

# FRM System Software Architecture Document

## Table of Contents

1.	Introduction	2
1.1	Purpose	2
1.2	Scope	2
1.3	Definitions, Acronyms, and Abbreviations	2
1.4	References	2
1.5	Overview	2
2.	Architectural Representation	2
3.	Architectural Goals and Constraints	4
4.	Use-Case View	4
4.1	Use-Case Realizations	<b>Error! Bookmark not defined.</b>
5.	Logical View	7
5.1	Overview	<b>Error! Bookmark not defined.</b>
5.2	Architecturally Significant Design Packages	7
6.	Process View	9
7.	Deployment View	9
8.	Implementation View	9
8.1	Overview	<b>Error! Bookmark not defined.</b>
8.2	Layers	<b>Error! Bookmark not defined.</b>
9.	Data View (optional)	10
10.	Size and Performance	10
11.	Quality	10

# Software Architecture Document

## 1. Introduction

### 1.1 Purpose

This document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system.

### 1.2 Scope

This overview will apply for the whole FRM System project.

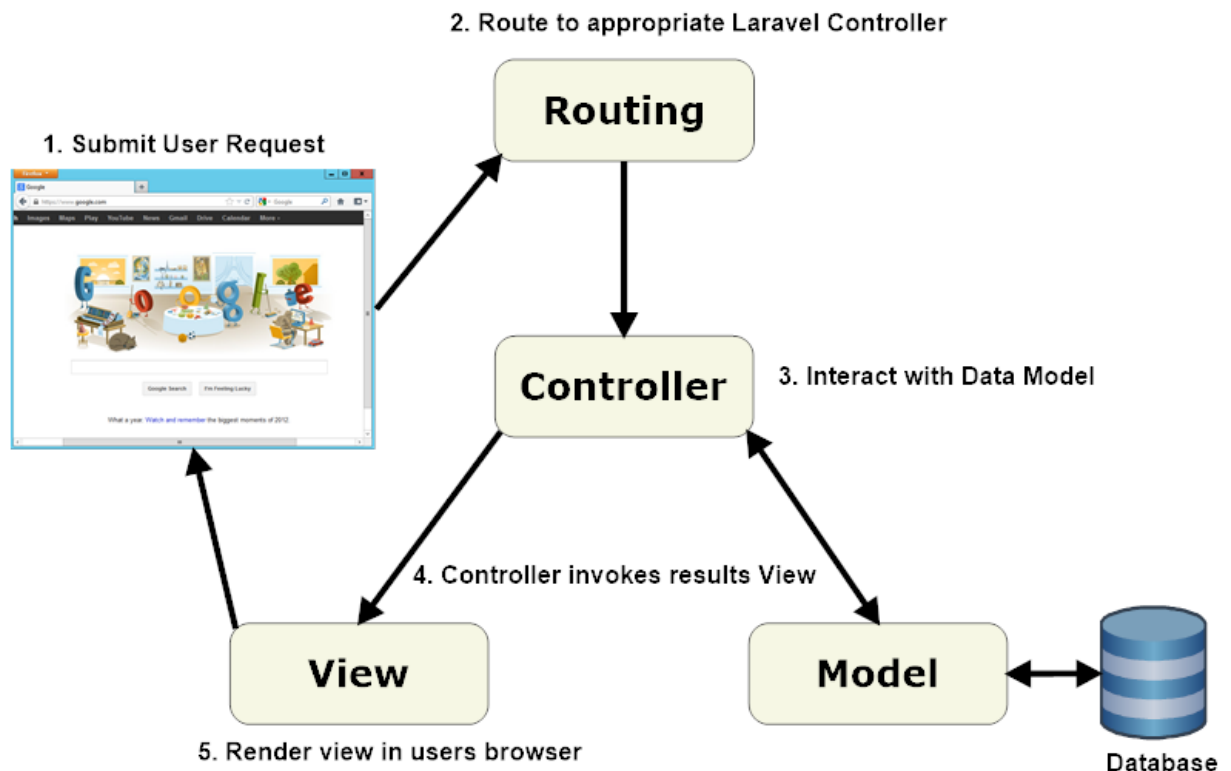
### 1.3 Definitions, Acronyms, and Abbreviations

### 1.4 References

### 1.5 Overview

The FRM System is a web application, that relies mostly on PHP. For displaying the information the user requests, the system uses HTML documents that can be viewed in many browsers such as Google Chrome, Mozilla Firefox, etc. To style these documents we use CSS files, that define how the browser is supposed to display the contained information of these HTML documents. For more in depth information on how the PHP backend generates these files and how it is structures, please view the sections below.

## 2. Architectural Representation



The FRM Systems architecture is relying on the popular Laravel PHP framework.

Every application that is based on Laravel has 3 main components: models, controllers and views.

Models are based on real-world items, e.g. machines, products or bank accounts.

They are mostly permanent and are stored outside of the application in a database, in our case this is a MySQL relational database, that gets initialized via Laravel migrations.

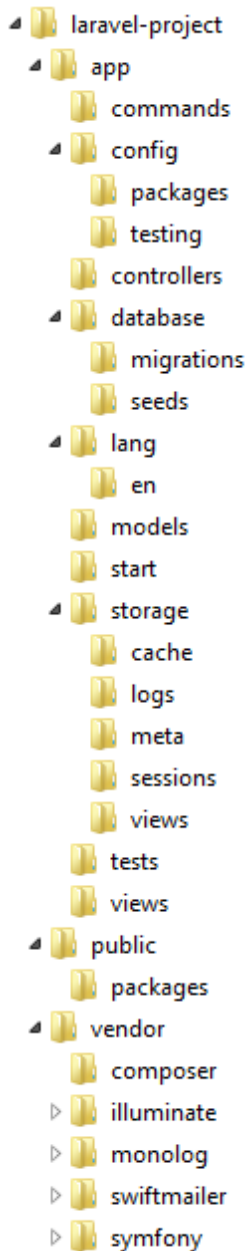
A model sets the rules and constraints of an application. Therefore it must be outside of the application to ensure that nothing in the application may break the rules.

Views contain the design and visual presentation of a model. Mostly this is achieved through a markup, the so called blade-templates, that the Laravel framework can render to HTML which the browser can read properly. Therefore the view builds the user interface based on the model.

Controllers link a view and a model. They process the input of the User through the user interface and conclude a respond, for instance rendering a new or directing to another page.

For more information on the architecture of Laravel we recommend the official documentation of Laravel which can be found here: <http://laravelbook.com/laravel-architecture/>

### 3. Architectural Goals and Constraints



On the left you can see the folder structure of a Laravel project. The app folder contains the controllers, models, views and assets.

The public folder must be linked to the web servers root directory. It contains the index.php which initiates the rest of the Laravel framework, so all requests can be redirected via the framework.

The vendor folder is dedicated to third-party code, it is used to hold other frameworks, most are installed via Composer.

The contents of the application folder are as follows:

The config folder holds the configurations of the application like runtime rules, database and session etc., in multiple config files.

For example:

The config file app.php contains application level settings as an example timezone, locale debug mode and encryption keys.

The config file auth.php contains the configuration for the user authentication.

The config file cache.php can hold the configurations for caching if a app uses this feature.

The config file database.php configures the database as an example default database engine.

The config file mail.php contains the options for the e-mail sender engine, in our case this redirects the send mails to MSMTMP.

The config file session.php controls how user sessions are managed i.e. session lifetime, generally there are cookies in use for stuff like this.

The folder controllers contain all classes that provide logic load views and interact with the data models.

The folders migrations and seeds are contained in the folder database.

The folder migrations contains PHP classes which are needed to update the Schema of the database while keeping the database in sync.

The folder seeds contains PHP files which allow Artisan to populate database tables.

The folder storage is a temporary file store for various of Laravel services such as sessions or cache. It is maintained by the framework and needs no further interaction.

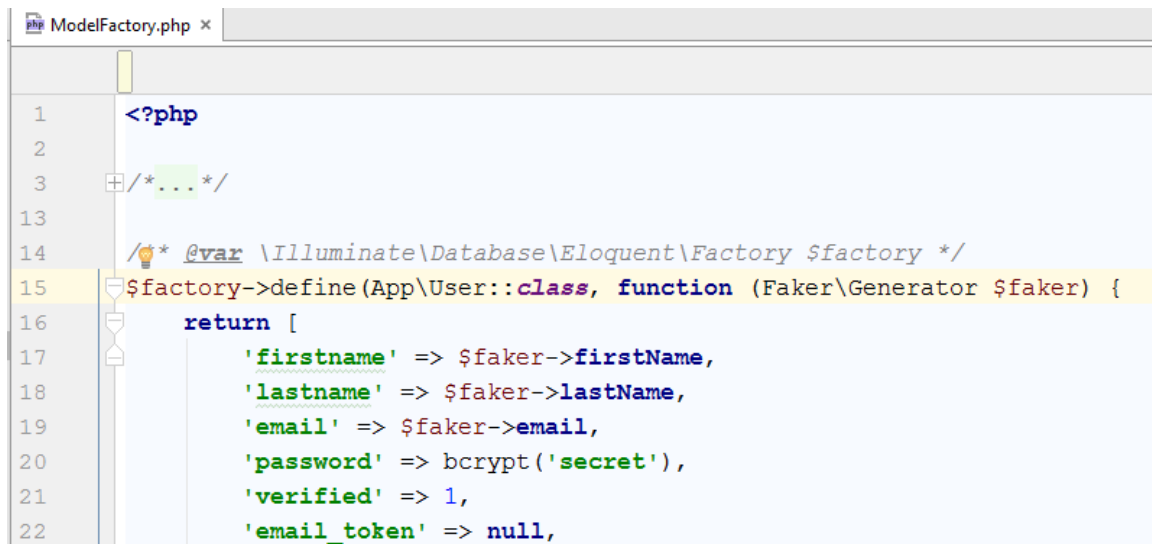
The views folder contains the HTML template files used by controllers or routes.

The routes folder contains multiple routing files which holds the routing rules that tell Laravel how to connect incoming requests.

Another goal is the use of **patterns**. Below you can find an explanation regarding the factory pattern that we used:

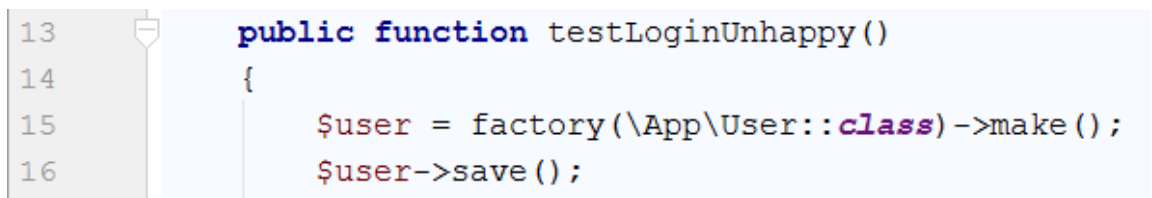
For testing you need lots of testdata. This is generated by faker, a php package, that generates data following usual data-patterns e.g. emails, names, dates etc. But it would be very tedious to generate a user

or a relationship by hand everywhere you need it in your tests. This is avoided by so called factories, the pattern type is also named “factory method”. Here you can see a snippet of our model factory declaration, we did this for all our models:



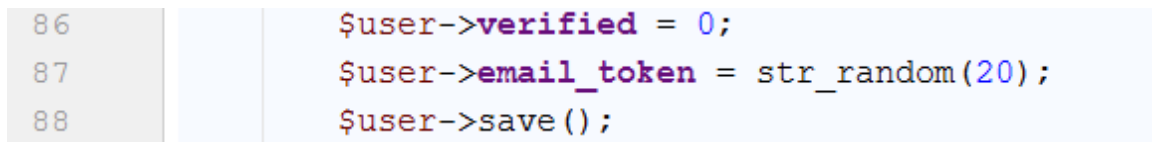
```
ModelFactory.php x
1  <?php
2
3  /* ... */
13
14  /* @var \Illuminate\Database\Eloquent\Factory $factory */
15  $factory->define(App\User::class, function (Faker\Generator $faker) {
16      return [
17          'firstname' => $faker->firstName,
18          'lastname' => $faker->lastName,
19          'email' => $faker->email,
20          'password' => bcrypt('secret'),
21          'verified' => 1,
22          'email_token' => null,
```

Here is an example how we utilize the pattern in our tests:



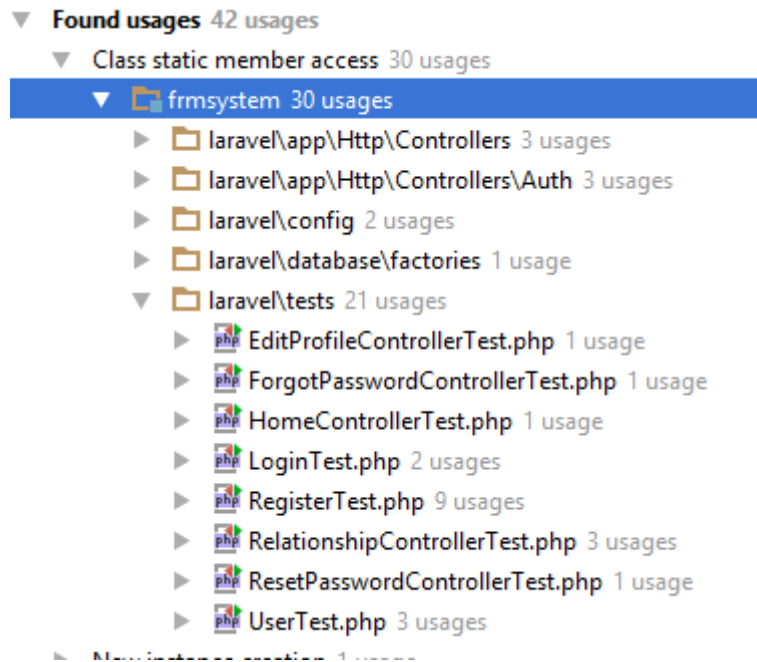
```
13  public function testLoginUnhappy()
14  {
15      $user = factory(App\User::class)->make();
16      $user->save();
```

We make use of the fact, that we can still modify the instance before saving it:



```
86      $user->verified = 0;
87      $user->email_token = str_random(20);
88      $user->save();
```

The utilization of the patterns is high, the following image shows the utilization of just the user factory:

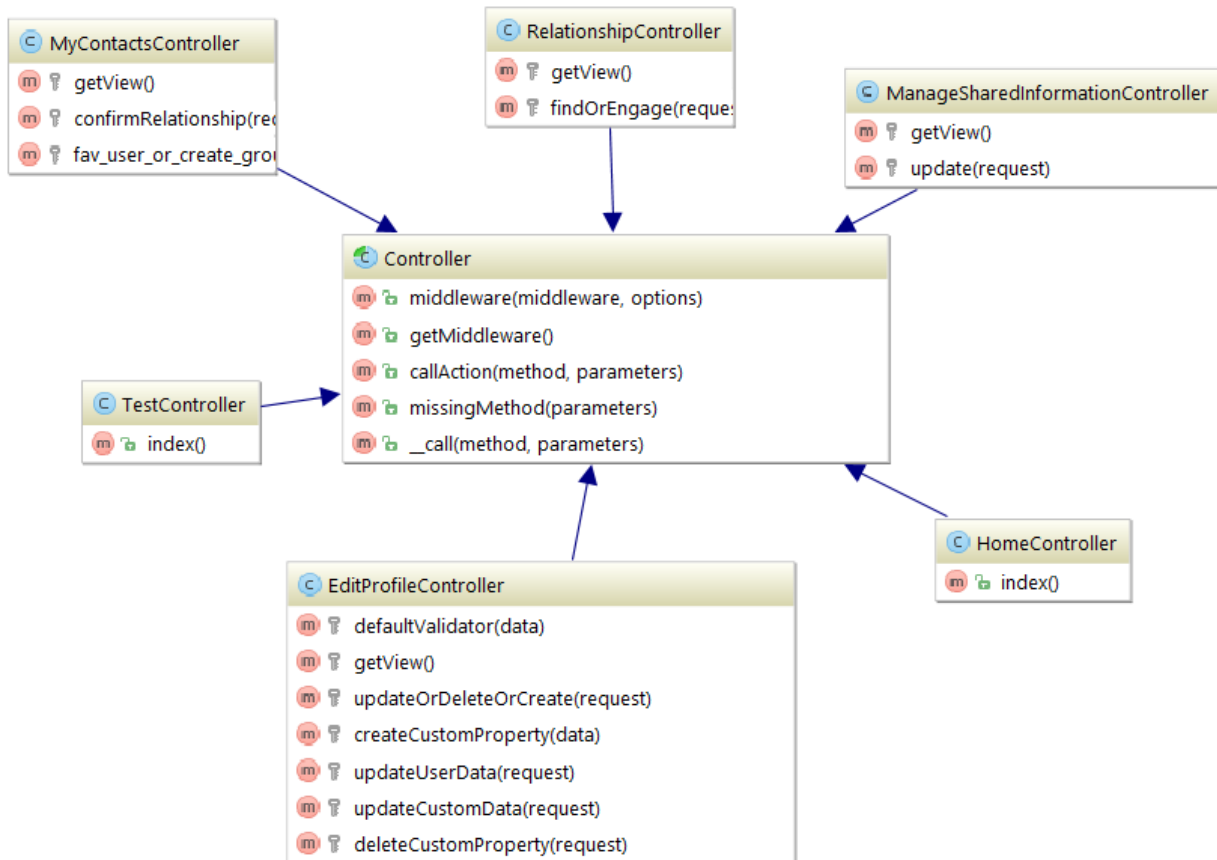


#### 4. Use-Case View

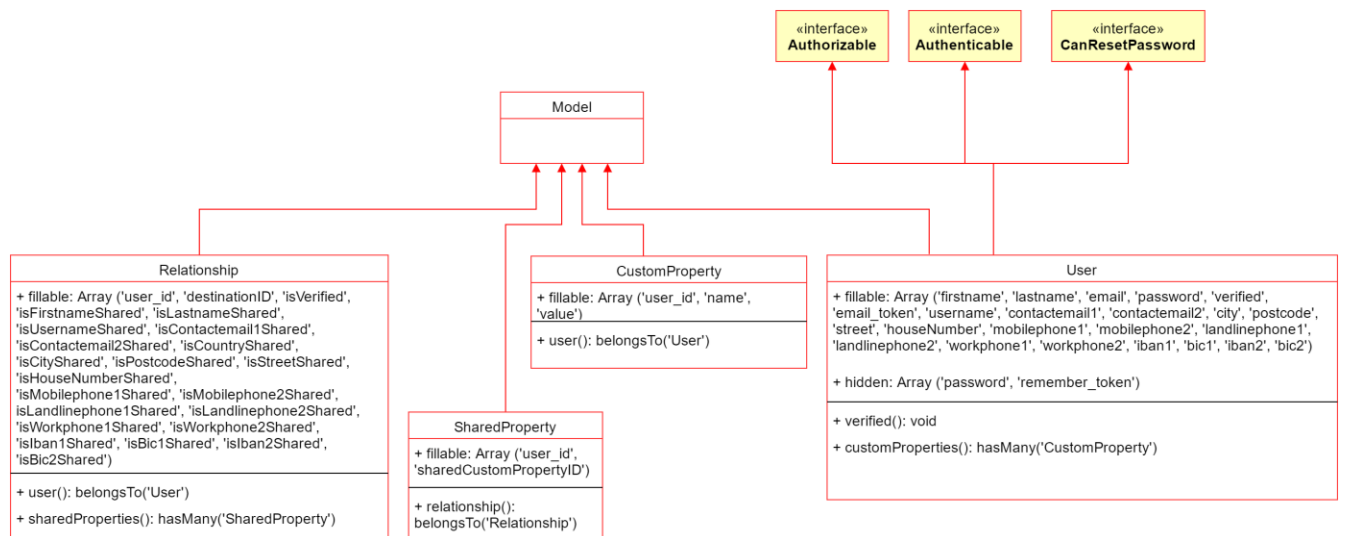
n/a

## 5. Logical View

### 5.1 Controllers



### 5.2 Models



Above you can see the class hierarchy of the FRM System-application. Since we are managing many different kinds of data, the model classes have lots of attributes to store the user's data and his relationships. It is important to note, that all of these classes also extend many of the framework-provided

classes, e.g. for managing dependencies between the models. To improve readability we did not include these in the above diagrams.



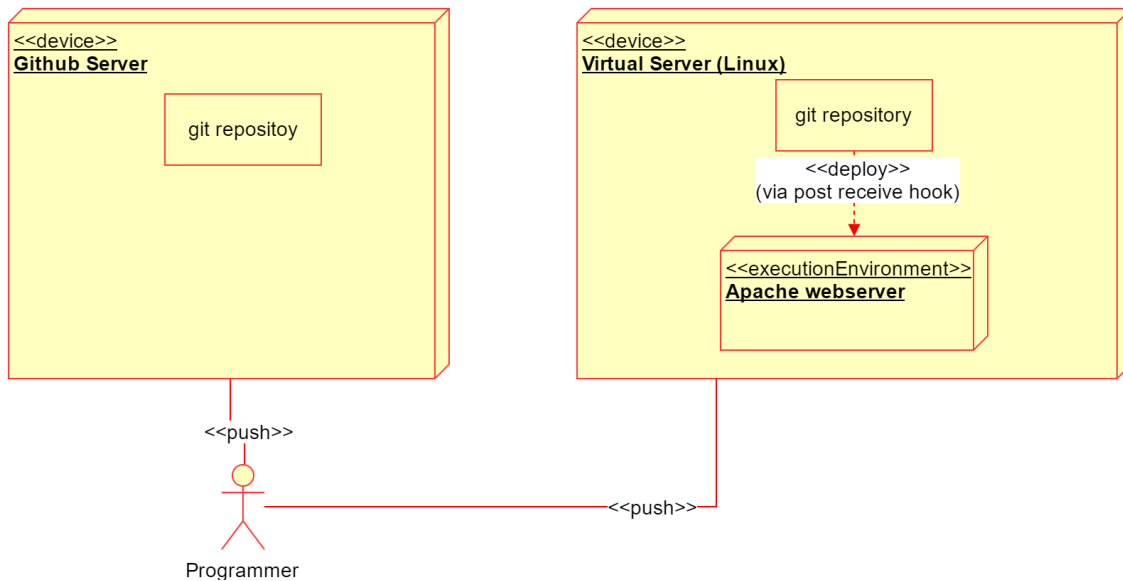
### 5.3 Views

The views for a Laravel application are – as already mentioned – realized as so called Blade-PHP-templates. They are rendered out to HTML documents by the framework before being sent back as an answer to a request by a user. They can be found here: [github.com](https://github.com)

## 6. Process View

n/a

## 7. Deployment View



We rented a virtual server on <https://www.server4you.de> with the following specs:

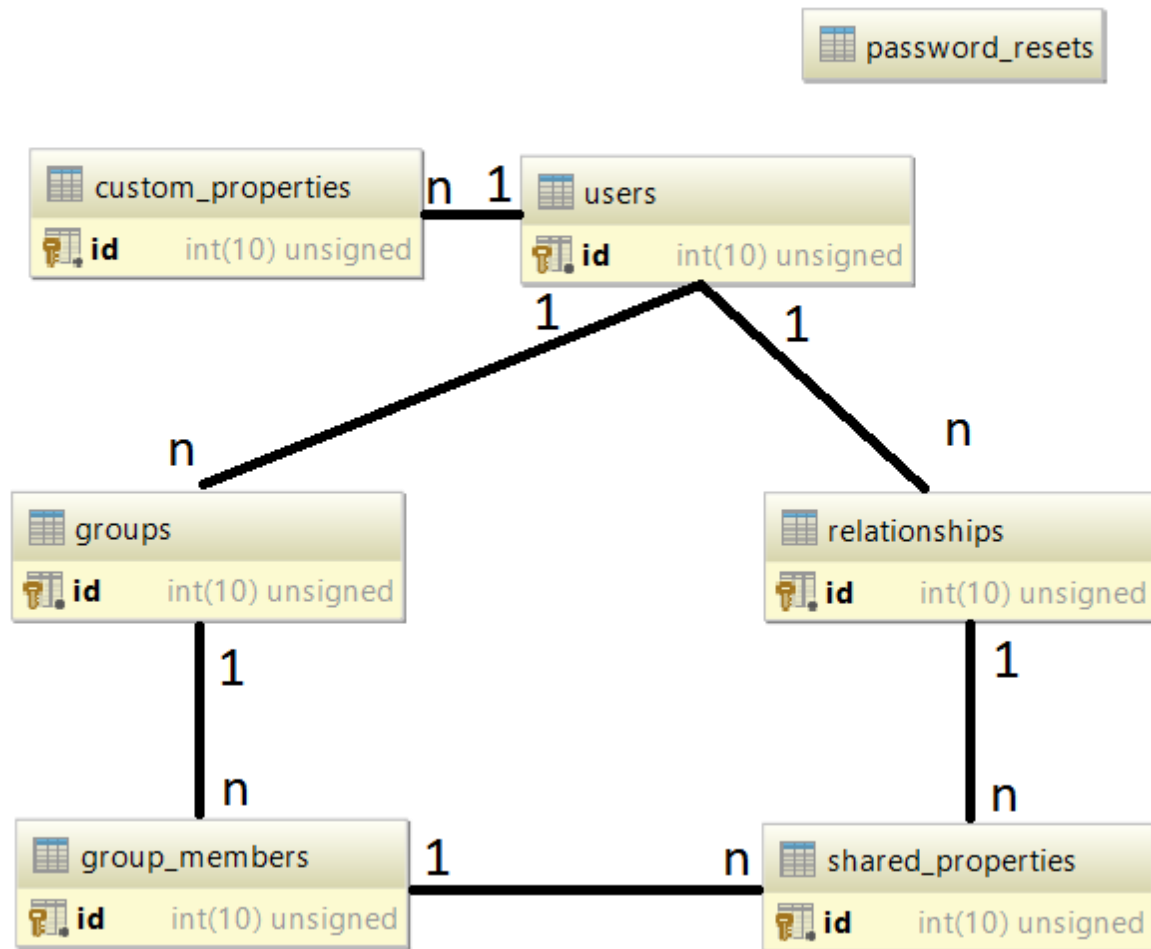
- 100 GB SSD
- 2vCores
- 4GB RAM
- Connection with 100 Mbit/s

Until now this server is handling all the tasks fine and without problems, but we are able to upscale it easily, as we see fit. It provides a maximum of flexibility that is needed to develop our application. The servers provided by DHBW would not have been sufficient for our tasks, sending mails for example would not be possible on them.

## 8. Implementation View

n/a

## 9. Data View (optional)



## 10. Size and Performance

n/a

## 11. Quality

n/a