Architecture Document

Introduction

The EPL Centennial is designed to have as decoupled as an architecture as possible. It is primarily split into 'back-end' and 'front-end' components both of which can be individually developed. Well defined APIs allow a simple way for the two components to be able to exchange data.

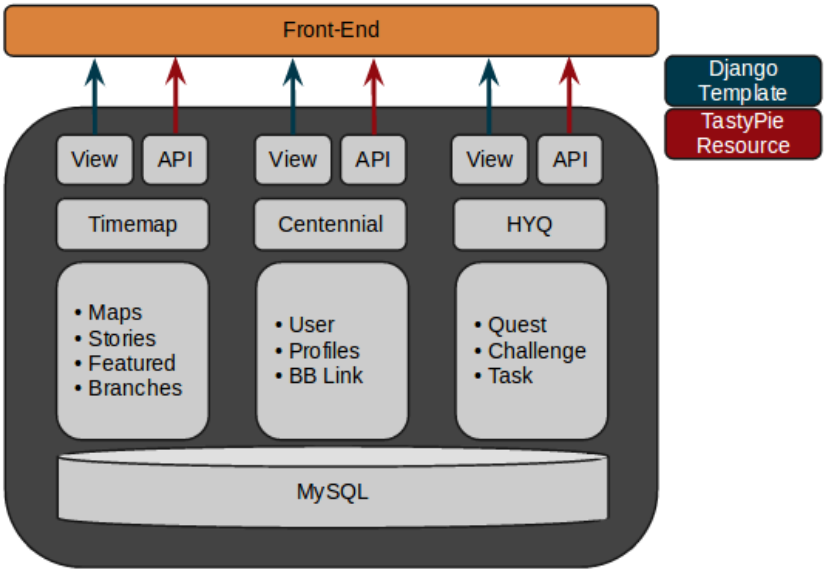
The back-end database is serviced with a MySQL database. Django, a python web development framework, is used to manage the data base as well as incoming and outgoing data requests. Various Django plugins can assist in managing these requests as well.

The front-end is presented to the user via HTML and JavaScript tools. This allows for an efficient way to present dynamically changing data in a clean way.

TimeMap data and HYQ data is transfered between the front-end and back-end via HTTP requests and responses. The data contaned within these responses is managed using the various tools within the back-end and front-end components.

Back-End

The Back-End architecture of the Centennial Celebration Timemap and Hundred Year Quest Game can be best described by the following diagram:



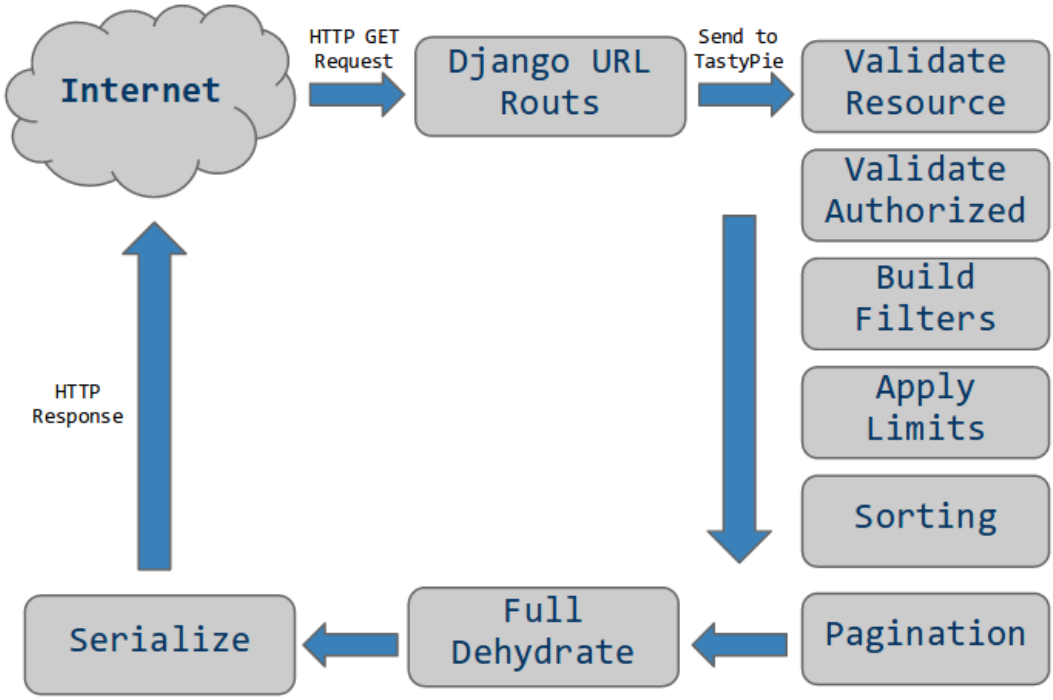
The system has been divided into three separate applications. First the Timemap where users can interact with SIMILE to explore the temporal information of the city of Edmonton. In this application the Stories, Maps, and Branches form the core of the data that needs to be modeled, stored and explosed.

On the other side of the architectural diagram the Hundred Year Quest game can be found. This application manages the Quests, Challenges, and Tasks that are proposed to the users.

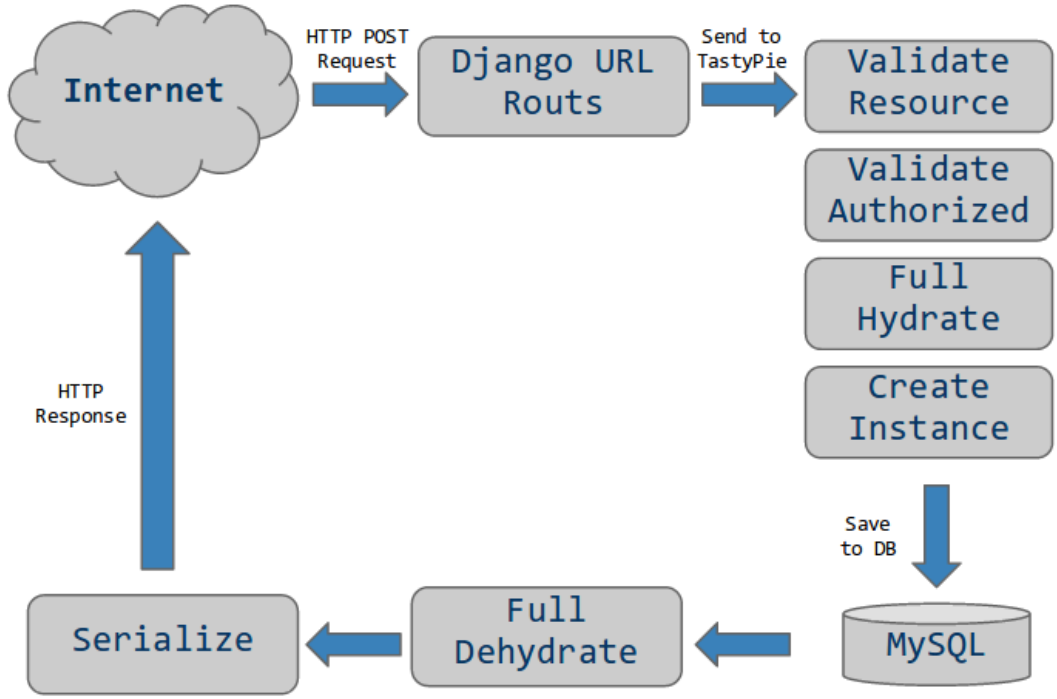
Both, the Timemap, and the HYQ, utilize the Centennial application. Here user information and their profiles is modeled. Information such as stories created and game points as well as quests completed is managed through this application.

All of this functionality is built on top of Django, a Python based web-development framework. Django has a huge community which allows us to ensure the framework will be supported for the concievable future. Besides the time guarantee, many applications are also available which allows us to quickly plug in functionality reducing development costs.

One of such applications is TastyPie which allowed us to expose all of our data through REST services. There are two main cycles that control how we respond to either *GET* or *POST* requests.



When we have a *GET* request Django will use it's routing to and forward the request to tastypie. Tastypie will then validate the resource requested exists, and then it will check the authorization and authentication of the user. If these two checks pass then the query filters are built, the query is performed and the results are sorted. Once the results are aquired pagination is done followed by a hydration cycle which transforms the data based on constraints on how it is to be exposed. This is then serialized into JSON and sent back as a response to the user.

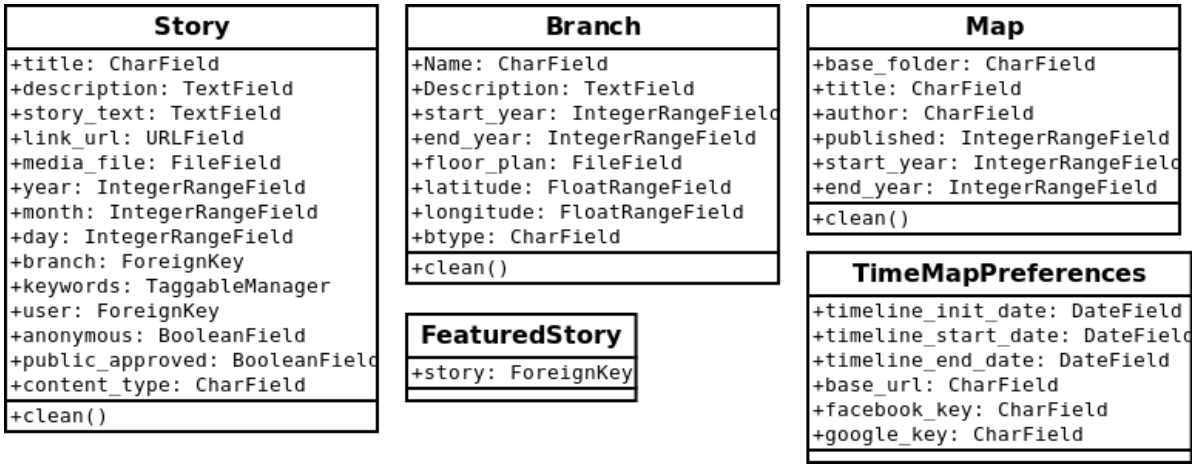


The cycle for a *POST* request, which will result in the addition of a new story is very similar. Once Tastypie has aquired the request it will validate the user is authenticated and authorized, then the data will be hydrated in order to make it fit with our models. Once the data is prepared an instance of the model is created and saved into the MySQL database. Then a response with the serialized and dehydrated data of the newly created resource is sent back as a reponse to the user.

Timemap Architecture

An overview of the Timemap Architecture has been presented in the previous sections. Our implementation follows the default Django application

skeleton and as such is very simple and maintainable. The following UML diagrams represent all of our models which handle our timemap data.



As you can see the models are also very simple as the data we manage is very well encapsulated and has no external depensencies. We have not built any stand alone classes as we were able to make everything fit within the provided Django framework. Django is renowned for it's clear and thorough documentation and we would be unable to do it justince in a document as short as this. The separation of the different applications as described previously makes it trivial for anyone to jump right in into this project.

Hundred Year Quest Architecture

System Design

The EPL Centennial Website is built using an AJAX Front End and the Django ORM overlaid on an SQL Server. Development was divided into Front End work and Back end work, which communicate over an agreed-upon API. For increased development speed, the Hundred Year Quest and TimeMap share bodies of code associated to Account management and creation.

Subsystems, Connectors, Relationships to Hardware

Backend

Django

The Django ORM acts as an interface between the database and the application logic. It also handles all dynamic page/api requests, and connects them to the appropriate view logic or API call.

Hundred Year Quest

The Hundred Year Quest runs all APIs powering the game, and maintains the Game state for all users. It is divided into the following submodules, which handle various aspects of the game's execution.

Models

```
hyquest/models.py
```

All database-backed objects are defined in the Models, which includes Questsand User Actions. All attributes of these objects are instantiated here.

Quest Manager

```
hyquest/questmanager.py
```

The Quest Manager handles which Users will see which Quests when. It is responsible for Quest Flow and updating of featured Quests.

Action Manager

```
hyquest/actionmanager.py
```

The Action Manager handles the backend logic of beginning and completing Quests for given users. It also performs ancillary tasks such as incrementing Users' points, and managing the redundant bookkeeping kept for speed.

Task Verifiers

```
hyquest/verifiers/*
```

The Task verifiers handle the actual checks performed to detect if a user has completed a Quest. Verifiers are split up by the type of Task they verify.

Admin Forms

```
hyquest/taskeditors/*
hyquest/admin.py
```

The Admin Forms handle the creation of the custom forms used to alter the settings of the various types of Tasks. They also handle the customizations which simplify filtering and altering quests.

Views

```
hyquest/views.py
```

The views handle all non-administrative, non-trivial data queries. For example, it exposes APIs for quest completion. It performs data sanitation, before handing off the calls to the responsible module.

API

```
hyquest/api.py
```

The API module configures TastyPie to expose the Quests over a REST API, to allow easy retrieval by the front end.

Centennial

The Centennial module handles common functionality relating to Bibliocommons and User Management

Models

```
centennial/models.py
```

Bibliocommons Interface

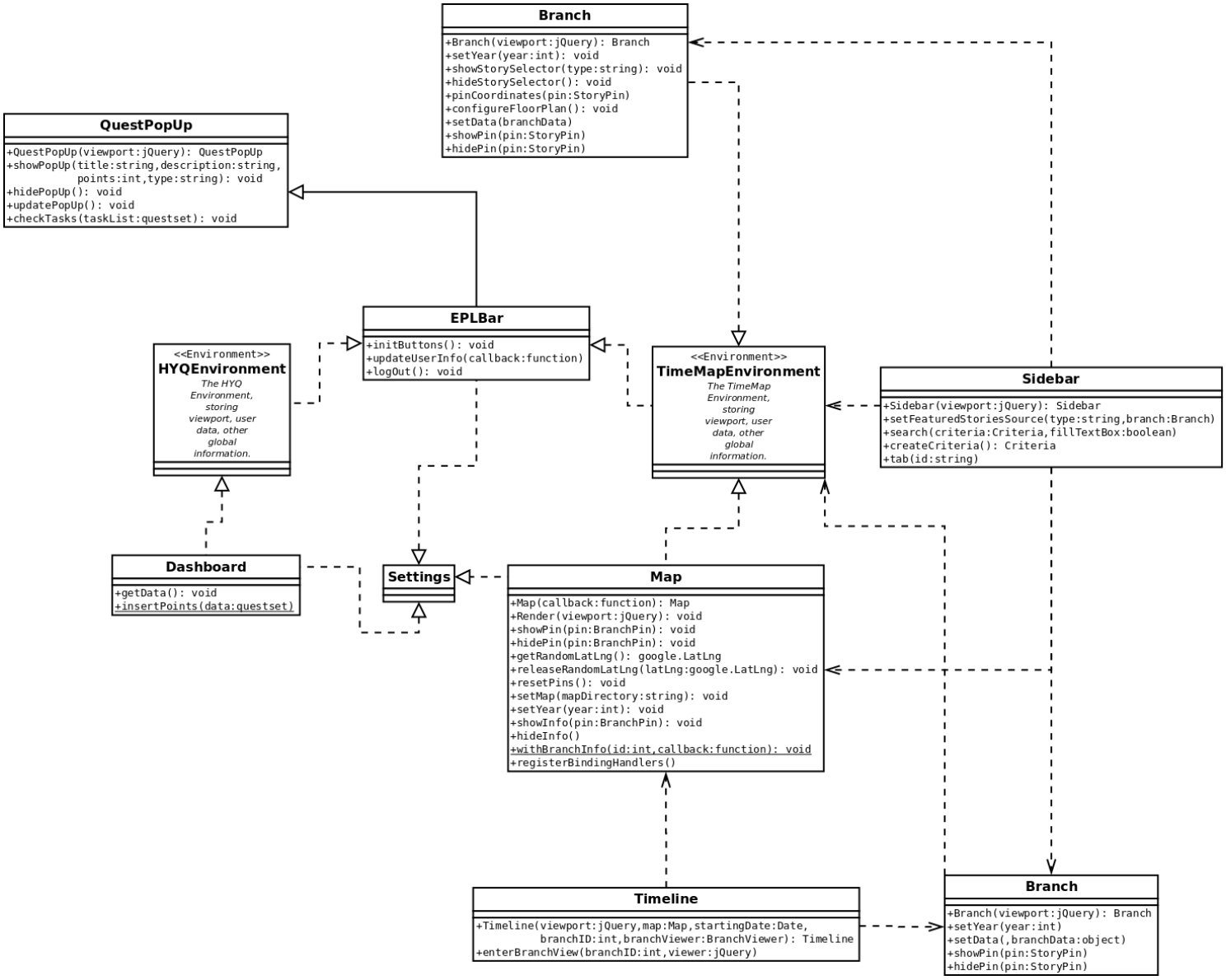
```
centennial/biblioauth.py
centennial/bibliocommons.py
```

The Bibliocommons interface interacts with the Bibliocommons API to allow for user authentication, and to retrieve user content.

Emailer

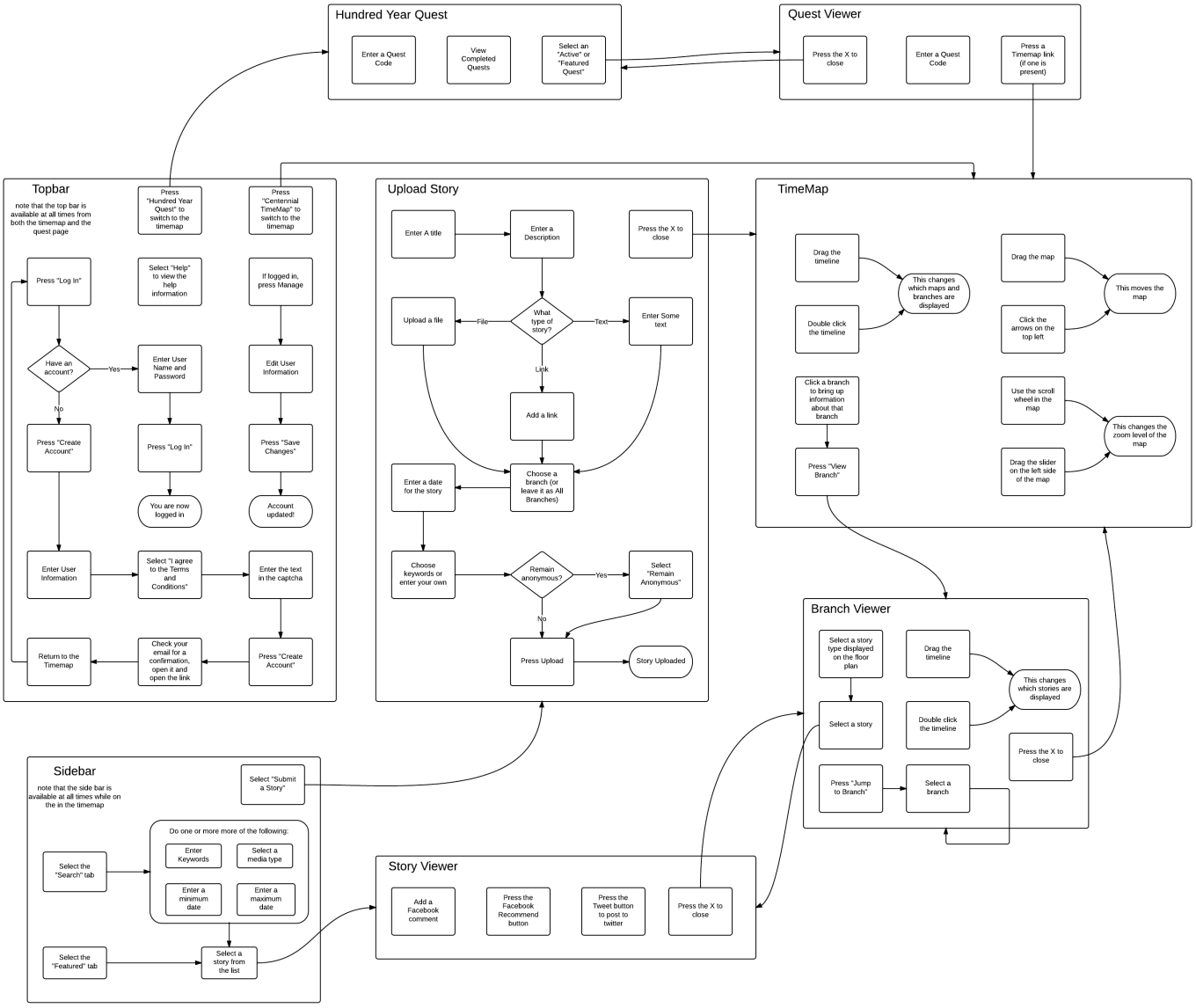
The emailer is built using python's builtin email modules. The e-mail modules simply establish a connection to a desired SMTP server and send an e-mail message to that server.

Logical Model

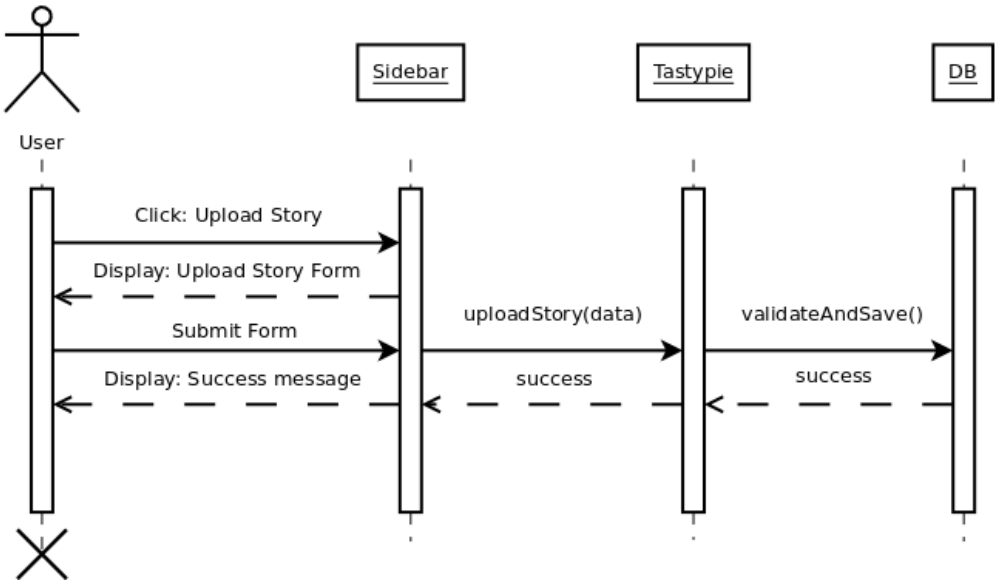


Behavioural Model

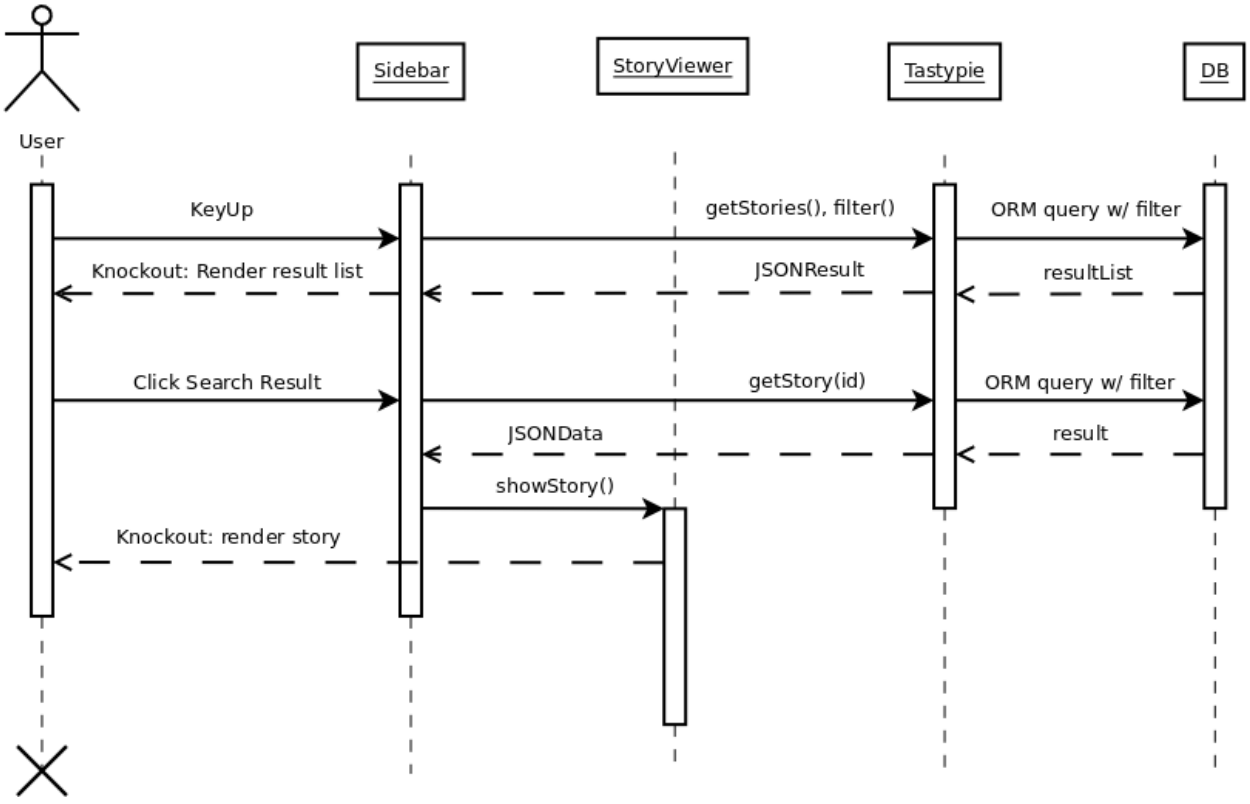
System Overview



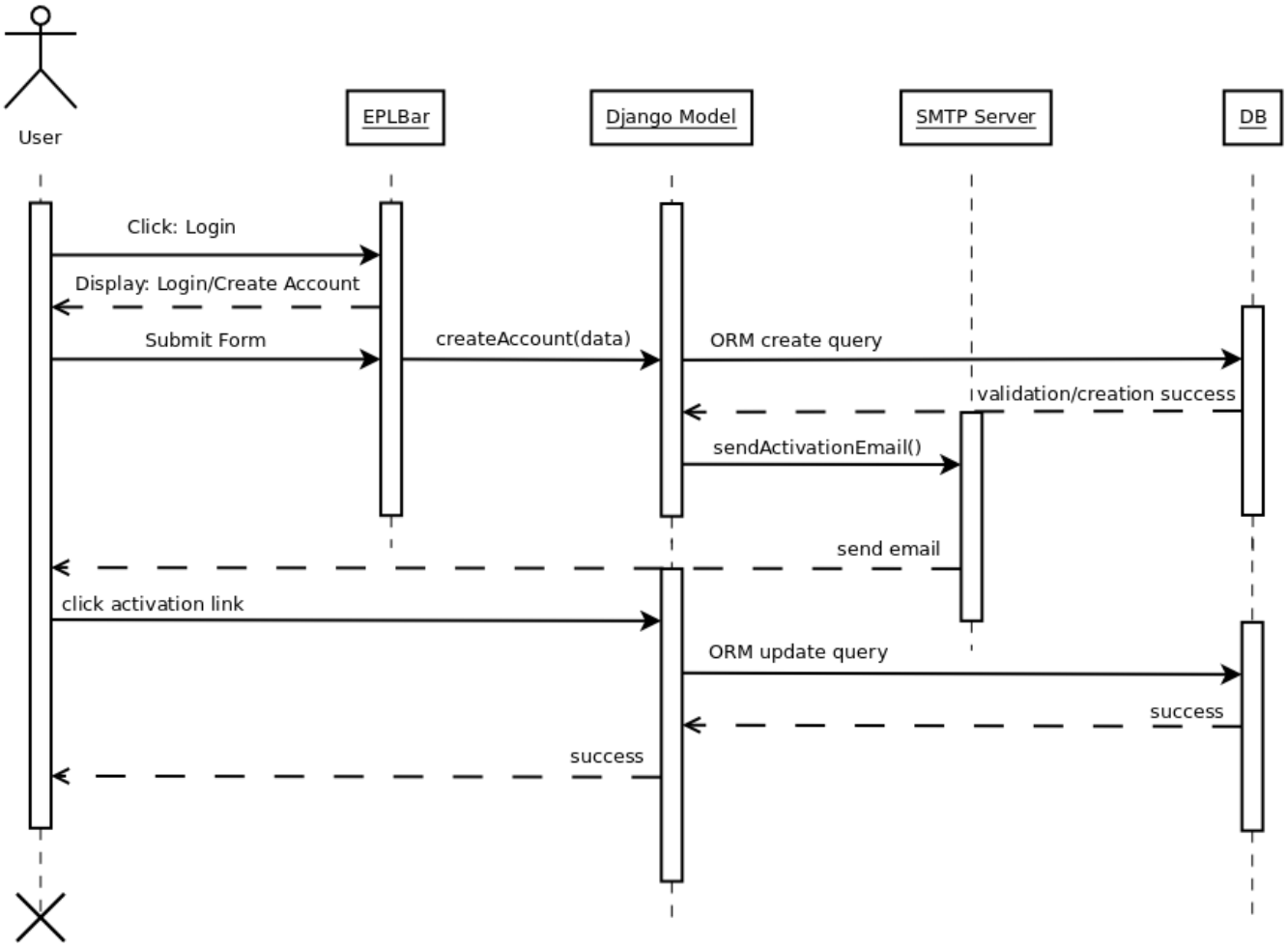
Uploading a Story



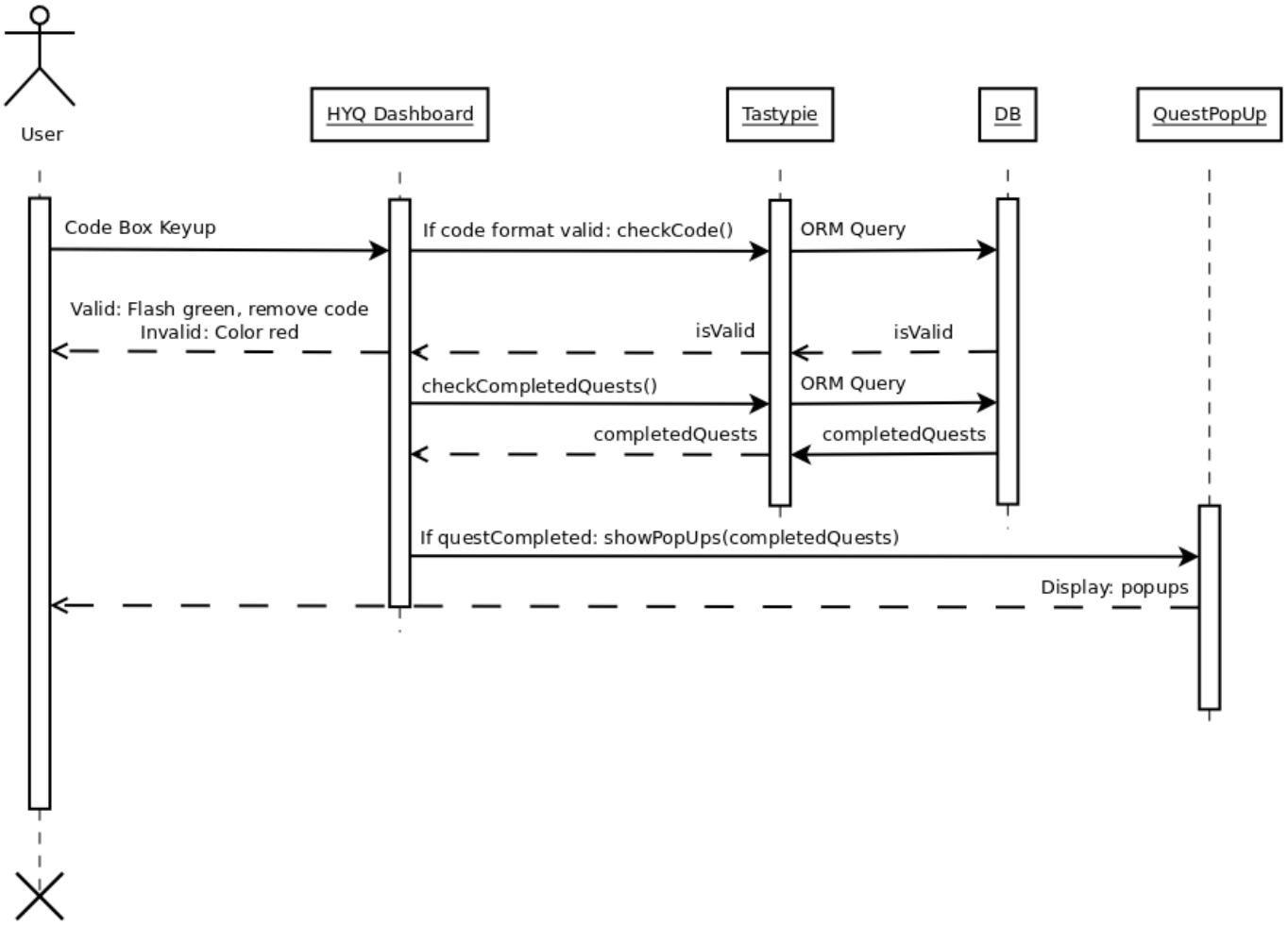
Searching for a Story



Creating an Account



Checking a Quest Code



Navigating Historical Maps