SDU✧

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

Master Thesis

Building a Package Manager for Jolie

Author:
Dan Sebastian Thrane

Supervisor:
Fabrizio Montesi

April 19, 2017

Contents

Introduction

Introduction goes here.

Background

Background goes here.

Microservices

- General stuff about microservices, actually introduce the thing

Introduction to Jolie

We need to cover:

- Basic syntax
- Control-flow, defines, interfaces, types, type-checking, that sort of thing
- Ports, protocols, that sort of thing
- Basic example (steal this one from article)

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, interfaces, types

Embedded, aggregates, redirection

The Jolie Interpreter

- Values and types

- Phases of the compiler

- Anything we need to explain the changes that were made

Jolie Modules

Modules

- Probably need to cover some problems with the current solution, to give some context to the actual solution

- Present the actual solution of modules

- Cover the implementation.

Configuration

- Introduce why we need it

- Configuration format, profiles, modules

Package Manager

Packages

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

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

In listing 2 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 3.

```
1  {
2      "name": "calculator",
3      "main": "calc.ol",
4      "description": "A simple calculator service",
5      "authors": ["Dan Sebastian Thrane <dathr12@student.sdu.dk>"],
6      "license" "MIT",
7      "interfaceDependencies": [
8          { "name": "math", "version": "1.0.0" }
9      ],
10     "dependencies": [
11         { "name": "sum", "version": "1.2.X" },
12         { "name": "multiplication", "version": "2.1.0" }
13     ]
14 }
```

Listing 2: A Simple Package Manifest

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

Listing 3: 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.

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 4:

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  |
+-----------+     +-------+     +------------+
```
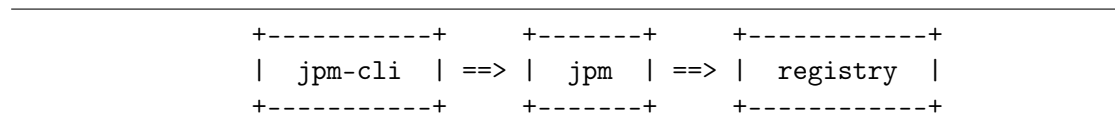
Listing 4: Ten-thousand foot view of the JPM architecture

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.

Authorization

In this section we need to discuss:

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

The Command Line Interface

We need to cover:

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

- Internal organization (commands)

- Technical architecture, and deployment

- Kind of wrap up how the entire thing works here

Integrity Checks

This is a section about integrity checks in JPM.

- Some background about checksums (probably)

- How we do it. Reason we don't go for something like code signing (in pkg manager)

- Code signing, and why we would prefer this.

Appendix

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.

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".

Table of Contents

- Format and Properties
    - name
    - version
    - license
    - authors
    - private
    - main
    - dependencies
    - dependency
        * name
        * version
        * registry
    - registries
    - registry
        * name
        * location

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 accidental 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")