Architecture - frontend

indexmenu_n_20

frontend, architecture

The application is divided into 2 technological lines: backend (java) and frontend (javascript). There is a REST interface between the two.

- FE - Frontend (javascript)
  - `NodeJS`
  - `Gulp`
  - `ReactJS` (view layer)
    - `React router` (page navigation)
    - `Redux` (non-invasive handing of events, keeps the whole application status)
    - `React redux router` (integration of router + redux for calling the navigation status)
    - `React Bootstrap` (overusing of some of the components)
  - `Babel` (allows writing of ES6 … makes conversion to ES5, also procures JSX → JS translation)
  - `Browserify` (loader/dependency module)
  - `BrowserSync` (changes in the source code will take effect immediately)
  - `Chai`, `Mocha` (testing)
  - `lodash`
  - `Immutable` (saving the application status in immutable structures - Map, List …)
  - `Isomorphic fetch` (client communication with REST services)
  - `i18next` (localization)
  - … other see `package.json`
- BE - Backend (java)

Compared to BE, the development of FE began with a few months head start. The aim was to use the new FE over the original BE for CzechIdM for the access of endpoint users - as the user interface. The administrator part of the user interface would remain in the original CzechIdM. Therefore, within this concept, a major part of react components and pages for editing user profile, changing password and the like was created. Using the resteasy framework, REST services were built over the original CzechIdM BE.

My vision was to take FE as it is and build REST under it with the same interface as over the original BE. In the end, we opted for a few conceptual changes, which will be further described in the backend chapter.

We are now building a client running on the (`Browserify`) client using the technologies mentioned above - in the future, it might be necessary to switch to an isomorphic web (NodeJS, Webpack). We are using the **ES6** syntax (JS format).

Application modules

Core module

Contains services, managers, components and individual core pages. More in modularity and in the next chapter with layers.

App module

= Application, executable application. Connects modules with environment configuration. The minimal executable application always contains the app + core modules.

Application layers

Going from the bottom:

RestApiService ⇒ `service` (AbstractService) ⇒ `manager` (EntityManager, redux) ⇒ ui `component` (basic, advanced) ⇒ `content` (=page, AbstractContent).

and now in a bit more detail...

Loading of data from a REST endpoint is done by the service = `service`. The class `AbstractService` was created for a service with base operations. When creating a new service communicating with a REST interface, all it needs to be done is to inherit from this class and define the url of the REST endpoint (implement the `getApiPath()` method). Virtually all application services are children of this service since the REST endpoints are built with the given convention as well. `RestApiService` is used for calling REST. It calls physically the `get`, `post`, `put`, `patch`, `delete` operations on the url within the service.

The service is used by a manager (redux manager), which adds asynchronity (dispatch, promise) and state (reducer, getState()) to the service. Similarly to the service, there is an abstract class `EntityManager` for the manager, which needs to be inherited and then implement the `getEntityType()` and `getCollectionType()` methods - they state the name of the entity type which the manager will work with.

Managers are accessed by individual UI components (basic, advanced) and pages (AbstractContent), where **the life cycle is controlled by redux**:

1. action start (requesting data loading, saving ...) e.g. `this.context.store.dispatch(manager.fetchEntities())` Notice that the manager operation is not called directly, but through redux context in order to send the action and the result through redux (action = dispatch + stated = reducer).
2. listening to state with a result e.g. `manager.getEntities(state)`. Since it is all asynchronous, the redux function `connect` is used for listening to state. The function is always called when the state changes.

All the described classes are in the `core` module and can be used in any application module (one by

one, I will add links to the key words and used libraries in Git).

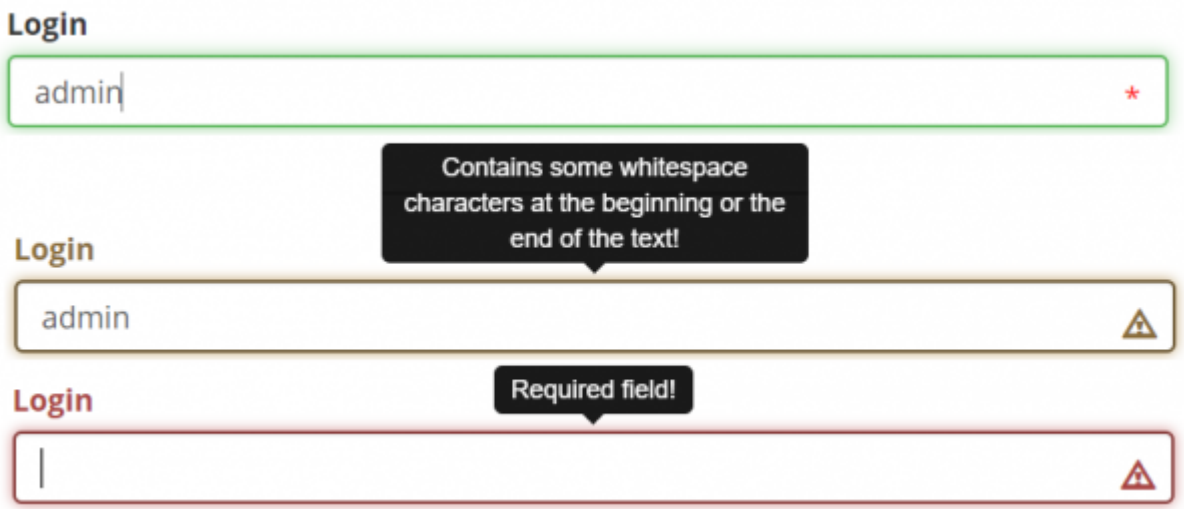
Frontend validations

validation

Validations on UI level

Validations of user inputs are important for preventing from entering a wrong values which could lead to IdM malfunction. Indication of the validation result gives a chance to user to immediately see where the problem is. Validations are applied on user inputs which are in the most general case some kinds of textual input, sometimes limited to a particular data type only e.g. numbers. From user point of view IdM uses two types of input components. Let's call them TextField and TextArea. TextField allows single line inputs only whereas TextArea is intended for multi-line inputs where several, sometimes unrelated, values are entered. Indication of the validation result is performed same way for both types of input.

The result of validation is indicated to user in several ways. We may see it in the picture below. It is an example of a TextField where are depicted all validation results when entering login for a new identity. Validation results are distinguished by using different colors. All states are accompanied with a hint bubble providing more information. The first line shows a state when the input is as expected. The second line is so called soft validation and is meant as a warning. That means the user is allowed to save and use this input but there may be anything unintended. In this case there are some leading or trailing white-space characters which might have been accidentally copied and then pasted to the input. The third line shows a case with an error. In case of occurrence of the error the filled form is not allowed to be saved. Error explains in the hint bubble that this field is mandatory and cannot be empty. Beside different colors there is also displayed an exclamation mark emphasizing occurrence of a warning or error result.



Validation results: 1-Ok 2-Warning 3-Error

Validation setting on component level

Validations used as an example in the previous section are for mentioned components (TextField and

TextArea) already implemented and are configurable via their properties. Most common validations are:
`required` - this field must not stay empty
`min` - enforces minimal length of the input
`max` - enforces maximal length of the input
If input value violates these properties then validation result is considered for being an error. User is not allowed to save the form until the problematic value is corrected.

The only currently implemented FE component property which turns on warning i.e. it enables soft validation is `warnIfTrimmable` property. When input value contains leading or trailing white-space characters the warning is displayed. In case of TextArea it is worth mentioning that every individual line must not contain these white-space characters not to display the warning. That validation is for Text Area component turned off by default. Text Field component has this validation turned on by default except the situation when it contains some confidential data e.g. a password. Following code snippet shows how to use implemented validations. Both types are booleans and can be set even conditionally.

```
<Basic.TextArea
   ref={ AbstractFormAttributeRenderer.INPUT }
   type={ attribute.confidential ? 'password' : 'text' }
   required={ this.isRequired() }
   warnIfTrimmable
   label={
   ....
```

Creating a new validation

The IdM FE uses Hapi Joi framework for validation of the user inputs. This framework contains many predefined validations for variety of data types. Beside simple checks such as minimal/maximal input length, mandatory field etc. it also facilitates check of more complex commonly used patterns e.g. email address or IP address format and many others.

Usage is quite straightforward. First it is necessary to create a validation schema, which defines a set of rules the input has to meet in order to be considered valid. The first line of the example below shows very simple rule enforcing input length in the range between 5 and 10 characters. Because every setting of a rule returns extended validation schema one can easily chain several rules together. It makes code concise and readable. Separately created schemas can be combined together using method concat as can be seen at the 3rd line. The last line invokes validation itself.

```
let schema = Joi.string().min(5).max(10);
const schema2 = Joi.any().required();
schema = schema.concat(schema2);
const result = schema.validate(value);
```

Return value **result** is an object containing result of the validation. One of its most important part is the **details** array, which contains individual detected errors. Every error is represented by an item

which provides more specific information. Particularly error type informing which rule was violated, message with more detailed description and path to the value where the error happened. See more in the official documentation.

If needed validation is not available as a configurable property of a component, developer is allowed to define one's own validation. This validation schema is created as shown above but instead of calling validation method on this schema it is passed as a parameter to component. The example below shows usage of a email address validation which is part of Joi framework. When validation schema is passed via validation property it is always treated as a hard validation i.e. invokes an error.

```
<Basic.AbstractForm
   ref="form"
   data={identity}>
   <Basic.TextField
     ref="email"
     label={this.i18n('email.label')}
     placeholder={this.i18n('email.placeholder')}
     validation={Joi.string().allow(null).email()}/>
     .....
</Basic.AbstractForm>
```