LANGUAGES AND SYSTEMS TO DEMOCRATIZE DEVELOPMENT OF DATA-DRIVEN WEB APPLICATIONS

by
Michailia Verou

B.S., Athens University of Economics and Business (2013)
S.M., Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2024

AUTHORED BY	Michailia Verou Department of Electrical Engineering and Computer Science August 29, 2024
CERTIFIED BY	David R. Karger Professor of Electrical Engineering and Computer Science
ACCEPTED BY	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Languages and Systems to Democratize Development of Data-Driven Web Applications

by

Michailia Verou

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2024, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

A cornucopia of systems exist to facilitate web application creation, yet most are either too complex for novices, or too limited to cater to people's diverse needs. This work explores avenues to better balance the tradeoffs between complexity and expressiveness.

A focus of this work is the Mavo language, a modular set of technologies that enables authors with basic HTML knowledge to rapidly transform a static HTML mockup into a fully-functional, persistent, data-driven web application. Mavo HTML makes schema creation implicit, and generates high fidelity direct manipulation interfaces for editing data. It provides reactive computation via Formula^2, a hierarchical reactive expression language for novices, and remote persistence via Madata, a design and protocol for a distributed authentication and storage ecosystem with a unified API. Lastly, it extends the reactive paradigm with data update actions, allowing users to add interactivity or automate repetitive tasks. We later explored exposing these concepts to end-users via Lifesheets, a domain-specific prototype visual app builder geared around one of the most common personal data management use cases: personal tracking.

Unlike platform-based *"low-code/no-code"* approaches, extending open web technologies provides universal, portable, decentralized solutions. Our studies show novices quickly learn these technologies and feel empowered to create tools they never thought possible. By lowering the barrier of web programming, I envision a future where end-users feel empowered to create tools for their needs, while maintaining agency over their data and its location, in line with the Web's original vision: *"This is for everyone"*.

Thesis supervisor: David R. Karger
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

They say that raising a child takes a village. Having some experience with both, I would assert that so does a doctorate. This has been a very long journey and I could not have done it without the love, help, and support of so many.

This thesis is dedicated to the two women who made it possible: *the one I succeeded and the one who succeeds me*.

The former is my mom **Maria Verou**, whose life inspired this journey, yet she never got to see it. She was fearless, a genuine trailblazer. In 1976 — a time most Greek women did not even go to college — she moved halfway across the world to follow her dream of doing research at MIT. Her groundbreaking research helped solve real problems, yet she only published her work in her Master's thesis [1] as life forced her to return to Greece, cutting her PhD short.

I grew up hearing her recount her years at MIT as *intense* but also the most wonderful, most intellectually stimulating time of her life, and as a little girl I dreamed of following in her footsteps. For decades she longed to go back and finish what she started but never took the leap again, until on January 4th, 2013, it was finally too late. Shortly after, I uprooted my life and career to pursue my own dreams of research — before I also ran out of time. As a tribute to her memory, the programming language at the core of this thesis is named after her (Mavo = **Ma**ria **V**er**o**u).

The latter is my daughter **Zoe**, who has been with me for half of this journey. Through a strange turn of events, my pregnancy of her saved my life, and thus without her this thesis would not exist — as neither would I. Raising a child does make pursuing a PhD harder, but also a lot more meaningful. I love you, Zoe. Sorry for all the time I had to spend on this instead of playing with you.

To my husband **Chris**, who has been my support system for almost this entire journey. Thank you for moving halfway across the world to be with me and being there for me through thick and thin. I love you.

To my advisor, **David Karger**, in whom I found a mentor and a friend. I would have never reached the finish line without you. Thank you for believing in me, treating me like an equal, for being there to guide me when I needed you, and for giving me space and freedom when I didn't. You made me a researcher.

To my thesis committee members, **Arvind Satyanarayan** and **Sam Madden**, for their patience, flexibility, valuable feedback, guidance, and approachable demeanor. You made a very stressful process a lot more bearable.

Haystackers past and present. From left to right: (a) Soya Park, Tarfah Alrashed, Farnaz Jahanbakhsh, and me, (b) Group dinner at David's house.

To my fellow **Haystackers, past and present** that we have spent time with: Eirik Bakke, Amy X. Zhang, Farnaz Jahanbakhsh, Tarfah Alrashed, Nouran Soliman, Theia Henderson, Luke Murray, Jumana Almahmoud, Soya Park. I already miss our office laughs, impromptu brainstorming sessions, brunches, and surprise birthday parties.

Eirik, thank you for welcoming me to the group, introducing me to the ins and outs of grad school, for being such a fun officemate for my first two years, and for always being willing to give (and take) feedback. And for SUS.

Amy, thank you for making the process of writing my first paper less scary by being a fantastic co-author, for teaching me so much about academic writing, and for being such a well-organized co-instructor for our class.

Tarfah, thank you for being one of the kindest, most giving people I have ever met, a wonderful co-author, and a great friend.

To my **mentees and students**, who have taught me as much as I have taught them, and especially **Dmitry Sharabin**, for being Mavo's biggest fan, a tireless maintainer for that and many related projects, and a wonderful apprentice who has helped me immensely, including on some of the typesetting of this thesis. You rarely find people so eager to absorb knowledge, and it has been a pleasure seeing him grow over the years. Also to **Barish Namazov**, who has been a great student and later TA, co-author, and collaborator.

To my **friends** who provided the warmth of a family away from home, and the fun that made all the hard work feel worth it.

Different subsets of the Warehouse People (WHP) over the years: (a) when we all started in 2014, (b) at my and Chris' wedding in 2018, (c) at my birthday in Sirma's yard in 2020.

I was so incredibly fortunate to meet some of them on the first few days, at MIT orientation: Judith, Sirma, David, Viirj, Valerio, Thras, Lukas, Tal, Martin, Alexandros, Prashan, aka **The Warehouse People (WHP)**, from the MIT dorm many of us lived in during our first year. Making such good friends so early on turned a scary experience into a fun adventure that we all went through together. And we grew together too: we have now been through each other's thesis defenses, weddings, births, and many other life events, happy and sad. At this point most of us have been scattered around the world, but whenever we meet again, it feels like no time has passed.

Being so furtunate once was already unlikely, what are the odds of being so fortunate *twice*? And yet, on our daughter's first day of school, we met our friends **Ana** and **Filip** who immediately felt like long lost family to all of us, and their daughter **Eva** soon became almost like a sister to Zoe. They have supported us through health scares, deadlines, disappointments, and celebrations.

This thesis would not have been possible without **funding sources** from various fellowships and companies. I was fortunate to be the recipient of several fellowships (Paris Kanellakis, George and Marie Vergottis, Thomas G. Stockham Jr) and I am grateful to the families who set up these funds, and to MIT who selected me among hundreds of other candidates. I would also like to thank Wistron Corporation and NSF for funding parts of this work. It would also not have been possible without all the people who participated in my user studies and who sat down to be interviewed by me — thank you.

Last but not least, to **Travis Chase** and **Dave Gandy** for offering me a job so exciting it gave me the strength to finally wrap this PhD up, spread my wings, and leave MIT's protective cocoon. Thank you for believing in me without expecting me to jump through hoops like a circus animal. I can't wait to start.

How lucky have I been. ❤

Contents

Online version: phd.verou.me

List of Figures

List of Tables

CHAPTER 1

Introduction

5,120 words (15 min read)

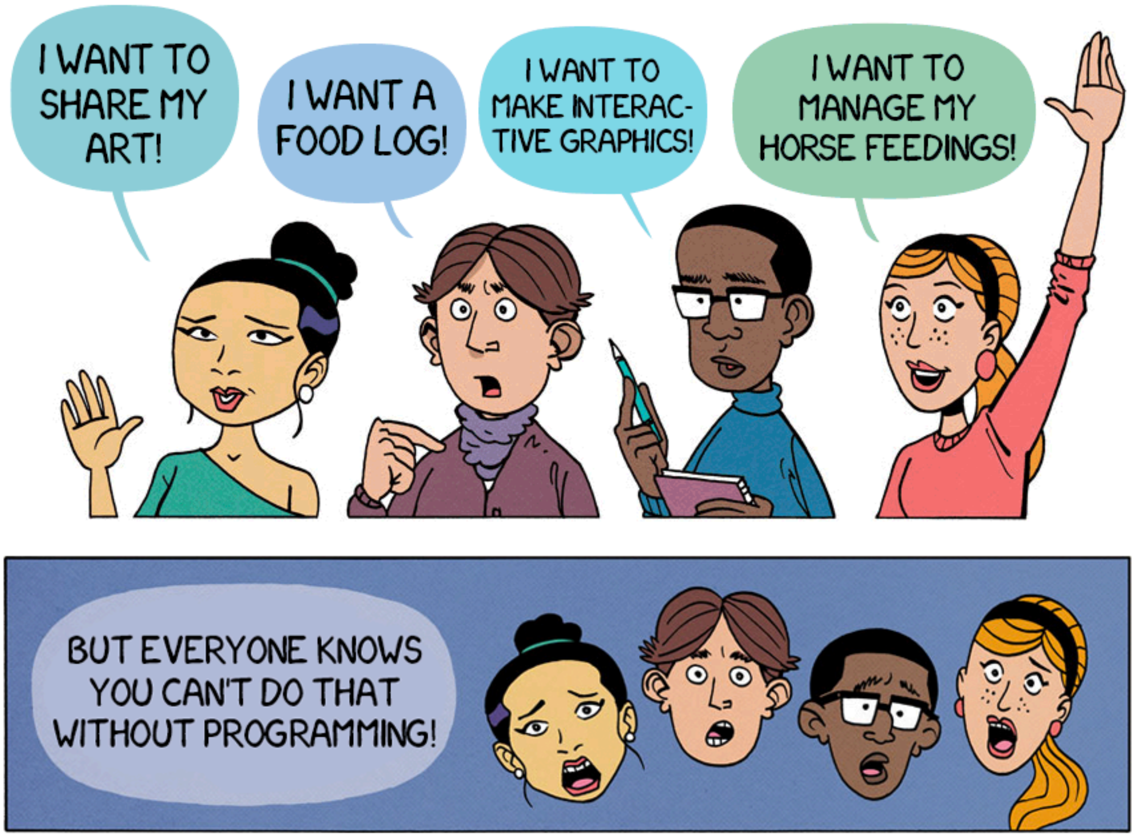

I WANT TO SHARE MY ART!
I WANT A FOOD LOG!
I WANT TO MAKE INTERACTIVE GRAPHICS!
I WANT TO MANAGE MY HORSE FEEDINGS!
BUT EVERYONE KNOWS YOU CAN'T DO THAT WITHOUT PROGRAMMING!

Figure 1.1 *Examples of applications novices wanted to build from our user studies. Graphic adapted from zenpencils.com/comic/98-alan-watts-what-if-money-was-no-object*

Most Web users have needs beyond what commercial web applications support. Automating common tasks, storing arbitrary data and performing calculations on them, tracking, are only a few categories of use cases.

Despite the Web originally being designed so that anyone could contribute, not just passively consume [1], these days the Web Platform[1] has grown tremendously in both complexity and power. It now takes years of training for someone to be in a position to create bespoke web applications, and even professional programmers with years of experience often lament the complexity of the modern web stack.

Even though trained programmers have the ability to create web applications for their own needs, the task is still so laborious, they rarely embark on it.

The goals of my research are three-fold.

1. The primary goal is to make web application development accessible to a wider audience and bring it within reach of *everyone*.
2. A secondary goal is to make it faster for *any* audience. If trained programmers can create prototype applications *really fast*, everyone wins.
3. Lastly, a tertiary goal is to contribute towards increasing the amount of machine-readable data on the Web and towards decentralization, not by attempting to convince users that these are worthy goals, but by creating technologies that incorporate them as a natural part of the interaction that does not require additional effort or even interest from the end-user.

1.1 Why Is Web Publishing Still Hard?

0.3 %

Few questions fill web practitioners with more dread than a variation of this deceptively simple query:

> *"I want to publish a simple personal website and be able to easily edit its content. Nothing much, just a bio, a portfolio, and a contact form. I can't afford to hire a web developer, but I'm a little technical, I think I could do it. What tools would you recommend?*

The reaction is typically a deer-in-the-headlights look, as if having explain to a small child that puppies die sometimes. It is true that a multitude of tools and services exists, but answering the question is less about picking the best tool for the job, and more about scrambling to figure out the lesser of many evils.

[1] The set of technologies used to develop web applications, see en.wikipedia.org/wiki/Web_platform

Social media services (e.g. Facebook or Medium) are likely the lowest threshold (see Section 1.4.1) solution, but also come with a very low ceiling. They afford no control over presentation, and data schema and storage is entirely controlled by the service provider. Similar downsides apply to website builders like Wix or Squarespace, though to a lesser extent.

Content Management Systems (CMSes) are meant to be a middle ground between the lack of control of centralized services and the complexity of writing a web application from scratch but are associated with high levels of dissatisfaction [2]: they still require a lot of technical skill to set up and maintain, they are bloated and heavyweight for most use cases, yet still too rigid for many common use cases.

On the other end, the highest ceiling solution is to write a web application from scratch. However, even for more technical users, this is a daunting task. Even a deceptively simple website like the one described above would require a lot of code, and deep understanding of many technical concepts such as authentication, templating, sending and receiving HTTP requests, data binding, handshakes, asynchronicity, security, and many more.

Despite the Web being originally envisioned as a read-write medium [1, 3], web publishing today suffers from numerous usability cliffs (see Section 1.4.2).

1.1.1 From Documents to Web Applications

0.6 %

Beyond publishing content, many users have data management needs that cross into the realm of *web applications*, requiring not just data binding, editing, and persistence, but also computation and interactivity. Examples abound: managing tasks, expenses, recipes, tracking life events, calculating interest rates and loan payments, or even more complex use cases like managing a small business or a community, to name a few.

Some of these use cases are common enough to make business sense for launching specialized commercial applications, but others are part of the very long tail of use cases that too niche to be served by commercial applications individually, yet vast in aggregate.

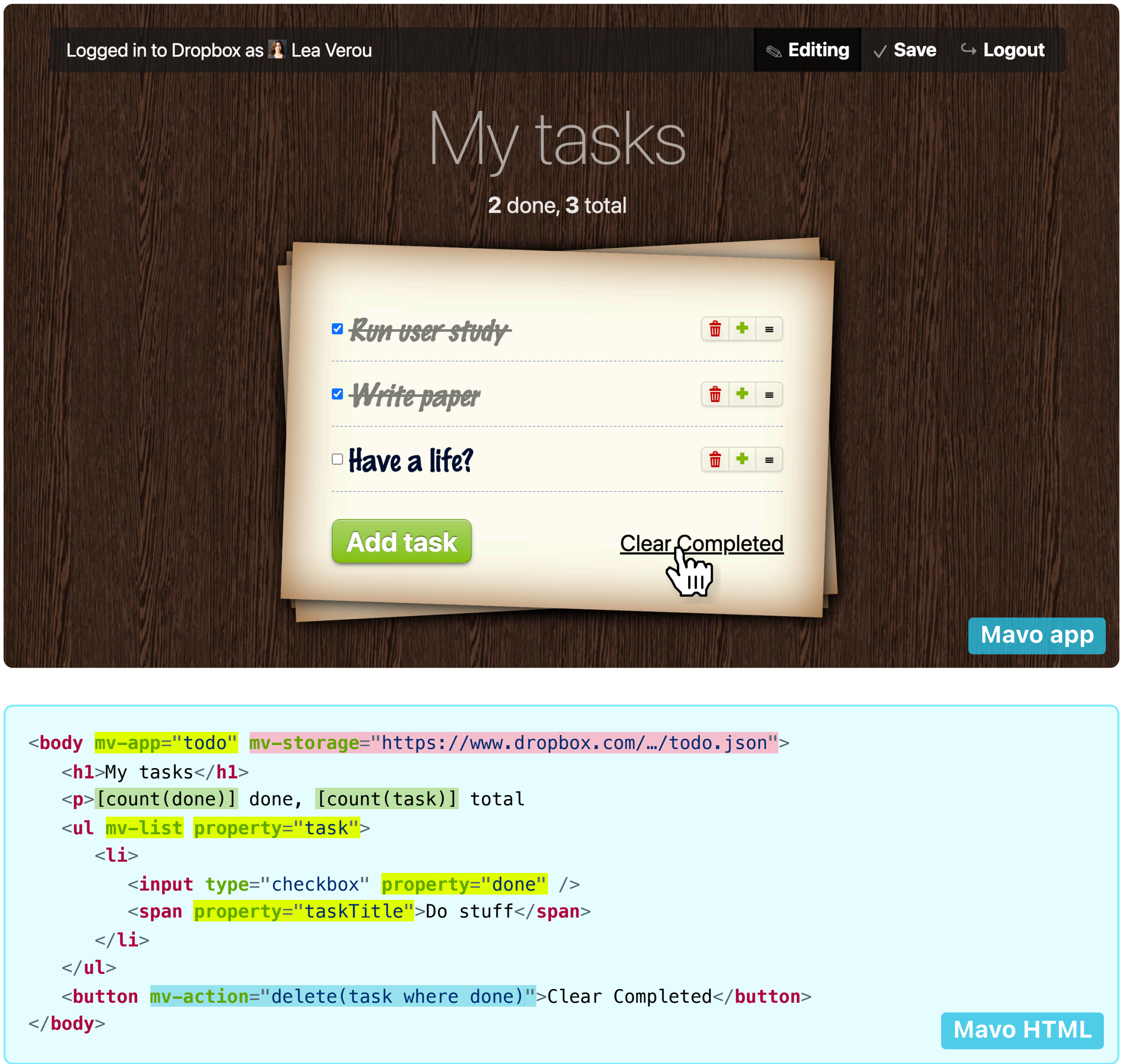
Even for use cases that on the surface appear to be well served by commercial applications, user needs are also varied and often not fully met by one-size-fits-all solutions. For example, let's take a simple use case like tracking household expenses. Some families have joint finances, others keep them separate. Some of the latter split expenses evenly, others proportionally by income, and others anywhere in between. Some families only need to deal with one currency, others travel enough that currency conversion is a frequent concern. Prefabricated applications either only deal to the subset of these needs that are most frequently encountered (known in product management as the 80/20 Pareto Principle [4]), or grow to enormous complexity (feature creep) if they try to cater to all of them. While avoiding feature creep is generally good, it does mean that the resulting applications skew toward mainstream needs, and often leave minorities behind.

The main alternative to prefabricated applications is to build one's own tools. Unlike the web publishing use cases, users rarely ask deceptively simple questions about this — they simply assume that building high fidelity tools for their needs is out of reach. When the delta between their needs and those catered by the prefabricated options, they typically try to adapt them to the tool. When it is too large, they resort to no-code tools such as spreadsheets, which do help with data management and lightweight computation, but are very limited in terms of presentation and interactions and many users struggle with authoring and debugging formulas [5, 6].

Creating websites and creating web applications is often treated as two distinct use cases, but the line between them is blurry. The need to manage structured data and share and display them on a webpage is very common. Consider a personal website displaying a portfolio, or a list of publications, speaking engagements, press mentions. Or a restaurant needing to manage and display their menu with dishes, prices, categories. Or a real estate website displaying listings of available properties. Or a wedding website that includes an RSVP. Or a conference website that includes a list of speakers, abstracts, and a schedule.

In all of these cases the data is *structured*, and cannot be managed (well) by interfaces essentially treating it as rich text. Furthermore, while these are often presented statically to end-users, end-users benefit tremendously [7] from the ability to interactively explore the data via filtering, sorting, aggregates, and other operations. The end-user need is so strong that there has been research in enabling such capabilities on websites not designed to provide it [8]. Making it easier for website authors to provide such functionality in the first place could provide tremendous value to end-users and have a ripple effect on the Web as a whole.

1.2 The Mavo Ecosystem for Low-code Web Application Development



```
<body mv-app="todo" mv-storage="https://www.dropbox.com/…/todo.json">
   <h1>My tasks</h1>
   <p>[count(done)] done, [count(task)] total
   <ul mv-list property="task">
      <li>
         <input type="checkbox" property="done" />
         <span property="taskTitle">Do stuff</span>
      </li>
   </ul>
   <button mv-action="delete(task where done)">Clear Completed</button>
</body>
```

Figure 1.2 *A to-do list application built with Mavo, showcasing all four core components: Mavo HTML, Formula², Madata, and Data Update Actions.*

It was these recurring pain points around managing, sharing, and transforming data on the Web that led me to design the Mavo language and associated components.

Mavo is a novel low-code programming language that extends the declarative syntax of HTML to describe small-scale web applications that manage, store and transform data (henceforth referred to as *data-driven web applications*).

Authoring HTML does require some technical skill (although the ACM cites knowledge of HTML and CSS to be at the K-12 level of computer literacy [9]), but **lowering the barrier of web application programming down to authoring HTML brings it within reach for everyone**. Even for end-users who have never written a line of code, learning HTML from scratch is a much more manageable task than learning the entirety of modern web development concepts.

The Mavo language consists of four key components:

1. FORMULA^2, a hierarchical formula language designed from scratch to be easy to use and understand, even when working with deep hierarchical data structures.
2. MADATA, a set of protocols and APIs which allow applications to read and store data either locally or to a variety of remote services, all with the same unified API.
3. The MAVO HTML syntax, which extends HTML with syntax to describe data-driven web applications and embeds reactive computation via Formula^2 expressions and unified storage via Madata URLs.
4. DATA UPDATE ACTIONS, an extension to both Mavo HTML and Formula^2, which allows authoring data manipulation sequences that are triggered by user actions while largely maintaining the same low *threshold* (see Section 1.4.1) of Mavo HTML and Formula^2.

While there is great synergy between these four components, each of them is an independent contribution of this thesis, and is useful even without the others.

All four Mavo components share the same design principles, which are also key features that enabled the growth of the early Web:

- *No installation, configuration, or maintenance.* Anyone could "join the Web" simply by putting an HTML file on a web server. Similarly with Mavo, one only needs to put an HTML file on a web server capable of serving static files (no server-side code execution is required) and they can immediately take advantage of Mavo. This is a direct corollary of Section 1.4.1.
- *Tinkerability*. A web application's entire logic is in its HTML file, and can be copied and tweaked. Furthermore, the data source of any Mavo app can be overridden by simply changing a URL parameter, which enables end-users to repurpose other people's Mavo apps for their own needs even without copying them to their own file space.

- *Incremental complexity.* Authors can add additional functionality and complexity in small steps, never needing to swallow a whole new set of ideas in one dose (see Section 1.4.2).
- *No network effect required.* Unlike social networking sites, Mavo provides immediate benefits to its first adopter, regardless of others' actions. It simplifies the author's management of their data, and offers visitors using existing web browsers a better interface to that data than can be built by typical web authoring tools with the same effort.
- *Robustness and fault tolerance.* Fault-tolerance is one of the design principles that guided the design of these technologies (see Section 1.4.4). Reminiscent of the design philosophy of Scratch [10], Mavo components generally attempt to do something sensible with most input rather than failing with an error message.

1.2.1 Formula^2: A Human-centric Hierarchical Formula Language

2 %

A key part of Mavo is its formula language called *Formula^2* (MavoScript in earlier literature) [11, 12]. Formula^2 expressions can be embedded almost anywhere in Mavo HTML by delineating them with certain syntactic tokens, or raw in certain attributes.

Formula^2 was designed with the explicit goal of reducing the amount of cognitive overhead around abstract data operations, and allow novices to write formulas that are closer to natural language, yet still unambiguous and easy to parse. To achieve this, it introduced several novel concepts, such as:

- *Implicit reference semantics*, where references are resolved based on the context of the formula, to alleviate users from complex mapping operations or long and fragile reference chains.
- *Seamless list-valued operations*, where operations on lists work just like operations on scalars, to reduce how much novices need to think about (or even know) the structure of their data.
- *Robust and forgiving syntax* in line with our design principles, which is unusual in the space of formula languages.

The contributions of Formula^2 are described in more detail in Chapter 4.

While originally developed for Mavo, Formula^2 has no particular dependence on Mavo concepts, and can be used to evaluate expressions against any arbitrary hierarchical data structure. That said, it is primarily useful for systems where the expression and the data have a natural mapping to elements in a visual layout, whose visual hierarchy largely follows the data hierarchy. Mavo is one such system, but so are most visual no-code systems.

1.2.2 Madata: Facilitating Data Ownership by Democratizing Data Access

2.2 %

One of Mavo's key features is its ability to store data remotely on a variety of cloud services, without requiring the author to register any OAuth [13] applications or write any authentication code. Storing and reading data remotely becomes almost as simple as storing it locally, and one storage service can be seamlessly swapped for another with the same capabilities without requiring any changes to the application code. All that users need to do is simply provide a URL that unambiguously identifies the storage location and Mavo takes care of the rest.

Originally hardcoded in Mavo HTML, after launching Mavo as an open source project in 2017, it quickly became clear that the potential reach of these concepts was broader than Mavo.

Reading and storing data is an integral part of many languages and systems. Yet, end-users typically have no control or ownership over their data. This is partly due to business reasons, but also because its is far easier for application developers to store data in a central location they control.

Madata makes it trivial to store data on any supported service, and swap out one service for another. Storage locations are specified by URLs, most of which can be easily obtained from the user interface of each service. Then, Madata takes care of the rest (authentication, data transformations, pagination, flags, etc.). Swapping one service for another is simply a matter of using a different URL, and requires no changes to the application code.

To ensure robustness and prevent centralization, extensibility is essential. Teaching Madata about new backends requires minimal JavaScript knowledge, especially for backends that follow certain known protocols (e.g. OAuth 2 [14]).

Madata frees authors from the need to procure servers that can run server-side code, a far more involved task. Nearly all of Madata runs client-side and interacts with APIs directly from client-side JavaScript.

There is one exception: **Authentication**. To facilitate experimenting with different storage locations without having to go through the hassle of registering applications, Madata introduces the concept of a *federated authentication provider*. This is a generalization of Mavo's original ad hoc authentication server (auth.mavo.io), which is now simply another Madata authentication provider. These are servers that encapsulate API keys for supported services, and handle authenticating end-users and ensuring that users are not misled by malicious applications.

The European Union establishes data portability as a fundamental human right [15]. Madata prototypes a future where end-users can own their data and choose its location by simply entering a URL in the settings of the application they are using. If they later change their mind, and wish to store their data elsewhere, all they need to do is change the URL. This data portability affords a federated version of data ownership that places no additional (time or technical skill) burden on end-users than centralized architectures.

1.2.3 Mavo HTML: Creating Web Applications by Authoring HTML

2.5 %

A key contribution of this dissertation is Mavo [11], a novel programming language that extends the declarative syntax of HTML to describe Web applications that manage, store and transform data (these will henceforth be referred to as *data-driven web applications*).

Using Mavo, authors with basic HTML knowledge define complex nested data schemas *implicitly* as they design their HTML layout. They need only a few HTML attributes and expressions to transform a static HTML template into a persistent, data-driven, access-controlled web application whose data can be edited by direct manipulation of the content in the browser. Mavo has been evaluated in lab studies, and in the real world, as an open source project.

Unlike current low-code/no-code approaches based on proprietary platforms, evolving the HTML language provides a solution that is universal and portable, with no dependence on any particular web infrastructure. By defining its syntax as an extension of HTML, all tools that process HTML — some of which *do* target end-users — can also process Mavo code.

This resulted in the following key ideas and primitives for Mavo HTML:

- *UI First.* User interfaces are less abstract than data, and thus require less technical expertise to reason about. With Mavo, authors are designing their interface with the tools they are used to; then they annotate where data goes in it. The data model is not specified separately, in the abstract; it is *generated* through these annotations. We believed that pointing to concrete places on a template is easier for novices than

the abstract data modeling tasks that traditional software engineering requires and our lab studies validated that hypothesis.
- *Editability.* Creating a WYSIWYG interface for editing data in place is as simple as naming the data and choosing an appropriate HTML element for it.

Furthermore, embedding Formula^2 expressions and Madata URLs in Mavo HTML results in these additional primitives:

- *Persistence.* Data can be stored locally or remotely, on one of the many supported cloud services, by simply providing a *storage URL*. Mavo takes care of authentication, if needed. Access control is enforced by the remote service.
- *Lightweight computation* through a reactive expression language called FORMULA^2 similar to spreadsheet formulas but designed for nested schemas like those organically created in most Mavos. A key feature of Formula^2 is its novel reference mechanism: properties can be referenced from everywhere in the template, and the relative placement of the expression to the data affects what the named reference resolves to.
- *Reactive defaults*, which are essential to many very common use cases such as smart default values, or editable formulas.

1.2.4 Data Update Actions

2.9 %

Originally, Formula^2 was purely reactive and side-effect free. However, we kept encountering use cases requiring programmatic data modification, triggerred by user actions. Often applications were almost entirely CRUD with lightweight computation and only one or two simple actions, but the inability to specify these actions made Mavo unsuitable for these use cases.

After exploring several alternatives, we decided to make these possible by extending Formula^2 with data update actions [12], which are only enabled in specific application-dependent contexts (e.g. an `mv-action` attribute in Mavo). We then did user research to ensure that our proposed syntax felt natural [16] to novices.

Our design adds minimal complexity but significantly expands the use cases that can be satisfied.

While our research focused on Mavo applications, the core concepts can be used to extend any reactive formula language with Data update actions (and since the publication of [12], some commercial no-code systems implemented similar ideas to great success).

In fact, data update actions do not even depend on hierarchical data structures, as this is a common spreadsheet user pain point. Perhaps this work could serve as a basis to address it.

1.3 Lowering the Threshold to End-Users

3 %

Mavo is a *low-code* language, rather than a *no-code* system[2] and targets HTML *authors* rather than end-users. While we have made the argument that the effort required for an end-user to become an HTML author is minimal, and certainly orders of magnitude smaller than the effort required to become a fully-fledged web developer, any amount of syntax is a barrier to entry for a large group of people.

My later research explored the question *"If we eliminate HTML syntax, would end-users be able to use and understand Mavo concepts?"*.

We hypothesized that a domain-specific visual app builder would be more effective. Since personal tracking use cases are both very common, and a class of applications with minimal network effects, we decided to start by prototyping LIFESHEETS, a visual IDE for building custom Quantified Self [17] applications.

In addition to demonstrating that Mavo concepts can largely be understood by end-users with no technical skill beyond spreadsheets, Lifesheets introduces a novel architecture for empowering users of all technical skills to create web applications that are portable, malleable, and not dependent on any particular infrastructure.

1.4 Design Principles

3.2 %

We conclude the introduction by describing a set of design principles that guided the development of the languages and systems presented in this thesis.

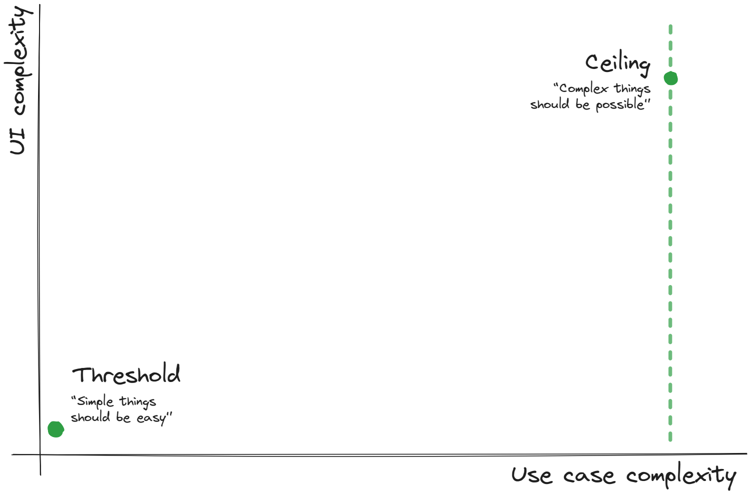
[2] There are currently no no-code *languages*, though advances in Artificial Intelligence may soon change this.

1.4.1 | Maximize the Distance Between Threshold and Ceiling

> *"Simple things should be easy, complex things should be possible" — Alan Kay (rumored)*

Decades later after Alan Kay, Myers et al formalized this idea by introducing the concept of *threshold* and *ceiling* [18].

The *threshold* is how difficult it is to learn how to use a system[3], i.e. its *learnability*; the *ceiling* is how much can be done using it, i.e. its *expressive power*.


UI complexity
Ceiling
"Complex things should be possible"
Threshold
"Simple things should be easy"
Use case complexity

Myers said that most successful systems are either low threshold / low ceiling (easy to learn but limited in expressiveness) or high threshold / high ceiling (hard to learn, but very powerful). In other words, most successful systems either trade off learnability for power or the opposite.

It seems clear that balancing a low threshold and a high ceiling would be ideal, but per Myers et al, it remains a challenge.

1.4.2 | Incremental User Effort Should Produce Incremental Value

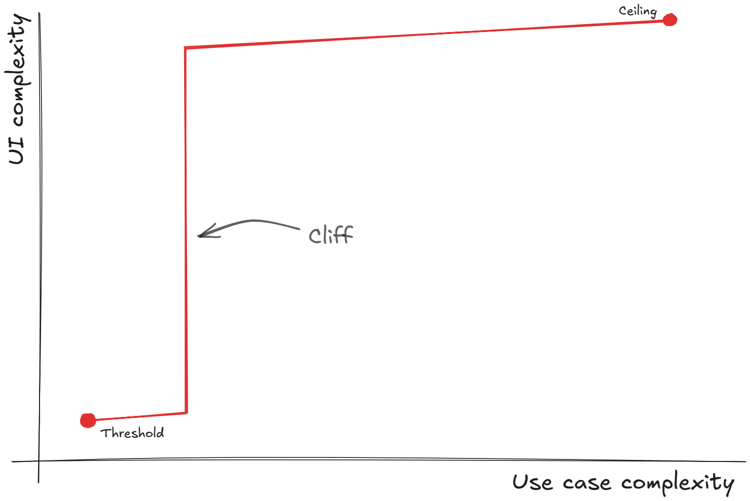
3.4 %

While a low threshold and a high ceiling are certainly desirable, and establish a usability bar that a good majority of systems cannot pass, they are not sufficient.

Many systems today achieve a low threshold and a high ceiling by simply combining a low threshold / low ceiling solution with a high threshold / high ceiling one. When more power is desired than what the low-threshold solution affords, users are directed to the

[3] "or language" is implied.

high-threshold solution. This introduces a "*usability cliff*", a point where a small increase in use case complexity results in a disproportionately large increase in UI complexity.



EXAMPLE

Relevant to this thesis is the example of the HTML5 `<video>` element. Its threshold is as low as HTML elements go: all it takes to embed a video on a webpage with a sleek video player is a single attribute to specify the video source and another to opt-in to the default playback controls:

HTML

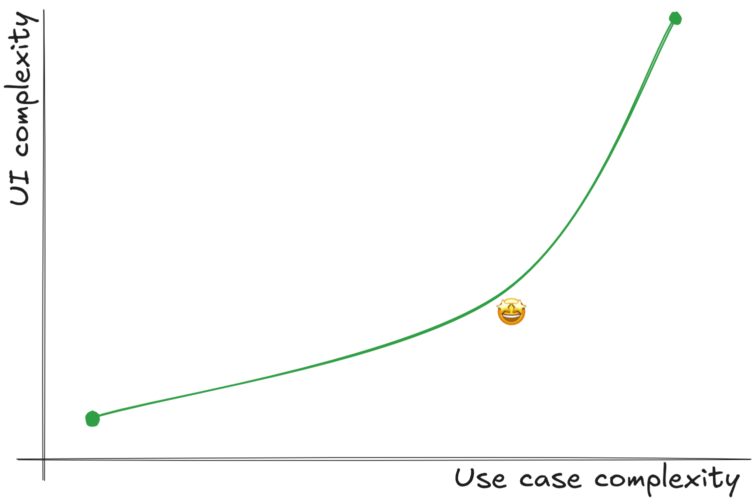
```html
<video src="myfile.mp4" controls></video>
```

However, authors cannot customize this playback toolbar beyond hiding buttons. Once any additional functionality is desired, such as a subtitle selector, or buttons to jump a few seconds back or forwards, the only option is to use the JavaScript API that these elements provide and write (*a lot* of) JavaScript to create a custom video player from scratch.

The *threshold* and *ceiling* merely establish the two extremes of a spectrum, but many use cases are not at either extreme. For optimal usability, we want a smooth *use case complexity to UI complexity curve*, where UI complexity increases gradually with use case complexity. Incremental user effort should result in incremental value; there should be no sudden jumps in complexity. The rate of increase matters too; the flatter, more horizontal the curve, the better.

Essentially, this is a corollary of the *Attention Investment Model of Abstraction Use* [19], whose core idea is that programmers have a finite supply of time and attention to invest. For an investment to be worthwhile, the expected payoff must exceed the cost, unless the

risk is too great. The cost of the investment is the amount of attention by the user that must be devoted to accomplishing a task. The expected payoff from that investment will be some saving of attentional effort in the future, such as by achieving a good abstract



formulation to reduce the amount of effort required to cope with similar problems. The perceived risk is the extent to which the user believes the investment will not produce the payoff, or that it will lead to even more costs that are not yet apparent. A cognitive simulation of programmer behavior has validated that this simple investment model can model many of the actions and decisions made during programming tasks, both by professional software engineers and end-user programmers [19], and there is evidence that it is effective in practical language design [19].

A lot of the work presented in this thesis is about either reducing the threshold of web programming, or making the curve of use case complexity to UI complexity more gradual.

1.4.3 Inference Should Be Escapable

3.8 %

Traditional programming languages often opt for explicit paradigms, where every parameter of the computation is specified by the programmer. Everything is clear cut, and there is no ambiguity, but to avoid potentially incorrect inference, this design offloads a lot of work to the programmer, increasing cognitive load.

This rigidity can be frustrating for novices, who are more familiar with the communication paradigms of natural language, which favor implicitness and ambiguity [20, 21]. In natural language, the receiver of a message will largely infer several concepts from context, and ask for clarification when needed. A compiler cannot ask for clarification, it can only produce errors. The “clarification” is essentially the programmer fixing the issue.

Heuristic algorithms that attempt to infer author intent from incomplete input can often improve user experience by reducing the amount of explicit input required and the amount of errors produced (which we know are discouraging). However, when the inference is incorrect, it is *essential* to provide a way for users to override the inferred behavior and provide explicit input.

> **EXAMPLE**
>
> CSS selectors [22] are a querying language for DOM trees, HTML's hierarchical object model. When declarations from two CSS rules conflict, the browser must decide which one to apply. Rather than a simple rule like "last one wins", CSS uses an elaborate algorithm taking many factors into account (*"The Cascade"*).
>
> One of these factors is the *specificity* of the selector, which assigns a weight to each selector based on its structure. Essentially **this is an inference mechanism that attempts to guess importance by proxy of querying logic**. For example, using an id selector (`#foo`) which in theory targets only a single element is more specific than using a class selector (`.foo`) which targets several elements, which in turn is more specific than using a tag selector (`div`) which targets any element of that type.
>
> This works somewhat well in practice, but there are many cases where the inference is incorrect. As a particularly egregious example, `:not(#foo)` targets all elements *except* one, yet enjoys the same high specificity as `#foo`.
>
> For years, this was a source of frustration for CSS authors, since CSS did not provide a general mechanism for *lowering* the specificity of a selector, only workarounds to *increase* it. This changed with the introduction of the `:where()` pseudo-class (proposed by the author in 2017[4]), and with Cascade Layers.

The importance of providing overrides depends on the frequency and consequences of incorrect inferences. In some cases, the precence of alternative ways to solve the same problem can be a sufficient escape hatch.

> **EXAMPLE**
>
> In JS, `array.concat(value)` attempts to infer intent based on the type of the argument(s) passed. If the argument is an array, it will append the array *values* to the original array, rather than appending the array itself. If the argument is *not* an array, it will append the argument itself, even when it's a different iterable, e.g. a `Set`. In this case, when a different behavior is desired the escape hatch is to use different language features, such as the spread operator, or `array.push()`, not to add options to `array.concat()`.

In some ways, this is a corollary of Section 1.4.2: inference is making simple things easy, while escape hatches are making complex things possible.

[4] github.com/w3c/csswg-drafts/issues/1170

1.4.4 Be Liberal in What You Accept

4.3 %

HTML is possibly the most tolerant mainstream computer language. This is no accident; tolerance was one of its earliest design principles [23–25].

Eliminating error messages does not eliminate errors. However, when a program does *something*, even if it is not correct, it feels closer to working and is less discouraging than a program that does not run (or compile) at all [10].

Per [26], there are no errors; all operations are iterations towards a goal. Typing mistakes or illegal statements can be thought of as an approximation. The language's job is then to aid the user in rapid convergence to the desired goal. In some cases, that may be achieved via inference (see Section 1.4.3), in others by failing gracefully. Notifying the user that there is a problem is important, but rarely requires complete and total failure.

This kind of *resilience* is especially important on the Web platform, where the environment is unpredictable and the user base is vast and diverse. There is no guarantee that when the error condition occurs, the user will be the website author. Thus, resilience ensures a better user experience for all Web users.

1.5 Thesis Overview

4.5 %

Chapter 2 positions this thesis in the broader context of related research and tools that aim to make web application development easier.

From there, Chapter 3 to Chapter 6, and then Chapter 9 describe various languages and systems democratizing web application development and empowering data ownership from different angles:

- Chapter 3 introduces the Mavo HTML language and briefly describes Formula^2 and Madata and how Mavo HTML integrates them.
- Chapter 4 expands on the Formula^2 hierarchical formula language.
- Chapter 5 expands on the Madata JavaScript API and federated authentication architecture.
- Chapter 6 introduces Data Update Actions, a way to add programmatic data manipulation to reactive formula languages.
- Chapter 9 introduces Lifesheets, a domain-specific visual application builder for building Mavo applications for personal tracking.

These chapters present the latest design of each technology, which is often the result of multiple iterations following insights from user studies and deployments. They include results from formative needfinding studies, example use cases, descriptions of system specifications, and implementation details.

Then, Chapter 7 provides an overview of the various studies conducted to evaluate these systems, and provides context for the status of Mavo technologies at the time each study was conducted. These include results from lab evaluations, case studies, and wide deployments as open source projects. We decided to present them after the description of all four languages and systems, as many studies were evaluating more than one component.

Chapter 8 presents a series of case studies of Mavo applications showcasing all technologies in the Mavo ecosystem working together to produce high fidelity applications. Some were created by Mavo users, and some by the author. Some are included because they showcase interesting patterns for common use cases, and others because they push the boundaries of what is possible with Mavo. Each case study is accompanied by a description of key points from its architecture and implementation, as well as a list of limitations it exposes in the current Mavo ecosystem.

Last, Chapter 10 summarizes design lessons from these languages and systems, their user studies and their deployments, discusses current limitations, and proposes future research directions.

The thesis concludes in Chapter 11 by reviewing and summarizing the contributions of this work.

CHAPTER 2

Background & Related Work

3,280 words (10 min read)

There is a multitude of systems that assist novice web developers and end-users with building dynamic, data-backed web applications, including research and commercial tools. This section provides an overview of the current landscape.

2.1 Web Publishing Tools

5.1 %

2.1.1 Social Media

In recent years, social media platforms for publishing web content have become a popular way for end-users to publish their content for free, without having to deal with any of the technical challenges of publishing a website they own.

Examples include Facebook Pages, blogging platforms (e.g. Medium, WordPress, Tumblr, etc.), or profiles on media-rich social platforms (e.g. Instagram, Flickr, TikTok, YouTube).

Out of all methods of publishing content on the web, this is certainly the one with the lowest threshold, which explains its popularity.

However, it comes with severe drawbacks and limitations on the type of content that can be published, what can be done with it, and how it can be displayed. These are typically designed around the most common, most generic use cases (e.g. a blog or a gallery of photos), which imposes severe limitations on the type of content that can be published, what can be done with it, and how it can be displayed. Because they are still part of the social platform that hosts them and need to maintain a consistent brand identity, they are usually limited in terms of personalization and customization. Custom functionality is typically not possible, and the platform may change or remove features at any time. Data is owned by the platform, and portability is hard or impossible. As a result, if the social media platform shuts down, content is (effectively) lost.

2.1.2 SaaS Visual Website Builders 5.2 %

Visual website builders like Wix and Squarespace have revolutionized web development by enabling novices to create and manage websites that look professionally designed through direct manipulation interfaces, making it feasible for small businesses, freelancers, and individuals to establish an online presence quickly and affordably. Being designed as creative tools, these afford much better customization than the social media solutions discussed in the previous section, as the central focus is the creative artifact produced, not the connections between users.

However their how threshold typically also comes with a relatively low ceiling, or the bifurcation described in Section 1.4.2 of having a low threshold and a high ceiling by combining a low-threshold/low-ceiling solution with a high-threshold/high-ceiling solution. These platforms rely heavily on pre-designed templates that dictate much of the site's layout and visual appearance. While these templates are often polished and professional, they are also quite rigid. Users can make changes within predefined sections and elements, but the overall structure is usually fixed. Advanced functionality is provided via predefined plugins, widgets, and integrations, which can be added to the site with a few clicks, but when these do not serve needs well, writing code is often the only escape hatch.

As a result, while these platforms excel in simplicity and speed for use cases that conform to the most mainstream of needs, they fall short to cater to the very long tail of specialized use cases that emerge in practice.

Additionally, since these are (usually proprietary) platforms, users have limited control over their data and content and transferring a website to another platform can be challenging.

2.1.3 Content Management Systems (CMSes) 5.4 %

CMSes are possibly the most popular way for end-users to publish their content on a website they control.

These include platforms typically hosted on one's own server which connect to some form of data storage (e.g. a database), and provide templating functionality and visual affordances for editing content. Examples of such systems include CMSes such as Wordpress, Drupal, or Joomla.

Previous work has explored the high levels of dissatisfaction with how rigid and heavyweight these are [27].

The drawback to many of these systems is that they often require using their own heavyweight authoring and hosting environments, and they provide pre-made plugins or templates that users can not customize without programming.

Another drawback is that they are structured around a very crude model of what is UI and what is data, typically consisting of a set of pages with content that is edited all at once, and no computation. While this model works for content-heaby *websites*, such as blogs or media portals, it does not work so well for displaying and editing structured data which is a lot more fine-grained than a single blob of text with a title and other metadata.

Displaying and editing structured data is a broad category of use cases that come up very frequently, even for content-heavy websites. For example, thing of the personal blog of a popular conference speaker, a textbook CMS use case. Displaying their list of talks, their list of publications, a list of press mentions, a list of interviews they have given, all of these are examples of structured data for which a CMS is not well suited.

2.1.4 Static Site Generators (SSGs) 5.6 %

The dissatisfaction around CMSes bred the growing community around static site generators, such as Jekyll [28] and Eleventy [29]. These do not have a visual interface at all, content is typically stored in Markdown files and HTML templaes, and the final HTML is generated by invoking a terminal command.

While these are a lot more lightweight and afford tremendous levels of control, they practically target exclusively web developers, as they require significant technical expertise to configure, and offer no graphical interface for editing data.

Many uses of CMSes are merely to enable non-technical users to edit website content, a use case that static site generators do not accommodate.

2.1.5 "Headless" CMSes 5.7 %

"Headless" CMSes are tools designed to bridge the gap between CMSes and SSGs, by combining the ease of use of the former with the control of the latter. However, these

typically require the SSG to first be configured normally, and then its templates painfully annotated to tell the CMS where data should go and how to edit it.

Moreover, they tend to fare poorly at providing a WYSIWYG preview of the rendered website, since it's not always clear to them what the content managed in the system will be used for in the end.

2.2 Spreadsheet Extensions

5.8 %

So far, the types of website builders discussed focus on editing *content* and making it look good via templates. Any computation is added via plugins, and if no suitable plugin exists, it requires programming.

However, there is already a very successful paradigm for end-users to store their data and perform computations on it: spreadsheets. Because of the popularity of spreadsheet applications, many researchers and practitioners have explored eliminating the usability issues of spreadsheets and pushing the boundaries of the spreadsheet paradigm.

Common extensions to the spreadsheet paradigm include:

- Extending the formula language to named references [30–32]
- Allowing the user to define datatypes, defaults, and formulas for entire columns [30–32]
- Extending the formula language and input affordances to support hierarchical data [33–35]
- Allowing the cells to be arranged in layouts other than a grid [36]
- Extending the output to richer data types such as interactive graphics [36–38] or maps [39]
- Making it easier to correlate data across multiple tables via relations [31, 32]

While spreadsheets address the user need for lightweight computation, most spreadsheet systems (research or commercial) share the same limitations:

- They afford very little to no customization in terms of input UI
- They are typically not portable: the data is stored within the spreadsheet, and the functionality cannot easily be repurposed to handle different data or moved to a different platform.

- Many only target single-user local web applications and do not address the unique challenges that Web applications raise.

Because of the popularity of spreadsheet applications, some researchers have explored using a spreadsheet for end-users to define and manage their data. For example, Quilt is a system that allows users to link a Google spreadsheet with a webpage and provides simple syntax to bind GUI elements with particular cells in the spreadsheet [40]. Similarly, Gneiss is a live programming environment that incorporates a spreadsheet editor and allows users to create bindings between GUI elements and spreadsheet cells [33].

2.3 Do-It-Yourself Database-driven Web Applications

6.1 %

A large class of web applications are purely CRUD (Create, Read, Update, Delete) interfaces to structured data. Databases allow storing and querying structured data, but integrating them into a web application is quite laborious even for professional programmers. A small study [41] found that the ratio of "plumbing" code to pure data code (business logic + SQL) in a web application was a whopping 24.4:1!

Therefore, many systems have focused towards making database systems more user-friendly for web application development, and/or building interfaces to easily display and edit database data.

One direction involved bridging the spreadsheet and database paradigms by exposing a database as a hierarchical spreadsheet [32, 35, 39, 42, 43].

SQL is a widely accepted data query and manipulation language, and its declarative nature means that relatively complex data queries and updates can be performed using even a single short line of SQL.

However, web designers with limited knowledge of databases might not be able to write SQL queries in order to make these edits programmatically. Several database-driven web application platforms have been developed to assist non-programmers to build web applications. WebML [44] presented a web modeling language that provided a graphical way of specifying the database schema and navigational structure of web application. However, WebML does not provide a mechanism to do programmatic updates to the data. A lot of work has been conducted on developing visual query languages [45–47]. These systems hide the SQL syntax from the users, but they still show the database schema and the relational tables, which could be overwhelming for non-programmers with limited knowledge of databases. They also do not offer any way for the user to create web pages on top of these visual query languages. Other systems have focused on creating form-based visual tools for creating queries, design database, and define views [48–50]. However, these tools do not offer a WYSIWYG environment and they similarly require the users to deal with joins across multiple tables, which has been shown to be unnatural for average users [51].

AppForge [52] tried to hide the complexity of building and editing databases by developing a graphical interface to navigate the database schema. And like our proposed extension to Mavo, it provided graphical primitives, in which developers can create and edit NRA views over the schema. Nevertheless, it exposes non-programmers to the complexity of databases. FORWARD [53] is another system that provides a powerful WYSIWIG environment for creating web applications, however, not only it requires writing SQL queries within HTML, but it also requires writing JavaScript if users need to create a custom visual layer. Other systems, like that presented by Kowalzcykowsi et al. [54] provide a WYSIWYG environment and do not require users to edit the database schema directly; nevertheless, they do not provide an abstraction for complex relationships, aggregation and nesting. Mavo [11] allows users with basic HTML knowledge to create Web applications that manage, store and transform data, and unlike some of the previous work it provides for nested data, but does not let you join one nested data blob to another, it also offers controls for adding, updating, and deleting individual items manually. However, the only data manipulation that Mavo presented is direct editing of a single item, although in many applications, even simple ones, there are more complex editing actions that need to be developed. In this work, we extended Mavo to support specifying such actions programmatically.

2.4 WYSIWYG Application Builders

6.6 %

Visual application builders like app2you [54] and AppForge [52] allow authors to specify the design of pages by placing drag-and-drop elements into a WYSIWIG-like environment. However, this approach limits authors to only the building blocks provided by the tools and provides very little control over the specifics of the interface created.

Some systems have been developed to provide a WYSIWYG interface that allows non-programmers to create web applications, without dealing with the complexity of databases and SQL queries, by using spreadsheet as the a back-end. Dido [55] allows users to visualize, edit, and store editable data directly in their browser. It allows web designers to integrate Dido into any web design and made it independent of any back-end system. Another system is Quilt [40], which integrates web applications to a Google spreadsheet, allowing web authors with no programming skills to gain access to lightweight computation.

Gneiss [33, 34] is another interactive system that extended a spreadsheet. It lets users retrieve JSON data returned from web services to a spreadsheet interactively without programming, unlike some of the previous work, Gneiss supports hierarchical data. However, since it depends on spreadsheets as the back-end, it does not really provide a mechanism to update data. None of the previous work provided a mechanism for programatically allowing end-users to specify data updates, without them having to write SQL queries, which can get complicated for nested schemas [51] or scripting. Our work is building on top of Mavo to make it more powerful, allowing users to specify computational data updates that are not evaluated reactively, but are executed based on user interaction. Rather than limiting users to only manually data editing, we want to empower them to create richer data interactions and ultimately, to build more powerful web applications.

2.5 HTML Extensions for Web Application Development

6.8 %

The idea of extending HTML to make it more powerful is not new; there have been many past attempts at extending it in different directions.

Many attempts to make HTML more powerful treat HTML as a shortcut for programmers to express programming concepts more succinctly. They focus on reducing the *amount* of programming code required, not its *difficulty*. One such system was FORWARD [41] which aimed to simplify the "plumbing code" needed to render and edit data stored in a SQL database into a web page. It was quite powerful, but required writing SQL queries within HTML. There are also several JavaScript frameworks with this philosophy, starting with AngularJS [56] in 2010 and more recently VueJS [57]. These adapt and extend HTML to present dynamic content through two-way databinding that allow for the automatic synchronization of models and views, but require the user to be well versed in JavaScript to use them.

ConstraintJS [58] extended HTML with a templating syntax and reactively evaluated constraints, but required the user to understand and write JavaScript.

Exhibit [7] (and later Dido [55], based on Exhibit) were some of Mavo's early influences. Exhibit extended HTML with language elements that visualized and stored editable data directly in the browser. This approach allowed a web designer to incorporate Dido into any web design and made Dido independent of any back-end system.

Quilt [40] was one of Mavo's biggest influences. It extended HTML with a language for binding an arbitrary web page to a Google spreadsheet "back-end", enabling web authors to gain access to lightweight computation without programming. While it afforded full creative freedom, and lightweight computation, like Mavo, it imposed the software engineer mindset of data modeling as a separate task, and UIs as views that need to connect to the data model as a separate step, which can be cognitively taxing for end-user programmers that tend to be goal-oriented. Additionally, it was by design limited to spreadsheets as the means for data storage.

2.5.1 Web Components

7.1 %

"Web Components" is the colloquial term for a set of standardized technologies that allow developers to encapsulate reusable functionality in HTML elements, and provide an extensibility point to HTML by allowing the creation of custom elements. While Web Components require (fairly advanced) JavaScript to create, because they can be packaged and distributed, the theory is that novices can import them and use them just like native HTML elements.

Web Components are not an alternative to Mavo: first, novices can only use them, not create them, and second, they exist at a different level of abstraction than Mavo which focuses on facilitating data interactions, rather than encapsulating UI functionality.

However, Web Components are **complementary** to Mavo. Because Mavo leverages existing HTML elements as foundational building blocks, the addition of custom elements through Web Components broadens the range of functionalities available to Mavo authors. Well-designed Web Components extend Mavo's capabilities while maintaining the same low threshold as native HTML elements, thereby expanding Mavo's utility without increasing its complexity.

2.5.2 HTML Transformation Languages

7.2 %

There are several languages designed around transforming data to HTML or simpler HTML to more complex HTML, in order to automate repetitive templating tasks.

Extensible Stylesheet Language Transformations (XSLT) [59] is a language designed for transforming XML documents into different formats, including HTML. It is primarily used to transform XML data into a presentable HTML format, applying styles and formatting rules that dictate how the content should be displayed in the browser. XSLT is powerful in environments where XML is the primary data format, allowing for the separation of content and presentation, but its syntax is complex and verbose, making it difficult for non-programmers to use.

Several templating languages also exist (e.g. Handlebars, Mustache, Jinja, etc.) that allow authors to write HTML templates with placeholders for data, and store the data separately in a structured format (e.g. JSON, YAML, etc.).

Cascading Tree Sheets [60] was a research language that essentially functioned as a templating language where both the data and the output were HTML. It allowed authors to write minimal HTML documents with only the elements required to hold their data, and add any superfluous presentational markup as transformations of that HTML, specified via CTS rules with a CSS-like syntax.

These languages are typically static one-time transformations that produce HTML from data, not dynamic data bindings.

2.6 The Semantic Web and Web Data Extraction

7.4 %

There has been a great deal of work on both encouraging and extracting structured data on the web [61]. However, automatic scraping techniques often have errors because they must infer structure from unstructured or poorly structured text and HTML markup.

Several efforts have been made to define syntaxes and schemas, such as RDFa [62] and Microdata [63], for publishing structured data in web pages to contribute to the *Semantic Web* and *Linked Open Data* [64]. However, novice users have had little incentive to adopt these standards — sharing data rarely provides direct benefit to them — and find them difficult to learn, potentially contributing to their limited adoption on the web.

It appears that the approaches that work best for increasing adoption of semantic web technologies are those that provide immediate benefits to them, such as search engines displaying richer results for structured data, or tools using structured data to improve user interfaces and/or make prose more informative [65].

Mavo contributes to this line of work by using a standards-compliant syntax that is machine-readable, yet produces tangible benefits. With Mavo, authors expend effort because it makes their static website editable or creates a web application. As a side effect, however, they enrich the Semantic Web by producing structured data.

2.7 Novice Mental Models & Natural Programming

7.5 %

In the last few decades, several research efforts have focused on how novice programmers or non-programmers struggle in learning how to program [66–68]. These studies showed that this is because of the mental models novice programmers build about the notional machine. Another study found that programming is more difficult than necessary because it requires solutions to be expressed in ways that are not natural for non-programmers [69]. The study examined the ways that non-programmers indicate solutions to common programming tasks, which are often vastly different than the ways programming languages require solutions to be expressed.

Natural Programming [16, 70, 71] is a research area that aims to make programming more accessible to non-programmers by studying what syntax and mental models feel most natural to them, and use these insights in designing languages and systems that allowing them to express solutions in ways that are more natural to them.

CHAPTER 3

Mavo: Creating web applications by authoring HTML

7,884 words (23 min read)

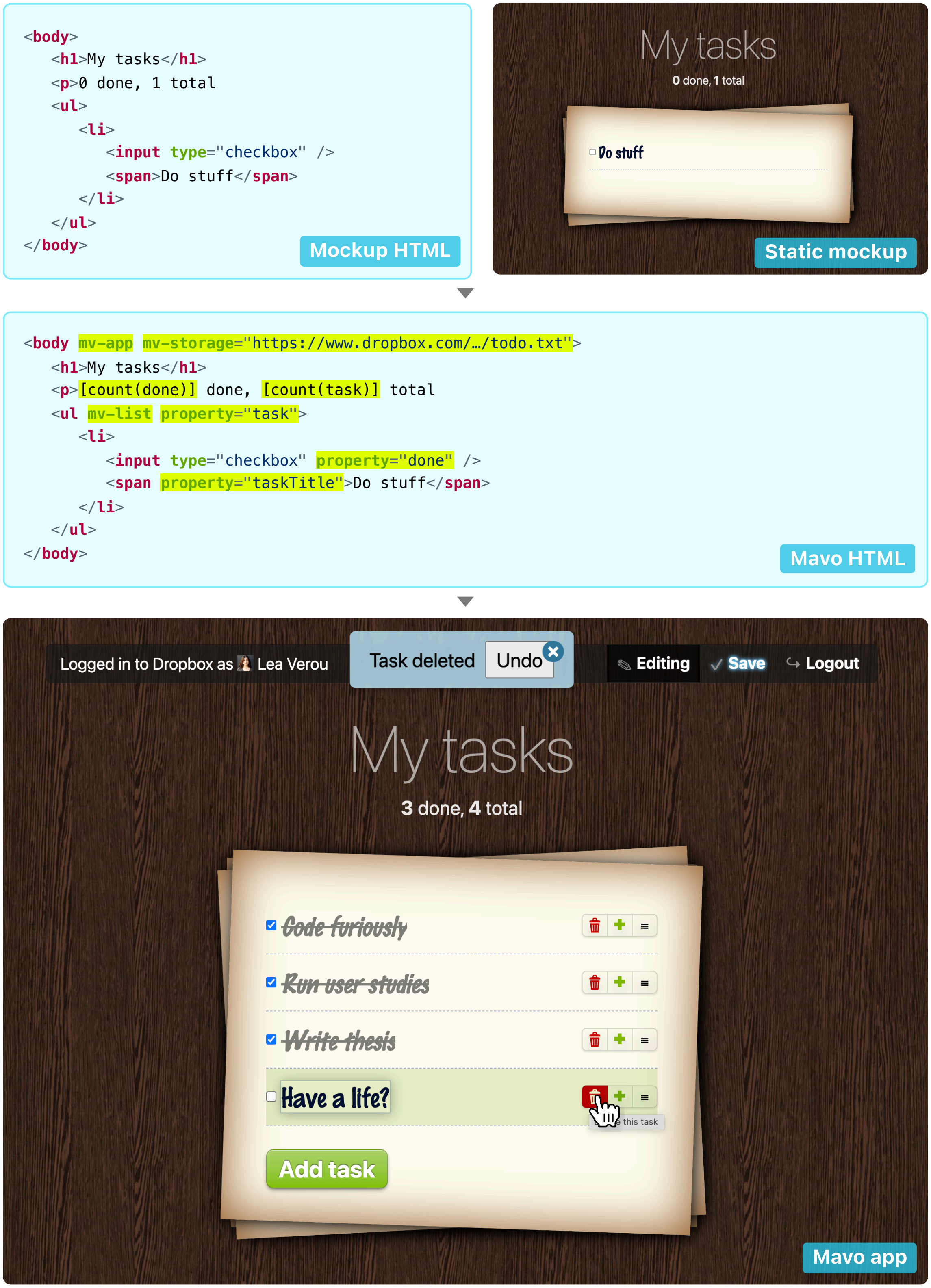
3.1 Introduction

7.8 %

Languages like HTML and CSS have characteristics that make them more natural [70, 71] to learn and use. They are *declarative*, *reactive*, *robust* and *forgiving* in terms of syntax. Authors assemble high-level concepts and constraints, rather than explicit instructions. Robustness is achieved in different ways across the two (HTML attempts to *correct* authoring mistakes, CSS to scope them tightly and ignore them), but both are designed with resilience and fault tolerance as a design principle.

These desirable properties have given rise to a large community of authors who are comfortable with HTML and CSS, yet not being comfortble with JavaScript or other traditional programming languages. While it is difficult to pinpoint the size of this community, it is likely large and growing. The ACM cites knowledge of HTML and CSS to be at the K-12 level of computer literacy [9].

Far more powerful than static pages are web *applications* that react dynamically to user actions and interface with back-end data and computation. Even a basic application like a to-do list needs to store and recall data from a local or remote source, provide a dynamic interface that supports creation, deletion, and editing of items, and have presentation varying based on what the user checks off. Creating such applications requires knowledge of JavaScript and/or other programming languages to support the necessary user interaction and to interface with a data management system, as well as understanding of some form of data representation, such as JSON or a relational database.



```
<body>
   <h1>My tasks</h1>
   <p>0 done, 1 total
   <ul>
      <li>
         <input type="checkbox" />
         <span>Do stuff</span>
      </li>
   </ul>
</body>
```

```
<body mv-app mv-storage="https://www.dropbox.com/…/todo.txt">
   <h1>My tasks</h1>
   <p>[count(done)] done, [count(task)] total
   <ul mv-list property="task">
      <li>
         <input type="checkbox" property="done" />
         <span property="taskTitle">Do stuff</span>
      </li>
   </ul>
</body>
```

Figure 3.1 *A fully-functional To-Do application made with Mavo, shown with its accompanying code and the starting HTML mockup. CSS not shown, but is only used for styling.*

There are many frameworks and libraries aiming to simplify creation of such Web applications. However, all target programmers and still require writing a considerable amount of code. It is indicative that even implementing a simple to-do application similar to the one in Figure 3.1 requires hundreds of lines of code:

Framework	LOC
AngularJS	294
Polymer	246
Backbone.js	297
React	421

Table 3.1 *Lines of JavaScript code required to implement a simple to-do application in popular JavaScript frameworks. Other frameworks are in the same ballpark. Comments not included in the count. Statistics from todomvc.com.*

Many people who are comfortable with HTML and CSS do not possess additional programming skills[1] and have little experience articulating data schemas [72]. For these novice web authors, using a CMS (Content Management System) is often seen as their only solution. However, research indicates that there are high levels of dissatisfaction with CMSs [2]. One reason is that CMSs impose narrow constraints on authors in terms of possible presentation–far narrower than when editing a standalone HTML and CSS document. When an author wishes to go beyond these constraints, they are forced to become a programmer learning and modifying server-side CMS code.

The problem worsens when authors wish to present structured data [27], which CMSs enable via plugins. The interfaces for these plugins do not allow authors to edit data in place on the page; instead they must fill out forms. This loses the direct manipulation benefits that are a feature of WYSIWYG editors for unstructured content.

Finally, CMSs provide a heavyweight solution when many authors only need to present and edit a small amount of data. For example, out of the over 7,000 CMS templates

[1] We carried out a snowball sample of web designers using a Twitter account followed by 70,000 Web designers and developers. Of 3,578 respondents, 49% reported little or no programming ability.

currently provided in ThemeForest.net, a repository of web templates, 39% are for portfolio sites, while another 31% are for small business sites.

This chapter presents and evaluates a new language called Mavo[2] that augments HTML syntax to empower HTML authors to implicitly define data schemas and add persistence and interactivity. Simply by adding a few HTML attributes, an author can transform any static HTML document into a dynamic data management application. Data becomes editable directly in the page, offering the ability to create, update, and delete data items via a WYSIWYG GUI.

3.1.1 Implicit Data Schema Definition

9 %

While programmers generally prefer to keep their data schema logic separate from presentation definition, end-users may not have the same preferences, and may instead be frustrated by the need to think about data in two separate places. Indeed, with a certain category of applications, including most CRUD applications, how the data is laid out on the page can easily translate to how the data should be organized. For end-users who are seeking to build these sorts of apps, it may be easier to define a proper schema in tandem with defining the layout.

Mavo authors never have to articulate a schema separately from their interface or write data binding code. Instead, authors add attributes to describe which HTML elements should be editable and how, unwittingly describing their schema by example in the process. With a few attributes, authors quickly imply complex schemas that would have required multiple tables and foreign keys in a relational database, without having to think beyond the interface they are creating. As an added benefit, Mavo's HTML attributes are part of the HTML RDFa standard [62] and thus contribute to machine-readable data on the Web.

Mavo is inspired by the principle of *direct manipulation* [73] for the creation of the data model underlying an application. Instead of crafting a data model and then deciding how to template and edit it, a Mavo author's manipulation of the visual layout of an application automatically *implies* the data model that drives that application. In addition, Mavo does not require the author to create a separate data editing interface. Users simply toggle an edit mode in their browser by clicking an edit button that Mavo inserts on their webpage. Mavo then adds affordances to WYSIWYG-edit whatever data is in view, with

[2] Open source implementation & demos available at **mavo.io**

appropriate editing widgets inferred from the implied types of the elements marked as data. Mavo can persist data locally or outsource storage to any supported cloud service, such as Dropbox or Github. Switching between storage backends is a matter of changing the value of one attribute.

In addition to CRUD functionality, Mavo also embeds Formula^2 expressions, allowing users to perform complex calculations on nested data with a natural syntax. Formula^2 has been described in detail in Chapter 4.

In contrast to the hundreds of lines of code demanded by the popular frameworks, Figure 3.1 shows how an HTML mockup can be transformed into a fully functioning to-do application by adding only 5 lines of Mavo HTML.

Our approach constitutes a novel way for end-users to transform static webpages to dynamic, data-backed web applications without programming or explicitly defining a separate data schema.

From one perspective, this makes Mavo the first *client-side CMS*, where all functionality is configurable from within the HTML page. But it offers more. In line with the vision of HTML as a *declarative* language for describing content so it can be *presented* effectively, Mavo extends HTML with a declarative specification of how the data underlying a presentation is structured and can be *edited*.

Fundamentally a language extension rather than a system, Mavo is completely portable, with no dependence on any particular web infrastructure, and can thus integrate with any web system.

Similarly, existing WYSIWYG HTML editors can be used to author Mavo applications. We offer Mavo as an argument for the benefits of a future *HTML language standard* that makes structured data on every page editable, persistent and transformable via standard HTML, without dependencies.

We conducted a user study with 20 novice web developers in order to test whether they could use Mavo to turn a static HTML mockup of an application into a fully functional one, both with HTML we provided and with HTML of their own creation.

We found that the majority of users were easily able to mark up the editable portions of their mockups to create applications with complex hierarchical schemas.

3.2 Related Work

Quilt [40] was a system that allowed users to link a Google spreadsheet with a webpage and provided simple syntax to bind GUI elements with particular cells in the spreadsheet. Similarly, Gneiss is a live programming environment that incorporates a spreadsheet editor and allows users to create bindings between GUI elements and spreadsheet cells [33].

Mavo combines ideas from three prior systems that addressed the downsides of CMSs. Dido [55] built on Exhibit [7], extending HTML with language elements that visualized and stored editable data directly in the browser. This approach allowed a web designer to incorporate Dido into any web design and made Dido independent of any back-end system. Quilt [40] extended HTML with a language for binding an arbitrary web page to a Google spreadsheet "back-end", enabling web authors to gain access to lightweight computation without programming. Gneiss [33, 34] was a web application within which authors could manage and compute over hierarchical data using an extended spreadsheet metaphor, then use a graphical front end to interact with that data.

These three systems introduced powerful ideas: extending HTML to mark editable data in arbitrary web pages, spreadsheet-like light computation, a hierarchical data model, and independence from back-end functionality. But none of these systems provides all of these capabilities simultaneously. Dido had no computational capabilities, could not manage hierarchical data, and was never evaluated. Quilt was dependent on a Google spreadsheet back-end, which left it unable to manage hierarchical data. Gneiss was a monolithic web application that only allowed the user to construct web pages from a specific palette. It did not offer any way (much less a language) to associate an arbitrarily designed web page with the hierarchical data Gneiss was managing, which meant that a web author faced constraints on their design creativity. Gneiss and Quilt both required users to design their data separately from their web pages.

Mavo is a *language* that solves the challenge of combining the distinct positive elements of this prior work, which are in tension with one another. It defines a simple extension to HTML that enables an author to add data management and computation to *any* web page. At the same time, it provides a lightweight, spreadsheet-like expression language that is expressed and evaluated *in the browser*, making Mavo independent of any particular back-end. The editing and expression language operates on *hierarchical data*, avoiding this limitation of traditional spreadsheet computation.

The combination of these ideas yields a novel system that is particularly well-suited to authoring interactive web applications. In Mavo (like Dido), the author focuses entirely on the design of the web page, then annotates that page with markup describing data and computation. The web page *implies* the data model, freeing the author of the need to abstractly model the data, manage a spreadsheet, or describe bindings between the two. At the same time, our expression language provides lightweight computation (Quilt and Gneiss), even on hierarchical data (Gneiss) without relying on any external services (Dido). Because they are part of the document (Dido), Mavo expressions can refer directly to data elements elsewhere in the document, instead of requiring a syntactic detour through references to cells in the associated spreadsheet. Finally, because it is an HTML language extension (Dido and Quilt), Mavo can be applied to *any* web page and authored with any HTML editor, freeing an author from design constraints.

In sum, we believe that the combination of capabilities of Mavo align well with the needs and the preferred workflow of current web authors. In particular, the independence of the Mavo authoring *language* from any back-end system (or even from any particular front-end interpreter) means that Mavo prototypes a future for HTML and the web browser itself, where data interaction becomes as much a basic part of web authoring as paragraphs and colors.

3.2.1 End-User Web Development

10.1 %

There are many systems that assist novice web developers with building dynamic and data-backed web applications. The drawback to many of these systems, however, is that they often require using their own heavyweight authoring and hosting environments, and they provide pre-made plugins or templates that users can not customize without programming. Examples of such systems include CMSs such as Wordpress, Drupal, or Joomla. The growing community around static site generators, such as Jekyll [28] is indicative of the dissatisfaction with rigid, heavyweight CMSs [27]. However, these require significant technical expertise to configure, and offer no graphical interfaces for editing data.

In the previous section, we described three systems—Dido [55], Quilt [40], and Gneiss [33]—from which we draw key insights. However, this work solves challenges in combining those insights into a single system, incorporates additional ideas, and contributes useful evaluation of the resulting system. Most importantly, Mavo demonstrates that the often-hierarchical data model of an application can be incorporated directly into the visual design on which a web author is focused, making the data modeling task an automatic side effect of the creation of the web design. Supporting hierarchical schemas is critical because they occur naturally in many data-driven apps on the web (53% according to [27]). Our evaluation studies users working with such hierarchical schemas.

3.2.2 The Semantic Web and Web Data Extraction

10.5 %

There has been a great deal of work on both encouraging and extracting structured data on the web [61]. However, automatic scraping techniques often have errors because they must infer structure from unstructured or poorly structured text and HTML markup. Several efforts have been made to define syntaxes and schemas, such as RDFa [62] and Microdata [63], for publishing structured data in web pages to contribute to the *Semantic Web* and *Linked Open Data* [64]. However, novice users have had little incentive to adopt these standards—sharing data rarely provides direct benefit to them—and find them difficult to learn, potentially contributing to their limited adoption on the web. Mavo contributes to this line of work by using a standards-compliant syntax that is machine-readable. Authors typically do not care about theoretical purity and are motivated to add additional markup when they see a tangible benefit. With Mavo, they expend effort because it makes their static website editable or creates a web application. As a side effect, however, they enrich the Semantic Web.

3.3 The Mavo Language

10.6 %

A description of the Mavo language follows. Its expression syntax, Formula^2 is described in detail in Chapter 4, so here we will focus on how Mavo HTML embeds formulas, and not their syntax. Similarly, its storage location is specified as a Madata URL (discussed in Chapter 5), so here we will focus on how Mavo HTML interfaces with Madata, and not the specifics of where data is stored.

We chose to use **declarative, HTML-based syntax** instead of new syntax for Mavo functionality because our target authors are already familiar with HTML elements, attributes, and classes and because HTML is inherently fault tolerant (Section 1.4.4) Whenever possible, we reused concepts from other parts of HTML. Using HTML5 as the base language also means a WYSIWYG editor for Mavo applications can be easily created by extending any existing WYSIWYG HTML editor — in fact, we discuss an attempt at this in Chapter 9. But as discussed previously, we consider it a key contribution of Mavo that it is a system-independent *language*. For example, we expect most Mavo authors to frequently take advantage of the ability to "view source" and work with arbitrary HTML.

"View source" is an essential methodology for learning and adopting new elements of web design. It permits authors to copy and tweak others' designs (even without fully understanding them) without worrying about new or conflicting system dependencies [27]. Source editing is essential to let authors circumvent any limitations imposed by graphical editing tools.

Per [18] and our Design Principles (Section 1.4.1), we want a low *threshold* (cost to get started) while allowing users escape the low *ceiling* (maximum achievable power) of GUI-based tool builders.

3.3.1 Creating a Mavo Application 10.8 %

To specify Mavo functionality on an HTML structure, the author places an `mv-app` attribute on the enclosing element. Its (optional) value provides an identifier that can be used to refer to this app's data from other Mavo apps on the page, and is used in a variety of other places. If not specified, Mavo looks at the element's HTML `id` attribute, and defaults to a generic identifier (e.g. `mavo3`) if that's also not present.

3.3.2 Data Persistence 10.9 %

By default, Mavo does not store data anywhere, which can be useful for calculator-type applications. Authors can specify a storage location via the `mv-storage` attribute.

The syntax of this attribute is a thin abstraction over a Madata URL (Chapter 5) to make these easier to specify for novices, and to provide sensible defaults, since Mavo has more information about the use case at hand.

For example, to store data locally in the browser, the Madata URL is `local:foo`, where `foo` is a unique identifier for the data store. To alleviate novices from having to understand what a custom protocol is or from the cognitive tax of finding a suitable key, Mavo uses a `local` keyword instead, and defaults the key to the app's identifier.

Similarly, to store data in a remote service, e.g. GitHub, Madata expects a URL that identifies a specific storage location (e.g. a repository and file path). These do not need to exist — for storage services with predictable resource URLs (such as GitHub), Madata will create any resources needed (e.g. files, repositories, etc.). However, simply having to choose suitable values can still be taxing to the novices who are be using GitHub as a backend because of its capabilities, but do not necessarily understand its constituent concepts. To make this easier, Mavo includes a set of defaults for each backend that facilitate underspecified URLs. To reuse the GitHub example, authors can specify a URL to their GitHub profile (e.g. github.com/leaverou), and Mavo uses defaults for the repository name (`mv-data`) and file name (`<app-id>.json`). That said, not every backend is conducive to this — for example a Dropbox URL does not have a predictable structure, and thus for Dropbox to be used, authors first need to upload an empty file and get a link to it.

Data Loading Overrides

11.1 %

By default, `mv-storage` specifices *both* the data source, and the data destination. If data does not yet exist, an empty dataset is rendered, and the data is created upon saving. However, there are certain common use cases that require more complex logic when *reading* data (storing data always goes to `mv-storage`).

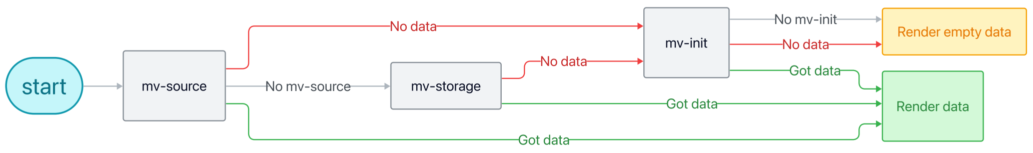


Figure 3.2 *A summary of the data loading algorithm.*

Some applications need *different backends for reads and writes*. These include applications that transform data and store the result elsewhere, certain performance optimizations, or simply read-only applications. To enable these use cases, Mavo supports an `mv-source` attribute, whose syntax is identical to `mv-storage`, but it has precedence over `mv-storage` for loading data.

For many types of CRUD applications, starting from empty data provides a poor user experience. Using a certain *default dataset if storage is empty* can help users better understand the application's purpose and functionality, or simply facilitate experimentation, while still allowing the user's own data to take precedence once it exists. Mavo supports an `mv-init` attribute for this purpose, which is only used if the main data source (either via `mv-storage` or `mv-source`) is empty or unavailable.

3.3.3 Data Definition

11.3 %

Our approach to data definition means that end-users define their data by defining the way they want their data to look on the page. This is in contrast to many systems which expect their users to define their data model *first* and then map their model into a view. In the spirit of direct manipulation, Mavo users are manipulating their data schema by manipulating the way the data looks. We believe that our approach is more natural for many designers, permitting them to directly specify their ultimate goal: data that looks a certain way.

Once Mavo is enabled on an HTML structure, it looks for elements with `property` or `itemprop` attributes within that structure in order to infer the data schema. These elements are henceforth referred to as simply *properties*. If the HTML author is aware of semantic Web technologies such as RDFa [62] or Microdata [63], these attributes may be already present in their markup. If not, they don't need to understand either technology — authors are simply instructed to use a `property` attribute to *name* their element in order to make it editable and persistent. An example of this usage can be found in Figure 3.1.

When an element becomes a *property*, it is associated with a data value. This value is automatically loaded from and stored to the specified `mv-storage` location. For many elements (e.g. `<span>`), the natural place for this value to be "presented" is in the element's contents. In others, such as `<img>` or `<a>`, the natural place for a value is a *primary attribute* such as `src` and `href`. RDFa [62] defines a small set of such attributes (mainly `href` and `src`), which Microdata [63] extends with many more, which Mavo adopts.

Explicit Primary Attribute

11.5 %

There are many cases where the data value cannot be predicted by these heuristics. For example, consider the following Mavo HTML fragment:

HTML

```html
<a property="twitter_username" href="https://twitter.com/[twitter_username]">leaverou</a>
```

Since this is an `<a>` element, its primary attribute is `href`, but the data value is in the element content. To handle such cases, both RDFa and Microdata use a `content` attribute as an escape hatch: If a `content` attribute is present, it overrides both the element contents, and the primary attribute. However, this requires duplicating data across where it actually lives and the `content` attribute.

Instead, Mavo adds an additional rule to these semantics: an `mv-attribute` which allows authors to *specify* the default primary attribute, with a `none` value for the element contents (which is what we would use above).

Objects

11.7 %

Properties that contain other properties become *grouping elements* (objects in programming terminology); this permits a user to define deep hierarchical schemas implicitly, simply as a natural consequence of spatial containment. For example, an element with a `student` property can contain other elements with `name`, `age`, and `grade` properties, indicating that these properties "belong" to the student.

Collections

A core value proposition of Mavo is automating the large amount of repetitive UI code that CRUD applications require to manage collections of data. Interactions like adding items, deleting items (with undo), reordering items via drag-and-drop, plus keyboard handling for efficiency and accessibility for all of the above, are all automatically handled.

To convert a property into a collection, all that authors need to do is mark what should be the collection *item* — i.e. the element that will be repeated — with an `mv-multiple` or `mv-list-item` attribute, or mark the collection itself with an `mv-list` attribute.

During editing, appropriate controls appear for adding and deleting new elements in the collection, as seen for the to-do items in Figure 3.1. Collection items can themselves be complex HTML structures consisting of multiple data-carrying elements and even nested collections. This enables the author to visually define schemas with one-to-many relationships.

To author a collection, the author creates *one* representative example of a collection item; Mavo uses this as the archetype for any number of collection elements added later.

As discussed earlier, this archetype can contain real data so it resembles actual output and not just a template, and can also provide default data values for new collection members.

Collection Syntax

Originally, there was no HTML representation for the *collection* itself — `mv-multiple` was added to the collection *item* and was the only way to define a collection. This was not an issue for purely CRUD applications with very simple computations — it was just framed as *"`mv-multiple` makes an element repeatable"*.

Regardless, putting the attribute on the collection *container* instead was a common slip, although easy to self-correct from feedback (see Section 7.1). However, as noted in Section 7.2, when authoring expressions for hierarchical schemas the lack of an HTML element to host the entire collection created some confusion.

Additionally, this design created some conceptual inconsistencies. Other Mavo attributes (e.g. `mv-default` or `mv-value`) had to heuristically determine whether they were being set on the *collection* itself or its *items*, with awkward escape hatches.

As a result, Mavo later adopted a more explicit syntax involving two separate attributes: `mv-list` and `mv-list-item`. However, per Mavo's philosophy of flexibility and fault tolerance, collections can be defined with only one of the two, and the other will be inferred. If necessary, suitable container elements to hold the collection or its items will be crated.

Relationship to RDFa

12.1 %

The Mavo syntax for naming elements is based on a simplified version of RDFa we call *Loose RDFa* that is designed to prioritize learnability over generality.

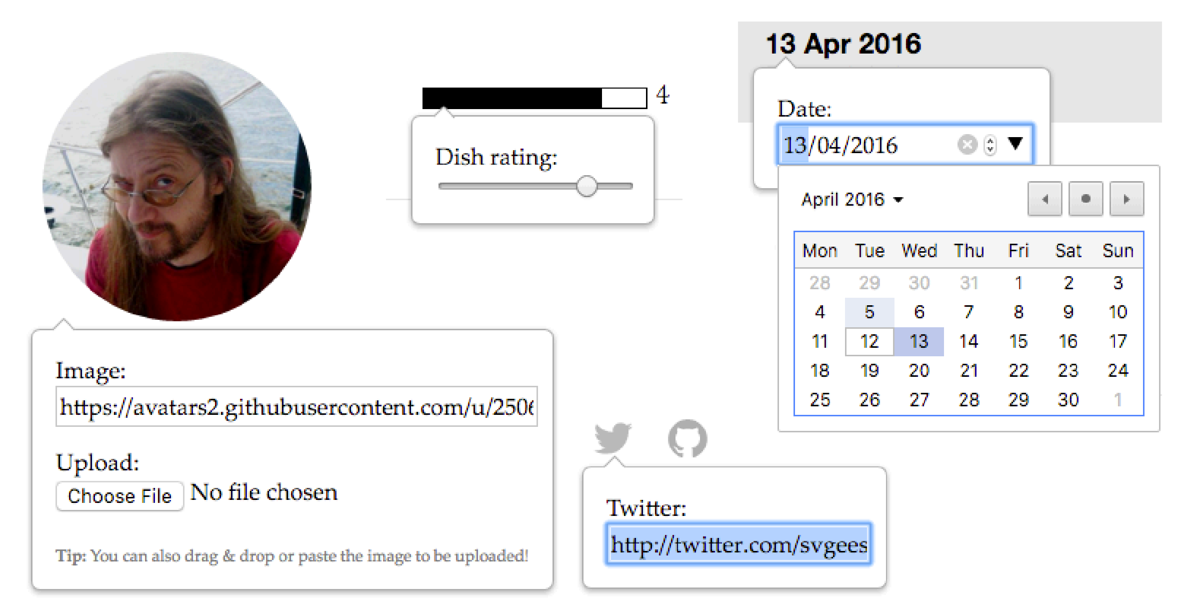
Its main differences from standard RDFa [62] are as follows:

1. Objects are inferred from the property structure instead of requiring a separate `typeof` attribute. Mavo then adds any missing `typeof` attributes. Authors can additionally add `typeof` attributes to explicitly declare objects for cases where the inference is incorrect.
2. Plain identifiers (rather than URIs or CURIEs [74]) are allowed even when no `vocab` is set.
3. `typeof` values are optional, to declare an object with no specific type.
4. The way primary attributes are inferred (see above), which extends RDFa's rather primitive heuristics.

3.3.4 Data Editing

12.2 %

Mavo generates UI (user interface) controls for toggling between reading and editing mode on the page, as well as saving and reverting to the last saved state (if applicable), as seen at the top of Figure 3.1. In editing mode, Mavo presents a WYSIWYG editing widget (called the property's *editor*) for any property that is (a) not already a form control, (b) not a computed property, and (c) not explicitly defined as read-only.



Date: 13/04/2016

April 2016

Mon	Tue	Wed	Thu	Fri	Sat	Sun
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	1

Image: https://avatars2.githubusercontent.com/u/250(

Upload: No file chosen

Tip: You can also drag & drop or paste the image to be uploaded!

Twitter: http://twitter.com/svgees

Figure 3.3 *Different types of editing widgets for different types of elements. Clockwise from the top left:* `<img>`, `<meter>`, `<time>`, `<a>`

In line with its philosophy of *implicit data schema definition*, **Mavo leverages available HTML semantics** to both optimize the editing interface and extract data type information. For instance, a `<time>` element is expected to hold temporal data, and will be edited via a date or time picker (depending on its `datetime` attribute), whereas an `<img>` element will be edited via a popup that allows specifying a URL or uploading an image (Figure 3.3).

Customization

12.3 %

These heuristics are intended to both reduce cognitive load on the author and to encourage the use of semantically appropriate HTML. However, per Mavo's design principles, inferred information should be escapable (Section 1.4.3). Indeed, both the data type and the editing widget can be overridden.

The inferred data type can be overridden by using the `datatype` attribute. This is an RDFs attribute, and thus does not need an `mv-` prefix, though Mavo uses it more loosely by accepting plain identifiers like `number` rather than CURIEs like `xsd:number`.

The generated editing UI can be customized in a variety of ways. For small tweaks, such as overridding an attribute, authors can specify attributes with an `mv-editor-` prefix, and they are copied to the generated editing widget. For more extensive customization, authors can provide their own editing widgets by linking to an existing element anywhere on the page via the `mv-editor` attribute, whose value is a CSS selector. If no data type is explicitly specified, this new editor element will be used for inferring it, just like default editors.

For example, if a property only accepts certain predefined values, authors can express this by linking to a `<select>` menu, essentially declaring it as an enum. Any changes to the linked form element are propagated to the property editors. This way, authors can have dynamic editing widgets which could even be Mavo apps themselves, such as a dropdown menu with a list of countries populated from remote data and used across multiple Mavo apps.

3.3.5 Computation

12.5 %

The aforementioned three attributes—`mv-storage`, `property`, and `mv-multiple` (or `mv-list`/`mv-list-item`) — are sufficient for creating any CRUD content-management application with a hierarchical schema and no computation. However, many CRUD applications in the wild benefit from lightweight computation, such as displaying a count of items, summing certain values, or conditionally showing text depending on a data value.

To accommodate these use cases, Mavo embeds reactive expressions in brackets (`[]`) in the HTML, as well as raw within certain attributes (`mv-value`, `mv-if`, etc.), called *directives*. Originally, Mavo supported two expression interpreters: (a) raw JavaScript (executed in a sandbox environment where properties become read-only variables), and (b) MavoScript, a Formula^2 precursor. Authors did not specify the flavor they were using. Instead, expressions would be first parsed as MavoScript expressions, and if that failed, as JavaScript expressions. This contributed to author confusion, and restricted the evolution of Formula^2, so it was quickly abandoned.

Our approach to expressions only partially meets the *declarative, direct manipulation* goal we described in our motivation. It is challenging to specify computation, an abstract process, entirely through direct manipulation. The expression language is similar to that in spreadsheets—fully reactive with no control flow, which nods towards declarative languages. The widespread adoption of spreadsheets provides evidence that this type of computation is within reach of a large population. Furthermore, placing the expression *in the document*, precisely where its value will be presented, as opposed to referencing values computed in a separate model "elsewhere", fits the spirit of direct manipulation in specifying the view. During our several user studies many participants volunteered observations that this was effective.

Embedding Formula^2 Expressions in HTML

12.8 %

There are two ways to embed a Formula^2 expression in HTML.

The first is to use an `mv-value` attribute, containing the expression. This is more verbose than the second method, but allows providing a fallback value (which is also used *before* the expression is evaluated) and allows Mavo to provide visible *feedback* when the expression is invalid, as there is an element to apply CSS to.

The second is to enclose it in square brackets (`[]`), which allows placing it anywhere inside the Mavo instance, including in HTML attributes (but not element names — yet).

To avoid triggering unrelated uses of brackets on individual elements, authors can use the `mv-expressions` attribute to customize the syntax or disable expressions altogether (`mv-expressions="none"`). The setting is inherited by descendant elements, unless they have a `mv-expressions` attribute of their own. For example, for the double-brace expressions common in many templating libraries, authors can use `mv-expressions="{{ }}"`.

The choice of brackets for delineating expressions was based on the observation that non-programmers often naturally use this syntax when composing form letters, such as email templates. In addition, many text editors automatically balance brackets.

An earlier version of Mavo used a more spreadsheet-like `=(expression)` syntax, but we found from preliminary user studies that few users realized the spreadsheet connection and found it difficult to determine where an expression terminated due to parentheses also being used inside expressions.

Named References

13 %

Instead of referencing mysterious row and column coordinates (a common source of errors in spreadsheets [75]), Formula^2 (and by extension Mavo) refers to properties by name. Every property defined in a Mavo instance becomes a (read-only) variable that can be used in expressions **anywhere** in the Mavo instance. The algorithm is described in detail in Section 4.4.1. These named references are necessary since Mavo has no predefined grid for row/column references. We consider this necessity a virtue.

We believe this will decrease bugs caused by misdirected references. Indeed, many spreadsheets offer *named ranges* to provide this benefit of understandable references (although no-one used them in our user studies, see Chapter 7). For spreadsheets, perhaps the main benefit of the row-column references is having formulas with "relative references" (e.g. to adjacent columns) to automatically update as they are copied down into new rows. But Mavo's automatic duplication of templates in collections means copies are never made by the user, obviating the need for this benefit.

Dynamic Collections

13.1 %

Formula^2 is not restricted to primitive values, but can also operate on lists and objects (and combinations thereof). Mavo harnesses this via the `mv-value` attribute. We saw earlier how `mv-value` can be used to specify scalar expressions with a fallback. However, `mv-value` follows the schema of the element it's being used on (adapting the data if necessary, see Section 3.6.2). When used on a collection, it renders its value as collection data and reactively re-renders the collection items whenever needed. Combined with Formula^2 list-valued semantics, this can produce powerful results with very little code. Furthermore, this dynamic collection can also be named and referenced elsewhere, creating a basic but powerful abstraction mechanism.

For example, suppose our To-Do app had a `priority` property for each task, and we wanted to display the number of tasks per priority (a pivot table). It could be as simple as:

HTML

```html
<dl property="tasks_per_priority" mv-list mv-value="task by priority">
  <dt property="priority">(No priority)</dt>
  <dd mv-value="count(task)">0</dd>
</dl>
```

3.3.6 Reactive Defaults

13.5 %

It is worth noting that in collections, the example item can be filled with real data, which makes the template really look like the output, unlike other templating languages where the template is filled with visible markup. In addition, this example data can easily become default values, by using the `mv-default` attribute without a value. Specifying a value for `mv-default` will set the default value to that, which is useful for static defaults.

However, `mv-default` becomes very powerful when combined with expressions, creating *reactive defaults*. To our knowledge, Mavo is the first novice programming language to provide reactive defaults.

When a property is set to a reactive default, it gets updated whenever the default value changes, unelss it has been edited by the user. This effectively provides a data/formula hybrid which enables many use cases that are otherwise cumbersome or even impossible.

The more obvious use case is facilitating smart defaults that are progresssively refined as other properties are edited. For example, suppose we have a calendar application whose events have `start_date`, `end_date`, `start_time`, `end_time` properties. `end_date` could default to `start_date` if `end_time` is after `start_time`, and to one day after `start_date` otherwise. It would look like this:

HTML

```html
<time property="end_date" mv-default="[ if(end_time > start_time, start_date, start_date + 1 * day())
]">
```

However, there is a large set of use cases where fields are essentially computed via an expression, but there are some few exceptions where the user might want to override the computed value. Using `mv-default` rather than `mv-value` in these cases preserves editability and allows the user to override the computed value.

3.3.7 UI Customization

13.7 %

We designed Mavo to be useful to HTML authors across a wide range of skill levels, including web design professionals. This presents a tension: Web designers want *control*, whereas novices want UI to be generated requiring as little involvement as possible.

The approach that Mavo follows is to generate a default UI, but expose hooks for customization at various levels of the ease-of-use to power curve. Smaller customizations can be done by simply styling the generated Mavo elements with CSS.

In addition, authors can provide *their own* UI elements that replace those generated by Mavo by using certain class names (such as `class="mv-add-task"` for a custom "Add task" button).

For the parts of Mavo most likely to be customized, such as the editing widgets or the toolbar, Mavo also provides specific attributes for high level customization with low effort (e.g. see Section 3.3.4.1 for customizing the editor).

> **NOTE**
> An earlier version of Mavo also supported nesting form elements inside properties to *implicitly* define an editor without the indirection of linking to a form element elsewhere, akin to nesting a form element in a `<label>` to implicitly associate the two without using a `for` attribute. However, practice showed that this frequently conflicted with the author's intent for the form element to be a part of the template, and not an editing hint, and thus it was removed.

3.4 Implementation

13.9 %

Mavo is implemented as a JavaScript library that integrates into a web page to simulate native support for our syntax. On load, Mavo processes any elements with an `mv-storage` attribute and builds an internal *Mavo tree* representation of the schema (Figure 3.4). It also inspects every text and attribute node on or inside every element looking for expressions, and builds corresponding objects for them. For every expression, a JavaScript parser is used to rewrite binary operations as function calls in order to enable array arithmetic.

Any remote data specified in the `mv-storage` attribute is then fetched via Madata and recursively rendered. Every time an object is created during editing or data rendering, it holds a reference to its corresponding node in the Mavo tree (Figure 3.4), which it uses as a template. This improves performance by only running costly operations (such as finding and parsing expressions) once per collection.

When the data in an object changes — via rendering, editing or expression evaluation — expressions within it or referring to it are re-evaluated to reflect current values. This occurs in a special execution context where current object data and Mavo functions appear to be

global scope. ECMAScript 2015 Proxies are used behind the scenes to conditionally fetch descendant or ancestor properties only when needed, to allow for identifiers to be case insensitive, to make the variables read-only, and to allow identifier-like strings to be unquoted.

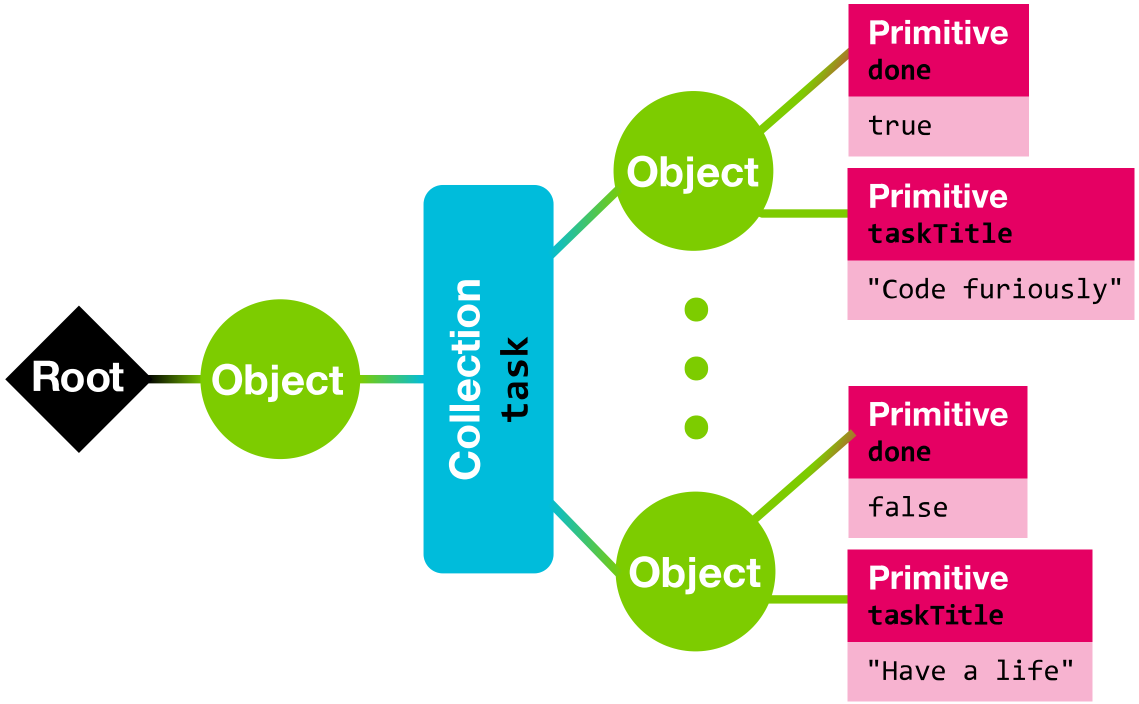


Figure 3.4 *The Mavo tree created for the To-Do app shown in Figure 3.1.*

3.4.1 Extensibility

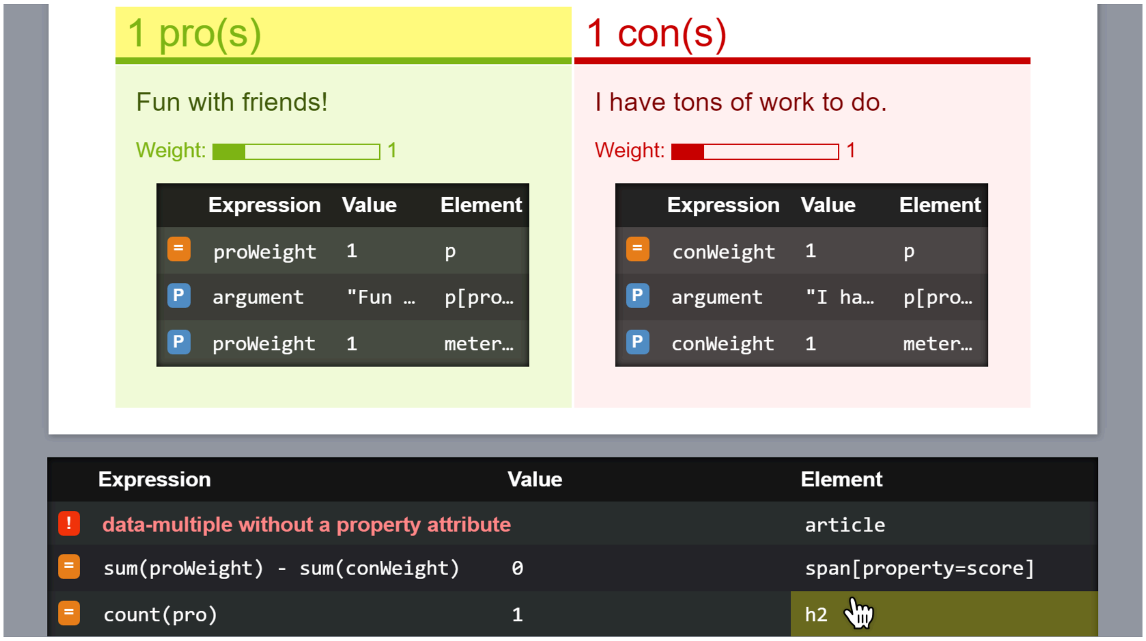
14.2 %

There are APIs in place for third-party developers to add new default editing widgets, new expression functions, and new storage backends. In addition, Mavo includes a hooks system for developing plugins that modify how it works on a lower level. For example, both the inline Mavo debugging tools and most directives are implemented as plugins and could be removed.

3.4.2 Debugging

Inline Debugging Tools

Mavo includes debugging tools that show the current application state, as expandable tables inside objects (Figure 3.5). This is enabled by placing a `mv-debug` class on any ancestor element or adding `?debug` to the URL. These tables display current values of all properties and expressions in their object, and warnings about common errors. Expressions shown can be edited in place, so that users can experiment in real-time.



1 pro(s)

Fun with friends!

Weight: 1

	Expression	Value	Element
=	proWeight	1	p
P	argument	"Fun …	p[pro…
P	proWeight	1	meter…

1 con(s)

I have tons of work to do.

Weight: 1

	Expression	Value	Element
=	conWeight	1	p
P	argument	"I ha…	p[pro…
P	conWeight	1	meter…

	Expression	Value	Element
!	data-multiple without a property attribute		article
=	sum(proWeight) - sum(conWeight)	0	span[property=score]
=	count(pro)	1	h2

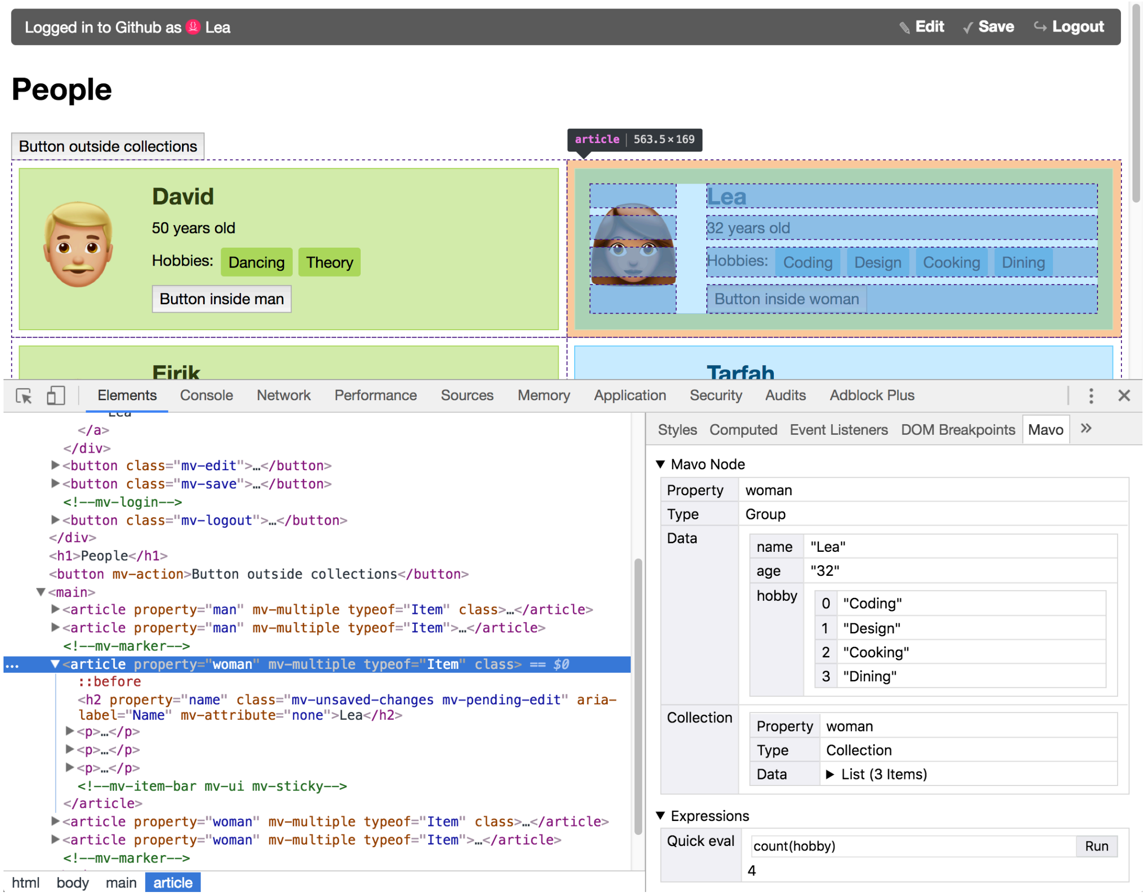
Figure 3.5 *The debug tools in action, showing local values and warnings.*

Mavo Inspector

While inline debugging tools may appear to work better with novices, they do produce a lot of clutter. Furthermore, our first user study reported low levels of author engagement with those early debugging tools.

We later also implemented a Google Chrome devtools extension (Figure 3.6) for inspecting the values of Mavo Nodes and experimenting with expressions. The extension adds a "Mavo" tab to the Element inspector of the built-in browser developer tools that displays current values, and allows one-off execution of expressions and data updates. Since property references in Mavo expressions are context sensitive, we used this inspector in our evaluation to demonstrate context sensitivity to our subjects, and later on in the study to allow them to quickly experiment with expressions in training tasks.

Later user studies used the Mavo Inspector instead, both to explore data, and to allow subjects to quickly experiment with expressions in training tasks.



```
      </a>
    </div>
  ▶<button class="mv-edit">…</button>
  ▶<button class="mv-save">…</button>
    <!--mv-login-->
  ▶<button class="mv-logout">…</button>
  </div>
  <h1>People</h1>
  <button mv-action>Button outside collections</button>
 ▼<main>
  ▶<article property="man" mv-multiple typeof="Item" class>…</article>
  ▶<article property="man" mv-multiple typeof="Item">…</article>
    <!--mv-marker-->
…  ▼<article property="woman" mv-multiple typeof="Item" class> == $0
      ::before
      <h2 property="name" class="mv-unsaved-changes mv-pending-edit" aria-
      label="Name" mv-attribute="none">Lea</h2>
    ▶<p>…</p>
    ▶<p>…</p>
    ▶<p>…</p>
      <!--mv-item-bar mv-ui mv-sticky-->
    </article>
  ▶<article property="woman" mv-multiple typeof="Item" class>…</article>
  ▶<article property="woman" mv-multiple typeof="Item">…</article>
    <!--mv-marker-->
```

▼ Mavo Node

Property	woman
Type	Group
Data	name "Lea" age "32" hobby 0 "Coding" 1 "Design" 2 "Cooking" 3 "Dining"
Collection	Property woman Type Collection Data ▶ List (3 Items)

▼ Expressions

Quick eval	count(hobby) Run 4

Figure 3.6 *The Mavo Inspector Chrome extension with a collection item selected in the Elements panel and the expression* `count(hobby)` *evaluated in that context .*

3.5 Discussion

14.4 %

In this section we discuss various issues brought up in our design and study of the Mavo language, and other potential future directions.

3.5.1 Data Formatting

Currently, data formatting is automatic. The only type that supports custom formatting is the `<time>` element, which can optionally nest an expression for presenting the date/time to override the default:

HTML

```html
<time property="start">[date(start)] at [time(start, "seconds")]</time>
```

For example, numbers are automatically formatted in a locale-aware way, with thousands separators and two decimal places. However, there are cases where authors might want to override this formatting. For example, in a recipe app, one would typically want to list ingredients with fractional quantities like “⅜ cups”, rather than “0.375 cups”. This is already possible by redirecting the number value to an attribute, and using a custom expression as the element contents:

HTML

```html
<span property="amount" mv-attribute="data-value">[custom_format(amount)]</span>
```

However, this is essentially a workaround, as its closeness of mapping [76] is poor. A more direct way to specify formatting would be beneficial, e.g. an attribute dedicated to this, alongside a richer set of built-in formatters.

3.5.2 Direct Manipulation of Data Schemas

14.7 %

Mavo’s approach of designing schmemas by designing the presentation of the data from those schemas works well because the presentation of data usually reflects its schema. If we have a collection of objects with properties, we generally expect those objects to be shown in a list, with each object’s properties presented inside the space allocated to that object. This is understandable, as the visual grouping conveys relationship to the viewer. We are simply inverting this process, arranging for the visual grouping to convey information to the underlying data later. Mavo may not be suitable for creating presentations that conflict strongly with the underlying data schema, should such presentations ever be wanted.

3.5.3 Target Users

14.8 %

Mavo is aimed at a broad population of users. There is no hard limit to what it can do, since JavaScript can be used as an extension point. However, this is not the primary target. Our focus is increasing the power payoff for a given investment of effort/learning that is accessible to novices (Section 1.4.2). Currently, even small web applications require substantial skill and effort to build. Too often, designers of essentially static websites are forced to deploy them inside CMSs, only so that their non-technical clients can update the site content. Mavo frees designers from these CMS constraints by providing an automatic WYSIWYG content management UI for plain HTML. Plain CRUD apps only need `mv-*` attributes “entirely in HTML” without application logic.

For users who want more, expressions add power: lightweight computation for application logic at a conceptual cost similar to spreadsheets. More complex functions provide more power, like advanced spreadsheet ones. Our user study traced out this ease/power curve and showed that most users can work with such expressions.

Although we have focused on Mavo as a tool to support non-programmers, skilled programers can also benefit from the ability to rapidly build dynamic CRUD interfaces. Even for programmers Mavo brings some of the benefits of *data typing* to the construction of the interface: declaring data types enables the system to provide appropriate input and data management without demanding that the developer write special purpose code for the typed content.

3.5.4 Handling Large Amounts of Data 15 %

The existing Mavo backends save all data in a single JSON file. This is convenient for use cases involving small amounts of data, and allows using any popular file hosting web service as a backend. However, making multi-user apps possible will create a pressing need for handling larger amounts of data with Mavo, even to create entire social websites

Mavo already supports displaying and editing part of the data, and already keeps track of what data has been modified, to highlight unsaved changes. Therefore, it is easy to implement incremental saving for web services that support it. Implementing a backend adapter for a cloud database service (e.g. Firebase), will also allow for fetching partial data. Such backends usually also support server-sent events, which would enable incremental updates for true bidirectionality.

3.5.5 Encouraging Semantic Web Best Practices 15.1 %

Mavo encourages semantic web practices by providing incremental value to authors when they add semantic markup. For example, using suitable HTML elements will produce more suitable editing widgets, and using good property names is *essential* for an editing UI that makes sense, since they are used in a number of places in the generated editing UI: button labels, tooltips, and input placeholders to name a few.

Additionally, its storage functionality typically produces structured, machine-readable data, contributing to the Semantic Web through another avenue. In fact, if a `vocab` attribute is used in the markup, Mavo will produce valid JSON-LD [77].

It could be argued that because of its looser syntax (Section 3.3.3.4), data produced by Mavo app users are not useful. However, we believe that reducing the burden of creating structured data and increasing the author perceived value of doing so is the only way to encourage widespread adoption of structured data practices, and a large quantity of imperfect structured data is more useful than a small quantity of perfect data created by the few users with intrinsic motivation to contribute to the Semantic Web.

3.6 Future Work

15.2 %

3.6.1 Separate Name for Collection and Collection Items

Using plural instead of singular nouns and vice versa is a common slip of Mavo authors, since currently the same name is used for both the collection itself, and each item.

As noted in Section 3.3.3.3.1, the current syntax for defining collections consists of two attributes: `mv-list` and `mv-list-item`. While that was not the motivation for the change, it opens up the possibility of defining a separate name to refer to the collection and its items (currently an authoring mistake handled by precedence rules).

This would allow using a plural name for the collection itself, and a singular alias for the items. This would also allow referencing the collection from within each item without having to do `propName.all`.

3.6.2 Handling Schema Mutations

Mavo's innovation of inferring schema from HTML presentation might be its Achilles' heel. After Mavo is used to create data, changes to the HTML may result in a mismatch between the schema of the saved data and the new schema inferred from the HTML, which could lead to data loss (or perceived data loss, where data is retained but not displayed, creating the appearane of data loss). Similar mismatches can occur when connecting Mavo apps to third-party data with a pre-existing structure that was not created by the Mavo app. Mavo does automatically handle some changes, such as:

- When properties are added, the schema is automatically extended to include them.
- When properties are *removed*, corresponding data is retained and saved, but not displayed. This protects a user from data loss if they stop displaying a property then bring it back later. It also enables the creation of multiple Mavo applications operating on different parts of the same dataset.

- When a singleton is rendered onto a collection, Mavo heuristically either converts the single item to a collection of one item or renders a list-valued property of the singleton, depending on their structure.
- When a collection is rendered onto a singleton (e.g. by removing the `mv-list`/`mv-list-item` attributes), the data is retained so it can be brought back later but anything after the first item is not displayed and cannot be edited or referred to in expressions.

However, if the disparity between the data schema and the inferred schema is too great, Mavo will not be able to automatically reconcile them without author intervention. Such intervention includes:

- Using `mv-alias` to read data from a property with a different (older) name.
- Using `mv-path` to "fast-forward" to a property that is now nested deeper in the schema.

More complete handling of schema changes and/or better author feedback about how they are handled are key open questions for Mavo.

3.7 Conclusion

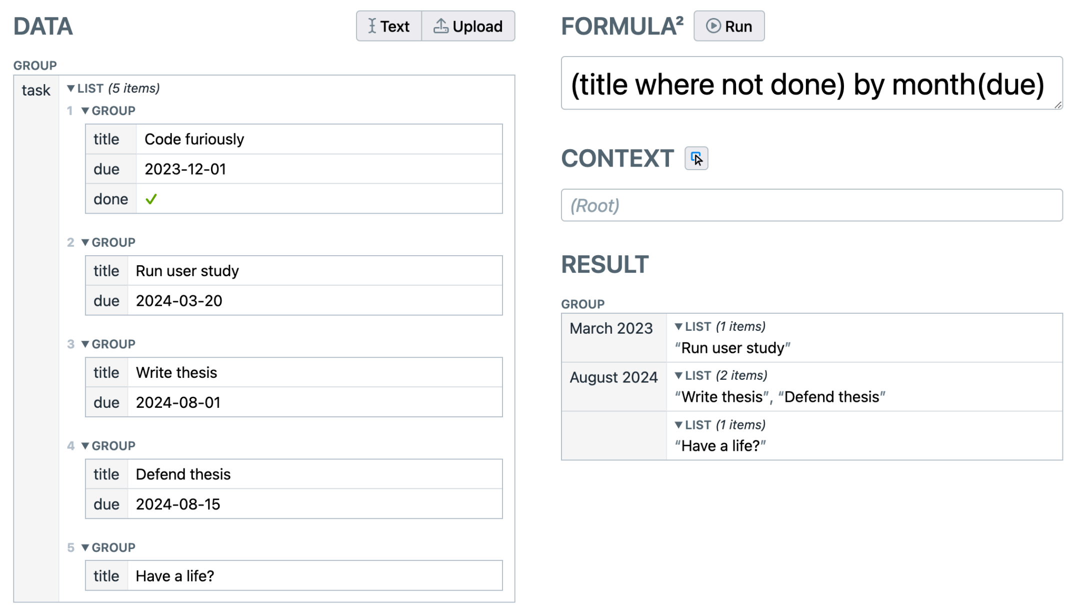
15.6 %

This chapter presents Mavo, a system that helps end-users convert static HTML pages to fully-fledged web applications for managing and transforming structured data. Our user studies showed that HTML authors can quickly learn use Mavo attributes to transform static mockups to CRUD applications, and, to a large extent, use Mavo expressions to perform dynamic calculations and data transformations on the existing data.

CHAPTER 4

Formula^2: A Human-centric Hierarchical Formula Language

9,425 words (27 min read)



GROUP

task ▾LIST *(5 items)*

1 ▾GROUP

title	Code furiously
due	2023-12-01
done	✓

2 ▾GROUP

title	Run user study
due	2024-03-20

3 ▾GROUP

title	Write thesis
due	2024-08-01

4 ▾GROUP

title	Defend thesis
due	2024-08-15

5 ▾GROUP

title	Have a life?

(title where not done) by month(due)

(Root)

GROUP

March 2023	▾LIST *(1 items)* "Run user study"
August 2024	▾LIST *(2 items)* "Write thesis", "Defend thesis"
	▾LIST *(1 items)* "Have a life?"

Figure 4.1 *A Formula² formula operating on a shallow hierarchical schema, showcasing implicit scoping (`title`, `done`, `due`), filtering & grouping operators, and temporal computation. Parentheses around the `where` expression are added for clarity — operator precedence rules would produce the same result.*

4.1 Introduction

Spreadsheets introduced reactive (aka *continuous evaluation* [36]), side-effect free formulas to the massses, and despite the well-known limitations of spreadsheets [78], no other end-user programming environment has managed to surpass their popularity.

Despite hierarchical data naturally occurring in more than half of data structures novices organically create [27], few formula languages support hierarchical data structures. Moreover, the few that do often prioritize compatibility with spreadsheets over natural programming and general HCI principles.

Formula^2 (*MavoScript* in earlier literature [11]) is a formula language designed from scratch to support hierarchical data structures in a way that is natural for novices and reduces the amount of abstract thinking about data required.

However, if the research question was simply *"How can we design a natural formula language?"*, the answer is simple — no structured language can beat natural language at that — almost by definition. But parsing natural language is resource-intensive and error-prone, so instead we designed Formula^2 to answer the question *"What would a formula language look like if it were designed from scratch today, with the explicit goals of prioritizing usability, natural programming principles [16, 70], and making common use cases easy, while still maintaining reasonable parsing and evaluation complexity?"*

Indeed, while our user studies repeatedly demonstrated that Formula^2 was easier for novices than alternatives, it can be parsed very efficiently with a simple Pratt parser [79].

While Formula^2 was originally developed for Mavo, it has no dependency on any particular host language or system. It could be used on hierarchical spreadsheet-like applications, compiled to JavaScript and used by JavaScript frameworks, or even used in a standalone environment against arbitrary data (see Figure 4.1). Consistency with JavaScript or other web technologies was not a primary design consideration for its syntax, but was used as a tie-breaker for otherwise equally valid design decisions.

4.2 Related Work

16.1 % Since LANPAR introduced reactive formulas in 1969, and VisiCalc augmented them them in 1979 with the A1 notation and most of the features we know today [78], a lot of work has been done to improve on their design.

4.2.1 Hierarchical Data in Spreadsheets

Gneiss [34, 80] was a spreadsheet system supporting hierarchical data. However, the bulk of its contributions were in the user interface. Its formula language prioritized compatibility with spreadsheets and thus, included the minimum amount of changes needed to support hierarchical data. References to data still used the obscure A1 notation, just extended it for nested data.

4.2.2 Reactive Formulas Operating on Databases 16.2 %

SIEUFERD [32] provided a spreadsheet-like hiearchical data interface to a SQL database, and thus had to design a formula language that could handle hierarchical data. While it used readable names rather than A1 notation, compatibility with spreadsheets was a primary design goal. More importantly, the primary means of querying list-valued data was its interface for visually constructing SQL queries, and thus the formula language is purposefully restricted to scalars as a return value.

4.2.3 Reactive JavaScript Frameworks 16.3 %

Spreadsheet formulas are not the only syntax in wide use for reactive formulas. While spreadsheet formulas are the most popular end-user programming syntax for reactive formulas, this section would not be complete without mentioning the many reactive JavaScript frameworks, such as VueJS [57], Angular [56], or Svelte [81] These frameworks typically support embedding a restricted subset of JavaScript in HTML, and detect dependencies by parsing the JavaScript code.

4.2.4 Other Dataflow Languages

There are many other dataflow languages, such as LabView [82], a visual programming language for dataflow programming. However, these are typically not designed for end-users, and are often not reactive in the same sense as spreadsheet formulas. The usability issues with LabView are well-documented [76].

Mashroom [83] was a dataflow language designed for mashups, and operating on nested tables. While some of its operators were similar to Formula^2, and it also centered around aggregation, it required a lot more technical knowledge to use and targeted different use cases.

4.3 Syntax and Core Concepts

16.4 %

In this section we summarize Formula^2's core concepts and syntax. A more detailed design discussion can be found in Section 4.5.

4.3.1 Separation of Concerns

We define the ***host environment*** as the environment in which Formula^2 is embedded. This could be a spreadsheet-like application, a no-code visual app builder, a web framework, or a standalone application that allows the user to directly import data, specify Formula^2 expressions, optionally select the *context node* (see Section 4.4.1), and see the result (Figure 4.1). We have already seen one host environment, Mavo HTML.

Since Mavo is the only host environment of Formula^2 that we have seen so far, it may be unclear where Formula^2 ends and the host environment begins. Formula^2 is responsible for (a) parsing and compling expressions, and (b) evaluating them against an arbitrary data tree. To maximize flexibility and make it easier to adopt by a variety of host environments, monitoring dependencies and re-evaluating expressions at the appropriate times is handled by the host environment.

It could be argued that this makes Formula^2 simply a functional, side-effect free language, since it is the host environment that may (or may not) implement it reactively. For example, the testing host environment shown in Figure 4.1 does not implement any reactivity. However, we believe this to be an implementation detail. The language design and semantics *assume* a reactive implementation. The fact that it can function (and even be useful) without reactivity is a testament to its flexibility.

4.3.2 Syntax and Semantics

16.6 %

An explicit design goal was to minimize the number of syntactic primitives that need to be learned, and instead allow as many combinations of these primitives as possible.

- **Literals**: Numbers, strings (enclosed in single quotes, double quotes, or none (see below)), booleans (`true`/`false`), empty values (`none`)
- **Identifiers**: Any string of letters, numbers, or underscores. `$` is also allowed, but is reserved for predefined Formula^2 identifiers.
- **Function calls** are invoked with the usual syntax of `functionName(arg1, arg2, ...)`. Commas are optional, but encouraged.
- **Operators**, which can be unary, binary, ternary, or n-ary.

Complex data types such as lists and groups are also constructed using functions and operators.

4.4 Core Contributions

16.7 %

In this section, we present the core concepts that position Formula^2 in the landscape of reactive formula languages.

4.4.1 Implicit Scoping and Aggregation

Consider the use case shown in Figure 4.1. If we were using JavaScript, the expression could look like this:

JS

```
tasks.reduce((acc, task) => {
   (acc[month(task.due)] ??= []).push(task.title);
}, {})
```

Going beyond syntactic differences, referencing in JavaScript (and most other programming languages) is *shallow*: we cannot reference any task properties without getting ahold of each task object, and we could do that without iterating over the list of tasks. There are two levels of indirection to go from the data root to the data we are interested in. This approach minimizes conflicts, but at the cost of verbosity and complexity.

In natural language, **scoping is a way to optionally narrow, not a prerequisite for meaning**. Any known object can be referenced, and the context of the conversation is implicitly used to resolve it. This process may even involve iteration or even a recursive walk through the hierarchy of known objects, yet humans handle it effortlessly.

We can say *"put all dirty dishes in the dishwasher"* without needing to specify whether the dishes are on the table, in the sink, or in the bedroom. We can *optionally* scope it further, e.g. *"put all dirty dishes* ***from the dinner table*** *in the dishwasher"*, but the scope is optional, and the *dirty dishes* concept has meaning without it, just broader.

In contrast, while programming languages also have a notion of context, references are typically more restricted. Context typically simplifies references to identifiers directly within the current scope, or in ancestor scopes, but that's about it. Getting descendant or sibling data typically requires a series of recursive mapping operations. To word the dirty dishes example using the concepts of most programming languages, we would have needed to say something like:

> *"Visit every room in the house and do the following. If there are any dirty dishes on the floor, put them in the dishwasher. Look for objects with horizontal surfaces, then look for dirty dishes on them. If you find any, put them in the dishwasher. Now look inside all containers larger than a plate. If you find any dirty dishes, put them in the dishwasher. If you find other large enough containers, repeat the process for them too."*

Imagine how tedious communication would be if we had to speak like this! And yet, we have accepted that this is a reasonable way to communicate with computers.

This could explain why scoping and referencing errors are so common among beginner programmers [67–69]. The need to understand scoping rules, write out lengthy namespaces, or — worse — mapping operations when multiple values are desired are all barriers to entry. To alleviate this, Formula^2 uses a novel scoping algorithm that will prioritize explicit references over implicit ones, but will attempt to resolve any known identifier to the most reasonable author intent, taking into account both the structure of the formula, the data schema, as well as the *placement of the formula relative to the data*.

In Formula^2, **any known data identifier can be used anywhere**. However, the value(s) it resolves to depends on the *placement* of the formula. Every evaluation of a Formula^2 expression can be associated with a *context node* in the data.

This is in contrast with scoping in most programming languages, which is either lexical (derived from the code hierarchy) or dynamic (derived from the call tree). Typically the context node would have a spatial association with the formula in the host environment, for example in Mavo it would be the closest containing property. In a spreadsheet it could be the cell the formula is in.

A given identifier can resolve to a single value or a list of values, depending on the relative locations of the context node, the data it references, and the data schema.

Conceptually, to resolve the value of an identifier, we perform a breadth-first search[1] on the data to find the shortest path(s) from the context node to properties with the given identifier. We then resolve these paths to values. If there are more than one paths, or the paths cross any arrays (even if they only contain one element), the result is a list of values.

If the property is not found at all, then ancestor nodes are searched for the property.

While this algorithm may sound complex, in practice it affords a more natural way to reference data, where references largely *just work* and scoping rarely needs to be explicitly considered.

4.4.2 Transparent List-valued Operations

17.4 %

Since lists are such common return types, built-in support for list operations is essential. But Formula^2 goes a step beyond that, and attempts to **blur the line between scalar and list operations**. Its answer to "Am I dealing with a list or a scalar?" is *"if you don't already know, it shouldn't matter"*, in the sense that the author providing a list or a scalar communicates *intent*, and that intent is generally honored, but whether a function's return value is a list or a scalar should largely be irrelevant, except when it's a predictable consequence of said author intent.

Part of this is a natural consequence of the scoping algorithm. Since `foo.bar` looks at descendant properties via a breadth-first search, the result will be the same whether `foo` is a single object with the `foo` property or a list containing said object as its only element (since Formula^2 lists don't include arbitrary data properties).

Nearly all Formula^2 operations (functions and operators) are list-aware. Operations on lists, between lists, or between scalars and lists require no special syntax or functions. They largely *just work*.

The algorithm used is simple. Operations (including functions) are defined in terms of their scalar equivalents.

[1] This is a simplification and would result in very poor performance. In practice, the Breadth-First Search is performed on a pre-computed schema of the data, and then the paths are resolved on the actual data.

1. When both operands are scalars, the scalar operation is applied.
2. When only one operand is scalar, the result is a list where each list item is the result of applying the operation to the scalar and the list item.
3. When both operands are lists, the operation is applied element-wise. If the lists have different lengths, `null` is used for missing values.

> EXAMPLE
>
> For example, if `rating` is a list of ratings across items, `rating > 3` returns a list of boolean values, with `true` for ratings over 3 and `false` for those equal to or lower than 3. This result can then be fed to a `count()` function, so that a nicely readable `count(rating > 3)` returns the number of items with a rating over 3.

Lists returned from operations are flattened, so that all data is either scalars or lists of scalars, i.e. authors do not need to deal with lists of lists. This was chosen to limit complexity, as lists of lists are less common in naturally occurring data structures [27], and create amgibuities for many operations (e.g. what should `count(list)` return if `list` is a list of lists? The number of lists or the total number of elements?).

4.5 Detailed Design Discussion

17.7 %

We now present the design of Formula^2, focusing on its syntax and semantics.

4.5.1 Fault-tolerance & Flexibility

17.8 %

The design principle of fault tolerance (Section 1.4.4), permeates many of the design decisions of Formula^2:

- *Case-insensitivity*: Function names and identifiers are case-insensitive, so `rating`, `Rating`, and `RATING` will resolve to the same result.
- *Graceful references*: Using the narrowing (`.`) operator on an empty value, does not produce an error, just a result that is also an empty value.
- *Unquoted strings*: Unquoted strings are allowed, but are a last resort if an identifier cannot be resolved as data.

4.5.2 Special Properties

Most formulas depend on the values of other data, and should update when those values change. However, some use cases require access to context data that may change even if no data has changed, and thus should influence when a formula is recalculated, essentially adding itself to its dependencies.

For example, a formula that displays the current time in hours and minutes should update at least every minute, even if no data has changed. In spreadsheets, this is expressed as a function, `NOW()`. However, function calls do not update dependencies, so `NOW()` has the confusing behavior that it does not actually reflect current time *"now"*, but the time the row was last edited (which is a useful piece of information in its own right, but needs a name that better describes its purpose).

In Formula^2, such computations are expressed as special, built-in properties beginning with `$`. For example, the current time is expressed as `$now`.

Other examples of special properties are:

- `$mouse`, which provides the current mouse position and updates whenever the mouse moves.
- `$hash`, which returns the current URL hash (without the `#` sign) and updates when the hash changes.
- `$today`, which returns the current date and updates at midnight.

There are also several tree-structural special properties, which are used to navigate the data tree. For example, `$parent` to reference the parent of the current context node, `$previous` and `$next` to reference its siblings, `$all` to go from an item to its containing list, or `$index` to get the index of the closest list item (or any list item, when used as a property)

While the `$` prefix is designed to prevent clashes with author identifiers by partitioning the namespace, we noticed in user studies (see Chapter 7) that authors often forget to use it, presumably because this is an unnatural use for the `$` symbol, which in natural language is used for currency. It remains an open problem how to partition the namespace in a way that is more intuitive to authors, without making it overly verbose (as in the case of a reserved prefix, e.g. `f2_now`).

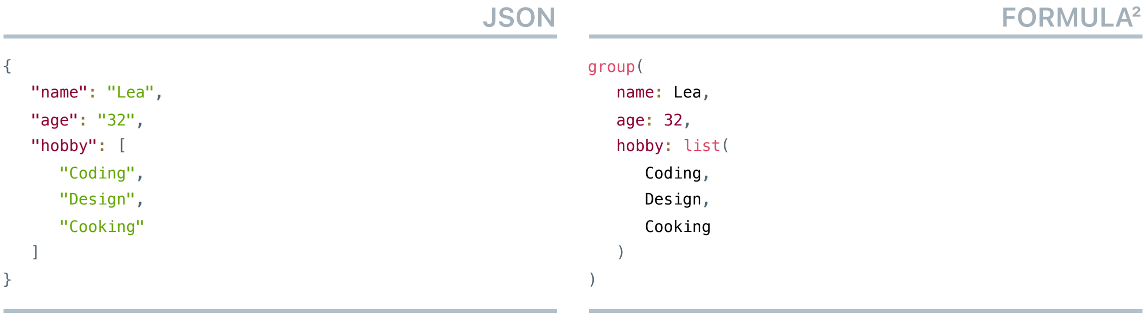
In line with its design principle of fault tolerance (Section 1.4.4), Formula^2 will attempt to resolve special property identifiers without the `$` prefix, but with a lower priority than those with the `$` prefix, only if no data properties exist with these names. It will also correctly resolve author identifiers that begin with `$`.

4.5.3 Data Types

18.1 %

Formula² supports the following data types:

- Numbers
- Strings, enclosed in single or double quotes. Unquoted strings are also supported if they only consist of identifier-compatible characters, but have lower precedence than data references. Since the schema can be mutated at runtime, this means their meaning could change if a new property is added anywhere in the data that matches the unquoted string.
- Booleans, `true` and `false`.
- Empty values (aka `null`)
- Lists (aka arrays or collections). Lists can contain any data type. There is no dedicated syntactic construct for lists, instead they are defined via a `list()` function.
- Groups (aka objects or dictionaries), which are sequences of key-value pairs. Groups are defined using a `group()` function and a colon to separate keys from values.



JSON

```
{
   "name": "Lea",
   "age": "32",
   "hobby": [
      "Coding",
      "Design",
      "Cooking"
   ]
}
```

FORMULA²

```
group(
   name: Lea,
   age: 32,
   hobby: list(
      Coding,
      Design,
      Cooking
   )
)
```

Figure 4.2 *A JSON object and the corresponding representation in Formula².*

The colon (`:`) is actually an operator that returns an object with a single key-value pair. The `group` function merely merges these objects. This has the very nice consequence that when there is only a single key-value pair, the `group()` function can be omitted.

4.5.4 Operators

18.5 %

Operators are essentially a nicer syntax to improve readability for certain functions, and to reduce the need for balancing nested parentheses. All operators are also implemented as functions (but the contrary is not true).

To improve learnability, only widely understood operators are symbols:

- common arithmetic operators: `+`, `-`, `*`, `/`
- common comparison operators: `>`, `<`, `>=`, `<=`, `=` (and `==`), `!=`,
- concatenation: `&`,
- the key-value pair operator: `:`
- the range operator: `..`

All other operators are words, including logical operators (`and`, `or`, `not`), a modulo operator (`mod`), filtering (`where`), grouping (`by` / `as`).

These are designed to eliminate many common types of errors across novice programming and formula languages [75, 84]:

- To avoid confusion between `=` and `==`, `=` and `==` are *both* used for comparison.
- It is common for `+` to do double duty: addition or concatenation, heuristically determined by the operands. This usually leads to errors, when the heuristics predict author intent incorrectly. Moreover, such a heuristic does not improve ergonomics since addition and concatenation are fundamentally distinct operations. In Formula^2, `+` is *only* used for addition, and `&` is used instead for concatenation. This allows `+` to work even when numbers are stored as strings, sparing novices from having to think about data types.
- Unlike most programming or formula languages, comparison operators are n-ary. For example, `3 < foo < 5` is perfectly legal and equivalent to `(3 < foo) and (foo < 5)`.

4.5.5 Operators That Affect Scoping

18.7 %

In certain cases, Formula^2's innovation of implicit aggregation becomes its Achilles' heel. A big class of these are operations whose operands are not independent, but one operand is implicitly scoped by the other.

Narrowing Operator (`.`)

18.8 %

The most trivial such case is the narrowing (or dot) operator, which is used to access properties of objects and to narrow down overly broad references.

With most operators, operands are resolved independently. For example, in the schema shown in Figure 4.1, a formula like `title & " " & due` would evaluate `title` to all task titles, `due` to all task due dates, and would then concatenate them element-wise separated by a space.

However, if the same approach were followed with the dot operator, it would produce nonsensical results. Imagine if given a reference like `students.name` we evaluated `students` and `name` separately!

Instead, `students` essentially adds a *constraint* to the resolution of `name`: it says "only consider `name` properties that are descendants of `students`". Whether this constraint is implemented by fetching all `students` objects and then looking within them, or by resolving `name` independently and then filtering it, is left up to implementations.

Filtering and Grouping Operators (`where` and `by`)

18.9 %

A less obvious example is the filtering operator (`where`). To enable filtering of list-valued properties, Formula^2 supports a `where` operator and a corresponding `filter()` function. The `filter()` function takes two lists and returns a new list that contains only the elements of the first list for which the corresponding element in the second list is not empty, false, or 0. Any surplus elements in the second list are ignored per existing list operation semantics (Section 4.4.2).

However, `a where b` is not simply syntactic sugar for `filter(a, b)`. There is one common ambiguity that `where` resolves: should properties referenced in predicates be resolved by the context node of the expression as a whole, or in a different way? This is best explained with an example. Consider the schema in Section 4.7.2. Assuming a `decision` item context node, what would you expect the expression `pro where weight > 2` to return? Presumably, you would strongly suspect it should return `pro` objects that have a `weight` property over 2.

However, if `where` were naïvely rewritten, it would be equivalent to `filter(pro, weight > 2)`. Like every function, each argument is resolved separately: `pro` resolves to all pros of the context decision, and `weight > 2` resolves to a list of booleans …for *all* weights — including weights for cons! To get the expected result with the `filter()` function, we need to use the narrowing (dot) operator and write `filter(pro, pro.weight > 2)`.

To prevent this terribly confusing behavior, `where` is implemented to apply *special scoping rules*, to preferentially resolve properties in the second operand as descendants of the first operand.

This is not the same as naïvely prepending identifiers with a narrowing operator involving the first operand, nor as using the first operand as a context node: When we write `foo.bar`, if `bar` is not found within `foo` and its descendants, the return value is empty[2]. However, in the case of `foo where bar > N`, if `bar` is not found within `foo` and its descendants, we do want the search to continue as normal within the context node of the formula. It could be argued that using `foo` as a context node and continuing the search to its ancestors is also reasonable, but in practice that is rarely desirable and has similar issues as dynamic scoping. The closest rewriting would be `(foo.bar or bar)`

19.2 %

Explicit Scoping Operator (`in`)

In some cases, it makes sense to evaluate an identifier using a different context node than the one of the whole formula. One area where this is needed is *nested aggregates*, i.e. aggregates of aggregates. We use nested aggregates when in a hierarchical schema we want to compute an aggregate within descendant lists and then aggregate the aggregates at a higher level of the tree.

This may sound obscure in the abstract, but use cases that require nested aggregates are quite common in data-driven applications. For example, assuming the data schema of a restaurant review website, *"average number of reviews"* is a nested aggregate: AVERAGE of COUNT.

To support this, Formula^2 supports an explicit scoping operator, `in`, which evaluates its first operand in the context of the second operand.

The `in` operator has slightly different semantics around list values than other operators (Section 4.4.2): if the second operand (the scope) is a list, the first operand is evaluated for each item in the list. This is because it actually *affects* what the first operand resolves to, so taking its shape into account would create a logical cycle.

The scoping operator facilitates nested aggregates using a concise, natural syntax. Returning to the restaurant review example, to calculate the average number of reviews per restaurant, we would write `average(count(reviews) in restaurants)`.

It can also facilitate complex object mapping operations, though the syntax can be a little more awkward. For example, assume we have a list of right triangles (`triangle`) as objects with `width` and `height` properties. We can easily calculate a list of hypotenuses with `sqrt(pow(width, 2) + pow(height, 2))`, but what if we want to *transform*

[2] The implementation of Formula^2 that the Mavo HTML prototype embeds does not actually follow this; a bug that has caused a lot of confusion.

it to a list of triangles with `width`, `height`, and `hypoteneuse` properties? This would work, but a novice would be unlikely to write it:

FORMULA²

```
group(triangle, hypotenuse: sqrt(pow(width, 2) + pow(height, 2))) in triangle
```

Due to the potential for confusion, its precedence is very high so that authors are forced to use parentheses to scope expressions with more than one term.

4.5.6 Escaping the Scoping Heuristic

19.6 %

In many common cases, Formula²'s scoping heuristic can save authors time and effort. However, as with many heuristics, there are cases where the heuristic would incorrectly predict author intent. Per our design principles (Section 1.4.3), inferred author intent should be overridable.

For such cases, Formula² provides several ways to disambiguate references, either by using more narrow scoping, or by providing ways to tweak reference resolution:

- If ancestor values are undesirable, a narrowing operator (`.`) can be used to scope references to be within a specific object.
- Special properties such as `$parent`, `$previous`, `$next`, `$item`, `$this` can be used to navigate the data tree of a given reference.
- If the entire list of values is desired from a formula within a list, the `$all` special property can be used.

> **EXAMPLE**
>
> For example, in a list of restaurants, if the formula `rating` was on or within each restaurant, it would resolve to a scalar value: the rating of that restaurant. However, if `rating` is used *outside* the list of restaurants, it would resolve to a list of ratings of *all* restaurants. So what if we want to show where the current restaurant stands compared to all others in the list?
>
> `100 * count(rating.all > rating) / count(rating.all)` would give us the percentage of restaurants with a higher rating than the current one, which would allow us to output things like "Top 5%".

4.5.7 Easier Temporal Computation

19.8 %

We identified temporal math as a common pain point when working with spreadsheets and later validated that our hypothesis was true (see Section 7.6). Yet, temporal computation is ubiquitous in so many types of web applications, from use cases like personal tracking and project management to simple things like showing the date a post was created, or how long has passed since.

Formula^2 provides a variety of functions to support manipulating and formatting temporal values across various points of the ease-of-use to power spectrum.

Dates, times, and date/times are represented as ISO 8601 [85] strings. If a mathematical operator is used with at least one string operand (which cannot be parsed as a number), Fomula^2 will check if the string is a temporal value, and will perform a temporal calculation if so.

This allows authors to specify temporal values by simply specifying them as strings, with no additional effort to convert them to a specific data type. Since no valid ISO dates are also valid numbers[3], this is a safe heuristic.

A variety of built-in functions and special properties are provided to make such operations natural to read and write.

For example `date($today + 1 * day())` will return tomorrow's date and `duration($now - "2019-07-12", 2)` will return how long it has been since July 12th, 2019 in a human-readable format with two terms (e.g. "3 years, 2 months").

Temporal expressions involving special properties such as `$now` should be updated by the host environment at a reasonable rate, up to display's framerate, to ensure that they are always up-to-date. While Formula^2 does not specify optimizations for this (e.g there is no need to update `readable_date($now)` at 60 fps), host environments are encouraged to do so.

The ISO 8601 format also supports timezones, and these are honored in any calculations. However, it remains important future work to provide more primitives for their handling. If no timezone is specified, the user's local timezone is assumed.

4.5.8 Internationalization & Localization

20.1 %

Any functions that produce human readable output (such as many temporal functions) are *locale-aware*: the host environment can optionally associate a locale with certain data

[3] The only exception is years, since something like `"1997"` is actually a valid ISO date. However, this does not participate in the heuristic discussed here, since if authors are doing math between years, there is nothing special to do — handling them as regular numbers works fine.

nodes (any node without a locale inherits the locale of its parent), and output follows this locale. For example, in Mavo, the locale is derived from the closest HTML `lang` attrbute.

4.6 Architecture

Here we summarize some architectural considerations for implementing Formula².

4.6.1 Expression Compilation

20.2 %

Grammar

Ease and efficiency of parsing was a key design consideration. The entirety of Formula²'s syntax can be parsed by a simple Pratt parser [79]. In fact, all that is needed to adapt a Pratt parser intended for JavaScript expressions[4] to a Formula² parser is to simply modify its operators.

To generate a Pratt parser for Formula², the operator associativities and precedences are shown in Table 4.1. On a high level[5], an EBNF grammar for Formula² could look like this:

CODE

```
       Expression: Operand | Compound
          Operand: Literal          | Identifier | UnaryExpression
                 | BinaryExpression | MemberExpression | TernaryExpression
                 | CallExpression   | '(' Expression ')'
    UnaryOperator: 'not' | '-' | '+'
  UnaryExpression: UnaryOperator Operand
   BinaryOperator: ['<' | '>' | '='] '='? | '!=' | '*' | '/' | '+' | '-'
                 | 'mod' |  'and' | 'or' | 'where'
 MemberExpression: [ [MemberExpression | Identifier] '.' ]+ Identifier
 BinaryExpression: Operand BinaryOperator Operand
TernaryExpression: Operand 'by' Operand [ 'as' Operand ]?
   CallExpression: [ Identifier | MemberExpression ] '(' Compound ? ')'
         Compound: Expression [ Separator [ Expression | Compound ] ]+
        Separator: ',' | ';' |
```

[4] JSEP: ericsmekens.github.io/jsep

[5] The productions do not enforce operator precedence, which is specified seprately, and do not describe the minutiae of number and string literals, which are on par with JavaScript or similar languages.

Operator	Description	Precedence	Associativity
`()`	Function call, grouping	15	Left
`.`	Narrowing operator	14	Left
`not`, `-`, `+`	Unary operators	13	Right
`*`, `/`, `mod`	Multiplication, division, modulus	12	Left
`+`, `-`	Addition, subtraction	11	Left
`<`, `<=`, `>`, `>=`	Comparison operators	10	Left
`=`, `==`, `!=`	Equality operators (equal, not equal)	9	Left
`and`	Logical AND	8	Left
`or`	Logical OR	7	Left
`where`	Filtering operator	4	Right
`by` / `by ... as`	Grouping operator	3	Right
`in`	Explicit Scoping operator	2	Left
`,`	Separator in function arguments, etc.	1	Left

Table 4.1 *Precedence and associativity of Formula² operators.*

CODE (continued)

```
whitespace
                | any other non-alphabetic, non-numeric character
         Literal: Numeric literals (e.g., 1, -42, 3.14)
                | Single or double-quoted string literals
      Identifier: ['$' | letter | '_'] [ letter | digit | '_' | '$' ]*
```

AST Transforms and Desugaring

20.9 %

After parsing, a number of AST transforms are applied to the parsed expression. These include:

- Flattening logical operators. For example, `3 < foo < 5` will be parsed as `(3 < foo) < 5`, which would produce an incorrect result. Only logical operators need to be flattened. Other operators are either correctly handled via normal precedence rules, or are associative and thus flattening would make no difference.
- Rewriting operators into their equivalent functions (with further rewriting for operators that affect scoping, see Section 4.5.5).
- If the left operand of `:` is an identifier, it is rewritten to a string literal.

4.6.2 Expression Evaluation

21 %

Building a Schema

> NOTE
>
> While Formula^2 is not dependent on any particular language or system, to avoid excessive abstraction, we will describe its implementation in terms of JavaScript concepts.

For Formula^2 to be able to evaluate expressions against a data tree, it first needs to walk the data tree to understand its structure and to store pointers from data objects to this stored information.

This involves:

1. Walking the data tree to build a schema (unless this is precomputed by the host environment). This is not a detailed data schema; for example it does not concern itself with data types at all. It is simply a nested data structure of all possible property paths to help resolve identifiers.
2. Linking each node to its parent, so that traversal in any direction is possible from any node.
3. Linking each object to its corresponding schema node, so that traversal in any direction is possible.

Building a schema is not merely a performance improvement: it is *essential* for reasonable identifier resolution. Imagine if the meaning of identifiers changed simply because a collection happened to have no items at the moment of evaluation! Of course it is *also* a performance improvement — performing a breadth first search on the data tree for every identifier would be prohibitively slow even for modestly sized datasets.

The host environment needs to notify Formula^2 when the data tree changes in a way that affects the schema so it can be rebuilt, otherwise expression evaluation could be incorrect.

JS

```js
{
   task: [
      {
         taskTitle: "Code furiously",
         done: true,
         tags: ["coding", "fun"],
      },
      {
         taskTitle: "Run user study",
         priority: "P2",
         tags: "science"
      },
      "Have a life?"
   ]
}
```

JS

```js
{
   task: [
      {
         taskTitle: true,
         done: true,
         priority: true,
         tags: [true]
      }
   ]
}
```

Figure 4.3 *A data tree on the left and its generated schema on the right. List items are merged into a single schema node, and arrays win out over scalars.*

4.7 Comparison with Other Languages

21.6 %

We now present a few examples that highlight how Formula^2 provides improved ergonomics over JS, spreadsheet formula languages, and SQL. Each example schema has been chosen as a representative example of a common data shape. First, a flat table. Then, a schema with divergent one-to-many relationships with the same schema. Finally, a schema with a deeper hierarchical structure.

For the comparison with JS, we will use the scoping convention of many JavaScript frameworks, that identifiers directly below the context node can be referenced directly, as well as ancestor properties, but everything else requires explicit scoping.

To compare with spreadsheets and SQL, there is a dilemma: how to best convert a hierarchical schema to a tabular one to maximize the comparison fairness?

There are three main ways:

1. **Denormalization**: Flatten the data structure into a single table with repeating values, akin to the result of a SQL join.
2. **Normalization**: Split the data structure into multiple tables, with foreign keys linking them. 2NF ([86]) is probably sufficient for this.
3. **Blank cells**: Use blank cells where (1) would repeat values.

The latter is what is most readable to humans and what they often naturally gravitate to, as it does not require duplication (1), yet does not involve the complexity and indirection of relations (2). Effectively it is trying to visually emulate a hierarchy, and it works — for (sighted) humans. However, it is actively discouraged [87] as it breaks any type of data processing (e.g. formulas, filtering, sorting, etc.).

Repetition is a common way, albeit tedious to manage manually. However, repetition becomes very awkward when we have divergent one-to-many relationships [32], as is the case in one of the schemas below. This only leaves us one option: normalization.

In many cases, defining additional data in the host UI (e.g. computed properties in Mavo, or additional columns in a spreadsheet) would make some of these expressions a lot simpler, but to ensure a fair comparison, we will only use the data as it is defined in the schema. Additionally, we know from Section 7.6 that it is a barrier for novices when a computation requires auxiliary data to be defined in the UI.

For tasks that involve a context node, it is mentioned. Otherwise, the context node is assumed to be the root of the data tree.

4.7.1 Flat Table: The To-Do List Schema

21.9 %

JS

```
{
   task: [
      {
         taskTitle: "Code furiously",
         done: true,
         priority: "P2",
         due: "2022-07-12T12:00:00Z",
      }
   ]
}
```

task

taskTitle	done	priority	due
Code furiously	true	P2	2022-07-12T12:00:00Z

Table 4.2 *The two schemas: hierarchical and tabular.*

This is a variation of the to-do app shown in Section 1.2.

Simple Aggregates: Percentage of Tasks Done

22.1 %

This is a simple aggregation task, so it mainly highlights syntactic differences across the four languages.

FORMULA²

```
count(done) / count(task)
```

SPREADSHEETS

```
COUNTIF(B2:B, true) / COUNTA(B2:B)
```

JS

```
task.filter(t => t.done).length / task.length
```

SQL

```
SELECT count(done) / count(*) FROM task
```

Grouped Aggregate: Count Tasks by Priority

22.3 %

Here we want a data structure that will give us priorities and the number of tasks with that priority. Aggregate functions like `count()` have a special behavior with the results of grouping, and return a list with the counts of all groups. We can get the group names either as `(task by priority).tasktitle` or simply `unique(taskTitle)`, since it is guaranteed to produce the same results, in the same order.

The colon operator can help us combine the two into a single object:

FORMULA²

```
unique(taskTitle): count(task by priority)
```

> **ASIDE**
>
> But what if a user needs to *actually* count the groups produced? How does one escape this heuristic (Section 1.4.3)? They can simply apply `count()` twice: `count(count(task by priority))`.

This is not possible to do in spreadsheets as the result of a single formula. Spreadsheets have a dedicated UI feature for this called *pivot tables*, although no-one tried to use them in our study (Section 9.7). The closest we can get with the regular primitives is to add a column E to the table, place the formula `=COUNTIF(C2:C, C2)` in E2, and drag it down.

Since UI interactions are not acceptable in this comparison, we will not consider this option. We *could* get an arrayformula with a list of strings like “P0: 2”, “P1: 3”, etc. like this:

SPREADSHEETS

```
=TEXTJOIN(", ", TRUE, ARRAYFORMULA(UNIQUE(C2:C7) & ": " & COUNTIF(C2:C7, UNIQUE(C2:C7))))
```

In JS, it could look like this:

JS

```
task.map(t => t.priority).reduce((acc, p) => {
   acc[p] = (acc[p] || 0) + 1;
   return acc;
}, {})
```

Although a grouping function has recently been added to the language (`Object.groupBy()`), it does not necessarily make aggregation easier:

JS

```
Object.fromEntries(
   Object.entries(
      Object.groupBy(task, t => t.priority)
   ).map(([k, v]) => [k, v.length])
)
```

Contrary to the other two, these tasks are SQL’s bread and butter:

SQL

```
SELECT priority, count(*) FROM task GROUP BY priority
```

Temporal: Display Time Left for Each Task

This is a simple temporal computation task. More than showcasing the strengths of Formula^2, this task highlights a weakness of the other three languages, in a task that is very common in data-driven applications.

FORMULA²

```
(taskTitle): duration($now - due)
```

The parentheses are needed because otherwise Formula² would create an object with a single `taskTitle` property (rather than evaluate `taskTitle`) due to the special handling of the first operand in the colon operator (see Section 4.6.1.2).

Spreadsheets have the same issue as with the previous task: to use a simple formula, we need to perform UI interactions such as dragging down a formula. Otherwise, our only recourse is unwieldy arrayformulas.

But here there is a much bigger issue at play: there is *no* high-level way to express an interval in a human-readable way. Spreadsheet applications often provide UI for formatting a number as a duration, but this is not available in formulas and typically produces cryptic separated values of predefined units, so a duration like "2 months, 8 days" may appear like `936:00:00`. Our Lifesheets user study explores this more (see Section 9.7).

Displaying values in any even moderately human-readable way is painful. Even if we only care about differences of a day or more, we need to do low-level wrangling like:

SPREADSHEETS

```
=TRIM(SUBSTITUTE(
IF(DATEDIF(D1, NOW(), "Y") > 0, DATEDIF(D1, NOW(), "Y") & " years, ", "") &
IF(DATEDIF(D1, NOW(), "YM") > 0, DATEDIF(D1, NOW(), "YM") & " months, ", "") &
IF(DATEDIF(D1, NOW(), "MD") > 0, DATEDIF(D1, NOW(), "MD") & " days", ""),
", ", ""))
```

It gets even worse if we want to show time units as well, as one normally would in a task manager:

SPREADSHEETS

```
=TRIM(SUBSTITUTE(
IF(DATEDIF(D1, NOW(), "Y") > 0, DATEDIF(D1, NOW(), "Y") & " years, ", "") &
IF(DATEDIF(D1, NOW(), "YM") > 0, DATEDIF(D1, NOW(), "YM") & " months, ", "") &
IF(DATEDIF(D1, NOW(), "MD") > 0, DATEDIF(D1, NOW(), "MD") & " days, ", "") &
IF(INT(NOW() - D1) = 0, IF(HOUR(NOW() - D1) > 0, HOUR(NOW() - D1) & " hours, ", "") &
IF(MINUTE(NOW() - D1) > 0, MINUTE(NOW() - D1) & " minutes", ""), "") &
IF(HOUR(NOW() - D1 - INT(NOW() - D1)) > 0, HOUR(NOW() - D1 - INT(NOW() - D1)) & " hours, ", "") &
IF(MINUTE(NOW() - D1 - INT(NOW() - D1)) > 0, MINUTE(NOW() - D1 - INT(NOW() - D1)) & " minutes", ""),
", ", ""))
```

The `ARRAYFORMULA` version of this is left as an exercise for the reader.

JavaScript also does not provide a built-in way to format durations in a human-readable way. In practice, this is often done with a library like `moment.js` or `date-fns`, but for the sake of comparison, we will use a simple, naïve implementation:

JS

```
task.map(t => {
   let start = new Date(task.due);
   let end = new Date();
   let elapsed = now - start;
   let units = [
      { unit: 'year', ms: 365 * 24 * 60 * 60 * 1000 },
      { unit: 'month', ms: 30 * 24 * 60 * 60 * 1000 },
      { unit: 'day', ms: 24 * 60 * 60 * 1000 },
      { unit: 'hour', ms: 60 * 60 * 1000 },
      { unit: 'minute', ms: 60 * 1000 }
   ];

   let rtf = new Intl.RelativeTimeFormat('en', { numeric: 'auto' });

   return units.reduce((result, { unit, ms }) => {
      const value = Math.floor(elapsed / ms);
      if (value !== 0) {
         result.push(rtf.format(-value, unit)); // Negative value to represent the past
         elapsed -= value * ms;
      }
      return result;
   }, []).join(', ');
});
```

Standard SQL is not well-suited for this task either, as it is not designed for temporal computations. It is possible to do it, but it is also not pretty:

SQL

```
SELECT taskTitle, CONCAT(
   IF(YEAR(due) - YEAR(NOW()) > 0, YEAR(due) - YEAR(NOW()) & " years, ", ""),
   IF(MONTH(due) - MONTH(NOW()) > 0, MONTH(due) - MONTH(NOW()) & " months, ", ""),
   IF(DAY(due) - DAY(NOW()) > 0, DAY(due) - DAY(NOW()) & " days, ", ""),
   IF(HOUR(due) - HOUR(NOW()) > 0, HOUR(due) - HOUR(NOW()) & " hours, ", ""),
   IF(MINUTE(due) - MINUTE(NOW()) > 0, MINUTE(due) - MINUTE(NOW()) & " minutes", "")
) FROM task
```

PostgreSQL however has an `age()` function that can make this very elegant:

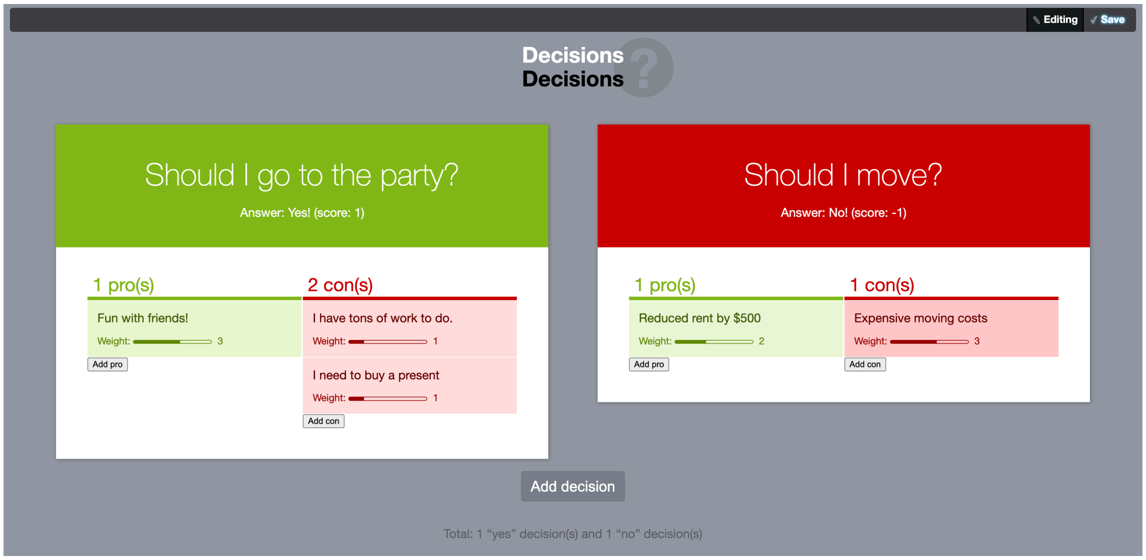
SQL

```
SELECT taskTitle, age(due, NOW()) FROM task
```

4.7.2 Divergent 1-N Relationships: The Decisions App Schema

26.7 %

This is the (Mavo-inferred) schema from the decision-making application that was used in the first lab study (Section 7.1). It's a hierarchical schema with two levels of nesting, where each `decision` item contains divergent one-to-many relationships.



Decisions
Decisions

Should I go to the party?

Answer: Yes! (score: 1)

1 pro(s)

Fun with friends!
Weight: 3

2 con(s)

I have tons of work to do.
Weight: 1

I need to buy a present
Weight: 1

Should I move?

Answer: No! (score: -1)

1 pro(s)

Reduced rent by $500
Weight: 2

1 con(s)

Expensive moving costs
Weight: 3

Total: 1 "yes" decision(s) and 1 "no" decision(s)

Figure 4.4 *A sample application using data with this schema.*

JS

```
{
  decision: [
    {
      decision: "Should I go to the party?",
      answer: null, // calculated
      score: null, // calculated
      pro: [
        { argument: "Fun with friends!", weight: 3 }
      ],
      con: [
        { argument: "I have tons of work", weight: 1 },
        { argument: "I will need to buy a present", weight: 1 }
      ]
    },
    {
      decision: "Should I move?",
      answer: null, // calculated
      score: null, // calculated
      pro: [
        { argument: "Reduced rent by $500", weight: 2 }
      ],
      con: [
        { argument: "Expensive moving costs", weight: 3 }
      ]
    }
  ]
}
```

decision

id	decision	score	answer
1	Should I go to the party?		
2	Should I move?		

pro

did	argument	weight
1	Fun with friends!	3
2	Reduced rent by $500	2

con

did	argument	weight
1	I have tons of work	1
1	I will need to buy a present	1
2	Expensive moving costs	3

Figure 4.5 *The two schemas: hierarchical and tabular.*

Calculate the Score of Each Decision

27.4 %

Context node: `decision.*`

This highlights aggregates that span two separate branches.

FORMULA^2

```
sum(pro.weight - con.weight)
```

or

FORMULA^2

```
sum(pro.weight) - sum(con.weight)
```

SPREADSHEETS

```
SUMIF(pro!A2:A, A2, pro!C2:C) - SUMIF(con!A2:A, A2, con!C2:C)
```

JS

```
pro.filter(p => p.decision_id == decision.id)
   .reduce((acc, p) => acc + p.weight, 0)
-
con.filter(c => c.decision_id == decision.id)
   .reduce((acc, c) => acc + c.weight, 0)
```

Since SQL does not have a context node concept, we calculate all scores:

SQL

```
SELECT decision.id, sum(pro.weight) - sum(con.weight)
FROM decision
LEFT JOIN pro ON decision.id = pro.did
LEFT JOIN con ON decision.id = con.did
GROUP BY decision.id
```

Filtered Aggregate: Count of Good (Score > 0) Decisions

28 %

FORMULA^2

```
count(score > 0)
```

SPREADSHEETS

```
COUNTIF(score!A2:A, ">0")
```

JS

```
decision.filter(s => s.score > 0).length
```

SQL

```
SELECT count(*)
FROM decision
WHERE score > 0
```

Nested Aggregate: Average Number of Arguments per Decision

28.2 %

This compares nested aggregates across different languages.

FORMULA²

```
average(count(pro) in decision + count(con) in
decision)
```

or

FORMULA²

```
average((count(pro) + count(con)) in decision)
```

SPREADSHEETS

```
=(SUMPRODUCT((pro!A:A=decision!A2:A3)+0) + SUMPRODUCT((con!A:A=decision!A2:A3)+0)) /
COUNTA(decision!A2:A3)
```

JS

```
decision.map(d => d.pro.length + d.con.length) / decision.length
```

SQL

```
SELECT
    d.id,
    d.decision,
    AVG(argument_count) AS average_arguments
FROM
    (SELECT
        p.did AS id,
        COUNT(p.argument) + COUNT(c.argument) AS argument_count
    FROM
        pro p
    LEFT JOIN
        con c ON p.did = c.did
    GROUP BY
        p.did) AS argument_counts
JOIN
    decision d ON d.id = argument_counts.id
GROUP BY
    d.id, d.decision;
```

4.7.3 Deep Nesting: The Restaurant Reviews Schema

28.9 %

This is the schema from the restaurant review log that was used in the first lab study (Section 7.1). It's a hierarchical schema with three levels of nesting.

JS

```
{
   "restaurant": [
      {
         "picture": "https://www.toscanoboston.com/common/images/thumb-cucina.jpg",
         "url": "http://www.toscanoboston.com/beacon-hill",
         "name": "Toscano",
         "visit": [
            {
               "date": "2016-03-15",
               "title": "Date night!",
               "dish": [
                  {
                     "name": "Filet mignon with black truffle and foie gras",
                     "dishRating": 4
                  }
               ]
            }
         ]
      }
   ]
}
```

restaurant

id	picture	url	name
1	“redacted.jpg”	“redacted”	Toscano

visit

rid	date	title
1	2016-03-15	Date night!

dish

vid	name	dishRating
1	Filet mignon with black truffle and foie gras	4

Figure 4.6 *The two schemas: hierarchical and tabular.*

Nested Aggregate (2 Levels): Restaurant Rating

29.3 %

Context node: `restaurant.*`

A restaurant’s rating is the average of all visit ratings, and the visit rating is the average of all dish ratings.

FORMULA^2

```
average(average(dishRating) in visit)
```

SPREADSHEETS

```
AVERAGEIFS(
    dish!C:C,
    dish!A:A,
    IF(
        ISNUMBER(MATCH(visit!A:A,
            IF(restaurant!A:A = A2, visit!A:A, ""),
            0
        )),
        visit!A:A,
        ""
    )
)
```

JS

```
visit.map(v => v.dishRating.reduce((acc, c) => acc + c, 0) / v.dishRating.length) / visit.length
```

SQL

```
SELECT
   r.id AS restaurant_id,
   r.name AS restaurant_name,
   AVG(vr.average_dish_rating) AS restaurant_rating
FROM restaurant r
JOIN visit v ON r.id = v.rid
JOIN
   (
      SELECT  vid, AVG(dishRating) AS average_dish_rating
      FROM dish
      GROUP BY vid
   ) vr ON v.rid = vr.vid
GROUP BY
   r.id, r.name;
```

30.1 %

Filtered Nested Aggregate: Count of Good Restaurants

We define a restaurant as *good* if its rating is above 3. This is a nested aggregate with three levels of aggregation, and a filter.

FORMULA^2

```
count((average(average(dishRating) in visit) in restaurant) > 3)
```

SPREADSHEETS

```
SUMPRODUCT(
    IF(
        MMULT(
            (visit!A:A = restaurant!A2:A100) *
            TRANSPOSE((dish!A:A = visit!A:A) * dish!C:C),
            TRANSPOSE((visit!A:A = restaurant!A2:A100) * (visit!A:A <> ""))
        ) > 3,
        1,
        0
    )
)
```

JS

```
restaurant.map(r => r.visit.map(v => v.dishRating.reduce((acc, c) => acc + c, 0) /
v.dishRating.length) / r.visit.length).filter(r => r > 3).length
```

SQL

```
WITH restaurant_ratings AS (
   SELECT
      r.id AS restaurant_id,
      AVG(vr.average_dish_rating) AS restaurant_rating
   FROM restaurant r
   JOIN visit v ON r.id = v.rid
   JOIN (
         SELECT
            vid,
            AVG(dishRating) AS average_dish_rating
         FROM dish
         GROUP BY vid
      ) vr ON v.id = vr.vid
   GROUP BY r.id
)
SELECT COUNT(*) AS good_restaurant_count
FROM restaurant_ratings
WHERE restaurant_rating > 3;
```

4.8 Discussion & Future Work

31 %

4.8.1 Limitations

Formula² was designed to make the kinds of computations that are common in small-scale data-driven applications easier to express by novices. There are numerous use cases arising in general application development that are currently either awkward or impossible to express in Formula².

Complex Mapping Operations

31.1 %

While Formula² can express simple mapping operations very easily, such as mapping objects to a descendant property or a combination of descendant properties, many of the kinds of arbitrary mapping operations that a developer can accomplish in an imperative language via a loop are not possible, or awkward to express with Formula² alone.

A large class of such use cases is *augmenting objects*, such as the triangle hypotenuse example in Section 4.5.5.3, which can be expressed but the formula to express them is beyond the capabilities of most novices.

31.2 %

String Parsing

One big category is **parsing** tasks: producing structured data from a string of text, for example color components from a serialized color, or SVG path segment information from an SVG path string (see Section 8.2.1).

Perhaps a novice-friendly way to express simple patterns could be developed and exposed in Formula^2 via suitable functions, but currently this remains an open question.

4.8.2 Is Flattening Always the Right Choice?

Possibly one of the most questionable design decisions of Formula^2 is its flattening of multi-dimensional arrays. E.g. `list(1, list(2, 3), list(4, 5))` is flattened to `list(1, 2, 3, 4, 5)`. The intent behind this was to simplify the number of distinct cases that need to be handled, since these structures did not naturally occur in Mavo apps, and since every multi-dimensional array can be expressed as an array of objects.

However, since the original design, some cases have emerged where this flattening is undesirable. The main example is grouping by nested lists. Consider a list of people (`person`), each of whom has a list of hobbies (just a list of strings) (`hobby`). Currently, `person by hobby` would not produce a reasonable result, since it depends on element-wise matching.

Perhaps a compromise solution could be to use a data structure that *behaves* like a flattened array, but is not actually flattened. Or the opposite: a flattened array that preserves metadata about the boundaries of its constituent arrays.

31.3 %

4.8.3 Dot Notation for Function Calls

We opted for a functional syntax, as it appears to be easier for novices to understand, and may be familiar from spreadsheets. We hypothesized that for example, `count(rating)` is easier to understand than `rating.count()`. However, it does come with the drawback of authors having to manage nested parentheses, a common authoring mistake. Additionally, if *some* functions are available as functions and some as methods, it increases the cognitive load on the author to remember which is which.

In line with our design principle of robustness, it appears an *optional* dot notation could greatly increase the efficiency and safety of Formula^2, without compromising its learnability.

Rather than authors having to remember which functions are methods, *every function* would be available as a method and vice versa. `arg.foo(arg1, arg2)` would be equivalent to `foo(arg, arg1, arg2)`.

This would allow authors to use the syntax that is most convenient for their use case, and feels most natural to them, without introducing additional error conditions or cognitive tax.

4.8.4 Notation for Unrestricted Identifiers

31.5 %

We have repeatedly seen in our user studies that novices struggle with the concept of identifiers having a restricted syntax. Furthermore, imposing restrictions on identifier names makes it awkward to work with data created by others, which may not conform to these restrictions.

To prevent syntactic ambiguity, allowing unrestricted identifiers would require a different syntax to tell the parser that a series of symbols is actually an identifier, and delineate where it begins and ends.

It appears that brackets (`[]`) could be a good fit for this purpose, and have some precedent ([32]). However, this would require substantial changes in Mavo, which uses brackets to embed Formula^2 expressions in literal text.

4.8.5 Quantities as a First-class Citizen

31.6 %

For functions returning a numerical value with no arguments, and constants, it is often more readable to be able to express multiplications with a fixed number as a number with a unit.

For example, `date($today + 2 * days())` will return the date two days from now. But it might be a lot more natural if we could write `date($today + 2days)`. Or, rather than calculating a circle's circumference as `radius * 2 * pi`, it may be more readable to write `radius * 2pi`.

Formula^2 could implement this as a general language construct: a number followed by an identifier is syntactic sugar for the multiplication of the number and the value of the identifier if the identifier is a number, or the value of calling the identifier as a function with no arguments if the identifier is a function. This could be a property of specific functions, since it only makes sense for a relatively small percentage of functions. Alternatively, the number and type of quantity could be retained in an object that can be coearced to a number when needed. This could allow expressing a lot more values as first-class citizens, e.g. currencies, temperatures, measurements, etc.

4.8.6 Optimizing Frequent Lookups

While performance optimizations do not affect the language itself, in practice they can have a significant impact on the user experience. Currently, the only performance optimization that the prototype implementation of Formula^2 employs is the use of a schema to precompute all possible property paths. Even caching the result of an expression so that expressions are not re-evaluated if the data has not changed, is relegated to the host environment.

However, given the lack of abstractions in the formulas novices write (see [88] and Section 9.8), more elaborate caching mechanisms are *essential*. By moving caching within Formula^2, individual operands can be independently cached, so that for example `task by priority` and `count(task by priority)` only needs to compute the grouping once.

Additionally, a common pattern we have observed (see Chapter 7) is for certain properties to be repeatedly used to filter the same structure, often acting essentially as *implicit primary keys*.

For an illustrative example, consider a collection of books and their metadata (possibly fetched from a remote API), and a reading log with a list of books read, ratings, and notes, maintained by the app user. The app author could display a dropdown of books to associate each entry with, and store the selected book ID in a `bookId` property.

Then, to display book metadata next to each entry, the author would have to run a filtering operation such as `books where id = bookId`. The problem with that is that this would iterate over the entire `books` collection for each entry, making lookups $O(N^2)$.

These types of tasks are the bread and butter of relational database systems: you simply declare `bookId` as a foreign key, and assuming `book.id` is a primary key, the database will take care of the rest. However, this is nontrivial for novices and end-user programmers, who struggle to think in terms of relations [51] and are goal-oriented, and therefore averse to preparatory work such as setting up schemas and indices.

However, perhaps end-users don't need to do this work. Assuming Formula^2 provided a mechanism for declaring primary keys and indices, creating them could be handled by the host environment, which has a lot more information about what expressions may be evaluated and how frequently. To use Mavo as an example, a simple heuristic would be to automatically create indices for any property used in a `where` or `by` clause within a collection.

Another (not mutually exclusive) direction could be for Formula^2 to automatically optimize expressions based on a combination of factors such as how frequently they are evaluated, and how large the collections they operate on are.

4.8.7 Short-circuit Evaluation

32 %

Most common programming languages support short-circuit evaluation for logical and conditional(ternary) operators. Later operands are only evaluated if earlier operands do not suffice to determine the value of the expression. For example, in `a and b`, if `a` is false, `b` is not evaluated, since we already know that the overall value must be false.

Due to its handling of list-valued operations, Formula^2 cannot support short-circuit evaluation, as all operands need to be examined to determine the shape of the result.

This is not an issue for its use as a reactive formula language, but becomes one once side effects are introduced (such as by Data Update Actions, see Section 6.3.4).

4.9 Conclusion

32.1 %

In this chapter, we presented Formula^2, a formula language designed for end-user programming in data-driven web applications, and optimized for hierarchical schemas.

Our user studies (described in Chapter 7) have shown that Formula^2 is easy to learn for novices, who often did not believe that expressions they wrote could work.

While Mavo is currently the only deployed Formula^2 host environment, we believe its potential is much broader, and look forward to seeing additional implementations in the wild.

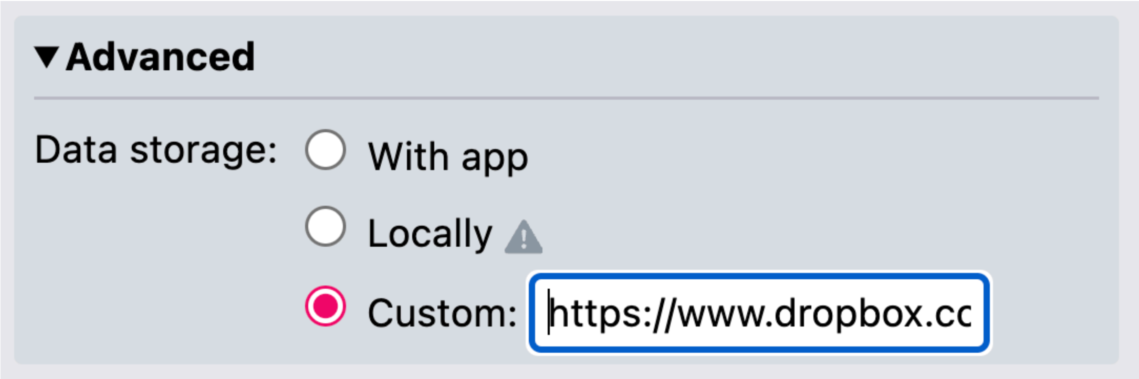
CHAPTER 5

Madata: Facilitating Data Ownership by Democratizing Data Access

5,102 words (15 min read)

5.1 Introduction

32.5 %



▼ Advanced

Data storage: ○ With app
○ Locally ⚠
● Custom: https://www.dropbox.cc

Figure 5.1 *Our experimental visual app builder Lifesheets (Chapter 9) is also the first prototype of a GUI application using Madata to offer users the freedom of being able to store their data in any location they choose.*

One of the big innovations of the Web [3] was that it unified the process of accessing a remote document and encoded all the information needed for the (previously) multi-step process of conncting to the right FTP server, navigating the remote filesystem to find the right file, downloading the resource, and opening it in the right application into a single step: loading a URL in the browser. However, when it comes to *writing* data to the Web, we still need to deal with complex, multi-step processes and disparity between APIs and data formats.

As a result, most systems that store user data do so either locally or (more frequently) on a single cloud service they control. Users have little to no data portability; effectively locking them into the service. Part of this is the common need for services to control user data. However, another component is simply that supporting user selection for data storage is *really hard*.

While reading data from remote sources is generally easy, persisting data remotely is one of the big *usability cliffs* of the web platform. Despite it being such a common need, implementing it requires understanding of many programming concepts which are non-trivial for programmers and entirely out of reach for beginners.

Worse, even after investing the effort to understand and use a specific authentication and storage mechanism, interfacing with a different service requires learning a completely new API.

Madata is a set of simple protocols and JavaScript APIs[1] that allows web applications to read and store data in a variety of locations, serialized in a variety of formats, all with the same unified API. Programmers can support additional services without *any* changes to their application code, as Madata abstracts differences between services away into a single API.

Since one of the primary goals is **portability**, inspired by the usability innovations of the Web, Madata introduces the concept of a *storage URL*. The storage location can be uniquely identified via a URL, from which Madata infers which service to use, the location of the data within the service, and how to access it. This prototypes a future where users can decide where their data is stored by simply entering a URL in the settings of the application they are using.

Most remote services require authentication, which is a complex process that requires registering an OAuth [13, 14] application and writing code to handle the authentication flow which involves a multi-step handshake. Madata simplifies this process by introducing the concept of a *federated authentication provider* or *FedAP*. A FedAP is a server that securely stores API keys for specific OAuth appliations in its supported services.

[1] Source code and documentation is available at madata.dev.

Instead of requiring developers to register a new application to experiment with a new API, FEDs allow several developers to share the same OAuth application. Developers have the *option* to use their own API keys that are not shared, but they don't need to.

This flow also provides user experience benefits to end-users: once they have logged in to a FED, they can log in to any app using the same FED with two clicks. Any server can become an authentication provider by implementing a simple API (Section 5.5.3).

While the Madata client library that requires programming to use, it has been designed to minimize the *"gulf of evaluation"* [89] and to maximize *"closeness of mapping"* [76] between the user's mental model and the API. It is indicative that Mavo [11], which targets non-programmers, provides an HTML-based API which is a thin abstraction layer over Madata objects and components.

While portability in terms of storage location is the core focus of Madata, it also allows for *data portability in terms of data serialization format*. Portability of format is essential for data longevity, as it allows data to be migrated to new formats as old ones become obsolete. While defaulting to JSON, Madata seamlessly parses and serializes data in a variety of formats, including CSV, YAML, TOML, BiBTeX, and more.

The European Union establishes **data portability as a fundamental human right** [15]. By making it *easier* for developers to offer end-users data portability than not to, Madata prototypes a future where data portability (and the data ownership it begets) are not a rare exception, but the norm.

5.2 Related Work

WebDAV [90–93] was a protocol with very similar goals to Madata. It was developed in the late 1990s as an extension to HTTP to enable users to collaboratively edit and manage files on remote web servers. Despite its promising features, WebDAV failed to achieve widespread adoption beyond certain niche domains. Some of the reasons were related to its high complexity which created performance issues, and its proneness to network effects, as it required web server support. Instead of attempting to *replace* existing protocols, Madata is designed to *"pave the cowpaths"* by reducting the friction of *interfacing* with them.

The Solid Platform [94, 95] has very similar goals as Madata. Solid is a decentralized platform for social Web applications. Like Madata, user data is managed independently of the applications that create and consume this data. User data is stored in a Web-accessible personal online datastore (or pod). Like Madata, Solid allows users to store data in many different providers, and easily switch between providers. However, the solution Solid is proposing is more heavyweight and involves higher complexity and more cognitive burden for end-users. Solid's approach requires users to understand concepts such as Pods and Linked Data, which can be complex for non-technical users, compared to Madata's simple URL-based approach, which imposes no requirements or demands on the data being exchanged. But most importantly, Solid requires adoption by the storage providers, thus being subject to network effects, while **Madata can work with any service that provides a Web API**.

Two very relevant projects that came after Madata are Scrapir [96] and Shapir [97], tackling similar goals of standardizing and democratizing access to disparate Web APIs. Scrapir [96] takes an assisted collaborative approach, with somewhat technical users adding support for new services via a GUI, so that non-technical users can then use those services. Shapir [97] also tackles similar goals of standardizing various Web APIs by mapping them to schema.org [98] entities that can then be read and written by modifying regular JavaScript objects. While there is some intersection, both of these are focused around reading and writing third-party data, while Madata is focused on reading and writing arbitrary user data.

5.3 Main Concepts

33.4 %

5.3.1 Backend Classes

A *backend class* (or for short, *backend*) tells Madata how to interface with a specific type of storage location. This is often a remote service (e.g. GitHub, Google Sheets, Dropbox), but it can be any I/O mechanism that supports hierarchical data. For example, there are backends like:

- *Local* for storing data locally in the browser (`localStorage` object)
- *Element* for "storing" data as another element's content (mainly useful for debugging)
- *URL* for "storing" small amounts of data as parameters of the current URL.

Not all backend classes provide the same capabilities. Backend classes declare which capabilities they support (`write`, `login`, `upload`). Applications using Madata can then read this information and adjust their UI accordingly or communicate to the user that their selection of backend is unsuitable for the current operation.

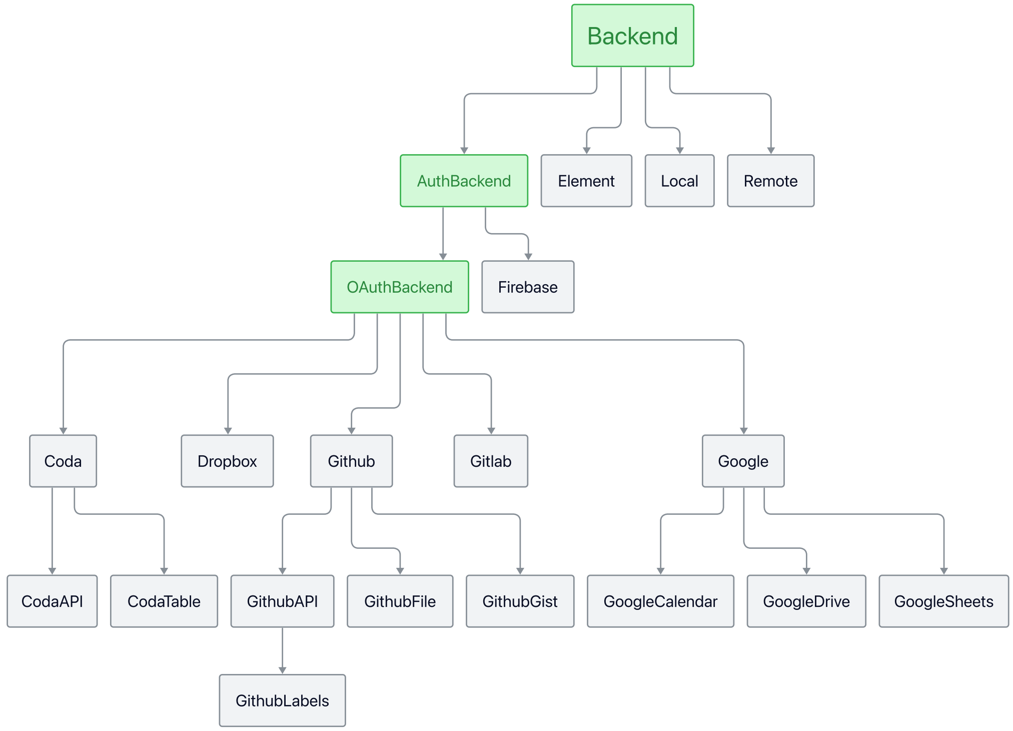


Figure 5.2 *The hierarchy of backend classes in Madata's prototype implementation as of August 2024. Abstract classes shown in green.*

Backends are organized in a **hierarchy**, with common patterns implemented in base classes. For example, the Google Sheets backend and the Google Drive backend both share the same parent backend, which defines authentication for many Google™ services (Figure 5.2).

5.3.2 Backend Objects

33.6 %

A *backend instance* encapsulates a specific storage location within a specific *backend class*. For example, a specific sheet in a Google Sheets spreadsheet, a specific file on Dropbox, or a specific key in the browser's `localStorage`.

In some cases the boundaries of what should be an object in a hierarchical data structure vs a separate backend object can be blurry. For example, should a spreadsheet backend object represent the entire spreadsheet, by returning an object with keys for every sheet, or a single sheet? The current Google Sheets backend has opted for the latter, as many use cases only involve a single sheet, and thus having to deal with an extra level of nesting would be cumbersome. However, both options are defensible.

5.3.3 Storage URL

33.7 %

As a design principle, it should be possible to identify storage locations by specifying a URL. While a URL should unambiguously identify the storage location, the same storage location *may* be described by multiple URLs. The URL should be *either* easy to compose, *or* easy to obtain from the service itself, and ideally both.

For example, one of the supported URLs for GitHub is the URL shown in the browser when viewing a file on GitHub. Or, the URL for a Dropbox file is the URL obtained when using its *"Share file"* UI feature.

> **EXAMPLE**
>
> For example, if the *storage location* is `https://github.com/mavoweb/mavo.io/blob/main/demos/todo/tasks.json`, Madata infers the following from it:
>
> - The service to use is GitHub,
> - The backend to use is "GitHub File" (as opposed to e.g. GitHub Gist),
> - The file is located at `demos/todo/tasks.json` in the `mavoweb/mavo.io` repository in the `main` branch.

Storage URLs are merely a portable way to represent the information needed to access a storage location. They do not need to be URLs browsers can natively load or actually resolve to the resource, altough both of these are desirable properties, as they assist with debugging.

However, in some cases, especially with non-typical backend classes, no HTTP URL pattern is a good fit. A last resort is using a custom protocol. For example, to save data in the browser's local storage using `mykey` as the key, the Madata URL is `local:mykey`.

33.9 %

Test URLs & Known URLs

URLs are defined as URL patterns [99], a standardized syntax for concisely specifying a set of URLs (for an example, see Section 5.5.1).

Each backend defines a set of zero or more *test URLs* and zero or more *known URLs*. *Test URLs* are those that are used to test if a given URL should resolve to a given backend. They need to be more precise, to avoid false positives. *Known URLs* are those that the backend knows how to process, but should not necessarily cause it to be selected.

Madata stores a list of all backends topologically sorted by their inheritance. To resolve a given storage URL to a backend, Madata iterates over the list of backends, trying their test URLs in order, until one matches. If none match, it falls back to the *default backend*, whose only capability is reading data from URLs.

These URL patterns perform double duty: their named groups are used to extract the necessary information from the URL, so that a second parsing step (which may be beyond the capabilities of many novice programmers) is not necessary.

5.3.4 Federated Authentication Provider (FedAP)

34 %

There are various authentication protocols in use today to facilitate secure access to third-party APIs. These authentication protocols are designed to be very secure, but they are also complex and require a lot of boilerplate code to use.

As an illustrative example, consider OAuth 2.0 [14], one of the most popular authentication protocols today. For a developer to use OAuth 2.0, they usually need to:

1. Register an application with the service they want to access, by describing what they intend to do and obtain a secret API key.
2. Obtain a server with the capability to run server-side code. Register a domain name, then provide a "calback URL" to the service they want to access.
3. From their client-side application, open a popup to a certain URL, so users can authenticate with the third-party service.
4. The popup asks the user to log in to the third-party service and authorize the application to access their data. Then, the popup redirects to the callback URL, with a temporary code in the URL.

9. The server-side code residing at the callback URL sends a `POST` request to a special URL at the third-party service (e.g. `https://github.com/login/oauth/access_token` for GitHub), providing the temporary code as well as its secret API key (which shouldn't be shared).
10. If everything went well, the response should (*finally!*) contain an access token. Extract that access token.
11. Now communicate the access token back to the client-side application, so it can use it to access the third-party service.

OAuth does support an easier *"implicit grant"* flow, but it is considered less secure and not supported by all services. Thus, even if the rest of the API is CORS-enabled [100], the authentication handshake often requires server-side code.

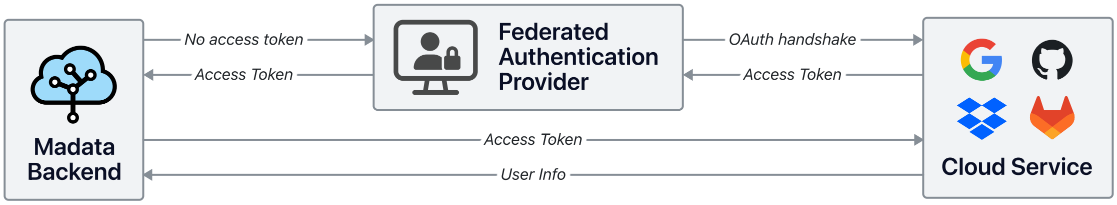


Figure 5.3 *The only role of a Federated Authentication Provider (FedAP) is to authenticate the user with the third-party service, obtain an access token, and communicate it back to the client-side application. It does not even need to store them (although it can). From that point onwards, the Madata backend is expected to communicate directly with the third-party service.*

We introduce the concept of a *Federated Authentication Provider (FedAP)*, to abstract all this complexity away into a single string: The FedAP's domain name.

FedAPs are servers that store API keys for specific OAuth applications in their supported services, take care of the authentication handshake, and communicate the resulting access token to Madata, all with no involvement from the application developer. Developers can change their authentication provider (from the default auth.madata.dev) by simply setting a static property on `Backend` objects. For example, to be able to take advantage of existing logins to Mavo applications, one would need to use the Mavo authentication provider:

JS

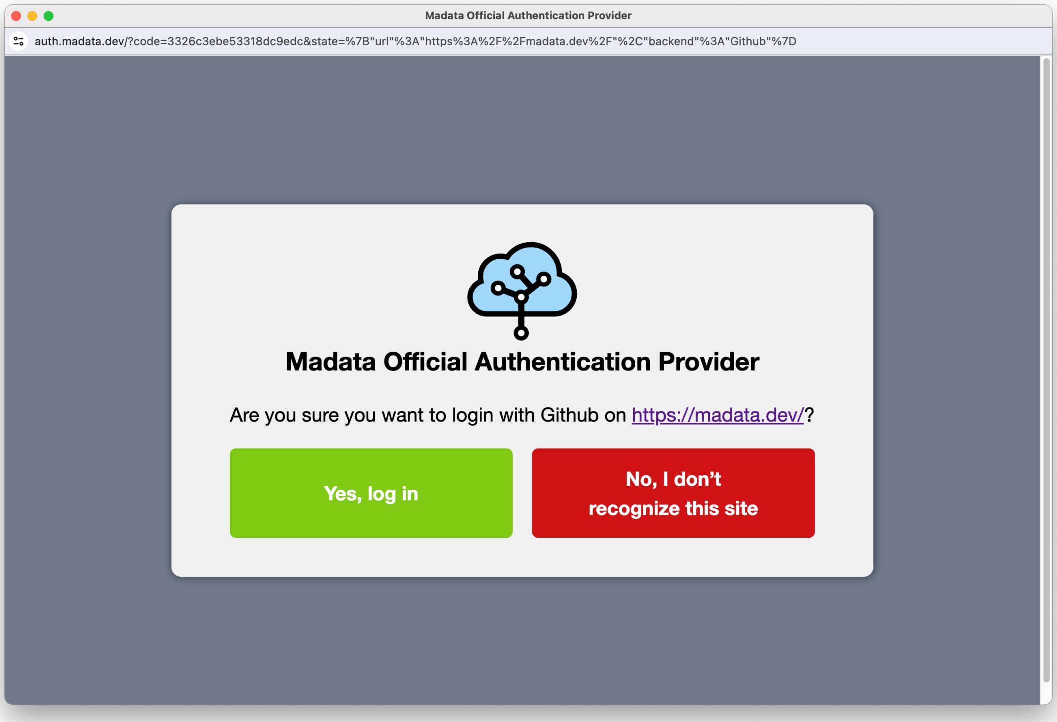
```js
Backend.authProvider = "https://auth.mavo.io";
```

Or, they could specify multiple, for redundancy:

JS

```js
Backend.authProvider = ["https://auth.mavo.io", "https://auth.madata.dev"];
```

At the end of the authentication handshake, the FedAP presents the user with a confirmation dialog (Figure 5.4). This step is **essential** for preventing malicious use. Without a confirmation, a malicious application could trick users into visiting the page, and then would get unfettered access to the user's data on *all* services they have used the FedAP to authenticate with.



Madata Official Authentication Provider

auth.madata.dev/?code=3326c3ebe53318dc9edc&state=%7B"url"%3A"https%3A%2F%2Fmadata.dev%2F"%2C"backend"%3A"Github"%7D

Madata Official Authentication Provider

Are you sure you want to login with Github on https://madata.dev/?

Figure 5.4 *An example of an authentication confirmation screen.*

If the user confirms, the FedAP then communicates the access token to Madata, and no further server interaction with the FedAP is necessary. Since the FedAP is only involved in the authentication handshake, this is generally a very low resource operation. As one data point, the author has used a very early precursor of this approach on a website [101] that served 50-100k users per month for five years (2013-2018) with no issues.

FedAPs can be used independently of Madata. When using a FedAP (without Madata), the above process looks like this:

3. From their client-side application, open a popup to a certain URL, so users can authenticate with the third-party service.
4. The popup asks the user to log in to the third-party service and authorize the application to access their data. Then, the popup redirects to the FedAP callback URL.
5. The FedAP takes care of the rest (steps 4-7 above), and communicates the access token back to the originating application by using the Window Cross-Messaging API `window.postMessage()` to send the token back to the originating application…

When using a FedAP with Madata, the process is even simpler:

3. From their client-side application, the developer calls `backend.login()`, which takes care of the rest.
4. The developer can `await` the result, which will be the user information (if the login was successful) or just listen to the `login` event.

FedAPs follow *introspection*; visiting a FedAP's root domain displays the list of services it supports (Figure 5.5). The same information can be obtained programmatically via `/services.json` which *should* be CORS-enabled.

5.4 Usage Examples

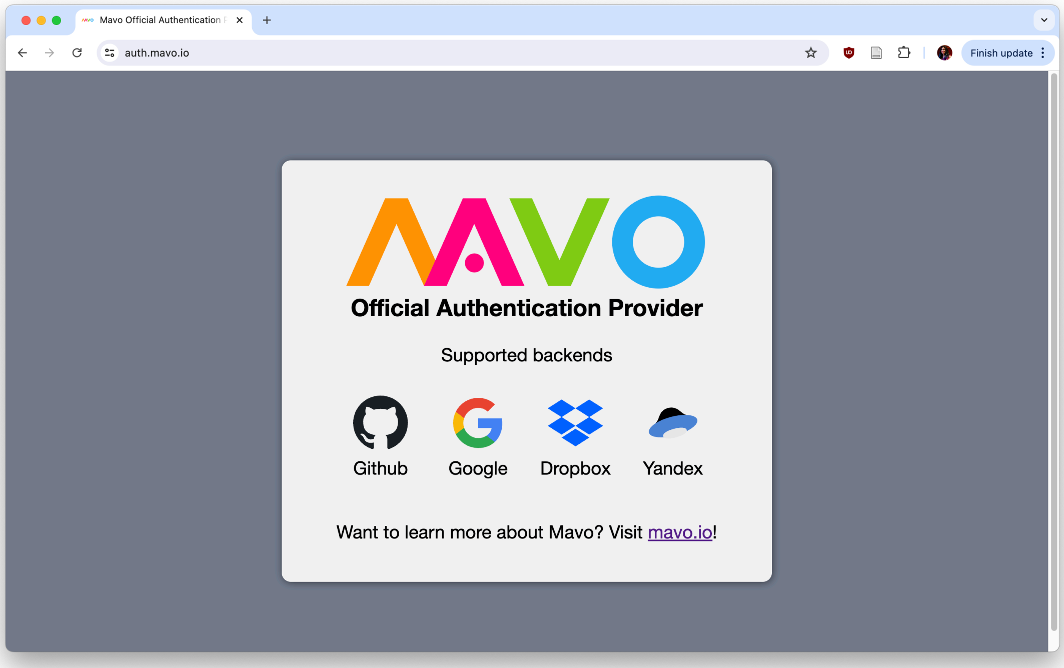
34.8 %

To create a backend object for a specific storage location, all that is needed is to call `Backend.from()` with the storage URL as the sole parameter, for example:

```JS
// Import Madata and all supported backends and formats
import Backend from "https://madata.dev/src/index.js";

let backend = Backend.from("https://github.com/leaverou");
```

From that point onwards, common data operations are a single function call away. We provide a few examples below.



Official Authentication Provider

Supported backends

Want to learn more about Mavo? Visit mavo.io!

Figure 5.5 *Visiting a FedAP's root domain displays a list of supported services. Their metadata is also available programmatically via `/services.json`.*

5.4.1 Authentication

34.9 %

Creating a backend will automatically log in a previously logged in user without showing a login prompt (*passively*).

To show a login prompt when the user is not logged in, we can call `backend.login()`. `Backend` objects emit `login` and `logout` events so that the UI can be updated accordingly.

Assume we have the following HTML:

HTML

```html
<button id="login_button">Login</button>
<button id="logout_button" hidden>Logout</button>
<img id="avatar" alt="">
<div id="username"></div>
```

All the JavaScript we need to write to make it work is this:

JS

```js
globalThis.backend = Backend.from("https://github.com/leaverou/repo/data.json");

backend.addEventListener("login", evt => {
   login_button.hidden = true;
   username.textContent = backend.user.username;
   avatar.src = backend.user.avatar;
});

backend.addEventListener("logout", evt => {
   logout_button.hidden = true;
   username.textContent = "";
   avatar.src = "";
});

login_button.onclick = event => backend.login();
logout_button.onclick = event => backend.logout();
```

Note that while this is may appear like a nontrivial amount of code, it is nearly all UI (DOM) code: setting up events, updating the UI, and handling button clicks. The actual data interaction with the data layer has been reduced to a single line of code for each operation.

5.4.2 Editing and Reading Data

35.6 %

Now suppose we want to show a number of upvotes for the current page, and allow any webpage visitor to see the same number, and any logged in user to add one or more votes.

It only takes a few additional lines of code:

HTML

```html
<div id="upvote_count">0</div>
<button id="upvote_button">👍</button>
```

JS

```js
let upvotes = await backend.load();

if (upvotes > 0) {
   upvote_count.textContent = upvotes;
}

upvote_button.onclick = event => {
   upvote_count.textContent = ++upvotes;
   backend.store(upvotes);
};
```

5.4.3 Uploading Files

36 %

Now assuming we have a simple HTML file upload UI with an image to display the uploaded file:

HTML

```html
<input type="file" id="uploader" accept="image/*">
<button id="upload_button">Upload</button>
<img id="uploaded_image">
```

This is all the JavaScript needed to make it work:

JS

```js
upload_button.onclick = async event => {
   let file = uploader.files[0];

   if (file) {
      let url = await backend.upload(file, {path: "images/"});
      uploaded_image.src = url;
   }
};
```

These 6 lines of code take care of uploading the file to the right location, fetching a URL that can be used to display it, and updating the image element with it.

5.5 Extensibility

36.4 %

For a paradigm like Madata to be successful, extensibility is key, not simply a nice-to-have. Like Mavo, the prototype implementation of Madata supports arbitrary extension points via hooks, but here we focus on the three core extensibility points: adding a new backend, adding a new format, and adding a new authentication provider.

5.5.1 New Backends

While adding support for a new backend requires writing JS, this does not mean that superfluous complexity is acceptable. Madata follows a class hierarchy where common patterns are implemented on base classes, so authors need only specify the backend-specific details, such as the specific authentication URLs, storage location URLs, or API calls needed. The less divergent the backend is to existing standards and patterns, the more declarative the code will be.

As an illustrative example, here is the full code[2] to add support for GitLab, a popular code hosting service,

JS

```js
import { OAuthBackend } from "madata";

export default class Gitlab extends OAuthBackend {
   static capabilities = { auth: true, put: true, upload: true };
   static defaultPermissions = { read: true };
   static fileBased = true;
   static apiDomain = "https://gitlab.com/api/v4/";
   static oAuth = "https://gitlab.com/oauth/authorize";

   static urls = [
      "http{s}?://gitlab.com/:id(.+)/-/blob/:branch/:path(.+)",
   ];

   static api = {
      user: {
         get: "user",
         fields: {
            username: "username",
            name: ["name", "username"],
            avatar: "avatar_url",
            url: "web_url",
         },
      },
      file: {
         get: ref => `projects/${ ref.id }/repository/files/${ ref.path }?ref=${ ref.branch }`,
         put: ref => `projects/${ ref.id }/repository/files/${ encodeURIComponent(ref.path) }`,
      },
   };

   async put (data, {ref = this.ref} = {}) {
      return super.put(data, {
         branch: ref.branch,
         content: this.stringify(data, {ref}),
         commit_message: this.constructor.phrase("updated_file", ref.path),
      }, "PUT");
   }
}
```

The `urls` field specifies the URL patterns that should be recognized as GitLab URLs when `Backend.from()` is called. While not shown here, there is also a `knownUrls` field that specifies additional URL patterns that are recognized, but do not participate in the URL matching algorithm.

[2] Slight simplifications for readability.

5.5.2 New Formats

37.6 %

Adding support for a new format entails specifying which file extensions and MIME types [102] should automatically be recognized as the format, plus defining two methods: `parse(string, options)` and `serialize(string, options)`, which are usually wrappers around a library that does the actual parsing and serialization.

For example, here is the *complete* code to add support for TOML [103], a generic configuration format language:

JS

```js
import { Format } from "madata";
import toml from "smol-toml";

export default class TOML extends Format {
   static extensions = ["toml"];
   static mimeTypes = ["application/toml"];
   static parse = toml.parse;
   static stringify = toml.stringify;
}
```

Note that because the format's `parse` and `stringify` methods simply pass their arguments directly to the library, we did not even need to implement these two methods, we simply assigned them.

5.5.3 New Authentication Providers

37.8 %

By definition, creating a new authentication provider requires a server. However, all it requires is copying the code (*"forking"*) of a template repository and deploying it to a server. Then, the server administrator would need to register OAuth applications with the services they want to support, and add their API keys to the server's secret configuration file (`.secret.json`), and their public metadata to a public configuration file (`services.json`).

As a nice synergy with the rest of the ecosystem, the homepage showing the list of supported services could also be a Mavo app for the server administrator to edit them, with the data saved to `services.json` directly.

5.6 Discussion

5.6.1 Security and Threat Model

The Madata ecosystem has many parties: the developer, the user, other users, the FedAP, the web site hosting the Madata-using app. An important question is, how much does each entity have to trust other entities? And what power does a malicious version of each party have?

As described above, a mailicous website that tricks users into visiting the page, will not be able to access any user data, since the user will not confirm the authentication. However, a malicious website that tricks users into authenticating (such as a phishing attempt), can do a lot more damage. While this is a risk, it is not unique to Madata; it is a security risk on par with the implicit (client-side) OAuth grant flow — Madata simplicy extends it to the explicit (server-side) OAuth grant flow.

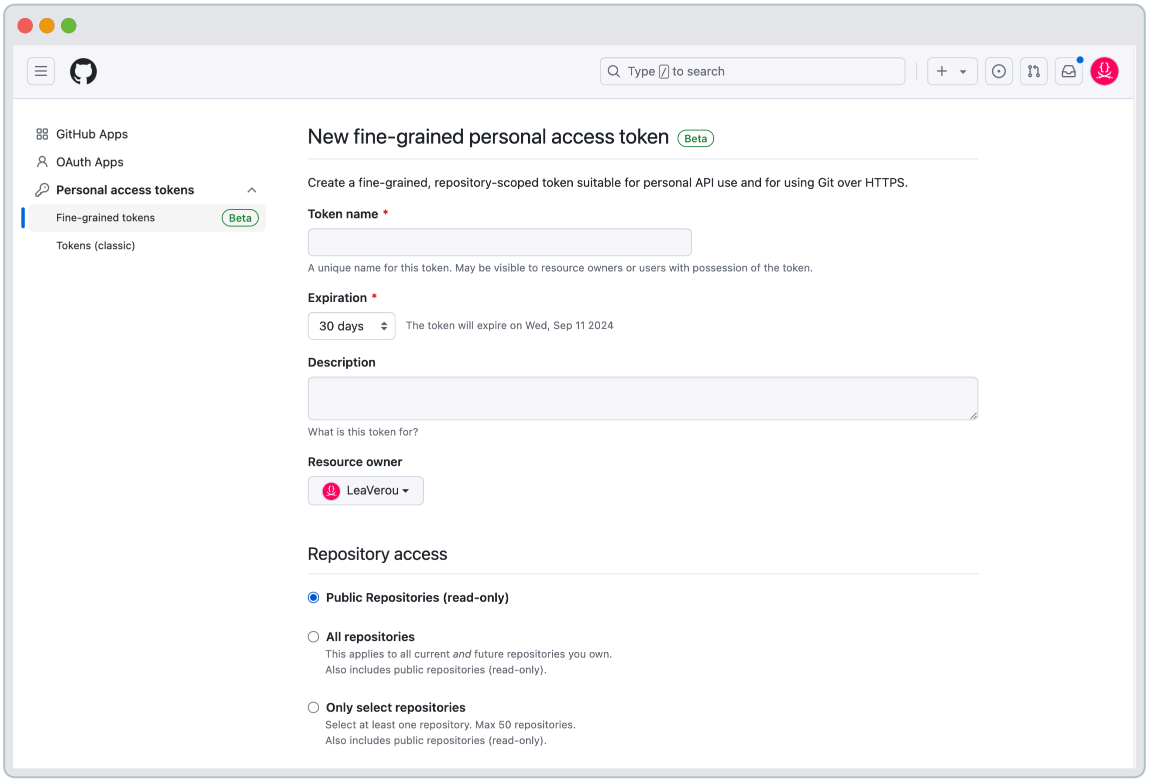
FedAPs need to be chosen carefully, as they require a high level of trust. A malicious FedAP could do a lot of damage, as it can access all user data on all services the user has used it to authenticate with. FedAPs do not need to store any user data (although nothing prevents them from doing so); once the access token is communicated to the client application, the FedAP's job is done. This means that even if a FedAP is compromised, the damage is limited to the access tokens of users who authenticate *while* it is compromised. Passive authentication (where the user has already previously logged in) is not affected, since the FedAP is not involved in that process.

Some of the risks could be mitigated by **allowing FedAPs to register multiple OAuth applications** for the same service, and distrubuting them to Madata-using applications based on a one-way hash of their URLs, so that users who have previously authenticated with one Madata application do not need to authorize more OAuth apps, but other Madata applications using the same FedAP do not need to share the same API keys with all other applications using the same service on the same FedAP.

OAuth does provide a mechanism of *scopes*[3] so that each OAuth application does not get unfettered access to user data. Using this mechanism here is tricky, because the scopes are shared across all Madata apps using the same service on the same FedAP, and thus currently Madata requsts very broad scopes when authenticating. However, perhaps the FedAP could start conservatively, and expand scopes as needed when users authenticate with a new Madata app.

[3] See tools.ietf.org/html/rfc6749#section-3.3 for the standard, and oauth.net/2/scope for a more human-readable explanation.

It should be noted that **Madata does not require use of FedAPs**. For increased security, app developers can use their own authentication server that handles communicating the access token to their application, and still take advantage of Madata's unified API for data access. However, this requires more work, as the OAuth handshake needs to be implemented on the server side. Perhaps Madata could make this easier by allowing users to register their own OAuth applications and securely store their API keys with the FedAP, which would take care of authenticating only a whitelist of app URLs with these API keys. This would still require trust in the FedAP, but it would prevent Madata applications from being prone to phishing attacks.



New fine-grained personal access token Beta

Create a fine-grained, repository-scoped token suitable for personal API use and for using Git over HTTPS.

Token name *

A unique name for this token. May be visible to resource owners or users with possession of the token.

Expiration *

30 days — The token will expire on Wed, Sep 11 2024

Description

What is this token for?

Resource owner

LeaVerou

Repository access

● Public Repositories (read-only)

○ All repositories
This applies to all current *and* future repositories you own.
Also includes public repositories (read-only).

○ Only select repositories
Select at least one repository. Max 50 repositories.
Also includes public repositories (read-only).

Figure 5.6 *GitHub is an example of a cloud service that allows users to create personal access tokens with very elaborate, fine-grained custom permissions.*

Another avenue is **user empowerment**. Many third-party services provide GUI for users to obtain access tokens directly with the service (see Figure 5.6 for an example). A planned improvement is for Madata to provide an optional way for more technical, privacy-conscious users to directly enter an access token, so that they can set their own scopes and access limits, and revoke access at any time. It could even make this easier by storing the link to the relevant settings page for each service in the FedAP's metadata.

5.6.2 Accountability

38.4 %

Madata facilitates simplicity and ease of use, but that does not come for free. Some tradeoffs were described in the previous section. Another potential tradeoff is *accountability*.

In the traditional OAuth model, application developers register their own OAuth applications to procure API keys, and are thus accountable for their actions. With Madata FedAPs, they do not need to register anything — they simply start using the service. FedAPs *do* have access to the app's requesting URL and thus the ability to block bad actors, but this is after the fact, since there is no review step involved.

However, it is questionable whether the review step in OAuth is effective at preventing bad actors, or whether it simply adds friction to the development experience for little benefit.

5.6.3 Lowering the Floor with Web Components

38.5 %

Madata is purely about the data layer. It does not construct any UI, which is left up to the application developer. While this makes it more flexible, it can also make it tedious to use, as a lot of interactions with the data layer are repetitive.

Mavo HTML is certainly a solution to this problem, but it is not a perfect one. While Mavo provides a very high abstraction level, the loss of control can be frustrating for programmers. To bridge this gap, Madata implements a set of Web Components[4] that encapsulate specific UI interactions such as authentication, or autosave with throttling.

For example, a `<madata-auth>` custom HTML element can be used to display user information and authentication controls with just a single line of HTML:

HTML

```html
<madata-auth src="https://github.com/leaverou/somerepo"></madata-auth>
```

[4] madata.dev/components

5.6.4 More Declarative Syntax

38.7 %

When Madata begun, as a Mavo component, it was a lot more imperative. To support a new backend, authors had to implement low-level `get()`, `put()`, `login()`, `upload()`, `getUser()`, etc. methods. While it did provide abstractions that made it palatable to implement these methods, it was still a lot more boilerplate. Over time, as more backends were added, common patterns emerged, and their code was abstracted away into base classes that only set static class fields as inputs.

It is an open question how far this process can go. Could we reach a point where **adding a new backend can be done entirely by specifying metadata**, without requiring any imperative code? And if so, would that bring it within reach of non-programmers?

5.6.5 Dynamic, Decentralized Backends

38.8 %

The discerning reader will have noticed that the promise of storage URLs is not yet fully realized. For Madata to support a given service, it needs to *"know"* about it in advance, and for a storage backend to exist and have been imported for that particular service. For true decentralization, there should be a standard protocol to enable services to declare all the necessary information that Madata needs to know, so that the necessary backend can be generated on the fly.

There is already such a mechanism: *well-known URIs*, defined in IETF RFC 8615 [104]. A well-known URI is a URI [RFC3986] whose path component begins with the characters `/.well-known/`. While the registry of well-known URIs is maintained by IANA[5], nothing prevents services from using well known URIs that are not registered, and those in widespread use would likely later become standardized. One could imagine a URL like `/.well-known/madata` or `/.well-known/madata.json` that would contain all the necessary information for Madata to interface with the service.

Additionally, authentication providers could also provide the necessary code or metadata for Madata to interface with a new service.

This is not without new risks. When allowed backends are imported by the programmer, they are in control of what code runs in their application. If backends can be generated on the fly, this control is lost. And if URLs are also user-supplied, an adversarial user

[5] iana.org/assignments/well-known-uris/well-known-uris.xhtml

could easily host malicious code on their own server, and then use a URL from said server.

The main way to mitigate this risk is to **turn code exchange into data exchange**. This could happen either at the point of interfacing with the backend, by only allowing entirely declarative backends from third-party services (see previous section), or at the point of data I/O, by running all untrusted third-party code in a sandboxed environment, where it can only access its inputs, and nothing else. Then, the sandbox communicates with the main application by exchanging plain *data*, which is generally safe.

Of these, declarativeness is a preferable approach; not only due to the benefits of declarative languages in general, discussed in previous chapters, but also because it is less likely to negatively affect ergonomics.

5.7 Conclusion

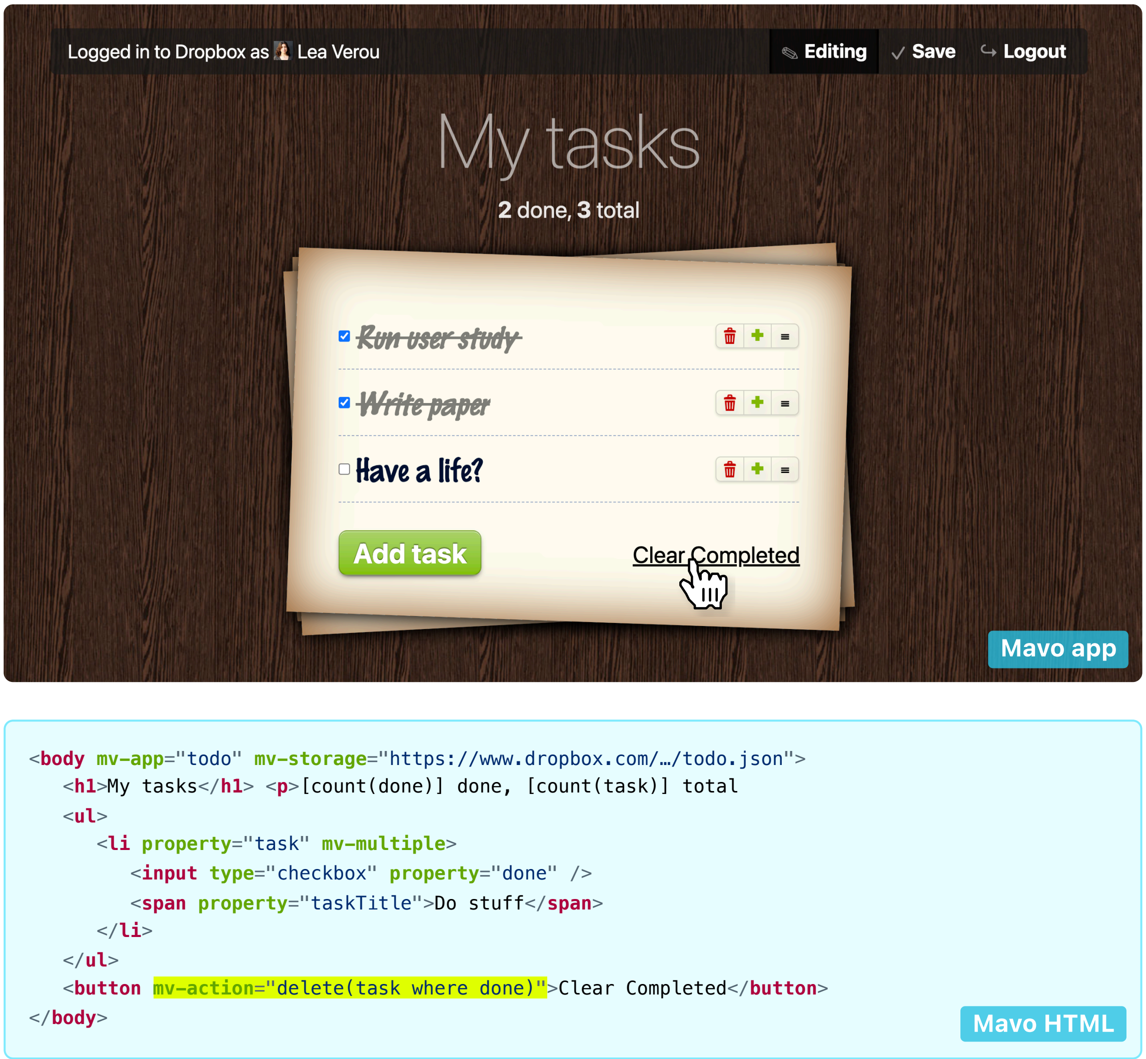
39.1 %

In this chapter, we presented Madata, a set of protocols and client-side APIs, designed to facilitate data ownership by democratizing data access. By unifying the processes of reading and writing data across diverse storage services and formats through a single API, Madata addresses not only one of the biggest usability cliffs in modern web development, but also suggests a model that could facilitate data portability more broadly.

CHAPTER 6

Extending a Reactive Formula Language with Data Update Actions

4,947 words (15 min read)



```
<body mv-app="todo" mv-storage="https://www.dropbox.com/…/todo.json">
   <h1>My tasks</h1> <p>[count(done)] done, [count(task)] total
   <ul>
      <li property="task" mv-multiple>
         <input type="checkbox" property="done" />
         <span property="taskTitle">Do stuff</span>
      </li>
   </ul>
   <button mv-action="delete(task where done)">Clear Completed</button>
</body>
```

Figure 6.1 *The* ***complete*** *HTML for a fully-functional To-Do app made with Mavo, with a data update action for deleting completed items. No JavaScript is needed.*

6.1 Introduction

Many systems and languages exist for assisting novice programmers to manage information, and/or create CRUD applications for this purpose. They range from the well known commercial spreadsheet systems to more complex application builders [54, 80] or simplified declarative languages [11, 40].

These usually generate an editing interface for elementary data manipulations (editing a data unit, inserting items, deleting items) and a mechanism for lightweight reactive data computation. As an example, in spreadsheets the editing interface is the grid itself, and the computation is the spreadsheet formula.

These tools typically offer only direct editing of specific data items by the end-user. Affordances may also be provided for aggregating certain kinds of commonly needed mass modifications. A few examples of these would be selecting multiple items for deletion or move, adding multiple rows or columns, or the spreadsheet fill handle. However, the set of potential data mutations is infinite, and it is not practical to predefine controls for every possible case. For more complex automation of data edits, users are typically directed to scripting or SQL queries. Learning a scripting language is almost as hard as learning a programming language, and SQL queries quickly become complicated when nested schemas are involved, which are represented by multiple tables and foreign keys [51].

Mavo [11] is an HTML language extension for defining CRUD Web applications by annotating a static HTML mockup to separate UI from data. A `property` attribute indicates that an element is data, and an `mv-multiple` attribute makes it repeatable i.e. turns it into a *collection*. Based on this markup, Mavo generates a suitable editing interface. Mavo stores its data locally or on a cloud service.

Many applications benefit from presenting values *computed* from their data, so Mavo implements a reactive expression language called *Formula²*, similar to what can be found in spreadsheets, for lightweight computation. Expressions can be placed anywhere in the HTML and are denoted by square brackets (`[]`) or certain attributes(`mv-if`, `mv-value` etc). An expression can reference properties, with its evaluation depending on the location of the expression relative to the referenced properties in the data tree. Referencing a multi-valued property on or inside a collection item resolves to its local value on that item, whereas referencing that property name outside the collection resolves to *all* values.

All operators and most functions can be used with both single values and lists. This makes many common expressions concise.

Our previous work [11] provided evidence that Mavo empowered users with no programming experience to author fully functional CRUD applications. But Mavo offered only direct editing of individual data items, while many applications call for richer, programmatic modification of large collections of data simultaneously. For example, while a simple To-Do list could be easily implemented with a few lines of HTML+Mavo, clearing all completed items was not possible.

Mavo did offer controls for deleting individual items, but no way to specify such actions that programmatically delete certain items.

6.1.1 Our Contribution

40.3 %

In this work, we extended Mavo with a new HTML attribute, `mv-action`, for specifying programmatic data updates. Its value describes the data mutation as an expression, and it is placed on the element that will trigger the action by clicking. The expression leverages Formula²'s existing expression syntax as much as possible, but adds functions that *modify* the data. Formula²'s filtering operator, as well as its data specification syntax (`group()` and `list()`) were also originally defined to facilitate actions. A short example implementing bulk deletion can be seen in Figure 6.1.

Given the small user bases of most research systems, there is little data on any kind of user requests. However, when it comes to spreadsheets, there is a large volume of user questions indicating a clear need for a way to programmatically manipulate data. It is indicative that searching for *"button to change value excel"*, which is *considerably* more specific, both in terms of task and in terms of system, yields at least 400 relevant questions on stackoverflow.com alone! [1]. Most askers did not mention their exact use case, but the abstract task they were trying to accomplish, which was primarily being able to press a button to change the content of one or more cells.

Our hypotheses are that (a) the set of primitives we have chosen is expressive enough to meaningfully broaden the class of data management applications that can be created

[1] We measured this by restricting our Google search to stackoverflow.com and inspecting all results in the 50 pages that were accessible

without undue complexity, and (b) novice web authors can easily learn to specify programmatic data mutations. To examine the first hypothesis, we list a number of case studies of common interactions and how they would be expressed with our data update syntax. To examine the second hypothesis, we conducted a user study with 20 novice web developers writing a variety of data mutations, using first their own imagined syntax and then ours (Section 7.2).

We found that the majority of users were easily able to learn and apply these data mutation expressions with 90% of our participants getting two thirds of the questions right on first try, with no iteration. We even found that in many cases, our syntax was very close to their imagined syntax.

Although the *presentation* of information is an important part of Mavo, our work here focuses on and extends the *computational power* of Formula^2. We believe our work suggests more broadly a way to increase the power of such functional reactive programming environments, including spreadsheets, for novice users.

6.1.2 End-User Reactive Programming

40.7 %

Although the *presentation* of information is an important part of Mavo, our work here focuses on and extends the *computational power* of Formula^2, an expression language with power similar to spreadsheet functions. And while we study Mavo, we believe our work has broader implications for such functional reactive programming languages.

As in spreadsheets, all Formula^2 expresions are *reactive*, updating immediately if their arguments change. But Mavo diverges from spreadsheets in two potentially useful ways. First, instead of referring to arguments through a grid-based coordinate system, Formula^2 uses the *names* defined in the data (through the `property` attribute in Mavo HTML).

Unlike the table model of spreadsheets, Mavo provides a *hierarchical data model* of objects containing properties and other objects, as well as collections. This data model permits storage and reference of more complex objects than spreadsheets. Nesting offers a limited amount of the power that *database joins* do, to connect relationships between multiple tables. The nested presentation structure of HTML makes it natural and easy for authors to define these nested data models by the way they look on the web page.

One of the primary reasons we chose to extend Mavo with this functionality is Formula^2's property reference mechanism (Section 4.4.1): Every property can be referenced from anywhere and the value depends on the location of the expression relative to the property in the data tree. Referencing a multi-valued property on or inside a collection item resolves to its local value on that item, whereas referencing that property name outside the collection resolves to *all* values. All operators and most functions can be used with both single values and lists. This makes many common expressions concise, a feature that extends to our mutations as well.

6.2 Related Work

Our work extends the Mavo language [11]. A full discussion of related work can be found there. In summary, many platforms and systems have been developed over the past few decades to help web authors build web applications, many of which are presented in Chapter 2. Some of these tools target web developers with limited programming and database knowledge [44, 52], allowing them to make programmatic changes to the data using SQL queries.

Others were developed for novice web designers who are interested in rapid development of web applications [27, 55, 105], with spreadsheets or a variation of spreadsheets as a back-end, but with limited or no mechanism to make data updates programmatically. Two lines of prior work are especially relevant, one on creating systems for graphically designing database schemas as well as building web page content, depending on SQL queries to make automated data updates (Section 2.3), and the other on developing WYSIWYG tools for creating web pages with spreadsheets as the back-end, with limited data update capabilities.

6.3 Mavo Data Update Language

6.3.1 The `mv-action` HTML Attribute

Data updates are specified via an `mv-action` HTML attribute, which can be placed on any element. Its value is an expression that describes the action that will be performed. The action is triggered by clicking (or focusing and pressing spacebar, for keyboard accessibility), as that appears to be the most common way to invoke an action on most types of GUIs.

6.3.2 Data Mutation Functions

41.1 %

We extended Formula^2 with four data modification functions (brackets indicate optional arguments):

- `set(reference, value)`
- `delete(ref1, [, ref2 [, ref3, ...])`
- `add(collection [, data] [, position])`
- `move(from, to)`

The first three are analogous to the SQL primitives UPDATE, INSERT, DELETE, whereas the latter is a composite mutation (delete, then add). Per [106], we used simple English words that have a largely unambiguous meaning. These functions are **only** available in expressions specified with the `mv-action` attribute. Regular Mavo expressions remain side-effect free.

We kept these functions minimal, to delegate selection and filtering logic to Mavo expressions. This maximizes the amount of computation specified in a reactive fashion, which is easier for novices to work with [107, 108].

41.2 %

Changing Values with `set(ref, newValue)`

The `set()` function only accepts two arguments: what to set, and the new value(s). However, both arguments can be multi-valued, in which case they are applied pairwise, in the same way as Mavo operators, i.e. if the lengths are different, the operation is only applied to the corresponding items, and the extra items are just returned with no modification.

Deleting Items with `delete(ref1, ref2, ...)`

The `delete()` function deletes all collection items passed to it via one or more parameters.

41.3 %

Adding Items with `add(ref [, data] [, position])`

The `add()` function adds one or more new items, optionally pre-filled with data (if the `data` parameter is used) and optionally at a specific position in the collection (if the `position` parameter is used). The first parameter can be a reference to a collection, or a collection item. In the latter case, the target defaults to the collection containing the referenced item, at the position of that item. This for example makes it easy to replicate Mavo's own "Duplicate item" behavior by invoking `add(name, name)`.

Moving Items with `move(from [, to] [, position])`

This function moves items to a different position within the same collection or to a different collection. Originally we did not plan to provide a `move()` function, since it can be simulated with a `delete()` and then `add()`. However, `delete()` displays UI for undoing the deletion which would be jarring in the case of moving items. Furthermore, even if that were not a problem, a separate `move()` function provides semantic clarity in the author's code.

41.4 %

Implementing References to Mavo Nodes

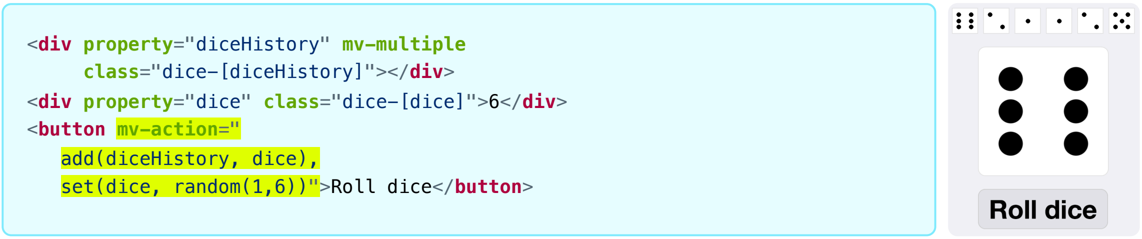
Previously, there was no way to infer which Mavo node had produced which data from within a function. The data passed to functions was essentially a combination of plain JavaScript objects, arrays, and primitives. This had the advantage that authors who were comfortable with a little JavaScript could easily extend Mavo by writing their own functions in plain JS. However, with data actions, it is *essential* to be able to trace values back to the Mavo node that produced them, so that the UI can be updated accordingly.

We added a non-enumerable Symbol[2] property to objects that correspond to Mavo Node data. This way, these node references do not interfere with normal expressions, but Mavo data mutations can still retrieve the nodes referenced and manipulate them. We also needed to convert all expression data to objects instead of primitives so they could have properties attached to them and used a special **Null** object for empty values, instead of the JavaScript `null` value (which also could not have properties).

Furthermore, we had to modify the implementation of several Formula^2 functions so that they do not inadvertently break these reverences by creating new data.

6.3.3 Multiple Sequentially Executed Function Calls per Formula

41.6 %



```
<div property="diceHistory" mv-multiple
     class="dice-[diceHistory]"></div>
<div property="dice" class="dice-[dice]">6</div>
<button mv-action="
   add(diceHistory, dice),
   set(dice, random(1,6))">Roll dice</button>
```

Figure 6.2 *A dice rolling application with a history of past dice rolls.*

[2] ecma-international.org/ecma-262/6.0/#sec-symbol-objects

Update actions often consist of multiple elementary updates, executed sequentially. The Dice Roller in Figure 6.2 demonstrates an example (`add()`, then `set()`). To facilitate this, we extended Formula^2 to support **multiple function calls in the same expression**. These can be separated by commas, semicolons, whitespace, or even nothing at all.

6.3.4 Using `if()` with Data Updates

41.9 %

Formula^2 provides an `if()` function that works similarly to the `IF()` function in spreadsheets, except it can also be applied element-wise if one or more of the arguments are arrays. In early pilots of the user study, we realized that subjects may try to use `if()` to filter the target of a data update. For example, in the data update "Rename every person older than 40 to Chris", it would be natural to use an expression like `if(age > 40, set(name, ’Chris’))`. However, given the way Formula^2 works, this would not filter the target of the update (`name`)), but merely the result of the `set()` function, a behavior which – while consistent with how functions work – is highly unlikely to meet author expectations.

To enable such expressions to produce the expected result, we defined a new set of data mutation functions: `setif()`, `addif()`, `deleteif()`, `moveif()`, whose first argument is a condition, and otherwise work the same way as their aforementioned counterparts. Then, we rewrite data update calls inside `if()` expressions to use these functions instead.

6.3.5 Multiple Function Calls per Expression

42 %

Since regular Formula^2 expressions have no side effects and only produce one output, there was no way to specify more than one sequential function call (and no reason to do so). However, with data mutations, there may sometimes be a need to perform more than one update as part of the same action. Figure 6.2 demonstrates an example of this. To enable this, we modified Formula^2 to allow multiple function calls, which can be separated by commas, semicolons, whitespace, or even nothing at all.

Implementation

42.1 %

We noticed that Formula^2's expression parser already parsed adjacent function calls as AST nodes of type "Compound" but serializes such tokens to JavaScript with no separator, which would result in an error when the JavaScript code produced is executed. Therefore, to enable multiple function calls we modified Formula^2's serialization to serialize Compound tokens into comma-separated values, as commas are an existing JavaScript operator that merely returns the last value.

Since the same Compound token was produced regardless of whether the function calls were separated by commas, semicolons, whitespace, or even nothing at all, providing this kind of flexibility essentially came for free. Another fortunate side effect of this modification was that this also enabled function arguments that are separated by whitespace.

6.4 Example Use Cases

42.2 %

Part of our argument is that Data Update Actions have sufficient power to easily specify a broad range of programmatic data manipulations in applications. To support that argument, we outline a few common interactions below. None of these can be implemented with the original Mavo alone. In each case, we show the source code, primarily HTML and original Mavo syntax; and we highlight the code leveraging data actions.

We first demonstrate how a few lines of Mavo can be used to define a number of frequently used general-purpose UI widgets, then present a few specific applications such as the ubiquitous shopping cart.

6.4.1 Generalizing Existing Mavo Data Updates

42.3 %

All Mavo data update controls except drag and drop can be expressed concisely as data actions, which facilitates UI customization. The following examples assume the updates are modifying a collection with `property="item"`. Note that `index` (or `$index`) is a built-in Mavo variable that resolves to the index of the closest collection item, starting from 0.

Action	Data update Expression
Add new item button *(outside collection)*	`add(item)`
Add new item after current	`add(item)`
Duplicate current item	`add(item, item)`

Delete current item	`delete(item)`
Move up	`move(item, index - 1)`
Move down	`move(item, index + 1)`

42.4 %

Heterogeneous Collections

It is common to have lists of items that share some properties but not others. In object-oriented programming, we would say objects that are subclasses of the same superclass. For example, a blog might have two types of posts: text and picture. All posts have a `title` and a `date`, but picture posts have an `image` property, and `text` posts have a `text` property.

This pattern can already be implemented in Mavo, by having a `type` property that determines the type of the item, and then using `mv-if` to show the right properties:

HTML

```html
<article property="post" mv-multiple>
   <meta property="type" mv-options="text, image">
   <h2   property="title"></h2>
   <img  property="image" mv-if="type = 'image'">
   <div  property="text"  mv-if="type = 'text'">
</article>
```

However, the user interaction is suboptimal: to add an image post we need to add a new post, then change its type to “image”. With data actions, we can make non-default types first-class citizens, by having different add buttons for them:

HTML

```html
<article property="post" mv-multiple>
   <meta property="type">
   <h2   property="title"></h2>
   <img  property="image" mv-if="type = 'image'">
   <div  property="text"  mv-if="type = 'text'">
</article>
<button mv-action="add(post, type: 'image')">New 🖼 </button>
<button mv-action="add(post, type: 'text')">New 📄 </button>
```

43.3 %

6.4.2 Common UI Widgets

One natural class of use cases for data actions is in creating rich new UI widgets. These widgets generally present underlying data from the traditional model in some novel fashion. But the widgets also tend to come with their own internal *view model* describing the state of their presentation. Data actions can be used to control the state of the view model, and thus to manage the widget's data presentation.

Select All

It is common to offer an affordance to simultaneously check or uncheck all items in a list.

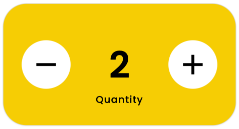
HTML

```html
<button mv-action="set(selected, true)">Select All</button>
<button mv-action="set(selected, false)">Unselect All</button>
<div property="item" mv-multiple>
   <input type="checkbox" property="selected" />
   <!-- other content -->
</div>
```

43.6 %

Spinner Control

While spinners (a widget that can increment or decrement a number) are native to HTML5, this is still useful when a customized presentation is desired, such as the one here.



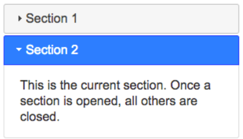
HTML

```html
<button mv-action="set(number, number - 1)">-</button>
<span property="number">1</span>
<button mv-action="set(number, number + 1)">+</button>
```

43.8 %

Accordion / Tabs

An accordion permits a user to show one of several distinct sections of content, while the rest are hidden. The same markup, with different CSS, could also be used to implement a tabbed view.



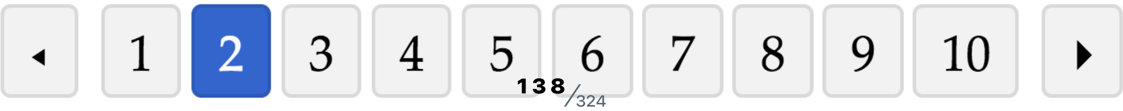
This is the current section. Once a section is opened, all others are closed.

HTML

```html
<details property="prop" mv-multiple open="[open]"
      mv-action="set(open.$all, false) set(open, true)"">
   <summary property="title"></summary>
   <meta property="open" />
   <!-- content -->
</details>
```

44.1 %

Pagination



The following markup implements a working pagination widget and the content it paginates. It uses the Mavo `mv-value` attribute to generate a dynamic collection of page markers, then `mv-action` to make them clickable. An expression on each item controls whether it should be displayed based on the current page.

HTML

```html
<meta property="current" content="1">
<meta property="per_page" content="10">
<meta property="pages" content="[ceil(count(item) / per_page)]">

<a mv-action="set(current, current - 1)" mv-if="current > 1">◂</a>
<a mv-multiple mv-value="1 .. pages" mv-action="set(current, page)">1</a>
<a mv-action="set(current, current + 1)" mv-if="current < pages">▸</a>

<!-- Content to be paginated: -->
<div property="post" mv-multiple hidden="[page != current]">
   <meta property="page" content="[ceil((index + 1) / per_page)]">
   <!-- content of one item -->
</div>
```

Slideshow / Carousel

44.8 %

A slideshow is essentially pagination with one item per page, frequently used for photos or other large objects.

HTML

```html
<meta property="current" content="0">
<div property="slide" mv-multiple hidden="[current != index]">
   <!-- content -->
   <button mv-action="set(current, next.index or 0)">▸</button>
   <button mv-action="set(current, previous.index)">◂</button>
</div>
```

Sorting Table by Clicking on Column Header

45.1 %

According to a wide survey of HTML authors in 2023 ([109]), the top missing element in HTML is data tables with filtering and sorting.

Mavo provides an `mv-sort` attribute whose value is the property name(s) to sort by. Data actions can dynamically change that via a helper property that holds the property name.

Name	Age ▾
Tarfah	30
Lea	32
Karger	50

HTML

```html
<meta property="sortBy" content="">
<table>
   <tr>
      <th mv-action="set(sortBy, 'name')">Name</th>
      <th mv-action="set(sortBy, 'age')">Age</th>
   </tr>
   <tr property="person" mv-multiple mv-sort="[sortBy]">
      <td property="name"></td>
      <td property="age"></td>
   </tr>
</table>
```

This does not address clicking twice for ascending sort, which can be done by introducing a conditional:

HTML

```html
<th mv-action="set(sortBy, if(sortBy = '-name', '+name', '-name'))">Name</th>
<th mv-action="set(sortBy, if(sortBy = '-age', '+age', '-age')))">Age</th>
```

To visually communicate the sorting criteria, we could either use an expression like `if(sortBy = '-name', '▲', if(sortBy = '+name', '▼', ''))` in each header cell, or use CSS.

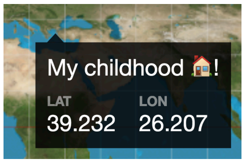
An alternative solution would be to overwrite the collection with a sorted version of itself each time the header is clicked:

HTML

```html
<th mv-action="set(person, sort(person, name)">Name</th>
<th mv-action="set(person, sort(person, age)">Age</th>
```

45.9 %

Adding Map Pins / Calendar Events



This example positions each collection item over a 720 × 360 equirectangular map by storing the mouse position at the time of clicking. Similar logic can be used for adding events to a calendar view.

HTML

```html
<img src="map.svg" mv-action="add(event, hoverPos)"/>
<meta property="hoverPos"
      content="group(lat: 90 - $mouse.y / 2, lon: $mouse.x / 2 - 180)">

<div property="place" mv-multiple
     style="top: [  2 * (90 - lat) ]px; left: [ 2 * (lon + 180) ]px">
```

HTML (continued)

```html
  <meta property="lat" />
  <meta property="lon" />
  <!-- other properties -->
</div>
```

E-shop: Add to Cart Button

46.4 %

Mavo supports direct manipulation to move an item from one collection to another by dragging it. But data actions enable authors to specify more natural mechanisms for common cases, such as the "Add to Cart" button of an e-shop.

This is a specific instance of copying an item from one collection to another, which Mavo offers natively through drag and drop. But nobody wants to force users to drag an item to a shopping cart! Instead, a dedicated button provides a much quicker interaction.

HTML

```html
<div property="product" mv-multiple>
   <!-- name, image etc properties -->
   <button mv-action="add(cart, product)">Add to cart</button>
</div>
<div property="cart" mv-multiple>
   <!-- subset of product properties -->
</div>
```

Invoice Manager: Two Ways to Copy Customer Details

46.6 %

Another commonly needed functionality is copying data from one item to another, for example copying shipping address to billing address, or this example of copying invoice details.

HTML

```html
<div property="invoice" mv-multiple>
   Copy customer details from invoice #<input property="copy" />
   <button mv-action="set(customer, customer.all where id = copy)">Go</button>

   <button mv-action="add(invoice, customer: customer)">
      New invoice for this customer</button>
   <!-- invoice properties -->
</div>
```

6.5 Discussion & Future Work

46.9 %

6.5.1 Extending Mavo HTML vs Extending Formula²

Over the years, we considered many alternatives for offering data update functionality in Mavo in a sufficiently general way. Early on, we were focused on designing an HTML-based syntax for specifying actions in a human-readable way.

An early syntax sketch included an `<mv-action>` element with the action type, property to operate on, value to set it to, etc. as attributes, which could either be fixed values, or use the bracket syntax to embed expressions.

HTML

```html
<button mv-action="#foo">Click me</button>
<mv-actions id="foo">
    <mv-action type="add" property="propertyName" value="[someProperty]">
    <mv-action type="edit" property="myProperty" value="[foobar]">
    <mv-action type="delete" property="someCollectionProperty" value="5">
</mv-actions>
```

While this syntax may have prevented some of the user errors we observed in our study, the learnability of the Formula² syntax appears to be sufficient that the verbosity of such a syntax does not appear to be justified. That said, it would be a good future direction to explore, and compare how it performs compared to the Formula² syntax.

This consideration is not specific to Mavo — the tension of whether to add new functionality via the user interface or the formula language is universal and relevant to many low-code and no-code tools. In Mavo, the UI is the HTML the author is writing, whereas in a GUI builder it would be the visual controls.

47.4 %

6.5.2 Improving the Learnability of Our Syntax

We will apply the user study findings to iterate on our syntax and make it more natural. Many participants wanted to use a `to` keyword, which can be easily added. Several participants were confused about the `group()` function, what it does, and when it is needed, so we will examine whether it is possible to design the language in such a way that `group()` is not required, possibly by using a variable number of arguments in `add()` or by requiring plain parentheses instead of `group()`.

We may decide to special case certain patterns to match user expectations: predicates will be allowed as the sole argument in `delete()` and will target the closest item.`set(a = b)` could be rewritten as `set(a, b)`. Repetition in `where` can be avoided by expanding `property where value` to `property where property = value`. Underspecified assignments, such as `set(age + 1)` could target the first named token.

We also need to improve the syntax for tasks which filter one property and set another (Q14 and Q16), since our user study indicated clear problems with the current syntax.

Perhaps exposing the `setif()` etc functions we have implemented would be sufficient or otherwise modifying the syntax of `set()`.

6.5.3 End-User Functional Reactive Programming

47.6 %

Our work has explored ways to extend Formula², a functional reactive programming language, to permit end-users to specify complex data updates. These ideas may generalize beyond Mavo. Spreadsheets are used to create quite complex data management applications, and we believe that needs for complex data updates are likely to arise in such applications. If we can provide a suitable syntax for end-users, we can broaden both the range and the fidelity of tools they can create in their spreadsheets.

6.5.4 More Triggers

47.7 %

Currently, our data actions are triggered by clicking or form submission. In the future we are planning to add the ability to specify different triggers, such as double clicking, keyboard shortcuts, dragging, or mousing over the element.

From a preliminary analysis of use cases, we found that unlike in applications such as interactive visualizations [110], CRUD applications rarely require the same level of interaction richness, and clicking appears to suffice for many data update use cases.

Furthermore, if actions are available, even if only triggered by clicking, authors can write JavaScript for the event handling, and have it programmatically click an element. For example, to trigger a data action by mouse over, one could do:

HTML

```html
<button onmouseover="this.click()" mv-action="...">Hover me</button>
```

While suboptimal, it's still a lot easier than specifying the data mutations entirely in JavaScript.

That said, there is certainly value in expanding the set of triggers available to authors, especially since over time Mavo is expanding beyond strictly CRUD applications (see Chapter 8). This could be done via an HTML attribute (e.g. `mv-action-trigger` or `mv-action-event`) whose value would be an "event selector", as defined by Satyanarayan et al. [110]. Event selectors facilitate composing and sequencing events together, allowing users to specify complex interactions very concisely.

Furthermore, since many of our target users are familiar with CSS selectors, they might be able to transfer some of that knowledge.

Another (potentially complementary) approach would be to also use a function-based syntax as the value of this attribute, and expose event-related information as declarative variables.

The majority of events in JavaScript represent a meaningful change in state of some natural variable: the key that is down, mouse x or y, the selected element, the hovered element. It is natural to expose these variables in expressions, and Formula^2 already does this to a small degree, by exposing special properties (Chapter 4), such as `$mouse.x`, `$mouse.y`, `$hash`, `$now` and others, which update automatically, even when no data has changed. We could expand this vocabulary to expose more event information (such as which key is pressed), which would also be useful for Formula^2 more broadly.

This leads naturally to thinking of triggering off changes to variables or changes to expressions over those variables. It remains a fascinating open question to resolve which metaphor is most intuitive for novice programmers. This approach would also make it possible to use data changes as triggers. While powerful, this could easily result in cycles, which may confuse novices.

6.5.5 Blurring the Line Between Computed and Regular Properties

48.2 %

"Computed properties" in Mavo are properties whose value is an expression. These can be simple primitives, or entire data structures by using the `mv-value` attribute on objects (groups) or collections.

Currently, data updates to *computed properties* (i.e. properties whose value is an expression) are ignored, since these properties are not editable by Mavo's editing interface either. However, there are valid use cases where one may want to temporarily replace or "freeze" the value of an expression, such as a stopwatch with a pause button.

More work is needed to determine the best way to address these cases. We *could* allow data updates to work with computed properties, and just remove the expression (essentially freezing them in time), but it is unclear how to reverse this (`clear()`?).

There could be a special syntax to declare that the value to set should not be a one-time evaluation but a new reactive formula, essentially getting our HTML authors to declare functions. This could be used to blur the distinction between computed properties and regular properties; any property could be set to a reactive formula and become temporarily computed, and any computed property can be set to a value and become a regular property. However, this is a higher level of abstraction, which may be confusing for novices.

6.6 Conclusion

48.3 %

This chapter extends Mavo HTML and Formula^2 by adding programmatic data updates that are triggered by user interaction. Our user study (Section 7.2) will show that HTML authors can quickly learn to use this syntax to specify a variety of data mutations, significantly expanding the set of possible applications they can build, with only a little increase in language complexity.

CHAPTER 7

Evaluation & Evolution

12,633 words (37 min read)

7.1 First Lab Study: Mavo Prototype

This was the first set lab of lab studies [11] we conducted on very early prototypes of Mavo HTML (nee Wysie), Formula^2 (nee WysieScript), and Madata (which did not yet have a separate name or implementation). Data Update Actions were not yet supported.

The primary focus of this first set of studies was to evaluate the usability of Mavo HTML, Formula^2 as it relates to Mavo, and to a much lesser extent Madata as it relates to Mavo.

48.6 %

7.1.1 Relevant Context

To better understand the results, it is essential to discuss the design of Mavo at the time of the study. since its constituent languages and components have evolved significantly since then (partly thanks to the findings from these studies!).

Mavo HTML

`data-*` over `mv-*`

Initially, all Mavo attributes (that were not part of any existing standard) used the prefix `data-` rather than `mv-`. For example, `mv-multiple` was then `data-multiple`. While HTML handles any attribute name well, to prevent future HTML features from breaking existing websites, the HTML5 specification [111] defines that attributes beginning with `data-` are reserved for custom data attributes, and any other unknown attribute should be considered invalid.

This leaves third-party languages, libraries, or frameworks that define multiple attributes with having to choose between using invalid HTML, using `data-` prefixes which makes it unclear which attributes belong to what third-party technology, or use a verbose prefix like `data-mv-`. The first version of Mavo HTML went with the second option, favoring HTML validity over brevity or clarity. Later versions switched to supporting *both* the former (`mv-*`) and the latter (`data-mv-*`), and eventually the latter was dropped for simplicity (and due to lack of use).

No `mv-app` Attribute

At the time there was no `mv-app` attribute (or even a `data-app` one) — `data-storage` served double duty: It *both* enabled Mavo functionality on an HTML subtree *and* specified the data location.

The original thinking was that since enabling Mavo functionality on a subtree without also asking Mavo to do *something* does not produce any visible change, which would violate the design principle that incremental user effort should result in incremental value. Since an explicit opt-in does not produce any visible change, it was creating a highly likely error condition, where authors use `data-storage` but forget to enable Mavo functionality on the subtree.

However, this was also a textbook case of undersirable concept overloading [112]. First, some awkward situations, such as when applications did not need to store data anywhere (e.g. a mortgage calculator), yet still had to specify a `data-storage` attribute with no value to enable Mavo functionality. Second, it seemed unclear why we privileged that particular attribute, so we later added more attributes to the set of attributes that could enable Mavo functionality, making it nontrivial to figure out which elements on a page were Mavo applications.

Furthermore, there was no way to provide a unique identifier for an application, since there was no attribute that did not also serve another purpose. Eventually, all of these issues led to the introduction of the `mv-app` attribute.

No Declarative Expression Evaluation (`mv-if`, `mv-value`, etc)

At the time, conditional logic and computation in general was only possible with expressions. The `mv-if` and `mv-value` attributes were added to Mavo as a result of this study.

Formula²

This initial prototype of Formula² used `iff()` instead of `if()`.

Madata

As a proof of concept, Madata at the time only supported local storage and Dropbox.

7.1.2 Study Design

In our evaluation, we examined whether Mavo could be learned and applied by novice web authors to build a variety of applications in a short amount of time. In order to understand both the usability and flexibility of Mavo, we designed two user studies. For a first STRUCTURED study, we authored static web page mockups of two representative CRUD applications and then gave users a series of Mavo authoring tasks that gradually evolved those mockups into complete applications. This study focused on learnability and usability.

For a second FREESTYLE study, *before* telling users about Mavo (so that they would not feel constrained by its capabilities), we asked them to create *their own* mockup of an address book application. Then, during the study, we asked them to use Mavo to convert their mockups into functional applications. This study focused on whether Mavo's capabilities were sufficient to create applications as envisioned by users.

We carried out the two user studies using three applications. The applications were designed with hierarchical data to test users' ability to generate hierarchical data schemas and perform computations on them.

To facilitate replication of our study, we have published all our study materials online[1].

7.1.3 Preparation

We recruited 20 participants (mean age 35.9, SD 10.2; 35% male, 60% female, 5% other) by publishing a call to participation on social media and local web design meetup groups.

Of these, 13 performed only the STRUCTURED study, 3 performed only the FREESTYLE study, and 4 performed both.

[1] mavo.io/uist2016/study

	HTML	CSS	JavaScript
Beginner	0	4	13
Intermediate	8	5	6
Advanced	9	6	1
Expert	3	5	0

Table 7.1 *Participant familiarity with web development languages.*

All of our participants marked their HTML skills as intermediate (rich text formatting, basic form elements, tables) or above. However, most (19/20) described themselves as intermediate or below in JavaScript (Table 7.1). When they were asked about programming languages in general, 13/20 described themselves as beginners or worse in *any* programming language, while 7/20 considered themselves intermediate or better.

	JSON	HTML Metadata	SQL
Never heard of it	0	10	0
Heard of it	6	6	5
Can read it	2	1	3
Can edit it	8	3	8
Can write it from scratch	4	0	5

Table 7.2 *User study participants' familiarity with data specification languages.*

In addition, when we asked participants about their experience with various data concepts, only 4/20 stated they could write JSON, 5/20 could write SQL, and none could write *any* type of HTML metadata (RDFa, Microdata, Microformats).

7.1.4 Pre-Study Interview & Tutorial

49.5 %

We began by asking participants a series of open-ended questions about their experience with web development and web publishing. We asked them what kind of applications and functionality they wished they could create but could not due to lack of time or ability.

Before either study, we gave each user a tutorial on Mavo, interspersed with practice tasks on a simple inventory application. This took 45 minutes on average and covered the `property` attribute (10 minutes), the `data-multiple` attribute (10 minutes), and expressions using the `[]` syntax, broken down into how to reference properties and perform computations (5 minutes), aggregates such as `count()` (10 minutes), and `iff()` syntax and logic (10 minutes).

7.1.5 The Structured Study

49.6 %

For the Structured study, we created two applications:

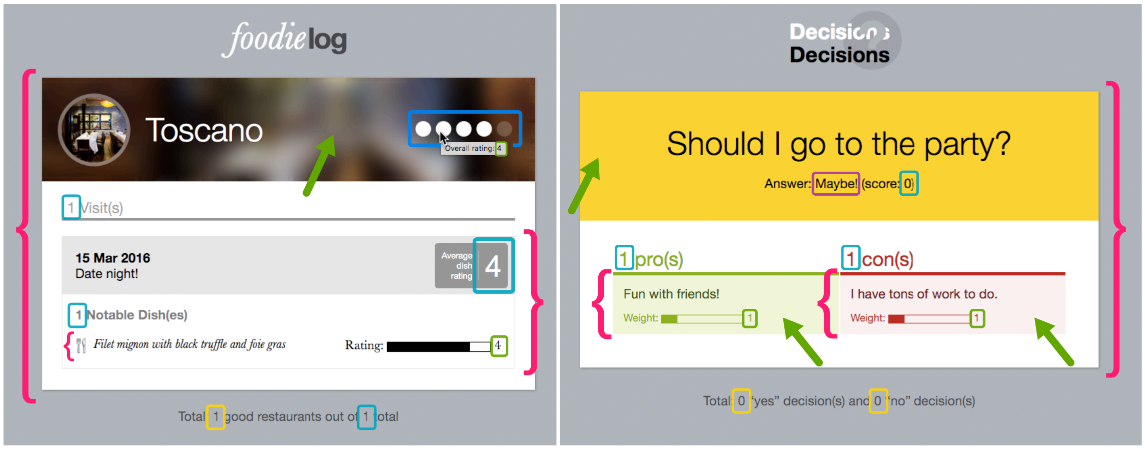
- **Decisions app**: A tool for making decisions by summing weighted pros and cons. The application also shows a suggested decision based on the sums of pro and con weights.
- **Foodie log**: A restaurant visit tracker that includes dishes eaten on each visit with individual ratings per dish. The application also computes average ratings for each visit and each restaurant.

17 subjects were given static HTML and CSS mockups of one of these applications and were asked to carry out a series of tasks by editing the HTML. These tasks tested their ability to use different aspects of Mavo, as shown in Figure 7.1. Eight of these users were given a mockup of a **Decisions app** and the other 9 were given a mockup of a **Foodie log**.

Each subject was shown a fully functional version of their respective application (but not its HTML source) before being given the static HTML template. While a CSS style file was provided, they did not have to look at it.

We provided tasks to the user one at a time, letting them complete one before revealing the next. Tasks were administered in the same order, and we measured the time each subject took to complete the task as well as screen recorded their typing.

Participants were asked to speak aloud their thoughts and confusions as they worked. Researchers were silent except to alert subjects to spelling mistakes and to explain HTML and CSS concepts—such as how to set a value on a `<meter>` tag — if subjects were unaware of them. If subjects spent over 15 minutes on a task but were not close to


foodie log
Toscano
Overall rating: 4
1 Visit(s)
15 Mar 2016
Date night!
Average dish rating
4
1 Notable Dish(es)
Filet mignon with black truffle and foie gras
Rating: 4
Total 1 good restaurants out of 1 total
Decisions
Should I go to the party?
Answer: Maybe! (score: 0)
1 pro(s)
Fun with friends!
Weight: 1
1 con(s)
I have tons of work to do.
Weight: 1
Total 0 "yes" decision(s) and 0 "no" decision(s)

	Task category	Example task	Example code	Med. time	Success
	Make editable Foodie: 1, Decisions: 1	*"Make the restaurant information editable (name, picture, url, etc)"*	`<h1 property="name">Toscano</h1>`	3:00	**100%**
🟥	Allow multiple Foodie: 3, Decisions: 2	*"Make it possible to add more pros and cons."*	`<article property="pro" data-multiple>`	1:15	**100%**
🟩	Simple reference Foodie: 3, Decisions: 3	*"Make the header background dynamic (same image as the restaurant picture)"*	`<header style="background: url([pic])">`	0:43	88%
🟦	Simple aggregate Foodie: 3, Decisions: 2	*"Make the visit rating dynamic (average of dish ratings)"*	`[average(dishRating)]`	0:55	**97.5%**
🟦	Multi-block aggregate Foodie: 1, Decisions: 0	*"Make the restaurant rating dynamic (average of visit ratings)"*	`<meter value="[average(visitRating)]">`	2:00	77.8%
🟨	Filtered aggregate Foodie: 1, Decisions: 1	*"Show a count of good restaurants"*	`[count(rating > 3)] good restaurants`	6:10	70.9%
🟪	Conditional Foodie: 0, Decisions: 1	*"Show "Yes" if the score is positive, "No" if it's negative, "Maybe" if it's 0."*	`[if(score>0, Yes, if(score < 0, No, Maybe))]`	5:28	75%

Figure 7.1 *User study tasks are shown in the mockups that were given to participants, and results are broken down by task category. The green arrows point to element backgrounds, which participants made dynamic via inline styles or class names. Page elements involved in specific tasks are outlined with color codes shown in the table. "Make editable" tasks are not shown to prevent clutter.*

succeeding, the researchers stepped in to offer hints or explain the answer, and marked the task as failed.

50.3 %

Study Tasks

In the case of the Decisions app, users had 10 tasks to complete, while for the Foodie log, users had 12 tasks. The tasks increased in difficulty in order to challenge the users. We grouped the tasks into 7 categories, where each category tests a particular aspect of Mavo.

Example tasks, code solutions, and the number of tasks in each category per application is in Figure 7.1. As footnoted earlier, all this task data is available online.

A description of each task type and what it entails follows:

- **Make editable** Adding `property` attributes to different HTML tags to make them editable.
- **Allow multiple** Turn an element into a collection, by adding `property` and `data-multiple`.
- **Simple reference** Display the value of a property somewhere else, via a `[propertyName]` expression.
- **Simple aggregate** Show the result of a simple aggregate calculation, such as the count or sum of something.
- **Multi-block aggregate** Aggregate calculation on a dynamic property, such as an average of counts.
- **Filtered aggregate** Show how many items satisfy a given condition.
- **Conditional** Show different text depending on a condition.

50.4 %

Results

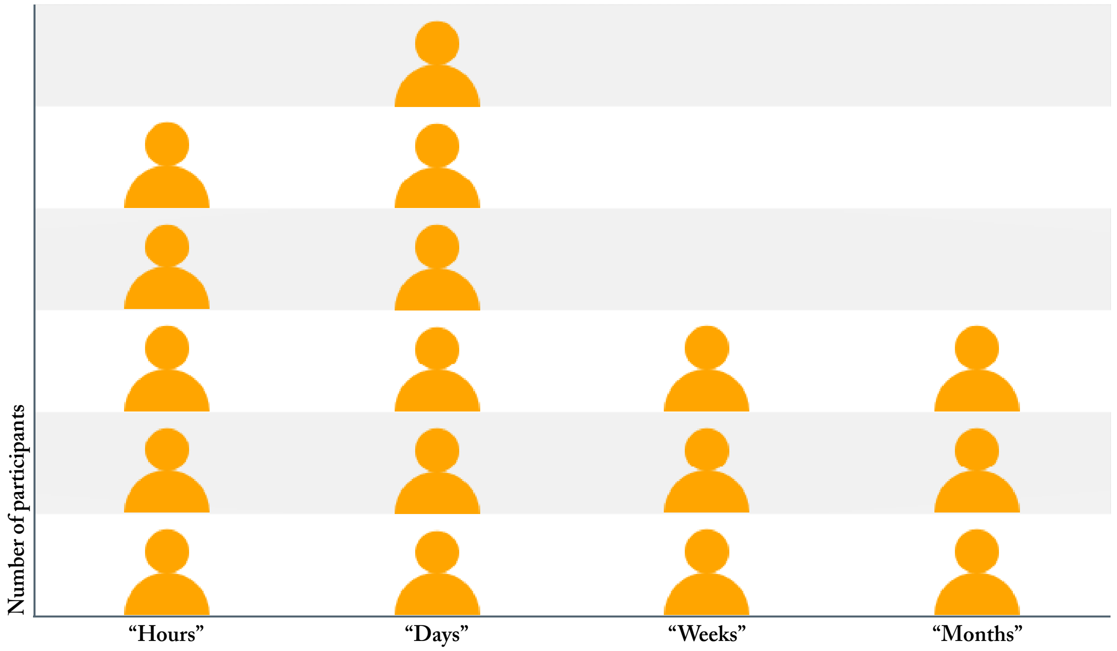


Figure 7.2 *Participant responses to the question “How long do you think it would take you to build this application?” before learning about Mavo.*

In the Structured studies, *before* providing the tasks, we showed users the finished application they were tasked to create and asked them how long they thought it would take them. Of the 17 users, 5 estimated it would take them several hours, 6 estimated days, 3 estimated weeks, and 3 estimated months. Some users said that they would need to learn new skills or that they had no idea where to start.

After going through the tutorial, 6 users went on to complete all the tasks for their application with no failures, 1 user had no failures but had to leave before the last task, and 10 users failed at one or more tasks. The 6 users who completed all tasks successfully took on average 17.3 minutes (Decisions, 10 tasks) and 22.5 minutes (Foodie, 12 tasks) to build the entire application.

Of the 10 people who failed one or more times, 5 failed on 1 task, 2 failed on 2 tasks, and 3 failed on 3 tasks. All failures were concentrated on expression tasks, usually the most difficult ones. The success rate for basic CRUD functionality was **100%**.

Figure 7.1 shows the median time taken and success rate for each category of task for all 17 users. As can be seen, some task categories were easier for participants to carry out than others. For instance, all participants quickly learned where to place the `property` and `data-multiple` attributes, taking a median of 3 minutes to make several elements editable via `property` and a little over a minute to turn single elements into collections.

Almost all participants were also able to display simple aggregates, such as showing a count of restaurant visits or a decision score (sum of pro weights - sum of con weights). However, some participants struggled with more complicated expressions, such as conditionals or multi-block aggregates. We explore some of the more common issues next.

We asked these 17 participants who built either the Decisions or Foodie app to rate the difficulty of converting the static page to the fully realized application. They were asked to rate this twice: once after seeing a demo of the final application but before learning about Mavo, and once after going through all the tasks with Mavo. On a 5-point Likert scale, the reported difficulty rating after building the app with Mavo dropped 2.06 points on average from its pre-Mavo rating.

50.8 %

Common Mistakes

We observed several recurring patterns in the errors made by participants.

Misplaced `data-multiple` Attributes

The most prevalent error was putting `data-multiple` on the wrong element — usually the parent container — with 40% of participants stumbling on it at some point. However, as soon as users saw that they were getting copies of the wrong element, they immediately figured out the issue. As the user's *intent* was always clear, a WYSIWYG editor would solve this in the future. Another similarly common and quick-to-fix mistake was forgetting `data-multiple` (25%). None of these mistakes led to task failures.

Concatenation

We noticed that users had a hard time grasping or realizing they could do concatenation. Both the Decisions and Foodie applications included 3 simple reference tasks. We noticed that the failure rate was significantly higher (20-25% vs 0%) **when the variable part was not separated by whitespace from the static part of the text**, as shown in Table 7.3. Perhaps the whitespace allowed users to see the variable part as a separate entity, and avoid building a mental model that involves concatenation.

HTML fragment	Success
`</meter> [rating]`	100%
`title="Overall rating: [rating]"`	100%
`</meter> [weight]`	100%
`style="background: url([pic])"`	77.8%
`class="weight-[weight]"`	75%
`class="answer-[answer]"`	75%

Table 7.3 *Success rate of simple references.*

Using `sum()` Instead of `count()`

Another common mistake was using `sum()` instead of `count()` (20% of participants). This may be because they are thinking of counting in terms of "summing how many items there are", Another theory might suggest that they are more familiar with `sum()`, due it being far more common than `count()` in spreadsheets. However, this is unlikely as there was no correlation between spreadsheet familiarity and occurrence of this mistake.

Abstraction and Intermediate Variables

We noticed that some participants frequently copied and pasted expressions when they needed the same calculation in different places. A DRY (Don't Repeat Yourself) strategy familiar to programmers would be to create an intermediate variable by surrounding the expression in one place with a tag (such as `<span>` or `<meta>`) that also has a `property`, so that it can be referenced elsewhere. These intermediate properties would reduce clutter and consequently reduce future mistakes down the road; they would also make it easier to modify computations globally. This idea might however be counterintuitive in Mavo as it calls for creating a tag in the HTML that is never intended to be part of the presentation, conflicting with the idea that one authors the application by authoring what they want to see.

Conditionals

The Structured tasks with the lowest success rate (70.9%) were those that required counting with a filter (`count(rating > 3)`). 25% of participants tried solving these with conditionals, usually of the form `iff(rating > 3, count(rating))`, which just printed out the number of ratings, since the condition is true if there is at least one rating larger than 3. Most who succeeded remembered or (more often) guessed that they could put a conditional inside `count` and seemed almost surprised when it worked. Another way of completing this task would be to declare intermediate hidden variables computing e.g. `rating > 3` inside each restaurant or decision and then sum or count them outside that scope. Only 10% of participants tried this method, again suggesting that intermediate variables are a foreign concept to this population.

Most participants found `iff()` to be one of the hardest concepts to grasp. 40% of subjects tried `iff()` when it was not needed, for instance in *simple* reference tasks. 25% of users were unable to successfully complete the conditional task, which required two nested `iff()`s or three adjacent `iff()` statements, each controlling the appearance of one of the designated words ("Yes", "Maybe", or "No"). The latter strategy was only attempted by 37.5% of participants.

In post-study discussions, some users mentioned how conditionals reminded them of what they found hard about programming:

> *“That’s some math and logic which are not my strong points. Just seeing those if statements…I did a little bit of Java and I remember those always screwed me over in that class. No surprise that that also tripped me up here.”*

Another user reflected on how having multiple ways of doing something made it more difficult:

> *“It’s hard because there are often multiple ways of doing something. And knowing which one would be the most efficient and best way to do it without making a mistake in the process was hard for me."*

7.1.6 Freestyle Study

51.4 %

Our second Freestyle user study involved a third **Own Address Book** application. During recruitment, subjects were asked to create *their own* static mock-up of an address book on their own time prior to meeting us, without being told why. The 7 subjects who complied were assigned to the Freestyle study (3 also did the Structured study first). During our meeting (and after the tutorial), they were asked to add Mavo markup to their own mockup to turn it into a working application.

We added this second study to address several questions. First, we wanted to be sure that our own HTML was not “optimized” for Mavo. Because users were not aware of Mavo at the time they created their application, their decisions were not influenced by perceived strengths and limitations of the Mavo approach. We can therefore posit that these mockups reflected their preferred concept of a contact manager application. Thus, this study served to test whether Mavo is suitable for animating applications that users actually wanted to create. At the same time, it tested whether users could effectively use Mavo to animate “normal” HTML that was written without Mavo in mind.

Study Tasks

51.6 %

Before this Freestyle study, we provided no specification of how the application should work or look, except to say that users only needed to use HTML and CSS; that if there were lists, they only needed to provide one example in the list; and that the mockup needed to contain at least a name, a picture, and a phone number. Then, during the study session, we asked them to use Mavo to make their mockup fully functional in any way they chose. If the application they envisioned was very simple, after they successfully implemented their application, we encouraged them to consider more complex features, as described in the a section below.

Since what the user worked on depended on their own envisioned implementation, we did not have explicitly defined tasks throughout. However, we did encourage users to try more advanced Mavo capabilities by suggesting the following tasks if they ran out of ideas:

1. Allow phone numbers (or emails) to have a label, such as “Home” or “Work” [Make editable]
2. Allow multiple phone numbers (and/or emails, postal addresses) [Allow multiple]
3. Provide a picture alt text that depends on the person’s name (for example, “John Doe’s picture”) [Simple reference]
4. Show a total count of people (and/or phone numbers, emails) [Simple aggregate]
5. Show “person” vs “people” in the heading, depending on how many contacts there are. [Conditional]

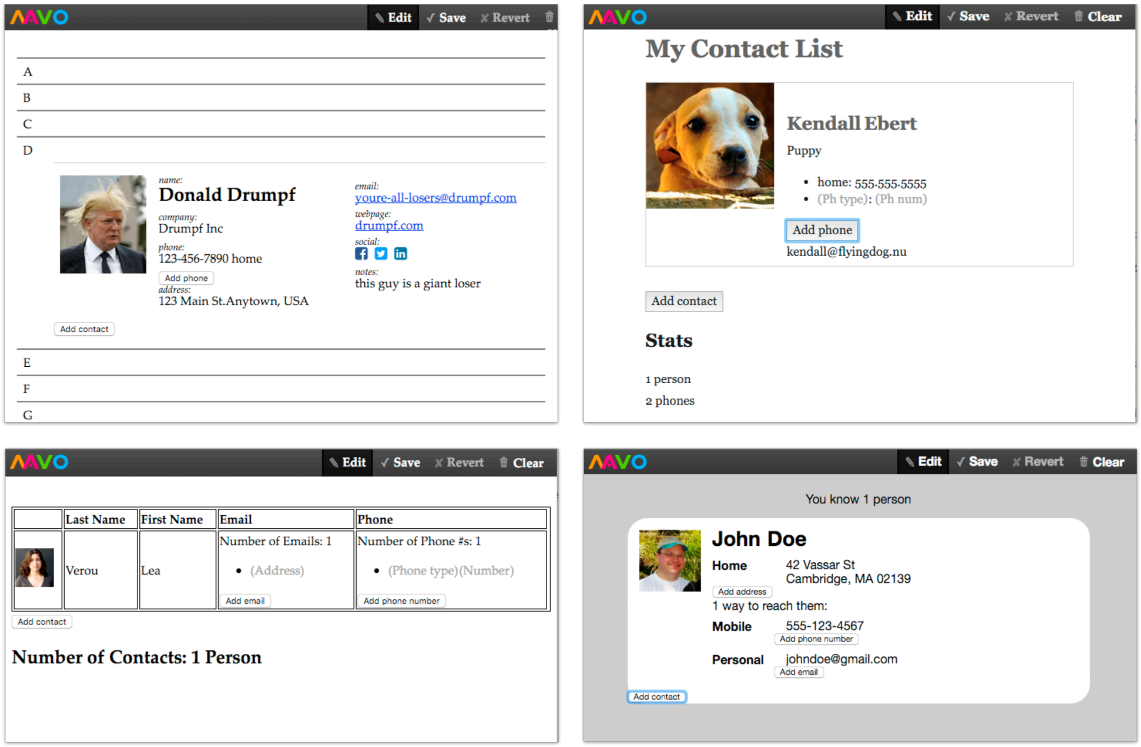
51.7 %

Results of Open-Ended Tasks

Of our participants, 7 brought in their own static mockup of an Address Book app and had time for the Freestyle study. We found a variety of implementations of the repeatable contact information portion. One person used a `<table>`, with each row representing a different contact. Three people used `<ul>`, with each contact as a separate list item, and the information about each contact represented inline or as separate `<div>` elements. Two people chose to only use nested `<div>`s, with each contact having their own `<div>`. Finally, one person chose to create a series of 26 `<div>`s, each one a letter of the alphabet, with the intended ability to add contacts within each letter.

When we asked users to use Mavo to improve their mockup in any way, all 7 users chose initially to use the Mavo syntax to make the fields of the app editable and to support

multiple contacts, and had no trouble doing so. 4 out of 7 chose, of their own accord, to support multiple phone numbers, emails, or addresses per contact. In all but one case, Mavo was able to accommodate what users envisioned, as well as our extra tasks. In one case (top left in Figure 7.3), the participant wanted grouping and sorting functionality, which Mavo does not support. She was still able to convert her HTML to a web application, but the user had to manually place each contact in the correct one of 26 distinct “first letter” collections. A sample of Own Address Book applications that users created are shown in Figure 7.3.



A
B
C
D

name:
Donald Drumpf
company:
Drumpf Inc
phone:
123-456-7890 home
address:
123 Main St.Anytown, USA
email:
youre-all-losers@drumpf.com
webpage:
drumpf.com
social:
notes:
this guy is a giant loser

E
F
G

My Contact List

Kendall Ebert
Puppy
- home: 555-555-5555
- (Ph type): (Ph num)

kendall@flyingdog.nu

Stats

1 person
2 phones

	Last Name	First Name	Email	Phone
	Verou	Lea	Number of Emails: 1 • (Address)	Number of Phone #s: 1 • (Phone type)(Number)

Number of Contacts: 1 Person

You know 1 person

John Doe
Home 42 Vassar St
Cambridge, MA 02139
1 way to reach them:
Mobile 555-123-4567
Personal johndoe@gmail.com

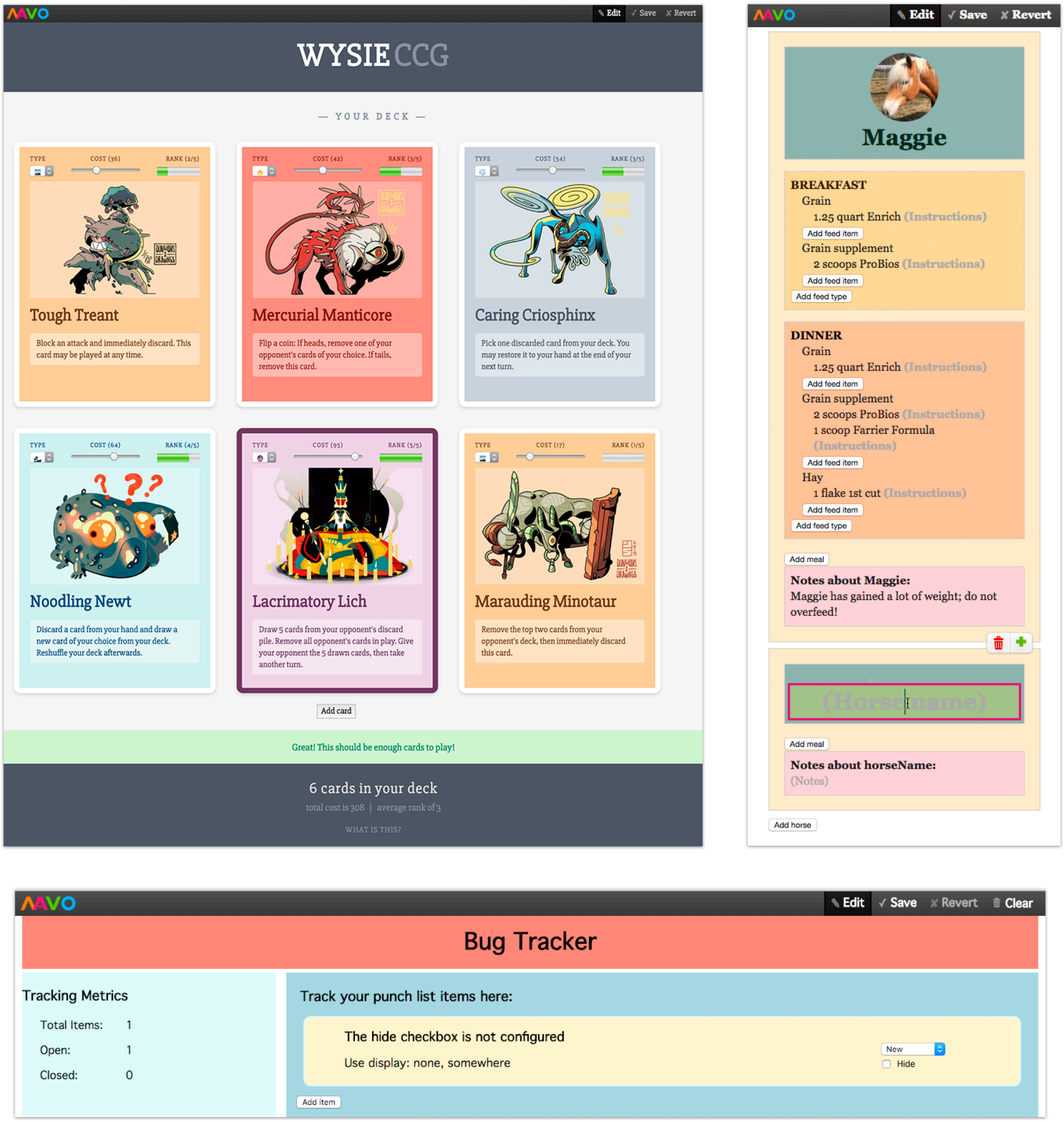
Figure 7.3 *A sample of Own Address Book applications created by users.*

Five more participants brought Contact Manager mockups, but did not have time to animate them due to participating in the Structured study first. However, all five mockups were suitable for Mavo and followed the same patterns already observed in the Freestyle study.

7.1.7 Aftermath

52 %

To further investigate its appeal, we encouraged participants to try out Mavo on their own time after the user study. Three of them went on to create Mavo apps for their own needs: (a) a collectible card game, (b) a bug tracker, and perhaps the most interesting of all, (c) a horse feed management application (Figure 7.4). The authors of the first two applications were programming novices, the latter intermediate.



WYSIE CCG

— YOUR DECK —

TYPE COST (36) RANK (2/5)
Tough Treant
Block an attack and immediately discard. This card may be played at any time.

TYPE COST (42) RANK (3/5)
Mercurial Manticore
Flip a coin: If heads, remove one of your opponent's cards of your choice. If tails, remove this card.

TYPE COST (54) RANK (3/5)
Caring Criosphinx
Pick one discarded card from your deck. You may restore it to your hand at the end of your next turn.

TYPE COST (64) RANK (4/5)
Noodling Newt
Discard a card from your hand and draw a new card of your choice from your deck. Reshuffle your deck afterwards.

TYPE COST (95) RANK (5/5)
Lacrimatory Lich
Draw 5 cards from your opponent's discard pile. Remove all opponent's cards in play. Give your opponent the 5 drawn cards, then take another turn.

TYPE COST (17) RANK (1/5)
Marauding Minotaur
Remove the top two cards from your opponent's deck, then immediately discard this card.

Great! This should be enough cards to play!

6 cards in your deck
total cost is 308 | average rank of 3
WHAT IS THIS?

Maggie

BREAKFAST
Grain
1.25 quart Enrich (Instructions)
Grain supplement
2 scoops ProBios (Instructions)

DINNER
Grain
1.25 quart Enrich (Instructions)
Grain supplement
2 scoops ProBios (Instructions)
1 scoop Farrier Formula (Instructions)
Hay
1 flake 1st cut (Instructions)

Notes about Maggie:
Maggie has gained a lot of weight; do not overfeed!

(Horsename)

Notes about horseName:
(Notes)

Bug Tracker

Tracking Metrics

Total Items:	1
Open:	1
Closed:	0

Track your punch list items here:

The hide checkbox is not configured
Use display: none, somewhere
New
□ Hide

Figure 7.4 *Mavo apps independently created by participants. Clockwise: Collectible Card Game, Horse feed management, bug tracker.*

7.1.8 General Observations

We conclude this section with some general observations applicable to both studies.

Spreadsheet Familiarity and Mavo

Approximately half (9/20) of our participants did not use spreadsheets frequently ("rarely" or "hardly ever"), while the rest used them frequently or daily. And while all users had used spreadsheets and spreadsheet formulas before, most (12/20) had never used the `VLOOKUP()` function necessary to do joins in spreadsheets.

While it is a plausible hypothesis that familiarity with spreadsheets would make Mavo or Formula² easier to learn, we did not observe any difference in outcomes between those familiar with spreadsheets and those not.

Debugging Behavior

Some participants used the Inline debugging tools provided to them while others ignored it, instead choosing to look at the visual presentation of the HTML to see where they went wrong. One user even commented out loud that they were not going to look at the debug table at all, then proceeded to fail on a task where a quick glance would have likely prevented this.

A possible explanation is that novices are not used to looking in a separate place for debugging information. The debug tables were visually and spatially disconnected from the rest of their interface, especially on (visually) larger objects, which to some extent violates the direct manipulation principles Mavo was based on.

When they were trying to solve an issue, they were looking at the part of the interface that was supposed to display the result, not elsewhere. Another possible explanation is that the information density of the table is intimidating to novices.

The users who did look at the debug tables found them useful for spotting spelling mistakes, missing closing braces or quotes, use of wrong property names, and for understanding whether properties were lists, strings, or numbers. Nobody experimented with editing expressions in the debug table, and few participants (15%) used the in-browser development tools such as the console and element inspector.

52.3 %

Overall Reactions

The overall reactions to Mavo ranged from positive to enthusiastic. One user who was a programming beginner but used CMSs on a daily basis, said

> *"Being able to do that…right in the HTML and not have to fool with…a whole other JavaScript file…That is fantastic. I can't say how awesome that is. I'm like, I want this thing now. Can I have a copy please? Please send me an email once it's out."*

Along similar lines, another non-programmer said *"When is this going to be available? This is terrific. This is exactly the stuff I have a hard time with"*.

Many participants liked the process of editing the HTML as opposed to editing in a separate file and/or in a separate language:

> *"It seems much more straightforward, everything is right there. You're not referring to some other file somewhere else and have to figure out what connects with what. It's…almost too easy"*.

Others liked how the Mavo syntax was reminiscent of HTML:

> *"It didn't seem like a lot of new things had to be learned because naming properties was just the same as giving classes and ids."**

> *"It's very simple. It's as logical as HTML. You are eliminating one huge step in coding, the need to call the answer at some point, which is really cool… Everything is where it needs to be, not in a different place"**.

Other users praised the ability to edit the data from within the browser as opposed to a separate file or data system. One person said,

> *"I'm convinced it's magic to basically write templating logic and have it show up and be editable. I think there's a lot less cognitive overhead to direct manipulation on the page, especially for a non-technical user"*.

This unprompted recognition of direct manipulation supports our argument that this approach is natural.

Though several users struggled with some of the more complicated tasks around expressions, *all* participants easily got the hang of defining a hierarchical data schema within HTML using Mavo. Several users felt that the Mavo attributes of `property` and `data-multiple` were powerful even without expressions, and mentioned wanting to use these attributes to replicate functionalities of CMSs that they used.

When asked what applications they could see Mavo being useful for, users mentioned using Mavo to build a color palette app, a movies-watched log, a basic blog, and an app for tipping. Two users mentioned using Mavo for putting out surveys and contact forms. Several mentioned using Mavo to build an online portfolio, with lists of projects.

Reaction to Formula^2

Many participants were enthused about Formula^2 expressions, even those who had failed at a few tasks. One participant said about them:

> *"It's simpler than I expected it to be. My anxiety expects it to be hard, then I just say 'write what you think' and it turns out to be right. It's very intuitive."*

Another user, after learning about filtered aggregates (e.g. `count(age > 5)`) said

"It's so expressive, it tells you exactly what it's doing!".

7.1.9 Discussion & Future Work

52.7 %

Declarative Conditionals

Users struggled with conditionals (`if()`), and their struggle multiplied when they were nested. Part of this was , partly due to syntax — balancing parentheses and commas is hard for novices [21, 113].

There are two ways to address this, not necessarily mutually exclusive: (a) in Formula^2, by reducing the need to balance parentheses (b) in Mavo HTML, by implementing a declarative, HTML-based syntax for conditional logic.

For (a), Formula^2 could adopt a ternary operator such as `if test then value1 else value2` or `value if test else value2` which is arguably more readable than the functional syntax for everyone.

For (b), Mavo HTML did adopt a declarative syntax for conditionals, by adding an `mv-if` attribute whose value is always interpreted as an expression. in addition to the functional syntax.

52.8 %

Generalizing Numerical Aggregates

`count()` is the only aggregate function that is meaningful for any data type. All others (`sum()`, `average()`, `median()`, `min()`, `max()`) are only meaningful for numbers, booleans (treated as 0 or 1), or strings containing numbers. Using them with lists of objects does not produce an error, since non-numbers are simply ignored, but it also does not produce a meaningful result.

In tasks where participants had to sum properties of objects in a list (e.g. summing the weights of pros and cons in the Decisions app), some tried using `sum()` on the list of items, rather than the list of numbers (e.g. `sum(pro)` instead of `sum(pro.weight)`). Others used `sum()` instead of `count()` to count the number of items in a list. In both cases, all values passed were ignored and the result was `0`.

It is an open question how numerical aggregates could be generalized in a way that produces a more meaningful result when used with objects. One way would be to look in the object's properties and operate on all of those that are numbers. This would enable our participant attempts to write things like `sum(pro)` to work as expected. While a useful behavior in its own right, when author intent is to actually sum multiple object properties, when it is the result of a slip, it could be quite fragile: you add a property to your schema and suddenly your sum changes!

Another option is to treat objects and other non-numbers as `1` in order to have `sum` generalize `count`. This would eliminate the second type of mistake, but it could be surprising as a general behavior, whereas descending to object numerical properties seems more inline with Formula^2 aggregation semantics.

53 %

Filtering and Sorting

While participants were enthusiastic about the potential of building apps with Mavo, there were also a few requested use cases that Mavo cannot presently accommodate. Sorting, searching and filtering were recurring themes. Simple filtering and searching is already possible via expressions and CSS, but not in a straightforward way. We plan to explore more direct ways to declaratively express these operations. Since Mavo makes collections and properties explicit, it doesn't take much more syntax to enable sorting and filtering of a collection on certain properties; however, the more complex question is to develop a sufficiently simple language that can empower users to fully customize any generated sorting and filtering interfaces beyond simple skinning.

One user wanted to filter a list based on web service data (current temperature). Mavo can already incorporate data from any JSON data source, so this will become possible once we support combining data from multiple Mavo instances on the same page.

After learning about conditional counting (e.g. `count(score > 0)`, one participant inquired about more complex queries, such as counting a different property than the one filtered. The syntax we are considering for this is optional extra filtering arguments on all aggregate functions. This would enable syntax like `count(gender == female, age > 40, height > 160)`.

53.2 %

7.2 Second Lab Study - Formula^2 & Data Update Actions

Following the introduction of data update actions, together with several Formula^2 improvements, we conducted a second lab study focused on the usability of these additions.

While data update actions were the focus of this study, it also serves as another evaluation for Formula^2 and Mavo HTML, especially around Formula^2's scoping and referencing, and its data specification and filtering mechanisms, which were new additions.

7.2.1 Goals

To design a data update language that feels natural to novice programmers, we took a two-pronged approach. First, we attempted an *unconstrained* elicitiation [114] of a syntax that users find natural. Second, we used our prototype language in a *constrained* elicitation, as we expected different insights from unconstrained responses compared to a prototype.

Several studies [67, 69] have investigated and tried to understand what is natural for novices, by examining the ways that non-programmers express solutions to common programming tasks. We decided to follow a similar, albeit slightly modified approach.

First, we authored static web page of a simple Mavo application, and we asked our participants to create their own syntax (what feels natural to them) to answer a series of Mavo data mutation tasks. This study focused on understanding the mental models that novices build about the notional machine. Second, we went over Mavo's data actions documentation with the participants, then asked them to write the syntax for the same series of tasks we have asked them in the first part of the study, except now they know the syntax of Mavo's data actions. Third, we authored static web page mockups of two representative CRUD applications and then gave users a couple of Mavo authoring tasks that gradually evolved those mockups into complete applications. Finally, freestyle study, before telling users about Mavo's data actions (so that they would not feel constrained by the capabilities of our new data mutation syntax), we asked them to create their own mockup of a shopping list application. In the last three sections, the study focused on the usability and learnability of the data actions syntax.

Is our syntax intuitive and can it be learned in a short amount of time?

1. What syntax feels most natural to novice web authors for expressing a variety of data mutations on nested data structures?
2. Is our syntax intuitive and can it be learned in a short amount of time?

7.2.2 | Preparation

	HTML	CSS	JavaScript	JSON
Not at all	0	0	6	4
Beginner	1	1	5	3
Intermediate	5	3	9	8
Advanced	12	11	0	5
Expert	2	5	0	

Table 7.4 *Participants' familiarity with web technologies.*

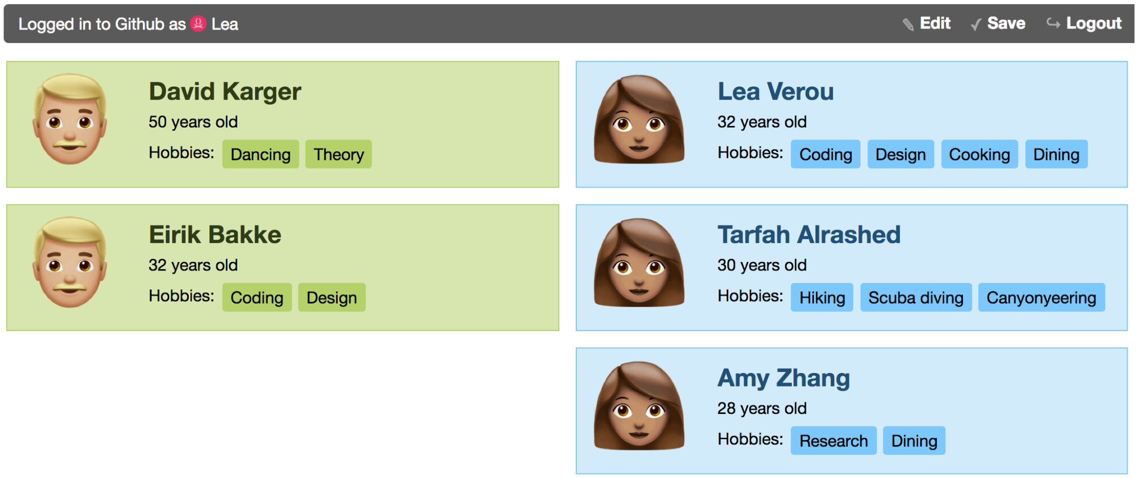
We recruited 20 participants (age μ=36.2, σ=9.25; 60% female, 40% male) by publishing a call to participation on social media and local web design meetup groups. Their (self-reported) skill levels in HTML and CSS ranged from beginner to expert, but intermediate or below in JavaScript. 11/20 described themselves as beginners or worse in *any* programming language, while 9/20 were intermediate.

Regarding data concepts, 5/20 stated they could write JSON, 4/20 could write SQL, and none could write HTML metadata (RDFa, Microdata, Microformats). We asked our participants to read through the Mavo Primer[2] and optionally to create a shopping list application with Mavo before coming in for the study.

[2] mavo.io/docs/primer

7.2.3 Study Design

Sessions were conducted one-on-one, in person and were limited to 90 minutes. Participants were shown a Mavo application with two collections (men and women) each containing a name, an age and a collection of hobbies (Figure 7.5). We decided on this schema because it is nested, and the properties have an obvious natural meaning.



David Karger
50 years old
Hobbies: Dancing Theory

Eirik Bakke
32 years old
Hobbies: Coding Design

Lea Verou
32 years old
Hobbies: Coding Design Cooking Dining

Tarfah Alrashed
30 years old
Hobbies: Hiking Scuba diving Canyonyeering

Amy Zhang
28 years old
Hobbies: Research Dining

Figure 7.5 *The people application, used for a variety of tasks*

We used the Mavo Inspector (Figure 3.6) to demonstrate the application in general and in particular the difference between referring to property values within the scope of a collection item versus outside the collection (See Section 4.4.1)

First, participants were asked to write expressions that compute counts for five questions of increasing difficulty, starting from the simplest ("Count all men") down to filtered counts (e.g. "count women older than 30", "count women who have 'Coding' as a hobby", etc), which participants found problematic in the first Mavo study. Participants were discouraged from iterating on their expressions, and were told we wanted to capture their initial thinking.

The purpose of this part of the study was three-fold: (a) to assess their understanding of existing Mavo capabilities, (b) to verify whether filtered counts were indeed harder, and (c) to prime them into thinking in terms of declarative functional expressions for the study that was yet to follow.

Not all five questions could be answered entirely with information from the Primer. For example, the primer did not include the dot notation for narrowing down references (e.g.

#	Question	Type	▼
1	Delete all men	delete	
2	Add new man (with no info filled in)	add	
3	Delete all people	delete	
4	Add a new man and a new woman	add	
5	Delete current man	delete	
6	Make current man 1 year older	set	
7	Make everyone 1 year older	set	
8	Set everyone's name to their age	set	
9	Delete women older than 30 years old	delete	✓
10	Move the current woman to the collection of men	move	
11	Add a woman with the name "Mary" and age of 30	add	
12	Add a woman with the name "Mary" and age of 30 to the beginning of the women collection	add	
13	Delete "Dining" as a hobby from everyone	delete	✓
14	Rename every man with age > 40 to "Chris"	set	✓
15	Move the current woman to the beginning	move	
16	Change the age of the woman named "Mary" to 50	set	✓
17	Move all men to the collection of women	move	

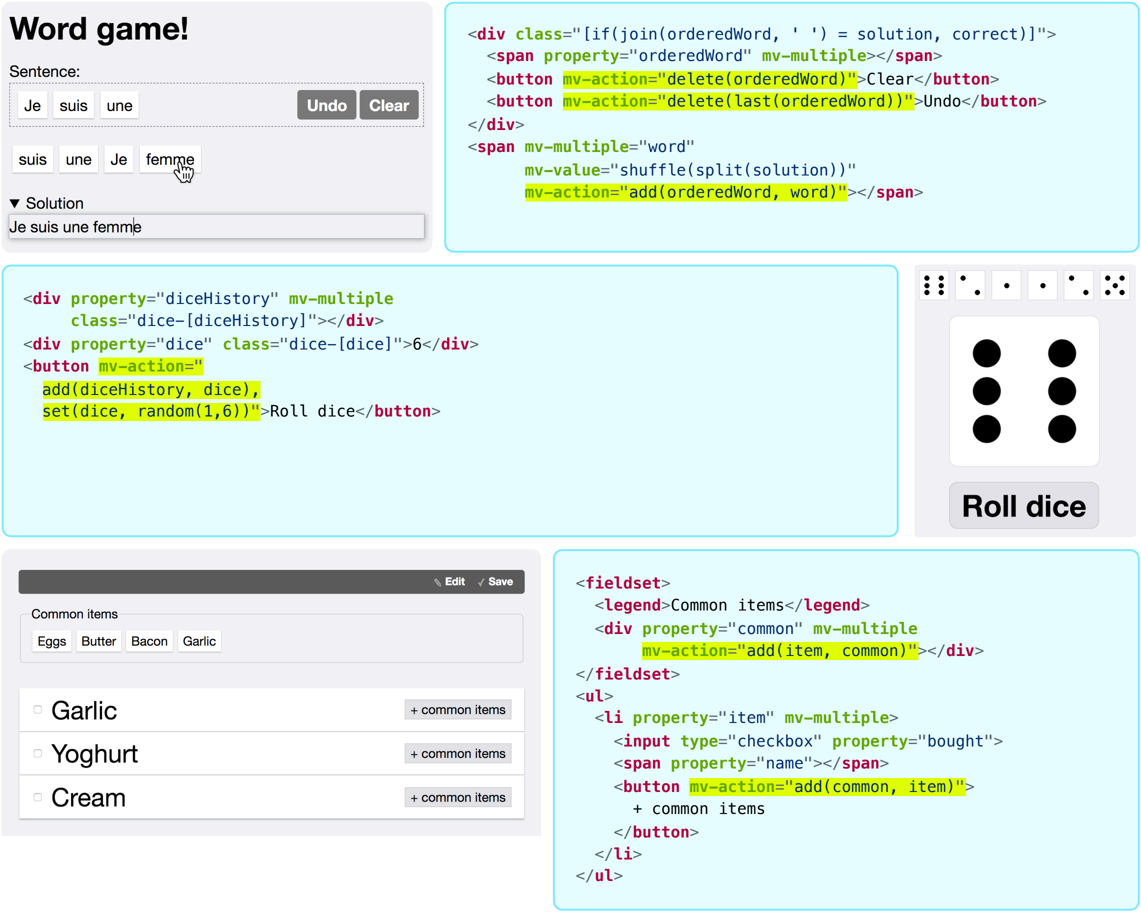
Table 7.5 *All 17 data manipulation questions. The third column indicates whether filtering was needed to answer the question.*

`woman.age` to get ages of women instead of `age` which would return ages from both men and women).

The second part was a natural programming elicitation study [114]. We briefly explained the problem that Mavo data updates are solving, as well as our idea for addressing it on a high level. More specifically, we mentioned the `mv-action` attribute, as well as the `set()`, `delete()`, `add()`, and `move()` functions, but presented this as ideas whose syntax we are not sure about and had not developed yet. We then asked participants to answer 17 data update questions of increasing complexity (Table 7.5) by writing the syntax that felt more natural to them. They were also encouraged to even use different function names, if that felt more natural to them.

After this stage, we revealed our language prototype so that they could experiment with it during the study. After a brief tutorial (5-10 minutes), participants had to answer the same questions, in the same order, using our syntax. After this section, participants were asked to choose 4 questions, one from each action type (set, add, move, delete) and try them out as a training task for the next part. Researchers would alert them to any mistakes and help correct them.

The final part of the study consisted of two sets of hands-on tasks where participants would try authoring data updates to complete the functionality of two different applications using our syntax prototype. For the first set, participants were randomly assigned one of two applications: a Dice Roller application with a history of past dice rolls, and a language learning Word Game where users click on words in the right order to match a hidden sentence, both having three tasks. The second set was the same for all users and extended a shopping list application, either one they made, or our template. For all hands-on tasks (Figure 7.6), participants were given the HTML, CSS and (original) Mavo markup, and only had to add `mv-action` attributes to complete their functionality.



Je suis une femme

```
<div class="[if(join(orderedWord, ' ') = solution, correct)]">
  <span property="orderedWord" mv-multiple></span>
  <button mv-action="delete(orderedWord)">Clear</button>
  <button mv-action="delete(last(orderedWord))">Undo</button>
</div>
<span mv-multiple="word"
      mv-value="shuffle(split(solution))"
      mv-action="add(orderedWord, word)"></span>
```

```
<div property="diceHistory" mv-multiple
     class="dice-[diceHistory]"></div>
<div property="dice" class="dice-[dice]">6</div>
<button mv-action="
  add(diceHistory, dice),
  set(dice, random(1,6))">Roll dice</button>
```

```
<fieldset>
  <legend>Common items</legend>
  <div property="common" mv-multiple
       mv-action="add(item, common)"></div>
</fieldset>
<ul>
  <li property="item" mv-multiple>
    <input type="checkbox" property="bought">
    <span property="name"></span>
    <button mv-action="add(common, item)">
      + common items
    </button>
  </li>
</ul>
```

Figure 7.6 *The hands on tasks with their solutions. From top to bottom: Words game, Dice Roller, our Shopping List (for participants who did not bring their own).*

After finishing all tasks, participants were asked a few questions about their experience in the form of a brief semi-structured interview, completed a SUS [115] questionnaire, and a few demographics and background questions.

55.9 %

7.2.4 Results & Discussion

Data Model and Referencing

While participants generally understood properties, groups, and collections, many participants were confused by the fact that no element in the HTML represented the actual collection, it was instead a data node that existed purely in the Mavo tree.

This partly motivated changing the collection specification syntax from the original `mv-multiple` to `mv-list`/`mv-list-item` shortly after the study, in addition to the many conceptual issues with the original design (discussed in Section 3.3.3.3.1).

56 %

Counting Questions

All participants correctly answered all Formula^2 counting questions, even when they had to count a deeply nested property, such as counting all hobbies from outside both collections of men and women. Also, they seemed to have no trouble with filtered aggregates like `count(age > 3)` with 17/20 getting them right, and the remaining three making only minor syntax errors.

Participants had some trouble disambiguating between nested properties with the same name across two collections (e.g. getting only women's ages or only men's hobbies). Like SQL, Mavo uses dot notation for this (`woman.age` only returns women's ages), which only 8/20 participants used. However, as there was no example of this in the Mavo Primer, we did not consider these failures a sign of poor understanding of Mavo functionality.

Freeform Syntax

In this part of the study, we wanted to explore what syntax participants found natural, with the only suggestion being that they had to use the four functions (set, add, move, delete). This suggestion was introduced to put participants in the mindset of writing expressions instead of purely natural language. They were even encouraged to use different function names if they wished to, and 6 did so at least once (half of them inconsistently).

Despite emphasizing that constraint, 6/20 participants did not use any functions in at least one question, but wrote statements instead (such as `age = age + 1`) and 4 more used a hybrid approach, with some parameters outside the function call, such as `add(woman) name=Mary age=30`.

The median time each participant spent answering each question was 28.5 seconds.

Scope

As mentioned in Chapter 4, in Formula^2 expressions, property names can be used anywhere and resolve to the local value or all values depending on the expression placement, enabling very concise references for common cases (see Section 4.4.1). Thus, `delete(man)` used "inside" a particular item in a collection of `man` objects would delete *only* that item, while `delete(man)` "outside" the collection would delete *all* those man objects.

However, in their own syntax, many subjects wanted to make this distinction explicit. 8/20 used a keyword or function to refer to all items (e.g. `man.all`) at least once, and 8/20 used an explicit keyword or function for the current item, such as `woman.current` or `this`. Only 3 participants did both. Interestingly, **none followed their own referencing schemes consistently**, using these explicit references only in some of the questions or some of the arguments, and plain property names elsewhere. This may indicate one reason why this referencing scheme is useful: it eliminates error conditions.

More work is needed to understand why our subjects attempted this more verbose language when the more concise one would work. Based on participant answers to probing questions, the survey format may have played a role: they were writing their answers in text fields, separately from the HTML, so the context of their expression was removed. In that setting, it may have been jarring to write the same expression as an answer to completely different questions (e.g. "Delete all men" and "Delete current man" are both `delete(man)` with our syntax). Perhaps if they'd been writing Mavo expressions inside actual HTML, the disambiguation through context would have eliminated the desire to disambiguate through syntax.

Another possibility is that novice programmers *prefer* verbosity. Pane et al. [69] showed that 32% of non-programmers constructed collections by using the keywords **every** or **all**. The use of such verbose syntax could be seen as a form of commenting, adding clarity over more concise code. It is easy to provide syntactic sugar to allow such explicit references. In fact, Mavo already defines special `all` and `this` variables that work in a similar way although we did not mention this.

We also observed the reverse, of users trying to be *more* concise. 7/20 participants indicated that the target of their action is the current item by **omitting** a parameter, such as writing `delete()` for deleting the current item or `move(man)` for moving the current woman to the collection of men.

Underspecified Expressions or Clever Heuristics?

Implicit Modification Target

Stylistic choices such as punctuation should be distinguished from expressions which must be regarded as incorrect because they are missing necessary information for the operation. However, even in those cases, it is hard to be certain that there is a logic error at play. Are the missing parameters actually missing, or did the participant have a clever heuristic in mind for inferring them? And if not, is it a logic error, or merely a slip? In this section, we describe some of the most common patterns of (ostensibly) underspecified expressions that we observed.

By far the most common one was `delete(<predicate>)` with no reference to the item(s) to be deleted. For example, `delete(woman.age > 30)` for deleting women older than 30, or `delete(hobby = Dining)` to remove the hobby "Dining" from all people. 18/20 participants did this at least once, and 10 did so in both of the conditional delete questions (Q9, Q13). One possible interpretation could be that in their mental model, specifying the target of the operation is only necessary for *disambiguation* — when the expression only includes one data reference, what else could we be targeting?

Another common pattern was using `set(age + 1)` to increment all ages. 15/20 participants used a variation of this syntax. This is consistent with the proposed interpretation above, that when there is only one data reference, they expect the update target to default to it.

As further evidence for that theory, there was no such underspecification in Q8 (*"Set everyone's name to their age"*), which involved two properties. None of the participants realized the inconsistency when they answered the latter and did not think to back and change their answer to the former. Asking a subset of participants about this at the end revealed that some thought that `age + 1` would function as an increment operator (like C's `age++`).

Both patterns indicate a distaste for parameter repetition, which on par with natural language: "parameters" are only explicitly specified when different and are otherwise implied.

Implicit Primary Keys

5/20 participants wrote their expressions as if the "name" property was special, i.e. was an implied primary key. For example they would use the identifier `Mary` to refer to the person that has ``name = "Mary"`, without specifying "name" anywhere in their expression. This did not seem to correlate with a lack of (self-reported) programming skill, as only one of them had not been exposed to programming at all. It is unclear whether this has to do with the word “name” itself, or with the fact that names were indeed unique in the data we gave them.

Implicit Value

Using objects as numbers was common, e.g. omitting “age” from `delete(woman > 30)` or `set(man + 1)`. Many participants attempted it at first, and 4/20 submitted their answers with it. In many cases this turned out to be a slip, but two participants articulated a consistent mental model: it automatically operates on all numeric properties! In Pane et al. [69], 61% of non-programmers modified the object itself instead of its properties, which is even higher than the percentage we observed.

56.9 %

Syntax

Argument Separation

While commas are likely the most widely used argument separator, they did not appear to be very natural to our subjects. 7/20 did not use any commas, but instead separated arguments by other symbols, or even whitespace. 5/20 only used commas for repetition of the same argument type (e.g. `delete(man, woman)`). From the remaining 8 subjects, only 2 used commas exclusively to separate arguments. The rest combined commas with separators that were more related to the task at hand.

16/20 subjects used `=` to separate arguments at least once, most commonly in `set()`. 9/20 used `to`, primarily in `set()` and `move()`. Other separators were used by 3 people or fewer (whitespace: 3/20, colons: 3/20, parentheses: 1/10).

Sequencing Function Calls

Only 8/20 participants used multiple function calls in an expression (such as `add(man) add(woman)`) The rest tried to express compound actions via arguments of one function call (such as `add(man, woman)`), even when this was inconsistent with their later responses.

In spreadsheets (like Formula^2 before data update actions), expressions have no side effects and only produce one output, therefore there is never a need for multiple adjacent function calls Therefore, using more than one function call may feel foreign and unnatural to these users.

Delimiters

In the prototype syntax, we had used different punctuation (period, comma, colon, semicolon, and spaces) to separate the key from its value, the object from its properties, different collection or items, and different functions. For example, for one of the questions that we asked our participants (*"add a new man and a new woman"*), the prototype syntax expected separate functions `add(man) add(woman)`, which could be separated by spaces, commas, or semicolons.

In another question where we asked our participants (Set everyone's name to their age), the answer should be `set(name, age)`, the two properties, the key and the value here, should be separated by comma, But 12/20 of our participants used `=` to separate the key and the value in the function `set()` ON the other hand, in the `move()` category questions (e.g. Move the current woman to the collection of men), 9/20 participants used the keyword `to`, so instead of writing `move(woman, man)` they wrote `move(woman to man)`.

Filtering

Four questions required filtering on a collection (cases where a corresponding SQL query would need a `WHERE` or `HAVING` clause) to specify the target of the data update. For example *"Delete women older than 30 years old"* (Q9), requires some way to filter the collection of women by age, then delete all matched items. Our prototype syntax supported this kind of filtering with a `where` operator, so `woman where age > 30` would produce a list of women whose age is over 30.

Half of our participants also defined a language-level filtering syntax, such as `if` or `where` keywords, or parentheses (e.g. `woman(age>30)`) whereas 6/20 expected that the data update functions would allow a filtering argument.

However, `if` appeared to be a slightly more natural keyword for our participants, with 5/20 using it at least once in these tasks, in contrast to 3/20 using `where`. 5/20 used filtering by predicate (e.g. `delete(woman.age > 30)`), 4/20 used filtering by parentheses, and 6/20 used filtering by argument.

5/20 expected that predicates would act as a filter of the closest collection item and consistently used them in that fashion For example, they expected that `man.age > 40` would return a list of men whose age was larger than 40, and wrote expressions like `set((man.age > 40).name to "Chris")` for Q16 However, in Mavo currently the inner expression returns a list of booleans corresponding to the comparison for each man.

Relationship to Prototype Syntax

Participant free-form syntax was consistent with our current prototype syntax (would have produced the correct result) with no changes in 4.35/17 answers on average (σ = 2.16) and with minor changes (different symbols or removing redundant tokens) in 8.6/17 answers on average (σ = 2.06).

Multiple Function Calls or Multiple Arguments?

Some questions were asking about multiple operations of the same type, to examine whether participants will use separate functions in the same action (e.g. `delete(man), delete(woman)`, or one function with separate multiple arguments (e.g. `delete(man, woman)`).

Our prototype syntax supports that kind of aggregation for `delete()`, because deleting an item does not require any other parameters. However, it cannot as straightforwardly be supported in `add()`, as it supports specifying other parameters for the new item. We were curious to see if our participants will be able to draw this kind of distinction by themselves. From our survey, we found that 9/20 participants used separate functions in general, and 7/9 who used separate functions used them in the case of adding a new man and a new woman However, 13/20 used one function (e.g. `add(man, woman)` It was also interesting to notice that only 3/20 used a separate functions for delete all man and women, which also works But 4/7 participants who used separate functions for adding a new man and a new woman `mv-action="add(man), add(woman)"`) did not use separate functions for delete all men and woman `mv-action="delete(man, woman)"`).

Which Position is first?

In the survey, we have asked our participants a couple of questions about moving an item to the beginning of its list (e.g. Move the current woman to the beginning of the women collection — `move(woman, 0)` in our prototype syntax). We wanted to understand how non-programmers would define “beginning”. Would they use a numerical index or a keyword? If a numerical index, would they use 0 or 1? If a keyword, would they use `start`, `top`, or something else?

8/20 used the number 0, 3/20 used number 1, 3/20 used the keyword `first`, 2/20 used the keyword `top`, and interestingly, others assumed that moving an item to the top of its list would be the default behavior if no position was specified (e.g. `move(woman)`).

7.2.5 Prototype Syntax

In this part, we revealed our syntax prototype to participants and asked them to answer the same questions, but this time using our syntax, to test the learnability and usability of our prototype. Participants were not allowed to test their expressions, and were discouraged from iterating as we wanted to capture their initial thinking. Therefore, correct answers in this section are equivalent to participants getting the answer right **on first try** and with no preceding training tasks.

Overall, 11 out of 17 questions had a correctness rate of 75% or above with 8 (Q1-3, Q5, Q8-10, Q17) having 90% or above, i.e almost every participant got them right on first try.

The most prominent patterns from the previous step persisted, though to a lesser extent. 7/20 participants remained unable to use a sequence of two function calls for Q4 despite this being covered in the tutorial, and wrote `add(man, woman)` or a variation. Curiously, based on later answers, all seemed to understand that the second parameter of `add()` holds initial data, yet none realized the inconsistency. Similarly, 4/20 participants still used `set(age + 1)` to increment ages, 2/20 used objects as numbers, and 8/20 used `delete(``<predicate>``)`.

Almost all failures in *"add with initial data"* questions (Q11-12) were related to grouping the key-value pairs, or incorrectly using equals signs (`=`) instead of colons (`:`) to separate them.

Two questions asked participants to delete items with a filter, but had vastly different success rates. 18/20 participants got Q9 correct, while only 9/20 got Q13 right, despite the superficial similarity of the two questions. The difference was that Q9 was operating on a list of primitives, so the values being deleted were also the same values used for filtering. It felt normal to write something like Participants had a very hard time using `hobby` twice in Q13 (The correct answer is `delete(hobby where hobby = 'Dining')` and even those that got it right hesitated a lot before writing it.

By *far* the hardest questions were Q14 and Q16, where participants had to filter on one property and set another. Only 7/18 of participants answered them correctly. All knew which function to use, and almost all used `where` correctly for filtering, but were then stuck at where to place the property they were setting. In Q14, a common answer was `set(man where age > 40, "Chris")`. Users when then unsure where to put `name`. The correct syntax in this case (if using `where`) would have been `set(man.name where age > 40, 'Chris')`, which is indeed confusing as one would expect the property being set to be grouped with its value, not with the filtering predicate.

58 %

Adding Items

For the `add()` category, we asked them several questions (e.g. *"Add a new man"*, *"Add a woman with the name "Mary" and age of 30"*, etc) that varies in their difficulties. 76.25% of participants answered these questions correctly on first try. We noticed that 81.67% of participants answered the questions that asked them to deal with one collection, man or woman, (e.g. *"Add a woman with the name "Mary" and age of 30"*), but only 60% of them answered the questions asking them to deal with both collections, man and woman, (e.g. Add a new man and a new woman). Even after seeing examples of this when shown the prototype syntax they still were unsure if they can write two functions in `mv-action` (e.g. `mv-action="add(man) add(woman)"`). 7/20 of participants still added both man and woman in the same function (e.g. `mv-action="add(man, woman)"`).

58.1 %

Deleting Items

For the `delete()` category, we asked several questions that varies in difficulties as well (e.g. Delete all men, Delete women older than 30 years old, etc). 83% of participants solved the questions in this category correctly from the first try. In this category, for the questions that include conditions (e.g. Delete "Dining" as a hobby from everyone), we found that they were more challenging for our participants than other questions that do not include conditions (e.g. Delete all people). 93.33% of participants solved the questions without conditions correctly and only 67.50% of them solved the ones with conditions. For the questions with conditionals (e.g. Delete "Dining" as a hobby from everyone), there were confused about hobby where hobby in `delete(hobby where hobby= "Dining")` I need to say why?

58.2 %

Setting Values

The `set()` category was the most challenging category for our users. In total, only 59.56% answered the questions in this category correctly. And like the `delete()` category, we had two different sets of questions. Some with conditions (e.g *"Rename every man with age > 40 to "Chris""*) and others without (e.g. *"Make current man 1 year older"*). 73.33% of participants solved the questions without conditions correctly, however, only 38.89% solved the ones with conditions correctly. For example, for the question (Make current man 1 year older), the right answer is `set(age, age+1)`, nevertheless, our participants were confused about how the set function works. They thought that they can just send the new value, without specifying what to set it to, for example, some of our participants thought that set(age+1) will automatically increase the age by one, others though that setting age+1 to the man would be sufficient to increase the man's age by one. Same thing with another question we asked them (Rename every man with age > 40 to "Chris"), they did not know what to set the name to, so they would do something like `set(man where age > 40,"Chris")` instead of `set(name where age > 40,"Chris")`, or they would not be sure about the order of setting values and using where condition, so they would do something like `set(Set (name, ’Chris’ where age > 40))`.

58.4 %

Moving Items

This involved questions such as *"Move all men to the collection of women"* (Q17), or *"Move the current woman to the beginning of the women collection"* (Q15). The category `move()` unlike `set()`, was much easier for users to understand. 93.07% of participants solved the questions in this category.

7.2.6 Hands-on Tasks

16 participants completed the hands-on section of the study (see Figure 7.6). Half were randomly assigned to the Dice Roller application, and the rest to the Words Game application. 13 also completed the Shopping List tasks.

58.5 %

Dice Roller Application

All participants solved the first two tasks correctly and were able to display a random dice roll (task 1) within a median time of **55 seconds** and to display it in the history (task 2) within a median time of **70 seconds**. 5/8 and 3/8 did so on first try. 5/8 participants hesitated before using multiple function calls in `mv-action`, even if they had answered Q4 with two function calls in the survey, but they eventually got it right.

The third task was to prevent the current dice roll from showing up in the history. Despite the second task being carefully worded to avoid implying a particular order, **all** 8 participants used `add()` after the `set()` they had written in the first task. This places the current die in the history as well as the main display. The opposite order would have rendered the third task redundant, yet nobody realized this. Furthermore, only 1 participant was able to solve the third task. All they had to do was use `add()` before `set()`, i.e. swap the order of the two functions. This would add the dice to the history *before* they replace its value with a new random value. **None** of the other 7 participants was able to figure out why this was happening, nor how to fix it.

Some participants thought that multiple function calls are executed in parallel, a common misconception of novice programmers [67]. This appears to be a general failure of computational thinking, not specific to Mavo or Formula^2.

58.6 %

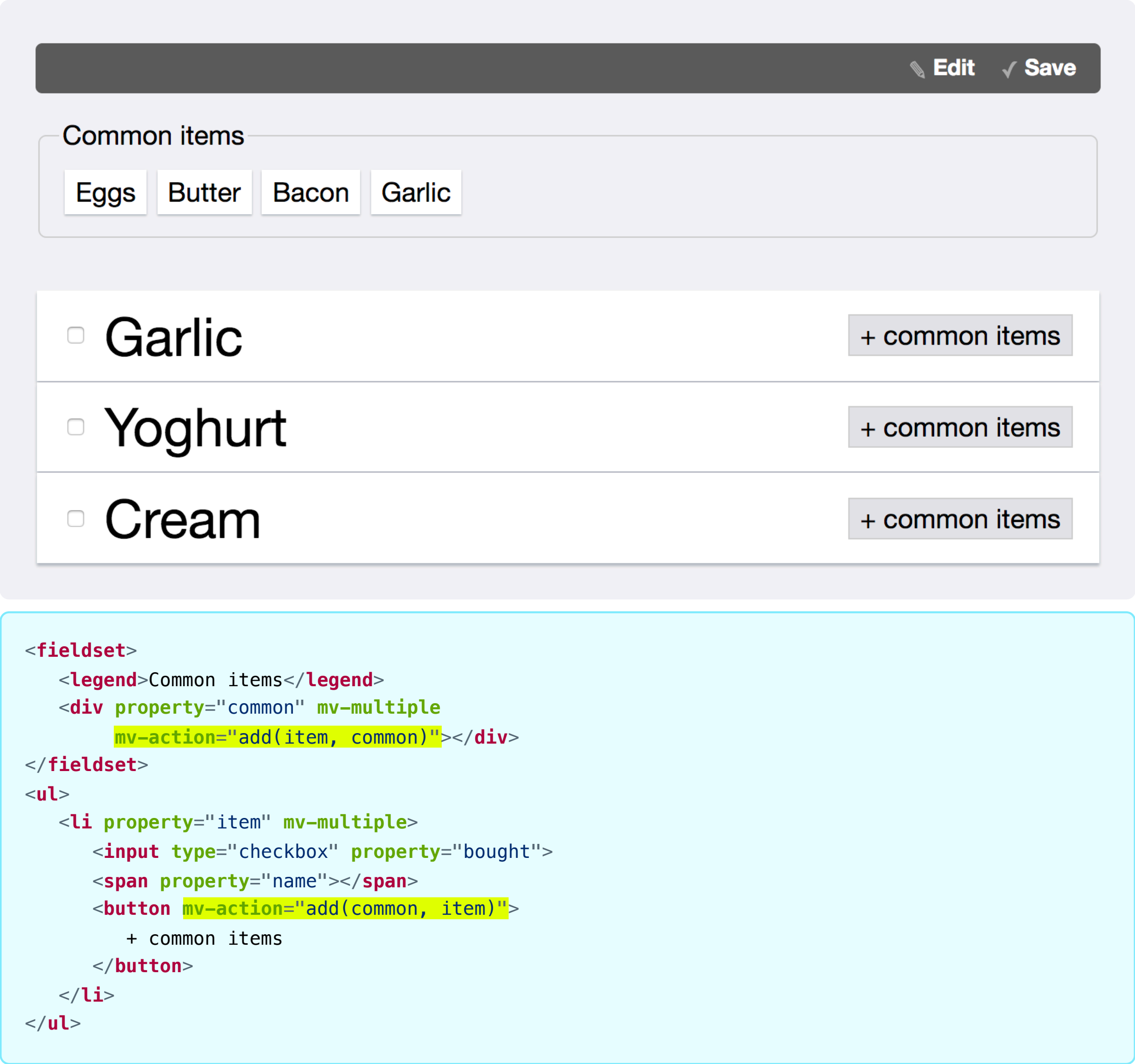
Words Game

This proved to be substantially easier than the dice roller. 6/8 participants succeeded in all three tasks. Clicking on words to add them to the sentence took a median time of 3.6 minutes, deleting the last word (Undo) took 43 seconds, and deleting all words took 2.9 minutes. For the first task, a common mistake (3/8 participants) was to use `move()` instead of `add()` to copy the clicked word into the sentence. Even after realizing their mistake, they were ambivalent about using `add()`.

58.7 %

Shopping List

13 subjects carried out the Shopping List tasks, copying to (task 1) and from (task 2) a “Common Items” collection. 6 participants brought their own application and 7 used ours.



```
<fieldset>
   <legend>Common items</legend>
   <div property="common" mv-multiple
        mv-action="add(item, common)"></div>
</fieldset>
<ul>
   <li property="item" mv-multiple>
      <input type="checkbox" property="bought">
      <span property="name"></span>
      <button mv-action="add(common, item)">
         + common items
      </button>
   </li>
</ul>
```

Figure 7.7 *The shopping list application with its solution (for participants who did not bring their own)*

Application	Copy to Common Items	Copy from Common Items
Ours	6/7 (55s)	5/7 (50s)
Theirs	6/6 (133s)	5/6 (55s)

Table 7.6 *Numbers of participants and median times for each Shopping List task*

Almost all participants succeeded in both tasks, with only 1/13 failing the first task and 3/13 failing the second one. It took slightly longer for participants using their own app to get started on the first task with a median of 133 seconds vs 55 seconds. By the second task the difference had been eliminated (55 vs 50 seconds). Three participants were confused about whether to use `move()` or `add()` to copy the shopping list item to the common items, but quickly figured it out after trying.

For the participants who used our Shopping List application: For the first task, 6/7 participants solved this task on first try within a median of 55 seconds. For the second task, also 5/7 solved the task correctly from the first try within a median time of 50 seconds. 1/7 participant was confused on where to add `mv-action` attribute.

For the participants who used the Shopping List application they created before the study using Mavo: For the first task, 6/6 participants solved it correctly within a median time of 133 seconds. For the second task, 5/6 were able to solve it successfully within a median time of 55 seconds. 4/5 participants solved from the first try.

7.2.7 System Usability Scale (SUS)

59.4 %

At the end of each session, subjects rated their subjective experience on a 7-point SUS scale with 10 alternating positive and negative questions. The answers were then coded on a 5-point scale and the SUS score was calculated according to the algorithm in [115]. We removed one participant who had selected “Agree” on all 10 questions (positive and negative), indicating lack of attention, a common problem with SUS.

Our raw SUS score was 76.3 ($\sigma_{\bar{x}}$ = 2.43), which is higher than 77.5% of all 446 studies detailed by Sauro [116] Our raw Learnability and Usability scores as defined by Lewis and Sauro [117] were 78 and 69.7 respectively.

7.2.8 General Observations

59.5 %

The overall reactions to Formula^2 + data mutation functions ranged from positive to enthusiastic. Several participants remarked on the perceived intuitiveness of the syntax. One participant answered several questions on the survey in one go, without looking at the documentation, then paused and said *“it’s so intuitive, I don’t even need to look at the docs!”*. Many other participants remarked on expressiveness; *“it is very easy to do complex things!”*, as one of our participants phrased it.

Most participants described our data update actions as easy, even those who made several mistakes. Example quotes:

> *“This is very application-centered, a page that can actually do something!”.*

> *"I think they [data mutation functions] are very useful, easy, and approachable"*

> *"it is definitely more accessible than having to program, so that's pretty cool"*

> *"They are easier and quicker to make things without worrying about technicality. It is very easy to use"*

As with the first study, many participants liked being able to use this functionality by editing HTML as opposed to editing in a separate file and/or language:

> *"Interesting to be able to do these things from the HTML!"*

> *"It is interesting!..being able to do this in HTML, I was able to use it pretty easily, once I knew what functions there were and the syntax it has it was very easy."*

Users also liked the fact that they can build applications that typically require programming.

> *"This is easier than JavaScript! If I wanted to do something complicated I would be frustrated to use JavaScript cause I'm not good at it, this is easier".*

> *"It's easier and quicker to make things without worrying about technicality. It's very easy to use".*

Several participants commented positively on the `where` operator.

> *"the where syntax is like natural language, I did not expect it to be there and written as if I am saying it".*

7.3 Mavo in the Wild Informal Interviews

59.8 %

In September 2020, we contacted Mavo authors we found from website access logs and interviewed five of them about their experiences. While this never resulted in a published study, a very prominent pattern was present across most interviews: users generally *loved* the parts of Mavo that were core to its design: they found the syntax intuitive and the capabilities very powerful.

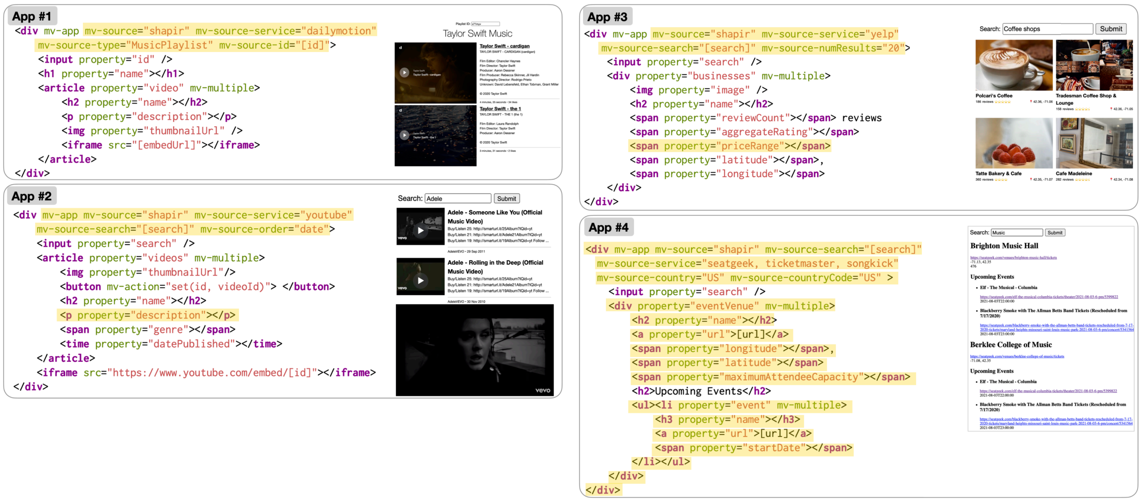
However, they were having a lot of trouble with superficial aspects of the design and prototype implementation, the most prominent of which were:

- They wanted server-side rendering for their content, not having data fetched client-side
- They did not like the loading indicator and found it too intrusive
- Performance was slow.

More often than not, these were insurmountable problems to them and eventually drove them away.

For Mavo to gain wider adoption, it is important to invest in addressing such issues, even if the research value of such work is only in the longer term.

7.4 Shapir: Standardizing and Democratizing Web APIs



App #1

```html
<div mv-app mv-source="shapir" mv-source-service="dailymotion"
   mv-source-type="MusicPlaylist" mv-source-id="[id]">
   <input property="id" />
   <h1 property="name"></h1>
   <article property="video" mv-multiple>
      <h2 property="name"></h2>
      <p property="description"></p>
      <img property="thumbnailUrl" />
      <iframe src="[embedUrl]"></iframe>
   </article>
</div>
```

App #2

```html
<div mv-app mv-source="shapir" mv-source-service="youtube"
   mv-source-search="[search]" mv-source-order="date">
   <input property="search" />
   <article property="videos" mv-multiple>
      <img property="thumbnailUrl"/>
      <button mv-action="set(id, videoId)"> </button>
      <h2 property="name"></h2>
      <p property="description"></p>
      <span property="genre"></span>
      <time property="datePublished"></time>
   </article>
   <iframe src="https://www.youtube.com/embed/[id]"></iframe>
</div>
```

App #3

```html
<div mv-app mv-source="shapir" mv-source-service="yelp"
  mv-source-search="[search]" mv-source-numResults="20">
    <input property="search" />
    <div property="businesses" mv-multiple>
       <img property="image" />
       <h2 property="name"></h2>
       <span property="reviewCount"></span> reviews
       <span property="aggregateRating"></span>
       <span property="priceRange"></span>
       <span property="latitude"></span>,
       <span property="longitude"></span>
    </div>
</div>
```

App #4

```html
<div mv-app mv-source="shapir" mv-source-search="[search]"
  mv-source-service="seatgeek, ticketmaster, songkick"
  mv-source-country="US" mv-source-countryCode="US" >
    <input property="search" />
    <div property="eventVenue" mv-multiple>
       <h2 property="name"></h2>
       <a property="url">[url]</a>
       <span property="longitude"></span>,
       <span property="latitude"></span>
       <span property="maximumAttendeeCapacity"></span>
       <h2>Upcoming Events</h2>
       <ul><li property="event" mv-multiple>
          <h3 property="name"></h3>
          <a property="url">[url]</a>
          <span property="startDate"></span>
       </li></ul>
    </div>
</div>
```

Figure 7.8 *The four apps completed by participants in the Mavo-Shapir study: (1) Dailymotion playlist viewer, (2) YouTube video search, (3) Yelp & Foursquare search, (4) Event searching app combining SeatGeek, Ticketmaster, Songkick.*

Alrashed et al [97] integrated Mavo with ShapirJS, a JavaScript library that normalizes data from various APIs into schema.org [98] ontologies and exposes them as *"live"* JavaScript objects that can seamlessly perform asynchronous API calls to fetch additional data and can update remote data via standard JavaScript object manipulation methods.

Combined, ShapirJS and Mavo make it possible to create standalone web applications that read, combine, and manipulate data over multiple web APIs without writing any JavaScript or back-end code.

Mavo-Shapir was implemented as a Madata backend, and packaged as a Mavo plugin. To support this integration, Formula^2 was extended to support asynchronous values, which opened up many new possibilities for what Formula^2 expressions.

They then evaluated this combined system in a lab study with 16 participants (9 female, ages 18-60). Of these, 8 identified as beginner or intermediate in HTML, and 8 as advanced or expert. Their programming skills ranged from none to skilled: 2 with no programming skills, 6 beginners, 6 intermediate and 2 skilled. In terms of Mavo familiarity, 7 participants had used Mavo before, 4 had heard of it but not used it, and 5 had never heard of it.

Participants were given functional Mavo apps operating on local data, and they had to adapt them to work with live data from various APIs. **All participants** were able to complete the tasks in **under 4 minutes**, with apps 2-3 taking them about **a minute** on average. There was no correlation between time taken and programming skill or Mavo familiarity.

Participants were generally very positive about the experience; they found the combination of Mavo and Shapir easy to use, and were impressed by how quickly they could build applications that combine data from multiple sources.

7.5 Wikxhibit: Using Mavo and Wikidata to Author Applications that Link Data Across the Web

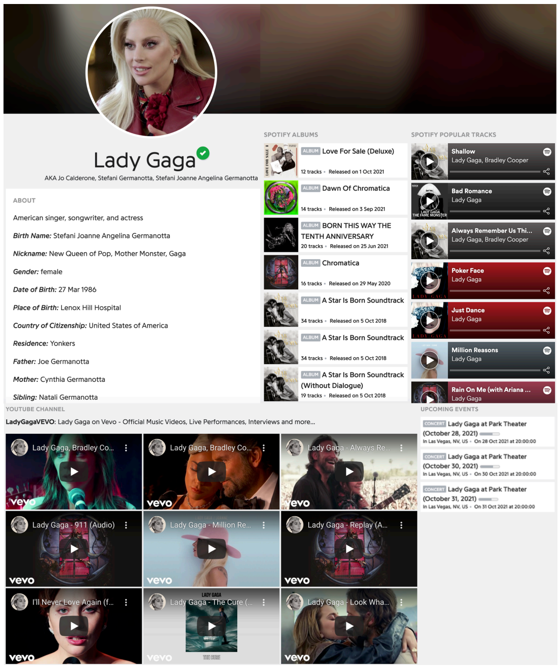
60.2 %

Alrashed et al later integrated Mavo-Shapir with Wikidata, a free and open knowledge base that can be read and edited by both humans and machines [118] to create mashups of data from multiple APIs. Effectively, Wikidata is used as a universal join table to cross-reference entities across different third-party APIs (see Figure 7.9 for an example).

They then evaluated this system in a lab study of 12 Wikidata users. Of the participants, 8 identifed as beginner or intermediate in HTML, and 4 as advanced or expert. Additionally, 7 described themselves as beginners or worse in any programming language, while 5 considered themselves intermediate or better.

HTML

```html
<div mv-app mv-source="wikxhibit"
     mv-source-service="wikidata"
     mv-source-id="Q19848">
<img property="image" />
<h1 property="label"></h1>
<div property="SpotifyArtist">
   <div property="albums" mv-multiple>
      <img property="image" />
      <h5 property="name"></h5>
      <p property="numTracks"></p>
   </div>
   <div property="tracks" mv-multiple>
      <iframe src="[embedUrl]"></iframe>
   </div>
</div>
<div property="YouTubeChannel">
   <div property="videos" mv-multiple>
      <iframe src="[embedUrl]"></iframe>
   </div>
</div>
<div property="SongkickArtist">
   <div property="events" mv-multiple>
      <h5 property="name"></h5>
      <p property="location"></p>
      <p property="startDate"></p>
   </div>
</div>
</div>
```



Lady Gaga

AKA Jo Calderone, Stefani Germanotta, Stefani Joanne Angelina Germanotta

ABOUT

American singer, songwriter, and actress

Birth Name: Stefani Joanne Angelina Germanotta

Nickname: New Queen of Pop, Mother Monster, Gaga

Gender: female

Date of Birth: 27 Mar 1986

Place of Birth: Lenox Hill Hospital

Country of Citizenship: United States of America

Residence: Yonkers

Father: Joe Germanotta

Mother: Cynthia Germanotta

Sibling: Natali Germanotta

SPOTIFY ALBUMS

ALBUM Love For Sale (Deluxe)
12 tracks • Released on 1 Oct 2021

ALBUM Dawn Of Chromatica
14 tracks • Released on 3 Sep 2021

ALBUM BORN THIS WAY THE TENTH ANNIVERSARY
20 tracks • Released on 25 Jun 2021

ALBUM Chromatica
16 tracks • Released on 29 May 2020

ALBUM A Star Is Born Soundtrack
34 tracks • Released on 5 Oct 2018

ALBUM A Star Is Born Soundtrack
34 tracks • Released on 5 Oct 2018

ALBUM A Star Is Born Soundtrack (Without Dialogue)
19 tracks • Released on 5 Oct 2018

SPOTIFY POPULAR TRACKS

Shallow
Lady Gaga, Bradley Cooper

Bad Romance
Lady Gaga

Always Remember Us Thi...
Lady Gaga, Bradley Cooper

Poker Face
Lady Gaga

Just Dance
Lady Gaga

Million Reasons
Lady Gaga

Rain On Me (with Ariana ...
Lady Gaga

YOUTUBE CHANNEL

LadyGagaVEVO: Lady Gaga on Vevo - Official Music Videos, Live Performances, Interviews and more...

UPCOMING EVENTS

CONCERT Lady Gaga at Park Theater (October 28, 2021)
In Las Vegas, NV, US • On 28 Oct 2021 at 20:00:00

CONCERT Lady Gaga at Park Theater (October 30, 2021)
In Las Vegas, NV, US • On 30 Oct 2021 at 20:00:00

CONCERT Lady Gaga at Park Theater (October 31, 2021)
In Las Vegas, NV, US • On 31 Oct 2021 at 20:00:00

Figure 7.9 *An artist page made with Mavo and Wikxhibit that displays integrated data from different websites: general information about the artist from Wikidata, their albums and tracks from Spotify, their videos from YouTube, and their events from Songkick.*

Of these participants, 7 participated in the structured study, where they were given static HTML & CSS scaffolding for three applications, and they had to write Mavo HTML to pull in different data sources and display their data in a suitable way.

After going through a brief tutorial, 6/7 users went on to complete all the tasks for their three applications with no failures (1/7 had to leave early). They took, on average, 14 minutes (Tech Company, 6 tasks), 9 minutes (US President, 5 tasks), and 5 minutes (Botanical Gardens, build an app from scratch) to build the entire application.

One study finding relevant to Mavo was that a common slip was participants forgetting to add `mv-multiple` attributes and being confused when they could only see one item. This is part of Mavo's schema mapping heuristics (Section 3.6.2), to allow seamlessly converting between scalars and lists without data loss and displaying the same data across different Mavo apps. However, perhaps `mv-path` could be used instead to explicitly opt-in to this behavior. A compromise could be to display all array items, but not add any UI for adding new ones until `mv-list-item` (nee `mv-multiple`) is specified.

Five participants participated in the later freestyle study, creating their own applications from scratch. All were able to accomplish the applications they set out to build, in less than 30 minutes (3/5 in less than 15). The applications created included a page about the movie "The Big Lebowski", a presentation showing a list of the superior courts of California a page showing political parties of a user-provided country specifed via an input field, and an application shows information about comic strips that are part of xkcd[3].

7.6 Lifesheets: Exposing Mavo Concepts to Non-programmers

The next chapter presents LIFESHEETS, whose contribution is twofold: First, to explore the value and feasibility of empowering end-users to create their own tracking applications, and second, to expose lightly abstracted Mavo concepts (especially Formula²) to non-programmers via a GUI.

Its user study provides insights on both, and is described in detail in Section 9.7. Here we pull in only the high level insights that relate to Mavo concepts.

By exposing forms to generate formula calls we were able to short-circuit many syntax errors, while still exposing enough of Formula² to get valuable insights. Additionally, unlike the previous Formula² studies, it included a control condition, where users were also writing spreadsheet formulas for the same tasks (*between subjects* on the task, *within subjects* on the condition).

[3] xkcd.com

7.6.1 Formula²

61.8 %

Overall, the study validated our hypothesis that novices can largely understand and use Formula² for grouping, aggregation, and temporal calculations, and that these tasks are very difficult with spreadsheets.

Despite spreadsheets supporting UI features that make these tasks easier (e.g. data validation, pivot tables), the fact that these are separate features that users need to know about in practice meant that they were not used. Instead, users were painfully trying to accomplish these tasks with formulas, until they gave up or did them manually or semi-manually (e.g. emulating a pivot table via a manual list of values and `SUMIF()` formulas).

A recurring theme we observed was the end-users' **distaste for indirection**. Participants generally expected to be able to be able to accomplish their goals via a combination of UI settings, or a single formula call, and continued trying different parameters until they either got the right result (often as a happy accident rather than an accurate mental model), or got frustrated and gave up.

While not appearing in this study, there were also use cases that required auxiliary data in Formula² (which can only be created via the host environment, e.g. Mavo). A common example is nested aggregates (e.g. average of averages) — before the `in` operator, they required a hidden Mavo property and could not be done with Formula² alone (see Section 4.7.3).

61.9 %

Temporal Calculations

A big takeaway from the study was that temporal calculations in spreadsheets were extremely painful, whereas the Formula² counterpart was generally a lot more understandable. Perhaps the most characteristic such case is calculating intervals and displaying them in a human readable way (see also Section 4.7.1.3), for which Formula² provides a high level primitive.

However, even with Formula² there were some recurring mistakes that highlight areas for improvement, mainly around values only being useful when used a certain way, and not another that users tried. This included:

- Attempting to do math with `duration()` (which returns an array of strings)
- Printing out `$now` and getting confused at the result (a number of milliseconds)
- Setting a time property to `$now` and getting confused that it didn't work (since it expects a string like `"HH:mm"`, and `$now` is a number)

These attempts are reasonable things that *should* and *could* work if these functions and special properties return objects that retain their metadata and thus can be used in a variety of ways, rather than primitives like numbers of strings.

7.6.2 Mavo & Madata

Mavo HTML was more highly abstracted, and thus many study findings are less directly applicable to it. However, there were still some insights that are relevant.

The biggest issue was related to *Saving*. Having to click a *Save* button to persist their data felt foreign to users, who expected their data to be saved automatically. While Mavo supports an `mv-autosave` attribute, it is discouraged for backends that save to version control systems, in order to maintain a meaningful edit history (which can later be exposed via a UI). Perhaps a best of both worlds approach could be to autosave to a local storage, that is periodically synced to the VCS.

Placeholder entries (which Mavo automatically creates unless `mv-initial-items="0"` is used) seemed natural while the application was being built, but baffled users when they attempted to use the deployed application for the first time.

CHAPTER 8

Case Studies

6,183 words (18 min read)

As an additional resource, a list of case studies is presented in this chapter. These showcase all technologies in the Mavo ecosystem working together to produce high fidelity applications.

Some of them are included because they represent particularly common use cases, others because they push the boundaries of what is possible with Mavo, and others because they showcase interesting patterns.

8.1 CRUD Applications

62.4 %

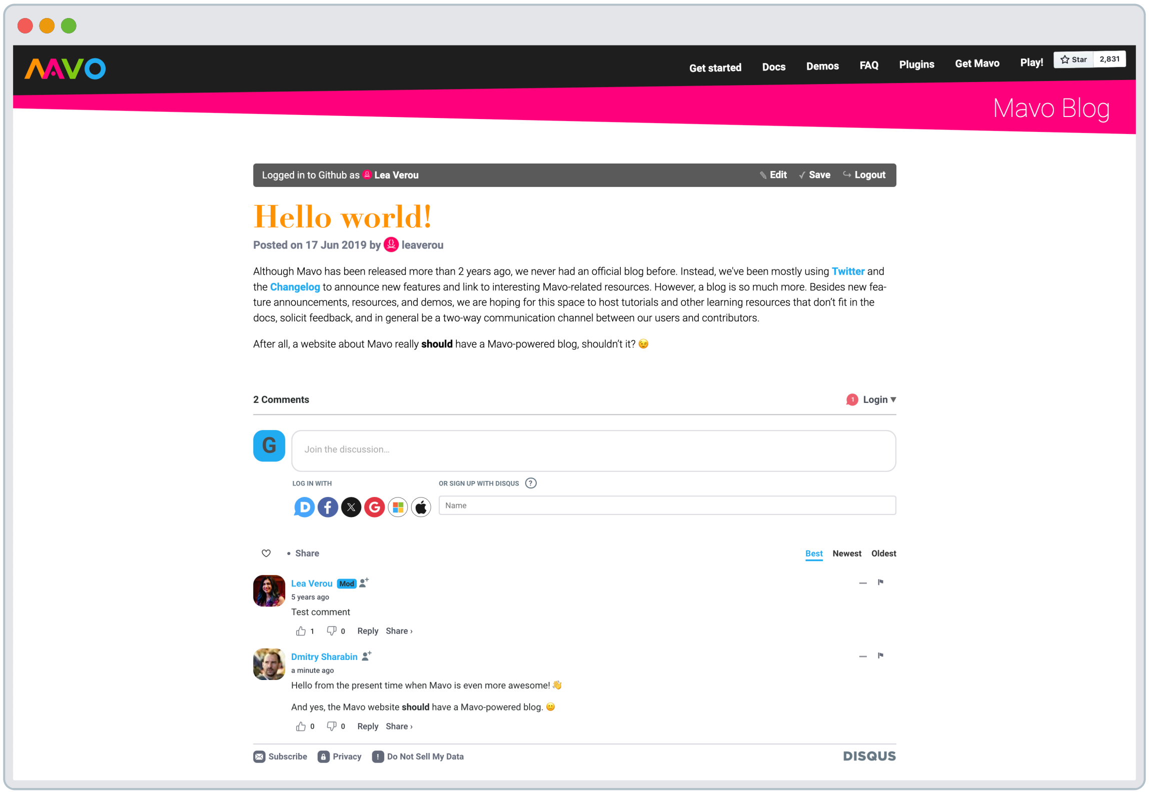
This section includes case studies of a few typical CRUD applications, some of very common mainstream use cases (blog, e-shop, portfolio), and some more specialized (research group website, CSS WG Disposition of Comments).

These types of applications are exactly the type of applications that Mavo was designed for, though some still showcase interesting patterns, such as integrations with third-party services.

8.1.1 Blog with Disqus Comments *(by Lais Tomaz)*

62.5 %

This case study showcases how Mavo can be used to build multi-page applications, and how it can integrate a third-party service (Disqus) to support comments.



Logged in to Github as Lea Verou

Hello world!

Posted on 17 Jun 2019 by leaverou

Although Mavo has been released more than 2 years ago, we never had an official blog before. Instead, we've been mostly using Twitter and the Changelog to announce new features and link to interesting Mavo-related resources. However, a blog is so much more. Besides new feature announcements, resources, and demos, we are hoping for this space to host tutorials and other learning resources that don't fit in the docs, solicit feedback, and in general be a two-way communication channel between our users and contributors.

After all, a website about Mavo really **should** have a Mavo-powered blog, shouldn't it? 😉

2 Comments

Join the discussion…

Share

Lea Verou Mod
5 years ago
Test comment
1 0 Reply Share ›

Dmitry Sharabin
a minute ago
Hello from the present time when Mavo is even more awesome! 👋

And yes, the Mavo website **should** have a Mavo-powered blog. 🙂
0 0 Reply Share ›

Figure 8.1 *A fully functional blog built with Mavo. The live version can be accessed at mavo.io/blog.*

Architecture

This application consists of two HTML files: `index.html` and `post/index.html`, with 40 and 60 lines of HTML respectively. It requires no JavaScript besides the edits needed to Disqus' embed code (described below).

Displaying a List of Posts

The `index.html` file lists all posts, with their metadata and an excerpt:

HTML

```html
<div mv-app="blog" mv-plugins="markdown"
     mv-storage="https://github.com/mavoweb/mavo.io/blog/posts.json">
   <section id="[id]" mv-multiple property="post">
      <h2>
         <meta property="id" mv-default="[ idify(title) ]">
         <a href="/post/?post=[id]" property="title" mv-attribute="none"></a>
      </h2>

      <div property="excerpt" class="markdown">[ excerpt(content, words: 100) ]</div>
      <footer>
         Posted on <time property="date" mv-default="[$today]"></time>
         by <a property="author" mv-attribute="none" href="https://github.com/[author]"></a>
```

HTML (continued)

```html
         &bull; <a href="https://mavo.io/post/?post=[id]#disqus_thread">Comments</a>
      </footer>
   </section>
</div>
```

An `id` is generated from each post based on the title, but can also be edited by the user. Note that `<meta>` elements are Mavo's recommended way for storing metadata that is not displayed to the end-user (but become visible and editable in edit mode).

Displaying A Single Post

The `post/index.html` file displays a single post in full. Its structure is very similar (except the `content` property is displayed in full). The main difference the `mv-path` attribute in the app definition:

HTML

```html
<div mv-app="blog" mv-plugins="markdown"
     mv-storage="https://github.com/mavoweb/mavo.io/blog/posts.json"
     mv-path="post/id=[ url('post') ]">
```

This defines a **two-way data transformation** that consists of subsetting the fetched data before it is rendered, so that the Mavo app defined within only needs to handle a single post, and then recombining the edited data back into the original data structure when saving. This can be applied both to the entire application, or to individual properties, which can be useful when handling pre-existing data, such as data fetched from an API.

Displaying Comments

To embed comments on a post, we use Disqus, a popular commenting service. Disqus provides a JavaScript snippet that needs to be included on the page where comments are to be displayed. As instructed by Disqus, this needs to be modified to communicate the page id and URL to Disqus. The parts we had to add or modify are highlighted below:

HTML

```html
<div id="disqus_thread"></div>
<script>
var disqus_config = function () {
   let id = Mavo.Functions.url("post");
   this.page.url = "https://mavo.io/blog/post/?post=" + id;
   this.page.identifier = id;
};

(function() { // DON'T EDIT BELOW THIS LINE
var d = document, s = d.createElement('script');
```

HTML (continued)

```
s.src = 'https://mavo-blog.disqus.com/embed.js';
s.setAttribute('data-timestamp', +new Date());
(d.head || d.body).appendChild(s);
})();
</script>
```

Disqus also provides JS code for displaying the number of comments for each post on the list page. To make that work with Mavo, it needs to be wrapped in an event listener. The code we added to Disqus' snippet is highlighted below:

HTML

```
<script id="dsq-count-scr" src="//mavo-blog.disqus.com/count.js" async></script>
<script>
   document.addEventListener("mv-load", evt => {
      DISQUSWIDGETS.getCount({reset: true});
   });
</script>
```

64.4 %

Limitations

URL Structure

Currently, our blog posts don't have very nice URLs, e.g. a blog post with an id of `foo` is accessed at `/post/?post=foo`. Ideally, we'd want a URL such as `/post/foo`. This *can* be done with Mavo, as its `url()` function handles both patterns, so we would not even need to modify our code. However, to do that we would need to first route unknown (not found) URLs of that format *to* the `post/index.html` page. Mavo cannot help with that, as it operates on the client side.

Most web servers can be configured to do this, but the code is not always straightforward code for novices to write. For example, in the popular Apache web server, this can be done with a `.htaccess` file, but it requires regular expressions:

CODE

```
RewriteEngine On
RewriteRule ^post/.*$ /post/index.html [L]
```

With other servers it can be simpler for common cases like this. For example, on Netlify, a *"serverless"* web hosting provider, this can be done with a `_redirects` file:

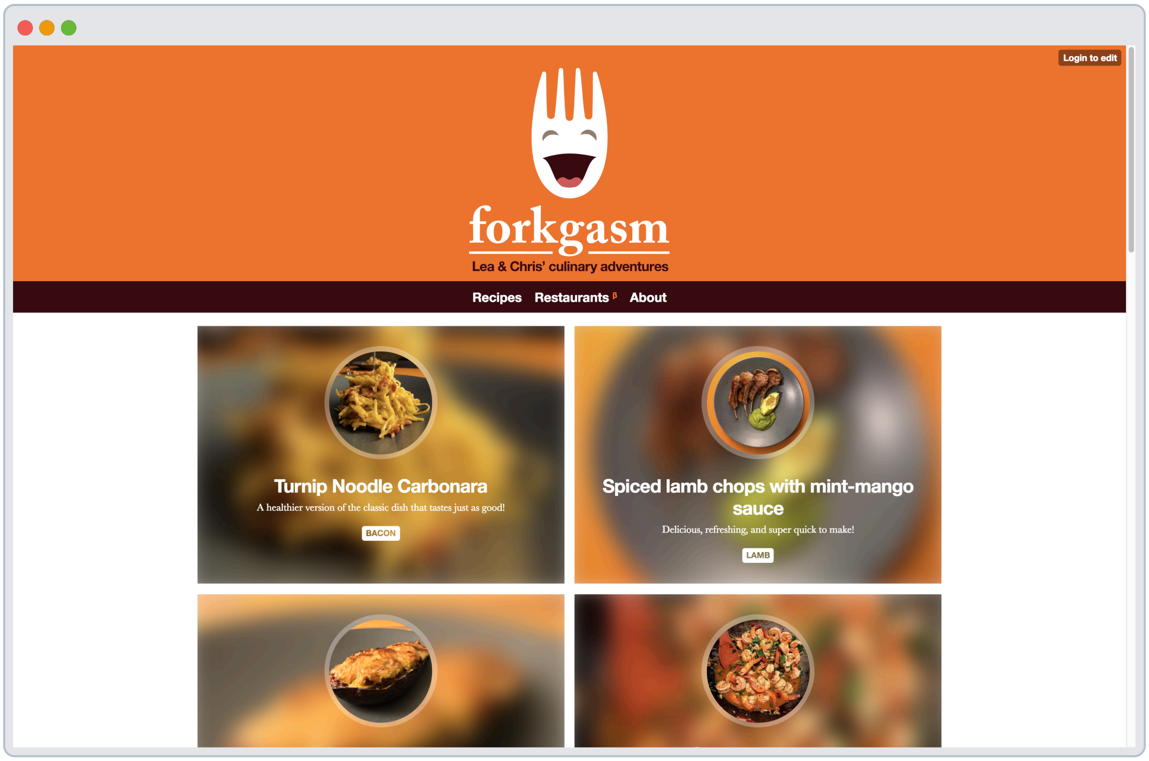
CODE

```
/post/* /post/index.html 200
```

Variations

Variations of this pattern can be used to build more specialized types of multipage websites. Some examples are presented below.

Recipe Manager



Turnip Noodle Carbonara

A healthier version of the classic dish that tastes just as good!

Spiced lamb chops with mint-mango sauce

Delicious, refreshing, and super quick to make!

Figure 8.2 *A multipage recipe manager built with Mavo. The live version can be accessed at forkgasm.com.*

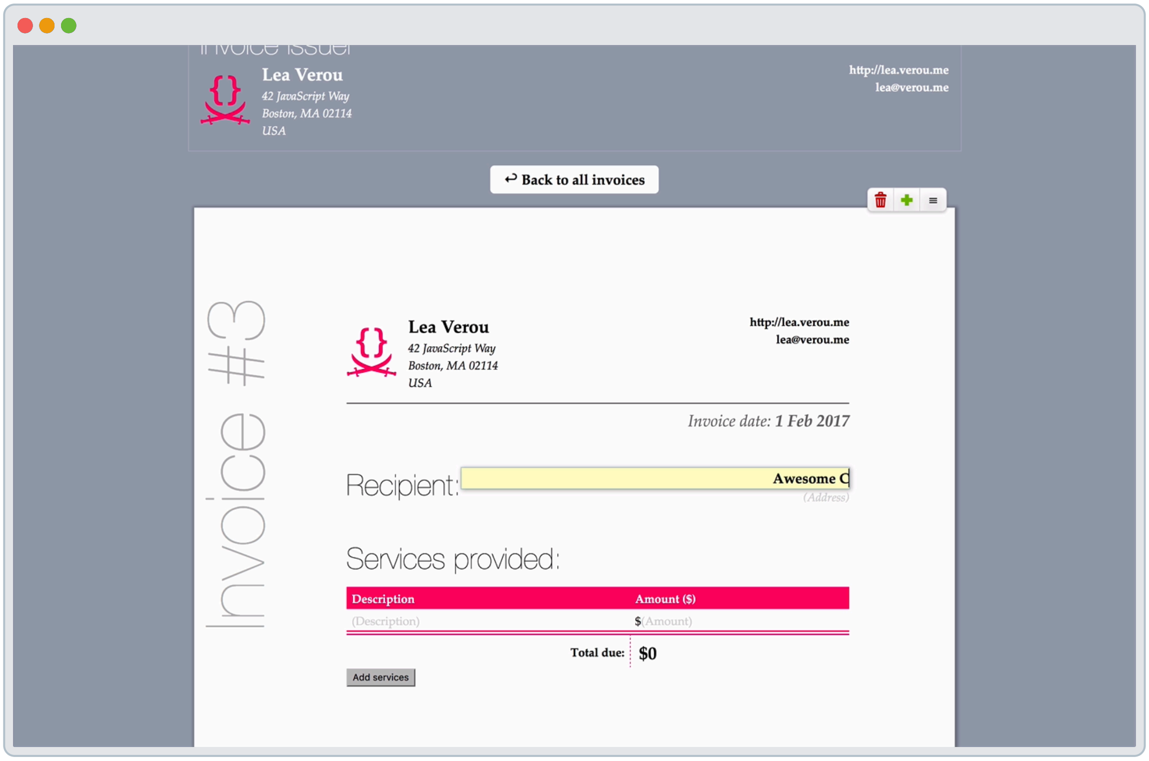
In this example, a page is used to list recipes and a different page to display a single recipe.

Instead of a single `content` property for the entry content, each recipe here contains a list of ingredients and a list of steps.

The recipe manager also showcases a current limitation of Mavo: the lack of a primitive for *data formatting*. Indeed, amounts need to be displayed as fractional numbers, such as

0.125, rather than the far more common fractional numbers that recipes typically use, such as ⅛.

Invoicing & Expensing Application



Invoice issuer

Lea Verou
42 JavaScript Way
Boston, MA 02114
USA

http://lea.verou.me
lea@verou.me

Invoice #3

Lea Verou
42 JavaScript Way
Boston, MA 02114
USA

http://lea.verou.me
lea@verou.me

Invoice date: 1 Feb 2017

Recipient: Awesome C
(Address)

Services provided:

Description	Amount ($)
(Description)	$(Amount)

Total due: $0

Figure 8.3 *A fully functional invoicing application built with Mavo. The live version can be accessed at mavo.io/demos/invoice.*

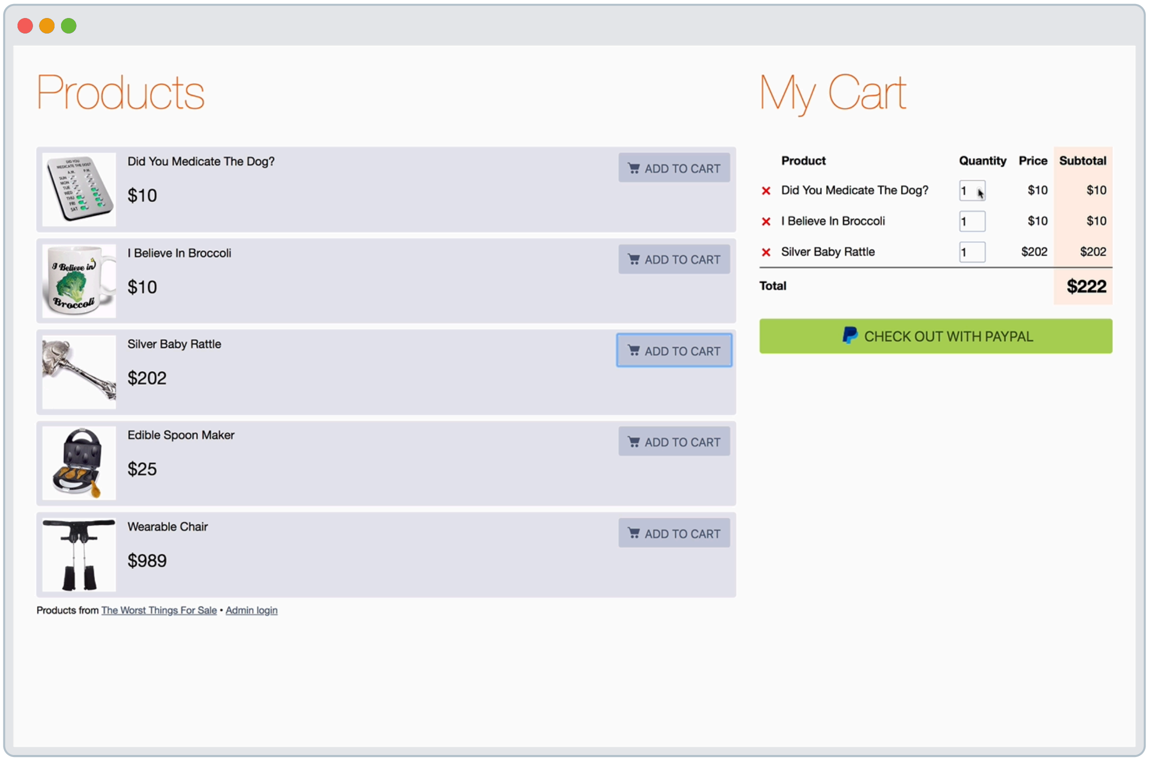
Here, a list of all invoices is displayed on one page, and each invoice is displayed on another. Each invoice includes a list of services provided, with amounts for each, and a total sum. The consultant's information (company name, address, logo etc.) is defined once as root properties, and copied on every invoice.

8.1.2 E-shop with PayPal Checkout

64.8 %

This is a fully-functional e-shop application, which integrates with PayPal, a well-known payment provider that provides an HTML integration for its API through form submission with a predefined parameter schema.

This case study highlights several concepts of the Mavo development approach, but is also an excellent example for the **interoperability** of HTML-based approaches. Neither



Products

Did You Medicate The Dog?
$10

I Believe In Broccoli
$10

Silver Baby Rattle
$202

Edible Spoon Maker
$25

Wearable Chair
$989

Products from The Worst Things For Sale • Admin login

My Cart

	Product	Quantity	Price	Subtotal
×	Did You Medicate The Dog?	1	$10	$10
×	I Believe In Broccoli	1	$10	$10
×	Silver Baby Rattle	1	$202	$202
Total				$222

Figure 8.4 *A fully functional e-shop built with Mavo. The live version can be accessed at mavo.io/demos/eshop.*

Mavo nor PayPal needed to know about each other, but because they both support HTML-based syntaxes, they work together seamlessly, with no need for any plugin or integration.

64.9 %

Architecture

While conceptually a single Mavo application, this is architected as two Mavo apps, each with different access control: one for managing the list of products, and one for managing the shopping cart. The former is viewable by everyone, but editable only by the e-shop administrator, and stored on a cloud service like GitHub:

HTML

```html
<div mv-app="eshop" mv-storage="https://github.com/…/products.json" mv-bar="no-login">
   <article property="product" mv-list-item>
      <img property="image" alt="" />
      <span property="name"></span>
      <button mv-action="add(product, cart.product)">Add to cart</button>
      <span property="amount" class="price"></span>
   </article>
</div>
```

The latter is editable by everyone and stored locally:

HTML

```html
<form mv-app="cart" mv-storage="local" mv-autosave mv-mode="edit"
      action="https://paypal.com/cgi-bin/webscr" method="POST">
   <table>
      <thead>
         <tr><th>Product</th> <th>Quantity</th> <th>Price</th> <th>Subtotal</th></tr>
      </thead>
      <tbody mv-list mv-initial-items="0" mv-item-bar="delete">
         <tr property="product" mv-list-item>
            <th property="name">
               <span class="mv-ui mv-item-bar"></span>
               <span property="name" mv-editor-name="item_name_[$index + 1]" inert></span>
            </th>
            <td>
               <input type="number" property="quantity" name="quantity_[$index + 1]" value="1">
            </td>
            <td property="amount" mv-editor-name="amount_[$index + 1]" inert></td>
            <td property="subtotal">[amount * quantity]</td>
         </tr>
      </tbody>
      <tfoot>
         <tr><th colspan="3">Total:</th> <td>[sum(subtotal)]</td></tr>
      </tfoot>
   </table>

   <button disabled="[count(product) = 0]">Check out with PayPal</button>

   <input type="hidden" name="cmd" value="_cart">
   <input type="hidden" name="upload" value="1">
   <input type="hidden" name="business" value="admin@fooshop.com">
   <input type="hidden" name="currency_code" value="USD">
</form>
```

The entire application is also a form that submits to PayPal. Any form elements with `name` attributes become part of the form submission, and communicate data to PayPal. Hidden form elements communicate variables that do not need to be displayed in the UI.

To send the product name and price to PayPal without allowing them to be edited by the user, we set a name on the generated editing element via `mv-editor-name` and added the HTML `inert` attribute to prevent the user from interacting with it and its descendants. A simpler but more verbose solution would be to use more hidden inputs, with formulas to copy the `name` and `amount` properties into them, and `mv-mode="read"` so that no editing UI is generated.

Cross-app Data Flow

The two apps interact with each other: the `eshop` app uses a data action to add products to the cart, by copying them to the cart app's storage. Note that Formula[2] **implicit scoping does not apply across apps**: since the trees are disconnected, the id of the other app (`cart`) had to be explicitly used. But once we obtain a reference to that app's data tree, implicit scoping works as expected.

This is not a bug or a limitation, but an intentional design decision. We felt that the small usability benefit of being able to reference properties from *other* data trees is outweighed by the potential for confusion and mistakes, since other Mavo applications may be very spatially disconnected from each other, and since cross-app references are relatively uncommon, the extra verbosity is not a big issue.

Edit-only Apps

By default, Mavo uses two modes: a read mode (the default) which is used for presenting data, and an edit mode which is used for editing data in place. However, certain applications, such as the shopping cart here only need one mode. Sometimes that is only a read-only mode. In these cases, the `mv-mode` attribute can be set to `read` to make the app (or parts of it) *read-only*. However, there are use cases that benefit from Mavo's editing controls, but do not need a read mode, they are *edit-only*. These can be specified via `mv-mode="edit"`.

The shopping cart is a good example of this: we want certain editing functionality to be always there (such as deleting collection items or editing quantities), certain editing functionality to *never* be there (such as adding new products). Editability is fragmented spatially rather than temporally.

Perisist Cart Across Devices?

To keep this demo simple, the cart is persisted locally. However most big e-shops support user accounts and then persist the cart across devices. While it is trivial to change the cart app to use cloud storage instead of local storage, this does not exactly give us the same pattern, unless that cloud storage allowed public writes and we were happy to store each guest's cart in it. But more likely, what we actually want is a hybrid: local storage for guests, and cloud storage for authenticated users.

This is still *possible* but certainly not *easy*. We can use formulas as the `mv-storage` value to use local storage for non-authenticated users and cloud storage for authenticated users, but this introduces a bit of a chicken-and-egg problem: how would the user log in to the cloud service, if there is no login UI because the app is using a local storage backend?

Since we can only have one storage backend per app, we would need a *third* app that is always cloud-based, and only contains the login UI, and take advantage of the fact that Mavo will synchronize user accounts across apps using the same type of backend (e.g. GitHub).

HTML

```html
<div mv-app="cart_login" mv-source="https://url/to/cloud">
```

We can then use a formula to set the cart's storage backend based on whether the user is logged in or not:

HTML

```html
<form mv-app="cart" mv-storage="[if ($user or cart_login.$user, 'https://url/to/cloud', 'local')]">
```

This is still not perfect: if a user starts shopping as a guest and then logs in, their cart will be lost. We can use `mv-init="local"` to address this, but that only works if the remote data is empty. It remains an open question what the best primitive is for handling such cases in a general way.

67.7 %

UI Customization

Discreet Authentication UI

The administrator gets the usual Mavo UI, with its promiment toolbar. For the public, the toolbar is hidden, by hiding its login button (via `mv-bar="no-login"`). Since logged out users see no other controls, this hides the entire toolbar for them. Instead, there is a discreet "Admin login" link in the footer:

HTML

```html
<a href="?login" class="mv-login">Admin login</a>
```

The link has class `mv-login` which makes it behave like a login button automatically (hidden for logged-in users, triggers login UI when clicked, etc.). Every toolbar button has a corresponding class so that its functionality can be reused on custom UI elements even outside the toolbar. The link target is `?login`, which is a special URL that triggers the login UI for the first Mavo app in the page. Such a link is available in every Mavo application (`?login` shows login UI for the first Mavo app in the page but a specific app id can be specified as the URL parameter's value). This allows for the login UI to be *entirely* hidden if the author desires.

Restricted List Editing

Since `cart.product` is a collection that is *only* edited via a data action, showing functionality to add items to it is meaningless. We can hide the add button using regular CSS:

CSS

```css
[mv-app="cart"] .mv-add-product {
  display: none;
}
```

To limit adding and reordering via the item bar, Mavo provides an `mv-item-bar` attribute that can be used similarly to `mv-bar`:

HTML

```html
<tbody mv-list mv-initial-items="0" mv-mode="edit" mv-item-bar="delete">
```

This will only show the delete button next to each item, and hide any controls for adding and reordering items.

8.1.3 Artist Portfolio

68.1 %

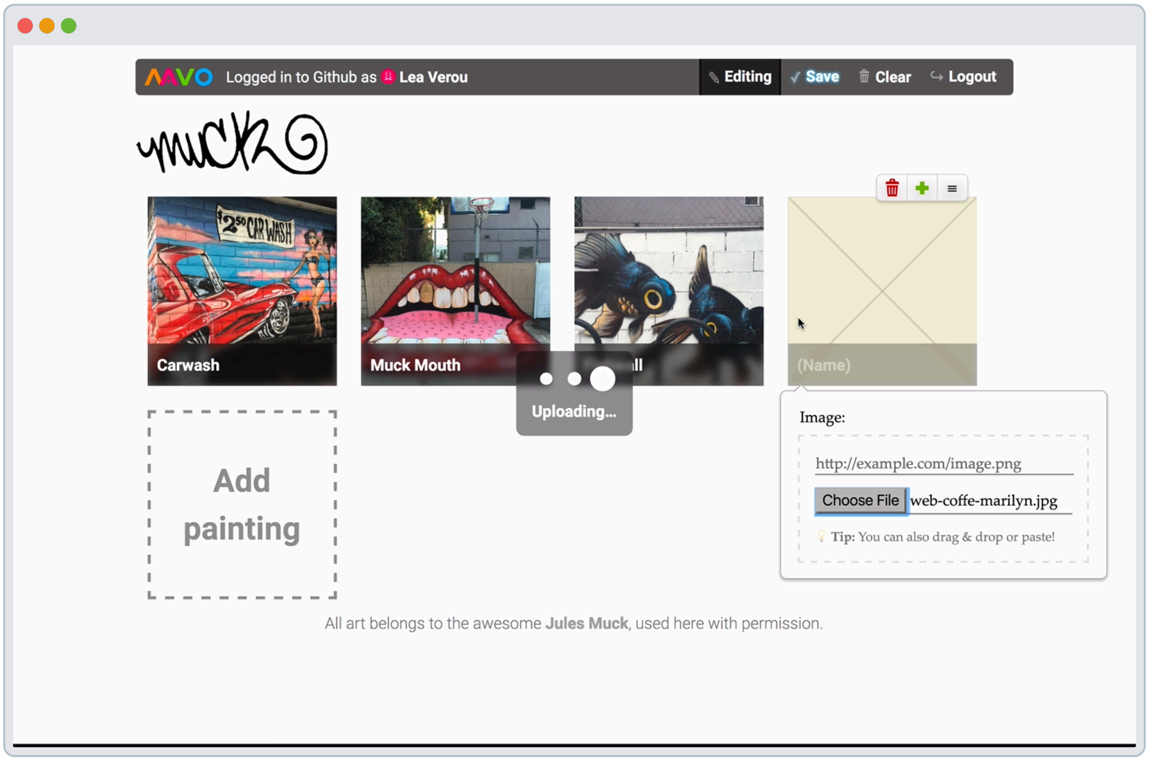
From a Mavo point of view, this is a very simple use case. So simple in fact, it is included here as a case study on how such a high fidelity application allowing images to be uploaded, pasted, or linked and edited in a number of ways can be built with such a small amount of code.

The entire page is 25 lines of HTML, with no JavaScript, of which only 8 are Mavo markup:

HTML

```html
<main mv-app="portfolio" mv-storage="https://github.com/mavoweb/data/portfolio">
  <div mv-list property="painting">
    <a mv-list-item mv-attribute="none">
      <img property="image" />
      <p property="name" mv-default="[ readable(to(filename(image), '.')) ]"></p>
    </a>
  </div>
</main>
```

Individual pages for each painting could be easily added using a similar pattern as the blog case study (Section 8.1.1). Note that reactive defaults crop up even in such a simple



Logged in to Github as Lea Verou

Image:

http://example.com/image.png

Choose File web-coffe-marilyn.jpg

Tip: You can also drag & drop or paste!

All art belongs to the awesome **Jules Muck**, used here with permission.

Figure 8.5 *An editable artist portolio. The live version can be accessed at mavo.io/demos/portfolio.*

application: the filename of the image is processed to create the default value for the `name` property, a common pattern for these types of applications.

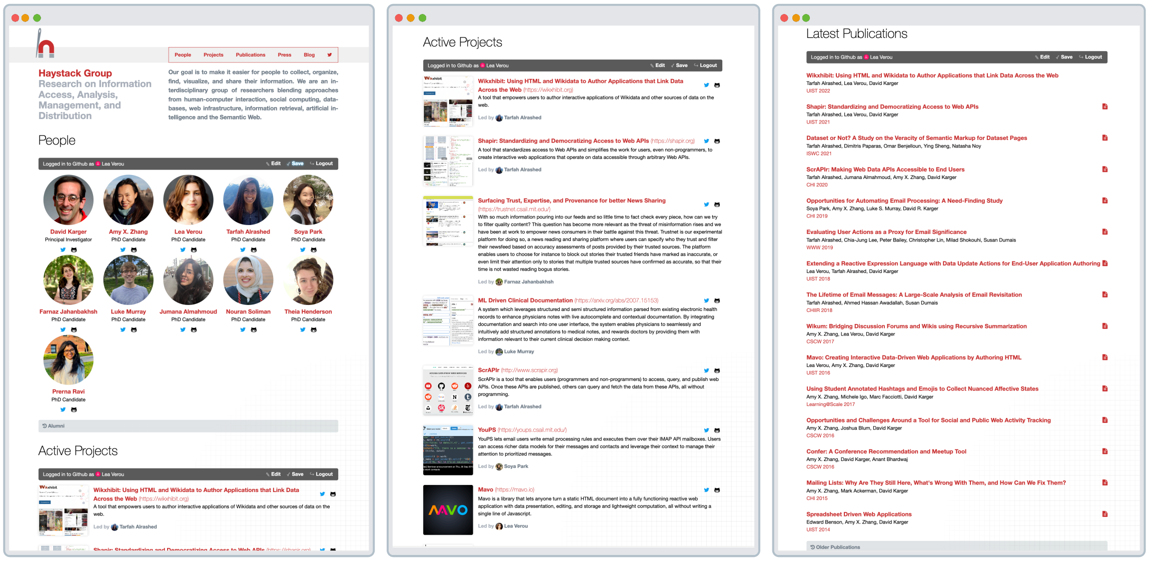
Limitations

Images are resized by the browser to display thumbnails, but the full image is still downloaded. It *would* be possible to handle thumbnail generation in Mavo via a plugin that generates thumbnails locally when uploading an image, but due to the Web's same origin policy [119, 120], this would not be possible for linked images.

68.6 %

8.1.4 Research Group Website

This application showcases how the current Mavo primitives can be used to emulate a graphical schema.



Haystack Group
Research on Information Access, Analysis, Management, and Distribution

Our goal is to make it easier for people to collect, organize, find, visualize, and share their information. We are an interdisciplinary group of researchers blending approaches from human-computer interaction, social computing, databases, web infrastructure, information retrieval, artificial intelligence and the Semantic Web.

People

Logged in to Github as Lea Verou

David Karger, Principal Investigator
Amy X. Zhang, PhD Candidate
Lea Verou, PhD Candidate
Tarfah Alrashed, PhD Candidate
Soya Park, PhD Candidate
Farnaz Jahanbakhsh, PhD Candidate
Luke Murray, PhD Candidate
Jumana Almahmoud, PhD Candidate
Nouran Soliman, PhD Candidate
Theia Henderson, PhD Candidate
Prerna Ravi, PhD Candidate

Alumni

Active Projects

Logged in to Github as Lea Verou

Wikxhibit: Using HTML and Wikidata to Author Applications that Link Data Across the Web (https://wikxhibit.org)
A tool that empowers users to author interactive applications of Wikidata and other sources of data on the web.
Led by Tarfah Alrashed

Shapir: Standardizing and Democratizing Access to Web APIs (https://shapir.org)
A tool that standardizes access to Web APIs and simplifies the work for users, even non-programmers, to create interactive web applications that operate on data accessible through arbitrary Web APIs.
Led by Tarfah Alrashed

Surfacing Trust, Expertise, and Provenance for better News Sharing (https://trustnet.csail.mit.edu/)
With so much information pouring into our feeds and so little time to fact check every piece, how can we try to filter quality content? This question has become more relevant as the threat of misinformation rises and we have been at work to empower news consumers in their battle against this threat. Trustnet is our experimental platform for doing so, a news reading and sharing platform where users can specify who they trust and filter their newsfeed based on accuracy assessments of posts provided by their trusted sources. The platform enables users to choose for instance to block out stories their trusted friends have marked as inaccurate, or even limit their attention only to stories that multiple trusted sources have confirmed as accurate, so that their time is not wasted reading bogus stories.
Led by Farnaz Jahanbakhsh

ML Driven Clinical Documentation (https://arxiv.org/abs/2007.15153)
A system which leverages structured and semi structured information parsed from existing electronic health records to enhance physicians notes with live autocomplete and contextual documentation. By integrating documentation and search into one user interface, the system enables physicians to seamlessly and intuitively add structured annotations to medical notes, and rewards doctors by providing them with information relevant to their current clinical decision making context.
Led by Luke Murray

ScrAPIr (http://www.scrapir.org)
ScrAPIr is a tool that enables users (programmers and non-programmers) to access, query, and publish web APIs. Once these APIs are published, others can query and fetch the data from these APIs, all without programming.
Led by Tarfah Alrashed

YouPS (https://youps.csail.mit.edu/)
YouPS lets email users write email processing rules and executes them over their IMAP API mailboxes. Users can access richer data models for their messages and contacts and leverage their context to manage their attention to prioritized messages.
Led by Soya Park

Mavo (https://mavo.io)
Mavo is a library that lets anyone turn a static HTML document into a fully functioning reactive web application with data presentation, editing, and storage and lightweight computation, all without writing a single line of Javascript.
Led by Lea Verou

Latest Publications

Logged in to Github as Lea Verou

Wikxhibit: Using HTML and Wikidata to Author Applications that Link Data Across the Web
Tarfah Alrashed, Lea Verou, David Karger
UIST 2022

Shapir: Standardizing and Democratizing Access to Web APIs
Tarfah Alrashed, Lea Verou, David Karger
UIST 2021

Dataset or Not? A Study on the Veracity of Semantic Markup for Dataset Pages
Tarfah Alrashed, Dimitris Paparas, Omar Benjelloun, Ying Sheng, Natasha Noy
ISWC 2021

ScrAPIr: Making Web Data APIs Accessible to End Users
Tarfah Alrashed, Jumana Almahmoud, Amy X. Zhang, David Karger
CHI 2020

Opportunities for Automating Email Processing: A Need-Finding Study
Soya Park, Amy X. Zhang, Luke S. Murray, David R. Karger
CHI 2019

Evaluating User Actions as a Proxy for Email Significance
Tarfah Alrashed, Chia-Jung Lee, Peter Bailey, Christopher Lin, Milad Shokouhi, Susan Dumais
WWW 2019

Extending a Reactive Expression Language with Data Update Actions for End-User Application Authoring
Lea Verou, Tarfah Alrashed, David Karger
UIST 2018

The Lifetime of Email Messages: A Large-Scale Analysis of Email Revisitation
Tarfah Alrashed, Ahmed Hassan Awadallah, Susan Dumais
CHIIR 2018

Wikum: Bridging Discussion Forums and Wikis using Recursive Summarization
Amy X. Zhang, Lea Verou, David Karger
CSCW 2017

Mavo: Creating Interactive Data-Driven Web Applications by Authoring HTML
Lea Verou, Amy X. Zhang, David Karger
UIST 2016

Using Student Annotated Hashtags and Emojis to Collect Nuanced Affective States
Amy X. Zhang, Michele Igo, Marc Facciotti, David Karger
Learning@Scale 2017

Opportunities and Challenges Around a Tool for Social and Public Web Activity Tracking
Amy X. Zhang, Joshua Blum, David Karger
CSCW 2016

Confer: A Conference Recommendation and Meetup Tool
Amy X. Zhang, David Karger, Anant Bhardwaj
CSCW 2016

Mailing Lists: Why Are They Still Here, What's Wrong With Them, and How Can We Fix Them?
Amy X. Zhang, Mark Ackerman, David Karger
CHI 2015

Spreadsheet Driven Web Applications
Edward Benson, Amy X. Zhang, David Karger
UIST 2014

Older Publications

Figure 8.6 *A fully functional research group website built with Mavo. The live version can be accessed at haystack.csail.mit.edu.*

68.7 %

Architecture

The application consists of three Mavo apps, one for each section of the website: `people`, `projects`, and `publications`, each reading and writing a different JSON file. This is a design decision primarily to produce reusable JSON files that can also function as a basic data API, and to keep their size manageable, but the website could have easily used the same file for all three sections.

As with the e-shop example, the Mavo toolbar is only visible for logged in users. Logged out users simply see a discreet *"Log in to Github to edit data"* link in the footer of each section.

The `people` application includes a collection of people with their names, images, URLs, and other information:

HTML

```html
<article property="member" typeof="Person" mv-multiple>
   <a property="url">
      <img property="image" alt="" mv-uploads="../images/people">
      <h3 property="name">Name</h3>
   </a>
   <!-- job title, social media handles, etc. -->
</article>
```

It then uses this to generate a `<datalist>` as a dynamic collection over the names of all people in the group:

HTML

```html
<div hidden>
   <datalist mv-list id="member_list" mv-value="name">
      <option mv-list-item></option>
   </datalist>
</div>
```

Then, on the collections that need to reference people, the generated editing UI is linked to this `<datalist>` to facilitate data entry via an (HTML-native) autocomplete widget. A hidden computed property is then reactively populated with the selected person's data. For example, this is how the leader of a project is selected:

HTML

```html
<footer>
   Led by
   <a href="[leader.url]">
      <img src="[leader.image]" alt="">
      <span property="leader_name" mv-edit-list="member_list">Leader</span>
      <meta property="leader" mv-value="people where name = leader_name" />
   </a>
</footer>
```

For publications, we only need to look up the author URL, so we can avoid defining the hidden property:

HTML

```html
<p mv-list class="authors">
   <a property="author" mv-attribute="none" href="[people.url where name = author]"
      mv-edit-list="member_list"></a>
</p>
```

This is essentially using the person's name as a unique, immutable identifier, i.e. a *primary key*. If there is no property that is a good candidate for this, a unique identifier can be generated using a formula (e.g. `idify(name)`), and used as the default value of a hidden property, so that it can be edited by the user:

HTML

```html
<meta property="id" mv-default="[ idify(name) ]" />
```

Lookups would then need to be adjusted accordingly.

70 %

Limitations

Many issues with this pattern stem from the fact that it is essentially a workaround around the lack of real primary keys and foreign keys in Mavo.

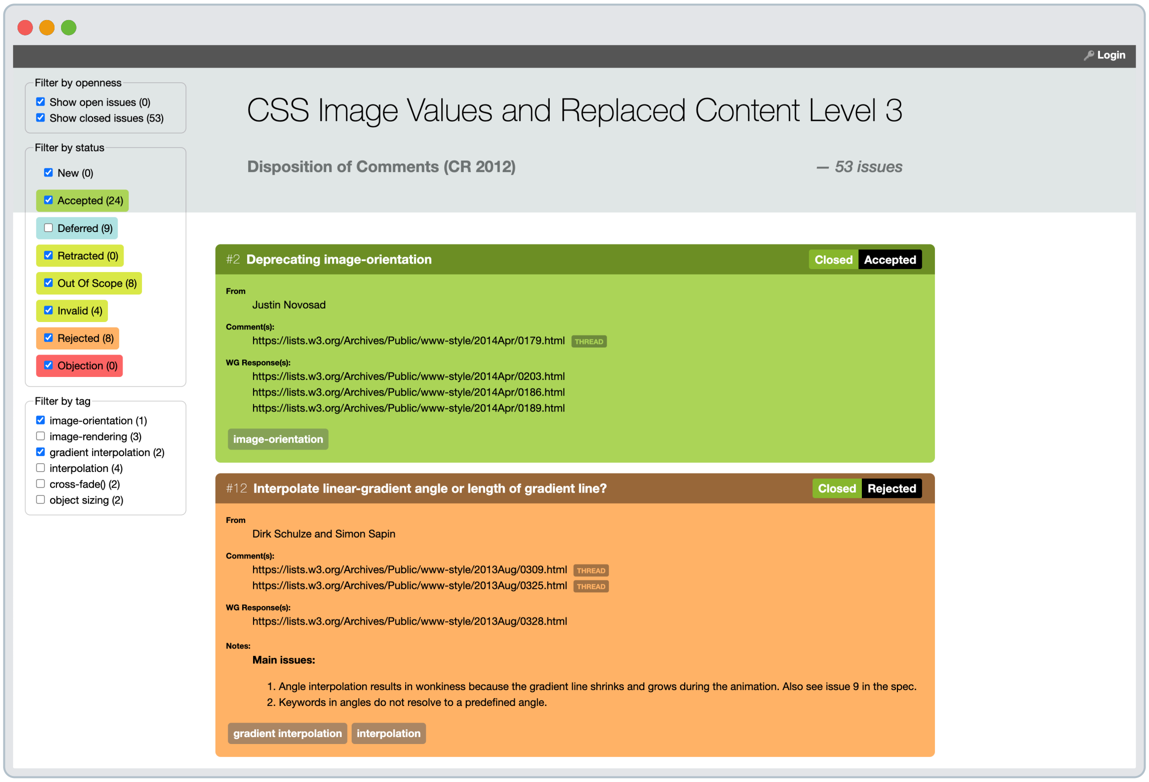
Uniqueness of the property used as a key is not enforced. It *could* be communicated to the user via form validation, but this would simply be informative, it would not prevent them from saving corrupted data.

But a bigger issue is that this suffers from poor *closeness of mapping*, breaking the declarativeness of general Mavo syntax. Authors cannot express their intent with the language primitives, so they have to express low-level steps for accomplishing their goal.

These issues are discussed in more detail in the next chapter.

8.1.5 CSS WG Disposition of Comments

70.1 %



Filter by openness
☑ Show open issues (0)
☑ Show closed issues (53)

Filter by status
☑ New (0)
☑ Accepted (24)
☐ Deferred (9)
☑ Retracted (0)
☑ Out Of Scope (8)
☑ Invalid (4)
☑ Rejected (8)
☑ Objection (0)

Filter by tag
☑ image-orientation (1)
☐ image-rendering (3)
☑ gradient interpolation (2)
☐ interpolation (4)
☐ cross-fade() (2)
☐ object sizing (2)

CSS Image Values and Replaced Content Level 3

Disposition of Comments (CR 2012) — *53 issues*

#2 Deprecating image-orientation Closed Accepted

From
Justin Novosad

Comment(s):
https://lists.w3.org/Archives/Public/www-style/2014Apr/0179.html THREAD

WG Response(s):
https://lists.w3.org/Archives/Public/www-style/2014Apr/0203.html
https://lists.w3.org/Archives/Public/www-style/2014Apr/0186.html
https://lists.w3.org/Archives/Public/www-style/2014Apr/0189.html

image-orientation

#12 Interpolate linear-gradient angle or length of gradient line? Closed Rejected

From
Dirk Schulze and Simon Sapin

Comment(s):
https://lists.w3.org/Archives/Public/www-style/2013Aug/0309.html THREAD
https://lists.w3.org/Archives/Public/www-style/2013Aug/0325.html THREAD

WG Response(s):
https://lists.w3.org/Archives/Public/www-style/2013Aug/0328.html

Notes:
Main issues:

1. Angle interpolation results in wonkiness because the gradient line shrinks and grows during the animation. Also see issue 9 in the spec.
2. Keywords in angles do not resolve to a predefined angle.

gradient interpolation interpolation

Figure 8.7 *A custom app to manage Disposition of Comments for the CSS Working Group, showcasing how Mavo can be used to build custom intersecting filtering widgets. The live version can be accessed at drafts.csswg.org/issues?spec=css-images-3&doc=cr-2012.*

Dispositions of Comments is part of the process for advancing a W3C specification to the next stage, by ensuring all issues raised against a draft specification by the community have been considered. It was typically managed as plain text at the time. This application was authored to improve this process (though as WG issues moved to GitHub Issues, it was later replaced by a convention based on GitHub labels).

This application showcases how filtering and sorting functionality can be implemented in Mavo, but is also an excellent example of the kinds of data management applications that Mavo was designed for: the *long tail* of use cases that are too niche individually, yet vast in aggregate.

70.2 %

Architecture

We can use a separate `filters` Mavo app to persist filters locally:

HTML

```html
<aside mv-app="filters" mv-storage="local">
   <!-- Filtering UI -->
</aside>
<main mv-app id="issues"
      mv-storage="https://github.com/w3c/csswg-drafts/[ url('spec') ]/issues-[ url('doc') ].yaml">
   <!-- List of issues -->
</main>
```

Each filter is a dynamic collection. For filters on single-valued properties, such as `status` the grouping operator can be used:

HTML

```html
<fieldset mv-list property="status" mv-value="issues.issue by status">
   <legend>Filter by status</legend>
   <label mv-list-item>
      <input type="checkbox" property="show" checked>
      <span property="status" mv-editor="#statuses"></span> ([ count($items) ])
   </label>
</fieldset>
```

> **NOTE**
> Note that while the `status` property in the dynamic collection is not editable (as `statusFilter` is a computed property), we had to specify `mv-editor="#statuses"` which points to a `<select>` menu, in order to get the same displayed text for each value. Admittedly, the *closeness of mapping* for this could be improved…

Unfortunately, as described in the Formula^2 chapter, the grouping operator does not currently work well with multi-valued properties, such as issue tags. For those, we need to use a lower level approach, essentially doing the grouping ourselves:

HTML

```html
<fieldset>
   <legend>Filter by tag</legend>

   <label property="tag" mv-list-item mv-value="unique(issues.tag)">
      <input type="checkbox" property="show" checked>
      <span property="text">[tag]</span>
      ([ count(issue.tag = text) ])
   </label>
</fieldset>
```

Then, to filter the issues based on the selected tags, we can define a `hidden` property as an *immutable collection*, with a value for each filter:

HTML

```html
<meta property="hidden" content="[ count(filters.status where show and id=status) ]" />
<meta property="hidden" content="[ intersects(tag, filters.tag where show) ]" />
```

And then use an aggregate disjunction of these properties to determine if each issue should be hidden:

HTML

```html
<article property="issue" mv-list-item hidden="[ or(hidden) ]">
```

> ASIDE
>
> Why use the HTML `hidden` attribute rather than Mavo's `mv-if`? Because `mv-if` actually affects the data model, and thus we would no longer be able to display values for hidden items, making it impossible to revert the filter!

Limitations

71.4 %

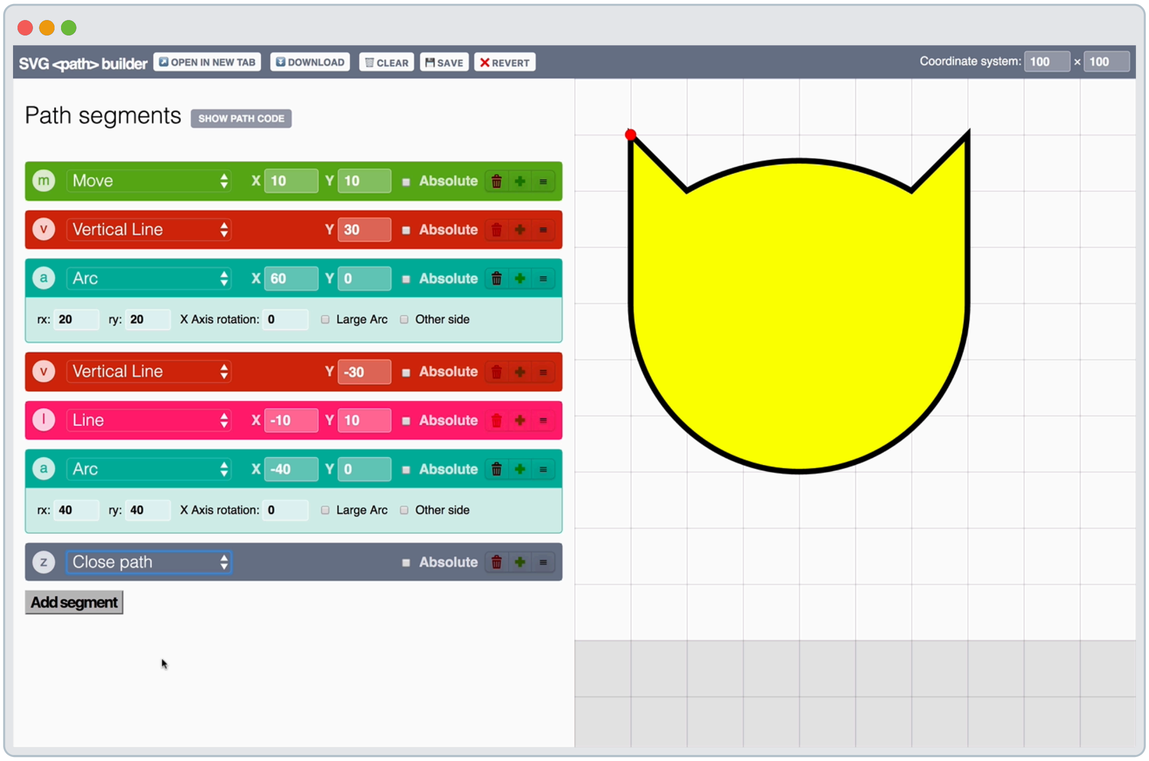
One limitation of this approach has already been discussed above: the lack of a good way to group by multi-valued properties in Formula^2. But the bigger issue is that this approach is *too low-level* for such a common task. This issue is discussed in more detail in the next chapter.

8.2 Graphics Builders

In this section, we present two applications that push the boundaries of what is possible with Mavo, by using it to build GUIs that generate graphics.

Both of these showcase a pattern for creating **heterogeneous collections**, as well as different ways to use Mavo to generate code in different languages: SVG in the first application, JavaScript and LOGO in the second.

8.2.1 SVG Path Builder



Coordinate system: 100 × 100

Path segments

m Move X 10 Y 10 ■ Absolute

v Vertical Line Y 30 ■ Absolute

a Arc X 60 Y 0 ■ Absolute
rx: 20 ry: 20 X Axis rotation: 0 □ Large Arc □ Other side

v Vertical Line Y -30 ■ Absolute

l Line X -10 Y 10 ■ Absolute

a Arc X -40 Y 0 ■ Absolute
rx: 40 ry: 40 X Axis rotation: 0 □ Large Arc □ Other side

z Close path ■ Absolute

Figure 8.8 *A fully functional e-shop built with Mavo. The live version can be accessed at mavo.io/demos/svgpath.*

This is certainly not a typical CRUD application, but this SVG path demo showcases many interesting patterns. It also makes a slightly different argument on interoperability: HTML-based approaches are only only interoperable with HTML itself, but also any syntax that can be embedded in it, such as SVG. This means that Mavo can be used to create interactive graphics and diagrams, or even — as this demo shows — entire editors.

Architecture

The entire application is 126 lines of HTML, with no JavaScript.

The main data structure of this application is a collection of path segments (`segment`), each item corresponding to a single SVG path segment command. There are also root level properties to parameterize the coordinate system (`width` and `height`) and style (`fill`, `stroke`, `strokeWidth`). of the generateed graphic.

A computed `pathsummary` property within each item is used to generate the actual SVG path string for that segment:

HTML

```html
<meta property="pathsummary"
     content="[if(absolute, uppercase(type), type)] [arcFlags] [x1] [y1] [x2] [y2] [x] [y]">
```

Then, outside the collection, all these are combined into another computed property, containing the entire path, which is also displayed to the end-user for copying:

HTML

```html
<textarea property="path">[replace(join(pathsummary, ' '), '  ', ' ', 10)]</textarea>
```

> **NOTE**
> The `replace()` is only there to make the path string more readable by collapsing sequences of spaces, and is not required (`10` is the number of times it should run on it own output).

Then, all that is needed to generate the download link is this:

HTML

```html
<a href='data:image/svg+xml,<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 [width] [height]">
   <path d="[path]" fill="[fill]" stroke="[stroke]" stroke-width="[strokeWidth]" />
</svg>' download="path.svg" target="_blank">⏬ Download SVG</a>
```

The actual inline preview is a little longer, as it also displays conveniences that improve feedback for the end-user, such as a grid, and a marker to display the current position of the last point of the path:

HTML

```html
<svg viewBox="0 0 [width] [height]" preserveAspectRatio="xMinYMin">
   <pattern id="grid" viewBox="0 0 10 10" patternUnits="userSpaceOnUse" width="10" height="10">
      <line y2="10" vector-effect="non-scaling-stroke" />
      <line x2="10" vector-effect="non-scaling-stroke" />
   </pattern>
   <marker id="pos" viewBox="-1 -1 1 1" markerUnits="userSpaceOnUse"
         markerWidth="1" overflow="visible" >
      <circle r="1" cx="0" cy="0" />
```

HTML (continued)

```html
   </marker>
   <rect width="100%" height="100%" fill="white" />
   <rect width="100vw" height="100vh" fill="url(#grid)" />
   <path d="[path]" marker-end="url(#pos)" fill-rule="evenodd"
         fill="[fill]" stroke="[stroke]" stroke-width="[strokeWidth]" />
</svg>
```

Heterogeneous Collections

This application showcases a common pattern for editing and storing heterogeneous collections, i.e. collections where each item has different properties.

Typically, these have some overlap across different types of items, otherwise there would be little reason to store them in the same collection. Path segments are no exception: they all have a type, a boolean `absolute` property, and X and Y coordinates for their end point (except one: close path (`z`) segments do not take any coordinates).

Frequently in heterogeneous collections, there is a main property that determines the item *type*, in this case the path segment type:

HTML

```html
<select property="type">
   <option value="m">Move</option>
   <option value="l">Line</option>
   <option value="h">Horizontal Line</option>
   <option value="v">Vertical Line</option>
   <option value="a">Arc</option>
   <option value="s">Smooth Bézier curve</option>
   <option value="c">Bézier curve</option>
   <option value="t">Smooth Quadratic Bézier</option>
   <option value="q">Quadratic Bézier</option>
   <option value="z">Close path</option>
</select>
```

Then we can use declarative `mv-if` conditionals to show and hide the appropriate fields for each type:

HTML

```html
<label mv-if="type != v and type != z">
   X <input type="number" property="x" />
</label>

<label mv-if="type != h and type != z">
   Y <input type="number" property="y" />
</label>
```

It is important to note that `mv-if` does not simply modify the HTML, like similarly namedj features in JS frameworks which treat HTML as purely a view. It actually *modifies the data* that expressions see (but not the data that is stored, to avoid data loss).

74 %

Limitations

Direct Manipulation

While this demo highlights many of Mavo's strengths, it also highlights one of its core limitations: it cannot handle complex interactions, such as those needed by a visual graphics editor (e.g. dragging points to change the path).

How to enable such interactions within Mavo is an open question.

Editing Existing Paths

For this to be truly useful, it should also support editing existing paths. Then authors could paste an existing path into the editor, edit it, and copy the result back out. However, there is no way to write a Formula² formula that parses (nontrivial) syntax.

End-user parser programming is a whole research area in itself, and is currently out of scope for Mavo.

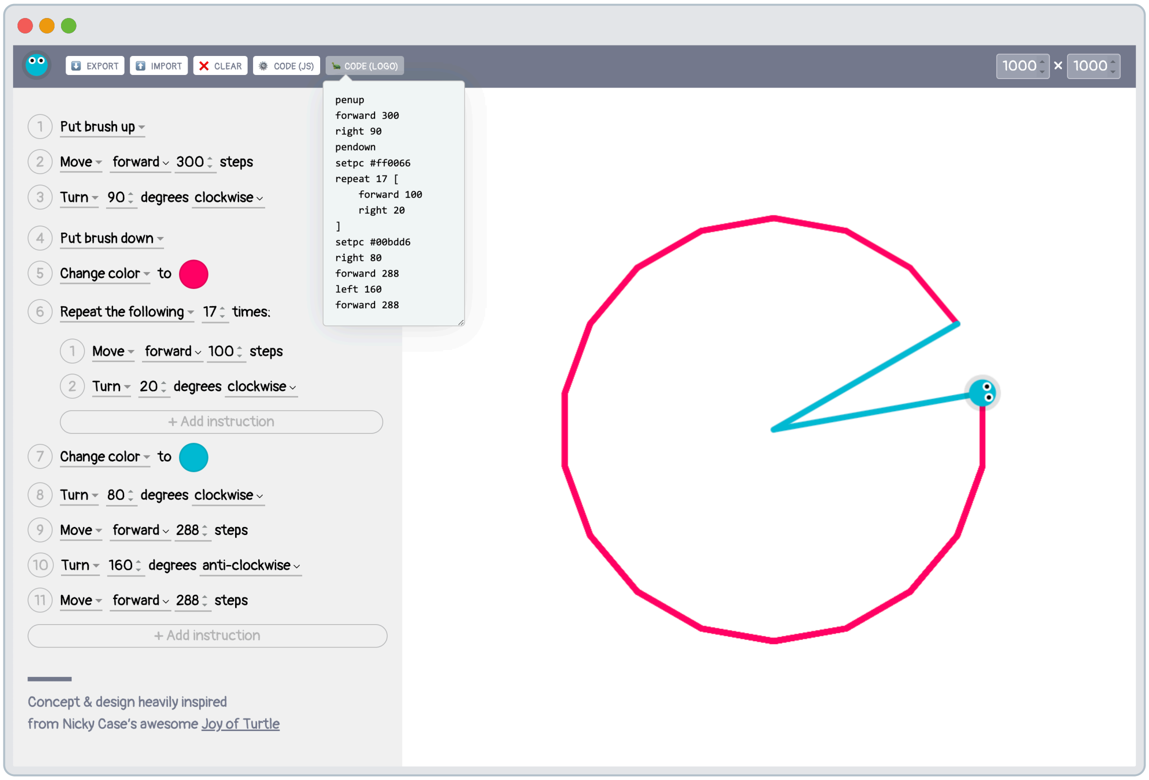
8.2.2 Turtle Graphics

74.1 %

This application fits squarely in the category of apps pushing the boundaries of what Mavo can do to the limit.

It is a visual programming environment for (a subset of) the LOGO programming language [121], a graphic programming language designed for teaching programming to children. It lets the user build a LOGO program by manipulating visual controls, displays a reactive preview on the right, and generates both LOGO code, and JS code (using the Canvas API). The user can not only tweak the parameters of commands, but rearrange them and move them in and out of loops via drag & drop.

It is implemented as a collection of commands (`instruction`) is progressively **generating JavaScript code** through Formula² expressions, and then rendered in an `<iframe>` to reactively draw the graphic on a `<canvas>` element, automatically re-rendering the graphic as the commands change.



1 Put brush up
2 Move forward 300 steps
3 Turn 90 degrees clockwise
4 Put brush down
5 Change color to ●
6 Repeat the following 17 times:
    1 Move forward 100 steps
    2 Turn 20 degrees clockwise
7 Change color to ●
8 Turn 80 degrees clockwise
9 Move forward 288 steps
10 Turn 160 degrees anti-clockwise
11 Move forward 288 steps

```
penup
forward 300
right 90
pendown
setpc #ff0066
repeat 17 [
    forward 100
    right 20
]
setpc #00bdd6
right 80
forward 288
left 160
forward 288
```

Concept & design heavily inspired from Nicky Case's awesome Joy of Turtle

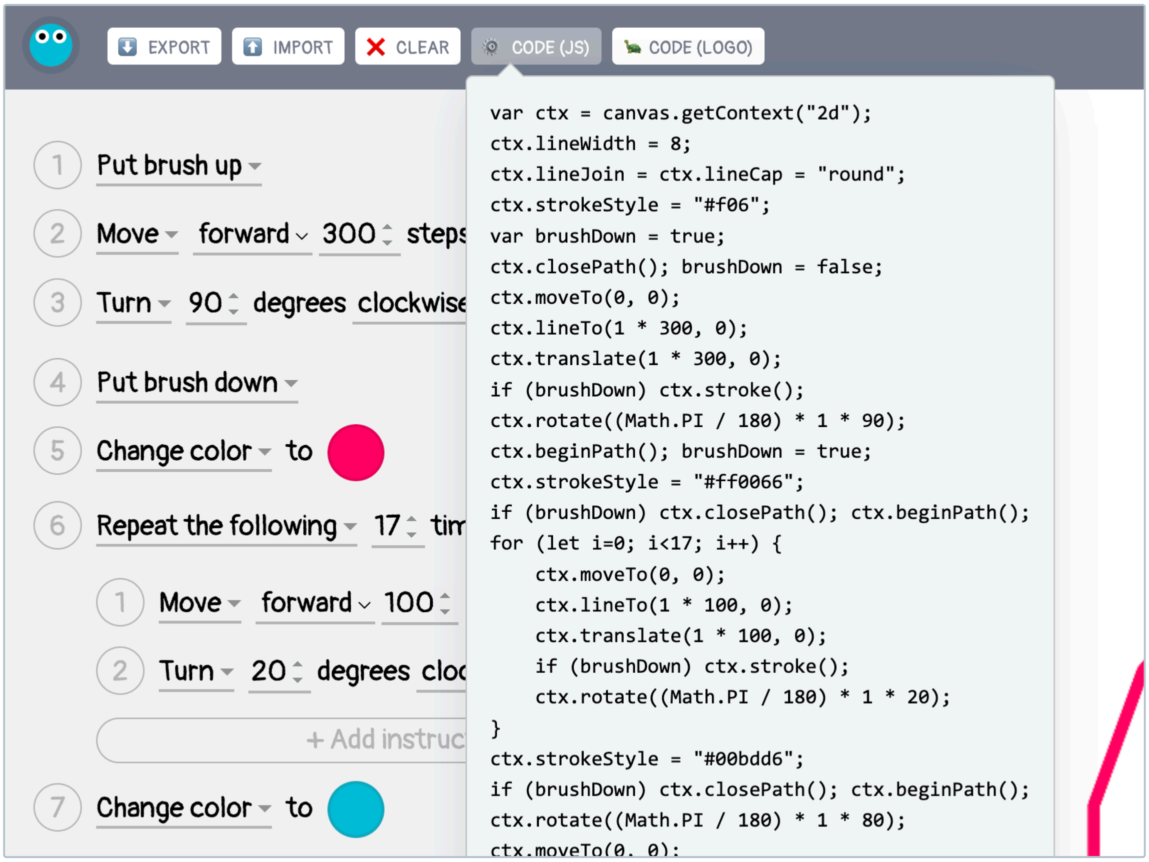
Figure 8.9 *A pure Mavo visual programming environment for (a subset of) the LOGO programming language. The live version can be accessed at mavo.io/demos/turtle. The concept and design was inspired from a demo by Nicky Case at ncase.me/joy-demo/turtle.*

Beyond the novelty of the application itself, it also showcases **recursive collections** via the `mv-like` attribute. Recursive collections are collections that have at least one child using the same template as the parent. A common use case is threaded discussion comments, where each post can have any number of replies as children, which can also have their own replies and so on. In this case `instruction` is a recursive collection, as some commands (loops, conditions) can contain other commands.

It also showcases how **metaprogramming** can work in Mavo HTML, by taking advantage of existing HTML primitives (`<iframe srcdoc>`) that allow rendering documents from arbitrary strings.

The entire application consists of approximately 200 lines of Mavo HTML, and its logic needs no JavaScript (beyond the JS code it *generates*, as described below).

Of the 19 commands in the 1969 design of the LOGO language [121], this application implements 16 (forward, backwards, right, left, pen up, pen down, setpencolor, repeat, home, setxy, set heading, setpensize, penerase, circle, arc, print). The main commands



1 Put brush up
2 Move forward 300 steps
3 Turn 90 degrees clockwise
4 Put brush down
5 Change color to ●
6 Repeat the following 17 tim
  1 Move forward 100
  2 Turn 20 degrees cloc
7 Change color to ●

```
var ctx = canvas.getContext("2d");
ctx.lineWidth = 8;
ctx.lineJoin = ctx.lineCap = "round";
ctx.strokeStyle = "#f06";
var brushDown = true;
ctx.closePath(); brushDown = false;
ctx.moveTo(0, 0);
ctx.lineTo(1 * 300, 0);
ctx.translate(1 * 300, 0);
if (brushDown) ctx.stroke();
ctx.rotate((Math.PI / 180) * 1 * 90);
ctx.beginPath(); brushDown = true;
ctx.strokeStyle = "#ff0066";
if (brushDown) ctx.closePath(); ctx.beginPath();
for (let i=0; i<17; i++) {
    ctx.moveTo(0, 0);
    ctx.lineTo(1 * 100, 0);
    ctx.translate(1 * 100, 0);
    if (brushDown) ctx.stroke();
    ctx.rotate((Math.PI / 180) * 1 * 20);
}
ctx.strokeStyle = "#00bdd6";
if (brushDown) ctx.closePath(); ctx.beginPath();
ctx.rotate((Math.PI / 180) * 1 * 80);
ctx.moveTo(0, 0);
```

Figure 8.10 *The app provides both LOGO and JavaScript code.*

missing are those related to defining procedures (`to ... end` and `output`) and stopping the program (`stop`).

74.5 %

Architecture

This is possibly the *weirdest* Mavo application ever built: The main program being authored is a collection of commands (`instruction`). Each `instruction` item is a heterogeneous collection with a `type` property for the type of command, which determines which other properties are shown.

HTML

```html
<select property="type">
   <option value="move">Move</option>
   <option value="turn">Turn</option>
   <option value="color">Change color</option>
   <option value="up">Put brush up</option>
   <option value="down">Put brush down</option>
```

HTML (continued)

```html
   <option value="repeat">Repeat the following</option>
</select>

<!-- Command parameters -->
```

Each command has `js` and `logo` properties which correspond to fragments of JavaScript and LOGO code respectively for that particular command.

For example, this is the implementation of the **turn** command:

HTML

```html
<label mv-if="type = turn">
   <input type="number" property="angle" value="20" /> degrees

   <select property="direction">
      <option value="1">clockwise</option>
      <option value="-1">anti-clockwise</option>
   </select>

   <meta property="logo" content="[if(direction = 1, 'right', 'left')] [angle]" />
   <meta property="js" content='ctx.rotate((Math.PI / 180) * [direction] * [angle]);'>
</label>
```

This of the **color** command:

HTML

```html
<label mv-if="type = 'color'">
   to <input type="color" property="color" mv-default="[color.$previous or '#ff0066']" />

   <meta property="logo" content="setpencolor [color]" />
   <meta property="js" content='ctx.strokeStyle = "[color]";
      if (brushDown) ctx.closePath(); ctx.beginPath();'>
</label>
```

The **pen up** and **down** commands have no parameters, and thus *only* contain the `logo` and `js` properties:

HTML

```html
<div mv-if="type = 'up'">
   <meta property="logo" content="penup" />
   <meta property="js" content='ctx.closePath(); brushDown = false;'>
</div>

<div mv-if="type = 'down'">
   <meta property="logo" content="pendown" />
   <meta property="js" content='ctx.beginPath(); brushDown = true;'>
</div>
```

The **repeat** command also includes a nested collection of `instruction` items:

HTML

```html
<div mv-if="type = 'repeat'">
   <label>
      <input type="number" property="times" />
      times:
      <meta property="logo" content='repeat [times] ["["]
         [ join(instruction.logo, "\\n") ]
      ["]"] ' />
      <meta property="js" content='for (let i=0; i<[times]; i++) {
         [ join(instruction.js, "\\n") ]
      }'>
   </label>
   <ol>
      <li property="instruction" mv-multiple mv-like="instruction"></li>
   </ol>
</div>
```

The `repeat` command is **recursive**: it can contain any number of commands, including other `repeat` commands. To implement this, we use the `mv-like` attribute to copy the template of the parent collection. Collections with `mv-like` implicitly have `mv-initial-items="0"`, otherwise this would create an infinite loop.

Outside the root `instruction` collection, the values of the `js` and `logo` properties are joined to generate the final code (the `js_before` property contains 5 lines of JavaScript code to set up the canvas and apply default styles):

HTML

```html
<meta property="js_combined" content="[js_before][join(js, '\\n')]" />
<meta property="logo_combined" content="[join(logo, '\\n')]" />
```

These are then displayed in the UI for the user to copy, and for educational purposes. The `js_combined` property is also used to generate the preview via a dynamic `<iframe>`:

HTML

```html
<iframe srcdoc='
<canvas width="[width]" height="[height]" id="canvas"></canvas>
<script>
try { [js_combined] } catch(e) {}
ctx.drawImage(parent.turtle_image, -25, -25, 50, 50);
</script>' frameborder="0"></iframe>
```

Since this is simply an attribute with an expression, the `<iframe>` automatically re-renders the graphic if any of the properties change.

77 %

Limitations

A big limitation of this approach is its **performance**: since an entire webpage (in the `<iframe>`) needs to be re-rendered every time the user makes a single change. Surprisingly, while the user experience is not as smooth as it would be with a custom implementation, it still very usable (at least in a Chrome browser in 2017), with the main issue being some flickering when quickly incrementing numbers.

Another is **lack of parsing**: it would have been useful if this allowed inputting LOGO code directly and displaying the corresponding visual program, rather than requiring it to be composed only via the GUI. This is a common limitation across many Mavo apps — we have seen it before in the SVG path builder example (Section 8.2.1) as well.

Last, **lack of direct manipulation**: the user cannot interact with the graphic in any way. While by design this is a graphic generated by a series of commands, it would be useful to at least be able to pan and zoom via direct manipulation, something the app this is inspired from (Nicky Case's Joy of Turtle Demo) does provide.

8.3 Mavo Games

77.1 %

Even though our driving use cases have been primarily CRUD apps, it was interesting to see how some Mavo authors pushed the boundaries of what types of applications can be built. One such category is **games**, as most games are the polar opposite of CRUD apps, with far more complex logic and interactions.

8.3.1 Memory Game *(by Dmitry Sharabin)*

77.2 %

This is an implementation of the popular game *Concentration* (also known as *Memory* or *Pairs*), where the player has to find matching pairs of cards. Its entire implementation is fewer than 200 lines of Mavo HTML (and no JavaScript), and that includes customizable themes, a timer, a performance rating, and a history of previous gameplays.

This use case argues the point behind the motivation for data actions quite well: the entire game is fully reactive, with only a single (nontrivial) data action which is triggered

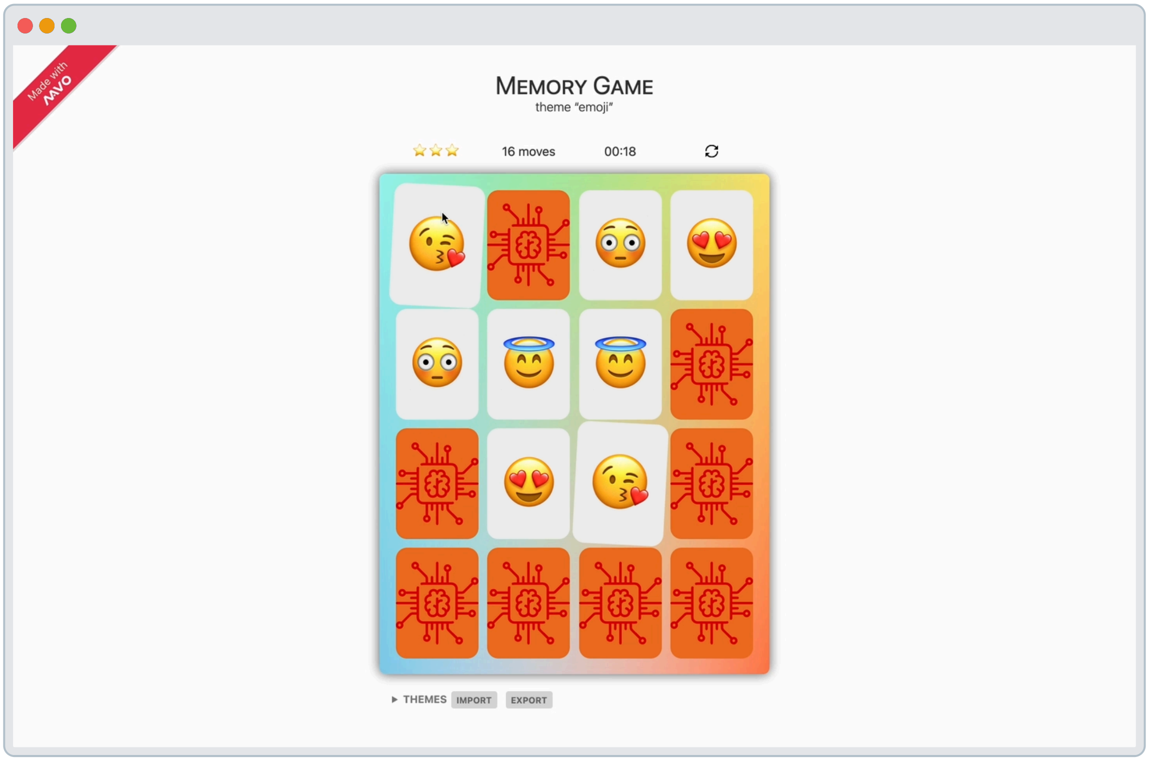


Figure 8.11 *A clone of the popular Memory game built with Mavo. The live version can be accessed at dmitrysharabin.github.io/mavo-memory-game.*

when a card is clicked. Yet, without that one data action, the game would not be possible to implement in Mavo.

77.3 %

Architecture

The application is implemented as six Mavo apps, but architecturally it only needs three, the rest is done for modularity:

- `themes`: Loads data about the available themes from a JSON file (colors, icons, etc.).
- `stats`: History of past gameplays, stored in local storage.
- The rest describes the game state and is broken down into four apps for modularity (`game`, `gameState`, `game-over`, *(unnamed)*), but this state could have been combined into a single app.

The game is implemented as a dynamic collection of cards (`card`), shuffling a list of symbols, with each symbol being present twice, via the formula shuffle(list(symbols, symbols)). While this is a reactive formula, it only gets re-evaluated if symbols changes, which should not happen during gameplay.

CSS classes are used to reflect the state of each card (flipped, matched, incorrect, etc.). The core game logic is implemented by this long data action, which is triggered when a card is clicked:

HTML

```html
<meta property="first_card" />
<meta property="move" content="0" />
<meta property="won" content="[ count(state.$all != 'matched') = 0 ]" />

<ul id="game-board" mv-list mv-value="shuffle(list(symbols, symbols))">
   <li mv-list-item class="card [ if(flipped, 'flipped') ] [state]"
       mv-action="
         // Start game on first move
         if(move = 0, set(game_started, true) & set(start_time, time($now, 'ms'))),

         if(not flipped, set(move, move + 1)),
         set(flipped, true),

         if(first_card,
            if(card.symbol = first_card.symbol, // Match!
               set(state, 'matched') & set(first_card.state, 'matched'),
               set(flipped, false) & set(first_card.flipped, false)
            ) & set(first_card, ''),
            set(first_card, card)
         ),

         if(won, add(group(moves: move, stars: rating, time: timer), stats.attempts))
       ">
      <meta property="state" />
      <meta property="flipped" datatype="boolean" />
      <span property="symbol"></span>
   </li>
</ul>
```

Beyond this, the rest of the game is almost purely reactive.

Limitations

As described in Chapter 6, data actions were envisioned as means to bridge the small gaps between purely reactive applications and the reality of what many CRUD applications needed. The kinds of data actions CRUD applications need rarely exceed one or two function calls, and as we have seen in our user study, novices struggle with sequences of actions.

78 %

The fact that they *can* be used to implement complex game logic is a testament to their flexibility, but using them in this way also reveals their weaknesses, since they were not envisioned as a general purpose programming language.

Namely, the lack of abstractions becomes painful the more complex the logic gets. The large data action above could be a lot easier to comprehend if it could be broken down into smaller, more manageable pieces.

This is an issue we will also see in the next game.

8.3.2 Mavordle (Wordle Clone) *(by Dmitry Sharabin)*

78.1 %

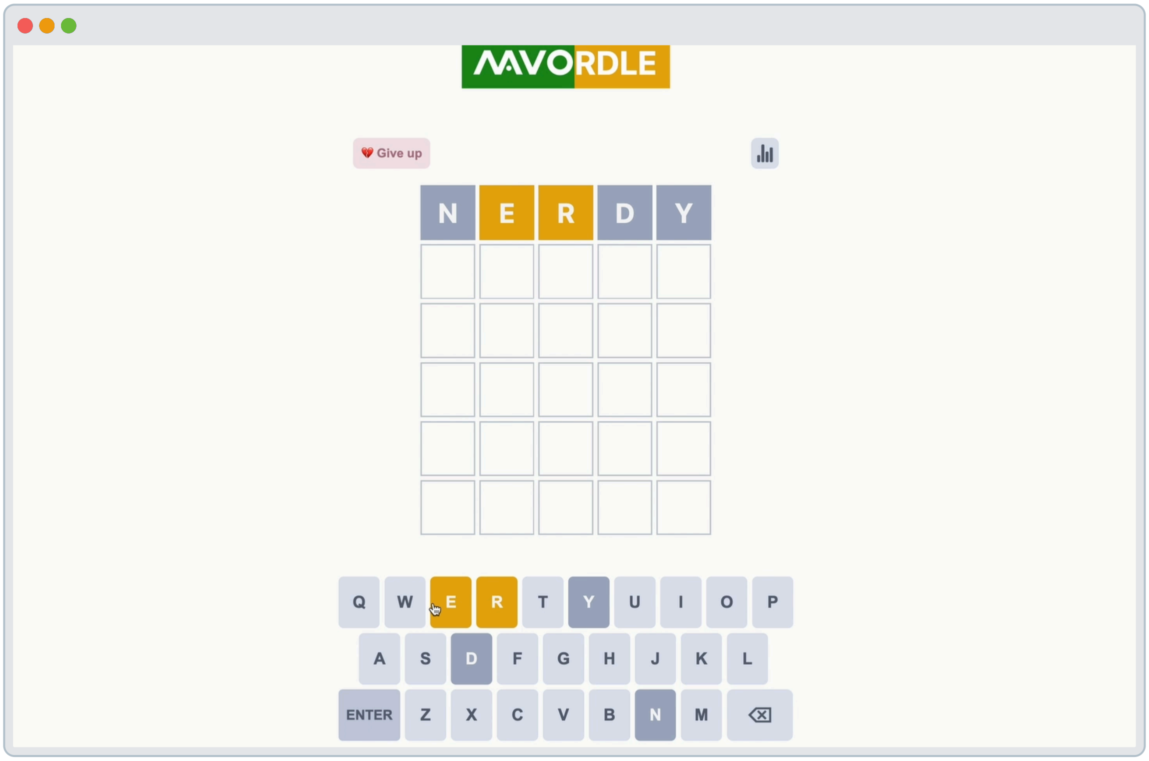


Figure 8.12 *A clone of the popular Wordle game built with Mavo. The live version can be accessed at dmitrysharabin.github.io/mavo-wordle.*

This is a clone of the popular Wordle game and needs fewer than 200 lines of Mavo HTML (and no JavaScript).

Architecture

78.2 %

Mavordle consists of several Mavo apps, each representing a different part of the game:

- Three apps (**`keyboard`**, **`possible_words`**, **`common_words`**) load data from JSON files that is then used by the rest of the code.
- An app for the statistics of past gameplays (**`statistics`**), whose data is stored in local storage.
- The remaining three Mavo apps (**`game`**, **`mode`**, **`popup`**) implement the core game logic and were implemented as separate apps for modularity — in terms of functionality, they could have been combined into a single app.

The gameplay is implemented as data actions for the onscreen keyboard buttons, which update state as needed.

The keyboard is implemented as a collection of letters, each with a data action that appends it to the current guess and a class that styles them based on their status:

HTML

```html
<button mv-list-item class="[status]" style="--row: [row]"
        mv-action="if(guess_length < 5,
         add(key, guess_letters) & add(count(guess_letters) - 1, used_indices)
        )">
   <span property="key"></span>
   <meta property="used_indices" mv-list mv-initial-items="0" />
   <meta property="used" mv-value="contains(join(guesses), key)" />
   <meta property="solution_index" mv-value="index_of(solution, key)" />
   <meta property="status" mv-value="
      if(solution_index = -1, 'absent',
         if(contains(used_indices, solution_index), 'correct', 'elsewhere')
      )" />
</button>
```

Most of the logic is on the Enter key, which has two versions, one for when the guess is incomplete (fewer than five letters), and one for when the guess is complete:

HTML

```html
<button id="enter" style="--row: 3; --column: 1"
        mv-if="0 < guess_length < 5"
        mv-action="toast('Not enough letters')">Enter</button>

<button id="enter" style="--row: 3; --column: 1"
        mv-if="guess_length = 5" mv-action="
   if(guess_valid,
      if(guess = solution, set(result, 'won'), setif(attempt = 6, result, 'lost'))
      & add(guess, guesses) & clear(guess_letters),
      toast('Not in word list')
   ),

   if(result = lost, toast(solution)),

   if(result,
      add(group('guess': if(result = won, count(guesses), 0)), statistics.games)
      & set(statistics.current_streak, if(result = won, statistics.current_streak + 1, 0))
      & set(statistics, group(
```

HTML (continued)

```
         solution: solution,
         guesses: guesses,
         states: used_letters.state,
         result: result,
         max_streak: max(statistics.current_streak, statistics.max_streak),
         date: $today
      )
   )
">Enter</button>
```

79.2 %

Limitations

Gameplay needs to happen via the onscreen keyboard — pressing the corresponding letters on the physical keyboard does not work. This is because Mavo does not provide a way to trigger actions based on key presses. This *could* be added, but it would require a few lines of JavaScript:

JS

```js
document.addEventListener("keyup", event => {
   let key = document.getElementById("key-" + evt.key.toLowerCase());
   if (key) {
      event.preventDefault();
      key.click();
   }
});
```

CHAPTER 9

Lifesheets: End-User Development of Quantified Self Applications

14,147 words (41 min read)

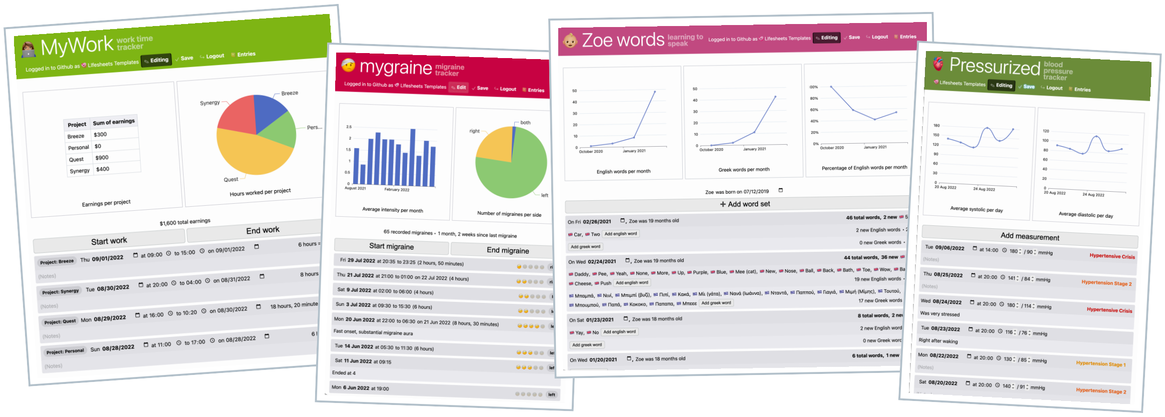


Figure 9.1 *A set of example Lifesheets applications, tracking various aspects of life. From left to right: Work time tracker, migraine tracker, bilingual child vocabulary tracker, blood pressure tracker.*

9.1 Introduction & Background

79.6 %

While Mavo significantly lowers the barrier for developing full-stack data-driven web applications, writing HTML still requires a certain nontrivial level of technical proficiency. Many end-users struggle to sufficiently conform to any type of syntax, even as permissive as HTML's.

One of the advantages of extending HTML syntax, is that any sufficiently generic WYSIWYG HTML editor can be used to create Mavo applications. However, while that would remove the syntactic pitfalls, a visual interface has tremendous potential to make the *concepts* easier to use and understand as well.

Some work in this direction explored adapting a general purpose WYSIWYG editor for HTML to support Mavo with moderately positive results [122].

I hypothesized that perhaps a domain-specific visual application builder could be even more effective, as narrowing down the set of potential use cases also narrows down the design space, and allows for certain assumptions to be made (e.g. about the data model or desired capabilities). This facilitates rapid iteration, and provides increased value per unit of user effort (Section 1.4.2).

Many domains could have been chosen for this, but personal tracking (aka Quantified Self) has several attributes that make it *particularly* well suited for this exploration:

- It has broad appeal to a majority of the population
- The use cases are not abstract but concrete and relatable,
- The data models involved tend to be shallow but not entirely flat,
- Data ownership and privacy are important concerns
- It makes use of all core Mavo primitives: data editing, lightweight computation (Formula^2), Data Update Actions, and unified remote data storage (Madata).

Furthermore, it is yet another domain where user dissatisfaction with available tools is well documented, and validated once more by our needfinding study.

79.8 %

9.1.1 Motivation & Core Contributions

Many people wish to record data about themselves or loved ones. Motivations vary: self-awareness, preservation of memories, comparison to others, self-discipline, and many others [123, 124]. Mainstream applications are not flexible enough to cater to users' diverse needs, and raise concerns about privacy, data ownership, and lock-in to proprietary systems [125–128]. As a result, people regularly resort to spreadsheets, unstructured documents, or even handwriting [129, 130]. Tracking involves high amounts of friction, so data is lost. Often, the process is abandoned altogether [131–133].

We present Lifesheets[1], a domain-specific End-User Programming (EUP) system empowering end-users to create *their own, custom* tracking applications. Lifesheets bridges a gap between generic EUP tools and customizable tracking applications: Being domain-specific allows it to produce higher fidelity applications with less effort than generic no-code tools (low threshold), while being designed as an end-user programming tool rather than a customizable tracker allows it to be more flexible and powerful than existing trackers (high ceiling).

We focus on manual tracking in this initial work; but plan to support semi-automated tracking [134] at a later stage. Manual tracking has been shown to improve awareness and facilitate reflection [135–137], but imposes a high capture burden, which Lifesheets reduces through programmable shortcuts, defaults, and input affordances.

Creating these applications requires no more, and often significantly less, technical skill than using spreadsheets. The resulting applications are far easier to use than spreadsheets and much preferred by our user study subjects: Using a standardized SUS questionnaire [115], participants rated the usability of both creating and using Lifesheets as *vastly* than a spreadsheet.

Lifesheets are **private** and **portable**. Each is represented as a simple Mavo HTML document, interpreted in the browser. Both the app itself, and its tracked data are stored in the user's own storage space on a cloud service such as GitHub or Dropbox. GitHub is used by default, but any Madata backend can be used, as long as it supports the capabilities necessary (`auth`, `write`, `upload`, `host`). Data is private to the user, and there is no dependence on any specific platform: if their chosen backend goes away, they can simply move their files to another cloud service and continue using all their applications. The HTML representation also allows power users to customize each lifesheet further with their own HTML or CSS.

Spreadsheets are an obvious default tool for tracking, thanks to their tabular structure. But we identify several limitations of spreadsheets: the visual layout of entries, laborious data capture, and challenges in authoring suitable calculations—especially over dates and times. We then show how to overcome these limitations through our design of Lifesheets.

Beyond customization and input flexibility, Lifesheets increases efficiency in two ways: both by lowering the capture burden per entry, but also *the time needed to begin tracking a new thing*. As the Greeks say, "*The beginning is half of every action*": if starting requires copious amounts of app store research [125] or tedious application building, it requires very high motivation for tracking to commence. Until then, data is lost.

[1] We refer to the system as *Lifesheets* (capitalized) and to the apps created by it as *lifesheet(s)*

Beyond our initial goal of studying whether end-users can understand Mavo concepts, or Formula^2 expressions, and to gauge their reaction to an application using the Madata approach to data storage, this work also includes several other contributions to the Quantified Self and End-User Programming literature:

1. We provide numerous insights regarding people's tracking habits through a survey of 85 participants. A focus is parents tracking data about their child(ren)'s development, a type of personal tracking underexplored in the Personal Informatics literature.
2. We identify numerous obstacles to the use of existing tools for personal tracking
3. We design and implement Lifesheets, a serverless application builder aimed at overcoming these obstacles
4. We evaluate Lifesheets for flexibility by implementing dozens of use cases collected via our needfinding survey, and for learnability and efficiency through a lab study of 10 participants.
5. We outline a machine-readable description of tracking applications and their tracked data. Using these, other interoperable systems could allow users to further explore and correlate their data.

The Greeks say "The beginning is half of every action". If beginning to track requires copious amounts of app store research or tedious application building, it requires very high motivation for tracking to commence. Until then, data is lost. While it is established in the literature that efficiency in *using* a tracking application is important for users to stick with it [123, 124], we hypothesized that lowering the barrier for *beginning* the tracking process is equally or more important. If tracking does not commence, there is nothing to stick *to*. Our needfinding survey confirmed this.

We envision a future where creating a tailored, efficient, personalized tracker that evolves together with the data and adapts to life or goal changes, yet is as high fidelity as commercial applications, can be as easy as creating a document or spreadsheet. In this vision of the future, marginalized groups can take tracking their data into their own hands, rather than depending on third parties who don't understand them [138] and don't necessarily have their best interests in mind.

9.2 Related Work

80.5 %

To our knowledge, our work is the first that bridges these two main research areas: (1) Personal Informatics, and (2) End-User Programming.

9.2.1 Personal Informatics

Needfinding

Several studies have explored user needs related to personal tracking. Recurring themes were input flexibility [129, 130, 139–141], style/layout customization [127, 129, 130, 139, 140, 142], and privacy [125–128, 130].

Co-design has been used to explore tracking needs, mainly within narrow domains such as food tracking [140] or specific subsets of users, such as youth [127]. This body of work provides additional motivation, as it demonstrates that even within these narrow scopes, user needs are still widely varied. However, this body of work does not answer the question of how these varied tracking applications would be built.

Pain Points

80.7 %

Previous work has found several usability issues and other barriers with existing tracking tools [123, 126, 138]. Relevant to manual tracking are: (1) low efficiency and usability of data entry [123, 126, 130], (2) data reflection not meeting user needs [123, 126], (3) lack of input flexibility [140], (4) insufficient customization [130], (5) inconvenience of data export [126], (6) privacy concerns [125–128], (7) biases inherent in the design of widely available tools that hamper suitability [138, 143], (8) difficulty in finding a suitable application [125]. We will see that Lifesheets effectively addresses all these issues.

Ultra-Customizable Trackers

80.8 %

Some attempts to address these issues resulted in highly customizable tracking applications. The only general purpose research system (to our knowledge) is OmniTrack [141]. Other research systems that focus on customizability, such as [128, 139] focus on very narrow tracking domains or use cases (e.g. multiple sclerosis care).

Beyond research systems, there is a variety of commercial applications that attempt to solve this problem, such as Bearable [144], Exist.io [145], DoEntry [146], Nomie [147], or KeepTrack [148].

All of these systems are essentially highly configurable applications, not empowering users to *create standalone applications*. They require the system that created them to work; they cannot function independently of it, nor outlive it.

There is no computation, nor any extension points to extend the ceiling. Settings accept predefined static values, not formulas.

Parental Tracking 80.9 %

The subject of personal tracking is not necessarily oneself, but often other loved ones; Parental record-keeping is a subset of personal informatics that is often studied separately. It is further broken down into safety monitoring through sensors [149, 150], and manual record-keeping to preserve memories, detect developmental delays, or as requested by a pediatrician [151–154].

Although parental record-keeping shares a lot with self-tracking, these use cases are distinct in several ways. One big difference is posterity: data needs to be preserved for decades, making data ownership and portability critical.

9.2.2 End-user Programming (EUP) Systems 81 %

There is a large body of work around empowering end-users to create their own read-write data-driven applications, discussed at length in Chapter 2. It could be argued that self-tracking applications are merely a special case that existing EUP tools can already accommodate. Our work adds to the small body of work around *domain-specific EUP systems* (e.g. [155–157]) and demonstrates that there are enough commonalities between the diverse set of manual tracking use cases — even after including those outside the common health & wellness domain — that a domain-specific EUP system can significantly smoothen the ease-of-use vs. power curve. We identify significant commonalities in personal tracking use cases that allow us to *specialize* end-user programming to the personal tracking task; this specialization makes Lifesheets much simpler and more efficient than a general purpose no-code tool. It also lets us create apps that are highly optimized for efficient personal tracking data entry, which is critical. If entering data takes too long, users are less likely to stick with it [124].

Commercial No-code Tools 81.2 %

In addition to the research literature, we also investigated several popular commercial no-code tools such as Airtable [158], Glide [159], and Coda [31] to compare them with Lifesheets. These tools utilize a spreadsheet-like interface and improve on many spreadsheet issues (e.g. named references, chart chooser interfaces, simple default values). However, they still fall short in many areas that are relevant to personal tracking (namely temporal calculations and charting, non-tabular schemas, reactive defaults, actions for automation).

Temporal calculations are easier in some of these tools, but still challenging. Many non-tabular schemas are still hard (e.g. global variables, multivalued properties).

81.3 %

Mavo

The Mavo suite of technologies is particularly well-suited to tracking: Managing *collections* of data (just like tracking entries) is one of its primitives. It facilitates UI customization and reactive defaults for efficient data entry. Its expression language, Formula[2], is novice-friendly by design and well-suited to data aggregation and temporal calculations. Thanks to Madata, data can be stored independently of the application, in a portable format, and its actions can create data entry shortcuts with a one line expression, reducing capture burden.

However, creating a custom tracker with fidelity comparable to commercial applications still requires a sophisticated understanding of Mavo itself, HTML, CSS, not to mention user interface design skills. Lifesheets wraps Mavo and exposes a higher level abstraction, making it approachable to users who are far less technical than its original target audience.

The Lifesheets GUI does expose several lightly abstracted Mavo concepts to end-users (e.g. expressions, properties, simple collections, actions, data and logic separation). This highlights another contribution of this work: it is one of the first to study exposing Mavo concepts to end-users who cannot necessarily write HTML.

Additionally, the generated Mavo HTML is visible and editable in various places in the Lifesheets GUI as an escape hatch for customization, to further extend the ceiling, while maintaining the low threshold of a GUI (see Section 1.4.1). This smoothens out a common usability cliff of no-code tools: once use cases outgrow the GUI, users are typically directed to scripting languages, which are much harder to learn [11].

9.3 Needfinding Survey Summary

81.5 %

We began our work with a survey of 85 participants, recruited through a call for participation on social media and local parent groups in July 2022. The goals of the survey were threefold: (a) collect outlines of tracking use cases (tracked data schemas, reflection, etc.) in bulk, to be used for evaluating the flexibility of Lifesheets, (b) recruit participants for our later user study, (c) collect data to guide the design (complementing existing needfinding literature),

While needfinding is not a primary focus of this work, there are a few novel aspects of this survey compared to existing needfinding work like [125, 129, 130, 142]: **(1)** Participants in similar studies often forget what they track [140]. To mitigate this, we used a modified version of web probing [160]: we primed participants with a list of 26 (+13 more for parents) common tracking cases, that they could complete in with their own custom cases, using 5 (+4 more for parental tracking) open text fields — a format shown to increase recall [161, 162]). **(2)** Most existing work studies things actually tracked; we also study things that people *want to track, but don't*. **(3)** We compare self-tracking and parental record-keeping across *the same population*. We intentionally did not select based on parental status, nor made any special mention of it in our recruitment materials to study it as a special case of personal tracking of self-trackers who also *happen* to be parents.

To keep the focus on the main contributions, we have placed the bulk of background details and survey findings in the appendix (Appendix A). Here, we focus on observations that specifically demonstrate the general need for programmable end-user tracking apps, surface particular requirements that our design should meet, or relate to the novel aspects of this survey mentioned above.

9.3.1 Overview

81.8 %

Participants tracked a median of 7 things for themselves (no difference by parental status), but parents *additionally* tracked a median of 4 things about their children. While automatic tracking (e.g. step count) was common, more than half of self-tracking use cases and nearly all parental tracking was by manual entry (indicating perhaps that despite the rise of "baby wearables"[149, 150], parental tracking largely remains a manual labor of love).

Popularity of tracking use cases largely validates existing results, but our methodology uncovered a few common use cases that are not present in existing literature. Most notably, sexual activity is the 9th most popular tracking use case (being tracked by 22.35% of participants — equally across genders), despite barely appearing in other studies.

9.3.2 Use Case Details

49/85 participants provided additional details on their (manual) tracking methods — a total of 217 tracking instances, 168 of which contained enough details to be included in the Tracking Use Cases dataset that was later used to evaluate Lifesheets for flexibility. 36/49 used a generic tool (handwriting, spreadsheets or note app) for at least one use case. Only half of the things tracked (37.5% for parental tracking) were tracked using a dedicated (web) application. The rest were mostly tracked in spreadsheets, and digital or paper documents.

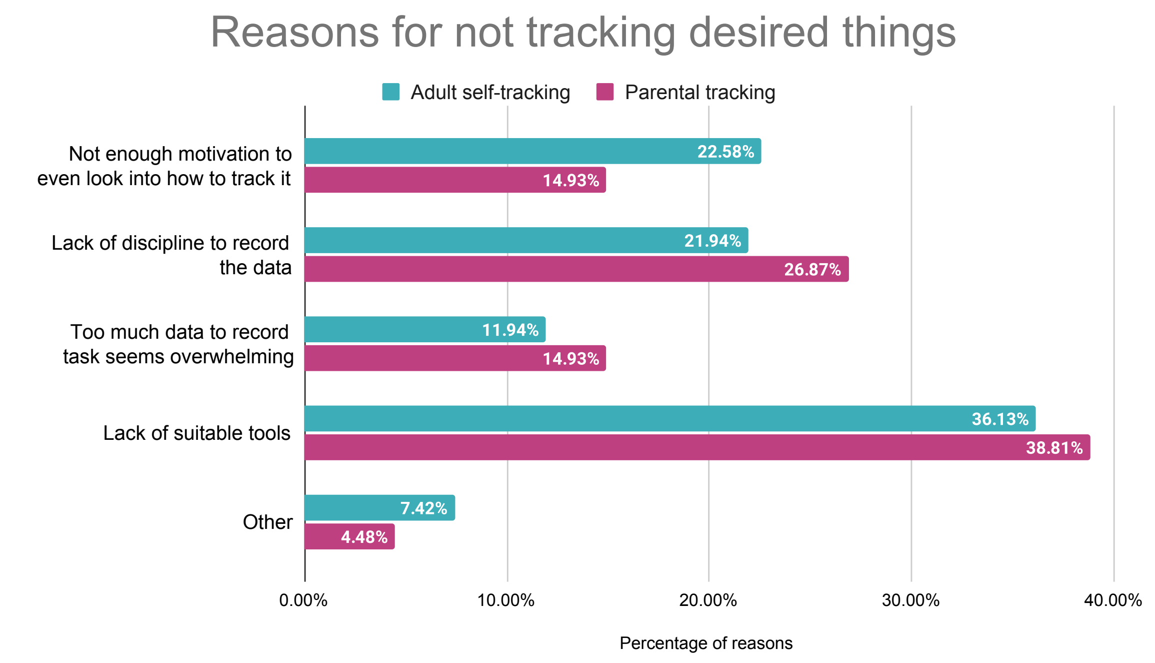


Figure 9.2 *Reasons given for not tracking the things participants have wanted to track, broken down by self-tracking use cases, and parental tracking use cases.*

Lack of suitable tools was by far the most common reason for not tracking desired things (Figure 9.2), with 33/49 participants reporting this as a reason that they don't track things they want to track.

9.3.3 Satisfaction and Complaints

Participants rated the self-tracking tools they use as satisfactory in 64.1% of cases and as unsatisfactory in 18.3%. Tools for parental tracking were associated with lower satisfaction (52.6%), with 18.4% of tools rated as unsatisfactory.

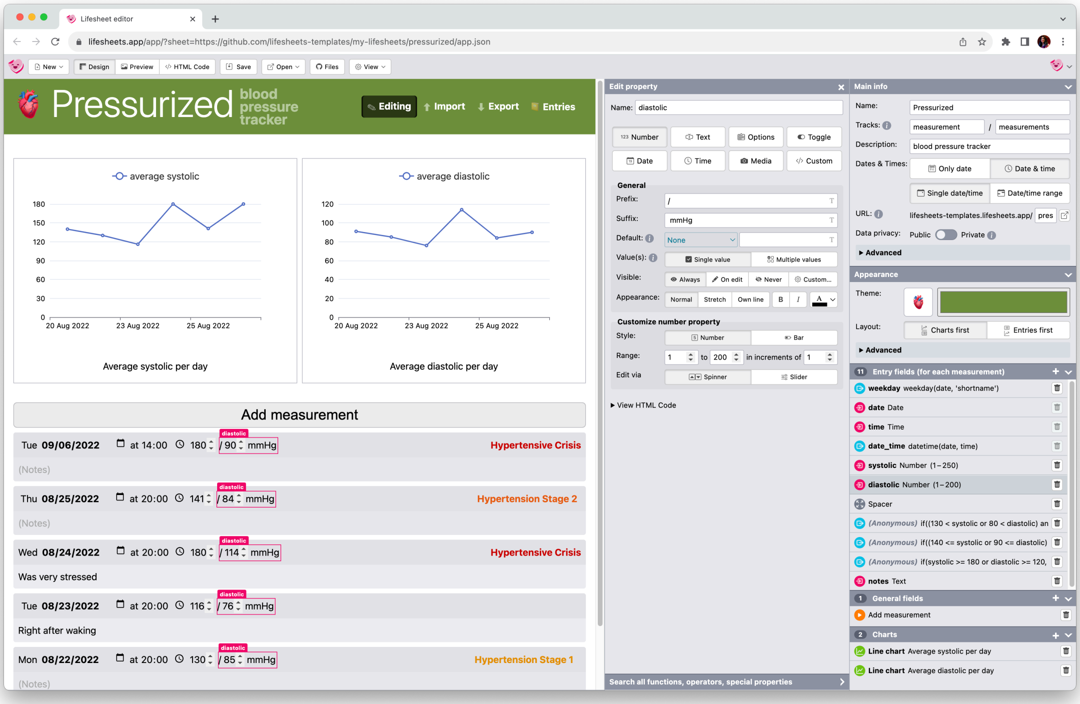
Out of 76 complaints people had with the tools they were using, data entry efficiency was the most common (17 complaints). P47 wants shortcuts and presets: *"having presets for exercise types could be useful to make adding records faster, limiting clicks, which would in turn make it less of a hassle to record."* P71 finds using a spreadsheet tedious, even with data validation: *"It's tedious to enter data, especially on a phone. I have to enter the date manually, no defaults. Even in the fields where I have dropdowns it's still tedious to fill them in."*.

The second most common issue was mismatch between the tracked data schema and their needs (10 complaints). Notably, P81 went into detail on how they use their menstrual tracker to track sexual activity: '*'In addition to what I hate about Flo in general, it's not really a sex tracker, so I have to encode what I actually want to record into its only two modes for recording sex: protected and unprotected. So I have the convention that if penetration happened I log it as "unprotected" otherwise I log it as "protected". But in actuality, both are protected, because I have an IUD! I'd also like to log things like how good it was, whether we both had an orgasm etc."*

Other common complaints were insufficient automation (9), the tool does not calculate desired insights (8), no reminders (5), lack of structured data (4), lack of customization (4), and privacy concerns (4). P23 stopped tracking details about their period due to privacy concerns: *"One day I had an out of ordinary heavy flow which I tracked in the app. Later that day I saw a targeted ad that I'd never seen before (or since) so I concluded my data had been shared immediately. Since then I've only entered start date and have not given detailed info to that app. An app selling such personal data was an affront."*

9.4 Lifesheets

82.2 %



lifesheets.app/app/?sheet=https://github.com/lifesheets-templates/my-lifesheets/pressurized/app.json

Tue **09/06/2022** at 14:00 180 diastolic / 90 mmHg **Hypertensive Crisis**
(Notes)

Thu **08/25/2022** at 20:00 141 diastolic / 84 mmHg **Hypertension Stage 2**
(Notes)

Wed **08/24/2022** at 20:00 180 diastolic / 114 mmHg **Hypertensive Crisis**
Was very stressed

Tue **08/23/2022** at 20:00 116 diastolic / 76 mmHg
Right after waking

Mon **08/22/2022** at 20:00 130 diastolic / 85 mmHg **Hypertension Stage 1**
(Notes)

Edit property
Name: diastolic

General
Prefix: /
Suffix: mmHg
Default: None
Value(s): ☑ Single value | Multiple values
Visible: Always | On edit | Never | Custom...
Appearance: Normal | Stretch | Own line | B | I | A

Customize number property
Style: Number | Bar
Range: 1 to 200 in increments of 1
Edit via: Spinner | Slider

Main info
Name: Pressurized
Tracks: measurement / measurements
Description: blood pressure tracker
Dates & Times: Only date | Date & time
Single date/time | Date/time range
URL: lifesheets-templates.lifesheets.app/ pres
Data privacy: Public Private

Appearance
Theme:
Layout: Charts first | Entries first

11 Entry fields (for each measurement)
weekday weekday(date, 'shortname')
date Date
time Time
date_time datetime(date, time)
systolic Number (1–250)
diastolic Number (1–200)
Spacer
(Anonymous) if((130 < systolic or 80 < diastolic) an
(Anonymous) if((140 <= systolic or 90 <= diastolic)
(Anonymous) if(systolic >= 180 or diastolic >= 120,
notes Text

1 General fields
Add measurement

2 Charts
Line chart Average systolic per day
Line chart Average diastolic per day

Figure 9.3 *The Lifesheets editor, in Design view, with a property selected (diastolic), shown in a browser window. Temporal and data privacy settings are found in the "Main info" panel on the top right.*

Motivated and informed by our preliminary study and the needs and issues identified in the literature, we designed Lifesheets (Figure 9.3). In this section, we describe Lifesheets' novel architecture and its main concepts. A functional prototype can be found online at lifesheets.app.

9.4.1 Architecture

82.3 %

Lifesheets introduces a novel "serverless" end-user programming ecosystem in which the application is built by the Lifesheets editor but stored in the user's own cloud storage space. Our prototype only supports GitHub, but other services are straightforward: all that is needed is file storage, static file hosting, and a Web API to control both programmatically. We started from GitHub as it meets all three requirements, simplifying the process. However, it is entirely possible in the future to support using separate services for file storage and web hosting, which would expand the range of services that can be used.

A major advantage of this type of decentralized design is **increased privacy and portability**: In contrast to the current ecosystem, in which users need to trust each tracking application individually to preserve their privacy, our approach minimizes the trust surface to a single cloud storage provider. Cloud storage providers typically build their reputation on security and privacy, but if trust becomes compromised, copying files to another provider is all it takes to migrate. Although the benefits of this architecture extend beyond tracking applications, they are especially important for tracking, as data preservation is a major need — and the main motivation for parental tracking according to our survey, comprising almost 40% of provided reasons (Chapter 9). In comparison, all other systems we reviewed store data opaquely in proprietary systems that can disappear at any time.

82.5 %

9.4.2 Anatomy of a Lifesheet

Lifesheets are standalone web applications consisting of a single `index.html` file, and a `style.css` file used for styling. They are Mavo [11] applications, and thus need no custom JavaScript code to function. They work on a phone, tablet or computer, and their UI adapts to available space. These web applications are PWAs[2] and thus can be installed on a phone just like a native application. Their code (HTML and CSS) is generated with readability in mind, to facilitate both learning Mavo, and tinkerability by power users.

Each lifesheet includes its own authentication UI (provided by Madata), which is separate from that of the Lifesheets editor. Lifesheets have two data privacy modes: Public and Private. In both cases, only the owner (and anyone they authorize) can edit data, but in the latter, only the owner can *read* it as well.

Data tracked with a lifesheet is stored separately from the application, in a cloud location of the user's choice, or locally in the browser. By default, this is a JSON file in the same directory as the web application (not deployed to the web if the lifesheet is private). The data location is exposed on the Lifesheet editor UI, so that users know where to find their stored data at any given point.

In fact, Lifesheets is the first GUI application to use Madata (Chapter 5) in an end-user facing way, and also the first to offer user-choice on two levels: the location of the *application itself*, and the location of the *tracked data*, which can be different. This is a markedly different model than the more common "data export" feature: this data file is **not an export, but the primary source of truth**. Power users could even write custom

[2] en.wikipedia.org/wiki/Progressive_web_app

tools to read it or even modify it, and their lifesheet would update to incorporate their edits.

9.4.3 Data Model

The data model of each lifesheet consists of an array of objects called `entries`, containing the main tracked data. Each entry object includes temporal properties (discussed below), as well as arbitrary tracked data. There may also be root-level data, useful for settings, constants etc.

9.4.4 Fields

Fields are the basic building blocks of a lifesheet.

There are four kinds of fields, with distinct purposes:

1. **Properties** hold stored data and are editable. There are several types of property fields, providing affordances commonly needed in tracking applications [141] (text, number, options, toggles, media, date, time), as well as *custom*: an arbitrary fragment of Mavo HTML.

   Each property can be single-valued (most common), or a *collection*, with controls to add, delete, and reorder items (for an example, see the list of words in each entry of Figure 9.10).
2. **Expressions** display reactive calculations, akin to spreadsheet formulas.
3. **Actions** are buttons that can automate data modifications, e.g. add entries with pre-filled values, set values on certain entries, and/or delete certain entries or elements of any named collection. Data can be static values (e.g. `2`) or dynamic expressions (e.g. `time($now)`). Every lifesheet starts with one general action: the button that adds new entries.
4. **Spacers** are no-op fields that facilitate layout.

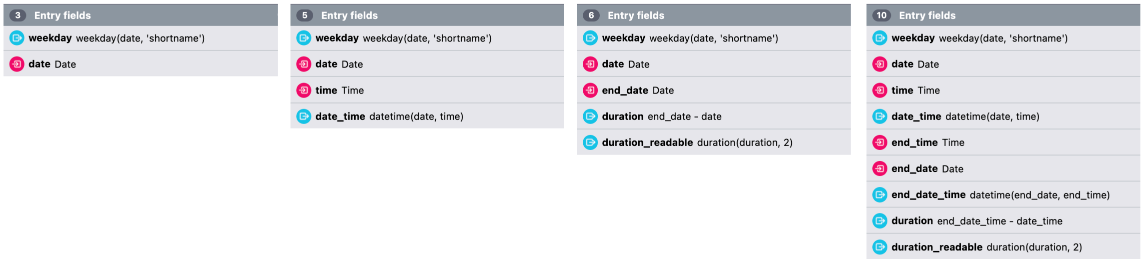
Properties and (optionally) expressions are named so so they could be referred in expressions and other places of the editor interface.

Fields may belong to entries and be repeated with them (*Entry fields*) or to the lifesheet itself (*General fields*). While most properties are Entry fields, General properties can be useful for global parameters (e.g. hourly rate, child's name and birthday, currency, etc.). Entry expressions are useful for reflection on the data of that particular entry, or how it relates to the previous entry (e.g. time since previous migraine), while general expressions are useful for aggregate calculations or calculations on a specific type of entry (e.g. first/last).

9.4.5 Coping with Time

83.1 %

One of the major ways that Lifesheets simplifies tracker construction is its abstractions over temporal data handling, which (as our user study shows) is still difficult with most general-purpose EUP tools.



3 Entry fields
weekday weekday(date, 'shortname')
date Date

5 Entry fields
weekday weekday(date, 'shortname')
date Date
time Time
date_time datetime(date, time)

6 Entry fields
weekday weekday(date, 'shortname')
date Date
end_date Date
duration end_date - date
duration_readable duration(duration, 2)

10 Entry fields
weekday weekday(date, 'shortname')
date Date
time Time
date_time datetime(date, time)
end_time Time
end_date Date
end_date_time datetime(end_date, end_time)
duration end_date_time - date_time
duration_readable duration(duration, 2)

Figure 9.4 *The predefined temporal fields generated with the selection of each temporal category. From left to right: (a) Single Date (b) Single date & time (c) Range of dates (d) Range of dates & times*

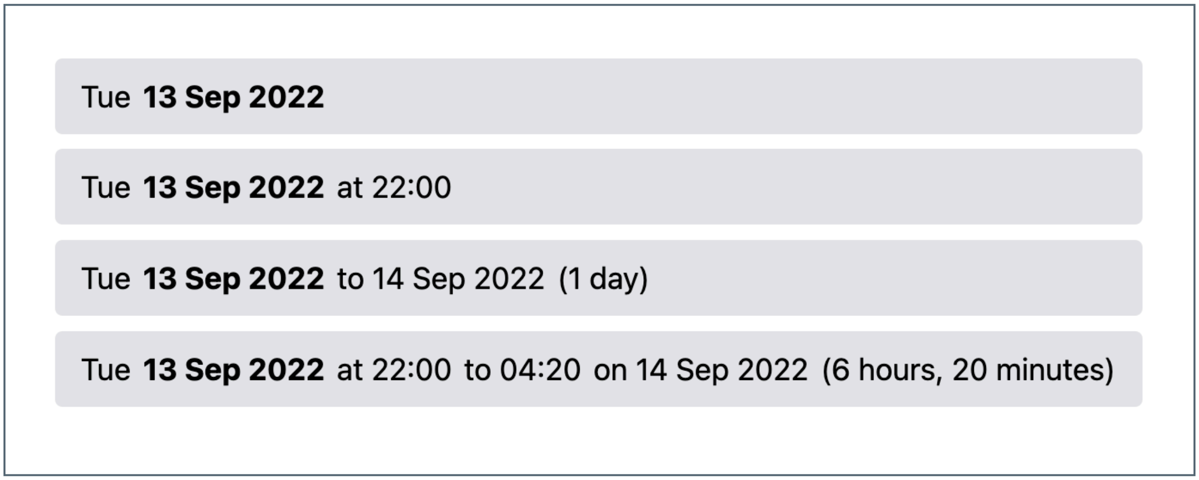
Types of Time

83.2 %

Nearly all tracking use cases involve tracking temporal data alongside other variables (the few that don't are out of scope for this work). However, not all use cases require the same type or granularity of temporal data. We identified four main categories, depending on (a) whether times are needed and (b) whether entries have duration.

- **Single date** Each entry includes only a date. These are used for events where time is not relevant, such as very infrequent events. Examples: Child milestones, Medical exam results, Monthly injections, Expenses.
- **Single date & time** Used for events that may be tracked multiple times a day but have no duration (or it is not relevant). Examples: Cigarettes smoked, Basal body temperature.
- **Range of dates** Examples: Travel.
- **Range of dates & times** Examples: Sleep, Nursing sessions, Migraines.

Lifesheets supports all four, through high-level Date & Time settings (Figure 9.3). Choosing one of these categories automatically creates the right fields (Figure 9.4, which include input affordances, computation, dynamic defaults (Figure 9.6) etc. Here are some example entries for each category with no other fields:



Tue **13 Sep 2022**

Tue **13 Sep 2022** at 22:00

Tue **13 Sep 2022** to 14 Sep 2022 (1 day)

Tue **13 Sep 2022** at 22:00 to 04:20 on 14 Sep 2022 (6 hours, 20 minutes)

The same use case could belong to a different temporal category depending on user needs:

- Tracking blood sugar would fall under "Single date" for most people who check it once or twice a year, but would be "Date & Time" for diabetes patients that measure it several times a day.[3].
- Menstruation may fall under "Range of dates" for people who only wish to record start and end dates, but could also be "Single date" for people who wish to record flow or other symptoms by day (in that case, expressions would calculate what the start and end is, or it could be marked explicitly).

83.5 %

Other Predefined Fields

To facilitate efficient tracker creation, in addition to the temporal predefined fields discussed above, every lifesheet also begins with:

- A `notes` multiline text property (Entry field), to capture freeform metadata about the entry. Formatting can be optionally enabled (through Markdown or a WYSIWYG editor), and the field can be deleted if notes are not desirable.
- An action (General field) for adding new entries. This can also be optionally deleted and replaced with more specific actions (e.g. that prefill properties with certain values, see Figure 9.12).

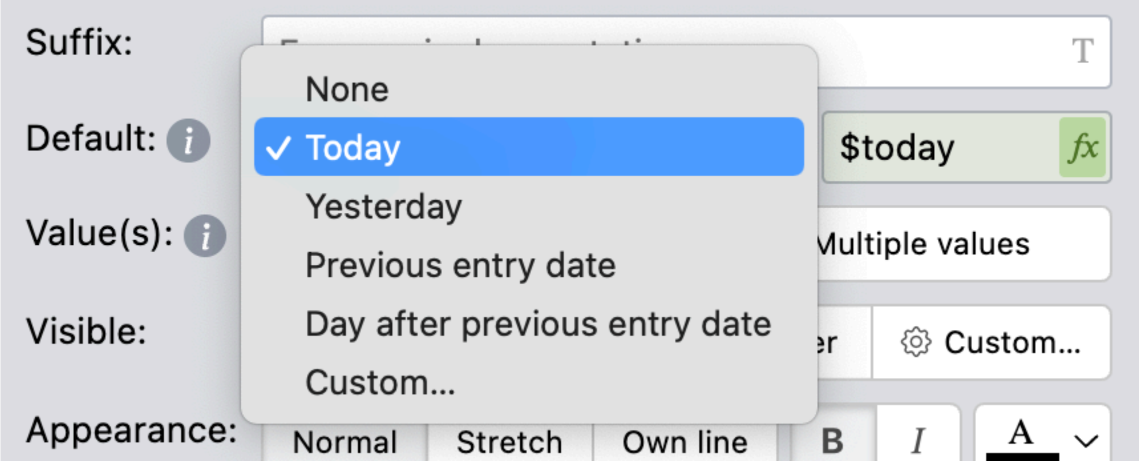
[3] mayoclinic.org/diseases-conditions/diabetes/in-depth/blood-sugar/art-20046628

This also means that for the simplest of tracking cases which are basically a diary with temporal info, a blank lifesheet, possibly with 1-2 clicks to set the temporal category, is all that is needed.

83.6 %

9.4.6 Static & Dynamic Default Values

Default values play an important role in tracking applications, as they can often save significant time in data entry. Lifesheets takes advantage of Mavo's reactive defaults (Section 3.3.6) to provide *"smart"* defaults that are computed based on other properties in the same entry, or even across entries, and update reactively as the data changes.



Default: $today

Figure 9.6 *Temporal properties come with several suitable defaults to facilitate efficient data entry.*

But it goes one step further than simply exposing `mv-default` via a GUI, and provides *default default values*, i.e. presets for commonly needed defaults, in line with our design goal that common use cases should not require typing expressions manually.

All properties include certain presets (e.g. value of the same field in the previous or next entry). In addition to their "default default" values, temporal fields come with additional default value presets such as *"day after"*, or *"start of the hour"*, to facilitate common tracking use cases. (Figure 9.6)).

Moreover, these presets serve double duty: the expressions that generate each default are displayed next ot it to facilitate tinkering and to gradually teach users Formula^2 syntax.

83.8 %

9.4.7 Style Customization

We saw earlier that style customization is one of the most common needs around tracking. Yet neither dedicated tracking applications nor most EUP tools afford much in the way of style customization.

In designing UIs for style customization, there is always the tension between simplicity and power. Too much control can give users too much rope to hang themselves, resulting in UI clutter. Too little and users are unhappy with the result. We incorporated a few basic settings to personalize each lifesheet (Figure 9.3):

Color

While simple, they allow for producing lifesheets that look fairly diverse (Figure 9.1, Figure 9.12, Figure 9.10, Figure 9.14). We also allow custom CSS for more extensive personalization — originally by power users, but we could easily see an ecosystem of lifesheet themes that non technical users can adopt emerging.

83.9 %

Layout

In terms of layout, custom applications include optimized layouts that take better advantage of space, and making each entry easier to process, while spreadsheets and most data-focused *"no-code"* tools are still bound by the limitations of the classic rectangular grid of spreadsheets, or close to it.

A major limitation of spreadsheets compared to custom applications is layout. The rectangular grid rules. Grids facilitate vertical scanning of property values across entries, but have several issues when used for arbitrary data. Each "record" generally occupies only a single row, which for large records might not even fit on the screen. Long values stretch the entire row and/or column. Sparingly used optional fields still occupy an entire column. If users manually violate the grid (e.g. using two rows per record, merging cells, etc.), formula evaluation becomes much harder, and navigation between records functions poorly. In contrast, custom applications often apply far more variability in their layout, taking better advantage of space, highlighting relationships between different fields of a record, and overall making the record easier to understand.

We wanted to bring some of that flexibility to the design of Lifesheets. But in designing any visual application builder, there is always tension around how much control to provide on the resulting layout. Too much control can give users too much rope to hang themselves, resulting in UI clutter. Too little, and users are unhappy with the result.

When laying out a collection of records, there is always the tension between horizontal (grid-based) vs vertical (top-down, “cards”) layouts. Top-down layouts work really well for longform data, but waste space for shorter data. Grid-based layouts have the issues discussed above. It appears that the optimal solution may be a hybrid form [32] where certain fields are laid out left to right, and others top-down. Furthermore, we wanted to support fields that could be next to each other and read as one logical unit, but also a more tabular layout, for use cases that require it.

The solution we chose was to support three appearance settings: “Normal”, “Stretch”, or “Own line” which could be applied on any field, as well as size constraints (min/max width and height). “Normal” (the default appearance) fields are sized based on their contents, which allows them to flow next to each other, and be perceived as one logical unit. “Own line” fields occupy an entire line, which is suitable for longform content. After allocating minimum required space on each line, any remaining space on the same line is distributed equally to fields with an Appearance setting of “Stretch”. “Stretch” is the default appearance for *Spacer Fields* whose most common use is to align fields on the two sides of the same line. Using only these three appearance settings and size constraints, users can create layouts that range from tabular, top to bottom, or anything in between.

9.4.8 Charts

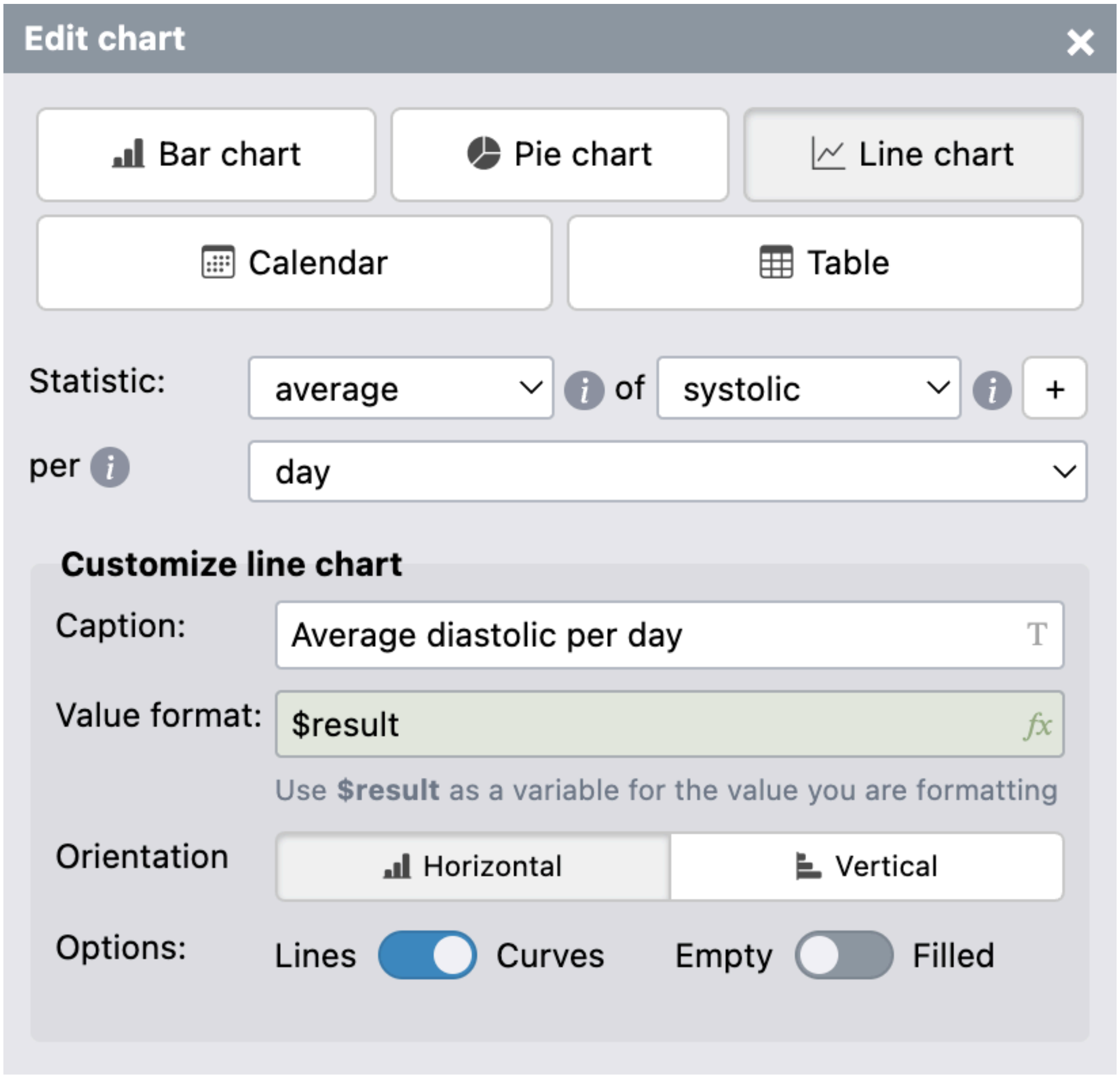
84.2 %

Charts and summary tables are an essential part of the *Reflection stage* of any personal tracking activity [123]. When designing Lifesheets we looked at the kinds of charts and tables existing tracking applications provided, and designed a minimalistic *chart chooser* [163] driven by them.

Lifesheets charts are generated based on three parameters: the field we are plotting (properties or named expressions), how to group it (either by temporal factors such as day or month, or distinct values of another named field), and how to combine values within the same group (e.g. count, sum, average, etc). Multiple aggregates over the same parameter can co-exist in the same chart.

To facilitate early detection of slips and aid the user rapidly converge to the desired goal, the default chart type is *“table”*. This forces users to verify they have the correct data before moving on to a more complex presentation such as a bar chart or a pie chart.

To support including dynamic charts in Mavo applications (even outside Lifesheets), we implemented an `<h-chart>` web component[4], which accepts the data to be charted as a child `<table>` element and the plotting parameters as HTML attributes.



Statistic: average of systolic

per day

Caption: Average diastolic per day

Value format: $result

Use **$result** as a variable for the value you are formatting

Orientation: Horizontal | Vertical

Options: Lines ● Curves   Empty ○ Filled

Figure 9.7 *The chart chooser interface (not shown: common controls for dimensions, visibility, appearance)*

84.5 %

9.4.9 Relationship to Mavo Concepts

Expressions and Actions directly map to the same concepts in Mavo, except they are partly generated via GUIs.

[4] projects.verou.me/h-chart

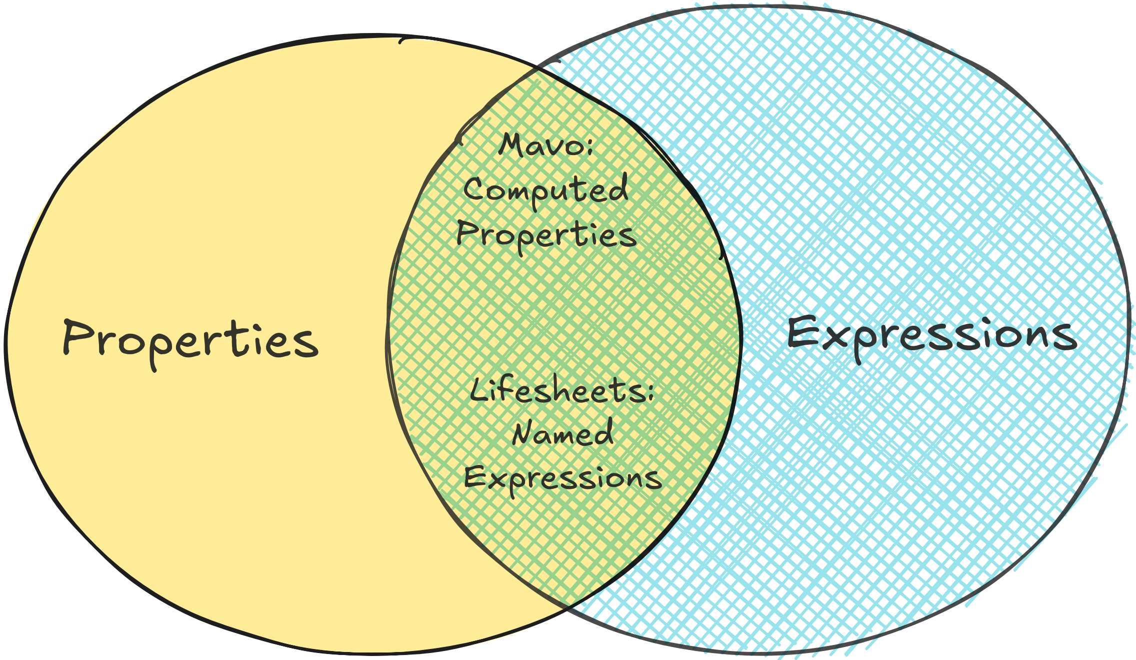
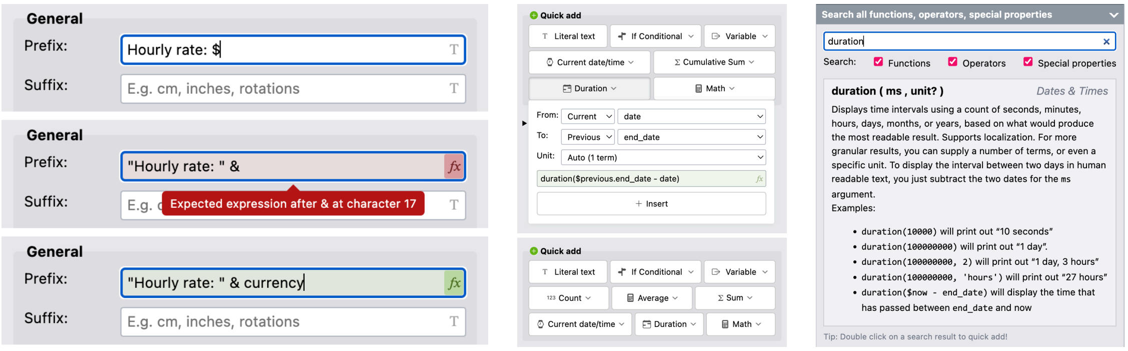


Figure 9.8 *Mavo concepts are largely exposed directly via the Lifesheets GUI, with one exception: computed properties.*



Prefix: Hourly rate: $
Suffix: E.g. cm, inches, rotations

Prefix: "Hourly rate: " &
Suffix: E.g. c
Expected expression after & at character 17

Prefix: "Hourly rate: " & currency
Suffix: E.g. cm, inches, rotations

From: Current date
To: Previous end_date
Unit: Auto (1 term)
duration($previous.end_date - date)

duration
Search: ☑ Functions ☑ Operators ☑ Special properties

duration (ms , unit?)

Displays time intervals using a count of seconds, minutes, hours, days, months, or years, based on what would produce the most readable result. Supports localization. For more granular results, you can supply a number of terms, or even a specific unit. To display the interval between two days in human readable text, you just subtract the two dates for the `ms` argument.

Examples:

- `duration(10000)` will print out "10 seconds"
- `duration(100000000)` will print out "1 day".
- `duration(100000000, 2)` will print out "1 day, 3 hours"
- `duration(100000000, 'hours')` will print out "27 hours"
- `duration($now - end_date)` will display the time that has passed between `end_date` and now

Tip: Double click on a search result to quick add!

Figure 9.9 *Expression authoring conveniences. From left to right: (a) The text field used in places where expressions are allowed in three states: literal text, invalid expression, valid expression (b) The Quick Add widgets with the Duration widget expanded. Top: entry, bottom: general. (c) The docs browser which opens automatically when an expression is focused, on a suitable entry based on the expression content and caret position.*

9.4.10 Expression Authoring

84.7 %

Expressions are allowed in almost any textual setting (e.g. prefix, suffix, visible, default value, etc).

To indicate this, a text field is used with an icon on the right that is T in literal text mode, or fx in expression mode (Figure 9.9) and can be clicked to toggle between modes. Expression mode includes basic syntax checking, error reporting, and parenthesis

balancing. There is also a sidebar widget (Figure 9.9) for browsing Mavo expression documentation that expands every time such an expression field is focused.

However, our goal is that users should not need to type expressions at all for the kinds of calculations that are common in tracking applications. There is a series of widgets that generate function calls for common calculations, such as durations (Figure 9.9), aggregates, or conditionals. The widget allows specifying all parameters through form controls, and displays the resulting expression fragment, to teach the user how to write expressions.

These function calls can be tweaked, then inserted in the main expression field, and then tweaked further. Unfortunately, this is currently only available for the main expression in Expression fields, and not anywhere an expression can be written.

9.4.11 Viewing Modes

85 %

The Lifesheets editor offers two view modes: Design (default) and Preview. The Design mode allows users to click to click on a field to open its configuration editor, as well as edit the main app info in a WYSIWYG way. The Preview mode allows users to use the app they are creating, verify that expressions, actions, and charts work as expected etc. The modes are necessary to differentiate what happens when a user clicks on a field—are they configuring that field or editing its value?

9.4.12 Supporting Tinkering and Iteration

85.1 %

Tinkerability is important in the design of any creative interface [164], and crucial for end-user programmers [165] who are less aware of what exactly they are doing. In Lifesheets, most edits to the application schema are non-destructive: users can create a

[4] projects.verou.me/h-chart

prototype of their tracking application, use it to store data, then return to the Lifesheets editor later and evolve it based on changes to their needs or new insights. If the author removes a field that has been used to enter data, its data is preserved in case the user decides to bring it back. This is also why range start date and time are named `date` and `time`, instead of `start_date` and `start_time`: to support experimentation with different temporal categories.

85.2 %

9.5 Standardizing Tracked Data

Currently, all existing tracking applications define their own tracked ad hoc data schemas. Even when data export is supported, there is little non-programmer users can do with this data. To our knowledge, there is no standard data schema for this type of data, so that different applications can interoperate.

We believe that there is value in converging towards a data schema that all these different applications can use. JSON is a good language for defining this, due to its popularity as a data interchange format and its support for nested data structures (unlike e.g. CSV). For example, one can imagine a user choosing to use one tracking application for its superior data entry capabilities, another for its data analysis capabilities, and a third one for correlations across data tracked by multiple different applications. Since Lifesheets had to be designed to accommodate a very wide variety of tracking use cases, a secondary contribution of this work is to pave the way towards such a standard schema for tracking data.

Even without a description of the tracking use case, the data Lifesheets produces has several commonalities that would allow a third-party application to process them meaningfully: all data includes a root object, with an `entry` key that includes an array of entries, as well as any global properties. Entries are objects, whose values are primitives or arrays of primitives. Temporal fields (`date`, `time`, `end_date`, `end_time`) are standardized across all use cases and their values are the same format (ISO 8601 [5]).

Lifesheets also produces a JSON description of the tracker, which can be used by third party apps to make sense of the rest of the data properties that are not shared across lifesheets.

[5] iso.org/standard/70907.html

The data from each tracker is stored as a JSON file in the user's chosen file space.

The general shape of all tracked data is a root object, with an "entry" key which contains the list of entries in an array, plus keys for any root properties, if any are present.

Entries are represented by JSON objects. Properties are represented by object keys with primitive values, except multi-valued properties which are stored as arrays of primitives. Expressions are not stored, even if they are named. ISO 8601[85] is used for dates and times.

We use `date` and `time` instead of `start_date` and `start_time` as the default property names for the start of the primary time interval; this keeps the names consistent if the author switches between time-point and time-interval data models in their app.

Dates and times are stored separately and joined into datetime values (per ISO 8601 [85]) via expressions at runtime. This design was chosen because in these kinds of use cases different data entries can be of different fidelities: e.g. one may want to record a life event that generally has a time granularity, but not remember the time for a given entry.

Here is an example data entry from the migraine application:

JSON

```json
{
  "date": "2021-09-13",
  "time": "14:40",
  "end_time": "16:30",
  "end_date": "2021-09-13",
  "notes": "Strong headache since wakeup",
  "intensity": 2,
  "side": "left"
}
```

Apart from the tracked data, the description of each Lifesheet is *also* stored as a JSON file, which is a higher level description than the generated Mavo application.

Using the tracker schema, any application could interpret the tracked data in the same way that Lifesheets does.

We believe if such schemas were to be standardized and used across tracking applications, even as an export format, it could benefit users by maximizing interoperability and data portability. For example, even if a particular tracking application does not allow the user to derive correlations on their data, there could be a third-party application that allows users to upload various tracking data and derive correlations and other insights.

Third-party applications can process data even without a description of the tracking application, but having that allows them to *interpret* the data too or even provide alternative editing interfaces.

Each application is represented by a JSON object with keys:

- `app`: Main app information (e.g. name, what is being tracked, etc)
- `temporal`: An object with two boolean keys: `times` and `ranges`, to select a temporal category
- `entry`, `root`, `charts`: Arrays of objects for entry fields, general fields, and charts respectively

Each field (including charts) is represented as an object with:

- a `type` key (values: property, expression, action, spacer, chart)
- a `name` (mandatory in properties, optional in expressions)
- a `flags` array with flags like `"temporal"` for storing special state about auto-generated properties
- a `settings` object with various options, depending on the field

9.6 Lifesheets Case Studies

85.9 %

Part of our argument is that Lifesheets has sufficient flexibility to easily specify a broad range of manual tracking use cases. To support that argument, we looked at the tracking use cases outlined by our needfinding survey participants (description, plus answers to “What data do you record for each entry?” and “How do the tool(s) you use help you understand your data?”). Three researchers worked together to review these use cases for implementability with Lifesheets.

For semi-automatic cases, we only considered the manual component.

Researchers split the 168 use cases into four categories: i) Invalid, not enough data to discern what the answer was trying to convey or participant had misunderstood the question (7) ii) Use cases where at least one of the researchers could outline a complete Lifesheets implementation (properties, expressions, actions, charts) were marked as “Implementable” (153, or 95% of valid use cases). If they were not sure, they implemented the use case. iii) Use cases that the researchers were certain could *not* be implemented, were marked as *"Not implementable"* (6) iv) The remaining two use cases were marked as "Borderline" (2).

9.6.1 Discussion

The data entry component of nearly all valid use cases (161/162 or 99.4%) was implementable with Lifesheets' design. The one that was not, was a running tracker with a running route drawn on a map, as Lifesheets does not support drawing as an input widget (drawing on a map and uploading it through a media property would have worked, but is suboptimal), not does it connect to any wearable location tracking devices.

For the remaining 7 unimplemented or borderline cases, the issues were:

- The 2/8 borderline cases needed correlations between different tracked data. Visual correlations are possible, by tracking all data on the same lifesheet and plotting two different fields on the same chart. However, it may be suboptimal to use the same tracker for both, and we cannot know if participants were referring to visual correlations or computed ones.
- 3/8 required access to existing large datasets or APIs (nutritional info of foods, locations of places). Small datasets (e.g. baby teeth duration, or WHO child growth data) can be entered as hidden general fields and used in expressions or plotted in charts. However, this becomes impractical above a certain number of data points (≈ 20 or so).
- 1/8 needed notifications
- 1/8 required a proprietary calculation of fitness level. This may be feasible with expressions, but there is no way to know without more details.

It is important to note that Lifesheets requires more *knowledge* than simply reaching for a widely available application. For example, in a menstrual/ovulation tracker (3 use cases), ovulation can be predicted by adding the average cycle length to the cycle start date and subtracting 14 days (in Formula[2] expressions: `date + avg_cycle_length - 14 * days()` where `avg_cycle_length` is a general expression with value `average(duration)`. However, not everyone would be able to perform this calculation. In practice, we expect that once Lifesheets is deployed widely, only a few users would need to know how to implement the calculations for common cases, and the rest would simply "fork" their sheets and customize them.

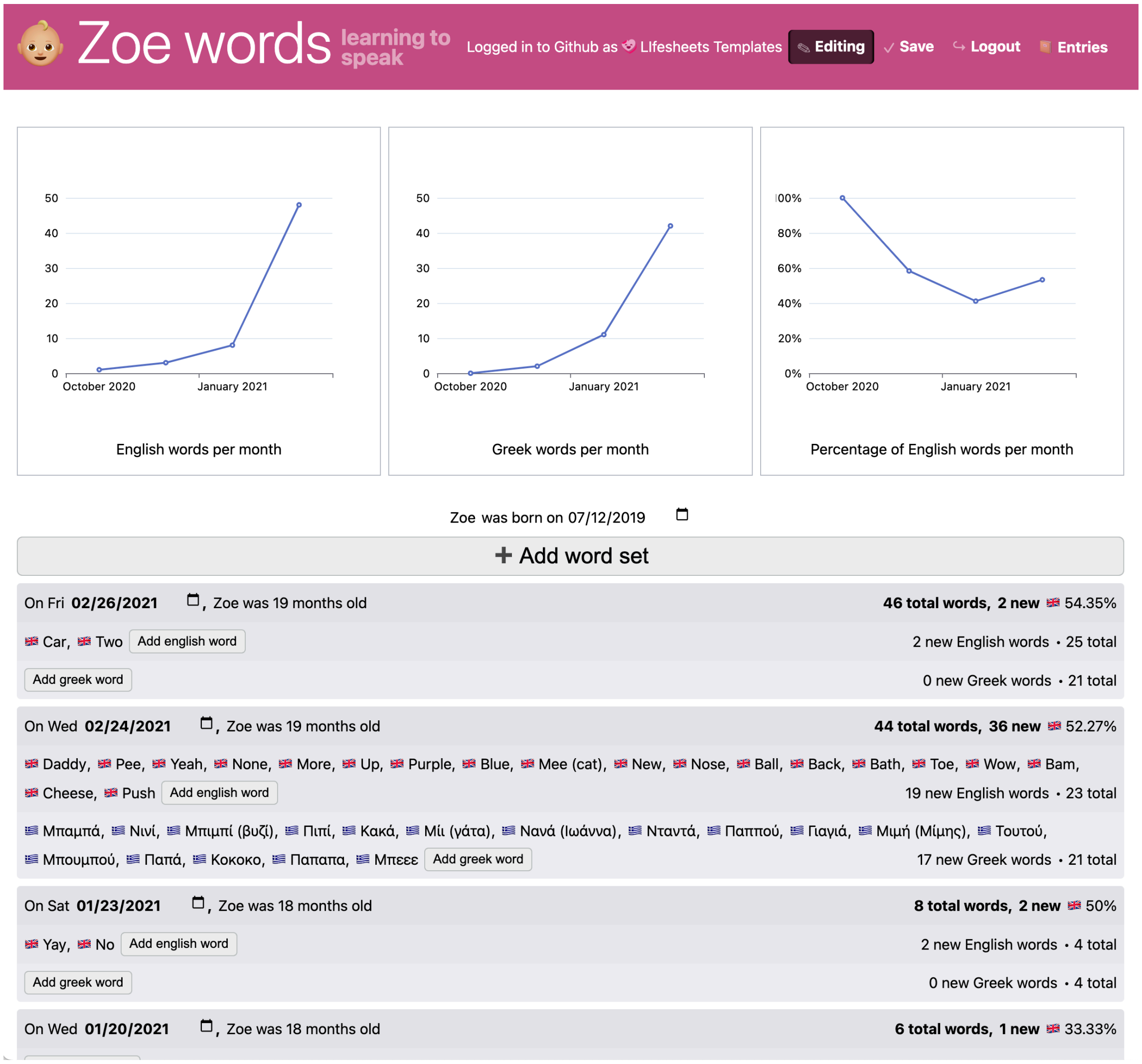
The use case data as well as the researcher assessments is included in the supplementary materials.

9.6.2 Detailed Case Studies

In this section, we present more details about some of the more interesting, complex cases we implemented.

Bilingual Child Vocabulary

While most respondents who wanted to track their child's language development only tracked words with dates, there was one who listed "language" as one of the fields they track.



English words per month

Greek words per month

Percentage of English words per month

Zoe was born on 07/12/2019

On Fri **02/26/2021**, Zoe was 19 months old — **46 total words, 2 new** 🇬🇧 54.35%

🇬🇧 Car, 🇬🇧 Two — 2 new English words • 25 total

0 new Greek words • 21 total

On Wed **02/24/2021**, Zoe was 19 months old — **44 total words, 36 new** 🇬🇧 52.27%

🇬🇧 Daddy, 🇬🇧 Pee, 🇬🇧 Yeah, 🇬🇧 None, 🇬🇧 More, 🇬🇧 Up, 🇬🇧 Purple, 🇬🇧 Blue, 🇬🇧 Mee (cat), 🇬🇧 New, 🇬🇧 Nose, 🇬🇧 Ball, 🇬🇧 Back, 🇬🇧 Bath, 🇬🇧 Toe, 🇬🇧 Wow, 🇬🇧 Bam, 🇬🇧 Cheese, 🇬🇧 Push — 19 new English words • 23 total

🇬🇷 Μπαμπά, 🇬🇷 Νινί, 🇬🇷 Μπιμπί (βυζί), 🇬🇷 Πιπί, 🇬🇷 Κακά, 🇬🇷 Μίι (γάτα), 🇬🇷 Νανά (Ιωάννα), 🇬🇷 Νταντά, 🇬🇷 Παππού, 🇬🇷 Γιαγιά, 🇬🇷 Μιμή (Μίμης), 🇬🇷 Τουτού, 🇬🇷 Μπουμπού, 🇬🇷 Παπά, 🇬🇷 Κοκοκο, 🇬🇷 Παπαπα, 🇬🇷 Μπεεε — 17 new Greek words • 21 total

On Sat **01/23/2021**, Zoe was 18 months old — **8 total words, 2 new** 🇬🇧 50%

🇬🇧 Yay, 🇬🇧 No — 2 new English words • 4 total

0 new Greek words • 4 total

On Wed **01/20/2021**, Zoe was 18 months old — **6 total words, 1 new** 🇬🇧 33.33%

Figure 9.10 *Bilingual child vocabulary tracker.*

Any vocabulary tracker (even a monolingual one) is an example of a tracking application that requires a hierarchical schema, and thus, is hard to do with spreadsheets or most no-code tools.

This is an outline of the lifesheet implementation:

- Temporal category is *Single Date*
- Two (one for each language) multi-valued Text properties with 0 initial items. We use prefixes to distinguish the two.
- Two named expressions to count the new words in each entry, e.g. `english_word_count` with value `count(english_word)`[^6]
- Two cumulative sum expressions to count total words, e.g. `total_english_word_count` with value `$previous.total_english_word_count + english_word_count`
- Named expressions for counting total vocabulary and total new words (e.g. `total_english_word_count + total_greek_word_count`)[^7]
- Expressions to calculate percentage of vocabulary by language (e.g. an `english_percentage` expression with value `100 * (total_english_word_count / total_word_count)`)
- An expression to show the age of the child when the entry was entered
- Three line charts, two for showing the average word count per month for each language and one for plotting the percentage of English per month

While it is not strictly necessary for implementing this use case for one's own needs, we also included a general `birthday` Date property and a `child_name` property (we could use

hardcoded values), to allow others to "fork" the sheet and use it for their own child(ren) without having to comb through every expression to figure out what to change.

The resulting application can be seen in Figure 9.10 and the implementation can be explored at lifesheets.app/app/?sheet=github.com/lifesheets-templates/my-lifesheets/zoe-words/app.json&copy=1. It can be trivially modified to support trilingual or other polylingual families.

86.6 %

Babysitting Calculator

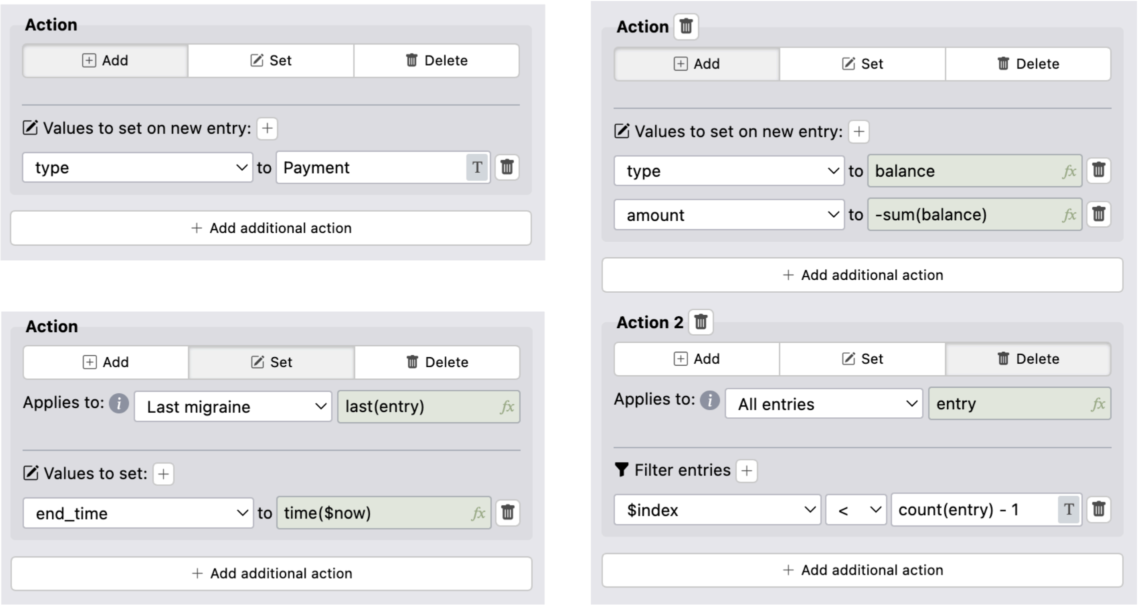
This lifesheet tracks hours worked by a babysitter, payments by the parents, as well as occasional expenses, and displays total balance owed.

This lifesheet showcases a useful pattern: how one can emulate heterogeneous entries with different temporal characteristics. Conceptually, it needs three types of entries:

- **Care** that is a regular date/time range entry with no other fields
- **Payment** which corresponds to a payment by the parents, and does not need anything more than a date and an amount.
- **Balance** which adjusts the balance by a specific amount. This is useful for a) consolidating past entries into a single equivalent entry and b) reimbursing the babysitter for expenses (e.g. museum tickets). Like Payment, this only needs a date and an amount.

To implement this, we add a `type` Options property with three values: *Care, Payment, Balance*. We select the widest temporal category that we need (range of dates & times).

We then set the visibility of `end_time`, `end_date`, `end_time` to an expression that only shows them when `type = ’Care’`. Similarly, `amount` is only shown when `type != ’Care’`.



Action

Values to set on new entry:

type to Payment

Action

Applies to: Last migraine last(entry)

Values to set:

end_time to time($now)

Action

Values to set on new entry:

type to balance

amount to -sum(balance)

Action 2

Applies to: All entries entry

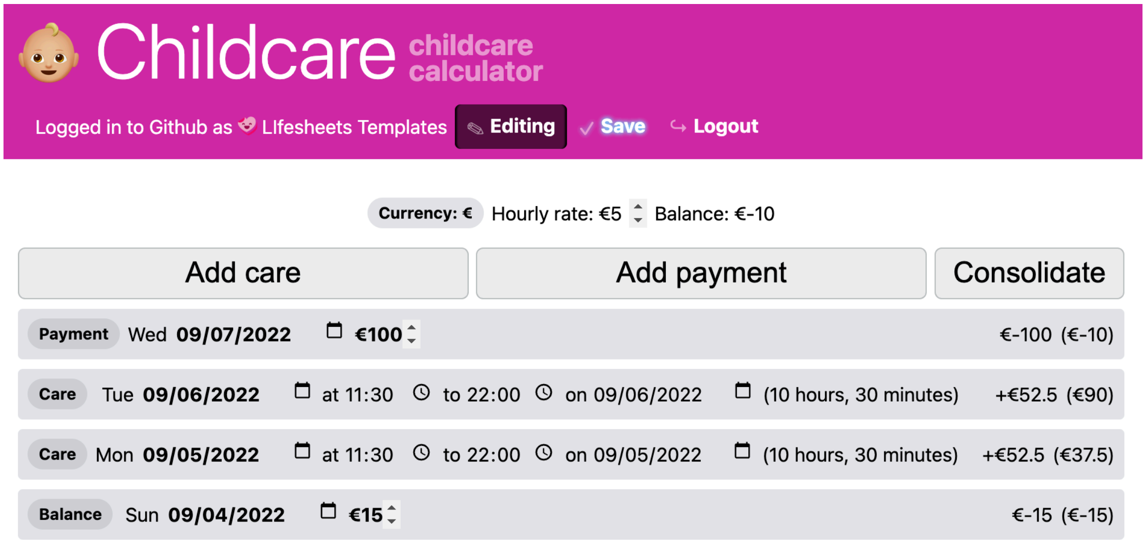
Filter entries

$index < count(entry) - 1

Figure 9.11 *UI for actions, showing two simpler actions and a more complex, composite one: i) Add new entry with prefilled fields ii) Set end date to current time (essentially marking an event as "finished") iii) "Consolidate" entries by replacing them with a new equivalent entry*

To facilitate data entry and automation, we implement three actions: i) Two actions for entering each type of entry and ii) An action to consolidate entries, which adds a Balance entry with the value of the total balance, and deletes all entries after it (see Figure 9.11).

The rest is simple properties, and expressions that can be directly generated with widgets. Like in the previous case, we have made this more generalizable by introducing general properties for currency and hourly rate and used them in the corresponding prefixes and calculations, so that it can be copied and used by others.



Currency: € Hourly rate: €5 Balance: €-10

Payment Wed **09/07/2022** **€100** €-100 (€-10)

Care Tue **09/06/2022** at 11:30 to 22:00 on 09/06/2022 (10 hours, 30 minutes) +€52.5 (€90)

Care Mon **09/05/2022** at 11:30 to 22:00 on 09/05/2022 (10 hours, 30 minutes) +€52.5 (€37.5)

Balance Sun **09/04/2022** **€15** €-15 (€-15)

Figure 9.12 *The finished care hours calculator. A live version of this application can be found in lifesheets.app/user/lifesheets-templates.*

The application can be seen in Figure 9.12 and the detailed implementation can be explored in lifesheets.app/app/?sheet=github.com/lifesheets-templates/my-lifesheets/childcare/app.json&copy=1 .

87 %

9.7 User Study

To evaluate Lifesheets' learnability and efficiency, we ran a user study of 10 participants. We used spreadsheets as a control because they are the most common tool people reach for when they cannot find a suitable application (Section A.1.6).

9.7.1 Participants

The 10 (2 female) participants were recruited among survey respondents that fulfilled the following inclusion criteria: (1) Had opted-in to participating in the user study, (2) had enough familiarity with spreadsheets to write basic formulas and (3) track or want to track at least 6 things total (self-tracking + parental tracking)

Ages ranged from 30 to 47 (median = 39, x̄ = 38.6, σ = 4.9). 5/10 were parents (1 female). 5/10 had little to no programming experience. 3/10 rated their ability to write spreadsheet formulas “advanced” and 7/10 “basic”. User study sessions were conducted over Zoom and took 90-120 minutes each.

Procedure and Study Setup

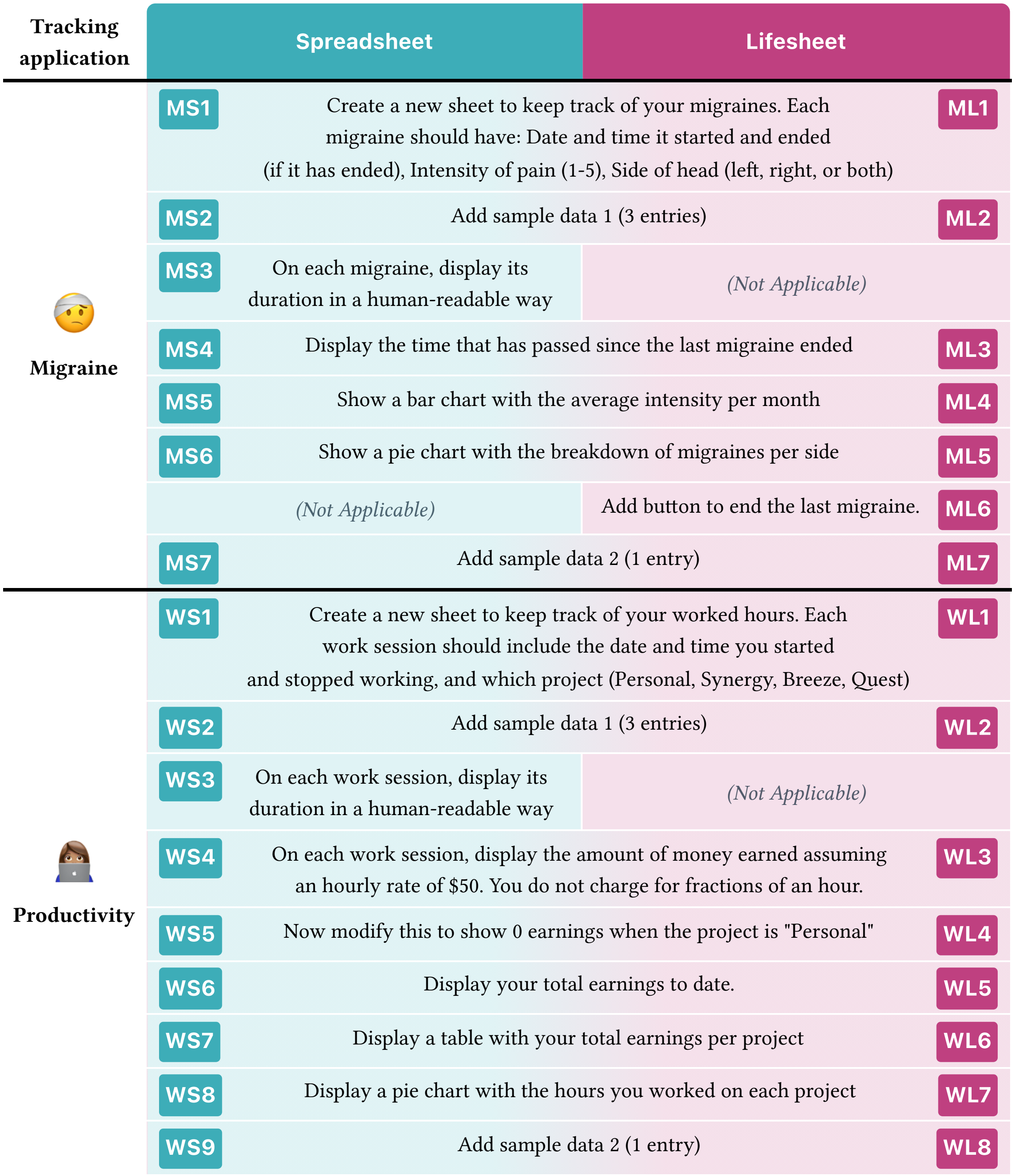
We used a within-subjects design, with each participant creating a specific tracking application with Lifesheets and a *different* one with Google Sheets. Both the assigned apps and the order of the two conditions were randomized and counterbalanced. The two applications were (a) a Migraine tracker and (b) a Productivity tracker. We selected these use cases as the two tasks for a variety of reasons: (a) both were among the most common use cases (Figure A.1) (b) both required date and time ranges, the most complex of the four temporal categories (c) they were implementable with Lifesheets, but nontrivial

Participants were first briefly interviewed about their tracking habits (primarily to build rapport). The control condition consisted of them creating their assigned application in Google Sheets step by step Table 9.1.

The experimental condition consisted of them watching a 7 min recorded tutorial about Lifesheets, then doing a small practice task (smoking tracker) for x̄ = 10min where the researcher could help them. A smoking tracker was selected because it was simple to implement, yet practiced all core Lifesheets concepts. Then they would go on to create their assigned lifesheet, without researcher help.

If there was time at the end (unfortunately only in 4/10 of sessions), participants were encouraged to work on a freeform task for their own needs.

After these tasks, participants completed a post-study questionnaire which included separate System Usability Scale (SUS) [115] questions about creating and using the application, for both conditions. It also included a few multiple-choice questions about their experience with Lifesheets and how it compares to spreadsheets and tailored tracking applications. Participants completed the post-study survey on their own, without researcher oversight.



Tracking application	Spreadsheet		Lifesheet	
🤕 Migraine	MS1	Create a new sheet to keep track of your migraines. Each migraine should have: Date and time it started and ended (if it has ended), Intensity of pain (1-5), Side of head (left, right, or both)		ML1
	MS2	Add sample data 1 (3 entries)		ML2
	MS3	On each migraine, display its duration in a human-readable way	*(Not Applicable)*	
	MS4	Display the time that has passed since the last migraine ended		ML3
	MS5	Show a bar chart with the average intensity per month		ML4
	MS6	Show a pie chart with the breakdown of migraines per side		ML5
		(Not Applicable)	Add button to end the last migraine.	ML6
	MS7	Add sample data 2 (1 entry)		ML7
👩‍💻 Productivity	WS1	Create a new sheet to keep track of your worked hours. Each work session should include the date and time you started and stopped working, and which project (Personal, Synergy, Breeze, Quest)		WL1
	WS2	Add sample data 1 (3 entries)		WL2
	WS3	On each work session, display its duration in a human-readable way	*(Not Applicable)*	
	WS4	On each work session, display the amount of money earned assuming an hourly rate of $50. You do not charge for fractions of an hour.		WL3
	WS5	Now modify this to show 0 earnings when the project is "Personal"		WL4
	WS6	Display your total earnings to date.		WL5
	WS7	Display a table with your total earnings per project		WL6
	WS8	Display a pie chart with the hours you worked on each project		WL7
	WS9	Add sample data 2 (1 entry)		WL8

Table 9.1 *The tasks for both applications and both conditions. The wording of each task has been lightly edited for length.*

9.8 Results & Discussion

	Spreadsheet		Lifesheet	
🤕 **Migraine**	Completion:	**0**/5	Completion:	**4**/5 👤👤👤👤
	Time:	*(N/A)*	Time:	14m 33s
	Failed tasks:	**3** (x̄ = 2.4)	Failed tasks:	**0** (x̄ = 0.4)
🧑‍💻 **Productivity**	Completion:	**2**/5 👤👤	Completion:	**3**/5 👤👤👤
	Time:	17m 56s	Time:	10m 4s
	Failed tasks:	**1** (x̄ = 1)	Failed tasks:	**0** (x̄ = 0.6)

Table 9.2 *Task completion rates, times, and number of failed tasks per condition. Numbers are medians unless otherwise noted. $\bar{x}$ is the mean.*

As shown in Table 9.2, participants generally succeeded more at implementing these applications with Lifesheets compared to spreadsheets, took less time, and failed at fewer tasks.

There were recurring patterns in the issues participants faced in either condition, which we discuss below.

9.8.1 Lifesheets UI

Personalization

The Lifesheets' personalization features were well-received, despite being spartan. All 10 participants customized at least one aspect of the application style (color, icon), even though they were not asked to by any task.

Name vs Prefix vs Suffix vs Label

Nearly all participants tried to use the "Name" field as a visible label, rather than an identifier for referencing. Most recovered once they saw the error in the UI, but 2/10 needed researcher intervention (and were counted as failures).

Even after, some struggled to find where to actually place a human-readable label, as they did not identify "Prefix" and "Suffix" as suitable and were looking for a "Label" field. All were able to recover, and *preferred* prefix/suffix over their initial expectation.

Preview Modes

Most participants did not understand the difference between the different preview modes (Design mode, Preview mode, opening the app in a new tab).

They expected to be able to save data in Preview mode, in which the data entered is currently intended for experimentation only. Some expected that "Save" in the Lifesheets editor would *also* save the data they had entered (it saves the lifesheet itself — which includes a separate Save button to save data).

The participants with the least programming experience expected the Design view to have more WYSIWYG features (e.g. moving fields around with drag & drop), while the programmers were generally content to use the sidebar and did not even use the WYSIWYG features that the Design view did support.

9.8.2 Data Entry

It is hard to measure the real efficiency difference of using a lifesheet in a lab setting: since data entry is artificial, with dates and times in the past, the user is fighting against the default values, rather than taking advantage of them.

Regardless, we observed a significant difference in the last data entry task (no signficant difference in the first task): adding one item took a median of 20s (migraine: 18s, productivity: 23s) in Lifesheets and a median of 57s (migraine: 53s, productivity: 59s) in spreadsheets. In the Migraine task this was largely due to the "End migraine" action from ML6 as the migraine to add had just finished. All participants used the action they had created instead of entering data manually. However, Producitivity did not include any actions, and yet the difference was comparable. We hypothesize that participants had simply grown more familiar with lifesheets by then and could take better advantage of the interface.

The predefined temporal fields of Lifesheets implement the common UI pattern where the end date is not shown until an end time is entered, as it uses a dynamic default value that depends on the start date, and both times. This initially confused 2/10 participants as they were looking to enter an end date before entering an end time.

Customization can be a double-edged sword: U9 had reordered their fields to experiment with the interface and had trouble during the data entry task because controls were in unpredictable places. 3/10 were initially confused that new entries were added to the top, they expected them to be appended. Interestingly, their expected behavior would have resulted in worse ergonomics: if entries were added to the end, one needs to scroll through all entries to see and edit their new entry, or just to see the most recent data.

3/10 participants did not notice that a placeholder entry existed when they began entering data, and just ignored it, then were confused that there was a superfluous entry. The placeholder entry was not surprising in the preview while *editing* the application, it only became suprising when actually *using* the application. To address this, we plan to start with 0 entries outside the editing environment in the future, and only use a placeholder entry in the preview.

9.8.3 Formulas

88.4 %

Referencing Data

Most participants struggled with spreadsheet cell references at least once, regardless of whether they used the selection GUI or tried to write out references.

Several participants made errors that would have been apparent with column names, but were difficult to discern with spreadsheet ranges such as e.g. `E2:E9`. In some cases, the error itself was that the participant attempted to reference a column by its header, rather than use a range. None used named ranges to overcome these issues.

There was no such issue in the Lifesheets condition, as names are used for referencing.

Datatypes and Operations

88.6 %

In both conditions, participants struggled with operations due to unexpected operand types. In spreadsheets, many struggled to perform operations between dates because their entered dates were treated as text. Some operations produced duration values (e.g. `2:5`) which they could not interpret or reformat to make more human-readable.

In Lifesheets, many tried to do math with the output of the `duration()` function, which is a text value (e.g. `"5 months"`). Predictably, those with programming experience figured the issue out far faster than the rest.

Specifically for durations, this pattern can be easily supported, by making `duration()` return a text value that when coerced to a number returns the number of milliseconds (which is what date/time functions expect).

Duration Math

Nearly all participants struggled to calculate durations and present them in a human-readable format with spreadsheets. The participants most successful at these tasks subtracted two date/time values and formatted the result as a duration, but this was still not particularly human-readable (it was presented as hh:mm:ss, even for durations over 24 hours, e.g. "607:0:0").

Their data modeling intuition played a crucial role in setting them up success: none of the 5/10 who started their spreadsheets with separate dates and times could figure out how to combine them to get a single date/time value to perform calculations with.

Subtracting date/times produced a fractional number of days, which they found confusing. Searching the documentation or googling for help yielded functions that did not help them (such as `DATEDIF()` or `DURATION()`, both of which are for date intervals, and not date/time intervals).

The Productivity condition includes a task where hours worked need to be multiplied by hourly rate to display earnings (WS4/WL3). 3/5 subjects struggled with this in the spreadsheet condition, as multiplying a duration with a number produces a result formatted as a duration with unclear units (e.g. `25:3`).

No-one was able to present the time since the last migraine in the spreadsheet. One participant thought of using `MAX()` to get the latest date, but could not present the difference between that and the current time in a human-readable way.

Participants were vocal about their frustration while attempting any tasks that involved math between date/time intervals in spreadsheets.

> *"this is giving me a headache"* — U1

> *"Honestly, I would rather calculate it manually and not use a formula"* — U4

> *"I never did something like that with spreadsheets and it really sucks. I can cry"* — U5

> *"That was frustrating"* — U6

> *"This is showing me how much I don't like Google Sheets"* — U9

All participants were able to accomplish these tasks in Lifesheets. For entry duration, there was no corresponding task because Lifesheets displays this in a human-readable way automatically (e.g. *"3 days, 5 hours"* or *"1 hour, 30 minutes"*

). Everyone used the Duration widget to calculate time since last migraine (ML3), though 2/5 were confused about the "Auto" unit (but tried it anyway).

88.9 %

Conditionals

Only the productivity tracker required the use of a conditional (WS5, WL4). 3/5 participants were successful in using `IF()` for it in spreadsheets, and 5/5 in Lifesheets. All 5 participants in Lifesheets used the Conditional widget instead of writing an expression, 4/5 successfully (U7 did not understand they could have an "Otherwise" value as well).

Despite this being a task that would really benefit from auxiliary data (e.g. a separate hourly rate field), only one participant in each condition thought to create a separate spreadsheet column or field to make the expression more readable. This validates the existing finding that end-user programmers reuse by cloning [88].

89.2 %

Aggregates

Only the productivity tracker required the use of an aggregate (total earnings, WS6/WL5). All participants were able to accomplish this, under both conditions. In Spreadsheets, 4/5 participants wrote the formula on their own, and one used the menu to insert a `SUM()` that they then edited. In Lifesheets, all 5 participants used the Sum widget. One had not given a name to their (in-entry) earnings expression, but named it then so that it could be used as the argument to the sum.

9.8.4 Data Modeling

Temporal Data

Handling temporal data was particularly difficult in spreadsheets. There is a tension: **meaningful temporal calculations require combined date/time values, but that is not a natural format for human data entry.** Our study verified this tension: half our participants started with separate columns for dates and times, then struggled to combine them for calculations. More than half (3/5) did not include an end date, only an end time, expecting to be able to calculate the end date from the other three columns. The other half started with combined date/time values, but then struggled to enter them textually, with 2/5 of which completely failing to figure out a textual format that was recognized as a date/time.

Despite spreadsheets providing a way to constrain input to a specific format (data validation), which also displays suitable input widgets (e.g. a date picker), only 2/10 participants used it for dates and times (and only 4/10 for *any* value).

In Lifesheets, all 10 participants selected the correct temporal category without hesitation.

89.3 %

Other Data Modeling Tasks

There were few differences in non-temporal data modeling tasks in spreadsheets. All participants were able to create the necessary properties with roughly similar names. Only 4/10 used any type of data validation.

In Lifesheets, 9/10 customized the entry name, 1/10 was happy with the default *"entry"*. In the Migraine task, all selected "migraine", whereas in the Productivity tracker, names were more varied: "hours worked" (2), "task", "session", and "work". This is expected: a migraine is a concrete event, while a work session is more abstract.

All 10 participants created suitable properties and correctly understood the difference between property types. All 10 participants selected suitable types for their properties with minimal experimentation. While none of the tasks required multi-valued properties, every participant seemed to understand what selecting "Multiple values" did (it turns the property into a collection), and could recognize that they did not need it for these tasks (4/10 after experimentation). Across both tasks, 4/10 participants experimented with selecting "Multiple values" on a *Text* property instead of selecting an *Options* property in the corresponding subtask, but seeing the collection management controls it created was enough feedback to help them realize it was not what they needed.

Some more specific data modeling observations:

- Migraine:
  - **Intensity**: All 5 participants selected a *Number* type
  - **Side**: 4/5 selected an *Options* property; one created two toggles instead: left, and right.
- Productivity:
  - **Project name**: All 5 participants used an *Options* property

9.8.5 Pivot Tables and Charts

Each structured task included two forms of reflection visualizations, that were essentially grouping tasks: Migraine called for a bar chart of average intensity by month (MS5/ML4), and a pie chart of the breakdown of migraines per side (MS6/ML5), while the Productivity app called for a table of total earnings per project (WS7, WL6), and a pie chart of hours worked on each project (WS8/WL7).

Participants generally struggled a lot with these in spreadsheets, especially those involving temporal grouping and aggregation. While it is known that spreadsheet charting interfaces are generally poor [5], the main issue at play seemed to be unrelated to these issues. **Indirection was a big barrier**: if creating the chart required auxiliary data such as a separate column or a pivot table, users struggled. Participants generally expected to be able to create the visualizations through a combination of settings in the charting interface, and continued trying different parameters until they got the right result (often as a happy accident), or got frustrated and gave up.

Non-programmers struggled more with indirection. We hypothesize this is because programmers are more used to creating "helper variables" as containers for intermediate values, while end-users expect to be able to achieve their goal by selecting the right combination of user interface settings. Interestingly however, even programmers had trouble with auxiliary columns in spreadsheets.

In the Productivity tracker, both visualizations required similar tables: projects on one dimension and earnings or hours on the other. **No participant used a pivot table** for this. The 2 that succeeded at this task used a hardcoded list of projects with `SUMIF()` to aggregate earnings and hours. It is debatable whether this should be marked as a success, as it is not a generalizable solution.

In the spreadsheet condition, *no* participant was able to display a bar chart of average migraine intensity per month. **Grouping by month** was particularly hard. 2/5 thought of creating a separate column but failed at doing so. Averages by month were also hard: if they could not extract the month, they could not use `AVERAGEIF()` to calculate an average on it. U5 said, *"I don't think I could do that. I would not be able to do that. I'm already thinking how to do that in a lifesheet, but in this case, I will give up."*.

The *migraines per side* pie chart was easier due to the lack of grouping and aggregation. Regardless, 2/5 participants failed. Of the 3 that succeeded only one understood how they achieved the right result, the other two simply tried different settings until they got the right result.

Participants had a significantly easier time with the Lifesheets charting interface, which is optimized for grouping tasks as a thin abstraction over .

All 5 were able to display a bar chart of average migraine intensity per month, in a median of 34s. 4/5 were able to display a pie chart of migraines per side in a median time of 24s, although they hesitated before grouping by a non-temporal factor. The one that couldn't, U8, had created two separate toggles for the migraine side: `left` and `right`. Creating the chart is possible with this design but requires a hidden expression to return a single value (e.g. `if(left and right, 'both', if(left, 'left', 'right'))` or `if(left, 'L') & if(right, 'R')` to avoid conditional nesting which novices find hard [11]) , and as discussed above participants have trouble with adding hidden auxiliary data.

All participants were able to display a table of total earnings per project, and unlike spreadsheets, the list of projects reactively updated with the data. None had a separate field for hours worked, so 2/5 failed at the subsequent task of displaying a pie chart of total hours per project. As discussed earlier, it felt far more natural to programmers to create a hidden auxiliary field than to non-programmers. Both participants that failed were in the non-programmer group, and all three that created a hidden `hours` field had programming experience.

Participant responses to the charting interface ranged from positive to enthusiastic. U9 during the bar chart task: "Out of the things you have going on here, the charting is remarkable. I just created this bar chart with zero effort and you saw how much I struggled in Google Sheets.". And then during the next pie chart task: "I thought I have no freaking idea how to do that, and then it was just right there!" U10: "Lifesheets made creating charts A LOT easier than Google Sheets. It makes a lot of assumptions for me that I don't have to manually tell it."

9.8.6 Actions

There was only one action task: A button to end the last migraine in the Migraine condition. In Lifesheets, this involved constructing an action to set `end_time` to `time($now)` for the last entry. There was no control for this since spreadsheets do not support a similar feature (except via script).

A core difference from Mavo is the handling of *computed properties*. In Mavo, these are framed as properties whose value happens to be an expression, and are typically neither editable nor saved in the data store. Lifesheets has an expression authoring GUI and a property configuration GUI, which are separate. Since configuring a property's editing UI is rarely useful for computed properties, but expression authoring affordances definitely are, these were instead framed as *named expressions*.

However, some participants of our user study did inquire about combining the two concepts, e.g. having custom widgets whose value was computed from an expression. This is already somewhat possible by setting a property's default value to an expression, but that is a workaround, not a first-class concept. This may indicate that a better direction might be to blur the line between the two concepts and make *both* types of UIs available to both of them, varying only which one is shown by default, and hiding only the parts that truly are not meaningful for computed properties.

Lastly, charts are simply dynamic Mavo collections that use grouping and aggregation to generate a table of data, which is then either output directly, as a pivot table, or wrapped in an `<h-chart>`[4:1] web component to transform it into a chart.

All 5 participants understood what actions are and how to specify one and 4/5 succeeded on this task. The one who failed (U8) selected the correct action type (Set) but could not figure out how to specify the current time (used `$now` instead of `time($now)`), which is quite a natural syntax that we *should* support.

All proceeded to use this button in the subsequent data entry task (ML7), which made this task far faster than the same data entry task in the control condition (MS7) (median of 18s vs 54s).

90.3 %

9.8.7 Overall Impressions

Ease of Creating Tracking Applications

The participants' frustrations with spreadsheets permeated their SUS evaluation: They gave spreadsheet creation a median SUS score of 33.75 ($\bar{x} = 36.25$, $\sigma = 21.3$), which corresponds to an adjective rating between "Worst imaginable" and "Poor" according to [166]. In contrast, lifesheet creation got a median SUS score of 78.75 ($\bar{x} = 75.75$, $\sigma = 10.3$), far above the acceptability threshold of 67, and corresponding to an adjective rating between "Good" and "Excellent".

U1 liked the temporal settings:

> *"Using spreadsheets to track something with dates and times is really hard, if I write dates it kind of works, but it's weird. One of the core principles of lifesheets is that times and dates are at the core, you get a lot of power from how this works."*

U4 found the expression creation UI easier in Lifesheets:

> *"Oh a lot easier than using the spreadsheet, because I didn't have to try and remember formulas. When you're confronted with a spreadsheet you have to think about a lot, how to arrange columns, here it's almost a drag & drop of what you want to do; a lot more intuitive."*

U8 found Formula^2 expressions and Lifesheet's visual builder helpful:

> *"Expressions are super helpful. That's the first step in getting something back from your data. The UI to create a sheet is really nice. It basically works around everything that I was just struggling with Google Sheets and the calculations I was trying to do. […] Very easy, can't compare to the spreadsheet, it's a breeze."*

90.5 %

User Satisfaction

Participants rated using a lifesheet far more pleasant than a spreadsheet. On a scale of 1 (Very pleasant) to 5 (Very unpleasant), they rated using a spreadsheet $\bar{x} = 3.6$ and the lifesheet $\bar{x} = 1.8$.

Spreadsheet usage got a median SUS [115, 166] score of 51.25 ($\bar{x} = 50.5$, $\sigma = 23.8$), which corresponds to an adjective rating of slightly below "OK". Lifesheet usage got a median SUS score of 90 ($\bar{x} = 87.25$, $\sigma = 9.4$), which corresponds to an adjective rating between "Excellent" and "Best imaginable".

U3 remarked on the efficiency of the data entry interface: *"Easier and more structured. Even this little "Add" button, it feels like adding things, like adding a new row to a spreadsheet. The interface is much friendlier than a Google sheet."* U4 said that they preferred the visual presentation of a lifesheet: *"The spreadsheet looks sterile, this is more visual. My current tracking is a blackboard on my fridge. I respond to things more visual."* U5 also liked the more structured data entry interface of a lifesheet, as well as the fact that it works well on a phone: *"working with data in spreadsheets feels horrible, working with the lifesheet was way better. […] Lifesheets is more structured, the way you work with datetimes is way better. Super cool that it works with your phone as well."* U6 liked actions and also mentioned mobile friendliness: *"Definitely easier than spreadsheets. The customizable buttons to add entries and create shortcuts, that's definitely an improvement over spreadsheets. Ease of entry is a big one. All the data and the visualization, it's great, easier, but the deciding factor is that I can't enter stuff into a spreadsheet when I'm in the middle of doing something."* U7 also mentioned the structured data entry, and also the charts: *"Having it automatically put together something that has this summary view at the top. With spreadsheets it's not organized the same way, maybe it can be but it requires more effort. This is a simpler view, with spreadsheets it's hard to tell where to look. Spreadsheets have this nature of sometimes there are things this way, or that way, and you don't know where to scroll. It can be difficult to do a handoff. As a manager I don't want to spend too much time figuring out where everything is, there's a lot of importance to that. I have a lot of challenges with spreadsheets, this guides people more, 'that's it, don't worry about the rest'."* U9—the participant that tracked the most things out of everyone else in the study—spontaneously made Lifesheets his #1 bookmark during his session and remarked on customizability and portability *"Better than other data trackers because it's so customizable. A huge benefit of what you're doing is that it's on my computer. You're one step ahead of things like Bearable [144] in that regard."*

90.8 %

Likelihood of Lifesheet Usage over Alternatives

All participants said they were more likely to use a lifesheet over a spreadsheet for tracking (6/8[6] much more likely, 2/8 slightly more likely). 5/7 participants even said they would be more likely to use a lifesheet over a widely available app for the same purpose! (3/7 much more likely)

90.9 %

Privacy, Data Ownership, and Portability

Several participants (U1, U5, U10) expressed privacy and data ownership concerns about using existing widely available tracking applications, consistent with one of the main barriers identified in the literature [123, 125–128].

> *"Sharing data with a random number of companies makes me uncomfortable." — U5*

[6] The question was added after the first 2 sessions

Consequently, participants liked that the data Lifesheets records are stored in their own space of their choosing, rather than the servers of a commercial application. Those who could program additionally remarked on data portability:

> *"I like that it stores the data on GitHub where I could use it on other projects. Also like that it saves history on GitHub."* — U3

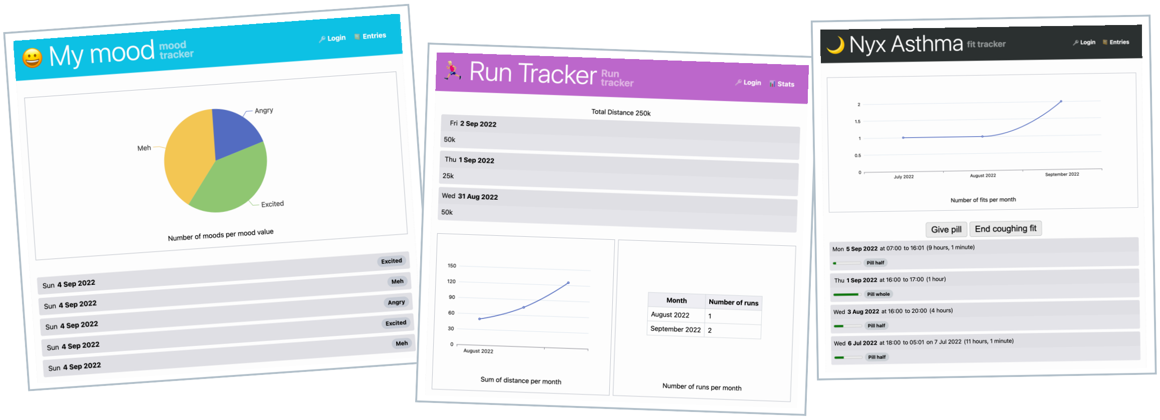
U10, a professional programmer, said that for certain use cases, any data stored on *any* server is not private enough:

> *"Biggest concern I have with these apps is privacy, even in a private repo. Some trackers are so private I wouldn't even want on GitHub. In the tracker I have built, the data is local and encrypted. Some use cases I wouldn't use lifesheets for because the data is ultra private. But the majority of my trackers could probably fit here."* — U10

Motivated by their comments, we added support for optionally storing data locally in the browser, rather than a cloud service. Madata already supported this, all that was needed was to expose it in the GUI.

91.1 %

9.8.8 In the Wild



Sun 4 Sep 2022 Excited
Sun 4 Sep 2022 Meh
Sun 4 Sep 2022 Angry
Sun 4 Sep 2022 Excited
Sun 4 Sep 2022 Meh

Total Distance 250k

Fri 2 Sep 2022
50k

Thu 1 Sep 2022
25k

Wed 31 Aug 2022
50k

Month	Number of runs
August 2022	1
September 2022	2

Number of runs per month

Mon 5 Sep 2022 at 07:00 to 16:01 (9 hours, 1 minute)
Pill half

Thu 1 Sep 2022 at 16:00 to 17:00 (1 hour)
Pill whole

Wed 3 Aug 2022 at 16:00 to 20:00 (4 hours)
Pill half

Wed 6 Jul 2022 at 18:00 to 05:01 on 7 Jul 2022 (11 hours, 1 minute)
Pill half

Figure 9.13 *Three of the lifesheets participants created for their needs during the user study. From left to right: mood tracker, run tracker, tracker for cat asthma attacks*

4/10 participants (U1, U2, U3, U4) had time to create a custom tracking application at the end of their session. They created: two mood trackers, a run tracker, and a tracker for cat asthma attacks. Three are shown in Figure 9.13 (the run tracker not shown is a simpler version of the one shown).

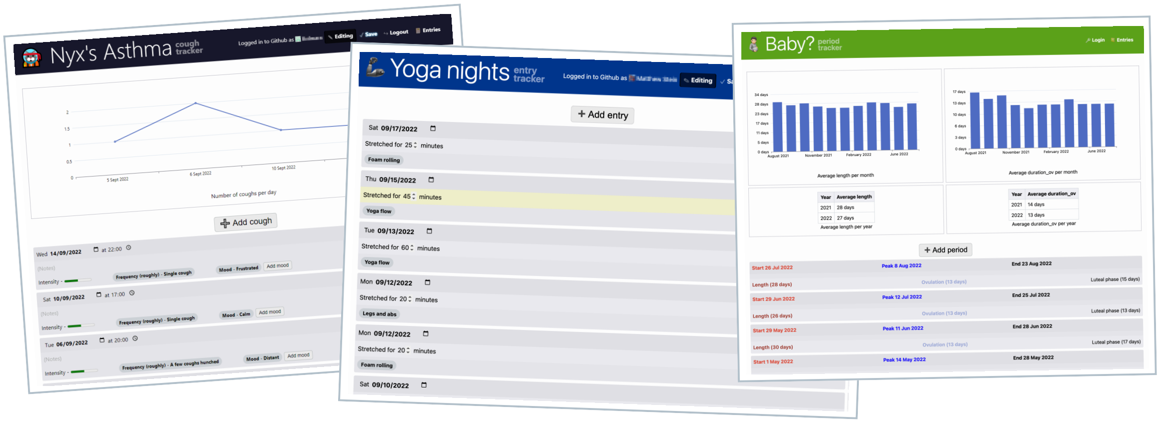


Figure 9.14 *The three lifesheets participants created for their needs in the days following the user study. From left to right: i) U4's improved cat asthma tracker ii) U6's stretching tracker iii) U7's custom period/ovulation tracker*

U4, U6, and U7 voluntarily spent time using Lifesheets on the days following the study (Figure 9.14). Interestingly, all were in the non-programmer group, indicating that perhaps even though non-programmers struggle more, they also perceive more value in a system like Lifesheets. U4 created an improved version of their cat asthma tracker with more granular data. U6 created a tracker for their stretching sessions and used it over a period of two weeks to track data until they realized they did not want to use a phone at all during these sessions. U7 created two trackers: a period & ovulation combination tracker and an app to track aspects of the manufacturing process in their materials engineering startup, to replace a legacy system. They preferred that we not screenshot the manufacturing tracker. They stopped using Lifesheets because they ran into bugs (they did not provide further details).

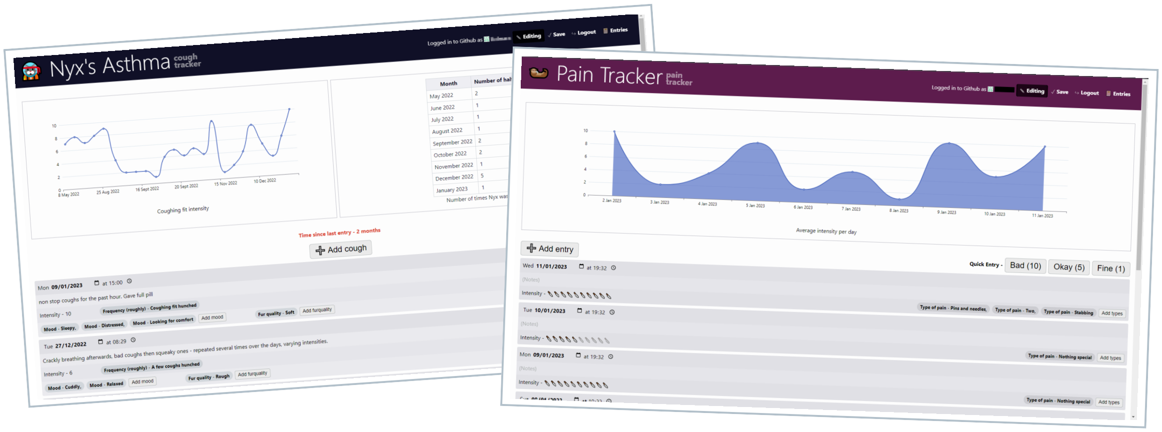
Longitudinal Follow-up Case Study

U4 used Lifesheets regularly for five months after the study. We interviewed them again seven months after their original user study session. They had continued to evolve their cat asthma tracker (Figure 9.15) and used it to track data until it was no longer medically relevant. They had placed a shortcut to this lifesheet on their browser's bookmark bar and said they recorded data immediately or at most a few minutes later. They shared this lifesheet with their veterinarian, who found it helpful. They mostly used it on their desktop computer, but also entered data through their phone if late at night.

Before Lifesheets, they were recording this data on a fridge blackboard and their phone's notetaking application. They felt that using a lifesheet improved both the rate and granularity with which they collected data (they estimated 1/15 missed entries with Lifesheets vs 3-4/15 with their previous methods). They were motivated enough that they manually ported all their older data to their lifesheet.

When asked about the reasons they preferred a lifesheet over a spreadsheet, they remarked on the efficiency of tracking data, its visual appearance (*"spreadsheets feel needlessly complicated and clunky"*), and customization (*"this feels like something I made"*). They also felt adding a field to a lifesheet required less *commitment*, whereas in a spreadsheet it would require *"a whole new column I have to scroll past"*.

At the time of the interview, they also said there were several more lifesheets they plan to create: a mood tracker, a back pain tracker, and a chore tracker. In addition to their own use cases, they talked about Lifesheets to many of their friends, and even made a prototype of a pain tracker for a close friend (Figure 9.15).



Month	Number of hal
May 2022	2
June 2022	1
July 2022	1
August 2022	1
September 2022	2
October 2022	2
November 2022	1
December 2022	5
January 2023	1

Number of times Nyx was

Time since last entry - 2 months

Mon 09/01/2023 at 15:00
non stop coughs for the past hour. Gave full pill
Intensity - 10
Frequency (roughly) - Coughing fit hunched
Fur quality - Soft
Add furquality
Mood - Sleepy, Mood - Distressed, Mood - Looking for comfort
Add mood

Tue 27/12/2022 at 08:29
Crackly breathing afterwards, bad coughs then squeaky ones - repeated several times over the days, varying intensities.
Intensity - 6
Frequency (roughly) - A few coughs hunched
Fur quality - Rough
Add furquality
Mood - Cuddly, Mood - Relaxed
Add mood

Wed 11/01/2023 at 19:32
(Notes)
Intensity -
Type of pain - Pins and needles, Type of pain - Two, Type of pain - Stabbing
Add types

Tue 10/01/2023 at 19:32
(Notes)
Intensity -
Type of pain - Nothing special
Add types

Mon 09/01/2023 at 19:32
(Notes)
Intensity -
Type of pain - Nothing special
Add types

Figure 9.15 *The lifesheets U4 worked on for the months following the study From left to right: i) Improved cat asthma tracker ii) Prototype of pain tracker*

9.9 Future Work

91.6 %

9.9.1 Templates and “forking”

While Lifesheets simplifies application building, there is still a learning curve. To fully embrace tinkerability [164, 165] it would be useful to provide starting templates for common use cases that a user could customize instead of starting from scratch. It is already possible to visit another user’s profile and “fork” their public lifesheets, but there is no discovery mechanism for other users. Perhaps there could be a “featured” users or lifesheets section to facilitate. Note that it is also possible to “fork” an existing public lifesheet by opening it in the editor and sharing the URL. This is not exposed in the UI though U6 and U7 discovered it by experimentation.

9.9.2 Autosave and Visible Edit History

91.7 %

Several participants mentioned that clicking *Save* felt foreign, as most modern applications autosave. The main reason Lifesheets require explicit saving is that data is saved in a remote version controlled system (GitHub) and every save creates a history entry (commit). We wanted to avoid polluting that history with too many entries, so that it’s usable for data recovery, as we plan to expose that history in the UI. Reasonable compromises could be: configurable autosave, persisting data locally until saved remotely, and/or coalescing minor consecutive edits into one history entry.

9.9.3 Reminders

91.8 %

4/10 user study participants (U3, U6, U9, U10) mentioned reminders as the one missing piece for them to fully adopt Lifesheets. This was also a popular complaint about existing tools in the survey, and is a prevalent need in the literature. In line with its programmable philosophy, Lifesheets could support optional notifications with either static values or expressions This could easily be a setting on the Lifesheets editor so that users can opt-in and customize the kinds of reminders they would receive.

9.9.4 Locations

Location is the only data entry type that was identified as common in [141] but Lifesheets does not yet support. An ideal implementation would support locations of various granularities (country, city, specific address, coordinates), as well as place names (e.g. specific restaurant). Suitable expression functions should be added to extract metadata from location properties (e.g. timezone, country, city, etc.), and suitable default values to automatically set from the current location.

9.9.5 Timezones

91.9 %

While Lifesheets abstracts temporal math into a high level setting, there is one missing piece: timezones. Formula^2 automatically takes timezones into account when specified, but currently, all tracking is presumed to happen within a single timezone. Date/time properties should support optionally specifying a timezone, with presets that read it from another Location property. This would also allow different start and end timezones, e.g. for travel.

9.9.6 Correlations

A common motivation for tracking is to find correlations between different kinds of tracked data. [167–169]. This was also requested by U4 and U9. Visual correlations can be supported by allowing data from other lifesheets to be "mounted" to the current one, and plotting their data in the same chart. Going further, Lifesheets could even automatically detect correlations, without requiring any user hypothesis.

9.9.7 Semi-automated Tracking

92 %

While Lifesheets works well for manual data entry, which—as discussed—has several benefits, semi-automated tracking [134], a combination of manual and automatic tracking, maintains these benefits but reduces the high data capture burden of manual tracking. It would be useful for lifesheets to be able to connect with external services and sync data, which could then be augmented further by the sheet. This was also requested by U4 and U9.

9.9.8 Collaboration and Competition

Right now there are two levels of privacy: in public lifesheets everyone can read the data but only the author can edit, and in private ones only the author can read or edit data. In the user study, U10 asked *"How do I share a private lifesheet with my wife?"*. There is no way to share a private lifesheet with other users through the Lifesheets interface. Technical users can do so through GitHub, but since all lifesheets are currently stored in the same repository, this would share *all* of their private lifesheets. Lifesheets could address this by optionally storing lifesheets in separate repositories and providing UI for managing collaborators.

U9 suggested allowing users to "compete" on certain publicly tracked metrics (e.g. exercise), to encourage themselves to do better.

9.9.9 Predicting Future Entries

92.1 %

For many tracking cases, predicting future occurrences is the primary motivation for tracking (*"to find patterns in the data and get insights"* was the most popular reason for self-tracking in our survey). Currently this can be done by encoding the prediction logic in expressions, but Lifesheets could provide a more sophisticated, high-level way out of the box.

9.10 Conclusion

92.2 %

In this chapter, I presented Lifesheets, a domain-specific End-User Programming system for creating personal tracking applications with minimal technical skill. We discussed the results of our needfinding study, which looked into the tracking needs of 85 people. We evaluated Lifesheets through case studies and a lab study of 10 participants, who all rated Lifesheets as more usable than spreadsheets for application creation, data entry, and reflection. We believe that Lifesheets can empower individuals to achieve their goals of preserving the data that matters to them about themselves and their loved ones, reflecting on them, and reaching higher levels of self-knowledge. Beyond evaluating Lifesheets, the lab study provided several insights on the usability of Formula^2, the viability of the vision behind Madata, and the learnability of Mavo concepts like properties and expressions embeded in static content.

CHAPTER 10

Discussion & Future Work

5,820 words (17 min read)

The preceeding chapters have probably spawned many questions and hopefully a few good ideas. This chapter offers a discussion of some of the higher-level concepts, limitations, and possibilities surrounding this work, and suggests some directions for future research.

There are three main categories of current limitations:

1. **The low-hanging fruit**: these can be overcome in relatively obvious ways, and it is a simple matter of specifying and prototyping the behavior. For these, we sketch out potential solutions and the tradeoffs of each.
2. **The harder problems**: these limitations are not inherent to the approach, but would require *sustantial* design work to be solved well. For these, we outline potental directions for future research.
3. And last, **the fundamental limitations**: these are inherent to the approach itself, and it is likely they may not be solvable within it, at least not without very major changes to Mavo or the environment. We describe these as well. Every scientist should have a healthy skepticism when faced with statements about the impossibility of a certain problem, and this should be no exception. It is entirely possible that even for this latter category, more research may reveal creative ways to work around them that are not currently obvious. However, they certainly would require more work and creativity to tackle than the first two categories.

10.1 Can This Model Build Non-Trivial Applications?

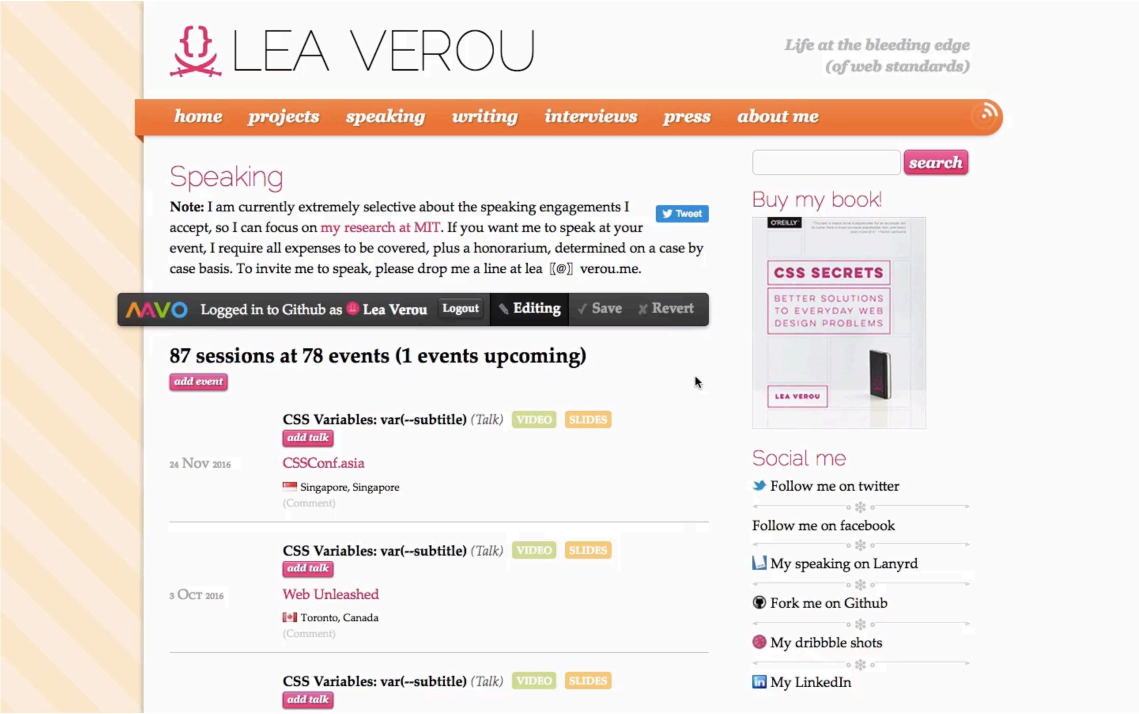
92.8 %

Empowering novices to create the small-scale data-driven applications they envision for their needs is already a worthy goal. But once this becomes possible, the question arises: **what are the boundaries of this approach?** Could it possibly be used to prototype real

-world applications like Facebook or Amazon? What about games like Fortnite or Minecraft? Or graphics editors like Photoshop or Illustrator?

Clearly, for any language that allows imperative programming as an escape hatch, the answer is yes, but that answer is of similar utility as the assertion that we can build these applications with any machine able to manipulate symbols on an infinite tape [170]. A more meaningful question would be whether Mavo could add enough *value* to substantially simplify the development of such applications. We explore several facets of this question below.

10.1.1 Scope



Speaking

Note: I am currently extremely selective about the speaking engagements I accept, so I can focus on my research at MIT. If you want me to speak at your event, I require all expenses to be covered, plus a honorarium, determined on a case by case basis. To invite me to speak, please drop me a line at lea [@] verou.me.

87 sessions at 78 events (1 events upcoming)

CSS Variables: var(--subtitle) *(Talk)*
24 Nov 2016 CSSConf.asia
Singapore, Singapore
(Comment)

CSS Variables: var(--subtitle) *(Talk)*
3 Oct 2016 Web Unleashed
Toronto, Canada
(Comment)

CSS Variables: var(--subtitle) *(Talk)*

- Follow me on twitter
- Follow me on facebook
- My speaking on Lanyrd
- Fork me on Github
- My dribbble shots
- My LinkedIn

Figure 10.1 *The very first Mavo demo: A web application to manage and publish a person's list of past and future conference talks*

The first Mavo use case was a list of conference talks (Figure 10.1), and the dozens that followed in its first few years were of a similar scale and spirit: small-scale, mostly CRUD applications plus very lightweight computation for editing structured data in a high fidelity WYSIWYG interface, with few editors and any number of consumers (readers). For these use cases, Mavo was envisioned as an easier yet more flexible, more lightweight, and more portable alternative to CMSes, that still provided higher fideity editing interfaces than said CMSes.

Over the years, as its formula language (Formula²) and data I/O (Madata) developed, and with the introduction of data actions, Mavo has become capable of building applications that break out of this mold. It has been used to build games, color pickers, interactive graphics editors, code playgrounds, social feedback apps, chat applications, and more. Many such examples are described in the Case Studies chapter.

However, applications whose main complexity lies in their *interactivity model* rather than their *data model* are still largely out of scope. For example, a drawing applications, or a VR game. Applications requiring parts of such interactions can still be built, if these interactions are encapsulated in web components, but Mavo is likely not a good fit for applications where these types of interactions are central.

10.1.2 Scalability

Because Mavo is implemented as a pure JavaScript library and all computation occurs on the client, serving a Mavo app to any number of users is as easy and scalable as serving static web pages. Scalability issues arise only around access to the *data*, which may be stored locally or outsourced to third-party storage providers such as Dropbox.

Mavo is therefore perfectly suited to so-called *Personal Information Management (PIM)* applications. These applications have a single author and reader, and the amount of data they manage is generally small. For the ultimate in scalability, the Mavo app web page can be stored ("installed") on the user's own machine and data stored locally in the user's browser. While this old fashioned approach sacrifices the access-from-anywhere advantages of cloud-based services, it frees the user of any dependence on the network. Even when operating in the cloud, PIM-oriented Mavo applications scale extremely well because each user's data is isolated. Each user's Mavo simply loads or stores their own small data file, which is the bread-and-butter operation of the popular storage services. A peer-to-peer synchronization service for web storage would allow users to manage information on all their devices while still avoiding dependence on any cloud services.

Mavo is also well suited to "web publishing" applications where an author manages and publishes a moderate-size hierarchical data model and present it to audiences of any size through views enriched by computation of scalar and aggregate functions over those items. This large space spans personal homepages, blogs, portfolios, conference websites, photo albums, color pickers, calculators, and more. Since only the author edits, these applications scale like the PIM applications for editing, while on the consumption side any number of consumers are all simply loading the (static) Mavo application and data file, which again is highly scalable. Conversely, Mavo can be used to supercharge web forms that *collect* information from large numbers of individuals—such as surveys and contact forms—to adapt dynamically to inputs and perform validation computations.

Mavo was not originally designed to make social or big data apps that present every user with the results of complex queries combining data from many users. This social/computational space is important, but so is the large space of "small data" applications that Mavo can provide. Even on big-data applications, Mavo may in the future be a useful component for simplified UI design if powerful back-end servers are used to filter down and deliver only the small amount of data any given user needs in their UI at a given time.

10.1.3 Data Model Structure

Mavo is designed to work well with **hiearchical data models**, such as those that can be represented as JSON objects. This is a superset of tabular data models, as a table can be expresssed as an array of objects. However, a reasonable question is **how could Mavo handle graphical schemas**, which per [27] occur in 27% of the applications they studied — though only 22% of these cyclic (6% of the total dataset).

Mavo can already express graphical data models, but it requires a lot of manual effort (an example is described in Section 8.1.4). Authors need to designate a certain property of each object as (conceptually) a primary key, peform manual effort to ensure it is unique (e.g. via suitable formulas doing data validation), write custom Formula² expressions to look up objects by this key and display them elsewhere, and author dynamic collections to facilitate data entry of foreign keys.

Inspired from the several decades of database system design, Mavo could introduce language primitives that make primary and foreign keys first-class citizens, which would greatly simplify this process.

These could include:

- **Enforcing uniqueness**: A way to specify that a certain property needs to be unique
- **Primary keys**: A way to designate a certain property as the one that uniquely identifies an object (which would also enforce uniqueness)

- **Foreign keys**: A way to designate a certain property as a *foreign key* that references another object's primary key.

Authors could then either template the referenced data in a custom way, or when this power is not needed, Mavo could provide a default rendering that summarizes the object without requiring authors to write a new template for it (possibly by reusing the same template as in the defining collection, but adding a certain CSS class than can be used as a styling hook).

These kinds of *implicit objects* can be useful for more than foreign key references. For example, entering a URL in a property could expose an object with metadata about the website (e.g. title, description, image), or entering an address could expose an object containing metadata about the location (e.g. GPS coordinates).

More research is needed about the best mental model and syntax for expressing these concepts in a way novices can understand.

93.7 %

10.1.4 Data Model Granularity

A big current limitation of both Mavo and Madata is the data model *granularity*. Fetching data from the server is typically only done once: when the page is loaded. The data powering a Mavo app is fetched as a whole (often from a JSON file) and stored as a whole, even if the app only needs to read and edit a subset (via `mv-path`). Access Control is also expected to function at this level: a user can either read or edit the whole dataset, or no part of it.

These limitations are no issue for the use cases that drove Mavo's design, such as personal information management applications, calculators, and web publishing applications, where the data is small and the editors are few and know each other. However, as authors who liked its novice-friendly syntax tried to push the boundaries of what was possible, these limitations started to become more restrictive.

Due to its distributed architecture, supporting data models with finer granularity is not a simple matter of programming, due to the number of moving parts involved. First, not all backends support finer granularity. The types of file-based backends that power the cloud storage of most Mavo applications (Dropbox, GitHub, Google Drive) typically only support reading and writing a single file.

It is important to tease apart the use cases that benefit from granularity, as solutions may vary depending on the need:

- **Security & Privacy**: Many multi-user applications require granular access control at the item or even property level, rather than the dataset level to protect sensitive data or prevent malicious usage.
- **Usability**: Improve the UI for collaborative use cases by hiding irrelevant elements or making certain properties non-editable
- **Conflict Minimization**: For multi-user applications, it is important to reduce the number of conflicts that can occur when multiple users edit the same data to a minimum, which requires frequent synchronization and push updates.
- **Performance**: The larger the dataset, the slower it is to fetch and save it in its entirety.

We now discuss some potential directions for future research in these areas.

Granular Access Control: Security, Privacy, and Usability

Presentational Access Control (PAC)

Even for Mavo's originally envisioned use cases involving small groups of editors acting in good faith, restricting edits of certain parts of the data to certain users can help prevent accidental data corruption, and hiding irrelevant elements can streamline the UI.

This is a lighter form of granular access control (henceforth referred to as *Presentational Access Control (PAC)*), as it does not actually require server-side enforcement, since none of the users involved is malicious.

PAC is largely already *possible* in Mavo, though not necessarily *easy*:

- `mv-if` can be used to hide elements based on formulas, and formulas can take include user information into account (e.g. username of current user).
- `mv-mode="read"` can be used to prevent certain subtrees from being editable, and its value can be an expression that depends on user information.
- Separate JSON files can be used to define groups of users and their permissions, and Formula^2 expressions can be used to check these permissions.

One conspicuous absence is the lack of a way to disable certain collection management operations (e.g. conditionally disabling item deletions, while still allowing additions). There are ways to hide the controls for these operations (e.g. the `mv-item-bar` attribute, or simply CSS), but this communicates a different *user intent* than disabling them (e.g. we may be removing them because we have implemented these functionalities via different controls). A primitive to disable the actual *action* would be higher level and thus more expressive and more future-proof.

It is important to note that **there are no primary use cases for PAC**, i.e. no use cases *require* it or benefit from it. There are only use cases for which it is an acceptable *workaround*, and use cases for which it is not. Furthermore, PAC workarounds need to be designed with care to avoid users assuming they are secure and building applications that contain security vulnerabilities.

Enforced Access Control

PAC may suffice for small-scale collaborative use cases, but for the types of multi-user applications mentioned in the beginning of this section, server-side enforcement of access control is necessary.

There is a cornucopia of very common multiuser use cases that require this kind of granularity. Some examples include:

- A comment section on a blog post: any user can add a comment, but only the post author can edit or delete it. *(restricted writes)*
- A survey, where anyone can submit a response, but can only see and edit their own response. *(restricted reads and writes)*
- A chat or IM application, where anyone can send a message, but only edit or delete their own messages.
- And of course, any social network.

In cases like these, it is not enough to merely hide the UI elements that allow editing — permissions need to be *enforced* by the backend. This surfaces a tension that our approach creates: by not controlling the backend, but merely the *choice* of backend, Madata is also bound by the limitations and capabilities of these backends. Very few of the backends that Madata currently supports have the ability to enforce fine-grained access control. For example, there is no way to instruct GitHub to only allow writing certain parts of a JSON file to certain users, or even certain files.

However, back-end services with richer access models exist. For instance, DataHub [171] provides row-level access control, where each table row is “owned” by different users. Similarly, Firebase [172] is a popular commercial hierarchical database, where each node can have its own access control rules (and is already supported by Madata).

Even though Madata can be used with backends like these (and already supports some), it does not yet support reading or reflecting granular permissions in its objects. Similarly, Mavo can be used with such backends, but the author needs to take care of implementing suitable access control on the backend service and then reflecting it in the UI (via the same patterns as those used in PAC).

Even if we only consider backends that support granular access control, there are several pieces that need to fall into place for the Mavo ecosystem to properly support it:

1. Madata needs to support **reading** these types of permissions from the backend (for the current user) and exposing them to the application developer in a unified format that is backend-agnostic.
2. Mavo needs to support reading granular permissions from Madata and automatically **reflecting** them in the app UI

These steps would already be a big improvement, as they would allow users to define elaborate access control rules in their backend service, and everything else would *just work*. However, typically the systems for defining these permissions require a higher level of expertise than the typical Mavo author possesses, and the syntaxes for defining these rules suffer from the same incompatibilities Madata was designed to normalize.

What if we could go further, and **have the Mavo app become the source of truth for these permissions**? Mavo could provide a novice-friendly syntax for authors to *define* granular access control rules in their Mavo templates, and Madata would define a backend-agnostic abstraction to describe them. Mavo would then be able to communicate these permissions to Madata, and the Madata backend would translate them to the appropriate backend-specific rules.

A Declarative Syntax for Granular Access Control?

This brings us to the question: if novices could declaratively define access control rules in their Mavo template, and have then be translated to the appropriate backend-specific rules, what would that syntax look like?

There are two potential directions:

1. A completely declarative micro-syntax, with keywords for permissions and roles (e.g. `mv-can-edit="own"`)
2. A formula-based approach, where elaborate logic can express complex rules (e.g. `mv-can-edit="post.owner = $username"`).

The first approach is has a lower threshold and is easier to translate to different backends, but also has a lower ceiling, and thus may not be expressive enough for all use cases. The second approach has a higher ceiling, but also a high threshold and imposes a higher burden on supporting additional backends, putting its generalizability at risk.

A **layered approach** could involve a hybrid of the two: a declarative syntax at first, covering the most common use cases, that can later expand to a formula-based approach for more complex rules, where the initial keywords (e.g. `everyone`, `own`) are translated to suitable formula expressions.

Two Backends, One App?

Rather than restricting granular access control to the few backends that natively support it, a different direction would be to allow a Mavo app to be backed by *multiple* backends, by extending the `mv-storage` attribute to apply to *any* property. This would facilitate granular access control by *partitioning*.

For example, it would make it possible to create a blog where the posts are stored in Dropbox and can only be edited by the author, with upvotes stored in a service that allows public writes. Or, a designer portfolio where project metadata is stored on GitHub in a public repository, but billing details and client notes are stored in a private repository.

Sufficiently granular partitioning, if done well, could solve a very wide range of use cases. For example, the blog comments use case could be solved by storing each user's comments in their own file space (e.g. a fork of the GitHub repository). This is not without its own set of challenges, which have been studied in the context of distributed systems, and are out of scope for this work.

It could be argued that this kind of partitioning is an *eigensolution* [173]: it does not only solve (many cases of) granular access control but also several other use cases, such as *mashups*. For example, a user could create a book reading log, where they store their ratings and notes in a personal Dropbox file, and display book metadata from a third-party API. Or a recipe manager, where the recipes are stored in a personal Google Drive file, and nutritional information is fetched from a third-party API. Mashups are already possible with Mavo, by using a separate Mavo app for each data source, and Formula² expressions to join the data, but the authoring experience is suboptimal.

Squint and Everything Becomes a Storage Backend

A creative solution (which can be combined with other ideas) is for Madata to support storing data in *"backends"* that are not storage services in the typical sense, and take advantage of their more powerful access control mechanisms.

For example, one could imagine a Madata backend that stores its data in GitHub Issues, with each comment storing JSON for a collection item. Or as comments on a hidden Facebook post.

This gives Madata developers (and thus, Mavo authors) access to a storage mechanism that supports the commonly needed paradigm where any registered user can append, but the data owner (and admins) can edit or delete, without needing to configure any access control rules or set up complicated services.

Incremental Data I/O: Conflict Minimization & Performance

Multi-user applications — especially realtime ones — require robust conflict resolution to scale, and larger datasets are slower to load and save. The typical solution to both is granular data reads and writes, where only the data that is needed is fetched or saved rather than the entire dataset. The less data that is transmitted over the wire, the faster data I/O will be, and (auto)saving granular changes frequently and pushing granular updates to the UI can reduce conflicts to a minimum that can be resolved via the UI.

For backends that support these capabilities, this becomes a matter of Madata (a) reading that they exist and interfacing with them, (b) exposing them to the application developer in a backend-agnostic way, and (c) Mavo handling such updates and propagating them to the displayed data.

Mavo already tracks edits in a granular way, so that it can communicate unsaved changes to the user, so it would be trivial to save these changes incrementally if Madata supported it. Similarly, applying partial updates can be relatively simple by recursively updating the data model in places where it has not been edited by the user.

However, this needs partial updates to be communicated by Madata as a hierarchical, ordered list of operations. This is because there is no way to know the identity of an object in a JSON file, as neither Mavo nor Madata enforce any kind of primary key. Approaches like JSON Patch [174] or some types of CRDTs [175] could be used for this purpose.

As with granular access control, this becomes a lot harder when the backend does *not* support these capabilities. For backends that do not support push updates and incremental writes, Madata could emulate them by polling the backend for changes, diffing changes with the fetched version of the data, then applying that diff to the displayed data. Similarly with saving, Madata could refetch the remote data, apply the user's changes to it as a patch, and save that result. This would not help with performance since it still involves fetching and saving the entire dataset (in fact, it would make it worse due to the additional diffing/patch steps), but it would help with conflict minimization.

10.1.5 Platform Limits

95.3 %

No discussion on the limits of Mavo would be complete without discussing the limits imposed by the platform itself. As a client-side extension, there are certain things Mavo simply cannot do, at least not without interfacing with a server-side component. The vast majority of these limitations are well-intentioned security protections, such as the same-origin policy [119] which prevents client-side code from reading HTTP responses from other origins [120], however it is unfortunate that there is currently no way for end-users to opt-out of these protections for certain trusted applications, which gives proprietary native platforms a competitive advantage.

Websites can opt-out of the same-origin policy for requesting sites by setting **`Access-Control-Allow-Origin`** and other CORS [100] HTTP headers. However, as with any opt-in mechanism, many neglect to do so. Additionally, CORS does not disable all cross-origin protections. For some, there is no opt-out mechanism, such as reading the contents of cross-origin iframes.

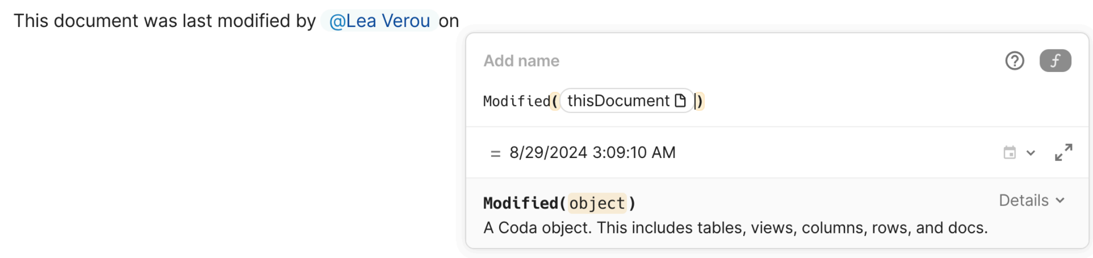
For example, it was mentioned in the Portfolio case study, while Mavo *could* generate thumbnails for uploaded images, it would not be able to do so for linked images. This also restricts the number of APIs that Madata can support, as many do not provide CORS headers.

10.2 Formulas as Data?

95.4 %

Spreadsheets blur the line between data and computation, with formulas used ad hoc in cells, and columns being able to contain a mix of data and formulas. This design trades off maintainability for flexibility. Repeating a formula across a column is managed by essentially copying it to each cell, with the UI providing affordances to make this less painful, and there is no way to ensure all copies of the formula stay in sync. However, it also it allows creating one-off exceptions, which can be useful in a world of messy data.

Most data-focused no-code tools take a different approach for tables: columns are either data or formulas. This is decided upfront, and each cell automatically follows the same rule. This design makes the opposite tradeoff, sacrificing flexibility for maintainability.



This document was last modified by @Lea Verou on

Add name

```
Modified( thisDocument |)
```

= 8/29/2024 3:09:10 AM

```
Modified(object)
```

A Coda object. This includes tables, views, columns, rows, and docs.

Figure 10.2 *Entering an inline formula in Coda*

An interesting pattern in some no-code tools is to allow formulas to be entered alongside rich text (e.g. entering `=` while writing kicks the editor into formula mode), to interleave one-off computations with prose (Figure 10.2).

This is a very powerful pattern, and is currently not possible in Mavo. It remains an open question how to best support this pattern in Mavo, likely in an opt-in way.

95.6 %

10.3 Higher-level Primitives for Data Exploration (Grouping, Filtering, Sorting)

Creating custom filters via dynamic collections and Formula^2 expressions has been possible in Mavo almost from the start. An example of this can be seen in the CSS WG Disposition Of Comments application.

Thanks to work by Sanchez D. [176], sorting via an `mv-sort` attribute (and corresponding Formula^2 `sort()` function) and grouping via the Formula^2 `by` operator became possible as well. There is no HTML syntax for grouping, making grouping *in place* tricky but possible: (it would require sorting by the grouping expression and inserting content with its value before the first item that has a different value).

While this approach affords a high ceiling, it also imposes a high threshold. Writing the logic for one's own filters is nontrivial, tedious, and error-prone, even for Mavo, and novices do not necessarily have the training to make good UI decisions in this area.

Higher-level primitives for data exploration such as those provided by Exhibit [7] could greatly simplify this process. Ideally these would automatically generate a suitable UI based on the shape of the data and the schema of the data model, with escape hatches to customize the UI if needed. Entirely custom widgets can still be used for more complex use cases, and even shared with others if abstractions (see Section 10.4) are added to the language.

Orthogonally, better view-update handling by Mavo could make it simpler to build custom data exploration UIs and still have editable data, not readonly copies.

10.4 End-user Abstractions

There is a scarcity of abstraction and reuse mechanisms across most no-code or low-code tools. Extension points typically involve use of scripting languages, rather than ways for users to compose and reuse primitives of the language or tool they are *already* using. Part of this may be lack of demand; we know that novices and end-user programmers prefer cloning over abstractions [88], and it is something we have also observed in our user studies (Chapter 7).

However, to use the example of the most successful end-user programming environment, spreadsheets have evolved a whole continuum of users, from expert programmers to individuals who only use spreadsheets created by others and do not do any formula writing themselves [177]. Abstractions would allow such power users to *share* their creations with others, enriching not just their own experience, but that of the entire community.

Complex Mavo applications can become difficult to manage due to the limited abstraction and reuse mechanisms. Currently, Mavo can build some pretty complex interactive widgets, but there is no mechanism to reuse them and compose them into larger applications besides cloning, which restricts its ceiling, since there is only so much that can be done in a single HTML file. In terms of its *abstraction gradient* cognitive dimension [76], Mavo is closer to *abstraction-hating* than *abstraction-tolerant*.

In this section, we discuss some potential directions for future abstraction and reuse mechanisms in Mavo.

Last, for no-code abstraction mechanisms to be successful, it is not enough to simply implement such mechanisms. The tool also needs to *teach* users how they can benefit, e.g. by providing ways to convert existing duplicated structures to use an abstraction, or even automatically identify good candidates for abstraction. This is certainly easier for visual builders like Lifesheets rather than languages like Formula² or Mavo HTML.

10.4.1 Reusable Components

96.1 %

The extension point of plain HTML is custom elements, implemented via JavaScript. Therefore, a natural extension point for Mavo HTML would be to allow users to define their own custom elements, which encapsulate Mavo functionality. Ideally, this process would be able to create regular custom elements that can be used *without* Mavo.

10.4.2 User-defined Functions

Authors can already define *computed properties* whose values are Formula² expressions. Functions are a natural next step, if viewed as *parameterized computed properties*.

Existing literature for adding custom functions to spreadsheets [178, 179] has explored ways to smoothly transition from the specific to the abstract, by allowing users to convert a regular cell formula to a parameterized function.

This approach could work well for Mavo too. Mavo already supports groups of properties (objects) some of which can be computed. Only two things are missing to turn such a group into a function: (a) A way to map input arguments to group properties, and (b) A way to select a specific property as the one containing the return value.

This could be opt-in (certain groups are marked as functions), or all groups could also be functions. The latter also opens up interesting interactions with Formula²'s scoping rules: *If scope can vary function code, is that a feature or a bug?*

10.5 Visual Mavo Builders and Direct Manipulation

96.2 %

Lifesheets (Chapter 9) has shown that Mavo concepts can reach a much wider audience when exposed visually. We believe this is a very small first step towards this goal. A higher fidelity visual interface could go a lot further in visualizing the data model, Formula²

’s identifier resolution algorithm, introduce more direct manipulation interactions, and elimiate syntax even further.

Additionally, Lifesheets was very narrowly scoped to personal tracking, but it became clear that many of its ideas could be generalized to a more general no-code visual builder. Its affordances for editable properties, formulas, actions, or charts have no particular dependence to the personal tracking domain. It remains an open question where the boundary lies between a GUI that is too general to be helpful, and one that is too specific to cater to most use cases.

10.6 Towards a Declarative, Transparent Web

96.3 %

Anyone visiting a webpage could peek under the hood, see its inner workings, copy and repurpose the code, and ultimately learn from the experience. Editing this HTML code did not produce any errors, and rarely if ever produced non-local failures. Many of today’s software engineers got their humble beginnings by tinkering with HTML and CSS in this way. The so called *tinkerability* [164, 165] of the early Web served as a gentle introduction to programming for many of today’s software engineers.

Today, while *“View Source”* still exists, it is of far more limited utility. More often than not, the HTML that reaches the browser is the obsfucated result of a complex build process, or a stub for a JavaScript framework to fill in.

We hope for Mavo to serve as a first step towards restoring some of this tinkerability of the early Web. Mavo applications are completely transparent: their UI, their logic, their storage location, and frequently their data, are all visible in the HTML. Tinkering with them does not require wading through dozens of JavaScript modules or understanding complex architectures, but merely looking at the HTML like it’s 1999.

But beyond learnability and tinkerability, there are other benefits to this approach, which hint at many possible future directions of research.

10.6.1 Interoperability

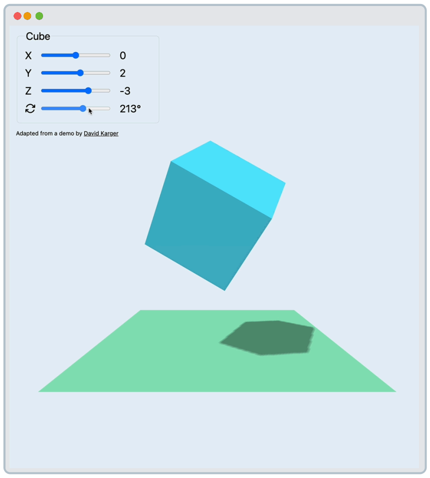
96.5 %

A big advantage of HTML-based approaches is that they are **naturally interoperable**, without requiring any plugin or integration to be written: the HTML itself is the integration point.

We saw an example of this in the e-shop case study (Chapter 8), where Mavo could interface with PayPal, a payment provider, without neither Mavo nor PayPal needing to know about each other, simply because they both spoke the same language: HTML.

HTML

```html
<a-scene>
   <a-box position="[boxX] [boxY] [boxZ]"
          rotation="45 [boxRotation] 45"
          color="#4CC3D9" shadow></a-box>
   <a-plane position="0 0 -5"
            rotation="-90 0 0"
            width="4" height="4"
            color="#7BC8A4" shadow></a-plane>
   <a-sky color="#E9F2F9"></a-sky>
</a-scene>
```



Cube

X 0

Y 2

Z -3

213°

Adapted from a demo by David Karger

Figure 10.3 *Using Mavo and A-Frame to create an interactive parameterized 3D scene with HTML. Interactive demo at mavo.io/demos/aframe.*

Another example is Web Components: any web component can be integrated with Mavo simply by using it in the Mavo HTML. An extreme example of that is this Mavo - AFame demo (Figure 10.3), where a 3D scene is created using the A-Frame web component library by Mozilla, and Mavo is used to parameterize it.

Last, as we saw in the SVG Path Builder case study (Section 8.2.1) this interoperability extends to other markup languages that can be embedded in HTML, such as SVG or MathML.

97 %

10.6.2 Accessibility

A declarative language that describes the application at a high level rather than the low-level steps to achieve its various interactions, can automate many things that are currently laborious and require expert knowledge.

An important example is **accessibility**. In the State of HTML 2023 survey [109], web developers cited lack of knoweldge and low organizational priority as some of their core pain points with making their web applications accessible to people with disabilities. The more that is known about the application, the more of accessibility can be automated, rather than depending on the individual author's knowledge and effort. Mavo already adds several ARIA [180] annotations to the author's HTML, but there is a lot more that can be done in this area.

10.6.3 Longevity

97.1 %

Declarative languages are better positioned for **longevity**. The very first page on the Web, created by Tim Berners-Lee in 1991, is still viewable and usable today. In contrast, most JavaScript-heavy applications from only a few years ago are already broken. One reason is that declarative languages can be standardized and used by a variety of competing but interoperating tools. Another is their fault tolerance; the HTML of this first website is now considered invalid. In fact, approximately *half* of its source is highlighted as an error or warning by the W3C Validator. And yet; every modern browser can render it flawlessly. This should be contrasted with the behavior of most imperative languages, where a single syntax error can break the entire application.

> ASIDE
>
> While it is easier to design fault-tolerant declarative languages than imperative ones, not *every* declarative language is fault-tolerant. In the beginning of the 21st century the tide had briefly turned *against* HTML's fault tolerance, and towards strict syntaxes with draconian error-handling. *Architecture of the World Wide Web* [181], published by the W3C Technical Architecture Group in 2004 echoes this contemporary philosophy:
>
> > *"Agents that recover from error by making a choice without the user's consent are not acting on the user's behalf."*
>
> XML (and its HTML serialization, XHTML) was the poster child of this philosophy, and the effects of these design decisions ripple through the Web to this day. Thankfully, it did not take long for most to realize that this philosophy was not practical for the Web, and the tide turned back a few years later.

10.6.4 The Future of Code Generation?

97.4 %

In recent years, LLMs are increasingly used to automate the authoring of code. This does not make approaches such as Mavo obsolete — quite the contrary. Humans still need to *verify* the generated code, and to do so, they need to *understand it*. The more readable and transparent the generated code, the more it facilitates this human verification step. We predict that code readability will become a top priority for future syntaxes, while efficiency of authoring will be less of a concern. We think that declarative languages like Mavo are excellent candidates for this trend. Our Lifesheets study (Section 7.6) has already touched on this, by making Mavo code more readable and editable by novices who would not have been able to author it. A lot more can be explored in this direction.

CHAPTER 11

Conclusion

318 words (1 min read)

Software development is currently a privilege enjoyed by a small minority of people. For the rest, using a computer or phone translates to using prefabricated experiences created by developers, with little hope for modification. The ability to shape the software you use or create new software from scratch is for most people indistinguishable from magic.

But there is no fundamental reason for software development to be a privilege enjoyed by few. It is merely a **failure of design**: languages and technologies that are overly complex, opaque, brittle, designed by and for experts, with little regard for the needs and mental models of novices.

Mavo aims to be one step towards demystifying this magic, and to add to the small but growing literature of programming languages and systems designed with HCI principles and methods.

This thesis has presented Mavo, a suite of languages and systems that reduce the barriers novices face today in managing, storing, sharing, and transforming data on the Web, and to facilitate web authoring practices that favor transparency and decentralization.

Mavo consists of three components, which are separate contributions: Mavo HTML, a declarative HTML extension for specifying data-driven web applications; Formula^2, a new formula language for operating on hierarchical data in aggregate; and Madata, a decentralized protocol and client library to reduce the complexities of remote data storage down to a single URL and a unified API.

The three components work together synergistically to enable end-users to create high-fidelity web applications with minimal effort. They have been designed with the design principles that made the Web successful in mind: smooth ease-of-use to power curve, and fault-tolerance. The Lifesheets experiment hints that paired with a visual interface, these concepts can become even more powerful and far-reaching.

Taken together, these languages and systems offer new ways of thinking about building and editing software on the Web, and the full potential of these ideas remains to be explored.

APPENDIX A

Personal tracking needfinding survey

2,455 words (8 min read)

A.1 Needfinding Survey Details

A.1.1 Population

Participants were recruited through a call for participation on social media (Facebook, Reddit, Twitter) and local parent groups. The survey received 85 responses in total, 42 being parents or guardians of at least one child. 22 were female-identifying, 32 were male-identifying, and 2 identified as non-binary. 29 chose to not disclose their gender. Median age was 39 ($\bar{x} = 41.4$, $\sigma = 9.3$).

A.1.2 Questions and Survey Flow

To jog participants' memory, the survey began by presenting a list of 26 common tracking cases, collected by browsing tracking applications on the Apple and Google App Stores (keywords: "tracking", "tracker", "journal", "logging") with five freeform fields at the end. For each, participants could select wither they had tracked it manually, automatically, or have wanted to track it (but didn't) at any point in time. These options were non-exclusive, to account for semi-automatic tracking [134] and different attitides for different time periods. Parents were additionally presented with another 13 parenting-related use cases, with four freeform fields.

Respondents could then provide more details about their selections. There were no further questions for things they selected they track automatically. The full list of questions can be found in Section A.2.

A.1.3 | Things Tracked

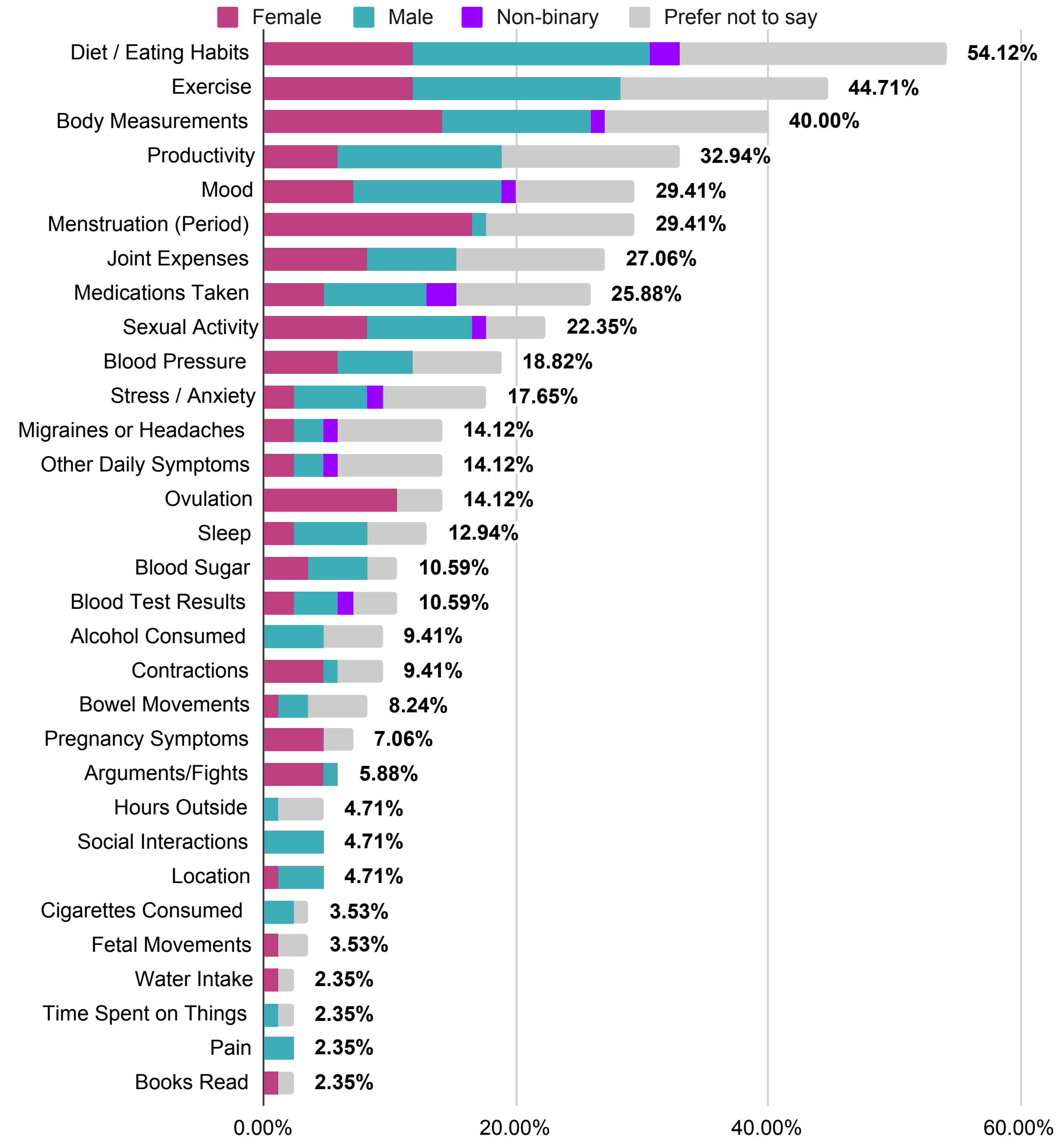


Figure A.1 *Self-tracking use cases from the survey, ordered by percentage of respondents that manually tracked them.*

Most participants were experienced trackers, self-tracking a median of 7 things ($\bar{x}$ = 8.4, σ = 5.3). This did not differ between parents and non-parents, but parents *additionally* tracked a median of 4 things ($\bar{x}$ = 4.8, σ = 4.1) about their children. While self-tracking use cases were almost equally split between manual and automatic tracking (median of 4 ($\bar{x}$ = 5.1, σ = 3.6) vs 3 ($\bar{x}$ = 3.3, σ = 2.7)) parental tracking was almost exclusively manual with only a median of 0.5 ($\bar{x}$ = 1.2, σ = 1.9) automatically, indicating perhaps that despite the rise of "baby wearables"[149, 150], parents keeping track of their children's development is largely still a manual labor of love.

Female-identifying people self-tracked slightly more than male-identifying people: a median of 8.5 things ($\bar{x}$ = 9.4, σ = 5.1) vs a median of 6 things ($\bar{x}$ = 7.6, σ = 5.1) respectively. However, when we look at parental tracking, the picture if vastly different: a median of 6 ($\bar{x}$ = 6.4, σ = 4.6) things tracked by mothers vs a median of only 2.5 ($\bar{x}$ = 2.9, σ = 2.9) by fathers.

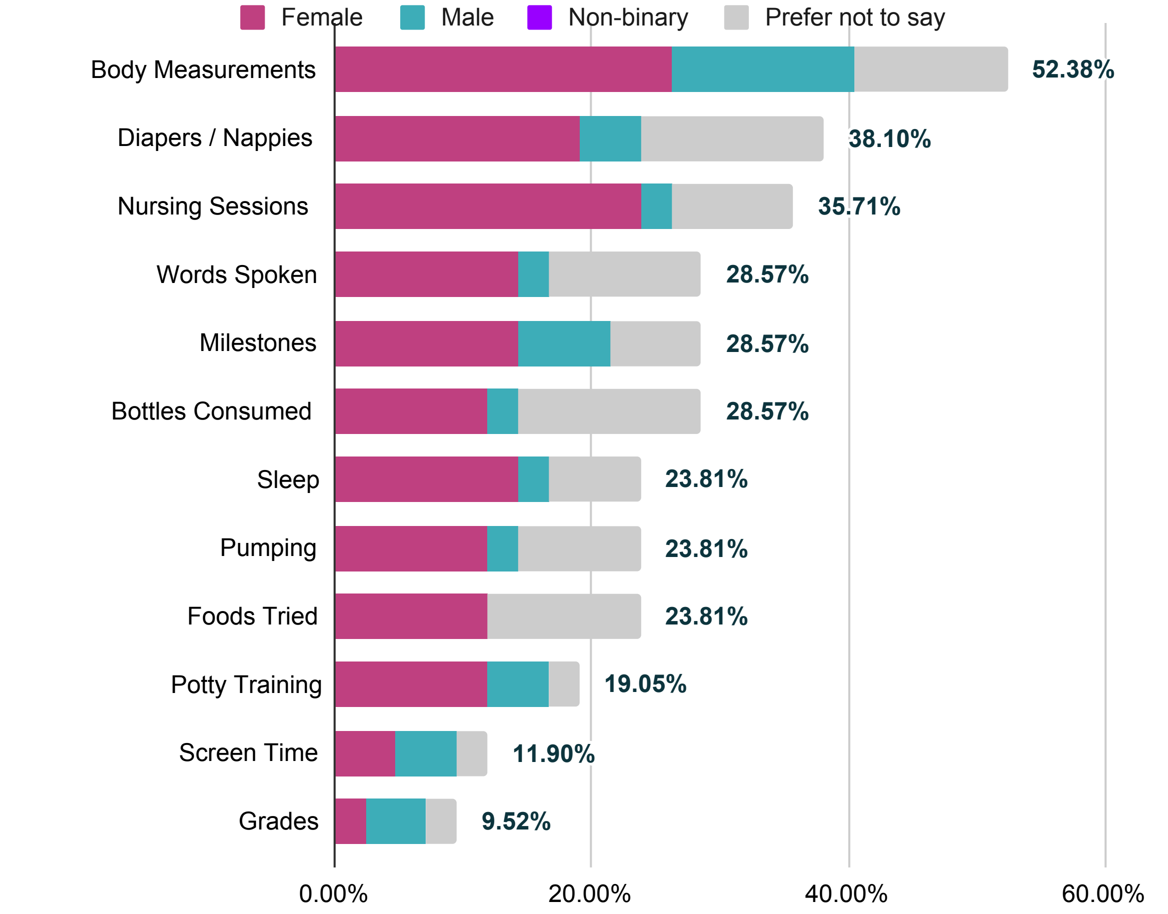


Figure A.2 *Child tracking use cases from the survey, ordered by percentage of respondents that manually tracked them.*

The most popular *manually* tracked items are shown in Figure A.1 and Figure A.2.

98.3 %

Custom Tracking Use Cases

Nearly a quarter (24.71%) of respondents tracked or have wanted to track one or more things not in the list of predefined common cases.

This was similar in the parental tracking set of questions, with 23.81% of parents entering item(s) in the freeform text fields.

It could be argued that this figure was high because the researchers missed certain common cases, but there was little overlap across subjects. Two researchers separately normalized differences in wording, then reconciled the result. Even after aggressive normalization, the only items that appeared more than once were "Books read" (3x), "Social Interactions" (3x), Location (2x), and "Personal expenses" (2x) for self-tracking, and "Teeth (when they come in and fall out)" (2x) for parental tracking.

Examples of unique custom use cases for adults were: *water consumption, cleaning, climbing progress, daytime sleepiness, scores on Lumosity and BrainHQ, time spent in Internet rabbit holes, being kinder, stretching sessions, progress in studying a foreign language, breast milk production, travel, groceries bought, yelling instances, prayers/mindfulness, teeth flossing, Aimovig injections*.

Examples of unique custom use cases for parental tracking were: *tummy time, baths, nail trimming, firsts, relationships, interests and wants, books read, prizes in reward system, accomplishments, funny things said*.

98.5 %

A.1.4 Motivations

Respondents' motivations are summarized in Figure A.3. By far the most common reason for self-tracking was *knowing thyself*: to find patterns in the data and get insights. While this was common for parental tracking as well, it was surpassed by data preservation for posterity. Comparing the tracking subject to others was also a far more popular reason for parental tracking than self-tracking. These differences are consistent with the literature, which finds that parents track primarily to preserve memories and detect developmental delays.[151, 153].

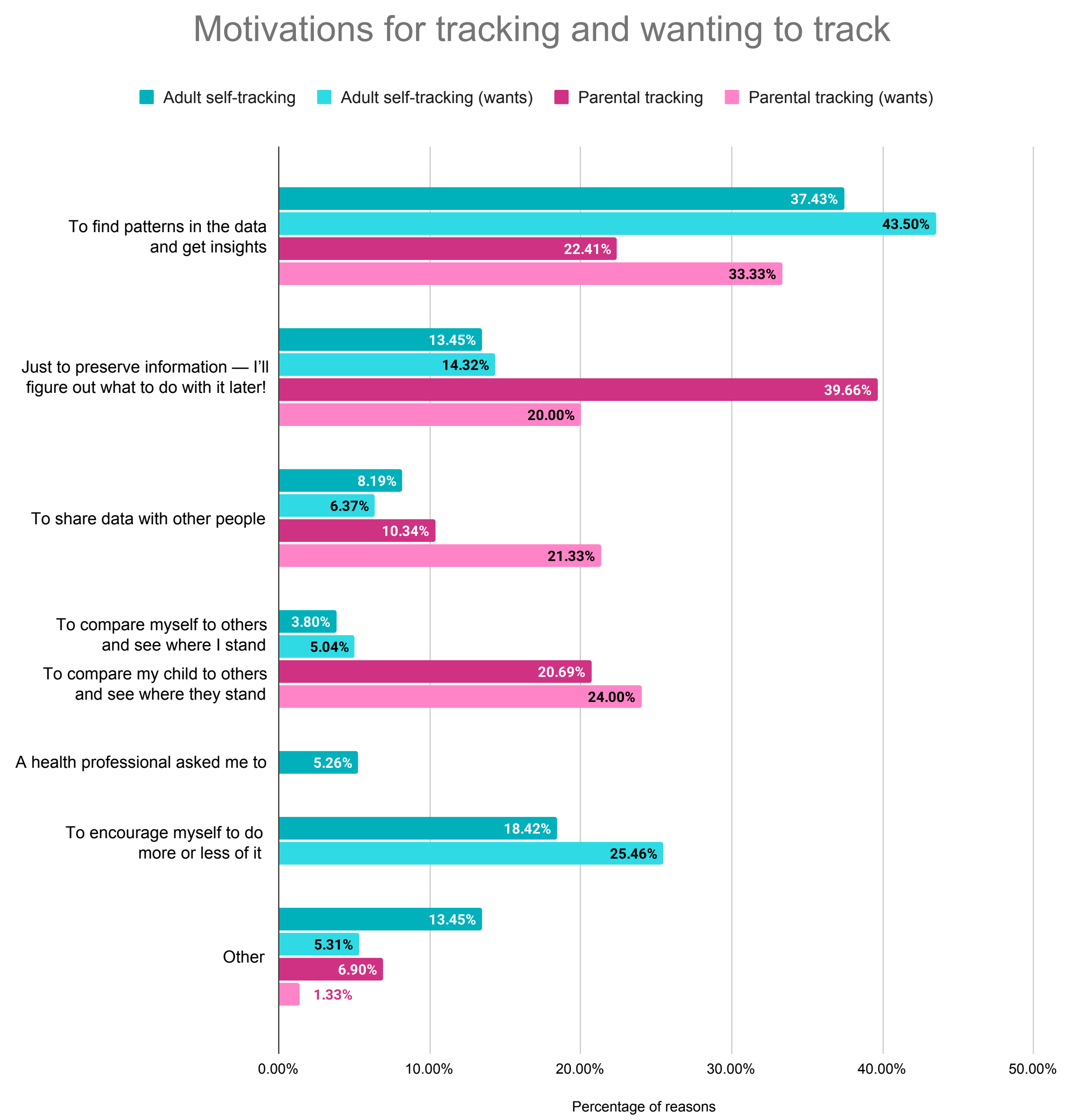


Figure A.3 *Respondent motivations for tracking or wanting to track, broken down by self-tracking and parental tracking use cases*

A.1.5 Reasons for Not Tracking

98.6 %

The most common reasons people gave for not tracking things they have wanted to track are shown in Figure 9.2. It is relevant that *Lack of suitable tools* was by far the most common reason both for self-tracking (36.1%) as well as parental tracking (38.8%). Lack of motivation seemed to be a less common reason in parental tracking, where the primary reasons for not tracking (in addition to lack of tools) were related to the overwhelm that parents feel, consistent with [151].

In 91% of cases, respondents said they would be more likely to record data if it were quick and easy (53.3% much more likely, 37.6% slightly more likely). Parents are even more eager to record data if the capture burden was reduced: Only 2.5% would still not record anything in that case (65% much more likely to record, 32.5% slightly more likely).

98.7 %

A.1.6 Tools Used

Types of Tools

The lack of suitable tools we discussed in the previous session became more apparent when we looked at the tools used.

For self-tracking, only half of tracked things are tracked with a widely available app or website. The rest are mostly tracked via spreadsheets (15.2%), documents (12.6%), or even handwriting (10%)

There's an even higher scarcity of suitable tools for parental tracking. Only 37.5% of things tracked are tracked with a widely available app or website. The rest are tracked via documents (40%), spreadsheets (17.5%), and handwriting (2.5%).

98.8 %

A.2 Needfinding Survey Questions

A.2.1 Tracking Use Cases

For the question *What do you track (or have tracked in the past) either for yourself or other adults (e.g. a partner, a friend, your parents etc)?* the 26 common cases presented to participants were (does not include the 5 freeform fields):

- General / Wellness
  - Exercise
  - Mood
  - Sleep
  - Stress / anxiety
  - Productivity
  - Menstruation (period)
  - Cigarettes consumed
  - Alcohol consumed
  - Hours outside
  - Screen time

- Relationships
  - Sexual activity
  - Arguments/fights
  - Joint expenses
- Health & Chronic conditions
  - Diet / eating habits
  - Body measurements
  - Blood sugar
  - Blood test results
  - Blood pressure
  - Migraines or headaches
  - Bowel movements
  - Other Daily symptoms
  - Medications taken
- Pregnancy & conception (if you have ever been pregnant)
  - Ovulation
  - Contractions
  - Pregnancy symptoms
  - Fetal movements

These were presented in a matrix, with columns:

- I have tracked this manually
- I have tracked this automatically
- I have wanted to track this (but never did)

Participants could check 0-3 of these per row. We decided against a N/A column to avoid clutter.

This does have the downside that this survey design cannot distinguish between things that do not apply at all (e.g. pregnancy-related things in a person without a uterus) and things for which the participant never had any desire to track.

There was also a short FAQ at the top (answers were collapsed but participants could expand them):

- **What qualifies as tracking?** Tracking is when you (manual) or a piece of software (automatic) records information about an aspect of you or someone else's life. E.g. checking the back statement every month is not tracking of joint expenses, but if you have a spreadsheet where you record things from these statements, then it is.
- **What if I track things about my children?** Do not include things you've tracked about your child(ren), as this is covered in a separate question.
- **What if someone else tracks these things about me?** If someone else tracks things for you (e.g. your partner or your doctor), please do not select that you track these things. Answer about the things you track yourself (either about yourself or other adults), and consider sending them this survey so they can participate too!
- **What is the difference between automatic and manual tracking?** Tracking manually is when data entry is performed by you, e.g. in a journal, document, spreadsheet, or app designed for this purpose. Tracking automatically is the kind of tracking where a device or app does the data entry for you with little to no intervention. Even if you have to approve or start each session manually, it's still automatic tracking for the purposes of this study if the actual data is produced automatically from device sensors and recorded automatically in an app without you having to enter it somewhere. Certain types of tracking may be a combination of automatic and manual, e.g. you use a sleep tracker that records how long and how deep you sleep, but then you go into the sleep log app and manually add notes about what you dreamed of. In that case, you'd tick both boxes.

  Example: If you use a smart blood pressure monitor to record your blood pressure and the monitor automatically records it in its own app, we'd consider it automatic tracking. If you manually go into a health tracker app and enter your blood pressure measurements, we'd consider it manual tracking. If the monitor records it mostly automatically, but sometimes the Bluetooth connection fails and you have to record it manually, you'd tick both boxes.
- **When do I tick the "I have wanted to track this (but don't)" box?** This means that the item applied to you at some point in your life and you have had a desire, interest, or need to track it, but for whatever reason you did not actually track it.

Participants were also asked "Are Are you a parent or caregiver for any children?" with options:

- No
- Yes, one child
- Yes, one or more children

Those who selected any of the Yes options were also presented with another question, "What have you tracked about the child(ren) in your care, if anything?" with another list of common cases pertaining to parenting (and 4 freeform fields):

- Body measurements (height, weight, head circumference etc)
- Sleep
- Foods tried
- Breastfeeding/chestfeeding sessions
- Pumping
- Bottles consumed
- Diapers / nappies
- Words spoken
- Potty training (pees, poops, accidents)
- Milestones
- Screen time
- Time outside
- Grades

There was also a short expandable FAQ about these:

- **What if I have tracked this for only one of my children?** Answer that you have tracked it, even if you have only tracked it for one of your children.
- **What if my child is too young for that metric but I do want to track it eventually?** If all children in your care are too young for that metric to make sense (e.g. a newborn baby has no "Foods tried", "Words spoken", or "Grades"), do not tick any of the "I have tracked this" columns. You can still select "I want to track this" if you plan to track it when your child is old enough.
- **What if someone else (e.g. another parent) tracks things about my child?** If someone else tracks things about your child (e.g. another parent or another caregiver) please do not select that you track these things. Please consider sending them this survey so they could participate too!

A.2.2 Follow-up Questions

For every item participants selected they had wanted to track (but didn't) they were presented with a series of follow-up questions:

- Why have you wanted to track this?
  - To find patterns in my data and get insights
  - To share data with other people
  - To compare myself to others and see where I stand
  - To encourage myself to do more or less of it
  - Just to preserve information — I'll figure out what to do with it later!
  - Other: (freeform)
- Why have you not tracked *[thing]*?
  - Lack of suitable tools
  - Not enough motivation to even look into how to track it
  - Lack of discipline to record the data
  - Too much data to record, task seems overwhelming
  - Other: (freeform)
- If recording this data was quick and easy how much more likely would you be to track *[thing]*? (1 choice)
  - Much more likely
  - Slightly more likely
  - I still would not record it
- Anything else you'd like to add about not tracking *[thing]*? (freeform)

For each thing participants said they tracked manually, they were presented with the following questions:

- Why do you track this?
  - To find patterns in my data and get insights
  - To share data with other people
  - To compare myself to others and see where I stand
  - To encourage myself to do more or less of it
  - A health professional asked me to
  - Just to preserve information — I'll figure out what to do with it later!
  - Other: (freeform)
- What do you use to track *[thing]*?
- What data do you record for each *[thing]* entry? (e.g. "date, time, intensity (1-5)")

- How do the tool(s) you use help you understand your data? (charts, insights, predictions, etc)
- How satisfied are you with the tool(s) you have used to track *[thing]*?
  - Very satisfied
  - Somewhat satisfied
  - Neither satisfied nor dissatisfied
  - Somewhat dissatisfied
  - Very dissatisfied
- What do you like about the tool(s) you use to track *[thing]*? (multiple choices, non-exclusive)
  - It records some of the data automatically with no effort from me
  - It makes it quick and easy to enter the data I want
  - It shows statistics and charts that give me insights about my data
  - It allows me to export my logged data
  - It's free
  - Other: (freeform)
- If you could, what would you change in the tool you use to track *[thing]*? What don't you like about it? What would make it better? (freeform)
- How long have you tracked (or did you track) *[thing]*?
  - A few days
  - A few weeks
  - A few months
  - 6 months to a year
  - 1 year or more
  - The entire time it was relevant (regardless of length)
- Do you still track *[thing]*?
  - Yes
  - No
- Why did you stop tracking it? (shown conditionally)
  - Too much effort to track correctly
  - No longer relevant
  - Other: (freeform)
- How complete is the data you keep on tracking *[thing]*?

    - Very complete, nearly every entry is recorded
    - A few entries are missed, but mostly complete
    - Very incomplete
- How could the tool(s) you use help you keep track of more complete data? (freeform)
- How did tracking this differ for each child and why? (shown conditionally)
    - I tracked this the same for all my children
    - I only tracked this for my first child, no time after!
    - Other: (freeform)

Before each block of such questions, participants were given a choice to skip to the end of the survey (the demographics questions). 12/85 did this.

The demographics questions were shown at the end, to eliminate stereotype threat:

- Your gender:
    - Female
    - Male
    - Non-binary
    - Other: (freeform)
    - Prefer not to say
- What is your age in years? (freeform, number)
- What is your occupation? (freeform)
- Anything else you'd like to add? (freeform)

Bibliography

[1] Berners-Lee, T., Cailliau, R., Groff, J. and Pollermann, B. 1992. World-Wide Web: The Information Universe. *Internet Research*. 2, (Jan. 1992), 52–58. doi 10.1108/eb047254.

Cited in 1, and 1.1

[2] Connell, R.S. 2013. Content management systems: trends in academic libraries. *Information Technology and Libraries (Online)*. 32, (2013), 42.

Cited in 1.1, and 3.1

[3] Berners-Lee, T. 1999. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. Harper San Francisco.

Cited in 1.1, and 5.1

[4] Koch, R. 2011. *The 80/20 principle: The secret of achieving more with less: Updated 20th anniversary edition of the productivity and business classic*. Hachette UK.

Cited in 1.1.1

[5] Chambers, C. and Scaffidi, C. 2010. Struggling to Excel: A Field Study of Challenges Faced by Spreadsheet Users. *2010 IEEE Symposium on Visual Languages and Human-Centric Computing* (Sep. 2010), 187–194. doi 10.1109/VLHCC.2010.33.

Cited in 1.1.1, and 9.8.5

[6] Chen, Y. and Chan, H. 2000. An Exploratory Study of Spreadsheet Debugging Processes. *PACIS 2000 Proceedings*. 12, (2000).

Cited in 1.1.1

[7] Huynh, D.F., Karger, D.R. and Miller, R.C. 2007. Exhibit: Lightweight Structured Data Publishing. *Proceedings of the 16th International Conference on World Wide Web - WWW '07*. (2007), 737. doi 10.1145/1242572.1242672.

Cited in 1.1.1, 2.5, 3.2, and 10.3

[8] Kushmerick, N., Weld, D.S. and Doorenbos, R. 1997. Wrapper Induction for Information Extraction. (1997).

Cited in 1.1.1

[9] Seehorn, D., Carey, S., Fuschetto, B., Lee, I., Moix, D., O'Grady-Cunniff, D., Owens, B.B., Stephenson, C. and Verno, A. 2011. CSTA K--12 computer science standards: revised 2011. (2011).

Cited in 1.2, and 3.1

[10] Maloney, J., Resnick, M., Rusk, N., Silverman, B. and Eastmond, E. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, (Nov. 2010), 16:1-16:15. doi 10.1145/1868358.1868363.

Cited in 1.2, and 1.4.4

[11] Verou, L., Zhang, A.X. and Karger, D.R. 2016. Mavo: Creating interactive data-driven web applications by authoring HTML. *UIST 2016 - Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (2016), 483–496. doi 10.1145/2984511.2984551.

Cited in 1.2.1, 1.2.3, 2.3, 4.1, 5.1, 6.1, 6.1, 6.1, 6.2, 7.1, 9.2.2, 9.4.2, and 9.8.5

[12] Verou, L., Alrashed, T. and Karger, D. 2018. Extending a reactive expression language with data update actions for end-user application authoring. *UIST 2018 - Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (2018), 379–387. doi 10.1145/3242587.3242663.

Cited in 1.2.1, 1.2.4, and 1.2.4

[13] Leiba, B. 2012. OAuth Web Authorization Protocol. *IEEE Internet Computing*. 16, (Jan. 2012), 74–77. doi 10.1109/MIC.2012.11.

Cited in 1.2.2, and 5.1

[14] Hardt, D. 2012. The OAuth 2.0 Authorization Framework. RFC 6749. Internet Engineering Task Force: *https://datatracker.ietf.org/doc/rfc6749*. Accessed: 2024-07-22. doi 10.17487/RFC6749.

Cited in 1.2.2, 5.1, and 5.3.4

[15] Kuebler-Wachendorff, S., Luzsa, R., Kranz, J., Mager, S., Syrmoudis, E., Mayr, S. and Grossklags, J. 2021. The Right to Data Portability: conception, status quo, and future directions. *Informatik Spektrum*. 44, (Aug. 2021), 264–272. doi 10.1007/s00287-021-01372-w.

Cited in 1.2.2, and 5.1

[16] Myers, B.A., Pane, J.F. and Ko, A.J. 2004. Natural programming languages and environments. *Commun. ACM*. 47, (Sep. 2004), 47–52. doi 10.1145/1015864.1015888.

Cited in 1.2.4, 2.7, and 4.1

[17] Tufekci, Z. 2019. Quantified Self. *Scientific American*. May (2019), 2019. doi 10.1038/scientificamerican0519-85.

Cited in 1.3

[18] Myers, B., Hudson, S.E. and Pausch, R. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction*. 7, (2000), 3–28. doi 10.1145/344949.344959.

Cited in 1.4.1, and 3.3

[19] Blackwell, A. and Burnett, M. 2002. Applying attention investment to end-user programming. *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments* (Sep. 2002), 28–30. doi 10.1109/HCC.2002.1046337.

Cited in 1.4.2, 1.4.2, and 1.4.2

[20] Bonar, J. and Soloway, E. 1983. Uncovering principles of novice programming. *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, Jan. 1983), 10–13. doi 10.1145/567067.567069.

Cited in 1.4.3

[21] Ma, L. 2007. *Investigating and Improving Novice Programmers' Mental Models of Programming Concepts* (Doctoral dissertation, University of Strathclyde).

Cited in 1.4.3, and 7.1.9

[22] Etemad, E. and Atkins, T. 2022. Selectors Level 4. W3C: *https://www.w3.org/TR/selectors/*. Accessed: 2024-07-31.

Cited in 1.4.3

[23] Berners-Lee, T. Principles of Design: *https://www.w3.org/DesignIssues/Principles.html*. Accessed: 2024-07-31.

Cited in 1.4.4

[24] Kesteren, A. van and Stachowiak, M. 2007. HTML Design Principles. W3C: *https://www.w3.org/TR/html-design-principles/*.

Cited in 1.4.4

[25] Verou, L. and Moon, S. Web Platform Design Principles. W3C: *https://www.w3.org/TR/design-principles/*. Accessed: 2024-07-31.

Cited in 1.4.4

[26] Norman, D.A. 1983. Design principles for human-computer interfaces. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, Dec. 1983), 1–10. doi 10.1145/800045.801571.

Cited in 1.4.4

[27] Benson, E. and Karger, D.R. 2014. End-users publishing structured information on the Web: An observational study of what, why, and how. *ACM CHI Conference on Human Factors in Computing Systems* (2014), 1265–1274. doi 10.1145/2556288/2557036.

Cited in 2.1.3, 3.1, 3.2.1, 3.2.1, 3.3, 4.1, 4.4.2, 6.2, and 10.1.3

[28] Jekyll • Simple, blog-aware, static sites: *https://jekyllrb.com/*. Accessed: 2024-07-22.

Cited in 2.1.4, and 3.2.1

[29] Leatherman, Z. Eleventy is a simpler static site generator: *https://www.11ty.dev/*. Accessed: 2024-07-22.

Cited in 2.1.4

[30] Google Inc. Google Sheets: Online Spreadsheets & Templates: *https://sheets.google.com/*. Accessed: 2024-07-22.

Cited in 2.2, and 2.2

[31] Coda Project, Inc. Coda: Your all-in-one collaborative workspace: *https://coda.io/*. Accessed: 2023-09-15.

Cited in 2.2, 2.2, 2.2, and 9.2.2

[32] Bakke, E. and Karger, D.R. 2016. Expressive Query Construction through Direct Manipulation of Nested Relational Results. *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, Jun. 2016), 1377–1392. doi 10.1145/2882903.2915210.

Cited in 2.2, 2.2, 2.2, 2.3, 4.2.2, 4.7, 4.8.4, and 9.4.7

[33] Chang, K.S.-P. and Myers, B.A. 2014. Creating Interactive Web Data Applications with Spreadsheets. *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2014), 87–96. doi 10.1145/2642918.2647371.

Cited in 2.2, 2.2, 2.4, 3.2, 3.2, and 3.2.1

[34] Chang, K.S.-P. and Myers, B.A. 2016. Using and Exploring Hierarchical Data in Spreadsheets. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, May 2016), 2497–2507. doi 10.1145/2858036.2858430.

Cited in 2.2, 2.4, 3.2, and 4.2.1

[35] Bakke, E., Karger, D. and Miller, R. 2011. A spreadsheet-based user interface for managing plural relationships in structured data. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, May 2011), 2541–2550. doi 10.1145/1978942.1979313.

Cited in 2.2, and 2.3

[36] Burnett, M., Atwood, J., Djang, R.W., Reichwein, J., Gottfried, H. and Yang, S. 2001. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*. 11, (2001), 155–206.

Cited in 2.2, 2.2, and 4.1

[37] Lewis, C. 1990. NoPumpG: creating interactive graphics with spreadsheet machinery. *Visual Programming Environments: Paradigms and Systems*. (1990), 526–546.

Cited in 2.2

[38] Wilde, N. and Lewis, C. 1990. Spreadsheet-based interactive graphics: from prototype to tool. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, Mar. 1990), 153–160. doi 10.1145/97243.97268.

Cited in 2.2

[39] Gonzalez, H., Halevy, A.Y., Jensen, C.S., Langen, A., Madhavan, J., Shapley, R., Shen, W. and Goldberg-Kidon, J. 2010. Google fusion tables: web-centered data management and collaboration. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), 1061–1066.

Cited in 2.2, and 2.3

[40] Benson, E., Zhang, A.X. and Karger, D.R. 2014. Spreadsheet driven web applications. *Proceedings of the 27th annual ACM symposium on user interface software and technology* (Honolulu, Hawaii, USA, 2014), 97–106. doi 10.1145/2642918.2647387.

Cited in 2.2, 2.4, 2.5, 3.2, 3.2, 3.2.1, and 6.1

[41] Fu, Y., Ong, K.W., Papakonstantinou, Y. and Petropoulos, M. 2011. The SQL-based all-declarative FORWARD web application development framework. *CIDR* (2011), 69–78.

Cited in 2.3, and 2.5

[42] Bakke, E. 2022. *Expressive Query Construction through Direct Manipulation of Nested Relational Results* (Doctoral dissertation, Massachusetts Institute of Technology).

Cited in 2.3

[43] Qian, L., LeFevre, K. and Jagadish, H.V. 2010. CRIUS: user-friendly database design. *Proc. VLDB Endow.* 4, (Nov. 2010), 81–92. doi 10.14778/1921071.1921075.

Cited in 2.3

[44] Ceri, S., Fraternali, P. and Bongio, A. 2000. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks*. 33, (2000), 137–157.

Cited in 2.3, and 6.2

[45] Zloof, M.M. 1975. Query-by-example: the invocation and definition of tables and forms. *Proceedings of the 1st International Conference on Very Large Data Bases* (1975), 1–24.

Cited in 2.3

[46] Benzi, F., Maio, D. and Rizzi, S. 1999. VISIONARY: a viewpoint-based visual language for querying relational databases. *Journal of Visual Languages & Computing*. 10, (1999), 117–145.

Cited in 2.3

[47] Murray, N., Paton, N. and Goble, C. 1998. Kaleidoquery: a visual query language for object databases. *Proceedings of the working conference on Advanced visual interfaces* (1998), 247–257.

Cited in 2.3

[48] Choobineh, J., Mannino, M.V. and Tseng, V.P. 1992. A form-based approach for database analysis and design. *Communications of the ACM*. 35, (1992), 108–120.

Cited in 2.3

[49] Embley, D.W. 1989. NFQL: the natural forms query language. *ACM Transactions on Database Systems (TODS)*. 14, (1989), 168–211.

Cited in 2.3

[50] Mitchell, K.J. and Kennedy, J.B. 1996. DRIVE: an environment for the organised construction of user-interfaces to databases. *Proceedings of the 1996 international conference on Interfaces to Databases* (1996), 5–5.

Cited in 2.3

[51] Jagadish, H.V., Chapman, A., Elkiss, A., Jayapandian, M., Li, Y., Nandi, A. and Yu, C. 2007. Making database systems usable. *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (New York, NY, USA, Jun. 2007), 13–24. doi 10.1145/1247480.1247483.

Cited in 2.3, 2.4, 4.8.6, and 6.1

[52] Yang, F., Gupta, N., Botev, C., Churchill, E.F., Levchenko, G. and Shanmugasundaram, J. 2008. WYSIWYG development of data driven web applications. *Proceedings of the VLDB Endowment*. 1, (2008), 163–175. doi 10.14778/1453856.1453879.

Cited in 2.3, 2.4, and 6.2

[53] Bhatia, G., Fu, Y., Kowalczykowski, K., Ong, K.W., Zhao, K.K., Deutsch, A. and Papakonstantinou, Y. 2009. FORWARD : Design Specification Techniques for Do-It-Yourself Application Platforms. *Proceedings of the 12th International Workshop on the Web and Databases (WebDB'09)*. 50, (2009), 2008–2009.
Cited in 2.3

[54] Kowalzcykowski, K., Deutsch, A., Ong, K.W., Papakonstantinou, Y., Zhao, K.K. and Petropoulos, M. 2009. Do-It-Yourself database-driven web applications. *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research (CIDR'09)* (2009).
Cited in 2.3, 2.4, and 6.1

[55] Karger, D.R., Ostler, S. and Lee, R. 2009. The web page as a WYSIWYG end-user customizable database-backed information management application. *Proceedings of the 22nd annual ACM symposium on User interface software and technology - UIST '09* (New York, New York, USA, 2009), 257. doi 10.1145/1622176.1622223.
Cited in 2.4, 2.5, 3.2, 3.2.1, and 6.2

[56] Google Inc. AngularJS — Superheroic JavaScript MVW Framework: *https://angularjs.org/*. Accessed: 2024-07-22.
Cited in 2.5, and 4.2.3

[57] You, E. Vue.Js - The Progressive JavaScript Framework: *https://vuejs.org/*. Accessed: 2024-07-22.
Cited in 2.5, and 4.2.3

[58] Oney, S., Myers, B. and Brandt, J. 2012. ConstraintJS : Programming Interactive Behaviors for the Web by Integrating Constraints and States. *Proceedings of the 25th annual ACM symposium on User interface software and technology* (New York, NY, USA, Oct. 2012), 229–238. doi 10.1145/2380116.2380146.
Cited in 2.5

[59] Kay, M. 2021. XSL Transformations (XSLT) Version 2.0 (Second Edition): *https://www.w3.org/TR/xslt20/*. Accessed: 2024-08-30.
Cited in 2.5.2

[60] Benson, E.E. and Karger, D.D.R. 2013. Cascading tree sheets and recombinant HTML: better encapsulation and retargeting of web content. … *International Conference on World Wide Web*. (2013), 107–117.

Cited in 2.5.2

[61] Cafarella, M.J., Halevy, A. and Madhavan, J. 2011. Structured data on the web. *Communications of the ACM*. 54, (2011), 72–79.

Cited in 2.6, and 3.2.2

[62] HTML+RDFa 1.1 — Second Edition. W3C: *https://www.w3.org/TR/html-rdfa/*. Accessed: 2023-09-15.

Cited in 2.6, 3.1.1, 3.2.2, 3.3.3, 3.3.3, and 3.3.3

[63] Microdata — HTML Living Standard. WHATWG: *https://html.spec.whatwg.org/multipage/microdata.html*. Accessed: 2023-09-15.

Cited in 2.6, 3.2.2, 3.3.3, and 3.3.3

[64] Berners-Lee, T., Hendler, J., Lassila, O., and others 2001. The semantic web. *Scientific American*. 284, (2001), 28–37.

Cited in 2.6, and 3.2.2

[65] Khalili, A. and Auer, S. 2013. WYSIWYM Authoring of Structured Content Based on Schema.Org. *Web Information Systems Engineering – WISE 2013* (Berlin, Heidelberg, 2013), 425–438. doi 10.1007/978-3-642-41154-0_32.

Cited in 2.6

[66] Du Boulay, B. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research*. 2, (1986), 57–73.

Cited in 2.7

[67] Ma, L., Ferguson, J., Roper, M. and Wood, M. 2011. Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education*. 21, (2011), 57–80.

Cited in 2.7, 4.4.1, 7.2.1, and 7.2.6

[68] Miller, L.A. 1974. Programming by non-programmers. *International Journal of Man-Machine Studies*. 6, (1974), 237–260. doi 10.1016/S0020-7373(74)80004-0.

Cited in 2.7, and 4.4.1

[69] Pane, J.F., Myers, B.A., and others 2001. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*. 54, (2001), 237–264. doi 10.1006/ijhc.2000.0410.

Cited in 2.7, 4.4.1, 7.2.1, 7.2.4, and 7.2.4

[70] Myers, B.A., Ko, A.J., Scaffidi, C., Oney, S., Yoon, Y., Chang, K., Kery, M.B. and Li, T.J.-J. 2017. Making End User Development More Natural. *New Perspectives in End-User Development*. F. Paternò and V. Wulf, eds. Springer International Publishing. 1–22. doi 10.1007/978-3-319-60291-2_1.

Cited in 2.7, 3.1, and 4.1

[71] Myers, B.A., Ko, A.J., Park, S.Y., Stylos, J., LaToza, T.D. and Beaton, J. 2008. More natural end-user software engineering. *Proceedings of the 4th international workshop on End-user software engineering* (New York, NY, USA, May 2008), 30–34. doi 10.1145/1370847.1370854.

Cited in 2.7, and 3.1

[72] Rosson, M.B., Ballin, J. and Rode, J. 2005. Who, what, and how: A survey of informal and professional web developers. *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on* (2005), 199–206.

Cited in 3.1

[73] Shneiderman, B. 1993. Direct manipulation: a step beyond programming languages. *Sparks of Innovation in Human-Computer Interaction*. 17, (1993), 1993.

Cited in 3.1.1

[74] Birbeck, M. and McCarron, S. CURIE Syntax 1.0. W3C: *https://www.w3.org/TR/curie/*. Accessed: 2024-08-03.

Cited in 3.3.3

[75] Panko, R.R. 2013. The cognitive science of spreadsheet errors: Why thinking is bad. *Proceedings of the Annual Hawaii International Conference on System Sciences* (2013). doi 10.1109/HICSS.2013.513.

Cited in 3.3.5, and 4.5.4

[76] Green, T.R.G. and Petre, M. 1996. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing*. 7, (Jun. 1996), 131–174. doi 10.1006/jvlc.1996.0009.

Cited in 3.5.1, 4.2.4, 5.1, and 10.4

[77] Sporny, M., Longley, D., Kellogg, G., Lanthaler, M. and Lindström, N. 2020. JSON-LD 1.1. W3C: *https://www.w3.org/TR/json-ld/*.

Cited in 3.5.5

[78] Galassi, G. and Mattessich, R.V. 2014. Some clarification to the evolution of the electronic spreadsheet. *Journal of Emerging Technologies in Accounting*. 11, (2014), 99–104.

Cited in 4.1, and 4.2

[79] Pratt, V.R. 1973. Top down operator precedence. *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73* (Boston, Massachusetts, 1973), 41–51. doi 10.1145/512927.512931.

Cited in 4.1, and 4.6.1

[80] Chang, K.S.-P. and Myers, B.A. 2017. Gneiss: spreadsheet programming using structured web service data. *Journal of Visual Languages & Computing*. 39, (Apr. 2017), 41–50. doi 10.1016/j.jvlc.2016.07.004.

Cited in 4.2.1, and 6.1

[81] Svelte • Cybernetically enhanced web apps: *https://svelte.dev/*. Accessed: 2024-08-05.

Cited in 4.2.3

[82] Kodosky, J. 2020. LabVIEW. *Proc. ACM Program. Lang.* 4, HOPL (Jun. 2020), 78:1-78:54. doi 10.1145/3386328.

Cited in 4.2.4

[83] Wang, G., Yang, S. and Han, Y. 2009. Mashroom: end-user mashup programming using nested tables. *Proceedings of the 18th international conference on World wide web* (New York, NY, USA, Apr. 2009), 861–870. doi 10.1145/1526709.1526825.

Cited in 4.2.4

[84] Mase, M.B. and Nel, L. 2022. Common Code Writing Errors Made by Novice Programmers: Implications for the Teaching of Introductory Programming. *ICT Education* (Cham, 2022), 102–117. doi 10.1007/978-3-030-95003-3_7.

Cited in 4.5.4

[85] 154, I. 2019. ISO standard. ISO 8601-1:2019 - Date and time — Representations for information interchange — Part 1: Basic rules: *https://www.iso.org/standard/70907.html*. Accessed: 2022-09-13.

Cited in 4.5.7, 9.5, and 9.5

[86] Codd, E.F. 1972. Further Normalization of the Data Base Relational Model. *Data Base Systems*. 6, (1972), 33–64.

Cited in 4.7

[87] Broman, K.W. and Woo, K.H. 2018. Data Organization in Spreadsheets. *The American Statistician*. 72, (Jan. 2018), 2–10. doi 10.1080/00031305.2017.1375989.

Cited in 4.7

[88] Scaffidi, C. and Shaw, M. 2010. Chapter 21 - Reuse in the world of end user programmers. *No Code Required*. A. Cypher, M. Dontcheva, T. Lau, and J. Nichols, eds. Morgan Kaufmann. 407–421. doi 10.1016/B978-0-12-381541-5.00021-3.

Cited in 4.8.6, 9.8.3, and 10.4

[89] Norman, D. 1986. Cognitive Engineering. *User Centered System Design: New Perspectives on Human-Computer Interaction*. 31–61. doi 10.1201/b15703-3.

Cited in 5.1

[90] Carter, S., Whitehead, E.J., Goland, Y.Y., Faizi, A. and Jensen, D. 1999. HTTP Extensions for Distributed Authoring – WEBDAV. RFC 2518. Internet Engineering Task Force: *https://datatracker.ietf.org/doc/rfc2518*. Accessed: 2024-08-12. doi 10.17487/RFC2518.

Cited in 5.2

[91] Hernández, L.O. and Pegah, M. 2003. WebDAV: what it is, what it does, why you need it. *Proceedings of the 31st annual ACM SIGUCCS fall conference* (New York, NY, USA, Sep. 2003), 249–254. doi 10.1145/947469.947535.

Cited in 5.2

[92] Dusseault, L.B. 2003. *WebDAV: Next Generation Collaborative Web Authoring*. Prentice Hall Professional Technical Reference.

Cited in 5.2

[93] Dusseault, L. 2007. HTTP extensions for web distributed authoring and versioning (WebDAV). RFC 4918. IETF: *http://tools.ietf.org/rfc/rfc4918.txt*.

Cited in 5.2

[94] Mansour, E., Sambra, A.V., Hawke, S., Zereba, M., Capadisli, S., Ghanem, A., Aboulnaga, A. and Berners-Lee, T. 2016. A Demonstration of the Solid Platform for Social Web Applications. *Proceedings of the 25th International Conference Companion on World Wide Web* (Republic, Canton of Geneva, CHE, Apr. 2016), 223–226. doi 10.1145/2872518.2890529.

Cited in 5.2

[95] Sambra, A.V., Mansour, E., Hawke, S., Zereba, M., Greco, N., Ghanem, A., Zagidulin, D., Aboulnaga, A. and Berners-Lee, T. Solid: A Platform for Decentralized Social Applications Based on Linked Data.

Cited in 5.2

[96] Alrashed, T., Almahmoud, J., Zhang, A.X. and Karger, D.R. 2020. ScrAPIr: Making Web Data APIs Accessible to End Users. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu HI USA, Apr. 2020), 1–12. doi 10.1145/3313831.3376691.

Cited in 5.2, and 5.2

[97] Alrashed, T., Verou, L. and Karger, D.R. 2021. Shapir: Standardizing and Democratizing Access to Web APIs. *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event USA, Oct. 2021), 1282–1304. doi 10.1145/3472749.3474822.

Cited in 5.2, 5.2, and 7.4

[98] Schema.Org - Schema.Org: *https://schema.org/*. Accessed: 2024-07-29.

Cited in 5.2, and 7.4

[99] URL Pattern Standard. WHATWG: *https://urlpattern.spec.whatwg.org/*. Accessed: 2024-08-06.

Cited in 5.3.3

[100] Fetch Standard. WHATWG: *https://fetch.spec.whatwg.org/*. Accessed: 2024-08-06.

Cited in 5.3.4, and 10.1.5

[101] Verou, L. 2013. *Dabblet: A visual IDE for rapid prototyping of client-side web development* (Bachelor's thesis, Athens University of Economics & Business).
Cited in 5.3.4

[102] Freed, N. and Borenstein, N.S. 1996. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046. Internet Engineering Task Force: *https://datatracker.ietf.org/doc/rfc2046*. Accessed: 2024-08-06. doi 10.17487/RFC2046.
Cited in 5.5.2

[103] Preston-Werner, T. TOML: Tom's Obvious Minimal Language: *https://toml.io/*. Accessed: 2024-08-06.
Cited in 5.5.2

[104] Nottingham, M. 2019. Well-Known Uniform Resource Identifiers (URIs). RFC 8615. Internet Engineering Task Force: *https://datatracker.ietf.org/doc/rfc8615*. Accessed: 2024-08-06. doi 10.17487/RFC8615.
Cited in 5.6.5

[105] Quan, D., Huynh, D. and Karger, D.R. 2003. Haystack: A Platform for Authoring End User Semantic Web Applications. *The Semantic Web - ISWC 2003* (Berlin, Heidelberg, 2003), 738–753. doi 10.1007/978-3-540-39718-2_47.
Cited in 6.2

[106] Pane, J.F., Myers, B.A. and Miller, L.B. 2002. Using HCI techniques to design a more usable programming system. *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments* (Sep. 2002), 198–206. doi 10.1109/HCC.2002.1046372.
Cited in 6.3.2

[107] Salvaneschi, G., Proksch, S., Amann, S., Nadi, S. and Mezini, M. 2017. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *IEEE Transactions on Software Engineering*. 43, (2017), 1–1. doi 10.1109/TSE.2017.2655524.
Cited in 6.3.2

[108] Salvaneschi, G., Amann, S., Proksch, S. and Mezini, M. 2014. An empirical study on program comprehension with reactive programming. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. (2014), 564–575. doi 10.1145/2635868.2635895.

Cited in 6.3.2

[109] Greif, S. and Verou, L. 2023. State of HTML 2023. Devographics. (2023). *https://2023.stateofhtml.com/*. Accessed: 2024-07-30.

Cited in 6.4.2, and 10.6.2

[110] Satyanarayan, A., Russell, R., Hoffswell, J. and Heer, J. 2016. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*. 22, (2016), 659–668.

Cited in 6.5.4, and 6.5.4

[111] HTML Standard. WHATWG: *https://html.spec.whatwg.org/multipage/*. Accessed: 2024-07-28.

Cited in 7.1.1

[112] Jackson, D. 2015. Towards a theory of conceptual design for software. *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (New York, NY, USA, Oct. 2015), 282–296. doi 10.1145/2814228.2814248.

Cited in 7.1.1

[113] Kelleher, C. and Pausch, R. 2005. Lowering the Barriers to Programming : a survey of programming environments and languages for novice programmers. *Science*. 37, (2005), 83–137. doi 10.1145/1089733.1089734.

Cited in 7.1.9

[114] Myers, B.A., Ko, A.J., Latoza, T.D. and Yoon, Y. 2016. Programmers Are Users Too: Human-centered methods for improving programming tools. *Computer*. 49, (2016), 44–52. doi 10.1109/MC.2016.200.

Cited in 7.2.1, and 7.2.3

[115] Brooke, J. 1996. SUS: A “Quick and Dirty” Usability Scale. *Usability Evaluation in Industry*. 189, (1996), 4–7. doi 10.1201/9781498710411-35.

Cited in 7.2.3, 7.2.7, 9.1.1, 9.7.1, and 9.8.7

[116] Sauro, J. 2011. *A practical guide to the System Usability Scale: Background, benchmarks & best practices*. Measuring Usability LLC.

Cited in 7.2.7

[117] Lewis, J.R. and Sauro, J. 2009. The factor structure of the system usability scale. *International conference on human centered design* (2009), 94–103.

Cited in 7.2.7

[118] Alrashed, T., Verou, L. and Karger, D. 2022. Wikxhibit: Using HTML and Wikidata to Author Applications that Link Data Across the Web. *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, Oct. 2022), 1–15. doi 10.1145/3526113.3545706.

Cited in 7.5

[119] Same Origin Policy - Web Security: *https://www.w3.org/Security/wiki/Same_Origin_Policy*. Accessed: 2024-08-28.

Cited in 8.1.3, and 10.1.5

[120] Barth, A. 2011. The Web Origin Concept. RFC 6454. Internet Engineering Task Force: *https://datatracker.ietf.org/doc/rfc6454*. Accessed: 2024-08-28. doi 10.17487/RFC6454.

Cited in 8.1.3, and 10.1.5

[121] Solomon, C., Harvey, B., Kahn, K., Lieberman, H., Miller, M.L., Minsky, M., Papert, A. and Silverman, B. 2020. History of Logo. *Proc. ACM Program. Lang.* 4, HOPL (Jun. 2020), 79:1-79:66. doi 10.1145/3386329.

Cited in 8.2.2, and 8.2.2

[122] Cicileo, F.R. 2018. *Mavo Create: a WYSIWYG editor for Mavo* (Master's thesis, Massachusetts Institute of Technology).

Cited in 9.1

[123] Li, I., Dey, A. and Forlizzi, J. 2010. A Stage-Based Model of Personal Informatics Systems. *ACM CHI Conference on Human Factors in Computing Systems* (2010).

Cited in 9.1.1, 9.1.1, 9.2.1, 9.2.1, 9.2.1, 9.4.8, and 9.8.7

[124] Choe, E.K., Lee, N.B., Lee, B., Pratt, W. and Kientz, J.A. 2014. Understanding quantified-selfers' practices in collecting and exploring personal data. *ACM CHI Conference on Human Factors in Computing Systems* (2014), 1143–1152. doi 10.1145/2556288.2557372.

Cited in 9.1.1, 9.1.1, and 9.2.2

[125] Lee, J.H., Schroeder, J. and Epstein, D.A. 2021. Understanding and supporting self-tracking app selection. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*. 5, (2021). doi 10.1145/3494980.

Cited in 9.1.1, 9.1.1, 9.2.1, 9.2.1, 9.2.1, 9.3, and 9.8.7

[126] Oh, J. and Lee, U. 2015. Exploring UX issues in quantified self technologies. *2015 8th International Conference on Mobile Computing and Ubiquitous Networking, ICMU 2015*. (2015), 53–59. doi 10.1109/ICMU.2015.7061028.

Cited in 9.1.1, 9.2.1, 9.2.1, 9.2.1, 9.2.1, 9.2.1, 9.2.1, and 9.8.7

[127] Potapov, K. and Marshall, P. 2020. LifeMosaic: Co-design of a personal informatics tool for youth. *Proceedings of the Interaction Design and Children Conference, IDC 2020*. (2020), 519–531. doi 10.1145/3392063.3394429.

Cited in 9.1.1, 9.2.1, 9.2.1, 9.2.1, 9.2.1, and 9.8.7

[128] Kim, N.W., Wang, A., Im, H., Gajos, K., Riche, N.H. and Pfister, H. 2019. Dataselfie: Empowering people to design personalized visuals to represent their data. *Conference on Human Factors in Computing Systems - Proceedings*. (2019), 1–12. doi 10.1145/3290605.3300309.

Cited in 9.1.1, 9.2.1, 9.2.1, 9.2.1, and 9.8.7

[129] Ayobi, A., Sonne, T., Marshall, P. and Cox, A.L. 2018. Flexible and mindful self-tracking: Design implications from paper bullet journals. *Conference on Human Factors in Computing Systems - Proceedings*. 2018-April, (2018), 1–14. doi 10.1145/3173574.3173602.

Cited in 9.1.1, 9.2.1, 9.2.1, and 9.3

[130] Abtahi, P., Ding, V., Yang, A.C., Bruzzese, T., Romanos, A.B., Murnane, E.L., Follmer, S. and Landay, J.A. 2020. Understanding Physical Practices and the Role of Technology in Manual Self-Tracking. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*. 4, (2020). doi 10.1145/3432236.

Cited in 9.1.1, 9.2.1, 9.2.1, 9.2.1, 9.2.1, 9.2.1, and 9.3

[131] Lazar, A., Koehler, C., Tanenbaum, T.J. and Nguyen, D.H. 2015. Why we use and abandon smart devices. *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (New York, NY, USA, Sep. 2015), 635–646. doi 10.1145/2750858.2804288.

Cited in 9.1.1

[132] Clawson, J., Pater, J.A., Miller, A.D., Mynatt, E.D. and Mamykina, L. 2015. No longer wearing: investigating the abandonment of personal health-tracking technologies on craigslist. *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing* (New York, NY, USA, Sep. 2015), 647–658. doi 10.1145/2750858.2807554.

Cited in 9.1.1

[133] Epstein, D.A., Caraway, M., Johnston, C., Ping, A., Fogarty, J. and Munson, S.A. 2016. Beyond Abandonment to Next Steps: Understanding and Designing for Life after Personal Informatics Tool Use. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, May 2016), 1109–1113. doi 10.1145/2858036.2858045.

Cited in 9.1.1

[134] Choe, E.K., Abdullah, S., Rabbi, M., Thomaz, E., Epstein, D.A., Cordeiro, F., Kay, M., Abowd, G.D., Choudhury, T., Fogarty, J., Lee, B., Matthews, M. and Kientz, J.A. 2017. Semi-Automated Tracking: A Balanced Approach for Self-Monitoring Applications Characterizing Semi-Automated Tracking. (2017). *http://quantifiedself.com*. doi 10.1109/MPRV.2017.18.

Cited in 9.1.1, 9.9.7, and A.1.2

[135] Li, I., Dey, A.K. and Forlizzi, J. 2012. Using context to reveal factors that affect physical activity. *ACM Transactions on Computer-Human Interaction*. 19, (2012). doi 10.1145/2147783.2147790.

Cited in 9.1.1

[136] Korotitsch, W.J. and Nelson-Gray, R.O. 1999. An Overview of Self-Monitoring in Assessment and Treatment. *Psychological Assessment*. 11, (1999), 415–425.

Cited in 9.1.1

[137] Choe, E.K., Lee, B., Kay, M., Pratt, W. and Kientz, J.A. 2015. SleepTight: Low-burden, self-monitoring technology for capturing and reflecting on sleep behaviors. *UbiComp 2015 - Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. (2015), 121–132. doi 10.1145/2750858.2804266.

Cited in 9.1.1

[138] Eveleth, R. 2014. How self-tracking apps exclude women. *The Atlantic*. 15, (2014).

Cited in 9.1.1, 9.2.1, and 9.2.1

[139] Ayobi, A., Marshall, P. and Cox, A.L. 2020. Trackly: A Customisable and Pictorial Self-Tracking App to Support Agency in Multiple Sclerosis Self-Care. *Conference on Human Factors in Computing Systems - Proceedings*. (2020), 1–15. doi 10.1145/3313831.3376809.

Cited in 9.2.1, 9.2.1, and 9.2.1

[140] Luo, Y., Liu, P. and Choe, E.K. 2019. Co-designing food trackers with dietitians: Identifying design opportunities for food tracker customization. *ACM CHI Conference on Human Factors in Computing Systems* (2019). doi 10.1145/3290605.3300822.

Cited in 9.2.1, 9.2.1, 9.2.1, 9.2.1, and 9.3

[141] Kim, Y.-H., Ho Jeon, J., Lee, B., Choe, E.K., Jeon, J.H. and Seo, J. 2017. OmniTrack: A Flexible Self-Tracking Approach Leveraging Semi-Automated Tracking. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol*. 1, Article67 (2017), 67. doi 10.1145/3130930.

Cited in 9.2.1, 9.2.1, 9.4.4, and 9.9.4

[142] Epstein, D.A. 2023. This Watchface Fits with my Tattoos : Investigating Customisation Needs and Preferences in Personal Tracking. (2023). doi 10.1145/3544548.3580955.

Cited in 9.2.1, and 9.3

[143] Epstein, D.A., Lee, N.B., Kang, J.H., Agapie, E., Schroeder, J., Pina, L.R., Fogarty, J., Kientz, J.A. and Munson, S.A. 2017. Examining menstrual tracking to inform the design of personal informatics tools. *ACM CHI Conference on Human Factors in Computing Systems* (2017), 6876–6888. doi 10.1145/3025453.3025635.

Cited in 9.2.1

[144] Bearable Ltd Bearable Symptom Tracker App | Pain & Mental Health Journal: *https://bearable.app/*. Accessed: 2023-09-14.

Cited in 9.2.1, and 9.8.7

[145] Hello Code Pty Ltd Exist · Understand your behaviour: *https://exist.io/*. Accessed: 2023-09-14.

Cited in 9.2.1

[146] Gyecsek, D. DoEntry | Guided Journaling: *https://doentry.com/*. Accessed: 2023-09-14.

Cited in 9.2.1

[147] Happy Data, LLC. Private Daily journal for short attention spans. Nomie: *https://nomie.app/*. Accessed: 2023-09-14.

Cited in 9.2.1

[148] chrono.me KeepTrack: *http://www.zagalaga.com/*. Accessed: 2023-09-14.

Cited in 9.2.1

[149] Wang, J., O'Kane, A.A., Newhouse, N., Sethu-Jones, G.R. and De Barbaro, K. 2017. Quantified baby: Parenting and the use of a baby wearable in the wild. *Proceedings of the ACM on Human-Computer Interaction*. 1, CSCW (2017), 1–19. doi 10.1145/3134743.

Cited in 9.2.1, 9.3.1, and A.1.3

[150] Gaunt, K., Nacsa, J. and Penz, M. 2014. Baby Lucent: Pitfalls of applying quantified self to baby products. *ACM CHI Conference on Human Factors in Computing Systems* (2014), 263–268. doi 10.1145/2559206.2580937.

Cited in 9.2.1, 9.3.1, and A.1.3

[151] Kientz, J.A., Arriaga, R.I., Chetty, M., Hayes, G.R., Richardson, J., Patel, S.N. and Abowd, G.D. 2007. Grow and know: Understanding record-keeping needs for tracking the development of young children. *ACM CHI Conference on Human Factors in Computing Systems* (2007), 1351–1360. doi 10.1145/1240624.1240830.

Cited in 9.2.1, A.1.4, and A.1.5

[152] Kientz, J.A., Arriaga, R.I. and Abowd, G.D. 2009. Baby steps: Evaluation of a system to support record- keeping for parents of young children. *ACM CHI Conference on Human Factors in Computing Systems* (2009), 1713–1722. doi 10.1145/1518701.1518965.

Cited in 9.2.1

[153] Suh, H., Porter, J.R., Hiniker, A. and Kientz, J.A. 2014. @BabySteps: Design and evaluation of a system for using twitter for tracking children's developmental milestones. *ACM CHI Conference on Human Factors in Computing Systems* (2014), 2279–2288. doi 10.1145/2556288.2557386.

Cited in 9.2.1, and A.1.4

[154] Pina, L.R., Sien, S.W., Ward, T., Yip, J.C., Munson, S.A., Fogarty, J. and Kientz, J.A. 2017. From personal informatics to family informatics: Understanding family practices around health monitoring. *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW.* (2017), 2300–2315. doi 10.1145/2998181.2998362.

Cited in 9.2.1

[155] Eisenberg, M. and Fischer, G. 1994. Programmable design environments: integrating end-user programming with domain-oriented assistance. *Proceedings of the SIGCHI conference on Human factors in computing systems celebrating interdependence - CHI '94* (Boston, Massachusetts, United States, 1994), 431–437. doi 10.1145/191666.191813.

Cited in 9.2.2

[156] Prähofer, H., Hurnaus, D., Schatz, R., Wirth, C. and Mössenbö k, H. 2008. Software support for building end-user programming environments in the automation domain. *Proceedings of the 4th international workshop on End-user software engineering* (New York, NY, USA, May 2008), 76–80. doi 10.1145/1370847.1370864.

Cited in 9.2.2

[157] Desolda, G., Ardito, C. and Matera, M. 2017. Empowering End Users to Customize their Smart Environments: Model, Composition Paradigms, and Domain-Specific Tools. *ACM Transactions on Computer-Human Interaction*. 24, (Apr. 2017), 12:1-12:52. doi 10.1145/3057859.

Cited in 9.2.2

[158] Airtable The platform to build next-gen apps: *https://www.airtable.com/*. Accessed: 2023-09-15.

Cited in 9.2.2

[159] typeguard, Inc. Glide: *https://go.glideapps.com/*. Accessed: 2023-09-15.

Cited in 9.2.2

[160] Behr, D., Meitinger, K., Braun, M. and Kaczmirek, L. 2017. *Web probing - implementing probing techniques from cognitive interviewing in web surveys with the goal to assess the validity of survey questions (Version 1.0)*. GESIS - Leibniz-Institut für Sozialwissenschaften. doi 10.15465/gesis-sg_en_023.

Cited in 9.3

[161] Meitinger, K. and Kunz, T. 2022. Visual Design and Cognition in List-Style Open-Ended Questions in Web Probing. *Sociological Methods & Research*. (Feb. 2022), 00491241221077241. doi 10.1177/00491241221077241.

Cited in 9.3

[162] Keusch, F. 2014. The Influence of Answer Box Format on Response Behavior on List-Style Open-Ended Questions. *Journal of Survey Statistics and Methodology*. 2, (Sep. 2014), 305–322. doi 10.1093/jssam/smu007.

Cited in 9.3

[163] McNutt, A. and Chugh, R. 2021. Integrated visualization editing via parameterized declarative templates. *ACM CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2021). doi 10.1145/3411764.3445356.

Cited in 9.4.8

[164] Resnick, M. and Rosenbaum, E. 2013. Designing for tinkerability. *Design, make, play*. Routledge. 163–181.

Cited in 9.4.12, 9.9.1, and 10.6

[165] Luo, T. 2011. The Effects of Tinkerability on Novice Programming Skill Acquisition. *E-Learn 2011--World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education*. (2011), 742–748.

Cited in 9.4.12, 9.9.1, and 10.6

[166] Bangor, A., Kortum, P. and Miller, J. 2009. Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale. *Journal of Usability Studies*. 4, (2009), 114–123.

Cited in 9.8.7, and 9.8.7

[167] Watson, S.M. 2013. Living with Data: Personal Data Uses of the Quantified Self. (2013), 48.

Cited in 9.9.6

[168] Liang, Z., Ploderer, B., Liu, W., Nagata, Y., Bailey, J., Kulik, L. and Li, Y. 2016. SleepExplorer: a visualization tool to make sense of correlations between personal sleep data and contextual factors. *Personal and Ubiquitous Computing*. 20, (2016), 985–1000. doi 10.1007/s00779-016-0960-6.

Cited in 9.9.6

[169] Daskalova, N., Kyi, E., Ouyang, K., Borem, A., Chen, S., Park, S.H., Nugent, N. and Huang, J. 2021. Self-e: Smartphone-supported guidance for customizable self-experimentation. *Conference on Human Factors in Computing Systems - Proceedings*. (2021). doi 10.1145/3411764.3445100.

Cited in 9.9.6

[170] Turing, A.M. 1937. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*. s2-42, (1937), 230–265. doi 10.1112/plms/s2-42.1.230.

Cited in 10.1

[171] Bhardwaj, A., Bhattacherjee, S., Chavan, A., Deshpande, A., Elmore, A.J., Madden, S. and Parameswaran, A.G. 2014. Datahub: Collaborative data science & dataset version management at scale.

Cited in 10.1.4

[172] Google Inc. Firebase | Google's Mobile and Web App Development Platform: *https://firebase.google.com/*. Accessed: 2024-07-29.

Cited in 10.1.4

[173] Verou, L. 2023. Eigensolutions: composability as the antidote to overfit • Lea Verou: *https://lea.verou.me/blog/2023/eigensolutions/*. Accessed: 2024-08-29.

Cited in 10.1.4

[174] Bryan, P.C. and Nottingham, M. 2013. JavaScript Object Notation (JSON) Patch. RFC 6902. Internet Engineering Task Force: *https://datatracker.ietf.org/doc/rfc6902*. Accessed: 2024-08-29. doi 10.17487/RFC6902.
Cited in 10.1.4

[175] Letia, M., Preguiça, N. and Shapiro, M. 2009. CRDTs: Consistency without concurrency control: *http://arxiv.org/abs/0907.0929*. Accessed: 2024-08-29.
Cited in 10.1.4

[176] Sanchez, D. 2018. *Adding sorting and grouping to the Mavo framework for end user web application authoring* (Master's thesis, Massachusetts Institute of Technology).
Cited in 10.3

[177] Nardi, B.A. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. MIT press.
Cited in 10.4

[178] Jones, S.P., Blackwell, A. and Burnett, M. 2003. A user-centred approach to functions in Excel. *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, Aug. 2003), 165–176. doi 10.1145/944705.944721.
Cited in 10.4.2

[179] Sestoft, P. and Sørensen, J.Z. 2013. Sheet-Defined Functions: Implementation and Initial Evaluation. *End-User Development* (Berlin, Heidelberg, 2013), 88–103. doi 10.1007/978-3-642-38706-7_8.
Cited in 10.4.2

[180] 2023. Accessible Rich Internet Applications (WAI-ARIA) 1.2. W3C: *https://www.w3.org/TR/wai-aria/*. Accessed: 2024-08-05.
Cited in 10.6.2

[181] Berners-Lee, T., Bray, T., Connolly, D., Cotton, P., Fielding, R., Jeckle, M., Lilley, C., Mendelsohn, N., Orchard, D., Walsh, N., and others 2004. Architecture of the World Wide Web, Volume One: *https://www.w3.org/TR/webarch/*.
Cited in 10.6.3