High-Level Modeling and Low-Level Adaption of Serverless Function Choreographies

Benjamin Walch

Bachelor Thesis

Supervisor:
Dr. Shashko Ristov

Department of Computer Science
Universität Innsbruck

Innsbruck, April 9, 2020

Eidesstaatliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.

Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

Datum	Unterschrift

Abstract

The Distributed and Parallel Systems group developed the *Abstract Function Choreography Language* (AFCL). It is a specification for describing serverless workflows. Furthermore, a Java API, to describe serverless application workflows programmatically, was developed. The product which results in using that API is a workflow being described in AFCL in a text based file. Until now, the workflow being described in the text file has to be created manually (by editing the file directly), or a programmer has to write Java code which utilizes the API to generate the file.

The aim of this bachelor project is to develop a visual workflow editor, which makes modeling of workflows possible at a high level of abstraction. The tool should not only be able to load, display and save workflows, but also optimize workflows for multiple FaaS provider(s) in case of quotas and limits, and also in case of performance.

Contents

1 Introduction

"Run code, not Servers" is a recent term in the cloud computing world. With the rise of serverless technologies during the last years, *Function-as-a-Service* (FaaS) became more and more popular. This cloud computing concept offers new advantages for software development: very high flexibility, nearly unlimited scalability and pay-per-use pricing. Global Players like Amazon, Google, IBM and Microsoft jumped on the train and provide their infrastructure to developers, with the ability to deploy and execute their code in the cloud.

Moving from *Infrastucture-as-a-service* (IaaS) over *Platform-as-a-service* (Paas) to Faas, developers can completely focus on the program they need to run and not worry at all about the machine it is on or the resources it requires [7]. An entire program can be defined by a workflow, which in turn consists of functions and control structure. Workflows, though, can be executed using serverless technology in the cloud.

Many FaaS providers offer a tool to create a workflow and execute it (Amazon Step Functions, IBM Cloud Functions, Microsoft Azure Functions, Google Cloud Functions, ...). Most of them also offer the option to export the composed workflow definition into a text file. But each provider has its own definitions on how a workflow is expressed, as well as different pricing models, limits and quotas. Additionally, different providers also support different programming languages. That being said, a workflow from one system is not compatible with another, what means, the user is bound to a specific provider, if he wants to reuse or change a workflow later.

The distributed and parallel systems group developed a specification to describe serverless function choreographies generically. This specification is called *Abstract Function Choreography Language* (AFCL) and was created to eliminate incompatibility of workflows between different providers.

1.1 Objective

Castro et al. mentions that "One of the major challenges slowing the adoption of serverless is the lack of tools and frameworks" [1].

Therefore, this thesis has the goal to implement a new tool providing a graphical user interface, able to model workflows at a high abstraction level and export them in AFCL. In addition, visualizing - in particular loading, displaying and editing - of existing AFCL workflows in should be supported. The tool should also be able to optimize composed workflows.

It is assumed that concrete implementations of the functions are already developed and deployed to a FaaS provider.

1.2 Requirements

General

- design for the overall application [coreUI]
- clean and intuitive UI [bootstrap]
- ability to add components/menu points easily [modularity]
- good user experience [SPA]
- frameworks: open source, large community (future-safe)
- cross-browser [?]
- performance [?]

Libraries

- open-source
- good documentation
- actively maintained
- large and active community (future-proof)

- stars on github / downloads on npmtrends
- performance
- pre-knowledge of the developer / learning curve

2 Background

This chapter contains important details on the terms and technologies used in this thesis and helps the reader to better understand the further topics.
First of all, the serverless concept, in particular Function-as-a-Service, Serverless Functions and Workflows are explained, before giving a short overview about AFCL. The tools and technologies used for the development are also presented to readers who are not yet familiar with them.

2.1 Serverless Computing

Serverless computing is widely known as an event-driven cloud execution model. In this model, the client provides the code and the cloud provider manages the life-cycle of the execution environment of that code. The idea is based on reducing the life span of the program to execute functionality in response to an event. Hence, the program's processes are born when an event is triggered and are killed after the event is processed [4].

Castro et al. define serverless computing as follows: "Serverless computing is a platform that hides server usage from developers and runs code on-demand automatically scaled and billed only for the time the code is running" [1].

The term 'serverless' can be misleading though, as there are still servers providing these backend services, but all of the server space and infrastructure concerns are handled by the vendor. Serverless means that the developers can do their work without having to worry about servers at all [3].

2.2 FaaS

FaaS is a form of serverless computing, which is disrupting the way applications and systems have been built for decades. By abstracting infrastructure provisioning and deployment, user-provided functions can be invoked and executed remotely. The user only has to worry about development and triggering of the function. This serverless runtime has not only the advantage that it avoids costly pre-allocated or dedicated hardware, but also offers almost unlimited possibilities in scalability and billing.

2.3 Serverless Functions

Serverless functions - or tasks - are single-purpose, mostly stateless, programmatic functions that are hosted on cloud infrastructure. These infrastructure is provided through a FaaS platform. They are event-driven and can be invoked through the internet, mainly using HTTP. Like conventional functions, they accept arguments and and return the result of the computation - with the difference that this is done over a network.

2.4 Serverless Workflows

On an abstract level, a workflow consists of a set of interdependent tasks that need to be executed to achieve a specific goal. [...] A task within a workflow has the following properties: dependencies on the software or service used by the task to perform its computation (software flow), dependencies on data (data flow), and dependencies on other tasks (control flow) [2].
Basic and advanced workflow control-flow patterns for workflows are shown in [6].

A serverless workflow is a complex workflow defined through the composition of serverless functions. Serverless workflows make it possible to combine and reuse serverless functions in order to build more complex applications. Workflows can declare the structure of applications - and using formats like YAML, XML or JSON, they can be described in a text-based file.

2.5 AFCL

The *Abstract Function Choreography Language* (AFCL) was created to eliminate incompatibility of workflows between different providers. As the name indicates, AFCL describes workflows in an abstract, provider independent way. For example, if a workflow exported from AWS should be executed on IBM, the workflow has to be ported to be compatible, what means it has to be recreated - for example using the exported workflow as a template.

This specification forms a step on the way to cross-cloud workflow execution. To be able to execute multi-cloud workflows described in AFCL, some more tools are needed: an Enactment Engine and a Scheduler[1].

2.5.1 Supported Control-Structures

2.6 Development Tools and Technologies

2.6.1 npm

The node package manager[2] is the world's largest software registry, where open-source software packages of developers and companies are shared all over the world. Over the last years, npm became a de-facto standard for package management in JavaScript development.

2.6.2 webpack

webpack is a module bundler, its main purpose is to bundle JavaScript code for the usage in a browser.[3] In particular, multiple modules (often hundreds of) with dependencies [to each other] are processed and bundled into a few files. To be able to process other types of files than JavaScript or JSON, webpack offers the opportunity to configure a **loader**. In this application, the following loaders are configured:

[1]https://dps.uibk.ac.at
[2]https://www.npmjs.org
[3]https://webpack.js.org

- babel-loader, to transform ECMAScript and React JSX to browser-compatible JavaScript
- sass-loader, to transform SASS to CSS
- css-loader, to transform CSS to CommonJS
- file-loader, to handle static resources like images and fonts

2.6.3 ECMAScript

The scripting language specification ECMAScript (ES) was created to standardize JavaScript. With the release of ES6 (also known as ECMAScript 2015), features like class declarations, module imports and arrow function expressions became possible. After ES6, every year a new edition of the ECMAScript standard was finalized and released, offering new features. Worth mentioning here is the rest/spread operator released with ES9, which occurs a lot in the source code of this thesis.
Since current browsers only have partial support of ECMAScript, a 'transcompiler' (or transpiler) is needed to transform the ECMAScript source code to JavaScript, which common browsers are capable of interpreting. This compiling process is done with Babel[4], which is configured to not only compile ECMAScript, but also JSX to JavaScript.

2.6.4 SASS

CSS, in its pure form, reaches its limits when one thinks about using variables, functions or nested rules. SASS[5] is a stylesheet language, which is - similiar to ES - compiled to CSS and offers the mentioned and even more features. A lot of CSS Frameworks also provide their source code in SASS, often served with a large variable set which makes it easy to customize it. The transpiler used in this thesis to transform SASS to CSS is node-sass[6].

[4]https://babeljs.io
[5]https://sass-lang.com
[6]https://github.com/sass/node-sass

3 Implementation

The Implementation of a Graphical User Interface to visualize and create or edit workflows, as well as a proof-of-concept for workflow optimization were the main goals of this bachelor thesis.

3.1 Architecture

Tthe application consists of two sub systems which are operating mostly decoupled from each other: a Java backend and a web-based frontend. Those communicate and share JSON encoded objects between each other through an API the backend provides and the frontend consumes. The backend is responsible for converting and optimizing workflows at low-level as well as persisting of function data. The conversion of workflows is needed because the frontend encodes the visual representation of a workflow in XML, which is given as input to the backend. The conversion process is done by utilizing the AFCL API[1].

The frontend displays and controls the application's user interface. It is responsible for visualizing the data and provide the canvas where the user can compose a workflow.

[1]http://dps.uibk.ac.at/projects/afcl/files/jar/afclAPI.jar

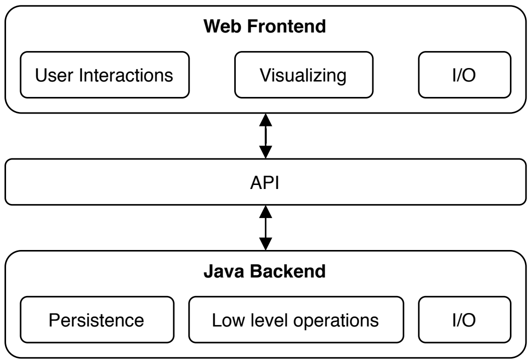


Figure 3.1: application architecture

3.2 Frontend

Since a web based frontend which runs in the user's web browser, is a hard requirement, the core technologies are limited to HTML, CSS and JavaScript.

3.2.1 Setup

The setup of the frontend application is a selection of tools and technologies for modern and complex web development. The base forms npm which is used to manage and resolve the dependencies. Additionally, npm's CLI is used as tool to execute development and build tasks. Webpack is in use to bundle all sources and assets into single files, and execute additional build steps by utilizing so-called 'loaders'.

Under the hood, JavaScript and several JavaScript Frameworks are in action to control the application consisting of several UI components. To give the application its look and feel, CSS and some CSS Frameworks are in use.

Babel is used as such a webpack loader, for compiling the ECMAScript and React JSX source code to browser-compatible JavaScript. The same applies for the CSS extension SASS, which is also integrated as a webpack loader, and is used to make the process of styling the user interface more efficient.

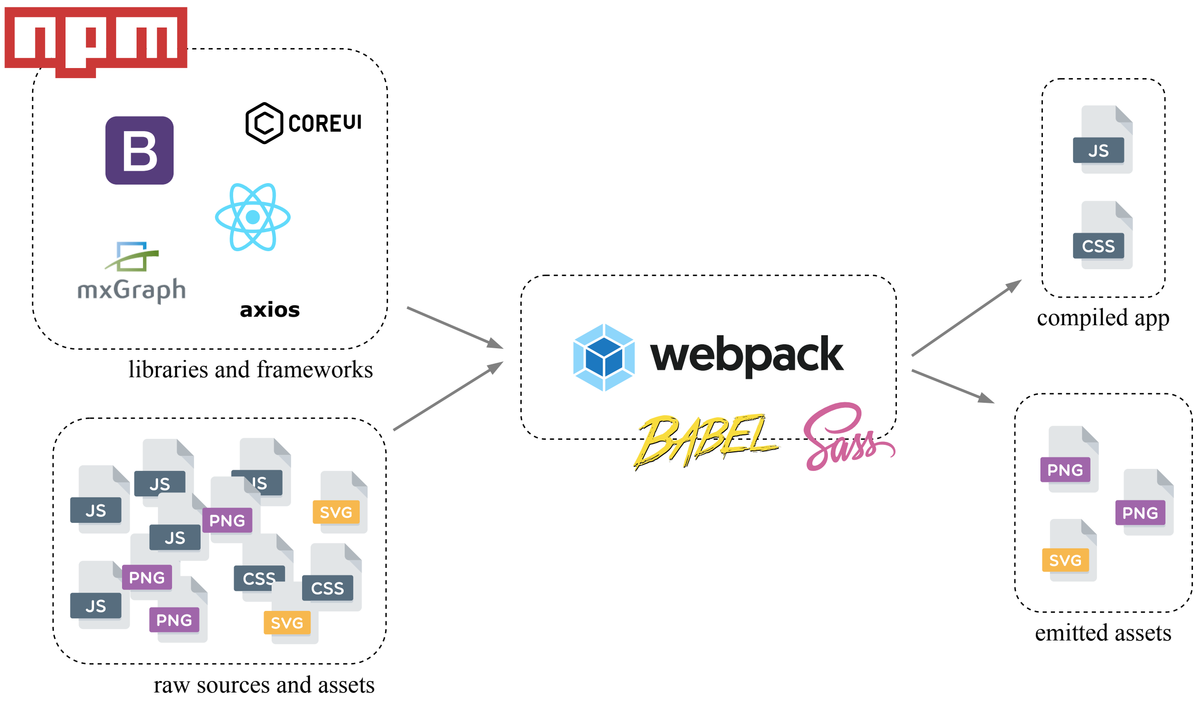


Figure 3.2: frontend development toolchain

3.2.2 Data model

All AFCL Java model classes have been ported to ECMAScript, to have all classes and properties available in the frontend, similiar like they are in the backend. Although plain JavaScript objects would also do the job, this adds a kind of type-safety to the ECMAScript sources and improves readability of the code. Also, the data exchange with the backend and the encoding of the graph benefit from this approach, since the XML node names map to constructor names.

3.2.3 Graph drawing

A workflow can be visualized as a directed acyclic graph (DAG). For reasons of efficiency, reusability, documentation and future safety, there is no question that an existing library has to be used to do the visualizing part. At the time of implementation, the JavaScript library mxGraph[2] was the best choice[3] to achieve the mentioned goals. It has an outstanding documentation, enriched with a lot of demos and example code which show many use cases and extensive features. The authors of mxGraph state:

> "mxGraph is pretty much feature complete, production tested in many large enterprises and stable for many years. We actively fix bugs and add features [...]."

In mxGraph, vertices and edges are represented by an `mxCell` model, which stores information about the cell's position, dimension, geometry and style.

[2]https://jgraph.github.io/mxgraph/

[3]see 3.2.6 to learn why

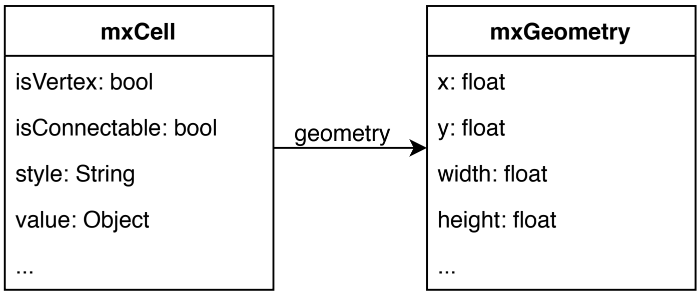


Figure 3.3: `mxCell` model

Additionally, a *user object* can be associated with a `mxCell` through the `value` property. User objects give the graph its context, they store the business logic associated with the visual part. [5] For simple cases, the user object may be a string to display the cell's label. In more complex applications, the user object may be an object and some property of that object will generally be the label that the visual cell displays. In this application, the cell's user objects are instances of the ported AFCL classes, respectively.

The following table shows the `mxCell`'s shapes which are implemented and used for composing a workflow. Beside a short description, the type of the associated user object is denoted.

Cell	Description	User Object
● Start	Defines the workflow's entry point	`String` (for the label)
○ End	Defines the workflow's termination	`String` (for the label)
▬ Function	An atomic function	`AtomicFunction`
◆ If-Then-Else	A mutual exclusive condition	`IfThenElse`
◆ Switch	A multi condition	`Switch`
▼ Merge	For merging previous branches	`null` (None)
⬚ Parallel	A container, where each child cell is meant to be executed in parallel.	`Parallel`
⬚ ParallelFor	A container, where each child cell is meant to be executed in a parallel for loop.	`ParallelFor`

3.2.4 Validation

Graph validation is implemented at two different stages: At runtime, when composing a workflow by adding and connecting the vertices, and before export, when saving the current graph. In AFCL, a valid workflow represented by a DAG always fulfills the following constraints:

- There exists only one starting point
- Start has only one outgoing edge
- There exists only one termination point
- End has only one incoming edge
- There exist no cycles
- Functions have only one incoming and only one outgoing edge
- IfThenElse have exactly two outgoing edges (then, else)
- ...

3.2.5 Encoding

The visual representation of the workflow and the included data targets human-readability. This data needs to put into another, proper format for storing. mxGraph provides built-in support for converting the visual representation to XML. An advantage of using XML over JSON is that constructor names (class names) are mapped to the XML node names, no additional property is required. XML is mostly convenient for the backend, since Java has excellent built-in XML support. The XML-encoded workflow is converted on backend-side internally to a AFCL object.

3.2.6 Layouting

Combination of nested hierarchical layout

3.2.7 Alternatives

One of the more difficult tasks was to choose a proper library for graph drawing. Since the graph drawing and visualization is one of the most important features of the frontend, a careful choice has to be made. Of course there exist multiple excellent JavaScript graph drawing libraries.

Below is a list of tested and researched JavaScript graph libraries with additional information why they were insufficient for the project.

- **jsPlumb** is a library of great quality and looks like it would fulfill all requirements for graph drawing and user interaction out of the box. It has a very good documentation, and a lot of examples. Furthermore its animations and drag and drop handling are outstandingly good. Unfortunately this software is not open source and a license costs a few thousand dollars.
- The same goes for **Rappid** (formerly known as JointJS) and **GoJS**, which is even more expensive.
- **diagram-js** is one of the newer diagram libraries. It seems to be a good candidate to draw BPMN[4] graphs. The lack of an appropriate documentation - even after 4 years of existence (there exists an open issue on github[5] for this) - is still an issue.
- **Cytoscape.js** is older than mxGraph but the development and the community is not less active. This library has a very good documentation and a lot of demos are provided. But it offers fewer examples and "out-of-the-box" features for modeling flowcharts than mxGraph does.
- **vis.js** is a star on npmtrends.[6] It has very smooth animations when manipulating the graph. There exists only one example of graph manipulation on its documentation, it is mainly used for visualization only.
- **Sigma js** is a tiny library with a small documentation. It clearly focuses on displaying and visualizing graphs, not on modeling them. This part would have to be developed manually. Furthermore, the last activity on this project on github was 2 years ago.

[4]http://www.bpmn.org
[5]https://github.com/bpmn-io/diagram-js/issues/78
[6]https://www.npmtrends.com/vis

3.2.8 Web Interface

The layout of the web interface is based on coreUI[7], an admin panel template, built on top of the web toolkit Bootstrap[8]. A clean and nested structure of components, which is typical for a React app, guarantees modularity and forms the whole frontend application. One of the benefits when using React is that the data model displayed to the user maps most of the time nicely to UI components. Basically, the components of the application respect the single responsibility principle[9]. The main component has four sub-components - where each of them consists again of multiple sub-components - representing the core features of the app.

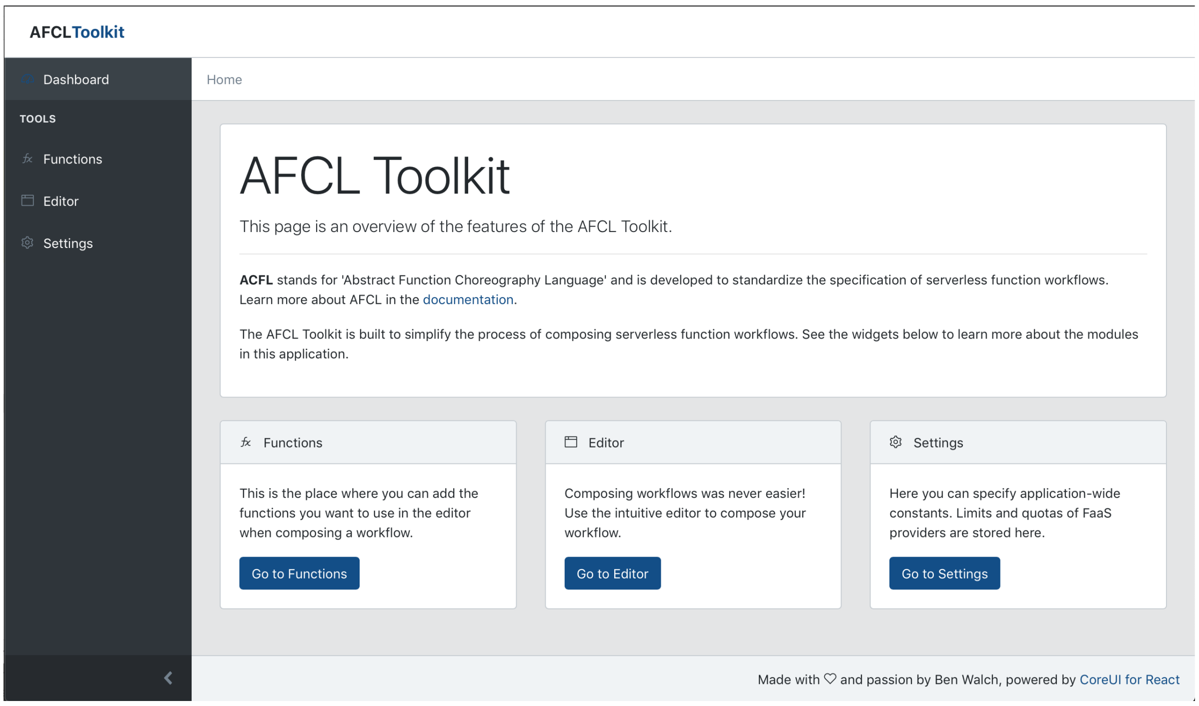
Dashboard

The dashboard is the entry point where the user lands after accessing the app. The purpose of this component is to give the user a quick overview of the application, provide short informational texts and links to the specific modules.

[7]https://coreui.io

[8]https://getbootstrap.com

[9]https://en.wikipedia.org/wiki/Single_responsibility_principle



AFCL Toolkit

This page is an overview of the features of the AFCL Toolkit.

ACFL stands for 'Abstract Function Choreography Language' and is developed to standardize the specification of serverless function workflows. Learn more about AFCL in the documentation.

The AFCL Toolkit is built to simplify the process of composing serverless function workflows. See the widgets below to learn more about the modules in this application.

Functions

This is the place where you can add the functions you want to use in the editor when composing a workflow.

Go to Functions

Editor

Composing workflows was never easier! Use the intuitive editor to compose your workflow.

Go to Editor

Settings

Here you can specify application-wide constants. Limits and quotas of FaaS providers are stored here.

Go to Settings

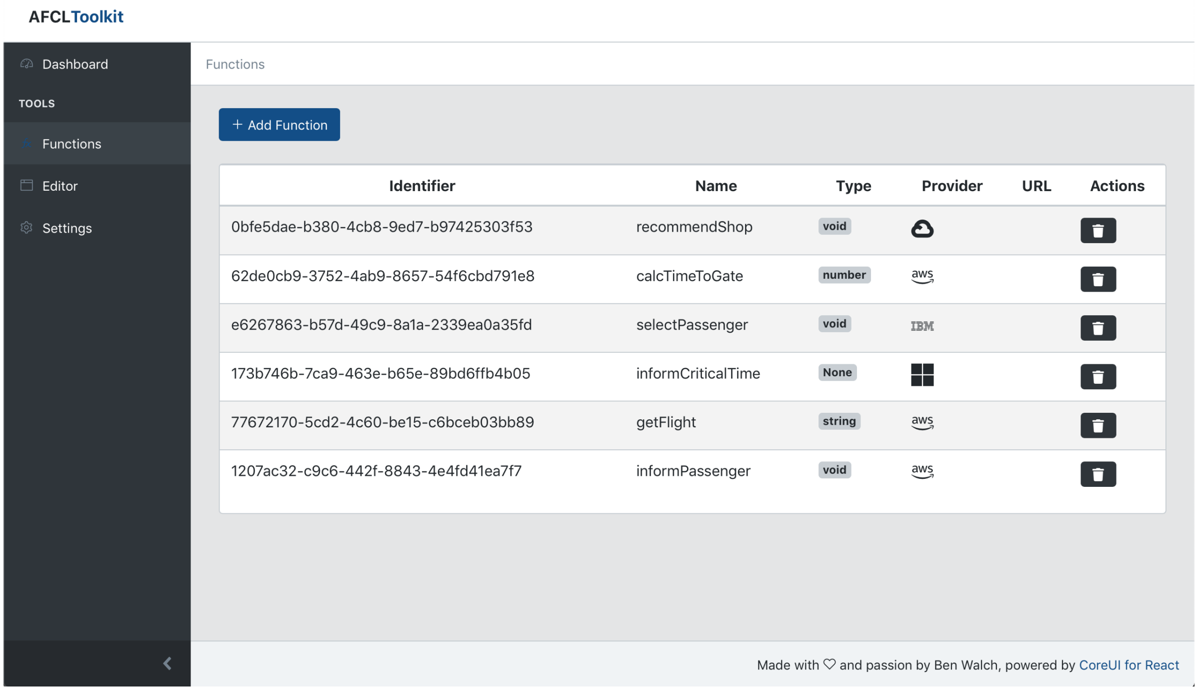
Made with ♡ and passion by Ben Walch, powered by CoreUI for React

Figure 3.4: the dashboard

Functions

A function repository is needed to know about available functions when composing a workflow. An item in the function repository holds the following data: `name`, `type` and `provider`. In the implementation of this thesis, this data is persisted to an internal file.[10]

[10]see 3.3.4 for detailed information on how data is persisted



Identifier	Name	Type	Provider	URL	Actions
0bfe5dae-b380-4cb8-9ed7-b97425303f53	recommendShop	void	☁		🗑
62de0cb9-3752-4ab9-8657-54f6cbd791e8	calcTimeToGate	number	aws		🗑
e6267863-b57d-49c9-8a1a-2339ea0a35fd	selectPassenger	void	IBM		🗑
173b746b-7ca9-463e-b65e-89bd6ffb4b05	informCriticalTime	None	⊞		🗑
77672170-5cd2-4c60-be15-c6bceb03bb89	getFlight	string	aws		🗑
1207ac32-c9c6-442f-8843-4e4fd41ea7f7	informPassenger	void	aws		🗑

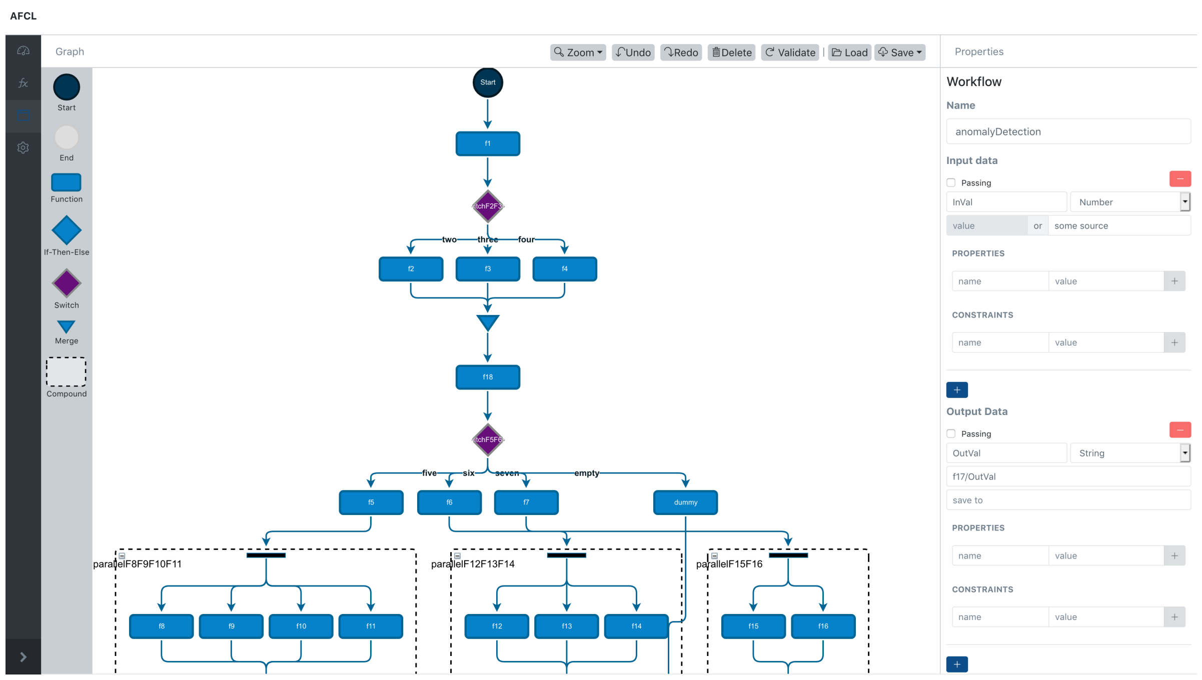
Made with ♡ and passion by Ben Walch, powered by CoreUI for React

Figure 3.5: the function repository

Editor

The interface of the component responsible for composing workflows is divided into two parts. On the left side is the editor, consisting of a toolbar and a drawing area, on the right side is the property view. In the editor, the user can choose a function from the toolbar to place it in the drawing area below. For the sake of clarity, each function has a defined shape and style. In the drawing area, these shapes (functions) can then be connected to each other by drag and drop. This leads to a directed graph, which represent the execution flow. The possible connection points (ports) of a shape will be displayed when hovering over it. These ports are input and output for most of the functions. When hovering a port, the port name appears in the tooltip.

When selecting a shape in the canvas, the property view changes. It displays each specific property of the selected function, with the ability to change, add or remove properties.



Workflow

Name: anomalyDetection

Input data
- Passing
- InVal | Number
- value or some source
- PROPERTIES: name | value
- CONSTRAINTS: name | value

Output Data
- Passing
- OutVal | String
- f17/OutVal
- save to
- PROPERTIES: name | value
- CONSTRAINTS: name | value

Figure 3.6: the editor

Settings

3.3 Backend

3.3.1 Setup

Maven, Servlets

3.3.2 Api

RESTful (Functions) and RPC-Style (Workflows)

3.3.3 Decoding

XML is good because built-in XPath support in Java.

3.3.4 Persistence

Composed workflows are saved to the user's file system by offering a download. The user can then choose the target location. The storing of function data, on the other hand, is abstracted from the user and persisting of the data is handled internally. An interface[11] for the repository was created on the backend side to be capable of supporting any data source. The repository which implements the interface in this thesis, stores the serialized data into a file. This could be easily replaced with any other implementation, for example an implementation which stores the data in a DBMS.

```java
public interface Repository<T> {

    public Collection<T> findAll();

    public T findOne(String id);

    public void add(T obj);

    public void remove(T obj);

    public void remove(String id);

    public void update(T obj);
}
```

Listing 3.1: Repository Interface

3.3.5 Optimization

3.3.6 Improvements/Known Issues

[11]https://docs.oracle.com/javase/specs/jls/se7/html/jls-9.html

4 Evaluation

5 Conclusion

Bibliography

[1] Paul Castro et al. “The Rise of Serverless Computing”. In: *Commun. ACM* 62.12 (Nov. 2019), pp. 44–54. ISSN: 0001-0782. DOI: `10.1145/3368454`.

[2] Erwin van Eyk. “The design, productization, and evaluation of a serverless workflow-management system”. 2019.

[3] Cloudflare Inc. *What Is Serverless Computing?* URL: `https://www.cloudflare.com/learning/serverless/what-is-serverless/` (visited on 03/08/2020).

[4] Ali Kanso and Alaa Youssef. “Serverless: Beyond the Cloud”. In: *Proceedings of the 2nd International Workshop on Serverless Computing.* WoSC ’17. Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 6–10. ISBN: 9781450354349. DOI: `10.1145/3154847.3154854`.

[5] JGraph Ltd. *mxGraph User Manual.* URL: `https://jgraph.github.io/mxgraph/docs/manual.html` (visited on 04/07/2020).

[6] Nick Russell et al. *Workflow Control-Flow Patterns: A Revised View.* 2006. URL: `http://workflowpatterns.com/documentation/documents/BPM-06-22.pdf`.

[7] Neil Savage. “Going Serverless”. In: *Commun. ACM* 61.2 (Jan. 2018), pp. 15–16. ISSN: 0001-0782. DOI: `10.1145/3171583`.