universität innsbruck

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, February 15, 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

In the FaaS world, there exist many providers who offer serverless execution of code (functions) in the cloud. Each of them has its own specifications and definitions on how functions are deployed and executed, and how workflows are created and executed.

The Distributed and Parallel Systems group developed the "Abstract Function Choreography Language" (AFCL). It is a specification for describing serverless workflows. Also, a Java API, to describe serverless application workflows programmatically was developed. The product which results in using that API is the workflow being described in AFCL in a text file. Until now, the workflow 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 to 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 Server" is the most recent term in the cloud computing environment. With the rise of the serverless technology during the last years, *FaaS* became more and more popular. With the serverless technology, a high level of flexibility in execution of code became possible. Global Players like Amazon, Google, IBM and Microsoft jumped on the train and provide their infrastructure to Developers, able to deploy functions and execute them in the cloud. Each system has its own definitions on how to define, deploy and run a function. Many providers also offer a tool to model a workflow and execute it (Amazon Step Functions, IBM Cloud Functions, Microsoft Azure Functions, Google Cloud Functions, ...)

The distributed and parallel systems group developed a specification

2 Background

2.1 FaaS

2.2 AFCL

3 Overview

4 Implementation

The Implementation of a Graphical User Interface as well as the optimization for Workflows were the main goals of this bachelor thesis.

4.1 Requirements

- 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 [?]

4.2 Frontend

One of the main parts of the application is the 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.

A prototyping phase was the kickoff for the development: Different frameworks and libraries have been researched, tested and selected. For the selection, the following criteria have been taken into account:

- open-source
- good documentation
- large and active community (future-proof)
- stars on github / downloads on npmtrends
- performance
- pre-knowledge of the developer / learning curve

This criteria ensure the usage of well-documented and open-source software and makes it easier for others (dps) to extend the application later. The result of this development phase was a working prototype as proof-of-concept. In the following sections, the setup and each chosen technology and frameworks are described in detail.

Setup

The setup of the frontend application is a selection of tools and technologies for modern and complex web development. 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. The base forms the node package manager (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 does much of heavy-lifting, by bundling all sources with the assets into single files, and execute additional build steps by utilizing so-called 'loaders'. 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.

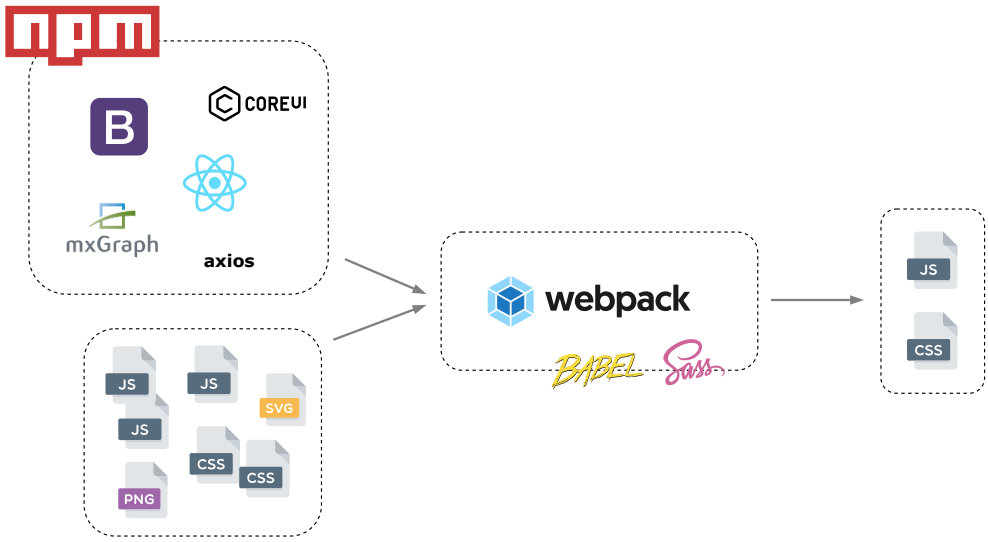

Figure 0.1: frontend development workflow

npm

The node package manager[1] 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.

webpack

webpack is a module bundler, its main purpose is to bundle JavaScript code for the usage in a browser.[2] 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 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

[1] https://www.npmjs.org
[2] https://webpack.js.org

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 compile 'transcompiler' (or transpiler) is needed to transform the ECMAScript source code to JavaScript common browsers are capable of interpreting. This process of compiling is done with Babel[3], which is configured to not only compile ECMAScript, but also JSX to JavaScript.

SASS

CSS, in its pure form, reaches its limits when one thinks about using variables, functions or nested rules. SASS[4] is a stylesheet language, which is compiled to CSS and offers the mentioned and even more features. A lot of CSS Frameworks also offer its source code in SASS with a large variable set which makes it easy to customize.

4.3 Web Interface

The layout of the web interface is based on coreUI[5], a admin panel template, built on top of the web toolkit Bootstrap[6]. A clean and nested structure of components, which is typical for a React app, guarantees modularity and forms the whole application. The main component has four sub-components - where each of them consists again of multiple sub-components represent 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.

[3]https://babeljs.io
[4]https://sass-lang.com
[5]https://coreui.io
[6]https://getbootstrap.com

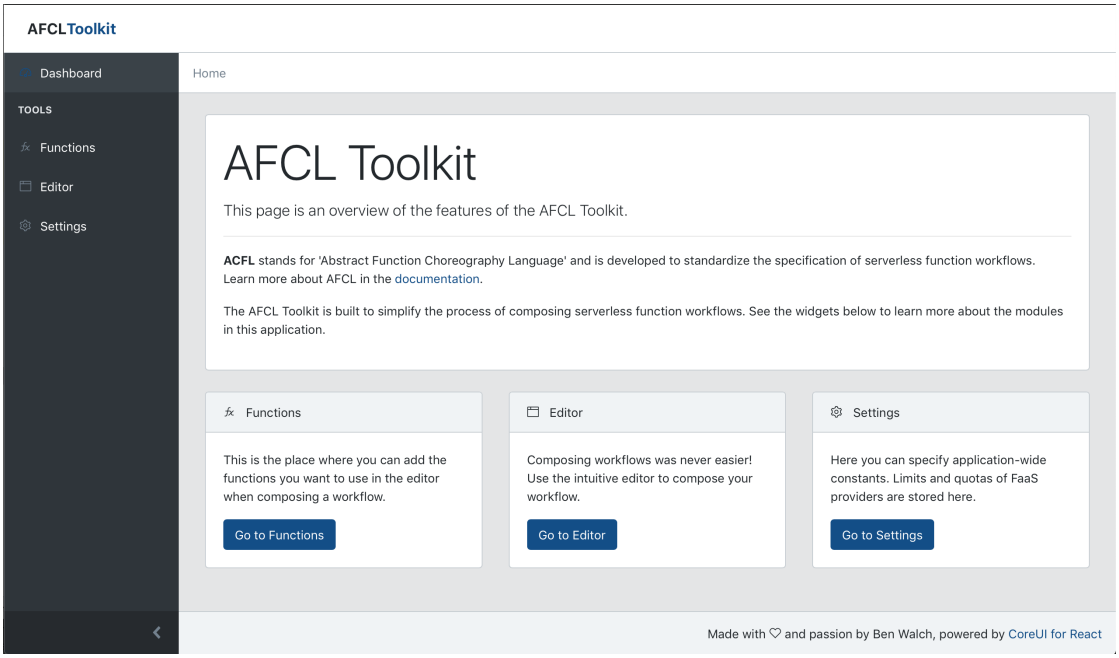

Figure 0.2: the dashboard

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. For In the implementation of this thesis, this data is persisted to an internal file.[7]

Editor

The interface of the component 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.

[7] see 4.4 for detailed information on how data is persisted

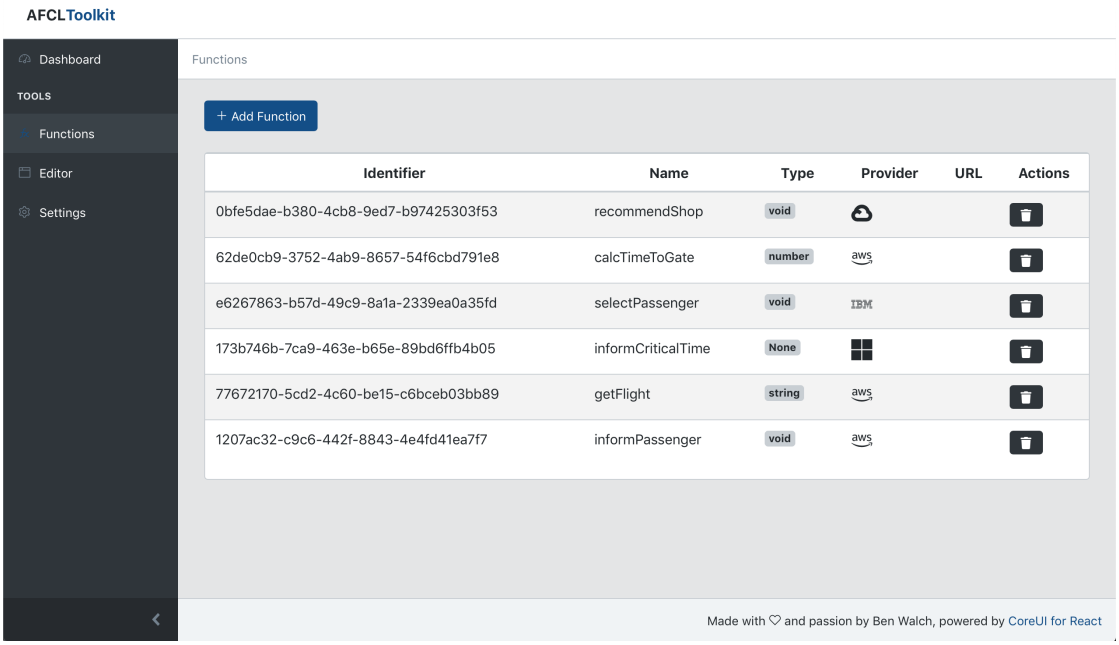

Figure 0.3: the function repository

Settings

React

mxGraph

CoreUI

Bootstrap

4.4 Backend

General

Maven

Servlets

Persistence

Composed workflows are be 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[8] for the repository implementation 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.

Api

4.5 Continuous Delivery

5 Improvements

6 Conclusion

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

Bibliography

[1] (2020, February) Webpack. [Online]. Available: https://webpack.js.org