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 8, 2017

Contents

Introduction

Introduction goes here

Background

Background goes here.

2.1 Microservices

- General stuff about microservices, actually introduce the thing

2.2 Package Managers

Some background on package managers.

2.3 Introduction to Jolie

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

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

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

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

Listing 1: A very simple Jolie program

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

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

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

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

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

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

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

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

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

    OneWay:
        hello(HelloType)
}
```

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

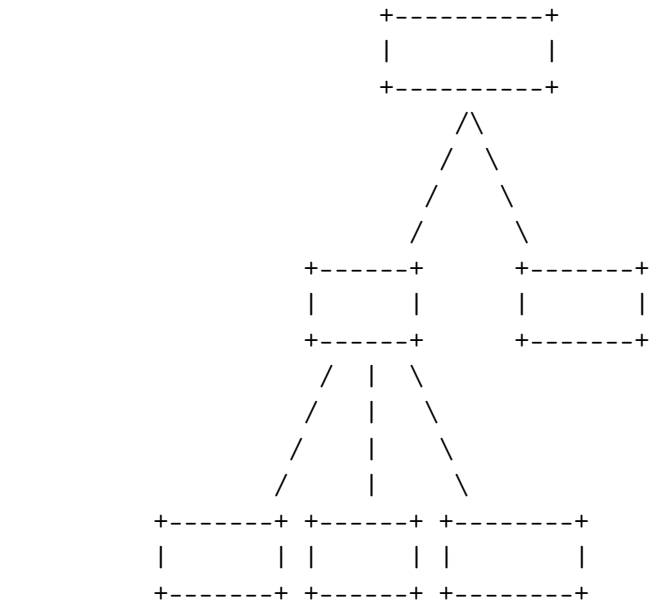
Whenever the Jolie interpreter invokes an operation on an output port, or receives a request on an input port, the types will be checked. This check ensures that we don't send out incorrect requests, and ensures that we do not attempt to process an incorrect request. Listing 4 show the request and response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

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

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

Listing 4: Jolie types are tree-like structures

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

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



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

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

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

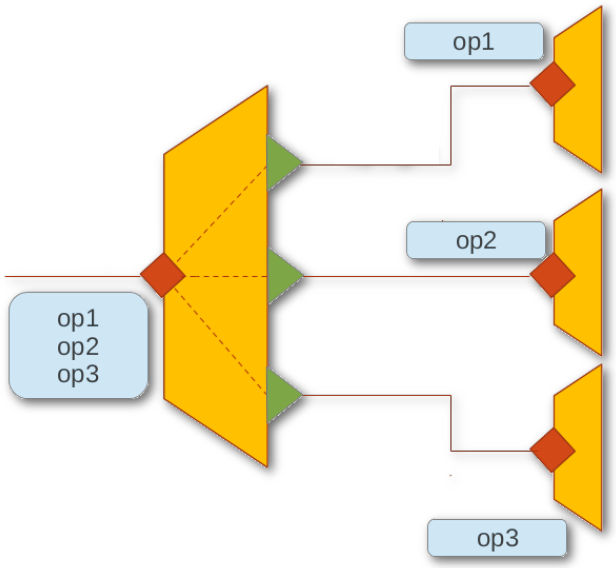


Figure 2.1: Aggregation is a generalisation of proxies and load balancers

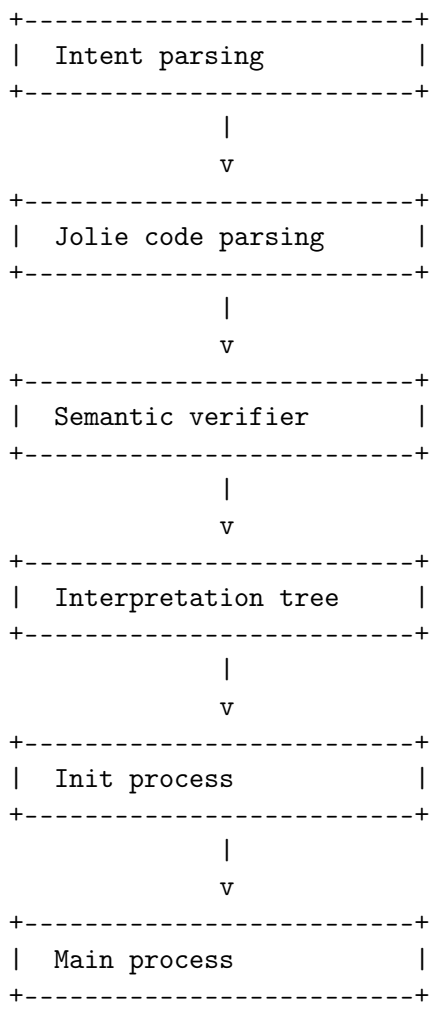
2.4 The Jolie Engine and Interpreter

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

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

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

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



Listing 5: A simplified view of the Jolie interpreter pipeline

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

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

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

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

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

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

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

2.5 A Complete Application with Jolie

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

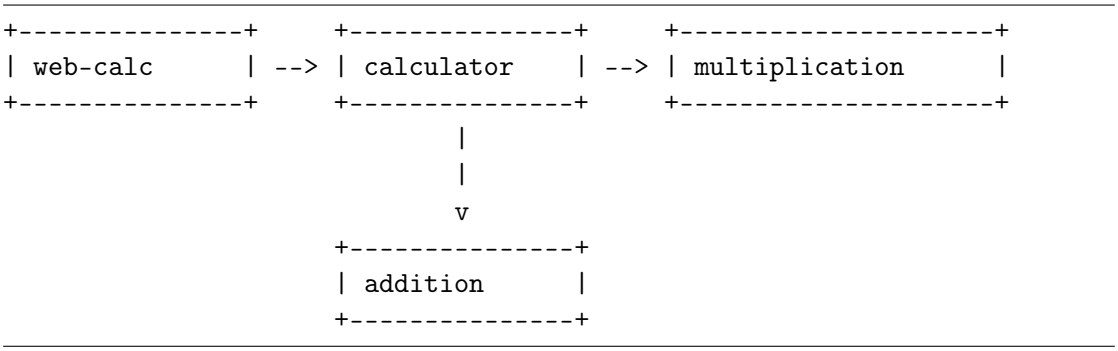
Along the way several patterns used for re-using code will be highlighted. This will serve as the core motivation for our solution.

2.5.1 Architecture

The application we will be building is a very simple calculator system. The architecture of our application is shown in Figure 6 we see an illustration of the system's architecture.

The calculator service will provide various operations for applying an operator on a sequence of numbers. In order to perform this, highly complex, action of applying these operators, the calculator will contact other microservices.

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



Listing 6: The Architecture of a Simple Microservice System

2.5.2 Implementing the Calculator Service

We will keep our focus on the *calculator* service, and its interacting with other services. Putting together the stuff learned from the previous sections, we can quickly setup an input port for the service, which has the appropriate interface. It is considered best practice to place the public interfaces that a service exposes in its own separate file. Files intended for other services to include typically have the file extension `.iol` as opposed to `.ol`. There is no technical difference between the two, but it allows for the developer to more easily express intent. 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 for the service implementation (`calculator.ol`), and another which can be used by other services (`calculator.iol`). The implementation and interfaces are shown in Listing 7.

```
// 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 7: TODO Caption

Observation 1. The interfaces and types of a service are typically separated into their own files. This files typically has 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.

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

The operations that the calculator exposes, needs to collaborate with the other services. In order for us to speak to them they need output ports.

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

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

Binding an output port to an external service is relatively easy. For example to let the `multiplication` port bind to a service using https, we might write `Location: "socket://mult.example.co`

and `Protocol: https`. The input port at the multiplication processor would also have to match this, to ensure it runs on the correct port and speaks the correct protocol.

It is a similar to bind an output port to an embedded service. However this is done by setting the `Location` or `Protocol` attributes. We must instead instruct the engine to embed the service, which most importantly requires us to point to some executable service. The Jolie engine supports several language for these embedded services, including Jolie, Java, and JavaScript. For our desired deployment, we wanted to embed the 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 9. Just like in the case of the external services, the input port of the receiving service *must match*. In the case of embedded service there must be an input port listening on the `"local"` location.

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

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

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

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

Listing 8: 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 are 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 may configure.

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

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

Listing 9: Embedding the `addition` service in the `Addition` output port

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

> **Observation 3.** Services intended to be used as libraries are often contained in a single `.iol` file. This file contain 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. For completeness sake this might look like shown in Listing 10.

```
[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 10: Implementing the checkout operation

Finally we want to reflect slightly on the file structure that this service ended up having. 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 service 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 11.

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

Listing 11: 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. The files required depend on if an embedding is required, 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 Summarizing the Problem

The goal of this thesis is to promote, and facilitate better code reuse in Jolie. In Section 2.5 we showed how a typical Jolie service would be written, and summarized a number of patterns we have observed.

3.2 Modules

A module in Jolie is defined as a project root directory, and optionally an entry-point. Modules are uniquely identified by a name.

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.

As an example, we may inform the engine about a module called "foo", which has its source code placed in `/packages/foo` with an entry-point in `/packages/foo/main.ol` we would write: `--mod foo,/packages/foo,main.ol`. This implementation technique is very close to how, for example, you would add a JAR file to the classpath of a Java program.

The result of this decision is that quite a lot of additional options may need to be passed to the engine. However this choice was purposely chosen, it is left for another tool to make this job easier. In this case a package manager is expected to take the heavy lifting, and figure out which modules exists. This allows for more freedom in how these tools are implemented, and another implementation strategy, than the one provided in the package manager, could be created without any changes to the language infrastructure. See Section TODO about how the package manager passes this information to the engine.

To allow for better organization, a new type of include has been added to the language. These include allows the developer to include from a particular package. The syntax of

this include is shown in Listing 12. When a package include is used, the search path will be altered, such that the project root is changed to that of the package (as opposed to where the engine was started). Any file included from within this package should perform its includes relative to its own root, rather than the project root.

TODO Example

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

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

Listing 12: Extension to the include statement, made for module imports

3.2.1 Include Algorithm for Jolie

TODO To this date I still don't understand how the include algorithm works. Ask Fabrizio about this.

3.2.2 Module Include Algorithm

TODO Module includes modify the search path. This also means the injection of the include directory, which is used as a convention. This helps maintain expected behaviour.

TODO In order to implement the root switching we use a stack.

3.3 Configuration

3.3.1 Motivation

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 isn't limited to just the desired configuration. As a result, a service developer cannot be certain that the configurator (entity who provides configuration) doesn't start messing with other details of the program. Deploying defensive programming[1] techniques against

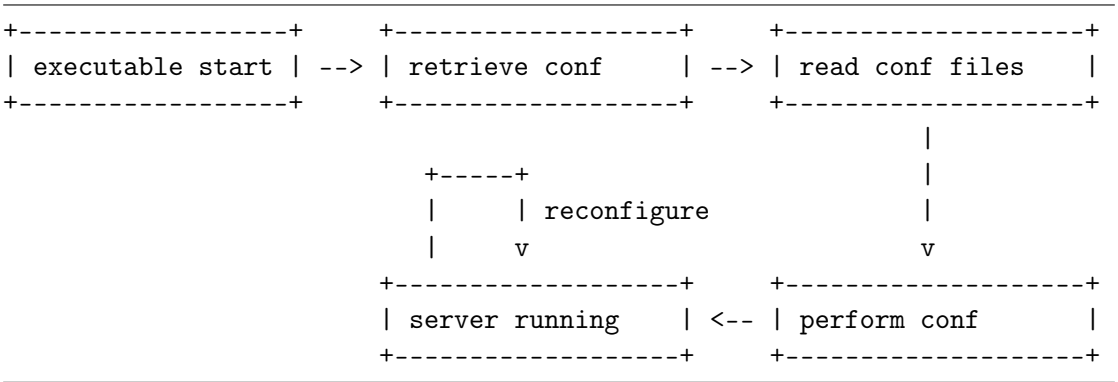
[1] Defensive programming techniques are usually employed for systems that require high availability, or where safety and security is required.

this becomes significantly more problematic, since no guarantees about the configuration source file can really be made.

Distributing re-useable packages is also problematic with this approach. Several features of package management requires that the package remains read-only. For example, updating of packages require this, without it source-code merges would be required.

We also saw that a lot of issues, that should have been a deployment issue, became a code issue.

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, could be Vert.x, it is a tool-kit for building reactive applications on the JVM. Examples of such "reactive applications" are microservices. The configuration workflow is shown in Figure 13. The system will retrieve, and read external configuration files, directed by the user code, and apply the configuration as needed.



Listing 13: Simplified workflow for configuration of Vert.x applications

TODO Defined deployment time somewhere

However implementing such as a system in Jolie has its problems, most of these come from the difference between general-purpose programming languages and specialized programming languages.

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. Doing this has multiple advantages, such as less complexity in user code, but it also means that user code is capable of doing less. Jolie user code can for example not control networking directly, but is instead forced to use the abstractions provided by Jolie (sending messages). The language puts

constraints on certain constructs being fully prepared directly in the source code. This is analogous to a programming language requiring the types of a struct's field to be present at compile time. As a result, not all constructs can be changed at run time. Concrete examples of this includes the input ports, which needs to be ready at deployment time. Thus without native support for configuration of these, it would not be possible to change the input port.

3.3.2 Configuration Units

A configuration unit is the basic entity, which encapsulates the configuration of a single Jolie module. A configuration unit is known by its name (known as its "profile"), along with which module it configures. Having multiple profiles for the same module can be useful for a variety of use-cases. A common use-case, could for example be to have separate profiles for development and production.

The units hold configuration for every possible type of configurable construct in Jolie. The ones supported are:

1. Input and output ports
    - Location
    - Protocol and protocol parameters
    - Embedding of other services (output ports only)
2. General purpose parameters
3. Interface rebinding

In the coming sections we'll mostly focus on the first two, in Section 3.4 we'll cover interface rebinding.

Configuration units are defined in configuration files, which may contain several units. These files may even include other files, to pull in more configuration units.

Jolie provides a new configuration file format. This format is custom, and made to mimic the syntax of Jolie. Listing 14 shows a very simple configuration unit. This units sets the location and protocol for the output port `A`, the location of the input port `ModuleInput`, and a parameter.

```
profile "hello-world" configures "my-module" {
    outputPort A {
        Location: "socket://a.example.com:3000"
        Protocol: sodep { .keepAlive = true }
    },

    inputPort ModuleInput {
        Location: "socket://localhost:80"
    },

    myParameter = 42,
    myParameter.subProperty = "hello"
}
```

Listing 14: 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 15 shows the embedding of output port `A`. Note that we need to make a reference to the module, 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".

```
profile "hello-world" configures "my-module" {
    outputPort A embeds "a-module" with "a-profile"
}

profile "a-profile" configures "a-module" {
    // configuration of a-module goes here.
}
```

Listing 15: Embeddings make reference to other configuration units

As we can see from the examples, it is not necessary to provide all the values of a port. It isn't necessary for two reasons. The first reason is that certain values may be provided by the underlying module, which uses this unit. If a module provides a value, then the configuration unit cannot override it. The second reason is that configuration profiles may extend other profiles.

Configuration units may extend another unit, which configures the same module. The tree of inheritance may be of an arbitrary depth, but each unit may only extend a single

unit, and they must configure the same module. The child is also wins when it comes to configuration. This means that if unit "B" extends "A", and they both configure the same value, then the values found in B is the one that is correct. Listing 16 shows an example of extension with units.

```
1  profile "a" configures "a-module" {
2      aValue = 42,
3      aValue.sub = "hello",
4
5      outputPort ExternalService {
6          Location: "socket://external.example.com:42000"
7      }
8  }
9
10 profile "b" configures "a-module" extends "a" {
11     aValue = 100
12     // aValue.sub = "hello"
13     // ExternalService.location = "socket://external.example.com:42000"
14 }
```

Listing 16: Configuration units may extend other units

The module developer is often aware of what the defaults should be. For this reason default configuration profiles may be shipped along the modules, which are implicitly imported into every configuration file. The Jolie engine will look for any `.col` file[2] 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.

It should be noted that no single unit is required to provide all configuration. The system doesn't have any "abstract"[3] configuration units. However it is required that the configuration file provides all the necessary configuration, as declared by the module. We'll learn more about how a module declares configuration in Section 3.3.3.

3.3.3 Configuration and the Core Language

In section 3.3.2 we learned about the configuration files, but never actually saw the corresponding Jolie modules. In this section we'll discover how the language has changed

[2]The file extension of the configuration file
[3]As in abstract classes, a concept often used in object oriented programming

to accommodate this configuration feature.

The configuration system provides the ability to configure any type of port, and additional provide the module with "parameters". Ports were covered in section TODO. Most importantly it should be noted that output ports do not require any values at deployment time, and can be changed at run time, while input ports require all their values at deployment time, and cannot be changed at run time.

```
outputPort Service {
    Location:
     ↪  "socket://service.example.com:443"
    Protocol: https {
        .debug = true,
        .debug.showContent = false
    }
    Interfaces: IService
}
```

```
outputPort Service {
    Interfaces: IService
}
```

Listing 17: Two valid output ports. (Left) A fully configured output port. (Right) A minimal output port.

Listing 17 shows a two different output ports, both perfectly valid.

TODO

When should we configure? Go through a few different approaches, always override source, always let code win, do we let everything be configurable? Should configuration stay final, or should we allow changing it later? How do we distinguish between accidentally leaving something out and wanting it to be configurable?

Output ports

Input ports, and why they are harder.

The `dynamic` keyword has been introduced to deal with which ports that should allow configuration, any port which does not have the dynamic keyword is static. Only ports which are static should be configurable. TODO Why is this a good idea.

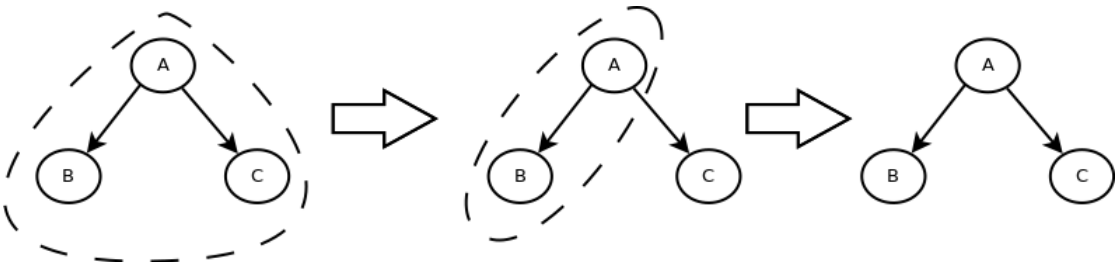
Parameters

3.3.4 Implementing the Configuration Format

Implementation, and fixing the AST

3.3.5 Example

TODO This is just copy pasted from some markdown document. Might be able to find a better example. Syntax is most likely also outdated.



(The dashed region corresponds to which services that are embedded together)

This use-case is intended to show how we can easily go from a prototype where we embed everything (this is easier to run locally) to hosting each service by itself.

```
// A.ol

ext outputPort A {
    Interfaces: AIface
}

ext outputPutPort B {
    Interfaces: BIface
}

constants {
    FOO: int
}
```

```
// A.col

include "B.col" // The includes work just like they do in Jolie
include "C.col"

// Previously called namespace. Configures makes it more explicit that we're
// talking about a specific package and not an arbitrary name
configures "A" {
    // Like always we just put the definitions here
    FOO = 42

```

```
12      // We alter the syntax slightly for output ports being embedded
13      outputPort B embeds B
14      // The second B refers to the profile B (if no profile is specified it gets
15      // the same name as the package it configures). This means that this
16      // configures block also has name "A". It could also have been written as:
17      // `profile "A" configures "A"`
18
19      outputPort C embeds C
20  }
```

Note: The previous proposal did not allow for embedding of services directly from the configuration. This is however needed since packages by themselves should be considered read-only. Thus if we want to configure a package to embed its dependencies, then this must be done from external configuration, we cannot do this in source.

```
1   // B.col
2
3   configures "B" {
4       inputPort B {
5           Location: "local"
6       }
7   }
```

```
1   // C.col
2
3   configures "C" {
4       inputPort C {
5           Location: "local"
6       }
7   }
```

In order to use an external B we need to update "A.col":

```
1   // A.col
2   include "C.col"
3
4   configures "A" {
5       outputPort B {
6           Location: "socket://b.example.org:8000"
7           Protocol: sodep
8       }
```

```
    outputPort C embeds C
}
```

In order to update the deployment file of B we simply need to update the input port to no longer be local.

```
// B.col
configures "B" {
    inputPort B {
        Location: "socket://localhost:8000"
        Protocol: sodep
    }
}
```

In most cases however this would be unnecessary since the default configuration file for "B" could already include a default input port.

In that case we could simply deploy directly from the default configuration. The restriction on configuration units to only configure a single package helps a lot. This restriction means that we cannot from any node configure any other node which isn't a direct child of it. Without this we wouldn't be able to easily swap out one configuration unit for another.

3.4 Interface Rebinding

Words

Jolie Packages

4.1 Introduction

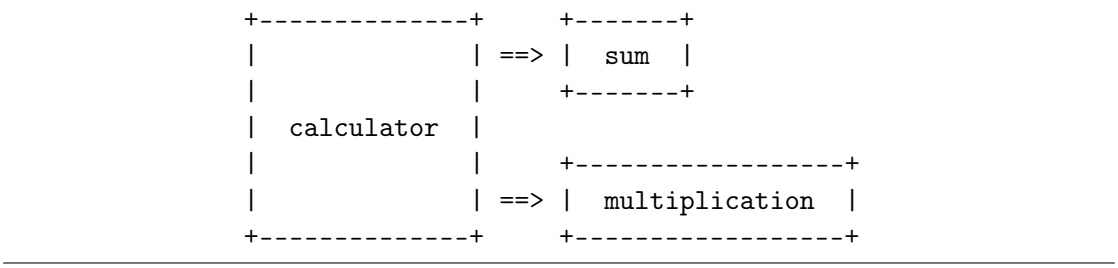
A Jolie Package is an extension of a Jolie Module. Recall that a Jolie Module was defined as a collection of resources, a name, and optionally an entry-point for the module. A package extends this concept by adding information required for package management.

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

In listing 18 we show a simple package manifest. This manifest showcases the most important features of the manifest.A complete specification of the package manifest format can be seen in Appendix A. The service that this manifest describes is shown in figure 19.

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

Listing 18: A Simple Package Manifest



Listing 19: A Calculator Service

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

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

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

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

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

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

4.2 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 A. However in this section

we'll also cover some of the reasoning behind the choices found in Appendix A.

4.2.1 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 A:

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 (see section 2.3 of RFC 3986)
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 4 states that package names are unique within a single registry. But this constraint actually goes even further, since these packages directly extend from the modules, we cannot have a system containing two packages named the same thing, even if they reside in two different registries. Generally we make no assumptions that registries communicate with each other, for this reason it is perfectly allowed that two distinct registries may contain name clashes, but any single package must not have a dependency which clashes either with another dependency, or the package who has the dependency itself.

4.2.2 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, 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.

4.2.3 The `version` Attribute

The `version` attribute used by packages declare which version of the source code the manifest describes. This field uses the Semantic Versioning (SemVer) 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 20.

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

Listing 20: 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[1] and NPM[2].

TODO Highlight benefits of using semantic versioning.

For dependencies the version field allows for a bit more flexibility using SemVer expressions, also known as SemVer ranges. 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.

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 4.4 we cover one of these problems, and its solution.

4.2.4 The TODO Name 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 (Section TODO). The attribute is also used for starting the package, we discuss this in Section TODO.

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 TODO.

[1] `http://doc.crates.io/manifest.html`

[2] `https://docs.npmjs.com/getting-started/using-a-package.json`

4.2.5 The `registries` Attribute

The package manager may communicate with one or more registries, when performing its work. This may either 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 21 shows an example.

```
1 {
2     "name": "my-registry",
3     "location": "socket://registry.example.com:9999"
4 }
```

Listing 21: 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.

TODO Should we talk about why it is good to have multiple registries here?

4.3 Dependencies

4.4 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, for example, Cargo, and Yarn.

Continuous integration (CI) is a software development process. 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. 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 correctly followed.

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 by 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 developers 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. This 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 even still left in. 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 22 shows the lock file, which would have been generated for the scenario above.

It should however be noted that fresh lock files are generated for every package. This means that the lock files are *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.

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

Listing 22: 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

4.5 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[3] provides such a feature.

All hooks are performed on the client-side, and never on the server side (i.e. 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 23 shows an example hook, which runs before publishing which ensure that all tests pass.

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

Listing 23: 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 (i.e. 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.

[3] `https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks`

4.6 The `.pkg` Format

The `.pkg` file format, is the format that JPM uses for distribution of packages. Simply 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. One such extension could for example be native code signing, which we discuss in more details in Section TODO. 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[4] format, and Office Open XML[5].

4.7 Integrity Checks

This is a section about integrity checks in JPM.

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

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

Package Manager

5.1 Architecture

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

At the ten-thousand foot view of the architecture, it consists of three core services, as shown in figure 24:

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

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

Listing 24: Ten-thousand foot view of the JPM architecture

5.2 Registry

Stuff we need to cover in this section:

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

5.3 Authorization

In this section we need to discuss:

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

5.4 The JPM Back End

5.4.1 Introduction and Responsibilities

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.

The most important partner the back end service has is the registry service. The back end will communicate with one or more registry services, which are covered more in depth in Section TODO.

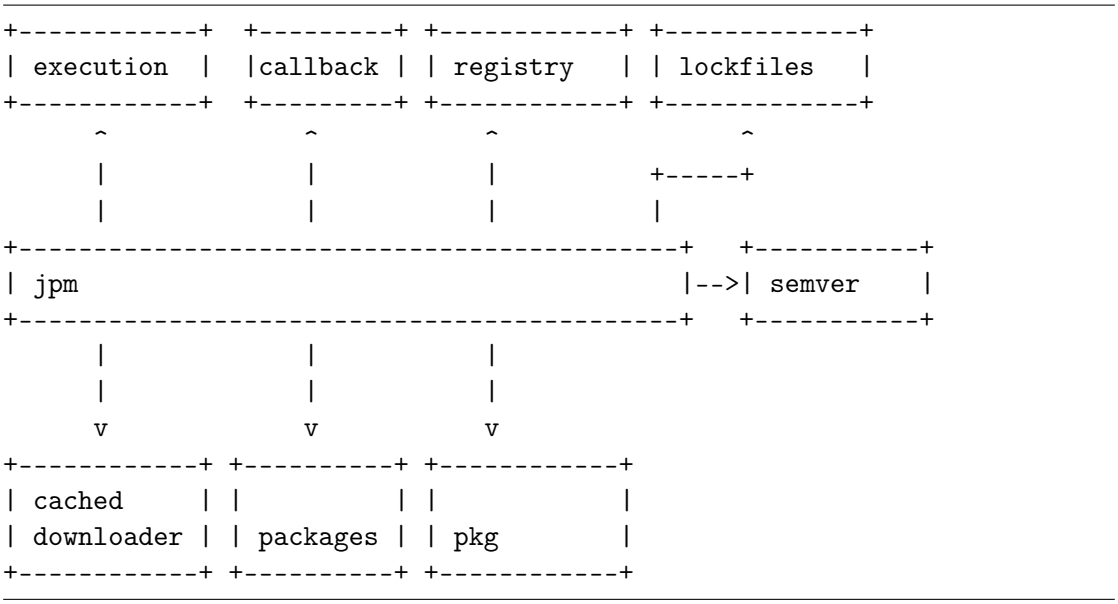
The responsibilities of the back end can roughly be divided into three categories:

1. **Package management (See Section TODO)**
    - Installing packages
    - Publishing packages
    - Upgrading packages
    - Searching for packages
2. **Account management (See Section TODO)**
    - Login/logout with each registry
    - Create user
    - Team management
3. **Helper scripts (See Section TODO)**
    - Creating a new package
    - Clearing the cache
    - Starting a package

5.4.2 Architecture

The JPM back end collaborates with a few external services, these are illustrated in Figure 25.

1. **registry**: Primary collaborator, responsible for storing packages. See Section TODO
2. **execution**: Used for starting package (Section TODO), and executing life-time hooks (Section TODO)
3. **lockfiles**: Handles the lockfiles feature (Section TODO)
4. **semver**: Library package for dealing with semantic versioning (Section TODO)
5. **cached downloader**: Acts as a cache for downloaded packages (Section TODO)
6. **packages**: Validator for the internal package format (Section TODO)
7. **pkg**: Responsible for creating the binary format used for distribution (Section 4.6)



Listing 25: System architecture, from the back end's point of view

5.5 The Command Line Interface

5.5.1 Introduction

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 not work by itself, but will instead delegate this to the back end (See Section TODO).

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 ]
```

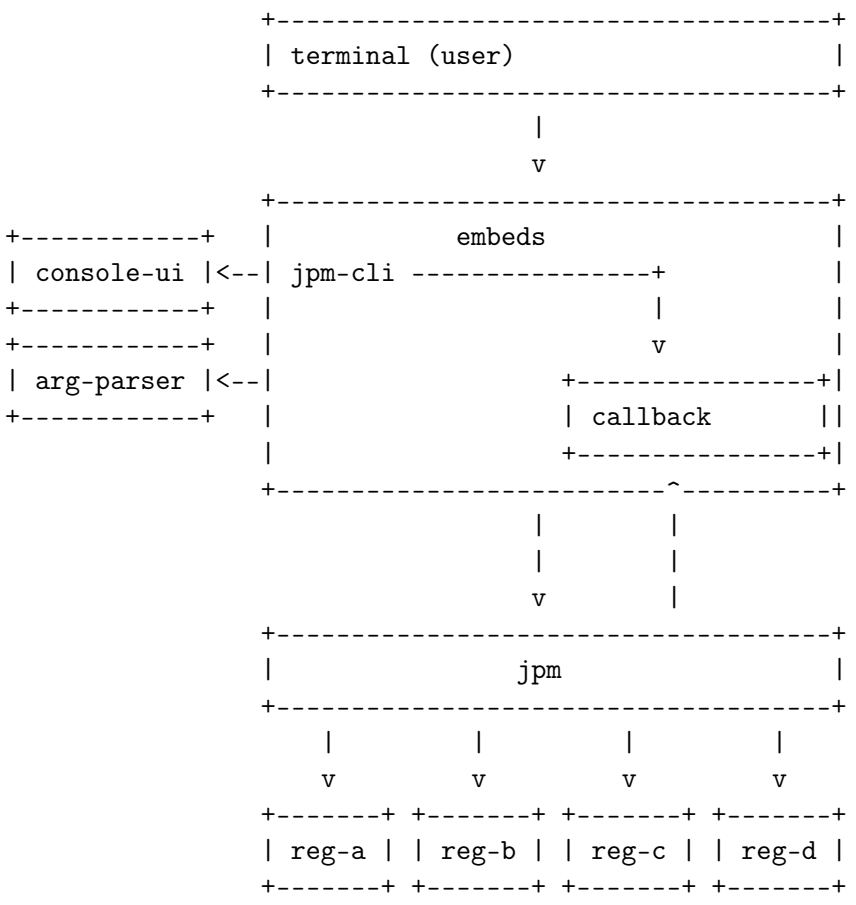
As clearly visible form this snippet, for the tool to do any work we must first give it something to do via a command. The commands that JPM understands almost directly mirror the functionality provided by the back end. To use JPM to create a new package, the user must simply use the `init` command. 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 26.

```
$ 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 26: The `jpm` tool provides a user interface for common tasks. In this example, creating a new package.

5.5.2 Internal Organization and Deployment

The command line tool delegates most of the work to the back end service (here called jpm). Figure 27 shows the architecture from the CLI's point of view. Most noteably 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.



Listing 27: The system architecture from the CLI's point of view

Quite a lot of services exists on the client side, notable `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[2]. This conversion would be relatively straight forward, since the server architecture required by the daemon is already implemented.

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

[2] `https://docs.gradle.org/current/userguide/gradle_daemon.html`

Appendix

A.1 Appendix A: JPM Manifest Specification

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

A.1.1 Purpose

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

A.1.2 Table of Contents

A.1.3 Format and Properties

name

Name: `name`

Optional: false

Type: `string`

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

Rules:

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

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

version

Name: `version`

Optional: false

Type: `string`

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

Rules:

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

license

Name: `property_name`

Optional: false

Type: `string`

Description: Describes the license that this package is under.

Rules:

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

authors

Name: `authors`

Optional: false

Type: `string|array<string>`

Description: Describes the authors of this package

Rules:

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

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

private

Name: `private`

Optional: true

Type: `boolean`

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

Rules:

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

main

Name: `main`

Optional: true

Type: `string`

Description: Describes the main file of a package.

Rules:

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

dependencies

Name: `dependencies`

Optional: true

Type: `array<dependency>`

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

Rules:

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

dependency

Type: `object`

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

name **Name:** `name`

Optional: false

Type: `string`

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

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

version **Name:** `version`

Optional: false

Type: `string`

Description: Describes the version to use

Rules:

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

registry **Name:** `registry`

Optional: true

Type: `string`

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

Rules:

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

registries

Name: `registries`

Optional: true

Type: `array<registry>`

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

Rules:

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

registry

Type: `object`

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

name **Name:** `name`

Optional: false

Type: `string`

Description: This property uniquely identifies the registry.

Rules:

- A name cannot be longer than 1024 characters

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

TODO:

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

location **Name:** `location`

Optional: false

Type: `string`

Description: Describes the location of the registry.

Rules:

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