České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Filip Ryšavý**

Studijní program: Otevřená informatika
Obor: Softwarové systémy

Název tématu: **Integrace knihovny AspectFaces do frameworku Angular 2**

Pokyny pro vypracování:

1. Prostudujte použití knihovny AspectFaces.
2. Prostudujte použití frameworku AngularJS 2 pro prezentaci dat.
3. Implementujte 2-3 prototypy aplikací ve framworku Angular 2.
3.1 První aplikace bude založená na ukázkách/prototypech AspectFaces.
3.2 Druhá aplikace bude založená na ukázkových příkladech Angular 2.
3.3 Třetí aplikací bude nasazení na aplikace registračního systému icpc.baylor.edu.
4. Navrhněte standardní API pro komunikaci klientské a serverové části aplikace
5. Porovnejte zhodnoťte přínos použití AspectFaces pro AngularJS2,
6. Porovnejte softwarové kvaility aplikace implementované ve frameworku AngularJS2 bez a s použitím AspectFaces

Seznam odborné literatury:

[1] On separation of platform-independent particles in user interfaces, Tomas Cerny, Michael J. Donahoo, Cluster Computing, 1--14, 2015, Springer US, doi:10.1007/s10586-015-0471-7

[2] On distributed concern delivery in user interface design, T Cerny, M Macik, MJ Donahoo, J Janousek, Computer Science and Information Systems, 2015, Volume 12, Issue 2, 655--681, doi:10.2298/CSIS141202021C

Vedoucí: Ing. Tomáš Černý, MSc.

Platnost zadání: do konce letního semestru 2016/2017

České vysoké učení technické v Praze Fakulta elektrotechnická L.S. -4-

doc. Ing. Filip Železný, Ph.D.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 23. 11. 2015

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering

Bachelor's Project

AspectFaces integration to Angular 2 framework

Filip Ryšavý

Supervisor: Ing. Černý Tomáš, Ph.D.

Study Programme: Open Informatics, Bachelor's degree

Field of Study: Software Systems

May 24, 2016

Aknowledgements

I would like to thank my supervisor Ing. Černý Tomáš, Ph.D. for his guidance and advices. I would also like to thank to all who supported me during studies.

Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

In Prague on May 24, 2016 ....

Abstract

This thesis aims to integrate the functionality of the AspectFaces library to Angular 2 framework. This integration solves complexities from the development perspective caused by the convention user interface design approach. These complexities are usually caused by the information restatements and mostly by mixing various concerns that result in tangling the user interface code. The integration aims to solve it by the aspect-oriented user interface design approach that can more effectively work with separation of concerns and thus result with reduced complexity of development and maintenance of the user interface.
Keywords: user interface; aspect-oriented programming

Abstrakt

Tato práce je zaměřená na integraci funkcionality knihovny AspectFaces do frameworku Angular 2. Tato integrace řeší komplexity vývoje způsobené tradiční přístupem k vývoji uživatelských rozhraní. Tyto komplexity jsou obvykle způsobovaný opakováním informací a hlavně mísení různých zájmů, což vede k těžko čitelnému kódu. Tato integrace si klade za cíl vyřešit tento problém použitím aspektově orientované přístupu k vývoji uživatelských rozhraní, který dokáže lépe pracovat se separací zájmů a tím snížit komplexnost vývoje a údržby uživatelského rozhraní.
Klíčová slova: uživatelské rozhraní, aspektově orientované programování

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Today's applications in a software engineering grow bigger and more complex. And with that comes a size and a complexity of their user interfaces especially with increasing demands on their usability, adaptability and dynamic behaviour [1].

Such demands on the user interface are difficult to achieve using the conventional user interface design approach. As such an approach we should consider the common component-based user interface design approach. This approach creates a lot of duplications in a code [2], which becomes complex, hard to maintain and cannot be used in different contexts [1].

A possible solution to this problem may provide the aspect-orient programming [3] with its applicability to an user interface design. The aspect-oriented user interface design approach results in low efforts when dealing with different contexts, information reuse or separation of concerns.

1.1 Motivation and goal of the thesis

The goal of this thesis is to design and create an implementation of the aspect-oriented user interface design approach for Java EE (Java Enterprise Edition) applications that use the Angular 2 framework in their presentation layers. The goal is to implement functionality of the AspectFaces library to the Angular 2 framework. The main motivation behind this to reduce efforts in developing user interfaces.

The implementation will use the AspecFaces library that provides the aspect-oriented programming engine for Java applications to generate a meta-model with the information about the user interface which will be then transformed into the code that will be rendered in the Angular 2 framework. A usage of the AspectFaces library ensures that demands in usability, adaptability and dynamic behaviour on the user interface will be fulfilled [4] as well as it ensures a presence of AspectFaces library features like supporting automated transformation of data properties to specific widgets and templates.

This topic of the implementation of the AspectFaces library to the Angular 2 framework was already processed in the parer *Aspect-Oriented User Interfaces Design Integration to Angular 2 Framework* [5] which this thesis expands.

1.2 Structure of the thesis

The chapter **Background** provides information about the conventional user interface design approach and the aspect-oriented user interface design approach. It describes and compares them mainly in terms of a re-usability of a code in different contexts and its maintenance. The chapter **Related work** presents some of already existing libraries, frameworks and widgets that are using the aspect-oriented user interface design approach such as AspectFaces with Java Server Faces [4, 6], Metawidget [2, 7], Google Web Toolkit [8], React [9] or the user interface generation by a server using Java Server Faces [10]. In the chapter **Design** the integration of the AspectFaces framework to the Angular 2 framework in designed. The chapter **Implementation** describes the implementation of the integration of the AspectFaces framework to the Angular 2 framework in designed. And finally in the chapter **Tests** there are presented some performance tests of the integration.

Chapter 2

Background

This chapter introduces a technical background behind the thesis. It describes principles of the conventional user interface design approach and the aspect-oriented user interface design approach and compares them mainly in terms of a re-usability of a code in different contexts and its maintenance. In this chapter there are highlighted reasons why the conventional user interface design approach is difficult to use and why the aspect-oriented user interface design approach provides a better solution to the problem.

2.1 Conventional user interface design

The most common way how to create an user interface is to use the conventional user interface design approach. As such an approach we should consider the common component-based user interface design approach. For example in Java SE (Java Standard Edition) we use Swing or JavaFX and in Java EE we use Java Server Faces.

2.1.1 Principle

This approach describes a particular user interface by various collections of components, widgets and other mechanisms [1]. This approach usually focuses only on the user interface part of the application which results in its isolation [1]. This is the reason for the inefficiency when addressing user interface variations or context-aware user interfaces [1]. Application information such as data definitions, data binding, field presentations, user interface layouts, security, input validation rules and other constraints must be defined several times across different layers of the application [1]. The resulting relation between the user interface and other layers of the application must be maintained manually [1]. This leads to mistype errors and inconsistencies [1]. Such a code is complex, hard to maintain and it is virtually impossible to create an user interface that is context related without its duplicate definition [1].

2.2 Aspect-oriented user interface design

Another way how to create an user interface is to use the aspect-oriented programming. We can do that using libraries and frameworks like AspectFaces with Java Server Faces [4, 6], Metawidget [2, 7], Google Web Toolkit [8], React [9] or the user interface generation by a server using Java Server Faces [10].

2.2.1 Aspect-oriented programming

Before the description of the aspect-orient user interface design approach it would be appropriate to describe the aspect-oriented programming. The most common programming approaches are the object-oriented programming and the procedural programming. There are many programming problems that can't be easily solved by using neither of them [3].

The aspect-oriented programming makes possible to express programs involving aspects [3]. An aspect in the aspect-oriented programming is a something that cannot be easily captured in a component [6]. An aspect in intended to capture a particular concern [1]. An aspect captures cross-cutting concerns separately from the components [1]. The goal of the aspect-oriented programming implementation is to be easy to understand and efficient [3].

2.2.2 Principle

Unlike the conventional user interface design approach where the code of individual aspects is mixed all together the aspect-oriented user interface design approach describes each individual concern separately [6]. A concern is a set of information that influences the source code [1] such as data definitions, data binding, field presentation, user interface layouts, security, input validation rules and others. This approach basically consists of a data model creation, a code inspection in order to generate a meta-model, their transformation and an integration of the transformed code into the user interface [1]. This approach produces a lesser amount of a code which is easier to maintain and is re-usable in a context related user interfaces [1]. Though this approach has a lot of benefits it fails on the user interface delivery to the client in terms of preserving concern separation [1].

2.3 Inabilities of conventional user interface design

The situation demonstrating the conventional approach and its inability to capture separate concerns effectively is provides by Figure 2.1 that shows various concerns in a form component. Next, Figure 2.2 shows the way conventional design collapses the individual concerns together. The aspect-oriented user interface design approach preserves the separation of concerns.

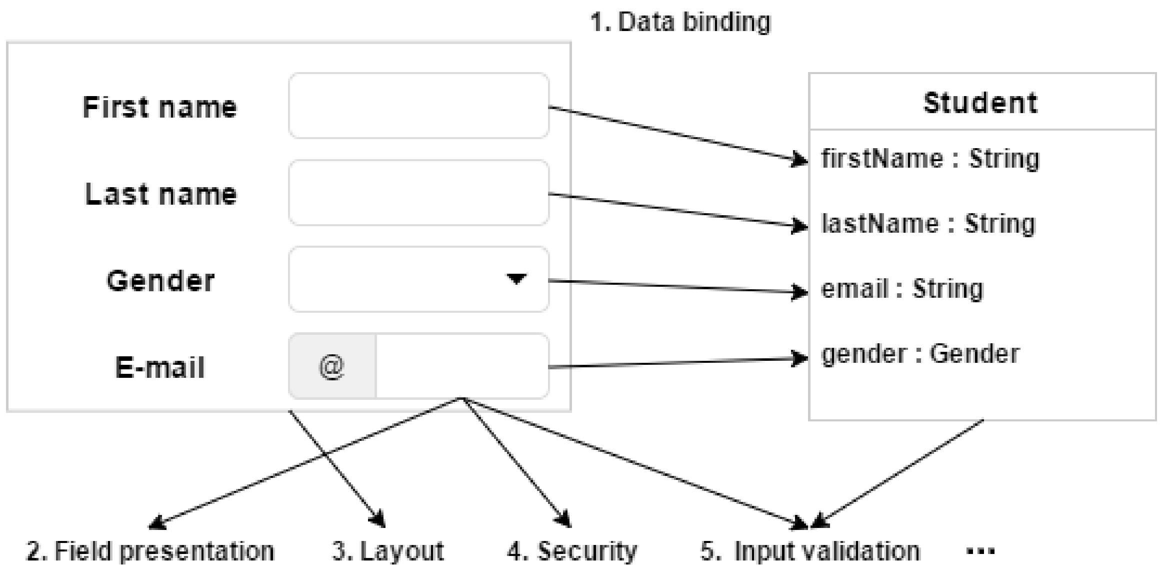


Figure 2.1: Conventional user interface design

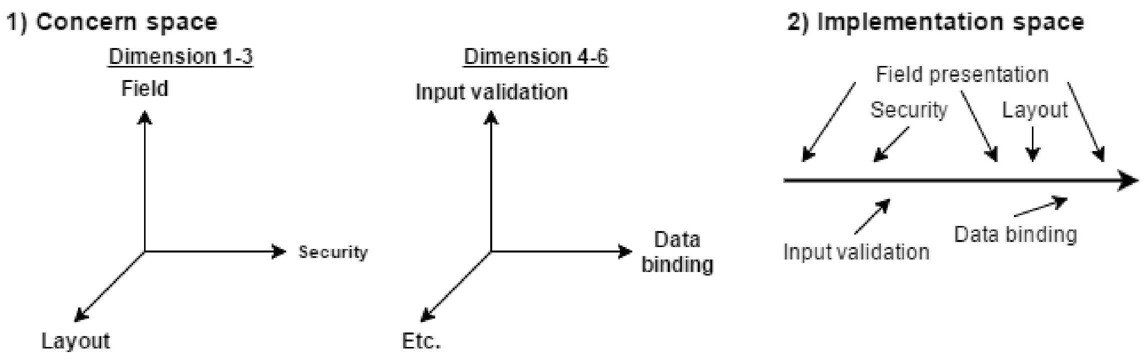


Figure 2.2: Cross-cutting concerns

Chapter 3

Related work

In this chapter already existing libraries, frameworks and widgets that are using the aspect-oriented user interface design approach. Those libraries, frameworks and widgets are AspectFaces with Java Server Faces [4, 6], Metawidget [2, 7], Google Web Toolkit [8], React [9] or the user interface generation by a server using Java Server Faces [10]. Examples of a user interface form are presented for each framework, library or widget.

3.1 AspectFaces with Java Server Faces

One of already existing libraries using the aspect-oriented user interface approach is the AspectFaces library with its support for Java Server Faces [4]. The AspectFaces library provides the aspect-oriented programming engine with its support for annotation descriptors, mapping to custom widgets and templates and other functionality that is utilized by Java Server Faces.

3.1.1 Usage of plain Java Server Faces

First lets examine an example of a user interface written in plain Java Server Faces. In plain Java Server Faces the developer has to make the instance accessible using Java Beans [10]. Then he can use them in the user interface as shown in Listing 3.1.

3.1.2 Usage of AspectFaces with Java Server Faces

The AspectFaces library makes possible to design individual concerns separately and then weaves them all together [4, 6]. To achieve that the developer has to also make the instance accessible using Java Beans. He can use the same beans that have been uses by a plain Java Server Faces user interface. It became a data source that can be used as the source of the information which can be passed to the AspectFaces custom component as shown in Listing 3.2.

The AspectFaces library has three basic stages: an inspection, a transformation and a runtime code integration [4]. During the inspection the information about the data instance are collected from annotations a the context [4]. In other words, a meta-model is

```
First name:
<h:inputText id="firstName" value="#{i.firstName}"/>
Last name:
<h:inputText id="lastName" value="#{i.lastName}"/>
Gender:
<h:selectMenu id="gender" value="#{i.gender}"/>
Email:
<h:inputText id="email" value="#{i.email}"/>
```

Listing 3.1: Example of a form developed manually using Java Server Faces

```
<af:ui instance="#{instance}"/>
```

Listing 3.2: Example of a form developed using AspectFaces with JavaServer Faces

built. The transformations uses the aspect-oriented user interface design approach to apply presentation rules, template composition and layout integration [4]. In the final stage the AspectFaces custom component produces the user interface code that represents the data form and correlates to the context [4]. All this happens at runtime.

3.2 Metawidget

Another solution of that uses the meta-programming and transformations to create the user interface design approach is Metawidget [2, 7]. Its usage is similar to the usage of the ApsectFaces library with Java Server Faces.

3.2.1 Usage

It inspects the entity model dynamically at runtime or statically using similar principles as mentioned above, while not contextual grammar or custom mappings are provided [7]. It can read information from many technologies, for example Annotations, Hibernate, JSON and others [7]. It creates user interface components native to existing front-end platforms such as Android, Google Web Toolkit, HTML5, Java Server Faces, Java Server Pages, Spring, Swing and others [7]. An example of usage the Metawidget can be see in Listing 3.3.

```
<m:metawidget value="#{student}"/>
```

Listing 3.3: Example of a form developed using Metawidget with JavaServer Faces

3.3 Other solutions

The AspectFaces library with its extension for Java Server Faces and Metawidget are not the only options that target separation of concerns. Currently there are available several frameworks, libraries and widgets that partially address such separation. Another examples are Google Web Toolkit [8], React [9] or the user interface generation by server using JavaServer Faces [10]. They all build on top of a data model that is inspected, transformed and the result is integrated into the user interface. A usage of these frameworks, libraries and widgets generally suffers because they are specific in the technology they use so there is no universal solution. Another problem is for example that Google Web Toolkit is practically dead and is about to be replace with frameworks like the Angular 2 framework which target solution of this thesis.

Chapter 4

Design

The already existing solution presented in the previous chapter target specific technology. In this chapter a design a design of the implementation of the aspect-oriented user interface design approach for Java EE applications that use the Angular 2 framework in their presentation layers is described. This integration consists of the server-side meta-model generation, the client-side form rendering and the communication between the server-side and the client-side part of the application.

4.1 Technologies

The integration of the AspectFaces library to the Angular 2 framework will be designed specifically to the functionality of the AspectFaces library and Angular 2 framework.

4.1.1 AspectFaces

The AspectFaces library provides the aspect-oriented programming engine for Java applications will inspect the code and generate meta-properties [4]. The engine also provides means of a type mapping and custom model annotation [4]. By default the AspectFaces library comes with the extension for Java Server Faces that was described in Section 3.1. The integration of the AspectFaces library to the Angular 2 framework will utilize the functionality of the the aspect-oriented programming engine provided by the AspectFaces library.

4.1.2 Angular 2

The Angular 2 framework is currently developed successor of the community popular framework AngularJS. The Angular 2 framework provides functionality such as for example a templating, a HTTP client or a form validation [11]. However the Angular 2 framework is still in beta phase at the time of making this thesis. This will bring a certain limitations during the implementation.

4.2 Meta-model generation

The first thing that needs to be solved is a way how to generate a meta-model from a data model. The meta-model generation will take place on the server-side part of the application and will utilize the aspect-oriented programming engine provided by the AspectFaces library.

4.2.1 AspectFaces meta-model generation

Lets assume that we have a data model prepared and annotated as shown in Listing 4.1. The engine provided by the AspectFaces library contains classes and methods that makes possible to generate the meta-model from a data model and a context [4]. The context is provided by the AspectFaces library and makes possible to get a configuration of a mapping of different properties to custom widgets and templates also provided by the AspectFaces library [4].

```
public class Student {

        @UiOrder(1)
        @UiPlaceholder("Enter your first name")
        private String firstName;

        @UiOrder(2)
        @UiPlaceholder("Enter your last name")
        private String lastName;

        @Email
        @NotNull
        @UiLabel("E-mail")
        @UiOrder(5)
        @UiPlaceholder("Enter your e-mail")
        private String email;

        @NotNull
        @UiOrder(6)
        @UiPassword
        @UiPlaceholder("Enter your password")
        private String password;

        ...

}
```

Listing 4.1: The data model representing a student

4.2.2 AspectFaces meta-model formating

However this meta-model are not in a suitable structure in which they can be easily passed to the Angular 2 framework and processed by it. At this point it is necessary to reformat them into the more suitable structure that could be sent to the Angular 2 Framework. A latter reformatting of the meta-model on the client-side part of the application would only cause performance issues. An example of such structure could be seen in Listing 4.2 The main goal here is to preserve as much of AspectFaces library functionality and features as possible in order to fulfil demands on the user interface usability, adaptability and dynamic behaviour.

```
{
    "name":"cz.cvut.fel.rysavfi1.afwa2.example.model.Student",
    "fields":[
        {
            "name":"firstName",
            "tag":"inputText",
            "constraints":{
                    "label":"First Name"
            },
            "options":null
        },
        ...
    ]
}
```

Listing 4.2: The meta-model generated form a data model

4.2.3 Encapsulation of meta-model generation

The final step of the server-side meta-model generation is to encapsulate its functionality into something that an user would easily use. An ideal case is to encapsulate it into one function.

4.3 Form rendering

The second thing that needs to be solved is a way how to render a form from the generated meta-model in the Angular 2 framework. The form rendering will take place on the client-side part of the application and will utilize Angular 2 functionality.

4.3.1 Meta-model transformation

At the client-side of the application in the Angular 2 framework the meta-model are transformed into the user interface. Specifically the custom component utilizes TypeScript or more precisely JavaScript to populate the content through the separately provided meta-model, cached templates and transformation rules. All the generation performs the client-side part of the application reducing the server-side part of the application involvements and resources.

4.3.2 Entry point

The integration of the AspectFaces library to the Angular 2 framework will provide an user with a custom component similar to the one from a usage of the AspectFaces library with Java Server Faces or a usage of Metawidget. The difference is that the structure and the values of the user interface are separated. This bring multiple options to improve performance regarding to the client-server communication for example by caching the structure of the user interface across an user navigation or a session.

4.4 Communication between server and client

The last thing that needs to be solved is the communication between the server-side part of the application and the client-side part of the application. Both parts of the application are designed to work together using some sort of communication method. Also the server-side part of the application and the client-side part of the application are designed so either of the can be substituted with appropriate alternatives.

4.4.1 Communication using REST

Because the integration is for Java EE application the integration can use benefits of Java EE such as great support or REST (representational state transfer) and JSON (JavaScript Object Notation) processing. Java EE allows easily to send a Java object as JSON through REST. This object can be a user interface structure or values. This means that only things that needs to be done is to create a suitable Java objects to contain a user interface structure and values.

Chapter 5

Implementation

In this chapter an implementation of the designed integration of the AspectFaces library to the Angular 2 framework presented in the previous chapter is described. This chapter contains a description of the implementation of the server-side meta-model generation in Java EE by using the AspectFaces library, the form rendering in the Angular 2 framework and the communication between the server-side and the client-side part of the application by using JSON and REST.

5.1 Meta-model generation

In the implementation of the server-side meta-model generation there are used JDK (Java Development Kit) version 1.8.0_92 and AspectFaces version 1.5.0-SNAPSHOT that were available at the time of making this thesis. The meta-model generation will be implemented in the `Generator` class.

5.1.1 AspectFaces meta-model generation

First thing that needs to be implemented in the `Generator` class is the meta-properties generation. The AspectFaces library provides the aspect-oriented programming engine [4] that makes possible to get the context as shown in Listing 5.1 and the meta-properties as shown in Listing 5.2. This aspect-oriented programming engine needs to be provided by a configuration which will be described in Section 5.1.4 and by a data model whose example was presented in Listing 4.1. A usage of the aspect-oriented programming engine in a list of the AspectFaces meta-properties.

5.1.2 AspectFaces meta-model formatting

In order to prevent performance issues on the client-side part of the application, the list of the meta-properties generated by the ApsectFaces library needs to be formatted into a more suitable structure. Since the meta-properties generated by the ApsectFaces library are in the list, it is simple to iterate through each meta-property and format them in the `Generator` class.

```
Context context = new Context();
Configuration configuration = ConfigurationStorage
        .getInstance()
        .getConfiguration("Angular2");
context.setConfiguration(configuration);
```

Listing 5.1: Getting the context from the AspectFaces library

```
JavaInspector inspector = new JavaInspector(entity);
List<MetaProperty> metaProperties = inspector
        .inspect(context);
UIFragmentComposer composer = new UIFragmentComposer();
metaProperties = composer
        .filterApplicable(metaProperties);
metaProperties = composer
        .getOrderedFields(metaProperties, context);
```

Listing 5.2: Getting the AspectFaces meta-properties

The AspectFaces library generates a lot of meta-properties by default [4]. A lot of them is a duplicate or useless for the designed integration. The useless meta-properties that are generated by the ApsectFaces library are removed. The useless meta-properties generated by the ApsectFaces library are `className`, `ClassName`, `dataType`, `DataType`, `DataTypeFullClassName`, `email`, `entityBean`, `field`, `fieldName`, `FieldName`, `fragment`, `fullClassName`, `FullClassName`, `instance`, `notNull`, `password` and `value`.

The AspectFaces library also generates a lot of meta-properties from data model annotations [4]. These annotation will be described in Section 5.1.6. The integration of the AspectFaces library to the Angular 2 framework makes use of following meta-properties: `label`, `minLength`, `maxLength`, `max`, `min`, `required` and `size`. The integration of the AspectFaces library to the Angular 2 framework also adds a `placeholder` meta-property. These meta-properties are based on attributes of HTML input and select tag.

Then the rest of the AspectFaces meta-properties is formatted into the more suitable structure which contains a name of the entity and a list of fields. The field consists of a name, a tag, a list of constraints and optionally a list of options. A constraint is a pair of a key and a value, an option consists of a label and a value. This representation corresponds with with the structure that was described in Listing 4.2. At the end of this step, the meta-properties are generated and formatted.

5.1.3 Encapsulation of meta-model generation

First, there are created Java classes `Structure`, `Field` and `Option` to represent the meta-model described in Section 5.1.2. These classes are also called DTO (data transfer object).

This representation of the meta-model in Java is visualised in Figure 5.1.

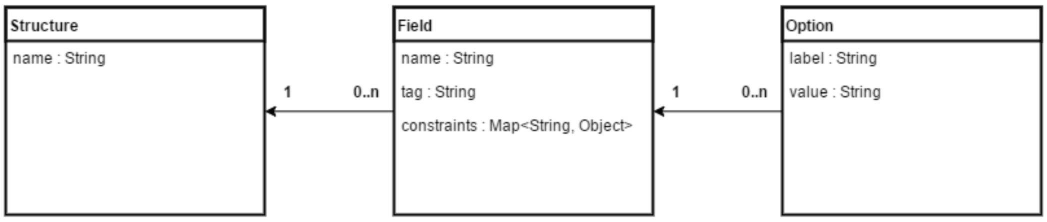


Figure 5.1: Class diagram of the meta-model structure

Secondly, the meta-properties generation and formatting implemented in Section 5.1.1 and 5.1.2 are encapsulated into the `generateStructure` method in the `Generator` class. This method needs to be provided with an entity described in Listing 4.1 and a configuration that will be described in Section 5.1.4. Result can be seen in Listing 5.3.

```
Generator.generateStructure(Student.class, "angular2");
```

Listing 5.3: The generation of meta-properties

5.1.4 Configuration

The AspectFaces library provides the basic configuration package [4]. This integration uses it and modifies it to it needs. A list of configuration files is listed bellow:

- `WEB-INF/web.xml` - This file is called the deployment descriptor. The AspectFaces listener must be registered in this file [4].
- `WEB-INF/apectfaces-config.xml` - The file contains a registration of configurations and a registration of annotations. This file was modified by adding the configuration and the annotations of the integration of the AspectFaces library to the Angular 2 framework.
- `WEB-INF/apectfaces.taglib.xml` - The file contains the configuration of the Java Server Faces extension of the AspectFaces library. Since the integration of the Aspect-Faces library to the Angular 2 framework doesn't use it, the file was removed.
- `WEB-INF/af` and `WEB-INF/af/profile` - The folders contains configurations. There are already presented configuration "html" and "table". They are replaced by "angular2" configuration.
- `aspectfaces.properties` - The file contains a basic configuration of a structure of a project.

The integration of the AspectFaces library to the Angular 2 framework adds its own configuration file to ones from AspectFaces:

- `afwa2-option-labels.properties` - The file was added by the integration of the AspectFaces library to the Angular 2 framework. It contains labels of enumerations crated by a user.

5.1.5 Type Mapping

The AspectFaces library provides means to map data types to different components [4]. The integration of the AspectFaces library to the Angular 2 framework provides several predefined components. They will be described later in Section 5.2.2. A usage of the AspectFaces library makes possible to create and map to user-created custom components. Example of mapping to the integration of the AspectFaces library to the Angular 2 framework components can be seen in Listing 5.4.

The integration of the AspectFaces library to the Angular 2 framework preserves this functionality. Only change is the attribute tag. Instead of containing the file with the tag, the attribute contains the name of the tag now. This is because of the current state of the Angular 2 framework and its capabilities. It is later described alongside the predefined tags in Section 5.2.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://aspectfaces.com/schema/aspectfaces"
    xsi:schemaLocation="...">

    <mapping>
        <type>String</type>
        <default tag="inputText"/>
        <condition
            expression="${not empty email and email == true}"
            tag="inputEmail"/>
    </mapping>

    ...
</configuration>
```

Listing 5.4: Example of the type mapping

5.1.6 Model Annotations

The AspectFaces library provides means to annotate a model by different kinds of annotations [4]. Those annotations can be from AspectFaces, Hibernate, Java Persistence API and others. The AspectFaces library also provides means to annotate a model by user-created custom annotations by creating and registering a descriptor for it [4].

The integration of the AspectFaces library to the Angular 2 framework preserves this functionality. If developer decides to create a custom annotation that generates a custom constraint, he needs to create a descriptor for it first. Then he needs to create a custom tag or component in which the constraint will be used. This creation is describe in Section 5.2.3.

The integration of the AspectFaces library to the Angular 2 framework also creates its own annotation:

- `UiPlaceholder` - This annotation is used to generate the `placeholder` constraint which is used to specify the place holder in the user interface.

5.2 Form rendering

In the implementation of the client-side form rendering there is used Angular 2 version 2.0.0-beta.17 that was available at the time of making this thesis.

5.2.1 Meta-model transformation

The Angular 2 framework has a component based structure [11]. In order to transform the meta model into the user interface, the `FormComponent` component was created. This component handles the user interface rendering. It needs to be provided with the structure generated by the server-side part of the application and by the values also provided by it. This component can also be provided with a boolean value with specifies if the user interface is editable.

5.2.2 Predefined tags

The integration of the AspectFaces library to the Angular 2 framework comes with several predefined tags or components:

- `inputText` - This tag is used for a plain text input. It utilizes HTML input tag with type attribute set to text.
- `inputEmail` - This tag is used for an email input. It utilizes HTML input tag with type attribute set to email.
- `inputPassword` - This tag is used for a password input. It utilizes HTML input tag with type attribute set to password.
- `inputBoolean` - This tag is used for a boolean input. It utilizes HTML input tag with type attribute set to boolean.
- `inputNumber` - This tag is used for a number input. It utilizes HTML input tag with type attribute set to number.
- `selectEnum` - This tag is used for selecting one item from a list of options. It utilizes HTML select tag.

5.2.3 Custom tags

The integration of the AspectFaces library to the Angular 2 framework also supports custom tags created by a developer. However they must be manually added to the `FormComponent` component because the current state of the Angular 2 framework. It is also worth to mentioning that if a developer creates a custom annotation that produces a custom meta-property respectively a custom constraint, he must create a custom tag to use it. The predefined tags don't recognise custom constraint without developer changing their source code.

5.2.4 Entry point

The integration of the AspectFaces library to the Angular 2 framework has an user friendly entry point similar to the one from AspectFaces with Java Server Faces [4] or Metawidget [7]. This entry point comes in form of the selector of the `FormComponent` component as shown in Listing 5.5.

```
<af-form [structure]="struct" [values]="vals" [edit]="edit">
        Loading...
</af-form>
```

Listing 5.5: Form developed using AspectFaces with Angular 2

5.3 Communication between server and client

Both parts of the application were implemented using REST and JSON. This is not mandatory so the alternative mean of communication can be used.

5.3.1 Communication using REST

The implementation is for Java EE application so on the server-side part of the application Java API for RESTful Services (JAX-RS) can be used. On the client-side of the application the Angular 2 framework has its own support for REST services.

5.4 Usage

Both parts the integration of the AspectFaces library to the Angular 2 framework are designed to work together but they can also be used separately. The work title of the distribution is AspectFaces with Angular 2 (AFWA2).

5.4.1 Usage of meta-model generation

The server-side meta-model generation described in Section 5.1 is distributed as a JAR file or through Maven. There is added the configuration package that was described in Section 5.1.4 to it. In case of the Maven project the installation consists of adding Maven repositories and dependencies for both AspectFaces [4] and its integration to the Angular 2 framework. Then the configuration package needs to be merge with the project. Examples of projects can be found on the included CD.

Once the developer finished the installation, he can create a data model which example was shown in Listing 4.1. Then he can use JAX-RS to create rest services in his application, Those service will provides values of the data model and its structure that was shown in Listing 4.2. The structure can be obtained by using the `Generator` class as shown in Listing 5.3

5.4.2 Usage of form rendering

The client-side form rendering described in Section 5.2 id distributed as a JS file respectively s a TS file that needs to be added to the project. The developer needs to add the Angular 2 scripts to his project either by Node.js or CDN [11]. The same must be done with the integration scripts. Examples of projects can be found on the included CD.

Once the developer finished the installation, he can import the `FormComponent` component to his component and use it and its custom tag shown in Listing 5.5 to render a form. This custom tag must be provided with a structure and values than can be obtained from the server-side part of the application using the Angular 2 HTTP module.

Chapter 6

Testing

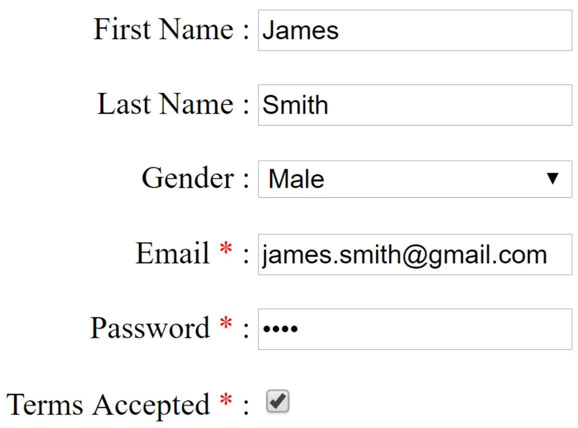
In this chapter there will be presented two example applications of the implementation of the AspectFaces library to the Angular 2 framework. The performance and usability tests will be performed on them.

6.1 Test applications

In order to present and test the implementation of the AspectFaces library to the Angular 2 framework two example applications were created. Those application are programmed in Java EE and Angular 2. They represents information system on which most likely be the implementation of the AspectFaces library to the Angular 2 framework deployed.

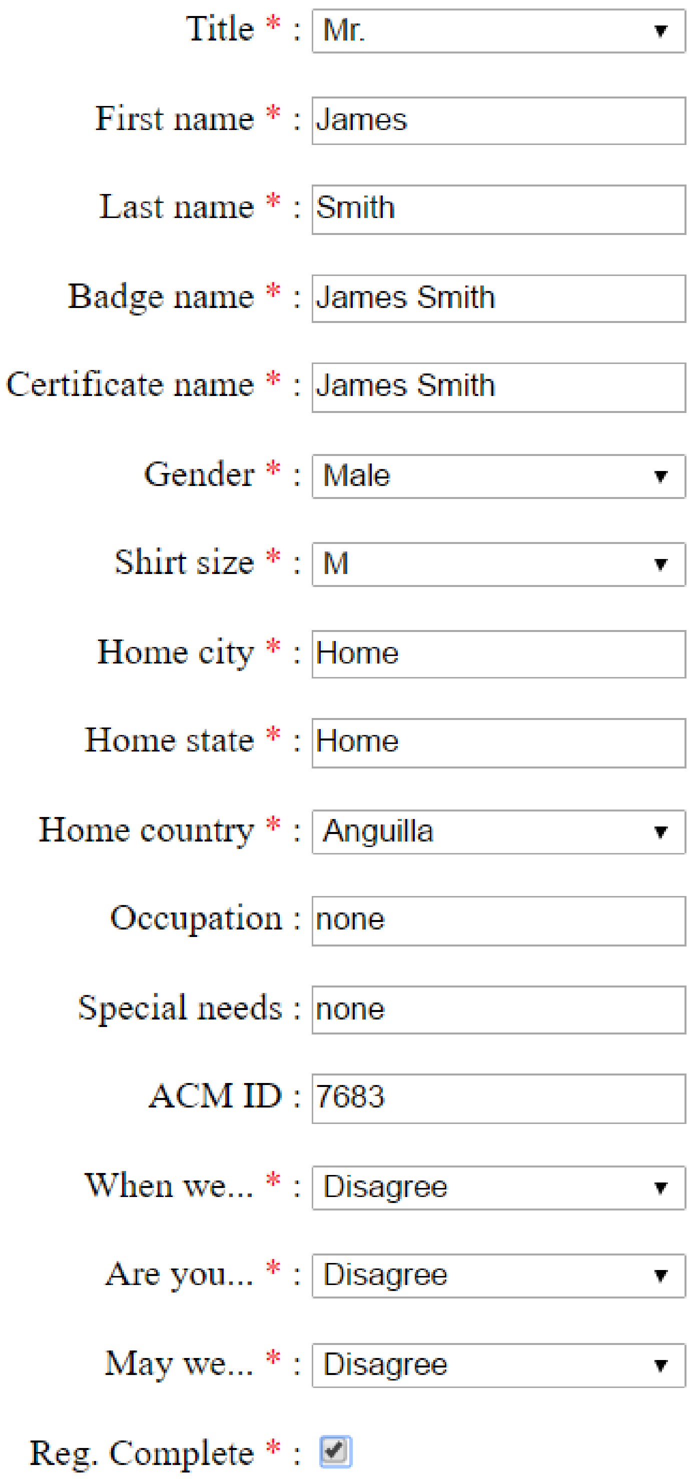
6.1.1 Application 1

This application represent a student registration form. This application was developed with the integration of the AspectFaces library to the Angular 2 framework and its purpose id to present all of its features. Example of a form generated by this application can be seen in Figure 6.1.



First Name :	James
Last Name :	Smith
Gender :	Male ▾
Email * :	james.smith@gmail.com
Password * :	••••
Terms Accepted * :	✓

Figure 6.1: Example application 1 developed using AFWA2



Title * :	Mr. ▼
First name * :	James
Last name * :	Smith
Badge name * :	James Smith
Certificate name * :	James Smith
Gender * :	Male ▼
Shirt size * :	M ▼
Home city * :	Home
Home state * :	Home
Home country * :	Anguilla ▼
Occupation :	none
Special needs :	none
ACM ID :	7683
When we... * :	Disagree ▼
Are you... * :	Disagree ▼
May we... * :	Disagree ▼
Reg. Complete * :	✓

Figure 6.2: Example application 2 developed using AFWA2

6.1.2 Application 2

This application is based on the AF intro application that was provided by Ing. Černý Tomáš, Ph.D. The AF intro application serves as demo for AspectFaces with Java Server Faces. The example with the integration of the AspectFaces library to the Angular 2 framework was added to it in order to compare those technologies.

6.2 Performance testing

In this section the performance testing will be described. It will test that the integration of the AspectFaces library to the Angular 2 framework doesn't slow the application and it will compare its speed with other technologies.

6.2.1 Results

The tested application load time was about 315 ms. However most of this time, about 300 ms, was a loading of Angular 2 scripts. The application itself took about 15 ms to load. For example loading the same application with with AspectFaces and Java Server Faces take about 20 ms and with PrimeFaces about 75 ms. If the fact that the Angular 2 framework is still under development is taken into consideration, there can be concluded that there is no negative performance impact.

6.3 Usability testing

In this section the application's user interface will be examined in terms of amount code and its maintainability. The results of the usability testing were directly taken from the parer *Aspect-Oriented User Interfaces Design Integration to Angular 2 Framework* [5].

6.3.1 Results

Let us examine the prototype's user interface in terms of amount code and its maintainability. First of all, the developer needs to import a custom component that is defined through a JavaScript library. Next, she needs to open a conventional HTML form tag and add there the custom component into it. This reduces the needs to manually design the form. Manual design would: restate information form backend, tangle various user interface concerns together, could expose a typological error not captured by weak type safety, extend development and maintenance efforts, reduce readability and finally loss correlation with the server-side once it evolves or changes the data stricture. The benefits of the automated derivation provided by the prototype are clear, all the before mentioned difficulties are addressed and resulting with a single line of code that provides the expected user interface outcome. Furthermore, optionally it is possible add form controls to influence the outcome. Next, the context information can be take into account so that different users receive different outcomes basing on their specific conditions. Security, us enforced by the AspectFaces library thus not extra information is being exposed to the end user.

To summarize, with our approach the developer only needs to define a particular widgets once and the resulting form correlated to the server-side data structure and becomes adaptive to changes at the server-side, which makes the solution efficient form the perspective of maintenance efforts. Input validations are enforced basing on the server provided meta-model, thus it cannot happen that client submits data that would not be accepted by the server-side because of basic constraint violations, such as length restrictions, textual format, requirements, information types, date restrictions, etc. The prototype thus enforces the structure as expected by the server-side, which reduces the information restatement and the single source of information moves only towards the server side.

Chapter 7

Conclusion

To summarize there was designed and implemented the aspect-oriented user interface approach for Java EE applications with Angular 2 in their presentation layers. The functionality of the AspectFaces library was implemented to the Angular 2 framework.

The implementation fulfils demands on usability, adaptability and dynamic behaviour of the user interface. The implementation doesn't limits a developer in terms of performance and usage. Because the Angular 2 framework is still in a beta phase. Our proposal is evaluated on a prototype application highlighting the benefits or the approach mostly in reduced development and maintenance efforts.

Even though Angular 2 is still in the beta phase we predict similar success as its predecessor Angular 1. So the first thing that needs to be done in the future is to update the implementation with new version of the Angular 2 framework. This will hopefully helps to remove some performance issues caused be the Angular 2 framework.

Other future work may consist of a application of the integration of the AspectFaces library to the Angular 2 framework on large production-level enterprise applications or applying this integration approach on other technologies like React.

Bibliography

[1] CERNY T. – MACIK M. – DONAHOO M. J. – JANOUSEK J. On distributed concern delivery in user interface design. *Computer Science and Information Systems.* 2015, 12(2), 655–681, ISSN 1820-0214, doi: 10.2298/CSIS141202021C. Available from: <`http://www.doiserbia.nb.rs/Article.aspx?ID=1820-02141500021C`>.

[2] KENNARD R. – STEELE R. Application of Software Mining to Automatic User Interface Generation. *New Trends in Software Methodologies, Tools and Techniques - Proceedings of the seventh SoMeT 2008, Frontiers in Artificial Intelligence and Applications.* 2008, 182, 244–254, doi: 10.3233/978-1-58603-916-5-244. Available form: <`http://ebooks.iospress.nl/publication/4851`>.

[3] KICZALES G. – John IRWIN J. – LAMPING J. – LOIGTIER J. – LOPES C. V. – MAEDA C. – MENDHEKAR A Aspect-oriented programming. *In ECOOP'97-Object-Oriented Programming, 11th European Conference* 1997, 1241, 220–242.

[4] ApectFaces documentation. Available form: <`http://wiki.codingcrayons.com/display/af/AspectFaces`>

[5] RYSAVY F. – CERNY T. Aspect-Oriented User Interfaces Design Integration to Angular 2 Framework. Submitted to International Conference on IT Convergence and Security 2016.

[6] CERNY T. – DONAHOO M. J. On separation of platform-independent particles in user interfaces. *Cluster Computing.* 2015, 18(3), 1215–1228, ISSN 1386-7857, doi: 10.1007/s10586-015-0471-7. Available from: <`http://link.springer.com/10.1007/s10586-015-0471-7`>.

[7] Metawidget documentation. Available form: <`http://metawidget.sourceforge.net/documentation.php`>

[8] HANSON R. – TACY A. GWT in Action: Easy Ajax with the Google Web Toolkit. Manning Publications Co., Greenwich, CT, USA, 2007.

[9] B. Speaker-Fisher. Flux: A Unidirectional Data Flow Architecture for React Apps. Applicative 2015, ACM, New York, NY, USA, 2015.

[10] BURNS E. – GRIFFING N. JavaServer Faces 2.0, The Complete Reference. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010.

[11] Angular 2 Documentation. Available form: <`https://angular.io/docs/ts/latest/`>

Appendix A

Nomenclature

AFWA2	AspectFaces with Angular 2
API	Application Programming Interface
CDN	Content Delivery Network
DTO	Data transfer object
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JAR	Java Archive
Java EE	Java Enterprise Edition
Java SE	Java Standard Edition
JAX-RS	Java API for RESTful Services
JDK	Java Development Kit
JS	JavaScript
JSON	JavaScript Object Notation
REST	Representational state transfer
TS	TypeScript
UI	User interface

Appendix B

Content of the included CD

```
|-- example                    The example application 1
|   |-- src                    Source code of the example application 1
|   |-- target                 Build of the example application 1
|   |-- ...
|
|-- example2                   The example application 2
|   |-- server                 Preconfigured server for the example application 2
|   |-- src                    Source code of the example application 2
|   |-- target                 Build of the example application 2
|   |-- ...
|
|-- form-rendering             The client-side form rendering
|   |-- src                    Source code of the client-side form rendering
|
|-- meta-model-generation      The server-side meta-model generation
|   |-- src                    Source code of the client-side form rendering
|   |-- target                 Build of the client-side form rendering
|   |-- ...
|
|-- rysavfi1-thesis-2016.pdf   The pdf version of this thesis
```