Insert here your thesis' task.

Czech Technical University in Prague

Faculty of Information Technology

Department of Software Engineering

Master's thesis

Application for Entering and Evaluation of Written Exams

Bc. Adam Havel

Supervisor: doc. Ing. Tomáš Vitvar, Ph.D.

29th April 2014

Acknowledgements

THANKS

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 29th April 2014 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Havel, Adam. *Application for Entering and Evaluation of Written Exams*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2014.

Abstrakt

V několika větách shrňte obsah a přínos této práce v českém jazyce.

Klíčová slova Replace with comma-separated list of keywords in Czech.

Abstract

Summarize the contents and contribution of your work in a few sentences in English language.

Keywords Replace with comma-separated list of keywords in English.

Contents

List of Figures

Introduction

Chapter 1

Analysis

Chapter 2

Realisation

2.1 Server

2.1.1 Architecture

2.1.1.1 Structure

2.1.2 Database

2.1.3 Internal communication

2.1.3.1 API

2.1.3.2 Websockets

2.2 Client

2.2.1 Architecture

2.2.1.1 Structure

2.2.1.2 Routing

2.2.1.3 Services

The core of the application lies in the so called services which are objects that are used to organize code into logical units and wired together using dependency injections to create new functionality. They are built upon services already present in Angular like `$http` or `$resource` and are instantiated only when another component depends on them. The so called service factory

then generates a single instance and provides a reference. Any other component that might require the same service will receive a reference to the same instance. Using this feature, data can be easily shared between different parts of the application.

Much of the core functionality is handled by aptly named services such as `ExamTake` or `NewBlueprint`. The user interface controller actually depends on many of these, so that it can provide information about their state to users. That is the reason most of the services get instantiated right at the initial load. Nonetheless, some of them are used only by a subset of users, depending on a user's role. That is why we want to know the role before we start creating instances of services a user might not be able to use anyway. Since the majority of the services also create event listeners, they would only pose as a potential memory hog.

This is where the `User` service comes in play. It is able to tell if a user is logged in and what is eventually his role and then provide that information to other components. If a user successfully authenticates, it emits a `loggedIn` message to `$rootScope`—kind of a common ground for all components that depend on it. Any service listening will receive the message and inspect the value of the role in return. Knowing that information, it can safely decide whether to self-initialize or not.

2.2.1.4 Directives

2.2.2 Optimalisation

2.2.2.1 Task automation

2.2.3 Graphical user interface

2.2.3.1 Design

Taking the proposed use of the application into account, it is crucial to embrace the idea that the design might actually be the most important aspect of the product and also the ultimate measure of its reach and usefulness. And when talking about design, we mean not just the aesthetics, of course, but also the whole field of user experience and usability. It is no secret that dealing with a poorly designed product can actually become a source of stress by itself. Or just a very unpleaseant experience at best. The situation one finds himself in while trying to use such application usually only helps to heighten

this feeling—and taking an exam can be quite gruelling as it is, as anyone can attest. Forcing students to use an application that is hard or simply obnoxious to use would epitomize nothing more than a major tell-tale sign of bad decision-making at every level of the design process.

As Dieter Rams once famously stated in his ten principles for good design (sometimes understandably referred to as the "Ten commandments"), "good design is as little design possible." And even though Rams dealt mostly with product design, the statement is applicable everywhere. Less, but better—such design can concentrate on the essential aspects and not burden the product with non-essentials. In recent years, this minimalist approach has once again proved to be in good health when the so called "flat" design seized the reigns of user interface design everywhere. The reason it is referred to as flat is because it removes any stylistic choices that give the illusion of three-dimensions (such as drop shadows, gradients, or textures) and is focused on simple elements, typography and primary colors.

In opposition to flat design lies what is called skeuomorphism. Esentially any interface that uses ornamental cues referring to an object in the physical world is said to be skeuomorphic. These cues are intended to help the user navigate the interface by creating a sense of familiarity. Among the many arguments against skeuomorphic design are those which states that skeuomorphic interface elements use metaphors that are more difficult to operate and take up more screen space than standard interface elements or that many users may have no experience with the original device being emulated. Prevalent is also the opinion that such design limits creativity by grounding the experience to physical counterparts.

Valid or not, these arguments helped pave the way for the domination of flat design. One does not have to look far for evidence—take for example the design of *Windows 8* and compare it to previous versions. Then do the same with *iOS 7*. The trend's existence is indisputable—but it is the author's belief that as with any trend, it is best not to get carried away and trust one's judgement first. It is undeniable that flat design allowed interface designs to be more streamlined and efficient, to quickly convey information while still looking visually appealing and approachable. But when used without questions and reservation, it can actually obfuscate the meaning behind an element.

This is what Dieter Rams was talking about when he said that good design makes a product understandable. Take for example an element representing

a button. If we blindly followed the principles of flat design (which many do), the element's visual identity would fall apart. But say we add a simple drop shadow—the metaphor then holds and so does the streamlined aesthetics. The points is that the button does not have to look exactly like a button to be recognized as one. Just give the user a cue and his mind will do the rest.

2.2.3.2 Typography

2.2.3.3 Performance

Chapter 3

Processes

3.1 Authentication

No part of the system is accessible without authentication. Upon loading, the client makes a `GET` request to a resource defined at `/api/user` and appends a session identifier if a cookie is found. The server then tries to look up the identifier in MongoDB-backed session store. If successful, it checks whether the session has not expired—if that is not the case, it sends the client a JSON object containing information about the particular user. The information is obtained by deserializing the user identifier from the session and using it to fetch relevant data from the database. The client application then fills its `User` service with the received object and the user is onward recognized as authenticated.

3.1.1 Logging in

In the opposite case—either no session was found or it has already expired—the server responds with a 401 status code which forces the client to show the user to a login form. After filling in the form, the client makes a new request to the `api/user` resource, this time using the `POST` method, sending along user credentials. These credentials are then checked against a faculty LDAP server running at `ldap.fit.cvut.cz` using a secure connection. If user's identity is verified, the LDAP server responds with basic information about the user. Back at the application server, the database is queried for additional data using the user identifier. It either finds a relevant entry or not. In the former case, the entry is updated with a new timestamp, representing the last login

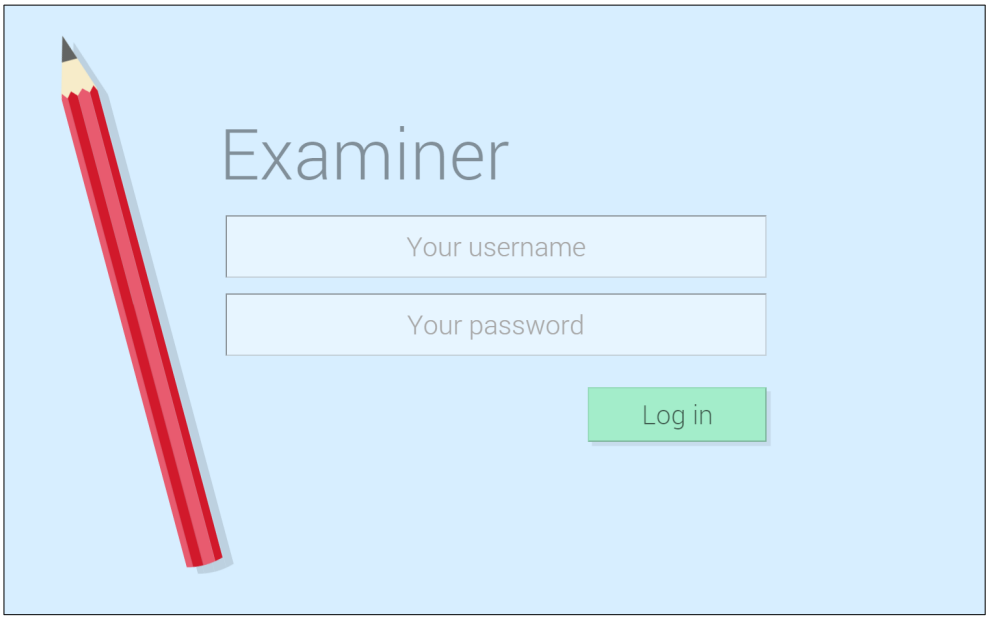

Figure 3.1: Login form

time, and the server sends it to the client as a JSON object, the same as before. If the latter is the case, it means that the user is logging in for the first time.

A new instance of `User` model is then created and filled with the available data. What remains unknown is the user's role—that is, if he or she is a student or a teacher. That information can be obtained by making a request to *KOSapi*, a faculty service that provides a REST API over the university information system *KOS*. To find out the role of a person with a given user identifier we can utilize the resource `/people/{uid}`. With that issue resolved, we can continue by looking up all courses the user either studies or teaches by using the resources `/students/{uid}/enrolledCourses` and `/teachers/{uid}/courses`, respectively. It is worth noting that *KOSapi* returns the data formatted as XML, which means the server has to transform it into a JSON object, for which it uses an external library called *x2js*. After obtaining all the neccessary data, the new user can finally be saved in the database and sent to the client.

But since both the role and the subjects of a user can change, the server has to periodically request the current states of the aforementioned resources and make neccessary changes in the application database. This is one of the

reasons the last login time is saved in the `User` model—using this information, the server makes sure not to ask for new data more than once a day. And since the majority of these changes occur at the start of a semester, long before any exam takes place, it should be frequent enough for most cases.

3.1.2 Client-side verification

When the client receives the authentication data and stores it in the appropriate service, the user can start to use the application with the scope assigned to his role. If for some reason the `User` service gets unset again, the user is shown a login form and any navigation elsewhere is disabled until he successfully logs in. In this manner, the authentication is verified only via the client-side service which might not be up to date with the server. This is sufficient for some operations and serves to mitigate any latency that would arise from communication between the client and the server.

If the user logs out, the `User` service is emptied and the client sends a `DELETE` request to the `/api/user` resource which prompts the server to discard the session. Nonetheless, the client session might also be terminated at the server end, either by force or because it has expired. It is therefore neccessary that the client recognizes this situation and acts accordingly. This is ensured by using a HTTP interception service that monitors every response the client receives from the server when it tries to reach a resource. Whenever a 401 status code shows up, it means the session has been ended and the user has to authenticate again if he wishes to continue.

The HTTP interceptor can be seen in the figure 3.2. It requires the `User` service so that it can tell whether a user is present and to log him out if neccessary. In any other type of component, the `User` service would be injected normally, but that is unfortunately not possible in this scenario. The problem arises from the fact that injecting a service into another component makes it dependent on that service. And since the `User` service is already dependent on `$http` and applying the interceptor esentially modifies the instance of the very same `$http`, making `$http` dependent on the `User` service through the interceptor would create what is called a circular dependence. That can be remedied by using the built-in `$injector` service which is able to get hold of another service reference without creating a dependance.

Something similar to the 401-filled responses also occurs when a user is not authorized to view a selected resource. This applies for example in the

```
app.factory('AuthInterceptor', function($q, $injector, Modal) {

  return {
    'response': function(response) {
      if (response.status === 401 && $injector.get('User').data) {
        Modal.open('alert', 'Your session has been ended. You will
            have to authenticate.', null, 'Log in');
        $injector.get('User').logout();
        return $q.reject(response);
      } else {
        return response || $q.when(response);
      }
    }
  };

}).config(function($httpProvider) {
  $httpProvider.interceptors.push('AuthInterceptor');
});
```

Figure 3.2: HTTP Interceptor

case when a user with the role of a student requests a different student's results. When such a thing happens, the server responds with a status code 403. Nevertheless, such request is never made on behalf of the client application and can only occur when the user tries to reach the server resource directly, via the provided API. That is why we do not have to handle this problem on the client side.

3.2 Blueprints

A blueprint represent what in real life would be the original copy of a written test. In the application, blueprints can be created, viewed and modified until the day of the exam. Quite obviously, all of these operations are allowed only for users that hold the role of a teacher. And even those can manage only the blueprints that belong to the courses they teach.

3.2.1 Resources

The blueprints are exposed as two resources, each of them designed for different use. The first one represents a collection of blueprints and is accessible

via `/api/blueprints`. It supports optional query parameters for filtering by subject, date and language. When no subject is specified, the result is automatically filtered using the list of subjects contained in the `User` service. If, on the other hand, a subject is given, it is tested against the same list and if no match is found, the server responds with a 403 code. Otherwise, the query is passed along.

In the end, the resource returns a collection of all the blueprints in the database that satisfy the inherent or supplied conditions. Anyhow, since the resource is used only for listings on the client side, it is unreasonable to request the actual content of the blueprints. Sending just the basic information will suffice and anything more than that would be a waste of bandwidth. To trim the response, the client can use the optional parameter `fields`. It is available to all the resources the API provides and when given, it prompts the server to return only the specified fields.

The other resource is used for operations dealing with a particular blueprint and can be found at URI `api/blueprint/{subject}/{date}/{language}`. Note that the variables are not optional and must be validated before any further processing, otherwise the server returns an error. The subject must be a valid subject code—starting with either `MI` or `BI` depending on the level of the subject, followed by a hyphen and ending with three word characters. The date must represent a string formatted as `YYYY-MM-DDThh:mm`, for example `2014-04-25T07:51`, and the language is validated against the standard *ISO 639-1* which allows only codes like `en` or `cs`.

The combination of these three variables is the only unique identifier of a blueprint in the application. There is a different type of key we might have used, which is the entry identifier assigned to every exam term in *KOS*, but because of the fact that an exam can be taken in different languages at once, it would not suffice. Even though such case is not very probable, it could still happen, and there is no reason for the application not to be as flexible as possible. It also helps retain a certain human-friendliness, as opposed to having to query the server API using generated identifiers.

3.2.2 Viewing blueprints

To view the blueprints, which only teachers are allowed to, the user must navigate to the state `Blueprints` which results in URL `/blueprints`. The items depicted are organized in stacks which reflect the logical distribution of blue-

prints among subjects. If only one subject is available, the user is presented directly with individual blueprints. Clear graphical distinction is made between what is considered stack and what is an individual "sheet" so that the user quickly recognizes which is which. Apart from using the browser history, the user can navigate the blueprints using a breadcrumbs navigation. This is possibly excessive in this case but makes more sense in different listings, for example when viewing exams where the structure depth can reach four levels. Also important is the fact that the current level is shared among all listings, which should help with the workflow. Anyhow, the listing is empty so far because we have yet to create our first blueprint.

3.2.3 New blueprint

To create a new blueprint, the user must enter the state `NewBlueprint` which is reflected in the URL as `/new`. She is then presented with a listing similar to the one before, only this time with exam terms instead of blueprints. Terms are the particular dates on which an exam takes place—we can get those by reaching the *KOSapi* resource `/courses/{subject}/exams`, using each of the user's subjects as the variable. The client actually asks the application server resource `api/examterms/{subject}` whose only job is to delegate the request to the aforementioned endpoint and return the transformed result. But before sending the request

3.3 Exams

3.3.1 Viewing exams

3.3.2 Ongoing exam

3.3.2.1 Overviewing an exam

3.3.2.2 Taking an exam

3.3.3 Exam evaluation

Conclusion

Bibliography

Appendix A

Acronyms

API Application Programming Interface

HTTP Hypertext Transfer Protocol

JSON JavaScript Object notation

LDAP Lightweight Directory Access Protocol

REST Representational State Transfer

XML Extensible Markup Language

Appendix **B**

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `exe` ..... the directory with executables
- `src` ..... the directory of source codes
  - `wbdcm` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format