Initial File System

The *Initial File System* feature for Golem introduces a significant enhancement to worker functionality, allowing pre-populated file systems upon worker instantiation.

This document outlines the feature's purpose, use cases, implementation details, and necessary changes across various Golem components. It covers the rationale behind the feature, permissions handling, live update considerations, and technical specifications for updating the Golem Manifest, REST API, CLI, Worker Gateway, and Worker Executor.

The document provides a comprehensive guide for developers and system architects to understand and implement this feature, including code snippets, API examples, and gRPC service definitions.

By enabling static content, configuration data, and reference datasets to be readily available to workers, this feature aims to align Golem's functionality more closely with developer expectations and container-like environments.

Use Cases

Developers may use the *Initial File System* feature for static content (such as HTML, CSS, and images), configuration data (YAML/TOML/HOCON), reference data sets, or even databases (SQLite) containing data that the application requires in order to function.

- HTML, CSS, Javascript
- Images, videos
- TOML, YAML, JSON, properties, and other configuration files
- Databases (SQLite, etc.)
- Standard libraries, for interpreted languages like Python, JS, etc.

Implementation Notes

Permissions

Each file in the initial file set may have a permission, which affects the ability for the file to be changed:

- **Read-only**. A read-only permission implies that the file may be read, but not updated. This permission is appropriate for static content that is not intended to change. Use of the read-only permission will result in more efficient storage in Golem, because the content does not have to be duplicated across every worker. This is ideal for videos, images, and other data files that are large and not intended to dynamically change from the business logic of the worker.
- **Read-write**. A read-write permission implies that the file may be both read and written. Such files will be duplicated on a worker-by-worker basis, and cannot be shared globally across all workers, therefore they incur additional storage and memory costs.

Live Update

An update of a component must allow update of the file set. This could happen if newer versions of a component require new configuration, or if there have been updates to static data, such as HTML, CSS, images, or databases.

If a worker is custom or auto-updated to the newer component version (”live update”), then after the update of logic and state has been performed, and the worker’s state has been fully updated, but before the worker resumes execution, it is theoretically possible to update the file set.

Files that are “read-only” can be safely updated at such a safe point. However, files that are “read-write” may never be updated, and must be skipped over, even if the newer version of the component has changed the contents of those files.

> 💡 It could be useful to issue a warning if live-update is attempted on a worker and the contents of read-write files have changed in the update.

Implementation Guide

In order to implement the *Initial File System* feature, a number of changes must be propagated across the various components that comprise Golem:

- **Golem Manifest**. The `golem.yaml` manifest file, which is used to declaratively describe build and deployment options for Golem applications, must be updated to support specifying the initial file set.
- **REST API.** The Golem REST API must be updated so that component and creation allow for specification of the initial file set.
- **Golem CLI**. The Golem CLI must be updated so that it properly utilizes the Golem Manifest and the REST API in order to properly upload the initial file set upon component creation or update.
- **Worker Gateway**. The Worker Gateway must be updated so that it can serve files from a worker.
- **Worker Executor**. The Worker Executor must be updated so that it correctly populates the file system for a new worker, and exposes a new API for retrieving worker files that can be used by the Worker Gateway for serving content.

The following sections detail these changes in more depth, and contain links to repositories.

Golem Manifest

The Golem manifest file is currently a YAML file that is prototyped in the `wasm-rpc` project, but which needs to be supported and integrated into `golem-cli`. The intention behind the manifest file is to provide a single source of truth about what a component depends on, what its configuration starts out being, what its initial files are, and so on.

This YAML file, as prototyped in `wasm-rpc`, currently only serves the needs of RPC, providing a way to declaratively describe the RPC dependencies of a component. However, as part of the *Initial File System* feature, the format can be extended to include a declarative specification of the initial file system of the component. This will involve not just extending the specification of the YAML file, but also adding support into Golem CLI for parsing and validating and using the YAML file, since currently, the YAML file is only used in the `wasm-rpc` in-development work.

Note that references to files that are not stored locally are supported, but will be resolved at deployment time, as Golem CLI downloads all such resources and packages them up.

The Golem Manifest file will need to be parsed by Golem CLI, so that when it creates or uploads a component, it can gather together, compress, and upload the files and permissions to the REST API.

```
component:
  name: component-abc
  ...
  properties:
      files:
        - sourcePath: ./a.txt
          targetPath: /a.txt
          permissions: read-only
        - sourcePath: ../../b.dat
          targetPath: /b.dat
          permissions: read-write
        - sourcePath: https://foo.com/bar?baz=quux
          targetPath: /bar.html
          permissions: read-only
```

💡 **Links to Resources**

- PR that introduces the Golem manifest format in wasm-rpc: https://github.com/golemcloud/wasm-rpc/pull/85
- The golem-cli project, which wraps wasm-rpc as a subcommand
  - Crate: https://github.com/golemcloud/golem/tree/main/golem-cli
  - Command wrapping: https://github.com/golemcloud/golem/blob/033576f388fc380bb7ddc72d01a505da30d281bc/golem-cli/src/oss/command.rs#L60-L65
- The files properties should defined and accessed as an extension of the models introduced in the above wasm-rpc PR; for possible extension points see:
  - *model::wasm_rpc::WasmComponentProperties unknown_properties* and
  - *model::oam::Component::typed_properties*
- Open Application Model

REST API

There are three APIs that must be updated:

- **Component API**. The component API must allow specifying the initial file set, together with their permissions, on both component creation and component update.

- **Worker API**. The worker API must support enumerating and retrieving files from a worker.
- **API Definition API**. The API Definition API must support a new binding type that will allow the Worker Gateway to easily serve file content from workers.

Some guidelines and technical details for these updates are provided in the sections that follow.

Component API

Currently, the REST API for component management uses multi-part encoding to accept new components with **name** and **component** fields. For updating existing components, it expects only the binary to be specified as the content for the request.

These APIs will have to be modified in the following ways:

- **Creation**. The component creation API will have to accept an arbitrary number of file uploads, each associated with a particular path in the initial file system. From the perspective of the REST API, the initial file system can be specified as a compressed archive (ZIP), which contains all files, together with their paths. A single new `files` field then becomes optional, whose content is the binary file containing all initial files. This archive is decompressed on the server side and expanded to the initial file system of the component, stored using the existing blob storage mechanism used to store the component binaries themselves. Another field, perhaps called `filesPermissions`, will be required to specify any custom permissions for the file set.

  > 💡 **Simpler Approach?**
  >
  > An alternate way to encode files and permissions might be repeating the multi-part fields `file` and `permission`. Each file could still be compressed, and while this would increase total request size (due to the inability to compress across files), it's a simpler format for purposes of specifying both file and permission.

- **Update**. The existing update API will have to be modified to support multi-part encoding, while still supporting the older API (in which the component binary is the only element being updated) for backward compatibility. In this multi-part encoding, and to match the creation API, the `component` field could specify the binary, the `files` field could specify the ZIP archive, and the `filesPermissions` field could specify file permissions.

> 💡 If permissions are not specified during creation or update, they should default to **read-only**, because read-only files will be stored more efficiently, and due to their immutable nature, could eventually be propagated to CDN.

When completed, these new and newly modified APIs must have rigorous validation and fail in constructive ways, consistent with the existing REST APIs.

The existing blob storage mechanism will be used to store the initial files and their permissions.

Worker API

The Worker API must be updated to expose the file system of a worker. This enhancement to the Worker API will allow tools like Golem CLI to access the worker file system (note that modifying the CLI to expose the worker file system is beyond the scope of this specification).

- `GET /v1/components/{component_id}/workers/{worker_name}/files/`. Retrieves the top-level listing of nodes in the worker's file system. The listing contains information on the type of each node, as well as other metadata.

```
[{"type": "directory", "name": "foo"}, {"type": "file", "name": "bar.txt"}, ...]
```

- `GET /v1/components/{component_id}/workers/{worker_name}/files/<path>`. Retrieves the file at the specified path or lists the contents of the directory at the specified path, depending on whether or not the path points to a file node or a directory node.

API Definition API

The API Definition API allows creating and updating of API Definitions. In addition, it supports importing an OpenAPI specification, augmented with custom Golem properties.

An API Definition consist of one or more *routes*, which define not only the HTTP Method, path segments (including variables), and other OpenAPI-like pieces of information, but also how the Worker Gateway should implement the route based on invoking a function of a specified worker.

In order to implement support for serving content in the Worker Gateway, it is necessary to support a new binding type. This new binding type, tentatively called `file-server`, allows specifying a Rib script, which will return the path and content type of the file that the Worker Gateway will serve.

The new binding type does not require the user to specification a `workerName`. In case the `workerName` is not specified, then the Worker Gateway is free to spawn a new worker for each request to the route (this will be a performance bottleneck but addressing it is not part of this ticket), so that it can access the files of the worker through the Worker Executor.

Once these changes have been implemented in the data models for API definitions, the following APIs can be updated:

- **API Definition Creation**. Creating a new API Definition must support the new binding type.
- **API Definition Update**. Updating an existing API Definition must support the new binding type.
- **API Definition Read**. The new binding type should be visible when retrieved by API.
- **API Definition Import from OpenAPI**. Golem extensions to OpenAPI must support the new binding type.

The following shows what the new format for API Definitions could look like under this change:

```
{
  "id": "twitter-pages",
  "draft": true,
  "version": "0.0.1",
  "routes": [
    {
      "method": "Get",
      "path": "/static/{+file}",
      "binding": {
        "type": "file-server",
        "componentId": {
          "componentId": "1b04d021-9e8e-4f44-832a-e73b32eeb0ba",
          "version": 0
        },
        "workerName": "<optional>",
        "response": "{status: 200, file-path: request.path.file, headers: {ContentType: \'
      }
    }
```

```
  ]
}
```

The following shows what the new format for OpenAPI import could look like under this change:

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "twitter pages",
    "version": "1.0.0"
  },
  "x-golem-api-definition-id": "twitter-pages",
  "x-golem-api-definition-version": "0.0.1",
  "paths": {
    "/static/{+file}": {
      "x-golem-worker-bridge": {
        "type": "file-server",
        "workerName": "<optional>",
        "component-id": "1b04d021-9e8e-4f44-832a-e73b32eeb0ba",
        "component-version": "0.0.1",
        "response": "{status: 200, file-path: request.path.file, headers: {ContentType: \'
      },
      "get": {
        "summary": "Get Twitter Home Page",
        "description": "Get the home page of twitter",
        "parameters": [
          {
            "name": "+file",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "OK",
            "content": {
              "image/png": {
                "schema": {
                  "type": "string",
                  "format": "binary"
                }
              }
            }
          },
          "404": {
            "description": "Contents not found"
```

```
          }
        }
      }
    }
  }
}
```

💡 **Links to Resources**

- Upload an API definition (Registering an API definition): https://github.com/golemcloud/golem/blob/main/golem-worker-service/src/api/api_definition.rs#L75
- Deploy the API in gateway: https://github.com/golemcloud/golem/blob/main/golem-worker-service/src/api/api_deployment.rs#L33

Golem CLI

The Golem CLI tool uses the REST API in order to create and update components. Once the Initial File System feature is implemented, then the CLI must be updated to support uploading the initial file system anytime a component is created or updated (see the `golem-cli component create` and `golem-cli component update` commands).

Since the *Initial File System* will be described inside the Golem Manifest file (`golem.yaml`), the CLI will need to identify and read this file, then package up all the files (both local and remote, by downloading remote files), and encode them appropriately for the REST API.

💡 **Links to Resources**

- Create/Update component command: https://github.com/golemcloud/golem/blob/main/golem-cli/src/command/component.rs#L31
- `golem-cli` component service that's responsible for doing the necessary steps before calling the server APIs:

  https://github.com/golemcloud/golem/blob/main/golem-cli/src/service/component.rs#L33

Worker Gateway

The Worker Gateway is the primary REST API interface to components deployed to Golem. The Worker Gateway is responsible for serving API Definitions by delegating to and invoking functions on specific workers.

In order to allow the *Initial File System* feature to be used for hosting content stored inside the file system of a worker, while still allowing this content to be under the control of the worker, it is necessary to add support for serving content directly to the Worker Gateway.

The Worker Gateway needs to be modified so that, for binding type of `file-server`, it no longer expects the Rib script to return a response that is converted to the response of the associated route. Rather, when the Worker Gateway encounters a binding type of a `file-server` for a route, it will understand the Rib script will return a file path (or an error). If the Rib script returns a file path, then the Worker Gateway will use the Worker Executor API in order

to stream the contents of the file located at the file path. This will allow the Worker Gateway to serve content from workers directly.

With the `file-server` binding type, the Rib script associated with a route must return a record containing at least the `file-path`, relative to the worker file system, and optionally, status and headers that may specify the content type (among other headers):

```
// Assumes route is something like: /static/{+file}
{file-path: request.path.file, status: 200, headers: {ContentType: "image/png"}}
```

If the record does not contain a header specifying content type, then the content type will be *inferred* based on the extension of the file, using a Rust library such as mime-infer.

💡 For convenience reasons, the worker gateway should support the Rib script returning the following alternatives to the "full record":

- A string, representing the file path. This can be used in very simple cases, where the Worker Gateway is responsible for inferring the media type of the file.
- A string wrapped in a `result`. This can be used, again, for simple cases, where the script can fail. If the script fails by returning `err` rather than `ok`, then the error message can be used by the Worker Gateway when generating the response. If the script returns `ok`, then the string result can be used as the file path.

💡 **Links to Resources**

- Serving a route (specified in the deployed API), where we resolve the worker binding (worker related information specified under each route in API definition) and then execute https://github.com/golemcloud/golem/blob/main/golem-worker-service-base/src/api/custom_http_request_api.rs#L98
- Rib expression which is part of worker binding is a separate module: https://github.com/golemcloud/golem/tree/main/golem-rib/src

Worker Executor

The Worker Executor contains an implementation of WASI file functions, which expose the underlying file system and allow the guest code to enumerate, read, and write files.

The Worker Executor needs to be modified in the following ways:

- **Presenting Initial Files**. The implementation of the file system interface must be changed to accurately present the contents of the initial file system.
- **Exposing Worker File System**. A new gRPC API for accessing the worker file system needs to be added to the worker executor.

Presenting Initial Files

Golem provides the following WASI interfaces that are related to the initial file system feature:

- https://wa.dev/wasi:filesystem

The core implementation of these host functions is coming from `wasmtime` , which uses `cap_std` under the hood. Golem's worker executor (`golem-worker-executor-base`) wraps this implementation to add observability and durability features on top of them. The API itself is based on the `descriptor` WIT resource, which points to either a file or a directory. The `preopens` interface returns the initial set of available descriptors - in Golem, that is the component's root directory (`/`). The component can only derive new descriptors using the existing ones.

We can avoid the need to copy/reimplement the whole filesystem interface if assume the following:

- In the manifest (and on all relevant APIs), directories in the initial file system and files are marked as "read-only" or "read-write".
- For read-only files, there is one copy of these files per Worker Executor (cached on the local disk of the Worker Executor) if there is at least one worker running using the component it belongs to.
- Read-write files are immediately placed in the worker's own directory, each worker having its own copy, which diverges from others.

In this case, the only worker-context change is to add more preopened directories with the proper permissions, which is already supported by `wasmtime` (although need to check what happens if one preopened directory is in the subtree of another).

💡 **Links to Resources**

- Link to the place where the initial file descriptors are specified: https://github.com/golemcloud/golem/blob/main/golem-worker-executor-base/src/wasi_host/mod.rs#L105
- Link to the code that downloads and caches information from the component service: https://github.com/golemcloud/golem/blob/main/golem-worker-executor-base/src/services/component.rs

Exposing Worker File System

The API for the Worker Executor is based on gRPC, and includes all of the functionality required by higher level components in Golem, such as the REST API or Worker Gateway.

Currently, there is no API which exposes the contents of a worker's file system. In order for the Worker Gateway to serve files, a new gRPC API must be added which supports the operations:

- Listing the children of a node in the file system of the specified worker, including any metadata that is easy to access.
- Retrieving (as a stream) the contents of a file in the file system of the specified worker.

The following is a sketch of what this gRPC API could look like:

```
syntax = "proto3";

service WorkerFileSystem {
  // List the contents of a directory
  rpc ListDirectory(ListDirectoryRequest) returns (ListDirectoryResponse) {}

  // Retrieve the contents of a file
  rpc GetFileContents(GetFileContentsRequest) returns (stream FileChunk) {}
```

```
}

message ListDirectoryRequest {
  golem.worker.WorkerId worker_id = 1;
  golem.common.AccountId account_id = 2;
  string path = 3;
}

message ListDirectoryResponse {
  repeated FileSystemNode nodes = 1;
}

message FileSystemNode {
  string name = 1;
  NodeType type = 2;
  uint64 size = 3;  // Size in bytes, applicable for files
  string permissions = 4;  // String representation of file permissions
  int64 last_modified = 5;  // Unix timestamp
}

enum NodeType {
  FILE = 0;
  DIRECTORY = 1;
}

message GetFileContentsRequest {
  golem.worker.WorkerId worker_id = 1;
  golem.common.AccountId account_id = 2;
  string file_path = 2;
}

message FileChunk {
  bytes content = 1;
}
```

Integration Testing Requirements

In addition to unit tests, there are three separate levels of integration tests in Golem and the **Initial File System** feature should be tested separately on each level:

- **Worker Executor Tests** (https://github.com/golemcloud/golem/blob/main/golem-worker-executor-base/tests/lib.rs#L42-L59) directly test the Worker Executor by running test components on it.
- **Integration Tests** (https://github.com/golemcloud/golem/blob/main/integration-tests/tests/worker.rs) start all the services of Golem locally and have access to the same test DSL as worker executor tests to upload and run test components on the system.
- **CLI Tests** (https://github.com/golemcloud/golem/blob/main/golem-cli/tests/main.rs#L14-L20) also work with all the services started locally, but they only interact with them by running the `golem-cli` app.