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

Benjamin Walch

Bachelor Thesis

Supervisor:
Dr. Shashko Ristov

Department of Computer Science

Universität Innsbruck

Innsbruck, May 2, 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 [8]. An entire program can be defined by a workflow, which in turn consists of functions, data dependencies and control structure. Workflows, though, can be executed using serverless technology in the cloud.

But each provider has its own definitions on how a workflow is expressed, as well as different pricing models, limits and quotas. One main issue is the maximum number of concurrent function invocations, which is not more than 1000 for current FaaS providers at time of writing[1]. 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 (vendor lock-in), if he wants to reuse or change a workflow later.

[1]checked providers: AWS, Google, IBM and Microsoft

The distributed and parallel systems group developed a specification to describe serverless workflows at a high abstraction level. This specification is called *Abstract Function Choreography Language* (AFCL) and was created to overcome weaknesses of current FaaS platforms; for example incompatibility of workflows between different providers. With the development of the AFCL specification, also a Java API was developed, which focuses on creating valid AFCL workflows from Java code.

Although many FaaS providers offer a tool to simplify creation of workflows, this is still a complex task, which also applies to AFCL workflows. Most of the time, a software developer is needed, because others simply lack the neccessary know-how; but even for developers, this task can be tedious and time-consuming.

Castro et al. mentions that "One of the major challenges slowing the adoption of serverless is the lack of tools and frameworks" [2]. Therefore, the goal for this thesis was to implement a new tool providing a system, able to model workflows at a high level of abstraction and export them in AFCL. In addition, visualizing - in particular loading, displaying and editing - of existing AFCL workflows in should be supported. The system should also be able to adapt composed workflows at low-level.

This thesis fulfills above goals and overcomes mentioned problems. With the developed system, which we call "AFCL ToolKit" in the further text, it is possible for non-developers (business guys) to develop a workflow in less time it would take a developer to write the workflow in AFCL. Another advantage is that exported workflows are compliant with the API and the risk of typos is minimized. Regarding provider limitations, a scheduler (which is not part of this thesis) should decide to distribute one big concurrent loop - with more iterations than the maximum number of function invocations allowed by the provider - onto multiple FaaS providers. For this purpose, a service is provided which offers this low-level adaptation.

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

This thesis is structured in three parts. At the beginning there is a short introduction to Faas, followed by explanations and additional information about AFCL.

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 [5].

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" [2].

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 [4].

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) [3].
Basic and advanced control flow patterns for workflows are shown in [7].

A serverless workflow is a complex workflow defined through the composition of serverless functions, connected by control- and data flow. 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

Below, a quick overview to AFCL and the control-flow constructs will be given. More detailed information about AFCL as well as the API documentation can be found at [1].

The *Abstract Function Choreography Language* (AFCL) was created to overcome weaknesses in current FaaS platform implementations. As the name indicates, this specification describes serverless workflows at a high level of abstraction, forming a step on the way to cross-cloud workflow execution.
Consider a workflow built with AWS Step Functions, should be executed on IBM Cloud. To make that work, the workflow has to be ported to the other platform to be compatible. In other words - it has to be recreated by a skilled programmer - for example by using the exported workflow as a template.

The AFCL specification also has more extensive control-flow and data-flow constructs available than current FaaS providers offer currently in terms of language features. Therefore, AFCL can not only eliminate the incompatibility of workflows between different providers (vendor lock-in), but also opens the door to execute a workflow by distributing its computation over multiple FaaS providers.

To be able to actually execute such multi-cloud workflows, some more tools are needed: an Enactment Engine and a Scheduler.

2.5.1 Overview

AFCL is based on YAML[1] and ships with a schema[2]. There exist two types of functions, base functions and compound functions, which can be connected by specifying control-flow and data-flow information. While base functions represent a single task, compound functions provide nesting - they can include some base functions or even other compound functions.

Base and compound functions have data input (dataIns) and data output (dataOuts) ports to define input or output data, respectively. Data input can refer to another data output or data input of another function by specifying the name in combination with the data ouput port of the other function.

2.5.2 Control-flow

In AFCL, base or compound functions are specified one after another, which means that they are executed sequential. However, the following control constructs are introduced with AFCL: `sequence`, `if-then-else`, `switch`, `for`, `while`, `parallel` and `parallelFor`.

2.5.3 Properties and Constraints

Additional, optional attributes for `dataIns` and `dataOuts` ports of a function, or for the function itself can be defined in `properties` and `constraints`. While those are simple key-value pairs accepting string values, AFCL has a few defined properties and constraints to specify concrete attributes like invocation type of a function, element index or data distribution information in loops.

[1]https://yaml.org

[2]http://dps.uibk.ac.at/projects/afcl/files/schema/schema.yaml

2.5.4 Data-flow

AFCL allows to express data-flow by connecting source data ports of functions to target data ports of functions. This offers support for more complex data-flow scenarios and might improve performance of the workflow. A source data port can be the input data port of the whole workflow, or a data port (input or output) of another function. So it is possible to even connect data ports of outer functions with a a lower nesting level to inner functions with a higher nesting level.

2.6 Tools and Technologies

2.6.1 npm

The *node package manager*[3] 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.[4] 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:

- babel-loader, to transform ES and React JSX to browser-compatible JavaScript
- sass-loader, to transform SASS to CSS

[3]https://www.npmjs.org
[4]https://webpack.js.org

- 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 was used a lot in this thesis.

2.6.4 Babel

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. Babel[5] is an industry standard to transpile ES to common JavaScript or lower ES versions.

2.6.5 SASS

CSS, in its pure form, reaches its limits when one thinks about using variables, functions or nested rules. *SASS*[6] 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.

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

3 Implementation

This chapter gives the reader a detailed overview over the system architecture as well as the system design.

AFCL ToolKit should not only visualize AFCL Workflows, but also give non-developers the opportunity to create and edit workflows through a Graphical User Interface (GUI). This GUI ships with a powerful editor, including additional features like real-time validation, versioning (change history) and automatic layouting.
Furthermore, workflows can be adapted at low-level in order to split large parallel loops - for example loops with more iterations than allowed by current providers - into several loops running concurrently. By doing this, AFCL ToolKit also adapts all existing data flow to the new concurrent structure.

3.1 System Architecture

AFCL ToolKit is built on top of two sub systems which are operating mostly decoupled from each other. On the one hand there is the backend service, responsible for low-level operations on workflows, on the other hand is the frontend with the GUI. Those interact with each other by sharing JSON-serialized objects through an API the backend provides and the frontend consumes.

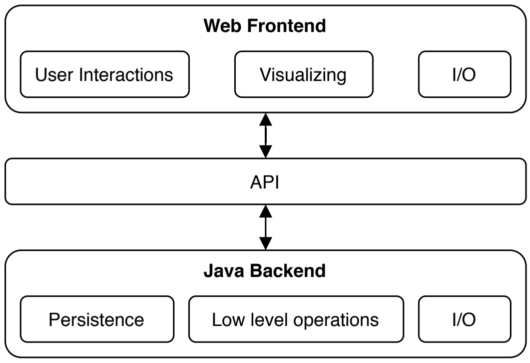


Figure 3.1: system architecture

3.1.1 Frontend

The frontend displays and controls the application's user interface. Its core component is the workflow editor, which is responsible for visualizing workflow data and provides the canvas with all additional control elements, allowing the user to compose and edit workflows. All features are described in detail in section 4

3.1.2 Backend

The backend is responsible for performing operations on workflows at low-level. The conversion of XML data from the frontend as well as persisting of function data is also a backend task.

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.

3.2 System Design

This section describes the design of the implemented application.

3.2.1 Frontend

The frontend is built as a web application with HTML, CSS and JavaScript. The JavaScript library *React* is primarily used to build the user interface.

The base forms *npm*, which is used to manage and resolve dependencies. *Webpack* is the main part which orchestrates transpiling and bundling all sources and assets into single files by utilizing so-called *loaders*.

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

webpack also executes some optimization plugins when building for production, which minimizes the size of the final bundle.

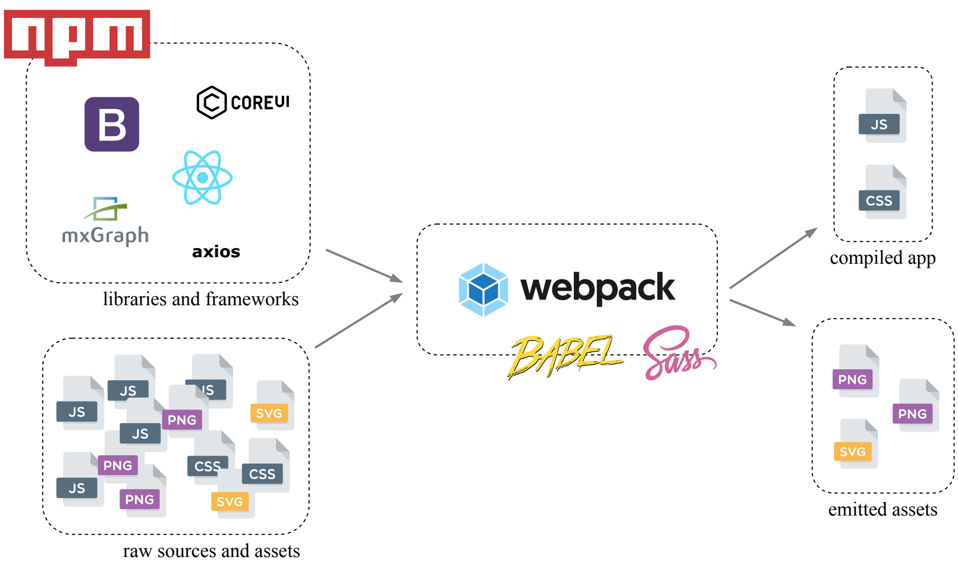


Figure 3.2: frontend stack

Libraries

Below is a list of selected libraries in this project. Care has been taken to ensure that all libraries used are open source.

- **React**
  React is an JavaScript library for building user interfaces.[1]

- **mxGraph**
  mxGraph is a JavaScript diagramming library.[2]

- **axios**
  Axios is a JavaScript library providing a promise based HTTP client for networking tasks.[3]

- **Bootstrap**
  Bootstrap is a toolkit for building web applications.[4].

- **CoreUI**
  CoreUI is a free admin panel template, based on bootstrap.[5]

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.

---

[1] https://reactjs.org
[2] https://jgraph.github.io/.mxgraph/
[3] https://github.com/axios/axios/
[4] https://getbootstrap.com/
[5] https://coreui.io/

Graph drawing

A workflow can be represented as a directed acyclic graph (DAG). DAG's are conventional models to present workflows, where nodes are tasks and edges are communications between tasks.

At the time of implementation, the JavaScript library mxGraph was the best choice[6] for the graph visualization part. It has an outstanding documentation, enriched with a lot of demos and example code which show many use cases and extensive features, as well as a powerful production-grade example.[7]
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.

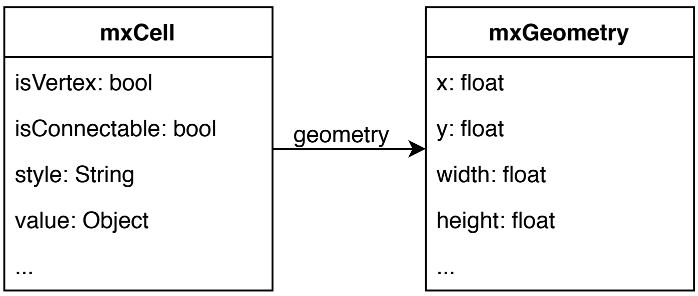


Figure 3.3: `mxCell` model

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

[6] see 3.2.1 to learn why
[7] https://draw.io

visual part. [6] 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 implemented `mxCell` shapes which are used as graphical elements representing a node in the graph. Beside a short description, the type of the associated user object is denoted.

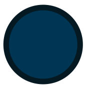
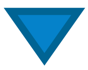
Cell	Description	User Object
 Start	Defines the workflow's entry point	`String` (for the label)
End	Defines the workflow's termination	`String` (for the label)
 Function	An atomic function	`AtomicFunction` *
 If-Then-Else	A mutual exclusive condition	`IfThenElse` *
 Switch	A multi condition	`Switch` *
 Merge	An element 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` *

Table 3.1: Implemented Shapes, *: AFCL Object

Validation

A DAG which represents a *valid* AFCL workfow always fulfills the following constraints:

- It exists exactly one Start
- It exists exactly one End
- It exist no cycles
- Every Cell has a path to Start
- Every Cell has a path to End
- Function, If-Then-Else, Switch, Parallel, ParallelFor and End have exactly one incoming edge
- Start, Function, Parallel, ParallelFor and Merge have exactly one outgoing edge
- If-Then-Else has exactly two outgoing edges (then, else)
- Switch has multiple outgoing edges
- Merge has multiple incoming edges

These constraints are enforced at two different stages: At modeling-time, when composing a workflow, more specificially when adding and connecting vertices, and before saving or adapting a workflow. Due the fact that the user cannot even create an invalid workflow, efficiency increases and errors in the corresponding backend services will be minimized.

Encoding

The visual representation of the workflow and the included data targets human-readability. This data needs to be 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. What makes XML also very convenient is Java's excellent native support for XML and XPath on the backend side. The XML-encoded workflow is converted on backend-side internally to an AFCL `Workflow`.

Layouting

Combination of nested hierarchical layout

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, but not all of them are suitable for the requirements of this project.

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 closed-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 graphs.[8] The lack of an appropriate documentation - even after four years of existence (there exists an open issue on github[9] 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.[10] 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 two years ago.

3.2.2 Backend

The backend of the application is built entirely with Java, using Maven as package manager and Apache Tomcat as Servlet Implementation.

Api

The API to interact with the frontend or other clients is based on HTTP and it is inspired by the principles of REST and RPC.

[8] http://www.bpmn.org

[9] https://github.com/bpmn-io/diagram-js/issues/78

[10] https://www.npmtrends.com/vis

RPC-based APIs are great for actions (that is, procedures or commands). REST-based APIs are great for modeling your domain (that is, resources or entities), making CRUD (create, read, update, delete) available for all of your data. [9]

Function data is stored on server-side, thus its API is a REST-based implementation. Workflows are stored in files on a user's disk and workflow adaptation is a "remote command" which just sends data to the server and asks it to process and return the result. Therefore the API to adapt workflows is RPC-based. Both use JSON as format to receive and return data.

Decoding Frontend Data

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

Persistence

The storing of function data, on the other hand, is abstracted from the user and persisting of the data is handled internally by a repository. An interface for the repository was created on the backend side to be capable of supporting any data source.[11] 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

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

Adaptation

The adaptation service, which has been developed in context of this thesis, is designed to be as generic as possible, such that certain functions can be reused later or in other projects for any other kind of workflow modifications.

In the following paragraphs of this section, terms highlighted with a mono-spaced font refer to classes or class properties of the AFCL Java API.

The AFCL Java API in its current state provides classes which define function and control-structure objects, as well as data-flow objects of an AFCL workflow. They can be arranged and nested in a tree structure in order to model the execution flow.

However, the API has some weaknesses. One is that a `Function` does not have a reference to its *parent object*. The *parent object* can be either a `Workflow` or `Compound`, which has its own implementation to access its enclosed functions - there is no common super class or interface providing a common method. The same problem occurs on `DataIns` `DataOuts`, and `DataOutsAtomic`, they do have properties (e.g. `source`) in common, but each of them has its own implementation to access them. This is also the case for `AtomicFunction` and `Compound` and the `dataIns` property.
Furthermore, tasks like iterating over all `Function` objects in a `Workflow` is not supported.

These limitations make it challenging to perform automated modifications and keep the code generic. During development, it turned out that the following tasks were required frequently while modifying a workflow programmatically. We define these task as *general adaptation tasks*:

- get a `Function` by its name
- get the *parent object* of a Function
- get the `List` which contains a given `Function`
- get all `DataIns` and `DataOuts` which use a given `Function` as source

- traverse a `Workflow`, in particular, iterate over all `Function` objects inside a `Workflow`

The generic requirement was achieved one the one hand by using *Reflection*, on the other hand by developing a generic *traverse function*, shown in listings 3.2 and 3.3.

```java
public static void traverseWorkflow(Workflow wf, BiConsumer<Function, Object> consumer) {
    traverseFunctions(wf.getWorkflowBody(), consumer, wf);
}
```

Listing 3.2: Traverse workflow

```java
public static void traverseFunctions(List<Function> functionsList, BiConsumer<Function, Object> consumer, Object currentParent) {
    if (functionsList == null) {
        return;
    }
    for (Function fn : functionsList) {
        consumer.accept(fn, currentParent);
        if (fn instanceof IfThenElse) {
            traverseFunctions(((IfThenElse) fn).getThen(), consumer, fn);
            traverseFunctions(((IfThenElse) fn).getElse(), consumer, fn);
        }
        if (fn instanceof Switch) {
            for (Case c : ((Switch) fn).getCases()) {
                traverseFunctions(c.getFunctions(), consumer, fn);
            }
        }
        if (fn instanceof Parallel) {
            for (Section s : ((Parallel) fn).getParallelBody()) {
                traverseFunctions(s.getSection(), consumer, fn);
            }
        }
        if (fn instanceof ParallelFor) {
            traverseFunctions(((ParallelFor) fn).getLoopBody(), consumer, fn);
```

```
        }
    }
}
```

Listing 3.3: Traverse functions

As the reader may have noticed, the traverse function accepts Java's `BiConsumer` as second argument, which makes it very versatile in its usage. The given consumer operation - which can be a function reference or a lambda expression - is executed for every function element in the workflow, providing the element itself and its *parent object* as arguments on traversal. This is indeed very powerful, reduces LOC, therefore improves readability, maintainability and efficiency. The only drawback is that all functions are always visited, because there is no proper way to break out of a lambda expression. There exist approaches by throwing an Exception, however this is considered to be bad practice.

For example, getting a function by its name, can be achieved as showed in listing 3.4. Note that mutating variables in lamda expressions is not thread-safe, an `AtomicReference` is used.

```java
final AtomicReference<Function> fRef = new AtomicReference<>();
traverseFunctions(fnList, (fn, parentObj) -> {
    if (fn.getName() != null && fn.getName().equals(name)) {
        fRef.set(fn);
    }
});
// do something with found function in fRef.get();
```

Listing 3.4: get a function by its name

As a result, the implementation of the *general adaptation tasks* rely on the *traverse function* and Reflection, which made it possible to overcome mentioned problems of the AFCL Java API for the major part.

The concrete adaptation developed in this thesis, which divides `ParallelFor` elements into multiple sections in a `Parallel` element, have been achieved by using a combination of the *general adaptation tasks* and adaptation-specific additional logic.

4 Web Interface

The layout of the web interface is based on coreUI[1], an admin panel template, built on top of the web toolkit Bootstrap[2]. 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[3]. The main component has four sub-components - where each of them consists again of multiple sub-components - representing the core features of the app.

4.1 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.

[1]https://coreui.io
[2]https://getbootstrap.com
[3]https://en.wikipedia.org/wiki/Single_responsibility_principle

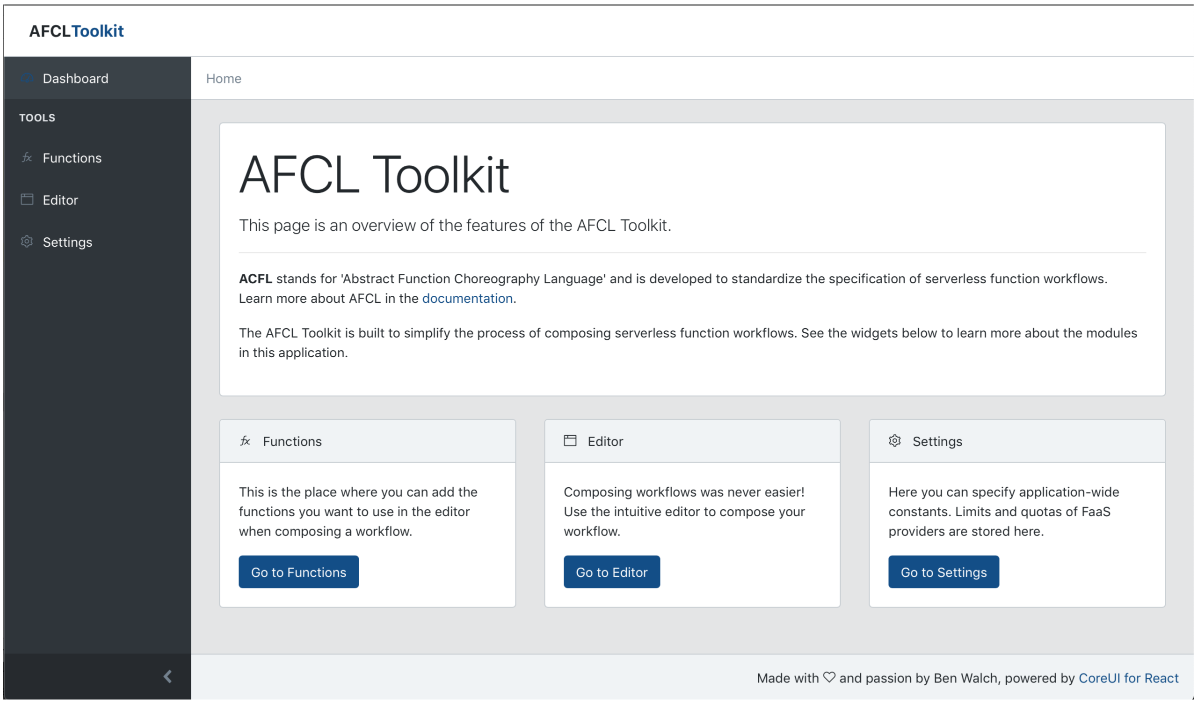


Figure 4.1: the dashboard

4.2 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.[4]

[4] see 3.2.2 for detailed information on how data is persisted

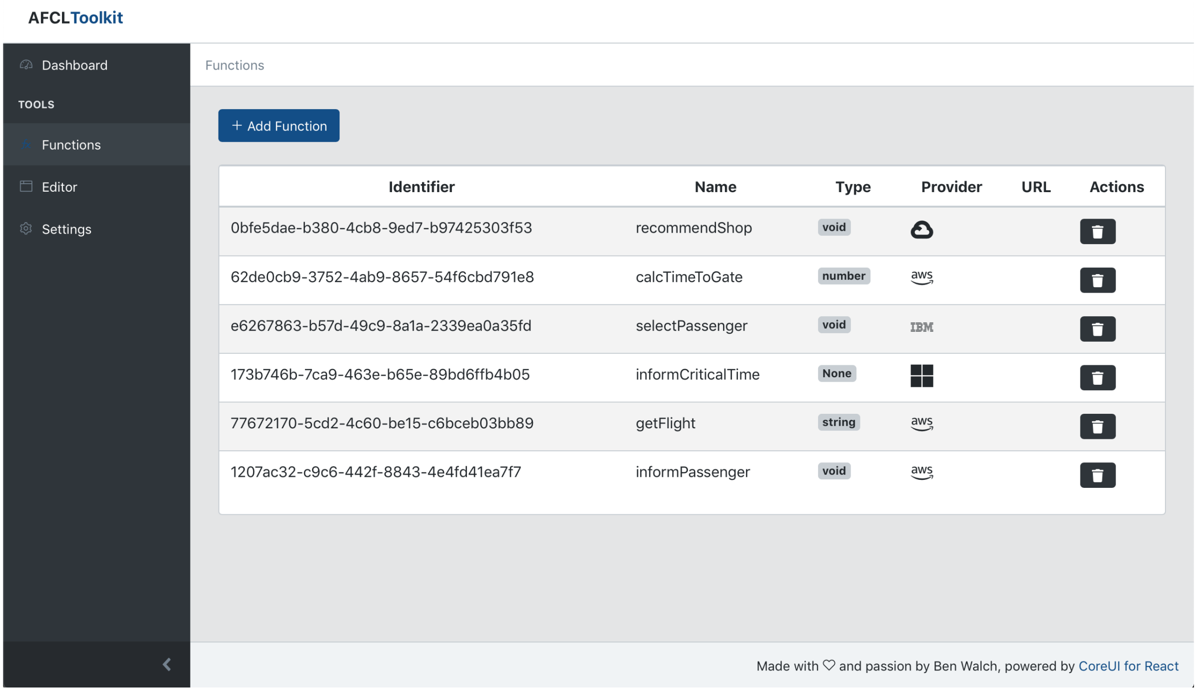


Figure 4.2: the function repository

4.3 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.

Composed workflows are saved to the user's file system by offering a download. The user can then choose the target location.

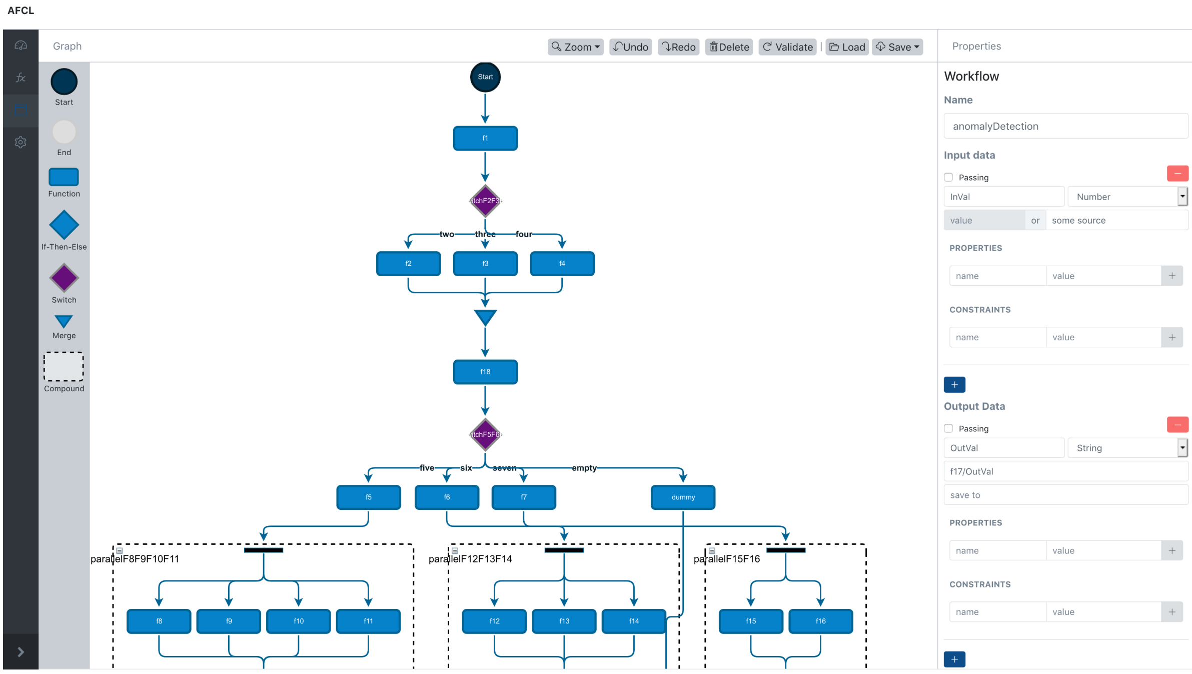


Figure 4.3: the editor

4.4 Settings

5 Evaluation

6 Conclusion and Future Work

Bibliography

[1] *AFCL - Abstract Function Choreography Language*. URL: `http://dps.uibk.ac.at/projects/afcl/` (visited on 04/10/2020).

[2] 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`.

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

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

[5] 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`.

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

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

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

[9] Phil Sturgeon. *Understanding RPC Vs REST For HTTP APIs*. URL: `https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/` (visited on 04/29/2020).