VRIJE
UNIVERSITEIT
BRUSSEL

# SOFTWARE ARCHITECTURE

## Assignment 2: Scala Play

Carlos Javier Rojas Castillo

July 28, 2020

Professor: Coen De Roover, Assistants: Kennedy Kambona, Camilo Velazquez

**Computer Science**
**Master in Software Languages and Software Engineering**

In this assignment, I use *Scala play* to implement a web application based on the *Model View Controller* pattern. In the remain, I focus on detailing the *MVC* design, provide some insight in security measures adopted by the app and inform about client-side dependencies.

**Models**

I use three different models; *User*, *Post* and *Comment*. The *User model* represents application members and corresponds with a *username* and *password*. The username is unique and the password expected to satisfy some strength. Both conditions are imposed at registration. Note that the password is saved as introduced by the user, that is, no hashing occurs to obfuscate the password, which is concerning for security reasons (e.g. server compromise, untrustworthy admins, etc.).

The *Post model* represents blog posts created by members. Each post is associated with a dummy identifier, a title, the textual content of the post, a date time and a number of likes and dislikes.

Each post can also have several *comments* associated with it. A comment is an instance of the *Comment model* and contains information about the writer of the comment, the textual comment itself and the post identifier to which the comment belongs.

Note that to facilitate app usage, dummy content is created when the application is started. A user with username "root" and password "root" is already present, ass-well as, posts about existing artists song.

**Controllers**

I use three different controllers in this app; *HomeController*, *UsersController* and *PostsController*. The *HomeController* is responsible to display the *homepage*, this involves retrieving the posts instances and passing it to the index view for further processing.

The *UsersController* is responsible for the correct *login*, *logout* and *user registration* operations. This involves using adequate *forms* for the login and registration steps (see *LoginForm* and *SignUpForm*). The forms are crucial because they validate user input and reject any invalid requests (e.g. reject weak passwords during registration). Additionally, once the user correctly authenticates, the *UsersController* associates content to the *session* making it apparent that the corresponding user is authenticated. This content is used to allowed/disallow authority operations (e.g. vote, comment, etc.) based on the presence/absence of that information and is removed once the user logs out.

The *PostsController* is about operations involving post model instances; *vote*, *comment*, *create post* and *postsByDate/postsByVote*. For the first two, the *PostsController* expects the requests to be *AJAX* requests. This concord with the idea seen in class, that is, moving towards a *single web page application* and removing the need to reload the whole page for small page changes. The *create post* request is used to create new blog posts and uses a form (see *PostForm*) to validate and reject invalid user input. The last two operations (*postsByDate* and *postsByVote*) are ordering operations that change the view mode of the posts in the homepage either to a by date or by vote order. Note that the ordering operations return a

whole html page as result. In mind of a *single web page application*, it is best to only return the ordered posts (e.g. as json).

**Views**
For the views, I tried to reuse views as much as possible. For this a main view is implemented (see *main.scala.html*) that provides reoccurring HTML structure (e.g. header, navigation bar, etc.). The main view defines placeholders that are filled by extensions on this view. For example, when generating the homepage, *index.scala.html* reuses *main.scala.html* to populate the placeholders with posts and JavaScript content.

Rendering posts can happen in two ways. 1) Either a post is not displayed in detail and a link to a detailed view of the post is included in the display (see *postsDetailsBriefView.scala.html*). This is for example used by *index.scala.html* to generate a homepage with short sized posts. 2) Or a post is rendered in a detailed manner, that is, all post content is displayed, including post comments (see *postDetails.scala.html*).

The remain of the views are about login (see *login.scala.html*), signup (see *signup.scala.html*) and post creation (see *create_post.scala.html*). Similarly, they all extend *main.scala.html* with custom HTML tags and JavaScript.

Note that depending on whether the requester is authenticated or not. Views render content differently. For example, we hide the *create post* link (present on the navigation bar) to non authenticated members, making it harder to perform authority operations.

**Security**
In this solution, security is not the main concern. Nevertheless, besides (dis)allowing authority operations to (non)authenticated users, one additional security measure is introduced namely *csrf-token*. The *cross-site-request-forgery token* is incorporated in each client-side form and is expected to be present when submitting the form. The token prevents malicious attackers to perform authenticated requests on behalf of users, given that they do not dispose of the token.

**Client-Side: JavaScript, Style and External Dependencies**
The client side JavaScript code is a single file (see *postsOperations.js*) used to implement the aforementioned AJAX requests to the server (e.g. vote post, comment, etc.). The JS code also reacts to the server responses i.e. responses to the AJAX requests. For example, a non authenticated vote results in a dialog box containing a disapproval message.

Several external libraries are used at client side. For example, *JQuery* is used to facilitate DOM manipulation. While *Bootstrap* and *Awesome fonts*, for the layout

and styling of each page. Bootstrap makes it possible to easily style client side forms, construct navigation bars and display server-side feedback through dialog boxes. *Awesome Fonts* is used to display the *thumbs up* and *thumbs down* icon of a post i.e. the clickable area of the post to up-vote or down-vote the post.