Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

ARTUR WASINGER

BACHELOR THESIS

IMPLEMENTATION OF A WEB-UI FOR A GUIDED CONFIGURATION WORK-FLOW

Submitted on 5 November 2018

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 5 November 2018

License

Erlangen, 5 November 2018

Abstract

Open Data has a huge potential, but due to its often poor quality and inconsistency most of this data can't be used, atleast not without a lot of effort. Here the Open Data Service is bringing remedy, because this tool can homogenize such data structures. Still, because the data has such great variety, the complexity of the structure is shifted onto the configuration possibilities, which makes the ODS unwieldy to use. Therefore this thesis presents the requirements of an configuration applications and provides an implementation in form of an Angular based web application, that is used to make the ODS more approachable by giving it the functionality of creating configurations in a better and more comfortable way.

Contents

1 Introduction

Data and data usage are now one of the most valuable resources, easily shown in the fact that the most valuable companies in the world all deal with data (Forbes, 2018). In particular a potential value to the economy has Open Data, where it was measured at 140 billion euros by the Konrad Adenauer StifungCh.H., 2016. Today, one of the main corporations to Open Data is the Open Knowlege Foundation[1]. It leads many other projects regarding Open Data, for example the Open Data Handbook, where they define Open data as follows:

"Open data is data that can be freely used, re-used and redistributed by anyone - subject only, at most, to the requirement to attribute and share alike" (Open Knowledge International, 2018, Chapter: What is Open Data?).

Public administrations also generate a lot of data, where it has not only enormous potential for the economy but also for civilians, politics and the administration. As this kind of data is already provided by many organization like [2]government organizations, or scientific groups, it is understandable that this data is provided in many forms and has no uniform structures. This makes the use very problematic as there is a lot of effort needed to process this data. This problem is one of the main difficulties to the use of Open Data, because it hinders to draw the potential economic value for it. To promote this potential there is a need to support the access and procession of this data. Remedy is given here by an open source project called Open Data Service (ODS). This tool was built to help with the procession of heterogeneous data, in particular Open Data. Here ODS provides a unified interface and can even further enhance data through filters, so that applications can receive already unified data in an easier way. But because data comes in many different forms, the ODS needs to have a big adaptability. This causes the ODS to not be intuitive to configure for an untrained mind.

This thesis presents an implementation of a user interface, that focuses on the configuration part of the ODS and should help to make the ODS more approachable. An implemented web user interface (web-UI) should help to make configuration

[1]https://okfn.de/ Official Open Knowledge Foundation page
[2]https://www.govdata.de/ data portal for german administrative data

process faster and more easy. It could also show people how to use the ODS Application Programming Interface (API) as an example platform. Up to date the configuration of the ODS is very strenuous, as one has many freedoms to how data is handled. one can perform direct requests to the ODS API or one can use the tool Postman[3], which can send predefined API requests. Where it is a great tool to send requests and test these, it is not so great in giving possible option choices to how the data is configured. Another disadvantage of Postman is the missing input validation. Together these points shows, that one needs in-depth knowledge of ODS either to use the direct API or Postman. This is where the acweb-UI gives the user choices and also provides him with the necessary options to create a working configuration set. As a quick overview, chapter 2 is delivering fundamental information, the used framework Angular and the ODS. Chapter 3 covers the requirements and Chapter 4 shows design and implementation choices. In return to chapter 3 the work will be evaluated in chapter 5. And chapter 6 leaves the reader with a summary and conclusion.

[3]https://www.getpostman.com/ Official Postman site

2 Fundamentals

This chapter covers some of the fundamental knowledge that is needed to understand the implemented web-UI. The first part is about some basic information about HTTP and REST, the second explains the ODS and the third is about some basis structures of the used framework Angular.

2.1 HTTP and REST

To clarify some of the terms in this thesis, there is a need to speak about Hypertext Transfer Protocol (HTTP) and Representational State Transfer (REST) [1]. HTTP is a transfer protocol heavily used in the world wide web, as for example already 30% of the current websites use the newest HTTP version HTTP/2 as their protocol (Surveys, 2018). The basic communication of HTTP is based on *HTTP-Requests* and *HTTP-Responses*, where the requests consist of different types, the most relevant for this thesis are: GET, PUT and DELETE. These types represent the action of the request. Where PUT is used to send data with the request, GET will retrieve data from the server and DELETE sends a delete-signal to the server. Together with a HTTP header, which contains a short description of the requested resource and authentication, a HTTP-Request message is send to the server. There the server sends back the HTTP-Response messages where information about that response is send back, like the status of the requested resource. The status code tells the client if the request was successful of if there emerged an error, which is also shown in the code number. REST is an architectural style described by Roy Fielding, to design loosely coupled applications over HTTP. It is mostly used to define a web service API and considers several constraints to be called a REST API. One calls it *RESTful* if all of these are met and *RESTlike* if they follow the REST constraints but don't support them all. Roy Fielding described the constraints of REST in his book Fielding, 2000, Chapter 5. The most important one for this thesis is the *Uniform Inter-*

[1] For an in-depth description to HTTP and REST look at https://www.crummy.com/writing/RESTful-Web-Services/html/

face constraint. Uniform Interface means that the sever provides an interface that logical to the representation of its resources to URLs. Fielding described this constraint in Fielding, 2000, Chapter 5.1.5. Other constraints are already universally applied, such as *Client-Server* constraint, that requires the division of of the client layer to the server layer, that are a standard in web applications.

2.2 Open Data Service

The ODS[2] is a software that is used to process *Open Data*. This data is often in heterogeneous forms and needs a lot of effort to change it to more usable data forms. Here ODS extracts such data from external sources, filters it and provides this then filtered and uniform data in its API. This helps to make this data more usable and accessible by external applications. ODS is based on the master thesis of Konstantin Tysin (Tysin, 2014), who implemented and prototyped the basic functionality of the system. Also there will be references to Mathias Zinnen's bachelor thesis (Zinnen, 2018), who described the current version of ODS on his way to implement his thesis work. In the following chapters I will describe *ODS*.

2.2.1 Architecture of the ODS system

To understand how to configure data for procession, one first has to look at the different parts of the ODS system. Here figure 2.1 from Tysin's master thesis shows clearly what main components ODS is based on. It can be divided into three parts:

- An import component, which extracts data from external sources.
- A database component, that persists data and other metadata for the import component.
- A server component, that enables communication over a REST API.

As figure 2.1 shows just a rough architecture, Zinnen showed a more detailed and more current version in figure 2.2. Here one can see that the ODS architecture is based on the structure of Tysin, but has evolved. It is not strictly divided and became more complex. For example is the API, here the so called REST-API, necessary to configure filter processors and the external datasources. For the next sections follows a more detailed explanation on how they work together.

[2]https://github.com/jvalue/open-data-service Official Github repository

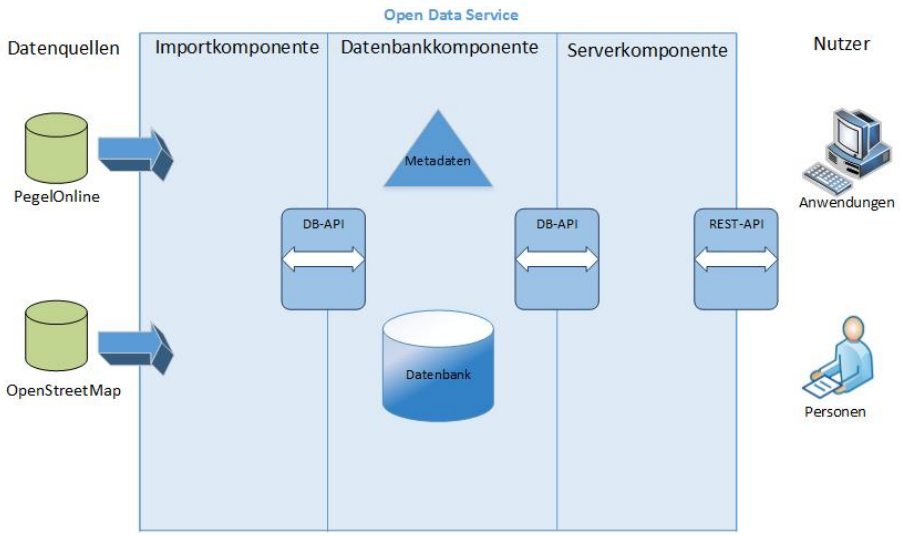

Figure 2.1: Basic ODS architecture Tysin, 2014

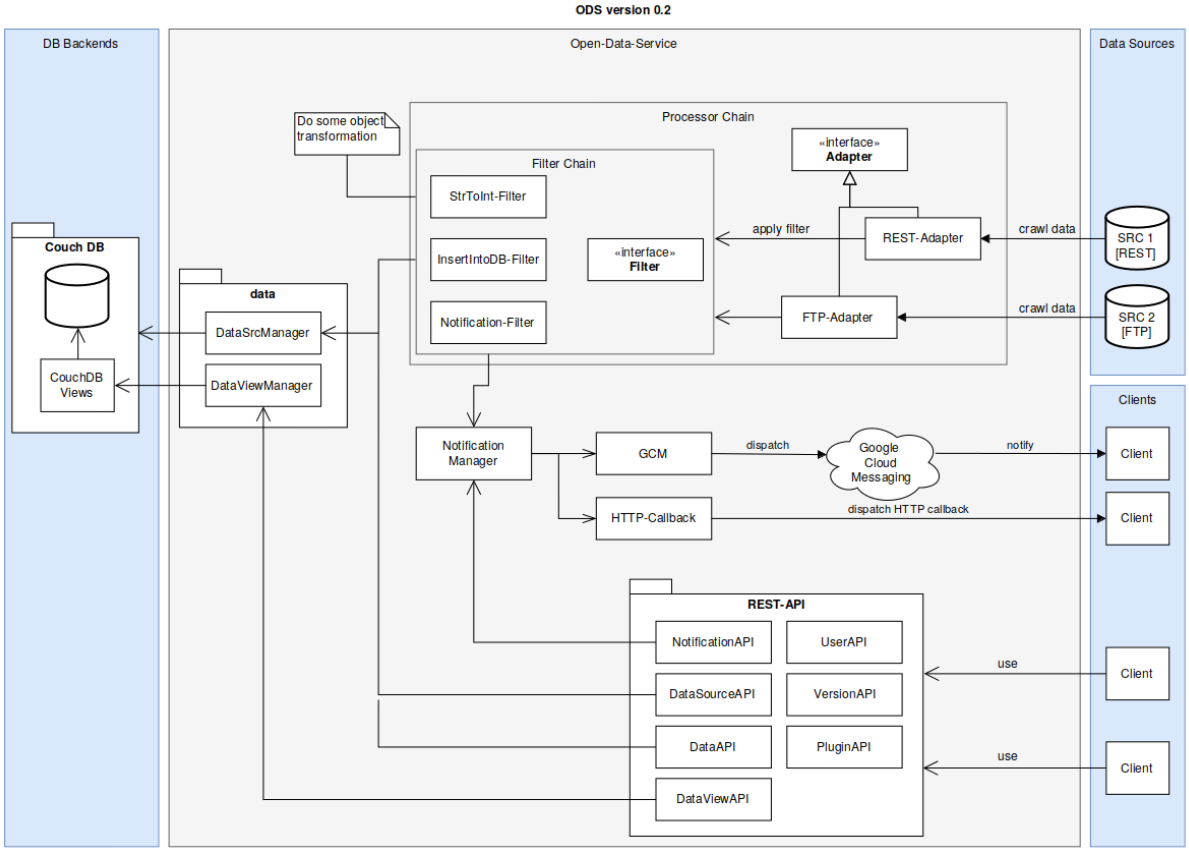

Figure 2.2: Detailed ODS architecture Zinnen, 2018

Name	Arguments
PegelBrandenburg	sourceUrl:java.lang.String
CsvSourceAdapter	sourceUrl:java.lang.String, csvFormat:java.lang.String
TransformationFilter	transformationFunction:java.lang.String
PegelBrandenburgMerger	
APIXUSourceAdapter	apiKey:java.lang.String, locations:java.util.ArrayList
JsonSourceAdapter	sourceUrl:java.lang.String
IntToStringKeyFilter	
InvalidDocumentFilter	
AddTimestampFilter	
XmlSourceAdapter	sourceUrl:java.lang.String
PegelOnlineMerger	
PegelPortalMvSourceAdapter	sourceUrl:java.lang.String
OsmSourceAdapter	sourceUrl:lava.lang.String
OpenWeatherMapSourceAdapter	apiKey:java.lang.String, locations:java.util.ArrayList
DbInsertionFilter	updateData:boolean
NotificationFilter	

Table 2.1: ODS processor list

Import Component

The Import Component's task is the data extraction from external sources, but to do so one first needs to follow some steps. The first one is to create the base *data source* resource that represents the metadata and schema of the external source. Then to import data, so called processors are needed, they represent the adapter and filter that are provided by the ODS. These processors are linked in a *processor chain* resource. This chain is based on the "pipes & filters"- architecture pattern, where the sequence of the processing steps is defined. The first step in a chain is always an adapter. This is a special processor that enables to connect to the target source API, for example an XMLAdapter that enables to crawl from API's that provide their data in XML. Table 2.1 shows the adapter and filter of the ODS, provided by ODS API. To further process, the data is then passed on through a chain of filters, where it will finally be persisted in the database if one selects the filter *DbInsertionFilter*. Additional processor chains can be added to target data sources via the API in run-time. But if one wants to add new processors they need to add a template in the source code of the ODS itself.

Database Component

The database component persists data that comes from the external source and of course the metadata of the configurations and existing model descriptions. The component is based on a NoSQL database. This is needed because the Data the

ODS is going to persist is not in a fixed schema.

The benefits of persisting data from external sources are its possibility to be able to consistently provide data, without being dependent on the targets source availability or to even provide historical data. For heterogeneous data, there is no good possibility to map every data schema to a relational database schema, regarding the flexibility of the database schema. Also there is the problem with scalability as firm data base schemes in relational database systems do not support elastic scalability [3]. The database component is dependant of the server component because through its API, it receives the information of the metadata and can also provide its stored data.

2.2.2 API & its Models

As the server component provides a REST API, it offers the question of what the API is used for. In this chapter there will be an overview of the current API and its core functions, by the example of its models. Models represent the resources that can be persisted by the database. As the current API is designed as a REST API Tysin, 2014, cf. p. 26, there is a constant textitroot endpoint in form of the URL *"/ods/v1/"*. One can see in Figures 2.3 and 2.4 what URL endpoints exist in the current API version. For example to receive a list of all *users* one has to send a HTTP request in form of "GET /ods/v1/users". This list is received in form of a JSON object as seen in Figure 2.5. The API represents the ODS models so for reference on can look at the different API endpoints. The models needed for a configuration are as follows:

- Data Source: For one metadata that a user can define is saved in this model. For the other there is the source schema, which defines the extracted data in form of a JSON schema[4]. This model is the key model in the ODS. As one can see in the API reference all models that are required to extract data are referenced from this model, for example the Processor Chain model, where its Base-URL is "/datasources/sourceId/filterChains". Here one can see that a Data Source is needed to create Processor Chains from the API.
- Processor Chain: A processor is either an adapter or a filter, where the Processor Chain is described as a chain, where the extracted data is followed through each processor-link. Adapters are needed to extract the data from the target source, in its arguments is defined where the target data is found. Filters are mainly there to manipulate the data, for example one can add

[3]https://www.mongodb.com/scale/nosql-databases-pros-and-cons Pro's and Con's of NoSql databases

[4]https://json-schema.org/ JSON schema is a standard to describe data formats

a timpestamp with the *AddTimestampFilter* or one can insert the filtered data into the database with the *DbInsertionFilter*

- Notification Client: This is where one can set up notification clients for a Data Source. Notification clients can currently notificate via Google Cloud Messaging (GCM)[5], Advanced Message Queuing Protocol (AMQP)[6] and HTTP. Note that one needs to insert a *NotificationFilter* in a Processor Chain that the clients are able to notificate.
- Data View: In relational databases it's possible to define views via queries. To be able to work with the extracted data in NoSQL databases, that are used with the ODS, on can also define views. Here these views are written in JavaScript in a style called MapReduce[7], where map- and reduce-functions are used to generate these views as they are very great with flexible data ("CouchDB The Definitive Guide", n.d.).
- Processor Specification: This is where the specifications and descriptions of the different processors are laying. In Table 2.1 one can see the data that is received from this API endpoint. Regardless if this model is not important in the data extraction workflow it is important to get a basic understanding of the possibilities that a configuration gives.

2.3 Angular

Today Angular is one of the most popular framework choices for designing large enterprise and business applications. Its currently developed and maintained by Google and even though its very complex it has a well-formed and consistent documentation. To understand the implemented web-UI, it is necessary to understand some basic fundamentals of the Angular framework. The following descriptions are based on the "Angular Documentation", 2018.

TypeScript

Angular uses TypeScript as its programming language. Typescript is a type oriented language, that is based on the ECMAScript 6 (ES6) standard, which is a javascript standard. In essence its a superset of ES6 and enables users to have ES6 features and strongly static type checking ("Typescript Documentation", 2018).

[5]https://developers.google.com/cloud-messaging/ Official GCM page
[6]https://www.amqp.org/ Official AMQP page
[7]Further information can be found at Ghemawat, 2004

Data	Base-URL: /datasources/{sourceId}	
Verb	URL	Description
GET	/data	Get data objects from {sourceId}
GET	/pointer:/.*	Get certain attributes of a data object
DELETE	/data	Delete all data objects from {sourceId}
DataSources	Base-URL: /datasources/	
Verb	URL	Description
GET	/	Get all data sources
GET	/{id}	Get a data source by its {id}
GET	/{id}/schema	Get the data schema for a data source
PUT	/{id}	Add a data source with a specified {id}
DELETE	/{id}	Delete a datasource
DataView	Base-URL: /datasources/sourceId/views	
Verb	URL	Description
GET	/	Get all data views
GET	/{id}	Get a data view by its {id}
PUT	/{id}	Add a data view with a specified {id}
DELETE	/{id}	Delete a datasource
Notifications	Base-URL: /datasources/sourceId/notification	
Verb	URL	Description
GET	/	Get all clients registered for notifications
GET	/{id}	Get a client by its {id}
PUT	/{id}	Register a client with a specified {id}
DELETE	/{id}	Unregister a client
Plugins	Base-URL: /datasources/sourceId/plugins	
Verb	URL	Description
GET	/	Get all plugins
GET	/{id}	Get a plugin by its {id}
PUT	/{id}	Add a plugin with a specified {id}
DELETE	/{id}	Delete a plugin
ProcessorChain	Base-URL: /datasources/sourceId/filterChains	
Verb	URL	Description
GET	/	Get all processor chains
GET	/{id}	Get a processor chain by its {id}
PUT	/{id}	Add a processor chain with a specified {id}
DELETE	/{id}	Delete a processor chain

Figure 2.3: ODS endpoint list I Zinnen, 2018

ProcessorSpecification	Base-URL: /filterTypes	
Verb	URL	Description
GET	/	Get all specifications
Users	Base-URL: /users	
Verb	URL	Description
GET	/	Get all users
GET	/{id}	Get a user by its {id}
GET	/me	Get currently logged in user
PUT	/{id}	Create new user with a specified {id}
DELETE	/{id}	Delete a user
Version	Base-URL: /version	
Verb	URL	Description
GET	/	Get version information

Figure 2.4: ODS endpoint list II Zinnen, 2018


```
[
    {
        "id": "698b423c-3e0a-43de-950b-abb2ff1f79ff",
        "name": "Admin",
        "email": "admin@adminland.com",
        "role": "ADMIN"
    },
    {
        "id": "d59dfebb-0c36-496a-90e8-0c8a5a0b8764",
        "name": "public123",
        "email": "bob@mail.com",
        "role": "PUBLIC"
    },
    {
        "id": "e2848aa7-723b-4cd8-b19d-c30ee9409b00",
        "name": "tester",
        "email": "test@test.com",
        "role": "ADMIN"
    }
]
```


Figure 2.5: ODS example "GET /ods/v1/users" response

2.3.1 Architecture of Angular

For the start, lets look at the basic architecture of an Angular app. Every application consists of so called `NgModules`, these are the basic building blocks for every Angular app. Each `NgModule` represents a set of related code and thus provides a compilation context for its components within. Some of the most common `NgModules` are for example the BrowserModule or the RoutingModule[8]. Every application has at least one root module, conventionally called `AppModule`. This module is the bootstrapping point of the app, meaning that this is where the app launches. Other modules are called feature modules. A class is considered an `NgModule` if it's decorated with `@NgModel()`. The `@NgModel()` decorator describes the `NgModule` properties with its meta data. Some of those properties are:

- `declarations`: Where all Module components are listed.
- `imports`: That lists all other Modules that are needed.
- `providers`: This is where services are set, so they can be used in the whole app. Note that it is possible and prefered to provide these in components or services.
- `bootstrap`: A `NgModule` should just have one bootstrap component, as it represents the root module.

Within the `NgModule` are two basic code sets: components and services. Components build the viewing part of the application, where they define views as sets of screen elements. Instead services provide functionality to components that is not related to views, like fetching data from an API. In Angular services can be made to injectable services with the `@Injectable`-decorator. Here can be declared where the service is provided from. The code line `@Injectable({providedIn: 'root'})` shows how the service is provided in the root module, like mentioned further up. That causes the injectable services to be available for all components, which makes them very reusable. For the constructor example `constructor(private service: HttpClient) {}` one can see how the service `HttpClient` service class is injected into another class. This design pattern is called Dependency Injection (DI). DI is a common design pattern in Angular where the class asks from external sources for their dependencies, instead of creating them itself[9]. This makes the code cleaner as one doesn't have to add dependencies.

[8] https://angular.io/guide/frequent-ngmodules Other common `NgModules`
[9] https://angular.io/guide/dependency-injection More information on DI

```
// declare a publishing operation
new Observable((observer) => { subscriber_fn });
// initiate execution
observable.subscribe(() => {
      // observer handles notifications
    });
```

Figure 2.6: short Observable example from "Angular Documentation", 2018

2.3.2 Observables

`Observables` are very common in Angular, as they provide support for messages between a publishers and subscribers. That makes them useful within services, as for example Angular's `HttpClient` returns `Observables`. These `Observables` can deliver multiple values of any type. For them to deliver their data, `Observables` first have to be `subscribed` to, where subscribing is a function that executes an `Observable`. One can see in Figure 2.6 how a publishing operation is defined in an `Observable` and then executed via `subscribe()`.

2.3.3 Material Angular

The User Interface (UI) component library Material Angular[10] got to be mentioned as this is the source of the UI modules used in the configuration UI. Here there exists a large variation of different UI components like buttons, sliders and tables, that can be imported into the root module.

2.3.4 Alternatives to Angular

There are several alternatives to Angular as a framework to build web services and frontend applications. The most popular of them are React[11] and Vue.js[12]. In comparison to Angular React is the current biggest rival. The main difference between them is that React is developed by an Open Source community whereas Angular is developed by Google. Both have their strengths as for instance React is easier to learn or has many libraries that are developed by a big

[10] https://material.angular.io/
[11] https://reactjs.org/
[12] https://vuejs.org/

open source community and Angular provides a uniform documentation or uses TypeScript(Team, 2018).

3 Requirements

The concept and design of a guided configuration workflow for a web-UI is followed by its requirements. To fully understand what is needed it's first necessary to clearly understand what functionality should be met.

3.1 Functional Requirements

The definition of a configuration is given by a set of resources that are needed to extract data from target sources. The requirement is met if it's possible to configure all resources that are needed for a working configuration. That means its necessary to be able to:

- create new configurations
- read configurations
- and delete configurations

These are similar to the convention of CRUD (create, read, update, delete), whereas the update-functionality is not implemented by design of the ODS server component API. A reconfiguration can be achieved here by creating a new configuration and removing the older version.

3.2 Nonfunctional Requirements

3.2.1 Guided Workflow

"The sequence of industrial, administrative, or other processes through which a piece of work passes from initiation to completion." - *workflow* definition (Oxford-Dictionary, 2018)

As mentioned in the section title, the workflow should also be *guided*. That means that the input of the user is *directed* to a working configuration set. This can be achieved with different methods, like constraints or help instructions.

3.2.2 Usefulness

The term useful can be defined as the additive from the terms usability and utility. For short usability is the attribute that assesses how easy interfaces can be used and utility is the attribute that measures if the interface provides all functionality that the user needs. Together they measure if a software, in this case the configuration client, is useful. To determine the usefulness, one can look at the quality attributes usability and utility. The utility attribute is in general more easy to define, because it describes the functionality. This part is already covered by the chapter 3.1. Describing usability is more subtile, like in Jakob Nielsen's Usability 101 (Nielsen, 2012) one can define the key quality components as follows:

- Learnability: Which describes how easy tasks are accomplished for the first time.
- Efficiency: Shows how quickly users can perform tasks, when they learned the design.
- Memorability: This is a measurement on how easy one can reestablish proficiency, when the design is not used for a period of time.
- Errors: This quality describes the misbehaviour of users in terms of how many errors do users make, how severe can these errors be and how easily can they recover from these.
- Satisfaction: Here, the question is asked, on how pleasant the use of the design was.

To delimit the usability radius more, one can first look at the actual user group. For instance, the guided configuration workflow can't easily be operated by users that don't know about the ODS system. This is simply proofed by the fact that somebody can't deliberately target a design purpose, if he doesn't know what purpose the design follows. Therefore the first restriction to the user group is the question, if the user is able to follow the design's intentions. One can say that, if the usability and utility quality components within the required user group is met, then this requirement is fulfilled. It's not fulfilled if there are drastic deficits in the quality attributes, that make the application unusable.

4 Design and Implementation

4.1 Design

This chapter covers the basic design choices of the web user interface (web-UI), moreover it should give an overview of the functionality of the workflow and what its intentions are.

4.1.1 Conception

To design a web-UI the first question is what tools are necessary. For this we decided to use the Angular framework for the ODS webclient[1]. As it is a state of the art framework for building web applications and gives a lot of pre build functionality. For example gives the tool *Angular CLI*[2] a very easy interface to generate new components, services and provides other functionality.

Like in Angular fashion the web application is mainly build with two basic building blocks: components and services. In the ODS webclient there are additionally the models classes that purpose as a bridge between component and service classes. Here is a quick overview of the main blocks and their uses:

1. Components generate the views. These views are the reason that something is showing in the browser. They can interact with services. For the main part this is where data is shown and can be inserted.
2. Services provide functionality. In case of the configuration workflow, this functionality is mainly focused on the use of the ODS API.
3. Models are the object classes in the webclient, they make sure data is type safe and represent the different model classes from the ODS API. These classes represent a bridge between the services and components.

[1] https://github.com/jvalue/ods-webclient
[2] https://github.com/angular/angular-cli/wiki

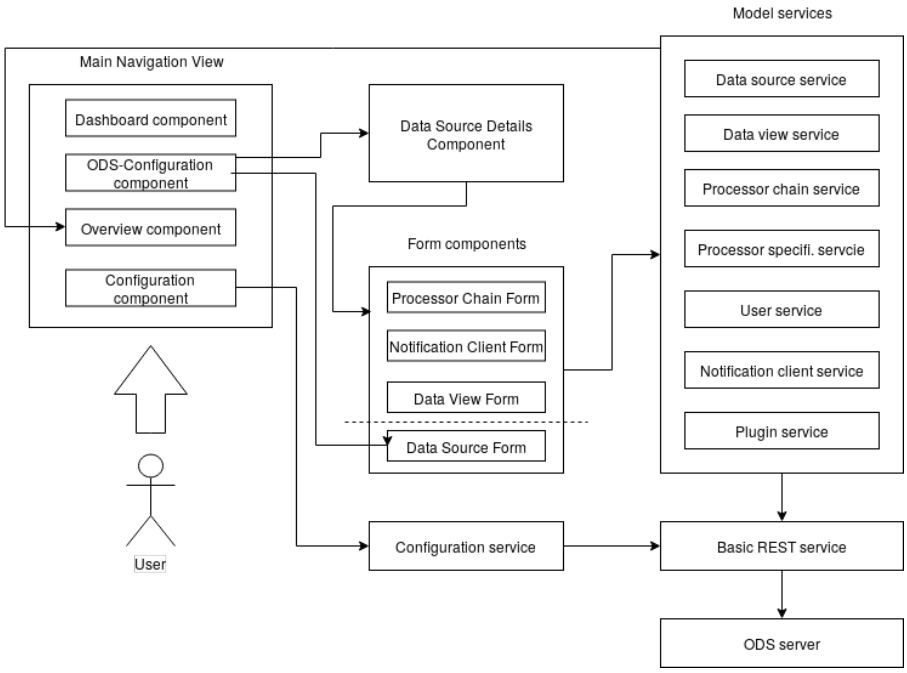

Figure 4.1: Design overview of the ODS webclient

In figure 4.1 one can see how the division between the building blocks is designed, as it shows a clear distinction between components in the main view and services on the other side.

4.1.2 Main View components

The first entry point of the web application is a navigation interface. This is the main dividing structure of the app, to do so this main component leads to other component views that are included into the main navigation:

- The Dashboard component, that acts as a welcome page, gives links, basic and other useful information.
- The ODS-Configuration component, where one can find an overview of existing data sources and manage them. Section 4.1.4 gives a more detailed description.
- The Overview component, that gives an overview of model resources, that are not bound by a Data Source URL, for example from the ODS User API, while Processor Chains are dependant on a Data Source model.
- The Configuration component, where you can modify the webclient configuration.

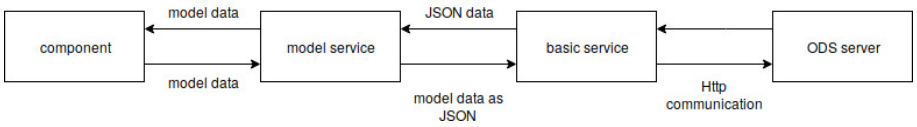

Figure 4.2: Design of the service pipe chain

These components are basically the main components, they can include components, too. These can give them other views based on the logic they need. For instance should the new ODS-configuration component give the possibility to add new data sources. This ADD-logic is then given in form of a new component.

4.1.3 Services

Communication with server structures always includes the use of a client, that can send and receive data. In this case Angular already gives a good solution in form of a injectable http client service, which is an extension of the hard to use XMLHttpRequest API. Here the design choice was to have one *basic service* that sends and receives data from a http client. And to have *model services*, that link between the basic service and the components. This service chain is further illustrated in the Figure 4.2. The model classes can either be send from component site, where the model is converted into a form that the server can use, or received from the *basic service*, where the server data is converted to a model data.

4.1.4 ODS-Configuration component

The ODS Configuration component is the heart of the ODS webclient. The reason one wants to use the ODS webclient is to configure the ODS in an easier way. To design a solution one first has to look at the ODS API. The API is subject under the REST architecture, and that means it is under the subject criteria of a *Uniform Interface*. This criterion is the reason, that the base resource of a configuration is set as a *Data Source*. Because as all other resources relevant for a configuration refer to its *sourceId* in their URL[3]. That means that a *Data Source* is the start for a configuration. In figure 4.1 on can see how the ODS-Configuration component uses the *Data Source Form component*, where new Data Source models are sent to the ODS API. And there is also a link to the *Data Source Details component*, where details of a Data Source can be viewed. Also one can here add or delete further configuration relevant resources. The basic idea was to first create a Data Source as a base block, where further resources

[3]Look at figure 2.3 and 2.4

like Processor Chains can be added. The creation of resources is always done by a form component. Here the web client can receive input from the user, so that the resource can be finally sent as a request to the ODS model services.

4.2 Implementation Details

The next parts will give more insight into noteworthy details of the source code and the problems that were dealt with.

4.2.1 Requesting Data from the ODS API

In figure 4.2 one can see how the fundamental communication with the ODS server to a component view is realized. The first link in this chain is the `BasicRestService` class. In figure 4.4 one can see the different API methods. Here the http client `http` builds the API endpoint `strings` and adds `HTTPHeaders` as `requestOptions`. The client is injected into the `BasicRestService` and enables to work with a `XMLHTTPRequest` API [4]. Contradictory to the name one can receive data not determined to be in XML form, as the definition of the data form is bound in the request header. One can also see that the request is followed by a `.pipe`-method, that pipes the response through a `catchError`-method, which handles the response in error case. The next layer are the model services, as shown in figure 4.3 one can see that the class methods represent the API endpoint of the ODS[5]. Here the Base-URL is given to the injected `BasicRestService`. Furthermore the model service methods return an `Observable<T>` with `T` being the specific model class of the service. This makes sure that the response data has the requested model class form as they are here type-checked. Note that even though the request is in theory complete, when build from a model service, it is *not* executed till the returned `Observable` is subscribed to. In figure 4.5 a Data Source is subscribed to and just now the data can be extracted and processed, as no notification is send to the Observable pre subscribtion. In this case one can see how the received data values are mapped to table elements that are then shown in a component view.

4.2.2 Sending Data to the ODS API with Reactive Forms

Angular has a wide offer to how input data is processed. Where the most basic one is the `<input>` html-tag. Here one can define input-tag-attributes like type,

[4] https://xhr.spec.whatwg.org/ the XMLHttpRequest Standard documentation
[5] compare to figure 2.3 and 2.4

```
  get(url: string): Observable<any> {
    return this.http.get( url: this.endpointUrl + url, this.requestOptions)
      .pipe(catchError(this.handleError));
  }

  post(url: string, data: any): Observable<any> {
    return this.http.post( url: this.endpointUrl + url, data, this.requestOptions)
      .pipe(catchError(this.handleError));
  }

  put(url: string, data: any): Observable<any> {
    return this.http.put( url: this.endpointUrl + url, data, this.requestOptions)
      .pipe(catchError(this.handleError));
  }

  delete(url: string) {
    return this.http.delete( url: this.endpointUrl + url, this.requestOptions)
      .pipe(catchError(this.handleError));
  }
```

Figure 4.3: Basic rest service HTTP methods

```
@Injectable({
  providedIn: 'root',
})
export class DataSourceService {

  dataSourceUrl = '/datasources';

  constructor(private service: BasicRestService) {}

  getDataSource(): Observable<DataSource[]> {
    return this.service.get(this.dataSourceUrl);
  }

  getDataSourceById(sourceId: String): Observable<DataSource> {
    return this.service.get(this.dataSourceUrl + '/' + sourceId);
  }

  getDataSourceSchemaById(sourceId: String): Observable<DataSource> {
    return this.service.get(this.dataSourceUrl + '/' + sourceId + '/schema');
  }

  addDataSource(body: DataSource) {
    const sourceId = body.id;
    const data = {
      'domainIdKey' : body.domainIdKey,
      'metaData' : body.metaData,
      'schema': body.schema
    };
    return this.service.put( url: this.dataSourceUrl + '/' + sourceId, JSON.stringify(data));
  }

  deleteDataSource(sourceId: String) {
    return this.service.delete( url: this.dataSourceUrl + '/' + sourceId);
  }

}
```

Figure 4.4: Data source service

```
this.dataSourceService.getDataSource().subscribe( next: value => {
  this.dataSourcesDataSource = new MatTableDataSource(value);
});
```

Figure 4.5: Data source subscription and mapping to MatTableDataSource

```
createMetaDataForm()  {
  return this._formBuilder.group( controlsConfig: {
    id: ['', Validators.required],
    domainIdKey: [''],
    metaData: this._formBuilder.group( controlsConfig: {
      name: [''],
      title: [''],
      author: [''],
      authorEmail: [''],
      notes: [''],
      url: [''],
      termsOfUse: ['']
    })
  });
}
```

Figure 4.6: Data source form initialization

the initial value and the *required*-attribute. With this simple tag one can create a lot of input forms, but if there are more complex requirements like cross-field-validation theses basic input forms are quickly nearing their limits. For this Angular gives a solution in form of the *R*eactive Forms. One of its main benefits are a model-driven approach to handling input that can change over time, for instance dynamic form requirements. Figure 4.5 shows how a form is build with a `FormBuilder` module. This module enables faster form creation. When the form is complete and valid, the form input is first transformed into a model object and then it is subscribed to. Following the model service, shown in figure 4.4, one can turn the model class to a JSON string with `JSON.stringify()` in the `addDataSource` method. Here it shows, that the Data Source model is separated into the *sourceId* and the rest, whereas the *sourceId* is outsourced. This is needed because on the one hand the *GET requests* deliver a *sourceId* in their JSON objects and on the other hand *PUT requests* need the JSON object without the *id*. That leads to a conflict, where much conversions are needed because of this. By having just one model class, one could convert directly to a JSON object.

Nested Reactive Forms

As the example with the Data Source should show the basic handling, how forms are send to the ODS server. Now comes a more complicated example of the reactive form used in the ODS webclient. This example is about the adapter form that is part of a Processor Chain. The Figure 4.7. shows the html-template of the adapter selection. This adapter is shown in a drop-down list, where each adapter is listed in the `adapterArray` and then shown as options. This array contains all adapter-specifications filtered from the Processor Specification API.

```
<h3>Configure the Adapter</h3>
<form *ngIf="adapterArray" [formGroup]="adapterFormGroup">

  <div>
    <div>
      <label>Adapter</label>
    </div>
    <div>
      <mat-form-field style="...">
        <mat-select placeholder="Please select an adapter" (valueChange)="updateAdapterSettings($event)"
                    formControlName="name">
          <mat-option *ngFor="let adapter of adapterArray" [value]="adapter.name">
            {{adapter.name}}
          </mat-option>
        </mat-select>
      </mat-form-field>
    </div>
```

Figure 4.7: Data source form initialization

```
<div *ngIf="currentAdapter" formArrayName="arguments">
  <div *ngFor="let argument of adapterConfigOptions; let i=index">
    <div [formGroupName]="i">

      <span style="...">{{argument}}</span>
      <mat-form-field style="...">
        <input matInput placeholder="Value" formControlName="{{argument}}"/>
      </mat-form-field>

      <span *ngIf="currentAdapter.argumentTypes[argument] + ''
                   == 'java.lang.String'">Please enter a String</span>
      <span *ngIf="currentAdapter.argumentTypes[argument] + ''
                   == 'java.util.ArrayList'">Please enter json as value for ArrayLists</span>
    </div>
  </div>
</div>
```

Figure 4.8: Data source form initialization

```
updateAdapterSettings(adapterName: string) {

  let adapter = null;
  for (const adap of this.adapterArray) {
    if (adap['name'] === adapterName) {
      adapter = adap;
    }
  }
  if (isNullOrUndefined(adapter)) {
    throw Error( message: 'Failed to locate the adapter in the adapterArray.');
  }

  this.currentAdapter = adapter;
  this.adapterConfigOptions = Object.keys(adapter.argumentTypes);

  while (this.adapterConfigFormArray.length > 0) {
    this.adapterConfigFormArray.removeAt( index: 0);
  }

  for (const type of this.adapterConfigOptions) {
    const formOptions: any = {};
    formOptions[type] = [''];
    this.adapterConfigFormArray.push(this._formBuilder.group(formOptions));
  }

}
```

Figure 4.9: Data source form initialization

On a `(valueChange)` in the `mat-select`-tab the method depicted in Figure 4.9 is executed. Here one can see that the adapter is first searched until found and then set as the `currentAdapter`. As the Processor Specification API give us not only the name and type of the different processors, but also the arguments, we can also dynamically show these arguments in the form fields for the Processor Chain form. Figure 4.9 shows how the arguments are taken as `Object.keys`, where one key represents the key of a JSON object key/value pair. To show these keys in the form template one first has to flush the old keys and then insert new form groups to the `adapterConfigFormArray`. Shown in Figure 4.8 is the template to show the argument of the selected adapter. There one can see that the adapter number can be variable as the `[formGroupName]` directive is declared as the current index of the `adapterConfigOptions` array. A problem with this approach is that the arguments can not be validated dynamically, as the only information the ODS API gives is the Java class type, which in turn is not enough information to how the required data should look like.

5 Evaluation

This Chapter will evaluate the requirements defined in Chapter 3. These will be compared to the implementation described in Chapter 4.

5.1 Configuration

There exist four different kind of resources that are needed for a configuration: Data Sources, Processor Chains, Data Views and Notification Clients. All these resources can be added in form components, and read or deleted in the ODS-Configuration component or in the Data Source Details component. Therefore this requirement is met.

5.2 Guided workflow

Currently the configuration is not in a strict workflow, that means that there is not one single workflow that leads to a ODS configuration. The workflow is divided into many single ones, because for example one first has to create a Data Source resource before it is possible to create a Processor Chain resource. That means there is not one single guided workflow process that directly leads to a configuration. The different workflows are divided onto the form components, that enable to process user input. These workflows are in theirselves guided, as for example the Notification Client Form enables the user to choose between the different Client types and guides a user to choose one of them. On the whole one can say that this requirement is partially met.

5.3 Usefulness

The requirement states five different quality components that are mostly subjected to individual interpretation, the evaluation will then build on the tendencies to determine if the implemented application is useful. Here its presupposed that a user has some in-depth knowledge of the ODS.

- Learnability: A user should be able to accomplish the tasks very easily.
- Efficieny: Tasks can be performed more quickly, than with previous methods[1].
- Memorability: As the design is kept very simple, proficiency is easily reestablished.
- Errors: For now this quality is not fully met. For one the user can misbehave in the Form components, for example is it not possible to fully dynamically validate processor arguments before sending them via the API, as there is currently too few information from the ODS API, and to hard code this information would lead to unwanted dependencies to newly implemented processor templates. For the other the user can't input resources that can cause severe errors as the ODS API is not accepting such data. .
- Satisfaction: The web-UI is more pleasant to use, because one can create configurations faster than with other methods.

Summa summarum the evaluation shows that most points are on the side of being met, so the requirement is met.

[1] directly via the API or via Postman, mentioned in the Introduction Chapter

6 Summary & Conclusion

The key target of this thesis consisted of the design and implementation of a web-UI to create configuration for the Open Data Service[1]. The first subject regarded the fundamentals, where among others the basic structure of the ODS was described (2.2) and the framework Angular was introduced (2.3). To concept and design the web-UI, it was needed to define the requirements (3). For one there was a definition of functional requirements (3.1) that based around the configuration creation and also nonfunctional requirements that should explain what a guided workflow requires (3.2.1) or what usefulness means in the context of a requirement (3.2.2). Chapter 4 depicted which design was developed and what its components were (4.1), while the following chapter cared about the implementation details (4.2). In the last part the web-UI implementation was compared against the requirements and evaluated (5). In conclusion the ODS webclient has met the requirement in the most part as there can always be improvements for a UI that can be discussed on. Possible improvements are for example more advanced validations of the ODS-Configurations. The goal is that the ODS webclient could evolve into a very useful tool whithin the jvalue project line, especially for users that are newer to the ODS.

[1] https://github.com/Keldami/ods-webclient GitHub Repository of the ODS webclient

7 Abbreviations

API	Application Programming Interface
ODS	Open Data Service
REST	Representational State Transfer
web-UI	web user interface
HTTP	Hypertext Transfer Protocol
DI	Dependency Injection
ES6	ECMAScript 6
UI	User Interface

References

Angular Documentation. (2018). Retrieved from https://angular.io/docs

Ch.H. (2016). Potenzial von open data. Retrieved from https://www.kas.de/statische-inhalte-detail/-/content/potenziale-von-open-data

CouchDB The Definitive Guide. (n.d.). Retrieved from http://guide.couchdb.org/draft/tour.html#mapreduce

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm Accessed at 3.11.2018.

Forbes. (2018). The world's largest public companies. Retrieved from https://www.forbes.com/global2000/list/#header:marketValue_sortreverse:true

Ghemawat, J. D. S̃. (2004). Mapreduce: Simplified data processing on large clusters. Retrieved from https://ai.google/research/pubs/pub62

Nielsen, J. (2012). Usability 101: Introduction to usability. Retrieved from https://www.nngroup.com/articles/usability-101-introduction-to-usability/

Open Knowledge International. (2018). The open data handbook. Retrieved from %5Curl%7Bhttp://opendatahandbook.org/guide/en%7D

Oxford-Dictionary. (2018). Definition of workflow. Retrieved from https://en.oxforddictionaries.com/definition/workflow

Surveys, W. T. (2018). Usage of http/2 for websites. Retrieved from https://w3techs.com/technologies/details/ce-http2/all/all

Team, K. (2018). Angular 6 versus react 16.3. Retrieved from https://kruschecompany.com/blog/post/angular-6versus-react-16.3

Typescript Documentation. (2018). Retrieved from https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md#1

Tysin, K. (2014). Design of a reflective rest-based query api. https://docplayer.org/8958351-Design-of-a-reflective-rest-based-query-api.html.

Zinnen, M. (2018). Design und implementierung einer restful api für heterogene daten. https://osr.cs.fau.de/wp-content/uploads/2018/10/zinnen-2018-arbeit.pdf.