Design Document - Wiki Studio

Wiki Studio provides a platform for people to produce short movies collaboratively. Let us describe a use case to demonstrate what this platform is all about.

Use Case

A user will be presented with a bunch of scripts to select from, where the scripts will be sorted by title. The user can select a script of his/her liking.

Figure 1: Use Case

A script typically comprises of 8 to 9 scenes (we focus more on short movies). Upon selecting a script, the user has an option to view existing videos for various scenes, uploaded by other users. The user can also view details such as dialogues, scene setting and character guidelines, of a specific scene. If a user has recorded a video clip for a specific scene, he/she can upload the video by selecting the scene and then specifying the path to the video's location in the local file system. There is an option to comment on the script as a whole. The use cases are depicted in Figure 1.

Architecture

Wiki Studio follows the client-server architecture. The client comprises of the browser, and the server hosts the actual website. The server logic is based on the Model View Controller design pattern. This is depicted in Figure 2.

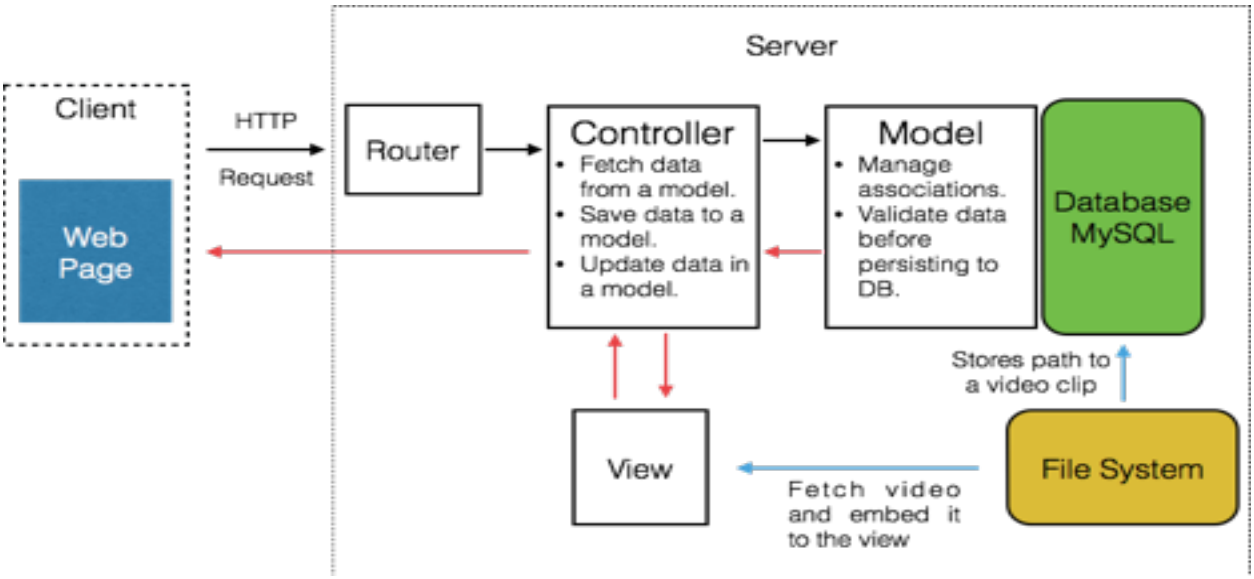

Figure 2: Architecture Diagram

The model is responsible for storing, updating, deleting and retrieving contents from the database. It also supports validation of data that is persisted to the database. When storing the video clips uploaded by a user, the actual clip is stored in the local file system(local to the server), and the path to the clip is stored in the database.
The view, as the name suggests, is the content that is presented to the user. It uses information generated by the model to render a HTML page, which is passed on to the client.
The controller is responsible for managing the flow of information between model and view, in the server. It is also responsible for servicing requests made by the client.
To understand the design better, let us describe the flow when a user wants to view a script. The client will initiate a HTTP GET request to the server. The router, which is the entry point to the server, will receive this request and evaluate the controller that will service it. This implies that there are several controllers, models and views. The association between these will be described later.
The controller will decide which of the several models has the information necessary to service the incoming request. It will request the models to provide the required information, manipulate it if necessary, and then pass it on to a specific view. In the use case described above, the controller would request for information like script name, script details like production notes and plots, number of scenes in the script, path to

video clips available for the script, user comments for the script, etc. This information will be passed on to the view, by the controller. The view will use this data to render a HTML page. For example, it will use the path of a video clip to fetch it from the file system and embed it on to the web page. The controller will then pass the HTML content to the client, encapsulated as a response.
This is how information flows between the client and server. Let us now move on to the various associations between models.

Associations

The website's data is encapsulated in four models. These are *script*, *scene*, *clip*, and *comment* models. The associations between each model is described in Figure 3. A *script* is associated to many *scene* and *comment* models. A *scene* is associated to many *clip* models.

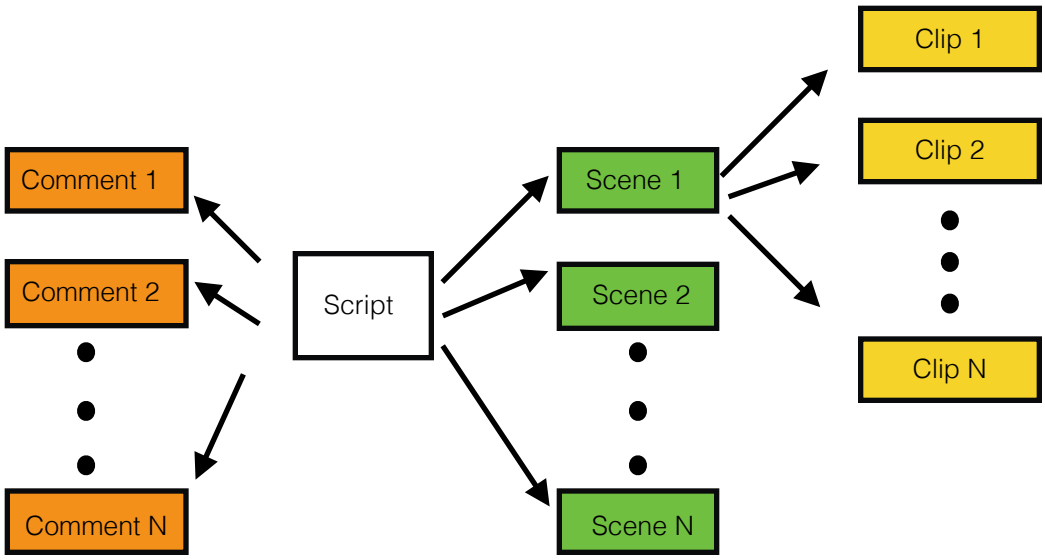

Figure 3: Associations between models

The *script* model holds all information about the script such as script title, plot, and production notes. It also holds reference to all the *comment* and *scene* models associated with it. Similarly, a *scene* model holds information such as scene number, scene setting, character guidelines and dialogues. It also holds reference to all the *clip* models and the single *scene* model associated with it. The *comment* model encapsulates comments and the commenter's name. It also stores the reference to the *script* model it is associated with.
Note that for each model, there exists a dedicated controller. Hence, there are a total of four controllers.

Folder Structure

This section discusses the structure in which the source code is organized. Majority of the source code is confined to the *app* folder. The *app* folder contains four major subfolders. The model source code is organized by model name in the *models* folder, the controller source code is organized in the *controllers* folder, and the views in the *views* folder. The *assets* folder consists of javascript and stylesheets associated with each view. This structure is depicted in Figure 4.

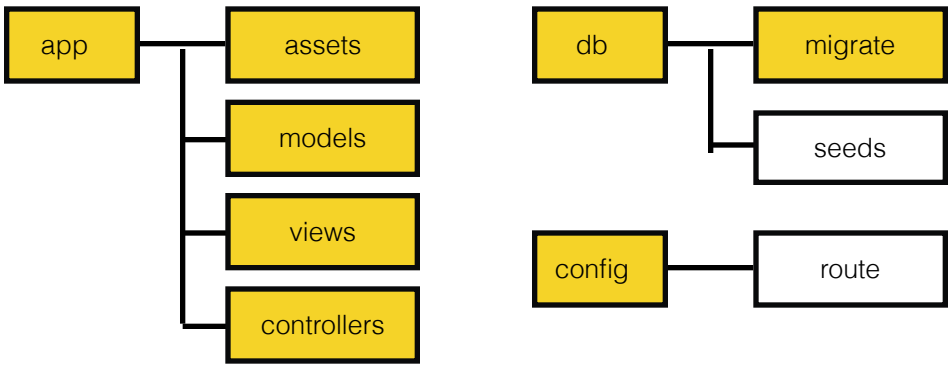

Figure 4: Folder Structure

The *db* folder consists of the migrations for each of the four models and seeds for each of the tables. To run a migration, navigate to the *bin* directory in terminal and execute the command *rake db:migrate*. To seed the database, execute the command *rake db:seed*.
The *config* folder contains the *route.rb* file which contains instructions about handling service requests. It specifies which controller should service a particular request.