Documentation - Museumsjakten

Group 12

Group members:

Simon Arneson (arneson), 920124-4036, arneson@student.chalmers.se
Victor Larsson (viclars), 930617-1738, viclars@student.chalmers.se
Sebastian Sandberg (sebsan), 860305-4712, sebsan@student.chalmers.se

Description of project

In the class “Praktiskt ledarskap inom innovation”, TEK475 we designed a business case regarding a quiz platform for museums to be used in educational purposes but also for regular visitors. The concept consists of questions printed as QR-codes which the museum can put up in its different exhibitions. These codes can then be scanned by visitors using an app. The visitor is then presented with a quiz question and is rewarded points for choosing the correct answer. In the app a potential visitor can also browse participating museums and check his/her score.

The project also needs an admin panel where museums can manage their quizzes, add new questions, check visitor statistics, and print QR-codes connected to their questions.

In this project we have developed three things:

- A Java EE backend with a REST-api and a connection to a Derby Database
- An admin panel website implemented in angular.js.
- An app for visitors in which they can scan QR codes written in Phonegap which is a framework where you can write javascript to create mobile applications and access native code via “plugins”.

The scope of the project

Our ER-model has been created to fit all of our current and coming use cases.
Due to the size of the project as a whole we have scaled down the features of the backend, app and admin panel to be what we consider our MVP (Minimal Viable Product), a version we can start using for user tests and gather feedback. This scaling has been done partly because we want to validate our idea but mainly because of the limited time we have during this course.

Users and permissions

Visitor:

Scan questions, answer questions, see their statistics, find museums and change his/her profile information and join groups. Logs in using his/her Facebook information.

Museum:

Logs in with a username and password. Can add, edit and remove quizzes, questions and answer options on questions. Can edit their own profile information. Can generate and print QR-codes for questions.

Team (only implemented in Database):

A team can be created by a teacher and he/she can also generate a QR-code used by students for joining the group. She can create quizzes and see statistics of her group members (students) and their performance on the quizzes.
Teams have two different types of memberships(roles): Admins (the teacher) and regular members (students)

Implemented use cases

A Visitor can:

- … signup using Facebook.
- … login using Facebook.
- ... scan a QR-code and get a question.
- … answer a question and receive points.
- … see his/her score.
- … browse museums which use Museumsjakten
- ... (scan a QR-code to join a team.)

A Museum can:

- … sign up using by entering a username and password (hash and salt implemented)
- ... login using username and password.
- … browse the quizzes created by this account.
- … create new quizzes.
- … update old quizzes.
- … add questions to a quiz.

- … view statistics for each question, ex. how many Visitors have selected each option.
- … edit a previously created question.
- … generate and print a formatted A4 containing the QR-code connected to a question.

Team (only implemented in Database):

Only implemented in ER-model and backend model, no functionality.

Technical design

UML diagram of Object Oriented Model

See "UML.pdf" in documentation folder

ER diagram of Database Model

See "ER-Diagram.pdf" in documentation folder

Software development approach

Since our system consists of both a website (the admin panel) and a mobile app a REST based approach was the best fit for us. We started off by designing or ER-model to get a full overview of our data models. To then use JPA to generate this model proved to be challenge due to the many relationships and the multiplicities and cardinalities of these relations. In the end we managed to do this and we are now satisfied with the structure of our persistence layer and REST-services.

UML diagram of physical setup

See "UML diagram of physical setup.pdf" in documentation folder

Layered UML view of the application

See "Layered UML.pdf" in documentation folder

Backend

The backend is divided into three packages.

- Core
- Persistence
- Service

com.ssv.museum.core

In the core package our main object oriented models are kept.

com.ssv.museum.persistence

The persistence package consists of one interface, IDAO, defining what methods our DataAccessObjects need to implement. The package also contains an abstract class, AbstractDAO, containing generic implementations of the methods defined in the IDAO interface. The remainder of the classes in the persistence package are Data Access Objects for each of the object oriented models from the core package which we need to explicitly persist.

com.ssv.museum.service

In the service package all classes defining our REST-api are stored. This package contains one abstract class, AuthedREST, which contains the implementations of the methods which do all authentication of both Visitors and Museum as well as hashing and salting the of the passwords connected to the Museum accounts. To allow our mobile app to access the REST-api a filter class, AccessControlResponseFilter is defined which set CORS(Cross-Origin Resource Sharing) on requests and responses. CORS is a security measure implemented in modern browsers.
We have separated our REST-api into several classes, each one containing the endpoints connected to that resource. For example if a museum wants to add a new quiz to their list of quizzes, they access and endpoint implemented in the MuseumREST class and if they want to add another question to one of their quizzes they access the endpoint implemented in the QuizREST class.

Side-note regarding the serving of QR codes.

The application revolves much around the use of QR-codes and that makes the endpoint /question/:id/qr found in the QuestionREST class (com.ssv.museum.service/QuestionREST.java) interesting. The code at the endpoint fetches a BufferedImage from the Question model based on the ID of the question. It then writes the data of the image to a ByteArray which it then sends as a response to the request. This solution results in an actual image being served as response to the requests which allows the app and admin panel to to refer to this endpoint as source for image tags and background-images in css directly without any static image hosted anywhere. This is a solution we are proud of. See example in addQuestionController.js in the admin panel website.

Side-note regarding authentication

Almost all endpoints require some sort of authentication from the request. The ones that do not are endpoints meant to be publically accessible such as listing museums, listing questions for example. The endpoints which require authentication are divided into two types with two different kinds of authentication. The endpoints which only logged in users are to be able to access are authenticated by checking that the request coming in has a valid Facebook access

token set as a header. The access token is then crossed checked to match a user in our database. The endpoints which only logged in Museum accounts are to access are authenticated using username and password. These values are also set as headers to the request. The received password is salted and hashed using methods in AutherREST and compared to the salted and hashed password which was stored in the database when the account was created.

Side-note regarding bidirectional one-to-many relations in JPA

A lot of the relations that were constructed and specified in the original ER diagram we bidirectional one-to-many relations. These allow the application to make the queries necessary present the museums, teachers and visitors with all statistics.
To create these in JPA proved to be a lot of hassle and as a result the Entity models have a lot of circular dependences between classes (for example, a Quiz has a list of Questions and each question has a reference to the Quiz). This was the only way to make JPA create the relations which was wanted. An example of a query using these relations can be found in the method getAnswerStatistics found in com.ssv.museum.persistence/QuizDAO.java

Side-note regarding placement of some API endpoints.

A decision was made to put the endpoints which create a new Quiz in the MuseumREST because this resulted in the best URI-mapping and was logical due to the fact that only Museums create Quizzes and every quiz is connected to a Museum. The same rule applies to the endpoint to create a new Question. This was moved to the Quiz API since every Question belongs to a specific Quiz. There are some more examples of this, all using the same reasoning.

External parts and APIs

All external libraries are added into the project as Maven dependencies added to pom.xml.
We access the Facebook Graph API to authenticate our Visitor accounts and fetch basic information about the Visitors. This is done using a library called RestFB(http://restfb.com).
The code using this API can be found in the methods getFacebookUserId() and getFacebookUsername() in the class AuthedREST (com.ssv.museum.service/AuthedREST.java)
The backend also uses GSON, Google's library for handling JSON. This was used because of the issue of parsing circular references in our models (which originates from the bidirectional one-to-many relations in the ER-model).
To generate QR codes which we later serve as resources to the admin panel and app a library called zxing(https://github.com/zxing/zxing) is used. This library is developed by Google. The library is used by code found in the getQR() method of the Question model (com.ssv.museum.core/Question.java).

Admin panel

The admin panel is written in javascript, html, css and uses a the javascript framework angularjs. The application is divided into different partials that are injected into the index.html file, located under the “Web Pages” folder. Inside the index.html file

Application structure

Login view
First thing a user encounters when opening the webpage, from here they can either login using their password and username or hit the registration button to register for an account.

Register view
When registering the user is prompted to type in name, username, password and email.

Admin view
This is page that the user is faced with after login or registration. For now it only displays a list over the quiz that are tied to the user. Future iterations will have more content on this page. On this page it possible to either select an already created quiz or create a new quiz.

Create New Quiz
This page contains one section where the user is prompted to fill in some basic information regarding the quiz such as name, description and total points. The user then hits save and the quiz is saved to the database and the user is redirected back to the admin page. The cancel option also redirects the user back the to previous page (admin page).

“Selected Quiz” overview (also known as manage quiz)
If the user selected an already existing quiz from the list on the admin page, the user would be faced with this page. The page consist of three sections, the first one is the same section as the one on the create quiz page, however the fields has been filled with values. The second section contains all of the questions of the quiz and a “add a new question” button (if the there were no question in the quiz, only the “add new question” button would be shown). Selecting a quiz will take the user to the question overview page where the user can edit an existing question. If the user where to press the “add new question” button, the user vill be taken to the same page but it wouldn’t contain any values.

The third and last section consist of statistics about each of the question. For now the only statistic shown is how many user have answered each of the options of the question. The statistics are illustrated with the use of pie charts. Later on we’re aiming towards implementing overall quiz stats like a top list of visitors and maybe the average score of visitors.

Question overview page
If selecting an existing question the input fields will already have values in them and the option to generate a QR-code will be available. When clicking on the “generate qr-code”, a qr code will be generated for the question and will take the place from the placeholder logos inside the qr code section. When generated, the option to print the qr-code will be available, clicking on that button will open up a new page with the printout preview of the question qr-code being displayed.

When creating a new question the input fields will be empty and the options to generate a qr-code will not be available. Submitting a valid question will make the option to generate a qr-code available.

Folder structure

- **WEB-INF** - *xml files for the glassfish server.*
- **App** - *contains the partials of the application with associated controllers and views, we have one controller for each view of the admin panel. All of the sub folders are partials except services.*
    - **addQuestion** - *Question view*
    - **admin** - *Admin view(list of quizzes)*
    - **createQuiz** - *Quiz view*
    - **directives** - *contains partials that are being used as component* objects on a page.
    - **login** - *Login view*
    - **register** - *Registration view*
    - **services** - *contains the services of the application that communicates with our api.*
    - **app.js** - *main angular module of the application, handles the injection of other modules such as ngRoute and chart.js.*
- **Img** - *local images resources.*
- **Styles** - *the stylesheets of the application.*
- **test** - *application tests.*
- **index.html** - *application skeleton, containing a navbar and a ng-view element where all of the other partials are being injected.*

Visitor application

Short intro to Phonegap (http://phonegap.com)

Phonegap is a platform which is build upon Cordova (https://cordova.apache.org).
The main concept is that you develop an application consisting of html, javascript and css just like any web app and then export the app to different platforms such as iOS, and Android. The html and javascript then run in a webview inside a native app on the device.

A big part of Phonegap is the plugins you can add to your app or develop on your own. These plugins are platform specific and are implemented by writing native code on each platform (Java for Android, Objective-C or Swift for iOS). These plugins can then be used to access native APIs which you cannot usually access from a browser or WebView such as accelerometers and more camera features. Basically you do all the design and basic controls in HTML with javascript and only write platform specific native code for the features that require it.

The Visitor app

In the Visitor app a simple interface with a side menu and login all written in HTML and css.
The application is an angular.js app which uses stateProvider for routing.
The code for the app can be found at /app/www in the git repo and it is structured and divided into different folders:

- views - Holds the html files describing the views
- img - Holds static image resources used in the app
- lib - Holds the cordova framework and other javascript libraries
- js - Holds all the application logic, written in javascript. See further down for internal structure.
- css - Holds the generated static css files.

The www folder also contains the index.html-file which is the skeleton of the application.

Javascript structure

The javascript is divided into several folders and files:

- App.js - Main entry point of the application which holds the routing and the imports of the controllers and services from the two other top level files.
- services.js - Collects all the services of the application and makes them available in the app.
- controllers.js - Collects all the application controllers and makes them available to the app.
- services/ - Folder containing the museumAPIService which is where all communication with our api is implemented and the locationService which fetches geolocation of the device.
- controllers/ - Folder which holds all of our controllers. We have one controller for each view.

The authentication against Facebook uses a Phonegap plugin to fetch an access token from Facebook which then is sent to the backend to find which Visitor it belongs to.
To scan barcodes another plugin is used which opens up the native camera and returns the data from the QR code if the data begins with our predefined string.
The css for the application is written in .scss-files in the folder app/scss which then is compiled into css and placed in the /app/www/css folder.

Testing

Backend

The REST-api of the backend has been rigorously tested using an application called Paw. This application lets us perform http request against our endpoints and set header-,path- and body-parameters. Using this we have tested of the endpoints of the REST-api throughout the development. Test runs using this application will be done during the presentation. This sort of testing was done to be able to keep the development process agile and to be able to do continuous testing without spending an excess amount of our limited development time on working with complicated testing frameworks. An interactive REST-api documentation file can be found in documentation/rest-api.html in the git repo. This file provides a full documentation of the REST-api and what parameters the endpoints expect. The urls of the endpoints are clickable and if the parameters are correct the response is printed on the page. This file was created to thoroughly test the backend system.

Admin panel

The admin panel has been tested with the help of Jasmine. It tests that the behavior of controllers and their functions are doing what they intend to. Jasmine is a behavior driven development framework that let us test controllers and functions individually with the help of mock data. With the help of these tests it’s easy to see that the expected behavior of the angular app is the outcome.

Visitor application

The Phonegap application has been tested using user testing and UI-testing. The endpoints to which the application make requests has, as previously mentioned, been done using Paw.