Codeline

Code sharing platform

# Analysis

* Specifically for school, dept
  + Growing CS dept, introducing CS at GCSE and A level
* Problem: accessing, keeping track of, storing (centralized), finding, checking and marking students’ code
* Process/stages of development
  + Iterative process with feedback from MWD, other students
  + User testing, amendments
  + Diagram?
  + Deployment
* What I did to find out more about problem
  + Interviews, talking to students and teachers
* Current system – putting code in collaborative area (Y drive)
  + Time consuming
* List of client requirements plus possible limitations (e.g. time, hard work for little gain)
  + Centralized storage of students’ code
  + Digital and internet based
  + Allow client to see a student’s history of submitted code to monitor progress
  + Students can add narrative with their submissions (outside of code comments)
  + View student’s posts
  + Ability to tag posts and filter by tags
  + Syntax highlighting (at least Python and C)
  + Date present on all posts
  + Can submit anonymously
* List of objectives that are realistic, achievable and measurable
* DON’T MENTION MARKING. FOCUS ON MONITORING

## Background

Dubai College’s Computer Science Department recently started offering Computer Science at GCSE and A-Level. With the expanding department, and a lot of growing interest in the subject, the teachers of the subject are increasingly pressured by various different responsibilities, including keeping lessons engaging and students interested; developing projects and problem sets to teach in ways other than textbook and PowerPoint; and consistently reading students’ code to monitor their progress. In the experience of Mr. Mark Wood, the head of the CS department at DC, the third point is often defaulted on, as a lot of energy goes into the other parts of teaching and not much is left for code review.

As a result, Mr. Wood and his colleagues find it difficult to keep track of their students’ progress, provide feedback on code written by students and collate code written by students iin and out of lessons.

Of course, tests can be used to gauge progress, but specifically for CS, it would be beneficial for the teachers to see the code that students are writing outside of test/exam conditions—programming is, after all, a large part of the subject.

I had a lengthy discussion with Mr. Wood that illuminated many of the problems the department is currently facing in regards to student monitoring. He told me that he genuinely wants to read students’ code, as he often gives programming tasks to the class for which he would like to observe each individual students’ approach. Looking through the code of each student would give a better understanding of each student’s progress, and insight into the effectiveness of certain projects or teaching styles for the cohort as a whole. Unfortunately, he explained, this observation often doesn’t happen as it is too time consuming. There are a few approaches he has tried, but each is suboptimal and fails in certain ways to match his workflow.

The first option is to get students to print out all of their code on paper and hand it in, much like perhaps a History essay would be presented. This works in getting all the code in one place, however is not optimal for a CS class which is trying to move towards a fully digital atmosphere (and, of course, minimize deforestation). Mr. Wood also expressed that he would much prefer digital submission as it is easy for students to lose track of old work in physical form. A digital submission method would allow both students and teachers to find work from any point in the year at any time, and access it both at home and school without needing to carry heavy physical books or folders.

Another option that was explored was having students put their work on their school network drives. This does allow students to keep all their work in one place, and allows teachers to access work for monitoring without need for physical paper, however is still significantly inconvenient as the teacher has to individually access each student area and locate the relevant file manually. This can be especially taxing when some students miss deadlines and so teachers spend time looking for work that isn’t there. The difficulty of accessing work that is not in a centralized location is especially visible with this approach.

The obvious solution to this is to ask students to submit their work to a shared area, perhaps on the school network or a cloud file sharing service such as Google Drive. Again, this allows students to keep their code in one place, and access it anywhere, with the addition of allowing the teacher to access all the code from one folder. There are still problems, however. First of all, students can access, modify and delete each other’s work, which can especially cause problems for younger and less mature classes. Additionally, the teacher still has to physically open each file individually, which can become rather tedious, especially for the small code snippets middle/high school students often end up writing.

Also, neither of the digital solutions make it easy for students to offer narrative with their code—teachers like to see students explaining their code to demonstrate an understanding of concepts, and while this is possible through the use of large inline comments in the code itself, or through screenshotting and word processing the code, neither of these methods are ideal.

Through my extensive research, which included one-to-one discussions, group discussions, surveys and shadowing I can conclude that there is a substantial problem within the Computer Science Department at Dubai College, which I feel I can help provide a solution to.

My initial thoughts are that an ideal solution would allow students to post their code, with narrative, in a centralized location for later access, and allow the teacher to view all of the relevant code in a single location (without any intermediate navigation of the filesystem for each student). Teachers will also be able to provide some form of feedback to the student via this solution.

I have decided to have another one-to-one meeting with Mr. Wood so that I can share with him my findings, my initial thoughts on a proposed solution (overview only, no detail) and to then firm together some agreed requirements, which I will then extend with specific objectives.

During my final meeting with Mr. Wood (in the analysis phase), we discussed at length the research I had done, the observations I had made and the we touched briefly on my proposed solution. However, I felt it necessary to let him (my end user) draw together his own requirements.

During our meeting I wrote down some key bullet points of the points/requirements he was making whilst we were talking. These are a copy of the bullet points I made:

## Requirements Discussion

* Web based digital solution
* Centralized storage of students’ code—ability to access after posting
* Allow teacher and students to see all of a student’s submissions chronologically
* Date present on any work posted
* Allow students to add narrative to their code (as well as being able to add comments to their code)
* Ability to tag any code, a student or teacher posts with multiple topics, and filter submissions by these tags
* Syntax highlighting for Python and C specifically
* Allow teachers and students to be able to offer feedback/comments to any code submitted
* Although the above was specifically stated Mr. Wood would like an alternative to simply commenting on the code i.e. he might want to correct some of their code, with comments, or provide alternative solutions with comments, it should not just be a 'comment' as used by most blogging and social media platforms.

As mentioned, the bullet points above were recorded directly by me during my meeting with Mr. Wood. I shared them with him after our meeting and asked if he felt these brief comments summarized his requirements for the proposed solution.

He was very clear in agreeing that they did. However, I have rewritten them below to be more concise where necessary or more detailed when appropriate:

## End User Requirements

My end user requires a solution that meets the following criteria:

* The solution must be a web based digital platform
* The solution should provide a centralized storage of students’ code
* Both teachers and students should be able to access any code or markup after posting
* The solution should allow teachers and students to see all student’s submissions chronologically
* The date a student posts code or markup to the portal must be present on the submission
* The solution should allow students to add narrative to their code (other than code comments)
* Anyone who posts to the platform must be able to tag posts with multiple topics or keywords
* All posts submitted to the portal can be filtered by one or more tags
* All posts submitted to the portal can be filtered by author
* All posts submitted to the portal can be filtered by title of the post
* The platform must provide Syntax highlighting for any code posted but specifically for Python and C programming languages
* Teachers and students should be able to provide some form of feedback or suggested improvements to the code written

The requirements outlined above describe a solution that would meet all of the needs of Mr. Wood’s department. The web based, centralized nature would ensure permanent and constant access to code submissions by both the teacher and the student from anywhere. Chronological sorting as well as dated submissions allow the teacher to look through a student’s history of submissions to understand their progression throughout the course, while filtering by topic tags allows a teacher to see how the entire class is doing with a certain topic. Syntax highlighting is of course necessary for user experience—the purpose of the entire solution is to improve the experience of reading students’ code, after all.

## Approach to project

The project development process will consist mostly of an iterative approach. I will build part of the platform, and bring it to the client to demonstrate. Any feedback the client has will be taken on board and the project will be amended as necessary. I will also look to my fellow computer science students for suggestions, regarding both the visual design and functionality of the website. This feedback will be immensely valuable, as these people will eventually be the users of the platform. Approaching it this way not only helps eliminate possible fatal design flaws that I overlook in development, but ensures the end product is desirable and usable by those who are going to use it.

I plan to investigate web application frameworks that will support the proposed solution. I have experimented with php and MySQL in the past, as well as basic HTML and CSS but feel that there might be alternative technologies that can be used.

I plan to develop an open source project where all my code will be hosted on GitHub so that anyone who wishes to contribute to the project, once it gains momentum, can do so. **I MUST MAKE IT CLEAR that during my completion of this course I will only use GitHub as a source of feedback from professionals and not as a contribution platform i.e. all of the work will be mine and no one else's**. However, once the project has been deployed, Mr. Wood sees this as a solution he plans to use with every student at the school next year and so having the GitHub community share and collaborate on this project with me will help ensure that this solution is sustainable, as well as flexible in the need for further scope to be suggested by others.

I can also use the GitHub community as a source during the testing phase, along with students and teachers at my school. This will ensure a wide variety of audiences are able to contribute to the success of the project.

# Documented Design

* Explanation of Django, MVC system
  + URLs and views
  + Templates
* More detail about database
  + What a post looks like
  + Diagram showing post’s anatomy
  + How posts are read and rendered
* Detail about libraries: Ace, highlight, handlebars
* Screenshots/diagrams of basic UI
* Exploration of logic behind some views

## High Level Design and User Experience

The platform will be a web application. This quickly checks the requirement of being web based and digital—the web application, hosted on the internet, will be accessible to teachers and students both from home and school at any time.

When the user opens website, they will be presented with a collection of posts organized chronologically which can be scrolled infinitely (until all the posts in the database are rendered on the page). The posts consist of cells, each containing either syntax highlighted code or plaintext (for narrative). The posts are also tagged, with tags specifying for example languages used, topics covered or techniques employed. Posts can further be filtered by tag or by the user that submitted them, as per the client requirements. This can be done through the left sidebar, but also by clicking on a username or a tag. The left sidebar will also contain links to log in or register to the website. Any visitor can create a post, whether logged in or not; only logged in posts can be filtered by submitter, however. All posts are publically viewable. The below diagram shows a rough representation of how the application’s front page will look.

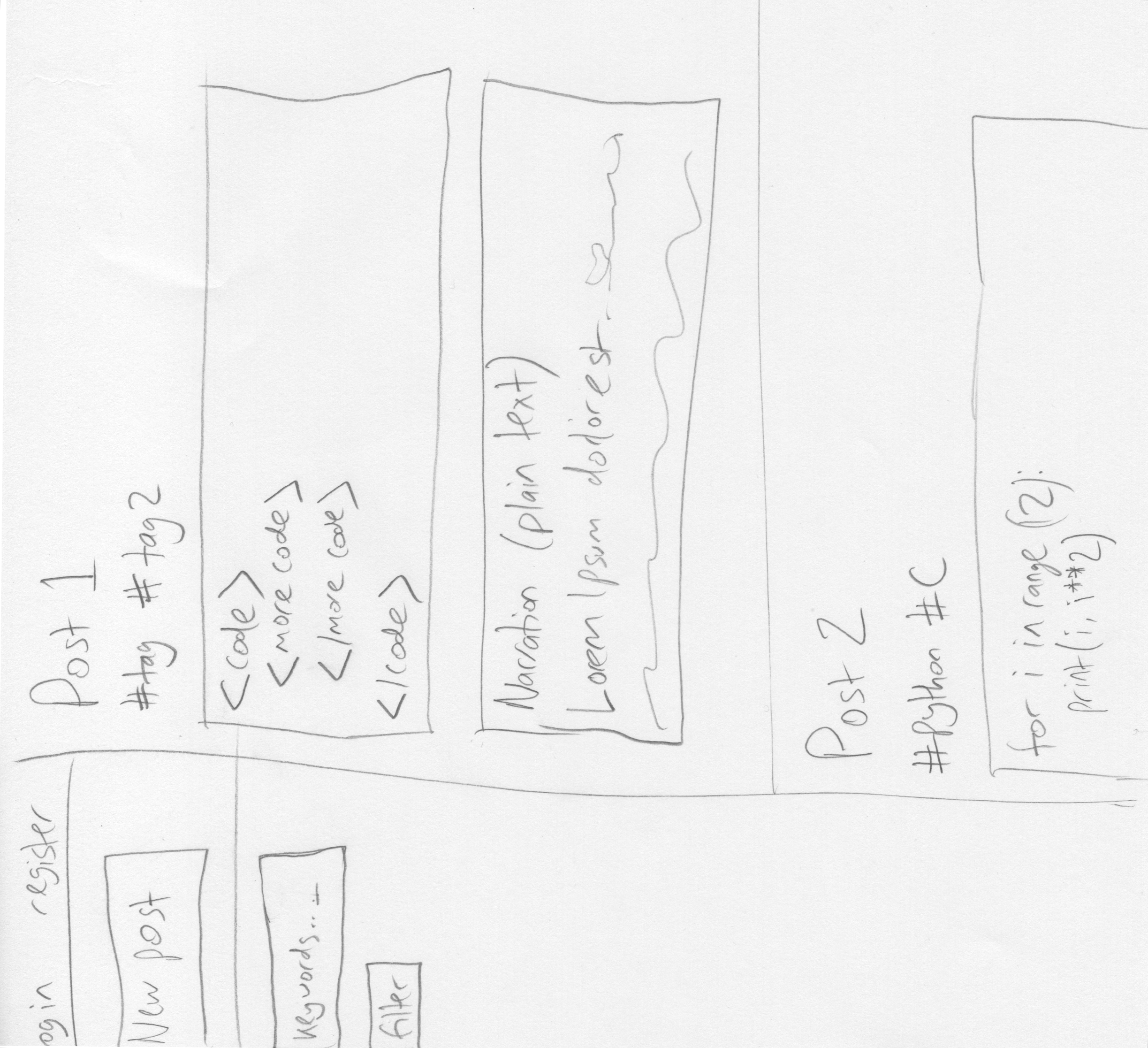


Figure – rough design of application front page interface

There will be a separate interface for submitting a post, which will allow a user to enter a post title, tags, and then provide a syntax highlighted text editor for each of the cells in the post. Cells can be added and removed as the user wishes.

## Technologies

The web application itself will be built using Django. Django is a web framework, written in Python, that operates in a manner similar to the traditional model-view-controller (MVC) architectural pattern but different enough to potentially cause confusion.

A Django application interfaces with a relational database, and entries in a database are described and interacted with using Python’s object orientation. This is an abstraction of the actual database engine used—the specific database engine doesn’t matter. A class is written to describe a database object, referred to as a *model*. The benefit of doing this is it allows relational aspects of a database, for example a post containing a field for post author, to be handled in a much more programmer-friendly way—the model instances (or database entries) being treated as objects means that foreign key fields—fields that reference an entry in another table in the database—can be treated as nested objects. This object oriented approach makes working with complex data models much easier.

The other main component of a Django application is its *views*, which are functions that are called when a request is sent by a client. Specific URLs are considered to call specific views The view accesses any relevant model and does any relevant processing. This includes, for example, saving a post to the database when a client submits one.

When a view wants to render some data for the user of the application, it does this through a *template*. The templating engine built into Django takes some context data assembled by a view and turns it into HTML with the relevant information for the user. Templates consist of HTML files with some special syntax that gets interpreted by the templating engine to fill the template with the correct content based on the context data sent by the view. Essentially, templates determine *how* the data is presented to users, and views determine *what* data to present.

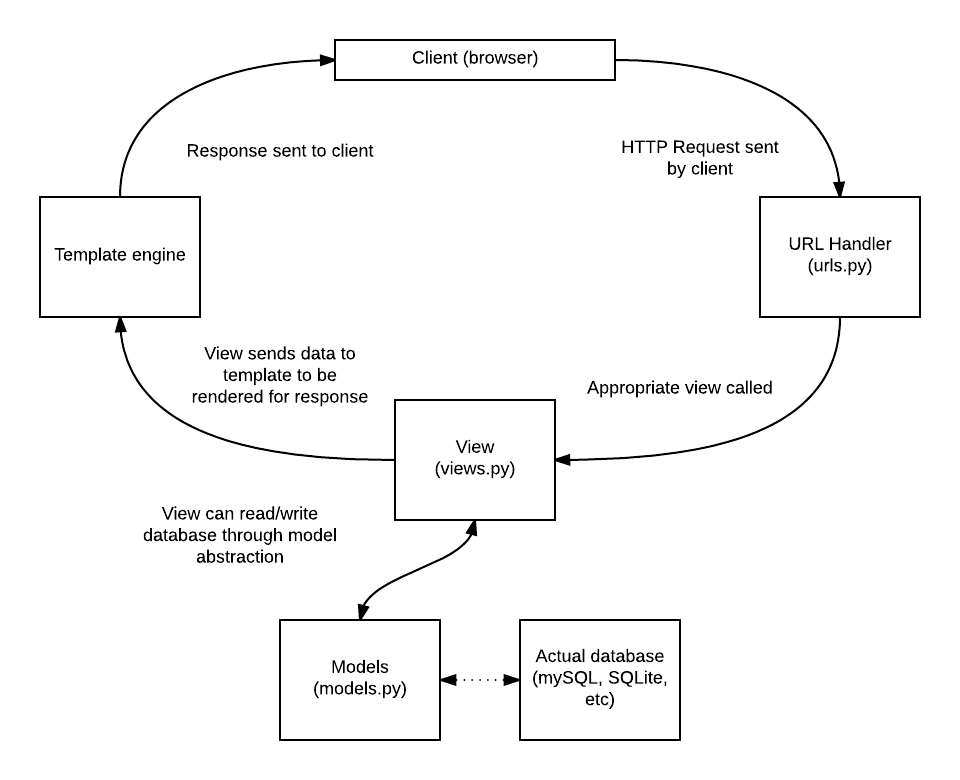


Figure – How Django renders a webpage

The specific functionality of Django will be covered in more depth in the context of this project further on in this document.

Several other libraries will be needed, however, to make the webapp functional. A library called highlightjs will be required to render syntax highlighting of code. Cloud 9’s Ace text editor, which is a lightweight javascript based text editor with syntax highlighting, will be embedded in the post creation interface. Bootstrap, a library containing several stylesheets, will be used to help with the style of the app and JQuery, a javascript library that makes manipulating HTML documents in the browser much simpler, will be used to help create responsive webpages and simplify much of the javascript required for the post submission interface.

## Models

The foundation of this web application will be its models—the objects that we want to store in the database. Usage of the application consists of manipulating these objects: viewing posts, submitting posts, filtering posts, and just about any dynamic functionality will require reading or writing to a database. Django provides some very useful abstractions in this regard, where the developer need only describe their models in terms of Python classes. This grossly simplifies relational matters, for example many-to-many relationships that require a link table in a database. Django does this all automatically and behind the scenes: the many-to-many relationship is simply treated as a Python set which is an attribute of the object with the relationship.

This application hinges on the Post model. Posts will have to contain a title, a date, and the body of a post, but also information about the tags it is tagged with and the user that submitted it. There will also be a date field for last edited, and a Boolean field describing whether the post has been edited since its creation. Posts will also have a foreign key relationship to their parent post if they are forked from another post. Tags will be a separate model, with each post having a many-to-many relationship with a number of tags. The author field of a post will be a foreign key relationship to the User model, which comes built in with Django. Below is the actual class definition for Post.

class Post(models.Model):

    title = models.CharField(max\_length=140)

    body = models.TextField()

    date = models.DateTimeField()

    last\_updated = models.DateTimeField()

    edited = models.BooleanField(default=False)

    author = models.ForeignKey(User, on\_delete=models.SET\_NULL, null=True)

    parent = models.ForeignKey('Post', null=True, on\_delete=models.SET\_NULL)

    tags = models.ManyToManyField('Tag')

    private = models.BooleanField(default=False)

    def \_\_str\_\_(self):

        return self.title

Note that *Post* is a subclass of models.Model, which is Django’s base model class. A field for private has been included for the potential future ability to create private posts. The author field has not been required (null=true) and if the user associated with a post is deleted, the user field gets set to null. As visible here, the creation of relationships in the database is much simplified by Django’s model system.

Here is the class definition for *Tag*:

class Tag(models.Model):

    name = models.CharField(max\_length = 140)

    lang = models.BooleanField(default = 0)

    def \_\_str\_\_(self):

        return self.name

The \_\_str\_\_ methods simply determine what happens when an object is cast to a string as a whole—changing this behavior is useful for debugging when using *print*.

Below is an ER diagram for the entire application, showing more clearly the relationships of the different models.

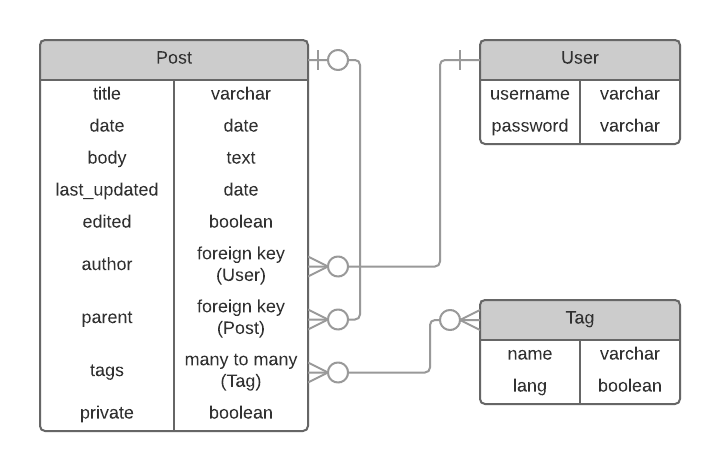


Figure - ERD for Post, User and Tag

Note that Django’s built in User model contains several other fields, but these will not be used in the application and so have been left out. Additionally, passwords are not stored as plain text, but as hashes—the field password doesn’t actually contain users’ passwords.

## More on Post

Post bodies can contain any number of cells, each requiring some metadata about the type of cell (code, text or markdown) and the language used (if the cell contains code). As such, storing the body of a post is not as simple as storing plaintext. Instead, information about the cells will be serialized as JSON and stored in the body field of each post. An example JSON post body will look like this:

{

  "cells": [

    {

      "lang": "c",

      "type": 2,

      "content": "int main(void){\n    for(int i = 0; i<12; i++){\n        printf(\"this code literally cannot run because i havent imported stdio.h\");\n    }\n    return 0;\n}",

      "title": "cell 0"

    },

    {

      "lang": null,

      "type": 1,

      "content": "## some markdown\n\* just\n\* for\n\* bants",

      "title": "md"

    },

    {

      "lang": null,

      "type": 0,

      "content": "this is just PLAIN TEXT",

      "title": "cell infinity"

    }

  ]

}

Note that the indentation will not be necessary to store as this JSON will be processed by the views before being displayed to the client.

## Views

Django views are functions that are called when a request comes in, and are responsible for returning a response. An HTTP response typically contains HTML for the browser to render, but it is also possible to return other data formats such as JSON.

The application will consist of one main front page view which shows all the posts in the database chronologically. This page will frequently need to ask the server for more posts or for a new list of posts when the user scrolls down or filters respectively. It can do this by sending requests to the server, much like a browser would when a URL is typed in, but instead of rendering the response in the window, the javascript controlling the webpage can process the data in the HTTP response sent by the server, and update the information on the page. This type of request is an AJAX (Asynchronous Javascript and XML) request. This means that the view that it accesses in the Django application will need to be able to return a response in some kind of serialized format so that the javascript controlling the live front page view can process the posts and modify the HTML in the browser window as necessary. The preferred format is JSON as it is much more suitable for data serialization than the XML for which AJAX is named.

However, the application will have distinct pages for other things, for example a page for an individual post, a page listing the posts of a single user, and a page showing all the forks of a given post. These pages will not need infinite scroll or live filtering, and so for these pages the data will be rendered as HTML and the response will be in HTML format, for the browser to directly display to the user (with no intermediary javascript controlling the page). So there are two different situations, outlined in the diagram following.

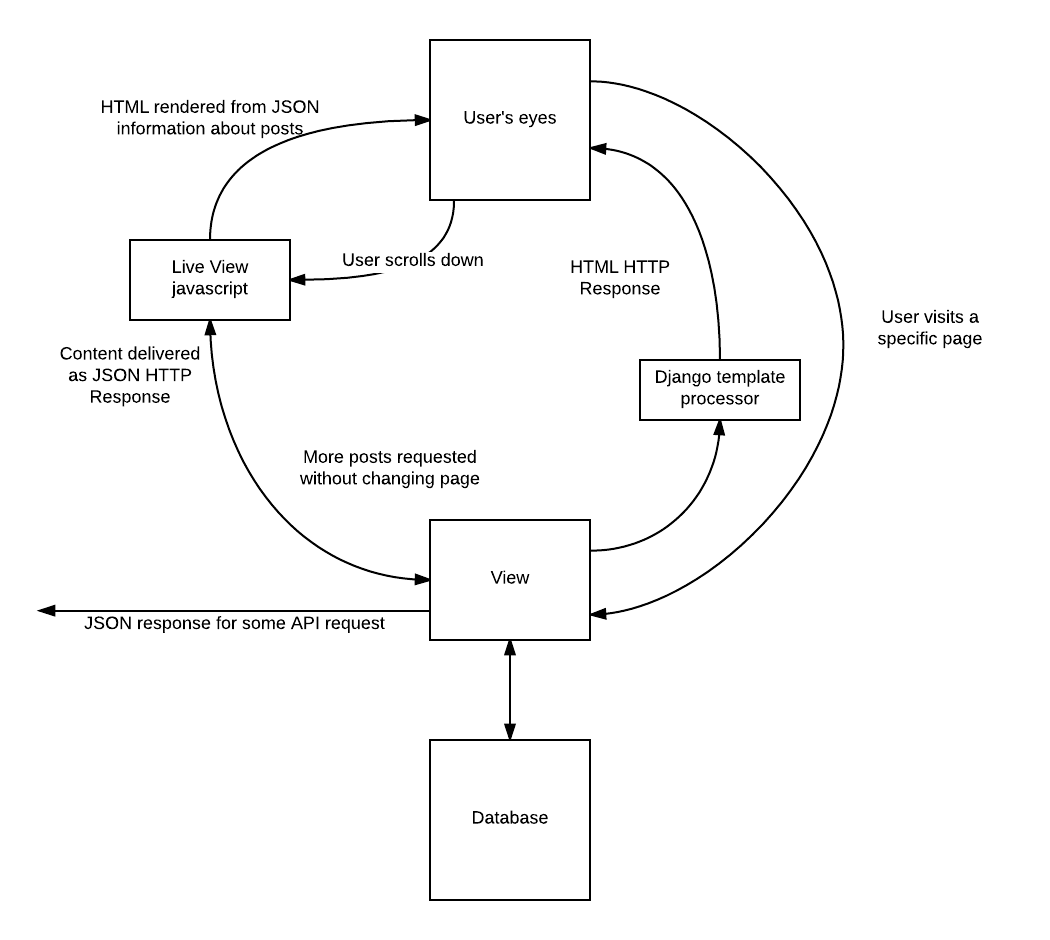


Figure - View structure

The diagram shows how views would return a JSON response, on the left, so that the javascript on the live view can control the addition of new posts to the infinite scroll, or the filtering of the visible posts. On the right, however, the view is returning an HTML response for direct rendering by the browser.

It would be beneficial if every view that displays posts could offer its output both as HTML and as JSON. The latter is not only useful for live views within the website, but also creates an API that can be used by other applications to retrieve posts (paving the way for a potential a mobile app for the platform in the future).

Views in Django are fundamentally functions. They are defined like any other Python function, and they are required at minimum to take in an HTTP request as their argument, and return an HTTP response. Here is an example of a basic view:

def logout\_view(request):

    logout(request)

    return redirect('/')

This view, unsurprisingly, logs out the user that is currently logged in. Thanks to Django’s built in authentication system, helpful functions such as logout make many views relatively simple to write. In this case, the response is an HTTP redirect response to ‘/’ (the root of the website).

Aside from these purely function based views, though, Django offers a system for creating class based views. This is immensely useful, as often a web application has several views that are very similar. In the case of the platform being developed here, many of the views are going to be some variation of “return a list of posts”. Variations such as “return a list of user *x*’s posts” and “return a list of posts that are forked from post *y*” are fundamentally still displaying lists of posts. These views can be thought of as “subclass” views of the main post list view.

If a class based view can be defined that can return a list of posts either as rendered HTML or as a JSON response, then this ability can easily be included in all of the special post list views. Django provides some nice generic class based views, such as ListView, which gets a list of objects of a specific model from the database. This is essentially what is required for most of the platform’s views.

A base PostListView class will be established with the ability to return a response as both HTML (through Django’s templating engine) and as JSON. This will allow easy creation of the following views:

* GlobalTimelineView (shows all posts reverse chronologically)
* UserTimelineView (shows all posts authored by a specific user)
* TagTimelineView (shows all posts that have one of a given list of tags)
* ForksView (shows all of the forks of a given post)
* FilterTimelineView (allows more fine grained filtering of posts based on GET parameters; will be used heavily in live view)
* SinglePostView (shows a single post)

Each of these views will return a JSON response if the GET key format is set to ‘json’, and an HTML response otherwise.

Other required views include NewPostView, ForkPostView and EditPostView, the views that deal with adding posts to the database (and editing existing ones). These views are responsible for two functions: displaying a webpage to a client, and handling a form submission. When the page for submitting a new post is accessed, the view must return a webpage on which the user can create their post. On submission of the form, i.e. the request sent to the server is a POST request, they must validate the data and then save the data to the database.

The major difference between NewPostView and the other two is that ForkPostView and EditPostView both require the page to be populated with the existing post’s body. As the three views are similar, it could make sense to have them be subclasses of each other: ForkPostView is just a NewPostView but with the contents of the post being forked being put into the post creation page, and EditPostView is just a ForkPostView where the post can only be forked by its owner, and the existing post is modified rather than a new one. These views will thus be implemented in this fashion: EditPostView is a subclass of ForkPostView which itself is a subclass of NewPostView.

Due to the complex nature of the posts on this platform, especially the variable number of cells, the form rendered by NewPostView will not be a standard HTML form. Instead, it will be a javascript controlled page where the user can synthesize their post, creating as few or as many cells as they wish. The javascript on the page will serialize the post—including title, tags, body and any other relevant information—as JSON before sending the POST request as an AJAX request, at which point NewPostView will unpack the JSON data sent with the request and save the new object in the database.

# Technical solution

# Testing

# Evaluation