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

May 31, 2017

Abstract

In this thesis, we develop a module and configuration system for the Jolie programming language. The features of these systems allow for new ways in which to create and distribute Jolie services. This makes it possible to change freely between binding to external services to internally embedded services entirely from configuration, and vice versa. Additionally, the configuration system introduces a notion of interface parametricity that can be used to develop generic Jolie services.

The module and configuration system lay the groundwork for the Jolie Package Manager (JPM). JPM is described and its implementation details are discussed in the second part of this thesis. JPM provides a helpful tool which makes it easier to develop and share Jolie services with other developers.

Resume

I dette speciale udvikler vi et modul- og konfigurationssystem til programmeringssproget Jolie. Funktionerne som dette system tilbyder, giver nye muligheder for at udvikle og dele Jolie servicer. Dette gør det muligt, at frit skifte, frem og tilbage, mellem eksterne bindinger til servicer og bindinger til en indlejret intern service. Desuden introducerer konfigurationsystemet en form for "interface parametricity", som tilllader udviklingen af generiske Jolie servicer.

Modul- og konfigurationssystemet er grundstenen, som tillader udviklingen af JPM (Jolie Package Manager). JPM beskrives og interessante implementerings detaljer diskuteres i den anden halvdel af dette speciale. JPM fremsætter en række værktøjer, hvilket gør det nemmere at udvikle og dele Jolie servicer med andre udviklere.

Acknowledgements

Acknowledgements goes here.

Contents

Introduction

TODO Something which gets us onto the package track.

Packages come in many different forms, and what they are mostly depends on the system that they serve. For example, binary package managers may ship applications which are then installed on the user's system. An example of such as package manager could be the iOS App Store. In general, a package manager primarily makes several things easier:

1. Publishing of Packages
2. Find packages to install
3. Automatically install dependencies
4. Handle available updates

These same principles have also been applied to the application level. Application-level packages are usually software components, which are intended to be run locally as a part of your application. Package managers also typically attempt to establish some sort of language/framework convention for developing packages. This is done such that packages may be imported into a software project in a consistent manner.

The goal of this thesis is to develop a package manager for the Jolie[Mon10; JOLA] ecosystem. Jolie is a service-oriented programming language. As such, we aim not only at covering the local packages (i.e., services running locally), but also external services. From the perspective of a local Jolie service, an external service is any service, which runs on an external virtual machine.

Before being able to achieve the goal of creating a package manager, a few hurdles must first be overcome. In this thesis, we will present a new natively supported module and configuration system for Jolie.

1.1 Structure of this Thesis

Chapter 1. This chapter. Introduces the thesis, explains the structure of it, and notations used throughout the thesis.

Chapter 2. Covers background material which is relevant for the entire thesis.

Chapter 3. Covers the module system and the configuration system which had been made to support the package manager efforts.

Chapter 4. Explains the concept of "Jolie Packages" which are what the Jolie Package Manager (JPM) supports. This chapter will also cover many of the features that are directly related to packages, which JPM implements.

Chapter 5. Covers architectural and implementation specific details for JPM.

Chapter 6. Conclusion of this thesis.

1.2 Notation Used

Service Diagrams

Simple box-and-line diagrams are used for illustrating how service are deployed, and how they depend on eachother. Figure 1.1 shows a simple diagram with two services `A` and `B`. In this diagram `A` depends on the services provided by `B`.

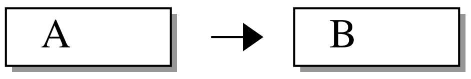

Figure 1.1: A service diagram showing two services: `A` and `B`. In this diagram service `A` depends on service `B`

We will illustrate embedding (a Jolie concept, see Chapter 2) by drawing a service inside another. The parent service will show its dependency on this service through similar means.

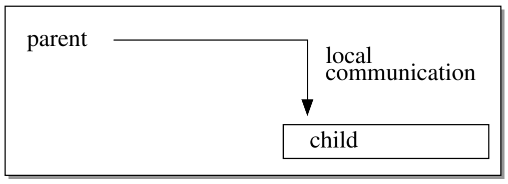

Figure 1.2: A `parent` service embedding a `child` service

File Listings

When code snippets are presented, where the file name and location is relevant then the file name and file path will be shown in a comment before the file contents. We will commonly show multiple files in a single listing. As an example, if we have the directory structure shown in Listing 1.1.

```
/
|-- a
|   `-- 1
|-- b
|   `-- 2
`-- c
    `-- 3

3 directories, 3 files
```

Listing 1.1: We will illustrate a directory structure in this manner

And the file contents of file `/a/1` is “A”, `/b/2` contains “B”, and `/c/3` contains “C”, then we would show the contents of these files as in Listing 1.2.

```
/* /a/1 */
A

/* /b/2 */
B

/* /c/3 */
C
```

Listing 1.2: Multiple files shown in a single listing where file path and name is relevant.

Command Line Illustrations

In certain cases, examples are made by showing the output of a command line session. The reader is expected to be familiar with a basic Unix-like shell. When the working directory is relevant, then it will be shown as part of the prompt. An example of this is shown in Listing 1.3. In this example two commands are invoked (`ls` and `cat hello-world.ol`. The `#` symbols mark the beginning of a command. Lines not containing this symbol should be considered as part of the output from the previous command. Before that the absolute path to the current working directory is shown, in these cases they were both `/this/is/the/path`.

```
dan@host:/this/is/the/path # ls
hello-world.ol

dan@host:/this/is/the/path # cat hello-world.ol
include "console.iol"
main {
    println@Console("Hello, World!")()
}
```

Listing 1.3: Illustration of two commands being invoked in a shell

1.3 Implementation Status

The module system has been implemented in a fork of the main Jolie repository. The fork is open-source and is available at `https://github.com/DanThrane/jolie`. As of writing this thesis, the work is available on the `feature-packages` branch. It is expected that this branch will eventually be merged into the master branch of the Jolie repository[1].

The package manager and the associated services are also made available, open-source. This repository is available at `https://github.com/DanThrane/jolie-packages`. This repository also contains a series of end-to-end tests, which also performs test made in the module system.

The software contains the features which this thesis describes. Although, in some cases, the software is missing minor features which could be desireable for public consumption. These minor features are almost entirely related to user interface tweaks and minor design adjustments.

[1]Available at `https://github.com/jolie/jolie`

Background

2.1 Microservices

In this section we will summarize the concept of microservices. This summary is mostly based on [Fow14].

Microservices are most easily explained by comparing them to, perhaps a more traditional architecture of, monoliths. A monolithic application is typically built as a single unit. This unit will contain all the components of which it is built.

Increasingly the industry has been feeling frustrations towards monolithic applications. Changes made to a monolith usually requires the entire application to be redeployed. Over time it may also become increasingly hard to keep the boundaries between components clean. Scaling of monoliths is also of concern, as it typically requires the entire application to be scaled, as opposed to just the components that need additional resources.

These frustrations led to a new architectural style, eventually named microservices. Microservices attempt to tackle each of these problems, but with a new architecture also comes new problems.

At its core, a microservice architecture will consist of many individual and independently deployable services. The fact that services must be designed to be independently deployable can make it significantly easier to scale. The services of a system will communicate with each other through message passing. There are no real restrictions on how services should communicate with each other. However, in practice services will typically communicate via the network, typically using “dumb” protocols, such as HTTP.

Microservices typically have a high focus on their own published interfaces. This comes, almost automatically, from the fact that they to communicate via message passing.

2.2 Package Managers

In this thesis we will work solely with application-level package managers. As mentioned in the introduction, application-level package managers deal with application-level packages. These packages are typically software components.

The work on the Jolie Package Manager (JPM), which this thesis presents, is hugely inspired by the following works:

1. **NPM:** The package manager for Node.js[NPMC].
2. **Yarn:** An alternative package manager for Node.js[YARNC].
3. **Cargo:** Build tool for the Rust language[CARB].
4. **Gradle:** Build tool, typically used for JVM languages (e.g. Java)[GRAB].

In the remainder of this section we will cover the most common traits from these works.

The primary role of a package manager, as the name would suggest, is to manage software packages. A package manager usually describes software packages through a special file, in this thesis we will refer to this file as the package manifest. NPM and Yarn both use a file called `package.json`, Cargo uses a `Cargo.toml`, Gradle using a `build.gradle`. Common for all of these are, that they typically describe the package itself, and the dependencies of this package.

Listing 2.1 shows an example for Cargo, and shows a similar manifest for NPM is shown in Listing 2.2. Common to both of these examples, is that their descriptions use a few attributes for describing the package itself. Additionally they have a section describing the dependencies, i.e., the packages that should be "managed". Most importantly of the attributes describing the package are the "name" and "version", which can uniquely refer to a specific version of the package.

```
[package]
name = "my-cargo-crate"
version = "1.2.3"
authors = ["Dan Sebastian Thrane <dathr12@student.sdu.dk>"]

[dependencies]
log = "0.3.8"
left-pad = "1.0.1"
```

Listing 2.1: A simple manifest for a Cargo package (a "Crate")

```json
{
  "name": "my-npm-package",
  "version": "1.0.0",
  "description": "This is a NPM package",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Dan Sebastian Thrane <dathr12@student.sdu.dk>",
  "license": "ISC",
  "dependencies": {
    "left-pad": "^1.1.3"
  }
}
```

Listing 2.2: A simple manifest for a NPM package

Most package managing tools come with CLI (Command-Line Interface) tools, which help the user perform various tasks. For example, both of the manifest examples (Listing 2.1, and 2.2) could be generated via their respective tools as shown in Listing 2.3.

```
dan@host:/ # cargo new my-cargo-package
    Created library `my-cargo-package` project

dan@host:/ # npm init
This utility will walk you through creating a package.json file.
It only covers the most common items and tries to guess sensible defaults.

( ... Cut for brevity ... )
```

Listing 2.3: Package managers typically ship CLI tools which help with common tasks

Similarly these tools can be used to download and install their dependencies, for Cargo `cargo build`, for NPM `npm install`. The packages, that these dependencies resolve to, can be stored in various places. A common approach, which we see both NPM, Yarn, and Cargo, is to use a default centralized registry. Gradle uses a similar approach but doesn't have a single default registry (which they refer to as repositories). The registries will act as a database of packages, which the tools will use to download, and install packages. Registries typically also have some type of user management. Such a feature becomes relevant when controlling who is allowed to publish updates for a package.

The reader may have observed already, that we have been referring to both Cargo and

Gradle as package managers, even though they were listed as build tools. The build tool description is pulled directly from their own descriptions. However, the border between package managers and build tools are typically blurry. Especially because many modern tools include dependency management as a core part of their features. Even tools, like NPM, which advertise themselves still build provide “build tool”-like features, such as running tests and starting the application. In this thesis, we will simply refer to our own tool, the Jolie Package Manager, as a package manager. This is done, since it is its primary responsibility, and technically there is nothing to build in Jolie, due to its dynamic nature.

2.3 Introduction to Jolie

Jolie is a service-oriented programming language and is built to support a microservice architecture natively. In this section, we will introduce the Jolie is, and show how it is 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, uses processes. A process has no arguments and does not contain any stack. There are two pre-defined processes, which will always be called by the interpreter, these are called `init` and `main`.

```
include "console.iol"

define PrintOutput {
    println@Console(output)() // Prints 'OK'
}

init { a = 1 }

main {
    b = 2;
    c = a + b; // c = 3
    if (c == 3) {
        output = "OK"
    } else {
        output = "Bad"
    };
    PrintOutput // Calls the defined process 'PrintOutput'
}
```

Listing 2.4: A very simple Jolie program

Listing 2.4 shows a very simple application written in Jolie.

The missing semicolons on lines 4, 13, 15, or 17 are not mistakes. The semicolon is not needed here, in fact, it would be a syntax error to add them. The reason for this is that the semicolon isn't used strictly for parsing purposes, but it instead used for composing statements in a process.

Semicolon statements, or sequence statements, have a syntax of `A ; B`. It should be read as: first, execute `A` then execute `B`. The sequence statement requires both operands to be present. Hence the syntax error for adding a semicolon to the last statement of a block.

A similar composition operator is, the parallel statement, which has a syntax of `A | B`, which reads as: execute `A` and `B` in parallel.

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.

Jolie has different rules for scoping than most languages. 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. To access the global scope we must explicitly state that we wish to use it. Accessing a variable `a` in global scope would require us to write `global.a`.

Several execution modes exists. The default execution mode, which was used in Listing 2.4, is `single`. With an execution mode of `single` the `main` process will run just a single time. Two additional execution modes exist: `concurrent` and `sequential`. Any of these two modes should be used if the Jolie program should act as a service. When running in one of these two, the Jolie program will act as a service.

When Jolie acts as a service, it will require the first statement of the main procedure to be an input. An "input" represents an operation call (request) going into the service, to which the Jolie program will need act upon. Listing 2.5 shows a Jolie service being capable of answering requests for either an operation called `echo` or `respondHello`.

```
main {
    [echo(request)(response) {
        response = request
    }]

    [respondHello(request)(response) {
        response = "hello"
    }]
}
```

Listing 2.5: A simple service capable of answering requests for either of the two types of operations: `echo` and `respondHello`

Ports are the primitive that Jolie uses for communication, two types of ports exist input and output. Ports describe a running service, where it is located (`Location`), how to speak to it (`Protocol`), and finally which operations it supports (`Interfaces`).

Listing 2.6 shows an output port `Example`. This output port will contact a web server running `http` on port 80 at `example.com`. Only ports deal directly with protocols, this means that the code interacting with the port remains the same, regardless of where it is located, or which protocol it uses.

```
outputPort Example {
    Location: "socket://example.com:80"
    Protocol: http
    Interfaces: IExample
}
```

Listing 2.6: A simple output port which contacts an example web server

The interface which the port uses is called `IExample`, a full definition of it can be seen in Listing 2.7. Two types of operations exist in Jolie, namely `RequestResponse` and `OneWay`. The difference is fairly self-explanatory, the first receives a request and returns a response, the other simply receives a request, and produces no response. When an operation call is made from a client, and the operation type is `RequestResponse`, then the client will wait for a response to come back before continuing. However, if the operation type is `OneWay`, then the client will not wait for anything, and simply return as soon as the message has been sent.

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

    OneWay:
        hello(HelloType)
}
```

Listing 2.7: 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 incorrect requests aren't sent. It also ensures that we do not attempt to process an incorrect request. Listing 2.8 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 2.8: 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. For example, a value of type `ResponseType` is shown in Listing 2.9.

```
value = 42; // root value
with (value) {
    .aFixedArray[0] = "A";
    .aFixedArray[1] = "B";
    .aNonFixedArray[0] = "foo";
    .fieldWithChildren.a.b = 1337
}
```

Listing 2.9: An example of a value which would type-check for the type `ResponseType`

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, through Jolie code, is done exactly the same way as with any other service. As a result, it is entirely transparent to the application code where the service is located.

Aggregation is a generalization of proxies and load balancers. Aggregation is useful for creating a wide variety of proxy like architectural patterns. The aggregation feature is often used alongside the courier and interface extender features. Couriers allow 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.

2.4 The Jolie Engine and Interpreter

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 2.1.

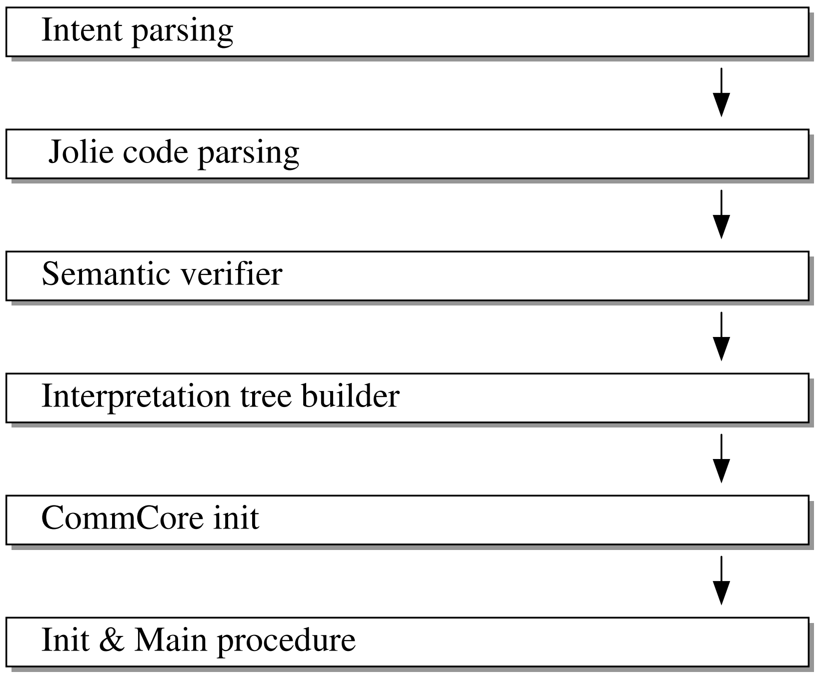

Figure 2.1: A simplified view of the Jolie interpreter pipeline

The first phase of the interpreter is parsing the intent. In this phase the interpreter will determine its purpose and, if any, modifiers to its ordinary behavior.

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]`.

Options change the overall behavior of the engine, all interpreter instances share these. These change the behavior of the interpreter.

Commands and program arguments, only belong to the interpreter that they were originally passed. These determine what the interpreter will do.

The intent parsing phase is responsible for locating and retrieving the program used.

A URI matching the input program is given to the program parser, the second phase. The parser 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. Includes are thus implemented by essentially inserting the contents of the include target. A quirk for Jolie includes, is that relative file includes are not resolved relative to the current file, but rather 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 that are syntactically correct, but not semantically valid. The checks performed in this phase are somewhat limited, given the 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, which are runnable.

Once the interpretation tree is built, we're ready to execute the actual code (which lives in the interpretation tree). The `CommCore` component is responsible for all communication, we'll discuss this briefly when relevant. Following its initialization, the user code of Jolie will start executing. The `init` procedure is executed first followed by the `main` procedure.

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.

Along the way, several patterns used for re-using code will be highlighted. This will serve as the core motivation for our solution. The observations made in this section are drawn from the Jolie Documentation[JOLB], the Leonardo[1] web server[LEO], and the source-code for the Jolie website[LEOB].

2.5.1 Architecture

The application we'll be building is a very simple calculator system. The architecture of our application is shown in Figure 2.2.

The calculator service will provide various operations for applying an operator on a sequence of numbers. In order to perform this, the action of applying these operators, the calculator will contact other microservices. This example is obviously contrived. Writing services for such simplistic services in the real world would be considered an anti-pattern. However, this example allows us to see services interact with each other without thinking that much about the business logic of the services, which in this case is fairly irrelevant.

[1]A web server written in Jolie by the Jolie team. Used to power the official Jolie website

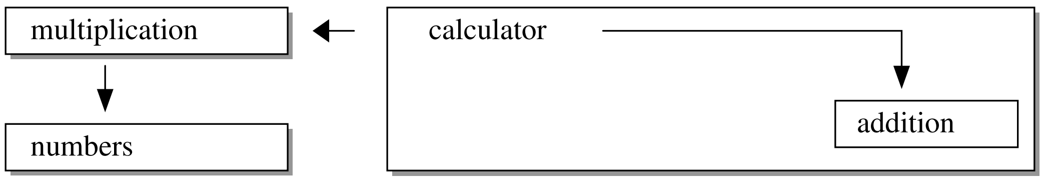

Figure 2.2: The Architecture of a Simple Microservice System

2.5.2 Implementing the Calculator Service

We will keep our focus on the `calculator` service, and its interactions with other services. Putting together the stuff learned from the previous sections, we can quickly create an input port for the service with 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. Files that are included can be placed in the `include` directory. This directory is always implicitly added to the search path.

Thus in order to implement our calculator service we create two files. One file for the service implementation (`calculator.ol`), and another which can be used by other services (`calculator.iol`). These files are shown in Listing 2.10.

```
// calculator.ol
include "console.iol"

inputPort Calculator {
    Location: "socket://localhost:12345"
    Protocol: sodep
    Interfaces: ICalculator
}

main {
    [sum(request)(response) {
        println@Console("Implementation
        ↪  goes here")()
    }]

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

```
// calculator.iol
type Numbers: void {
    .numbers[2, *]: int
}

interface ICalculator {
    RequestResponse:
        sum(Numbers)(int),
        product(Numbers)(int)
}
```

Listing 2.10: Interface and implementation of the `calculator` service

Observation 1. The interfaces and types of a service are typically separated into their own files. These files typically have the extension `.iol` to indicate that it does not contain a service implementation, but is rather intended for inclusion by a service which requires it.

These files are typically put in the `include` directory. This directory is always added to the search path natively by the Jolie engine.

The operations which the calculator exposes needs to collaborate with the other services. In order for us to speak to them, they need output ports. The output ports need interfaces, just like the calculator service these should be exposed from a dedicated file.

Secondly, these output port needs to be reached. The port can either be embedded or we can provide external bindings to it. For output ports, we may change this binding dynamically at runtime. This is unlike input ports which must be ready at deployment time.

Binding an output port to an external service, is done by setting the `Location` and `Protocol` properties. For example, to make `Multiplication` bind to an external service we could define it as shown in Listing 2.11.

```
outputPort Multiplication {
    Location: "socket://mult.example.com:443"
    Protocol: https
}
```

Listing 2.11: The `Multiplication` port is bound to an external service

It is possible to define embeddings of services in the `embedded` block. For our desired deployment, we wanted to embed the addition service inside of the calculator service. Assuming that the addition service is written as a Jolie service, and its service is implemented in `addition.ol`, then we may create an embedding as shown in Listing 2.12. 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 listening on the `"local"` location.

```
embedded {
    Jolie:
        "addition.ol" in Addition
}
```

Listing 2.12: Embedding the `addition` service in the `Addition` output port

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 solutions to change the deployment of a service in Jolie, all require us to *modify the source code of a service.* Best practices in Jolie attempt to make this less of a problem by including a configuration file which contains constants. The `addition` service, might include a file called `addition.iol` which sets up the location and protocol of the service. An example of this is shown in Listing 2.13.

```
// addition.ol
include "addition.iol"

inputPort Addition {
    Location: ADDIITON_LOCATION
    Protocol: ADDIITON_PROTOCOL
}
```

```
// addition.iol
constants {
    ADDIITON_LOCATION = "local"
    ADDIITON_PROTOCOL = sodep
}
```

Listing 2.13: A common Jolie practice for solving configuration of a service, is to include a file containing constants with the desired configuration.

Observation 2. Most configuration of Jolie services is done via constants. Jolie constants, different from most other languages, can be simple literal types, or they may even contain identifiers. This makes them rather versatile in what they can configure.

The constants are typically placed in a separate source file, which is included by the service requiring the configuration.

In Listing 2.10 the output port `Console` is used. This output port isn't visibly defined in the source code we see, it is instead defined in the included file `console.iol`. This is common practice in Jolie. This is especially used for services that work in a library-like fashion, that is not intended as a stand-alone service. This pattern is used for almost all services in the Jolie standard library[JOLB].

Observation 3. Services intended to be used as libraries are often contained in a single `.iol` file. This file contains everything required to set it up, included interfaces, types and an embedded output port.

This makes it very easy to use the service, however, it also makes it impossible to bind to this service externally, without first getting an embedding. This isn't possible since we cannot include the interfaces and types by themselves.

With the output ports correctly configured, we may now implement the actual business logic for our calculator. This might look like shown in Listing 2.14.

```
[sum(request)(response) {
    total = 0;
    for (i = 0, i < #request.numbers, i++) {
        // Add current number with total, and store result in total
        add@Addition({
            .a = total,
            .b =  request.numbers[i]
        })(total)
    };
    response = total
}]
```

Listing 2.14: Implementing the checkout operation

Finally, we want to reflect slightly on the file structure that this service has. In order to use external services, we had to include files which contain these interfaces. This version is worse when an embedding is desired, since the entire service with its implementation is suddenly required.

The files from these external services are also hard to manage. It isn't possible to simply move the source code of a service into its own directory. This is not possible since includes, unlike in most other languages, are not relative to the file performing the include, but rather relative to the project level root. Thus if we wish to move the service to a new directory, all includes in the source code would have to be updated. A common result of this is that every single file gets dumped into the project level root. When all files are stored in the same directory, we also get a much larger chance of having name conflicts. As a result, names tend to become rather large, to make it less likely that a conflict occurs.

The final file structure of the calculator service is shown in Listing 2.15.

```
.
|-- addition.iol
|-- addition.ol
|-- calculator.ol
|-- include
|   `-- calculator.iol
`-- multiplication.iol
```

Listing 2.15: File structure of the calculator service

Observation 4. In order to collaborate with other services, it is often needed to manually copy files into the service. Which files are required depend on the deployment. When an embedding is required, we will need the entire service (and any services it may depend on itself). If we just need to interface with it, we can simply include the files required for its interface files.

Structuring the files is problematic, due to `include` statements not being relative to file performing inclusion.

Jolie Modules

3.1 Modules: Design and Implementation

The module system of Jolie is made to facilitate better options for the modularization of code. This is what allows for a package-manager to exist. We discuss the package manager in detail in Chapter 4 and 5. A Jolie module is described by the following three properties:

1. **A name.** For a single running Jolie engine, this will uniquely identify a module
2. **A file path to the Jolie code-base.** We will refer to this as the project root or just root.
3. **(Optional) The entry point to the Jolie code.** This is a file path given relatively to the project root.

The Jolie engine needs to know about these modules. The engine is informed about these modules from the intent, passed as options, which starts the engine. This information is collected in the "intent parsing" phase and is made available to any later phase that might need it. Since the intent is passed as an option, all interpreter instances inside the engine will know about each module. As a result, any embedded service will also be aware of the same modules.

Consider a module "foo", with its source code placed at `/packages/foo` and an entry-point at `/packages/foo/main.ol`. It is possible to inform a Jolie interpreter about this module using `--mod foo,/packages/foo,main.ol`.

As a result, a lot of additional options must be passed to the engine. In this case, a package manager is expected to take the heavy lifting and figure out which modules exist. This allows for more freedom in how these tools are implemented. If another approach were to be found later, then a new implementation could simply be created without breaking the language.

A new include primitive has been added, a module include. This has been created to allow for better internal organization. This primitive allows the developer to include files from a particular module. The new primitive is shown in Listing 3.1. When a module include is used, the search path will be altered, such that the project root is changed to that of the package. As opposed to the original include primitive, which would look for files relative to the current working directory.

The new include primitive, along with native support for modules, allows for the code to be properly organized. With this, a module may be placed inside of its own directory, without any code has to change.

```
include "<file>" from "<module>"
```

Listing 3.1: Extension to the include statement, made for module imports

3.1.1 Module Include Algorithm

A module include is an extension of ordinary includes which specify which module to perform the inclusion from, that is `include "file" from "module"`.

The base case of such an include is simple. We make a "context switch" from our working directory into the root of the module and start searching for the file in that folder. This works quite well, however, consider the following scenario. From a client we perform a module include: `include "interface.iol" from "dependency"`.

```
/* /dependency/interface.iol */
include "types.iol"

interface Dependency { /* ... */ }

/* /dependency/types.iol */
type SomeRequest: void { /* ... */ }
```

Listing 3.2: Module includes may cause more ordinary includes. We must make sure the switch in context is kept

With the code in Listing 3.2, the system would incorrectly look for `types.iol` in the current working directory! To fix this a stack is kept. This stack gains an entry when a module include is performed, this entry is popped off the stack once the contents of that include has finished parsing. The stack is then used to inject the module's root into the search path for the ordinary includes.

3.2 Configuration

From Observation 2 we saw that most configuration was done via the inclusion of source code. This source code would expose constants (read: literal values *and* identifiers). The included source code, however, can do anything that Jolie source normally can, and is not limited to just the desired configuration. As a result, a service developer cannot be certain that the configurator (entity providing configuration) doesn't start messing with other details of the program. Deploying defensive programming[1] techniques against this becomes significantly more problematic, since no guarantees about the configuration source file can be made.

Distributing re-useable packages is problematic with this approach. Common features of package managers expect the source code of packages to be read-only, for example, when updating packages. Without this property, source-code merges would be required.

This gives us plenty of reason to explore the need for a native configuration format. Most other systems would most likely go for a system defined in user code, as opposed to natively. An example of such framework, is Vert.x, a tool-kit for building reactive applications on the JVM. Examples of such "reactive applications" are microservices. The configuration workflow of Vert.x[VERTA] is shown in Figure 3.1. The system will retrieve, and read external configuration files, directed by the user code, and apply the configuration as needed.

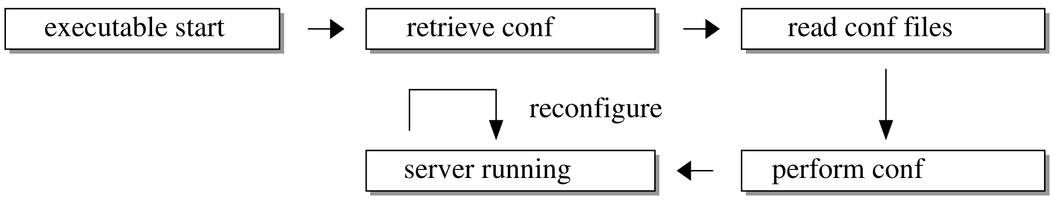

Figure 3.1: Simplified workflow for configuration of Vert.x applications

However, it is not straight forward to implement such a system in Jolie. In general-purpose languages, the constructs (such as a server's socket) for the microservice architecture are created in user code. As a result they are entirely accessible from user code. This make it feasible to change their behaviour, since code can run before deployment occurs.

In Jolie, the constructs are managed directly by Jolie. This has multiple advantages, such as less complexity in user code, but Jolie user code also loses power in the process. For example, a Jolie program cannot control the network directly, but must instead use the abstractions provided by the language (message-passing).

The primary need for a native solution comes from input ports. Input ports in Jolie require their values to be set at deployment time. It is not possible to reconfigure their

[1]Defensive programming techniques are usually employed for systems that require safety, security, or high availability [Jon05].

location during runtime, and for good reasons. TODO Reasons can't come up with any right now.

Additionally, creating a native solution provides many opportunities in terms of creating more powerful features. For example, the configuration format allows us to easily move between embedding a service, and binding to an external service.

3.2.1 Introducing the Jolie Configuration Format

In the coming section we will cover the Jolie Configuration Format (file extension: `.col`).

This new configuration format should allow a developer to provide configuration for a service, which works in the setting of packaged services. This will entirely replace the old system of using a mix of constants and hard-coded values.

The configuration system will only allow configuration of constructs which has been marked as such by the module. This should also allow a developer, and potentially tools, to quickly identify which constructs allow configuration. As much as possible, the code of a module should act as documentation of itself.

Additionally the system should work with the existing checking tools, e.g., the `--check` flag. The checking tools of Jolie are, for example, used by editor plugins to show errors. These checks do not execute any code. The checking algorithm should check that the provided configuration is valid.

A configuration file is build up of *configuration units*. A configuration unit is the basic entity, which encapsulates the configuration of a single Jolie module.

A unit is uniquely identified by its profile name and module. Having multiple profiles for the same module can be useful for a variety of use-cases. A common use-case, for example, could be separate profiles for development and production.

The units can hold configuration for every possible type of configurable construct in Jolie. The configuration format support the following constructs:

1. Input and output ports
    - Location
    - Protocol and protocol parameters
    - Embedding of other modules (output ports only)
2. General purpose parameters (values accessible from Jolie programs)
3. Interface rebinding

We'll start by introducing the configuration format through examples. The configuration format is custom, and made to mimic the syntax of Jolie. This decision was made to make it easier to convert existing Jolie code.

Listing 3.3 shows a simple configuration unit. This units sets the `Location` and `Protocol` of the output port `A` (line 3 and 4), the location of the input port `ModuleInput` (line 8), and two parameter assignments (line 11 and 12).

```
 1  profile "hello-world" configures "my-module" {
 2      outputPort A {
 3          Location: "socket://a.example.com:3000"
 4          Protocol: sodep { .keepAlive = true }
 5      },
 6
 7      inputPort ModuleInput {
 8          Location: "socket://localhost:80"
 9      },
10
11      myParameter = 42,
12      myParameter.subProperty = "hello"
13  }
```

Listing 3.3: A simple configuration unit named `hello-world` configuring the module `my-module`

Embedding of output ports can be performed from within a configuration unit. This moves the embedding from being a code problem to, what it should have been, a deployment problem.

Listing 3.4 shows the embedding of output port `A`. Note that we need to make a reference to the module (line 2), since the profile names are placed under a namespace for each module. This way multiple services can share the same name, a situation which is likely to occur with common profile names, such as "development" and "production".

```
1  profile "hello-world" configures "my-module" {
2      outputPort A embeds "a-module" with "a-profile"
3  }
4
5  profile "a-profile" configures "a-module" {
6      // configuration of a-module goes here.
7  }
```

Listing 3.4: Embeddings make reference to other configuration units

As seen in the examples, providing partial configuration, as opposed to complete configuration, is allowed. In certain cases, partial configuration is provided because the remaining values are defined by the module itself. In other cases, partial configuration is provided due to the unit being an extension of another unit.

A configuration unit may extend, at most, one other unit. The two, however, must configure the same module. If the child and parent disagree on configuration, the child always decides. For example, if unit “B” extends “A”, and they both configure the same value, then the constructs found in B are the ones that are used. Listing 3.5 shows an example of inheritance.

```
profile "a" configures "a-module" {
    aValue = 42,
    aValue.sub = "hello",

    outputPort ExternalService {
        Location: "socket://external.example.com:42000"
    }
}

profile "b" configures "a-module" extends "a" {
    aValue = 100
    // aValue.sub = "hello"
    // ExternalService.location = "socket://external.example.com:42000"
}
```

Listing 3.5: Configuration units may extend other units

The module developer is often aware of what the default configuration should be. For this, default configuration profiles can be shipped along the modules. The default configuration units are implicitly imported into every configuration file which uses a particular module. The Jolie engine will look for any `.col` file in the `conf` folder. This folder should be placed relative to the module’s root. For example, if module "a" has a file called `conf/my-defaults.col`, which contains a unit called "default". Then the user of the package may either write a configuration unit which extends this, simply by writing `profile "something" configures "a" extends "default"`, or the default directly. There is no need for any inclusion of this file, due to the implicit include.

While no configuration unit is required to be complete, the one used by an interpreter must be complete. If an interpreter is instructed to use a partial configuration unit an error will be produced.

3.2.2 Input and Output Ports

Input and output ports, both existing constructs, are defined just like before in the Jolie code.

Externally configurable ports are created by leaving out fields that should be configurable. Listing 3.6 shows a configurable input port.

Only the fields which are not already listed become configurable. This is useful when building a service that needs to make assumptions about its input ports. For example, if a developer is building a generic web-server, it is useful to allow the developer to change the location, but the protocol should remain fixed.

One typical assumption that Jolie services make about their input ports come from aliases that are made in the protocol configuration, an example is shown in line 4 of Listing 3.6. The pointer statement takes two variable paths and makes the one on the left link to the one on the right. Setting `statusCode` will cause the HTTP status code to be set accordingly. It should noted, configuration units *do not* support aliases, and there are no other ways of accessing variables hidden in the protocol configuration. As a result, any service which needs to do something special with its protocol configuration, like aliasing, must do it in source code. Since the aliased variable now also takes on a new semantic meaning, it also makes sense that it must be named directly in source code, and not left to configuration, where it could take any arbitrary name.

```
1  inputPort MyWebServer {
2      Protocol: http {
3          .keepAlive = true;
4          .statusCode -> statusCode;
5      }
6      Interfaces: MyWebServerInterface
7  }
```

Listing 3.6: A bare-bones configurable input port for a web-server

However, simply leaving out fields, and assuming they must be configurable proves somewhat problematic for output ports. Output ports require no fields, other than the interfaces, to be defined at deployment time. It is quite common for dynamic ports to not add a default location or protocol, assuming it is changed before use. To deal with this problem, it was decided to add a new keyword (`dynamic`) for output ports which need to change their location and protocol dynamically. Only output ports marked with this keyword are allowed to be changed at run-time. Dynamic ports are also not configurable.

3.2.3 Parameters

Parameters are read-only values which are provided to a Jolie module at deployment time. A parameter is type-checked at deployment time, to ensure that its type matches what the underlying service expects. Listing 3.7 shows a parameter definition and its use.

The type-checking feature functions both as a mean of documentation and ensuring that the supplied configuration is valid. Like any other type-checked feature of Jolie, it is possible to opt-out simply be setting the type to be `undefined`.

```
// service.ol
parameters {
    myParameter: string {
        .foo: int
        .bar: bool
    }
}

main {
    println@Console(myParameter)(); //
      ↪  "Root"
    println@Console(myParameter.foo)(); //
      ↪  42
    println@Console(myParameter.bar)() //
      ↪  false
}
```

```
// service.col
profile "my-profile" configures "service" {
    myParameter = "Root",
    myParameter.foo = 42,
    myParameter.bar = false
}
```

Listing 3.7: A parameter definition and its use

From a high-level point of view, parameters are very similar to constants. However parameters weren't implemented as an extension of constants due to the existing implementation of constants. Namely, constants aren't implemented using the "Value" system, which is used by all variables and messages in Jolie, but rather implemented at the parser level.

When the Jolie parser encounters a constant definition, it will save the token from the assignment and associate it with the identifier on the left-hand side. Only identifier tokens and simple value tokens are allowed. The constant definitions are limited to only a single token. As a result, it is not possible to define more advanced tree-like values, which parameters such as `myParameter` from Listing 3.7 would require.

Whenever the parser reaches a place where a constant would be allowed, it will look at the next token, check if it is an identifier, and try to replace the current token with the replacement token defined by the constant. This produces some rather surprising results and has limitations which are not commonly found in other programming languages. Listing 3.8 illustrates how the Jolie parser processes constants.

```
constants {
    FOO = 42
    BAR = ActualInterface
}

init {
    println@Console(FOO)()
}

courier Foo {
    [interface BAR(req)(res) {
        /* ... */
    }]
}
```

```
init {
    println@Console(42)()
}

courier Foo {
    [interface ActualInterface(req)(res) {
        /* ... */
    }]
}
```

Listing 3.8: Constants in Jolie works by replacing tokens at the parsing level. Left: The input program. Right: The program which the parser ends up seeing

While it would have been possible to extend the value system to support values it was ultimately decided against. Adding both optional type-checking of constants and expanding to values was seen as too big a departure from the original intent of constants. For backward compatibility reasons constants would also still not have been pure values, but rather either an identifier or a value.

The new parameters block is an addition to the AST of Jolie programs. This addition currently only works in collaboration with the configuration and module system. If any parameter is not defined by configuration or has the wrong type the checking scripts of Jolie will throw an error.

Even though the parameters block is added to the AST it will not be visible in the final interpretation tree. We'll learn more about this in Section 3.3.

3.2.4 Interface Rebinding

In this section we will first look at the existing aggregation, and courier features of Jolie. This leads to the introduction of interface rebinding which is explained following that.

Jolie provides the "aggregation" feature. We first introduced aggregation in the background material for Jolie. In short, this feature is a generalization of proxies. The aggregation feature simply forwards requests from one service to another. A simple proxy to the calculator service is shown in Listing 3.9.

```
include "calculator.iol" from "calculator"

inputPort Self {
    Location: "socket://localhost:12345"
    Protocol: sodep
    Aggregates: Calculator
}

outputPort Calculator {
    Location: "socket://calc.example.com:12345"
    Protocol: jsonrpc
    Interfaces: ICalculator
}
```

Listing 3.9: A calculator proxy: This service will proxy any call to the calculator service bound in the output port `Calculator`

Aggregation can be extended by using couriers, which allows for the service to run code associated with requests that are proxied. Courier choices may work either for an entire interface, or any particular operation. A courier may even decide not to forward a particular call. Listing 3.10 adds a courier to the proxy.

```
// The name of a courier matches its input port
courier Self {
    // Courier choices can match an entire interface
    [interface ICalculator(request)(response) {
        println@Console("Received call for calculator!")();
        forward(request)(response)
    }]

    // Or just a particular operation
    [sum(request)(response)] {
        // The courier may choose to forward a request, or answer the
        // request itself
        if (#request.numbers > 2) { forward(request)(response) }
        else { response = request.numbers[0] + request.numbers[1] }
    }
}
```

Listing 3.10: A courier allows code to run alongside a potential forwarding of operations

Additionally, Jolie supports "interface extenders". These can, like the name suggests, extend the types of operations with additional fields. These may add additional fields for every operation, or specific operations. These can also add faults to the type signature of an operation. The extensions to types are automatically striped before forwarding the message.

Like with any other output port, the service relies entirely on the interface listed in the output port to be correct. There is no communication with the target service about the "correct" interface. If the interface doesn't match, it will simply fail when attempting to invoke the operation.

A consequence of this is that it isn't possible to create entirely generic proxies without knowing the interface.

The aggregate and courier features, allow Jolie to implement many different common proxy-like patterns. These patterns mostly deal with their target service in a generic fashion, making very few, if any, assumptions about the target service. However, because the aggregation feature needs the interface it would be impossible to write a fully generic proxy service.

The interface rebinding feature fixes this problem, by allowing configuration files to redefine an interface at deployment time. This way, the generic service may write ordinary code, making no assumptions about the underlying service, and only at deployment time will it know which interface the target service has.

Creating a configurable interface is done by leaving the body empty, as shown in Listing 3.11.

```
interface ITarget
```

Listing 3.11: Configurable interfaces are defined by leaving the body empty

The interface is bound from the configuration file, in a similar fashion to other configurable constructs, as shown in Listing 3.12.

```
interface ITarget = ICalculator from "calculator"
```

Listing 3.12: Rebinding the `ITarget` interface to the `ICalculator` interface from the calculator module

3.2.5 Syntax of the Configuration Format (COL)

In this section the formal grammar of the configuration format is presented. The grammar is presented in a ABNF-like syntax. Complete syntax is presented in Appendix 2.

All literals are case sensitive.

A configuration file starts by a list of includes, followed by a list of config-units, as shown in start rule `configuration-tree`.

```
configuration-tree = *include *config-unit

include = "include" qstring

config-unit = config-unit-header "{" config-unit-body "}"
config-unit-header = [ "profile" qstring ] "configures" qstring
                [ "extends" qstring ]
config-unit-body = *definition
definition = port | interface | parameter
```

The definitions inside of a config-unit correspond to the different configurable units. The syntax is made to closely mimic the syntax of existing Jolie code.

```
port = input-port | output-port
input-port = "inputPort" identifier "{" port-body "}"
output-port = embedded-output-port | external-output-port
external-output-port = "outputPort" identifier "{" port-body "}"
embedded-output-port = "outputPort" identifier "embeds" qstring
                       "with" qstring
port-body = *port-property
port-property = location-property | protocol-property
location-property = "Location" ":" qstring
protocol-property = "Protocol" ":" identifier [ protocol-config ]
protocol-config = inline-tree

interface = "interface" identifier "=" identifier "from" qstring

parameter = variable-path "=" value
variable-path = var-id *var-node
var-id = ( "(" qstring ")" ) | identifier
var-node = "." var-id [ "[" unsigned-int "]"  ]
value = ( primitive [ inline-tree ] ) | inline-tree
inline-tree = "{" *(tree-child ",") tree-child "}"
tree-child = "." variable-path "=" value
primitive = qstring | int | long | double | bool
```

The values very closely resemble what Jolie allows, primarily just removing variable references from the syntax. Variable identifiers can be strings, like in Jolie, to allow for special variable names which would otherwise not be valid identifiers. This is primarily useful for building dictionary like structures with arbitrary string keys.

Additional configuration file supports C-style comments, these are allowed anywhere and are simply ignored.

```
comment = single-line-comment | multi-line-comment
single-line-comment = "//" <any text except line breaks>
multi-line-comment = "/*" <any text except "*/"> "*/"
```

3.3 Implementing the Configuration Format

The new interpreter pipeline is as follows, new phases marked in bold text:

1. Intent parsing
2. Jolie code parsing
3. **Configurator**
4. Semantic verifier
5. Interpretation tree builder
6. **Late verification**
7. Communication core init
8. Init & main process

The primary work occurs in the configurator service, but some of the verification work, namely the parameters, are postponed until after the interpretation tree has been build.

3.3.1 The Configuration Phase

The configuration phase builds upon the result of the intent and parsing phase. From the intent phase, we receive information about modules and the selected configuration unit. From the parsing phase we, obviously, get the AST.

The configuration phase works by modifying the AST to insert the configuration gathered from the selected configuration unit. The processed AST is handed to the later phases, and the remainder of the pipeline works mostly like it has before.

Processing of Configuration Files

Resolving configuration files The first step of the configurator is to gather a fully processed configuration unit. This process starts by looking for the configuration file which was passed from the intent. The selected configuration unit is passed with: `--conf <unitName> <filePath>`.

If the file path is an absolute file path, then that file will always be used. However if the file path is relative some searching is required. The base case for resolving this is to look for the configuration file relative to the current working directory. However this approach stops working once modules starts embedding other modules in a static fashion.

Consider three modules `A` and `B`, `C`. Module `A` depends on `B`, and `B` depends on `C`.

```
/* /a/main.ol */
outputPort B { Interfaces: BInterface }

/* /a/config.col */
profile "a-deployment" configures "a" {
    outputPort B embeds "b" with "b-deployment"
}

profile "b-deployment" configures "b" { /* ... */ }

/* /b/main.ol */
outputPort C { Interfaces: CInterface }

embedded {
    Jolie:
        "--conf c-deployment embedded.col c.mod"
}

/* /b/embedded.col */
profile "c-deployment" configures "c" { /* ... */ }
```

Listing 3.13: Relevant source code for modules `A` and `B`

Relevant code is shown in Listing 3.13. Assume that we start module `A` with `--conf a-deployment config.col a.mod`. The problem arises because the fixed embedding of module `C` performed in `B`. Because the files are resolved relative to working directory in which the engine was started, the engine would look for the configuration file in module `A`. To fix this, every time an embedding is performed the engine will implicitly add a `--parent` flag. This flag informs the new interpreter who performed the embedding. If the parent flag has been given, the configuration file will be found relative to that of the parent.

The working directory, throughout this, is `/a/`. The following will happen:

1. Start engine (from CLI) with: `--conf a-deployment config.col a.mod` File resolved to `/a/config.col` because of relative path and no parent.
2. Configuration causes embedding of `b-deployment` with:
   `--parent a --conf b-deployment /a/config.col b.mod`. File resolved to `/a/config.col` because of absolute path.
3. Source embedding in `B` causes embedding of `c-deployment` with:
   `--parent b --conf c-deployment embedded.col c.mod`. File resolved to `/b/embedded.col` because of relative path with parent.

Parsing of configuration files The resolved configuration file is passed to the configuration parser along with known modules. The parser is written in the same style as the Jolie code parser. The parser is a hand-written recursive descent parser with one token lookahead.

The parser directly outputs the AST. At the top level the AST consists of `ConfigUnits` which directly map to the configuration unit abstraction. Each `ConfigUnit` are namespaced under the module that they configure. Internally each namespace maps the name of the `ConfigUnit` to the actual `ConfigUnit`. The top-level node of the AST contains a dictionary mapping between the name of modules and their namespaces. The Java type for this mapping ends up being `Map<String, Map<String, ConfigUnit>>`, this allows for easy access of `ConfigUnits` when they are needed. This does however also mean that duplicates must be caught during parsing, since they would otherwise just silently override previous entries. This would typically have been performed in a later stage to keep the parser "pure". In this case some purity was sacrificed for some efficiency and ease of use.

The grammar of the configuration format closely follows the grammar of Jolie code. As a result the configuration format also re-uses AST nodes where possible. As we will see later, this proved helpful when outputting a configured AST. The configuration grammar only supports a subset of the features that Jolie code does.

Recall from Section 3.2.1 that each module may publish default configuration units, which can be used by user-level configuration units. For this reason the parser will make special note of references to other configuration units. Such references currently only appear in output port embeddings, e.g.,
`outputPort Foo embeds "foo" with "a-foo-unit"`.

Once the initial file has been parsed, every single configuration file placed in the "conf" directory of referenced modules are parsed and placed in the same AST. Configuration files parsed during this phase may itself have references to other modules, in that case these will be pulled in as well. A set of parsed defaults is kept to avoid parsing the same file multiple times, or potentially reach infinite loops.

Merging configuration files With the entire configuration tree in place, merging is performed to create a single `ConfigUnit` which represents the actual configuration of this module. The merging occurs between a `child` and its `parent` unit (that is, the one it extends, if any) and results in a `merged` unit. The `parent` is always merged, via a recursive call, first. The first `child` selected is the `ConfigUnit` matching the profile name and package given by the intent.

For ports, the merging algorithm works by inserting all the ports of the `child` into the `merged` unit. The ports of the `parent` are then merged with the preliminary output of the `merged` unit. The process of merging a `port` of the `parent` with the `merged` unit is shown, in pseudo code, in Listing 3.14. In short the merging will favor the `child`, and the `parent` will only provide values if the `child` doesn't.

```
procedure merge(port: Port, merged: ConfigTree): Port {
    if (!merged.contains(port)) return port
    child = merged.getExisting(port)

    if (child.isComplete) return child
    // The child cannot be an embedding, since those cannot
    // be partial.

    for (property in Port.properties) {
        if (child[property] == null) {
            child[property] = merged[property]
        }
    }
    return child
}
```

Listing 3.14: Pseudo code for merging a `parent`'s `port` into a `merged` configuration tree

For parameters the same idea applies. However each parameter definition node in the AST potentially only provides a partial value. For example the node `foo.bar = 42`, where the type of `foo` is `string { .bar: int }`, only provides a partial definition of `foo`. For this reason the parameter assignments are first added from the `parent` and then by the `child`. This allows the definitions of the `child` to override those of the `parent`.

Applying Configuration to the AST

The `merged` unit from the previous section is then used along with the AST corresponding to the Jolie program. The last phase of the configurator will process the entire AST and output a new, fully configured, AST. This process works by running a `process` function on every single top-level node. The `process` function will inspect the node and return a list of replacement nodes.

Throughout this process the configurator will also verify that no configuration is performed on nodes which aren't configurable. This includes dynamic ports and non-empty interface definitions.

Ports When a port node is encountered in the AST, the `merged` unit is searched for an output port matching its name. If none is found, the node is returned as it is. Otherwise the configuration proceeds. For input ports and non-embedding output ports the process is the same. The properties of the AST port and the port represented in the configuration unit are compared one-by-one. If any property is defined by both, an error is returned pointing to the AST node and the place that the configuration took place. At the end, if no errors are returned, a new port is returned with their properties merged together.

For output ports which embed another module, the process is slightly different. The configurator will first verify the existence of the configuration unit by looking it up in the configuration tree. If it is not found an error will be returned. Otherwise the configurator will output a new embedding node. The embedding node will contain a reference back to the same configuration file the current `ConfigUnit` comes from. When the embedding is performed, the configurator of the embedded service will lookup the correct configuration unit from the same file. This will cause an addition parsing of the same file, but a caching layer at the configuration parser could take care of this. The embedding process is demonstrated in Listing 3.15.

```
embedded {
    Jolie:
        "--conf foo-profile
         ↪  <inputConfigFile> foo.mod" in
         ↪  Foo
}
```

```
outputPort Foo embeds "foo" with
 ↪  "foo-profile"
```

Listing 3.15: AST nodes generated, shown as code (left), by configuration (right) for an output port containing an embedding

Interface Rebinding Just like ports, when an interface definition is found, the `merged` unit is searched for a matching interface definition. Assuming no errors, the configurator must now replace the dummy interface definition with the real interface definition, as the configuration specifies.

Quickly recapping interface rebinding in the configuration is shown in Listing 3.16. Note that the only information the configurator has for locating the replacement is the name of the interface and its module.

```
interface Dummy = Concrete from "my-module"
```

Listing 3.16: Rebinding the interface `Dummy` to match the interface `Concrete` from the module `my-module`

The interface will be found by parsing the module, starting at its entry point. Its AST, if it exists, will then be searched for an interface matching the replacement. Simply replacing the dummy with its replacement, however, isn't enough. Consider, for example, the scenario shown in Listing 3.17. In this scenario, the processed AST will no longer be valid due to the missing types (`FooRequest` and `FooResponse`).

```
type FooRequest: void {
    .child: string
}

type FooResponse: int

interface Concrete {
    RequestResponse:
        foo(FooRequest)(FooResponse)
}
```

Listing 3.17: Simply copying the interface definition is not enough, the types must also be copied

The relevant type definitions are found by iterating through every operation of the replacement interface. If the types listed in the operations are type links those are followed. This leads to a recursive algorithm, which looks through every type looking for type links, which will cause more types to be copied over. In order to avoid infinite loops a set of already visited types is kept. If a type has been seen before then its children will not be visited.

In the end, the original interface definition will be replaced, and all necessary type definitions will also be added to the program.

Parameters Parameters being a new concept in Jolie simply outputs new nodes for every single parameter assignment. This along with the definitions gathered from ordinary Jolie code parsing are processed in the interpretation tree building phase.

3.3.2 Additions to the Semantic Verifier

As we have seen, several syntax changes were also made to the core languages. These modifications to the syntax changed which programs were legal. As an example, interfaces are now allowed to be empty at a syntactical level, to signal that they should be replaced by configuration. However having an empty interface not being replaced should still cause an error.

This means that additional checking must now be performed, since it no longer will be caugt during parsing, to ensure that programs are semantically valid. Such a phase already exists in the compiler, the semantic verifier.

The semantic verifier works by performing an AST traversal and validate the semantics of each node.

The verifier has been updated to not allow empty interfaces, this was previously enforced at a syntax level. Since the configurator runs before this phase and updates the AST, this will only cause errors for non-configured trees.

Non-dynamic ports are ensured to be constant by using features already implemented by the semantic verifier.

The semantic verifier is the last phase of the Jolie interpreter when run with the `--check` flag. As we will discuss in the coming section, it is not practical to perform type-checking of parameters at this point. As a result, it will not be checked using the `--check` flag.

3.3.3 Additions to the Interpretation Tree Builder and Late Verification

Interpretation tree building was expanded to support parameters. All other features required no changes to AST structure, and could simply reuse existing features.

This section will discuss the technical details of how the Interpretation Tree Builder (ITB) works. It will cover issues that are specific to this particular implementation of the Jolie engine.

The Interpretation Tree Builder (ITB) is the last step before a Jolie program is ready to be executed. The ITB works by traversing the AST. The ITB will collect information from the AST in the form of definitions and inform the AST about their presence. These definitions include ports, interfaces, types, and code procedures.

Within code procedures the ITB will produce "processes" from the AST nodes which are runnable. It is from processes the meat of code execution is implemented. For example, it is from processes everything from addition to operation calls are implemented.

In this phase, the ITB will be gathering complete information about types. Type building is, for the most part, a one-to-one translation from the AST nodes into the internal representation. An important job of the ITB with respect to type building is resolving of type references. This turns references to a name into an actual type reference. Because types aren't built until this phase is what makes type-checking during semantic verification problematic. The semantic verifier would essentially have to duplicate the work of the ITB in order to perform type-checking earlier. This also means that type-checking cannot be performed until the ITB is finished.

The ITB will need to initialize the system with certain values. For example, the ITB will initialize the `location` and `protocol` variables of output ports. These variables are local only to a single request. At this point, however, the state of the Jolie program has not yet been initialized.

In Jolie, state is associated with the current thread of execution. Jolie uses a separate thread of execution for handling every message. As a result every single request gets its own fresh state. All state is copied from the initialization execution thread. This thread performs work requested by the ITB followed by the `init` procedure. Thus right after the ITB is done, and the initialization execution thread is running, code execution starts. This includes both user code and loading of any embedded services.

It should be noted that any process which works with state must be run on an execution thread, since it depends on the state being present on the thread.

The ITB receives two new node types from the configurator and code parser: parameter assignments, and parameter definitions.

For parameter definitions, the ITB will resolve the types. Ensuring that all links are properly resolved. This will output a "processed" parameter definition which the ITB will inform the interpreter about.

In a similar fashion, the ITB will process parameter assignments and inform the interpreter. Type-checking cannot be performed at this point, due to types not being ready before the ITB finishes. Simply outputting processes for configuration will make it impossible to perform type-checking before code execution starts, due to their dependency on state.

The processed assignments contain an ordinarily processed LHS (of the assignment), which contains the variable path. The RHS has been processed into an expression, which can be evaluated into a native Jolie value on which type-checking can be performed.

Because of these constraints a new "late-checking phase" has been created. This phase runs right after the ITB, but before code execution starts. In this phase, it is not possible to directly touch the init state. It is however possible to both create values, and evaluate expression. This phase creates a synthetic state by creating a new value which will act as the state. Expression are evaluated against this root and later type-checked. The result of evaluating the parameter assignments are then copied into the init state, assuming, of course, that type-checking was successful.

3.4 Examples

In this section, we will introduce two examples which will showcase different aspects of the Jolie module and configuration system. These examples will be revisited again in Section 5.8 where they will be shown in the context of packages.

3.4.1 A Calculator System

In this example, we will look at the calculator example shown in the introductory chapters. The calculator system will be updated to work under the new Jolie module system. We will also show how what was previous problems that required code changes, can now be performed entirely in independent configuration files.

Quickly recapping from previous chapters. Figure 3.2 shows the system architecture for the calculator system. The system contains three services. The main point-of-contact for a client is the `calculator` service. This service will delegate the "hard" work to the other two services, that is `addition` and `multiplication`.

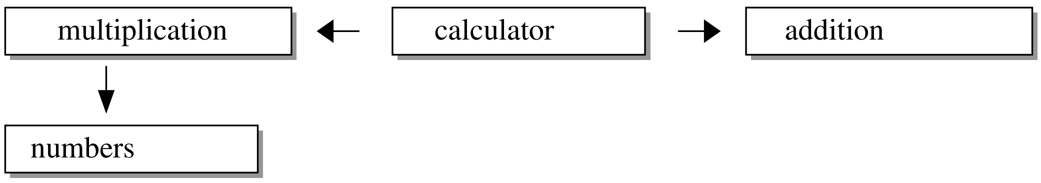

Figure 3.2: System architecture for the "calculator system"

The file structure of our services contained a lot of duplication previously. The file structure is shown in Listing 3.18. As we can see from this there is quite a bit of duplication of files, and we cannot simply copy the directories of our dependencies, as this would place the files in the wrong places. We also had to resort to code files for distributing default configuration profiles. Switching between embedding a service and binding to it externally would also require significant changes.

```
// multiplication
.
|-- multiplication.ol
|-- include
|   |-- mult_config.iol
|   `-- multiplication.iol
`-- numbers.iol

// numbers
.
|-- include
|   `-- numbers.iol
`-- main.ol
```

```
// calculator
.
|-- addition_config.iol
|-- addition.iol
|-- addition.ol
|-- calculator.ol
|-- include
|   `-- calculator.iol
|-- mult_config.iol
|-- multiplication.iol
|-- multiplication.ol
|-- numbers.iol
`-- numbers.ol
```

```
// addition
.
|-- include
|   |-- addition_config.iol
|   `-- addition.iol
`-- addition.ol
```

Listing 3.18: Initial file structure of the calculator system

The first step is to update all includes which require files from other modules, to use the new module includes. For example, `multiplication` depends on the file `numbers.iol` from `numbers`. Their includes change from the old `include "numbers.iol"` to the new module include
`include "numbers.iol" from "numbers"`.

Using module includes provides a certain level of namespacing for file names. This system will allow for two files to be named the same thing, without causing problems, as long as they exist in separate packages. This allows us to give files better names, and allow many rather useless prefixes to be deleted. For example, the file `addition.iol` can be renamed to a more meaningful name such as `interface.iol`. Causing includes to say `include "interface.iol" from "addition"`.

The next step is to convert the legacy configuration into the new configuration format. The contents `mult_config.iol` previously contained constants for configuring the input port and parameters of the service. The legacy configuration is shown in Listing 3.19.

In order to convert ports, we must first decide which ports need to be configured, and which fields should be configurable. In this case, we should only configure our input port. Since the module makes no assumptions about this service's input port, we can simply allow it all to be configurable.

For parameters we must determine the actual type each parameter can take. This must be put in the `parameters` block. The complete transformation is shown in Listing 3.20.

```
constants {
    MULT_INPUT_LOCATION = "socket://localhost:12345",
    MULT_INPUT_PROTOCOL = sodep,
    MULT_MAX_INPUT = 1000,
    MULT_MIN_INPUT = 0,
    MULT_DEBUG = false
}
```

Listing 3.19: Legacy configuration file for the `multiplication` module

```
inputPort Multiplication {
    Interfaces: IMultiplication
}

parameters {
    minInput: int,
    maxInput: int,
    debug: bool
}
```

Listing 3.20: Updated configuration for the `multiplication` module

We're not quite finished yet with the configuration. The original service provided default configuration. A new directory named `conf` is created. Inside which we can make as many defaults as we wish. For now, we create a file named `default.col`. In Listing 3.21 we show the default configuration. The inheritance feature allows us to easily create profiles for various deployment configurations which a user of the module might wish to use. The `"self-hosted"` profile would be ideal for hosting the service as standalone, while the `"embedded"` and `"debug"` profiles would be ideal for development.

```
profile "default" configures "multiplication" {
    minInput = 0,
    maxInput = 1000,
    debug = false
}

profile "self-hosted" configures "multiplication" extends "default" {
    inputPort Multiplication {
        Location: "socket://localhost:12345"
        Protocol: sodep
    }

    outputPort Numbers {
        Location: "socket://numbers.example.com:22000"
        Protocol: sodep
    }
}

profile "embedded" configures "multiplication" extends "default" {
    inputPort Multiplication {
        Location: "local"
        Protocol: sodep
    },

    outputPort Numbers embeds "numbers" with "default"
}

profile "debug" configures "multiplication" extends "embedded" {
    debug = true
}
```

Listing 3.21: Providing defaults for a Jolie module. Using inheritance we can create suitable profiles for typical deployment configurations.

At this point we can now clean-up in the directories of each module. The Jolie module system technically allows for any organization of modules, but for now we will simply put the source code of each module inside a directory called `modules`. In this directory we will place a directory named the same as the module, containing the complete source code of the modules.

After doing this, and renaming files to give them more meaningful names, we end up with the file structure shown in listing 3.22.

```
// multiplication
.
|-- conf
|   `-- default.col
|-- include
|   `-- interface.iol
|-- main.ol
`-- modules
    `-- numbers
```

```
// calculator
.
|-- include
|   `-- interface.iol
|-- main.ol
`-- modules
    |-- addition
    |-- multiplication
    `-- numbers
```

```
// addition
.
|-- conf
|   `-- default.col
|-- include
|   `-- interface.iol
`-- main.ol
```

```
// numbers
.
|-- include
|   `-- interface.iol
`-- main.ol
```

Listing 3.22: Final file structure of the three services for the calculator system

This file structure, as we shall also discuss later, is much better for potential package managers. Launching the service now requires a longer command. Currently in order to launch this program the following would be needed:

```
dan@host:/calculator # jolie                                     \
    --mod addition,modules/addition,main.ol                      \
    --mod multiplication,modules/multiplication,main.ol \
    --mod numbers,modules/numbers,main.ol                        \
    --mod calculator,.,main.ol                                   \
    calculator.mod
```

In contrast, this is quite a bit more than previously:

```
dan@host:/calculator # jolie calculator.ol
```

However, most of this disappears when using a package manager or similar tool. In fact it will be as simple as:

```
dan@host:/calculator # jpm start
```

This is also similar to how, for example, the Java ecosystem ordinarily does its additions of libraries. From the CLI this will require manually listing every JAR dependency. However such projects are usually managed through more advanced build tools, such as: Maven, Gradle or Ant.

The configuration system also helps significantly with changing between various different deployment configuration, which switch between embedding services and having them be external.

Before the difference between embedding a service, and not embedding a service would require drastically different work. The first step here would be to make sure that all files were included in the directory. Since there were no module system, these files would have to live in the same directory as all the other code, or alternatively rewrite all includes in the dependency. Secondly it would require changing all relevant configuration, and inserting an embedded section inside of our source code.

With the module system this will require absolutely no code changes. The calculator module would simply have to define the output ports for `addition` and `multiplication` as configurable. This is shown in Listing 3.23.

```
outputPort Addition         { Interfaces: IAddition         }
outputPort Multiplication   { Interfaces: IMultiplication   }
```

Listing 3.23: Definitions of the output ports for `addition` and `multiplication`

Creating configurations for embedding these and for externally binding is trivial as shown in Listing 3.24. No changes to the existing code is required at all. These profiles can safely use the `"debug"` profiles created in the default configuration files of both `addition` and `multiplication` since those are included by default.

```
profile "calculator-with-embedding" configures "calculator" {
    outputPort Addition embeds "addition" with "debug"
    outputPort Multiplication embeds "multiplication" with "debug"
}

profile "calculator-with-mixed" configures "calculator" {
    outputPort Addition embeds "addition" with "debug"
    outputPort Multiplication {
        Location: "socket://mult.example.com:12345"
        Protocol: sodep
    }
}
```

Listing 3.24: Creating vastly different deployments no longer require changes to the code. They can instead be solved entirely from configuration files

Jolie Packages

4.1 Introduction

A Jolie Package is an extension of a Jolie Module. Recall that a Jolie Module was defined as a directory root, 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 read and write for both humans and machines.

In listing 4.1 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 1. The service that this manifest describes is shown in Figure 3.2.

```
1  {
2      "name": "calculator",
3      "main": "main.ol",
4      "description": "A simple calculator service",
5      "authors": ["Dan Sebastian Thrane <dathr12@student.sdu.dk>"],
6      "license" "MIT",
7      "version": "1.0.0",
8      "dependencies": [
9          { "name": "addition", "version": "1.2.X" },
10         { "name": "multiplication", "version": "2.1.0" }
11         { "name": "numbers", "version": "1.0.0" }
12     ]
13 }
```

Listing 4.1: A simple package manifest

This manifest configures the ongoing example of the calculator system. Lines 2-3 take care of the module definition, the root is implicitly decided by the location of the manifest. 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 version we will cover shortly, but it is used for package management. The rest of the manifest describes the dependencies of this package.

4.2 Architecture

Before continuing with the packages we'll take a quick de-tour to look at the overall architecture of JPM. This should help the reader understand in which context the packages exist. The details of this system will be covered in Chapter 5.

The services 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. Some library services in the ecosystem are written in Java, using the powerful embedding features of Jolie, which allow certain foreign languages to be embedded as services.

At the ten-thousand foot view of the architecture, it consists of three core services, as shown in figure 4.1:

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.

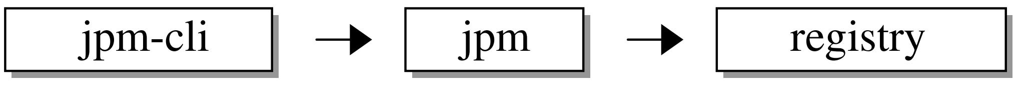

Figure 4.1: Ten-thousand foot view of the JPM architecture

4.3 Package Manifest

In this section we'll cover, in details, the format of Jolie manifests. The complete specification for the package manifest can be found in Appendix 1.

4.3.1 The `name` Attribute

The `name` attribute uniquely identifies a package. We'll start this discussion by looking at the constraints imposed on it, from Appendix 1:

1. The name of a package is *not* case-sensitive
2. The length of a name is less than 255 characters
3. Names may only contain unreserved URI characters (Section 2.3 of [RFC3986])
4. Names are US-ASCII
5. Every registry must only contains a single package with a given name.

Points 1 through 4 all contribute towards a common goal: packages should be displayable in URLs. This will allow us to more easily create, for example, a web application where a developer can browse through packages.

Putting some constraints on the package name is also helpful in multiple places of the package manager. Consider for example a system that might store packages organized by their name (say for example a caching mechanism). If one package would be named "../foo", then that might accidentally override the contents of the package named "foo". Of course these name constraints are not a general solution for these types of injection attacks, but it does allow for several services working with packages to not worry as badly about the package names.

It should also be noted that the package names are not compatible with the rules of identifiers. This has affected the design of the Jolie module system. This is, for example, the reason that strings are used in both configuration files and module includes, when making references to other modules.

Rule 5 states that package names are unique within a single registry.

4.3.2 The Meta Data Attributes (`license, authors, description`)

These attributes are mostly used for search results. They allow for a developer seeking information about a package, to quickly gather the most important information.

License identifiers are validated against a pre-existing list of license identifiers. This is done to ensure that typos are not made in license names. The identifiers used are from the SPDX license list[SPDX], which contains a list of commonly found licenses. Currently only the license identifiers are supported, this could be extended to use SPDX License Expression Syntax, which allows for references to custom licenses.

4.3.3 The `version` Attribute

The `version` attribute used by packages declare which version of the package the manifest describes. This field uses the Semantic Versioning (SemVer)[SEMV] specification for version numbers. It is a requirement to use SemVer versions, if a version is specified, which is not a valid SemVer version, then validation of the manifest will fail. Enforcing this standard allows for several other features, which will be covered in a moment.

SemVer version numbers consists of three fields, all integers, those being the *major*, *minor*, and *patch* version numbers. Additionally a version may have a pre-release label or additional meta data. Labels and build meta-data won't be covered, as they are not that important for our use-case. The (simplified) syntax of SemVer version number is shown in Listing 4.2.

```
<major>.<minor>.<patch>[-<label>][+<meta-data>]
```

Listing 4.2: Simplified syntax of a SemVer version number

Precedence of version numbers are determined by comparing each unit numerically in the order of major, minor, and finally patch. The first differentiating units determine the precedence. Thus, for example: $1.0.0 < 1.0.1 < 1.1.0 < 1.10.0 < 2.0.0$.

Semantic Versioning dictates rules for how version numbers should change, depending on how the public API changes. The rules can be summarized as:

1. The patch version number *must* be incremented, if only backwards compatible bug fixes are introduced.
2. The minor version number *must* be incremented, if backwards compatible functionality is introduced.
3. The major version number *must* be incremented, if incompatible changes are introduced.

Having a major version of 0, indicates that the product is not yet stable. At this point normal rules for incrementing version numbers do not apply. As a result version 1.0.0 indicates the first public API.

Semantic Versioning has become a commonly used standard across many different package managers, examples include Cargo[CARA] and NPM[NPMA].

For dependencies the version field allows for a bit more flexibility using SemVer expressions, also known as SemVer ranges[NPMB]. These expressions allows the developer more freedom in which version to use. In short the expressions allows the developer to both whitelist certain ranges for use, and blacklist certain ranges for use.

SemVer expressions allows for a wide range of different comparators, and range builder syntaxes. For more information see [NPMB]. A few examples are shown below, along with a description of what they mean:

- `1.2.3`: Exactly version `1.2.3`
- `1.0.X`: Major must be 1, minor must be 0, patch can be anything
- `1.0.*`: Major must be 1, minor must be 0, patch can be anything
- `>1.0.0`: Any version larger than version `1.0.0`
- `1.0.0 - 1.2.3`: Any version between `1.0.0` and `1.2.3` both inclusive

This provides both a convenience factor, i.e., the developer can express that she doesn't care which specific version is used, as long as it has feature X (released in some known version).

This feature also provides benefits for when the dependency tree is to be calculated. Consider, for example, two dependencies A and B which both depend on C with the same level of functionality, which was released in version 1.2.0. Assuming that expressions were not allowed, these two services would have to describe their dependency in an exact version. Service A might have been written at a time when 1.2.3 of service C was the newest, while B was written a bit later and is thus depending on version 1.2.6. The problem is now that the client depending on A, and B, cannot do so, because A and B have conflicting requirements, despite the fact that they both require the same level functionality. With SemVer expressions these services may more accurately describe what they actually depend on, service A might state that it depends on ">=1.2.0", while B is a bit more conservative and wants exactly version 1.2.6. In this case the package manager will be able to choose version 1.2.6, since this version fulfills the requirements of all dependencies.

Allowing for SemVer expressions to be used, is however not without its drawbacks. In Section 4.5 we cover one of these problems, and its solution.

4.3.4 The Lifetime Attributes (`events`, `main`)

These attributes control aspects of the package related to the execution of scripts, by the package manager, related to the package.

The `main` attribute controls the entry-point of the module. This is used both in informing the Jolie Engine where the entry-point is located. The attribute is also used for starting the package.

The `events` attribute defines a series of lifetime hooks, which are scripts that are run when specific lifetime events occur, this is described in Section 4.6.

4.3.5 The `registries` Attribute

The package manager may communicate with one or more registries, when performing its work. This can be to install dependencies, or it could be to perform some of the more organizational commands, such as creating an account.

For this the package manager will need to know about the different registries. This is what the `registries` attribute describes, it is an array containing definitions for all known registries (known to the package).

A registry is defined by a unique (within the package) name, and a location (a URI following the same rules that Jolie does for its port locations). Listing 4.3 shows an example. Note that the protocol is always fixed to `sodep`.

```
{
    "name": "my-registry",
    "location": "socket://registry.example.com:9999"
}
```

Listing 4.3: A registry named `my-registry` being hosted at `registry.example.com` running on port 9999

Every package manifest has an implicit entry which is named "public". This entry points to the public registry. Every command which needs to use a registry, will optionally receive a registry name. If this name is specified, it will look within the `registries` attribute. If no name is specified, the "public" entry will be used.

4.4 Dependencies

A JPM dependency describes a dependency on another JPM package. Since JPM packages are extensions of Jolie modules, what we're really describing is a code dependency.

There is an important, but subtle, distinction between the dependencies of a Jolie service and the dependencies listed in a JPM package manifest. For the sake of clarity, we will refer to the first one as service dependencies, and the latter as package dependencies.

Consider, the Jolie service shown in Listing 4.4. For this service we can define one explicit service dependency on the `Example` service and one implicit service dependency on the client calling the `Self` service.

However given this service, encapsulated in a package, it would have no package dependencies. For JPM a package dependency is a dependency on the code of another module.

In this example, however, we have no dependency on external code.

```
// service.ol
interface ExampleInterface { RequestResponse: example(undefined)(undefined) }

outputPort Example {
    // ...
    Interfaces: ExampleInterface
}

interface SelfInterface { RequestResponse: hello(undefined)(undefined) }

inputPort Self {
    // ...
    Interface: SelfInterface
}

main {
    [hello(request)(response) {
        // ...
        example@Example(exampleRequest)(exampleResponse);
        // ...
    }]
}
```

Listing 4.4: A simple Jolie service served defined in a single file (`service.ol`)

If we change the previous example, such that the interface of `Example` is defined in another package named `"example"`, then we could define a package dependency on the `"example"` package. This is shown in Listing 4.5. This service dependencies in this example remain unchanged.

```
/* /example/package.json */
{
    "name": "example",
    "version": "1.0.0"
    /* remaining fields omitted */
}

/* /self/package.json */
{
    "name": "self",
    "dependencies": [{ "name": "example", "version": "1.0.0" }]
    /* remaining fields omitted */
}
```

```
/* /example/interface.iol */
interface ExampleInterface { RequestResponse: example(undefined)(undefined) }

/* /self/main.ol */
include "interface.iol" from "example"

outputPort Example {
    // ...
    Interfaces: ExampleInterface
}

// input port and interface of Self omitted

main {
    [hello(request)(response) {
        // ...
        example@Example(exampleRequest)(exampleResponse);
        // ...
    }]
}
```

Listing 4.5: A package dependency is a dependency on the code of another package

A package dependency can be described by the 3-tuple $(name, version, registry)$, as covered the manifest will implicitly set the registry to `"public"` if not specified.

The dependency tree of a JPM package is build starting at the package itself. The algorithm for calculating the dependency tree is shown, in pseudo-jolie, in Listing 4.6.

Some details, like lockfiles and data marshalling, are left out. Error handling is illustrated by the `Error` procedure. In reality the error handling is slightly more advanced, but not relevant to our discussion.

```
define DependencyTree {
    // "manifest" is set at entry to this procedure
    dependencyStack << manifest.dependencies;
    currDependency -> dependencyStack[0];

    while (#dependencyStack > 0) {
        RegistrySetLocation; // Registry now points to origin of dependency

        // Lookup version information from the registry
        getPackageInfo@Registry(currDependency.name)(packageListing);
        if (#packageListing.results == 0) { Error };

        resolved -> resolvedDependencies.(currDependency.name);
        if (is_defined(resolved)) {
            // Check if 'resolved' matches requirement of 'currDependency'
            satisfies@SemVer({
                .version = resolved.version, .range = currDependency.version
            })(versionSatisfied);
            if (!versionSatisfied) { Error };
        } else {
            allVersions -> /* extract versions from packageListing.results */;

            // Find the best match for our version expression
            sortRequest.versions -> allVersions;
            sortRequest.satisfying = currDependency.version;
            sort@SemVer(sortRequest)(sortedVersions);
            if (#sortedVersions.versions == 0) { Error };

            // Insert resolved dependency
            with (information) {
                .version << sortedVersions.versions
                    [#sortedVersions.versions - 1];
                .registryLocation = Registry.location;
                .registryName = registryName
            };
            resolved << information;

            // Insert dependencies of this dependency on the stack
            dependenciesRequest.packageName = name;
40              dependenciesRequest.version << resolved.version;
41              getDependencies@Registry(dependenciesRequest)(depsResponse);
42              for (i = 0, i < #depsResponse.dependencies, i++) {
43                  item << depsResponse.dependencies[i];
44                  dependencyStack[#dependencyStack] << item
45              }
46          }
47      }
48  }
```

Listing 4.6: Pseudo-Jolie code for calculating a dependency tree

The algorithm keeps two primary data-structures: the `dependencyStack` and the `resolved` versions. The algorithm starts by inserting all the listed dependencies from the `manifest` onto the `dependencyStack`, this is done in lines 2-5. The algorithm then enters the main loop (line 7).

This loop will pop a dependency off the stack and start the process of resolving the dependency to a known version. Recall that dependencies use SemVer expressions in their version field. We need to resolve this field to a particular version. In this process we will also confirm that the registry has knowledge of this version.

This process starts by asking for a complete version listing for a package matching the name of our dependency (lines 8-12). At this point we can be certain that the registry at least know of a package by that name. We then check if a previous dependency already has resolved this dependency. The `resolved` dependencies data structure is a dictionary, mapping a package name to a resolved version. There are two important takeaways from this decision:

1. **A package cannot have two (or more) identically named dependencies.** Even though it is technically possible to have two identically named dependencies, as long as they do not originate from the same registry, this is still disallowed. The reason for this lies in the module implementation, where modules are uniquely identified only by their name. As a result we would have to either identify all packages by their name and registry, or simply disallow two identically named dependencies. We chose the latter to simplify the use of packages in code.

2. **A package cannot depend on two (or more) different versions of a package**. This is similar to the first point, but made worse by the lack of any namespacing in Jolie. Because of this, if two packages were to provide any two constructs which clash in name, the Jolie engine would be unable to distinguish them.

If a dependency has not been resolved (lines 20-46) we will start the resolving phase. The package manager will *always* select the newest version available in the registry which matches the SemVer expression as defined by the dependency (lines 21-27). When the package has been resolved its dependencies are looked up (by contacting the registry, we don't have the actual manifest). These dependencies are then added onto the stack (lines 38-46).

If the dependency has already been resolved, we must check if this dependency works under these conditions, by matching its SemVer expression to the resolved dependency. If this is possible we can continue. *The package manager makes no additional attempts to resolve the constraints imposed by the manifests.*

These constraints mean that there plenty of dependency trees can be constructed, such that the greedy approach of selecting the newest version, will lead to conflicts. This approach was chosen for its simplicity. More advanced approaches were not explored. The greedy approach will most likely also be significantly more efficient. At least one other package manager was also found to be using this exact same approach to constraint solving[YARNA].

4.5 Lock Files

Lock files are a feature designed to solve some of the drawbacks associated with allowing SemVer expressions in the version field of dependencies. To see how SemVer expressions pose a problem, we must first inspect the typical engineering process in which it is used. This feature is by no means original, prior work includes: Cargo[CARA], and Yarn[YARNB].

Continuous integration (CI) is a software development process[Fow06]. This process is used to avoid the so called "integration hell", in which the time taken to integrate, or merge, the changes from multiple developers into a single track, takes longer than developing the individual features. The process works by, multiple times a day, integrating all these changes, this way failure in integration is found much earlier. We'll quickly summarize one possible implementation of CI here. This is not the only way to do it, but summarizes the most important parts, which we'll use to highlight potential problems with SemVer expressions.

For this process to work efficiently it frequently uses automated tests and builds, in order to perform this integration. The automated tests and builds are then performed by a dedicated server. The server will most likely pull the source code from a single repository, which represents the most up-to-date track of development.

When new features are developed, the developer will pull the most up-to-date source code, and develop the feature locally with this. This includes performing all of the builds and tests. Thus when the feature is pushed back onto the source control all of these tests should still pass. The role of the CI server is then to verify that the process is upheld.

How the build and testing is performed obviously depend on which technologies are used. A typical JPM based project might perform the following steps:

1. Pull source code from version control
2. Perform the build
   (a) Download JPM dependencies, i.e., `jpm install`
   (b) Potentially build other necessary artifacts (such as Java libraries)
3. Run tests

It is in step 2(a) the need for lock files arises, the problem is most easily demonstrated through an example.

Consider the following scenario: a developer wishes to develop a new feature which uses package X of at least version 2.1.0. For this reason the developer adds the following line to the dependencies section: `{ "name": "X", "version": ">=2.1.0" }`. Afterwards the developer performs an install of this package, at the moment this dependency to version 2.1.3. At this point the developer continuous with developing the feature, without having to perform any additional installs of the package. Once the feature is developed, the developer performs all of the tests, and see that they pass.

At this point the code is pushed onto the source control system, where it is picked up by the CI server for testing. The CI server will then perform all of the steps listed above, including installing the dependencies anew. However, since the initial install by the developer had been performed a new version has been released, say version 2.3.0. This version still fulfills the expression listed in the dependency, and is hence chosen, since this is the best fitting package for this dependency. Remember, the CI server has no way of knowing which package was originally installed, it only has the package manifest to work with.

Installing a different version of the package may however lead to the tests no longer working! As a result, we cannot guarantee what works on a developer's machine, necessarily works on any other machines that might try to run it. As a result, some might choose to never use SemVer expressions to avoid this pitfall.

Lock files offer a compromise between these two extremes. A lock file contains the exact versions that every dependency was resolved to. The lock file should be put into source control, thus when another machine attempts to perform the build the package manager will know which exact version should be used for the build.

The convenience factor of SemVer expressions is present. When we perform the initial install, we might not care so which version to use, and the package manager will still be able to just pick the best version for us. Additionally the package manager can provide upgrade scripts, which uses these expressions as guidelines.

The lock files is placed in the package directory, and is called `jpm_lock.json`. Listing 4.7 shows an example of a lock file, which would have been generated for the scenario above.

The lock files of a dependency is *not* used by a client package. As a result library developers should still be careful not specifying too wide expressions. This feature is only intended to guarantee that a package will have the same dependencies regardless of which machine it runs on. It is not intended as a replacement for specifying exact versions. If a package requires a specific version, then that should be reflected in the package manifest.

```json
{
    "_note": "Auto-generated file notice",
    "locked": {
        "X@>=2.1.0/RegistryName": {
            "resolved": "2.1.3",
            "checksum": "..."
        }
    }
}
```

Listing 4.7: A lock file showing that the dependency for package X of at least version 2.1.0 has been resolved to version 2.1.3

4.6 Lifetime Hooks

A lifetime hook is a script that "hooks" onto certain lifetime events that occur during normal use of the package manager. These scripts allows the package developer to augment the package manager, and potentially affect the work it performs.

This is a feature which is available in many other types of software, for example the popular version control system Git provides such a feature[Cha09].

All hooks are performed on the client-side, and never on the server side, i.e., on registries. Only the hooks of the current package will be run, the hooks of a dependency will not be executed. Hooks that run before a certain action, denoted by a `pre-` prefix, may stop the corresponding command from being executed. These scripts may do so by returning a non-zero exit code.

These features become useful for augmenting the workflow with in-house conventions. This may for example include ensuring that, for example, tests pass before publishing a new version. Using a simple process interface allows for the user to write these scripts in whichever technology they choose.

The hooks are defined in the "events" section of the package manifest. Listing 4.8 shows an example hook, which runs before publishing which ensure that all tests pass.

```
{
    // ...
    "events": {
        "pre-publish": "./run_tests.sh"
    }
}
```

Listing 4.8: Defining a lifetime hook, which runs the script `run_tests.sh` before publishing the package to the registry

The following are the hooks that JPM supports:

- `pre-start`: Script is run right before the `start` command is invoked on a service.
- `post-start`: Run right before the `start` command terminates. There is no guarantee that this script is run, e.g., if the service was forcefully terminated.
- `pre-install`: Runs before the `install` command is invoked.
- `post-install`: Runs after the `install` command has finished.
- `pre-publish`: Runs before the `publish` command is invoked.
- `post-publish`: Runs after the `publish` command has finished.

4.7 The `.pkg` Format

The `.pkg` file format, is the format that JPM uses for distribution of packages. Simply put, the file format contains a Jolie package zipped up into a single file.

Currently the contents of the zip file maps directly with the underlying package. The format is however open to extension, by allowing for new files to be introduced that can be used in various extensions. An obvious implementation strategy would involve adding the necessary files directly into the ZIP archive.

Extending the ZIP file format is a quite common occurrence. Especially when shipping what is essentially a collection of files. Popular examples of other file formats using this approach includes the JAR[1][ORAA] format, and Office Open XML[2][OXML].

[1]Package file format typically used for aggregating many Java class files together
[2]Developed by Microsoft, used in their office applications

Package Manager

5.1 Introduction

The Jolie Package Manager is build to facilitate the use of packages, which were introduced in the previous chapter. The package manager provides a variety of tools, these can roughly be divided into three categories:

1. **Package management**
   - Installing packages (Section 5.6)
   - Publishing packages (Section 5.4)
   - Upgrading packages (Section 4.3.3 and 4.5)
   - Searching for packages (Section 5.5)
2. **Account management (Section 5.7)**
   - Login/logout with registries
   - User management
   - Team management
3. **Helper scripts**
   - Creating a new package (Section 5.8.1)
   - Starting a package (Section 5.8.1)
   - Interacting with the cache (Section 5.6)

These responsibilities are services by the package manager ecosystem which were briefly covered in Section 4.2. Many of the features this ecosystem provides were also covered in Chapter 4. In this chapter we will cover the final details of this ecosystem.

5.2 The Command Line Interface

The command line application serves as the user interface to JPM. The application is, perhaps not unsurprisingly, named `jpm`. The tool will be used in several examples.

The command line application is responsible for displaying a more user friendly interface to inner workings of JPM. The tool will perform almost no work by itself, but will instead delegate this to the back end (See Section 5.3).

When first running the tool, the user will be welcomed with the following message:

```
JPM - The Jolie Package Manager
Version 1.0.0

Usage: jpm <COMMAND> <COMMAND-ARGUMENTS>

Command specific help: jpm help <COMMAND>

Available commands:
-------------------
init           Initializes a repository
search         Searches repositories for a package
install        Install dependencies
publish        Publish this package
start          Start this package.

[ Remaining commands removed from snippet ]
```

As clearly visible form this snippet, for the tool to do any work we must first give it something to do via a command. The commands that JPM understands almost directly mirror the functionality provided by the back end. To use JPM to create a new package, the user must simply use the `init` command. This will display a prompt, guiding the user through the mandatory field, and automatically create a package with the required structure. This is shown in Listing 5.1.

```
$ jpm init
Package name
------------
> my-package

Package description
-------------------
> This is my package

Author: [Format: name <email> (homepage)]
----------------------------------------
> Dan Sebastian Thrane <dathr12@student.sdu.dk> (github.com/DanThrane)

Private package? [Y/n]
----------------------
> n

$ cat my-package/package.json | json name
my-package
```

Listing 5.1: The `jpm` tool provides a user interface for common tasks. In this example, creating a new package.

5.2.1 Internal Organization and Deployment

The command line tool delegates most of the work to the back end service. Figure 5.1 shows the architecture from the CLI's point of view.

Most notably is the callback server, this server is responsible for receiving information about events that occur in the back end. These are primarily used to communicate progress, this is especially useful for long running processes, such as downloading dependencies. The back end will in these cases these events to a callback server, which can then choose to display information about this event. The need for a separate service comes mostly from a limitation in Jolie. In Jolie all communication follows either a one-way, or request-response communication pattern. As a result of this, it isn't possible for JPM to send back information while a request is being processed. To handle this the front end (in this case `jpm-cli`) will inform the back end of where it should send events.

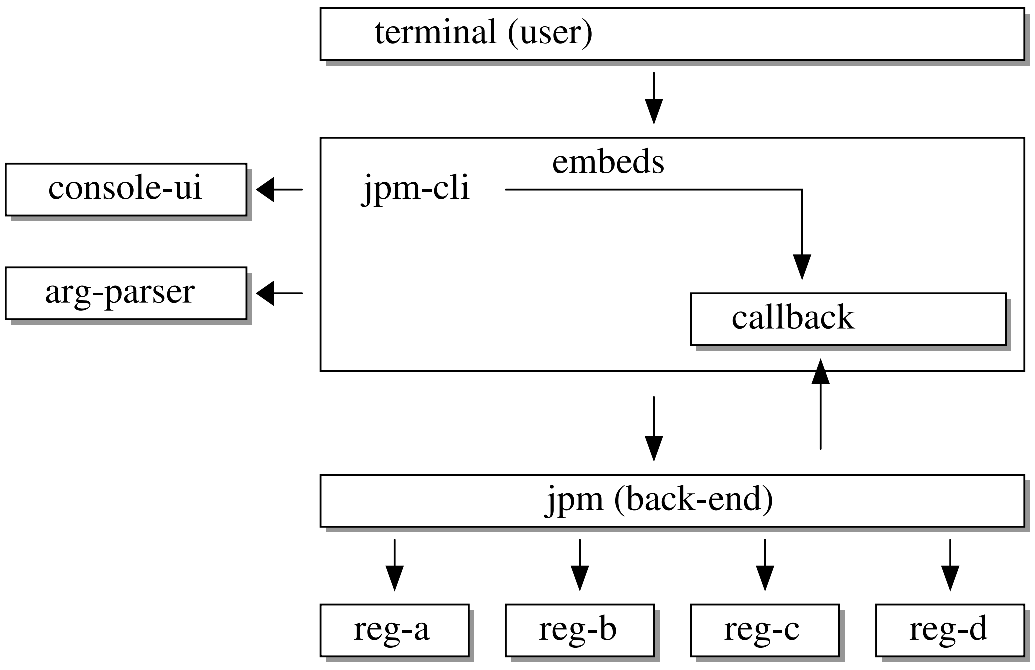

Figure 5.1: The system architecture from the CLI's point of view

Quite a lot of services exists on the client side, notably `jpm-cli` and `jpm`. From a usability perspective this is less optimal. For this reason the `jpm` binary, which ships with the package manager, will use a deployment which embeds all of the core services together. Future versions of the package manager, may wish to optionally spin up the required services, and run them as daemons[1]. This will remove quite a bit of the overhead associated with spinning up the JVM (required for the Jolie engine) and every embedded service. Such a practice is done by similar tools, as an example the JVM based build tool Gradle provides such a daemon[GRAA]. This conversion would be relatively straight forward, since the server architecture required by the daemon is already implemented.

5.3 The JPM Back End

The back end service is responsible for performing most of the heavy lifting, for the commands that are send to it from the front end. Some of the jobs are performed directly by this service, while quite a few others are delegated.

[1]A daemon is a background process running on a computer

The most important partner the back end service has is the registry service. The back end will communicate with one or more registry services.

The JPM back end collaborates with a few external services, these are illustrated in Table 5.2. The work performed by the back-end has mostly already been discussed or will be discussed (see Table).

Service Name	Responsibilities	Sections
`registry`	Responsible for storing packages	5.4
`execution`	Used for starting packages and executing life-time hooks	N/A
`lockfiles`	Handles the lockfiles feature	4.5
`semver`	Library package for semantic versioning	4.3.3
`jpm-downloader`	Acts as a cached downloader and package installer	5.6
`packages`	Validator for the package format	4.3
`pkg`	Creates the binary format used for distribution	4.7
`callback`	Callback service. JPM calls this about events	5.2

Table 5.2: Collaborators with the JPM backend

5.4 Registry

In this section we will briefly cover the responsibilities of JPM registries. For the most part we will cover the specific features it provides in other section and make references to them, when relevant.

The primary responsibility of a JPM registry is to manage packages and serve them to clients who request them. When a developer publishes a packages then this is done to a particular package registry. When a developer downloads a package this is done from a registry. Almost anytime the `jpm` tool needs to do anything it must contact a registry for information.

Many registries can exist in the JPM ecosystem. A default registry, the public registry[2], is made for sharing open-source packages with the Jolie community. Operations in the `jpm` tool that needs to speak to a registry will default to this registry.

[2]As of writing this thesis, the module system and package system has not yet been merged into the Jolie language. As a result, when referring to the public registry, we're referring to a registry which might exist in the future. The software required to run a registry is, however, implemented, as described in this thesis. A public registry does not exist yet.

Having multiple registries is beneficial for example if an organization wishes to share packages, but only wishes to share them internally within the organization. The JPM registry, and all other packages in the JPM ecosystem, are JPM packages. They are available from the public registry. This makes it relatively easy to host your own registry, since the process of this is the exact same as any other Jolie package.

The `registry` package also provides user and team management. Different rights may be assigned to these users, which can control who may perform certain actions. This is needed for controlling who's allowed to publish updates for a package. This also allows for a registry to be configured to only allow users from within an organization to use it. These features are covered by the `security` package which is described in Section 5.7.

Keeping track of package information is delegated to the `registry-db` package. We cover this package in Section 5.5.

5.5 The Package Database (`registry-db`)

The `registry-db` package is responsible for storing information about packages. In this section we will cover in depth how this service works and how it is used.

This package provides storage, and querying facilities for package information. This package does not provide any storage of actual packages. It is left to the client of this package to perform this. This allows freedom in how contents is stored, which may change between clients.

This package is consumed by both the `registry` and the `cache` packages. How these packages communicate with the `registry-db` package will be covered in their respective packages. The `registry-db` package essentially acts as a front-end to a SQL database. We won't cover the technical details of how this works, but for the remainder of this section we will cover what is stored and which facilities this package provides.

The core data stored in the database is shown in Figure 5.2. The core database model consists of three tables: `package`, `package_dependency`, `package_versions`. The `package` table describes a particular known package. Using any particular entry in the `package` table, we may find all known versions of this package. From a particular version of the package we may find all its known dependencies.

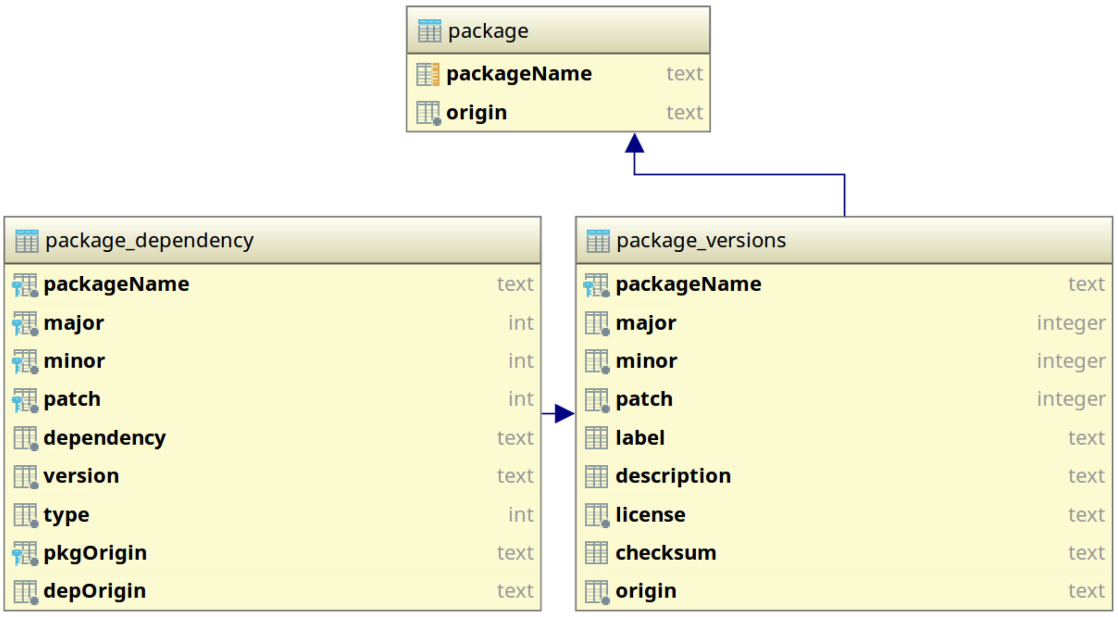

Figure 5.2: The core database model used in the `registry-db` package

The `origin` fields which appear in all three tables reference the origin registry. This is used for handling both cross-registry dependencies, but also for storing information about packages from different registries. The latter is useful for cache-like services, which collect information from many different registries.

Any particular package version also contains a `checksum` field. This field can contain any generic checksum which can be serialized as text, that is the actual checksum algorithm is defined by the client.

The remainder of the fields stored in the database simply mirror the most important fields of the package manifests. Certain fields from the package manifests, like the lifetime hooks, are left out of this database model. The reason for this being lack of interest by clients. The remaining fields, like name, description, license, are all of interest when it comes to discoverability of packages.

Table 5.4 shows the operations provided by the `registry-db` package. Generally the operations provided by this package for querying operations operate on a key matching either the `package` or `package_versions` table. Operations that update the tables usually operate on package manifests (typically provided by the `packages` package).

Operation	Description
`query`	Searches for packages matching a text query
`checkIfPackageExists`	Check if any package exists with name origin
`getInfoAboutPackage`	Lists all versions of a package matching name, origin, and possibly version
`compareWithNewest`	Checks if a package manifest is newer than what is known by this registry
`insertNewPackage`	Inserts a new version of a package manifest
`createPackage`	Creates a package with name and origin
`getDependencies`	Returns a list of dependencies for a package (name, origin, version)

Table 5.4: Operations provided by the `registry-db` package. Operations named changed slightly for formatting reasons

5.6 The JPM Cache

The `cache` of JPM is responsible for downloading packages, and maintaining the packages in a local cache.

The `cache` acts as a database of known packages. This database contains both the information needed to know about the database, as well as the binary containing the package data.

The `cache` consists of previously downloaded packages, along with a locally kept database which matches the type that the `registry` uses. The database kept also track the origin `registry` , since the `cache` will potentially download from multiple. The key used for this is the location URI of each `registry` . The name cannot be used, since these differ between each package.

The primary operation that the `cache` exposes is the `installDependency` operation. This operation will install a dependency directly into a package. JPM assumes that dependencies are installed under `<pkgRoot>/jpm_packages/<depName>`. Having packages stored under a common folder makes it easy to locate. This is, for example, useful when needing to filter out the packages. Common places for this would be version control, or from JPM when publishing a package.

The `cache` will first attempt to retrieve the package from its local database, if not found it will forward the request to the registry. In order to forward the request, the operation is fed both the registry and authentication token.

When a new package is received from a `registry` , two parts will be returned: the binary itself, along with the checksum that the `registry` believes the package to have. Checksums are both checked when initially receiving the package, but also at every install from the cache. This helps prevent initial corruption and further protects from the cache itself being corrupted.

The `cache` will maintain a `registry-db` just like the `registry` component does. When new entries are loaded into the `cache` , it will updated the `registry-db` in a similar fashion. As a result the `cache` is technically capable of performing the exact same responsibilities as the `registry` . This could for example allow for fully offline downloads assuming the data is already loaded into the cache.

5.7 Security

The authorization of JPM is implemented in the `security` service. The `security` service provides two different services, which work together to form the `security` service:

1. **An authentication system:** Responsible for proving identity of users.
2. **An authorization system:** Responsible for deciding which users are allowed to do what.

In this section, the client represents a service which uses the `security` services. The system is written to assume that a client *is not* an actual user. As a result most operations have been written to support various options which should not be available to actual users. This means that when the `security` service is deployed, care must be taken to ensure that it is not reachable by any ordinary user. Only the services which use the `security` service should be able to reach it. The services that use the `security` service, will then proxy only the relevant parts along.

5.7.1 Authentication

The authentication system has a user based system. Users authenticate themselves using a password. A group is a collection of users, which have an associated set of permission. A single user may be part of many groups, it is from groups that a user gains permission to use parts of the system. We cover group permissions in Section 5.7.2.

The authentication system is build following recommendations from OWASP[OWA1; OWA2; OWA3]. In this section the authentication implementation, and rules surrounding the system are summarized.

Username and Password Rules

The usernames used for the system are unique and case in-sensitive. The username is required to be between 1 and 64 characters long. All character types are allowed in usernames. Encoding of the usernames (and passwords) are deployment dependent, specifically it depends on the protocol and its configuration.

Passwords are case sensitive. The only limitation put on password is a length requirement between 5 and 128 characters.

The maximum lengths used for these are mostly to establish reasonable limits. Accepting an unlimited amount of data can be of concern for both storage and amount of network traffic. For data that should be presented, e.g., username, having a maximum is helpful for the design of user interfaces.

Storage

The usernames and passwords are stored in a SQL database. The database itself, along with its driver is configurable via the configuration system (See Section 3.2). Passwords are hashed with BCrypt[3][PM99], each individual password has its own salt. "Salting" a password is the act of adding a randomly generated string to each password. The salt itself isn't secret, but is instead used to avoid precomputed reverse lookup tables for the hashes (rainbow tables).

Security Concerns

Limiting number of login attempts within a period is often recommended. This is done to avoid brute-force attacks. This isn't possible in Jolie, since the sender isn't available to Jolie user code. This is most likely due to the fact that no unique sender ID exists, since Jolie can accept messages from various types of networks.

Authentication Process

We start the discussion of how the authentication process works by looking at how a client will successfully authenticate itself, this is shown in Figure 5.3.

During the initial request nothing special really happens. The `security` service collaborates with both the `bcrypt` service, along with its own database. The database contains both `user` objects and `session` objects. The `user` object has already been explained, and simply contains the username, the hash of the salted password, and the salt itself. The `session` object represents an active session. It contains the actual token, which is

[3]BCrypt is a password hashing algorithm which has been used by OpenBSD and others

a string generated by a cryptographically secure random number generator. Along with this token, the object contains a timestamp for generation, and a reference to the user object which created the session. The timestamp will be used for implementing timeouts for the session.

At the end of the initial request, the client receives the token. This token is what the client will have to provide for every privileged operation. An invalidation of a session may occur. This would ordinarily happen due to a logout, or potentially a timeout which is handled by the client. The `security` service also supports checking the "freshness" of a session token, by making sure it isn't older than some amount of time. This could be useful for ensuring re-authentication before critical operations, such as changing your password.

Figure 5.3 showed us the successful case. Internally, validation will be performed at most steps. In the case of failure at any of these steps a generic error message will be returned to the client. The error message will only indicate if it was a user error or a server error. This way an attacker cannot abuse the error messages for information. For example given the error message "Invalid password" an attacker would most likely be able to assume that the username itself is correct.

5.7.2 Authorization

The authorization system is based on an "Access Control Matrix"[SS94] to define its security model. An access control matrix, is a matrix A, where A_{ij} contains the *rights* that *role* i has for *resource* j.

A *right* is a single piece of information. It describes something that an entity is allowed to do for a specific resource in a role. In this implementation a right is a simple string.

A *resource* represents any entity that can have any rights associated with it. This could, for example, be a file, which might have associated rights such as "read" and "write".

A *role* is some entity which have a set of rights for resources. In the implementation roles are represented by *groups*. A group is a collection of users, which are provided by the authentication system.

Table 5.5 shows an access control matrix for a few files, which can be either read from or written to.

	File 1	**File 2**	**File 3**
Group 1	read, write	read	
Group 2	read	read	read, write

Table 5.5: A sample access control matrix for some computer files, which can be read from and written to by different groups

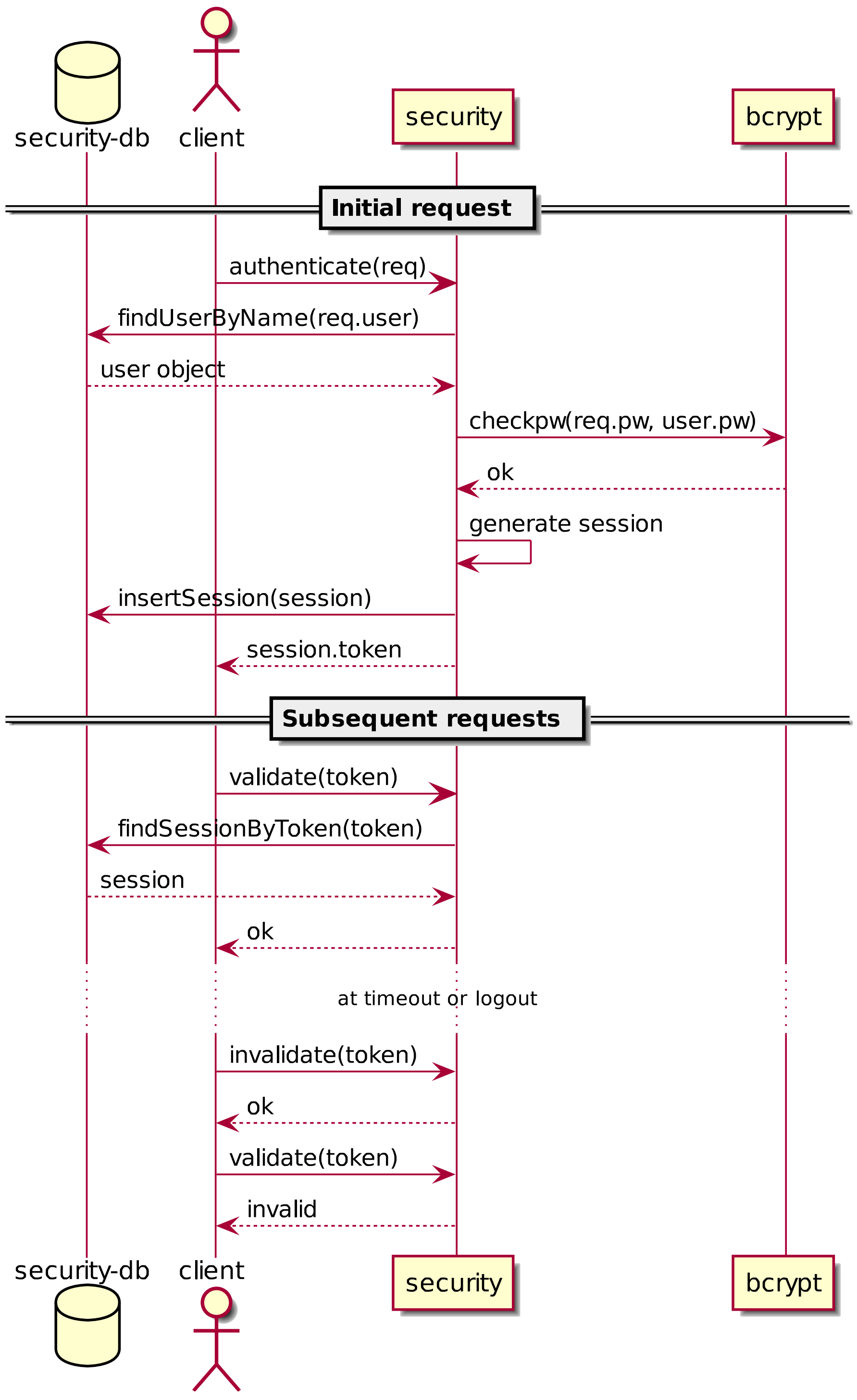

Figure 5.3: The authentication process when successful

5.7.3 Permissions in Registries

The `registry` service is the only consumer of the `security` service. On top of this service the `registry` service implements the security model JPM uses. JPM supports users these map one-to-one with the user model of the `security` service. On top of this JPM supports teams which are supported by a single underlying group. The purpose of a team is to allow for multiple users to all have ownership of a single package. Every single user and team get their own group, user groups only have a single member, while a team group has members corresponding to the team. The user groups are named `users.<name>`, and teams `groups.<name>`.

When a package is downloaded from or published to the `registry` the user's permissions will be checked. For a user to download a package, the user must be in a group having the `read` (or `write` for publishing) right for the corresponding resource. Each package has an associated resource called `packages.<name>`. A wild card resource (`packages.*`) also exists. The wild card resource represents every single package, thus if a group holds the `packages.*/read` right then that group is allowed to download ever package in the `registry` . This is exactly how a `registry` might provide guest downloads.

5.7.4 Authentication in JPM

When a user successfully authenticates, the authorization framework will provide an authentication token proving the user's identity.

For user convenience the JPM system does not require a valid username and password combination for every privileged operation. To avoid this, JPM will store the authentication token returned by the `registry` . These are stored in a file on the file-system.

The authentication token is saved instead of the username and password. Since if an attacker were to gain access to both the username and password, she would be able to gain full control over the user, this includes being able to generate new authentication tokens.

If an attacker instead were to gain only the authentication token, the damage is not as bad. The attacker would still be able to pose as the user, but only while the authentication token is valid.

5.8 Examples and Discussion

5.8.1 Creating, Sharing, Using, and Upgrading a JPM Package

In this example we will discover a typical workflow with JPM.

First we will create a new JPM package by using the `init` command. This will display prompts which guides the user through creating a new package. Following this, a new directory is created containing the basic files needed for a package.

```
dan@host:/ # jpm init
Package name
------------
> my-package

( ... Remaining cut for brevity ... )

dan@host:/ # cd my-package

dan@host:/my-package # ls
package.json

dan@host:/my-package # cat package.json | json
{
  "private": false,
  "name": "my-package",
  "description": "description",
  "version": "0.1.0",
  "authors": "Dan"
}
```

We're now ready to actually implement our service. With the service created, we can now publish the package to the public JPM registry. Before we can publish a package we must create a user on the registry.

```
dan@host:/my-package # jpm register
Username
--------
> dan
Password:
Password (Repeat):
```

It's possible to ask a registry who we are currently authenticated as:

```
dan@host:/my-package # jpm whoami
dan
```

When authenticated, it is possible to publish package to a registry.

```
dan@host:/my-package # jpm publish
```

Now lets pretend we another user, who wishes to use our package. In order to download from the JPM public registry there is no need to be authenticated (although other registers may require this). Lets create a package and add it as a dependency. This requires us to update the `package.json` files generated by JPM:

```
{
    "dependencies": [{ "name": "my-package", "version": "0.1.X" }]
}
```

The dependency here uses a SemVer expression, as explained in Section 4.3.3. This expression states that the newest version of the package should be picked, as long as that version has a major version number of 0 and a minor version number of 1.

We can now download all the dependencies of our package:

```
dan@host:/client # jpm install
Downloading          my-package@0.1.0
Completed            my-package@0.1.0

dan@host:/client # tree .
.
|-- jpm_lock.json
|-- jpm_packages
|   `-- my-package
|       |-- interface.iol
|       |-- main.ol
|       `-- package.json
`-- package.json

2 directories, 5 files
```

We can see that this generates a lockfile (Section 4.5) which states that “my-package” has been resolved from the public registry as version 0.1.0.

```
dan@host:/client # cat jpm_lock.json | json
{
  "locked": {
    "my-package@0.1.X/public": {
      "resolved": "0.1.0"
    }
  },
  "_note": "Auto-generated"
}
```

If we publish a new version of “my-package” and run `install`, then we still be left with the same version. It is not until we run `upgrade` that we receive the newest version.

```
dan@host:/client # jpm search my-package
my-package@0.1.1/public
  description

dan@host:/client # jpm install
Downloading          my-package@0.1.0
Completed            my-package@0.1.0

dan@host:/client # jpm upgrade

dan@host:/client # jpm install
Downloading          my-package@0.1.1
Completed            my-package@0.1.1
```

We can start our package by setting the “main” attribute of the package manifest. This allows us to write:

```
dan@host:/client # jpm start
Service running...
```

We can even run with a particular configuration:

```
dan@host:/client # jpm start --conf my-profile config.col
Service running...
```

5.8.2 The Calculator System

Continuing with the “modularized” version of the calculator system from Section 3.4.

The first step in making these usable from the package manager is of course to create package manifests. This action is fairly straight forward, especially since we already have a system working with the Jolie module system. The package manifest for the `multiplication` package is shown in Listing 5.2. The only significant modification here is perhaps a version number, and a formalization of dependencies (in this case the `numbers@1.0.0` package).

```json
{
    "name": "multiplication",
    "description": "An multiplication package",
    "license": "MIT",
    "authors": ["Dan Sebastian Thrane <dathr12@student.sdu.dk>"],
    "main": "main.ol",
    "version": "1.0.0",
    "dependencies": [{ "name": "numbers", "version": "1.0.0" }]
}
```

Listing 5.2: Package manifest for the `multiplication` package

The other package manifests will look fairly similar. The dependency graph ends up mimicking the system architecture exactly. However there are some subtle problems associated with the current approach.

Currently the calculator will always download the `numbers` package, despite not directly depending on it. This happens due to the dependency on `multiplication` which itself depends on `numbers`. However the `calculator` service really only depends on `numbers` if `multiplication` is embedded. At this point this may seem like a minor detail, but this can get vastly more complicated if there are more dependencies (which themselves may have dependencies and so on).

The solution to this problem, is to make sure the package manager will only download the files we actually need, and no more. This would mean if no embedding is ever performed, we will need only the dependencies required to perform interfacing. If embedding is required we will need both the interface dependencies along with a concrete implementation.

For the sake of simplicity the same dependency system is used. It is recommended that every service separately publishes its interface and concrete implementation. This approach has the added benefit of allowing multiple implementations of same interface. For closed source services a single package containing only the interface and types could be published.

Depending on if an embedding is desired the concrete implementation can simply be listed as a dependency, the interface dependency will implicitly be picked up (from the concrete implementation's dependencies). If an embedding is *never* desired a dependency can be made to just the interface package.

Additionally JPM might benefit from optional dependencies which are only when certain conditions are made, for example during some "development build". This might prove to be a decent compromise, allowing for the convenience of embedding without directly forcing users of the package to also depend on it.

5.8.3 Using Lifetime Hooks to Improve Development Workflow

In this example we will discover how the use of Jolie lifetime hooks can improve the development workflow. In this example we will look at a hybrid Jolie-Java service. The service itself will be written in Jolie, using an internal Java service to perform some of the computational heavy-lifting. The `packages` service, which performs package manifest validation, used in the package manager is an example of such a package type. The primary flow and communication is performed entirely by Jolie, while the more "computationally heavy" parts, like parsing, are handled by the internal Java service. This architecture is illustrated in Figure 5.4.

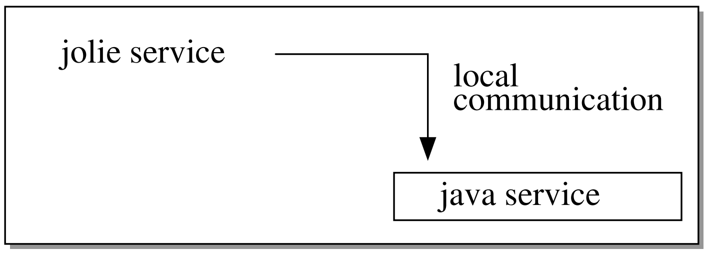

Figure 5.4: A Jolie service using an internal Java service

The Java service uses Gradle as its build system. The build system is capable of producing a JAR file. The Jolie engine requires this file to be included in the classpath in order to perform the embedding. If the file is placed in the `lib` directory this will happen automatically.

A small bash script was made to run JAR creation through Gradle, and then copy the artifact into the `lib` directory. Interested readers may find both the Gradle build script and bash script in Appendix 3. Both scripts are relatively simple, but should be a sufficient starting point.

The file structure of the project is shown in Listing 5.3. The `build` file is the script which performs the entire build and copies it into the `lib` directory.

```
.
|-- build
|-- java-service
|   |-- build.gradle
|   `-- src/main/java/dk/thrane/jolie/JavaService.java
|-- main.ol
`-- package.json
```

Listing 5.3: The file structure of a Jolie-Java hybrid package

Assuming a lot of changes are made to the Java service, this would mean two commands would almost always be required to create a new iteration of the build. This can be minimized by creating a `pre-start` hook as shown in Listing 5.4. Having it as a part package manifest makes the intent very clear. Having some home-made build script would instead rely entirely on convention. Such conventions may vary from project to project, while having it as a part of the package manifest would remain consistent across all JPM packages.

```
{ "events": { "pre-start": "./build" } }
```

Listing 5.4: A `pre-start` hook for performing compilation of the internal Java-service

The JPM lifetime hooks aim to enforce the implicit workflow a project typically has by making it explicit. This could for example be used to enforce the constraint that testing must succeed before publishing to a registry. This can be implemented by using the `pre-publish` hook. If the script being run returns a non-zero exist code, then the publishing action will be canceled (similar behavior applies to all `pre-*` hooks).

Currently however it would be useful if a Jolie package could allow for multiple entry-points. This would make it easier to write tests written directly in Jolie. However having multiple entry-points would bring up the question of which to use for embedding. We could keep the constraint of a single main entry-point used for embedding, and multiple others used for various other tasks, such as testing.

Conclusion

In this thesis, we have presented two new extensions to the Jolie programming language, namely, the module system and the configuration system.

The module system allows the Jolie engine to think about collections of files which are rooted in a particular directory. This introduced the new module include primitive.

The configuration system is built upon the module system to allow configuration of modules. The configuration system introduced the new configuration format (COL) which allows the user configuration of modules. COL files entirely replace the old way of doing configuration, through ordinary Jolie source files. COL files present a way of configuring a module, in a similar syntax to that of ordinary Jolie code.

Input ports and output ports, the primitives used by Jolie for communication, can be configured. The configuration of ports, follow the features that the Jolie language already provided. As a result, it is now possible to move freely between externally bound output ports to embedding the same service, entirely from the configuration. This was previously not possible without changing the source code of a service.

Parameters allow for configuration values to be provided to a service. Parameters are automatically type-checked at deployment time of a service. Due to constraints within the Jolie engine, a new verification stage was added to the Jolie engine.

The configuration system introduced a notion of interface parametricity through interface rebinding. This feature allows developers to write generic Jolie services which use the, already existing, aggregation feature. This was also not previously doable without changing the source code of the service.

We also presented the Jolie Package Manager (JPM). JPM manages a new concept of packages. Packages are an extension of the Jolie modules, also developed in this thesis. Package extends modules, by adding a package manifest which describes the package.

Packages can be published to a registry. Published packages are downloaded from the registry, by the JPM tool, and installed into packages. The JPM tool also provides a variety of features that include registry account management, lifetime hooks, and integrity checking of packages.

Appendix

A.1 JPM Manifest Specification

name

Field 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

Field 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

Field 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

Field 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

Field 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

Field 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

Field 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.

dependency.name **Field 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.

dependency.version **Field 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)

dependency.registry **Field 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

Field 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.

registry.name **Field 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

registry.location **Field 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"

A.2 COL Grammar

Presented as an ABNF-like syntax. All literals are case sensitive.

```
configuration-tree = *include *config-unit

include = "include" qstring

config-unit = config-unit-header "{" config-unit-body "}"
config-unit-header = [ "profile" qstring ] "configures" qstring
                     [ "extends" qstring ]
config-unit-body = *definition
definition = port | interface | parameter

port = input-port | output-port
```

```
input-port = "inputPort" identifier "{" port-body "}"
output-port = embedded-output-port | external-output-port
external-output-port = "outputPort" identifier "{" port-body "}"
embedded-output-port = "outputPort" identifier "embeds" qstring "with" qstring
port-body = *port-property
port-property = location-property | protocol-property
location-property = "Location" ":" qstring
protocol-property = "Protocol" ":" identifier [ protocol-config ]
protocol-config = inline-tree

interface = "interface" identifier "=" identifier "from" qstring

parameter = variable-path "=" value
variable-path = var-id *var-node
var-id = ( "(" qstring ")" ) | identifier
var-node = "." var-id [ "[" unsigned-int "]"  ]
value = ( primitive [ inline-tree ] ) | inline-tree
inline-tree = "{" *(tree-child ",") tree-child "}"
tree-child = "." variable-path "=" value
primitive = qstring | int | long | double | bool

letter = <any Unicode code point recognized by Java via Character#isLetter>
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
bool = "true" | "false"
int = unsigned-int | "-" unsigned-int
long = int ( "l" | "L" )
unsigned-int = 1*digit ; Jolie discards leading 0s
double = 1*digit "." 1*digit [ double-exp ] | 1*digit double-exp
double-exp = ( "e" | "E" ) [ "-" | "+" ] 1*digit
qstring  = ( <"> *(qdtext | quoted-pair ) <"> )
quoted-pair = "\" ( "\" | "n" | "t" | "r" | <"> | "u" )
qdtext = <any TEXT except <">>
```

A.3 Scripts from Lifetime Hooks Example

File: `build`

```
#!/usr/bin/env bash
cd java-service
gradle jar
cd ..
```

```
mkdir -p lib
cp java-service/build/libs/*.jar lib
```

File: `java-service/build.gradle`

```
group 'dk.thrane.jolie'
version '1.0-SNAPSHOT'

apply plugin: 'java'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.11'
    compile files("/usr/lib/jolie/jolie.jar")
}

jar {
    from {
        configurations.compile.findAll { it.name != "jolie.jar" }.collect
        ↪  { it.isDirectory() ? it : zipTree(it) }
    }
}
```

Bibliography

[CARA] The Cargo Authors. *The Manifest Format.* http://doc.crates.io/manifest.html. Accessed 26-05-17.

[CARB] The Cargo Authors. *Cargo: packages for rust.* https://crates.io. Accessed 31-05-17.

[Cha09] S. Chacon. *Pro Git.* Books for professionals by professionals. Apress, 2009. ISBN: 9781430218333. URL: https://books.google.de/books?id=3XcW4oJ8goIC.

[Fow06] Martin Fowler. *Continuous Integration.* https://www.martinfowler.com/articles/continuousIntegration.html. Accessed 28-05-17.

[Fow14] Martin Fowler and James Lewis. *Microservices.* https://www.martinfowler.com/articles/microservices.html. Accessed 30-05-17.

[GRAA] The Gradle Authors. *The Gradle Daemon - Gradle User Guide Version 3.5.* https://docs.gradle.org/current/userguide/gradle_daemon.html. Accessed 26-05-17.

[GRAB] The Gradle Authors. *Gradle Build Tool.* https://gradle.org/. Accessed 26-05-17.

[JOLA] The Jolie Team. *Jolie Programming Language - Official Website.* http://jolie-lang.org/. Accessed 29-05-17.

[JOLB] The Jolie Team. *Jolie Standard Library Reference.* http://docs.jolie-lang.org/. Accessed 30-05-17.

[Jon05] M. Tim Jones. “Defensive Programming”. In: (Feb. 2005). URL: http://www.drdobbs.com/defensive-programming/184401915.

[LEO] The Jolie Team. *The Jolie Web Server - GitHub repository.* https://github.com/jolie/leonardo/tree/de335885. Accessed 30-05-17.

[LEOB] The Jolie Team. *The Jolie Website - GitHub repository.* https://github.com/jolie/website/tree/d4aeb68e. Accessed 30-05-17.

[Mon10] Fabrizio Montesi. “Jolie: a service-oriented programming language”. PhD thesis. 2010.

[NPMA] The NPM Authors. *05 - Using a ‘package.json‘ | npm Documentation.* https://docs.npmjs.com/getting-started/using-a-package.json. Accessed 26-05-17.

[NPMB] The NPM Authors. *semver | npm Documentation*. `https://docs.npmjs.com/misc/semver`. Accessed 26-05-17.

[NPMC] The NPM Authors. *npm*. `https://npmjs.com`. Accessed 26-05-17.

[ORAA] Oracle. *Using JAR Files: The Basics (The Java™ Tutorials > Deployment > Packaging Programs in JAR Files)*. `https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html`. Accessed 26-05-17.

[OWA1] OWASP. *Password Storage Cheat Sheat*. `https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet`. Accessed 22-05-17.

[OWA2] OWASP. *Authentication Cheat Sheat*. `https://www.owasp.org/index.php/Authentication_Cheat_Sheet`. Accessed 22-05-17.

[OWA3] OWASP. *OWASP Secure Coding Practices Quick Reference Guide*. `https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf`. Accessed 22-05-17.

[OXML] Microsoft. *Open XML Formats and file name extensions - Office Support*. `https://support.office.com/en-us/article/Open-XML-Formats-and-file-name-extensions-5200D93C-3449-4380-8E11-31EF14555B18`. Accessed 26-05-17.

[PM99] Niels Provos and David Mazieres. “A Future-Adaptable Password Scheme.” In: *USENIX Annual Technical Conference, FREENIX Track*. 1999, pp. 81–91.

[RFC3986] R. Fielding T. Berners-Lee and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. Jan. 2005. URL: `https://www.ietf.org/rfc/rfc3986.txt`.

[SEMV] Tom Preston-Werner. *Semantic Versioning 2.0.0*. `http://semver.org`. Accessed 28-05-17.

[SPDX] The Linux Foundation. *SPDX License List | Software Package Data Exchange (SPDX)*. `https://spdx.org/licenses/`. Accessed 28-05-17.

[SS94] Ravi S Sandhu and Pierangela Samarati. “Access control: principle and practice”. In: *IEEE communications magazine* 32.9 (1994), pp. 40–48.

[VERTA] The Vert.x Authors. *Vert.x Application Configuration*. `http://vertx.io/blog/vert-x-application-configuration/`. Accessed 30-05-17.

[YARNA] The Yarn Developers. *Yarn GitHub repository - yarn/src/package-constraint-resolver.js*. `https://github.com/yarnpkg/yarn/blob/0477a392/src/package-constraint-resolver.js`. Accessed 28-05-17.

[YARNB] The Yarn Developers. *package.json | Yarn*. `https://yarnpkg.com/en/docs/package-json`. Accessed 28-05-17.

[YARNC] The Yarn Developers. *Yarn*. `https://yarnpkg.com/en/`. Accessed 28-05-17.