



VRIJE
UNIVERSITEIT
BRUSSEL



ASSIGNMENT 2: MVC USING SCALA PLAY

Software Architectures

**NAME: ROBIN DE HAES
STUDENT NUMBER: 0547560**

1. User Perspective

A short video showing the application in action is available at: https://youtu.be/Hb_4oxZ9WOg

2. Model

Most business logic has been implemented in the models package, so the controllers could be kept as small as possible and mainly focus on validation, calling the correct model methods and view selection.

Main classes

The **User** class represents a network user. Its `userName` field is the identifier in our dummy database. This suffices for our purposes as validation has been added to ensure `userName` entries are unique, but in a real application using an additional separate `userID` field would probably be better (e.g. for keeping historical data after a user is deleted while permitting another user to reuse that `userName`, etc.). A constant field `UnstoredUserName` has been added as well, which can be used for representing a User that is unauthenticated.

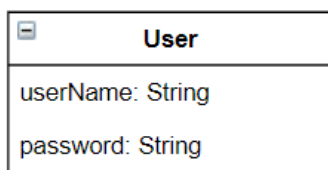


Figure 1.1. Model class User

The **Post** class represents a post made by a User. It has a `postID` field which is used as identifier in the database. There is also a constant field `UnstoredPostID` representing the ID of posts that are created, but not yet stored in the database. When storing a Post, the ID will be filled in with a valid auto-generated value (by `PostDao`).

The field `posterName` is a String referencing the `userName` of the User that posted this. Alternatively, we could have also stored all Posts as a field on the User but for our purposes we mostly needed access from the Post to the User and not the other way around.

The field `likes` is a list of `userName` Strings referencing the users that liked the post. This field and the `comments` field have only been made constructor parameters for easily filling up our dummy database, since in most cases they would be empty when constructing a new Post.

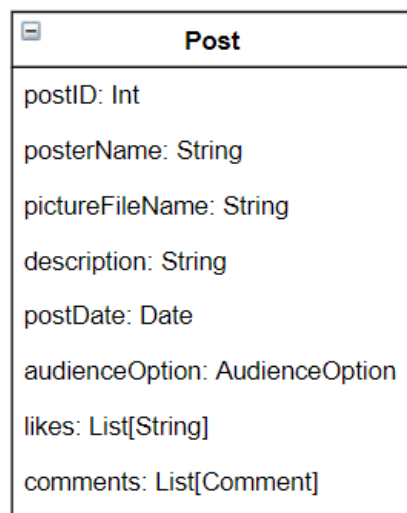


Figure 1.2. Model class Post

The **AudienceOption** class is used for selecting the audience that can view a Post. The option field represents the choice for making a Post Public (i.e. everyone can see it), Private (i.e. nobody but you can see it) or for a Selected Audience. In the latter case the audience field is filled with a List of userName Strings to represent the audience that was selected.

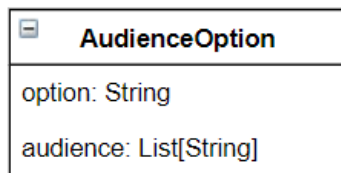


Figure 1.3. Model class AudienceOption

The **Comment** class represents a comment. The field commenterName references the userName of the commenter. Comments could have been stored with an incrementing ID as well, but this seemed redundant for our purposes so they were just stored directly under their related Post object.

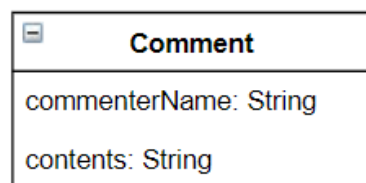


Figure 1.4. Model class Comment

DAO

Dao singletons are the main point of communication for the controllers with the model classes and their business logic.

UserDao is used to add and find users in the dummy database. Methods for manipulating a User return a Boolean representing success of the database operation.

PostDao is used for adding, updating, deleting and finding Posts. Database methods generally return a Boolean representing success of the operation, except when adding a Post. The latter returns a copy of the Post with a valid auto-generated ID, so this object can be used further if needed. PostDao also handles adding and finding comments. No additional Dao is made for handling Comments, since currently they are only used in the direct context of Posts.


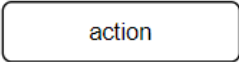

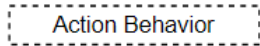

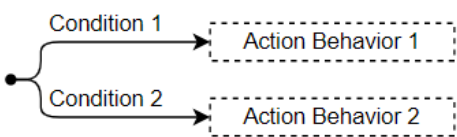
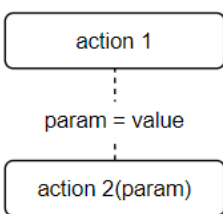
Global Objects

Global is an object encapsulating general keys that are used throughout the application. It has a key for accessing the currently logged in user and a key for displaying a query string message. The latter is used for showing messages after an AJAX call, since flashing is not supported in that case. A short message can be displayed on the target page by adding a query string to the URL with the aforementioned key, e.g. adding `?message="my%20message"` to a URL will cause "my message" to be displayed.

UserSession is mostly meant as a helper object that provides methods for easy access to user related session information, while **FileStorage** is a helper object that provides methods for easy access and manipulation of stored files.

3. Controllers

Below you can find the key used for all diagrams in this section about Controllers.

Key	
<div>Controller</div> 	Controller object
<div>action</div> 	"Regular" Action inside a Controller
<div>authenticatedUserAction</div> 	An AuthenticatedUserAction inside a Controller. This action requires the user to be logged in.
<div>Action Behavior</div> 	Description of the behavior resulting from invoking an Action from a Controller.
	Connection between an Action and its behavior or between two sequential behaviors.
	Conditional splitting point between two possible behaviors for an Action.
	An Action "action 1" which is simply an implementation that calls another Action "action 2" with a fixed parameter value.

Home

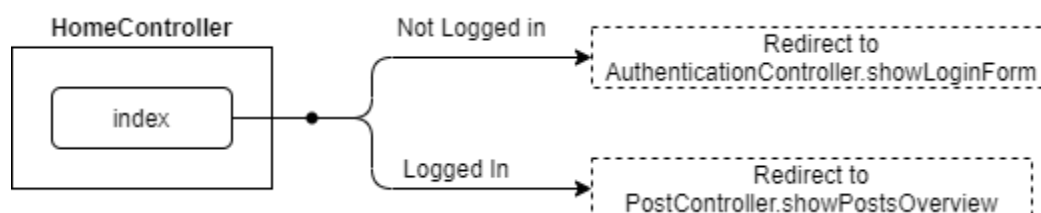


Figure 2.1. Diagram of HomeController

Unauthenticated users can visit the posts' exploration page, but no posts are visible until they log in. Therefore, a distinction was made for the home page as the login page is a better choice for them.

JavaScriptRoutes

A **JavaScriptRouteController** has been added to be able to easily perform controller actions via AJAX. This controller will expose routes to the specified controller actions, so JavaScript code can be generated to handle routing from client side back to the application. These routes are included in the main view template via a script tag with `src="@routes.JavaScriptRouteController.javascriptRoutes"`.

Although all required behavior could have been implemented without AJAX, reloading the entire page when performing small actions is not user-friendly. This is why simple operations such as liking a post, adding comments, sorting, etc. are performed via AJAX instead. The routes of all actions supporting AJAX have been exposed via this controller.

Authentication

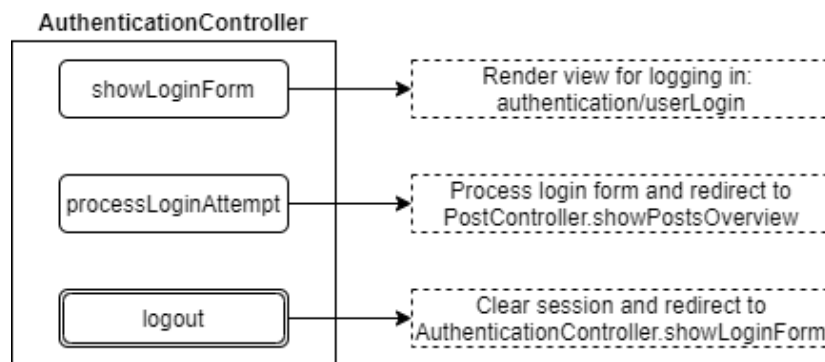


Figure 2.2. Diagram of AuthenticationController

A user is considered to be logged in if its `userName` is currently stored in the session. Therefore, logging in simply implies checking if a user exists in the database and storing it in the session while logging out can simply be done by creating a new session without a stored `userName`.

An **AuthenticatedUserAction** has also been developed which checks if the user is logged in before performing the “normal” action behavior to prevent certain actions of being performed by unauthenticated users. It will simply redirect unauthenticated users to the login page.

User related actions

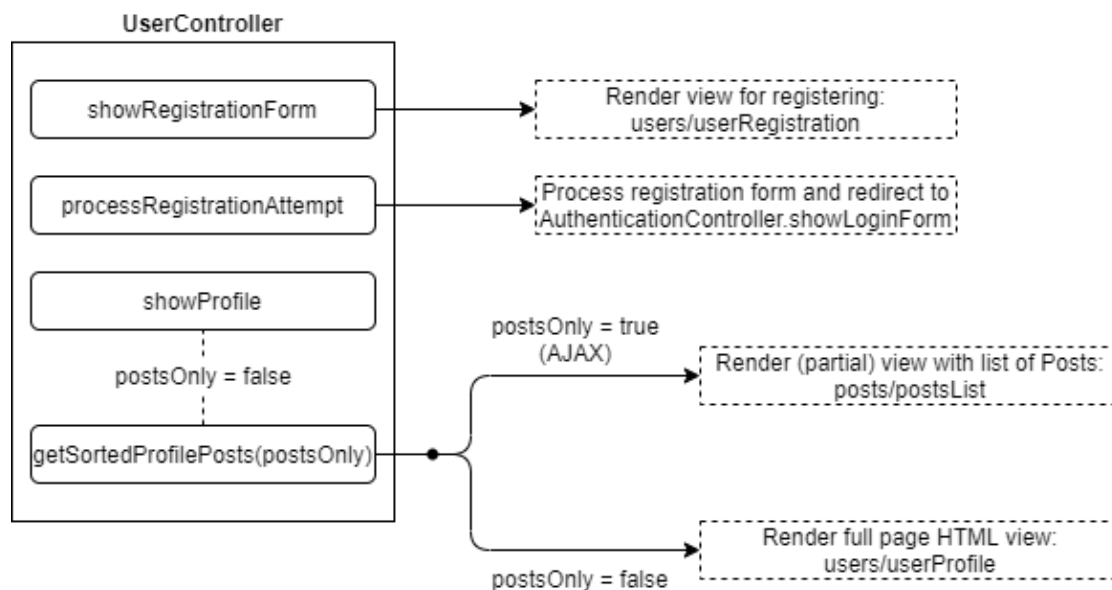


Figure 2.3. Diagram of UserController

A **UserController** takes care of all actions related to showing, creating or modifying user data.

Registration involves User creation, so the UserController handles it. Form validation includes username uniqueness and minimal password strength (uppercase, lowercase, numbers, length, etc.).

Showing the user profile happens via the `getSortedProfilePosts` action. This action returns a sorted, audience filtered overview of a user’s posts, but its behavior differs depending on a Boolean parameter `postsOnly`. If `postsOnly` is false, a view rendering a full HTML page is returned. If `postsOnly` is true, a view rendering only the list of HTML formatted posts is returned. The latter is meant to be used in AJAX calls to prevent full page reloads when merely changing the sorting option.

Currently, the user profile mostly shows posts and could therefore actually be moved to the PostController. However, for modifiability and extension purposes it seemed wiser to put it in the UserController so user information that is not post-related could straightforwardly be added later.

Post related actions

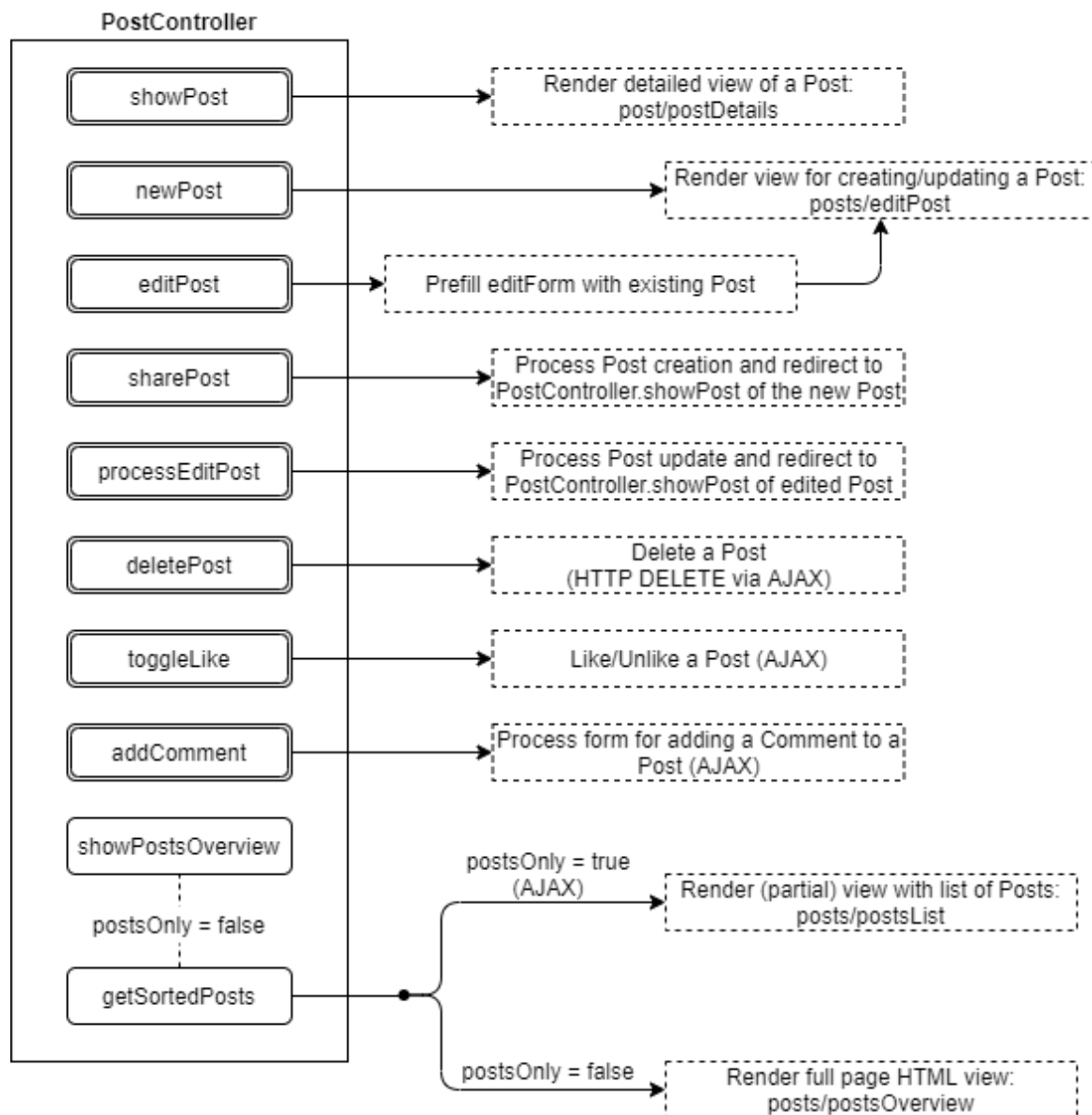


Figure 2.4. Diagram of PostController

A **PostController** takes care of all actions related to showing, creating, modifying or deleting Post data.

The same form and view template is used for post creation and editing, with the main difference being that the form is prefilled or not. A delete button is added (via a template parameter in the `editPost` view) in case we're dealing with an editing form. Modifying another User's Post is of course forbidden.

Creating or updating a Post also includes the upload and deletion of image files. These files are uploaded via multipart/form-data encoding and are validated to be of types jpeg, png or gif. We also check whether a file with the given name already exists before moving the upload to its destination directory. If the filename is already taken, we generate a new name to avoid unintentionally overwriting a previous file.

Before a Post is deleted via a button on the edit form, a confirmation modal is shown to prevent any unintentional removals. Preventing unintentional deletion is also the reason we did not put a quick deletion button on the user's profile itself. Such an action should not be too easy, since it is a rather invasive operation. Furthermore, the delete action is implemented as an AJAX call instead of a simple form submission, since we wanted to explicitly use the HTTP DELETE method and not an HTTP POST.

Toggling the likes (i.e. liking/unliking a post) and adding a comment to a post also happens via AJAX, since these are rather minor actions for which a full page reload seemed excessive.

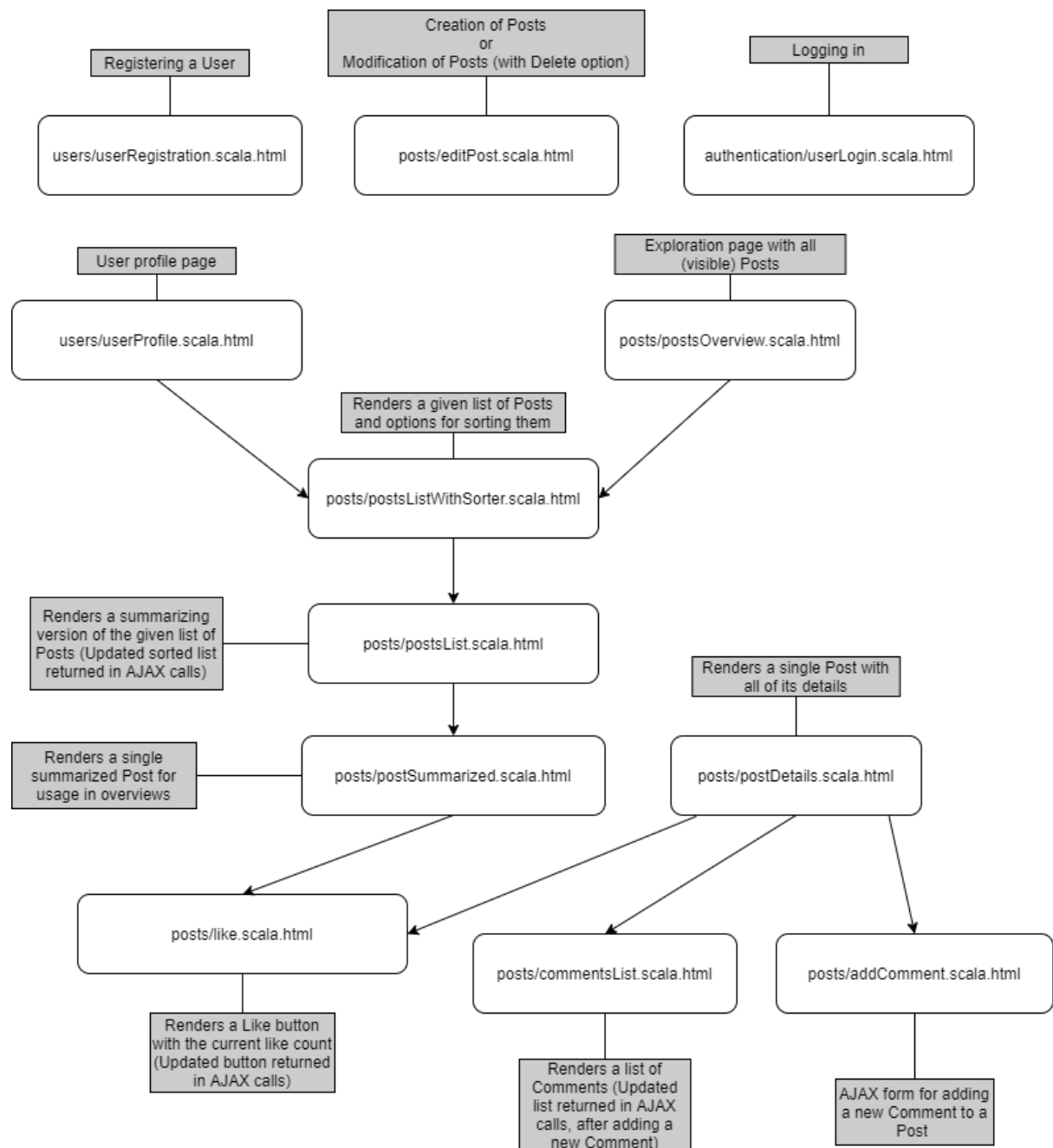
Showing an overview of all posts happens via the *getSortedPosts* action. This action supports both regular and AJAX calls, similarly to the *getSortedProfilePosts* action (but without filtering on the user). It should also be noted that liking a post in overview pages does not cause an immediate resorting, since it would be rather inconvenient for the user if the post jumps up or down when he (un)likes it.

4. Views, CSS and JavaScript

For creating responsive layouts we made use of Bootstrap 3. Furthermore, we used the Play Framework library for Bootstrap Forms of Adrian Hurt (Hurt, 2020) to easily generate Bootstrap compatible forms. Color use is rather limited -as I'm actually color blind-, but JavaScript is often included in various templates and forms to provide a better user experience.

A main view template was created that contains all necessary headers, navigation, our custom CSS (*css/main.css*) and JavaScript (*js/main.js*) files. Other views will call this main template with a parameter containing HTML content to be inserted. Additional JavaScript tags are also included in the main template from the calling templates via a *scripts* parameter.

All other templates have been made as modular, configurable and reusable as possible. Like buttons, comment lists, post lists, sorters, etc. are all separated into smaller templates which can then be used in the larger ones. Figure 4.1 shows some of these relations.



Key	
Comment	A comment providing additional information about the usage of a view.
View	A Scala Play view.
View A → View B	View A includes View B when being rendered.

Figure 4.1. Overview of the internal relations between views and their general usage

References

Hurt, A. (2020, April 25). Play-Bootstrap, a Play Framework library for Bootstrap.
<https://adrianhurt.github.io/play-bootstrap/?version=1.6.1-P28-B3>