The present work was submitted to
the Research Group
Software Construction

of the Faculty of Mathematics,
Computer Science, and
Natural Sciences

Bachelor Thesis

Developing a Service-Oriented Interface for a Heterogeneous Code Generator

Entwicklung einer Service-Orientierten Schnittstelle für einen heterogenen Code Generator

presented by

Marco Bähr

Aachen, September 25, 2017

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr.-Ing. Ulrik Schroeder

Supervisor

Dipl.-Inform. Andreas Steffens

Statutory Declaration in Lieu of an Oath

The present translation is for your convenience only.
Only the German version is legally binding.

I hereby declare in lieu of an oath that I have completed the present Bachelor's thesis entitled

Developing a Service-Oriented Interface for a Heterogeneous Code Generator

independently and without illegitimate assistance from third parties. I have use no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Official Notification

Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whosoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 to 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

I have read and understood the above official notification.

Eidesstattliche Versicherung

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Bachelorarbeit mit dem Titel

Developing a Service-Oriented Interface for a Heterogeneous Code Generator

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, September 25, 2017 (Marco Bähr)

Belehrung

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicher ung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen.

Aachen, September 25, 2017 (Marco Bähr)

Acknowledgment

I would like to thank the KISTERS AG for accompanying this thesis, as well as, Dipl.-Inform. Andreas Steffens for providing guidance. Lastly, I would like to thank the reviewers Prof. Dr. rer. nat. Horst Lichter and Prof. Dr.-Ing. Ulrik Schroeder for taking the time to review this thesis. *Marco Bähr*

Abstract

When combining different architecture patterns and DevOps software projects become complex in structure. Using a code generator can ease the creation of new projects, but introduces the need to install the tool and keep it up to date. By providing it as a web service only one instance has to be updated.

This thesis introduces a concept on how to serve an already existing code generator for heterogeneous infrastructure over the web using a REST-API. By adding a web application to interact with the service the usage is made more intuitive.

Additionally, this thesis introduces a concept for dynamic generation with the example of Jenkins build-pipelines. The build-pipelines are made decomposable to allow for a fine-grained configuration.

Contents

List of Tables

List of Figures

List of Source Codes

1. Introduction

Contents

To increase re-usability and separation of concerns during software development architecture patterns are used. As a result, the development environment becomes more complex and requires the repetitive creation of similar components[Gee17].

A code generator can be used to automatically create these components and provide boilerplate code [Gee17].

The need to install and update such a generator might make it less attractive for some. A service-oriented implementation eliminates these requirements for the individual developer and therefore makes it more appealing.

In this thesis, an implementation for a service-oriented interface for an already existing heterogeneous code generator is introduced. Additionally, a first front-end implementation is provided to make the service usable for developers.

Code generation can also be used in other areas. For this reason, dynamic code generation with the example of Jenkins build-pipelines is implemented and examined.

1.1. Structure of this Thesis

The following chapter presents the motivation for this thesis as well as a definition of the problem. In chapter 3 similar concepts already implemented are evaluated against the current requirements. The next chapter then introduces some basic knowledge which will be assumed as given for the rest of the thesis. Next, the concept to solve the in chapter 2 introduced requirements is presented and followed by some implementation details provided by chapter 6. The last chapter is then used to analyze to what extent the requirements are met by the introduced solution.

2. Motivation and Problem Statement

Contents

Using a code generator can decrease the overhead of creating new software projects [Gee17].

However, the requirement to install the generator can be unpleasant and lead to developers ignoring the option. By serving a it over the web, this is no longer needed and the service only has to be installed once.

Having a single instance that is being used by all developers results in a single point of maintenance. If the generator has to be updated it has to be done only once instead of on every developer's machine. This makes the update process a bliss and guarantees that all developers use the same version.

By definition a service allows the implementation of multiple clients. These clients only have to provide the needed information to the generator and are therefore lightweight. This makes it easy to implement plug-ins for various *Integrated Development Environments (IDEs)* or a web application.

Implementing a web application with an intuitive interface can help lower the learning effort required to get familiar with the generator. It makes it look more appealing to developers and will, thereby, increase its usage.

2.1. Problem Statement

The goal of this thesis is to make the code generator implemented by Ralph Geerkens in his master thesis easier to reuse by offering it as a web service. This allows to integrate it into different scenarios and introduces the mentioned single point of maintenance.

To make the service more useful, a web application will be provided as a first front-end implementation. This web application provides a graphical user interface to interact with

the service.

Adding to this, the thesis tries to prove that the Pagen generator can be extended to generate a complete software component. As an example, a generator for a component developed by the KISTERS AG will be added to the generator.

Lastly, the dynamic generation of code is examined. This is achieved by generating Jenkins build-pipelines that are dynamically requested and executed during an already running build-pipeline execution.

2.2. Requirements for a Service-Oriented Code Generator

In this section the functional and non-functional requirements of a service-oriented code generator are presented.

2.2.1. Functional Requirements

1.1. Generator discovery Implementing new generators should not require to register them in the service.

1.2. Generator Information A list of available generators and there parameters should be available.

1.3. Execution of multiple generators It should be possible to run distinct generators that produce a common artifact.

1.4. Test execution Before the download it should be possible to verify the generation by executing tests.

1.5. Persisted configuration To be able to reproduce or improve the generated project, the used configuration should be preservable to allow some sort of incremental usage.

1.6. Statistics To give an overview of most used generators, statistics should be set up.

1.7. Client application A first client should be shipped with the service in form of a front-end application.

1.8 Software Component Generation Pagen should be extended with the ability to generate complete software components.

1.9. Dynamic generation The service should be extended with the ability to generate code dynamically on the example of Jenkins build-pipelines [Jen].

2.2.2. Non-functional Requirements

1.9. Reusability The generator interface should be reusable. It should allow multiple front-end implementations.

1.10. Extensibility The integration of new operations in the processes should be possible without having to change the majority of the code.

2.3. Requirements for a Front-End Application

The previous section described the functional requirement of a front-end implementation. The requirements for such a web application are listed below.

2.3.1. Functional Requirements

2.1. *Single-Page Application* The front-end should be implemented as a *SPA*.

2.2. Generator list A list of all available generators should be presented to the user. This list should allow to filter generators by used architecture or technology.

2.3. Generator parameters The parameter inputs for the generators should be dynamically adjusted to the current selection.

2.4. Process information The progress and test information has to be continuously available to the user.

2.5. Optional artifact retrieval The retrieval of the artifact should be optional in the case of failed tests.

2.6. Charts Statistics should be presented as charts.

2.3.2. Non-Functional Requirements

2.7. Intuitivity The user-interface has to be intuitive and easily understandable to achieve a faster learning experience.

2.4. Requirements for Dynamic Jenkins Pipelines

In section 2.2 the requirements for the service included the dynamic generation at the example of Jenkins build-pipelines. This section describes the functional and non-functional requirements for such a build-pipeline.

2.4.1. Functional Requirements

3.1. Decomposability A dynamic build-pipeline should be decomposable to include or exclude certain parts of the pipeline.

3.2. Parameterization Pipelines should allow the usage of parameters, which are configured before execution.

3.3. Template Version To provide reliability, the template version used in a project should be specifiable.

2.4.2. Non-functional Requirements

3.4. Effortless Integration The integration into Jenkins should require a minimum amount of effort.

3.5. Maintainability There should only be a single point of maintenance, meaning a modification of a pipeline template should result in an upgraded pipeline configuration in all projects.

3. Related Work

Contents

In this chapter already available solutions for the introduced problems are presented and evaluated against the requirements.

3.1. Code Generator Services

3.1.1. Spring Initializr

The Spring Initializr is a web service that offers to generate quickstart projects for the Spring Boot framework [Sprc; Spra]. The project is split into three sub-modules.

The `initializr-generator` module is the library generating the code. It offers the core functionality and can be embedded in any project.

It is executed by the web service implemented in the `initializr-web` sub-module. This module offers a *REST-API* to configure the generator execution and a web interface.

The web interface offers the selection between Gradle and Maven as build systems, as well as, the programming language and the Spring Boot version [Mav; Gra]. The supported programming languages include Java, Kotlin and Groovy [Jav; Kot; Gro]. Next, it requires to input the projects name with the Maven naming convention and a optional description. The selection of the output format (*JAR* or *WAR*) and Java version is also avaiable.

If the generation is started, the Spring Initializr generates a basic example project containing: a basic Maven configuration and a small class containing the Spring Boot initialization.

For further configuration it allows adding commonly used libraries to the project. These include additional web frameworks, database drivers, template engines, integration of social networks and many more. These, however, only affect the generated Project

Object Model (POM) Maven configuration [Mav]. This means no additional boilerplate code is generated.

The generator is designed to create a bare-bone project structure and does not allow the generation of example code and makes the execution of tests is unnecessary. Spring Initializr also does not offer any way to share or preserve the used configurations settings.

3.2. Configurable Build-Pipelines

3.2.1. Jenkins

Jenkins offers functionality for parameterized builds by default. It allows the user to include placeholders in their `Jenkinsfile` pipeline configuration. While this allows reusing the pipeline across different projects that have the same build steps, but different parameters. To be able to include or exclude certain steps, conditional blocks and parameters have to be used. This results in a less readable and therefore less maintainable pipeline code.

A newer addition to the Jenkins build ecosystem are shared libraries. Shared libraries are designed to create new reusable pipelines steps and are generally loaded from a Source Code Management (SCM) system like Git [Gita]. Commonly used pipelines can thereby be implemented as a new pipeline step and reused in different Jenkinsfiles. The version of the library can be specified per Jenkins instance or Jenkinsfile.

3.2.2. TeamCity

TeamCity is a continuous integration server developed by JetBrains [Tea]. TeamCity, like Jenkins, allows builds to have parameters in form of either system properties or environment variables. These enable sharing common pipeline configurations across projects, but do not allow a single point of maintenance since they need to be duplicated for each new build configuration. For this reason TeamCity supports build templates. Build templates can be used to maintain similar build configurations. A configuration can inherit settings and build steps from one build template and might change some parameters. A configuration, however, can only be associated with a single build template. Therefore, parts of the template can not be excluded.

TeamCity also supports the concept of meta-runners. Meta-runners allow combining several build steps into a custom build step that can be included in different configurations. This allows summing up part of the process and share it across builds similar to Jenkins shared libraries.

3.2.3. BuildKite

BuildKite is a build server that does not have a specific pipeline language [Bui]. The pipelines are defined in a `pipeline.yml` file that contains its steps. Each step has a label; an identifier of the agent, which should execute the step; and a command to run. If the step is executed, the specific agent runs the command and therefore allows the

steps to be implemented in a programming language of choice. BuildKite also supports creating pipelines dynamically. Inside an existing pipeline, a build step can create a new `pipeline.yml` file and run it.

3.2.4. GoCD

GoCD is a continuous integration server that uses a *XML* based pipeline format [Goc]. Pipelines in GoCD can be abstracted using templates. A template pipeline defines the build steps but not the configuration. Pipelines using the template are required set these parameters. This allows to use the same build steps for different projects that have the same build steps and result in a single point of maintenance.

3.3. Summary

This chapter showed that there was no web service found that fulfilled all introduced requirements. There are, however, several build-pipelines that support functionality similar to what is required. Only one of those allows for a dynamic generation of pipeline code.

4. Background

Contents

In this chapter some concepts and architecture patterns are introduced, which were used as a basis for the concept and implementation of this thesis.

4.1. Model View Controller

The *Model View Controller* architecture pattern is used divide the business logic of a web application from its *UI* [BHS07].

The functionality of the application is implemented in a so called **model** [BHS07]. This model might be divided into several domain objects to allow to spread the responsibilities realized. The model has no knowledge of the user presentation nor does it have to.

The *User Interface* is presented by a **view** [BHS07]. There can be a view for every aspect of the model. A view is responsible to retrieve data from the model and convert it to the output [BHS07]. This allows to change the view without having an effect on the model.

The third component of the *MVC* pattern is the **controller**. The controller receives inputs from the user and manipulates the model to react to the events [BHS07].

By separating the concerns it is possible to, for example, change the *User Interface* without affecting the application's functionality [BHS07].

4.2. Model View ViewModel

Model View ViewModel (MVVM) is an architecture pattern based on the previously introduced *MVC* [Mvv]. The goal of *MVVM* is similar to the previous. Achieve separation

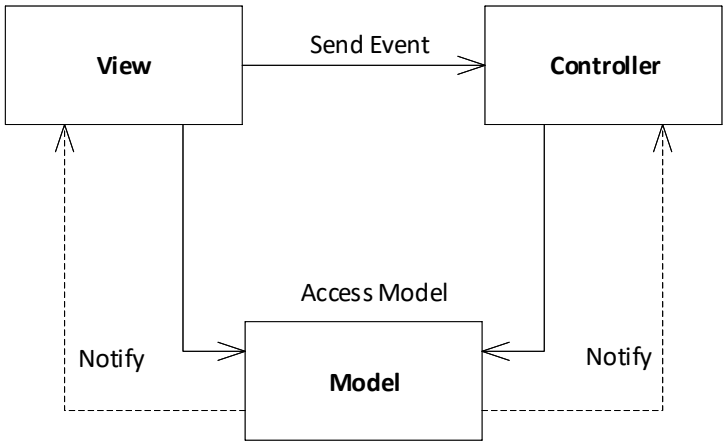


Figure 4.1.: *Model View Controller* [BHS07]

of concerns by splitting the application into different classes with predefined separate objectives. The result is easier to test, re-use and maintain than before [Mvv].

The pattern uses three components: the view, the model and the view model [Mvv]. The first two are familiar from the *MVC* pattern. The model implements the application's domain objects and the view is the visual representation of the data presented to the user. The controller of the *MVC* pattern is now replaced by a **view model**. The view model acts as a mediator between the model and the view [Mvv]. It invokes methods of the model and hands reformatted data to the view. It also manages the applications state and executes operations based on user interaction. Data between the view and the view model is linked using two-way data biding. If the data in the view changes, it effects the view models state and vice versa [Mvv].

A benefit of this approach is that the interface functionality can be tested by testing the view model. This is easier than having to test the from the view produced user interface itself. An overview of the components relationships is presented in figure 4.2.

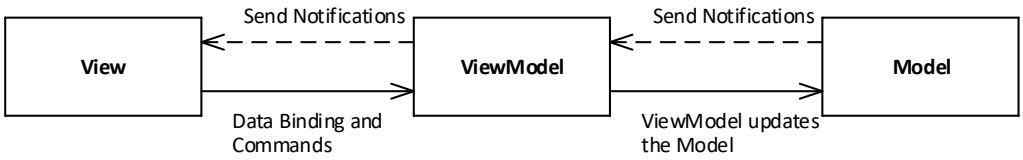


Figure 4.2.: *Model View ViewModel* Diagram [Mvv]

4.3. Web Services

Web services are defined by the W3C as

> "[...] a software system designed to support interoperable machine-to-machine interaction over a network. [...] Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

[Weba]

4.4. RESTful Web Services

Representational State Transfer (REST) is a style defining how *HTTP* requests should be used [Jak08]. The W3C categorizes web services implementing such a behavior as "REST-compliant" [Weba]. These web services are resource-oriented and allow addressing using a directory like *URI*. This makes an implemented Application Programming Interface (API) more intuitive for the user.

To modify a resource *CRUD* operations in the form of *HTTP* requests are used. Resources are retrieved using `GET`, created using `POST`, modified using `PUT` and deleted using `DELETE` [Jak08].

All operations executed are stateless, meaning the result is independent from the previous requests and always contains the full information [Jak08].

Generally, resources can be represented by any format, however, it is recommended to use either *XML* or *JSON*. These formats are implemented in a variety of clients and make the service widely usable [Rod08].

REST is the current state-of-the-art in *Application Programming Interfaces (APIs)* and is widely used in web services. Since it is only dependent on *HTTP* and when used in combination with *XML* or *JSON*, it can be easily consumed by any client side application written in a modern programming language. The stateless approach eliminates the need to persist a client's state on the server and thereby reduces overhead.

4.4.1. Pipes and Filters

The pipes and filters architecture pattern divides steps of a process into small chunks, called filters. Each filter has a input and an output and serves only a single function. Filters are connected by pipes, which are used to pass the results from one filter to the next.

Unix shells utilize this pattern to chain programs, which by design only serve a single purpose. For instance, the statement `cat hello.txt | wc -l` consist of the two filters `cat` and `wc`. Using the pipe ("`|`") the output of `cat hello.txt` is used as input for `wc -l` resulting in the number of lines in the file `hello.txt`.

4.5. Pagen

Pagen is a incremental code generator designed and developed by Ralph Geerkens during his master thesis [Gee17]. The generator tries to fulfill the following four requirements.

The generator should support the generation of heterogeneous software and architecture. This means it has to be able to adapt to different programming languages, as well as, architecture patterns. To achieve this it has to enable the generator developer to implement different naming strategies and code organizations [Gee17].

Instead of only providing a bare-bone project for a developer to use, the generator should be useful during the development phase by providing incremental generation. Therefore, the generator allows the user to generate components into an existing project [Gee17].

To make the generator more accessible, the ease of use was considered during development. This includes an easy installation, update and execution [Gee17].

The last considered aspect is knowledge sharing. Knowledge sharing should allow developers to share best practices by providing exemplary code as a result of the generation [Gee17].

The generator is constructed using a layered architecture presented in figure 4.3.

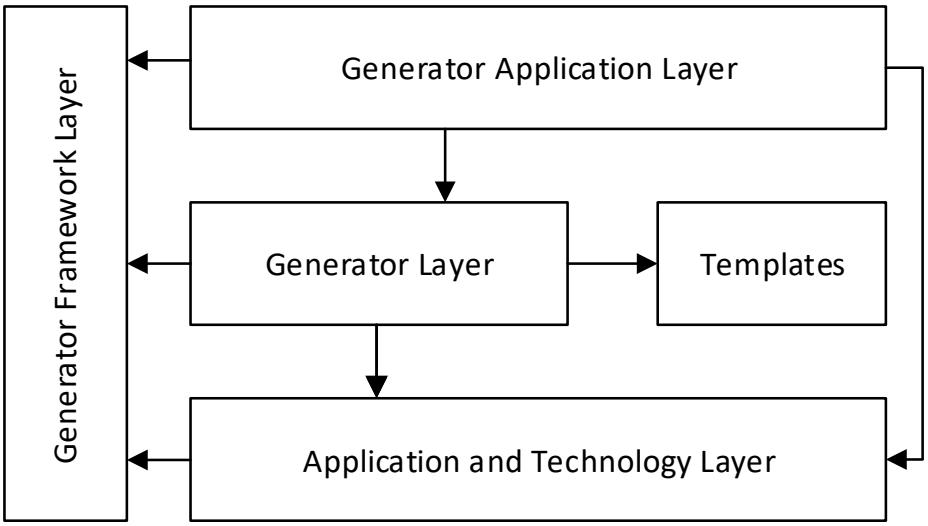


Figure 4.3.: Pagen Layers Overview [Gee17]

4.5.1. Generator Framework Layer

The first layer is the generator framework layer. It provides the functionality and base classes required to implement a code generator matching the described requirements.

This layer also contains the User Query API and the generator life-cycle [Gee17].

The User Query API is used by concrete generators to prompt the user questions on a *CLI*. Each question is represented by a class implementing the `Prompt` interface (shown in listing 4.1). Therefore, all prompts contain a name, message and a method to interact with the user [Gee17].

```
1 public interface Prompt<T> {
2
3     String getName();
4
5     String getMessage();
6
7     T prompt();
8
9 }
```

Source Code 4.1: Prompt.java

Each concrete generator implements the three life-cycle phases shown in figure 4.4.

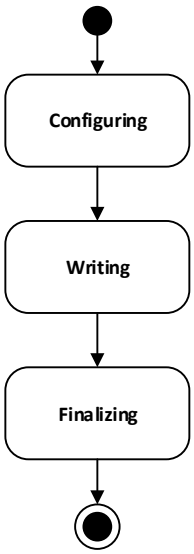


Figure 4.4.: Pagen Generator Life-cycle [Gee17]

The `configuring` phase is used to register any sub-generators (Pagen allows composed generators) and prompts before evaluating the inputs [Gee17].

The following `writing` phase performs all operations on the template files. Since files can be expected to be dependent on each other a Intermediate File Storage (IFS) is introduced. The *IFS* is responsible for retrieving the templates and keeping them in

memory until told otherwise [Gee17]. This allows to perform actions like placeholder replacement and rearrangement without causing load on a permanent file storage.

When executing the `finalizing` phase the *IFS* is instructed to output the results of the generation to disk [Gee17].

4.5.2. Application and Technology Layer

The second layer is the application and technology layer. This layer implements naming conventions and other definitions required to support the heterogeneous nature of the generation [Gee17]. It also contains abstract generator implementations for the different technologies and architecture patterns [Gee17].

4.5.3. Generator Layer

The third layer is the generator layer. The generator layer contains the concrete generator implementations and templates. Templates are stored separately and are fetched during the generator execution [Gee17]. Each generator follows the in the framework layer defined life-cycle steps and produces the final generation output [Gee17].

4.5.4. Generator Application Layer

The last layer is called generator application layer. This layer represents the *CLI* application implemented as a Maven plug-in [Mav; Gee17]. It offers the user a selection of available generators and executes the selected. On execution the respective generator prompts the user with the configured questions and generates the code in the current directory [Gee17].

In this thesis Pagen will be made available using a web service. For this reason, the generator application layer is of no use and therefore will not be mentioned again.

4.5.5. Jenkins Pipelines

A Jenkins pipeline (or more specific build-pipeline) is a collection of plug-ins for the Jenkins build automation server [Jen]. It allows the system to support *Continous Delivery* processes using a Domain Specific Language (DSL) [Jen].

Continous Delivery (CD) is the process of serving a software system to the user automatically. This process includes the retrieval of source code, compilation, testing and finally installation on a target machine [Jen].

Jenkins pipelines are defined in a `Jenkinsfile` containing different stages. A stage is a collection of steps that serve a common purpose [Jen]. An example for a stage might be the compilation phase. The steps represent actions executed by the pipeline. These could include a *SCM* checkout, shell script execution and so on. Listing 4.2 shows a pipeline example grabbed from the official documentation [Jen].

```
1 pipeline {
2     stages {
3         stage('Build') {
```

```
4               steps {
5                   sh 'make'
6               }
7           }
8           stage('Test'){
9               steps {
10                  sh 'make check'
11                  junit 'reports/**/*.xml'
12              }
13          }
14          stage('Deploy') {
15              steps {
16                  sh 'make publish'
17              }
18          }
19      }
20  }
```

Source Code 4.2: Jenkinsfile

5. Concept

Contents

In this chapter the basic concepts and architecture is introduced. The concept is divided into three parts:

- A **web service** implementing the capabilities defined in section 2.2 and section 2.4.
- A **web application** fulfilling requirements described in section 2.3
- And a **Jenkins library** to be able to use the generated Jenkins build-pipelines.

Figure 5.1 gives an overview about the listed components.

5.1. Web Service

The web service consists of two parts that are shown in figure 5.1. A Pagen service component and a Jenkins service component.

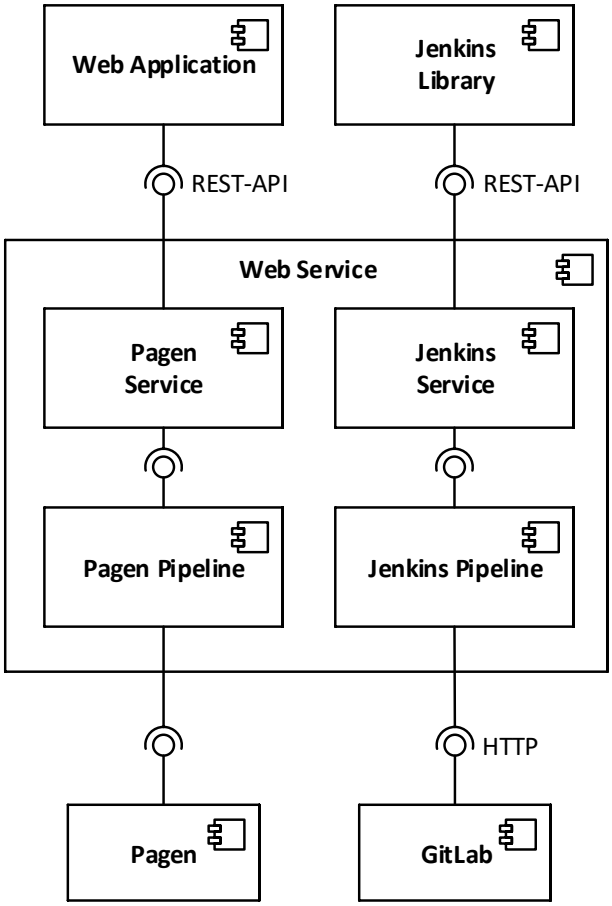


Figure 5.1.: Architecture

The Pagen component is responsible for offering a web service interface to Ralph Geerken's Pagen generator. This interface is used by the web application and can also be used by other *REST* compliant clients (requirement 1.9). The Pagen generator is here used as a dependency.

The Jenkins component is responsible for creating the dynamic Jenkins pipelines and serving them, again, over a *REST*-API (requirement 1.9). The templates are stored in a GitLab repository and are retrieved over the *HTTP*. Instead of the web application, the interface is designed to be consumed by a Jenkins library, which requests and executes the Jenkins build-pipeline.

The actual generation of both components is done in a separate component, called pipeline. These either do the generation themselves (Jenkins service component) or delegate it to the Pagen generator.

To avoid confusion the Jenkins pipelines will be called build-pipelines for the rest of this thesis.

5.1.1. Technology

Spring Boot

The web service is implemented with the help of Spring Boot[Spra]. Spring Boot is a simplification of the Spring Framework [Sprb]. It allows the user to create stand-alone Spring applications without the need for extra configuration or external Java Servlets. The Spring Framework uses the Model View Controller (MVC) pattern and includes a functionality to create RESTful apllications, as well as, features like Dependency Injection, database integration and integrated *JSON* marshalling/unmarshalling. Since it is a Java framework the integration of Pagen can be achieved by importing the project as a dependency and using it as a library. This allows to keep the execution in a single *JVM*.

The Spring request work-flow is visualized in figure 5.2. The framework is request-

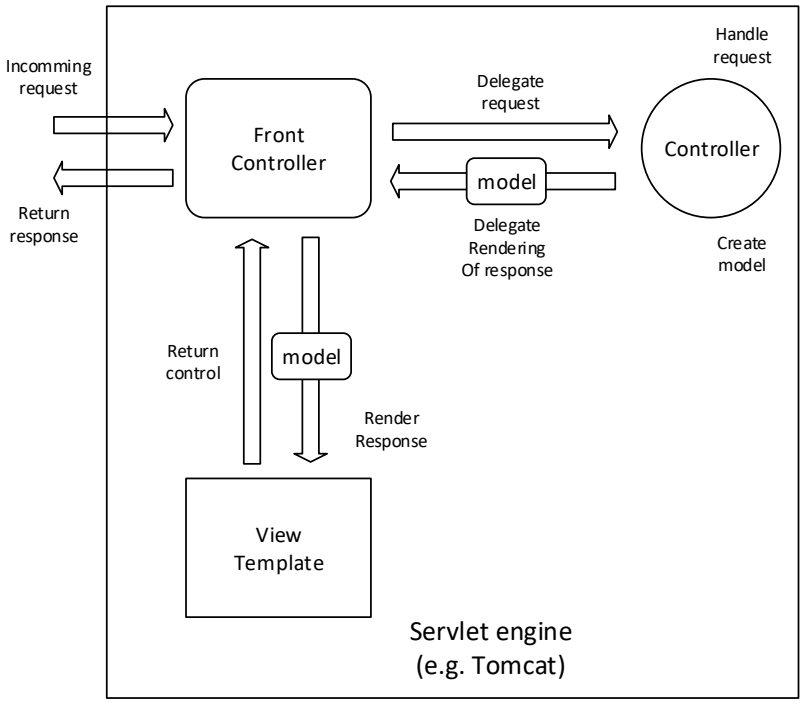


Figure 5.2.: Request Processing in Spring Web MVC `DispatcherServlet` [Sprb]

driven and uses a central `DispatcherServlet` (illustrated in the graphic) to delegate work to specific controllers and render the model to a view [Sprb].

5.1.2. Web Service Class Type Overview

The service's classes are divided into three types.

The **controller** is introduced in the illustration figure 5.2 and handles specific requests forwarded from the Spring framework. These represent the endpoints of the *REST*-API.

Like suggested by Fowler, the domain layer is split into domain models and a service layer [Fow02]. The **service layer** in this implementation contains the business logic while the other objects mainly serve as ***Data Transfer Objects (DTOs)*** [BHS07]. This makes the used data independent from the used generator.

5.1.3. Pagen Service Component Overview

The Pagen component of the service is shown in figure 5.3. It consists of six DTOs, three controllers and three services.

Data Transfer Objects

The `Generator` class is used to represent a Pagen generator. It includes its name, its command (used to identify the generator) and other meta information (requirement 1.2). Additionally, it contains the list of prompts. These prompts are represented by the `Prompt` DTO and include the message and value type of the prompt. Adding to this a map to store nested prompts is added. The map saves the name of the prompts as keys and other `Prompt` DTOs as values. These classes are used to provide the client with information about the available generators. To initiate a generation, the client inputs a `GeneratorConfig`. The `GeneratorConfig` defines the generators (identified by command) that should be executed and a list of parameters, which should be used as prompt inputs.

When a generation is started, the user is provided which an identifier for the specific job, to be able to check its progress. The identifier is modeled by the `Job` class and the status information by the `JobStatus` class. The `JobStatus` contains information about the current state of execution, test results and the current output log.

The remaining DTO (`Response`) is used to transmit `String` messages to the client. This is used to give information in the case of failure.

Controllers

The controllers `GeneratorController`, `JobController` and `ArtifactController` represent the *REST*-Endpoints of the application. They execute the respective methods of their service and compose error messages with the help of the `Response` domain in case the service fails.

Services

The service classes implement the business logic of the web service.

The `GeneratorService` acts as a facade for the user query mapping, input validation and pipeline composition [Fow02]. The query mapping (explained in section 5.1.5) and input validation is outsourced using a strategy pattern [Gam+95].
The `GeneratorService` executes the `PromptMapingStrategy` on all generators ,as soon as the generator overview is requested, and adds a prompt for the optional test execution to the returned list of `Prompts` DTOs. They are than added to the created `Generator` DTOs and returned to the user. A to *JSON* transformed example is presented in appendix D. If a new generation is initiated, the service validates provided inputs using the `PromptValidationStrategy` and assembles the pipeline using a `JobPipelineBuilder` (Explained later in this chapter). The assembly is performed dynamically, if the user requested to run test the pipeline is configured accordingly. All available Pagen generators are registered in the `GeneratorService`, during the pipeline composition the chosen generator instances are passed to the pipeline for execution. After the pipeline is composed, the `GeneratorService` passes the already running pipeline to the `JobService` and provides the user with a matching identifier in form of a `Job` DTO.

After that, the `JobService` stores the pipeline reference and is responsible for creating `JobStatus` responses. If a status is requested and the pipeline is finished, the service passes the pipeline's final artifact to the `StorageService` and provides the client with a matching artifact id.

Using this artifact id, the client is able to download the result of the generation from the `StorageService`'s in-memory storage. Artifacts can be expected to be small in the case of a code generator. Therefore, the in-memory storage is sufficient. If a client downloads an artifact it gets immediately removed from the storage.

It is, however, possible that a generated artifact gets abandoned by the client. This is common if, for example, the tests failed. In such cases the service has to discard the generated artifact. For this reason a time-to-live is introduced. Each artifact gets a time-stamp as soon as it is added to the storage. Every two minutes, the service checks for artifacts with a time-stamp older than two minutes and removes them. This results in a worst-case storage time of four minutes.

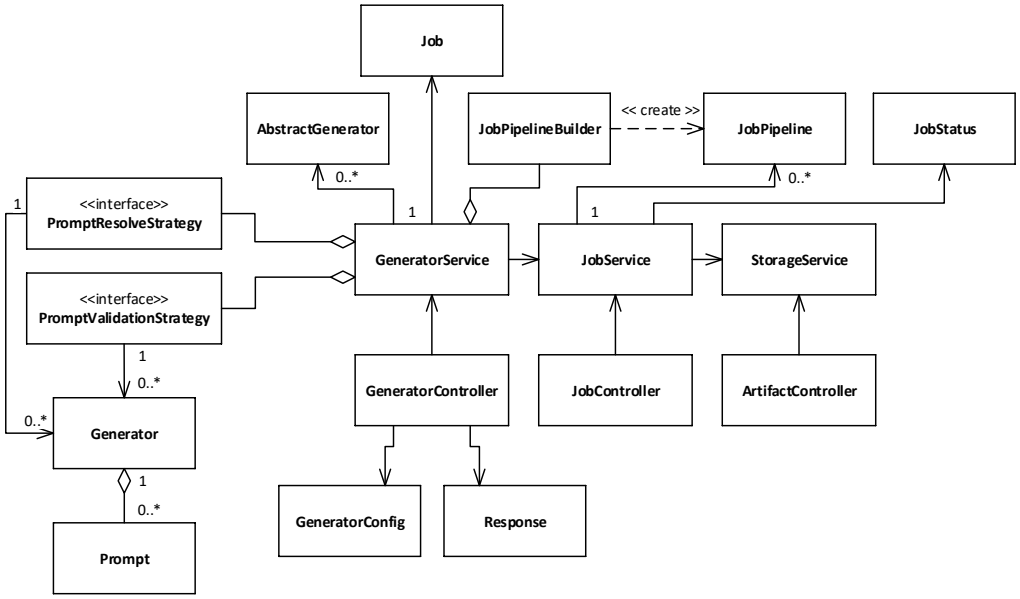


Figure 5.3.: Class Diagram of the Pagen Component

5.1.4. Pagen Generator Discovery

To ease the integration of new Pagen generators an annotation is added which contains additional meta-data of the generator. This includes architecture type, technology and a category. This annotation is used to scan for generators using Spring's bean discovery [Spra]. The discovered generators can than be injected in the service without having to be explicitly added (requirement 1.1).

5.1.5. Pagen User Query API Mapping

To present the user with the parameters required by the selected generators Pagen's User Query API has to be mapped to the `Prompt` DTOs (requirement 1.2).

Pagen uses a builder pattern to compose queries in form of prompts [Gam+95]. A prompt is implemented as a interface, which includes the prompts name and message. Here, the name of the prompt represents the queries key and the message the output presented to the user. There are several types of prompts including input, confirmation and selection prompts. The only ones used in the generators itself are the input and the confirmation prompt. These either ask the user to input a string or answer a question with yes or no. The value of the input in case of the confirmation prompt is saved as a boolean value. Therefore, the only constraints from the prompts are: queries represented by the `InputPrompt` require a value of type `String` and queries represented by `ConfirmPrompt` require a value of type `Boolean`. It has to be noted that prompts can be nested. This means a prompt of type `ConfirmPrompt` can contain another builder

composing additional queries if the prompt is selected. Therefore, the `Prompt` DTO has to allow a hierarchical structure. Furthermore, generators can set parameters of there sub-generators by using `InjectPrompts`. Values set by these prompts should not show up in the mapping and have to be filtered out.

5.1.6. Pagen Execution Process

The process of executing pagen is divided in three steps. First is the actual execution of one or more generators followed by the injection of the used configuration for re-usability and lastly the execution of the included tests.

Generation

At the start of the process Pagen is used to generate the desired source code (the artifact of the generation). Since a Java framework was chosen, Pagen can be utilized as a library. Pagen is designed to execute a single generator and output the generated files in the current directory. The service, on the other hand, should be able to run a selection of generators. During execution the files are kept in a Intermediate File Storage (IFS) and are written to disk in the `finalizing` phase of the generator. Therefore, it is a good idea to run the generators in sequence with a common *IFS*. This way, the operations are kept in memory until the complete generation is finished.

Configuration Injection

After the generation the used configuration of the generators is stored in the artifact (requirement 1.5). This enables developers to share useful configurations and minimizes startup times for new projects.

Testing

The last stage of the process is the testing stage. In this phase Maven is used to execute the tests included in the templates used by Pagen to verify if the generated source code is executable (requirement 1.4) [Mav]. Maven is the build system used in the templates and is able run all tests of a project by executing `mvn verify` in a directory containing the Maven Project Object Model (`pom.xml`). As already mentioned, the test execution is optional since it can take time.

5.1.7. Pagen Pipeline

The in section 5.1.6 described process can be realized using the pipes and filters architecture pattern introduced in section 4.4.1. Figure 5.4 shows the execution of Pagen with two generators, the attachment of the configuration and the test execution in form of a pipes and filters pattern. The pattern offers high flexibility making it easy to add new filters, like additional generators (requirement 1.3, 1.10), or to exclude them, like the optional testing filter.

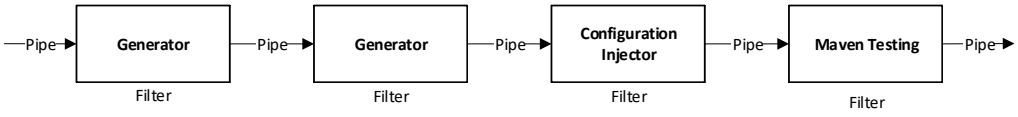


Figure 5.4.: Pipes and Filters Pagen Process

Figure 5.5 shows a concrete class digram of the processes model.

The `Pipeline` and `Filter` interfaces are the definition of the pattern. The pipeline is an active pipe representation that executes all registered filters in sequence and passes the output (artifact) of one filter to the next. The concrete implementation of `JobPipeline` has a artifact named `JobArtifact`. This artifact is the only allowed input and output format of the filters. It contains the common *IFS*, which therefore gets passed from generator to generator.

The three concrete filter types represent the process steps presented in the previous chapter.

To ease the usage of the pipeline it is equipped with a `JobPipelineBuilder` following the builder pattern. A builder pattern eases the construction of complex composed objects by providing methods to add a supported object to the composition [Gam+95]. Here, the builder is used to compose a `JobPipeline` by adding concrete filters to it.

5.1.8. Jenkins Service Component Overview

The Jenkins component of the web service only consists of one DTO , one controller and one service.

Data Transfer Object

The `Request` DTO represents a request by a client. It contains the configurable parameters of the pipeline, like the fragments and variables, as well as, the *URL* to the template repository and the commit *SHA*. The commit is optional and can be used to set a specific version of the pipeline instead of using the newest one.

Controller

Again the `JenkinsController` acts as the definition of the *REST*-Endpoint. It calls the `JenkinService` to generate the build-pipeline.

Service

The `JenkinService` is responsible for assembling the pipeline and starting the execution. Like the Pagen component it again uses a builder. After the pipeline is finished either the generated build-pipeline code gets returned or a build-pipeline code displaying an error message during Jenkins' execution.

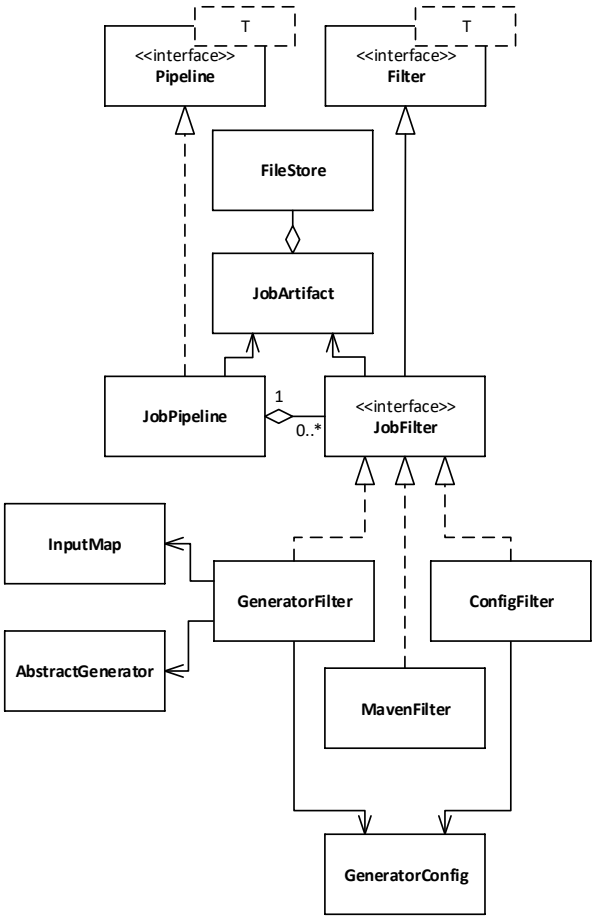


Figure 5.5.: Pagen Pipeline Class Diagram

5.1.9. Jenkins Templating

In order to exclude or include certain parts of a Jenkins build-pipeline, the original build-pipeline has to be split into fragments (requirement 3.1). The original pipeline might contain a segment on which any further step of the build-pipeline depends. In this segment dependencies might be downloaded or the environment might be configured. These steps are vital for the build-pipeline execution and are therefore required in order for a fragment to work. For this reason, a build-pipeline is able to define a **mandatory** fragment, which contains the explained actions. Each fragment, including the mandatory fragments, should be saved in a individual file. These files and a name identifying the fragment are stored in a `tempalte.json` file, which is located in the root of the template repository. Furthermore, the file should contain the location of a template's root directory and a list of variables, which should be provided by the client when a

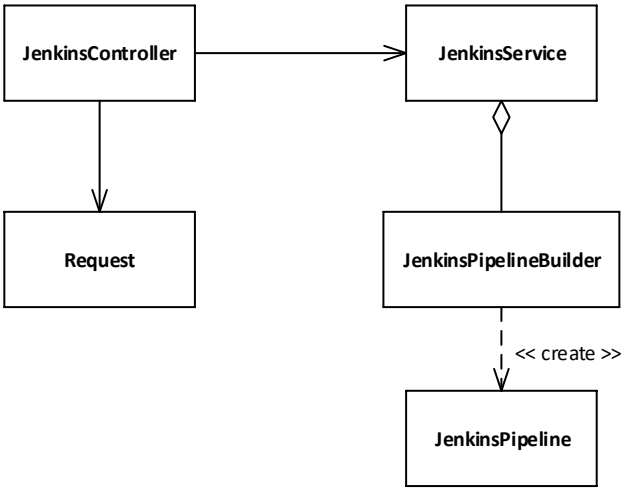


Figure 5.6.: Class Diagram of the Jenkins Component

build-pipeline is requested (requirement 3.2).

5.1.10. Jenkins Build-Pipeline Generation Process

The process of generating dynamic Jenkins build-pipeline can also be divided into several stages. The first stage being the download of the templates from GitLab optionally with a specific commit (requirement 3.3) [Gitb]. After that the `template.json` has to be unmarshalled to be able to read the information. Next, the variables provided by the request can be validated and added to the final build-pipeline, which will be returned after the process. The last stage is the assembly of fragments. For each fragment that is requested the respective files content needs to be added to the final build-pipeline.

5.1.11. Jenkins Pipeline

The in section 5.1.10 described process can again be modeled using the pipes and filters pattern.

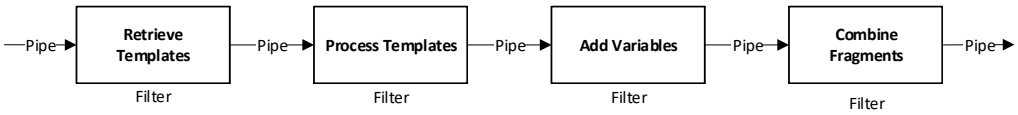


Figure 5.7.: Pipes and Filters Jenkins Process

Like the Pagen pipeline, the Jenkins pipeline uses implementations of the `Pipeline` and `Filter` interfaces to model the architecture.

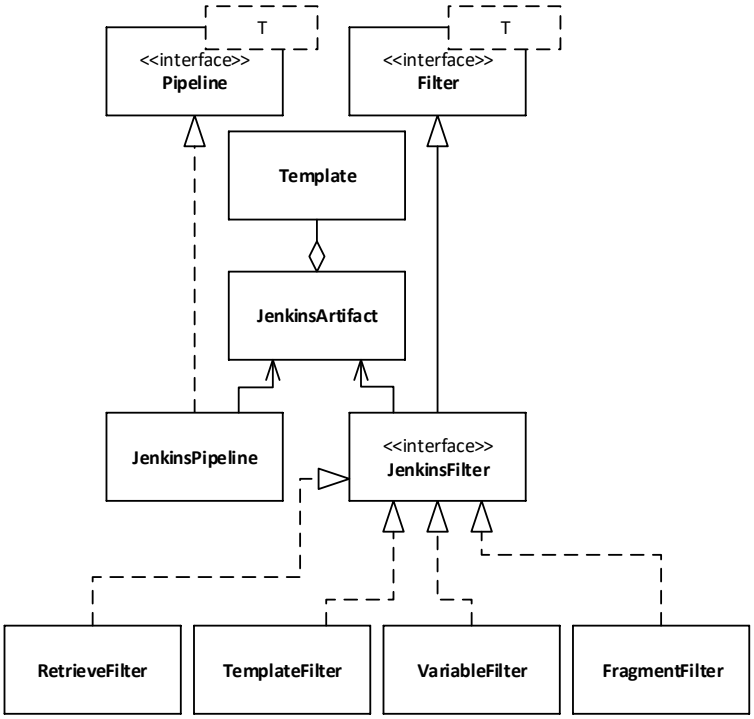


Figure 5.8.: Jenkins Pipeline Class Diagram

In this case the `JenkinsArtifact` contains the `Template` DTO, the actual templates retrieved from the GitLab repository and the current build-pipeline code[Gitb]. Therefore, unlike in the Pagen pipeline, it is not possible to rearrange the filters. Again, the concrete filters represent the steps described in section 5.1.10.

5.1.12. Statistics

To give the generator developer an idea which generators are used frequently the web service offers metrics on what generators are used how often and how many tests have failed (requirement 1.6). To achieve this the service takes advantage of Spring's actuator package to extend the included metrics with statistics [Sprb].

5.2. Extending Pagen with Component Generators

To test the extensibility of Pagen, a generator for the KiScriptServer component, developed by KISTERS AG is added to the generator (requirement 1.8). When choosing the

generator, the KiScriptServer component should be added to the project without any parameters.

To achieve this the application and technology layer has to be extended with fitting naming conventions and a factory to create the convention implementation [Gam+95]. Since the KiScriptServer follows the naming conventions implemented for the ports and adapters architecture the same implementation was chosen for the component. New component specific conventions can always be added afterwards. Additionally, an abstract generator for the software components was added to this layer.

Next, the generator layer has to be extended with the concrete implementation of the KiScriptServer generator. Since the component is already complete it just has to be moved into its own folder.

5.3. Web Application

The web application offers a user-friendly interface to operate the Pagen part of the web service in form of a *Single-Page Application* (requirement 2.1). The user should be able to select the desired generators and provide the required parameters. Furthermore, it should display the user with progress updates and test results, as well as offer the generated artifact as download.

5.3.1. Technology

Vue.js

The web application uses Vue.js as a framework[Vuea]. Vue.js is a progressive Javscript framework with support for creating *Single-Page Applications (SPAs)* using the *MVVM* pattern. It uses a declerative component-based approach with a virtual Document Object Model (DOM) for responsiveness. It also allows the user to define a component in a single .vue file, which includes the components template, javascript functionality and style, with the help of tools like Webpack [Webb].

Vuex

Vuex is a library which is used to manage global state in a Vue.js application [Vueb]. To modify the global state, mutations have to be implemented in order to keep the outcome predictable. The introduction of a global state is useful when dealing with state that is distributed over multiple components. When a component changes the state is mutated globally eliminating the need to gather the states of the individual components.

Bulma

Bulma is a *CSS* framework based on the flexbox layout mode of CSS3 [Bul]. It is a responsive and mobile first.It is therefore usable on any device. In contrast to other frameworks such as Bootstrap, Bulma does not include any JavaScript. Therefore, it

is lightweight while offering all needed components for building a generator front-end. It also allows to implement the needed functionality in the chosen js framework and eliminates the hassle of mixing imperative with declarative paradigms.

5.3.2. Generator Selection

Generators have to be displayed to the user in order to be selected (requirement 2.2). The generators are divided into categories and should be displayed accordingly. A generator has a name and a command which is only used to identify the generator. The easiest representation is to display generators as a list of `checkboxes` with their name as label. Since the amount of generators is expected to increase over time, filter methods are provided to only show generators that generate code of a certain programming language or architecture pattern.

5.3.3. Displaying Query Information

Each generator has a specific set of parameters. These parameters are provided by the web service in form of prompts. Since these prompts vary from generator to generator the union of the parameters has to be computed to eliminate duplicates (requirement 2.3). In section 5.1.5 it was determined that two types of parameters are required: `String` and `Boolean`. `String` parameters can easily be represented by the input type `textfield`, while `Boolean`s can be represented by `checkbox`. These inputs are than labeled with the query message retrieved from the prompts in the web service. Furthermore, the queries of type `Boolean` can contain nested queries, which should only be visible if the corresponding `checkbox` is checked. To clarify a mockup is provided in figure 5.9.

5.3.4. Progress Information

While the web service is working, the web application presents the user with the current log and the status of the execution (requirement 2.4). If the process succeeds, the web application should automatically download the artifact. If it fails, the user should decide if he still wants to download the file (requirement 2.5).

5.3.5. Charts

To display the from the web service computed statistics charts are added to the web application to give a quick an easy to interpret overview of the data.

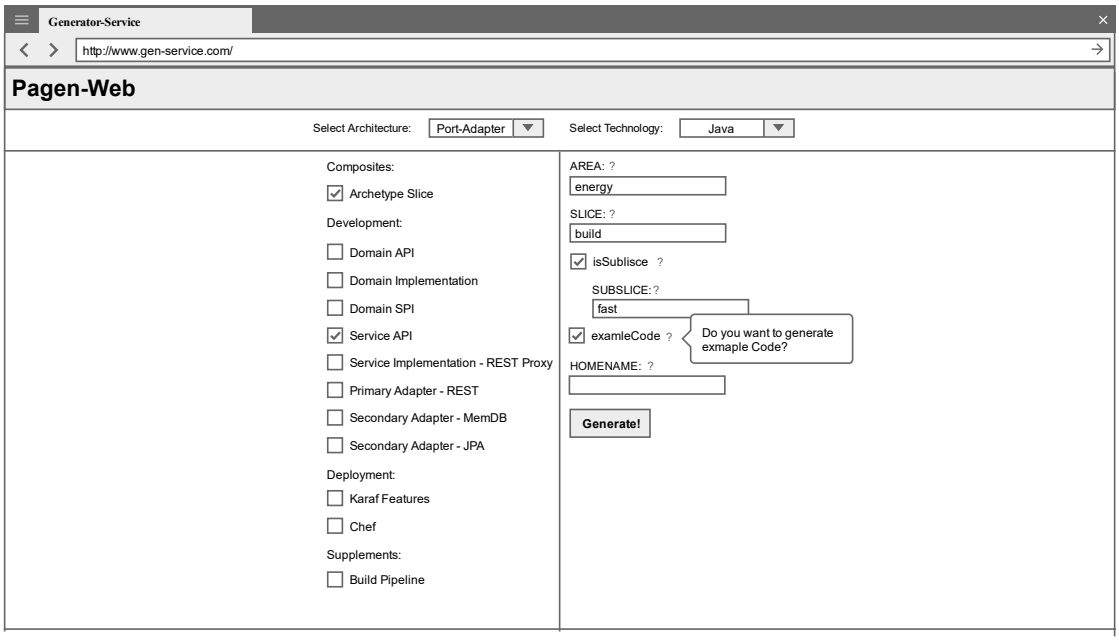


Figure 5.9.: Front-End Mockup

5.4. Jenkins Library

To be able to execute the build-pipeline, generated by the web service, dynamically, Jenkins needs to be extended. This is done by implementing a shared library. Shared libraries are registered inside Jenkins using, for example, a git repository (requirement 3.4, 3.5). They can either be made available globally, for all build-pipeline, or registered to a single build-pipeline. The library also offers a interface to define the name of the desired build-pipeline template, it's version (in form of commit) (requirement 3.3), the variables and fragments to include in the final product. With the help of those parameters it downloads the final assembly and executes it in the same context.

6. Realization

Contents

In this chapter the necessary changes to the Pagen generator and the implementation of the most important concepts explained in chapter 5 are explained [Gee17].

6.1. Pagen Modifications

Since Pagen is designed to be a *CLI*-based application some changes have to be made in order to utilize it as a library.

6.1.1. User Query API

First an foremost, the User Query API has to be modified. Each generator owns an instance of a `UserQueryBuilder` which is a class utilizing the builder pattern. Internally the queries and its name are stored in a map data structure. On generator execution the `UserQueryBuilder` is called with a already given `InputMap` (`HashMap` wrapper containing input data). If more inputs than given are needed, the respective query prompts the user for the input. To avoid these prompts a already complete list of inputs has to be provided to the generator. Therefore, the needed data needs to be known before generator execution. This can be achieved by adding a getter to the `UserQueryBuilder` for the prompt list and a getter to the generators in order to retrieve the builder.

6.1.2. Generator Life-cycle

The next step of retrieving the inputs is to change the generators life-cycle. In the original life-cycle model the generator specific queries are added in the `configuring` and the builder is executed immediately afterwards. This gives no opportunity to retrieve the queries without getting prompted for input. Therefore, the `configuring` step is split into a `initializing` and a `configuring` step. The new `initializing` step takes care of the query configuration and the registration of sub-generators, while the `configuring` step executes the builder and evaluates the generated `InputMap`. The modified generator life-cycle is presented in figure 6.1.

Next there needs to be a way to inject a complete list of inputs before the generator's execution. Therefore, the generators a equipped with an additional `run` method that accepts a configured `InputMap`.

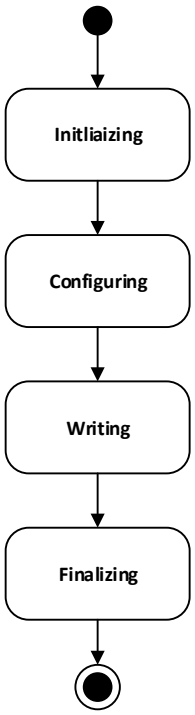


Figure 6.1.: Modified Generator Life-cycle

6.1.3. Output Formats

Apart from the user query changes the generated code format has to be adjusted. In addition to generating files and folders on the operating systems file system, there is now the need for archive support. The Intermediate File Storage (IFS) is responsible for

handling the files and writing them to disk with the `commit` method. By creating an abstract class with an abstract `commit` template method the implementation of multiple output formats is a matter of implementing different specific file stores [Gam+95]. The original `commit` implementation is extracted into a `DiskFileStore` and a new implementation for *ZIP*-archives is added in form of `ZipFileStore`. The relationships are shown in figure 6.2.

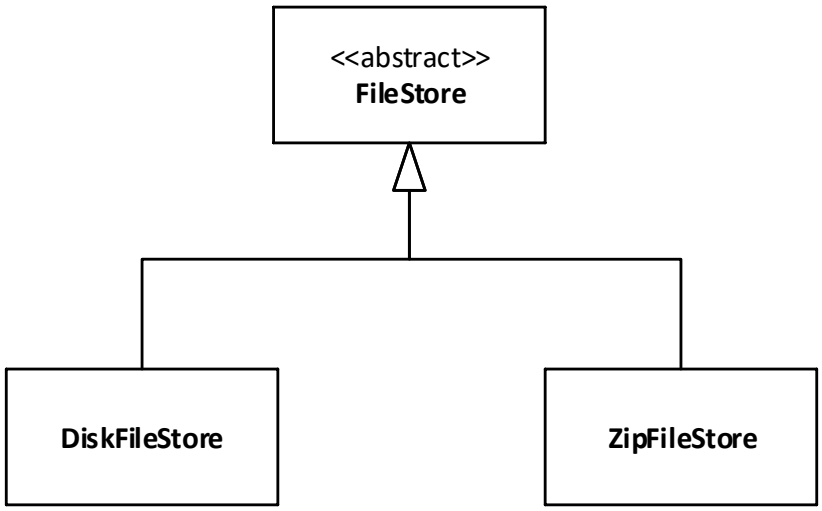


Figure 6.2.: FileStore Class Relationships

Again the generator `run` method needs to be modified to allow a selection of output formats. The final method signatures are presented in listing 6.1.

```
1 public void run(String... args) throws IOException
2 public void run(InputMap inputMap, OutputFormat format, Path path)
    throws IOException
3 public void run(InputMap inputMap, FileStore fileStore) throws
    IOException
4 public void run(InputMap inputMap, OutputStream out) throws
    IOException
```

Source Code 6.1: Generator run methods

The first signature is derived from the `Task` interface and used by the Maven plugin. The others allow the injection of a pre-configured `InputMap` as introduced in section 6.1.1. Furthermore, they allow output format configuration by either choosing a format and a destination or by providing an own `FileStore`. The signature in line 4 is used to write the archive into a desired `OutputStream`.

6.1.4. Metadata and Generator Registration

The current implementation requires a hard-coded list of generators and getter methods to configure meta-information. By introducing a `RegisterGenerator` annotation the meta-data can be separated from the generators logic and offers the opportunity to import generators by scanning for the annotation in a base-package. This eases the integration of new generators and allows for filtering by, for example, technology. The implemented annotation includes fields for architecture, technology, category and a description. A listing is provided by listing 6.2

```
@Retention(RetentionPolicy.RUNTIME)
public @interface RegisterGenerator {
    String description() default "";
    Architecture architecture() default Architecture.OTHER;
    Technology technology() default  Technology.OTHER;
    Category category();

    enum Architecture {
        PORT_ADAPTER,
        MVVM,
        OTHER
    }
    enum Technology {
        JAVA,
        ANGULAR,
        OTHER
    }

    enum Category {
        COMPOSITE,
        DEPLOYMENT,
        DEVELOPMENT,
        SUPPLEMENT,
        COMPONENT
    }
}
```

Source Code 6.2: RegisterGenerator.java

The `@Retention(RetentionPolicy.RUNTIME)` is required to keep the annotation in the compiled byte-code, otherwise evaluation at runtime would not be possible.

All that needs to be done to register a new generator is now to add a annotation, like shown in listing 6.3, in front of a generator class.

```
@RegisterGenerator(
    category = RegisterGenerator.Category.COMPOSITE,
    architecture = RegisterGenerator.Architecture.PORT_ADAPTER,
    technology = RegisterGenerator.Technology.JAVA
)
```

```
6 public class PortAdapterArchtypeGenerator extends
      AbstractPortsAndAdaptersGenerator {
```

Source Code 6.3: Archetype Generator Registration

6.2. Web Service

This section introduces the most important implementations of the concepts introduced in section 5.1.

6.2.1. Generator Discovery

Spring Boot detects its components by scanning the with `@SpringBootApplication` annotated classes package and sub-packages [Spra]. This allows the user to skip the process of registering all beans in a *XML* file. The same technology can be utilized for the generator discovery. Since all generators are already annotated with `@RegisterGenerators` (see section 6.1.4) all that needs to be done is to configure Spring to search for the annotation in the `de.kisters.pagen` package.

This is implemented by providing the Spring Java configuration shown in listing 6.4.

```
1  @Configuration
2  @ComponentScan(
3      basePackages = {"de.kisters.pagen"},
4      includeFilters = {
5          @ComponentScan.Filter(type = FilterType.ANNOTATION,
6          value = RegisterGenerator.class)
7      },
8      nameGenerator = GeneratorNameGenerator.class,
9      scopeResolver = GeneratorScopeMetadataResolver.class
10 )
11 public class GeneratorConfig {
12 }
```

Source Code 6.4: GeneratorConfig.java

The `basePackages` attribute points Spring to the start of the package tree to search, while `includeFilters` configures it to search for classes annotated with `@RegisterGenerator`. The default Spring naming strategy for beans is to use the class name, but since the generators have a predefined command to identify them a custom naming strategy is provided. This is achieved by implementing the `generateBeanName` method of the `BeanNameGenerators` interface. The listing 6.5 shows the implementation.

```
1  @Override
2  public String generateBeanName(BeanDefinition definition,
       BeanDefinitionRegistry registry) {
3      try {
```

```
4           Class<?> clazz =
                Class.forName(definition.getBeanClassName());
5
6           for (PropertyDescriptor pd :
                Introspector.getBeanInfo(clazz)
7                .getPropertyDescriptors()) {
8                if (pd.getReadMethod() != null &&
                    pd.getReadMethod().getName().equals("getCmd") &&
                    !pd.getName().equals("class")) {
9                    return (String)
                        pd.getReadMethod().invoke(clazz.newInstance());
10               }
11          }
12
13          return clazz.getSimpleName();
14      } catch (Exception e) {
15          springLogger.error(e.getMessage());
16      }
17
18      return definition.getBeanClassName();
19  }
```

Source Code 6.5: GeneratorNameGenerator.java

The generator command is defined in a getter method, which needs to be executed using reflection. This is done by iteration over the class properties until the `getCmd` method is found. If existent, it is invoked on a new instance and returned. As a fall-back, the name of the class is returned.

Last but not least, the bean scope has to be defined. A new instance of a generator is required for every request. To achieve this the bean scope has to be set to `request` (defaults to `singleton`). This is again realized by implementing another interface.

```
1   public class GeneratorScopeMetadataResolver implements
        ScopeMetadataResolver {
2       @Override
3       public ScopeMetadata resolveScopeMetadata(BeanDefinition
            definition) {
4           ScopeMetadata metadata = new ScopeMetadata();
5
6           metadata.setScopedProxyMode(ScopedProxyMode.NO);
7           metadata.setScopeName("request");
8
9           return metadata;
10      }
11  }
```

Source Code 6.6: GeneratorScopeMetadataResolver.java

The `resolveScopeMetadata` method presented in listing 6.6 sets the scope to `request` and disables the scope proxy. The scope proxy is not needed, since the service using the generators has the same scope as the generators.

6.2.2. Pagen User Query Mapping Implementation

The algorithm used to parse the `UserQueryBuilder` object is encapsulated in a **strategy** pattern shown in figure 6.3 [Gam+95]. The context is here represented by the `GeneratorService`. Listing 6.7 shows the implementation.

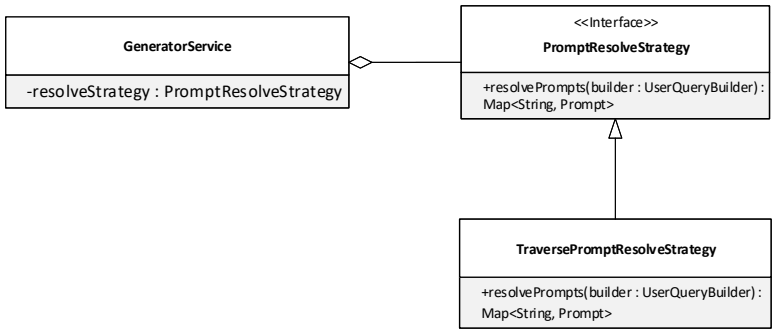


Figure 6.3.: Prompt Resolve Strategy Class Diagram

```
1  public class TraversePromptResolveStrategy implements
       PromptResolveStrategy {
2      @Override
3      public Map<String, Prompt> resolvePrompts(UserQueryBuilder
           builder) {
4          Map<String, Prompt> prompts;
5          List<String> remove;
6
7          prompts = new LinkedHashMap<>();
8          remove = new ArrayList<>();
9
10         builder.getPrompts().forEach((key, value) -> {
11             if (value instanceof ConfirmPrompt) {
12                 ConfirmPrompt confirm = (ConfirmPrompt) value;
13                 // Nested prompts?
14                 Prompt promptModel = new Prompt("Boolean",
                       value.getMessage());
15                 if (confirm.getBuilder() != null)
16                 promptModel.setAdditional(this.resolvePrompts(confirm.getBuilder()));
17                 prompts.put(value.getName(), promptModel);
18             } else if (value instanceof InjectPrompt) {
19                 if (!remove.contains(key))
20                 remove.add(key);
```

```
21                } else
22                prompts.put(value.getName(), new Prompt("String",
                      value.getMessage()));
23            });
24
25            // Delete all predefined entries
26            remove.forEach(key -> prompts.remove(key));
27
28            return prompts;
29        }
30    }
```

Source Code 6.7: TraversePromptResolveStrategy.java

The algorithm starts at the top-level builder and parses the registered prompts. If a prompt is an instance of `ConfirmPrompt`, it's message and name gets stored in a `Prompt` DTO with the addition of a value type of `Boolean`. The same is applied to `InputPrompts`, but with the value type of `String`. `ConfirmPrompts` can additionally have nested prompts, which are stored in another `UserQueryBuilder` instance in the prompts object. The nested builder's prompts are added to the current list of prompt DTOs using recursion. The third type of prompt is the `InjectPrompt`. These are used to set values for prompts that are already registered. Since these values are already set they do not need to be presented to the user and are removed after the traversal. A result of the traversal in *JSON* format can be examined in appendix D.

6.2.3. Pipes and Filters Implementation

The definition of the pipes and filters architecture pattern described in section 4.4.1 is implemented in the listings 6.8 and 6.9.

```
1  public interface Pipeline<T> {
2      void addFilter(Filter<T> filter);
3      void execute();
4      T getArtifact();
5      boolean isSuccessful();
6  }
```

Source Code 6.8: Pipeline.java

The pipes are implemented as an active pipeline, which has a list of filters that are applied in sequence. The pipeline passes the output of the previous filter in form of an artifact to the next as an input. Additionally, the pipeline provides a `Boolean` indicating if the filters were executed successfully.

```
1  public interface Filter<T> {
2      T execute(T artifact);
3      boolean isSuccessful();
```

```
4 }
```

Source Code 6.9: Filter.java

The filters are passive in the implementation. They get called by the pipeline and provides the result of the operation as a return object of the `execute` method. Whether the filter was successful is indicated by the `isSuccessful` getter method.

6.2.4. Pagen Pipeline

The implementation of the Pagen pipeline is contained in the package `de.kisters.genservice.pagen.pipeline`. The `JobArtifact` represents the input and output format of every filter. It acts as a container for the file store used in the generators and includes a method to generate the final *ZIP*- file. Every generator executes the `commit` method of the file store in its `finalizing` phase, which causes the file store to generate the zip after every generation. Since mulitple generators are run in sequence a custom file store listed in listing 6.10 is used to omit the output until all generators are finished.

```
1  public class ReusableZipFileStore extends ZipFileStore {
2      public ReusableZipFileStore(OutputStream out) {
3          super(null, null, out);
4      }
5
6      @Override
7      public void commit() {
8          // Skip commit to allow reuse in other generator
9      }
10
11     public void finalCommit() {
12         // Call ZipFileStore.commit() to create ZIP file
13         super.commit();
14     }
15 }
```

Source Code 6.10: ReusableZipFileStore.java

Like introduced in section 5.1.6 there are three filters implemented in the pipeline.

GeneratorFilter

The `GeneratorFilter` uses the `run` method added in section 6.1.3 to provide the respective Pagen generator with the in the `JobArtifact` wrapped file store and thereby executes a single generator.

ConfigFilter

The `ConfigFilter` utilizes `Jackson` to convert the generator configuration, provided as the body of the received `POST`-request, to a *JSON* file, which is added to the file store. Posting the file to the service therefore produces the same output as the current execution of the service.

MavenFilter

To be able to execute the tests, the *ZIP* first has to be generated by the file store and extracted to a temporary directory. The `JobArtifact` offers a `getBytes()` method, which calls `finalCommit()` of the file store and returns the *ZIP* file as bytes.

The `MavenFilter` uses the `MavenInvoker` package to execute the `mvn verify` command on the project object model included in the generated source code.
The `MavenInvoker` uses the pre-installed Maven executable located in the `MAVEN_HOME` folder with the settings located in the `M2_HOME`.

The pipeline (`JobPipeline`) executing the filters is implemented as a thread. This is due to the possibly long execution time of the generators and `MavenFilter`. A client, which has requested a generation, might timeout before the pipeline is finished. Therefore, the execution is run as a background tasks, while the client can request status updates including the current execution log. After the pipeline is finished, the client receives a artifact id, which he can use to retrieve the generated source as a *ZIP*. The client has also the ability to cancel the pipeline. If he chooses to do so, the thread gets terminated after the current filter finishes. The filters are made atomic to avoid leaving waste on the disk. The `MavenInvoker` and Pagen are both not designed to be interrupted and therefore offer no cleanup methods to remove any temporary files created.

To ease the usage of the pipeline, a `JobPipelineBuilder` is implemented, which offers a *DSL* to configure a pipeline.
`builder.generator(gens, "archetype", inputs).config(config).maven();`
would create a pipeline with a `GeneratorFilter` executing the `archetype` Pagen generator, a `ConfigFilter` adding the used configuration to the file store and a `MavenFilter` executing the test. By calling `execute()` the builder would start the pipeline and return the reference.

6.2.5. Jenkins Pipeline

The Jenkins pipeline is simpler than the Pagen pipeline and does not require the execution to run in the background. The generation of the pipeline code is quick enough for the client to not timeout and therefore the `JenkinsPipeline` is not implemented as a thread.

Again, the process is as described in section 5.1.10 is implemented as four filters.

RetrieveFilter

The `RetrieveFilter` is responsible for downloading the templates from GitLab. It uses Java's `FileSystem` class to open the downloaded archive and stores the information in the `JenkinsArtifact` container.

TemplateFilter

After the `RetrieveFilter` the `TemplateFilter` is executed and resolves the `template.json` file inside the *ZIP*'s root directory. It than uses `Jackson` to parse the content into the respective `Template` DTO and adds it to the artifact container.

VariableFilter

The `VariableFilter` is used to check the provided variables for completeness and checks their types. It than adds the variables and their values to the top of the final build-pipeline.

FragmentFilter

The `FragmentFilter` is the final filter of the Jenkins pipeline. It resolves the selected fragments with help of the `Template` domain model and adds them to the final build-pipeline code.

Like for the Pagen pipeline, the Jenkins pipeline also includes a builder in form of `JenkinsPipelineBuilder`. The complete configured pipeline looks as shown in listing 6.11.

```
1 builder.retrieve(templateUrl, commit).template().variable(name,
     variables).fragment(name, fragments).execute();
```

Source Code 6.11: Jenkins Pipeline

6.2.6. Statistics

Statistics are provided using the Spring Boot Actuator [Spra]. The Spring Boot Actuator generates automatically generates metrics for the applications uptime, memory usage and others. It also allows to add custom metric data by using the `CounterService`. The `CounterService` can be retrieved using Spring's dependency injection and offers methods to increase or decrease a specific counter by a string name representation. These additional counters are automatically added to the metrics endpoint and are used to create the generator statistics.

6.2.7. REST-Endpoints

The *REST*-Endpoint presented in table 6.1 are the result of the controller classes introduced in this chapter.

All Endpoints with prefix `/api/pagen/generators/` are implemented in the s `GeneratorController` of the Pagen component. They can be used to list all available generators, execute a single generator by providing input parameters or executing multiple generators using the `GeneratorConfig`. Example *JSON* results can be found in appendix D.

Endpoints prefixed with `/api/pagen/job/` refer to a running `JobPipeline` and are used to retrieve the status information or cancel a specific job.

After a successful generation the result of the pipeline is available to download via the `/api/pagen/artifacts/` endpoint.

The endpoint `/api/jenkins` belongs to the `JenkinsController`. It is used to provide the build-pipeline configuration and returns the generated build-pipeline.

Metrics and generation statistics can be viewed with the help of the `/metrics` endpoint generated by the Spring Actuator package. It contains the current performance metrics and different generation statistics.

The web service also offers a live documentation powered by Swagger, which can be found under the `/swagger-ui.html` *URL* [Swa].

REST-API Endpoints		
Request Type	Endpoint	Description
	Generators	
GET	/api/pagen/generators	List of available generators
POST	/api/pagen/generators/{cmd}	Run generator {cmd}
POST	/api/pagen/generators	Run generators
	Jobs	
GET	/api/pagen/jobs/{jobId}	Get status of job {jobId}
DELETE	/api/pagen/jobs/{jobid}	Cancel job {jobId}
	Artifacts	
GET	/api/pagen/artifacts/{artifactId}	Download artifact {artifactId}
	Jenkins	
POST	/api/jenkins/{name}	Retrieve pipeline {name}
	Spring Actuator	
GET	/health	Application health information
GET	/info	General information
GET	/metrics	Metrics and generator statistics
GET	/trace	Last HTTP requests

Table 6.1.: List of *REST*-API endpoints

6.3. Extending Pagen with Component Generators

First the component had to be made executable on its own. The *POM* of the module contained a reference to a parent component. This reference was deleted and the project was able to build with `mvn verify`. Next, a `template.json` had to be added for Pagen to work. Since the component should not be parameterized, the file was left empty.

The current supported types in Pagen are Angular components and Port Adapter implementations. To allow complete components,
a `AbstractSoftwareComponentGenerator` class was added to the `pagen-core` module. It does not contain any implementations except from the User Query Builder initialization.

Next is the implementation of the concrete generator in the `pagen-templates` module. This required setting the repository (in the constructor) and copying the files, located in the root of the repository, in its own folder inside the `writing` life-cycle method. The implementation is shown in listing 6.12.

```
1  @Override
2  public void writing(Path destinationPath) {
3      Path sourcePath = Paths.get("");
4      Path targetPath =
           destinationPath.resolve(convention.getFullName());
5
6      newGeneration(fileStore)
7      .from(sourcePath)
8      .to(targetPath)
9      .copyDirectories();
10 }
```

Source Code 6.12: ScriptServerComponentGenerator writing method

To register the generator all that was needed is to add the `@RegisterGenerator` annotation for the service to discover it.

This should show that with the help of the new generator discovery and the current Pagen interface, the implementation of new architecture components can be done with little effort.

6.4. Web Application

The web application implementation uses the Vue.js webpack template [Tem]. The template includes a complete webpack configuration with minification, Vue single file components and hot reload. The configuration was slightly modified to configure a development proxy for the web service and to be able to include the code in the *JAR*.

The list of generators is implemented as a list of generator components. Each of these generator components displays its name and a `checkbox`. If a generators `checkbox` gets checked, the corresponding command gets added to the list of activated generators in the global state store managed by Vuex [Vueb].

The same technique is also used to store the query information. Each prompt is represented by either the `InputPrompt` or the `ConfirmPrompt` component. Since these share a large portion of their functionality, a `mixin` is used to outsource these parts. A `mixin` is Vue's form of abstraction and "mixes" its JavaScript properties into the properties of the components using this mixin [Vuea]. Again, the query values are stored in the Vuex store.

Both component types now store their value in a central data store, which makes is easy to compose the *AJAX* request without having to traverse all components.

After a generation is initiated, the web application polls the current status of the execution every three seconds and allows the user to download the generated source code after it is finished.

All *AJAX* requests are implemented in a plugin making it easy to swap out the library or change the *REST*-endpoints.

The resulting web application can be examined in figure 6.4, further screen-shots can be found in appendix C.

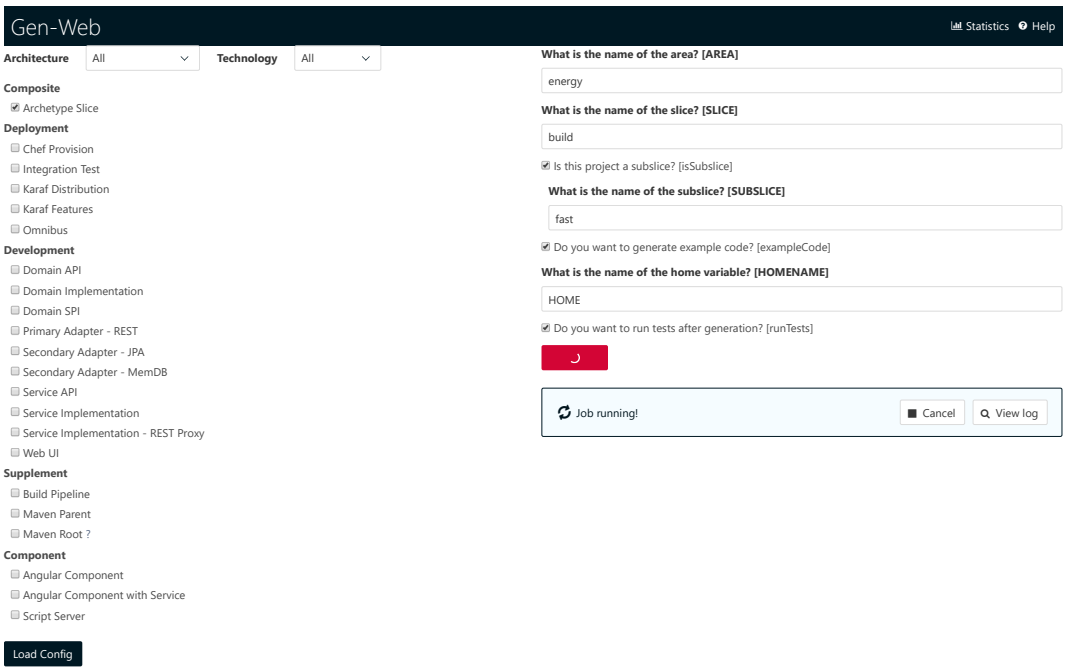


Figure 6.4.: Web Application Screenshot

6.5. Jenkins Shared Library

The Shared Library registers the function `loadPipeline` globally. The implementation of the function is shown in listing 6.13.

```
1  import groovy.json.JsonOutput
2
3  def call(Closure body) {
4      def config = [:]
5      body.resolveStrategy = Closure.DELEGATE_FIRST
6      body.delegate = config
7      body()
8
9      if (!config.url || !config.templateUrl || !config.fragments ||
           !config.variables ||
10     !(config.fragments in List) || !(config.variables in Map))
11     error 'Invalid config!'
12
13     def data = [
14     templateUrl: config.templateUrl,
15     commit: config.commit,
16     fragments: config.fragments,
17     variables: config.variables
18     ]
19
20     def json = JsonOutput.toJson(data)
21
22     // Retrieve pipeline from service
23     def response = httpRequest consoleLogResponseBody:
           config.debug, contentType: 'APPLICATION_JSON',
24     httpMode: 'POST', requestBody: json, url:
           "${config.url}/${config.name}",
25     validResponseCodes: '200'
26     def exec
27
28     // Load pipeline
29     node("master") {
30         writeFile file: "pipeline.groovy", text: "def call()
               {\n${response.content}\n}\nreturn this;"
31         exec = load "pipeline.groovy"
32     }
33
34     // Run pipeline
35     exec()
36 }
```

Source Code 6.13: loadPipeline.groovy

The `call` function gets executed everytime the `loadPipeline` function gets called.

The parameter `body` of the function refers to the body of the `loadPipeline` block (since parentheses can be omitted in Groovy) [Gro]. The block is used to provide the configuration and is parsed into a `Map` by setting the delegate context to the desired variable. After executing the closure the `config` map contains every variable and its value defined inside the `body` block.

Now, the existens of the required fields in the configuration is validated and the build-pipeline is requested from the back-end using the `HTTPRequestPlugin`.

To execute the downloaded build-pipeline it has to be written to disk first in order to be loaded by the `load` Jenkins step. For the `load` step to work, the build-pipeline has to be wrapped inside a `call` function. This has the consequence that no functions can be used in the build-pipeline templates and have to be replaced by closures.

A usage overview of the library can be found in listing 6.14

```
1 loadPipeline {
2     url = ''// URL to the service
3     templateUrl = ''// URL to the template ZIP (retrieve from
          gitlab)'
4     name = '' // Name of the template project
5     fragments = [] // Fragments to include
6     debug = true // Output pipeline before execution?
7     variables = [:] // Map of variables (needs to match definition
          in template.json)
8 }
```

Source Code 6.14: Library Overview

7. Evaluation

Contents

In this chapter the generator service developed will be evaluated by checking whether the requirements introduced in chapter 2 are matched. Afterwards the quality of code is examined using a SonarQube analysis [Son].

Following the code quality some results of a evaluation done at KISTERS AG are presented by discussing requested features.

In the last part of the chapter some possible improvements for the current realization and concept are introduced.

7.1. Requirement Analysis

In this section the requirements introduced in chapter 2 are compared to the implemented functionality.

7.1.1. Web Service

The web service includes the implementation of the requirements presented in section 2.2. In this section the functional and non-functional requirements are evaluated in there degree of implementation.

Generator Discovery

Adding a new generator to the list of available generators should not require a manual registration. For this reason Spring's component scan is used to search for generators in a specific project package and its sub-packages. This makes it easy to add or remove generators from the list without having to modify the service.

7.1.2. Generator Information

The implemented User Query API mapping algorithm adjusts to changed generator parameters and also supports nested prompts. It only implements the currently used types and has to be extended to allow for new prompt types.

Multiple Generations and Artifact

The generator should be able to execute multiple generators. Pagen, by design, only allows to execute one generator at a time [Gee17]. During execution, the user is prompted to select one of the available generators, fill in the parameters and start the generation. The generator creates the files in the current directory and terminates.

To allow for the sequential execution of generators a modified *Intermediate File Storage* was used. This *IFS* is shared between the generator executions and does not write to disk until all generations are finished. This allows to keep all data in memory until an output is required and is especially important if the generated artifact should be available as an archive. Instead of generating the files on disk and compressing them in a *ZIP* archive, the complete source code is written directly from memory in a ZIP format. It therefore decreases waste and processing on storage media, such as *Hard Disk Drives (HDDs)*.

Test Execution

In the presented implementation the last filter executed in the Pagen pipeline runs Maven to verify the generation. Pagen templates should be executable code and therefore stay executable after the generation process. The automatic verification makes sure the user known whether the generation was successful before including the source code in his project.

Incremental Use

The concept of incremental usage as implemented by Pagen is not easy to translate to a web service. A web service requires the user to download the generation and extract the archive. For the original implementation of Pagen as a Maven plug-in this is not necessary. The plug-in can directly generate the new component's files inside the project. To achieve the same effect with a web service a client implementation has to be added. This client has to download the generated sources and extract it into the projects folder.

To at least allow the user to regenerate a certain project or improve its generation, a configuration file is added to the artifact. This configuration includes the selected genera-

tors and their parameters. Creating a `POST` request to the `/api/pagen/generators` endpoint with the configuration as body will result in the same source code as the original generation. This enables a developer to share his presets or store them for future projects. It also allows to slightly modify the generation if something was missing.

To sum up, incremental usage is supported, but for effective usage an additional client is needed. The reuse of generation presets is however supported and allows for fast project initialization and sharing between developers.

Reusability

The web service executing the code generator should be reusable. This should allow the implementation of different front-ends, like plug-ins for IDEs. By choosing a *REST*-API for the interface implementation the only requirement for a client is to be able to send *HTTP* requests. The endpoints are documented using Swagger, which also presents the used *JSON* format. This makes it easy for third parties to implement their own client.

One inconvenience of the interface is the implementation of a job's status. The status of a generation has to be checked frequently to be able to determine when the final artifact is available for download. This means unnecessary overhead for the implementing client if it does not require a continuous update of the progress. One of the limitations of Representational State Transfer (REST) is the data retrieval. If a client request an endpoint it always returns all available information. In the case of the progress retrieval this always leads to the inclusion of the process log. If this data is not required in the consuming application it presents more unnecessary overhead.

All in all the implementation of a *REST*-API offers a easily adaptable interface for third parties.

Extensibility

The processes implemented in the web service are easily expendable. This is possible due to the usage of the pipes and filters pattern. It allows create new process steps by implementing the respective filter interface. Since the functionality of all filters is encapsulted the only change that needs to be made is to add the filter to the current pipeline. Furthermore, there are no dependencies between filters. Therefore, the order of filters can be changed to serve the requirementes without breaking the system.

7.1.3. Web Application

In this section the implemented features of the front-end are compared against the requirements listed in section 2.3.

The front-end is developed in form of Single-Page Application. It supports the selection of generators and dynamically adjusts the parameter form. The parameters presented display the same input messages as the Pagen plug-in and support nested prompts. During the execution of a generation process it is able to present the current execution

log, as well as, the test results. If tests failed, the download of the generated artifact is optional while it is automatically downloaded if all tests passed.

Statistics are offered in form of charts that give a quick overview over the most used generators and might indicate which of them can be composed into a composite generator.

To increase the learning experience, a help modal is added. It gives a quick overview about the interface components and thereby presents a basic work-flow. This helps reduce the time needed to get started with the generator and thereby increases efficiency.

The intuitiveness of the user interface is evaluated by asking employees of KISTERS AG for their opinion. The results are presented in section 7.3.

7.1.4. Dynamic Jenkins Pipelines

In this section the requirements of section 2.4 are evaluated against the library and template implementation.

Decomposability

The generator should allows pipelines to be decomposable. This means parts of the pipeline should be optional and can be excluded. This is implemented in the generator by introducing fragments. The original pipeline is divided into fragments, which can be included into the result. All of these are optional (except the mandatory fragment) and therefore allows the developer to only choose the fragments that are required for a project.

Parameterization

Pipelines should be configurable through the usage of parameters. These parameters are implemented as variables. If a client sets a parameter, it gets added as a variable at the top of the pipeline script. This makes it globally available inside all blocks of the pipeline.

By using variables instead of replacing placeholders the development process of pipelines is simplified. Instead of having to replace every occurrence of a variable or value by a placeholder, the developer can define his test parameters as variables at the top of the script and delete them afterwards. This makes the error prone replacement of values obsolete and eases the development process.

The included type checks make sure that unexpected behavior in the execution caused by implicit type casting of variables can not happen. The user receives an error if a variable is missing or if its type is incorrect before the build-pipeline's execution.

7.1.5. Template Version

To achieve reliability a commit parameter is added to the services request. This commit allows the developer to set a specific template version. Thereby an error in the template does not necessarily result in broken build-pipelines. Before a developer decides to update

the used template version of his project, he is able to check for any errors and does not get surprised by failing pipelines not caused by faulty code.

Maintainability

Pipelines should be editable in one location resulting in an updated pipeline for all using projects. This is achieved by serving the generation as a service and pulling the templates from GitLab. This way all changes in the git repository resolve in changes in the using pipelines.

Integration

To integrate the library into Jenkins, the *SCM* link hast to be added to the Jenkins configuration. There are now two options to use the library. Either the used version is fixed in the Jenkins settings or hast to be defined in any using pipeline. It is also possible to include the library automatically in every pipeline. This way, the library does not have to be specifically imported at the beginning of every script, but forces all pipelines to use the same library version.

This makes the integration and update of the library very convenient in comparison to an implementation as a Jenkins plug-in.

7.2. Code Quality

To ensure the quality of the implemented service and web application SonarQube was added to the used build-pipeline [Son]. As shown in the report screen-shot provided in figure 7.1 no bugs, vulnerabilities or code smells were found in the source code.

The source code sums up to 1.987 lines of code, which only includes Java and JavaScript code (no *HTML*, *CSS*, etc.). This is not a considerably large amount of code for the implemented functionality. As a result it is easier to manage than larger code bases and thereby decreases the amount of work needed to fix bugs.

The small amount of code is partly a result of the usage of libraries like Lombok, which was used to automatically create getters and setters, as well as, using the declarative Vue.js [Lom; Vuea].

7.3. Evaluation at KISTERS AG

To evaluate the usability of the service, the web application and Jenkins library and template structure was presented to two employees of KISTERS AG. They were given an overview of the user interface and how to use it, as well as, a usage example of the Jenkins library and their template implementation. Both were handed an evaluation sheet (see appendix A) to rate the usability and implemented functionality. The Likert scale presented in table 7.1 was used to measure the satisfaction of the employees. Results of the evaluation are available in appendix B

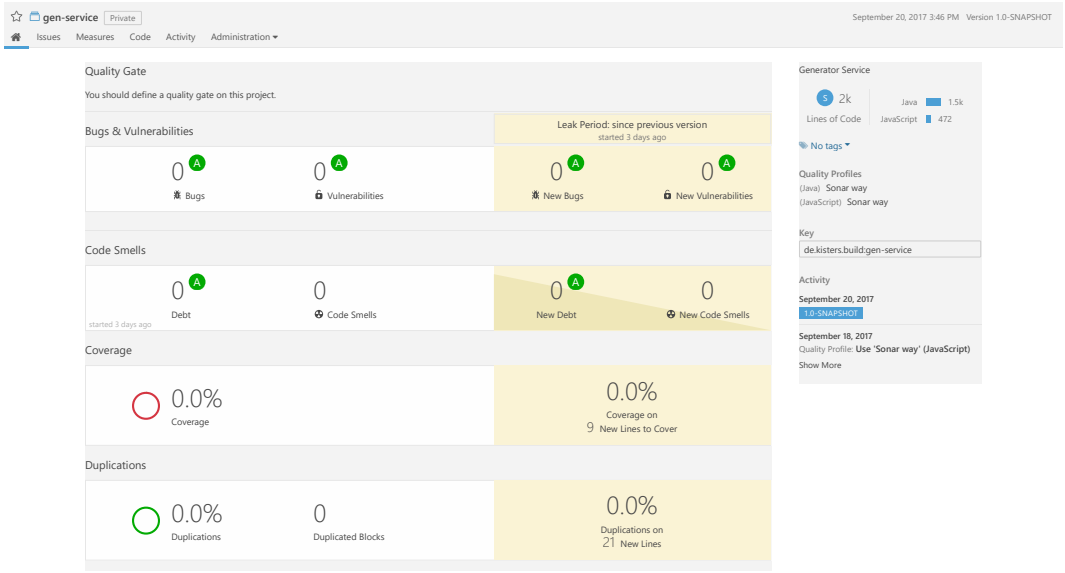


Figure 7.1.: SonarQube Report [Son]

The used terms in the application (question 1.4) achieved the worst rating. This is due to acronyms like MVVM, which are used in the filter options. These are directly retrieved from the architecture defined in the annotation introduced in section 6.1.4. It, however, can be expected for developers using the generator to know the acronym as they work with the pattern every day.

Some suggested improvements are the integration of links to the used template. This knowledge is however only available to Pagen and can therefore not be implemented in the service. Another suggestion was the addition of examples to required generator parameters. This information is, again, provided by Pagen and can therefore not be added in the service.

Another requested feature is the encoding of the current configuration inside the *URL* to be able to share it with co-workers. This is somewhat already implemented as the configuration file added to the archive. It can be loaded inside the *UI* and shared between employees. The addition of an *URL* encoding might be more convenient in some cases, but does not offer any real improvements.

A feature requested, which was already implemented after the evaluation, is a prefix in the generated file's filename. The previous version would use the artifact identifier as the name for the archive. The added prefix should eliminate any confusion about the file's

origin.

	Likert Scale
1	Strongly disagree
2	Disagree
3	Neutral
4	Agree
5	Strongly agree

Table 7.1.: Likert Scale [AS07]

7.4. Discussion

In this section some design choices and their advantages and disadvantages are discussed.

7.4.1. Pagen Component

When taking a look at the Pagen components class diagram it becomes obvious that the `GeneratorService` has many relationships. The class manages a large portion of the components responsibilities, which is not a good design. However, the class itself does not implement much functionality at all, it mostly delegates work to other classes. It acts as a facade for the Pagen generation pipeline and user query mapping strategies.

All discovered Pagen generators are owned by this class and are passed to the `GeneratorFilter` for execution. This was done to ensure there are no dependencies between the pipes and filters implementation and Spring Boot. This allows to reuse the pipeline in other Spring independent projects without having to make any changes.

A possible improvement would be to divide the object into a new class handling the query mapping and a class handling the pipeline configuration.

The Pagen generator pipeline uses a thread to be executed in the background. This could be improved by introducing thread pools to limit the load on the server. Since a generator can be expected to be only used occasionally this should not be an issue in general usage.

7.4.2. Jenkins Component

The Jenkins component of the service has its own implementation of basic code generation. The generation functionality is implemented in the `VariableFilter` and `FragmentFilter` classes. It currently does not depend on Pagen since the generation process of single file build-piplines is very straight forward. An integration into Pagen would, however, be desirable to take advantage of the concepts provided by the generator. It would allow composition of build-pipelines spread over multiple files while the capability

of merging files has to be added. The current implementation can be seen as a proof of concept for the idea of generating build-pipelines dynamically and not necessarily as a perfect solution.

Another point that could be improved is the Jenkins library. It currently requires to save the downloaded build-pipeline to disk before execution. A direct evaluation of the code was tested but turned out to be problematic since Jenkins requires the pipeline to be serializable during execution. This is for some reason not given when using direct code evaluation of a closure. A closure was wrapped around the pipeline code to be able to inject the Jenkins pipeline context into the evaluation environment by setting the closure delegate to the libraries current context. This could be improved by implementing the client code as a plug-in. A plug-in should be able to pass the serialization but would make the integration and update routine a lot harder.

The current Jenkins library implementation uses the `load` step to evaluate the retrieved code. This is problematic since the code needs to be wrapped inside a function. Therefore, the implementation of functions inside the build-pipeline templates is not possible and have to be replaced with closures.

7.4.3. Performance Improvements

To improve the performance of the service caching mechanisms could be introduced. This is especially useful for the result of query mapping. Instead of having to parse the query builder object on each request, it could be parsed once and cached for the rest of the execution since there is no possibility of changes. The same goes for the retrieval of Pagen generator information. The process of gathering the meta-data of all discovered generators could be done once and the saved result returned afterwards.

This is, however, not applicable for generation artifacts. The results of the generation is dependent on the templates located in a GitLab repository and might therefore change in between generations [Gitb].

7.4.4. Validation

The current input validation is not as robust as it should be for production. This could lead to error messages getting exposed to the client. Before exposing the service to the outside world validation should be improved to prevent leakage of sensible information.

7.5. Summary

The requirement analysis at the beginning of this chapter proofed that most of the required features are implemented. While the current implementation might not be optimal, like discussed in section 7.4, it provides a good reusable basis and a proof of concept for the introduced problems.

8. Conclusion

Contents

8.1. Summary

The goal of this thesis was to create a web service for an existing code generator. This should eliminate the installation and update process for individuals using the generator and therefore ease the introduction to code generation.

By creating a web application an exemplary client implementation of the service was provided to make it easy to interact with the service.

To evaluate the extensibility of the Pagen generator a software component generator was implemented. It turned out this can be achieved with minimal effort by implementing just a few classes to support the new architecture.

Lastly, the concept of dynamic generation was examined by implementing decomposable build-pipelines for Jenkins.

Chapter 3 introduced some already existing applications offering similar functionality as introduced in this thesis. On the one hand, it was concluded that there were no generator services found that support similar principles than the here introduced service for Pagen. On the other hand, a few build-pipeline principles were presented that allowed similar functionality as required. However, these were either not available for the Jenkins build server or not decomposable.

In chapter 4 some knowledge required to understand the rest of the thesis was introduced. This included the in the frameworks used architecture patterns as well as the later used pipes and filter pattern. Furthermore, a quick overview of the Pagen generator developed by Ralph Geerkens was provided.

The next chapter was used to introduce the services structure and design principles. At the beginning a brief overview of the implemented components was presented, followed by a more detailed introduction to the mentioned components. This included the implementation of the pipes and filters architecture and how the generation processes were mapped to filters.

In the following chapter 6 the concrete implementation of the concepts were described.

The last chapter than provided a requirement analysis to prove that all requirements were met. It, however, also showed some areas of the current implementation that can

be improved.

8.2. Future Work

The web service presented in this work can be improved in different ways, some of them were already mentioned in the thesis.

One idea already mentioned is the implementation of a client application, which utilizes the service's generation capabilities while allowing to modify a project's directory structure directly. This would allow for the same incremental usage as provided by the Maven plug-in implemented by Ralph Geerkens while keeping all the benefits of a service approach.

Another mentioned aspect is the integration of the Jenkins generation into the Pagen generator itself. This would allow to utilize the advanced functionality of Pagen, but would require implementation of problem-specific features, like the ability to edit a files contents apart from placeholders.

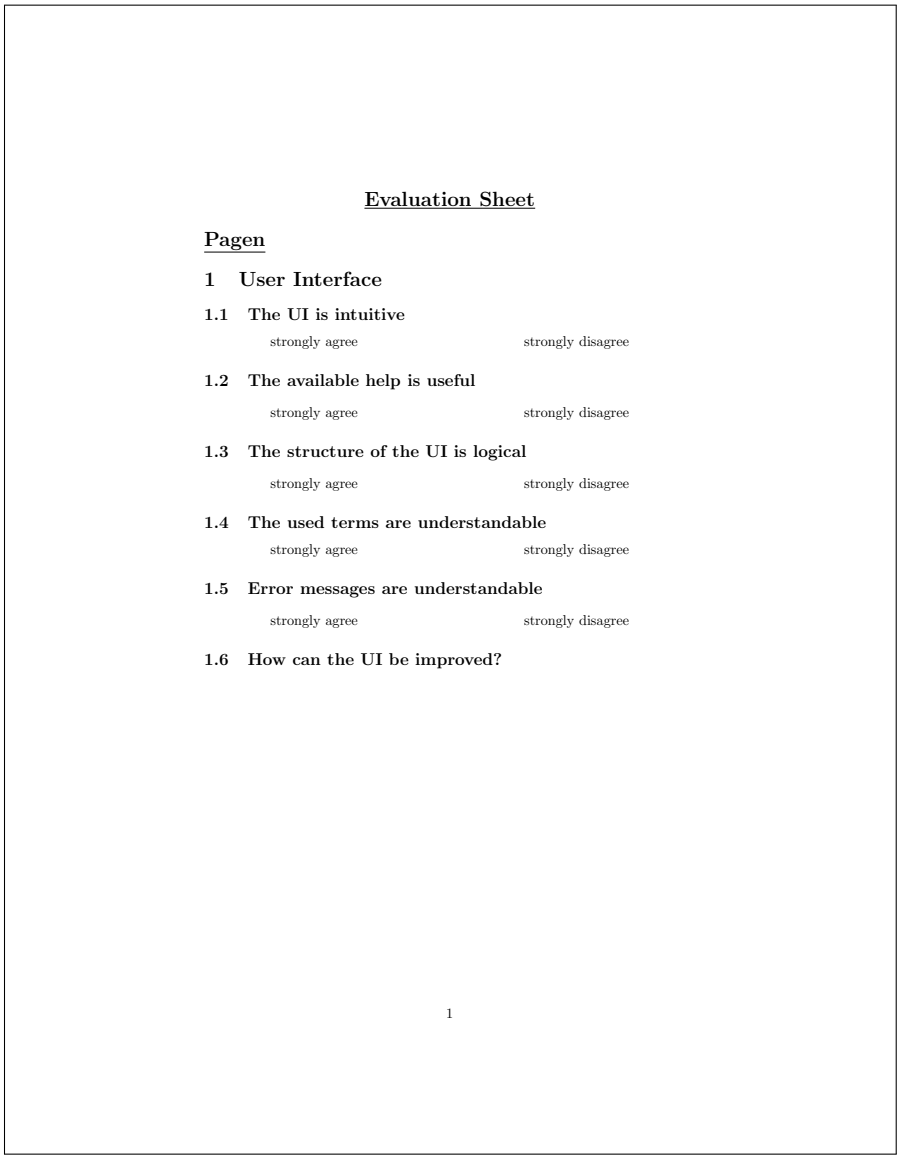
The current implementation of the service only allows the execution of multiple different generators sharing a common parameter set. The execution of the same generator multiple times with different parameters is currently not supported. To add this functionality only a few changes have to be made in the service while the web application would require a makeover.

Adding to this, the service currently does not resolve a composite generator's sub-generators. A user could therefore unknowingly select a generator which is already included in the generation. This does not cause any trouble, but a visual indication of already indirectly selected generators would be a good idea. Furthermore, without the possibility to resolve a composite's sub-generators the service cannot verify if all parameters required by sub-generators are matched. It can, therefore, happen that the generator pauses execution and waits for input, which will not be provided. This is currently not the case since the only composite has all required queries registered in the composite class, but might be an issue in the future. To prevent the execution pause, Pagen could also be extended with a non-interactive mode. This mode would return errors instead of requesting missing parameters.

Input validation is another area of the service that requires improvement. The service is currently not very robust. Faulty input data can cause errors that are returned to the client. To prevent this further validation rules should be added to the service.

Finally, statistics are currently not implemented for the Jenkins components. The implementation is however not necessary if the process gets added to Pagen.

A. Evaluation Sheet



Evaluation Sheet

Pagen

1 User Interface

1.1 The UI is intuitive

strongly agree strongly disagree

1.2 The available help is useful

strongly agree strongly disagree

1.3 The structure of the UI is logical

strongly agree strongly disagree

1.4 The used terms are understandable

strongly agree strongly disagree

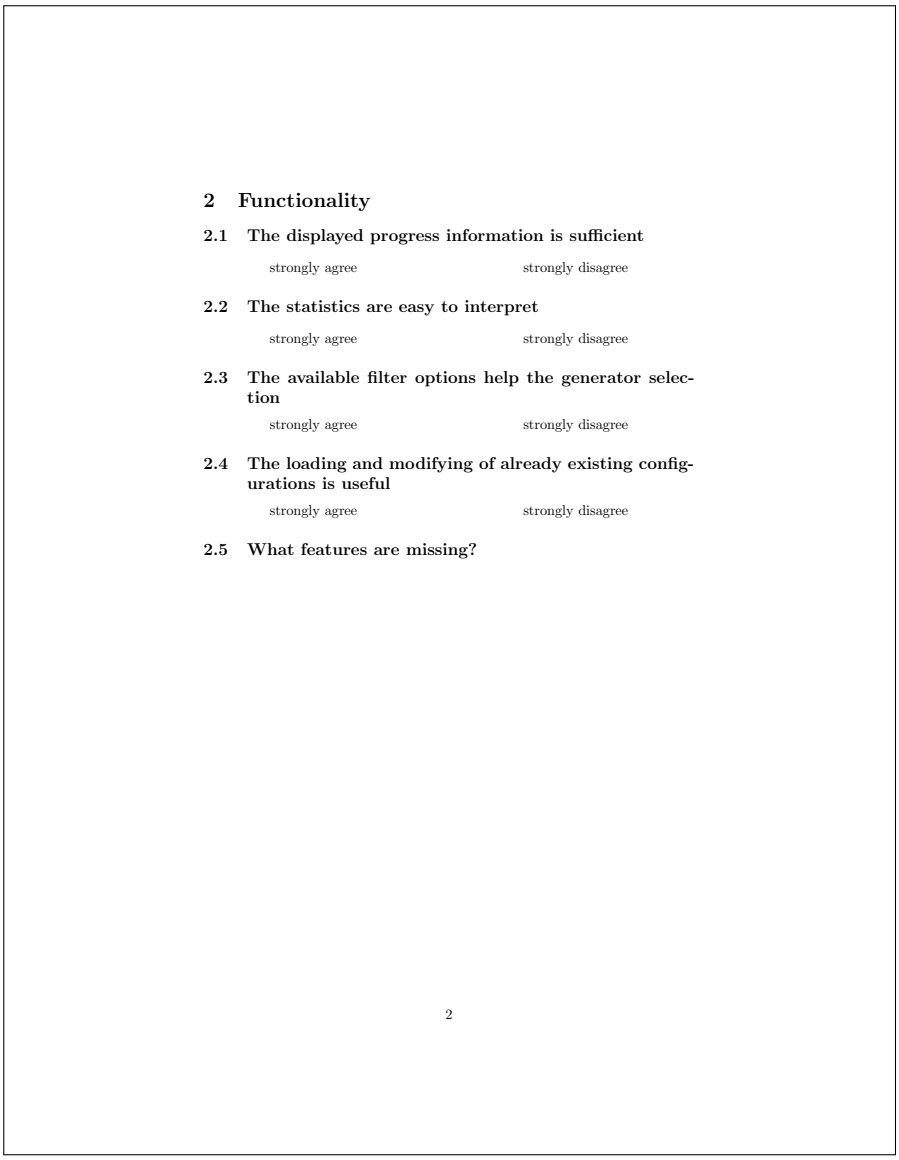
1.5 Error messages are understandable

strongly agree strongly disagree

1.6 How can the UI be improved?

1

Figure A.1.: Evaluation Sheet page 1 of 4



2 Functionality

2.1 The displayed progress information is sufficient

strongly agree strongly disagree

2.2 The statistics are easy to interpret

strongly agree strongly disagree

2.3 The available filter options help the generator selection

strongly agree strongly disagree

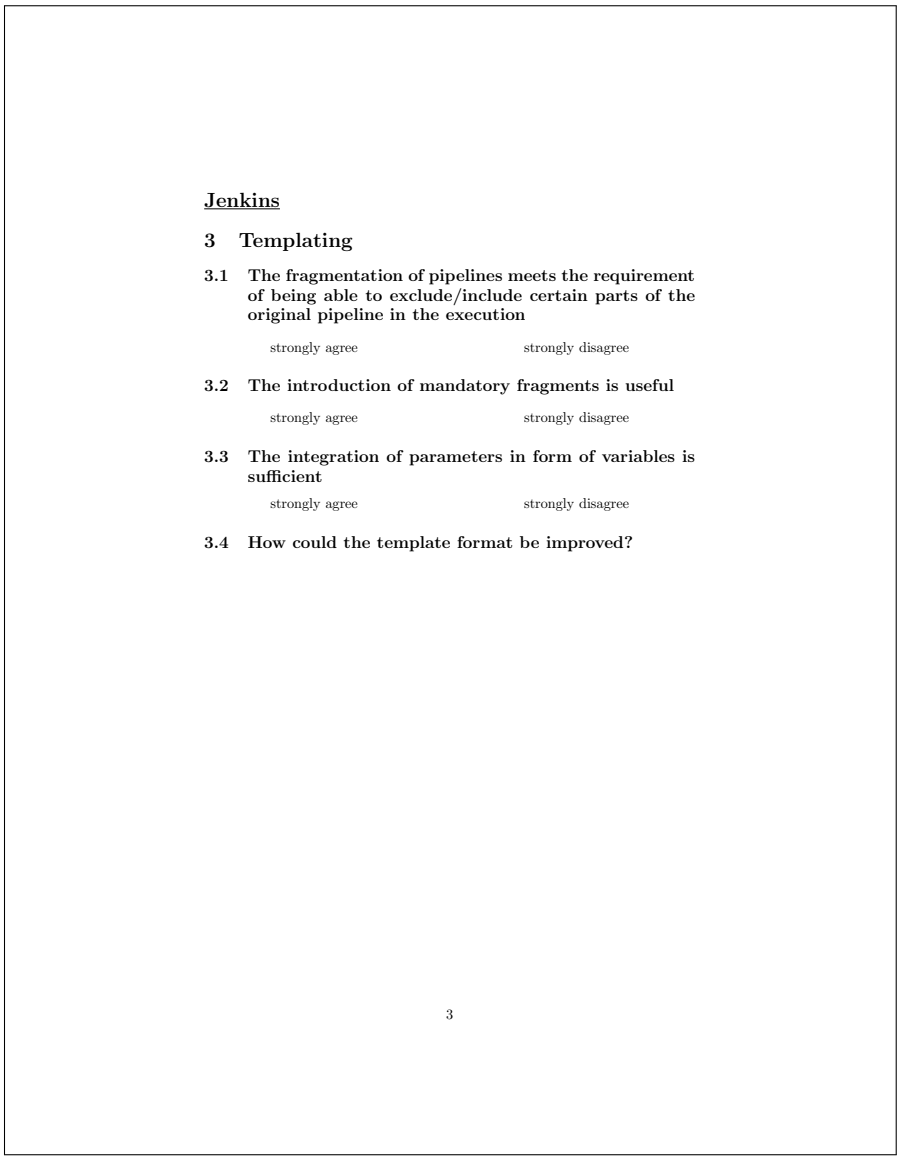
2.4 The loading and modifying of already existing configurations is useful

strongly agree strongly disagree

2.5 What features are missing?

2

Figure A.2.: Evaluation Sheet page 2 of 4



Jenkins

3 Templating

3.1 The fragmentation of pipelines meets the requirement of being able to exclude/include certain parts of the original pipeline in the execution

strongly agree strongly disagree

3.2 The introduction of mandatory fragments is useful

strongly agree strongly disagree

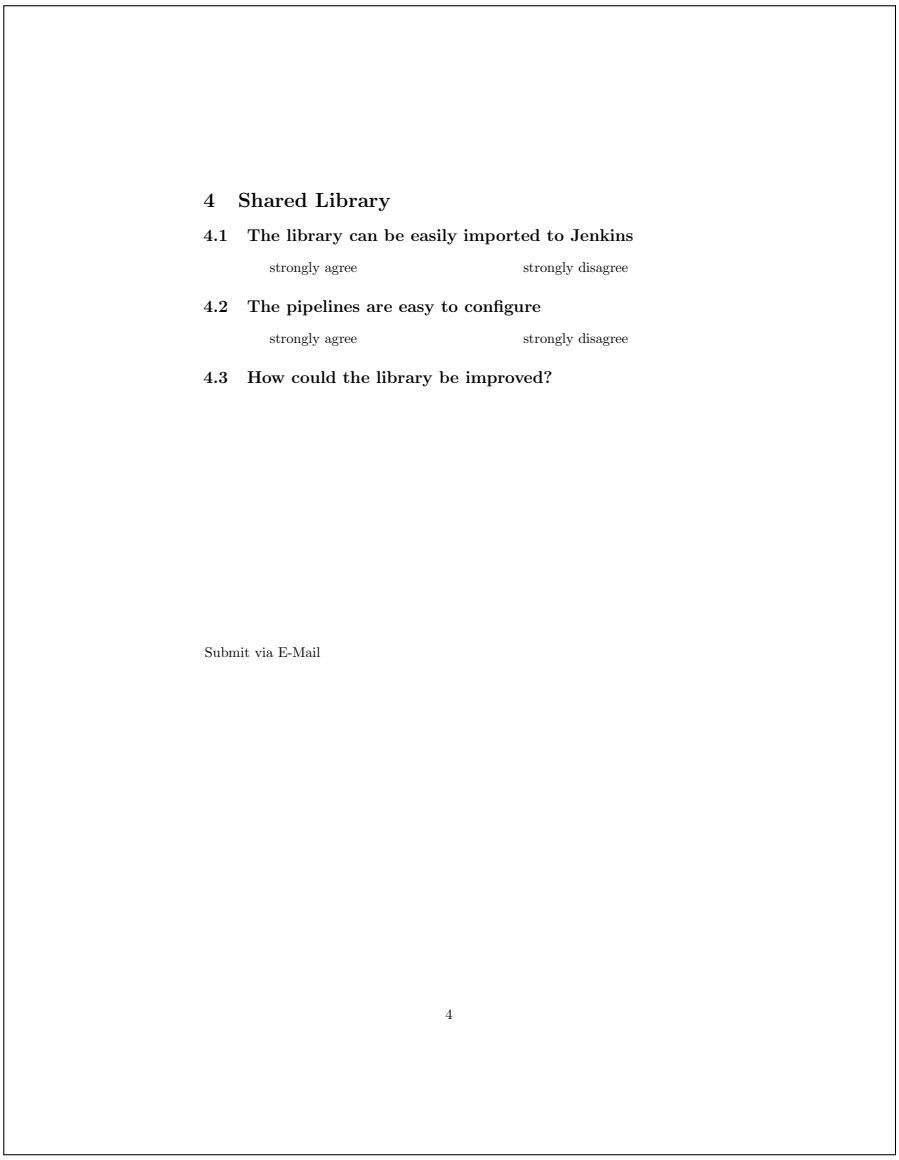
3.3 The integration of parameters in form of variables is sufficient

strongly agree strongly disagree

3.4 How could the template format be improved?

3

Figure A.3.: Evaluation Sheet page 3 of 4



4 Shared Library

4.1 The library can be easily imported to Jenkins

strongly agree strongly disagree

4.2 The pipelines are easy to configure

strongly agree strongly disagree

4.3 How could the library be improved?

Submit via E-Mail

4

Figure A.4.: Evaluation Sheet page 4 of 4

B. Evaluation Results

Evaluation Results		
Question	Rating	Rating
1.1	4	5
1.2	5	5
1.3	5	5
1.4	3	4
1.5	4	5
1.6	Links to used templates	Add examples for area, slice etc.
2.1	4	5
2.2	4	5
2.3	3	5
2.4	5	5
2.5	Encode configuration in URL	Generated file could start with app name
3.1	4	5
3.2	5	5
3.3	3	5
3.4	-	-
4.1	4	5
4.2	4	5
4.3	-	-

Table B.1.: Evaluation Results

C. Web Application Screenshots

Figure C.1.: Execution Log

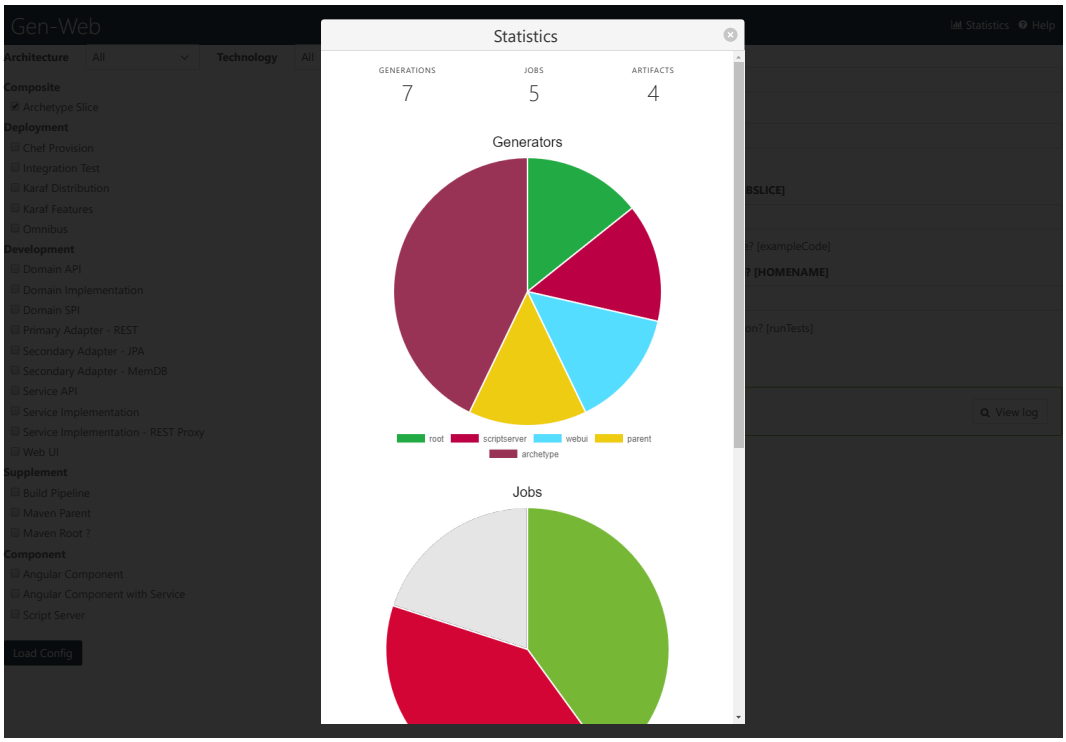


Figure C.2.: Statistics Modal

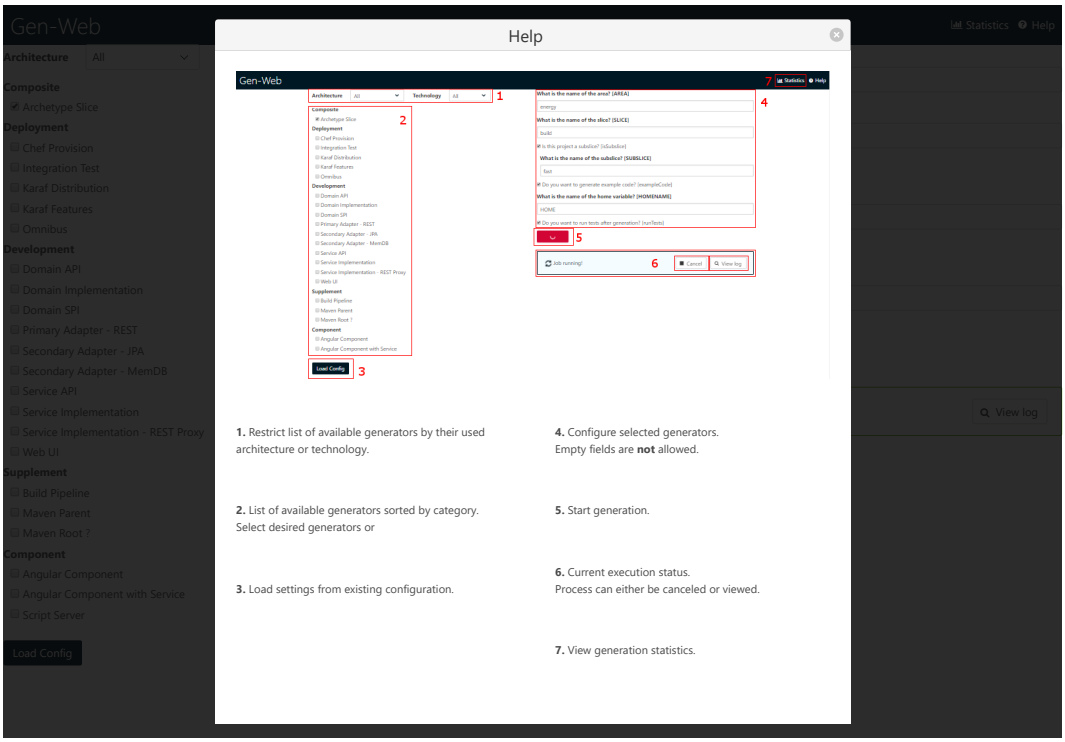


Figure C.3.: Help Modal

D. JSON Examples

```
{
    "name": "Archetype Slice",
    "cmd": "archetype",
    "category": "COMPOSITE",
    "architecture": "PORT_ADAPTER",
    "technology": "JAVA",
    "prompts": {
        "AREA": {
            "type": "String",
            "message": "What is the name of the area?"
        },
        "SLICE": {
            "type": "String",
            "message": "What is the name of the slice?"
        },
        "isSubslice": {
            "type": "Boolean",
            "message": "Is this project a subslice?",
            "additional": {
                "SUBSLICE": {
                    "type": "String",
                    "message": "What is the name of the
                       subslice?"
                }
            }
        },
        "exampleCode": {
            "type": "Boolean",
            "message": "Do you want to generate example code?"
        },
        "HOMENAME": {
            "type": "String",
            "message": "What is the name of the home variable?"
        },
        "runTests": {
            "type": "Boolean",
            "message": "Do you want to run tests after
               generation?"
        }
    }
```

```
39      }
```

Source Code D.1: Generator Information

```
1       {
2           "cmds": ["archetype"],
3           "inputs": {
4               "AREA": "energy",
5               "SLICE": "build",
6               "isSubslice": true,
7               "SUBSLICE": "fast",
8               "exampleCode": true,
9               "HOMENAME": "HOME",
10              "runTests": true
11          }
12      }
```

Source Code D.2: GeneratorConfig

Bibliography

[AS07] I. E. Allen and C. A. Seaman. “Likert scales and data analyses”. In: *Quality progress* 40.7 (2007), p. 64 (cited on page 55).

[BHS07] F. Buschmann, K. Henney, and D. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. John Wiley & Sons, 2007. ISBN: 0470059028, 9780470059029 (cited on pages 11, 12, 22).

[Bui] *BuildKite*. `https://buildkite.com/` (cited on page 8).

[Bul] *Bulma*. `http://bulma.io/` (cited on page 30).

[Fow02] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002 (cited on pages 22, 23).

[Gam+95] E. Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2 (cited on pages 23, 24, 26, 30, 35, 39).

[Gee17] R. Geerkens. *An Incremental Code Generator for Heterogeneous Software and Infrastrucutre*. 2017 (cited on pages 1, 3, 14–16, 33, 50).

[Gita] *Git*. `https://git-scm.com/` (cited on page 8).

[Gitb] *Gitlab*. `https://about.gitlab.com/` (cited on pages 28, 29, 56).

[Goc] *GoCD*. `https://www.gocd.org/` (cited on page 9).

[Gra] *Gradle*. `https://gradle.org/` (cited on page 7).

[Gro] *Groovy*. `http://groovy-lang.org/` (cited on pages 7, 48).

[Jak08] M. Jakl. “Rest representational state transfer”. In: (2008) (cited on page 13).

[Jav] *Java*. `https://www.java.com` (cited on page 7).

[Jen] *Jenkins*. `https://jenkins.io/` (cited on pages 4, 16).

[Kot] *Kotlin*. `https://kotlinlang.org/` (cited on page 7).

[Lom] *Lombok*. `https://projectlombok.org/` (cited on page 53).

[Mav] *Maven* (cited on pages 7, 8, 16, 25).

[Mvv] *The MVVM Pattern*. `https://msdn.microsoft.com/en-us/library/hh848246.aspx` (cited on pages 11, 12).

[Rod08] A. Rodriguez. “Restful web services: The basics”. In: *IBM developerWorks* (2008) (cited on page 13).

[Son] *SonarQube*. `https://www.sonarqube.org/` (cited on pages 49, 53, 54).

[Spra] *Spring Boot*. http://projects.spring.io/spring-boot/ (cited on pages 7, 21, 24, 37, 43).

[Sprb] *Spring Framework*. http://projects.spring.io/spring-framework/ (cited on pages 21, 29).

[Sprc] *Spring Initializr*. https://github.com/spring-io/initializr (cited on page 7).

[Swa] *Swagger*. https://swagger.io/ (cited on page 44).

[Tea] *TeamCity*. https://www.jetbrains.com/teamcity/ (cited on page 8).

[Tem] *vue-webpack-boilerplate*. https://github.com/vuejs-templates/webpack (cited on page 45).

[Vuea] *Vue.js*. https://vuejs.org/ (cited on pages 30, 46, 53).

[Vueb] *Vuex*. https://vuex.vuejs.org (cited on pages 30, 45).

[Weba] *Web Services Architecture*. 2004 (cited on page 13).

[Webb] *webpack*. https://webpack.js.org/ (cited on page 30).

Glossary

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

CD Continous Delivery

CLI command-line user interface

CRUD Create Read Update Delete

CSS Cascading Style Sheets

DOM Document Object Model

DSL Domain Specific Language

DTO Data Transfer Object

HDD Hard Disk Drive

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

IFS Intermediate File Storage

JAR Java Archive

JSON JavaScript Object Notation

JVM Java Virtual Machine

MVC Model View Controller

MVVM Model View ViewModel

POM Project Object Model

REST Representational State Transfer

SCM Source Code Management

SHA Secure Hashing Algorithm

SPA Single-Page Application

UI User Interface

URI Uniform Resource Identifier

URL Unifrom Resource Locator

WAR Web Application Archive

XML Extensible Markup Language

ZIP ZIP