University of Southern Denmark
Department of Mathematics and Computer Science,
IMADA

Master Thesis

Building a Package Manager for Jolie

Author:
Dan Sebastian Thrane

Supervisor:
Fabrizio Montesi

April 25, 2017

Contents

Introduction

Introduction goes here.

Background

Background goes here.

2.1 Microservices

- General stuff about microservices, actually introduce the thing

2.2 Package Managers

Some background on package managers.

2.3 Introduction to Jolie

Jolie is a service-oriented programming language, and is build to support a microservice natively. In this section we will cover what kind of language Jolie is, and how it is currently used.

Jolie has a C-inspired syntax, and is dynamically typed. Its interpreter is written in Java.

The language has no native functions or methods, but instead works in processes. A process has no arguments, and does not contain any stack (in the case of recursive calls). There are two pre-defined processes, which will always be called by the interpreter, these are called `init` and `main`.

```
1  include "console.iol"
2
3  define PrintOutput {
4      println@Console(output)() // Prints 'OK'
5  }
6
7  init { a = 1 }
8
9  main {
10     b = 2;
11     c = a + b; // c = 3
12     if (c == 3) {
13         output = "OK"
14     } else {
15         output = "Bad"
16     };
17     PrintOutput // Calls the defined process 'PrintOutput'
18 }
```

Listing 1: A very simple Jolie program

Listing 1 shows a very simple programming language, in what looks like what you might expect from a dynamic language with C-inspired syntax. However a few things may also strike you as odd.

First of all there are typos on lines 12, 14 or 16, the semicolon is not needed here, in fact it would be a syntax error. The reason for this is that the semicolon isn't used strictly for parsing purposes, but it instead for having multiple statements in a process. The "semicolon" statement, also called a sequence statement, has a syntax of `A ; B`, which should be read as: first perform statement `A`, then perform `B`. The sequence statement requires both of the operands to be present, hence the syntax error. Another similar statement is the parallel statement, which has a syntax of `A | B`, which reads as: do `A` and `B` in parallel. Using these operators together allows the programmer to easily create a fork-join workflow. This is typically used in microservices when we want to collect data in parallel, and continue once all of the data has been retrieved.

Secondly has slightly different rules for scoping. In Jolie everything not defined in the global scope goes into the same scope. This also persists through calls to defines. This is the reason that `PrintOutput` can use the output variable.

Several execution modes exists. The default execution mode, which was used in Listing 1 is `single`. This means that the `main` process is run just a single time. Two more modes exists, those being `concurrent` and `sequential`. TODO Some more stuff

Ports are the primitive that Jolie uses for communication, two types of ports exists: input and output. Ports describe a running service, where it is located (`Location`), and how to speak to it (`Protocol`), and finally which operations it supports (`Interfaces`). In Listing 2 we see a simple output port which contacts `example.com` on port 42000, using `http`. Note that it is only the ports that deal with the protocol, everything inside of the code is completely agnostic with respect to the protocol being used for communication. As a result it is easy to change a service from communicating using one protocol to another.

```
outputPort Example {
    Location: "socket://example.com:42000"
    Protocol: http
    Interfaces: IExample
}
```

Listing 2: A simple output port which contacts the Google website

The interface which the port uses is called `IExample`, a full definition of it can be seen in Listing 3. Two types of operations exists in Jolie, namely `RequestResponse` and `OneWay`. The difference being fairly self-explanatory, the first receives a request and returns a response, the other simply receives a request, and produces no response.

```
interface IExample {
    RequestResponse:
        anOperation(RequestType)(ResponseType)

    OneWay:
        hello(HelloType)
}
```

Listing 3: An interface in Jolie defines which operations a port exposes

Whenever the Jolie interpreter invokes an operation on an output port, or receives a request on an input port, the types will be checked. This check ensures that we don't send out incorrect requests, and ensures that we do not attempt to process an incorrect request. Listing 4 show the request and response type of `anOperation`.

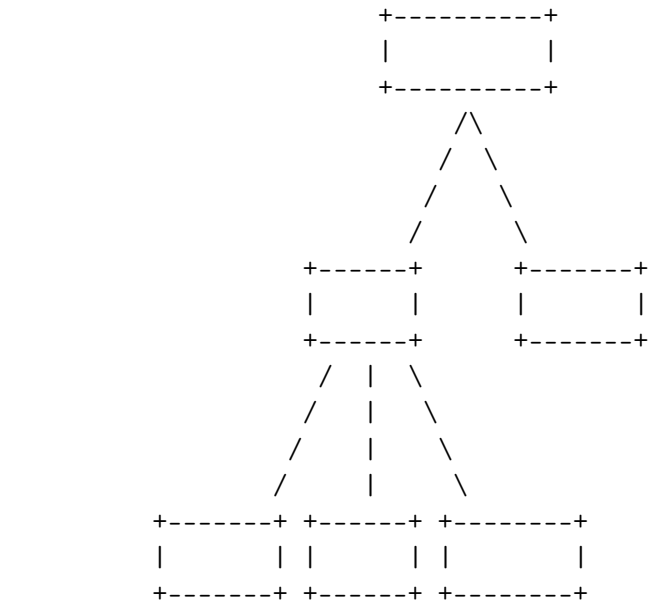
```
type RequestType: void {
    .a: int
    .b: string
    .c: bool
    .d: double
    .e: any // any primitive type
}

type ResponseType: int {
    .aFixedArray[1, 3]: string
    .aNonFixedArray[0, *]: string
    .fieldWithChildren: void {
        .a: void {
            .b: int
        }
    }
}
```

Listing 4: Jolie types are tree-like structures

In Jolie types are tree-like structures, very similar to how, for example, XML would be represented. Importantly the root may also contain a value, this is different from how most other programming languages work. This also means that certain encodings may have problems with this. JSON a popular format for serialization does not support root values, as a result Jolie will encode the root value under a special key to work around this fact.

TODO Actually use this illustration for something, should probably work with the actual example provided.



Jolie natively supports a variety of techniques for composition of services. The most important (for this work), which we will cover here are **aggregation** and **embedding**.

Embedding allows for a larger service to run smaller services as inner components. These services communicate with each other using more efficient local communication. These embedded services can be other Jolie services, but may also be services written in, for example, Java or even JavaScript. Communicating with these services is done exactly the same way as with any other service, it is entirely transparent to the application code where the service is located. TODO Can probably say a few more words about this subject. Might be easier to add this later when we know what we actually need.

Aggregation is a generalisation of proxies and load balancers. An illustration of this concept can be seen in Figure 2.1. Aggregation is useful for creating a wide variety of proxy like architectural patterns. The aggregation feature is often used along side the courier and interface extender features. Couriers allows the developer to insert code in-between the receiving the request and forwarding it. These features for example allow you to add authentication to a service which otherwise doesn't have it. This is done entirely without having to touch the original service.

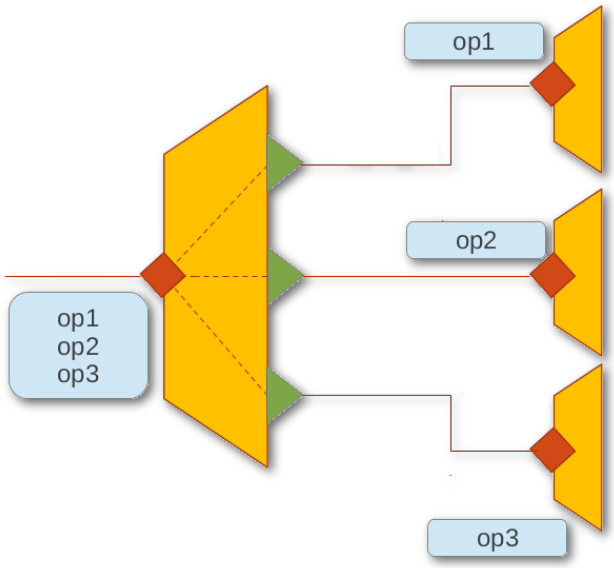


Figure 2.1: Aggregation is a generalisation of proxies and load balancers

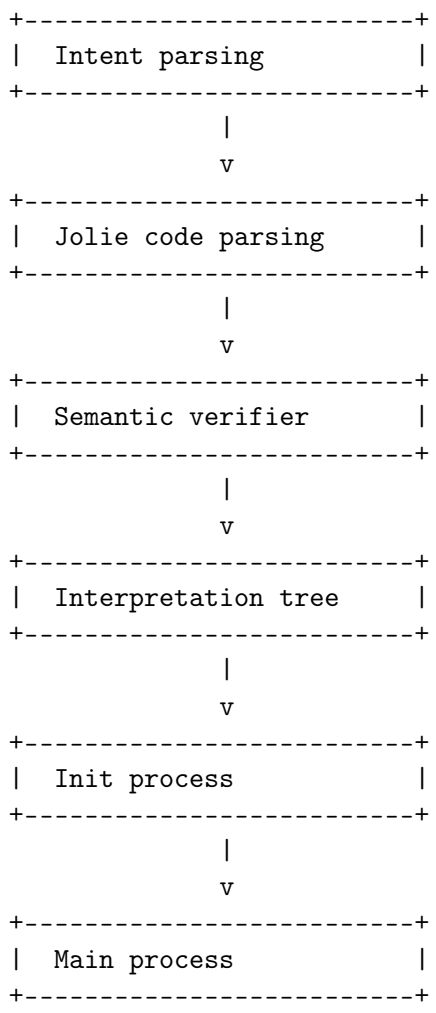
2.4 The Jolie Engine and Interpreter

- Values and types
- Phases of the compiler
- Anything we need to explain the changes that were made

In this section the internals of the Jolie engine will be introduced. This should give the reader the necessary knowledge to understand the changes made to support a module and package system for Jolie.

At the core of the Jolie engine is the interpreter. Each interpreter is responsible for parsing and executing a Jolie program. A single engine may run several instances of the interpreter, this is most commonly the case when embedding several other Jolie services.

A simplified view of the Jolie interpreter's pipeline can be seen in Figure 5.



Listing 5: A simplified view of the Jolie interpreter pipeline

The first phase of the interpreter is parsing the intent. In this phase we essentially figure out why the interpreter has been created, and what actions it should perform.

The Jolie engine can be invoked from the command-line, the command-line is the source of the intent which starts the first interpreter. The syntax for the command line is (roughly) as follows: `jolie [commands and options] <program> [program arguments]`. The options passed change the overall behaviour of the engine, and all interpreter instances share these. Commands and program arguments, however, only belong to the interpreter that they were originally passed.

The intent parsing phase is also responsible for locating and retrieving the program used. This is passed to the program parser. The program parser is the second phase, and is responsible for creating the Abstract Syntax Tree (AST) which represents the input

program. The parser will produce only a single root node, namely the `Program` node. As a consequence of this, any file which is included is semantically identical to copying and pasting the source code of that file into the original file, in place of the include statement. It should also be noted that include paths are *not* relative to the file that includes, but rather relative to the current working directory (i.e. where the engine was started).

The third phase traverses the AST to make sure that it is semantically valid. This weeds out programs which are syntactically correct, but do not make sense. The amount of work done in this phase is somewhat limited, given the otherwise dynamic nature of the language.

Given a semantically valid AST the interpreter is ready to build the interpretation tree. The interpretation tree contains new nodes, known as processes. Each of these processes can be run, to execute the correct behaviour.

Once the interpretation tree is build, we're ready to execute the actual code (which lives in the interpretation tree). First the init block is run, followed by the main block.

2.5 A Complete Application with Jolie

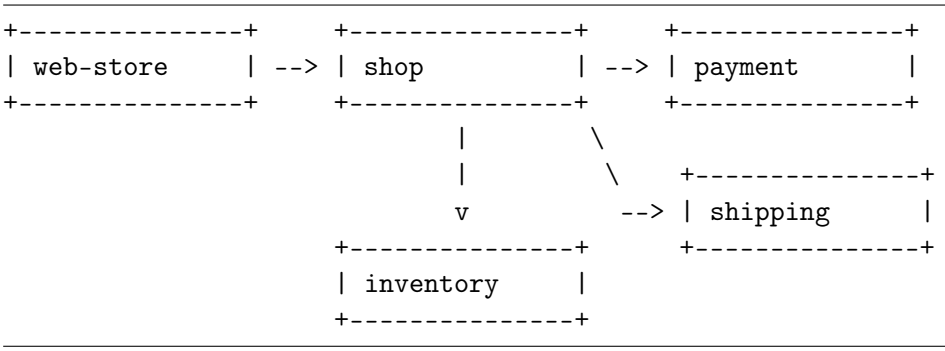
In this section we will describe a complete application written in Jolie. The application will contain several services, and will be written using best practices from before the module and package system.

2.5.1 Architecture

The application we will be building is a very simple shop application. This *shop* application will be able to look in inventory, and reserve a product, and arrange for *shipping*, while charging *payment* from the customer.

The architecture of our application is shown in Figure 6 we see an illustration of the system's architecture. The *shop* service will be contacted by the front-end service, which in this case is *web-store*.

The dashed region (TODO) displays services that should be running in the same Jolie engine. This is accomplished via embedding, which was introduced in a previous section.



Listing 6: The Architecture of a Simple Microservice System

2.5.2 Implementing the Shop Service

We will keep our focus on the *shop* service, and it's interacting with peering services. Putting together the stuff learned from the previous sections, we can quickly setup an input port for the service, which has the appropriate interface. It is considered best practice to place the public interfaces that a service exposes in its own separate file. Files intended for other services to include typically have the file extension `.iol` as opposed to `.ol`. There is no technical difference between the two, but it allows for the developer to more easily express intent.

Thus in order to implement our shop service we create two files, one for the service implementation (`shop.ol`), and another which can be used by other services (`shop.iol`).

```
// shop.iol
interface IShop {
    RequestResponse:
        checkout(CheckoutReq)(CheckoutRes),
        browse(BrowseReq)(BrowseRes)
}

// shop.ol
include "console.iol"
main {
    [checkout(request)(response) {
        println@Console("Implementation goes here")()
    }]

    [browse(request)(response) {
        println@Console("Implementation goes here")()
    }]
}
```

Listing 7: TODO Caption

The operations that the shop exposes, needs to collaborate with the shipping, payment, and inventory services. In order for us to speak to them they need output ports.

First of all the output ports needs interfaces. Like we did with the shop service, the other services have exposed their interfaces in a special file intended for inclusion. As a result we will have to copy the `.iol` files of these services into our own.

Secondly these output port needs to be reached. We can either embed the services, making it run inside of the same Jolie engine as our shop service, our we can provide external bindings to it. For output ports we may change this binding dynamically at runtime. Note that this is unlike input ports which must be ready at deployment time.

Binding an output port to an external service is relatively easy. For example to let the `Payment` port bind to a service using https, we might write `Location: "socket://paymentprocessor.com:4` and `Protocol: https`. The input port at the payment processor would also have to match this, to ensure it runs on the correct port and speaks the correct protocol.

It is a similar to bind an output port to an embedded service. However this is done by setting the `Location` or `Protocol` attributes. We must instead instruct the engine to embed the service, which most importantly requires us to point to some executable service. The Jolie engine supports several language for these embedded services, including Jolie, Java, and JavaScript. For our desired deployment, we wanted to embed the inventory service inside of the shop service. Assuming that the inventory service is written as a

Jolie service, and its service is implemented in `inventory.ol`, then we may create an embedding as shown in Listing 9. Just like in the case of the external services, the input port of the receiving service *must match*. In the case of embedded service there must be an input port listining on the `"local"` location.

Quite often the location of an input port is considered a deployment problem. We see this quite clearly in the case of embedding a service. All current solutions in Jolie, require us to *modify the source code of a service*, simply to change where the service should listen. Best practices in Jolie attempt to make this less of a problem by including a configuration file which contains constants. The inventory service, might include a file called `inventory_config.iol` with constants setting up the location and protocol of the service. An example of this is shown in Listing 8.

```
// inventory_config.iol
constants {
    INVENTORY_LOCATION = "local"
    INVENTORY_PROTOCOL = sodep
}

// inventory.ol
include "inventory_config.iol"

inputPort Inventory {
    Location: INVENTORY_LOCATION
    Protocol: INVENTORY_PROTOCOL
}
```

Listing 8: A common Jolie practice for solving configuration of a service, is to include a file containing constants with the desired configuration.

```
embedded {
    Jolie:
        "inventory.ol" in Inventory
}
```

Listing 9: Embedding the `inventory` service in the `Inventory` output port

With the code from Listing 7 where the output port `Console` is defined. The output port points to an embedding of the console service, and is included directly in the `console.iol` file. This is a fairly common pattern used in Jolie, especially for services that work in a library-like fashion (i.e. not intended as a stand-alone service). This pattern is used for almost every single service in the Jolie standard library.

With the output ports correctly configured, we may now implement the actual business logic for our shop. For completeness sake this might look like shown in Listing 10.

```
1 [checkout(request)(response) {
2     // Do some local calculations
3     checkForStock@Inventory(/* ... */)(hasStock);
4     if (!hasStock) throw(OutOfStockFault);
5     reserve@Inventory(/* ... */);
6     charge@Payment(/* ... */)();
7     send@Shipping(/* ... */)()
8 }]
```

Listing 10: Implementing the checkout operation

```
.
+-- include
|   +-- shop.iol
|-- inventory.iol
|-- inventory.ol
|-- payment.iol
|-- shipping.iol
+-- shop.ol
```

Listing 11: File Structure of the Shop Service

Jolie Modules

3.1 Modules

- Probably need to cover some problems with the current solution, to give some context to the actual solution
- Present the actual solution of modules
- Cover the implementation.

3.2 Configuration

- Introduce why we need it
- Configuration format, profiles, modules

Package Manager

4.1 Packages

A Jolie Package is an extension of a Jolie Module. Recall that a Jolie Module was defined as a collection of resources, a name, and optionally an entry-point for the module. A package extends this concept by adding information required for package management.

A Jolie Package is described by a package manifest. The package manifest is a JSON file, which is always placed at the root of the package, and must be called `package.json`. The fixed location allows for the package manager to easily identify a package. The JSON format was chosen as it plain-text, and easy to both read and write for both humans and machines.

In listing 12 we show a simple package manifest. This manifest showcases the most important features of the manifest.A complete specification of the package manifest format can be seen in Appendix A. The service that this manifest describes is shown in figure 13.

```
{
    "name": "calculator",
    "main": "calc.ol",
    "description": "A simple calculator service",
    "authors": ["Dan Sebastian Thrane <dathr12@student.sdu.dk>"],
    "license" "MIT",
    "interfaceDependencies": [
        { "name": "math", "version": "1.0.0" }
    ],
    "dependencies": [
        { "name": "sum", "version": "1.2.X" },
        { "name": "multiplication", "version": "2.1.0" }
    ]
}
```

Listing 12: A Simple Package Manifest

```
+--------------+     +-------+
|              | ==> |  sum  |
|              |     +-------+
|  calculator  |
|              |     +------------------+
|              | ==> |  multiplication  |
+--------------+     +------------------+
```

Listing 13: A Calculator Service

Lines 2-3 take care of the module definition. The remaining attributes, however, are entirely unique to packages. Some attributes included in the manifest are there for indexing and discoverability purposes, examples of such attributes are shown in lines 4-6. The rest of the document describes the dependencies of this package.

A JPM dependency is defined as a "code dependency". What this means is that the dependencies listed, should only be packages that we depend on for code. This might be different from the typical definition in microservices. For example, one might state that the calculator also has a dependency on the client who speaks to it. In JPM however, we will not need to list the client, as we do not depend on any code that the client has (in fact we know nothing about how the client works).

JPM deals with two different types of dependencies: interface dependencies, and ordinary dependencies. These two serve similar, but slightly different purposes, and all depends on how the dependency is to be used.

An ordinary dependency should be used if we wish to use the code of some other service, and want to option of embedding it. This will tell JPM to download both ordinary dependencies, as well as interface dependencies.

Interface dependencies are instead dependencies that are only required to interface with the package itself, and dependencies we need to interface with others. An interface dependency will only cause us to download the interface dependencies of each sub-dependency.

The interface dependency type isn't common in other package managers, but having multiple types of dependencies is fairy common. For example, a package manager might provide dependencies which are only used during testing, we see this in Maven, or it might provide dependencies only used during development, seen in NPM.

4.2 Architecture

The entirety of the ecosystem around JPM is written in Jolie, using the features that the Jolie Module System provides, along with the features that JPM itself provides.

At the ten-thousand foot view of the architecture, it consists of three core services, as shown in figure 14:

1. **Registry**: Responsible for serving packages known to the registry.
2. **JPM**: Provides the back-end of JPM. This includes communication with one, or more, registries, for example to download packages.
3. **CLI**: Provides the front-end of JPM. The front-end is responsible for displaying a user-facing interface, and will communicate with the back-end to perform the actual work.

```
+-----------+     +-------+     +------------+
|  jpm-cli  | ==> |  jpm  | ==> |  registry  |
+-----------+     +-------+     +------------+
```

Listing 14: Ten-thousand foot view of the JPM architecture

4.3 Registry

Stuff we need to cover in this section:

- Core responsibilities, such as: publish and download
- Secondary responsibilities: Package information and dependencies
- Tertiary responsibilities: Users and groups
- Once responsibilities are in place we should talk about how we split up the workload.

4.4 Authorization

In this section we need to discuss:

- Access control matrix, and how that stuff works.
- User registration and best practices for handling passwords

4.5 The Command Line Interface

We need to cover:

- Basic introduction of how to use the tool (we need this)

- Internal organization (commands)
- Technical architecture, and deployment
- Kind of wrap up how the entire thing works here

4.6 Integrity Checks

This is a section about integrity checks in JPM.

- Some background about checksums (probably)
- How we do it. Reason we don't go for something like code signing (in pkg manager)
- Code signing, and why we would prefer this.

Appendix

A.1 Appendix A: JPM Manifest Specification

This document covers the specification of the file which defines a package. The format used for this document will be JSON, but the format and whether or not to allow for several documents is still up for discussion. For now we should avoid using any features which the generic Jolie value cannot support.

A.1.1 Purpose

The purpose of the package document is to define what a package is. Every Jolie package will contain such a document, and it describes several important properties about the package. These properties are described in the section "Format and Properties".

A.1.2 Table of Contents

A.1.3 Format and Properties

name

Name: `name`

Optional: false

Type: `string`

Description: The `name` property uniquely defines a package in a registry. Every registry must only contain a single package with a given name.

Rules:

- The name of a package is *not* case-sensitive
- The length of a name is less than 255 characters
- Names are US-ASCII
- Names may only contain unreserved URI characters (see section 2.3 of RFC 3986)

If any of these rules are broken the JPM tool should complain when *any* command is invoked. Similarly a registry should reject any such package.

version

Name: `version`

Optional: false

Type: `string`

Description: This property describes the current version of this package.

Rules:

- The version string must be a valid SemVer 2.0.0 string (see http://semver.org/spec/v2.0.0.html)

license

Name: `property_name`

Optional: false

Type: `string`

Description: Describes the license that this package is under.

Rules:

- Must be a valid identifier. See https://spdx.org/licenses/

authors

Name: `authors`

Optional: false

Type: `string|array<string>`

Description: Describes the authors of this package

Rules:

- The array must contain at least a single entry
- Each entry should follow this grammar:

```
name ["<" email ">"] ["(" homepage ")"]
```

private

Name: `private`

Optional: true

Type: `boolean`

Description: Describes if this package should be considered private. If a package is private it cannot be published to the "public" repository.

Rules:

- By default this property has the value of `true` to avoid accidential publishing of private packages.

main

Name: `main`

Optional: true

Type: `string`

Description: Describes the main file of a package.

Rules:

- The value is considered to be a relative file path from the package root.

dependencies

Name: `dependencies`

Optional: true

Type: `array<dependency>`

Description: Contains an array of dependencies. See the "dependency" sub-section for more details.

Rules:

- If the property is not listed, a default value of an empty array should be used

dependency

Type: `object`

Description: A dependency describes a single dependency of a package. This points to a package at a specific point on a specific registry.

name **Name:** `name`

Optional: false

Type: `string`

Description: Describes the name of the dependency. This refers to the package name, as defined earlier.

Rules: A dependency name follows the exact same rules as a package name.

version **Name:** `version`

Optional: false

Type: `string`

Description: Describes the version to use

Rules:

- Must be a valid SemVer 2.0.0 string
- (This property follows the same rules as the package version does)

registry **Name:** `registry`

Optional: true

Type: `string`

Description: This describes the exact registry to use. If no registry is listed the "public" registry will be used.

Rules:

- The value of this property must be a valid registry as listed in the `registries` property.

registries

Name: `registries`

Optional: true

Type: `array<registry>`

Description: Contains an array of known registries. See the registry sub-section for more details.

Rules:

- This property contains an implicit entry which points to the public registry. This registry is named "public".

registry

Type: `object`

Description: A registry describes a single JPM registry. A JPM registry is where the package manager can locate a package, and also request a specific version of a package.

name **Name:** `name`

Optional: false

Type: `string`

Description: This property uniquely identifies the registry.

Rules:

- A name cannot be longer than 1024 characters

- The name cannot be "public"
- No two registries may have the same name

TODO:

- Encoding of name
- Should the length limit be dropped? There is no technical reason for the limit

location **Name:** `location`

Optional: false

Type: `string`

Description: Describes the location of the registry.

Rules:

- Must be a valid Jolie location string (e.g. "socket://localhost:8080")