2016

Project Documentation P!aceS

SOFTWARE ENGINEERING TINF 14A

HERGET, MARIUS/ KRALJIC, KRALO/ NIESLER, CHRISTIAN

HEWLETT PACKARD | IBM

Table of Contents

1. Project Setup

1.1. Way of collaboration

1.1.1. Responsibilities

Responsibility	Christian	Marius	Karlo
Environment for API	X		
API Specification	X		
API Documentation	X		
Frontend Design			X
Routing between Views			X
Services	X		
Controller	X		X
Backend		X	
Documentation	X	X	X

Table 1: Responsibilities per team member

1.1.2. Team work

Everybody will receive a sub task of the overall project. We will use the time during the lecture before the exam week to implement the project. If we cannot finish the project within that time, we will resume our work in our spare time after the exams.

1.2. Development Environment

1.2.1. Repositories

We will use a GitHub repository as a Version Control System (**VCS**). It will be a hidden repo until we submit the project to the lecturer. Afterwards, the repository will be publically available.

1.2.2. Tools

- Version Control System (**VCS**): SourceTree resp. GitKraken
- Integrated Development Environment (**IDE**): Visual Studio Code resp. Atom
- Runtimes: Node.js
- Database (**DB**): MongoDB
- Languages: HTML, CSS, JavaScript
- Frameworks: MetroUI, JQuery, AngularJS, Mongoose
- Quality Tools: JsHint, JSCS (Code Style Tools)
- Automation: Grunt (Javascript Task Runner)

1.2.3. MongoDB (Backend)

The backend uses a MongoDB database. The database can be installed on the local machine (where the NodeJS backend is installed) or on a remote server. We decided to install the MongoDB database instance in a dockerized (Docker container – Virtualization) cloud environment. For security reasons we didn't open the port to be accessible from the internet, but we use SSH port forwarding instead. We connect via the SSH protocol on port 22 to the hosted Linux server and forward our local

port 27017 (default for MongoDB) to the server (source port 27017 to destination 127.0.0.1:27017). A description on how to enable SSH port forwarding can be found here.[1]

The server details:

```
User: webengineer1
Password: kcEng14As4
IP: 13.95.155.108 (static)
```

1.2.4. Running the Application

The application is started by clicking on the run.bat in the root directory. In order to start you will have to establish a proper port forwarding to the MongoDB instance as described in 1.2.3. The Web Application can be accessed by typing “localhost:3000“ in your browser.

2. SCOPE

2.1. FUNCTIONS THAT WILL BE IMPLEMENTED (WITH RESPECT TO OUR TARGET SPECIFICATION):

- PF-000: Only for new Users
- PF-020
- PF-050: Without specific search
- PF-100: Without management functionality
- PF-120: Without management functionality
- PF-150: Without search functionality

The following User Interfaces (UI) will be implemented for a responsive Web Application instead of a Mobile Application to so that we can also use this project for our Web engineering course:

- U-A20
- U-A30
- U-A40: Without search functionality

Other functionalities, i.e. the monitoring functionality or the professional functionalities won’t be implemented due to time reasons.

2.2. MACRO STRUCTURE

Since we will implemented only the web part of the application, we decided to implement a MVC structure:

[1] https://howto.ccs.neu.edu/howto/windows/ssh-port-tunneling-with-putty/

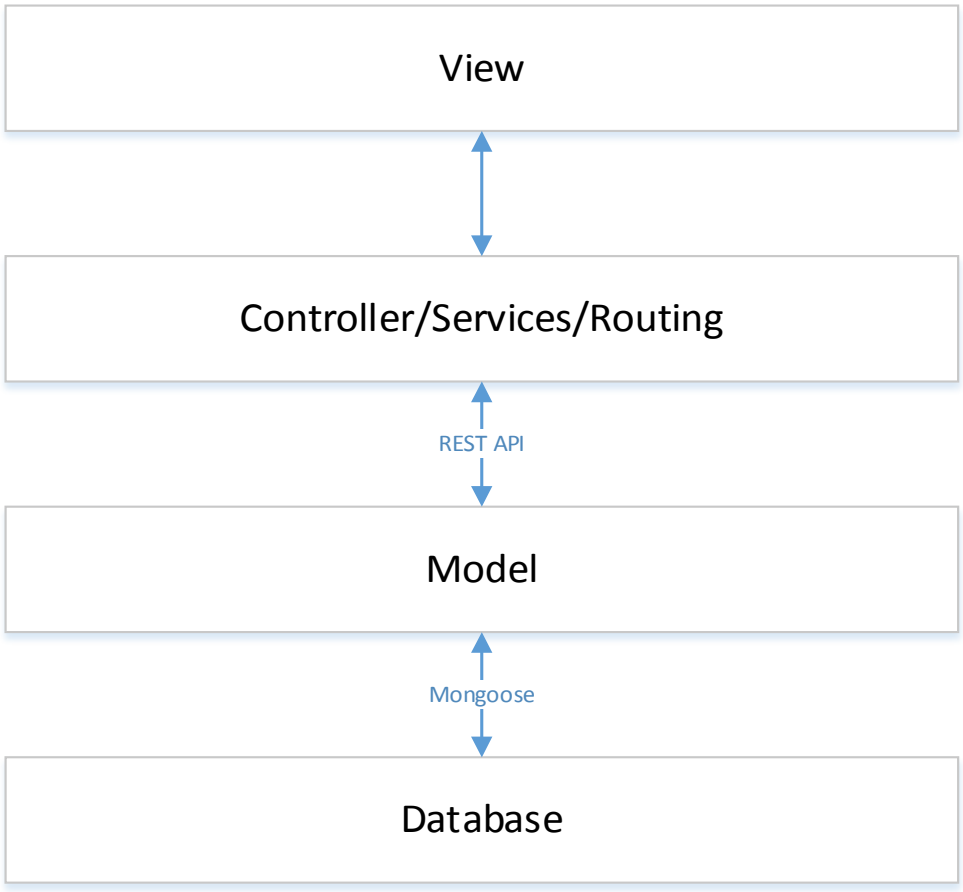

Figure 1: Macro Structure of P!aceS

Our project is split in a frontend (views and controller) and a backend (model and database) divided by a REST API.

The views are HTML files that contain the static part of each page. To make the page more dynamic, we decided to implement the frontend logic with controllers, services and routing features written in AngularJS.

The Model contains the backend logic communicating via Mongoose with the MongoDB.

2.3. Micro Structure

The micro structure will look as follows (please zoom in):

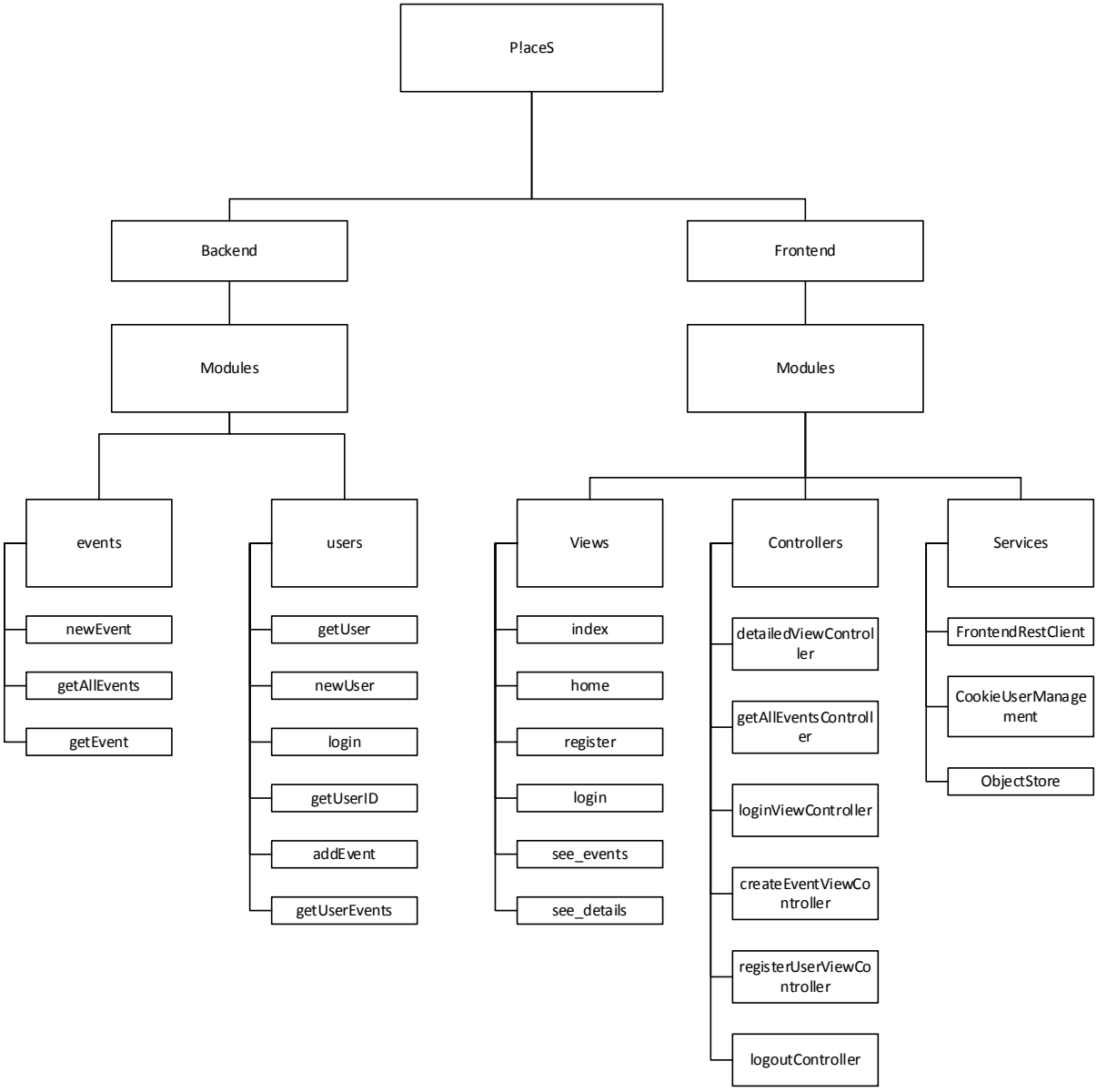

Figure 2: Micro Structure of P!aceS

The project is divided into a frontend and a backend. The backend exists out of 2 modules, one for the event logic and the other for the user logic. The naming of each method under the modules is self-explanatory.

The frontend is split in views, controllers and services. Views are HTML files that bind the controllers' and services' *.js files. Each controller is bound to the corresponding view. The services are view independent and are called by the controllers, where necessary.

All controllers and services are concatenated via a Grunt task. The views only bind the concatenated service and controller files. Therefore developers have to run the Grunt task (or double click on *grunt--force.bat* on Windows systems) before they can test their changes manually. The concatenation is used to maintain a clear structure of Services and Controllers. Controllers and Services can be maintained in

separate files and then concatenated in a single file for productive usage. Otherwise each controller and services had to be linked with a separate script tag within the HTML files.

3. Implementation Details

3.1. API Specification

The Web Application consists out of a frontend and a backend. Both parts are linked through an API. If the API is defined first there are several advantages. The frontend developers don't need to wait until the backend developers finish implementing, they can rely on the API definition. The second advantage is that an API that is defined first is usually more structured than an API that grew while implementing the backend.

There are several ways to define an API. We decided to use RAML (REST API Modelling Language) to define our API. We want to use a resource oriented architecture and therefore define a REST-API.

3.2. Reading the API with RAML

3.2.1. Resources

Resources are introduced by a "/" followed by the resource name. Nesting is realized by indentation. Same indentation means that the resources are on the same level.

```
/user
  /categories
```

Code 1: Nesting example in RAML

The service can extract information out of the URL. URL parameters are values between /… / within the URL. Placeholder for URL parameters are defined with curly braces: `/{userid}`

Query parameters are another way of retrieving information from the URL. Query parameters are key-value pairs that are appended to the URL like `www.example.com/resource?key=value&key2=value.` Those are specified in RAML as follows:

```
queryParameters:
    page: integer
    userid: string
```

Code 2: Query Parameter example in RAML

The key value pairs are specified by `key : type`.

3.2.2. Methods on resources

After a resource specification, the type of the http method is defined with the definition of the method type and a double colon. Everything that is intended below the method belongs to the method.

3.2.3. Requests

The "body" specifies the request- followed by the content-type, which is `application/json`. Afterwards, it is possible to define a JSON schema with either "type:" or "scheme:" The schema can be defined in place or in a separate file being referenced earlier in the RAML document.

```
post:
```

```
description: |
    Create a new item
body:
    application/json:
        type: item
```

Code 3: Request example in RAML

3.2.4. Responses

Responses are defined in analogously as requests. The only difference is that the http response code comes first. Multiple responses with multiple http response codes are possible:

```
200:
 description: |
   Successful retrieval of category list
     body:
         application/json:
         schema: categorylist
      404:
        description: |
         There are no categories to show
```

Code 4: Response example in RAML

3.3. FRONTEND

3.3.1. UI design

Not representative surveys among economy students have shown that Windows is the best Operating System worldwide. Moreover, they have shown, that the famous Metro look is highly favored in those circles of education. Therefore we decided to create an UI based on the Windows 8 Metro look & feel. The Metro-UI project by Sergey Pimenov and Aleksander Zadorozhnyi[2] delivers us the best framework and design guidelines to achieve this target.

3.3.2. UI structure

Internally, we will implement a single page Web Application. We only need one formally correct HTML file and substitute predefined `<div></div>` parts of the code with content from other HTML files. This provides a more dynamic way of filling pages with content resulting in the same view- and, with respect to the developers: Static content like navigation bars or bound dependencies that do not differ between the views don't have to be changed or deleted in all HTML files but in one single HTML file.

Precondition is a predefined routing concept, which looks as follows:

[2] https://metroui.org.ua/

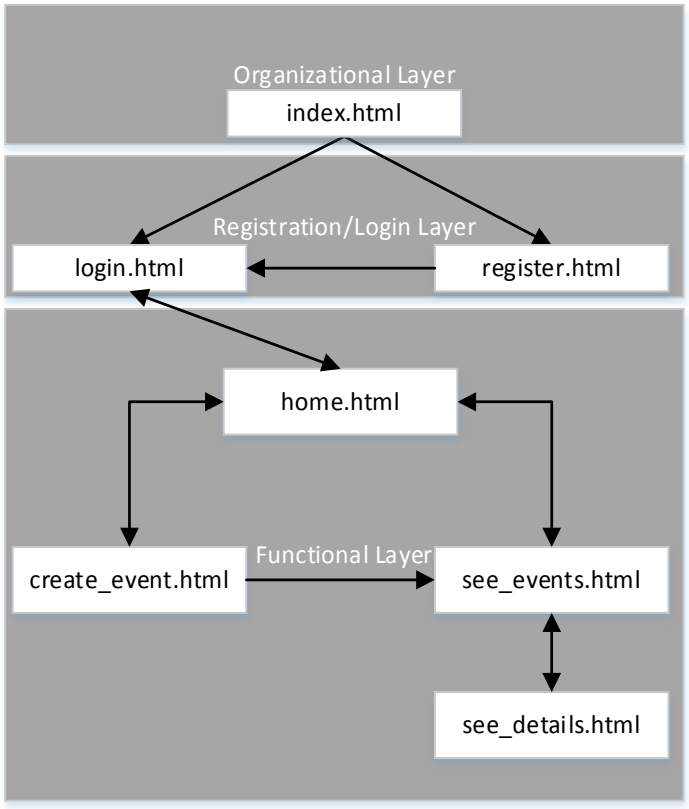

Figure 3: Routing between different Views

The index.html contains all dependencies. The content from layers below are loaded dynamically in the specified `<div>` part of the index.html. Initially, users have to register or login on the Registration/Login Layer to achieve the Functional Layer. The Registration/Login Layer will contain menus and functionalities necessary for this purpose, including one cookie as a "key" to join the Functional Layer (including key functionalities of the application) that is saved after a successful login.

The Functional Layer contains a menu and all functionalities needed by the application. Therefore we have an abstraction layer between Registration/Login & Functional Layer, and will automatically enforce potential users to register to use the application.

The arrows demonstrate the considered dynamic routing between views by elements of the page (buttons, without navigation through the navigation bar or other unforeseen events, i.e. closing the browser).

3.4. BACKEND

3.4.1. Setup

We decided to use a simple NodeJS Setup, because it has the following advantages compared with other setups (Java, PHP, simple HTML setup, .NET).

- JavaScript based (simple to use, consistent programming language in front- and backend)
- Active and vibrant community
- Simple setup, implementation and use

- Fast response time

Besides that, a NodeJS application is about 20% faster than a Java implementation for our requirements[3].

In addition to NodeJS we decided to implement a MongoDB instance with Mongoose as NodeJS interface. We use REST to provide a simple and sample HTTP(S) calling opportunity. In addition, several NodeJS addons which are described in chapter 3.4.3.

3.4.2. Scope

To show our working process we only implemented the API calls "user" and "events" (target specification: PF-100, PF-120, PF-140, PF-150, PD-100). Those represent all structures and best practices.

3.4.3. Design

The following sequence diagram shows the overall model of a simple REST call in the backend.

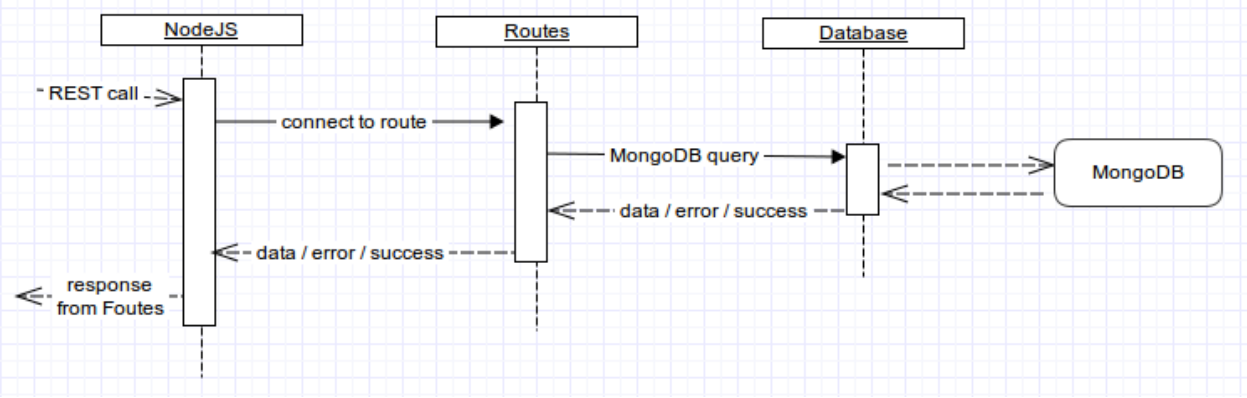

Figure 4: Sequence Diagram of a REST call in the backend

As previously mentioned, we use different NodeJS addons:

- Body-parser: A middleware module to parse requests bodies
- Cookie-parser: Middleware software to use cookies in NodeJS (TBD: implementation of sessions/login)
- Express: Implements simpler, more performant routing, HTTP implementation and better overall performance
- Mongoose: Addon to communicate simpler with the MongoDB.
- Other addons: required to run the NodeJS server

3.4.4. Folder/File structure

As best practice basement we have used our own[4] best practice guide for NodeJS DB backends. The resulting folder structure is as follows:

- Database/
    - category.js

[3] https://blog.shinetech.com/2013/10/22/performance-comparison-between-node-js-and-java-ee/
[4]Tom Burmeister, Dennis Heinze, Marius Herget and Marvin König - no publication yet

    - events.js
    - user.js
- Models/
    - schemas/
    - events.js
    - user.js
    - user.schema
- Routes/
    - api.js
    - category.js
    - events
    - user.js

Each folder has its own functionality:

- *Database*: In this folder are all DB queries and connections
- *Models:* We define our schemas to implement them in the MongoDB instance. Schemas are modeling documents which define the structure of the MongoDB internal data management
- *Routes:* Every API path has its own routes which define the events running at an API call

API Management

The routing is split into multiple files. The main file api.js forwards the calls to the called resource (event or user) and those files implement the resource specific implementation like **`POST`** or **`GET`**.

The app.js implements all these routes under /service. It includes a simple error handling and a body parser to parse all bodies.

3.4.5. Implementation of API

We will show our whole implementation process and structure by the example of the "user"-route: Every function in the code must have a valid callback to integrate a static data float.

Define schemas:

```
{
"nickname": String,
  "password": String,
  "firstname": String,
  "lastname": String,
  "properties": {
      "mobilephone": String,
      "email": String,
      "facebook" : String,
      "birthday" : String,
      "registertime": Number
  },
  "participateevents": [{ type: mongoose.Schema.Types.ObjectId, ref: 'Events' }],
  "ownevents": [{ type: mongoose.Schema.Types.ObjectId, ref: 'Events' }]
}
```

Code 5: Schema of MongoDB

Due to this schema our database holds all important user information like "nickname" or "email" in one place. In the arrays **`participateevents`** and **`ownevents`**, the database refers to event entries which indicate at which event the user will participate and which events the user is hosting.

Database connection and queries

The file *Database/user.js* holds all functions and information the application needs to run "user"-database calls.

Example for manipulating DB:

The function `newUser` is able to add users to MongoDB (*/database/userjs:41-77*). It parses the sent data from the API request into a suitable JSON object. Additionally it checks if there is an existing user with the same nickname.

```
1.  UserModel.findOne({
2.          nickname: user.nickname
3.      }
```

Code 6: Manipulating Data in DB example

If the query responds without any error, the requested INPUT event is valid for the user and mongoose saves the data `user.save()` and response to the parent function with a success callback and the user array.

If there is an error or the user already exists, the function throws an error-message (505 response).

Example for getting DB data:

The function `getUser` realizes a search through the DB. It responses user data *(/database/userjs:9-39*). We call a Mongoose search for users by `UserModel.findOne` with the wanted nickname as search parameter.

If the user does not exist, it throws an error.

If the user was found, `getUser` parses the data in a suitable JSON array and responds with this object. This parsing is not necessary but advisable as best practice to have the opportunity to add constants like "querytimestamp" or the “like” function for potential social media extensions.

Routes

The file */routes/user.js* implements the actual REST call. We implement `GET` to receive data and `POST` to manipulate data.

Example (`GET`):

The `GET` route `/:nickname` represents a possibility to receive user data under the address */service/user/test123/ (/routes/userjs:27-41*). "test123" is the query nickname. The colon signals a variable in the route providing query data without sending additionally headers or bodies.

Example 2 (`GET`):

The `GET` route */:nickname/events* implements a possibility to receive user events under the address */service/user/test123/event* (*/routes/userjs:43-83*). First we call the database function `getUserEvents`. If the user does not exist, an error is thrown. If the user exists, we receive the lists “ownevents” and “participateevents”. Then we invoke every element of the lists and access the events information with

assistance by the "events" database function `getEvent(eventid)`:

```
events.owneventslist.forEach(function(eventid){
    eventManager.getEvent(mongoose.Types.ObjectId(eventid), function(err, event){
        if (err) {
                throw(err);
        } else {
                tmpownevents.push(event);
            }
     });
});
```

Code 7: GET /:nickname/events example

Therefore we have to convert the ID from the list from a string to the Mongoose type ObjectId. When we get the event's data, the function pushes it to a temporary array we provide as a JSON response at the end.

Example (`POST`):

To manipulate the DB we can do a `POST` API call. In this example we add a new user by providing the data in a HTTP(S) body on the address */service/user/* (*/routes/userjs:85-96*). Therefore we call the database function `newUser` (see above) with the HTTP(S) body. The only thing we need to do is to catch reported errors or send a success response (200).

Connect the written call opportunities with the API:

The api.js file implements the connection between the NodeJS application and the provided API calls. We use the express function `express.Router().use("/address", file)`.

List of Codes

List of Figures

List of Tables