Technical Documentation

Event Booking

PHP-Programming Assignment

Applied Computer Science Programme

Baden-Württemberg Cooperative State University

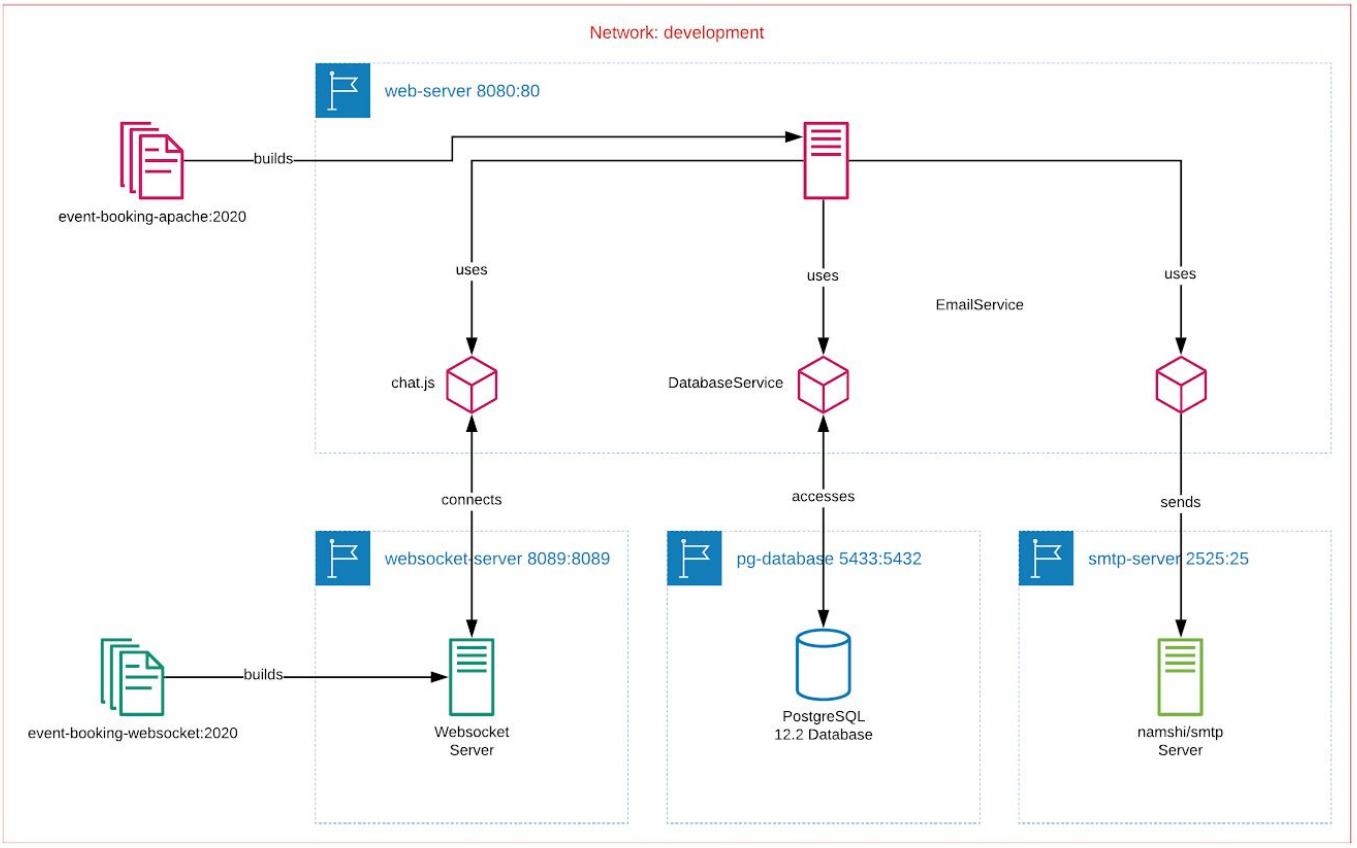
Stuttgart (DHBW)

Project period	02.03.2020 - 26.05.2020
Class	TINF18B
Editors	Andrea Budimir
	Luca Massa
	Timo Ströhlein
Lecturer	Martin Spörl

Contents

1. Architecture

1.1 Overview



1.2 Docker

The website needs to run on an Apache server with attached PostgreSQL (more on this in 3.) server inside a docker container. The SMTP and Websocket server are optional, however, without them some of the functionalities of the website are not accessible.
A **Dockerfile and docker-compose** are available for running the setting up of said container. For specific instructions on this, see the README.

The **Dockerfile** generates an image based on php:7.4.3-apache, meaning the latest php version. Also we are enabling a2enmod rewrite in order to use the Rewrite commands in .htaccess, which is needed for routing, further explained in 2.2.3. In order to support debugging we are installing and enabling xdebug with:

```
RUN pecl install xdebug-2.9.4 && docker-php-ext-enable xdebug
```

As a database we are using PostgreSQL, so wie are also installing all necessary dependencies for that.

The **root of the project** in the container will be the project's /src folder. All file references on the website need to be adjusted accordingly. Similarly, any project files outside the /src folder will not be available on the server. That means that we are copying all files from /src to the webserver directory /var/www/html in the container.

The **docker-compose** file contains four services, the **database, web, websocket** and **SMTP Mail server**. The **database** is using the latest postgres version 1.12 and running on port 5433. It has two volumes, one containing the database data, one holding the init.sql script which is executed when the container is being created.

The **webserver** is using our image created with the **Dockerfile** running on port 8080 depending on the database. It also has two volumes, one storing the webserver data and the other holding the files in the /src directory. Both services are using the same network development with the driver bridge.

The **SMTP mail server** is needed to communicate with common mail providers (e.g. Gmail). This exchange can be seen in the smtp-server container logs upon sending an email via the website. The image namshi/smtp is used: https://hub.docker.com/r/namshi/smtp/
It is configured to run internally on port 25 without authentication, since it is not reachable from outside the docker network or host machine. The hostname is set via the MAILNAME environment variable to "dhbw-event.com", meaning this is the domain the SMTP server sends emails from.

1.3 Composer

The project uses composer to manage, install and make dependencies available.
Every needed dependency needs to be added to the composer.json in the root of the project.
In order to **make the dependency available in the container**, it needs to be copied from the /vendor folder to /src/resources/assets/<package>. In order to do this, the package needs to be added to the post-install-cmd scripts in the composer.json. A shell script (composer-post-install.sh) is referenced in these scripts which refreshes the assets folder with every new needed dependency.

Shell copy and delete commands need to be added for every new needed dependency file/directory. Only needed files should be copied to the resources folder in order to keep the source folder as small as possible.

2. Website

2.1 Structure

2.1.1 Model-View-Controller

The website is structured after the Model-View-Controller (MVC) principle:

Views define the appearance of a module's functionality.
Every view consists of a separate folder in the project. Each of these folders has three files:

1. **PHTML**
   The PHTML file defines the structure of the page in the browser.

2. **JavaScript**
   The JavaScript file handles things like AJAX processes, button clicks or animated/responsive parts of the page.

3. **CSS**
   The CSS file defines the style of the page. No style tags should be present in the HTML file.

The JavaScript and CSS file are optional, they can be left out if no scripting or styling is needed.
Every view includes /views/shared/shared.phtml and shared.css to import resources that every view needs (like Bootstrap). Every view also has the option to import the header.phtml and/or footer.phtml in views/shared/… depending on the page.

Controllers control the logic and dataflow of views. Every Controller is called by the routing engine (more in 2.2.3 Core) and has the ability to invoke its view, inject variables into this view and redirect to a different view.

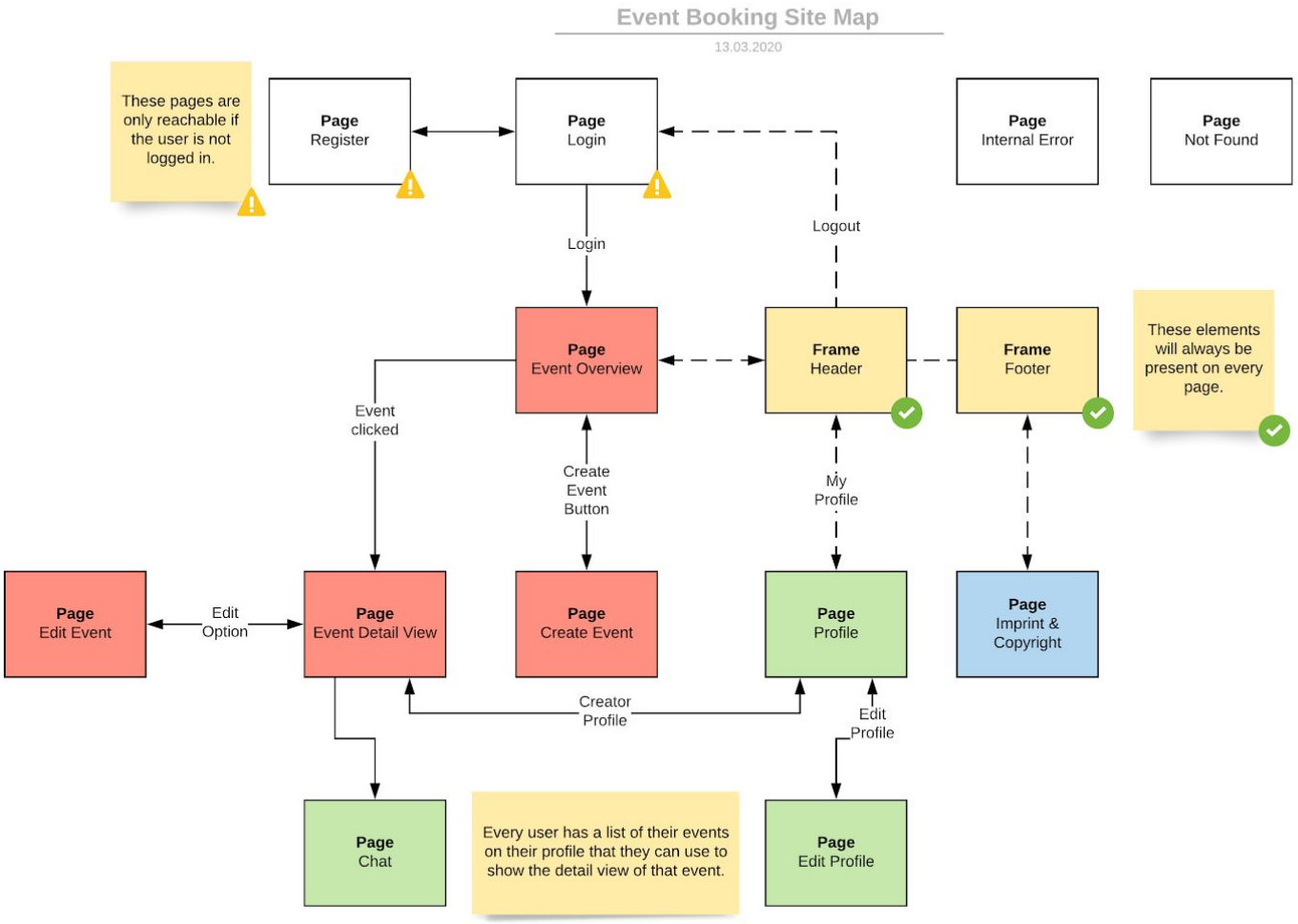
Models define the structure of database objects, offer queries and are used by controllers.

Modules are made up by a view, its controller and the used model(s). The module name is the name of the view.

Components are logical entities which define general functions and logic needed in multiple modules, for example an email service, database access and more.
If a fatal error occurs, they are the only ones that reroute the user to the internal error page.

2.1.2 Sitemap

A sitemap describes which pages exist and from where they can be reached.



2.2 Components

2.2.1 Authorization

The AuthorizationService is a service that manages everything surrounding sessions. Sessions are used to store user errors for display and more importantly, information whether a user is logged in or not. For this, a session is stored on the server with the needed information:

- Session ID
- User ID
- Login Time

The session is fetched from the server via the Session ID (SID) which is stored locally in the browser's cookies. The session contents are not editable by the user since they are stored on the server. However, a user could edit their cookies with a different SID and thus hijack a logged in user's session. To prevent this, the **AuthorizationService checks against the database** in which every session is saved until expiry.

When the service is called on a page that requires a logged in user, 3 checks have to pass before the page is displayed:

1. **User logged in**

The user_id session variable is only set if the user is logged in. It is also used by other modules to fetch all information of that user. If the user_id is not set, the user is redirected to the login page.

2. **Login not expired**

If the user is logged in, the login_time session variable is checked for expiry. If the login time is too far in the past, the user is logged out and redirected to the login page.

3. **Current connection information matches database information**

If the user is found to have a valid login, the user's IP address and HTTP User Agent is compared to the values that were stored in the database when the user first logged in with this session. If it matches, the user gains access to the called page. If it doesn't match, the user is logged out and redirected to the login page.

This authorization check happens on every page that requires a user to be logged in.

Apart from checking the session, the service also **creates the session** upon successful login and **destroys the session** upon logout, login time expiry or data validity check fail.

2.2.2 Database

The database component offers access to the PostgreSQL database running in the container pg-database.
In order to use it, either an existing instance of the Database class has to be called (getInstance) or a new one has to be created (newInstance). If a new one is created, PDO options can be passed. If no PDO options are passed, the default options are used.

The Database component makes three core database functions available:

1. **Connecting to the database**

Connection to the database happens via a PDO object. The parameters it uses are configured in /config.ini.php. This connection happens automatically when a new/existing model is called. **Thus, every model object has its own database connection.**

2. **Fetching data from the database**

With the method `fetch()`, a query and parameter data can be sent as a SELECT query to the database. It returns the fetched objects by default.
Both the query and data must be parameterized to prevent SQL Injections. The controller passes the needed data to the model, while the model provides the query and parameterized data.

The **data** must be in following format:
- Array type
- Key equals the parameter in the query

For example, the query

```
SELECT * FROM users WHERE username = :username AND email = :email
```

needs an array in the following format (note the leading colons matching the query)

```
[
   ":username" => "some username",
   ":email" => "some@email.com"
]
```

Values need to adhere to the following conventions before being passed to the Database component:
- must be escaped with `htmlspecialchars()`

- if needed must be sanitized and validated with `filter_var()`)
- empty strings (`""`) must be converted to `null`
- Booleans must be string: `"false"` instead of `false`

The function returns an object.

3. **Executing queries**

All other queries apart from SELECT can be executed with the `execute()` function. The same data specifications as in 2. apply here.

When inserting new data into tables, following things need to be considered (additionally to the conventions in 2.):

- Primary key UUIDs must receive the keyword DEFAULT in queries because they are automatically generated by PostgreSQL's library uuid-ossp
- Creation dates (users, events) must receive the keyword DEFAULT in queries because they are automatically generated by PostgreSQL's function NOW()

2.2.3 Core

The core component handles core functions, namely the routing engine of the website, a class autoloader as well as utility functions.

1. **Autoloader**

In order to access other classes within different folders, namespaces with the keyword *use* are being used. However the use instruction isn't enough by itself - including the file with either *include* or *require* is also needed. A better way of doing this is using an autoloader. It takes the namespace of a class and maps it to the directory structure, in order to load the class with *require_once*.

```
$file = $_SERVER['DOCUMENT_ROOT'] . DIRECTORY_SEPARATOR . $className .
".php";
```

As you can see above, it takes the directory to the document root, appends the directory separator "/" , the class name (namespace, e.g. components\core\Router, "\" is being replaced by "/") and the ".php" file extension.

Thanks to PHPMailer's very interesting choice of name spaces, an additional autoloader was introduced, the **resource autoloader**. It takes the first part of the use statement as a folder name and the last part of the use statement as the .php file name. It is a (sadly) necessary workaround to support the dependency file structure in /resources/assets.

2. **Router**

The Router of the application is responsible for **routing every request** against localhost:8080 (or its corresponding counterpart). Apache's rewrite engine redirects every request to index.php, where the router is called.
The router then receives the information and uses it to find the corresponding Controller, creates a new instance of it, assigns the view that belongs to the controller and shows it.
For example, the redirect localhost:8080/event-overview redirects to the EventOverviewController with the event-overview.phtml view in /views/event-overview. It is important to adhere to the UpperCamelCase naming convention for Controllers and the lower-snake-case naming convention for views (folders and files).
If no controller is found to the given route, the NotFoundController is created and displays a not found page.

In order to activate the rerouting, Apache's Rewrite Engine needs to be enabled. This is done via the .htaccess configuration file and Dockerfile commands (see 1.2 Docker for more information).

3. **Utility**

The Utility class contains **universally needed functions** like for example access to the .ini file of the project. The functions must be static.

2.2.4 E-mail

The EmailService.php is responsible for **triggering an outgoing email**. A generic sendEmail function receives the recipient, subject and message. The message is then wrapped automatically in a header and footer, as defined in the service. A template would be more elegant but the header and footer are very small.
In order to send HTML emails, the headers are also set accordingly in the service.

There are two ways to send emails:

1. **PHPs native mail() function**

PHP offers the native mail() function to send mail via the container's sendmail package. The sendmail package is installed in the Dockerfile:

```
RUN apt-get install -y sendmail
```

Furthermore, we tell PHP how to use the sendmail command via the "sendmail_path" variable in the php.ini:

```
RUN echo "sendmail_path = /usr/sbin/sendmail -t -i" >>
/usr/local/etc/php/php.ini
```

With just this, sendmail is successfully called, however, the email never arrives because it is not routed through (e.g.) an SMTP server. A SMTP server is available in the docker container “smtp-server”, but sendmail would need to be configured to use it. To adjust this in /etc/mail/sendmail.mc:

```
dnl define('SMART_HOST', 'smtp-server')dnl
```

needs to be added. Furthermore, the sendmail interface needs to be told to listen not just on localhost for connections. This requires changes to the “Addr” parameter of the DAEMON_OPTIONS in sendmail.mc (Addr needs to be removed). In order to write the changes over to /etc/mail/sendmail.cf, /etc/mail/make needs to be run.
Considering this extensive configuration needed to use a SMTP server with the standard PHP mail() function, a framework is used instead to send emails:

2. PHPMailer Framework

The PHPMailer framework offers a PHPMailer object that can be configured programmatically via PHP. All values are set according to the config.ini.php and the given parameters from the sendMail() function in the Email Service.
PHPMailer is configured to use the SMTP server to send mails through.

These mails arrive at the passed email address, however they will most likely be detected as spam. Working around this is a science in and of itself which is why this spam filtering is disregarded for now. More info on how to avoid these filters can be found in PHPMailer’s documentation:
https://github.com/PHPMailer/PHPMailer/wiki/Improving-delivery-rates,-avoiding-spam-filters

Emails in general can be fully configured with the config.ini.php. They can be:

- turned on or off (EMAIL_ENABLED). If off, needed information is displayed in the browser
- configured to use mail() or PHPMailer (PHPMAILER_ENABLED)
- configured, what address and name to use (EMAIL_FROM_ADDRESS and EMAIL_FROM_NAME)

SMTP settings (Host, Port, use authentication,...) can also be configured. **All Email settings are working as is.**

Troubleshooting:

If the website says the email was successfully sent but you have not received the email, take the follow steps to troubleshoot:

1. Check your **spam** folder
2. Try a **different email provider** as a recipient address
3. Check the **docker logs of the SMTP server** for any errors

   Note: Email service providers can block SMTP mails from certain IP addresses (and SMTP servers):

   ```
   275   SMTP<< 550-5.7.1 [79.200.118.14] The IP you're using to send mail is
   not authorized to
   275           550-5.7.1 send email directly to our servers. Please use the
   SMTP relay at
   275           550-5.7.1 your service provider instead. Learn more at
   275           550 5.7.1
   https://support.google.com/mail/?p=NotAuthorizedError g2si6233058edf.29 -
   gsmtp
   ```

   The SMTP server in use by the website has probably been blocked by Google due to multiple suspicious test emails.

4. If there is an error in the SMTP logs that you cannot resolve with ease, it is best to just **disable the email function in the config** (“IS_EMAIL_ENABLED” to false)

2.2.5 Validation

The validation component consists of classes that are called by controllers. They mostly **handle user input and check the sanitized data for any invalid values** as defined by the data dictionary or simply by logic. The validations were moved from inside the controllers to an internal component in order to keep the controllers more readable and focussed on the main logic, rather than tedious and long data validations.

A validator throws a custom **ValidatorException** that is caught in the controller in order to display the error message of the exception.

2.3 Modules

This section explains **how to use each module** and what their technical specialities are.

Some very basic modules like the *not found* or *internal error* page are omitted from this list since their functionality is limited to displaying a page with minimal content.

2.3.1 Chat

The chat module offers a page in which a user can chat with another user if **both of them** are currently on the website module /chat with each other.
If the chat partner is **not currently logged in and on the /chat window**, they will not receive the message. Instead, the sending user will receive a notification about this with the option to email the partner.
If the chat partner is **currently logged in and in a /chat window** with anyone other than the sending user, the partner will receive a notification and the option to switch to that window. The sent message cannot be seen retrospectively.

A private chat with another user can be **accessed** in multiple ways:

- on every /chat page, there is a search field that can be used to chat with a user if the username is known
- anyone that can see a specific event can message the attendees via the attendees list
- anyone that can see a specific event can message the creator via the “Message creator” button at the bottom of the event detail page

Technical notes:

- It would be possible to offer an automatic email notification, however this would require access to the Email Service either from the Websocket server or JavaScript. We’ve decided against an AJAX request from the chat.js to send an automatic email in order to avoid an unnecessarily large AJAX framework
- Messages can be retrospectively saved so they would be visible when a user receives a notification and switches to that window. This would require database saving. It is out of scope and thus not implemented

2.3.2 Confirm

This module is responsible for confirming an email address via a link that was sent to the user in an email (or shown in the browser if emails are disabled).

The module can be called in three different formats:

1. **/confirm?hash=[HASH]**

[HASH] needs to be replaced with the verification hash that was generated per user upon registration.
If the **hash is available** in the database, the verified value of the corresponding user is set to true.
If the **hash is not found**, the user is redirected to the error page (see case 2).

2. **/confirm?error**

Shown when a non-existent hash or no hash is passed.

3. **/confirm?success**

Shown when case 1 passed successfully.

2.3.3 Event Create

The event create module is represented by the **event create form** in *event-create.phtml*. It's purpose is to give the user the ability to create events. This module is only reachable when the user is logged in.

The form consists of two parts, the **required** (e.g. title, description, date, visibility) part and the **optional** (e.g. location, time, maximum attendees, price) part. Users can be **invited** when editing the event (2.3.7).

The data is being validated in two ways:

1. **HTML5 Form checks**

Every input field has a type, e.g. text, date, number, etc. in order to validate. Therefore if the type is a number, the user can't write letters in the input field. Every input field also has a maximum length, which is also checked via *maxlength="length"*.

2. **PHP Data validation**

HTML5 data validation isn't good enough on its own, because the user always has the ability to edit the HTML file, so the data is being double checked in PHP. That means that the type and length must be correct. Also it is being double checked if all required fields have been set.

2.3.4 Event Overview

The event overview module is represented by the **event overview page** in *event-overview.phtml*. Its purpose is to give the user the ability to have an overview of all events and to create new events.

Each event is represented by a card holding the most relevant information (e.g. title, description, location, date, time, price). On each card there are two buttons, one to **display** all details and one to **book** the event.

The user also has the ability to **filter the events**. For that purpose the user can search by the event name or location or can specify a timespan (from date to date). In order to prevent refreshing the page on every search, all the searching is done via javascript.

2.3.5 Event Detail

The **event detail page** is represented in the *event-detail.phtml*. The site can be accessed through the event overview page, via the details button for each event. The id of the event will then get passed to the **event detail controller**.

The page **shows all data** of an event e.g. title, creator, creation date, description, location, date, time, visibility, maximum attendees, current attendees and the price. All the data can be seen. If there is some data empty, e.g. optional fields, the field is just empty.
If the current logged in user is the event creator, the creator has the option to **edit** this event. See more information on this in 2.3.7.

In addition to that, the user has the option to **contact** the creator of the event or to **book** the event. Booking of the event is only possible when the event is **public or the user is invited** and the user is not the event creator.

2.3.6 Event Cancel

The event creator has the option to cancel the event via the **event detail page**. When canceling the event, the creator has the option to give a reason why this event has been canceled. After confirming the cancellation, all attendees will be removed and notified that the event has been canceled by the creator, in addition to the reason. After that the event will be permanently deleted.

2.3.7 Event Edit

The event creator can edit the event via the **edit** button on the **event detail page**. When editing he has the possibility to edit the following fields:

- title
- description

- location
- date
- time
- maximum attendees
- current attendees (invite and remove)
- price

When **inviting** a new user, the creator can enter the username or email of the user to be added. Every user can only be added once to the attendees list.
After completion the creator has the option to notify all attendees that the event has been updated.

2.3.8 Login

The login module consists of a **login form** in login.phtml that is accessible only when the user is not logged in. This form provides the user the ability to log into the website with the data provided on registration. This is however only possible after the user has been verified.

The form consists of two input fields, both of which are **required** for login. The first input field provides the user the ability to login either with their username or email address and the second field requires the corresponding password.

The data in both fields pass through a series of **sanitizing and escaping** to avoid malicious code. Additionally, the inserted data goes through **verification** to check if the user exists and the password matches the username or email, as well as if the user has **confirmed** their email address.

Should the user not yet be validated, they are prompted by an **error message** and have the option to resend the verification email for these purposes.

Finally, the form offers two links – one that **redirects** the user to the registration form and the other to a form to reset their password, both of which are handled in separate modules.

If valid data has been successfully inserted, the user is **redirected** to the home page and has access to all parts of the website.

2.3.9 Profile

The profile module consists of **two sub-modules**, represented by profile.phtml and profile-edit.phtml. Both are accessible only if the user is logged in and provide the ability to either view or edit personal details.

The profile sub-module is a page that **displays** a user's personal information. This includes the username and email and optionally a first and a last name, if the user provides this

information. If the user has created any events, these will be displayed on the right hand side of the user information card in tabular form.

Under the user icon there is an **edit profile** button that redirects to the profile editing sub-module. In case the person viewing the profile is not the current user, this button instead changes to a **message** button and provides the ability to contact the person whose profile the user is viewing.

The profile edit sub-module consists of a form with four input fields. These display the existing user information but are also editable. All inserted information goes through **sanitizing and escaping** to avoid malicious code or unwanted characters.

The username and email also go through a **verification process** to check if either of them belongs to an existing user. Should this be the case, the user is warned by an **error message** and should then attempt the change with different information.

In addition to this, if the email address has been altered, a new **verification email** is sent to the new address with a newly generated **verification hash** for confirmation purposes, just as is the case in the registration process.

The user also has the option to reset their password. This is however not directly done in the profile edit form but through a link which redirects to a separate form and is handled in a separate module.

The database is then **updated** with the new information and the information that has not been changed is kept as it is.

2.3.10 Password Reset

The password reset module consists of two sub-modules, both of which serve the purpose of changing the password.

The first one however, represented by the password-reset.phtml, is only accessible via the *Forgot password* link on the login page. It consists of a form with a single input field that prompts the user to insert the email address they registered with. After the entered email address goes through **sanitizing**, a verification email is sent to the user with a **verification hash**.

By clicking on the link provided in the email, the user is redirected to the second sub-module, represented by the password-save.phtml. This is also a form that consists of two input fields, one for the new password and the second to repeat the new password. If the two entries match each other, the new password is sanitized and its form is checked, after which it is matched to the user with the provided verification hash and saved.

The password-save sub-module is also reachable through the edit profile module. In this case, the user simply goes through the process of entering the new password, without having to receive a verification email.

After the password has successfully been saved, the user is redirected to the home page.

2.3.11 Register

The register module is represented by the **register form** in register.phtml. Its purpose is to give the user the ability to register themselves to the website and access the actual website contents. The register form is only reachable when the user is not logged in.

The form consists of two parts: the **required** part (username, email and password) and the **optional** part (first and last name and age).

All entered data passes three “stages” of **sanitizing and escaping** in which malicious code or html tags are removed. Invalid data such as a single whitespace are also removed.
Username and email must both be **unique**, meaning that they cannot already exist in the database. This is checked via a query.
All problems with validity or format of the data is relayed to the user via a **user-friendly error message**.

Before inserting the valid data into the users table in the database, the **password is encrypted**. A salt (defined in the config.ini.php of the project) is used to make the `md5()` algorithm more secure. It is vital for the password encryption security to keep this salt secret, otherwise it can be easily decrypted.
Additionally, a **verification hash** is generated for the purpose of confirming the user’s email address. This verification hash changes with every email confirmation.

If all data was valid and the insert was successful, the user receives a **success message** with the verification link either displayed in the browser (if emails are disabled) or sent to the registered email address.

2.3.12 Shared

The shared module includes the **header and footer** as well as a shared php file that includes generic libraries that are needed everywhere (like Bootstrap) as well as a style sheet for every page.

3. Database

The database schema can be described as an **ERM** in Addendum I.

All fields and **data specifications** can be found as a Data Dictionary in Addendum II.

For **connection** between the webserver and the database, see 2.2.2 Database.

4. Websocket

4.1 Websocket Server

The websocket server is a PHP class that is run as a **docker container**. It uses Utility functions from the Utility class and .ini settings that are accessed via a utility function. The general work cycle of the server can be described with the activity chart in Addendum III.

The Websocket Server handles **multiple client sockets** that are **identifiable by the User ID** of the user that initiated the connection. This is done via an IDENT message right after the handshake. This IDENT message is used to associate a socket with a user ID. More on this communication in 4.2.
Connected sockets can then send a JSON message with a “to” parameter that will be used to route the message to the corresponding socket (if it is connected).

An optional **timeout** can be set in the config.ini to control how fast the listen loop should run. Lowering this might negatively impact performance of your host machine. Increasing it might cause unexpected behavior and high latency. Generally, the higher you set it, the more you suffer. The lower you set it, the more your CPU suffers.

Generally, this server is a very delicate creature that WILL cry and boot a client (or itself) if you poke it too hard.

It is far from perfect, so here are some **open issues and general improvements** to have in mind while using it:

- **User Identification can be manipulated**

  A user can pretend to be someone else simply by editing the variables in the HTML script tag. Although there is surely a way to make this more secure, it is simply out of scope.

- **PING messages** are currently not handled

  Firefox has a habit of pinging the websocket server if a certain amount of seconds have passed without activity. The architecture for handling this via opcode is there, but simply not implemented yet.

- **Continuous frames** are currently not supported

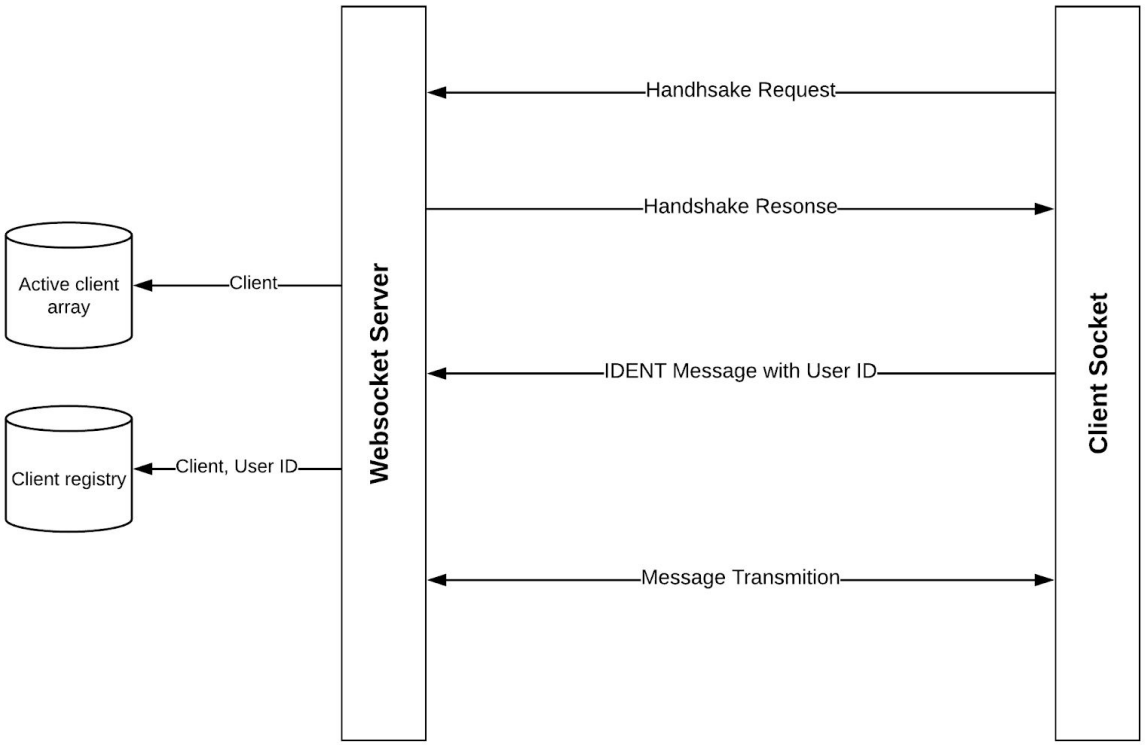
  Similarly to ping messages, continuous frames are detected but not handled.

- **Performance** is god awful

  Running the server on no/low timeout is extremely resource-heavy. Thankfully, docker limits the container to just one core, but frying bacon on your CPU is not a feature a websocket server should offer.

4.2 Website Integration

JavaScript is used in the website to support the asynchronous and continuous connection between server and client. A connection is established as soon as any /chat page is called. Clients interaction from the website to the server are as follows:



This derives the following **four message types** and their handling:

- **System** messages (Handshake, Connection closed, PING,...)

  Handshakes are detected via the socket. If the master socket has news, it is a handshake request. Responses to these messages are not masked.
  Other system messages are identified via the messages opcode.

- **Error** messages

  Error messages are usually received by the client to handle specific errors. They are of the CAPITAL_SNAKE_CASE format, starting with ERR. These messages can easily be checked in JavaScript and handled.

- **Identification** messages

Identification messages start with the keyword “IDENT” and everything behind the space separator is used as the **user ID to identify the client socket**.
No response to this message is sent. Although one could be implemented to notify the client that it is now free to send messages, the handling of this message is so fast that the user practically can’t send a message after connection is established and before the IDENT message is sent.

- **Chat** messages

  Chat messages are a JSON payload in following format:

```
{
      "from": [User ID that sent the message]
      "to": [User ID that should receive the message]
      "message": [Actual message payload]
}
```

  The “to” value is looked up in the client registry. If the recipient is connected, he receives a JSON payload similar to the one above (without the “to” variable).
  If the recipient is not connected (User ID not found in client registry), the error message ERR_USER_NOT_FOUND is sent back to the client.

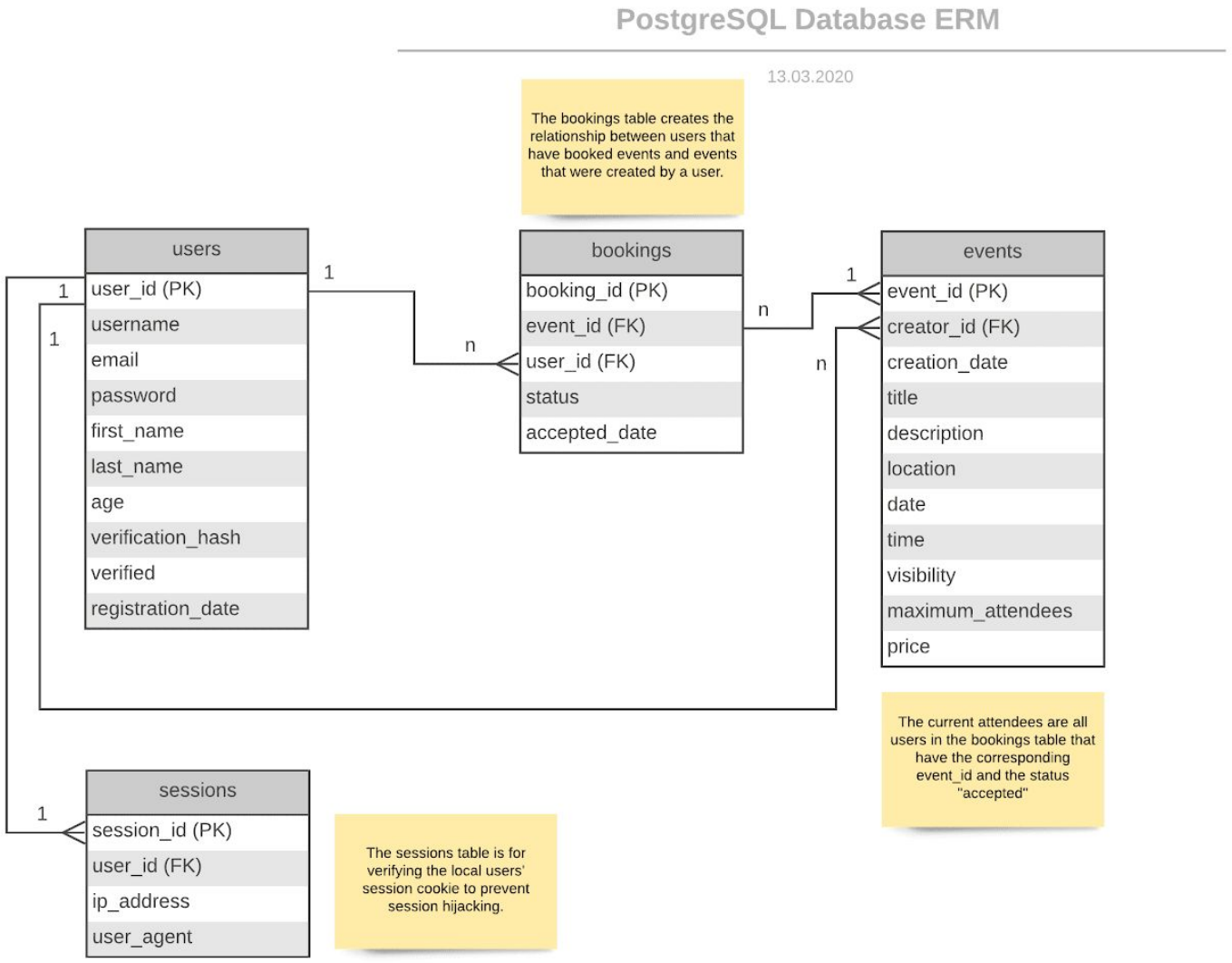
  A **sent chat message** is packed and sent to the server as well as printed to the frontend.
  A r**eceived message** is unpacked and the message value printed to the frontend.

  If a client receives a message from a User ID that they are currently not chatting with, a **notification** is shown.

  If a connection fails for whatever reason, an **error message** is shown.

Addendum

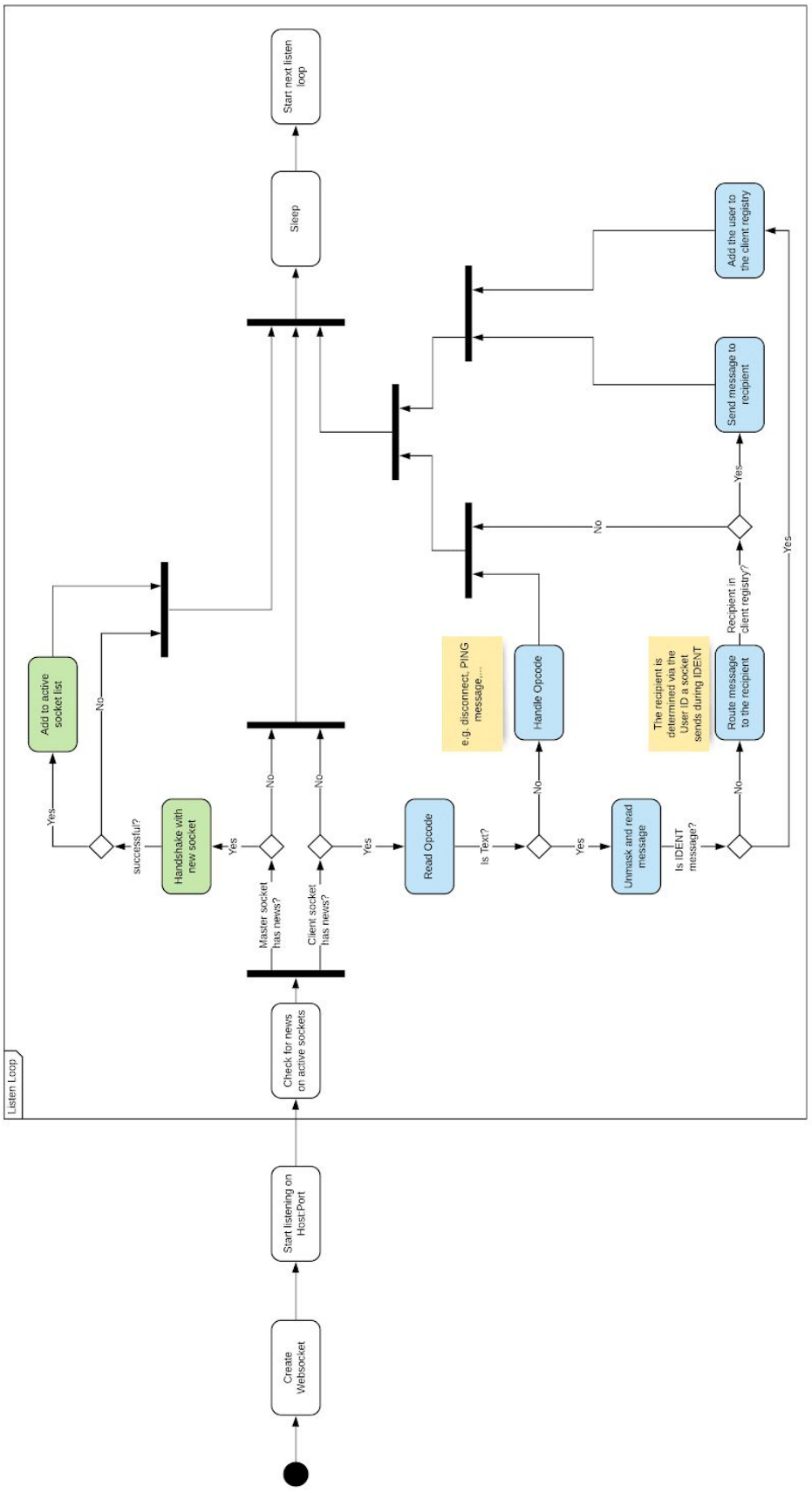
I Entity-Relationship-Model



II Data Dictionary

Table Name	Field Name	Description	Data Type	Data Format	Length	Nullable	PK/FK	DEFAULT
users	user_id	The indentifier of the user	UUID	v4 RFC 4122, ISO/IEC 9834-8:2005	36		PK	X
	username	The display name of the user	Text		32			
	email	The email address of the user	Text	name@domain.topleveldomain	64			
	password	The password of the user	Text	Hash Value	255			
	first_name	First name of the user	Text		32	X		
	last_name	Last name of the user	Text		32	X		
	age	Age of the user	Small Integer		-	X		
	verification_hash	Hash for email verification	Text	pseudo-random bytes of length 16	16			
	verified	Shows if the user has verified their email	Boolean		-			
	registration_date	Date and time when the user registered	Date	yyyy-mm-dd hh:mm:ss	-			X
bookings	booking_id	The indentifier of the booking	UUID	v4 RFC 4122, ISO/IEC 9834-8:2005	36		PK	X
	event_id	The indentifier of the event	UUID	v4 RFC 4122, ISO/IEC 9834-8:2005	36		FK	
	user_id	The indentifier of the user	UUID	v4 RFC 4122, ISO/IEC 9834-8:2005	36		FK	
	status	The status of the booking (invited/accepted)	Enum	{ "invited", "accepted" }	-			
	accepted_date	The date on which the booking was accpted	Date	yyyy-mm-dd	-	X		
events	event_id	The indentifier of the event	UUID	v4 RFC 4122, ISO/IEC 9834-8:2005	36		PK	X
	creator_id	The indentifier of the user who created the event	UUID	v4 RFC 4122, ISO/IEC 9834-8:2005	36		FK	
	creation_date	Date on which the event was created	Date	yyyy-mm-dd hh:mm:ss	-			X
	title	Title of the event	Text		32			
	description	Description of the event	Text		256			
	location	Location of the event	Text		32	X		
	date	Date on which the event happens	Date	yyyy-mm-dd	-			
	time	Time at which the vent happens	Time	hh:mm	-	X		
	visibility	Visibility settings (public/invite-only)	Enum	{ "invite-only", "public" }	-			
	maximum_attendees	Maximum number of attendees (bookings)	Number			X		
	price	Price of the event	Double	2 decimal place	8,2	X		
sessions	session_id	The SSID of the logged in user	Text		36		PF	X
	user_id	The ID of the user	UUID		36		FK	
	ip_address	The IP address of the logged in user	Text		16			
	user_agent	The HTTP User Agent of the login session	Text		-			

III Websocket Server Activity Flow



IV Websocket Integration