# BackFlash

**Marcus Presa Käld**
**Aleksandar Jelicic**

**A Project Work in Web Development - Advanced Concepts**

**Jönköping University 2019**

# Introduction

Backflash is a online forum where users can interact with each other and discuss different subjects by creating posts and then commenting on each individual post.

The forum can be used for anything that the users deem interesting, posting about your favorite book or the new movie that was just released anything is possible with this website, because users can discuss all subjects on this single forum. The website is for everyone and is not limited to any specific age/gender/interest group, the website welcomes every user to share her or his feelings on the forum.
Anyone can read posts and the comments but to be able to interact in the comment section or create a post a user has to have created an account, by doing this it allows the account to have access to the functions the website has to offer. The picture below is what is presented for the user when they first visit the website (*figure 1*).

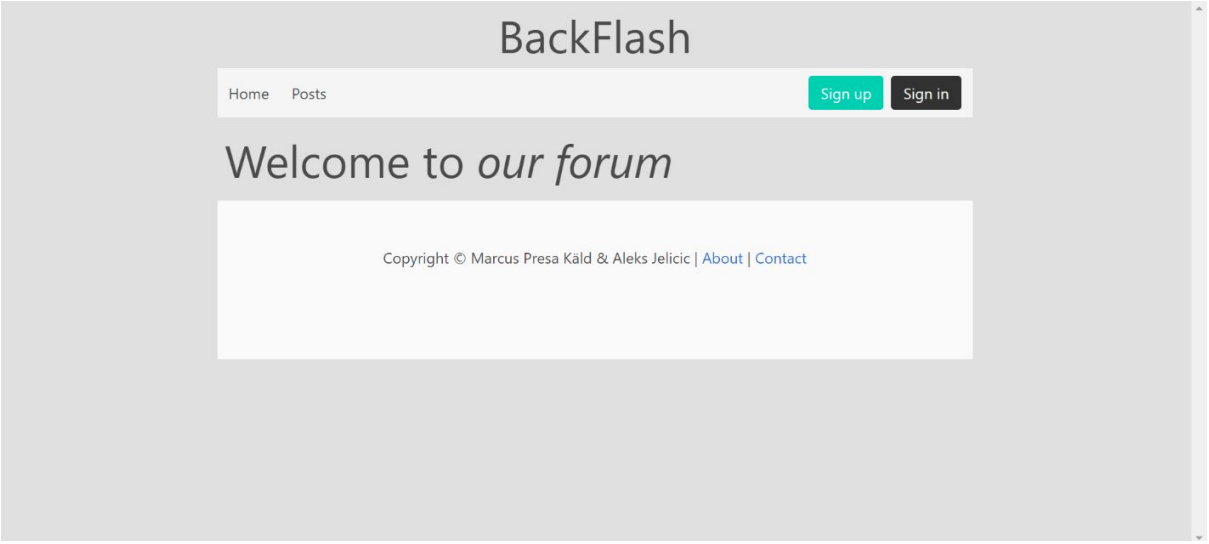


*figure 1, Front-Page*

# Architecture

The architecture of this website is structured in a three-layered-architecture style, the website consists of the main web application, a database connected to the main application and the web browser with end users that the application runs on. Users are also able to access the website through an smartphone via single-page application and REST API (*figure 2*).

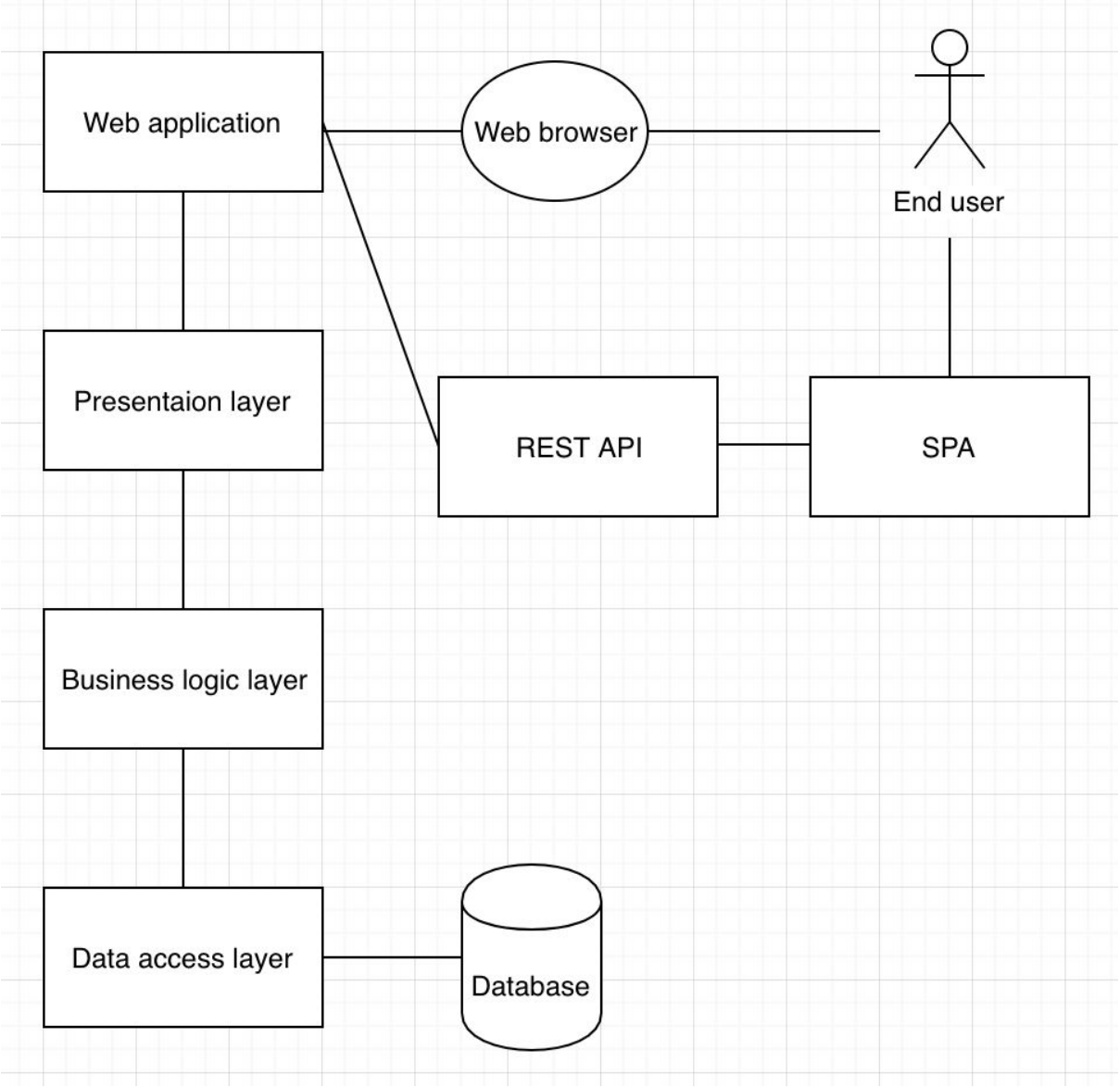


*figure 2, Architecture*

Docker is used with the web application, four containers are used which run the images  of the main web application, database, redis and Nginx that run independently of each other. (*figure 3*).

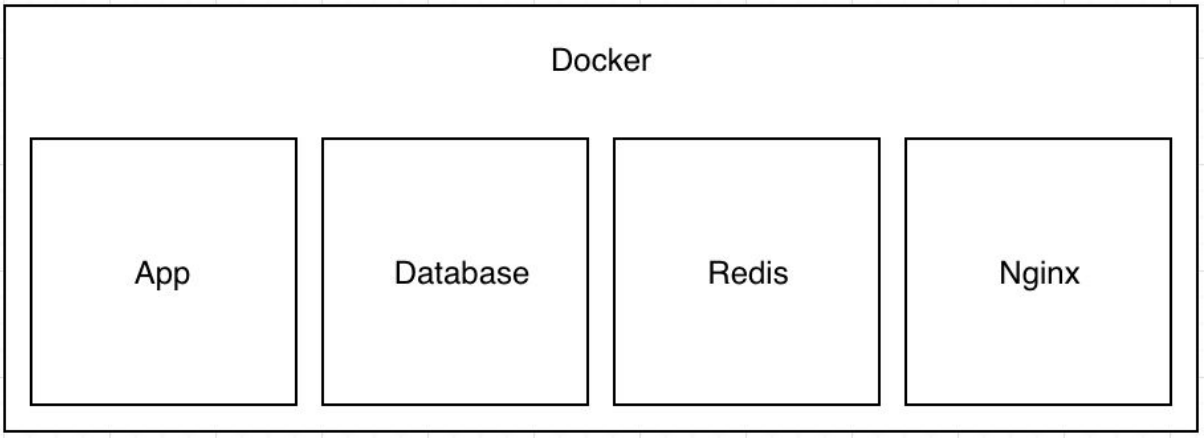


*figure 3, Docker containers*

With the help of Docker it is easier to share source code while working together with git, as Docker also includes the runtime environment for the web application, which means that there is no need to install npm packages manually as Docker does that automatically depending on which ones are used.

# Database

All data is stored in an relational database(*figure 4*). MySql is used as the web applications database in the data access layer. Another data access layer is also implemented  by using an object-Relational mapping framework, for this Sequelize is used with the existing MySql database by implementing the same interface as the other one as they have exactly the same functions.

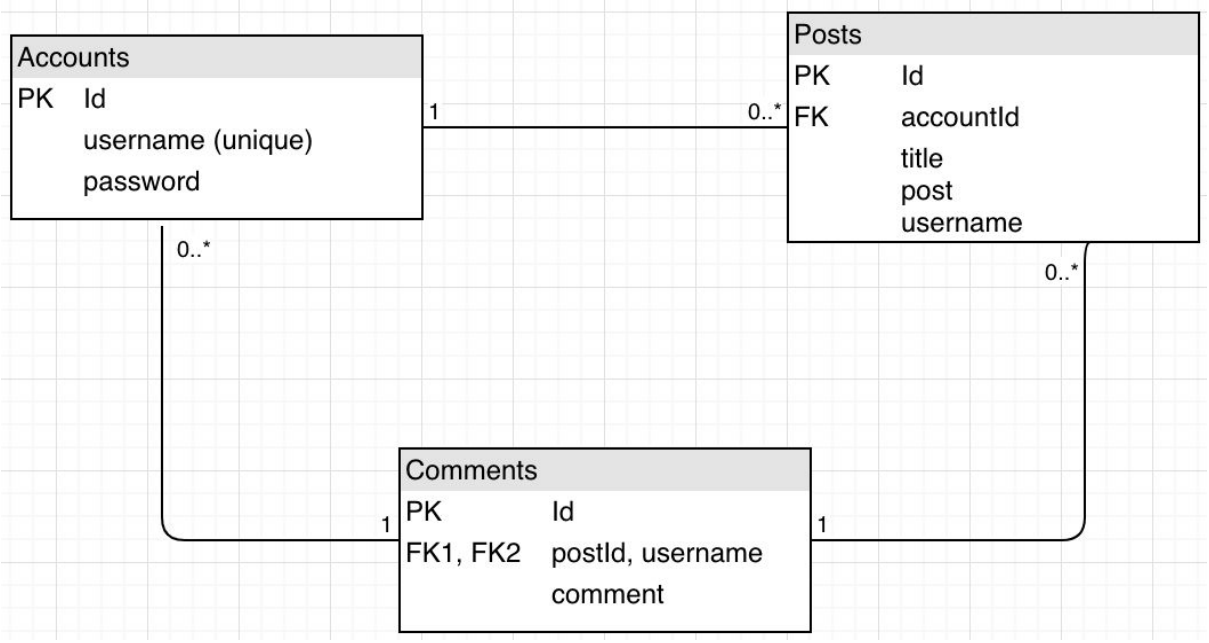


*Figure 4, Database*

The database contains accounts, posts and comments, each account can have zero or more posts and comments. Posts can have zero or more comments while comments can only belong to one account and one post. SQL queries are sent in the data access layer which updates the tables everytime there is a new account, comment or post.

Changing the database such as adding or removing tables and rows, is done by changing the “initialize-database.sql” file which contains all sql code. Using the command “docker-compose down” will remove all tables and the command docker-compose up will then add the tables and rows according to the file.

# Security

Security one the web application has been improved to protect against SQL injections, Cross-site scripting, and protect passwords by hashing them.

To the deal with SQL injections vulnerability the queries to the database are sent with placeholders such as “?” instead of the value. Templates are used when receiving values that escapes the HTML which protects against Cross-site scripting. To protect passwords, hashing is used, as hash values cannot be reversed accurately to the original passwords again.

# Web Application

The web application runs on Docker and is written in JavaScript and HTML using visual studio code with git to share source code.

Every docker image runs in a container independently of each other  which gives each container a single responsibility and makes it easier to scale.

Most of the design is made up by the Bulma framework with some added CSS. A model-controller-view pattern has been used to implement the web application, with the main file ”app.js” as the controller because it controls the application. The model is based of the database as it contains all data for the web application and the rest of the .hbs files work as the view as they reflect the model.

A Three layered architecture has been used which means that the code has been written in different files for different purposes. On the top most layer which is presented to the user, is where all different views that represent the website are located. There is a .hbs file for every page on the website that contains HTML and Bulma framework code. Some pages such as posts.hbs which represents posts from users, make use of templates to list resources from the database.

Under the top layer, lies the business layer which communicates with the data-access layer which lies below the business logic layer. As the business logic layer handles authorization and validation, it tells the data-access layer what to do.

The lowest layer, which is the data-access layer, contains all code that makes the database function. The data-access listens to the business layer so it knows what to do and can perform all the CRUD operations on the database.

Many npm packages have been used in the web application, but the most important is Express, which is used to set up middlewares that respond to HTTP requests and makes it possible to render HTML pages dynamically based on templates and passing arguments. To easily view resources on the web application, templates are used which comes from Express-handlebars.

Errors are handled in the web application. Error messages are shown when a user tries to enter a non existing page or if something is wrong while signing up or signing in. Users are prompted to fill out fill out all required fields when creating an account, posting or when signing in.

# REST API

REST API  is a architectural style, this web application exposes this concept. On the single-page-application when a user requests for a specific resource that belongs to the user, the application handles that request and sends back responses in the form of status code and json data format. For example when a user wants to create an account the user submits a POST request on the URI /accounts with the following values in json format.

```
{
    "Username":"Homer",
    "Password":"Simpson"
}
```

The header specifies that the format being sent is a json format, with the help of the code line bodyParser.json the application that the single-page application is communicating with can parse the json format and insert it into the database.

When the user has successfully created their account they will have to login on the website and authenticate their account. By submitting a POST request to /tokens the api checks if the user exists and that the grant_type is of a type password, if the user does not exists it will only return a response of 404 not found. If the user has put in the correct credentials then the api will grant the user an accessToken that can be used for authorizing for the different resources that belongs to the users account, and an idToken that the website can use to identify the user. If the user now wants to use the forum and post a post they can do it easily through the /new-post URI, when a user creates a new post the json format looks like this

```
{
    "title":"Fish",
    "post":"Fish is bad",
    "username":"Homer",
    "accountId":"Users accountID"
}
```

the user sends their username and their unique accountId, the application can access this information through the accessToken & idToken.

Since every post is connected to an accountId the user can easily access his or her own posts with the get request on the /your-posts/:accountid. When the user requests their resources to be fetched through this URI the API requests the accountID and fetches the correspondent posts that belongs to that id this time sending back the posts in json format. The response json format is the same as the code mentioned above, the user can now if they would like go in and check a specific post and perform either PUT or DELETE method on that specific post. By requesting the /your-post/:id URI the user get access to the specific

post that they would like to view, when the user chooses to use the PUT method the same URI is used for this operation. The PUT method checks so that the post really belongs to the user with the help of accessTokens before any action is done to the post. Same goes with the DELETE method, the method checks if the resource really belongs to the user before any action is taken.

# Single-Page Application

When implementing a single-page application there is a lot of client-side javascript code that edits the existing html code on the site with the help of unique identifiers such as ids on the different elements. The core of the single-page application is the navigation javascript file, the navigation file handles the different events and reacts to these events. In single-page applications no default requests are done instead the application prevents this and decides what to do with the values instead. For example if a user clicks on the Sign Up button the navigation file takes the uri and changes so that the correct content shows up on the page it also calls for the right functions that are needed. When a user presses the sign up button the navigation handles this as mentioned before, the values that the user later inputs on the site get handle by the sign-up javascript file. Here the values are fetched by getting the elementId for the input fields, the values are later sent as a json format so that the account can be created. If any error would occur on the website everything is displayed in different status codes, by doing this the site allows so that the user can easily see what error is displayed and adjust themselves depending on the problem. Single-page application are a great tool if you want a fast and responsive website on top of this single-page applications is great for user interfaces with good transitions between pages and a good interactive site for the users.