HOCHSCHULE
DER MEDIEN

Bachelor's Thesis

Implementation of a modular schema library in TypeScript with focus on bundle size and performance

presented by Fabian Hiller
at Stuttgart Media University on September 4, 2023
for the academic degree Bachelor of Science

First Examiner: Prof. Walter Kriha
Second Examiner: Miško Hevery
Practical Supervisor: Ryan Carniato

Abstract

The amount of JavaScript code in web applications is increasing year by year. Since this can have a negative impact on performance, this thesis uses schema libraries to investigate how the JavaScript bundle size can be reduced. It also examines what factors affect startup and runtime performance, as well as tooling in Visual Studio Code. To answer the research question, a modular and type-safe schema library called Valibot was implemented and compared to existing solutions. The research indicated that a modular design can significantly reduce bundle size, as tree shaking and code splitting allow JavaScript code that is not used to be removed by a bundler in the build step. A performance benchmark showed that Valibot's modularity positively affects startup performance, but other optimizations play a more critical role in runtime performance.

Contents

List of Figures

List of Listings

List of Tables

List of Abbreviations

API Application Programming Interface

CDN Content Delivery Network

TTI Time to Interactive

SEO Search Engine Optimization

1 Introduction

Zod, a TypeScript schema library that enables type-safe validation of structural data with static type inference, has seen rising popularity and adoption since its release three years ago [1, 2]. Its weekly downloads have increased over the past 12 months from 0.8 million to 5 million, indicating a growth rate of 525 % [3].

Responsible for this success is, among other things, the trend towards more robust software and better developer experience through TypeScript as well as the growing ecosystem of Zod, which makes the schema library versatile [4, 5, 6]. There are interfaces for API libraries, such as tRPC, and form integrations, such as React Hook Form, making Zod used in the front and back end [6].

1.1 Motivation

Regardless of the functionality and complexity of a schema, the minified and compressed bundle size of Zod in version 3.21.4 starts at 11.8 kB [7]. Since validating a string with a simple JavaScript function using an if condition requires less than 0.1 kB, the question arises whether the same functionality and developer experience can be achieved with a significantly smaller bundle size.

```
function string(input: unknown): string {
  if (typeof input !== "string") {
    throw new Error("Invalid type");
  }
  return input;
}
```

Listing 1: TypeScript function to validate if an unknown input is a string

The motivation behind this question is influenced by the current developments of widely used web frameworks that try to optimize the performance and reduce the JavaScript bundle size that has to be downloaded by the browser initially. Examples include React with React Server Components, Astro with Astro Islands, and Qwik with Resumability. [8, 9, 10]

One reason for these developments is that websites and web apps tend to become more interactive and functional, which leads to the fact that current JavaScript frameworks require more and more JavaScript to render a website. This substantially impacts the Time to Interactive (TTI) benchmark, which is particularly important for low-powered devices such as smartphones [11, 12]. Furthermore, as long as electricity is also generated from fossil fuels, downloading more JavaScript than needed harms the CO2 balance of a website [13].

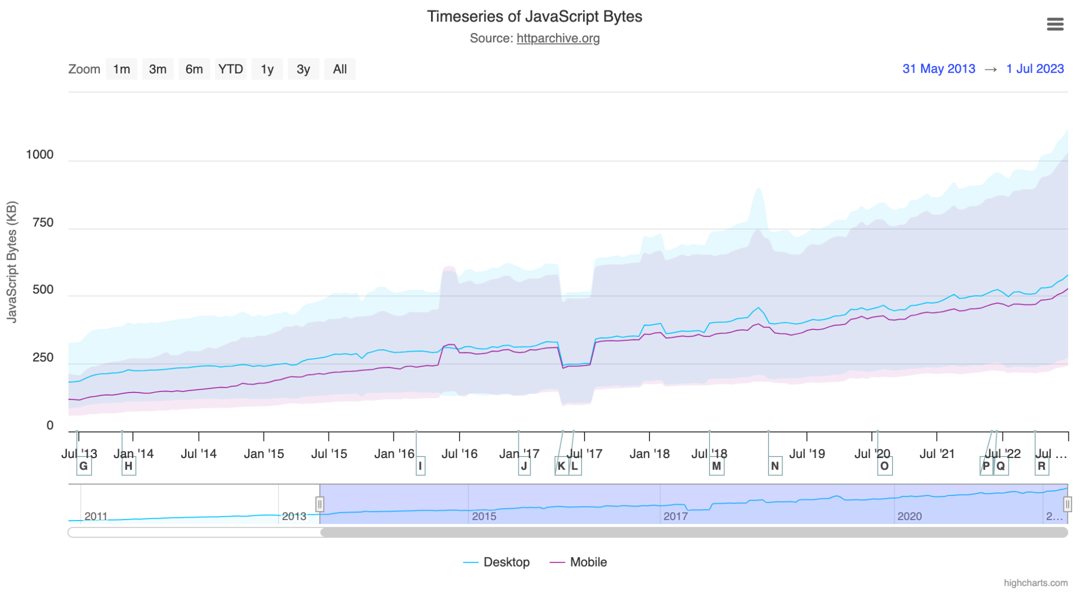


Figure 1.1: Time series of JavaScript bytes used in websites over the last 10 years [14]

Furthermore, more and more developers use full-stack frameworks such as Next.js and Nuxt, making it particularly easy to use the same code in the front and back end, sometimes even within the same file [15, 16, 17]. If the validation in the front and back end was previously implemented completely separately and partly also with different libraries, a schema library that works with forms libraries offers the possibility to use the same schema for the validation of the input in the browser as well as on the server [18, 19].

This makes the validation between front and back end consistent by default and reduces the complexity of the source code since the same logic does not have to be implemented twice. It also improves type safety between the front and back end since only one source, the schema, defines the type definition for both sides. [18, 19]

1.2 Objective

This paper examines what factors influence bundle size and how JavaScript code can be reduced in the context of a library when used in other projects. The aim is to investigate how software with a completely modular structure affects the bundle size and what the consequences of this approach are for performance and developer experience.

Part of this paper is also to investigate current schema libraries and, with a focus on modularity and type safety, to develop a new schema library that enables current bundlers to reduce the final bundle size of the portion of the library used to a minimum. It will also be investigated how such a schema library affects the runtime performance and the tooling in Microsoft's code editor Visual Studio Code.

The results of this paper should be compared at the end with current schema libraries. However, it should not be about functional parity but the differences resulting from another architecture and implementation of the software. This procedure should ensure that the findings of this paper can also be used to improve libraries with entirely different functionality.

1.3 Structure

At the beginning of this paper, relevant basics that are important for the main part are explained. This includes TypeScript and the importance of type definitions and type safety. Then validation is covered, and in which use cases it is helpful or necessary. At the end of the fundamentals, technical terms that will be referenced several times in the main part of the thesis are explained.

Since web development is very versatile and would exceed the scope of this paper to explain every technical term, this paper assumes basic knowledge about the web and the JavaScript programming language. Thus, this thesis is aimed at a specific technical audience and will likely be difficult to understand for the general public.

The main part starts with the analysis of current schema libraries. In addition to functionality, highlighting differences in API design and implementation will be particularly important here. This is followed by investigations into bundle size, performance, and tooling.

The results of this research then provide the basis for the implementation of a new schema library. Apart from the implementation and functionality, the library's practical use is shown and compared with the previously analyzed schema libraries.

At the end of this paper, the final results are evaluated, and the advantages and disadvantages are worked out. Part of the conclusion is also the future of the newly created schema library as an open-source project.

2 Fundamentals

In preparation for the research and results, this chapter provides the fundamentals necessary to understand the main part of the thesis. In addition to TypeScript and various technical terms, validation, and its use cases are explained in detail.

2.1 TypeScript

TypeScript is a programming language developed by Microsoft based on the ECMAScript standard. This makes it a superset of JavaScript with additional features such as static typing and interfaces. Before execution, for example, in the browser, the programming language is transformed to JavaScript by the TypeScript compiler. [5, 20]

2.1.1 Type Definition

JavaScript is a dynamically typed language, which has the disadvantage that you cannot be sure about the data type of a value [21]. Therefore, for larger and more complex programs, it is necessary to understand every context and process so that a wrong data type is not accidentally used, leading to bugs that can only be detected at runtime or with explicit tests [22].

TypeScript solves this problem with type definitions. For example, assigning the data type `number` to a variable can prevent it from accidentally passing a `string` in the following program code since the code editor and the TypeScript compiler will alert the programmer regarding this problem [5, 23, 20].

```
let value: number;
value = 123; // This line works
value = "123"; // But this causes an error
```

Listing 2: Example that causes a TypeScript compiler error

However, it should be noted that the TypeScript compiler removes the type definitions when transforming to JavaScript. Therefore, they only indicate problems during development and compile time. At runtime, there is no validation that guarantees type safety. [5, 23]

2.1.2 Type Safety

The thorough assignments of type definitions lead to type safety. This is important not only to prevent bugs but also to enhance the developer experience. With precise typing, a code editor like Visual Studio Code can display suggestions and autocomplete parts of the program code for the programmer. [23]

It also increases maintainability and helps other developers to understand program code more quickly. Type safety is, therefore, particularly important for external libraries since the type definitions also partially document the source code. [22]

2.2 Validation

Validation is a process in which something is checked against certain specifications [24]. In the context of a schema library, it is structured data. In contrast to unstructured data such as image and audio files, the information in structured data is subject to a predefined schema. An example of this is the customer and product data of an online store.

Validation is required when the creator or sender of data cannot be trusted [24]. Furthermore, validation can also be used to point out the outstanding requirements of the input when capturing data, for example, with forms, to enhance the user experience [25].

2.2.1 If/Else Conditions

In TypeScript, single values can be validated with conditional statements. The programming language can even automatically assign the correct type definition to unknown data in certain cases, for example, when using the `typeof` keyword [26].

```
if (typeof input === "string") {
  // `input` is of type `string`
}
```

Listing 3: TypeScript example code with a `typeof` type guard

However, for more complex and dynamic data, such as objects and arrays, TypeScript is no longer fully capable of automatically typing this data using conditional statements. Also, if/else conditions become confusing with large data sets, making the code error-prone due to the lack of typing.

2.2.2 Schemas Validation

A schema that can be used to validate data is an abstraction layer that uses conditional statements internally but provides a concise API externally [27]. TypeScript does not have this functionality, so external solutions, such as Zod, are used [1].

```
const LoginSchema = z.object({
  email: z.string()
    .min(1, "Please enter your email.")
    .email("The email address is badly formatted."),
  password: z.string()
    .min(1, "Please enter your password.")
    .min(8, "Your password must have 8 characters or more."),
});
```

Listing 4: Zod schema mapping the data of a login form

The advantage of such a schema is that it is similar to the code of a TypeScript type definition, and with Zod, it is also technically possible to derive the type definition of a schema [28]. Thus, even extensive and complex data sets can be mapped and typed [29, 30]. Furthermore, less code has to be written by the developer in total for the same functionality.

```
// TypeScript
type Object1 = Partial<{ key: string }>;
type Object2 = Pick<Object1, "key">;

// Zod Schema
const object1 = z.partial(z.object({ key: z.string() }));
const object2 = z.pick(object1, ["key"]);
```

Listing 5: Similarities between TypeScript and a Zod schema

2.3 Use Cases

The use cases of schema validation can be quite different. They can be part of the security concept of a software to prevent targeted attacks and manipulations, but they can also help improve an application's developer and user experience [24, 25].

2.3.1 Server Requests

For external requests on a server, the validation of the incoming data is important for the security and functionality of a service. Validation must, on the one hand, prevent attackers from deliberately manipulating the service and, on the other, prevent unexpected errors from occurring due to incorrect input by regular users [24]. Furthermore, the request data must be correctly typed since it is usually declared `unknown` or `any` by default [31].

2.3.2 Form Validation

A schema can be used in many ways and offers various advantages when creating forms. For example, a type-safe schema prevents a mismatch between the form's type definition and the input validation since both extend from the same source. Combined with a type-safe form library such as Modular Forms, this can reduce error-proneness and enhance the developer experience. [32, 18]

In addition, the same schema can be used for input validation in the browser and on the server. This reduces the lines of code needed for validation to a minimum and prevents a mismatch between the front and back end. Client-side validation increases performance and usability, as input can be validated directly on the user's device, and server-side validation prevents attacks and tampering. [25, 18, 19]

As a result, the user receives immediate feedback on any outstanding requirements, and in the end, only a single network request is necessary to send the final data to the back end successfully [25].

2.3.3 Config Files

Configuration files are another use case in which schemas can be helpful. By validating configuration data ahead of time, errors can be detected and fixed directly using informative error messages. A concrete example are environment variables, which are often located in the local development environment in `*.env` files and must be manually transferred to the hosting service before deploying to production [33].

2.4 Terminology

Technical terms that are particularly important for this paper and may have a different definition depending on the context are explained in detail below and defined in the context of the thesis.

2.4.1 Bundle Size

Bundle size is the memory size of program code measured in bytes. It plays an essential role for web applications that are transferred to the browser via the Internet since the number of transmitted bytes influences performance, usability, and Search Engine Optimization (SEO). Especially with a poor network connection and slow hardware, a large JavaScript bundle size has a negative effect because, on the one hand, the transfer of the JavaScript files takes time, and on the other, the entire code has to be parsed by the browser engine before execution. [34, 35, 36]

Besides optimizing the source code, tree shaking, code splitting, minification, and compression can be applied afterward to reduce the final bundle size send over the network [37, 38, 39, 40].

Tree Shaking

Tree shaking is the process of removing software parts that are demonstrably not used. A bundler such as Webpack, Rollup, or esbuild can identify this, for example, by the import statements of JavaScript modules. If only a particular module is imported, the remaining program code unrelated to the imported part can be removed. [37, 41]

Code Splitting

Code splitting is similar to tree shaking, with the difference that code is not removed but split into several chunks. This has the advantage that only the part of the JavaScript code that is required for a respective action must be initially transferred. [38, 42]

However, in contrast to tree shaking, code splitting cannot be applied across the board since the structure of the own program code, as well as the libraries and frameworks used, have a strong influence on it. For example, Angular, React, and Vue require the entire source code of a page to render the HTML elements and place the event listeners, which makes code splitting partially impossible. [43, 44]

Minification

Minification reduces the number of characters in the source code without changing its functionality. For example, unnecessary characters such as spaces, line breaks, and comments are removed. Furthermore, functions and variables can be renamed to single characters, and smaller parts of the source code can be restructured. [39]

Compression

By encoding, data can be compressed so that fewer bytes are sent over the network. In order to not damage the program code, lossless compressions such as gzip or Brotli are used. These can recognize patterns in the program code with special algorithms and thereby represent the same information in a unique format with fewer bytes. [40, 45]

2.4.2 Performance

Performance refers to the ability of a system to run programs quickly and efficiently. Relevant to this paper is the startup and runtime performance. Startup performance or TTI describes how long it takes for a website to be usable after it has been requested. This benchmark is mainly influenced by the network connection, bundle size, program code and the device's hardware and software performance. [46, 47, 48, 49]

The runtime performance is influenced by the program code that is executed, apart from the hardware and software of a device. Since JavaScript is an interpreted language, the JavaScript engine must also be considered in benchmarks. [50, 51]

2.4.3 Modularity

Modularity means that a system is composed of several small independent components. Each component has a specific functionality. Modular software has the advantage that the individual components can be tested, maintained, extended, and reused more easily. [52, 53]

In the case of a JavaScript library, a modular design with independent exports also enables tree shaking and code splitting. However, extreme modularity can compromise the developer experience and increase complexity while maintaining the same functionality. [53, 54]

2.4.4 Tooling

Tooling in the context of this thesis refers to the functionality of a code editor to assist the developer when writing code and thus improve the developer experience. It is not a functionality of the programming language and should not be confused with it. In Visual Studio Code, for example, the tooling can be extended by plugins. [55, 56]

3 Research

In this chapter, three different schema libraries are presented and included in the investigations into bundle size, performance, and tooling. Furthermore, it is examined how the bundle size can be reduced to a minimum and which factors of the program code affect the run time performance.

3.1 Schema Libraries

The schema libraries presented and examined differ significantly in API design and implementation. Thus, they provide an overview of the technical possibilities as well as the advantages and disadvantages when creating a type-safe schema library.

3.1.1 Zod

Zod was initially developed three years ago by Colin McDonnell and is a type-safe schema library whose API design is composed of functions and methods [1, 27]. Internally, Zod is implemented object-oriented with classes [57].

API Design

Each data type Zod supports is mapped with a schema function with the same name as the data type. This procedure makes it possible to define a schema based on functions, which can be used, on the one hand, for type-safe validation at runtime and, on the other hand, for inferring type definitions. [1, 57]

```
// Create product schema with functions
const ProductSchema = z.object({
  title: z.string(),
  description: z.string(),
  price: z.number(),
  currency: z.enum(["USD", "EUR"]),
});

// Infer type of product data from schema
type ProductData = z.infer<typeof ProductSchema>;
```

Listing 6: Example of a simple Zod product schema

When called, a schema function returns an instance of a class with various information and methods. The information contains the properties of the schema. The methods are divided into specific and general methods. The specific methods can be used to further specialize the validation of a data type. For example, a minimum number can be specified with `.min`, and a maximum number with `.max`. [57]

```
const ProductSchema = z.object({
  title: z.string().min(1).max(30),
  description: z.string().min(1).max(500),
  price: z.number().min(1).max(10000),
  currency: z.enum(["USD", "EUR"]),
});
```

Listing 7: Example of an extended Zod product schema

The general methods are available for each schema. They can be used to define how the schema should behave during validation. For example, `.default` can be used to add a default value that will be used if the input is `undefined`, and `.transform` can be used to transform the input after validation. [1, 57]

```
const ProductSchema = z
  .object({
    title: z.string().min(1).max(30),
    description: z.string().min(1).max(500),
    price: z.number().min(1).max(10000).default(100),
    currency: z.enum(["USD", "EUR"]),
  })
  .transform((data) => ({
    ...data,
    createAt: new Date().toISOString(),
  }));
```

Listing 8: Example of an advanced Zod product schema

With `.parse`, `.parseAsync`, `.safeParse`, and `.safeParseAsync`, the schema can be used to validate unknown data. `.parse` and `.parseAsync` work like an assertion function that throws an error if the input does not match the expected data type and returns the output of the schema otherwise. `.safeParse` and `.safeParseAsync` on the other hand return a discriminating union containing either the data or error information. [57, 58, 59]

```
const productData = ProductSchema.parse({
  title: "Apple",
  description: "Red apple from Lake Constance",
  price: 89, // Amount in cents
  currency: "EUR",
});
```

Listing 9: Example of validation with a Zod product schema

Implementation

Internally, Zod uses an object-oriented approach with classes. An abstract class called `ZodType` contains all the general methods from which the other schema-specific classes, such as `ZodNumber` and `ZodNull`, inherit. The classes of the various schema functions contain the schema-specific methods. With a few exceptions, all methods return an instance of their class, making it possible to chain the methods. [57]

```
export abstract class ZodType<...> { ... }

export class ZodNumber extends ZodType<...> { ... }
```

Listing 10: Example of the ZodType and ZodNumber class [57]

To avoid having to instantiate the instances of a class when using Zod with the `new` keyword, each class contains a static `.create` method that returns an instance when called. Zod assigns these static methods to a constant in order to be able to choose the name of the schema functions freely in the `export` statement. This is necessary because a few functions use the name of reserved keywords, such as `null` and `undefined`. [57]

```
export class ZodNull extends ZodType<null, ZodNullDef> {
  static create = (params?: RawCreateParams): ZodNull => {
    return new ZodNull({
      typeName: ZodFirstPartyTypeKind.ZodNull,
      ...processCreateParams(params),
    });
  };
}

const nullType = ZodNull.create;

export { nullType as null };
```

Listing 11: Example of instantiation and export of the ZodNull class [57]

When a schema is created by executing the schema functions and methods, the library internally collects all schema-relevant information. This includes the data type and all specific validation information, such as the minimum and maximum value of a number. If a schema is now used to validate unknown data, this information is picked up in a private `._parse` method to check whether the data conforms to the schema. If a problem is detected, it is added to the context and output via an instance of the `ZodError` class. [57]

3.1.2 ArkType

ArkType has been under development by David Blass for a year and a half and is a schema library that builds heavily on the functionality of TypeScript. Using complex TypeScript types, ArkType makes it possible to define type-safe schemas using only strings, objects, and tuples. [60, 61]

API Design

The API design of ArkType is simple. Instead of several functions, only the `type` function is needed to create a schema. The first argument of this function can be used to define the schema. The idea is to use strings, objects, and tuples to emulate a TypeScript type definition visually. [60, 61]

```
// Create product schema with `type` function
const ProductSchema = type({
  title: "string",
  description: "string",
  price: "number",
  currency: "'USD'|'EUR'",
});

// Infer type of product data from schema
type ProductData = typeof ProductSchema.infer;
```

Listing 12: Example of a simple ArkType product schema

With the help of special characters, the runtime validation can be further individualized. For example, a minimum value can be specified for a number with `X<=` at the beginning and a maximum with `<=X` at the end. The argument of the `type` function is type-safe, which prevents typos, and a code editor like Visual Studio can display suggestions and supports the developer with auto-completion. [61]

```
const ProductSchema = type({
  title: "1<=string<=30",
  description: "1<=string<=500",
  price: "1<=number<=10000",
  currency: "'USD'|'EUR'",
});
```

Listing 13: Example of an extended ArkType product schema

Implementation

When the type function is called, the library initializes a context which is subsequently filled with information from the schema definition of the first argument. If the schema is called with unknown input, the data is parsed using the cached schema information. [61, 62]

```
const { data, problems } = ProductSchema({
  title: "Apple",
  description: "Red apple from Lake Constance",
  price: 89, // Amount in cents
  currency: "EUR",
});
```

Listing 14: Example of validation with a ArkType product schema

The source code of ArkType is composed of various classes that use utility functions in their methods. Another essential part are the complex type definitions of the schema definition, which are implemented with many nested conditional types. Since schema creation originates from a single function, the complexity of this function is high. This is due to the fact that this function must be able to parse and validate every supported data type, regardless of whether a data type is actually used. [61]

3.1.3 Typia

Typia has been under development by Jeongho Nam for more than a year and is a schema library that uses TypeScript types to generate the most optimal JavaScript code possible in a compile step, which can be used for validation at runtime [63, 64].

API Design

Instead of relying on special schema functions or replicating a TypeScript type definition, Typia uses the existing type information. Therefore, no additional code is required to create a schema. The developer only needs to pass the type definition as a generic together with the unknown data to the `validate` function provided by Typia. [63, 65, 64]

```
type Product = {
  title: string;
  description: string;
  price: number;
  currency: "USD" | "EUR";
};
```

Listing 15: Example of a simple Typia product type

In order to validate other specifications apart from the data type, a comment tag can be added to a TypeScript type. This is a comment that starts with the @ symbol. For example, `@minimum` can be used to specify a minimum number, and `@maximum` to specify a maximum number. [66, 64]

```
type Product = {
    /**
     * @minLength 1
     * @maxLength 30
     */
    title: string;
    /**
     * @minLength 1
     * @maxLength 500
     */
    description: string;
    /**
     * @minimum 1
     * @maximum 10000
     */
    price: number;
    currency: "USD" | "EUR";
};
```

Listing 16: Example of an extended Typia product type

Since v5, Typia also supports type tags in addition to comment tags. As the research of this paper is based on Typia v4, type tags are not considered.

Implementation

The functions exported by Typia, such as `validate`, do not contain any logic. They will throw an error if they are executed without the required compile step. Typia replaces the contents of these functions with code generated using the type information passed as generic when transforming from TypeScript to JavaScript. [63, 67]

```
const productResult = typia.validate<Product>({
  title: "Apple",
  description: "Red apple from Lake Constance",
  price: 89, // Amount in cents
  currency: "EUR",
});
```

Listing 17: Example of validation with Typia's `validate` function

3.2 Bundle Size

In order to keep the bundle size as small as possible, the source code must either be optimized for tree shaking, code splitting, minification, and compression, or a compile step is required that individually generates only the validation code required for the respective validation [37, 38, 39, 40]. Since a compile step limits what a library can be used for and is a topic on its own, this paper focuses on the first option.

For example, a compile step prevents the library from being loaded directly from a Content Delivery Network (CDN) into a website via a <`script` /> element. It also excludes environments such as Deno, where an individual compile step is not possible [68]. Furthermore, the library's functionality is also limited with this approach, as it prevents the dynamic creation of schemas at runtime.

3.2.1 Modularity

Optimizing the library for tree shaking and code splitting requires a modular design since every functionality must be exported separately [37, 38]. This excludes the approach of ArkType with the `type` function, which can validate any data type.

Also, Zod's object-oriented approach is not suitable since the various methods of the classes contain more functionality than is needed for many use cases.

In order to achieve the smallest possible initial bundle size, it should also be ensured that the individual functions are reduced to the bare minimum and only contain the code required for each execution. Program code that is necessary only in individual cases should be able to be supplemented, for example, with dependency injection.

3.2.2 Shorthands

Furthermore, shorthand notations of the programming language can be used to reduce the bundle size apart from tree shaking and code splitting [69]. For example, the ternary operator can be used instead of an if/else block to assign a different value with fewer characters to a variable depending on a condition [69, 70].

```
// Longhand with if/else block
let value1;
if (Math.random() > 0.5) {
  value1 = "A";
} else {
  value1 = "B";
}

// Shorthand with ternary operator
const value2 = Math.random() > 0.5 ? "A" : "B";
```

Listing 18: Difference between if/else block and ternary operator

Since shorthand notations make the code more confusing in some cases, it should be taken into account that a bundler can automatically apply some shorthand notations in the build step during minification. For example, if several variables are declared one after the other, it is not necessary to use a shorthand notation since the bundler can perform this optimization during minification afterward.

More shorthand coding techniques can be found in the article "25+ JavaScript Shorthand Coding Techniques" by Michael Wanyoike and Sam Deering [69].

3.2.3 Minification

Another way to reduce the bundle size is to optimize the source code for minification and compression [39, 40]. Ensuring that a bundler can reduce the source code to as few characters as possible when minifying is crucial. For example, to allow a bundler to rewrite the code to the shorthand when declaring variables, all the required variables of a function should be declared one after the other, ordered by `let` and `const`.

```javascript
// Minified: let a=1;const b=2;let c=3;const d=4;
let a = 1;
const b = 2;
let c = 3;
const d = 4;

// Minified: let a=1,c=3;const b=2,d=4;
let a = 1;
let c = 3;
const b = 2;
const d = 4;
```

Listing 19: Difference in minification with SWC as bundler

3.2.4 Compression

On the other hand, to increase the compression rate, it helps if certain patterns are repeated, and names that cannot be renamed during minification occur more than once in a meaningful way. For example, the source code of two functions with similar functionality should also be structured similarly to achieve the highest possible compression. Also, with logic that can be implemented differently, it should be paid attention to a uniform process. For example, that can mean using a `for...of` loop whenever possible instead of sporadically mixing it with `.forEach`.

```
// 111 bytes with gzip
for (const number of [1, 2, 3]) {
  console.log(number);
}
['a', 'b', 'c'].forEach((char) => {
  console.log(char);
});

// 97 bytes with gzip
for (const number of [1, 2, 3]) {
  console.log(number);
}
for (const char of ['a', 'b', 'c']) {
  console.log(char);
}
```

Listing 20: Difference in gzip compression of similar code

The optimization that can be achieved by naming refers to the names of variables, functions, object keys, and the content of strings. In a schema library, for example, it can reduce the bundle size if a function that validates a number is called `number` since the string `"number"` is already required for validation with the `typeof` operator, and thus "number" occurs several times in the source code. Also, error messages should follow a uniform structure to increase the compression rate by the repeated occurrence of the same words.

```
function number(input) {
  if (typeof input !== "number") {
    throw new Error("Input is not a number");
  }
  return input;
}
```

Listing 21: Validation function in which the word "number" occurs 3 times

3.3 Performance

Three areas are relevant for examining the performance of a schema library. These include downloading and parsing the program code, creating or initializing a schema, and validating unknown data [47, 46]. The downloading and parsing of the program code and the direct initialization of global schemas affect the startup performance. However, this is only important if the library is executed directly in the browser as part of a website. For runtime performance, which affects the front and back end, the initialization of a schema and the validation of data must be considered.

The investigations of this section are primarily concerned with initialization and validation. As described in the fundamentals, the time required for downloading and parsing the program code, apart from external factors such as the hardware and network connection, is primarily impacted by the bundle size, which was already covered in the previous section.

3.3.1 Initialization

Before running more specific performance tests and using analysis tools like the Chrome DevTools, the source code should be read thoroughly to understand what happens during initialization and validation.

Zod, for example, creates instances of the internally used classes during initialization and adds all the validation specifications, such as the minimum and maximum number, one after the other, to the context of an instance in the form of an array [57]. ArkType behaves similarly but must additionally parse the strings, objects, and tuplets beforehand, which requires additional CPU time [61]. Based on the source code, it can be assumed that Zod is faster than ArkType for initialization.

To optimize the performance of the initialization, the API design and the implementation of a schema library must be geared towards executing as little code as possible and generating as few objects and arrays as possible. All required information should already be available initially in order to avoid having to collect it during initialization.

Typia is an exception here since no initialization is done at runtime. Typia generates an optimized validation function at compile time for each type definition used for validation, which can be executed at runtime without initialization [63].

3.3.2 Validation

During validation, Zod first checks the data type of the schema in the private `._parse` method and then executes further conditions or a loop with conditions to also check the specifications collected during initialization. Each if condition checks the input for a specific property and adds an entry with error information to the context in case of an error. If the input is a complex data type, for example, an array, Zod executes the `._parse` method of the nested schemas in a loop for each entry. [57]

ArkType is implemented differently internally but technically works similarly. When validation is performed, a recursive loop validates the input using the information collected during initialization [71]. By collecting information and creating various objects, the code of both libraries indicates that initialization requires more memory and CPU time than successful validation, and validation in case of error is slower than in case of success due to the creation of error information.

Since the generated code of Typia is almost exclusively in a single function and only contains conditions for the required checks, the validation can be executed more efficiently and thus with fewer resources than with Zod and ArkType [63]. Typia can also reduce the number of loops to a minimum, which positively impacts performance.

3.3.3 Bottlenecks

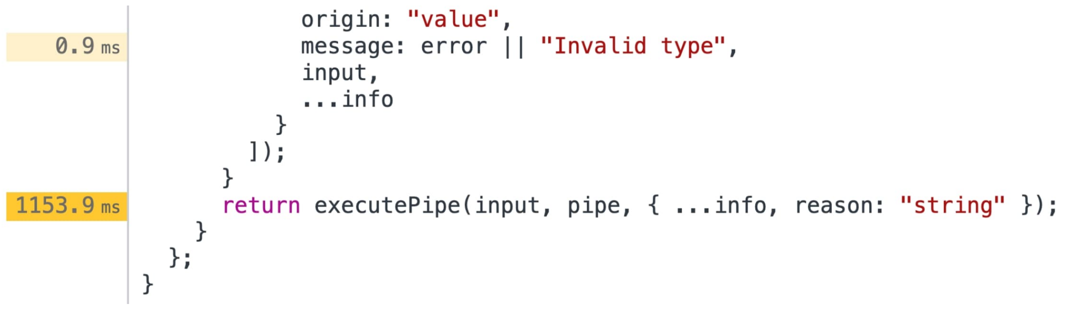
The code required for validation can only be reduced to a limited extent. Therefore, it is essential to avoid known performance bottlenecks for the code that is mandatory for the library's functionality. This includes, for example, reducing the creation of new objects and arrays and reusing existing ones as much as possible. Also, the spread operator should be avoided within a loop to fill an array since this increases the execution time exponentially with each pass. Changing a single line that did not consider this in the open-source library React Table improved performance by 100,000 % in one use case [72].

```
// Create immutable array with spread operator
map.set(resKey, [...previous, row]);

// Mutates existing array with `.push` (faster)
previous.push(row);
```

Listing 22: Line of code that improved performance of React Table [72]

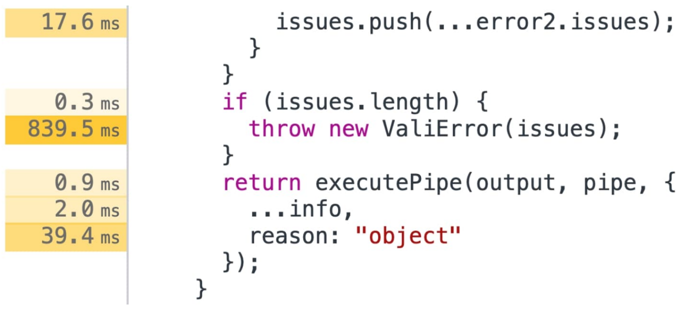
Another bottleneck that should be avoided is to use the spread operator to copy the contents of a known object into a new one. As in the previous example, the spread operator performs a dynamic operation for each entry to add every key-value pair to the new object. However, this is not necessary if the object is known because the assignment can be executed directly. [73]



```
                            origin: "value",
      0.9 ms                message: error || "Invalid type",
                            input,
                            ...info
                          }
                        ]);
                      }
   1153.9 ms          return executePipe(input, pipe, { ...info, reason: "string" });
                    }
                  };
                }
```

Figure 3.1: Signs of a hotspot in Chrome DevTool caused by the spread operator

Another bottleneck can be the internal error handling of a validation library since JavaScript provides a specific syntax for this with try/catch, which is, therefore, obvious to use. Furthermore, throwing errors with the JavaScript `Error` class or a class that inherits from it is also obvious. However, removing both has increased Zod's performance in case of errors by up to 20,000 % [74].



```
     17.6 ms              issues.push(...error2.issues);
                        }
                      }
      0.3 ms          if (issues.length) {
    839.5 ms            throw new ValiError(issues);
                      }
      0.9 ms          return executePipe(output, pipe, {
      2.0 ms            ...info,
     39.4 ms            reason: "object"
                      });
                    }
```

Figure 3.2: Signs of a hotspot in Chrome DevTool caused by throwing errors

The `Error` class is a bottleneck because it captures additional information, such as the stack trace, apart from the error message. However, this is only required for the final error thrown by a schema library. Removing only the error class increased validation performance for invalid data by almost 300 % in Zod [75].

The performance improvement that can be achieved by avoiding try/catch blocks is due to the fact that it takes less CPU time to add an object with error information to an array instead of throwing it as an error and catching it in a `catch` block. It is important to note that this is only about the internal error handling of the library. Externally, an error can still be thrown at the end of validation.

In order to find performance bottlenecks like the ones just mentioned, analysis tools like Chrome DevTools can be used. Through performance and memory profiling, it is possible to examine which functions take up the most CPU time and memory to find hotspots within the program code. [50]

3.3.4 Benchmarks

In order to be able to classify the results of the next chapter through comparative values, a benchmark was prepared for a specific use case. This benchmark covers the download, initialization, and validation of a product schema that contains dates and nested objects within arrays, apart from strings and numbers. The data was measured with Zod v3.22.2, ArkType v1.0.19-alpha, and Typia v4.3.2. [76]

To calculate the download time, the same schema was implemented in each of the three schema libraries, and the final JavaScript output was bundled, minimized, and compressed. Therefore, the test considers the library code and the custom code required to create the schema and validate unknown data. The final bundle size was used to calculate how fast the minimized and compressed program code can be downloaded over a 3G, 4G, and 5G connection.

Library	**Minified**	**Gzipped**	**Compression**
Zod	53.7 kB	12.9 kB	76 %
ArkType	47.7 kB	16.2 kB	66 %
Typia	35.8 kB	9.72 kB	73 %

Table 3.1: Bundle size of the same product schema with different libraries

Library	1.5 Mbit/s	15 Mbit/s	150 Mbit/s
Zod	69 ms	7 ms	1 ms
ArkType	86 ms	9 ms	1 ms
Typia	52 ms	5 ms	1 ms

Table 3.2: Download speed of the product schema at 3G, 4G, and 5G

To calculate the time of initialization and validation, a script was implemented that executes the operations and stops time in parallel. Each operation executes in a loop 10, 100, 1,000, 10,000, and 100,000 times to take into account that a JavaScript engine like Google's V8 can optimize the code at runtime. Each benchmark runs five times for each number of repetitions to reduce random variation by averaging. Benchmarks were run using Node.js v18.15.0 and a 2019 MacBook Pro with a 2.3 GHz 8-core Intel Core i9 CPU and 32 GB of 2667 MHz DDR4 memory.

Library	10	100	1,000	10,000	100,000
Zod	8 ms	35 ms	161 ms	1 s 175 ms	11 s 201 ms
ArkType	14 ms	56 ms	315 ms	2 s 345 ms	21 s 862 ms
Typia	-	-	-	-	-

Table 3.3: Initialization of the same product schema with different libraries

Library	10	100	1,000	10,000	100,000
Zod	7 ms	22 ms	75 ms	443 ms	3 s 866 ms
ArkType	3 ms	12 ms	61 ms	449 ms	4 s 139 ms
Typia	1 ms	1 ms	5 ms	37 ms	140 ms

Table 3.4: Successful validation of the same product schema with different libraries

Library	10	100	1,000	10,000	100,000
Zod	10 ms	39 ms	207 ms	1 s 581 ms	15 s 134 ms
ArkType	5 ms	25 ms	149 ms	1 s 191 ms	11 s 307 ms
Typia	2 ms	5 ms	21 ms	126 ms	769 ms

Table 3.5: Erroneous validation of the same product schema with different libraries

The investigations have shown that Typia's bundle size is the smallest and that the compile step, which generates optimal JavaScript code, leads to significant advantages in run-time performance. Since the current implementation of Typia prevents tree shaking and thus the entire library must be imported, it is theoretically possible to reduce the bundle size to less than 2 kB in this use case [77].

When comparing Zod and ArkType, it is noticeable that Zod's minified bundle size is larger, but the compressed bundle size is smaller than ArkType's because Zod's code can be compressed more efficiently. For schema initialization, Zod performs better, but for validation, ArkType performs better. However, it is interesting to note that Zod is faster than ArkType in successfully validating product data from 10,000 runs.

David Blass, the author of ArkType, is currently experimenting with the `Function` constructor to dynamically generate a function with optimal validation code at initialization to increase the performance of ArkType [78, 79]. An early test version of ArkType with the `Function` constructor, provided exclusively for this paper, increased initialization time by 324 % but was able to decrease validation time by 70 to 88 %.

3.4 Tooling

Besides a small bundle size and good performance, tooling should also be examined to ensure a good developer experience. This includes the automatic import of modules, code suggestions, auto-completion, and error highlighting. A distinction must be made between the tooling that Visual Studio Code includes by default and tooling supplemented by extensions. AI tools such as GitHub Copilot are not considered in this paper.

3.4.1 Zod

Zod has excellent tooling by default due to its API design. This starts with the fact that when "z" is typed into the editor, Visual Studio Code suggests importing the `z` object from Zod. By adding a point, IntelliSense displays the contents of the `z` object, and by adding more characters, this selection can be filtered. When selecting a suggestion, the editor completes the missing code. Because Zod does not export its functionality individually, the `z` object contains, besides various types, almost exclusively the schema functions, which makes the selection concise.

After creating a schema by calling a schema function, IntelliSense again makes suggestions for a schema's general and specific methods by adding a point. With the ability to chain most methods, Zod can thus provide a good developer experience. Due to the type-safety of the API, many bugs caused by typos can be prevented in advance, as the editor immediately displays an error if, for example, an incorrect data type is passed as an argument to a method.

3.4.2 ArkType

Like with Zod, ArkType's `type` function can be imported automatically by Visual Studio code. Since the `type` function is typed in detail, the editor suggests possible data types supported by ArkType when entering an empty string. As with Zod, the suggestions can be filtered by entering additional characters. If a suggestion is selected, the string is automatically completed. Also, typos can be prevented by the type definition. The development experience is, therefore, similar to Zod up to this point.

However, there is a difference when adding further specifications to the validation. These cannot be suggested and completed in some cases, such as when defining a minimum and maximum number. An advantage, however, is the ArkType extension for Visual Studio Code, which highlights the strings in various colors, making it easier to see a schema's data types than with Zod. Another advantage is that when the mouse pointer is moved over the `type` function, the final type definition of the schema is displayed.

3.4.3 Typia

Typia does not require specific tooling initially, as it builds on existing TypeScript types [63]. Once the validation becomes more detailed, Typia uses comment tags as an API to define additional specifications [66]. Since the comment tags Typia uses are not standardized, the editor cannot show any suggestions for them. Also, by default, Visual Studio Code cannot give a hint if a typo is made. Since there is currently no extension that supplements this functionality, the developer experience in this regard is worse than with Zod and ArkType.

4 Results

The schema library created as part of this thesis is called Valibot. Within a few weeks after the announcement, it reached 3,000 stars on GitHub and 25,000 downloads on npm [80, 81]. Furthermore, an ecosystem has already grown with integrations for tRPC, React Hook Form, VeeValidate, Hono, Drizzle ORM, and several other projects [82].

In this chapter, the results of Valibot are presented. Decisions regarding the API design and implementation are explained and the library's bundle size, performance, and tooling are compared with Zod, ArkType, and Typia.

4.1 Demonstration

The core functionality of Valibot is the creation of a schema that can be used at runtime to validate unknown data. In addition, the library is fully type-safe, allowing the schema's type definition to be inferred.

4.1.1 Schemas

Similar to how types can be defined in TypeScript, Valibot allows the creation of a schema with various small functions. This applies to primitive values like strings as well as more complex data sets like objects and arrays.

```
// TypeScript
type LoginForm = {
  email: string;
  password: string;
};

// Valibot
const LoginSchema = object({
  email: string(),
  password: string(),
});
```

Listing 23: Similarities between a TypeScript Type and a Schema

In addition, the library allows it to perform more detailed validations and transformations with pipelines. For example, this can be used to ensure that a string is an email and ends with a specific domain. Pipelines are optional and always added in the form of an array as the last argument of a schema function.

```
const EmailSchema = string([email(), endsWith("@example.com")]);
```

Listing 24: Example of the pipelines feature for validating an email

An object is returned with helpful information if an issue is detected during validation. The included error messages can be overridden by the first optional argument of a schema or validation function. Custom error messages can be used to improve the usability of an application by providing specific troubleshooting hints and returning error messages in the user's native language.

```
const LoginSchema = object({
  email: string("Your email must be a string.", [
    minLength(1, "Please enter your email."),
    email("The email address is badly formatted."),
  ]),
  password: string("Your password must be a string.", [
    minLength(1, "Please enter your password."),
    minLength(8, "Your password must have 8 characters or more."),
  ]),
});
```

Listing 25: Example of a login schema with individual error messages

Since Valibot offers the possibility to transform the data type after validation, there may be a difference between a schema's input and output when deriving its type definition. The input can be inferred with the `Input` and the output with the `Output` type. If no transformation is applied, the input and output types are identical.

```
type LoginInput = Input<typeof LoginSchema>;
type LoginOutput = Output<typeof LoginSchema>;
```

Listing 26: Example of inferring the input and output type of a schema

4.1.2 Validation

Valibot provides three different APIs to check if unknown data matches a schema. This allows developers to choose which API better fits the code base. With `parse`, Valibot throws an error if the data does not match the schema and otherwise returns its output. It is an assertion function that can be used with the try/catch syntax of

JavaScript.

```
try {
  const EmailSchema = string([email()]);
  const email = parse(EmailSchema, "jane@example.com");

  // Handle error if one occurs
} catch (error) {
  console.log(error);
}
```

Listing 27: Example of the `parse` function with a try/catch block

If the error information should be returned instead of being thrown, `safeParse` can be used. This function returns a discriminated union, which either returns the output of the schema if `.success` has the value `true` or the issues that occurred otherwise.

```
const EmailSchema = string([email()]);
const result = safeParse(EmailSchema, "jane@example.com");

if (result.success) {
  const output = result.output;
} else {
  console.log(result.issues);
}
```

Listing 28: Example of the `safeParse` function with an if/else condition

A type guard is another way to validate data that can be useful in some cases. It is an if-condition which, if `true`, sets the parsed data to the input type of a schema. The error information cannot be accessed if a type guard is used. Also, transformations have no effect, and unknown keys of objects are not removed. Therefore, this approach is less safe and capable than `parse` and `safeParse`.

```
const EmailSchema = string([email()]);
const data: unknown = "jane@example.com";

if (is(EmailSchema, data)) {
  const email = data; // string
}
```

Listing 29: Example of the `is` type guard with an if condition

4.1.3 Methods

Strictly speaking, methods are modification functions. However, since they do the job that methods usually do, they bear this designation in the context of the library. The methods either add additional functionality to a schema, create a new schema based on an existing schema, or help to use a schema.

For example, `coerce` can be used to coerce the input of a schema in order to change the data type before the validation starts. This can be used, for example, to convert a string of numbers to a number. Another method is `transform`. It works similarly to `coerce`, with the difference that the function of the second parameter is executed only after the validation.

```
const NumberSchema = coerce(number(), Number);
const numberOutput = parse(NumberSchema, "1234"); // 1234

const StringSchema = transform(string(), (input) => input.length);
const stringOutput = parse(StringSchema, "hello"); // 5
```

Listing 30: Example using the `coerce` and `transform` method

Apart from the general schema methods, there are special methods for object schemas that are strongly oriented towards the functionality of TypeScript. For example, the values of an object can be made optional with `partial` or required with `required`. With `merge`, multiple object schemas can be merged, and with `pick` or `omit`, specific values of an existing object can be included or excluded.

```
// TypeScript
type Object1 = Partial<{ key1: string; key2: number }>;
type Object2 = Pick<Object1, "key1">;

// Schema Library
const object1 = partial(object({ key1: string(), key2: number() }));
const object2 = pick(object1, ["key1"]);
```

Listing 31: Similarities between TypeScript and the schema library

4.2 API Design

The API design of Valibot is primarily geared towards a small bundle size and fast performance and secondarily towards a good developer experience. It is almost completely modular, which reduces the bundle size to a minimum and provides high flexibility for external extensions and customizations.

4.2.1 Composition

In order to keep the bundle size as small as possible through tree shaking and code splitting, care was taken to ensure that each functionality is imported individually and only as required. This has the consequence that a schema is composed of different imports. The dependency injection pattern was chosen to avoid that functions being nested on several levels to extend the validation of a schema by additional functionality.

```
// With function nesting
const EmailSchema = string(minLength(1, email()));

// With dependency injection
const EmailSchema = string([minLength(1), email()]);
```

Listing 32: Example of a deeply nested API vs. dependency injection

This approach has several advantages. On the one hand, a formatter like Prettier can format the code more clearly if a single line becomes too long since the functions are then placed under each other with the same indentation due to the structure of the API. On the other hand, it also reduces the code required for initialization, which reduces the bundle size and improves performance since the information is already available in an array and does not have to be collected first, like with Zod or ArkType.

```
// With function nesting
const EmailSchema = string(
  minLength(1, email(endsWith("@example.com")))
);

// With dependency injection
const EmailSchema = string([
  minLength(1),
  email(),
  endsWith("@example.com"),
]);
```

Listing 33: Difference in formatting for different API designs

The use of functions, compared to strings as in ArkType or TypeScript types as in Typia, also provides a high level of flexibility, as additional information can be added to a schema in a type-safe manner through the arguments of a function. For example, passing a string allows the error messages to be individualized.

```
const EmailSchema = string("Your email must be a string.", [
  minLength(1, "Please enter your email."),
  email("The email address is badly formatted."),
]);
```

Listing 34: Example of an email schema with individual error messages

In order to represent complex schemas like an object, they can be nested within each other. This implies that the individual schemas are independent and can be reused as often as required in other complex schemas.

```
const EmailSchema = string([minLength(1), email()]);

const LoginSchema = object({
  email: EmailSchema,
  password: string([minLength(1), minLength(8)]),
});
```

Listing 35: Example of reusability of a schema in other schemas

4.2.2 Naming

The schema functions are named after the data type they represent and validate. This has the advantage that users do not have to learn new names. It also reduces the bundle size since the name of the data type occurs multiple times within a schema's source code, thus improving the compression rate. For methods, the naming is based on TypeScript, resulting in a high similarity, which reduces the learning curve. Apart from the names oriented to data types or TypeScript, similar names were chosen as in Zod.

4.2.3 Compromises

The modular design of the API, apart from the small bundle size, has the advantage that the functions can be tested, maintained, extended, and reused more easily. However, this approach also brings disadvantages. For example, the tooling is worse than with Zod and ArkType by default, because the editor can not make prescient suggestions due to the modularity. Therefore, a custom Visual Studio Code extension is required to overcome this disadvantage.

Also, the modular design can lead to naming conflicts with reserved keywords. For example, the function that validates the data type `null` cannot be named `null` because JavaScript does not allow reserved keywords as function names. This makes it necessary to choose a different name or add a prefix or suffix. Although the function could be renamed back to `null` with an alias when exporting, this would require a wildcard when importing. For a consistent approach, the compromise was made to add the suffix "Type" to schema functions with name conflicts.

Another compromise in API design are the pipelines. Since they are optional, the code necessary to execute a pipeline is not required for every schema. This code would have to be outsourced to a separate function for complete modularity. However, since this would make the API design more cumbersome, and the pipeline feature will likely be used, the decision was made to include this code into most schema functions by default.

```
// When the pipeline is included
const EmailSchema = string([email(), maxLength(50)]);

// When the pipeline is supplemented
const EmailSchema = string(pipe([email(), maxLength(50)]));
```

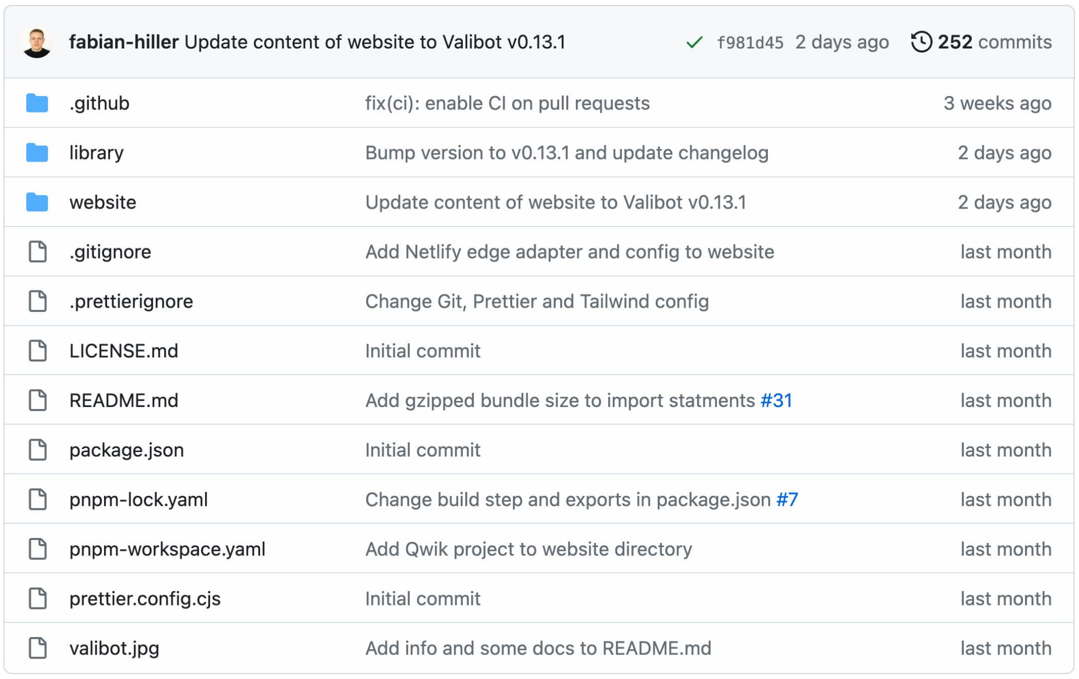
Listing 36: Impact on API design if the pipeline code needs to be supplemented

4.3 Implementation

The implementation of Valibot is composed of one class, 147 functions, and 140 type definitions. It differs significantly from Zod, ArkType, and Typia due to its modular structure. Without blank lines and comments, the source code of Valibot consists of about 3,500 lines of TypeScript code. [80]

4.3.1 Repository

Valibot uses a monorepo because apart from the library's source code, the official website with the documentation is also part of the repository. This makes it possible to keep the documentation in sync when making changes to the source code. In the repository's root directory are global configuration and information files, such as `.gitignore` and `LICENSE.md`, as well as the two directories `/library` and `/website`.

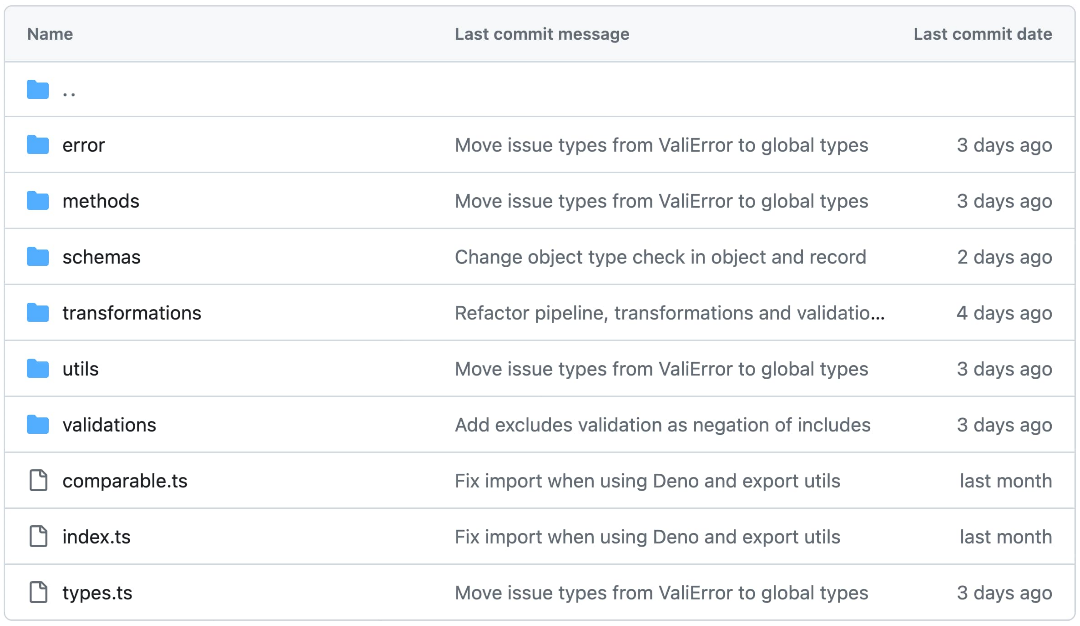


fabian-hiller Update content of website to Valibot v0.13.1 ✓ f981d45 2 days ago 252 commits

.github	fix(ci): enable CI on pull requests	3 weeks ago
library	Bump version to v0.13.1 and update changelog	2 days ago
website	Update content of website to Valibot v0.13.1	2 days ago
.gitignore	Add Netlify edge adapter and config to website	last month
.prettierignore	Change Git, Prettier and Tailwind config	last month
LICENSE.md	Initial commit	last month
README.md	Add gzipped bundle size to import statments #31	last month
package.json	Initial commit	last month
pnpm-lock.yaml	Change build step and exports in package.json #7	last month
pnpm-workspace.yaml	Add Qwik project to website directory	last month
prettier.config.cjs	Initial commit	last month
valibot.jpg	Add info and some docs to README.md	last month

Figure 4.1: Screenshot of the root directory of the GitHub repository [80]

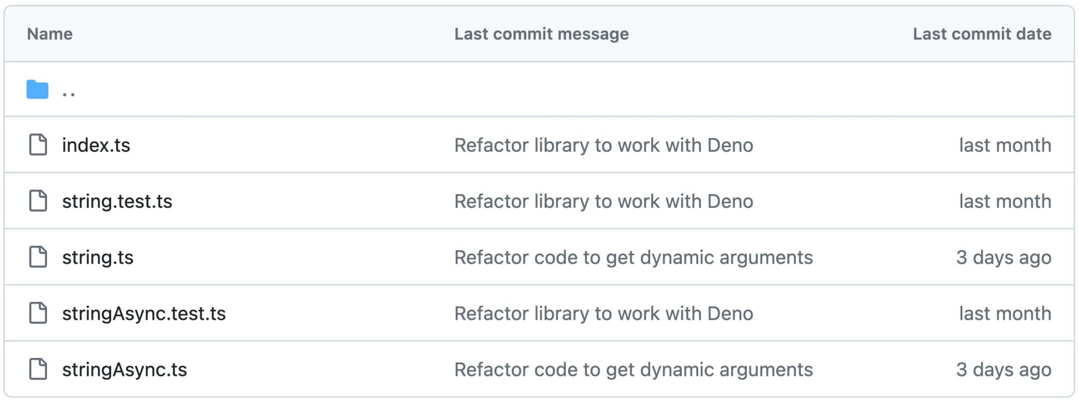
The `/library` directory contains local configuration and information files of the library as well as the `/src` directory where the source code is located. The source code is divided into six folders, apart from the `types.ts` file containing global type definitions. Under `/error` is the implementation of the `ValiError` class and a function to format error information. `/methods` contains the modification and validation functions, and `/schemas` all schema functions. The folders `/transformations` and `/validations` contain the transformation and validation functions that can be used in the pipeline of a schema. `/utils` is the last missing folder, where small utility functions are outsourced.



Name	Last commit message	Last commit date
..		
error	Move issue types from ValiError to global types	3 days ago
methods	Move issue types from ValiError to global types	3 days ago
schemas	Change object type check in object and record	2 days ago
transformations	Refactor pipeline, transformations and validatio...	4 days ago
utils	Move issue types from ValiError to global types	3 days ago
validations	Add excludes validation as negation of includes	3 days ago
comparable.ts	Fix import when using Deno and export utils	last month
index.ts	Fix import when using Deno and export utils	last month
types.ts	Move issue types from ValiError to global types	3 days ago

Figure 4.2: Screenshot of the `/src` directory in the GitHub repository [83]

Each class or function within the six folders is in its own directory. Besides the implementation, this contains a file for unit tests and, if necessary, an asynchronous implementation for asynchronous validation. Each folder contains an `index.ts` file, which determines what content is exported to the outside.



Name	Last commit message	Last commit date
..		
index.ts	Refactor library to work with Deno	last month
string.test.ts	Refactor library to work with Deno	last month
string.ts	Refactor code to get dynamic arguments	3 days ago
stringAsync.test.ts	Refactor library to work with Deno	last month
stringAsync.ts	Refactor code to get dynamic arguments	3 days ago

Figure 4.3: Screenshot of the `/string` directory in the GitHub repository [84]

4.3.2 Initialization

The schema functions that are called during initialization return an object with information about the schema and a validation method. To ensure that the schemas are compatible with each other, the basic structure of this object must be unified for each schema. For this reason, the `types.ts` file in the `/src` directory contains, with `BaseSchema`, the type definition to which each initialized schema must conform. This includes the `.async` and `._types` properties and the `._parse` method. `._types` and `._parse` start with an underscore to clarify that they are internal APIs that should not be accessed from outside the library.

```
export type BaseSchema<TInput = any, TOutput = TInput> = {
  async: false;
  _parse(input: unknown, info?: ParseInfo): _ParseResult<TOutput>;
  _types?: { input: TInput; output: TOutput };
};
```

Listing 37: Implementation of the `BaseSchema` type definition

`.async` is a boolean indicating whether the schema is validated asynchronously and `._types` is an object that stores the input and output type of a schema. `._types` is marked as optional in the type definition by a question mark since this property is only used within TypeScript types to derive the type definition of a schema and is not actually added to the object.

```
export function nullType(error?: string): NullSchema {
  return {
    schema: "null",
    async: false,
    _parse(input, info) {
      // Check type of input
      if (input !== null) {
        return getIssues(
          info,
          "type",
          "null",
          error || "Invalid type",
          input
        );
      }

      // Return input as output
      return { output: input };
    },
  };
}
```

Listing 38: Implementation of the `nullType` schema function

Function overloads make the optional specification of an individual error message and pipeline type-safe. This TypeScript feature allows different overload signatures of a function to be defined together with a single implementation. The implementation must be able to handle each overload signature. For this reason, many schema functions use the utility function `getDefaultArgs` to extract the error message and pipeline from the dynamic arguments of the function.

```
export function string(pipe?: Pipe<string>): StringSchema;
export function string(error?: string, pipe?: Pipe<string>): ...
export function string(
  arg1?: string | Pipe<string>,
  arg2?: Pipe<string>
): StringSchema {
  // Get error and pipe argument
  const [error, pipe] = getDefaultArgs(arg1, arg2);

  // Create and return string schema
  return {
    schema: 'string',
    async: false,
    _parse(input, info) { ... },
  };
}
```

Listing 39: Implementation of the `string` schema function

The validation and transformation functions called within the pipeline of a schema at initialization return a closure function called for validation and transformation within the `._parse` method. This process is more efficient compared to Zod and ArkType as the schema functions return only an object at initialization, and the validation and transformation functions are already in an array, so no additional code needs to be executed for initialization.

```
export function minLength<TInput extends string | any[]>(
  requirement: number,
  error?: string
) {
  return (input: TInput): PipeResult<TInput> => {
    if (input.length < requirement) {
      return {
        issue: {
          validation: 'min_length',
          message: error || 'Invalid length',
          input,
        },
      };
    }
    return { output: input };
  };
}
```

Listing 40: Implementation of the `minLength` validation function

4.3.3 Validation

When data is validated with `parse`, `safeParse`, or `is` using a schema, the `._parse` method of the schema is executed with the input as the first argument. This validates the data type of the input, usually with the `typeof` or `instanceof` operator, and then, if necessary, executes the validation and transformation functions of the pipeline in a loop. If it is a complex data type, the entries are validated in between using the `._parse` method of the nested schemas. In the end, the functions of the pipeline and the `._parse` method return a discriminated union. In case of an error under `.issues` or `.issue`, the problems found are located or otherwise under `.output`, the output of the validation or transformation.

```
export function parse<TSchema extends BaseSchema>(...): ... {
  const result = schema._parse(input, info);
  if (result.issues) {
    throw new ValiError(result.issues);
  }
  return result.output;
}
```

Listing 41: Implementation of the `parse` method function

This unified approach allows modification functions such as `coerce` and `transform` to change the input and output of a schema by overriding the `._parse` method.

```
export function coerce<TSchema extends BaseSchema>(
  schema: TSchema,
  action: (value: unknown) => Input<TSchema>
): TSchema {
  return {
    ...schema,
    _parse(input, info) {
      return schema._parse(action(input), info);
    },
  };
}
```

Listing 42: Implementation of the `coerce` method function

To abort the validation process early in case of a problem, the entire validation can be aborted early with `abortEarly`, or the validation of a pipeline can be aborted early with `abortPipeEarly`. This can improve performance if only the error information of the first issue is needed.

```
const output = parse(EmailSchema, "jane@exmaple.com", {
  abortPipeEarly: true
});
```

Listing 43: Example of the `abortPipeEarly` property when calling `parse`

4.3.4 Compromises

Compromises had to be made not only in the library's API design but also in its implementation. Most often, this involved a tradeoff between bundle size and performance. For example, copying a known object into another with the spread operator reduces the bundle size. However, the performance investigations of the last chapter showed that the spread operator creates a hotspot, which harms performance during validation. For this reason, it was decided to prioritize performance over bundle size and explicitly assign the object entries. Another example are several for-loops in the source code, which were preferred to a shorthand notation because of performance improvements.

4.4 Comparison

Even though the API design of Valibot resembles Zod at first glance, the implementation and structure of the source code is quite different. Also, it differs in various aspects from ArkType and Typia. These differences also affect the bundle size, performance, and tooling.

4.4.1 Bundle Size

Due to the modular design of Valibot, the minified and compressed bundle size starts at less than 300 bytes, depending on the scheme function, and only gets bigger when additional functionality is needed. For a direct comparison to Zod, ArkType, and Typia, the identical product schema was also implemented with Valibot.

Library	Minified	Gzipped	Compression
Valibot	3.71 kB	1.31 kB	65 %
Zod	53.7 kB	12.9 kB	76 %
ArkType	47.7 kB	16.2 kB	66 %
Typia	35.8 kB	9.72 kB	73 %

Table 4.1: Bundle size of the same product scheme compared to Valibot

The comparison shows an extreme gap to Zod, ArkType and Typia. Valibot's bundle size in this comparison is 90 % smaller than Zod's, 92 % smaller than ArkType's, and 87 % smaller than Typia's. Apart from a modular API design like Valibot's, a comparable bundle size can only be achieved with a compile step. However, common bundlers cannot remove unused code since Typia is currently not optimized for three shaking and code splitting.

Valibot is also smaller in total library bundle size. Compared to Zod and ArkType, Valibot's source code is simpler and more efficient and can be compressed more. For Typia, it has to do with the fact that the library contains other functionality besides validation, and the bundle size includes dependencies. In general, it must be considered that the functionality is different between the libraries. Not every schema implemented with Zod can currently be implemented with Valibot, ArkType, and Typia.

Library	Minified	Gzipped	Compression
Valibot	30 kB	5.95 kB	80 %
Zod	54.6 kB	13.3 kB	76 %
ArkType	47.4 kB	16.1 kB	66 %
Typia	28.2 kB	8.61 kB	69 %

Table 4.2: Bundle size of Zod, ArkType, and Typia compared to Valibot

4.4.2 Performance

Due to Valibot's small bundle size and various initialization optimizations, the library also outperforms Zod, ArkType, and Typia in the startup performance of a website. When downloading at 15 Mbit/s and initializing 10 product schemas, Valibot takes about 82 % less time than Zod, 87 % less than ArkType, and 40 % less than Typia in this benchmark.

Library	**1.5 Mbit/s**	**15 Mbit/s**	**150 Mbit/s**
Valibot	7 ms	1 ms	0 ms
Zod	69 ms	7 ms	1 ms
ArkType	86 ms	9 ms	1 ms
Typia	52 ms	5 ms	1 ms

Table 4.3: Download speed of the product schema compared to Valibot

Library	**10**	**100**	**1,000**	**10,000**	**100,000**
Valibot	2 ms	5 ms	19 ms	77 ms	449 ms
Zod	8 ms	35 ms	161 ms	1 s 175 ms	11 s 201 ms
ArkType	14 ms	56 ms	315 ms	2 s 345 ms	21 s 862 ms
Typia	-	-	-	-	-

Table 4.4: Initialization of the same product schema compared to Valibot

Valibot performs also better than Zod and ArkType in terms of runtime performance but remains behind Typia's speed despite various optimizations. Due to the optimized code of the compile step, Typia has to execute less code than Valibot and can perform the validation more efficiently.

Library	**10**	**100**	**1,000**	**10,000**	**100,000**
Valibot	3 ms	10 ms	41 ms	244 ms	2 s 235 ms
Zod	7 ms	22 ms	75 ms	443 ms	3 s 866 ms
ArkType	3 ms	12 ms	61 ms	449 ms	4 s 139 ms
Typia	1 ms	1 ms	5 ms	37 ms	140 ms

Table 4.5: Successful validation of the same product schema compared to Valibot

Library	**10**	**100**	**1,000**	**10,000**	**100,000**
Valibot	5 ms	18 ms	101 ms	787 ms	7 s 543 ms
Zod	10 ms	39 ms	207 ms	1 s 581 ms	15 s 134 ms
ArkType	5 ms	25 ms	149 ms	1 s 191 ms	11 s 307 ms
Typia	2 ms	5 ms	21 ms	126 ms	769 ms

Table 4.6: Erroneous validation of the same product schema compared to Valibot

4.4.3 Tooling

Since Valibot has many exports due to its modular design and IntelliSense cannot provide precise suggestions, the tooling is worse than with Zod and ArkType. This may require reading the documentation first to understand the structure and naming of the API. To achieve a similar developer experience, the tooling needs to be improved with an extension that pre-filters the suggestions based on the context so that, for example, only the validation and transformation functions compatible with the respective schema are suggested in its pipeline.

Compared to Typia, the tooling is different, but the developer experience is similar. Because Typia builds on TypeScript types, it is easier to get started. However, as soon as a schema becomes more complex and comment tags are required, the advantages of Valibot's type safety may outweigh the disadvantages since typos in the comment tags are not highlighted by Visual Studio Code.

5 Conclusion

In the introduction, the question was raised whether Zod's functionality and developer experience can be achieved with a smaller bundle size. The goal was to investigate how a modular structure affects the bundle size and how this approach affects performance and tooling.

Valibot, the schema library created as part of this thesis, has shown that a modular design is technically possible and that this approach greatly reduces the bundle size and thus can improve the startup performance of a website. However, the research also showed that the impact on runtime performance is small, as performance bottlenecks or a compile step that generates optimized code have a more significant effect. Tooling, and therefore the developer experience, deteriorates due to modularity, as Visual Studio Code cannot display precise suggestions without an extension, unlike with Zod.

With more than 3,000 stars on GitHub and 6,700 weekly downloads on npm five weeks after the announcement, Valibot's numbers indicate that the project is addressing an existing problem and that there is a need for innovation in this space [80, 81].

5.1 Evaluation

Valibot has shown that a modular API design and implementation can significantly reduce the bundle size of a library. A bundler can thus remove unused code through tree shaking and code splitting. For a library like Zod and ArkType, where a smaller portion of the functionality is needed in many use cases, this process can reduce the bundle size by more than 90 %. However, it also became clear during the research that the same problem can be solved in many different ways and that each approach has advantages and disadvantages.

5.1.1 Advantages

The most obvious advantage is the bundle size, which starts at less than 300 bytes and only increases when additional functionality is needed. This has a particularly positive impact on the TTI benchmark of a website, making Valibot especially suitable when schemas are used in the front end. In general, it can be concluded that a modular design can be an advantage for JavaScript libraries and SDKs used in the front end, if not the entire functionality is needed in every use case.

A practical use case for a schema library are forms. Coupled with a full-stack framework and a form library, Valibot provides end-to-end type safety and validation between the front and back end with a small footprint. An example is the implementation of Modular Forms for Qwik [32, 18, 19].

Another advantage, apart from the bundle size, is that the source code of a modular library can be individualized and extended more easily. If a specific functionality is missing or a function behaves differently than expected, it can be extended or replaced with custom code. This approach allows Valibot to add more schema, validation, and transformation functions without increasing the bundle size for all users.

In addition, the fact that the functionality of a single function is clearly defined reduces the complexity of the individual components, making them more accessible to external developers. This simplifies collaboration, especially in an open-source project, since contributors usually only need to look at a few functions rather than going through the entire library. It also simplifies testing, as it is easier to test the functionality of many small functions with a straightforward feature set than a few large and complex ones.

5.1.2 Disadvantages

One of the disadvantages is that the tooling is somewhat inferior by default. Thus, more effort is required when using the library for the first time to understand the modular structure of the API and the naming conventions. However, an extension for Visual Studio Code and extensive documentation can partially overcome this disadvantage. Also, the suggestions from AI tools like GitHub Copilot are becoming more and more precise, so a modular API may not be a disadvantage in the long run.

It is important to note that the main benefit of a modular library, the bundle size, almost exclusively affects the front end. On the back end, a small bundle size and fast initialization can help when scaling a serverless system, such as AWS Lambda and Google Cloud Functions, to reduce the cold start time of new instances, but otherwise plays a minor role.

5.2 Prospect

Within a few weeks, a community and an ecosystem formed around Valibot. There are already several blog posts, podcast episodes, and YouTube videos that introduce and explain the schema library. With the support of the community, the ecosystem is also growing from week to week. Currently, Valibot is in version 0.13.1, with plans to fix existing issues and achieve feature parity with Zod by the time v1 is released. It is also planned to expand the API reference of the official documentation step by step.

The research and results of this thesis, as well as the proven interest in Valibot, can provide a foundation to disrupt libraries in various areas through a modular design. It can create awareness of the impact of large bundle sizes and encourage rethinking. Since modularity requires a completely different approach for API design and implementation, this thesis also offers, beyond the theory, a practical starting point that can be used for inspiration with the source code of Valibot.

Bibliography

[1] Colin McDonnell. *Zod Website*. URL: https://zod.dev/ (visited on 08/04/2023).

[2] npm-stat.com. *Statistics about Zod downloads for the last 3 years*. URL: https://npm-stat.com/charts.html?package=zod&from=2020-07-30&to=2023-07-30 (visited on 08/04/2023).

[3] npm-stat.com. *Statistics about Zod downloads for the last 12 months*. URL: https://npm-stat.com/charts.html?package=zod&from=2022-07-30&to=2023-07-30 (visited on 08/04/2023).

[4] npm-stat.com. *Statistics about TypeScript downloads for the last 3 years*. URL: https://npm-stat.com/charts.html?package=typescript&from=2020-07-30&to=2023-07-30 (visited on 08/05/2023).

[5] Microsoft. *TypeScript Website*. URL: https://www.typescriptlang.org/ (visited on 08/04/2023).

[6] Colin McDonnell. *Zod Ecosystem*. URL: https://zod.dev/?id=ecosystem (visited on 08/04/2023).

[7] bundlejs.com. *Minimum bundle size of Zod v3.21.4 for a single export*. URL: https://bundlejs.com/?q=zod%403.21.4&treeshake=%5B%7B+boolean+%7D%5D (visited on 08/04/2023).

[8] React Core Team. *Introducing Zero-Bundle-Size React Server Components*. 2020. URL: https://react.dev/blog/2020/12/21/data-fetching-with-react-server-components (visited on 08/04/2023).

[9] The Astro Technology Company. *Astro Islands*. URL: https://docs.astro.build/en/concepts/islands/ (visited on 08/04/2023).

[10] Inc. Builder.io. *Resumable vs. Hydration*. URL: https://qwik.builder.io/docs/concepts/resumable/ (visited on 08/04/2023).

[11] Ryan Carniato. *Why Efficient Hydration in JavaScript Frameworks is so Challenging*. 2022. URL: https://dev.to/this-is-learning/why-efficient-hydration-in-javascript-frameworks-is-so-challenging-1ca3 (visited on 08/04/2023).

[12] Miško Hevery. *From Static to Interactive: Why Resumability is the Best Alternative to Hydration*. 2022. URL: https://www.builder.io/blog/from-static-to-interactive-why-resumability-is-the-best-alternative-to-hydration (visited on 08/04/2023).

[13] Tom Greenwood. *Greening the Web: How We Can Create Zero Carbon Websites*. 2022. URL: https://kinsta.com/blog/zero-carbon-websites/ (visited on 08/04/2023).

[14] HTTP Archive. *JavaScript Bytes*. URL: https://httparchive.org/reports/state-of-javascript#bytesJs (visited on 08/04/2023).

[15] npm-stat.com. *Statistics about Next.js downloads for the last 3 years*. URL: https://npm-stat.com/charts.html?package=next&from=2020-07-30&to=2023-07-30 (visited on 08/04/2023).

[16] npm-stat.com. *Statistics about Nuxt downloads for the last 3 years*. URL: https://npm-stat.com/charts.html?package=nuxt&from=2020-07-30&to=2023-07-30 (visited on 08/05/2023).

[17] Vercel. *Server Actions*. URL: https://nextjs.org/docs/app/building-your-application/data-fetching/server-actions (visited on 08/04/2023).

[18] Fabian Hiller. *Validate your fields*. URL: https://modularforms.dev/qwik/guides/validate-your-fields (visited on 08/08/2023).

[19] Fabian Hiller. *Handle submission*. URL: https://modularforms.dev/qwik/guides/handle-submission (visited on 08/04/2023).

[20] Microsoft. *TypeScript for JavaScript Programmers*. URL: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html (visited on 08/08/2023).

[21] MDN contributors. *JavaScript data types and data structures*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures (visited on 08/08/2023).

[22] Microsoft. *Why create TypeScript*. URL: https://www.typescriptlang.org/why-create-typescript (visited on 08/08/2023).

[23] Microsoft. *The Basics*. URL: https://www.typescriptlang.org/docs/handbook/2/basic-types.html (visited on 08/08/2023).

[24] CheatSheets Series Team. *Input Validation Cheat Sheet*. URL: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html (visited on 08/08/2023).

[25] MDN contributors. *Client-side form validation*. URL: https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation (visited on 08/08/2023).

[26] Microsoft. *Narrowing*. URL: https://www.typescriptlang.org/docs/handbook/2/narrowing.html (visited on 08/08/2023).

[27] Colin McDonnell. *Zod*. URL: https://github.com/colinhacks/zod (visited on 08/10/2023).

[28] Colin McDonnell. *Zod Type Inference*. URL: https://zod.dev/?id=type-inference (visited on 08/08/2023).

[29] Colin McDonnell. *Zod Objects*. URL: https://zod.dev/?id=objects (visited on 08/29/2023).

[30] Colin McDonnell. *Zod Arrays*. URL: https://zod.dev/?id=arrays (visited on 08/29/2023).

[31] Express.js. *body-parser*. URL: https://github.com/expressjs/body-parser (visited on 08/08/2023).

[32] Fabian Hiller. *Define your form*. URL: https://modularforms.dev/qwik/guides/define-your-form (visited on 08/08/2023).

[33] T3 contributors. *Environment Variables*. URL: https://create.t3.gg/en/usage/env-variables (visited on 08/08/2023).

[34] Nolan Lawson. *JavaScript performance beyond bundle size*. 2021. URL: https://nolanlawson.com/2021/02/23/javascript-performance-beyond-bundle-size/ (visited on 08/08/2023).

[35] Ben Schwarz. *Small Bundles, Fast Pages: What To Do With Too Much JavaScript*. 2021. URL: https://calibreapp.com/blog/bundle-size-optimization (visited on 08/08/2023).

[36] Nolan Lawson. *Why does speed matter?* 2020. URL: https://web.dev/why-speed-matters/ (visited on 08/08/2023).

[37] MDN contributors. *Tree shaking*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking (visited on 08/08/2023).

[38] MDN contributors. *Code splitting*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Code_splitting (visited on 08/08/2023).

[39] MDN contributors. *Minification*. URL: https://developer.mozilla.org/en-US/docs/Glossary/minification (visited on 08/08/2023).

[40] MDN contributors. *Compression in HTTP*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Compression (visited on 08/08/2023).

[41] Jeremy Wagner. *Reduce JavaScript payloads with tree shaking*. 2018. URL: https://web.dev/reduce-javascript-payloads-with-tree-shaking/ (visited on 08/08/2023).

[42] Webpack. *Code Splitting*. URL: https://webpack.js.org/guides/code-splitting/ (visited on 08/08/2023).

[43] Ryan Carniato. *Why Efficient Hydration in JavaScript Frameworks is so Challenging*. 2022. URL: https://dev.to/this-is-learning/why-efficient-hydration-in-javascript-frameworks-is-so-challenging-1ca3 (visited on 08/08/2023).

[44] Ryan Carniato. *Resumable JavaScript with Qwik*. 2022. URL: https://dev.to/this-is-learning/resumable-javascript-with-qwik-2i29 (visited on 08/08/2023).

[45] Google. *Brotli*. URL: https://github.com/google/brotli (visited on 08/08/2023).

[46] MDN contributors. *Web performance*. URL: https://developer.mozilla.org/en-US/docs/Web/Performance (visited on 08/08/2023).

[47] MDN contributors. *JavaScript performance optimization*. URL: https://developer.mozilla.org/en-US/docs/Learn/Performance/JavaScript (visited on 08/08/2023).

[48] Philip Walton. *User-centric performance metrics*. 2022. URL: https://web.dev/user-centric-performance-metrics/ (visited on 08/08/2023).

[49] Philip Walton. *Time to Interactive (TTI)*. 2022. URL: https://web.dev/tti/ (visited on 08/08/2023).

[50] Kayce Basques. *Analyze runtime performance*. 2017. URL: `https://developer.chrome.com/docs/devtools/performance/` (visited on 08/08/2023).

[51] Esther Christopher. *How Does JavaScript Work Behind the Scenes? JS Engine and Runtime Explained*. 2023. URL: `https://www.freecodecamp.org/news/how-javascript-works-behind-the-scenes/` (visited on 08/08/2023).

[52] Kenneth Leroy Busbee and Dave Braunschweig. *Modular Programming*. 2018. URL: `https://press.rebus.community/programmingfundamentals/chapter/modular-programming/` (visited on 08/08/2023).

[53] Millie Macdonald. *Modular programming: beyond the spaghetti mess*. 2023. URL: `https://www.tiny.cloud/blog/modular-programming-principle/` (visited on 08/08/2023).

[54] Dan Onoshko. *How to build tree-shakeable JavaScript libraries*. 2022. URL: `https://dev.to/cubejs/how-to-build-tree-shakeable-javascript-libraries-35fa` (visited on 08/08/2023).

[55] Microsoft. *IntelliSense*. URL: `https://code.visualstudio.com/docs/editor/intellisense` (visited on 08/08/2023).

[56] Microsoft. *Extension API*. URL: `https://code.visualstudio.com/api` (visited on 08/08/2023).

[57] Colin McDonnell. *zod/src/types.ts*. URL: `https://github.com/colinhacks/zod/blob/v3.21.4/src/types.ts` (visited on 08/10/2023).

[58] Microsoft. *Assertion Functions*. URL: `https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html#assertion-functions` (visited on 08/10/2023).

[59] Microsoft. *Discriminated Unions*. URL: `https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes-func.html#discriminated-unions` (visited on 08/10/2023).

[60] David Blass. *ArkType Website*. URL: `https://arktype.io/` (visited on 08/10/2023).

[61] David Blass. *ArkType*. URL: `https://github.com/arktypeio/arktype/tree/arktype%401.0.19-alpha` (visited on 08/10/2023).

[62] David Blass. *arktype/src/scopes/scope.ts*. URL: https://github.com/arktypeio/arktype/blob/arktype%401.0.19-alpha/src/scopes/scope.ts (visited on 08/10/2023).

[63] Jeongho Nam. *Pure TypeScript*. URL: https://typia.io/docs/pure/ (visited on 08/10/2023).

[64] Jeongho Nam. *Typia*. URL: https://github.com/samchon/typia (visited on 08/10/2023).

[65] Jeongho Nam. *validate() function*. URL: https://typia.io/docs/validators/validate/ (visited on 08/10/2023).

[66] Jeongho Nam. *Comment Tags*. URL: https://typia.io/docs/validators/comment-tags/ (visited on 08/10/2023).

[67] Jeongho Nam. *typia/src/module.ts*. URL: https://github.com/samchon/typia/blob/v4.2.1/src/module.ts (visited on 08/10/2023).

[68] Kitson Kelly. *Typescript Custom Transformers Support*. 2021. URL: https://github.com/denoland/deno/issues/3354#issuecomment-755712488 (visited on 08/11/2023).

[69] Michael Wanyoike and Sam Deering. *25+ JavaScript Shorthand Coding Techniques*. 2019. URL: https://www.sitepoint.com/shorthand-javascript-techniques/ (visited on 08/11/2023).

[70] MDN contributors. *Conditional (ternary) operator*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_operator (visited on 08/29/2023).

[71] David Blass. *arktype/src/scopes/scope.ts*. URL: https://github.com/arktypeio/arktype/blob/arktype%401.0.19-alpha/src/traverse/traverse.ts (visited on 08/10/2023).

[72] JP Camara. *Making Tanstack Table 1000x faster with a 1 line change*. 2023. URL: https://jpcamara.com/2023/03/07/making-tanstack-table.html (visited on 08/29/2023).

[73] Naru. *Reduce spread operators to improve runtime performance*. 2023. URL: https://github.com/fabian-hiller/valibot/pull/46 (visited on 08/29/2023).

[74] Matti Lankinen. *Performance improvements*. 2021. URL: https://github.com/colinhacks/zod/pull/492 (visited on 08/29/2023).

[75] Colin McDonnell. *Comment of Colin McDonnell*. 2021. URL: https://github.com/colinhacks/zod/pull/492#issuecomment-860339774 (visited on 08/29/2023).

[76] Fabian Hiller. *thesis-benchmarks*. URL: https://github.com/fabian-hiller/thesis-benchmarks (visited on 08/29/2023).

[77] Fabian Hiller. *Why does Typia initial have a bundle size of 7.6 kB?* 2023. URL: https://github.com/samchon/typia/issues/752 (visited on 08/29/2023).

[78] David Blass. *arktype/src/type.ts*. URL: https://github.com/arktypeio/arktype/blob/7933305e1470a2d5952124920a5dbbf28a730a2e/src/type.ts (visited on 08/29/2023).

[79] David Blass. *arktype/dev/utils/src/functions*. URL: https://github.com/arktypeio/arktype/blob/7933305e1470a2d5952124920a5dbbf28a730a2e/dev/utils/src/functions.ts (visited on 08/29/2023).

[80] Fabian Hiller. *valibot*. URL: https://github.com/fabian-hiller/valibot/tree/v0.13.1 (visited on 08/29/2023).

[81] npm-stat.com. *Statistics about Valibot downloads for the last weeks*. URL: https://npm-stat.com/charts.html?package=valibot&from=2023-07-12&to=2023-08-28 (visited on 08/29/2023).

[82] Fabian Hiller. *Ecosystem*. URL: https://valibot.dev/guides/ecosystem/ (visited on 08/29/2023).

[83] Fabian Hiller. *valibot/library/src*. URL: https://github.com/fabian-hiller/valibot/tree/v0.13.1/library/src (visited on 08/29/2023).

[84] Fabian Hiller. *valibot/library/src/schemas/string*. URL: https://github.com/fabian-hiller/valibot/tree/v0.13.1/library/src/schemas/string (visited on 08/29/2023).