

National College of Ireland

BSc in Computing

2017/2018

Seán Bickmore

X14428372

X14428372@student.ncirl.ie or seanbickmore@icloud.com

Reflux

Technical Report



TABLE OF CONTENTS

Table of Figures	9
Executive Summary	12
Introduction.....	13
Background	13
Aims.....	14
Technologies	14
Laravel.....	14
Semantic-UI	15
Sass	15
jQuery	15
Sequel Pro/MySQL.....	15
System.....	16
Requirements.....	16
Functional Requirements	16
Use Case Diagram	18
Abuse Case Diagram.....	19
Requirement 1: Registration	20
Requirement 2: Login.....	22
Requirement 3: User Profiles.....	24
Requirement 4: User Settings.....	26
Requirement 5: Content Submission	28
Requirement 6: Content Modification	30
Requirement 7: Content Deletion.....	32
Requirement 8: Content Filtering	34
Requirement 9: Content Tagging	36
Requirement 10: Content Searching.....	38
Requirement 11: Rating System	40
Requirement 12: Comments/Replies.....	42

Requirement 13: Night Mode.....	44
Non-Functional Requirements.....	46
Performance/Response Time	46
Availability	46
Recovery.....	46
Reliability.....	46
Maintainability.....	47
Extendibility.....	47
Reusability	47
Security.....	48
Data Requirements	48
User Requirements	49
Usability Requirements.....	49
Design and Architecture	50
Implementation.....	51
Frontend.....	51
Blade Templating	51
Compiling Assets.....	53
Syntax Highlighting	55
Eloquent ORM	56
Eloquent: Relationships.....	56
Routing.....	59
Scalability	60
Languages.....	60
Security	61
Authentication.....	61
Middleware	64
CSRF (Cross-Site Request Forgery) Protection.....	65
Hashing.....	66

Sensitive Data Exposure	67
Validation and Error Handling	68
Database	72
Database: Query Builder	72
Database: Pagination	77
Database: Seeding.....	78
Graphical User Interface (GUI) Layout.....	79
Registration	79
Login	80
User Profiles	81
User Settings	82
Content Submission	83
Content Modification.....	84
Content Deletion	85
Content Filtering.....	86
Content Tagging.....	87
Content Searching	88
Rating System	89
Comments/Replies.....	90
Night Mode	91
Testing	92
Creating Tests	92
Running Tests	93
Generating Factories.....	93
HTTP Testing.....	94
Guest User Tests	94
Authenticated User Tests	97
Test Results	101
Customer Testing	101

Question 1.....	101
Question 2.....	102
Question 3.....	102
Question 4.....	103
Question 5.....	103
Question 6.....	104
Question 7.....	104
Question 8.....	105
Question 9.....	105
Question 10.....	106
Evaluation	106
Overview	106
Responses	107
Question 1	107
Question 2	107
Question 3	107
Question 4	107
Question 5	107
Question 6	107
Question 7	108
Question 8	108
Question 9	108
Question 10.....	108
Conclusions.....	109
Further Development or Research	110
References.....	111
Appendix	112
Project Proposal	112
Objectives.....	112

Background	113
Technical Approach.....	114
Technical Details.....	114
Evaluation.....	115
Project Plan	116
Tasks	116
Gantt Chart.....	117
Monthly Journals.....	118
Introduction	118
September 2017	119
September 20 TH , 2017	119
September 25 TH , 2017	119
October 2017.....	119
October 2 ND , 2017	119
October 20 TH , 2017.....	120
October 26 th , 2017.....	120
November 2017	121
November 1 st , 2017	121
November 9 th , 2017.....	121
November 29 th , 2017	122
December 2017	123
December 12 th , 2017	123
February 2018	124
February 5 TH , 2018	124
March 2018	125
March 9 th , 2018	125
March 17 th , 2018	131
March 21 st , 2018.....	134
March 23 rd , 2018	138

March 24 th , 2018	140
March 25 th , 2018	142
March 27 th , 2018	145
March 31 st , 2018.....	145
April 2018.....	146
April 2 nd , 2018	146
April 6 th , 2018.....	146
April 7 th , 2018.....	147
May 2018	149
May 2 nd , 2018.....	149
May 6 th , 2018	151
May 13 th , 2018	151
Other Material Used.....	152
Requirements Elicitation Survey.....	152
Question 1	152
Question 2	152
Question 3	153
Question 4	153
Question 5	154
Question 6	154
Customer Testing Survey.....	155
Question 1	155
Question 2	155
Question 3	156
Question 4	156
Question 5	157
Question 6	157
Question 7	158
Question 8	158

Question 9	159
Question 10	159
User Manual	160
Accessing the Project.....	160
Creating an Account	161
Logging in to an Account.....	162
Logging out of an Account	163
Creating a Post	164
Text Post.....	164
Code Post	165
Editing a Post.....	166
Removing a Post.....	167
Searching	168
Filtering.....	169
Liking/Unliking a Post	169
Commeting/Replying	170
User Profiles.....	171
User Settings.....	173
Night Mode	175

TABLE OF FIGURES

FIGURE 1: REFLUX'S USE CASE DIAGRAM	18
FIGURE 2: REFLUX'S ABUSE CASE DIAGRAM	19
FIGURE 3: REGISTRATION USE CASE DIAGRAM	20
FIGURE 4: REGISTRATION ABUSE CASE DIAGRAM	20
FIGURE 5: LOGIN USE CASE DIAGRAM.....	22
FIGURE 6: LOGIN ABUSE CASE DIAGRAM	22
FIGURE 7: USER PROFILE USE CASE DIAGRAM.....	24
FIGURE 8: USER PROFILE ABUSE CASE DIAGRAM	24
FIGURE 9: USER SETTINGS USE CASE DIAGRAM	26
FIGURE 10: USER SETTINGS ABUSE CASE DIAGRAM.....	26
FIGURE 11: CONTENT SUBMISSION USE CASE DIAGRAM.....	28
FIGURE 12: CONTENT SUBMISSION ABUSE CASE DIAGRAM	28
FIGURE 13: CONTENT MODIFICATION USE CASE DIAGRAM	30
FIGURE 14: CONTENT MODIFICATION ABUSE CASE DIAGRAM	30
FIGURE 15: CONTENT DELETION USE CASE DIAGRAM	32
FIGURE 16: CONTENT DELETION ABUSE CASE DIAGRAM.....	32
FIGURE 17: CONTENT FILTERING USE CASE DIAGRAM.....	34
FIGURE 18: CONTENT FILTERING ABUSE CASE DIAGRAM	34
FIGURE 19: CONTENT TAGGING USE CASE DIAGRAM.....	36
FIGURE 20: CONTENT SEARCHING USE CASE DIAGRAM	38
FIGURE 21: CONTENT SEARCHING ABUSE CASE DIAGRAM.....	38
FIGURE 22: LOGIN USE CASE DIAGRAM.....	40
FIGURE 23: COMMENTS/REPLIES USE CASE DIAGRAM	42
FIGURE 24: COMMENTS/REPLIES ABUSE CASE DIAGRAM	42
FIGURE 25: NIGHT MODE USE CASE DIAGRAM	44
FIGURE 26: REFLUX'S ARCHITECTURE.....	50
FIGURE 27: MASTER LAYOUT FILE.....	51
FIGURE 28: DYNAMIC CONTENT	52
FIGURE 29: STATIC CONTENT.....	52
FIGURE 30: LARAVEL MIX FILE	53
FIGURE 31: SASS MAIN FILE	53
FIGURE 32: SASS PARTIAL FILE	54
FIGURE 33: JAVASCRIPT MAIN FILE	54
FIGURE 34: JAVASCRIPT PARTIAL FILE	54
FIGURE 35: SYNTAX HIGHLIGHTING	55
FIGURE 36: TINYMCE LANGUAGE ARRAY	55
FIGURE 37: USER MODEL	56
FIGURE 38: POST MODEL	57
FIGURE 39: TAG MODEL.....	57
FIGURE 40: COMMENT MODEL.....	58
FIGURE 41: VOTE MODEL.....	58
FIGURE 42: REGISTRATION ROUTES	59
FIGURE 43: BLANK REGISTRATION CREATE AND STORE METHODS.....	59
FIGURE 44: STORAGE: RETRIEVING A JSON ARRAY OF LANGUAGES.....	60
FIGURE 45: STORAGE: RETRIEVING A JSON ARRAY OF COUNTRIES.....	60
FIGURE 46: CREATING A NEW USER RECORD.....	61
FIGURE 47: LOGGING INTO AN ACCOUNT	62
FIGURE 48: RETRIEVING AN AUTHENTICATED USER	62

FIGURE 49: SESSIONS CONTROLLER: LOGGING OUT.....	63
FIGURE 50: REGISTRATION CONTROLLER: MIDDLEWARE CONSTRUCTOR.....	64
FIGURE 51: POSTS CONTROLLER: MIDDLEWARE CONSTRUCTOR.....	64
FIGURE 52: CSRF: BLADE FLAG.....	65
FIGURE 53: CSRF: EXAMPLE OF THE GENERATED INPUT.....	65
FIGURE 54: PASSWORD HASHING USING THE HASH FAÇADE.....	66
FIGURE 55: GRAVATAR EMAIL HASHING USING MD5.....	66
FIGURE 56: USER RECORD.....	67
FIGURE 57: HIDDEN ATTRIBUTES.....	67
FIGURE 58: JAVASCRIPT: VALIDATION.....	68
FIGURE 59: REGISTRATION CONTROLLER: VALIDATION.....	69
FIGURE 60: JAVASCRIPT REGULAR EXPRESSION.....	70
FIGURE 61: PHP REGULAR EXPRESSION.....	70
FIGURE 62: POSTS CONTROLLER: BLANK INDEX AND SHOW METHODS.....	72
FIGURE 63: RETRIEVING ALL POSTS AND PASSING TO THE VIEW.....	72
FIGURE 64: ROUTE FOR A SINGLE POST USING A WILDCARD.....	73
FIGURE 65: POSTS CONTROLLER: SHOWING A SINGLE POST VIA ROUTE MODEL BINDING.....	73
FIGURE 66: POSTS CONTROLLER: VALIDATING, CREATING, AND TAGGING A POST.....	74
FIGURE 67: POSTS CONTROLLER: EDIT METHOD.....	75
FIGURE 68: POSTS CONTROLLER: UPDATING A POST.....	75
FIGURE 69: POSTS CONTROLLER: DELETING A POST.....	76
FIGURE 70: DISPLAYING DATA PASSED TO THE VIEW.....	76
FIGURE 71: POSTS CONTROLLER: PAGINATING RESULTS.....	77
FIGURE 72: FRONTEND: PAGINATION PARTIAL.....	77
FIGURE 73: DATABASE SEEDING: USER ACCOUNT.....	78
FIGURE 74: REGISTRATION VIEW.....	79
FIGURE 75: LOGIN VIEW.....	80
FIGURE 76: USER PROFILES VIEW.....	81
FIGURE 77: USER SETTINGS VIEW.....	82
FIGURE 78: CONTENT SUBMISSION VIEW.....	83
FIGURE 79: CONTENT MODIFICATION VIEW.....	84
FIGURE 80: CONTENT DELETION MODAL.....	85
FIGURE 81: CONTENT FILTERING COMPONENT.....	86
FIGURE 82: CONTENT TAGGING COMPONENT.....	87
FIGURE 83: CONTENT SEARCHING COMPONENT.....	88
FIGURE 84: RATING COMPONENT.....	89
FIGURE 85: COMMENTS/REPLIES COMPONENT.....	90
FIGURE 86: NIGHT MODE COMPONENT.....	91
FIGURE 87: CREATING A UNIT TEST.....	92
FIGURE 88: LOCATION OF UNIT TEST.....	92
FIGURE 89: RUNNING UNIT TESTS.....	93
FIGURE 90: GENERATING A FACTORY.....	93
FIGURE 91: GENERATED POST FACTORY.....	93
FIGURE 92: GUEST USER: HOMEPAGE TEST.....	94
FIGURE 93: GUEST USER: REGISTRATION TEST.....	94
FIGURE 94: GUEST USER: LOGIN TEST.....	94
FIGURE 95: GUEST USER: VALID CREDENTIALS TEST.....	95
FIGURE 96: GUEST USER: INVALID CREDENTIALS TEST.....	95
FIGURE 97: GUEST USER: USER PROFILE TEST.....	95

FIGURE 98: GUEST USER: USER SETTINGS TEST	96
FIGURE 99: GUEST USER: CONTENT SUBMISSION TEST	96
FIGURE 100: GUEST USER: CONTENT MODIFICATION TEST	96
FIGURE 101: AUTHENTICATED USER: HOMEPAGE TEST	97
FIGURE 102: AUTHENTICATED USER: REGISTRATION TEST	97
FIGURE 103: AUTHENTICATED USER: LOGIN TEST	98
FIGURE 104: AUTHENTICATED USER: USER PROFILE TEST	98
FIGURE 105: AUTHENTICATED USER: AUTH USER SETTINGS TEST	99
FIGURE 106: AUTHENTICATED USER: OTHER USER SETTINGS TEST	99
FIGURE 107: AUTHENTICATED USER: CONTENT SUBMISSION TEST	100
FIGURE 108: AUTHENTICATED USER: CREATING A POST	100
FIGURE 109: UNIT TESTING RESULTS	101
FIGURE 110: CUSTOMER TESTING: QUESTION 1 RESULTS	101
FIGURE 111: CUSTOMER TESTING: QUESTION 2 RESULTS	102
FIGURE 112: CUSTOMER TESTING: QUESTION 3 RESULTS	102
FIGURE 113: CUSTOMER TESTING: QUESTION 4 RESULTS	103
FIGURE 114: CUSTOMER TESTING: QUESTION 5 RESULTS	103
FIGURE 115: CUSTOMER TESTING: QUESTION 6 RESULTS	104
FIGURE 116: CUSTOMER TESTING: QUESTION 7 RESULTS	104
FIGURE 117: CUSTOMER TESTING: QUESTION 8 RESULTS	105
FIGURE 118: CUSTOMER TESTING: QUESTION 9 RESULTS	105
FIGURE 119: CUSTOMER TESTING: QUESTION 10 RESULTS	106

EXECUTIVE SUMMARY

The following documentation pertains to the fourth year cyber security web application project entitled Reflux. The context of this application is as a body of knowledge, specifically for developers to share or query for solutions based in code with support for a plethora of languages. It can be held within the same realm as competitive technologies such as Stack Overflow, GitHub, or JS Fiddle. Within each of these is a feature-set that which Reflux aims to encompass in part and satisfy beyond the scope of what is offered by these competitors with heavy focus on communication/comprehension of the application itself as derived from its user interface. The project must additionally satisfy security requirements with particular respect to those as outlined by the Open Web Application Security Project (OWASP). Reflux itself is not deeply rooted in security thematically, but consists of the most common elements prone to vulnerabilities and exploitation, such as manipulating inputs with malicious intent to achieve an attacking objective that, in practice, compromises some or all of three facets, confidentiality, integrity, and availability or authentication.

Reflux is built on top of the Laravel PHP framework, which streamlines otherwise complex setup processes that are common to most web applications, such as routing, authentication, and session handling. It is a model view controller framework which compartmentalises the frontend contextual functionality that defines Reflux in a business sense, and its security implementations that occur in the backend that intrinsically link with said contextual functionality. As such, Reflux's internal setup means that database associations, backend, and frontend all communicate but locate externally and separate to one another. There are a plethora of helper functions defined in custom syntax that evoke many of the most important features of Laravel, inclusive of some security measures employed by Reflux. For example, by default, Reflux uses middleware to discern what type of user a user is, providing a layer of access control that can be called in any circumstance to gate views or derive what functionality of the site a given user type is allowed to execute. This instance, and any instance whereby a user is validated or something the user has provided is validated, are all processed through appropriately named and manageable controllers that link with a given singular or set of views.

Eliminating the context of Reflux, it encompasses key aspects that most web applications elicit in managing content in some fashion. Features that govern its capabilities in handling and displaying data as submitted by its user-base. The importance of security does not present itself on the frontend in Reflux, but is the most vital considerable non-functional requirement at play that ensures that each functional requirement cannot be utilised in undesirable ways, or ways in which the CIA triad of the application itself is at stake. In a business sense, the sellable idea of Reflux is within the context as outlined, and not within these security implementations. However, it stresses the valuable and necessary inclusion and consideration of security layers in applications that function similarly to Reflux, such as competitive technologies or any that exude a capacity of allowing an established user-base to get, post, patch, and destroy data of a contextual or arbitrary nature. In a given circumstance of normal usage, validation is performed in most if not all cases of executing a function of the site, especially when handling data. Most prominently, access control and input validation provide a backend foundation of aspects of the site that perform the aforementioned. When examining competitive solutions, security implementations were not a presented issue, but rather in how said solutions communicate the intended executable action behind elements in a given situation. It was discerned that the user interface itself must cleanly and concisely achieve this communication, which is an area whereby, in some respects, alternatives falter. Stylistically, a modern approach to design was taken in melding elements to the functionality that which they govern in a fashion that achieves just as what was proposed, and in Laravel, the blade templating engine ensures that Reflux's views include reusable components that are global to the site, reducing redundancy, establishing a set theme, and streamlining the refactoring process.

INTRODUCTION

BACKGROUND

With respect to the cyber security specialisation, the project idea was initially envisioned to encapsulate the meaning and importance of security within the idea itself both thematically and functionally. This heavily affected the scope of concocted ideas, as there were not many that held the capacity of satisfying both adequately. From this born the idea for a password manager web application. In a business sense, in both the platform and nature of this idea were two major issues. Many alternative solutions were backed by entire organisations, and said organisations solutions spanned a multitude of platforms, inclusive of the latest authentication technologies specific to some of these platforms, such as three-factor authentication. This meant the project idea would need to consider these aspects, enhance on those that are common across all, and deliver a unique selling point, all of which trumps competitive alternatives to some degree. As it was envisioned to be a web application, this additionally meant that certain avenues could not be explored that which the competition could with solutions based in desktop and mobile conjunctively. Thusly, it would not be able to encompass and satisfy all prospects presented by competitive technologies to a satisfactory standard.

This idea was abandoned in pursuit of a previously explored idea that could be expanded upon both in scope with respect to its context, as well as its security implementations. The baseline functionality of this application meant that while security would not be contextually represented in the frontend, it could be represented in the backend to a standard that satisfies many of the vulnerable and exploitable factors defined present in web applications as outlined by the Open Web Application Security Project (OWASP). In its basest form, it plays host to the functionality expected of an application of this nature, allowing a user-base to get, post, patch, and destroy data. Previously, this functionality was present, but security implementations were not within the scope of consideration nor could the feature-set reach a potential to be classified as a sellable idea in the span of time allocated for its first implementation. In this case, all bases could be covered, and due to its previous exploration, the requirements elicitation process was simplified. Between then and now, both the competition and requirements elicited did not deviate too much from what was previously defined. In both cases, the application hosting the most common thematic context to this idea was Stack Overflow. Within both, users can create topics with the intent to spark conversation for the purpose of acquiring new knowledge. Stack Overflow is a questions and answers site that is most commonly associated with developers seeking code solutions. However, there are a plethora of subdomains in use that all belong to the same family, but define different contexts of the questions that are allowed to be asked. Through examination of competitive solutions whereby particular correlations between feature-sets could be drawn, such as stylistically, syntax highlighting, and the common capacity to allow an established user-base to submit content to these sites, the scope of Reflux was identified.

With particular focus on Stack Overflow, this meant building upon this idea as a baseline conjunctively with uniquely differentiating it thematically. The idea was not as a question and answers site, but as a developmental hub for developers whereby queries may be posted, but also solutions to accomplish objectives without query, and of any context. In the latter sense, this meant unifying what are subdomains on Stack Overflow for segregating questions and answers of different contexts, and providing the means for users to compartmentalise what they are looking for without visiting a different site entirely. Other competitors, such as GitHub and JS Fiddle, are defined as such based on the features that correlate, and not within the entirety of the contexts that they occupy. With GitHub, it is aesthetically modern and allows the submission of code for the purpose of version control. Both it and Stack Overflow provide very basic syntax highlighting that satisfies the purpose of displaying a code block. JS Fiddle runs segmented pieces of code together with the intent of showing the output of these code blocks conjunctively with colourful syntax highlighting, but only within the realm of the languages it supports. There are facets to these applications that, when unified, provide an enhanced experience for developers that has not currently been explored and thusly represents the background behind Reflux.

AIMS

Reflux aims to satisfy the two major components that define it. The frontend with respect to user experience and comprehension, and the backend with respect to fulfilling its functional and non-functional requirements. Subsequently, the former equates to what differentiates Reflux from competitive solutions or those that comprise of correlating functionality and presentation. When examining Stack Overflow, GitHub, and JS Fiddle, while each plays host to fulfilling varying marketable ideas, foundationally speaking, they provide developers a means to query, answer, demonstrate, and provide open source code solutions to an encompassed community. The idea was not to represent any of these products explicitly in totality, but to build an application that accomplishes an overarching goal that each of them satisfy singularly when viewed hypothetically within a unified system. This hypothesis defines what Reflux aims to achieve. In its basest form, Reflux is about allowing a target market of individuals to query and provide solutions, spark discussion of topics, and do so within an encompassing user interface that provides a comprehensive, modern stylistic experience that expertly communicates the intended functionality behind elements and encourages both returning usage and maintenance of the community it envisages to host. This defines Reflux's context. However, it is aptly additionally aimed that the application ensures that for each differentiating functional requirement that satisfies Reflux as a marketable idea, that the backend in turn satisfies non-functional security requirements that layer each of these features. By disregarding Reflux's context in a business sense, it correlates with any web application that allows a user-base to view, submit, modify, and delete content. Therein lies the capacity with which the foundation of the Reflux's security implementations stem and where, despite its context, it correlates with other web applications. Broadly, most of Reflux's functional requirements involve displaying and manipulating data, often on account of user interaction and input. Thusly, for each of these requirements there must be a subsequent non-functional security requirement. In any given circumstance, the user-base must be constantly under validation with respect to the actions they are executing and the views they are accessing. These components define the objectives that which Reflux aims to accomplish.

TECHNOLOGIES

LARAVEL

Reflux is built using Laravel, which is a PHP model view controller framework. It is vastly extensive in its outlined plethora of helper functions and provisioned implementations that derive a swift development experience with an accessible learning curve. Laravel's eloquent ORM is a simple active record implementation that enforces that for each database table, there is a corresponding model. These models allow for querying of the database as well as to draw associations between tables that have a relationship with one another. With Reflux, this comes into play heavily, as data from other tables is consistently associated with a given user record based on their submissions. In turn, submissions themselves also relate to other data, and it cascades from there. Controllers segregate backend validation and functionality from Reflux's views. In the same way that there is a model for each database table, so too is there a controller for each functional requirement if said requirement involves communication with the database. In essence, when a user views, submits, modifies, or deletes a post, this functionality is entirely encompassed within a controller relating specifically to posts. This provides an efficient structure to the application from a development standpoint, and each controller uses the same naming convention in every circumstance that defines the methods encapsulated in each. With respect to the frontend, Laravel provides the blade templating engine, allowing for the creation of blade PHP files with the express purpose of defining layout files and partials. Layout files represent instances of the user interface that are used globally throughout the entirety of the site or a specific set of views. Partials are inclusions in multiple views that represent an element singularly. This means that Reflux's user interface is modifiable in isolated instances and updated globally within the views that which the modified file pertains to.

SEMANTIC-UI

Semantic-UI is a free open source CSS framework for creating concise, intuitive, and simple user interfaces. The main principle, is that it treats words and classes as interchangeable concepts, and draws relationships between multiple classes based on word order. In turn, this increases the readability of HTML by linking said concepts behind what is achieved by a given class name by communicating as such through the class name itself. The decision to use Semantic over alternative CSS frameworks stems from the aforementioned in addition to its simplicity and customizability on account of the former. Because components are relatively basic, there is a vast scope to work with in terms of melding them to a state whereby Reflux attains its identity. In essence, by using Semantic, it is not entirely obvious that the application uses Semantic. When viewed comparatively to alternatives such as Bootstrap or Materialize, there are facets that take intricate workings to override components of these frameworks in order to adapt them beyond the scope of what the framework intends them to be both aesthetically and functionally. While everything can be taken from the framework and used as is, everything is mutable, which plays heavily into the theming of Reflux.

SASS

Sass was used in the development of Reflux. Sass is a CSS pre-processor that expands the scope of features that CSS is currently capable of by introducing variables, nesting, partials, imports, mixins, extension/inheritance, and operators. With regards to Reflux and Semantic, this allows for defining variables that equate to values such as font-weights and colours, and with partials, the aforementioned may be located external to the main CSS file, then imported, thusly compartmentalizing the applications stylesheets. Similarly to Laravel's blade templating, this means that if a variable from a partial that is used in multiple places is modified, that the value will then update in every instance whereby said variable was used. This structures Reflux's stylesheets, allowing for efficient and consistent theming, and refactoring. Laravel supports Sass with command line tools and an internal project dependency file that determines the location of an input sass file and output CSS file that will be populated upon watching for changes in said sass file.

JQUERY

jQuery is a JavaScript library that simplifies client-side scripting for HTML. In Reflux, it was used for client-side validation and as a dependency of Semantic-UI's JavaScript components. Like Sass, Reflux's JavaScript is segmented into manageable, isolated chunks that are then compiled into a single JavaScript file by watching for changes in Laravel's internal project dependency file that specifies an input and output file. Each file works similarly to a blade template or Sass partial, in that they are inclusions of a main file, but are simpler to locate, modify, and manage as the actual code pertaining to a specific component is found within a file solely dedicated to them. With respect to Semantic, certain components are unusable without JavaScript, such as input validation, dropdowns, and popup messages. In these instances, the code relating to these components are located in appropriately named JavaScript files that are then included in a main file. Upon building the project, these culminate within a single output file.

SEQUEL PRO/MYSQL

Sequel Pro is an OSX database management application for working with MySQL databases. From a local development perspective, the Laravel PHP artisan command line tools are utilised to manipulate a MySQL database pertaining to Reflux. Reflux's environment file is used to define an association between the Laravel project and an instance of a MySQL database encapsulated by Sequel Pro. The artisan tools are used to create migration files in the file structure of the project that represents a table. These files are manually modified to include the appropriate rows. Said tables are migrated to the MySQL instance defined in the environment file.

SYSTEM

REQUIREMENTS

FUNCTIONAL REQUIREMENTS

The following are representative of Reflux's functional requirements as elicited prior to and during development:

- ❖ Registration
- ❖ Login
- ❖ User Profiles
- ❖ User Settings
- ❖ Content Submission
- ❖ Content Modification
- ❖ Content Deletion
- ❖ Content Filtering
- ❖ Content Tagging
- ❖ Content Searching
- ❖ Rating System
- ❖ Comments/Replies
- ❖ Night mode

Registration

Refers to the process of allowing a user to create an account given satisfaction of all enforced validation requirements.

Login

Refers to the process of allowing a user to sign into an account given satisfaction of all enforced validation requirements. Provided input must match that of an existing user record.

User Profiles

Refers to the inclusion of personalised user profiles relating to the authenticated user and the user-base in totality. Allows access to a user's submitted posts, liked posts, and comment history.

User Settings

Refers to the process of allowing a user to alter some of the details of an existing account, inclusive of those that are null by default (i.e. bio, affiliate site link, email address) given the user is authenticated.

Content Submission

Refers to the process of allowing an authenticated user to submit a post posed in the context of a query or solution, with or without the inclusion of technical material such as code, given satisfaction of all enforced validation requirements.

Content Modification

Refers to the process of allowing an authenticated user to modify an existing post given satisfaction of all enforced validation requirements. Post in question must be associated with the user account (i.e. ownership). The title of a post cannot be altered. This ensures that the context of a post cannot be changed.

Content Deletion

Refers to the process of allowing an authenticated user to remove an existing post given input of the posts title as confirmation. Post in question must be associated with the user account (i.e. ownership). Deletion of a post in turn removes all associated data from other tables such as votes and comments.

Content Filtering

Refers to the process of allowing a user to order site content in accordance with defined metrics (e.g. by score, newest, oldest).

Content Tagging

Refers to the process of allowing authenticated users to tag content during both the submission and modification stages that identifies languages encompassed by the post or the general context of what the post itself relates to (e.g. HTML, CSS, JavaScript).

Content Searching

Refers to the process of allowing users to search for site content in accordance with user input matching or partially matching a posts title, tags, or author.

Rating System

Refers to the process of allowing authenticated users to vote on existing posts. A vote can either be placed or revoked, but not of a negative nature such as disliked.

Comments/Replies

Refers to the process of allowing authenticated users to leave comments on existing posts, and leave replies to existing comments.

Night Mode

Refers to the process of allowing users to alter the sites theme, adjusting the colour-scheme for ease-of-use circumstantially dependant on the time of day. It reduces the harshness of white light emitted as presented by the default theme.

USE CASE DIAGRAM

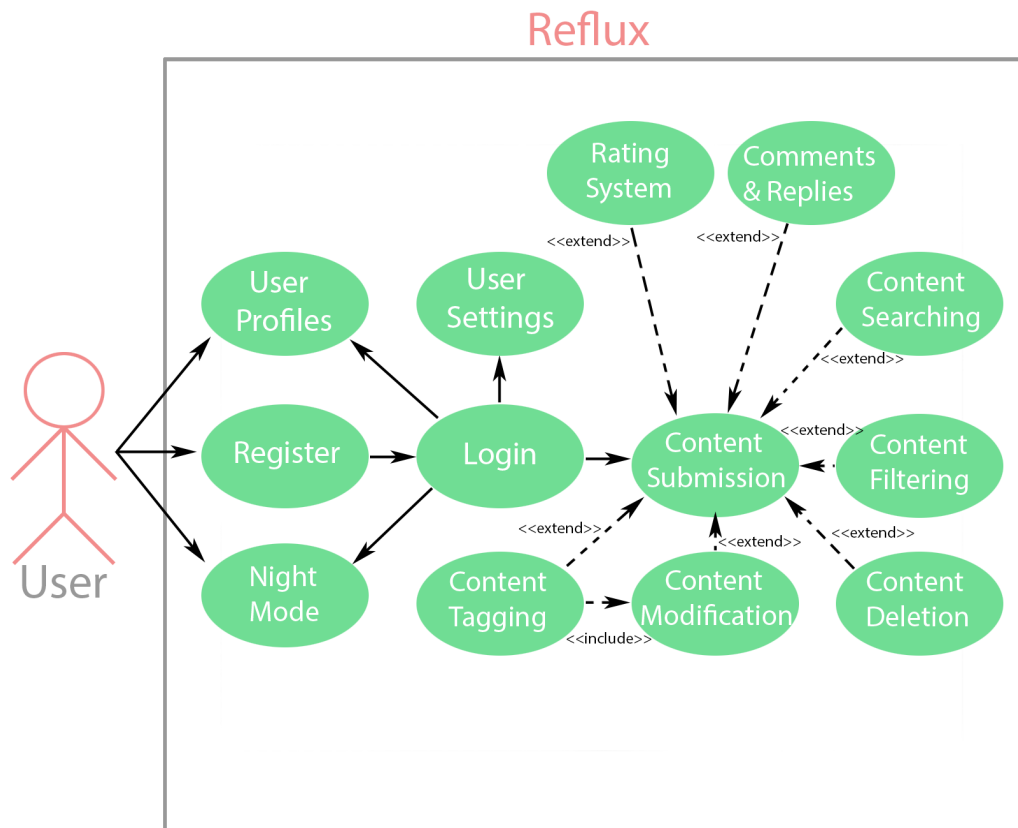


Figure 1: Reflux's Use Case Diagram

ABUSE CASE DIAGRAM

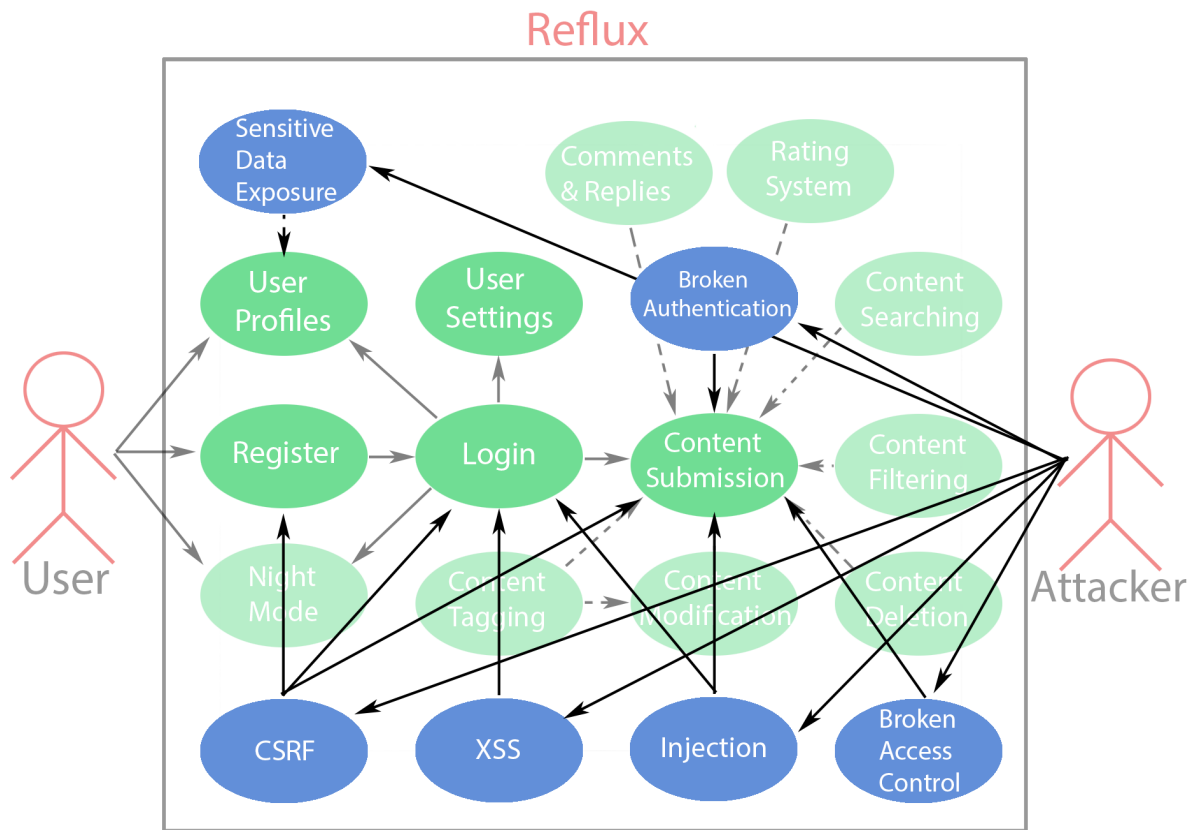


Figure 2: Reflux's Abuse Case Diagram

REQUIREMENT 1: REGISTRATION

Description and Priority

Allows users to register an account with the system. Feature is a prerequisite necessity for all aspects of the application that involve manipulating data, as an authenticated user is required to do so.

Use Case

Scope

The scope of this use case is to allow for a user to be able to register an account with the system to further engage with the application.

Description

This use case describes the process through which a user must follow to register an account with the system.

Use Case Diagram

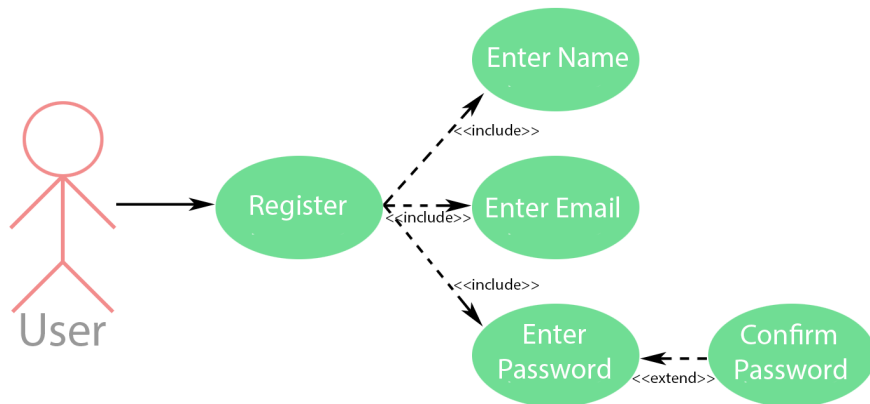


Figure 3: Registration Use Case Diagram

Abuse Case Diagram

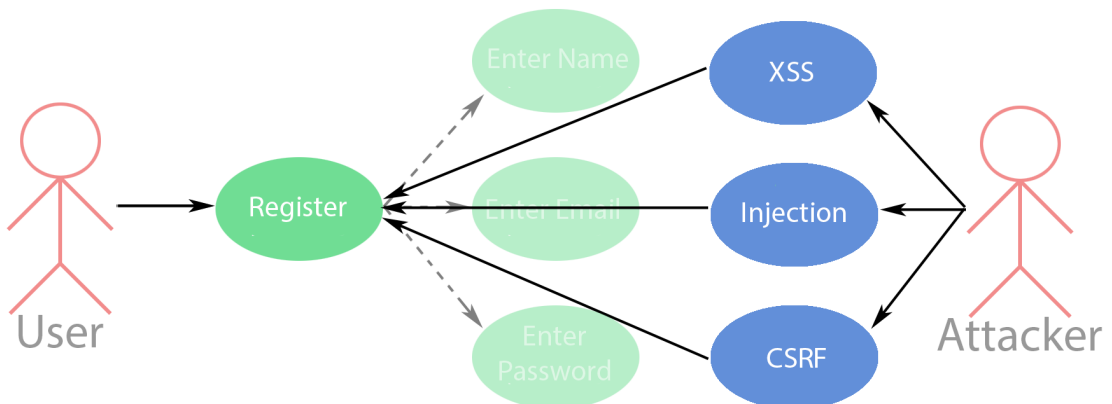


Figure 4: Registration Abuse Case Diagram

Flow Description

Precondition

The application is hosted and running in an online environment.

Activation

Use case commences when a user engages with the registration view of the application.

Main Flow

1. The user accesses the registration view from one of several elements that accomplish as such.
2. The system directs the user to the registration view.
3. The user enters details [A1: Unfulfilled Validation]
4. The user submits the details as aforementioned for validation with the system [A2: Existing User].
5. The system creates a new user record equating to the details provided.
6. The system redirects the user to the homepage.
7. The use case ends successfully.

Alternative Flow

A1: Unfulfilled Validation

1. The system notifies the user any unfulfilled validation requirements.
2. The user satisfies the stipulation.
3. The use case continues from step 4 of the main flow.

A2: Existing User

1. The system notifies the user that an account encompassing these details already exists.
2. The user satisfies the stipulation.
3. The use case continues from step 5 of the main flow.

Exceptional Flow

E1: Cancellation

1. The user ceases the registration process by performing a navigable action to another view.
2. The system directs the user to the view in question.
3. The use case ends with a failure condition.

Termination

Scenarios

1. The user ceases the registration process.
2. The user registers an account with the system successfully.

Post Condition

Success

The users details, as provided, are validated by the system thusly creating a new user record. This record is then used to log the user in following submission of valid credentials.

Failure

The system remains in the state prior to the users engagement with the registration process. Details that may or may not have been provided, are not logged with the system.

REQUIREMENT 2: LOGIN

Description and Priority

Allows users to sign into an existing account provided credentials that match a record. Feature is a prerequisite necessity for all aspects of the application that involve manipulating data, as an authenticated user is required to do so.

Use Case

Scope

The scope of this use case is to allow for a user to be able to sign into an existing account to further engage with the application.

Description

This use case describes the process through which a user must follow to sign into an existing account as created through the registration process.

Use Case Diagram

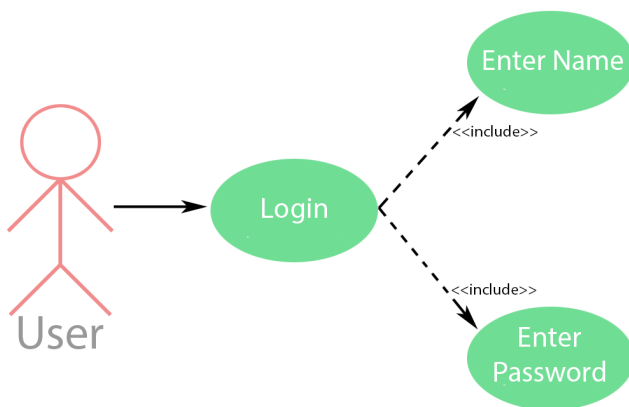


Figure 5: Login Use Case Diagram

Abuse Case Diagram

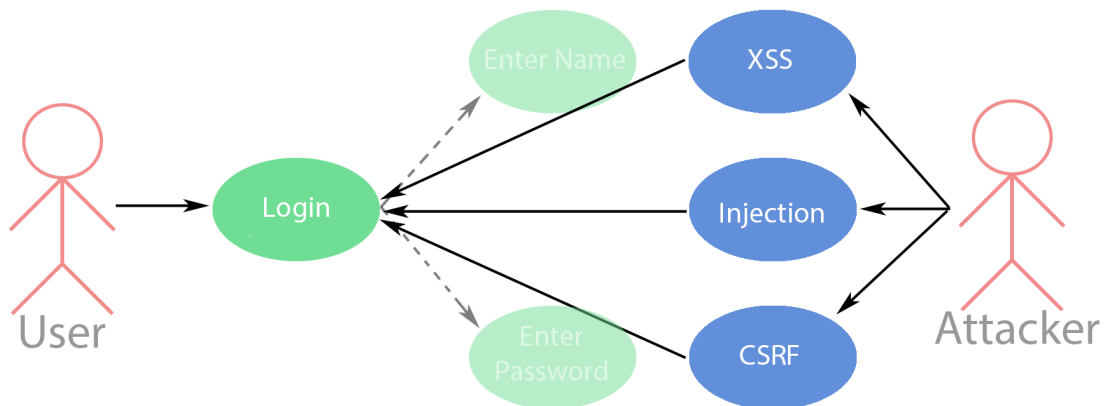


Figure 6: Login Abuse Case Diagram

Flow Description

Precondition

The application is hosted and running in an online environment.

Activation

Use case commences when a user engages with the login view of the application.

Main Flow

1. The user accesses the login view from one of several elements that accomplish as such.
2. The system directs the user to the login view [E1: Cancellation].
3. The user enters details [A1: Unfulfilled Validation]
4. The user submits the details as aforementioned for validation with the system [A2: Incorrect Credentials | E2: Non-existent User].
5. The system queries for a user record matching the provided input.
6. The system redirects the user to the homepage.
7. The use case ends successfully.

Alternative Flow

A1: Unfulfilled Validation

1. The system notifies the user any unfulfilled validation requirements.
2. The user satisfies the stipulation.
3. The use case continues from step 4 of the main flow.

A2: Incorrect Credentials

In step 4 of the main flow, if the users has provided incorrect credentials:

1. The system notifies the user that the data pertaining to a valid user record is incorrect (e.g. password).
2. The user satisfies the stipulation.
3. The use case continues from step 4 of the main flow.

Exceptional Flow

E1: Cancellation

From step 2 of the main flow:

1. The user ceases the login process by performing a navigable action to another view.
2. The system directs the user to the view in question.
3. The use case ends with a failure condition.

E2: Non-existent User

From step 4 of the main flow:

1. The user submits the details for an account that does not exist.
2. The system notifies the user that a record pertaining to the provided input is non-existent.
3. The use case ends with a failure condition.

Termination

Scenarios

1. The user ceases the login process.
2. The user signs into an existing account successfully.

Post Condition

Success

The users details, as provided, are validated by the system thusly accessing an existing user account.

Failure

The system remains in the state prior to the users engagement with the login process. Details that may or may not have been provided, are not logged with the system.

REQUIREMENT 3: USER PROFILES

Description and Priority

Allows guest users to access the profiles of other users, and authenticated users to do the former conjunctively with their own personalised profile. Feature attributes to users their submissions, liked posts, and comment history, as well as optional identifying details.

Use Case

Scope

The scope of this use case is to allow for a user to be able to sign into an existing account to further engage with the application.

Description

This use case describes the process through which a user must follow to sign into an existing account as created through the registration process.

Use Case Diagram

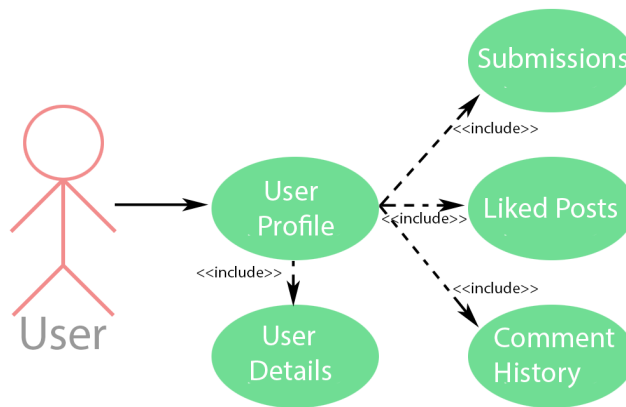


Figure 7: User Profile Use Case Diagram

Abuse Case Diagram

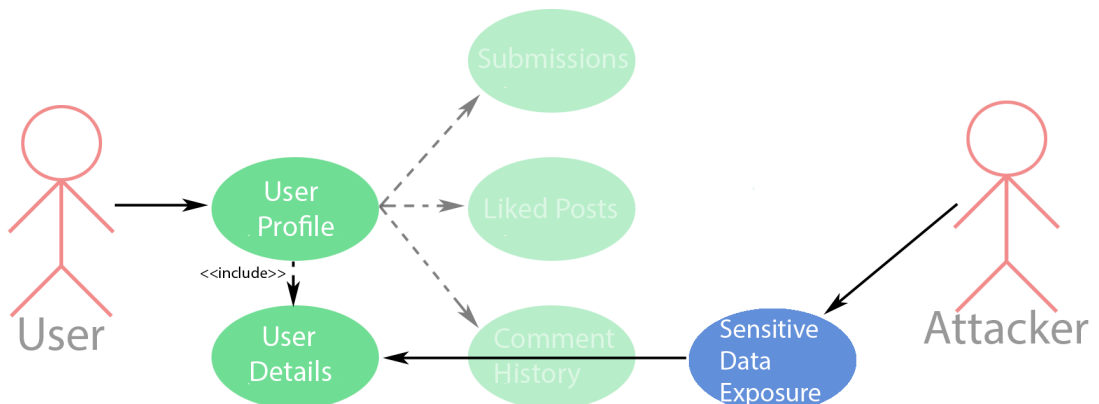


Figure 8: User Profile Abuse Case Diagram

Flow Description

Precondition

The application is hosted and running in an online environment.

Activation

Use case commences when a user engages with the view pertaining to another users profile or their own profile.

Main Flow

1. The user accesses the user profile view from one of several elements that accomplish as such.
2. The system directs the user to the user profile view.
3. The user selects a tab [A1: Default Tab | A2: Likes Tab | A3: Comments Tab]
4. The use case ends successfully.

Alternative Flow

A1: Default Tab

1. The system displays data pertaining to a given users submissions.
2. The use case continues from step 4 of the main flow.

A2: Likes Tab

1. The system displays data pertaining to a given users liked content.
2. The use case continues from step 4 of the main flow.

A3: Comments Tab

1. The system displays data pertaining to a given users comment history.
2. The use case continues from step 4 of the main flow.

Exceptional Flow

-

Termination

Scenarios

1. The user accesses their own or another users profile.
2. The user accesses additional information pertaining to a given users submissions, liked posts, and comment history.

Post Condition

Success

The profile view is populated with the relevant data associated with the users account as accessed by the user.

REQUIREMENT 4: USER SETTINGS

Description and Priority

Allows authenticated users to modify specific settings pertaining to their account. Feature facilitates the inclusion or exclusion of optional, additional identifying information.

Use Case

Scope

The scope of this use case is to allow for a user to be able to modify settings pertaining to publically displayed information within submitted posts and user profiles.

Description

This use case describes the process through which a user must follow to modify account settings with respect to the optional inclusion or exclusion of information.

Use Case Diagram

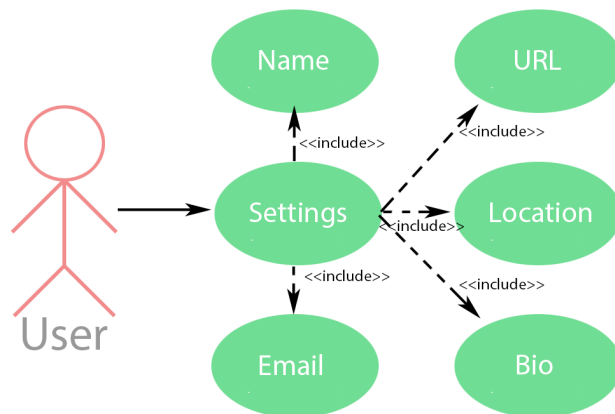


Figure 9: User Settings Use Case Diagram

Abuse Case Diagram

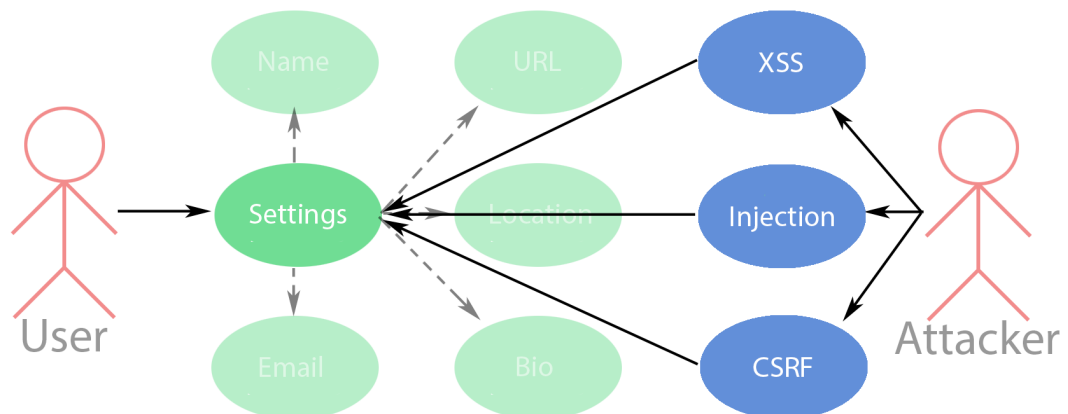


Figure 10: User Settings Abuse Case Diagram

Flow Description

Precondition

The application is hosted and running in an online environment.

Activation

Use case commences when a user engages with the settings view of the application.

Main Flow

1. The user accesses the settings view encompassed by the navigation dropdown menu.
2. The system directs the user to the settings view [E1: Cancellation].
3. The user modifies any or all of the mutable settings (i.e. name, email, URL, location, bio) [A1: Unfulfilled Validation].
4. The user submits the details as aforementioned for validation with the system [A2: Existing User].
5. The system updates the details associated with the currently authenticated user.
6. The system redirects the user to the homepage.
7. The use case ends successfully.

Alternative Flow

A1: Unfulfilled Validation

1. The system notifies the user any unfulfilled validation requirements.
2. The user satisfies the stipulation.
3. The use case continues from step 4 of the main flow.

A2: Existing User

1. The system notifies the user that an account encompassing these details already exists.
2. The user satisfies the stipulation.
3. The use case continues from step 5 of the main flow.

Exceptional Flow

E1: Cancellation

1. The user ceases the alteration of their details within the settings view of the application by performing a navigable action to another view.
2. The system directs the user to the view in question.
3. The use case ends with a failure condition.

Termination

Scenarios

1. The user ceases the modification to their details.
2. The user modifies their details successfully.

Post Condition

Success

The users details, as provided, are validated by the system thusly modifying the associated user record. Changes are reflected on the users profile.

Failure

The system remains in the state prior to the users engagement with the registration process. Details that may or may not have been provided, are not logged with the system.

REQUIREMENT 5: CONTENT SUBMISSION

Description and Priority

Allows authenticated users to submit content to the system relating to a query or solution, and inclusive or exclusive of technical facets (i.e. code blocks).

Use Case

Scope

The scope of this use case is to allow for a user to be able to submit content.

Description

This use case describes the process through which a user must follow to submit content.

Use Case Diagram

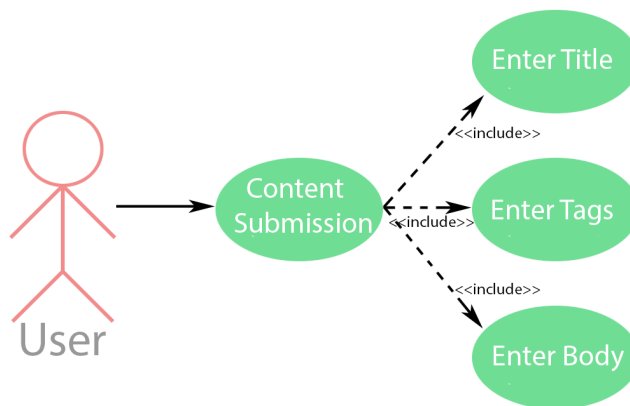


Figure 11: Content Submission Use Case Diagram

Abuse Case Diagram

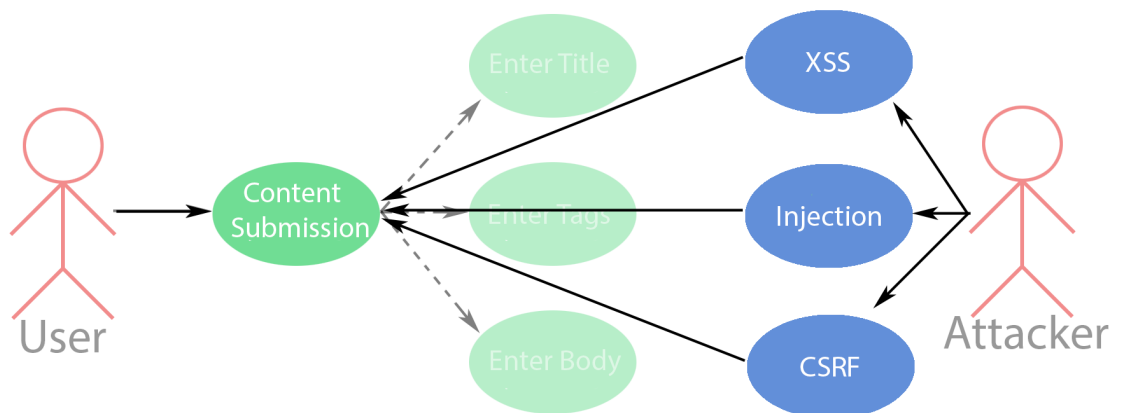


Figure 12: Content Submission Abuse Case Diagram

Flow Description

Precondition

The application is hosted and running in an online environment.

Activation

Use case commences when a user engages with the post creation view of the application.

Main Flow

1. The user accesses the post creation view from one of several elements that accomplish as such.
2. The system directs the user to the post creation view [E2: Cancellation].
3. The user enters details [A1: Unfulfilled Validation].
4. The user submits the details as aforementioned for validation with the system.
5. The system creates a new post record equating to the details provided.
6. The system redirects the user to the homepage.
7. The use case ends successfully.

Alternative Flow

A1: Unfulfilled Validation

1. The system notifies the user any unfulfilled validation requirements.
2. The user satisfies the stipulation.
3. The use case continues from step 4 of the main flow.

Exceptional Flow

E1: Cancellation

1. The user ceases the post creation process by performing a navigable action to another view.
2. The system directs the user to the view in question.
3. The use case ends with a failure condition.

Termination

Scenarios

1. The user ceases the post creation process.
2. The user creates a new post successfully.

Post Condition

Success

The post details, as provided, are validated by the system thusly creating a new post. This post is then attributed to the user in question and associated with variable other data sets.

Failure

The system remains in the state prior to the users engagement with the post creation process. Details that may or may not have been provided, are not logged with the system.

REQUIREMENT 6: CONTENT MODIFICATION

Description and Priority

Allows authenticated users to modify existing content attributed to them, relating to a query or solution, and inclusive or exclusive of technical facets (i.e. code blocks).

Use Case

Scope

The scope of this use case is to allow for a user to be able to modify existing content.

Description

This use case describes the process through which a user must follow to modify content.

Use Case Diagram

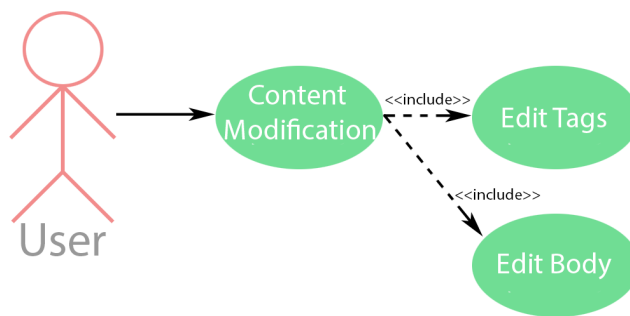


Figure 13: Content Modification Use Case Diagram

Abuse Case Diagram

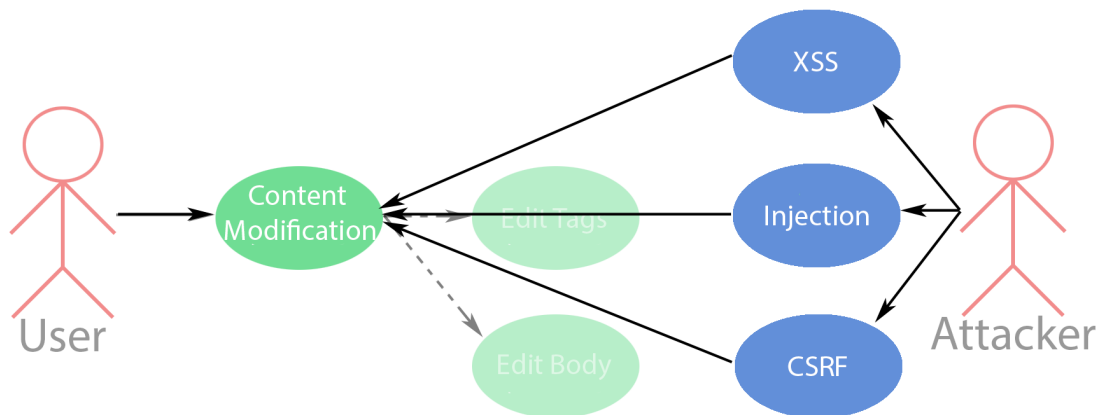


Figure 14: Content Modification Abuse Case Diagram

Flow Description

Precondition

The application is hosted and running in an online environment.

Activation

Use case commences when a user engages with the edit post view of the application.

Main Flow

1. The user accesses the edit post view from one of several elements that accomplish as such.
2. The system directs the user to the edit post view [E1: Cancellation].
3. The user modifies any or all of the fields that which pertain to an existing posts tags or body [A1: Blank Field(s)].
4. The user submits the details as aforementioned for validation with the system.
5. The system modifies that existing post equating to the details provided.
6. The system redirects the user to the homepage.
7. The use case ends successfully.

Alternative Flow

A1: Unfulfilled Validation

1. The system notifies the user any unfulfilled validation requirements.
2. The user satisfies the stipulation.
3. The use case continues from step 4 of the main flow.

Exceptional Flow

E1: Cancellation

From step 2 of the main flow:

1. The user ceases the edit post process by performing a navigable action to another view or executing the cancel element.
2. The system directs the user to the view in question.
3. The use case ends with a failure condition.

Termination

Scenarios

1. The user ceases the edit post process.
2. The user modifies a post successfully.

Post Condition

Success

The post details, as provided, are validated by the system thusly modifying an existing post.

Failure

The system remains in the state prior to the users engagement with the edit post process. Details that may or may not have been provided, are not logged with the system.

REQUIREMENT 7: CONTENT DELETION

Description and Priority

Allows authenticated users to delete existing content attributed to them, relating to a query or solution, and inclusive or exclusive of technical facets (i.e. code blocks).

Use Case

Scope

The scope of this use case is to allow for a user to be able to delete existing content.

Description

This use case describes the process through which a user must follow to delete content.

Use Case Diagram

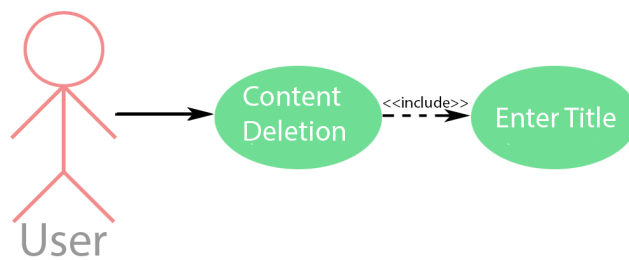


Figure 15: Content Deletion Use Case Diagram

Abuse Case Diagram

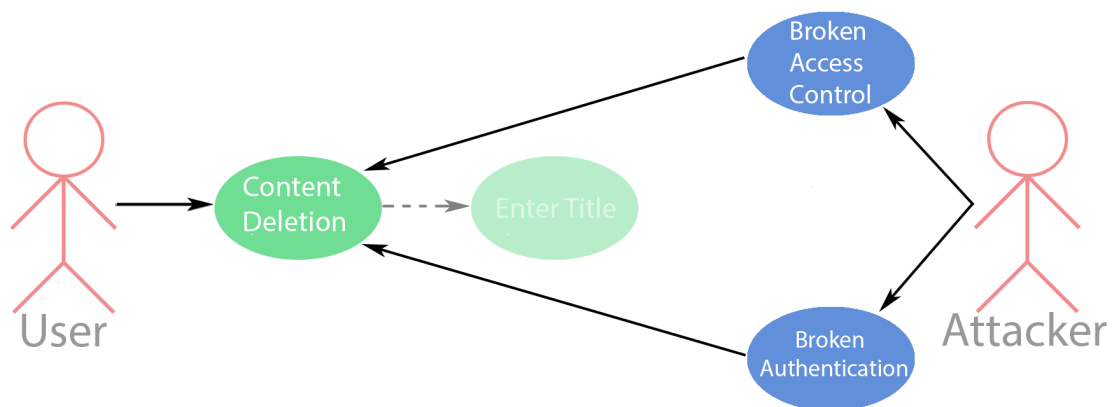


Figure 16: Content Deletion Abuse Case Diagram

Flow Description

Precondition

The application is hosted and running in an online environment.

Activation

Use case commences when a user engages with the post deletion action encompassed by a self-attributed post.

Main Flow

1. The user selects the post deletion action encompassed by a self-attributed submission.
2. The system displays a modal to the user. [E1: Cancellation | E2: Disabled JavaScript]
3. The user enters the title of the post as confirmation of its deletion [A1: Unfulfilled Validation].
4. The user submits the details as aforementioned for validation with the system.
5. The system deletes the existing post.
6. The system closes the modal and refreshes the page.
7. The use case ends successfully.

Alternative Flow

A1: Unfulfilled Validation

1. The system does not make the deletion action accessible. The form cannot be submitted.
2. The user satisfies the stipulation.
3. The use case continues from step 4 of the main flow.

Exceptional Flow

E1: Cancellation

1. The user ceases the post deletion process by performing a navigable action to another view or by closing the modal window presented by the system.
2. The system directs the user to the view in question or closes the modal.
3. The use case ends with a failure condition.

E2: Disabled JavaScript

1. The delete action becomes accessible regardless of entry of the post title.
2. The user does not enter a title, but proceeds to execute the delete action
3. The user submits the form for validation with the system.
4. The system processes the form server-side.
5. The system does not delete the post as the enforced stipulation was not satisfied.
6. The use case ends with a failure condition.

Termination

Scenarios

1. The user ceases the post deletion process.
2. The user deletes a post successfully.
3. The user removes client-side validation in an attempt to bypass the post title requirement.

Post Condition

Success

The post details, as provided, are validated by the system thusly deleting the post.

Failure

The system remains in the state prior to the users engagement with the post deletion process. Details that may or may not have been provided, are not logged with the system.

REQUIREMENT 8: CONTENT FILTERING

Description and Priority

Allows users to filter posts based on a variety of defined metrics related to the data associated with them (e.g. most popular based on votes, and newest or oldest based on timestamps).

Use Case

Scope

The scope of this use case is to allow for a user to be able to filter content.

Description

This use case describes the process through which a user must follow to filter content.

Use Case Diagram

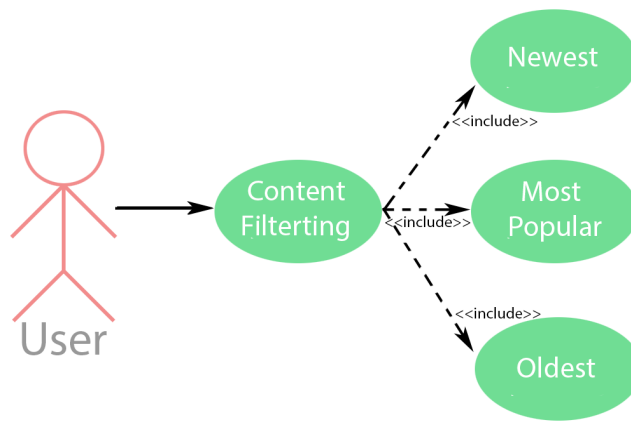


Figure 17: Content Filtering Use Case Diagram

Abuse Case Diagram

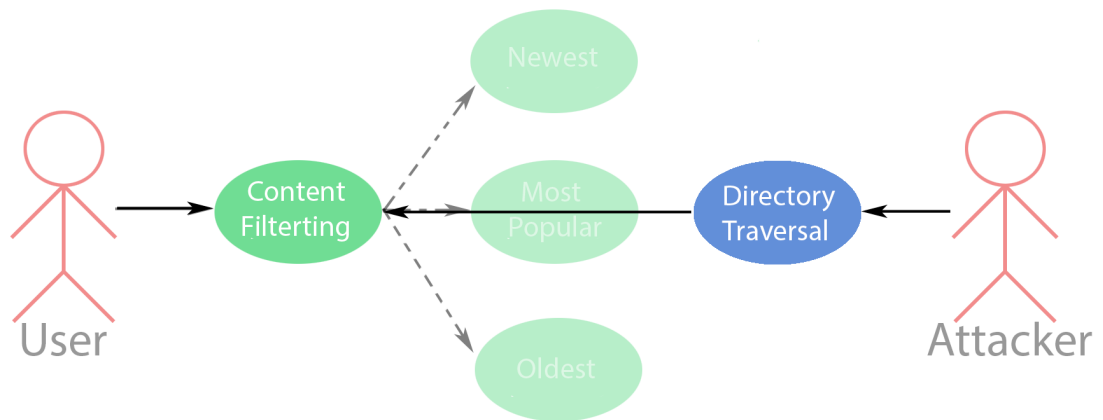


Figure 18: Content Filtering Abuse Case Diagram

Flow Description

Precondition

The application is hosted and running in an online environment.

Activation

Use case commences when a user engages with the filter content action located within the home view via dedicated dropdown menu.

Main Flow

1. The user selects the filter content dropdown menu.
2. The system reveals the optional metrics by which content may be filtered [E2: Cancellation].
3. The user selects one of the presented options.
4. The system dynamically alters the query, changing the order of displayed content in accordance to the metric selected by the user [A1: Pagination Request Input].
5. The use case ends successfully.

Alternative Flow

A1: Pagination Request Input

1. The system appends both filter and pagination request inputs to the URL.
2. The use case continues from step 5 of the main flow.

Exceptional Flow

E1: Cancellation

1. The user ceases the content filtering process by performing a navigable action to another view or by closing the associated dropdown as presented by the system.
2. The system directs the user to the view in question or closes the dropdown.
3. The use case ends with a failure condition.

Termination

Scenarios

1. The user ceases the content sorting process.
2. The user sorts content successfully.
3. The user sorts content and traverses pages, maintaining the selected sort option.

Post Condition

Success

The selected metric is validated by the system resulting in modification to the query governing how content is presented to the user. State is maintained during page traversal.

Failure

The system remains in the state prior to the users engagement with the content filtering process.

REQUIREMENT 9: CONTENT TAGGING

Description and Priority

Allows authenticated users to, during the submission stage, tag content with context relating to technical facets (e.g. HTML, CSS, JavaScript).

Use Case

Scope

The scope of this use case is to allow for a user to be able to tag posts.

Description

This use case describes the process through which a user must follow to tag a post during submission.

Use Case Diagram

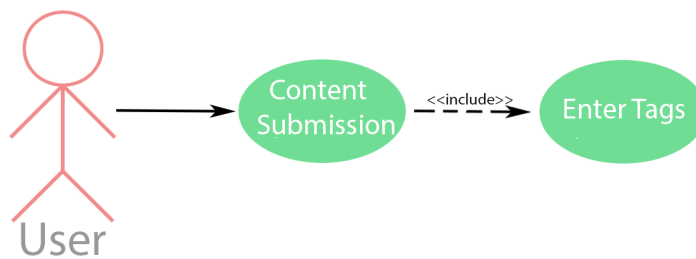


Figure 19: Content Tagging Use Case Diagram

Flow Description

Precondition

The application is hosted and running in an online environment.

Activation

Use case commences when a user engages with the content submission view of the application.

Main Flow

1. The user accesses the content submission or content modification views via their respective navigable elements.
2. The system directs the user to the view as selected [E1: Cancellation].
3. The user enters details [A1: Blank Fields].
4. The user submits the details as aforementioned for validation with the system.
5. The system creates a new post or updates an existing post accompanied by the tags as defined.
6. The system redirects the user to the homepage.
7. The use case ends successfully.

Alternative Flow

A1: Unfulfilled Validation

1. The system notifies the user any unfulfilled validation requirements.
2. The user satisfies the stipulation.
3. The use case continues from step 4 of the main flow.

Exceptional Flow

E1: Cancellation

From step 2 of the main flow:

1. The user ceases the content tagging process by performing a navigable action to another view.
2. The system directs the user to the view in question.
3. The use case ends with a failure condition.

Termination

Scenarios

1. The user submits a new post with accompanying tags successfully.
2. The user modifies an existing posts tags successfully.
3. The user ceases engagement with the creation or modification of a posts tags.

Post Condition

Success

The posts details, as provided, are validated by the system thusly creating or modifying a posts tag data. Tag records are then associated with a given post and displayed alongside them.

Failure

The system remains in the state prior to the users engagement with the content tagging process. Details that may or may not have been provided, are not logged with the system.

REQUIREMENT 10: CONTENT SEARCHING

Description and Priority

Allows users to search for specific site content based on defined metrics with respect to the search query (e.g. post title, post tags, user).

Use Case

Scope

The scope of this use case is to allow for a user to be able to search for specific site content via search query that encompasses details of a post or user.

Description

This use case describes the process through which a user must follow to search for content.

Use Case Diagram

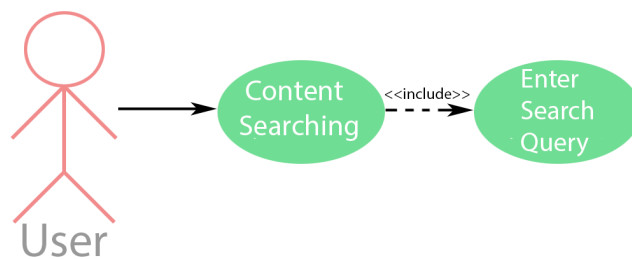


Figure 20: Content Searching Use Case Diagram

Abuse Case Diagram

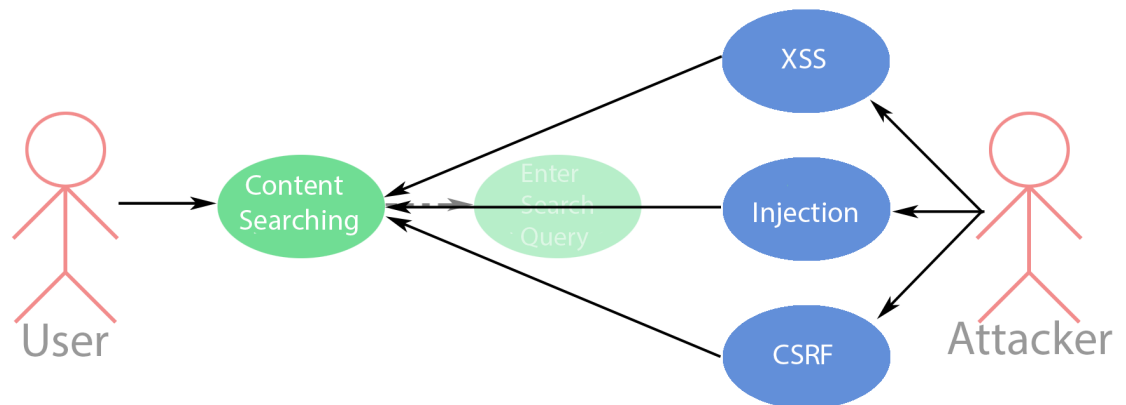


Figure 21: Content Searching Abuse Case Diagram

Flow Description

Precondition

The application is hosted and running in an online environment.

Activation

Use case commences when a user engages with the search input via the navigation menu.

Main Flow

1. The user accesses a view whereby the navigation menu is a component of the layout.
2. The user focuses on the search input inclusive of the navigation menu [E1: Cancellation].
3. The user enters details [E2: Empty].
4. The user submits the details as aforementioned for validation with the system.
5. The system returns post or user records that match or partially match the input as provided by the user.
6. The use case ends successfully.

Alternative Flow

-

Exceptional Flow

E1: Cancellation

1. The user ceases the content searching process by performing a navigable action to another view or revoking focus on the search input element.
2. The system directs the user to the view in question or does nothing.
3. The use case ends with a failure condition.

E2 Empty

1. The system does not execute a search query.
2. The system redirects the user to the page prior to submission of the form.
3. The use case ends with a failure condition.

Termination

Scenarios

1. The user executes a search query successfully.
2. The user executes a blank search query that does not retrieve records.
3. The user ceases engagement with the content searching process.

Post Condition

Success

The post or user details, as provided, are validated by the system thusly executing a valid search query. Returned records are reflective of full or partials matches to the provided input.

Failure

The system remains in the state prior to the users engagement with the content searching process. Details that may or may not have been provided, are not logged with the system.

REQUIREMENT 11: RATING SYSTEM

Description and Priority

Allows authenticated users to like content, revoke a like from previously liked content, but not dislike content.

Use Case

Scope

The scope of this use case is to allow for a user to be able to vote or revoke a vote on existing submitted content of their own or others.

Description

This use case describes the process through which a user must follow to vote on existing submitted content.

Use Case Diagram

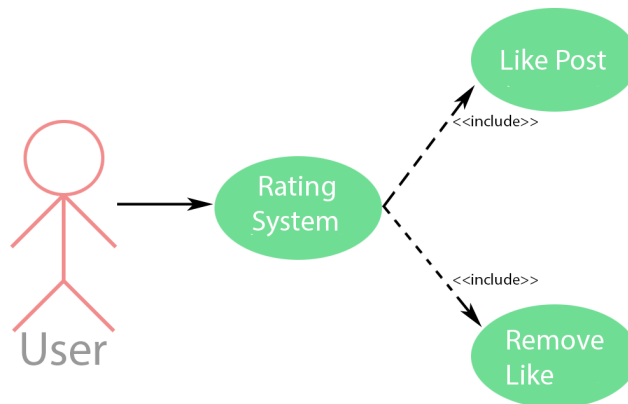


Figure 22: Login Use Case Diagram

Flow Description

Precondition

The application is hosted and running in an online environment.

Activation

Use case commences when a user engages with a view on the application containing post data (i.e. home page, expanded post, user profiles) that which is inclusive of the voting component.

Main Flow

1. The user accesses a view whereby post data is present that which is inclusive of the voting component.
2. The system directs the user to the view in question [E1: Cancellation].
3. The user executes a vote on a given post via click of the UI element representing it [E2: Guest User].
4. The vote is submitted for validation with the system [A1: Already Liked].
5. The system dynamically updates the state of the vote on a given post.
6. The use case ends successfully.

Alternative Flow

A1: Already Liked

1. The system deletes the record pertaining to the post previously voted on.
2. The use case continues from step 6 of the main flow.

Exceptional Flow

E1: Cancellation

From step 2 of the main flow:

1. The user ceases the rating process by performing a navigable action to another view or revoking focus on the element.
2. The system directs the user to the view in question or does nothing.
3. The use case ends with a failure condition

E2: Guest User

1. The system redirects the user to the login view of the application.
2. The use case ends with a failure condition.

Termination

Scenarios

1. The user ceases the rating process.
2. The unauthenticated user is prompted to login.
3. The user likes a post successfully.
4. The user revokes a like successfully.

Post Condition

Success

The users vote, as provided, is validated by the system thusly creating a new vote record or deleting an existing vote record. Vote records are then associated with a given post and displayed alongside them.

Failure

The system remains in the state prior to the users engagement with the rating process.

REQUIREMENT 12: COMMENTS/REPLIES

Description and Priority

Allows authenticated users to comment on existing posts or reply to existing comments encompassed by existing posts.)

Use Case

Scope

The scope of this use case is to allow for a user to be able to comment on a post and reply to a comment on a post.

Description

This use case describes the process through which a user must follow to comment on a post or reply to a comment.

Use Case Diagram

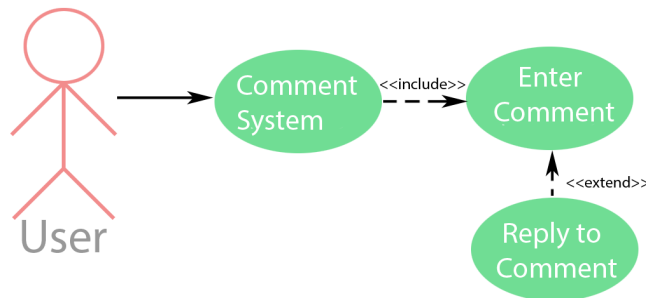


Figure 23: Comments/Replies Use Case Diagram

Abuse Case Diagram

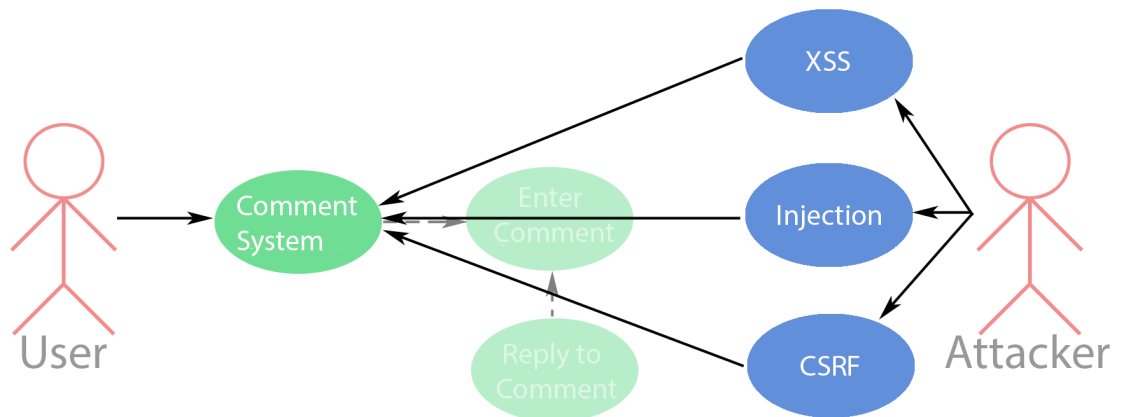


Figure 24: Comments/Replies Abuse Case Diagram

Flow Description

Precondition

The application is hosted and running in an online environment.

Activation

Use case commences when a user engages with the view of the application encapsulating expanded data pertaining to a single post record.

Main Flow

1. The user accesses a thread pertaining to a single post record.
2. The system directs the user to the view [E1: Cancellation].
3. The user enters details [A1: Unfulfilled Validation].
4. The user submits the comment as aforementioned for validation with the system [A2: Child Comment].
5. The system creates a new comment record equating to the input value as provided.
6. The system refreshes the view.
7. The use case ends successfully.

Alternative Flow

A1: Unfulfilled Validation

1. The system notifies the user any unfulfilled validation requirements.
2. The user satisfies the stipulation.
3. The use case continues from step 4 of the main flow.

A2: Child Comment

1. The system creates a new comment record equating to the input value as provided in addition to a value instructing that the record is a reply.
2. The use case continues from step 6 of the main flow.

Exceptional Flow

E1: Cancellation

1. The user ceases the comment/reply process by performing a navigable action to another view or revoking focus on the element.
2. The system directs the user to the view in question or does nothing.
3. The use case ends with a failure condition.

Termination

Scenarios

1. The user ceases the comment/reply process.
2. The user creates a comment successfully.
3. The user creates a comment reply successfully.

Post Condition

Success

The comment details, as provided, are validated by the system thusly creating a new comment record. This record is then associated with the given post.

Failure

The system remains in the state prior to the users engagement with the comment/reply process. Details that may or may not have been provided, are not logged with the system.

REQUIREMENT 13: NIGHT MODE

Description and Priority

Allows users to alter the sites presentation on the client-side to a theme that alleviates eye-strain and increases visibility during night time usage.

Use Case

Scope

The scope of this use case is to allow for a user to be able to change the thematic presentation of the application

Description

This use case describes the process through which a user must follow to change the thematic presentation of the application.

Use Case Diagram

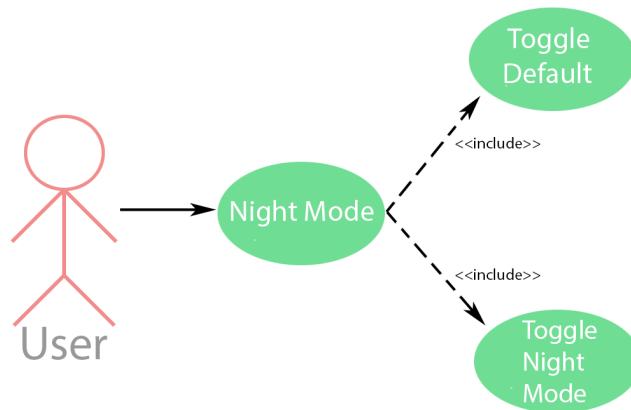


Figure 25: Night Mode Use Case Diagram

Flow Description

Precondition

The application is hosted and running in an online environment.

Activation

Use case commences when a user engages with the dropdown encompassed by the navigation menu pertaining to settings (variable options based on user type).

Main Flow

1. The user accesses the settings dropdown of the navigation menu [E1: Cancellation].
2. The system expands the dropdown to reveal the option pertaining to altering the sites theme.
3. The user executes the site theme toggle switch [A1: Night Mode | A2: Day Mode].
4. The system dynamically changes the sites theme.
5. The system creates a local storage variable representative of the toggled state.
6. The use case ends successfully.

Alternative Flow

A1: Night Mode

1. The system modifies the theme to night mode.
2. The use case continues from step 5 of the main flow.

A2: Day Mode

1. The system modifies the theme to day mode.
2. The use case continues from step 5 of the main flow.

Exceptional Flow

E1: Cancellation

1. The user ceases the process by performing a navigable action to another view or by closing the dropdown menu.
2. The system directs the user to the view in question or closes the dropdown menu.
3. The use case ends with a failure condition.

Termination

Scenarios

1. The user ceases the night mode process.
2. The user toggles the sites thematic presentation to night mode.
3. The user toggles the sites thematic presentation to the default.

Post Condition

Success

The system logs the toggled state in local storage or modifies the existing state if previously executed. This value determines the site current presentation.

Failure

The system remains in the state prior to the users engagement with the night mode process.

NON-FUNCTIONAL REQUIREMENTS

PERFORMANCE/RESPONSE TIME

Performance/response time with respect to Reflux refers to the process of conveying successful operation of the system from an end-user perspective. There are many use case scenarios whereby data is retrieved or manipulated on account of interaction from users. Users engage with frontend UI components that operate backend procedures, often executing controller methods that pertain to a singular or set of views. It is imperative that these processes not only conclude successfully with regards to their main flow and intended outcome, but to do so in a manner that comprehensively communicates that users have executed an action, to which the system responds both the processing and outcome of said action within a standardised timeframe that evokes successful operation from their perspective.

AVAILABILITY

Availability with respect to Reflux refers to the periods in which the application is hosted and running as intended. Aspects that affect availability pertain to circumstances whereby the aforementioned are not upheld. An attack on the system or scheduled maintenance are two scenarios in which the availability requirement of the system may be impeded. This is especially prevalent due to the nature of the platform by which Reflux occupies as a web application. Within reason, it is imperative that the system have a higher ratio of uptime than downtime while circumstances that may affect said goal are taken into consideration with regards to bolstering the system for either preventative or remediation reasons, and in ensuring the availability requirement is satisfied.

RECOVERY

Recovery with respect to Reflux refers to its dependencies on data-sets as pushed and retrieved from an associated database. Most of the functionality integral to the application involves getting, posting, patching, and destroying data in relation to the sites user submissions and data associated with said submissions. Additionally, this connection is what maintains Reflux's user-base. Thusly, recovery is an insurance that, should a circumstance befall the application which affects the integrity of the data or truncates it entirely, then there is a restoration procedure in place that remediates the occurrence, rebuilding an instance snapshot of the database as it was from within a previous timeframe.

RELIABILITY

Reliability with respect to Reflux refers to the ratio in which the outcome of a given executed function is successful. Thusly, it refers to the systems reliability in its functionality from an end-user perspective. It is to gauge the occurrence-rate of how well the application performs in a circumstance, and how well it communicates to users when the result of engagement with a feature results in an exception. If the exception does not originate from user error, then a given feature cannot be classified as reliable in its intended execution. An integral aspect to ensuring reliability across Reflux's processes is in automated unit testing.

MAINTAINABILITY

Maintainability with respect to Reflux refers to the difficulty of maintaining the application in relation to modifying it for purposes such as to patch bugs present in existing components. In this sense, it is specifically with reference to the source code with which the hosted application is derived from. In order to accomplish as such, it needs to be comprehensible. With Laravel, the application is compartmentalised into isolated sectors representative of the model view controller structure. This means that, in Reflux's case, sections of the site are isolated dependent on the contextual functionality they pertain to. Frontend refers to views, backend refers to controllers linked with elements of the former, and the model represents the sites database tables with which the methods from controllers communicate with. Thusly, in the circumstances as defined, the comprehensibility of maintaining the application is directly derived from the model view controller structure it employs.

EXTENSIBILITY

Extensibility with respect to Reflux refers to the difficulty of expanding the application in relation to introducing new features or components. Like maintainability, the ease or difficulty of satisfying this requirement is directly derived from how comprehensible the applications code-base is. In this sense, that means it directly mirrors the circumstances that define the fulfilment of the maintainability requirement, in that said comprehensibility is directly derived from the model view controller structure. Contextually, this could refer to the addition of a new model, views, or controllers. Specifically to Laravel, the inclusion of those as aforementioned is accomplishable via the PHP artisan command line tools which generates them, further amplifying the satisfaction of the extensibility requirement.

REUSABILITY

Reusability with respect to Reflux refers to recycling code in circumstances whereby its duplication otherwise results in redundancy. Specifically to Reflux, aspects of the user interface are global within a set of defined views such as navigation. In essence, if a component is reusable, and is therefore encompassed by multiple views, changes made to this component in the future should occur from a singular origin point. Changes are then reflected on a global scale within the views said component was present. Laravel employs the blade templating engine, which allows for the inclusion of layout and partial files. Reflux consists of these files whereby the most prominent cases of reuse locate within a master layout file comprising of these global components. Anything that differs from them are dynamically updated in accordance to the view.

SECURITY

Security with respect to Reflux refers to the applications imperative security implementations. For each functional requirement where a controller communicates with a model and returns a response to the frontend, security must be considered. In this context, the applications most prominent vulnerable endpoints relate to its manipulation of data, and the governance of features and views accessible dependant on the type of user a user is. While classified as non-functional inclusions, they define the stipulations that gate a user's access to executing actions they should or should not be able to execute in addition to validating what the system is provided in relation to user input. Following are the implementations as defined given context to their use case with respect to Reflux:

Access Control & Authorisation

Access control and authorisation refers to gating a user's access to somewhere or doing something based on their level of privilege. This defines the difference between a guest and authenticated user in Reflux. Middleware is used to do this and is employed directly within controllers that host the methods relating to a given route. When utilised, it decides which method within a controller a user type is and is not allowed to execute. This can refer to both features and views.

CSRF Protection

Cross-site Request Forgery is an attack that manipulates an authenticated user into executing actions they did not intend to execute. In Laravel, in any instance whereby a form is present, a CSRF hidden input must be instated. The framework forces this inclusion to protect against this vulnerability. It generates a token that is matched against the request. If both match, then it is occurring internally and is thus not an instance of CSRF.

Validation

Validating user input is the most prominent security layer employed by Reflux in reflection to its encompassed feature-set that comprises of many instances prompting as such. In each case, client-side validation via JavaScript is performed for the sole purpose of comprehensively communicating exceptions to users while, concurrently, the same validation is occurs in the backend within each controller. In addition, the nature of Reflux is, in part, the presentation of technical facets such as code blocks. Naturally, these are not executable, and thusly are transformed into character code alternatives safe for textual display via the PrismJS syntax highlighter and TinyMCE text editor.

DATA REQUIREMENTS

The following are representative of Reflux's data requirements as elicited prior to and during development:

Users Table	Posts Table	Votes Table	Tags Table	Comments Table
ID	ID	ID	ID	ID
Name	User ID	User ID	Post ID	User ID
Email	Score	Post ID	Name	Post ID
Password	Title	Type	Created at	Parent ID
Bio	Body	Created at	Updated at	Comment
URL	Created at	Updated at		Created at
Location	Updated at			Updated at
Created at				
Updated at				

USER REQUIREMENTS

The following are representative of Reflux's user requirements in reference to the functional requirements as outlined with respect to their operation. As such, a user should be able to:

- ❖ Register an account by providing the necessary details and satisfying enforced validation stipulations.
- ❖ Sign in to an existing account by providing the necessary details that match an existing record.
- ❖ Visit their own or other user profiles and access their own or other users submissions, liked posts, and comment history.
- ❖ Modify account settings such as name, email, URL, location, and bio.
- ❖ Submit a post posed as a query or solution with or without technical aspects (i.e. code blocks).
- ❖ Modify an existing post that is rightfully associated with them (i.e. ownership).
- ❖ Delete an existing post that is rightfully associated with them (i.e. ownership).
- ❖ Filter content in accordance with defined metrics, such as listing the most popular posts governed by their vote count.
- ❖ Tag posts with the language(s) that which define the context of the topics discussion and or the language(s) the post encapsulates.
- ❖ Search for a post or user based on their input whereby said input returns results that match or partially match it.
- ❖ Access views and execute actions that which is governed by their current user type (e.g. authenticated user = submit content, guest user = register an account, but not vice versa).
- ❖ Vote on existing posts and revoke an existing vote on a post at any time.
- ❖ Leave comments on existing posts and replies to existing comments.
- ❖ Change the aesthetic presentation of the application to better represent the time of day for easier usage within said circumstance with respect to viewing.

USABILITY REQUIREMENTS

The following are representative of Reflux's usability requirements that define its user experience with regards to its user interface:

Comprehension

- ❖ Basic UI components that expertly communicate the intention behind them.
- ❖ UI components are spaced accordingly, communicating further clarity of the system.
- ❖ The overarching purpose of the system with respect to the context of the application should additionally be communicated through the aforementioned.

Learnability

- ❖ Each functional requirement should briefly define its intention to the user through engagement with an action, but prior to committing to the execution of the action.
- ❖ Feature definitions should be non-intrusive, but communicate their intention fluidly through usage. Present prompted confirmation before foregoing committing to an action.
- ❖ As such, ensure the systems learning curve is accessible.

Presentation

- ❖ Approach the applications UI design through an established, tried and true methodology or standard (e.g. global font-weight and button sizes, material design methodology).
- ❖ Eliminate redundancy. Establish a set of reusable components as partials that which define a consistent theme and the identity of the site.

DESIGN AND ARCHITECTURE

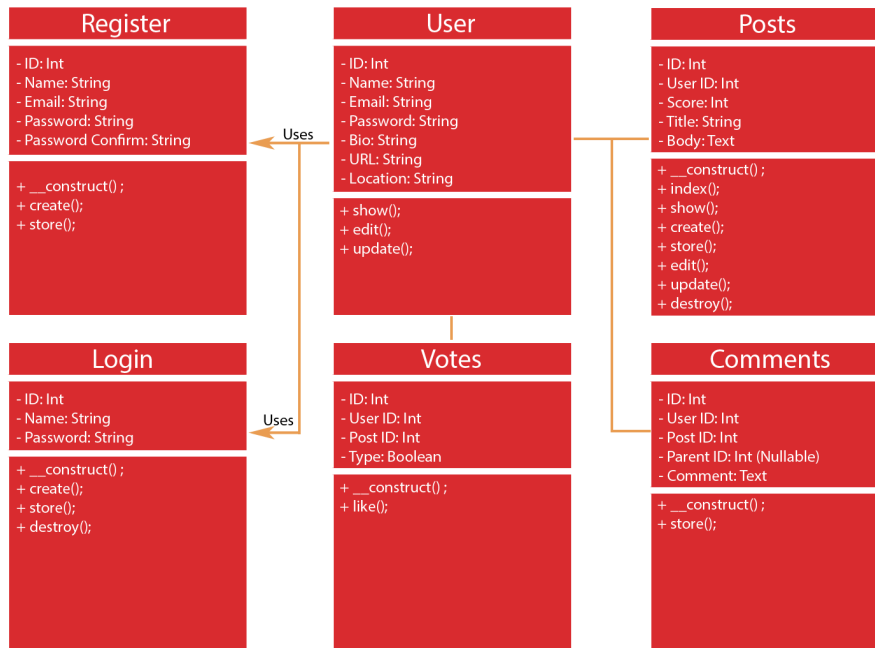


Figure 26: Reflux's Architecture

The system architecture depicted is representative of Reflux's models and controllers, and the associations drawn between them. The variables of each class mirror their respective database columns, and the methods mirror those encompassed by their respective controller. A class does not represent a single feature of the application explicitly, but rather is comprised of many methods that do so. For example, the methods of the post class describe each feature associated with posts, inclusive of those that only return views. The naming convention of these methods is reused in each class where applicable. The following define them:

- ❖ **__construct():** Defines which of the subsequent methods a guest or authenticated user may access.
- ❖ **index():** Viewing all posts on the home page. Additionally, sorting and searching.
- ❖ **show():** Showing a single post.
- ❖ **create():** Directs to the view that displays the form to create a new post.
- ❖ **store():** Stores a new post record in the database.
- ❖ **edit():** Directs to the view that displays the form to edit an existing post.
- ❖ **update():** Updates an existing post record in the database.
- ❖ **destroy():** Deletes an existing post record in the database.

IMPLEMENTATION

FRONTEND

BLADE TEMPLATING

Reflux's user interface boasts components that occur within multiple views for the same purpose. In a set of static pages, this would mean duplicating the code pertaining to the element within each case whereby it is used, inducing redundancy. This is problematic when considering maintainability, extendibility, and reusability. In a given circumstance, an element may be conducted in each of the three aforementioned processes, meaning that it must be engaged in each file that includes it. In Reflux, the blade templating engine is used to establish layout and partial files, meaning that an element need only be added or altered in a single location. This location is referenced in files that require the presence of this element. Thusly, changes are updated on a global scale.

MASTER LAYOUT FILE

```
    {{-- Navigation menu --}}
    @include('layouts.menu')

    {{-- Left and right content --}}
    <div class="ui grid stackable container">
        <div class="row" style="padding-top: 50px;">
            <div class="twelve wide computer sixteen wide mobile column"
id="sticky">

                {{-- Dynamic content --}}
                @yield('content')
            </div>
            <div class="four wide column computer only">

                {{-- Side menu --}}
                @include('layouts.side-menu')
            </div>
        </div>
    </div>

    <script src="{{ asset('js/prism.js') }}"></script>
    <script src="{{ asset('js/app.js') }}"></script>
```

Figure 27: Master Layout File

This illustrates an excerpt of Reflux's master layout file as encapsulated by body tags. It represents the elements present in each view in many cases. It itself comprises of partials representing isolated components, such as the navigation menu. The logic is that these components are *included* and do not change, while anything that is *yielded* represents dynamic content.

DYNAMIC CONTENT

```
@extends('layouts.master')

@section('content')
    {{-- Dynamic Content --}}
@endsection
```

Figure 28: Dynamic Content

In a file representing dynamic content (i.e. the content that changes depending on the view), the structure of a HTML document is not repeated, as this is an inclusion of the master layout file. Instead, the file extends the layout file and defines one or many sections of content that are dynamic. In the master layout file, the yielded section is entitled “content”, thusly the same naming convention is referenced here, which draws the association between both files and governs the content that is displayed within said section in accordance with its location as called within the layout file.

STATIC CONTENT

```
    {{-- Like button --}}
    <div class="hidden content">
        <form action="/vote/{{ $post->id }}" method="post" postid="{{ $post->id
}}">
            @csrf

            <input type="hidden" name="postid" value="{{ $post->id }}">
            <button class="ui icon {{ $post->votes->where('user_id', auth()-
>id()->where('post_id', $post->id)->first() ? 'blue' : 'basic blue' }} button vote
vote_type_{{ $post->id }}">
                <i class="thumbs up icon"></i>
            </button>
        </form>
    </div>
```

Figure 29: Static Content

In a file representing static content (i.e. content that does not change), the file is structured as an excerpt of a HTML component. Unlike dynamic content, there is no specialised format or syntax. The file is inclusive of the component as it would be written if it was included within the layout file itself manually. The above illustrates a section of the post component pertaining to the like button.

COMPILING ASSETS

LARAVEL MIX

```
/*
|-----|
| Mix Asset Management
|-----|
*/

mix.js('resources/assets/js/app.js', 'public/js')
  .sass('resources/assets/sass/app.scss', 'public/css');
```

Figure 30: Laravel Mix File

In the project code, everything is compartmentalized, inclusive and exclusive of the model view controller structure. With respect to assets, this refers to the sites stylesheets and JavaScript implementations. Laravel Mix defines an asset pipeline using common CSS and JavaScript pre-processors. In this case, these are Sass (Syntactically Awesome Stylesheets) and vanilla JavaScript. The file defines two paths for each case, one representing an input file where production takes place, and one representing an output file.

SASS

```
@import "partials/colors";

/*
|-----|
| Modals
|-----|
*/

/* Delete modal */
.ui.mini.modal.delete_modal {
  background: transparent;

  .header {
    color: $white;
    background: $red;
  }
}
```

Figure 31: Sass Main File

This above is a snippet of Reflux's main Sass file illustrating compartmentalisation in practice. For each post that a currently authenticated user owns, they may access a delete modal window. The styling of this modal is represented, but utilises pre-defined thematic variables pulled from an external partial Sass file pertaining purely to colours.

```

/*
|-----|
| Base Colors
|-----|
*/
$white: #FFFFFF;
$black: #1B1C1D;
$grey: #ECF0F1;
$red: #DB2828;
$blue: #2185D0;
$bg-color: #EEF2F5;

```

Figure 32: Sass Partial File

Within the partial Sass file, are colour variables relating directly to those as utilised within the main file, thusly segregating aspects of the sites styling contextually. Similarly to blade templating, maintenance of an elements styling only requires the value of the variables it uses to be changed, resulting in it updating globally.

JAVASCRIPT

```

/* Modules */
require('./modules/ajax.js');
require('./modules/dropdown.js');
require('./modules/popup.js');
require('./modules/preloader.js');
require('./modules/theme.js');
require('./modules/tinymce.js');

```

Figure 33: JavaScript Main File

The same logic as discussed applies to compiling Reflux's JavaScript. The above is the entirety of the input file. Each JavaScript component is segregated. These are then all pulled into one file and compiled to an output JavaScript file that combines all of these modules, essentially making it seem as though they were written within one file. This enables ease-of-access to these components in isolation when considering maintainability and extendibility. The following represents one such segregated component that is compiled to the main file.

```

/*
|-----|
| Dropdowns
|-----|
*/

$('.nav-dropdown')
  .dropdown({
    action: 'select'
  })
;

```

Figure 34: JavaScript Partial File

SYNTAX HIGHLIGHTING

Syntax highlighting is handled via the implementation of PrismJS, and is used synonymously with the TinyMCE rich text editor. Combined, users are able to input code blocks within the confines of the languages provided. Given their selected option and inserted code block, it is then coloured appropriately to fit the context of the language it pertains to.

```
<pre class="language-markup">
  <code>Markup Post!</code>
</pre>
```

Figure 35: Syntax Highlighting

Within a post, this is how a code block is represented. The assigned class is defined given the users language selection as previously discussed. In this case, the code block pertains to mark-up, such as HTML or XML. The list is defined within the initialisation of the TinyMCE text editor via JavaScript:

```
tinymce.init({
  selector: '.mce_field',
  menubar: false,
  branding: false,
  plugins: 'codesample',
  codesample_languages: [
    {text: 'HTML/XML', value: 'markup'},
    {text: 'CSS', value: 'css'},
    {text: 'JavaScript', value: 'javascript'},
    {text: 'PHP', value: 'php'},
    {text: 'JSON', value: 'json'},
    {text: 'Java', value: 'java'},
    {text: 'Git', value: 'git'},
    {text: 'Ruby', value: 'ruby'},
    {text: 'Python', value: 'python'},
    {text: 'C#', value: 'csharp'},
    {text: 'Objective-C', value: 'objectivec'},
    {text: 'Perl', value: 'perl'}
  ],
  codesample_dialog_height: 400,
  codesample_dialog_width: 400,
  toolbar: 'undo redo | styleselect | bold italic | blockquote | codesample'
});
```

Figure 36: TinyMCE Language Array

ELOQUENT: RELATIONSHIPS

Models represent elements of the database. Reflux comprises of the following:

- ❖ User
- ❖ Post
- ❖ Tag
- ❖ Comment
- ❖ Vote

With respect to eloquent relationships in Laravel, they define as such between tables that, in turn, allow Reflux to present data in tandem based on these associations. For example, the user model represents an authenticated user. Users of the site can post content, therefore, that post must be attributed to them in some way. They have ownership of the content they have submitted. This relationship is reflected both in the user and post models, as well as the logic employed when these records are created in the controllers that relate to them. This creates a one-to-many relationship between a user and posts, and a one-to-one relationship between a post and a user. In essence, a user is allowed to own many posts, but a singular post can only be owned by one user. This logic is used when querying in specific circumstances, such as retrieving post records for a user profile.

The following exhibits the associations drawn between each of the defined models:

USER MODEL

```
/* User has many posts */
public function post()
{
    return $this->hasMany(Post::class);
}

/* User has many votes */
public function votes()
{
    return $this->hasMany(Vote::class);
}

/* User has many comments */
public function comments()
{
    return $this->hasMany(Comment::class);
}
```

Figure 37: User Model

The user model exhibits the relationship the user has with the other models. They can have many posts, comments, and votes.

POST MODEL

```
/* Post belongs to a user */
public function user()
{
    return $this->belongsTo(User::class);
}

/* Post has many votes */
public function votes()
{
    return $this->hasMany(Vote::class);
}

/* Post has many comments */
public function comments()
{
    return $this->hasMany(Comment::class);
}

/* Post has many tags */
public function tags()
{
    return $this->hasMany(Tag::class);
}
```

Figure 38: Post Model

The post model exhibits the relationship posts have with the other models. They belong to a user singularly, and can have many votes, comments, and tags.

TAG MODEL

```
/* Tag belongs to a post */
public function post()
{
    return $this->belongsTo(Post::class);
}
```

Figure 39: Tag Model

The tag model exhibits the relationship tags have with posts singularly. In that sense, they only belong to a post.

COMMENT MODEL

```
/* Comment belongs to a user */
public function user()
{
    return $this->belongsTo(User::class);
}

/* Comment belongs to a post */
public function post()
{
    return $this->belongsTo(Post::class);
}

/* Comment has many of itself */
public function replies()
{
    return $this->hasMany('App\Comment', 'parent_id');
}
```

Figure 40: Comment Model

The comment model exhibits the relationship comments have with the other models. They belong to a user and a post singularly, and can have many replies, which is a self-referential definition. Replies are also comments, thusly there is no differentiation between them. In essence, it means that comments themselves may also comprise of many other comments. What actually distinguishes them, occurs in controller logic.

VOTE MODEL

```
/* Vote belongs to a user */
public function user()
{
    return $this->belongsTo(User::class);
}

/* Vote belongs to a post */
public function post()
{
    return $this->belongsTo(Post::class);
}
```

Figure 41: Vote Model

The vote model exhibits the relationship votes have with the other models. They belong to a user and a post singularly.

ROUTING

Routes refer to Reflux's endpoints. In Laravel, these are defined and used to determine which method within a given controller is executed based on the type of request the endpoint receives. The request type, the route, and the controller, inclusive of the method within said controller, are specified.

```
/* Auth */
Route::get('/register', 'RegistrationController@create');
Route::post('/register', 'RegistrationController@store');
```

Figure 42: Registration Routes

In the above, the routes pertaining to the registration view are defined, but the contexts differ:

- ❖ If it is a GET request, the *create* method is executed.
- ❖ If it is a POST request, the *store* method is executed.

```
class RegistrationController extends Controller
{
    /* GET Request */
    public function create()
    {

    }

    /* POST Request */
    public function store()
    {

    }
}
```

Figure 43: Blank Registration Create and Store Methods

The above illustrates how these methods appear within the Registration Controller. Contextually, they relate to displaying the registration form and storing a new user record, respectively. This process applies for every route and request type.

SCALABILITY

LANGUAGES

In its basest form, the project is inclusive of a plethora of common languages and formats commonly implemented in solutions of the first type, such as JSON. Notably, the PrismJS syntax highlighter supports many more than are presented by default, and their inclusion in the system is as simple as appending a language or format to the list currently in use. In that sense, this component of Reflux's system is easily extendible. Users select the language of a code-block from a dropdown list when creating a post, but this list is not hardcoded. Instead, it is pulled from a JSON file of languages using the storage component of Laravel. In essence, just like passing queried results to a view, so is the JSON array of languages passed.

```
public function create()
{
    $languages = Storage::disk('local')->get('languages.json');
    $languages = json_decode($languages, true);

    return view('posts.create', compact('languages'));
}
```

Figure 44: Storage: Retrieving a JSON Array of Languages

Since users are presented this information when creating a post, it thusly occurs within the Post Controllers create method (i.e. the view that presents the post submission form). A less integral use of this method is also elicited for a list of countries, as users may update their location via their profile settings page:

```
public function edit(User $user)
{
    $countries = Storage::disk('local')->get('countries.json');
    $countries = json_decode($countries, true);

    return view('profiles.edit', compact('countries'));
}
```

Figure 45: Storage: Retrieving a JSON Array of Countries

SECURITY

AUTHENTICATION

Authentication is governed by two controllers, the Registration Controller and Sessions Controller. Each refer to creating a new account and accessing an existing account, respectively. Laravel provides a default authentication foundation with which to build the methods required to accomplish as such, simplifying the process when viewed comparatively to base PHP.

REGISTRATION

```
class RegistrationController extends Controller
{
    /* Show Registration Form */
    public function create()
    {
        return view('registration.create');
    }

    /* Register New User */
    public function store()
    {
        $user = User::create(
            [
                'name'      => request('name'),
                'email'     => request('email'),
                'password' => Hash::make(request('password'))
            ]
        );

        auth()->login($user);

        return redirect('/');
    }
}
```

Figure 46: Creating a New User Record

The create method presents users with the registration form, while the store method takes an instance of the user model, and creates a new user record based on the request input values provided via input. The Laravel authentication provided takes the created user model assigned to the variable user, and uses these details to login to the newly created account following its registration.

LOGIN

```
class SessionsController extends Controller
{
    /* Show Login Form */
    public function create()
    {
        return view('sessions.create');
    }

    /* Login to Account */
    public function store()
    {
        if(!auth()->attempt(request(['name', 'password']))) {
            return back();
        }

        return redirect('/');
    }

    /* Logout of Account */
    public function destroy()
    {
        auth()->logout();

        return redirect('/');
    }
}
```

Figure 47: Logging into an Account

The create method presents users with the login form, the store method attempts authentication on account of the details provided by the user, and the destroy method terminates a session. If there is no associated account relating to the details provided, users are presented the login form again, otherwise the process is successful.

RETRIEVING THE AUTHENTICATED USER

```
/* Get the Authenticated User: Method 1 */
$user = Auth::user();

/* Get the Authenticated User: Method 2 */
$user = auth()->user();

/* Get the Authenticated Users ID: Method 1 */
$user = Auth::id();

/* Get the Authenticated Users ID: Method 2 */
$user = auth()->id();
```

Figure 48: Retrieving an Authenticated User

LOGGING OUT

Logging out uses the destroy method of the Sessions Controller. There are two use cases of this method. The first is in deleting data, and the second relates to destroying the current user session (i.e. logging out). The latter is as such:

```
public function destroy()
{
    auth()->logout();

    return redirect('/');
}
```

Figure 49: Sessions Controller: Logging Out

MIDDLEWARE

ACCESS CONTROL AND AUTHORISATION

Reflux employs middleware in each of its controllers that governs what a given user type can and cannot do, and where they can or cannot go.

```
class RegistrationController extends Controller
{
    public function __construct()
    {
        $this->middleware('guest');
    }
}
```

Figure 50: Registration Controller: Middleware Constructor

A constructor method encapsulates the middleware, which is executed when a user of any type sends a request to a route. In this case, the middleware stipulates that only a guest user has access to the methods contained within the Registration Controller. In other words, an authenticated user cannot execute these methods.

```
class PostsController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth')->except('index', 'show');
    }
}
```

Figure 51: Posts Controller: Middleware Constructor

The Posts Controller introduces more complexity to this logic. Regardless of a user's level of privilege, they should be able to view posts in totality and singularly. This is dictated by the inclusion of the exception provider, which takes the method names to be exceptions as parameters, meaning that a guest user does not hold the privilege to submit, modify, or delete content, but they may view content. Both the index and show methods represent as such.

CSRF (CROSS-SITE REQUEST FORGERY) PROTECTION

Laravel trivialises CSRF attacks by enforcing a stipulation in any instance where a form is present within the application. Said stipulation is the inclusion of a CSRF flag via blade syntax.

```
{{-- Registration Form --}}
<form action="/register" method="post" class="ui form">

    {{-- Generates a Hidden Input --}}
    @csrf

</form>
```

Figure 52: CSRF: Blade Flag

This creates a hidden input that generates a CSRF token related to the active user session. When an authenticated user makes a request, it is used to verify that it is actually them who have done so by comparing the token.

```
{{-- Registration Form --}}
<form action="/register" method="post" class="ui form">

    {{-- Generates a Hidden Input --}}
    @csrf

    {{-- Example of the Generated Input and Token --}}
    <input type="hidden" name="_token"
value="iwSpWZHUCKlo8qmUUtH9c2EGjzwCSj0JPZcdg4AM">

</form>
```

Figure 53: CSRF: Example of the Generated Input

HASHING

PASSWORDS

Password hashing occurs solely within the Registration Controller using the Laravel Hash façade. By default, the hashing mechanism of this façade is Bcrypt. It occurs when a new user record is being created with the controllers store method whereby the request input value pertaining to the password field is intercepted.

```
$user = User::create([
    'name'      => request('name'),
    'email'     => request('email'),
    'password' => Hash::make(request('password'))
]);
```

Figure 54: Password Hashing using the Hash Façade

GRAVATAR

Reflux employs Gravatar, which generates a unique user profile image provided an email hash. It is not related to any module sported by Laravel, and is required to be an MD5 hash.

```
/* Gravatar based on users hashed email */
public function getGravatarAttribute()
{
    $hash = md5(strtolower(trim($this->attributes['email'])));
    return "http://www.gravatar.com/avatar/$hash?s=400&d=retro";
}
```

Figure 55: Gravatar Email Hashing using MD5

SENSITIVE DATA EXPOSURE

This mainly pertains to user data, and occurs within Reflux's user model. When data is retrieved relating to the user for display in the frontend, this could include something as harmless as a name or email, or return an array of user data in relation to a given record. In the case of the latter, in general circumstances this could potentially include the hash for a given user records password. Although manipulation of the data presented on the frontend is not applicable, protection against such an instance is employed.

```
{
  "id":1,
  "name":"Se\u00e1n",
  "email":"seanbickmore@icloud.com",
  "bio":"Welcome to Reflux, my fourth year Cyber Security project!",
  "url":null,
  "location":null,
  "created_at":"2018-05-11 21:05:41",
  "updated_at":"2018-05-11 21:05:41"
}
```

Figure 56: User Record

The above pertains to a retrieved user record. Notably, there is no password or remember token, which are inclusions of the user table. This is achieved via a method of the user model.

```
/**
 * The attributes that should be hidden for arrays.
 *
 * @var array
 */
protected $hidden = [
    'password', 'remember_token'
];
```

Figure 57: Hidden Attributes

It is within this method that attributes may be defined which are not retrieved in an instance similar to the above. Thusly, in a scenario if such a vulnerability were applicable, these details would be excluded from the record.

VALIDATION AND ERROR HANDLING

Any instances where user input is taken, said input is validated via JavaScript on the client-side, and within the respective controllers pertaining to the given circumstance on the server-side, concurrently. The former is used to communicate exceptions to the user if provided input does not match a defined specification of the data the application expects to receive. The latter mirrors validation in the backend to ensure provided input satisfies the specification before executing any queries.

CLIENT-SIDE: JAVASCRIPT

JavaScript validation is a provisioned component of the Semantic-UI CSS framework. Take the following excerpt of the validation JavaScript file:

```
$(document).ready(function(){
  $('.ui.form')
  .form({
    fields: {
      name: {
        identifier: 'name',
        rules: [
          {
            type : 'empty',
            prompt : 'Please enter a username'
          }
        ]
      }
    }
  })
});
```

Figure 58: JavaScript: Validation

The above illustrates capturing a given form, defining fields, the stipulations associated with those fields, and the exception returned to users should the stipulation not be satisfied. In this case, if the user leaves the name field blank, an appropriate message representing this exception is presented to them. This is the process elicited in all cases via the inclusion of additional forms and encompassed fields.

SERVER-SIDE: CONTROLLERS

In Laravel, validation typically occurs as part of a given controllers store method directly preceding an instance whereby a new record is created or modified. An array of request items is defined conjunctively with attached associated attributes that dictate the enforced validation requirements for each input. In the section pertaining to authentication, validation of the users provided input when creating an account was excluded. Take its inclusion:

```
class RegistrationController extends Controller
{
    public function store()
    {
        $this->validate(
            request(),
            [
                'name'      => 'required|unique:users',
                'email'     => 'required|unique:users|email',
                'password' => 'required|confirmed|min:10'
            ]
        );

        $user = User::create(
            [
                'name'      => request('name'),
                'email'     => request('email'),
                'password' => Hash::make(request('password'))
            ]
        );

        auth()->login($user);

        return redirect('/');
    }
}
```

Figure 59: Registration Controller: Validation

Prior to execution of the query, each request input value is evaluated against a number of definable attributes. Those as presented represent the following:

- ❖ Required: Field cannot be empty.
- ❖ Unique [Table]: Value must be unique when checked against the database.
- ❖ Email: Value must be of a valid email format.
- ❖ Confirmed: Field must have an associated confirmation input that matches its value.
- ❖ Min [Length]: Value must meet a minimum length.

This process of validation is followed whenever user input is taken, such as with posting content.

REGULAR EXPRESSIONS

Regular expressions define a search pattern with respect to stipulating more complex validation enforcements in Reflux. In context, this can be seen in how the application validates user passwords when creating a new account both on the client and server-side. Said stipulation is that a provided password must contain at least 10 characters, and be inclusive of a lowercase, uppercase, numeric, and special character, singularly at a minimum.

```
password: {
  identifier: 'password',
  optional : 'true',
  rules: [
    {
      type : 'empty',
      prompt : 'Please enter a password'
    },
    {
      type : 'regExp[^(?=.*[a-z])(?=.*[A-Z])(\\w{10})(?=.*[!@#$%^&*.,?]).+$]',
      prompt : 'Regex Message!'
    }
  ]
}
```

Figure 60: JavaScript Regular Expression

The above enforces this validation in JavaScript in order to present the user the password requirements without allowing form submission to go through. Notably, this differs slightly from the expression via PHP, as in the cases of defining the minimum characters and numeric requirements, these require backslashes. Singularly, JavaScript removes two backslashes, thusly this is accounted for so that one may remain for the purposes of the regular expression to function as intended.

```
$this->validate(
  request(),
  [
    'name'      => 'required|unique:users',
    'email'     => 'required|unique:users|email',
    'password'  => 'required|confirmed|min:10|regex:/^(?=.*[a-
z])(?=.*[A-Z])(?=.*\d)(?=.*[!@#$%^&*.,?]).+$/'
  ]
);
```

Figure 61: PHP Regular Expression

This represents the validation that occurs within the Registration Controller inclusive of the addition of the regular expression. In this case, backslashes need not be accounted for with respect to the aforementioned issue present in JavaScript strings. It does however, require a backslash that precedes and follows the regular expression in PHP. These are called delimiters, which specifies a boundary. Functionally, both are identical and enforce additional measures to ensure that users provide passwords with an innate, relative strength.

In addition to those as illustrated, the following represent Reflux's full implementation requirements enforced by regular expressions in all contexts:

❖ **Registration**

- Name: Must be one word, contain no special characters, and contain no spaces.
- Password: Must be at least 10 characters and contain at least 1 uppercase, lowercase, numeric, and special character.

❖ **Post Creation**

- Title: Cannot contain any special characters.

❖ **User Settings**

- Name: Must be one word, contain no special characters, and contain no spaces.
- URL (Optional Field): Cannot contain HTTPS/HTTP preceding the URL, must contain at least one period, and cannot end with a period.

DATABASE

DATABASE: QUERY BUILDER

Laravel simplifies the database querying process. Behind the convenience of its syntax, the built queries use PDO parameter binding, which enforces protection against injection related vulnerabilities. The query builder is integral to fulfilling Reflux's functional requirements that create database records based on user input, and its non-functional security requirements when considering malicious input.

GETTING DATA

The largest use case of retrieving data in Reflux is in relation to posts. Posts govern the sites content. In addition, they illustrate each aspect of retrieval elicited in the application. For instance, everything is retrieved, everything is retrieved and sorted a specific way, and a singular record is retrieved.

```
class PostsController extends Controller
{
    public function index(Request $request)
    {
        /* Retrieve all Posts */
    }

    public function show(Post $post)
    {
        /* Retrieve a Single Post */
    }
}
```

Figure 62: Posts Controller: Blank Index and Show Methods

Thusly, the above are representative of the methods that achieve the aforementioned where index retrieves all posts and show retrieves one post.

```
/* GET data */
$post = Post::all();

/* Pass data to view */
return view('posts.index', compact('posts'));
```

Figure 63: Retrieving all Posts and Passing to the View

This showcases the ease of retrieving all of a resource without limiting the query in any way. The result is then passed to the view that utilises the retrieved data for presentation to the user in the frontend.

Retrieving a specific record in isolation requires the use of route model binding in Laravel, which uses a defined route and the model in question pertaining to the resource to be retrieved.

```
/* Route for a Single Post via Wildcard */  
Route::get('/posts/{post}', 'PostsController@show');
```

Figure 64: Route for a Single Post using a Wildcard

This represents the route to execute the show method for expanding the details of a post that the user selects from a list of all posts. This differs from other routes that present a static view, in that the URL utilises what is known as a wildcard. A wildcard represents something that differentiates one record from another. By default, this will be a table's ID column. Therefore, if a user selects a post, the URL will equate to the post route and ID of the selected post. Thusly, the appropriate data pertaining to that post is retrieved.

```
public function show(Post $post)  
{  
    if (!$post->exists()) {  
        return redirect('/');  
    }  
  
    return view('posts.show', compact('post'));  
}
```

Figure 65: Posts Controller: Showing a Single Post via Route Model Binding

When using route model binding, a query to retrieve a post singularly does not need to be defined. The model itself is representative of these resources in isolation. By passing the model as a parameter to the show method, it can be used in this way. It encompasses the data pertaining to single records based on the ID the wildcard is equal to upon navigation. In this case, it is also stipulated that if a post does not exist (i.e. an ID is entered manually in the address bar that doesn't relate to an existing post record), the user is redirected to the home page.

POSTING DATA

Posting data is functionally identical to the creation of a new user record as illustrated in the section pertaining to authentication. Contextually speaking however, posts are the prime use case of posting data within Reflux. It involves two methods: One that presents the user a form in the guise of a view, and another that takes the input from this view and creates a new record. As previously outlined, these are the create and store methods, following the established method naming convention. With respect to posts, there are three facets to consider:

- ❖ Validation
- ❖ Creation of the Post Record
- ❖ Accounting for Multiple Tags

```
public function store(Request $request)
{
    $this->validate(
        request(),
        [
            'title' => 'required|max:50',
            'body'  => 'required',
            'tags'  => 'required'
        ]
    );

    $post = Post::create(
        [
            'title'    => request('title'),
            'body'     => request('body'),
            'score'    => 0,
            'user_id' => auth()->id()
        ]
    );

    $tags = $request->input('tags');
    foreach ($tags as $tag) {
        Tag::create(
            [
                'post_id' => $post->id,
                'name'    => $tag
            ]
        );
    }

    return redirect('/');
}
```

Figure 66: Posts Controller: Validating, Creating, and Tagging a Post

The request input values are evaluated against the defined stipulations. The record is then created. Notably, in this instance, since a post must be attributed to the authenticated user, the user ID isn't set within the form, but the controller. Thusly, it equates to the authenticated users ID. Tags are a separate table entirely, but have a relationship with the posts table. When a post is created, tags are a required request input. They are a pre-set list of languages. Depending on the quantity, they are looped through, thus creating a new record for each.

UPDATING DATA

Updating data follows similar conventions to posting data, with the exception that it alters an existing record instead of creating one. However, validation still occurs prior to the modification query. Just like when posting data, there are two methods that represent showing a form and modifying the data. In this case, the edit method displays the view, while the update method executes the query.

```
public function edit(Post $post)
{
    $user = User::where('id', auth()->id()->first());

    if($post->user_id !== auth()->id()) {
        return redirect('/');
    }

    return view('posts.edit', compact('user', 'post'));
}
```

Figure 67: Posts Controller: Edit Method

The edit method displays a form for modifying a post by using route model binding, the same logic that applies when viewing a post singularly. In essence, the URL to edit a given post utilises the posts ID as a wildcard. In this instance, users are redirected away from this view if the user ID attached to the post does not match that of the authenticated user. Regardless, the modification query only allows for instances where the authenticated users ID equates to the posts user ID column, thus attributing the post to them.

```
public function update(Post $post)
{
    $this->validate(
        request(),
        [
            'body' => 'required',
        ]
    );

    $post = Post::where('id', $post->id)->where('user_id', auth()->id()-
>update(
    [
        'body' => request('body')
    ]);

    return redirect('/');
}
```

Figure 68: Posts Controller: Updating a Post

With respect to the data pertaining to the post table in isolation (i.e. not tags), the user is only allowed to modify the body of the post. This ensures that they cannot misrepresent the original context of the content.

DELETING DATA

Deleting a record is represented in controllers by the destroy method. It is one of the use cases of this method alongside destroying a user session. In the case of posts, this means deleting records located in three interlinking tables: Posts, Votes, and Tags. The latter two are associated with a post via their respective post ID columns. When a post is removed, its ID will never be used by another post again, but removing all of its associations regardless is an orderly methodology to follow.

```
public function destroy(Post $post)
{
    $this->validate(
        request(),
        [
            'post_title' => 'required',
            Rule::in([$post->title])
        ]
    );

    Post::where('id', $post->id)->where('user_id', auth()->id()->delete();
    Vote::where('post_id', $post->id)->delete();
    Tag::where('post_id', $post->id)->delete();

    return redirect()->back();
}
```

Figure 69: Posts Controller: Deleting a Post

In Reflux, an additional confirmation step is asked of users to ensure that the post they are trying to delete is an intentional action. This is done by validating the title of the post, provided as input. It must equate to the title of the post they are trying to delete. This is a similar implementation to how GitHub handles deletion of repositories.

DISPLAYING DATA IN THE FRONTEND

```
@foreach($posts as $post)
    {{-- Post Table --}}
    {{ $post->title }}
    {{ $post->body }}

    {{-- Tags Table --}}
    @foreach ($post->tags as $tag)
        {{ $tag->name }}
    @endforeach
@endforeach
```

Figure 70: Displaying Data Passed to the View

Post records are looped through similarly to the above implementation with the exclusion of the HTML elements they are attached to. In addition, tags can be retrieved simply by referencing their association to the post model.

DATABASE: PAGINATION

When illustrating the retrieval of all posts via Laravels query builder, Reflux's exact query was not represented. In context, Reflux presents a list of all posts to users up to a defined maximum per page. As the application has the potential to display an abundance of posts built by the community, querying a exponentially increasing large data-set at once affects the performance and response time requirement of the application.

QUERY BUILDER: PAGINATION

```
public function index(Request $request)
{
    $user = User::where('id', auth()->id()->first();
    $posts = Post::latest()->paginate(5);

    $sort = $request->input('sort');
    switch ($sort) {
        case 'top':
            $posts = Post::orderBy('score', 'desc')->paginate(5);
            break;
        case 'new':
            $posts = Post::latest()->paginate(5);
            break;
        case 'old':
            $posts = Post::orderBy('created_at', 'asc')->paginate(5);
            break;
        default:
    }

    return view('posts.index', compact('user', 'posts', 'tags'));
}
```

Figure 71: Posts Controller: Paginating Results

Pagination is essentially a request input variable appended to the end of a given navigable URL that determines returned records based on its value. In Reflux, this is used for pagination and sorting content conjunctively. On its own, pagination results in the same outcome as would be expected by limiting the amount of records returned. In essence, the query itself does not create the necessary frontend components that execute page traversal. This facet is represented by a blade partial:

```
@if ($paginator->lastPage() > 1)
    <div class="ui inverted teal pagination menu">
        @for ($i = 1; $i <= $paginator->lastPage(); $i++)
            <a href="{{ $paginator->url($i) }}" class="{{ ($paginator->currentPage() == $i) ? 'active ' : ' ' }}" item">{{ $i }}</a>
        @endfor
    </div>
@endif
```

Figure 72: Frontend: Pagination Partial

This calculates the amount of pages to generate based on number of posts per page defined in the controller.

DATABASE: SEEDING

In order to create dummy data representative of posts and users, Laravel provides a Database Seeding class that seeds migrated tables via the PHP artisan command line tools during said process. From a development perspective, this allows for lossless data-sets when considering maintenance and extensibility whereby new tables are introduced, for example.

```
class UsersTableSeeder extends Seeder
{
    public function run()
    {
        DB::table('users')->delete();

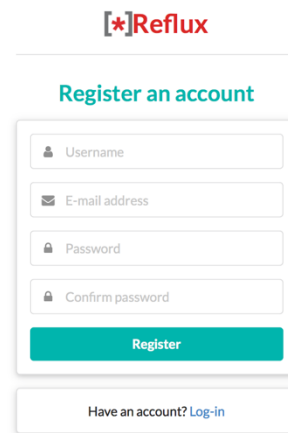
        User::create(
            [
                'name'      => 'Seán',
                'email'     => 'seanbickmore@icloud.com',
                'password'  => Hash::make('tester'),
                'bio'       => 'Welcome to Reflux, my fourth year Cyber Security
project!'
            ]
        );

        User::create(
            [
                'name'      => 'Naxes',
                'email'     => 'naxes@example.com',
                'password'  => Hash::make('tester')
            ]
        );
    }
}
```

Figure 73: Database Seeding: User Account

The above references two dummy accounts that are generated whenever database migration occurs, meaning that, with respect to manually engaging with the application, a user account does not need to be created manually. This cascades to posts, so that associations between tables can be checked and any functionality that is accessible to users who have ownership over posts they have submitted, such as modification and deletion.

REGISTRATION



The registration form is centered on the page. At the top, it features the Reflux logo, which consists of a red asterisk in a square followed by the word "Reflux" in a bold, black, sans-serif font. Below the logo is a horizontal line. Underneath the line, the text "Register an account" is displayed in a teal, sans-serif font. The form itself is a white rounded rectangle with a thin grey border. It contains four input fields, each with a small icon on the left: a person icon for "Username", an envelope icon for "E-mail address", a padlock icon for "Password", and another padlock icon for "Confirm password". Below these fields is a prominent teal button with the word "Register" in white, bold, sans-serif text. At the bottom of the form, there is a link that says "Have an account? Log-in" in a small, grey, sans-serif font.

Figure 74: Registration View

The registration view comprises of the form that executes the query in relation to creating a new user account. It is accessible only to users of type guest. Authenticated users cannot access the view or methods that relate to the registration process. A user must provide a name, email, password, and confirmation of the password.

[*]Reflux

Log-in to your account

Username

Password

Login

New? Sign Up

Figure 75: Login View

The login view comprises of the form that executes the query in relation to accessing an existing user account. It is accessible only to users of type guest. Authenticated users cannot access the view or methods that relate to the login process. A user must provide a name and password.

USER PROFILES

The screenshot displays a user profile for 'Seán'. At the top, there is a navigation bar with a home icon, a search bar, and a user profile icon. The main content area is titled 'Your profile' and features three tabs: 'Your posts', 'Your likes', and 'Your comments'. The 'Your posts' tab is active, showing a grid of six posts. Each post includes the number of votes, the title, submission time, author name, and comment count. To the right of the posts is a profile card for 'Seán', which includes a profile picture, name, join date, bio, submission count, and email address.

Post Title	Votes	Submitted	Comments
HTML	1.3K	9 minutes ago	3
JavaScript	1.5K	9 minutes ago	0
Java	436	9 minutes ago	0
Ruby	781	9 minutes ago	0
C#	404	9 minutes ago	0
Perl	256	9 minutes ago	0

Seán
Joined May 6, 2018
Welcome to Reflux, my fourth year Cyber Security project!
6 submissions
seanbickmore@icloud.com

Figure 76: User Profiles View

User profiles encompass data pertaining to a given user in isolation and any data that has been attributed to them via engagement with the application. Thusly, users can view information of the user as encompassed by the user table (i.e. name, email, bio, location, URL), the posts they have submitted, liked, and comments history. If the profile in question is the authenticated users, they may manage the content they have submitted and voted on (i.e. modification, deletion, revoke a vote).

USER SETTINGS

The screenshot displays the 'User Settings' interface. At the top, there is a navigation bar with a home icon, a search bar, and a user profile icon. The main content area is titled 'Public profile' and contains a form for editing profile information. The form has the following fields:

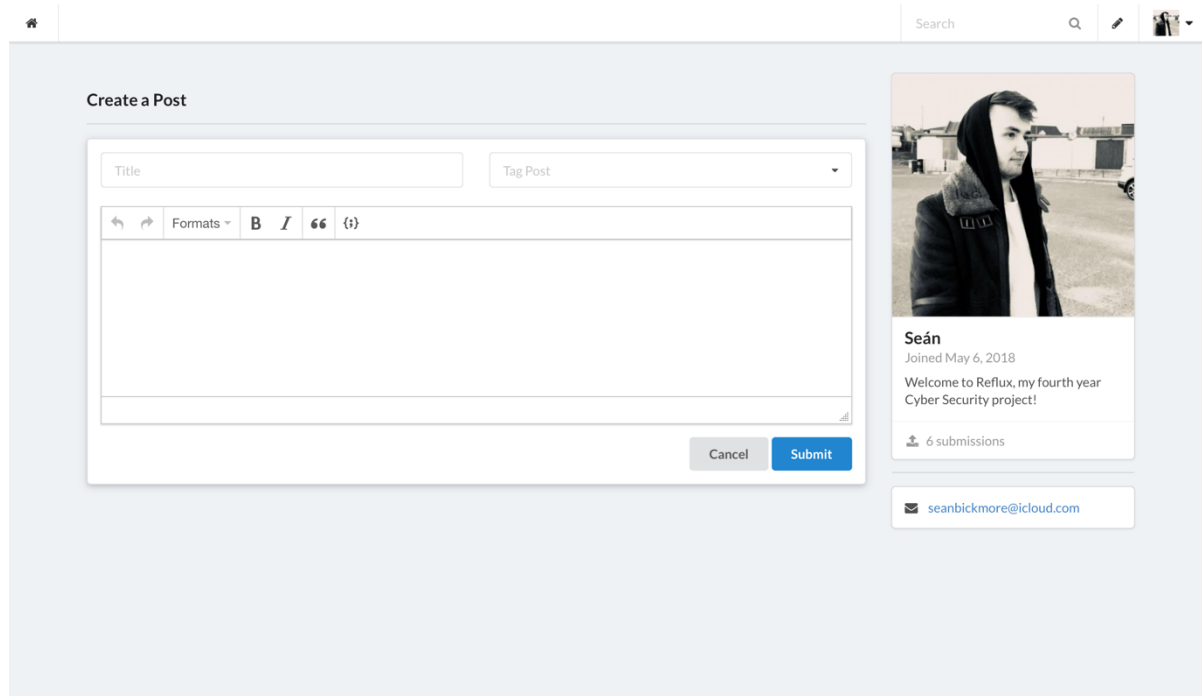
- Name:** A text input field containing 'Seán'.
- Email:** A text input field containing 'seanbickmore@icloud.com'.
- URL:** An empty text input field.
- Location:** An empty text input field.
- Bio:** A text area containing the text 'Welcome to Reflux, my fourth year Cyber Security project!'.

At the bottom of the form are two buttons: 'Cancel' and 'Save'. To the right of the form is a profile card for 'Seán', which includes a profile picture, the name 'Seán', the join date 'Joined May 6, 2018', the bio 'Welcome to Reflux, my fourth year Cyber Security project!', and a link to '6 submissions'. Below the profile card is an email address 'seanbickmore@icloud.com' with an envelope icon.

Figure 77: User Settings View

The user settings view allows users to modify aspects of their profile information inclusive of inputs that, by default, are null upon creation of an account. In essence, optional data. Data related to that as was prompted during account creation or login cannot be excluded.

CONTENT SUBMISSION

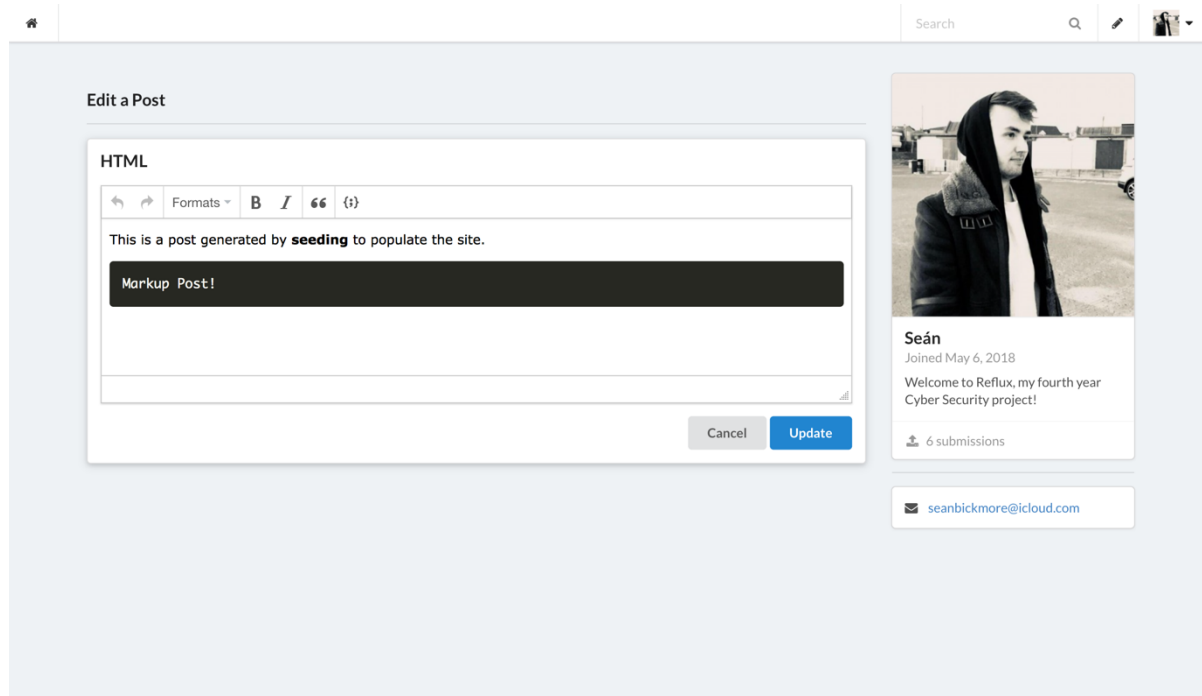


The screenshot shows a web interface for creating a post. At the top, there is a navigation bar with a home icon, a search bar, and a user profile icon. The main content area is titled "Create a Post". It features a form with a "Title" input field and a "Tag Post" dropdown menu. Below these is a rich text editor with a toolbar containing undo, redo, a "Formats" dropdown, bold (B), italic (I), quote ("), and code ({}). The form has "Cancel" and "Submit" buttons at the bottom right. To the right of the form is a user profile card for "Seán", who joined in May 2018 and has a bio about a Cyber Security project. Below the profile card, it shows "6 submissions" and an email address "seanbickmore@icloud.com".

Figure 78: Content Submission View

The content submission view allows users to submit a post inclusive or exclusive of technical facets (i.e. code block). Users must provide a post title, body, and at least one tag that which defines the context of the post.

CONTENT MODIFICATION



The screenshot shows a web interface for editing a post. At the top, there is a navigation bar with a home icon, a search bar, and a user profile icon. The main content area is titled "Edit a Post". On the left, there is a text editor with a "HTML" tab selected. The editor's toolbar includes "Formats", "B" (bold), "I" (italic), "“”" (quote), and "(i)" (code). The text area contains the message "This is a post generated by seeding to populate the site." Below this is a dark code block containing the text "Markup Post!". At the bottom of the editor are "Cancel" and "Update" buttons. On the right side of the interface, there is a user profile card for "Seán", who joined on May 6, 2018, and has a bio that reads "Welcome to Reflux, my fourth year Cyber Security project!". Below the bio, it shows "6 submissions" and an email address "seanbickmore@icloud.com".

Figure 79: Content Modification View

The content modification view allows users to edit an existing post inclusive or exclusive of technical facets (i.e. code block). Users may only modify a posts body. This ensures that the context of the post cannot be changed or misrepresented.

CONTENT DELETION

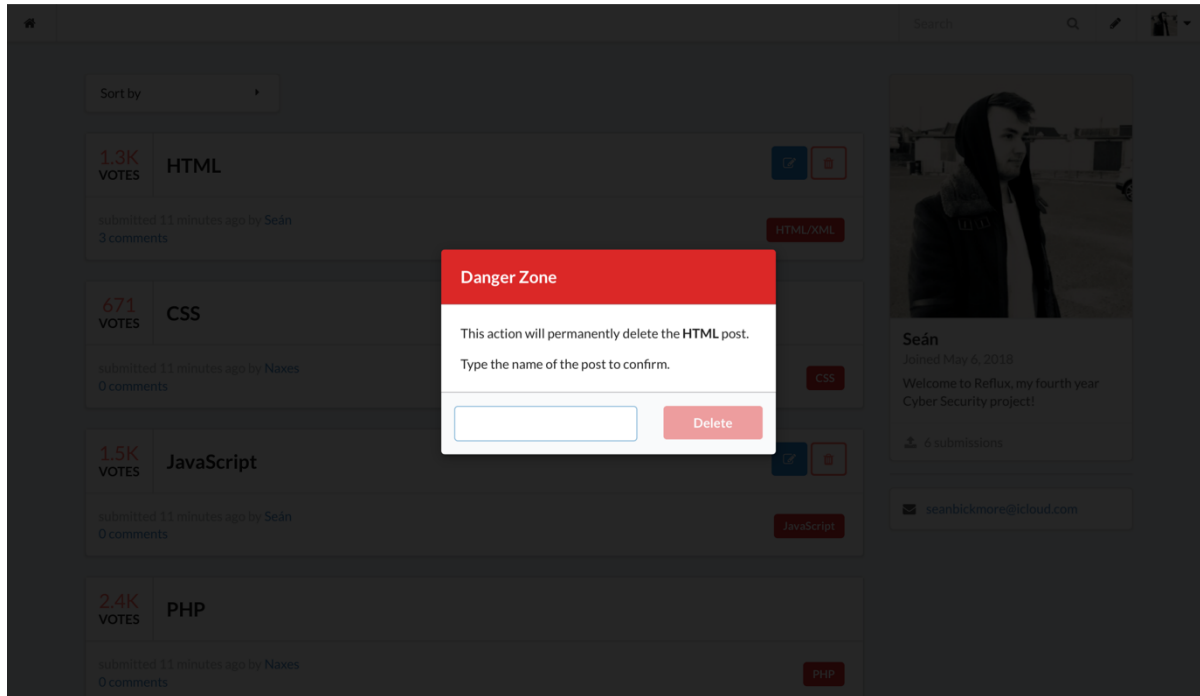


Figure 80: Content Deletion Modal

The content deletion modal allows users to remove an existing post. Users are prompted for confirmation of the action by provision of the post in questions title.

CONTENT FILTERING

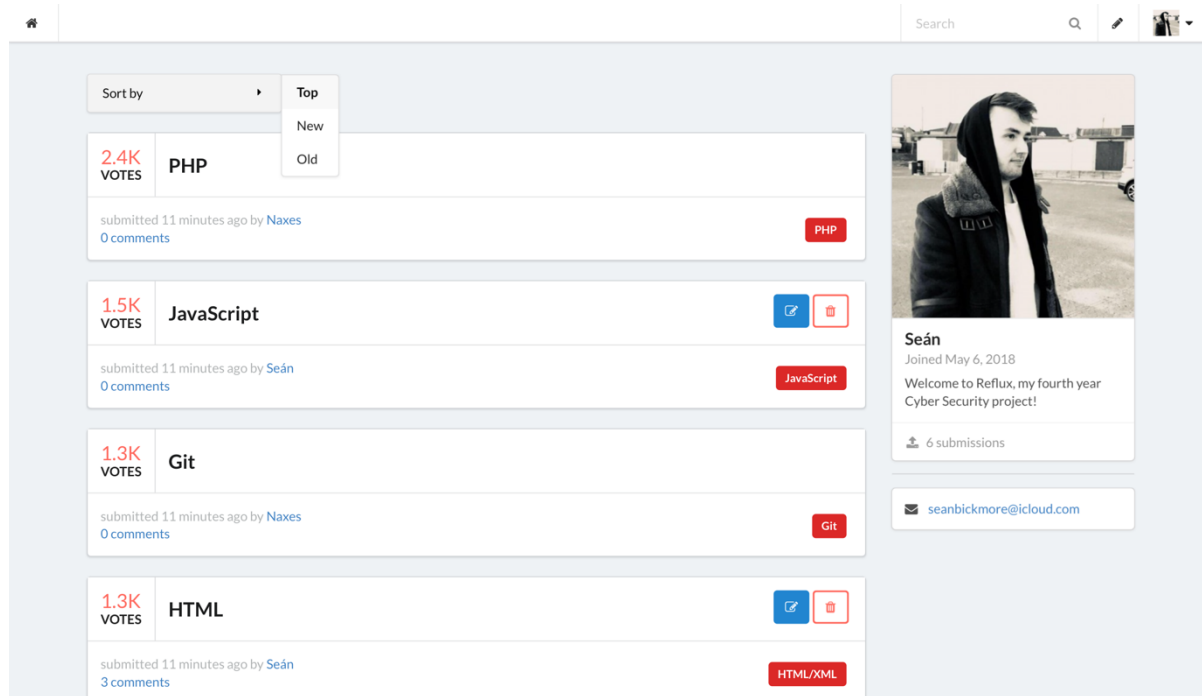
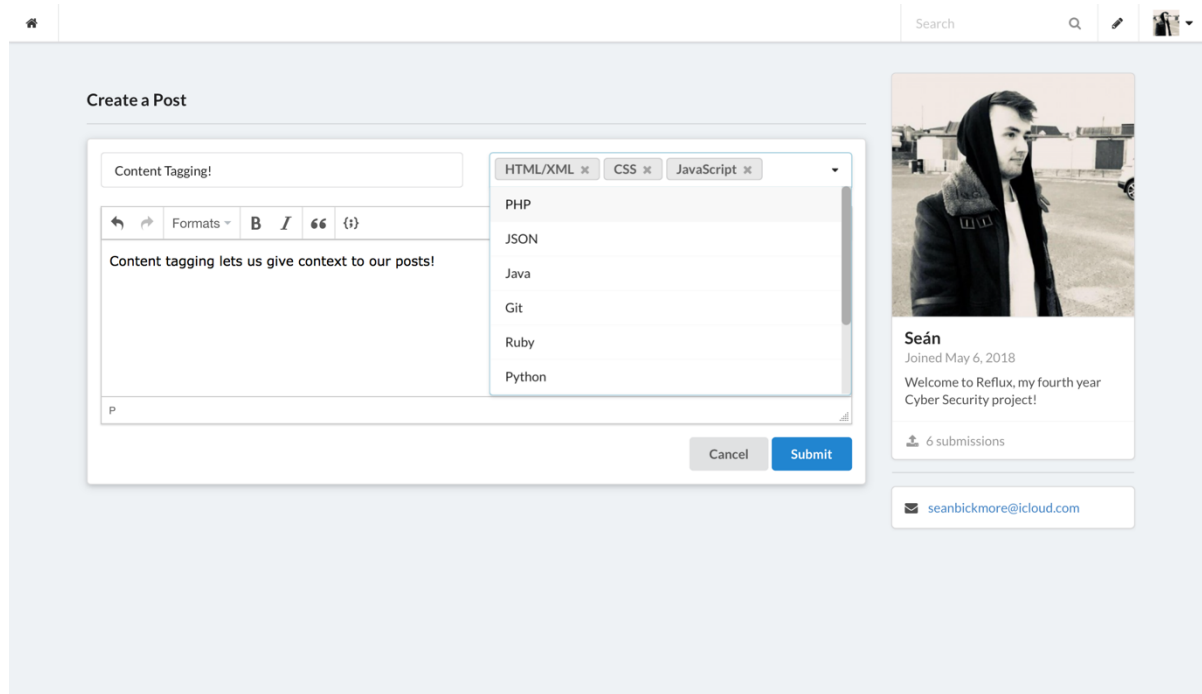


Figure 81: Content Filtering Component

Content filtering is a component of the user interface located on the home page of the application. It utilises a request input variable (sort) that, when validated by the system, alters the order of content based on the selectable metrics (e.g. top, new, old).

CONTENT TAGGING



The image shows a 'Create a Post' form with a content tagging component. The form has a title 'Content Tagging!' and a text area containing the text 'Content tagging lets us give context to our posts!'. Below the text area is a 'P' icon. To the right of the text area is a dropdown menu with the following options: HTML/XML, CSS, JavaScript, PHP, JSON, Java, Git, Ruby, and Python. The dropdown menu is currently open, showing the first five options. Below the dropdown menu are 'Cancel' and 'Submit' buttons. To the right of the form is a user profile card for 'Seán', who joined on May 6, 2018, and has 6 submissions. The profile card also includes a bio: 'Welcome to Reflux, my fourth year Cyber Security project!' and an email address: 'seanbickmore@icloud.com'.

Figure 82: Content Tagging Component

Content tagging is a component of the content submission and content modification functional requirements. It is a required input in either case that defines one or many tags that which relate to and describe the context of a given post. For each tag given, a new record is created in the tags table.

CONTENT SEARCHING

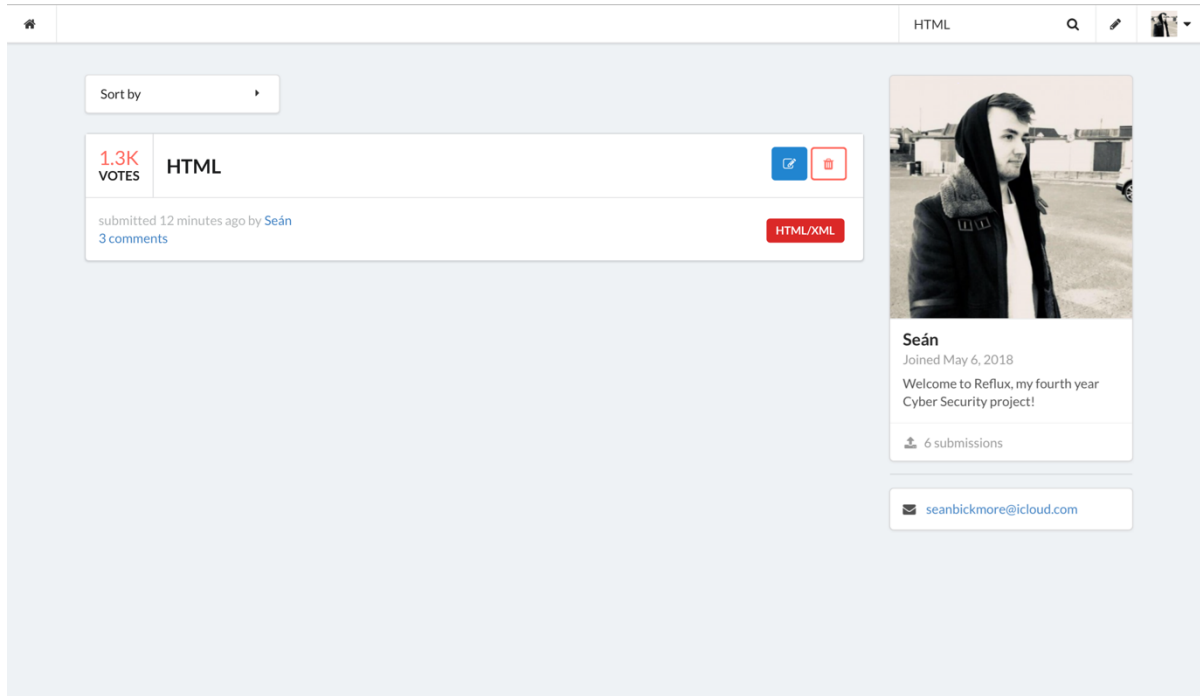


Figure 83: Content Searching Component

Content searching is component of the user interface, specifically the navigation menu. Users may perform a search query through input of a posts title or user. This retrieves results that match or partially match the provided input.

RATING SYSTEM

The screenshot shows a web interface with a 'RATING SYSTEM' header. The main content area features a list of programming language posts. Each post includes a 'VOTES' count, the language name, submission time, author name, and comment count. A 'Sort by' dropdown is located at the top left. On the right side, there is a user profile for 'Seán', which includes a profile picture, name, join date, bio, and email address.

Language	Votes	Submitted by	Comments
HTML	0	Seán	3
CSS	671	Naxes	0
JavaScript	1.5K	Seán	0
PHP	2.4K	Naxes	0

User Profile: Seán
Joined May 6, 2018
Welcome to Reflux, my fourth year Cyber Security project!
6 submissions
seanbickmore@icloud.com

Figure 84: Rating Component

The rating system is a component of the user interface interlinked with retrieved posts. It is an isolated table that, when engaged, creates a record of a new vote associated with the post and user. Votes may either be created or removed, but a post cannot be disliked.

COMMENTS/REPLIES

The screenshot displays a web interface for managing comments and replies. At the top, there is a navigation bar with a search icon and a user profile icon. Below this is a text editor with a 'Save' button. The main content area is divided into two sections: 'Comments' and a user profile. The 'Comments' section shows three comments: a parent comment by 'Seán', a child comment by 'Naxes', and another parent comment by 'Seán'. The user profile for 'Seán' includes a bio, a join date, and an email address.

Comments

- Seán** 15 minutes ago
This is a parent comment.
Reply
- Naxes** 15 minutes ago
This is a child comment.
Reply
- Seán** 15 minutes ago
This is another parent comment.
Reply

Seán
Joined May 6, 2018
Welcome to Reflux, my fourth year Cyber Security project!
6 submissions
seanbickmore@icloud.com

Figure 85: Comments/Replies Component

Comments/replies are a component of posts when viewed in isolation via selection by the user. It is an isolated table associated with a post and user. Comments may be a parent or child. Parents comments represent top-level comments, while children represent replies to parents. A comment can comprise of textual content only.

NIGHT MODE

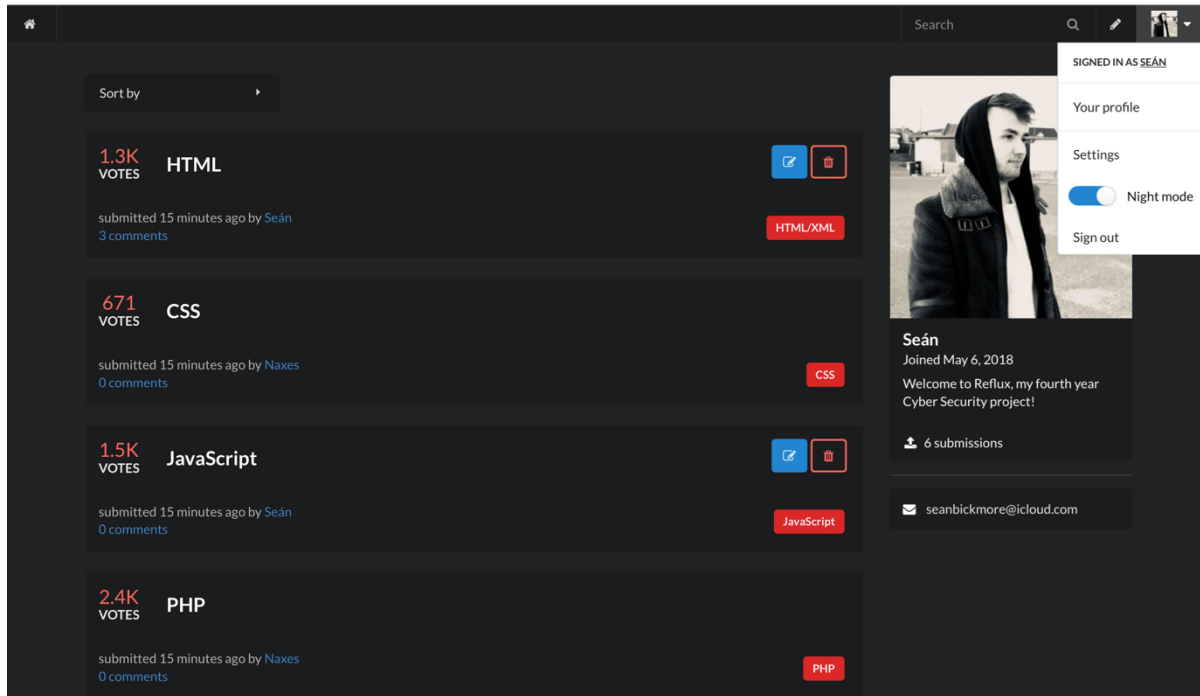


Figure 86: Night Mode Component

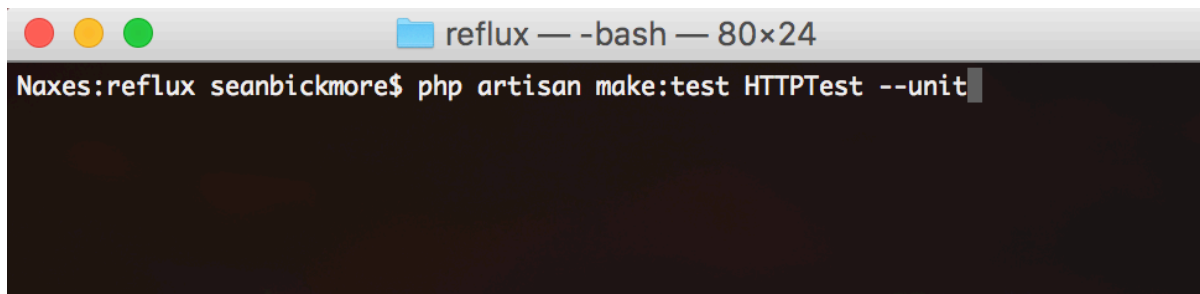
Night mode is a component of the user interface, specifically the navigation dropdown menu. Users can use this element to toggle the thematic presentation of the site to better facilitate visibility during the day or night, respectively.

TESTING

Laravel comes packaged with PHPUnit. Tests are defined within the project structures appropriately named tests directory, and created through the PHP artisan command line tools. In Reflux, HTTP unit tests are used for verifying the navigability of routes under certain circumstances. These circumstances derive the methods a given user type has access to. Assertions are used to determine the success criteria of each test represented by HTTP status codes. In some cases, accessing a route is a successful outcome, while in others it is not. The idea of testing, is to create the tests as a prerequisite to the development of the feature, then build the feature around making these tests succeed.

CREATING TESTS

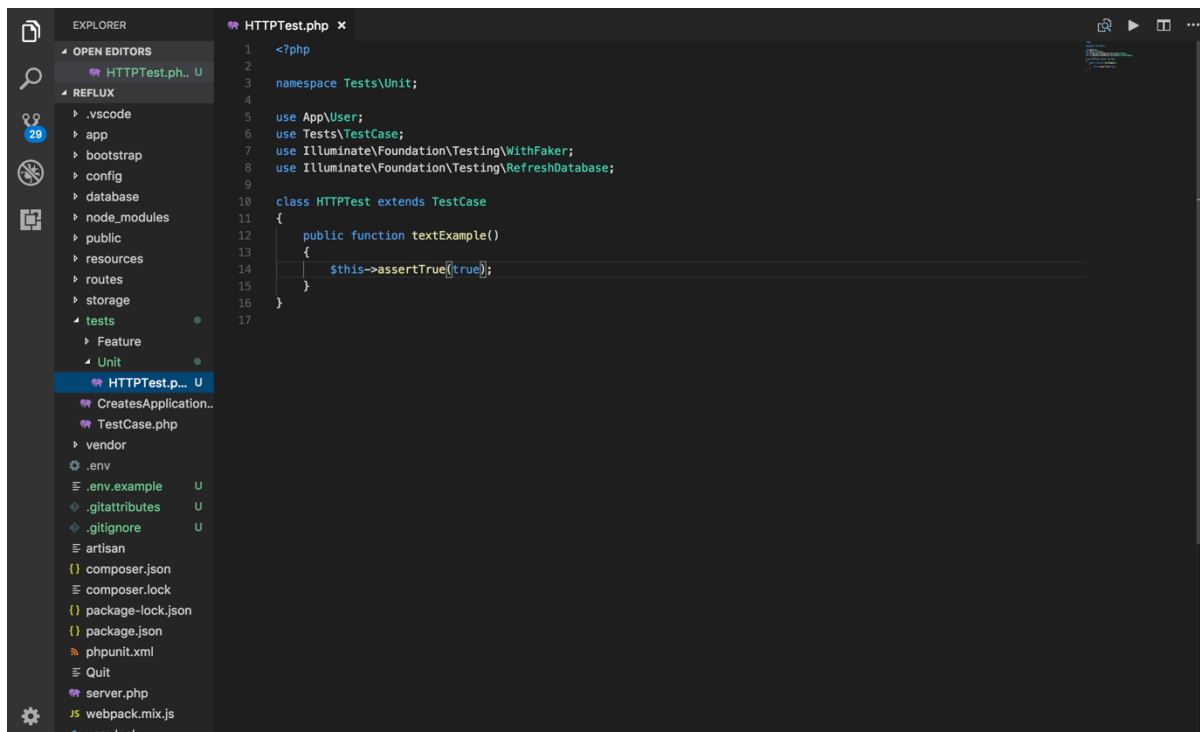
Tests are generated via the PHP artisan command line tools as follows:



```
reflux — -bash — 80x24
Naxes:reflux seanbickmore$ php artisan make:test HTTPTest --unit
```

Figure 87: Creating a Unit Test

This creates the test of the provided name within the projects directory:



```
EXPLORER
├── OPEN EDITORS
│   └── HTTPTest.ph... U
├── REFLUX
│   ├── .vscode
│   ├── app
│   ├── bootstrap
│   ├── config
│   ├── database
│   ├── node_modules
│   ├── public
│   ├── resources
│   ├── routes
│   ├── storage
│   ├── tests
│   │   ├── Feature
│   │   └── Unit
│   │       └── HTTPTest.p... U
│   ├── CreatesApplication...
│   ├── TestCase.php
│   └── vendor
├── .env
├── .env.example U
├── .gitattributes U
├── .gitignore U
├── artisan
├── composer.json
├── composer.lock
├── package-lock.json
├── package.json
├── phpunit.xml
├── Quit
├── server.php
├── webpack.mix.js
└── yarn.lock
```

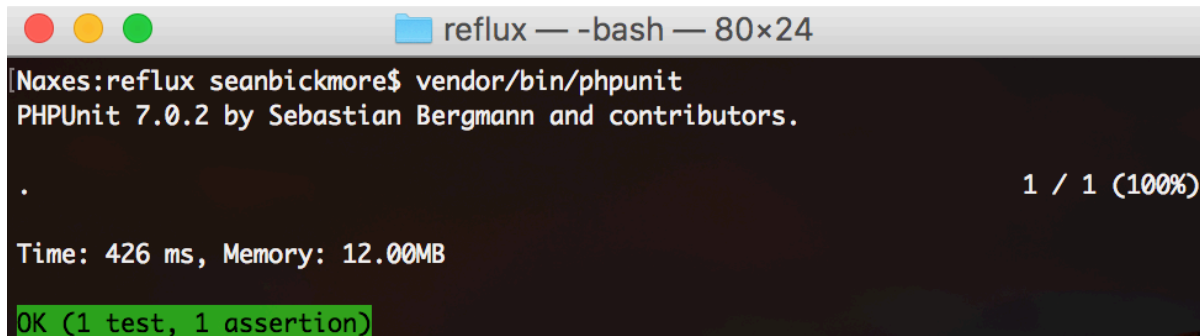
```
HTTPTest.php
1 <?php
2
3 namespace Tests\Unit;
4
5 use App\User;
6 use Tests\TestCase;
7 use Illuminate\Foundation\Testing\WithFaker;
8 use Illuminate\Foundation\Testing\RefreshDatabase;
9
10 class HTTPTest extends TestCase
11 {
12     public function textExample()
13     {
14         $this->assertTrue([true]);
15     }
16 }
17
```

Figure 88: Location of Unit Test

This process was followed in the creation of all subsequent tests to follow.

RUNNING TESTS

Tests are run via the PHPUnit command. All tests are run conjunctively. In any case whereby a test fails, the exception is highlighted:

A terminal window titled 'reflux — -bash — 80x24' showing the execution of PHPUnit. The prompt is 'Naxes:reflux seanbickmore\$'. The command 'vendor/bin/phpunit' has been executed. The output shows 'PHPUnit 7.0.2 by Sebastian Bergmann and contributors.' followed by a single dot representing a passed test, '1 / 1 (100%)', and performance metrics 'Time: 426 ms, Memory: 12.00MB'. The final result is 'OK (1 test, 1 assertion)' highlighted in green.

```
Naxes:reflux seanbickmore$ vendor/bin/phpunit
PHPUnit 7.0.2 by Sebastian Bergmann and contributors.

.                                                    1 / 1 (100%)

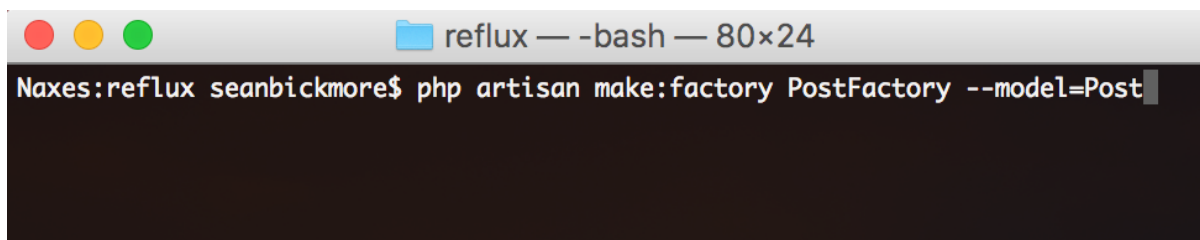
Time: 426 ms, Memory: 12.00MB

OK (1 test, 1 assertion)
```

Figure 89: Running Unit Tests

GENERATING FACTORIES

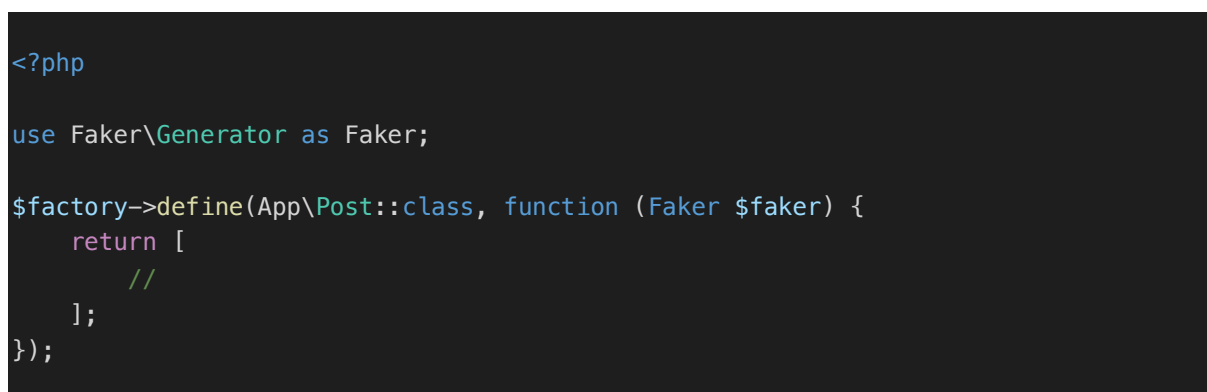
In some cases, to mimic manipulation of a resource, factories must be created. By default, Laravel generates a factory pertaining to the user model, allowing for subsequent testing whereby an authenticated user is required. To generate factories for other models, the following PHP artisan command is used:

A terminal window titled 'reflux — -bash — 80x24' showing the command 'php artisan make:factory PostFactory --model=Post' being entered at the prompt 'Naxes:reflux seanbickmore\$'.

```
Naxes:reflux seanbickmore$ php artisan make:factory PostFactory --model=Post
```

Figure 90: Generating a Factory

This generates a Post Factory whereby a dummy post may be defined for use in the unit testing process:

A code block showing the generated PHP code for a Post Factory. It includes the Faker library and a define method for the Post model.

```
<?php

use Faker\Generator as Faker;

$factory->define(App\Post::class, function (Faker $faker) {
    return [
        //
    ];
});
```

Figure 91: Generated Post Factory

HTTP TESTING

These tests are used to verify the routes a given user type has access to, which determines the methods they may execute. The outcome of the test can be asserted by stipulating what the returned HTTP status code should be. For example, a guest user should not be able to visit the view relating to post creation, therefore they are redirected. The status code for redirection is 302, which becomes the success criteria of the test.

GUEST USER TESTS

TEST 1: HOMEPAGE

```
/* Homepage */
public function testGuestUserHome()
{
    $response = $this->get('/')
                ->assertStatus(200);
}
```

Figure 92: Guest User: Homepage Test

Test asserts that a user of type guest should be able to visit the homepage.

TEST 2: REGISTRATION

```
/* Registration */
public function testGuestUserRegister()
{
    $response = $this->get('/register')
                ->assertStatus(200);
}
```

Figure 93: Guest User: Registration Test

Test asserts that a user of type guest should be able to visit the registration page.

TEST 3: LOGIN

```
/* Login */
public function testGuestUserLogin()
{
    $response = $this->get('/login')
                ->assertStatus(200);
}
```

Figure 94: Guest User: Login Test

Test asserts that a user of type guest should be able to visit the login page.

TEST 4: VALID CREDENTIALS

```
/* Valid Credentials */
public function testGuestValidCredentials()
{
    $this->assertCredentials(['name' => 'Naxes', 'password' => 'tester'],
$guard = null);
}
```

Figure 95: Guest User: Valid Credentials Test

Test asserts that provided credentials match that of an existing user record in the database.

TEST 5: INVALID CREDENTIALS

```
/* Invalid Credentials */
public function testGuestValidInvalidCredentials()
{
    $this->assertInvalidCredentials(['name' => 'InvalidUser', 'password' =>
'tester'], $guard = null);
}
```

Figure 96: Guest User: Invalid Credentials Test

Test asserts that provided credentials do not match that of an existing user record in the database.

TEST 6: USER PROFILES

```
/* User Profiles */
public function testGuestUserUserProfile()
{
    $response = $this->get('/Naxes')
->assertStatus(200);
}
```

Figure 97: Guest User: User Profile Test

Test asserts that a user of type guest should be able to visit another users profile.

TEST 7: USER SETTINGS

```
/* User Settings */
public function testGuestUserUserSettings()
{
    $response = $this->get('/edit/Naxes')
        ->assertStatus(302);
}
```

Figure 98: Guest User: User Settings Test

Test asserts that a user of type guest should not be able to visit the user settings page pertaining to another users profile.

TEST 8: CONTENT SUBMISSION

```
/* Content Submission */
public function testGuestUserContentSubmission()
{
    $response = $this->get('/posts/create')
        ->assertStatus(302);
}
```

Figure 99: Guest User: Content Submission Test

Test asserts that a user of type guest should not be able to visit the content submission page.

TEST 9: CONTENT MODIFICATION

```
/* Content Modification */
public function testGuestUserContentModification()
{
    $response = $this->get('/posts/edit/1')
        ->assertStatus(302);
}
```

Figure 100: Guest User: Content Modification Test

Test asserts that a user of type guest should not be able to visit the content modification page of a post associated with another user.

AUTHENTICATED USER TESTS

TEST 10: HOMEPAGE

```
/* Homepage */
public function testAuthUserHomepage()
{
    $user = factory(User::class)->create();

    $response = $this->actingAs($user)
        ->withSession(['foo' => 'bar'])
        ->get('/')
        ->assertStatus(200);
}
```

Figure 101: Authenticated User: Homepage Test

Test asserts that an authenticated user should be able to visit the homepage.

TEST 11: REGISTRATION

```
/* Registration */
public function testAuthUserRegister()
{
    $user = factory(User::class)->create();

    $response = $this->actingAs($user)
        ->withSession(['foo' => 'bar'])
        ->get('/register')
        ->assertStatus(302);
}
```

Figure 102: Authenticated User: Registration Test

Test asserts that an authenticated user should not be able to visit the registration page.

TEST 12: LOGIN

```
/* Login */
public function testAuthUserLogin()
{
    $user = factory(User::class)->create();

    $response = $this->actingAs($user)
        ->withSession(['foo' => 'bar'])
        ->get('/login')
        ->assertStatus(302);
}
```

Figure 103: Authenticated User: Login Test

Test asserts that an authenticated user should not be able to visit the login page.

TEST 13: USER PROFILES

```
/* User Profiles */
public function testAuthUserUserProfile()
{
    $user = factory(User::class)->create();

    $response = $this->actingAs($user)
        ->withSession(['foo' => 'bar'])
        ->get('/') . $user->name
        ->assertStatus(200);
}
```

Figure 104: Authenticated User: User Profile Test

Test asserts that an authenticated user should be able to visit their own or another users profile. The former is represented within the test in this case via the inclusion of the faux users name appended to the URL.

TEST 14: USER SETTINGS (AUTH SETTINGS)

```
/* User Settings (Auth Settings) */
public function testAuthUserUserSettings1()
{
    $user = factory(User::class)->create();

    $response = $this->actingAs($user)
        ->withSession(['foo' => 'bar'])
        ->get('/edit/' . $user->name)
        ->assertStatus(200);
}
```

Figure 105: Authenticated User: Auth User Settings Test

Test asserts that an authenticated user should be able to visit their own settings page.

TEST 15: USER SETTINGS (OTHER USER)

```
/* User Settings (Other Settings) */
public function testAuthUserUserSettings2()
{
    $user = factory(User::class)->create();

    $response = $this->actingAs($user)
        ->withSession(['foo' => 'bar'])
        ->get('/edit/Naxes')
        ->assertStatus(302);
}
```

Figure 106: Authenticated User: Other User Settings Test

Test asserts that an authenticated user should not be able to visit another users settings page.

TEST 16: CONTENT SUBMISSION

```
/* Content Submission */
public function testAuthUserContentSubmission()
{
    $user = factory(User::class)->create();

    $response = $this->actingAs($user)
        ->withSession(['foo' => 'bar'])
        ->get('/posts/create')
        ->assertStatus(200);
}
```

Figure 107: Authenticated User: Content Submission Test

Test asserts that an authenticated user should be able to visit the content submission page.

TEST 17: CREATING A POST

```
/* Creating a Post */
public function testContentSubmission()
{
    $post = factory(Post::class)->make();
    $users = factory(User::class)
        ->create()
        ->each(function ($u) use ($post) {
            $u->post()->save($post);
        });

    $this->assertDatabaseHas('posts', ['title' => $post->title]);
}
```

Figure 108: Authenticated User: Creating a Post

Test asserts that an authenticated user can create a new post. This is validated by checking for the newly created posts title via assertion parameter. Test utilises the default user factory and previously generated post factory.

TEST RESULTS

```
reflux — -bash — 80x24
Naxes:reflux seanbickmore$ vendor/bin/phpunit
PHPUnit 7.0.2 by Sebastian Bergmann and contributors.

.....                                     17 / 17 (100%)

Time: 418 ms, Memory: 16.00MB

OK (17 tests, 17 assertions)
```

Figure 109: Unit Testing Results

CUSTOMER TESTING

Customer testing took place via engagement with the application and subsequent survey questions querying participants on their experience. 10 instances were recorded for evaluation comprising of colleagues that which represent the target market. The main objective of testing was to inquire the comprehensibility of Reflux from an end-user perspective in addition to garnering which facets – if any – successfully improve on existing competitive implementations. Each participant was required to create an account with the system with the express intent of capturing their experience in accomplishing as such when factoring in validation enforcements. The following are the questions as provided to participants following usage of Reflux:

QUESTION 1

How long did it take to register an account?

10 responses

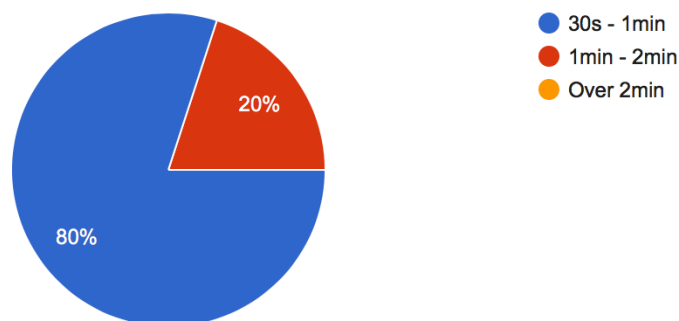


Figure 110: Customer Testing: Question 1 Results

QUESTION 2

On a scale of 1 to 10, how comprehensive was the user interface?

10 responses

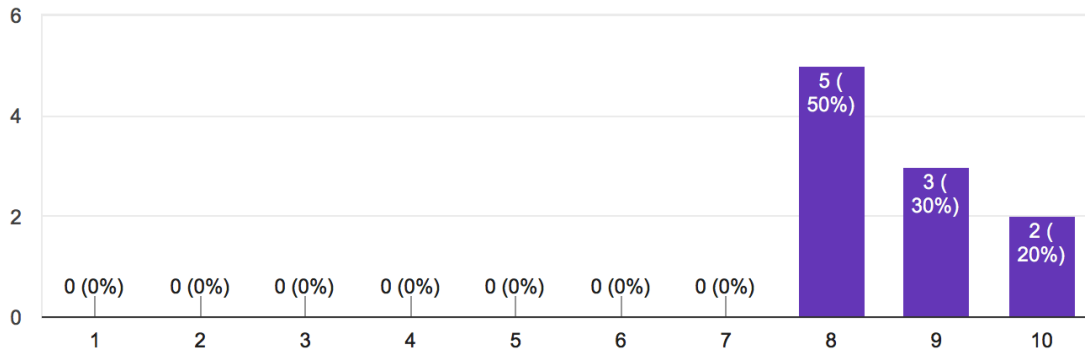


Figure 111: Customer Testing: Question 2 Results

QUESTION 3

Did enforced validation impede using the application? (e.g. max character count)

10 responses

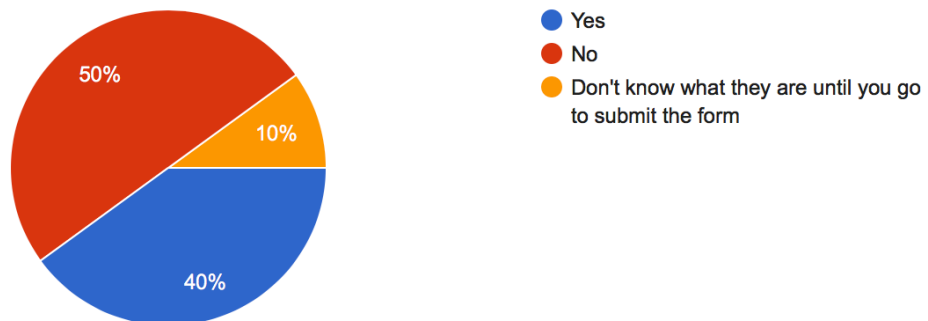


Figure 112: Customer Testing: Question 3 Results

QUESTION 4

On a scale of 1 to 10, how well does the system communicate these enforcements?

10 responses

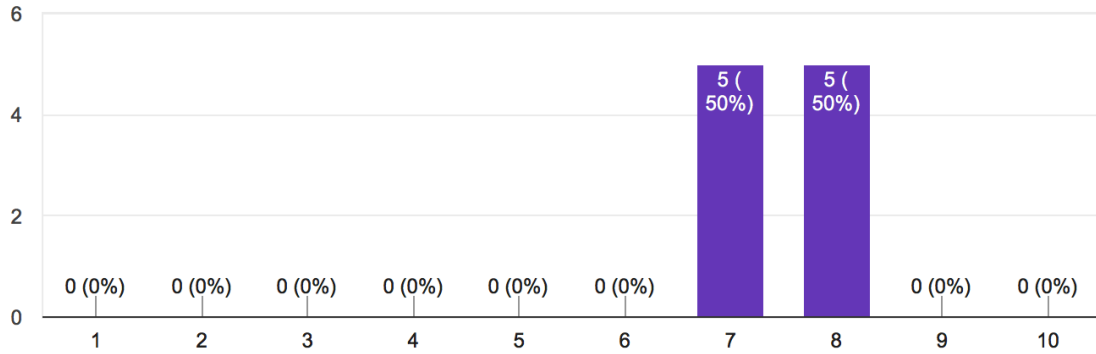


Figure 113: Customer Testing: Question 4 Results

QUESTION 5

Is it clear what the intent of the system is?

10 responses

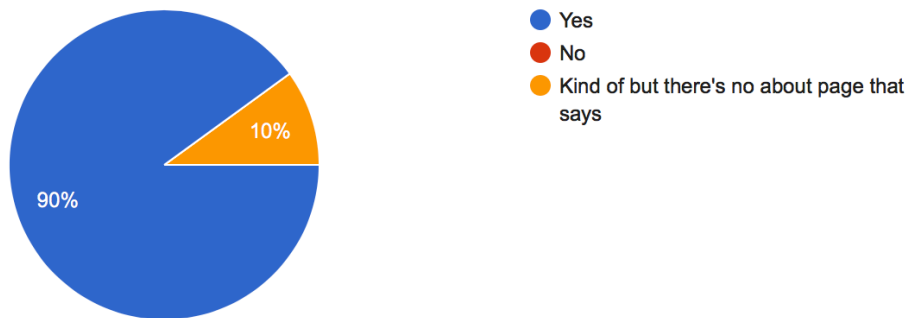


Figure 114: Customer Testing: Question 5 Results

QUESTION 6

Does the system invoke the sense that it is secure?

10 responses

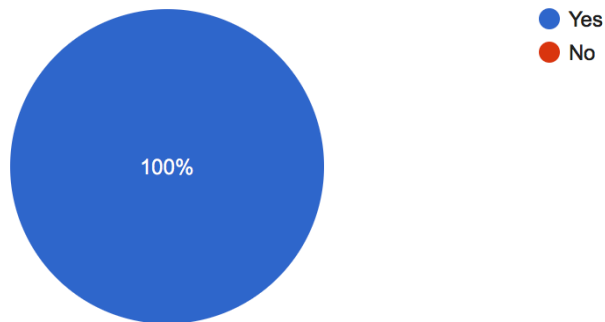


Figure 115: Customer Testing: Question 6 Results

QUESTION 7

On a scale of 1 to 10, do you agree or disagree that security is important, even if it is not necessary for the application to work?

10 responses

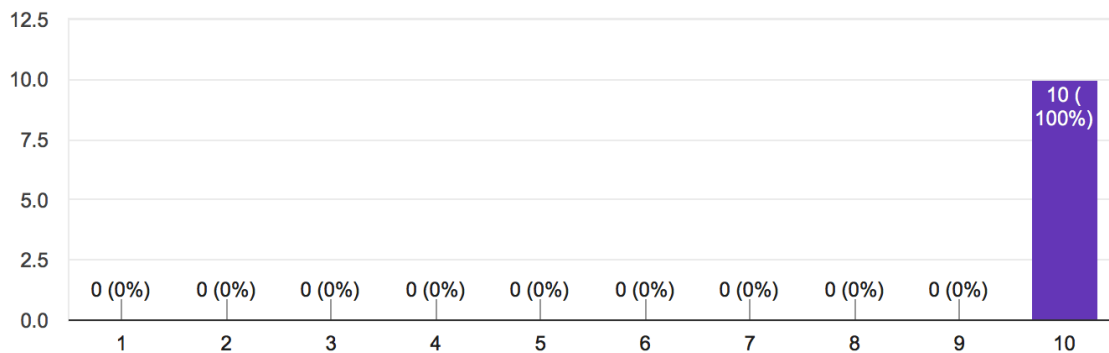


Figure 116: Customer Testing: Question 7 Results

QUESTION 8

Which of the following is the system most like:

10 responses

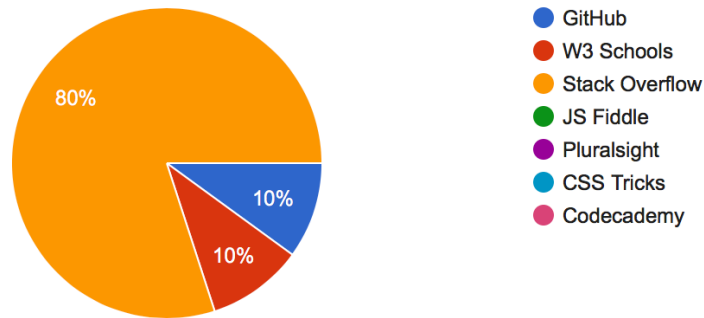


Figure 117: Customer Testing: Question 8 Results

QUESTION 9

What aspects of the system - if any - are improved comparatively to your selection above?

10 responses

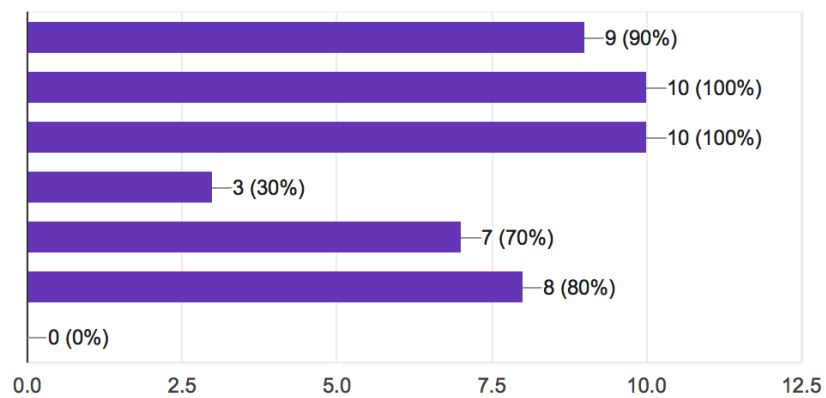


Figure 118: Customer Testing: Question 9 Results

QUESTION 10

What aspects of the system - if any - could be improved?

10 responses

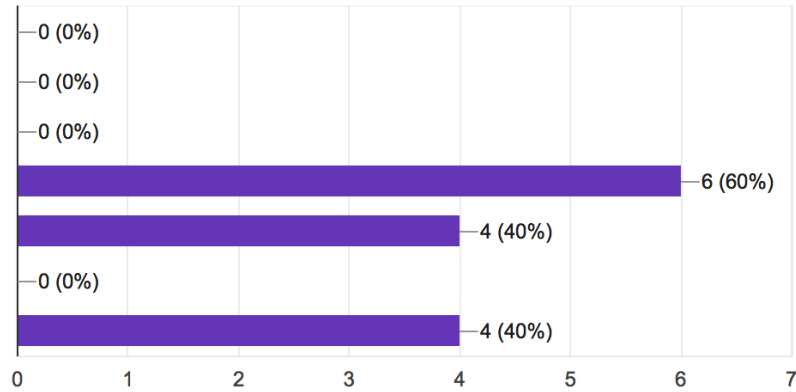


Figure 119: Customer Testing: Question 10 Results

EVALUATION

OVERVIEW

In summation, the responses elicited from the 10 participants satisfied the objective of customer testing in identifying the comprehensibility of the system from an end-user perspective, the processes improved over competitive technologies, and those that were not. In the onset of the testing procedure, participants were required to create an account with the system, as this is a prerequisite to experiencing the system in its entirety, as well as within the fact that it is an encompassed question in the subsequent survey. This was to garner how little or how much validation requirements of the form may frustrate and or impede a user's progress with respect to gaining access to an authenticated state.

Notably, the system was praised in terms of the overall user experience, as well as its general communication of intention with respect to the user interfaces elements, and within divulging the actual contextual nature of the application from a business perspective, in most cases. Every participant unitarily agreed on the importance of integrating an underlying security layer regardless of the fact that such an implementation is non-functional, and thusly not technically a requirement in fulfilling the marketable aspect of the idea. Unsurprisingly, most recorded responses noted Reflux as most similar to Stack Overflow amongst the plethora of provisioned competitors. However, this is in addition to noting improvements over the aforementioned in many respects, inclusive of user experience, syntax highlighting, the process of creating/modifying/deleting content, community features (user profiles, comment section, rating system), and security implementations. The latter being derived from discussion with participants. The most negative aspect presented about the system lay within its capacity in allowing users to search and sort content, which is an agreeable facet when viewed comparatively to the features initial scope prior to development. The component itself satisfies the requirements as defined, but does not excel or differentiate itself to that of competitive search algorithms.

RESPONSES

The following represent the percentage response of each selectable option presented to participants:

QUESTION 1

How long did it take to register an account?

- ❖ **30s – 1min:** 80%
- ❖ **1min – 2min:** 20%

QUESTION 2

On a scale of 1 (incomprehensible) to 10 (comprehensive), how comprehensive was the user interface?

- ❖ **8:** 50%
- ❖ **9:** 30%
- ❖ **10:** 20%

QUESTION 3

Did enforced validation impede using the application? (e.g. max character count)

- ❖ **Yes:** 40%
- ❖ **No:** 50%
- ❖ **Other:** 10%
 - “Don’t know what they are until you go to submit the form”

QUESTION 4

On a scale of 1 (badly) to 10 (exceptionally), how well does the system communicate these enforcements?

- ❖ **7:** 50%
- ❖ **8:** 30%

QUESTION 5

Is it clear what the intent of the system is?

- ❖ **Yes:** 90%
- ❖ **Other:** 10%
 - “Kind of but there’s no about page that says”

QUESTION 6

Does the system invoke the sense that it is secure?

- ❖ **Yes:** 100%

QUESTION 7

On a scale of 1 to 10, do you agree or disagree that security is important, even if it is not necessary for the application to work?

- ❖ **10: 100%**

QUESTION 8

Which of the following is the system most like?

- ❖ **GitHub: 10%**
- ❖ **W3 Schools: 10%**
- ❖ **Stack Overflow: 80%**

QUESTION 9

What aspects of the system – if any – are improved comparatively to your selection above?

- ❖ **User Interface/User Experience: 90%**
- ❖ **Syntax Highlighting (Code Blocks): 100%**
- ❖ **Creating/Modifying/Deleting Posts: 100%**
- ❖ **Searching and Sorting Content: 30%**
- ❖ **Community Features (User Profiles, Comments System, Rating System): 70%**
- ❖ **Security Implementations: 80%**

QUESTION 10

What aspects of the system – if any – could be improved?

- ❖ **Searching and Sorting Content: 60%**
- ❖ **Community Features (User Profiles, Comments System, Rating System): 40%**
- ❖ **None: 40%**

CONCLUSIONS

In summation, from a development standpoint, Reflux outlined the processes integral to planning and integrating security layers into an application, which is something that had never been previously considered. Comparatively to the previous iteration of this project idea, these aspects were an after-thought to that of its functional requirements. However, it became evident that security is the most functional non-functional requirement with respect to web applications. There was an initial blockade in deriving what aspects of a web application relate to what vulnerabilities, how they can be accounted for, and how they apply when considering them within the context of Reflux explicitly. When viewed from a broader spectrum however, Reflux boasts the same functionality present in many web applications that involve taking data and doing something with it. For example, personal blogs or social media sites. In their basest form, they are not representationally the same, but without context, the architecture is identical in many respects. In that sense, there is correlation, and it is within said correlation that allowed for applicable logic in development regardless of the sites context as a sellable idea. Realising that the overarching functional objectives correlate to the architecture of other sites made the overall process of development easier when deriving Reflux's system architecture and how components communicate with one another to draw associations and relationships.

The Laravel PHP framework was decided upon from the initial conception of the idea. This had not been utilised for development at such a scale before, making the process a learning experience. The learning curve of this framework was exceptionally slim. Each component comprehensively communicates its intent, thusly there is less of a barrier-to-entry. Initially, the model view controller architectural structure was difficult to grasp with respect to following the intended convention and directory structure, but it swiftly became prevalent how each component of this structure communicates. The choice to use this framework came from the simplicity of its setup. It does not automate or generate code to fulfil any of Reflux's functional requirements, but it does reduce the time to create them manually. Laravel essentially paraphrases the processes it would take to develop a feature, such as authentication, via base PHP.

Despite these aspects, there are facets of the project that were difficult to deliberate and account for. Planning and the elicitation of requirements deviated heavily from those that were initially derived prior to development. In some cases, requirements spawned sub-requirements. For example, multiple tags for a post was not possible within a single table in a relational database. Thusly, this was later segregated to its own requirement. The same is true for votes. Although the increase in scope was slight, it was an increase all the same. There were also the social aspects to consider. In Reflux, this is represented by user profiles, voting, and commenting. With specific focus on the latter, commenting was originally intended to allow users to post code blocks the same way as can be achieved via creation of posts. The issue with this arose via the text editor in use, TinyMCE. Comparatively to the default text area, there was a massive performance hit, as the editor itself is being initialised for each comment there was. This meant that, given even as low a number as ten comments, the performance of the system was impacted greatly. This resulted in comments being stripped back to allow only for plaintext, as the same functionality of the text editor was not achievable using only a single text area input. Where applicable, there is only a single instance of the text editor called in any given circumstance in order to account for this discovery. Regardless, the outcome of the project aesthetically satisfies its requirements with respect to user experience, architecturally follows structures and methodologies that aid in fulfilling non-functional facets, and effectively enforces security implementations in each endpoint where data is handled.

FURTHER DEVELOPMENT OR RESEARCH

There is potential in the context of Reflux in a business sense, within its elicitation of the latest technologies, and within its security implementations when reflected against the latest vulnerabilities as outlined by the Open Web Application Security Project (OWASP). From a user experience perspective, the user interface has been highlighted within customer testing as fulfilling the requirement of comprehension in most instances, which was an initial goal of the project. Functionally, the application could be bolstered both in maintaining and extending the feature-set. There are unexplored ideas that were purposefully excluded to balance a manageable project scope while correlating, in part, with what was offered by competitive technologies. For example, displaying the culminated result of a posts multiple encompassed code blocks in cases where said code blocks relate to one another, such as with HTML, CSS, and JavaScript. This would combine the informational aspects of Stack Overflow and GitHub, and the kinetic aspects employed by JS Fiddle, allowing users to interact with and manipulate the result to gain a better understanding of the code. Additionally, when looking at the aforementioned as commercial ideas, it was evident that there is not much to sell without purposefully gating features from users that are not intrinsically valuable. While deliberating this and researching the Laravel framework, the site Laracasts came to mind. It is essentially a site like Udemy but specific to Laravel video tutorials. This idea of introducing additional educational aspects to Reflux is a major point when considering future prospects. Unlike code blocks and arbitrary user posts on their own, meticulously designed course material, conjunctively with its established feature-set, would be something of monetary value. The introduction of this would involve heavy reworking of the project in its current state to better segregate content based on the type of content it is (i.e. questions, solutions, tutorials). In this sense, it would be differentiated from all competitive technologies whereby correlations can be drawn, as the system would represent a unification of many systems that fulfil a market in isolation.

REFERENCES

- ❖ Otwell, T. (2018). Laravel - The PHP Framework For Web Artisans. [online] Laravel.com. Available at: <https://laravel.com/docs/5.6> [Accessed 8 May 2018].
- ❖ Way, J. (2017). Laravel 5.4 From Scratch. [online] Laracasts. Available at: <https://laracasts.com/series/laravel-from-scratch-2017> [Accessed 8 May 2018].

OBJECTIVES

The objectives of the following project are to develop a rich internet application encompassing social networking features such as user accounts and profiles, as well as to provide a comprehensive platform from which this user-base can submit and upload code-blocks in a variety of languages that will be displayed with their syntax preserved in terms of white spacing and highlighting. Users can vote to increase or decrease the popularity of a particular submissions, sort the content based on this value and other variables, and search for content based on the name of the post entry or the language tag assigned to the post. User credentials will be encrypted using an internally developed algorithm to ensure the security of their account as well as any sensitive data attached in conjunction with the creation of the account. Administrative roles will be assigned by the system inclusive of discretionary measures and operations that can be performed by users with higher level authority which provides a necessary level of access control functions, blocking access to certain pages or tampering with submitted content that is not of the origin of the user whom submitted it.

In order to facilitate these underlying functionalities, a number of elements must be established. This means the development of a GUI which will later intrinsically link to the applications functionality. This must be clean and comprehensive. In this sense, we are talking about both the graphical, interactive components of each respective web page, as well as the web pages themselves, consisting of registration, login, profile, content submission, home, and administrative panel, for instance. Following this, the pages themselves describe the required underlying functionality that must then be developed. In other words, a user must be able to register an account, login to an account, be able to submit a code snippet and label it within the spectrum of an appropriate coding language, update or remove their own submissions, comment and rate the submissions of others, and add details and images to their respective profile page for other users to view. In terms of security, this aforementioned user information will need to be encrypted appropriately. In essence, it must be a completely secure application. This means, for example, the usage of weak encryption such as md5 is not a viable implementation of security. The idea is thusly to develop an encryption algorithm that is unique to the application itself, borrowing from existing concepts and implementations of encryption to hopefully concoct an algorithm that outperforms the security of other web services and inspire users of the applications safety in handling their sensitive data. As part of the cybersecurity stream, this aspect of the application is one that is mandatory and of vital importance. It is the main selling point of the application besides the context itself. To accomplish this, it is important to review the existing popular techniques of encryption, as well as engage in a studious assessment of relevant literature. In terms of the data itself, this will of course need to be stored in a database. This will be done locally during the development of the project before it is hosted. These are the essential and necessary objectives that must be fulfilled to ensure a successful development cycle.

BACKGROUND

The origin of this project idea stems from a previously developed project that did not have the necessary time allocated to be inclusive of the initial scope in terms of the feature-set conjunctively with a newly generated idea specifically for the 4th year software project. In the beginning, the project that was devised was to do with storing user credentials of various other web services securely. This was met with hesitance during the proposal stage, as the market of competitors is high, and in order to achieve a high mark, I would essentially be required to develop better encryption than the competition, which consists of products that are the result of a large, established organization. After being given permission to develop this project, I met with my project supervisor to discuss the details as to how I would go about developing a project that had a seemingly high magnitude of expectations that would need to be filled in order to have a chance of achieving a 1:1. Discouraged by the venture, I expressed my desire to have saved a previously explored project idea for the 4th year software project. I was informed that it would absolutely be possible to revamp an old idea with the only stipulation being that none of the code from the previous project could be used.

That is the origin of the project idea in terms of revamping and utilizing the idea again for this software project, but the first time the idea was devised was due to an interest in providing a platform for developers to share code snippets for free that could help in a variety of avenues, from basic HTML layout for ease of access, to useful functions on how to accomplish certain CSS effects or jQuery smooth scroll transitions, as a for instance, but the scalability of the idea far transcends just those aforementioned languages. Within the current state of affairs this idea has been similarly explored yet implemented differently. There are online markets from which to purchase code, Stack Overflow with which to ask questions about code, and JSFiddle with which to share and showcase working solutions to coding issues with an output of the function of the code. None of these examples have any overlapping functionality when compared to one another. In essence, Lexicode was, at the time, an idea generated to alleviate and unify these platforms into a single, comprehensive web application providing the social connectivity and feedback systems that Stack Overflow provides, mixed with the ability to showcase blocks of code akin to JSFiddle wrapped within a sleek format in which each element of the application serves to provide clarity to the user as to its purpose and function with no muddled confusion.

Some of these aspects were explored in the previous iteration of the project that was successfully completed as a part of Academic Internship, but the time allocated was not enough to implement everything that was required to unify the platforms. Specifically, in terms of the social interactivity between the user-base. It was non-existent. This older implementation of Lexicode provided the basis for something that could still be improved upon, hence the desire to return to the idea to realize the potential that its initial scope had defined previously. In light of the criticisms I had received for the initial project idea, the idea of Lexicode was in the back of my mind as the concept I wished I had presented in the first place, with the only thing stopping me being that I had done the idea before.

TECHNICAL APPROACH

Arguably, the most vital aspects of the project lie within the areas of how well it encrypts data and how secure the authentication procedure is in order to access a set of users' sensitive information and submissions. For this, I am undergoing thorough research of the subject of cryptography through revision of literature relevant to the subject in order to learn methods of encryption and how they can be applied to my chosen implementation language with respect to the backend. In turn, this will also highlight some of the base requirements of the project that will help to define its scope and avoid scope creep. Most, if not all, of these requirements lie in what the application hopes to provide as a service and in how securely it stores data and authenticates users. These are aspects and components of many applications. As previously discussed, there are some existing examples of the project idea on the market, albeit not in unified states, that, through research, provide a well of knowledge that which to draw from both requirements that they satisfy, and necessary features that should be implemented within my own iteration of the idea. This helps to improve upon these existing features as well as nail down unique selling points. In terms of this application, that will be through better encryption and the unified and comprehensive format of the application. The project will be developed on a local machine linked with a database and tested under local circumstances before being implemented in the cloud for testing in a simulated deployed state.

TECHNICAL DETAILS

The project will employ a number of frameworks and languages conjunctively. Firstly, the latest version of Bootstrap will be used for its responsive grid system as well as a number of useful UI components it provides. Font Awesome, for its provision of icons to suit any scenario or web application. With respect to the server-side functionality and the MVC architectural pattern previously discussed, the Laravel PHP framework will be used for its wealth of useful features and efficiency in condensing PHP down into simpler components. Inclusive is the blade templating engine which enables the creation of partials for reusable components within the application, such as the navigation. It can also be used to maintain the same layout on each page with only certain pieces of content changing. For local testing purposes, the Atom text editor and Sequel Pro database GUI for OSX will be utilised in the development of the application as well as the various command line dependencies required by Laravel in order to create a dedicated Laravel project, install particular useful components, and tinker with the application to test the storage, updating, retrieval, and deletion of data from the database. Later, the Cloud9 IDE will be used to test the application within a simulated deployed state.

EVALUATION

In order to evaluate the applications correctness functionally and securely, incremental and system-wide tests must be performed both during and post-development. The application has several components of functionality, most of which do not communicate with one another. This decreases the likelihood that integrating them into a singular system will introduce problems. Firstly however, I will conduct unit testing in order to evaluate that these components that make up the application function as expected when needed or requested individually. This includes registration, logging in, submitting code blocks, encrypting credentials, account management, access control, vote and comment on a submission, and updating or removal of submissions. Many of these components touch the same information but are not intrinsically linked or dependent upon the other with the exception of the fact that in order for the modules that involve the retrieval of information to do just that, they require a module that submits the information that they are setting out to retrieve. In this fashion, integration testing can be performed to test that these elements are working in unison as they are affecting the same data. The final performable test is a system test, that which involves going through each element of the application progressively as a whole to evaluate whether or not everything is functioning as it is expected to. This testing stage follows the same process as evaluating the system with an end user, in that we would have them create an account, logout, login to the account, submit a code block, vote and or comment on a submission, manage their own submissions, and manage their account. If all of those components are successful, then the evaluation is a success.

PROJECT PLAN

TASKS

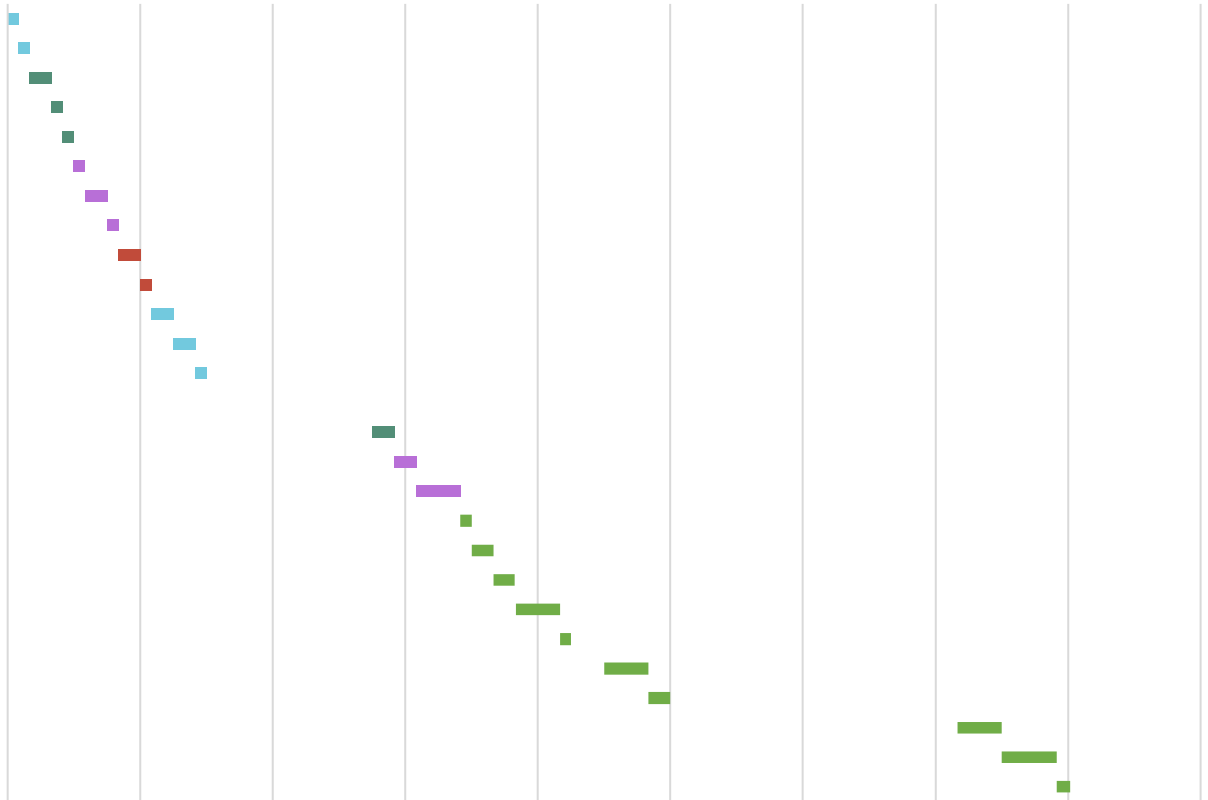
Task Name	Start Date	End Date	Duration (Days)
GUI			
Registration	28/10/2017	29/10/2017	1
Login	29/10/2017	30/10/2017	1
User Profiles	30/10/2017	01/11/2017	2
User Settings	01/11/2017	02/11/2017	1
Content Submission	02/11/2017	03/11/2017	1
Content Modification	03/11/2017	04/11/2017	1
Content Deletion	04/11/2017	06/11/2017	2
Content Filtering	06/11/2017	07/11/2017	1
Content Tagging	07/11/2017	09/11/2017	2
Content Searching	09/11/2017	10/11/2017	1
Rating System	10/11/2017	12/11/2017	2
Comment/Replies	12/11/2017	14/11/2017	2
Night Mode	14/11/2017	15/11/2017	1
Functionality			
Registration	30/11/2017	02/12/2017	2
Login	02/12/2017	04/12/2017	2
User Profiles	04/12/2017	08/12/2017	4
User Settings	08/12/2017	09/12/2017	1
Content Submission	09/12/2017	11/12/2017	2
Content Modification	11/12/2017	12/12/2017	2
Content Deletion	13/12/2017	17/12/2017	4
Content Filtering	17/12/2017	18/12/2017	1
Content Tagging	21/12/2017	25/12/2017	4
Content Searching	25/12/2017	27/12/2017	2
Rating System	22/01/2018	26/01/2018	4
Comment/Replies	26/01/2018	31/01/2018	5
Night Mode	31/01/2018	01/02/2018	1

GANNT CHART

28/10/2017 09/11/2017 21/11/2017 03/12/2017 15/12/2017 27/12/2017 08/01/2018 20/01/2018 01/02/2018 13/02/2018

Registration UI
 Login UI
 User Profiles UI
 User Settings UI
 Content Submission UI
 Content Modification UI
 Content Deletion UI
 Content Filtering UI
 Content Tagging UI
 Content Searching UI
 Rating System UI
 Comment/Replies UI
 Night Mode UI

 Registration
 Login
 User Profiles
 User Settings
 Content Submission
 Content Modification
 Content Deletion
 Content Filtering
 Content Tagging
 Content Searching
 Rating System
 Comment/Replies
 Night Mode



With respect to the Gantt Chart above, the deadline dates for particular components are based around the college schedule in conjunction with what is required of the project to be done incrementally. With previous experience working on this project idea, it is likely that the GUI and accounts functionality will be completed in time for the midpoint presentation with the potential to run ahead of schedule. It is also likely that certain elements that are due further along will be dabbled in to provide context for their implementation later and to showcase the uniqueness of the project early, in an incomplete state. This is especially true of the internally developed encryption which, as of present, has been assigned to start development following the Christmas holidays. However, encryption being one of the main selling points means there must be at least evidence of its involvement in the project in time for the midpoint presentation, hence the likelihood that it will in fact have a presence within the application prior to its actual designated development period, albeit in an unfinished format. Arguably, backend functionality is the most important aspect, but the reason it is placed so far along and completely separate to the development of the GUI is within the fact that visuals representing yet undeveloped functionality help concentrate focus on what exactly needs to be applied to these visual components, and has worked for me in the past. This also helps to cement the project idea in the minds of those who will be evaluating it, as it provides visual stimulus as to what the end-product will be once the functionality is intrinsically linked with the graphical elements further along in development. The accounts system has been placed as priority as it is a prerequisite to the functionality of the rest of the application due to the fact that in order to submit, edit, vote, and comment on submission, it is mandatory for there to be an accounts system implemented, as well as in the fact that I have the most experience with this component of the application from knowledge acquired in the development of previous projects throughout college. The aforementioned are the gist of the reasoning behind the project schedule outlined in the provided Gantt Chart.

INTRODUCTION

The following will be a journal documenting the progression of my 4th year software project from beginning to end. I'm hopeful this will become a resource that I can consult throughout the course of 4th year in its entirety and help keep in focus the scope of the project I'll be undertaking for a nine-month period. I am a computing student in the National College of Ireland, having started in 2014 when I was 19 years old. Currently, I am 21 going on 22 in November. I am confident enough in my abilities that I will have a coherent and functioning project by the end. If there's anything that I'm worried about, it's that I will fail in taking into account aspects of the project that will induce scope creep. This has been common place in the projects I've developed over the past three years, especially ones in which it was a combined effort as part of a team.

While I have successfully completed these projects, the increasing goals make it ever more difficult to be satisfied with what has been produced by the end. Another factor that did not help, is that the majority of these team projects left me as the sole programmer. I have witnessed this in abundance with not only myself, but colleagues I've found in similar circumstances. At this stage, I believe most of these individuals to have been weeded out, for lack of a better word. The reason I bring this up however, is that on paper the goals of these projects were clear, but with a lack of understanding from my teammates as to the code involved in achieving these, it was an impossibility for them to comprehend whenever we mistakenly forgot to take some aspect of some feature into account, and thus scope creep was imminent. I know myself what I am capable of, and because of these aforementioned facts I am all the better in terms of designing and developing the actual software. In order to achieve what I wish and hope to set out to achieve however, I have to be able to define the intricacies of the project from the beginning. This provides a vital foothold into what I can expect to need to do when the time comes when the development of a specific component comes around. That being said, I think my weakest aspect as a result is, in fact, documentation. Specifically, business-oriented documentation. This is prevalent even in the idea generation stage. It has been difficult to come up with an idea each and every year that has to satisfy a target market, have requirements and user stories defined, as well as be innovative or completely unique in some respects. This tends to result in a project idea that is oftentimes wacky and has no use case in the real world beyond being a college project. Or, if you do want to put your own spin on something that is already on the market, you're expected to either have a unique selling point or outdo the competition. The latter point is often outlandish, resulting in a David versus Goliath scenario due to the fact that the competition is generally the product of an entire organisation against a single student.

With respect to documentation, and this journal specifically, I expect its submission to be either early or late on occasions. Whenever an event of note takes place I plan to document it accordingly, as well as on days when actual development, presentations, or other assignments are due and or occur. The most important things I can keep track of that are the most beneficial to me in light of my weaknesses are on how the project is progressing and changing as time goes on. The fact that I am on my own for this project is almost no different from my previous experience in development, meant with no ill will. It just serves to highlight that I believe I will be able to handle the project going forward without support. As confirmed back in the Summer, my 4th year specialisation is Cybersecurity, which is an area that has been briefly touched on in other years, but one that I have found extremely insightful and intriguing. Knowing that most of what I have made over the years is nowhere near a level that one could label secure, I want to thematically represent security. Security is an aspect of applications that is invisible, because it happens unbeknownst to users, so I feel the need to devise a project that does this as well as having the context of being about security. An application that encrypts and protects users' sensitive data. The name I have for this is a combination of the word encryption and a pseudonym for weakness, kryptonite. Hence, Encryptonite.

SEPTEMBER 2017

SEPTEMBER 20TH, 2017

I devised a project idea while in discussion with a software developer friend of mine that perfectly encompasses what I want in order to create something that visibly represents security. It functions similarly to my last project which I developed during Academic Internship in college over the course of the Summer, which was a web application with social networking features designed to allow users to submit snippets of code in varying languages that were all displayed perfectly in syntax with preserved white-spacing. In terms of the context, the idea is actually completely different, but the underlying functionality required to create it is quite similar. Essentially, the idea is to create a web application for storing users' credentials for other web services that they elicit the use of. Storing their details such as usernames, emails, and passwords. Using this application would allow them to instead opt to using unique passwords for every service that they use. A password generator would also be present, allowing the generation of a seemingly random string of characters that, combined, form a strong password. This would have varying levels of customisability in terms of length, and whether or not a keyword is included. One issue with this idea is, again, that there are several competitors with existing products of their own that function the same, if not better. However, I knew this even when the idea was devised. My reasoning for wanting to do it doesn't lie in any potential post-course monetary gain or future prospects, but as a personal technical challenge and as something to showcase that can also represent the cybersecurity stream better than a project whose base idea has no relevance to security whatsoever.

SEPTEMBER 25TH, 2017

Even though I have the opportunity to create any idea I want, with the only stipulation being that it has to be completely secure, I am opting to do the idea I devised previously with my friend. I think I am capable of making something that, at the very least, matches some or all of the competitors out there in terms of functionality and aesthetics. It may not be a project of grandeur or innovation, but I hope that my own twist on the idea with a unique selling point here and there to be enough for the project to be accepted when the proposal presentations commence next week. My only worry is what I have already addressed here, that it will be too generic or not innovative enough to do. However, I have no qualms announcing my lack of interest in creating something just for the sake of it being completely unique. I would like to do something that I am interested in doing.

OCTOBER 2017

OCTOBER 2ND, 2017

I was second on the list of the entire year to propose my project to three judges. These were Thibaut Lust, Arghir Moldovan, and Pdraig De Burca. The first of these judges I was not acquainted with, but Arghir is my Artificial Intelligence lecturer, and I also had Pdraig as a lecturer during my Academic Internship over the Summer. They essentially brought the exact things that I was worried about up during the proposal. Arghir specifically said what I have already stated in this document, the word "generic". I was fine with this as I told them I'm not interested in creating anything with any future prospects, to which I was told that marks are going for innovation, and that I probably couldn't get a 1:1 doing this project. Pdraig was probably the harshest in a sense, in that he spent less than a minute looking at his phone just to show me the number of competitors there were to my idea on the Google Play store despite the fact I already know, considering I researched my own idea. Another aspect of the proposal that was bizarre, was extensive questioning behind the password generation component of the project idea. Thibaut asked if the generator could be used to crack passwords. I did not understand this, but my guess is that he means if someone uses the same keyword as someone else, would it produce the same result?

I told them that most likely this module would produce a random string of characters with options that are tweakable by the user. To this Padraig said that “there is no true random”. I was confused as to the relevance of this statement with respect to the project, especially for a component where something that’s randomly produced has no weight on the overall security. It’s merely a useful component for producing strong passwords. That’s the idea anyway. Because of what was said though, I was absolutely certain the project idea was going to be rejected. None of them seemed interested at all, especially after the remark about me not being able to get a 1:1. To my surprise they let me go ahead, albeit with some reluctance and a huge stipulation that I need to have a unique selling point, whether that be something none of the competitors are doing or do what they’re currently doing better. I think I’d find the latter quite something to accomplish considering the alternatives are the result of an entire organisation of people, like I’ve mentioned before. I’m not sure what my unique selling point could be, but at least I get to do the project.

OCTOBER 20TH, 2017

I’m starting to write up my project proposal. It’s a little vague so far, as I’m still not confident of all I want to be in it. I think having some level of ambiguity is helpful however, as it provides some breathing room as to how expansive a certain feature can be. I need to be careful in case it expands the scope though. I’ve also organised to meet with my project supervisor Glen Ward next Thursday. Hopefully he can give me some insight into having a unique selling point as I still haven’t nailed one down. I had the idea of providing a service that is a comprehensive guide to the various levels of security you’ll find on the internet, such as multi-factor authentication and captcha, for instance. I’m positive nobody else is doing such a thing, as it’s quite a quirky little feature that isn’t exactly necessary. If anything, I think it would be a quality of life feature, as there are certain aspects of security that frustrate users. Knowing what to expect and what additional measures to elicit might alleviate this some.

OCTOBER 26TH, 2017

Today I met with my project supervisor, Glen Ward. At this point I was starting to feel that doing this project was no longer a good idea and that, despite the harsh criticisms I received, it might be for the best. I told Glen that I would rather they had rejected the idea outright. The complexities behind it lie in the fact that I would have to devise my own encryption algorithm that outdoes the competitions. This would be an enormous undertaking, and is the only security-related unique selling point that I could possibly have. I expressed my worry of going forward with this project to Glen, and that I’d prefer if I had saved the idea I did in Academic Internship for my software project. Glen gave me some very useful and encouraging advice, and to my delight he said that there was no problem in rehashing an old idea, the only stipulation being that I can’t use any of the code from the previous iteration of course. So, while I was initially deadest on Encryptonite regardless of the criticisms I received, Glen told me that it was perhaps one of my mistakes in devising the project idea in that I was trying to create something purely about security, which is not mandatory. Overall, I’m quite happy with the outcome of today’s meeting, and I plan to meet with Glen again the Thursday I come back from reading week. The only sacrifice to changing my project idea at this stage is that I have to redo my project proposal. That is okay, as in the long run I will be working on a project idea I never had the time to fully flesh out, as well as having a feature-set that has a little bit more meat on the bones than Encryptonite did. Keeping track of this change in this journal is slightly enlightening in that this entry essentially completely voids what I had initially set out to do, which was even mentioned in the introduction, including the name I had for the project. You really can’t account for everything. Besides this, no more focus will be placed on the project until the reading week due to the need to work on other assignments, specifically, Strategic Management.

NOVEMBER 2017

NOVEMBER 1ST, 2017

It is the Wednesday of the reading week. So far, I haven't had much time to research or plan anything about the new project I'm doing. I did however, manage to completely redo my project proposal before the deadline. I did it the same day I met with Glen. He told me it would be okay to leave it as it was until the midpoint presentation, but I figured I already knew what to write as I've dabbled in the idea before. He wants me to research how top organisations like Facebook implement encryption, for the purpose that I will still be trying to develop my own method of encryption regardless of the project change. It is still the main selling point as I am in the cyber security stream. The new idea allows for a lot of breathing room though, as even if I fail or only partially implement the encryption, there's a plethora of other features to fall back on. If I had gone with Encryptonite, regardless of success or failure, I wouldn't have much else bar encrypting user credentials. I think it was the right move, and the project itself was very fun to work on the first time through. I've been concocting new ideas that I couldn't hope to implement in the time-frame of any previous projects. In general, you have a month or two at most. Firstly, I plan on using Laravel. It's a PHP framework that utilizes MVC. I've never made a project that wasn't a complete mess under the hood in terms of code placement, directory organisation, and naming conventions. Functionally and aesthetically sound projects, but messy. Over the Summer I dabbled in Laravel just to see how it works and I've been very impressed. Not much else is happening with the project other than thoughts at the moment. Currently I am working on an assignment for AI, Web API, and studying for a test in Secure Application Programming for the following Tuesday. Hopefully I'll be able to engage the project a little more after that. I'm looking forward to the second semester because there is more focus on the project there than there are other classes. There are a lot less than now. If we had even one less class my stress would be significantly reduced.

NOVEMBER 9TH, 2017

I met with Glen, my project supervisor, again today. He's a great supervisor to have and really puts you at ease with respect to what needs to be done. There's no miscommunication. We spoke about my progress with the project so far, which is next to none on the coding front, but there's been a lot in the development of the idea. For starters, we discussed the possibility of utilizing an API to validate the code snippets that are uploaded. This cannot be done for all languages, but would be useful for the ones it does support. He does not expect me to develop my own compiler or anything like that, just use what's out there. It is an interesting idea that I'm certainly considering. Firstly though, I want to focus on setting up the user accounts systems and implement the uploading of code snippets on a basic level. Essentially, I hope to have somewhat of the project I developed the last time ready for the midpoint presentation. If I do, I'll at least be able to provide visual context as to what it is I'm doing. I have my doubts that I'll have code validation by the time the midpoint rolls around, but I'll be trying to implement as much as I can to at least be able to comprehensively explain my roadmap. In terms of the security aspect, the best I have at the moment is the encryption of sensitive user data, with the selling point being that the algorithm will be developed by myself, piggybacking off of some of the better implementations out there and how they accomplished them. At the very least, with this idea I have a larger scope of features to fall back on if one or two do not come to fruition, but for the most part I'm confident enough that I'll be able to develop the majority of what I've set out to do.

NOVEMBER 29TH, 2017

I haven't been in college last week nor this week due to the workload and deliverables which are required by the end of this week, next week, and the week after that. There is so much that needs to be done in such a short span of time that it's baffling the semester was structured this way. To surmise, I need to have a prototype for my software project, the technical report of the software project, the AI project, and the API project done this week alone. It was my birthday yesterday, but you wouldn't think it to be honest. I was doing my software project.

I was supposed to meet Glen last week but I decided to instead opt to stay home and work on all of this stuff. At the moment, I have a GUI for the project done using the materialize CSS framework. It offers a lot of good components for UI development. As of writing this it's only in its alpha stages, but it is still very impressive. I have also got it hooked up to a local database using Sequel Pro on the Mac.

I chose to use the Laravel PHP framework for my project which is built off of the MVC architectural pattern. I had dabbled in it over the Summer but never really made anything. I'm learning as I go along while developing the project. Surprisingly it is quite a bit of fun and it's an amazing framework. Once you have all of the necessary components installed it is a matter of minutes before you have a basic project up and running that's connected to a database of your choosing.

I'm worried about not having enough to show for the midpoint. The truth is I probably won't have anything major functionally, but that's okay in my book. Glen told me to maybe have code validation for a single language done, but to be honest, I don't think I'm going to go through with that idea at all. When you think on it, it's actually quite implausible. For example, if you're uploading a code block in Java, having code validation would mean it would throw all sorts of errors due to a lack of dependencies such as imports, proper declarations, and whatnot. Sometimes though, you just want to upload a single method, not have to write an entire class and have users sift through it every single time to find what they're looking for. I'll explain this if asked, I just don't think it's possible using an API unless you were to develop it yourself, which is not happening.

The rest of this week is dedicated to having something half-decent to show for the midpoint, the AI which is due this Friday, and having the Web API project up to scratch which is luckily a group endeavour.

DECEMBER 2017

DECEMBER 12TH, 2017

No updates in a while, nor have I been in college still due to the workload. I generally work better from home than in a rowdy class. Anyway, since the last time I have done my midpoint presentation for my software project, finished my AI chess project, and finished my Secure Application Programming project. With respect to the midpoint, it went extremely well. As is standard, your project supervisor is one of the individuals present alongside someone else. This meant I had Glen and Dermot Killen, both of whom were extremely nice during and post presentation. Dermot expressed that there was nothing of a coding support system like Lexicode or Stack Overflow when he was younger, and seemed quite delighted overall. The demo went smoothly, no hitches whatsoever. Worth mentioning as well, due to the curse of my surname conjunctively with alphabetical order, I was one of the first presentations to occur at 9:00am on the dot. Needless to say, it rejuvenated my ambitions for working on the project. I shook hands with Glen and Dermot at the end and they wished me the best saying that it went very well.

In terms of AI, getting done was tricky and it definitely isn't perfect. The deadlines for things were beginning to clash and so I got done what I deemed enough. In essence, random move and next best move work great, but the minimax is another story. It works to an extent, but I don't know if it's working as is standard for the implementation of a minimax algorithm. You define the search depth in the code, and at the beginning of the game the AI opponent goes into a visual frenzy of the pieces flashing representing it making a decision. From there onward though, this never happens again and it appears to make moves instantly. Other than that, I think it's enough to do well enough in the module.

Secure Application Programming is something that took somewhat of a reasonable amount of time to do, but was made easier by the fact that implementing security in a project are things I've already done for my software project. My project partner (Dan) and I, discussed what we should do for the project. Initially we were going to develop a playground of sorts that would allow users to test various security vulnerabilities while also teaching them what to do to protect against them. By the time we got around to doing this project however, the idea felt too ambitious, and when reading the project specification, it seems you could literally develop anything as long as it is secure. To this end we decided to develop a blog/forum centred around the area of security called... Encryptonite! This is the name of the first idea I had for my software project, so we are coming full circle.

Despite the aforementioned, the software project is what matters here with respect to this journal. However, it's worth noting any obstacles, that of which take the form of my other assignments. The plan going forward is to put the project on the wayside for the time being while I get everything else I need in order. This is the last week, and I have yet to finish my Web API project or present my AI and Secure Application Programming projects.

FEBRUARY 2018

FEBRUARY 5TH, 2018

I received my exam results today. Mostly good. AI was absolutely dreadful, as expected, but not from lack of trying. I figure it was due to the fact that it used to be a third-year module, which would make it a level 7 and now that it has been moved to fourth year its difficulty was adjusted accordingly. I believe it's very common for students to study a multitude of past papers in addition with a good Hail Mary prayer in hopes that some combination of these past question come up. For AI, this was not the case. It strayed very far from the previous standard set by eight years' worth of papers. As of writing this, a classmate created a poll on our Facebook student group to get the gist of everyone's results from the exam. Currently, out of 59 votes, it's sitting at a pretty 49.02% of students who achieved less than 35% in the exam. From there onwards very few got above 50%. Besides that, acing the chess AI was enough to pass me the module thankfully. With the rest of my results it's possible to achieve a 1:1 if I manage to keep the standard set. I'm banking on this software project since it's worth the most credits out of anything, and these previous modules were only worth five comparatively.

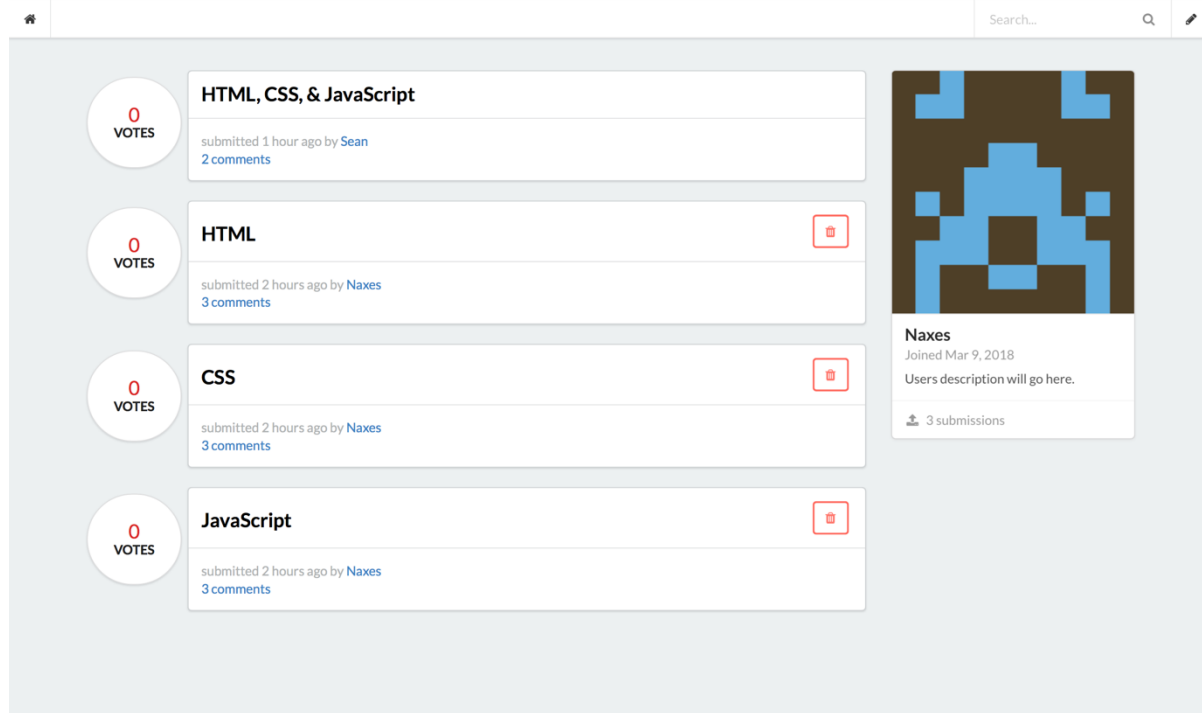
Speaking of the project, I have not touched it since the midpoint presentation honestly. When that was out of the way I absolutely had to focus on the myriad of other deliverables. What generally tends to happen as a result, and it has happened, is that I lose track of where I even was or what I was doing last time I worked on it. So, I'm pretty much restarting it from scratch, still using Laravel mind, but with a new CSS framework called Semantic-UI. This might sound crazy from an outside perspective but developing what I had developed previously for the midpoint was honestly not much at all, and the previous version of this project I had done in my academic internship took easily less than a month. Motivating yourself to do it though is another story. Chalk it down to a warm up, because I honestly don't even remember some of the Laravel commands and aspects of the MVC framework that I knew previously since it's been so long. Basically, I'm still confident that I'm on track. I also haven't seen my project supervisor since the midpoint, but at the moment, there isn't much reason to.

Additionally, I'm keeping in my mind's eye that there is an optional Analysis and Design Specification document upload for software project. That being said, my Advanced Secure Programming lecturer Irina Tal told us today that it is important to stick to deadlines and that this is also vital to ensuring you have a place at the end of the semester when it comes to showcasing your project. All I've noted from classmates from other streams is that everyone is completely enamoured with using raspberry pi's to accomplish things you don't even need it for. Regardless, I've just now begun integrating a blank Laravel project with Semantic-UI to see if I could get it working, as it's not as simple as just downloading it in the same vein as Bootstrap. It recommends a gulp installation, but none of this is tailored to Laravel or any other framework you might be using for that matter. Luckily, I found a solution using npm and moving a couple of files around. Following that, it's about settling on a design I like, getting back to where I was, and hopefully relearning what I used to know along the way using the previous iteration as reference.

MARCH 2018

MARCH 9TH, 2018

I've been working incrementally on the project over the past month. It's a slower process than anticipated but it's shaping up to be, at the very least, aesthetically pleasing.



This is the home screen. Basically, a centralised area where none of the content is sorted in any fashion initially. I'm using *gravatar* to generate a unique retro avatar based on an MD5 hash of a user's email. In this case, MD5 is okay. It's nothing more than the method required by *gravatar* itself to achieve this.

I'm unsure of how votes look next to posts here, but otherwise I'm happy with how this looks. Also, the segment encompassing the user's avatar and details on the right is fixed as you scroll through the content. This will be illustrated better when we look at another view. Before that however, the final aspect to discuss in this view are the trash icons next to some posts. These represent posts that can be deleted because they are your own. In the code this is simply hidden for posts that don't match your ID, but even if that were not the case, I am validating that you can only remove your own posts and not those of others under the hood.

The screenshot shows a web interface for a post. On the left, there is a circular badge with '0 VOTES'. The post title is 'HTML', submitted 2 hours ago by 'Naxes'. The content is a code block containing HTML boilerplate code:

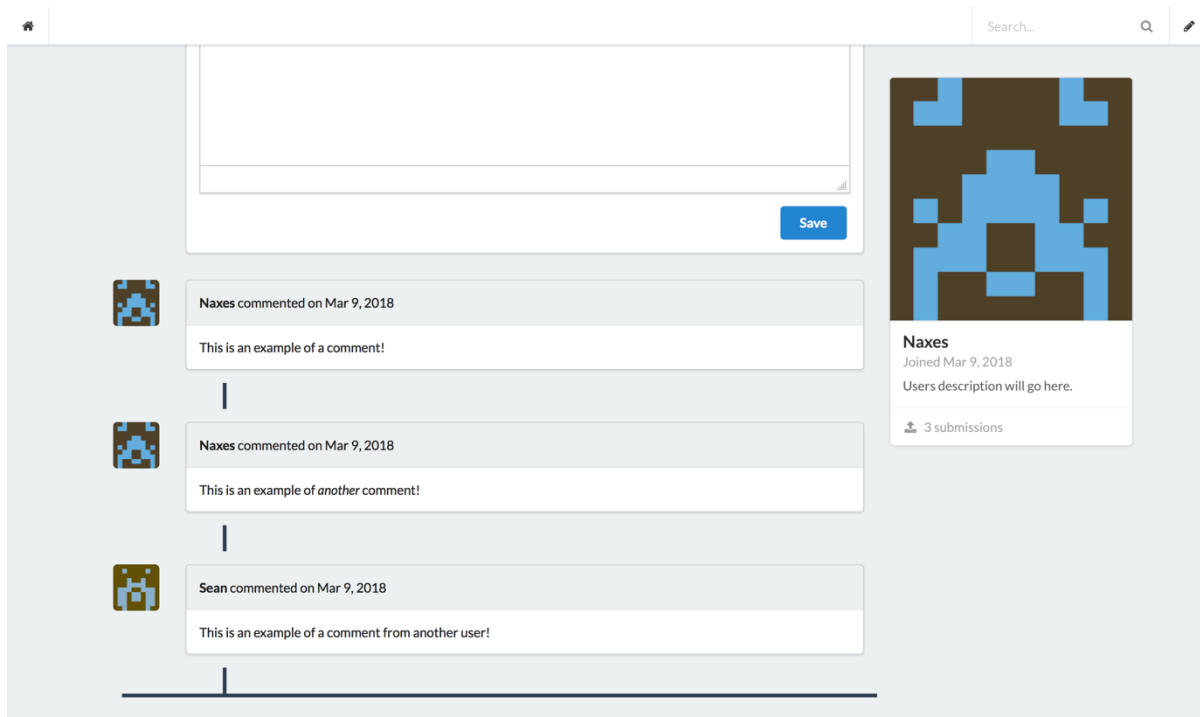
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
</body>
</html>
```

 Below the code block, it says '3 comments'. At the bottom, there is a rich text editor toolbar with 'Formats', bold, italic, and link icons. On the right, there is a user profile for 'Naxes', joined Mar 9, 2018, with a placeholder for a user description and '3 submissions'.

This view represents viewing the contents of a selected post. There's a couple of things to highlight. Firstly, as we can see, code blocks have appropriate styling thanks to PrismJS. Below this, I am using TinyMCE conjunctively with its code sample plugin. Funny enough, this plugin actually uses PrismJS just with the default theme that simply needed to be switched out with the darker one I'm using above.

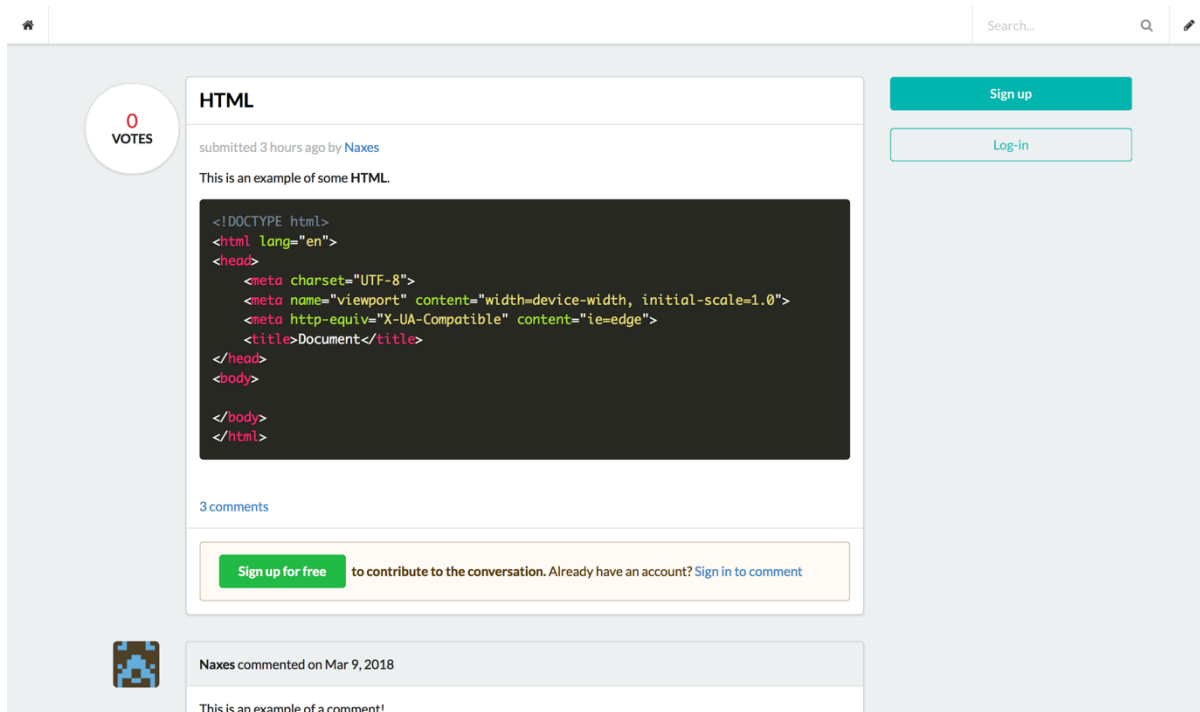
This allowed me to streamline a *lot* of the process I had originally implemented for posting content and comments. Basically, I had around four different fields for creating a post. These were title, body, code, and language of the snippet. It looked very convoluted.

With TinyMCE this has brought both the body and code sections together in one. In addition, since it handles assigning the language of the code itself, I no longer needed a field dedicated to this. Instead, I replaced it with a multiple selection input so that users can tag their content. The reason for this is that now posts can contain more-than-one segment of code.



This is the latter half of the same post. I'm just showing how the comments section is shaping up. I'm pretty proud of how this looks. It's inspired by how GitHub's comments look, inclusive of the timeline effect connecting the comments together. Since I'm working with a relational database, hierarchal data beyond a depth of two makes implementing replies a bit of a struggle. I may end up just leaving it like this, which by the way, is exactly how GitHub's is anyway. As we can also see, the segment containing our details follows us down the page.

The last thing here to show is that, when a user is not logged in, we show a nice message as opposed to the TinyMCE comment text area:



Speaking of accounts, the registration and login forms look as follows:

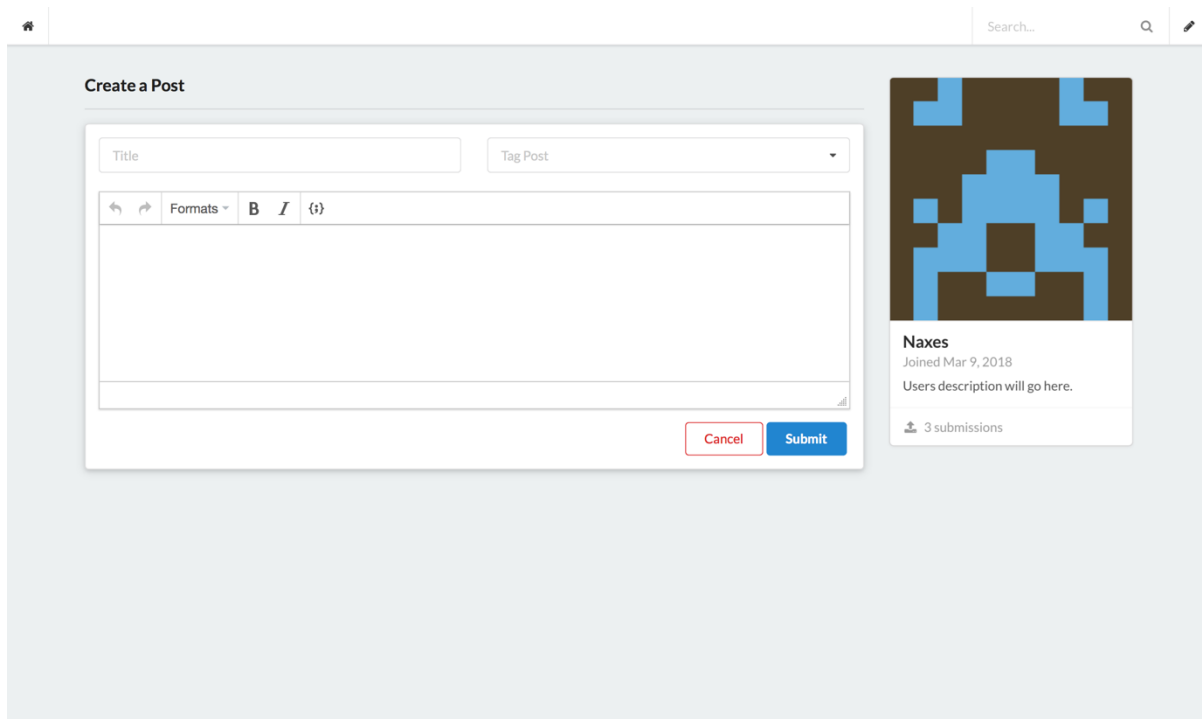
Register an account

[Have an account? Log-in](#)

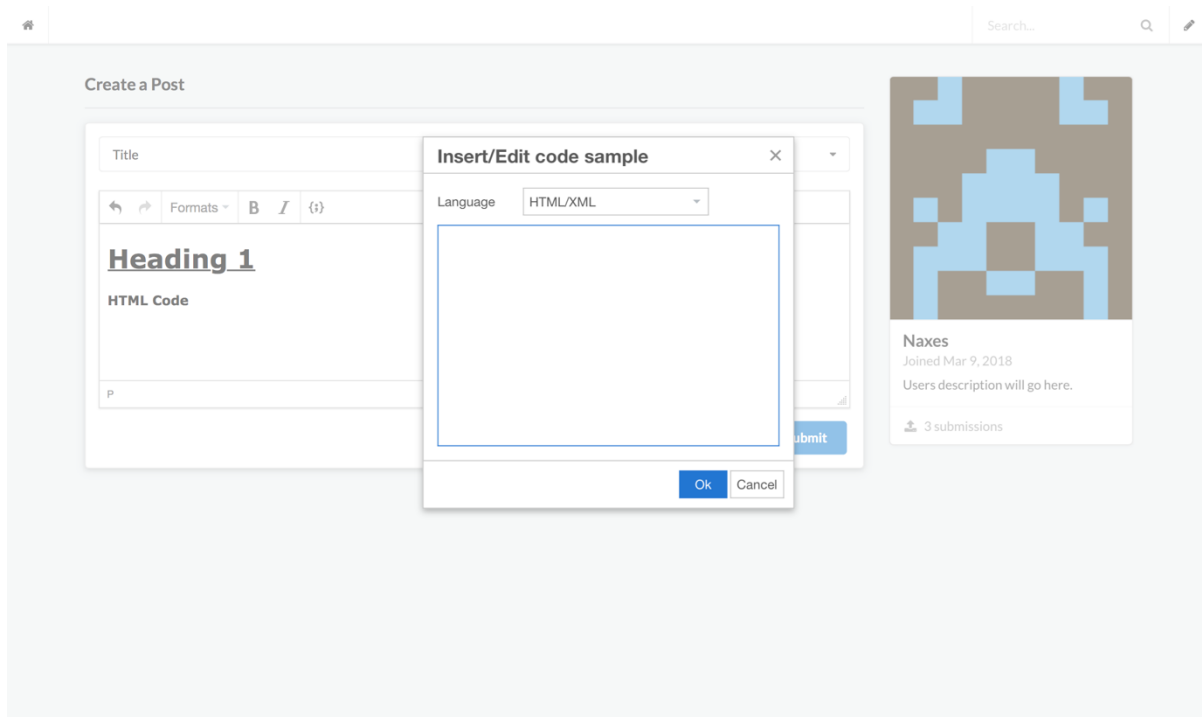
Log-in to your account

[New? Sign Up](#)

When a user comes to these forms, I ditch the master layout file that the rest of the site uses so that complete focus is on the form. I've seen many session forms styled this way in the past.



This is the final view I have to show off for the moment. As previously mentioned, this form used to consist of four different fields, each of which just did not sit well. It now looks for compact. Not only this, but users can utilise many rich text editing features that are simply not possible in a regular text area.



Here's how it looks when filling out the form all the way up until pasting in a snippet of code. A modal window becomes accessible for this purpose by selecting the rightmost option of the toolbar.

Once finished, the post looks as follows:

The screenshot shows a 'Create a Post' interface. At the top, there is a search bar and a home icon. The main area is titled 'Create a Post'. It features a 'Title' input field, a dropdown menu with 'HTML/XML', 'CSS', and 'JavaScript' options, and a rich text editor with 'Formats', 'B', 'I', and 'i' buttons. The rich text editor displays 'Heading 1'. Below it is an 'HTML Code' editor with a dark background, showing the following code:

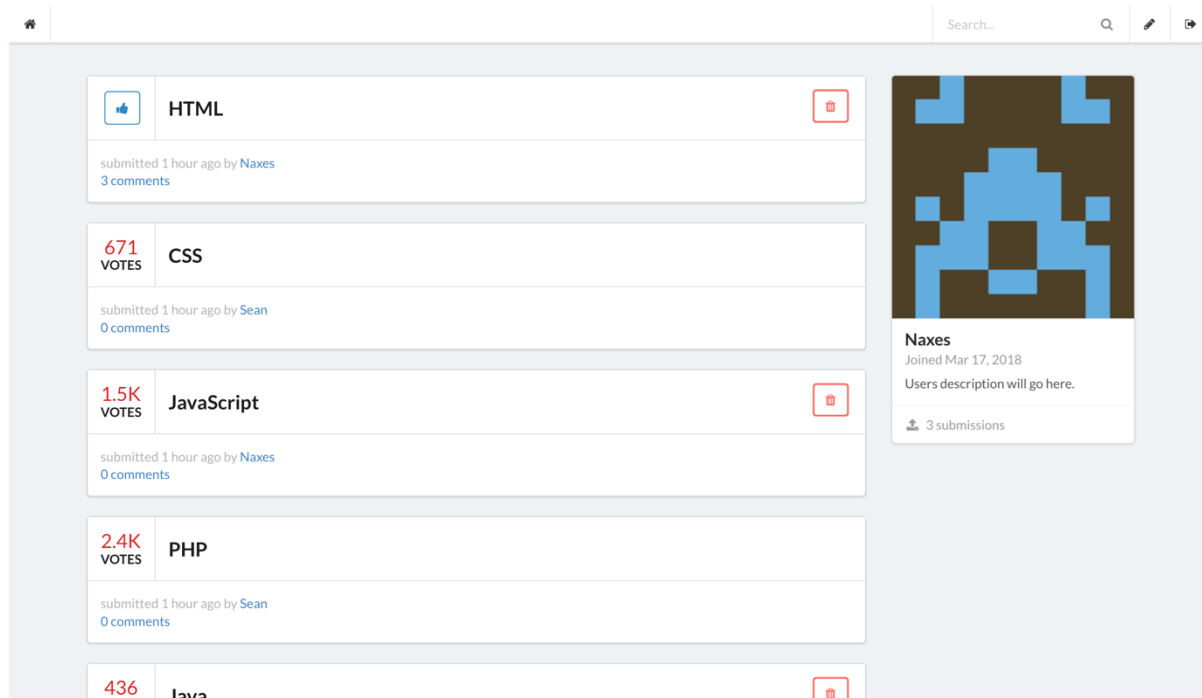
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
</body>
</html>
```

At the bottom of the HTML editor are 'Cancel' and 'Submit' buttons. To the right of the editor is a user profile card for 'Naxes', which includes the text 'Joined Mar 9, 2018', 'Users description will go here.', and '3 submissions'.

Now, we have a title, body, and some tags attached to the post. In this case, there is just HTML, but for showcasing purposes I've included multiple tags. The last step is simply submitting the post. The result will retain the exact rich text editing we've applied in the editor above. That's pretty much it in terms of the project at the moment. There is a way to go before I can consider it complete. I think it looks pretty good, but I'm biased. The aim is that it's clear, concise, and easy to use, so I hope that comes across at least. I'm not looking forward to the AJAX portion in order to get the voting system working. The last time, while I did manage it, it was very finicky and not consistent. I just never use AJAX in general and so hopefully this will forcibly shed some light on the topic.

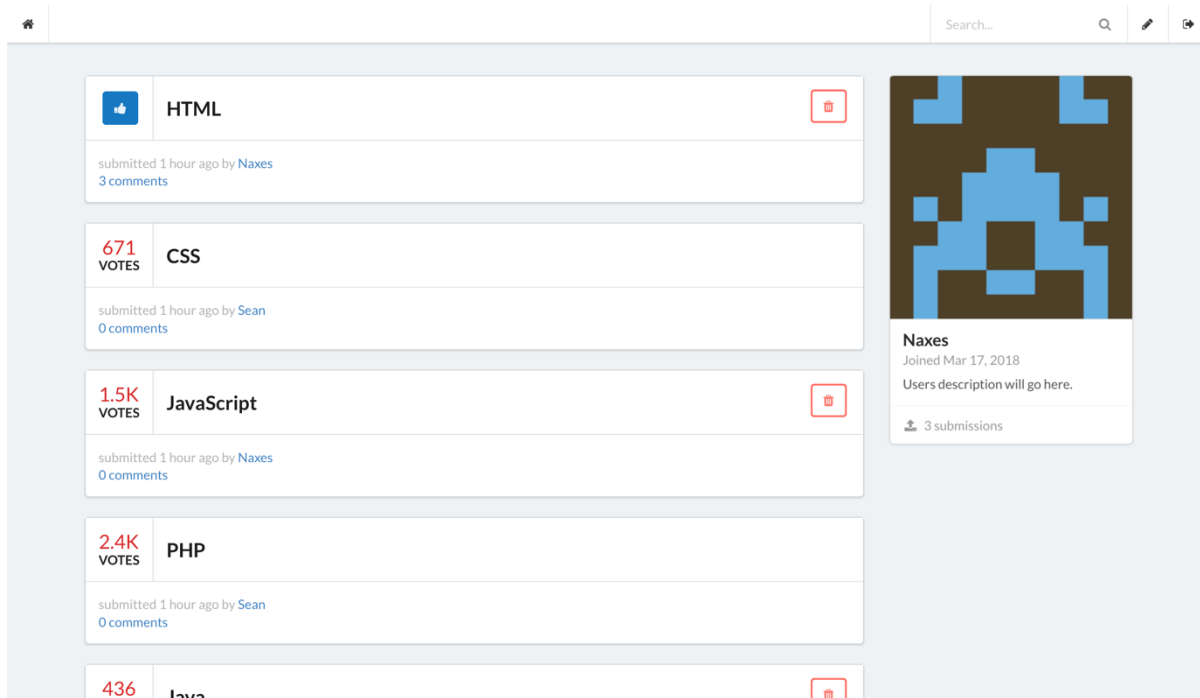
MARCH 17TH, 2018

Made some great progress since the last time. I have the voting system for posts working with ajax. Now, when a user hovers over the score of a post, a like button will be revealed with a neat animation thanks to a component of semantic:



Inclusive in this logic is that a user can only like a comment once, and once they have, we are checking if they have liked that particular post before by querying the database for an entry matching their user ID, the posts ID, and a Boolean value representing a like. So, if they *have* liked the post before, this entry will be removed. This could also be done with disliking a post, but I much prefer simply liking and un-liking something, especially since the style of button matches that of the delete component and fits nicely beneath the score segment. If I were, say, to include two buttons in here, there would be an overflow issue.

With ajax we are sending a post request to a route that triggers a method within a dedicated *VotesController* where this check is performed. Additionally, on the client side, we update the look of the button to illustrate to the user they have liked this post as well as reloading the portion of content representing the score. When the page is reloaded, the databases logic handles this aspect by, again, checking for all of the given user's instances of likes on posts, and updating the button and score accordingly to match the style attributed to having already liked something:



I ran into a bit of an annoyance when developing and testing as I went along with migrating the database over and over to test new tables, relationships, and functionality in that I had to keep making posts to conduct such tests. Luckily, I learned that in Laravel I can seed the database with a myriad of test data. So, all the posts you see in the above screenshots represent this seeded data, including the post content, score, user who owns the post, etc. Take for instance, generating a couple of user accounts I can login as from the get-go:

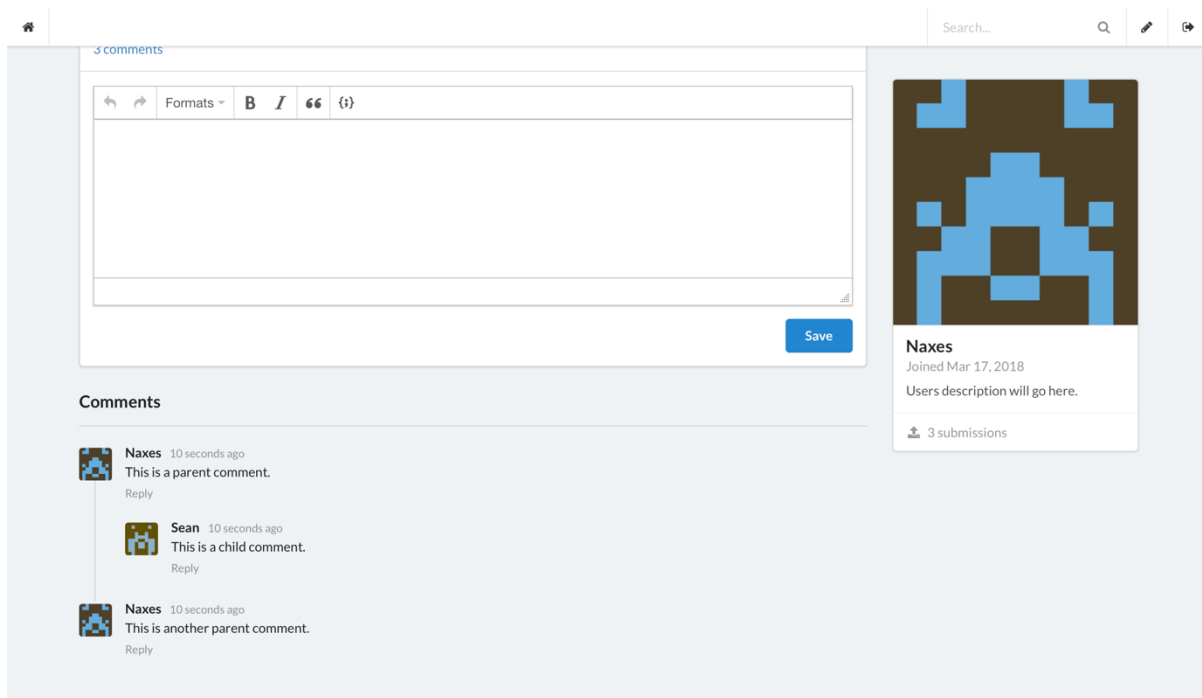
```
class UsersTableSeeder extends Seeder
{
    public function run()
    {
        DB::table('users')->delete();

        User::create(
        [
            'name'      => 'Naxes',
            'email'     => 'naxes@example.com',
            'password'  => Hash::make('tester')
        ]);

        User::create(
        [
            'name'      => 'Sean',
            'email'     => 'sean@example.com',
            'password'  => Hash::make('tester')
        ]);
    }
}
```

Looking at some popular websites like Stack Overflow and Reddit, it became abundantly clear that an extremely popular post is going to have an extraordinarily large number of votes attached. I took this into mind by formatting the number shown next to a post once it breaches a certain threshold. In this case, a thousand. Once a post reaches this point I format the number to one decimal place for readability purposes and so that we don't encounter some segments encompassing the score to become stretched and larger than others. The word votes under each score is what is determining the width of the segment, so if the score is less than or equal to this character space, all segments remain the same width.

In terms of the changes made to comments, I have implemented the database logic behind replies, which will be shown at an indentation depth of one at the most, even if we are replying to a reply. Currently, I have not put in the inputs for this, but with the logic there, it is as simple as hooking up the input to the comment. Additionally, I have changed the styling as I discovered there is a component of semantic specifically for comments. It is important to note, that this is still a tricky endeavour despite the components provided as they are simply CSS. I still have to build the logic around these CSS components.



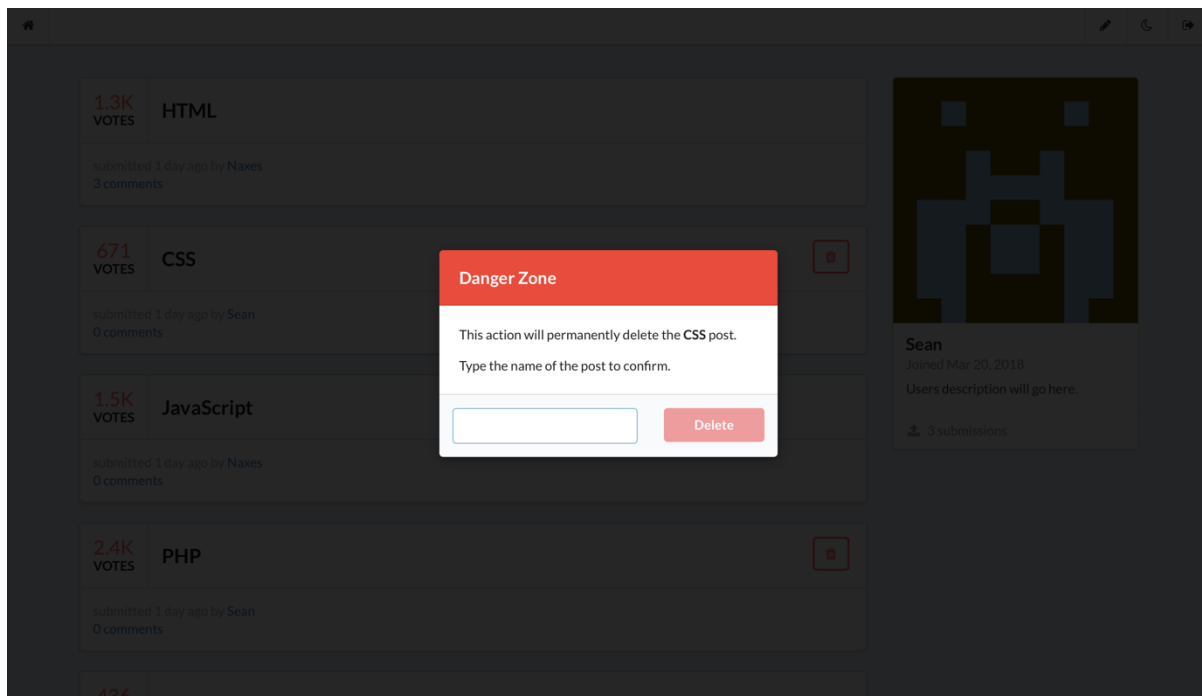
We still have a connector between comments, but now we have replies that I manually seeded in the database. Essentially, I included a new column called *parent_id* that is a *nullable integer*. When a top-level comment is posted, the value will be null, which represents a parent comment. When we reply to this parent comment, the ID will be set to the ID of the top-level comment. In the above case, the first comment has an ID of one, and a parent ID of null. The reply comment has an ID of two, and a parent ID of one. Now we know, using the parent ID, that that reply comment is the child of the top-level comment whose ID is one. This cascades down how many levels you want, but with semantics styling, setting up the logic in the view to delve deeper into the tree of comments past one is tricky. Instead, I will leave it as is but include the name of the user they're responding to such as @Naxes, for instance. If we have a reply of a reply, they will be at the same depth, but we'll know if they're responding to Sean, or any other user in the thread.

MARCH 21ST, 2018

I've completed the functionality for deleting a post. A basic query was already in place, but I had no confirmation popup or anything of the sort. I decided it would be a good idea to do it the same way GitHub handles the deletion of repositories. So, an extra step enforced in the validation process is that the user enters in the name of the post they're currently trying to delete. This is part of the query, and the form is required. On the JavaScript side, I made it so that:

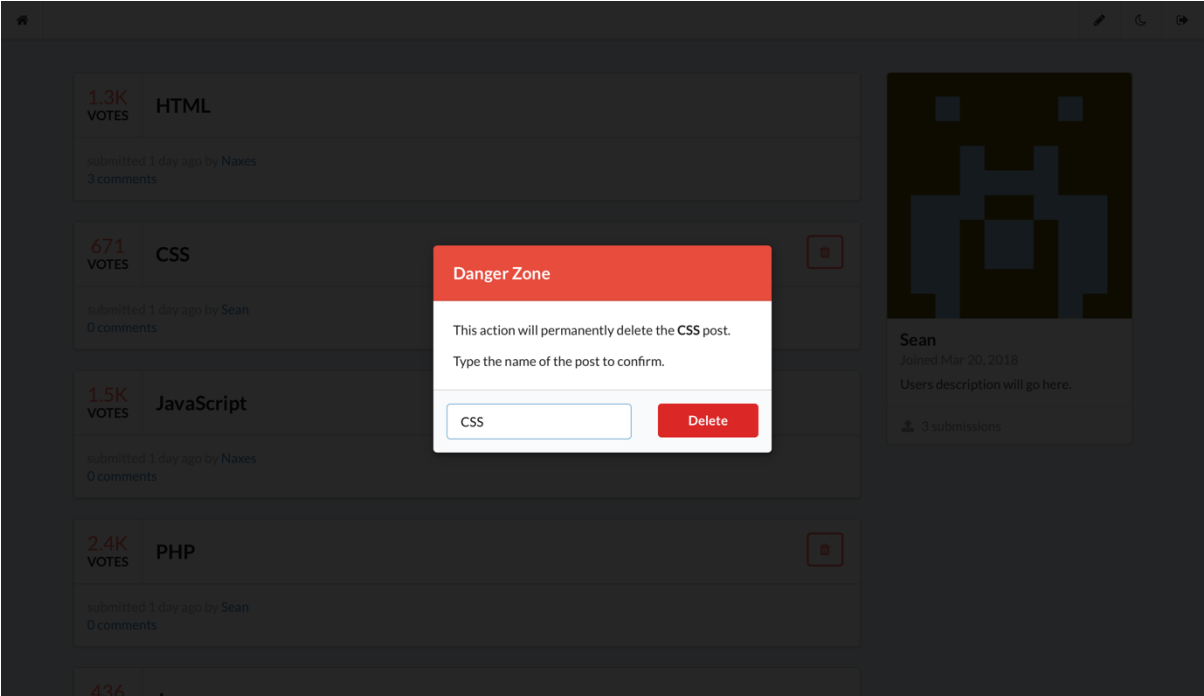
1. The form cannot be submitted by pressing the enter key.
2. The delete button doesn't become accessible until the correct post name is entered (case sensitive).

Even if JavaScript is disabled, we are handling the real validation server-side.

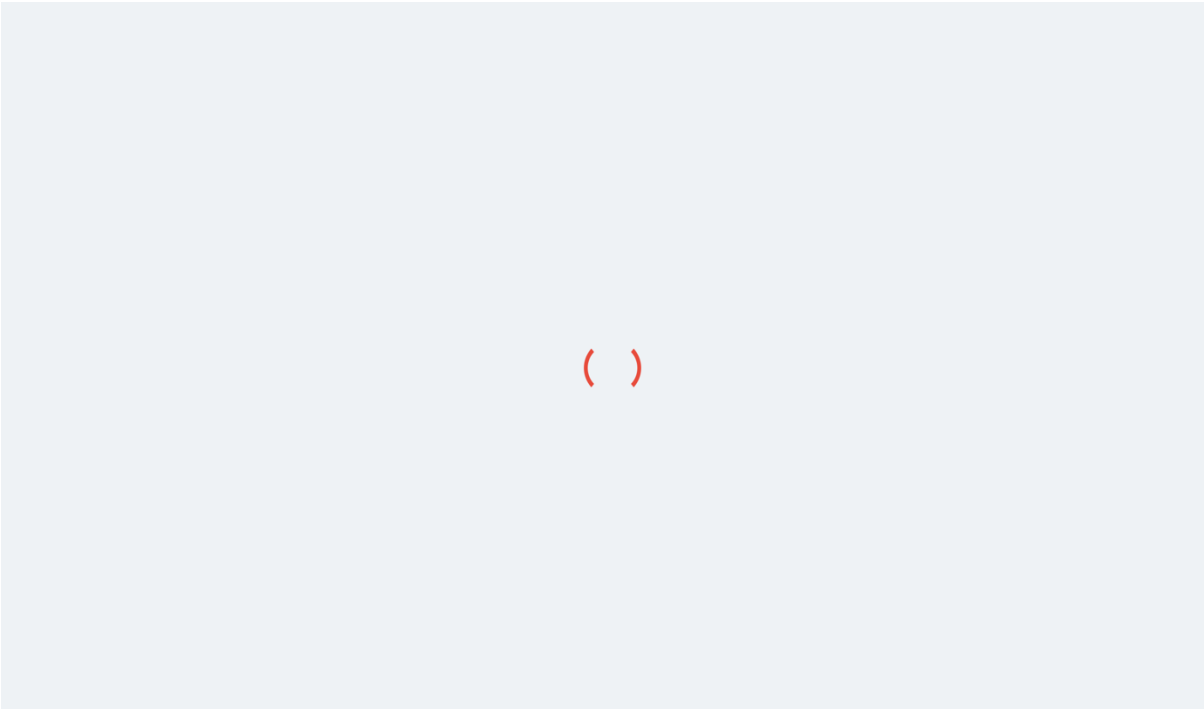


Example, user *Sean* is trying to delete their submitted post entitled *CSS*.

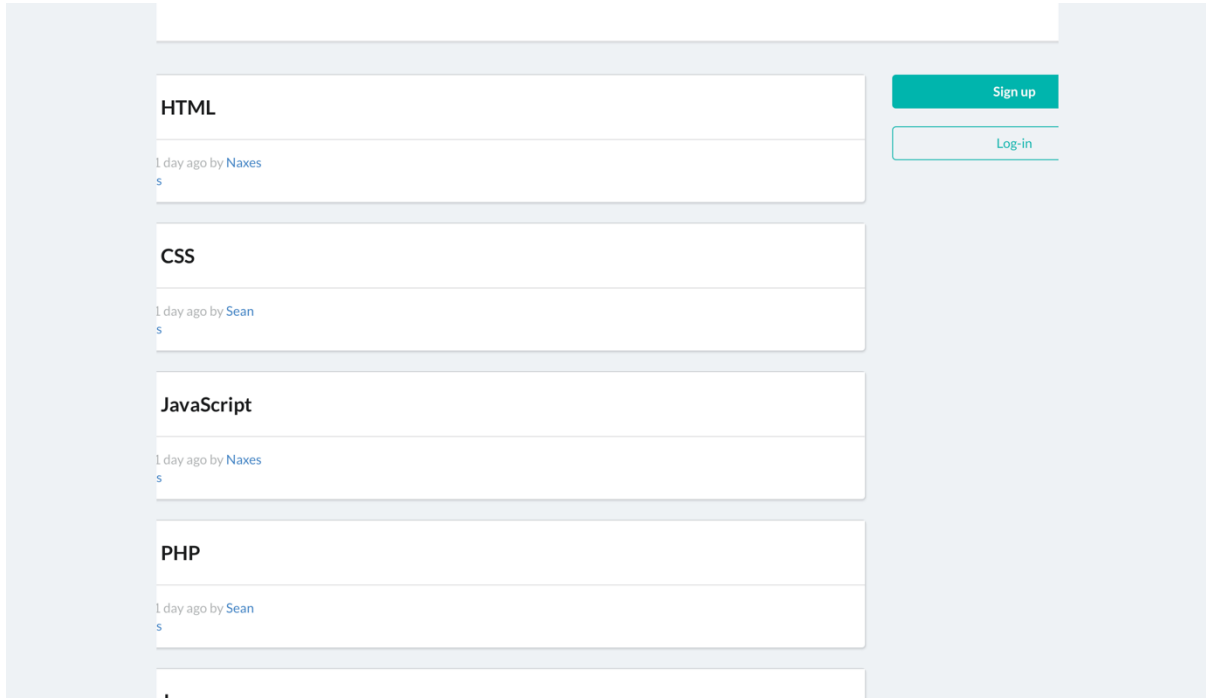
When we correctly enter the name of the post we gain access to the delete function. At the moment, ajax is also used so that the page is not reloaded and the entry is removed naturally, but I'm undecided on this.



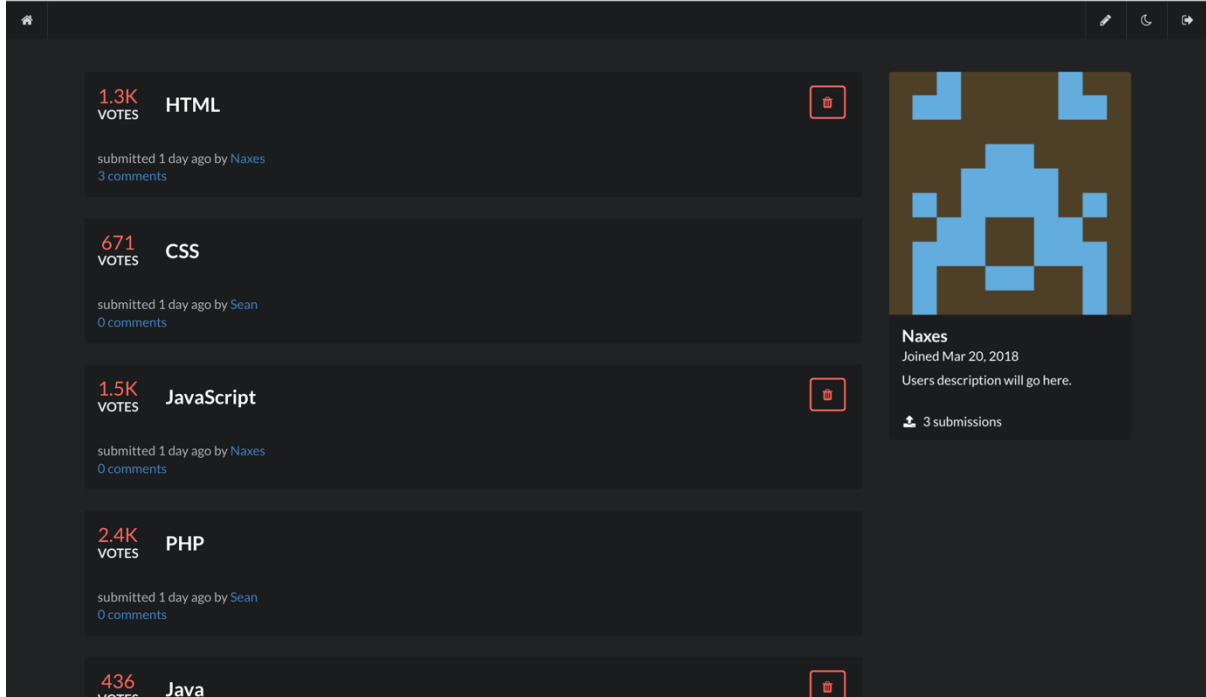
The next thing I made was a pre-loader. It consists of a spinning element, and two elements that act as curtains. When the page is loaded, these elements are removed stylistically. The first stage, when the page is loading, looks as such:



Naturally, the middle element is in a constant clockwise rotation. When the page has successfully loaded, only then do we animate the removal of these elements. Firstly, the spinning element fades out while the remaining two animate to either side of the user's viewport to simulate a curtain effect:

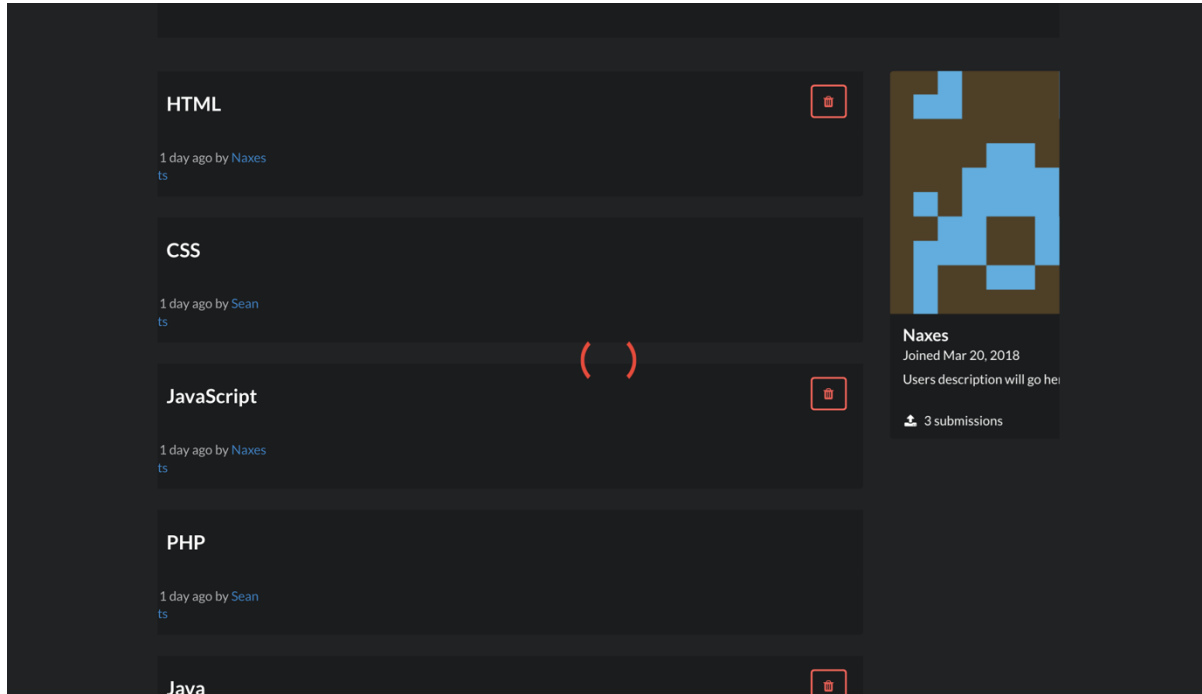


To show this, I've made it so that they animate only 75% of the way and won't disappear, in order to snap a screenshot. This pre-loader runs on any page linked to my master layout file. So, this excludes the login and registration forms, for instance. The final thing I worked on was something I'd always planned in my mind. IT's not exactly functional, but you see it in many web applications these days. Night mode.



It's a bit finicky to pick and choose the elements to style. For some, it was as simple as adding semantics' *inverted* class, but other elements don't support that. For the moment, I'm using a moon icon in the navigation menu for

testing, but this will be changed to a toggle switch in the future. Most likely nested in some dropdown menu. For this feature I'm utilising local storage. In essence, it's not something we need to tie to a user's account or store in the database in general. So, we just have a simple Boolean 1 or 0 representing the regular theme and the night theme. This also extends to the pre-loader, making it black!

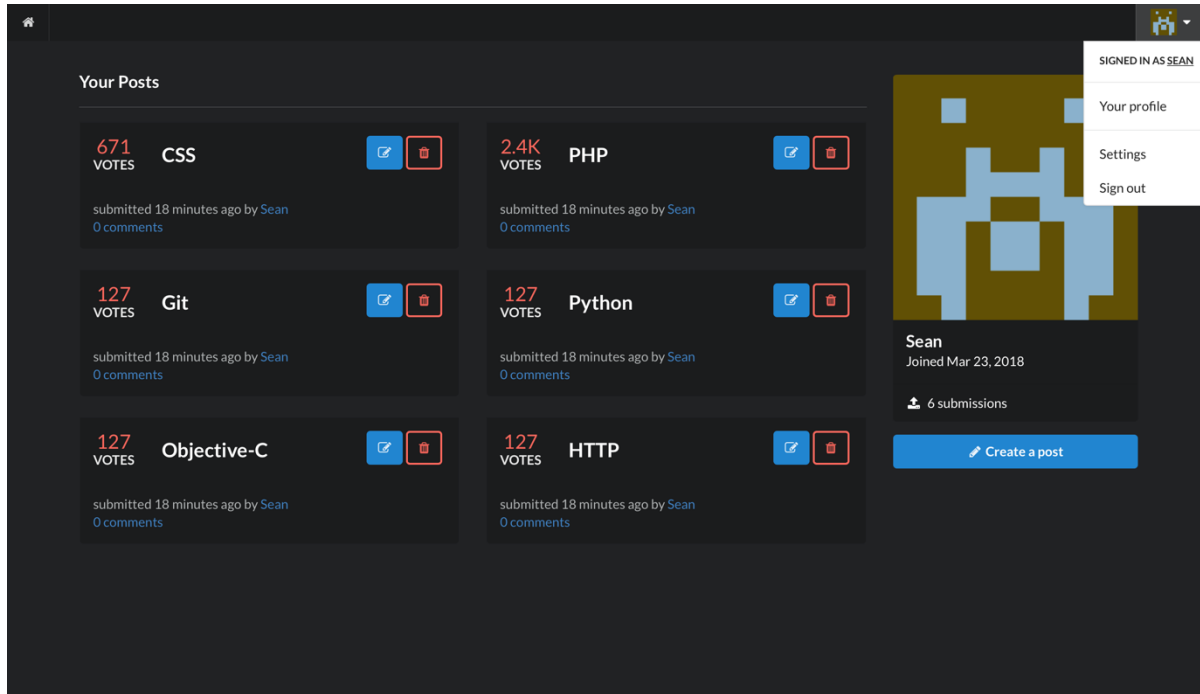


I might have the black pre-loader on both themes in future updates. Whenever I'm using the night mode, since it is the first element to load, it flickers white before the correct class changing it to the night variety is inserted. Plus, I just think it looks nicer.

Just to note, off the top of my head, I still need to do dedicated profile pages, editing user details, editing posts, editing comments (?), displaying the reply form, searching for content, sorting the content, tagging the content, and implementing an admin panel of some description.

MARCH 23RD, 2018

I think I've laid out how I want user profiles to look. It took some reworking of the side panel card containing user details, so some changes have been made on that front. In essence, we have this:

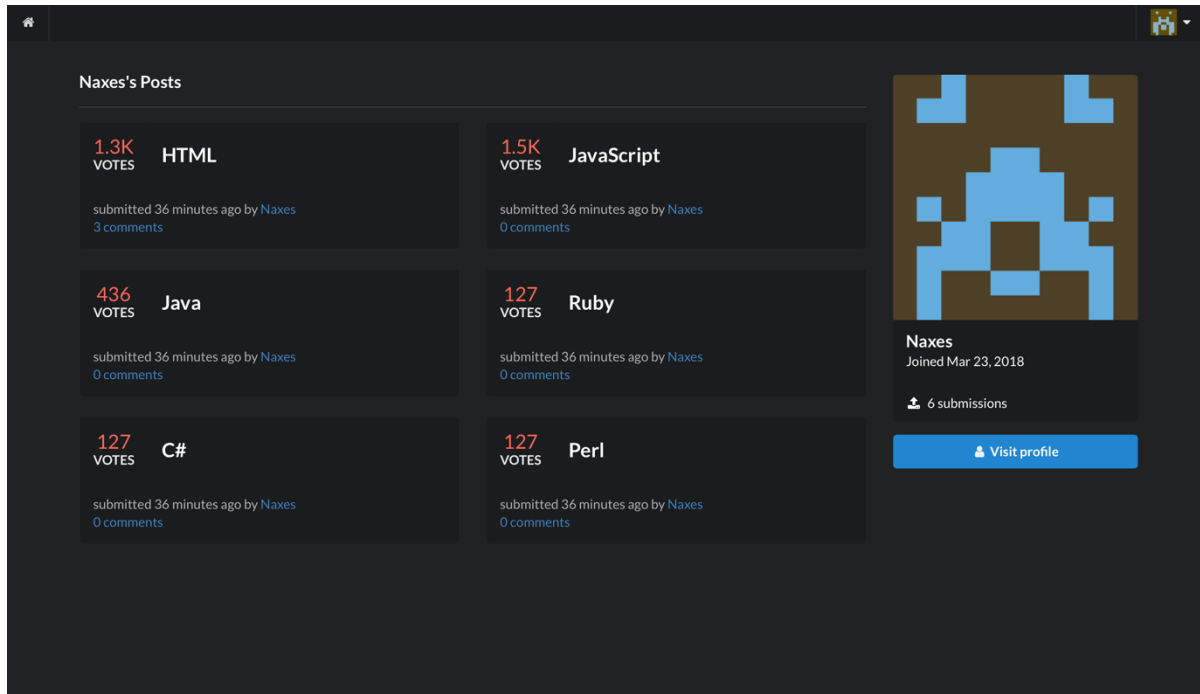


So, now I have a dropdown with a couple of options in preparation for other features. The rest of the screen shown represents our logged in user's profile page consisting of only the posts they've submitted. I've also added an edit button with appropriate modal window for when I implement that feature. Since I've only done this today, it's not finalised, but I think a more compact presentation of posts works well for the profile view. In addition, there is still pagination. The max on a single page here is six at the moment.

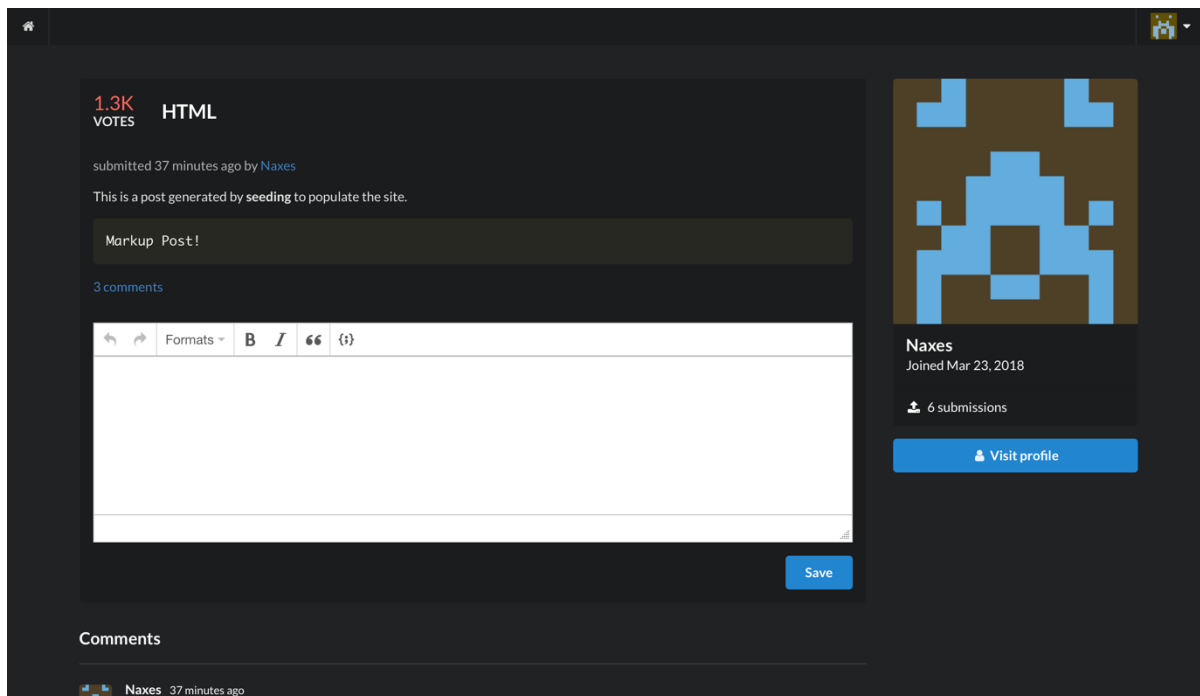
What's nice is that because we are extracting, for example, the delete button and modal from a different view as a partial, all I had to do was call it here and everything still works. I can still remove posts. Along the way I've tried to separate what I think are components that will be reused, but there are still some occurrences where this becomes awkward and a little unwieldy. The idea of using layout files and partials is in order to avoid instances where you have the same code in two different files, meaning if you need to change something, you have to change both. I've avoided this very well, but there are still times I have failed to do so. It's difficult to deliberate an element that is going to be reused if you haven't planned for it and it's this spontaneous decision.

Perfect example. With posts, there is no difference between how they look on the home page and how they look on a user's profile other than I'm using a smaller column size in the latter. However, with viewing a post, there *is* a difference because now we have to show the contents. So, making the whole thing a partial is tricky to do, because you're using some parts but not others. When it came time to link the user's name from a post to their profile I then had to change this in two places because the bottom segment containing this link also contains the body of the post while the bottom segment on the home page does not have this aspect.

As for the *settings* option in the dropdown, this is where I plan to let user's change their own details as well as the theme of the site.



Here we are visiting another users profile. This goes the same for viewing another user's post. Instead of seeing our own details, we will instead see the details of that user. On the home page and our own profile we see our own card displayed just like before.

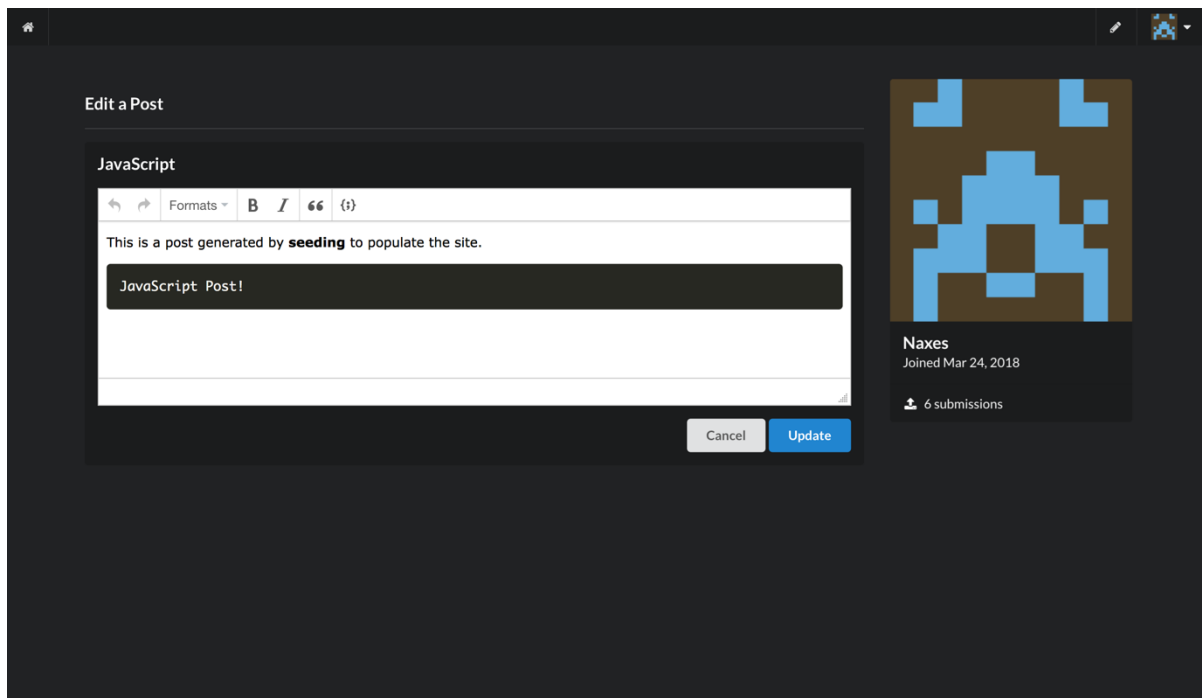


So, we are logged in as *Sean*, but viewing a post by another user, hence we'll show their card instead. This was the basis of getting profiles to work. Incidentally, the link to a profile is simply the users name, not the ID of the user in the user's table like how we view a post, where it is possible to have posts of duplicate names.

MARCH 24TH, 2018

Worked on editing posts today. I was set on making it in a modal window just like the delete functionality, but I found that the tinyMCE text editor doesn't function inside of a modal window without setting a particular setting of the modal window called *detachable* to false. This is rather hacky, so I instead decided to devote the functionality to their own views that work the same way as viewing a post with the exception that the posts body is inside the editor ready to go for any alterations.

I think doing it this way is for the best, as I feel as though there'd have to be some kind of performance issue with having all of the contents of every post we are looping through inside of uniquely identified modal windows. Fact is, even if the modal window isn't shown initially, that content still counts among the content being loaded when we access the home page as I'm not using ajax to dynamically query for the content for display. Anyway, the view in question is identical in design to the view for creating a post:

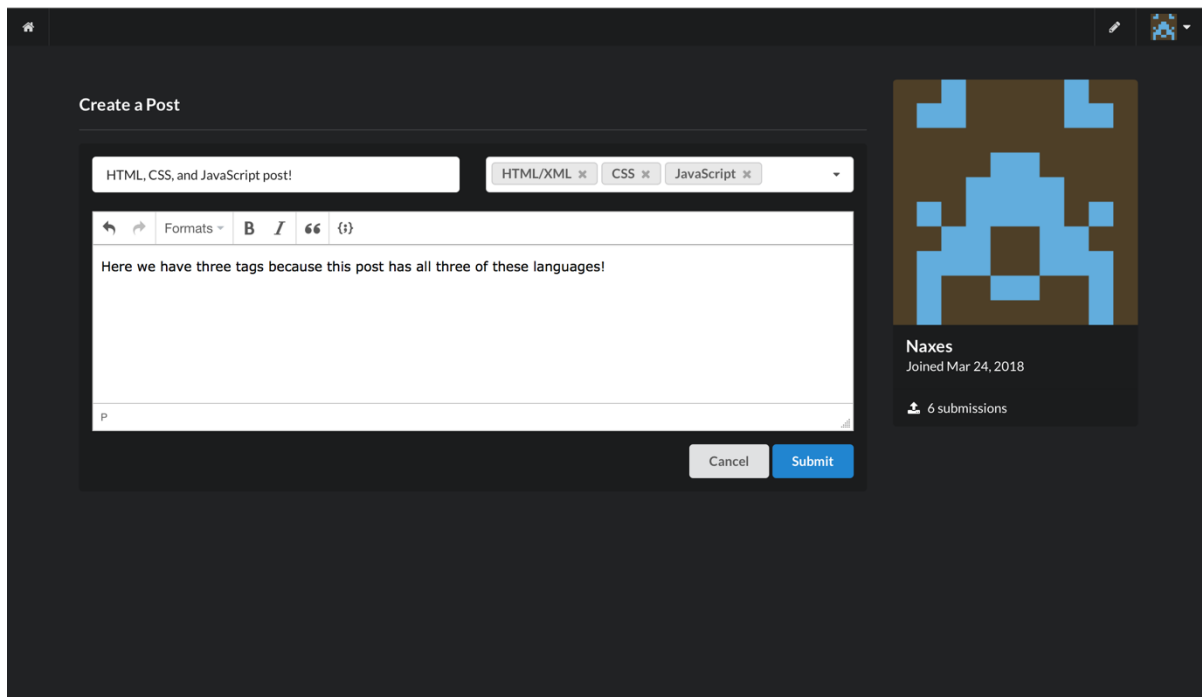


I've also made sure you cannot access an edit view for a post that is not your own, and even if you could, the backend validation ensures the query will fail if the user ID of the post doesn't match the authenticated user's ID. Additionally, I have it so that only the body of the post can be edited. Otherwise, a user could completely change the context of a post. So, say the post was initially Java related. In the future, they could change the title and body to be something completely different.

I also worked on tagging posts. This was a tricky one, but it turned out I was making it more complex than it actually was. I just had to apply the same logic I applied when creating the voting system. In essence, have a separate *tags* table with associations drawn between it and posts. The tricky part was using a single multiple selection input that would result in a separate row for each tag.

```
$tags = $request->input('tags');
foreach ($tags as $tag) {
    Tag::create(
        [
            'post_id' => $post->id,
            'name' => $tag
        ]
    );
}
```

This is how I managed it. Basically, we're getting the input values of the tags input field, then for however many there are we create a separate row in the tags table. For instance:



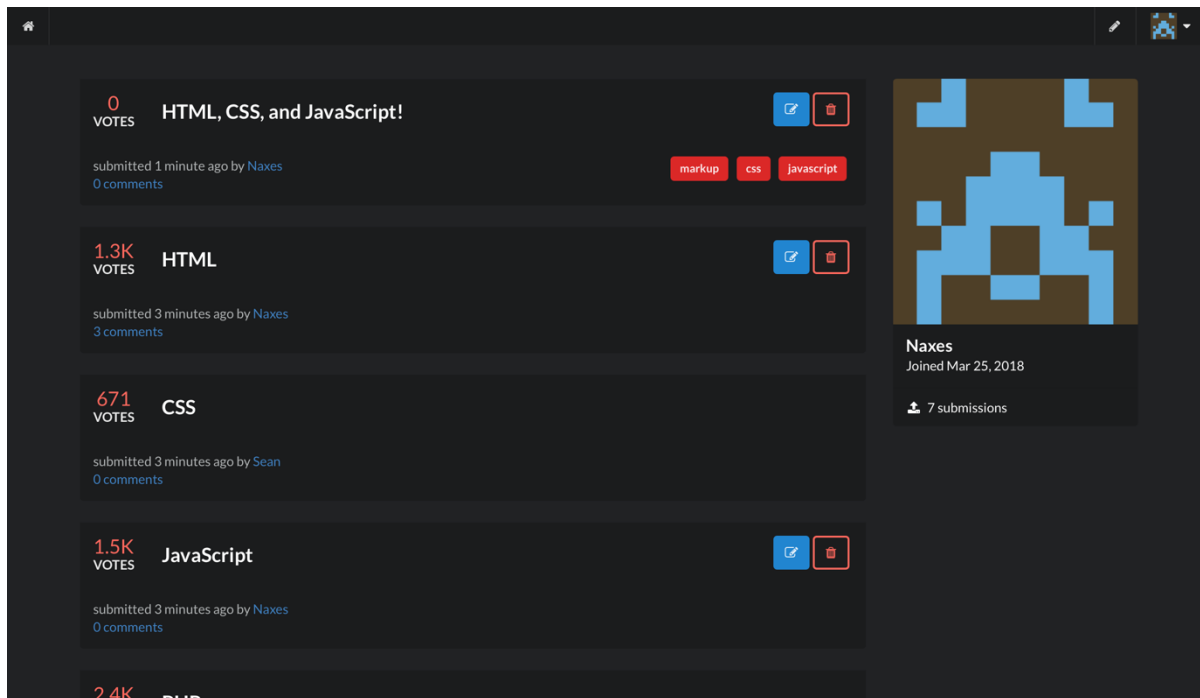
In the top right, we have tagged this post as having HTML, CSS, and JavaScript. The result is, these tags will be added as separate entries, but all collate to this one post through association of the post ID.

	id	post_id	name	created_at	updated_at
comments	1	13	markup	2018-03-24 23:08:46	2018-03-24 23:08:46
migrations	2	13	css	2018-03-24 23:08:46	2018-03-24 23:08:46
password_resets	3	13	javascript	2018-03-24 23:08:46	2018-03-24 23:08:46
posts					
tags					
users					
votes					

Here are the three tags segregated, but we can see that all of them relate to the post with the ID of thirteen. Now, I'll be able to loop through these in a view to display what users have tagged posts as.

MARCH 25TH, 2018

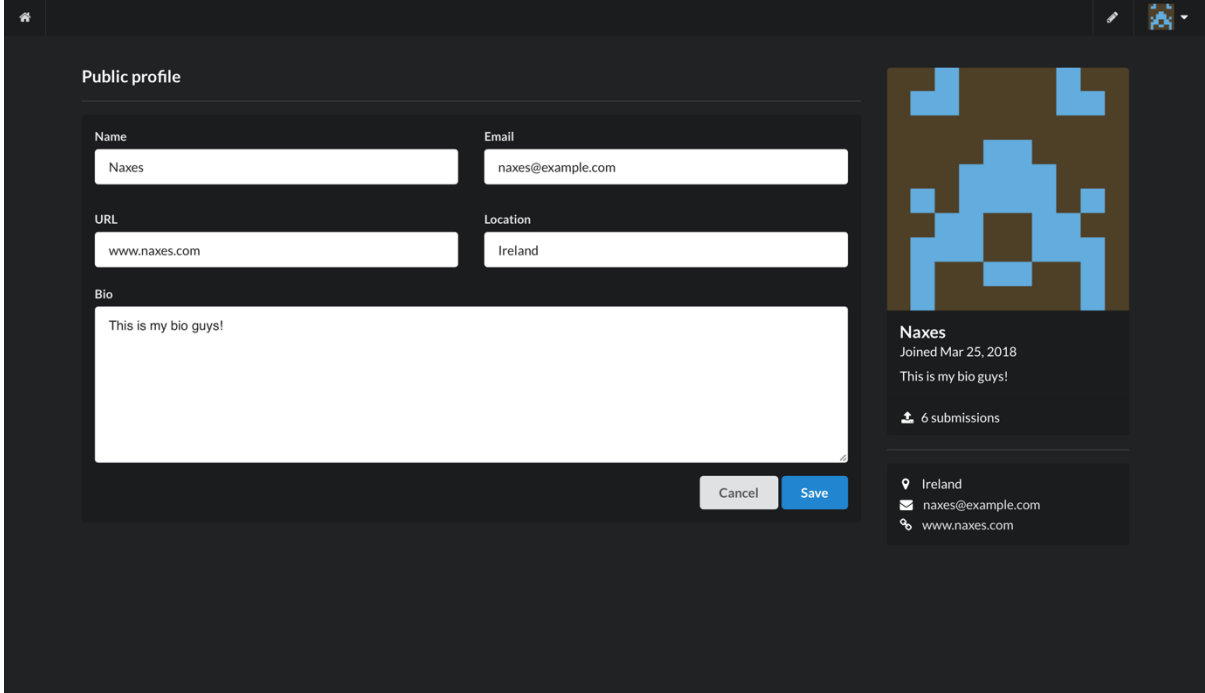
Following on from yesterday, I have the tags displaying next to posts.



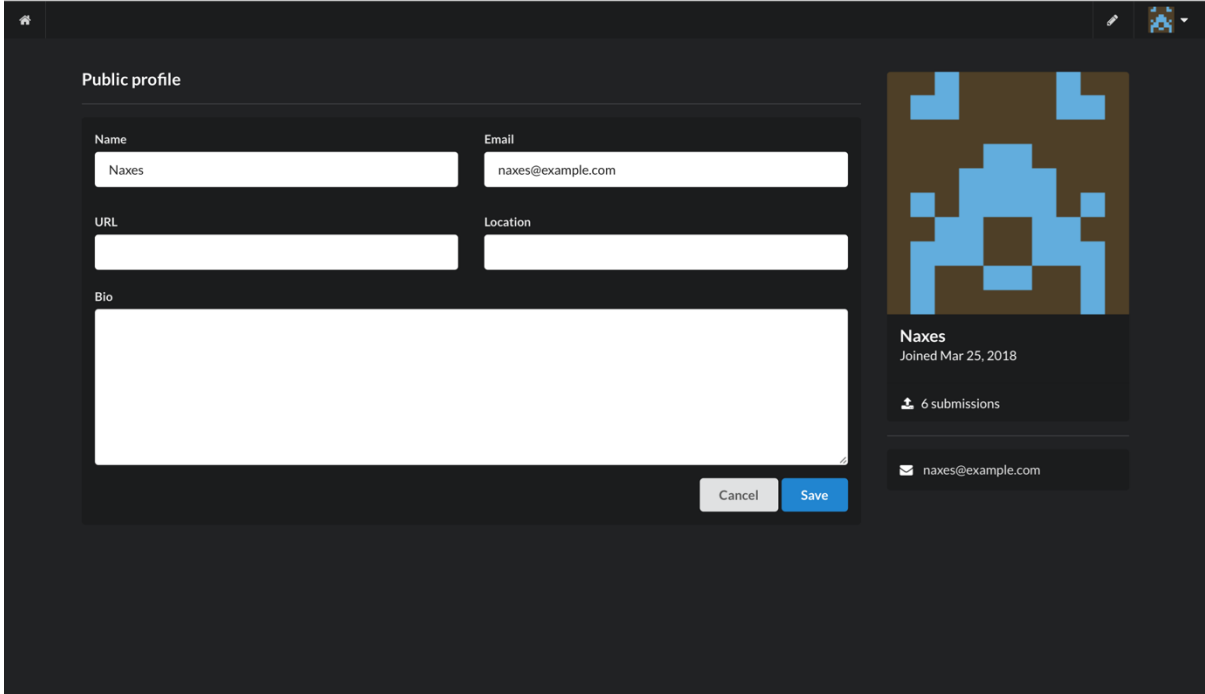
These tags only display on the home page for the time being. There is not enough space in the columns on the profile page, and in viewing a post I'd be constricting the posts body to 50% of the original space allocated. So, if I do display them on these pages it will be in a different fashion. Same goes for mobile view. In that case, I don't show the tags next to posts at all. Now, when I go to add the search functionality, I can possibly make these tags links that filter content based on posts tagged with, for instance, mark-up. Additionally, tagging posts is now a required input when creating a post. The other posts are generated by seeding, so that's why they don't currently have any tags.

In the past few days I've managed to get the profile pages, editing content, and tagging content facets of the project done. Most of the important things left relate to sorting content using URI parameters. This includes searching. I've done this before, but not in Laravel. It could be tricky. For example, I need to have both pagination parameters conjunctively with a content sorting parameter. By default, sorting the content will replace what page the user is on and vice versa.

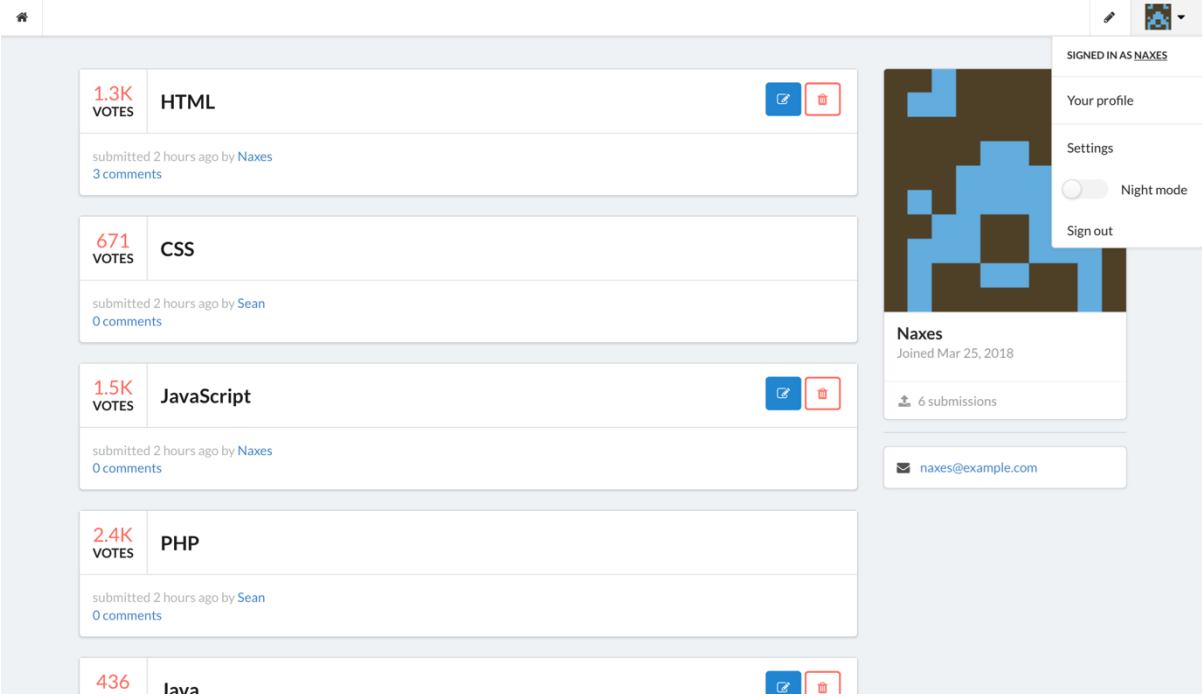
I made the view for users to change their details. This includes some that are allowed to be null-able, so I added some extra columns to the table for their bio, location, and a URL. It was tricky to figure out because I allow users to change their name and email but having the stipulation that these must be unique (i.e. not match other users details), meant that if I wanted to, for example, update my location but leave my name and email the same, the query would fail. This is because it is checking the uniqueness of my name and email against themselves and saying that they already exist. Luckily, I found a solution to this.



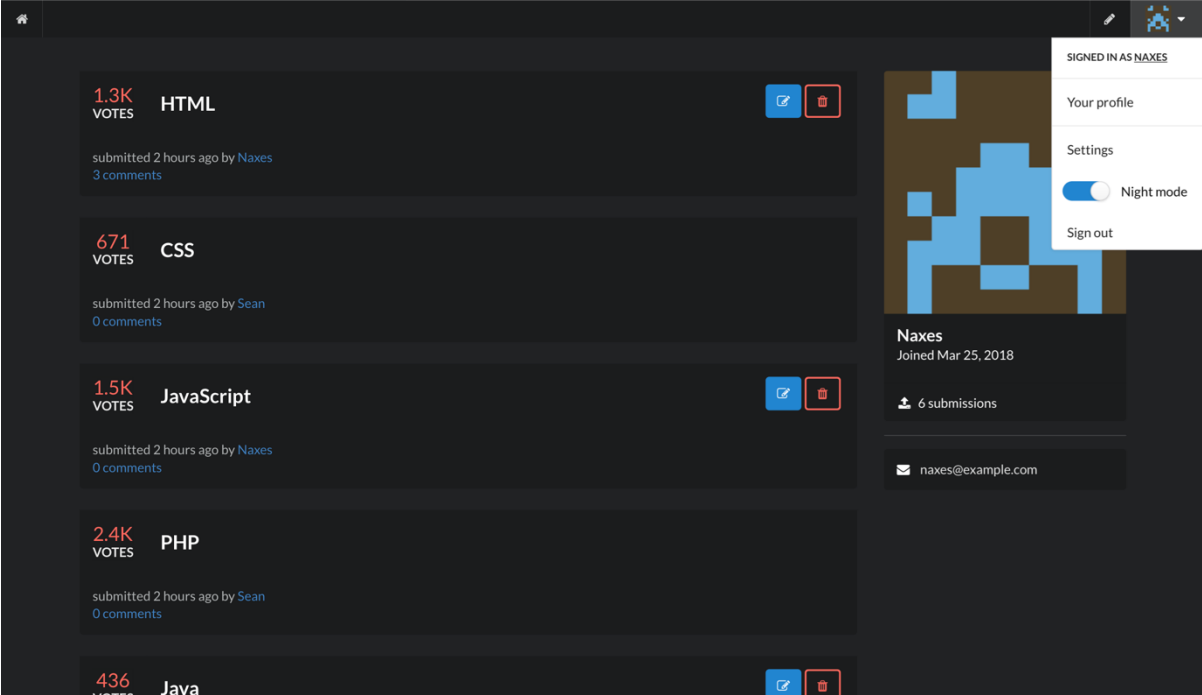
It's a simple view, simple form. The good with this is that it matches my other views for creating or editing a resource. If I remove one of the above that are allowed to be null-able (i.e. have no value), this will be reflected and removed from the list of details on the right-hand side under the original user card. The exception is the email, as this will always be present since it cannot have no value. So, naturally, if I remove these it looks like so:



I think I mentioned the option to swap to *night mode* would be in the settings view, but I actually figured out how to put it into the dropdown menu in the navigation bar after all. The trouble was, I was nesting the input in an element with the *item* class like other children of semantics list module. Not including this class made it work flawlessly. Items are selectable and close the menu, which was the issue. Also, for some reason, this appended the checkbox input to replace the parent element that opens the dropdown menu in the first place. Anyway, it now functions like so:



When we toggle the checkbox, the state of the toggle is maintained even on reload or load of a different view.



MARCH 27TH, 2018

Not much change at the moment. I started implementing the functionality to sort content, but I don't have any of the queries set up yet. Maybe later in the day. What I have done, is made sure that the URI variables won't overwrite each other, at least when changing pages. So, if I've sorted the content by most popular and proceed to page two, the page variable won't replace my sort variable.

On another note, college is coming to a close soon and there's still a good chunk of work to be done unrelated to this project. In particular, Irina's Java security project. I haven't even started it. It's quite worrying. When I'm back for my final two weeks after this week I'll need to get cracking on that. In the meantime, after I've sorted out the content sorting functionality I've opted to focus on the penetration testing CA. It seems like it's very similar to the first, so I'm hoping the same processes will apply. If I could have that done this week that would be a huge help as I could focus purely on the Java CA.

It's become a little unwieldy the past few weeks in the cyber security stream. We've been shafted in terms of organisation of CA's and classes. For one, we swapped penetration testing lecturers not even half way through, had our secure programming CA test delayed until the week I go back (on top of all the other work that has to be done), had changes to the grading rubric of the CA from the same class, and had the time table changed thrice this semester due to some of the aforementioned. Things will be looking up if I at least manage to get the penetration testing CA done. The software project can, fortunately, afford to sit at the wayside for the time being because it's nearly there anyway, and I'll have more time to finish it even after my exams at the end of April.

MARCH 31ST, 2018

No updates on the software project. I am currently working my way through making a basic boilerplate android application for my Advanced Secure Programming module. Since it is based in Java, we have the okay to make one. *Should* cover the criteria of security it needs to. Basically, we have to make something of any context that has at least four of the following:

1. Authentication and password management.
2. Cryptographic practices.
3. Input validation.
4. Access control.
5. Communication security.

I thought about this for a while, given the time I have left. In the end, I decided that making a boilerplate app with authentication pretty much covered most of the above points. Authentication is authentication, so that's one. Naturally, we hash passwords, so that's a cryptographic practice. We validate the input through established stipulations of the authentication forms. For instance, we want an email to be of type email and the password to have minimum six characters. Technically, access control *could* be the authentication itself blocking users from accessing the view(s) beyond that procedure, but I took it a step further.

I made it so that a user registers with an email and password, then they are brought to a view where they have to give a name. At this point, the email and password have already been stored. If they exit the app, these are the same details that are needed to login. However, we *need* the name, as it's something we could assign to, for instance, posted content. Before securing the routes, or views, this meant a user could login without giving a name which we require. Basically, even if they exit before conforming to this process, I have it gated so that the user will be redirected to the view prompting them for a name each time. It's a cheap way to communicate access control. I'm hoping something this simple is okay. There isn't really enough time to spend making a fully-fledged android application at the same level of completeness as my software project. That's all for now though.

APRIL 2018

APRIL 2ND, 2018

Just finished up my CA for advanced secure programming. Still some tweaks that can be made, but nothing that would take too long. I only have two more official college CA's. Both are reports. I have to perform penetration testing on a web application called *Bangers and Mash*. The VM is provided for us. I haven't begun yet, but I don't see *much* difference in this than the previous CA. The last is for digital forensics. Essentially, we have to use a hex editor and try to extract hidden images from a USB image.

I'm not sure if I will finish up the project before or after my exams on the 23rd and 25th, respectively. I am *not* looking forward to them, *obviously*. Studying is tough. It comes down to hoping the topics you researched come up. It's luck, really. Either you don't try, or you do try. If the latter, it will either work out, or not. It's impossible to cover everything. My advice, and what has worked for me every year consecutively for the most part, is to *literally* learn off the answers to questions from the past papers from three years ago *max*. If there *aren't* papers that go back that far, you're in the lucky position of some kind of course re-structure (sarcasm). That was the case with our AI exam in the first semester, as it used to be a third-year module, which explains the increased difficulty. Despite this, I have netted good results with this method. Additionally, if there is only *one* paper to go on, chances are your paper will not deviate *too much* from its contents. Chances are I will be graduate with a 2:1. It's *possible* I could get a 1:1, but the former is the safest bet, and it will be *very* close. My averages would place me just on the cusp of a 1:1 only if I manage to do well.

That's all for the moment. I may have a project-related update when the aforementioned CA's are out of the way. Luckily, I believe Irina's was the most cumbersome, at least comparatively to penetration testing and digital forensics, so it's good to have out of the way.

APRIL 6TH, 2018

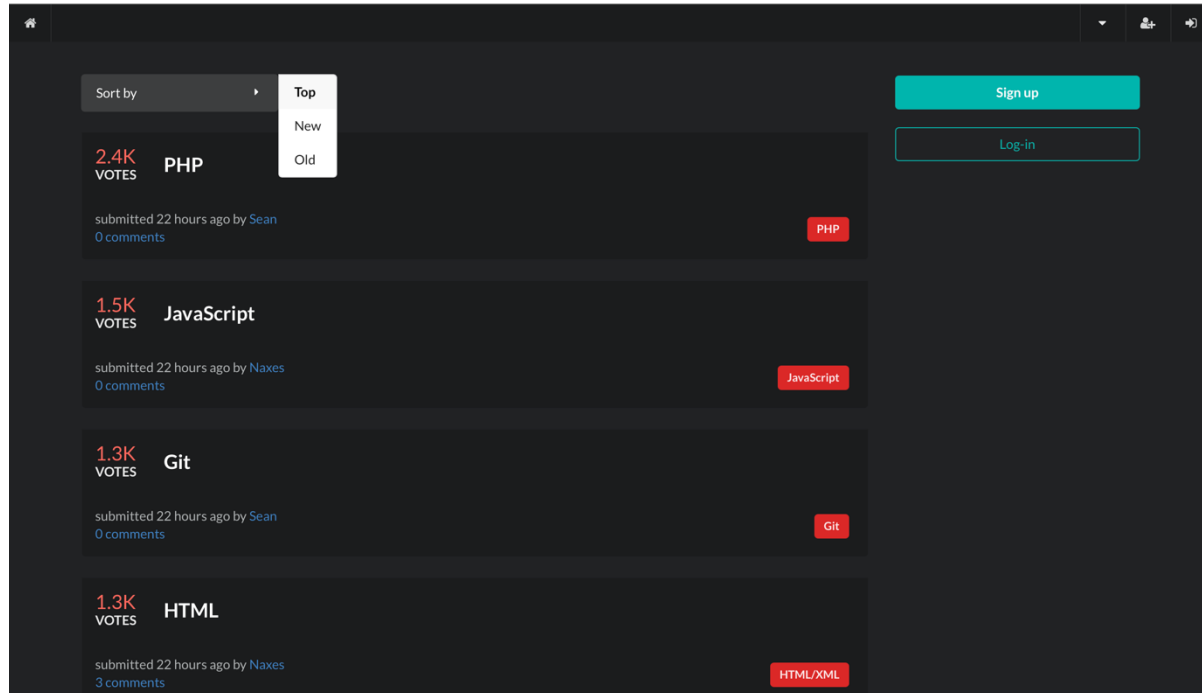
My advanced secure programming CA (test) was pretty abysmal. A lack of interest in C doesn't help. It's strange, because I don't require any of that knowledge relating to vulnerabilities in C to do the project CA for this class. The android application ticks most of the boxes despite having no context, and I understand the vulnerabilities I've covered pretty well. It's the same as for my software project. I'm capped out at almost 5,000 words with the accompanying documentation. I'm still worried for the result of the project, because last semesters project I thought was pretty well done, documentation included. The result wasn't bad, but I thought it would do better. The android application is naught compared to that project, so I'm not sure.

Yesterday, on the bus journey home from the terrible CA test, I decided to hop back into the project a bit to take my mind off of things. I set up sorting content relatively quickly. I'll probably change how it *looks*, but functionally, it's all there. From the last time I worked on it, I had set up that swapping pages *wouldn't* interfere with any other URI variables that get thrown into the mix, but not vice versa, which is as intended. In other words, if I sort the content based on the highest score, swapping pages will maintain this filter. However, if I change the filter, the page variable *will* be reset back to the first page. This isn't an issue just to note, that's how it should be. You wouldn't want to sort content and remain on, say, page five. That wouldn't make any sense.

Initially, I thought that it wasn't working because, in addition to creating a filter to sort the content by the most popular, I threw in ones for oldest to newest and vice versa, just to see. Nothing changed on those, but then I remembered that I am *seeding* the database with posts that are all created at the same time, hence there would be no change. So, I created a new post manually, and all was well. I'll include screenshots when I settle on a look. For the time being, it is just a dropdown selector.

APRIL 7TH, 2018

I just realised that this seems to be the last day where a reflective journal upload is required. I'll update the document regardless to keep track of things, but for the purposes of the upload I thought I'd put in screenshots for some of the updates I've made.



This is the content sorting I discussed yesterday. It works through the implementation of a *sort* request input variable. Based on the value of this variable, the content is sorted accordingly. I did this in the index function of my posts controller like so:

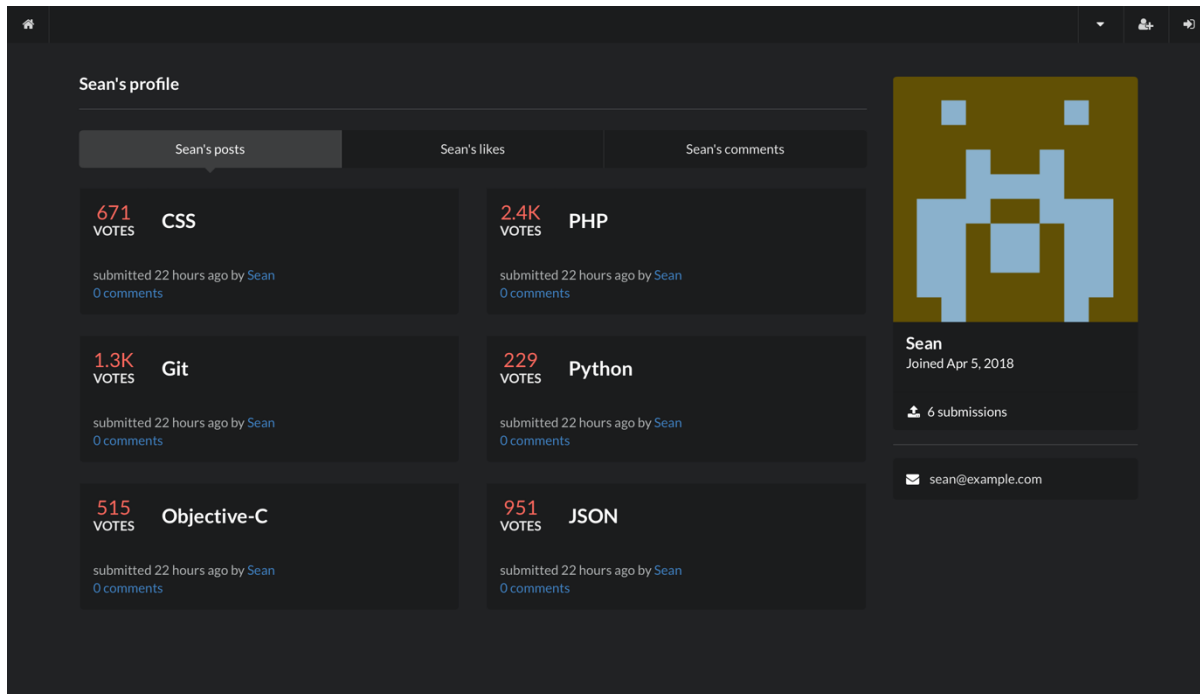
```
public function index(Request $request)
{
    $user = User::where('id', auth()->id()->first());
    $posts = Post::latest()->paginate(5);

    $sort = $request->input('sort');
    switch ($sort) {
        case 'top':
            $posts = Post::orderBy('score', 'desc')->paginate(5);
            break;
        case 'new':
            $posts = Post::latest()->paginate(5);
            break;
        case 'old':
            $posts = Post::orderBy('created_at', 'asc')->paginate(5);
            break;
        default:
    }

    return view('posts.index', compact('user', 'posts', 'tags'));
}
```

Essentially, we have switch statement checking for cases where it equals *top*, *new*, and *old*, respectively. The filters are subject to change. Additionally, *new* has been the default since I began, but I may change this as well.

Secondly, there are a few additions to user profiles.



Because I figured out the inclusion of pagination and other URI variables conjunctively, I thought it would be a neat implementation to update the query present on a user's profile based on the tab selected. This way, we can see the user's posts, what posts they've *liked*, and their comments in a given post. I'm not sure about the latter, but I thought I'd give it a go. These currently don't do anything yet, but it's the exact same logic as present in sorting content on the homepage. A URI variable, a switch statement, and a query within each case of the switch.

After this, the last things I have to do are searching for content and comment replies. I thought about the admin panel, which was labelled as *access control* in some of my earlier documentation, but I'm not sure about it. All I can say is that I will do it if I have time to spare, but it's not a priority. I understand it's inclusion to showcase higher level accounts with additional privileges, but it's no longer a case of showing access control as it was initially presented, because I have already implemented such things elsewhere. For example, guests cannot access views relating to posting content. Subsequently, authenticated users cannot access the views for registration and login. It's not *just* about denying access however, we are also anchoring important or dangerous actions, such as editing and deleting a post, to only those that belong to the current user.

That is all from me in this upload. Despite what I have shown, I am still not fully engaged in working on the project again until I finish up my final two assignments for penetration testing and digital forensics. Until then!

MAY 2018

MAY 2ND, 2018

Classes and exams are over, so I'm back working on the project full time. Between now and the 13th of May, I have to submit a poster, my final report, and the project source code. It's not so bad considering the amount that is done, it is simply a matter of whether or not it is adequate. Also, I met my project supervisor Glen for the first time since the midpoint, so I know the brunt of what needs doing for my final documentation and presentation. I will probably redo the document from scratch, even though what has already been marked will not be remarked. That being said, a lot has changed since its inception, and so I feel a complete rework is the only thing that will ensure I don't accidentally miss something pertaining to the older implementation. Some elements of it are okay. For example, Glen told me that my system architecture diagram was perfect, just a few minor tweaks required. I missed out on 15% the last time by not including all of my use cases, so I must make sure I have those as well. I had an idea that, for testing, I *could* do a self-penetration test on my project, but I'll have to see if something like that is in the scope with the time that remains. Otherwise, I am planning to have a focused test group consisting of some family, friends, and peers, presuming I have the site hosted in time to do so remotely. If not, I will physically present them with a build of the project and query individuals on site.

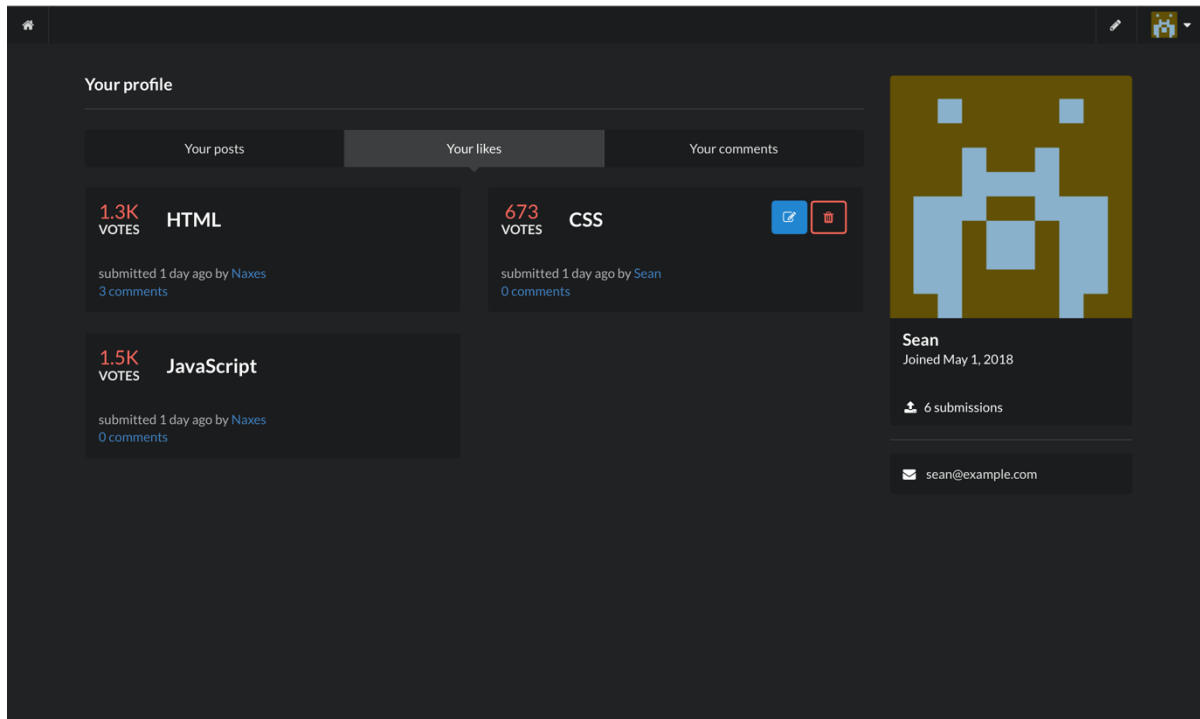
Onto more tangible things, I am currently working on the profile tabs that I outlined in my last post from April. It's just swapping out what query is enforced based on the value of a request input variable called 'sort'. For showing what I user has liked, I needed to employ a different technique I have not needed until now. Since I have associations drawn between the tables, I was trying to think of how I could get all posts that a given user has liked, remembering that the posts and votes tables are separate tables. Luckily, from scouring the Laravel documentation, I found just how to do this.

```
public function show(Request $request, User $user)
{
    $posts = Post::where('user_id', $user->id)->paginate(6);

    $sort = $request->input('sort');
    switch ($sort) {
        case 'likes':
            $posts = Post::whereHas('votes', function ($query) use ($user) {
                $query->where('user_id', $user->id);
            }->paginate(6);
            break;
        default;
    }

    return view('profiles.show', compact('user', 'posts'));
}
```

Select the posts that have votes, but where those votes have the user ID associated with the user profile we are currently viewing. The latter element of the function pertaining to including the user variable is vital. Even though the user variable is in use outside of this query, it is out of scope of the function unless we explicitly *use* it.



This is the result of the tab. There's an issue with this though. If a user is on their own profile and chooses to revoke a previous vote, it results in the AJAX request updating the votes data of the post in question to be blank. Presumably, this would be because it is trying to retrieve the resulting votes in real-time via a query in which we are only interested in things the user *has* voted on. Easy solution is making a new partial that looks the same, but the like button triggers some different functionality. Not sure though, I will have to investigate further.

```
/**
 * Navigation dropdowns (i.e. links | sorting)
 */
$('.nav-dropdown')
  .dropdown({
    action: 'select'
  })
;

/**
 * Tag selection dropdown
 */
$('#tags')
  .dropdown()
;
```

The only other change I've made today was with dropdowns. Previously, they were all universally tied to the same piece of JavaScript in order to function. There was, what I thought, an abnormality with selecting an item from the dropdown list whereby the parent element's value would be replaced with that of the selected item. It is actually an intentional default of the dropdowns in the Semantic CSS framework, so I segregated the dropdowns where I did *not* want this to happen. Cases where I do want it to happen are with dropdowns such as selecting tags for posts, where the input is populated with the selection from the list. The 'select' action ensures that the aforementioned won't occur.

MAY 6TH, 2018

I am currently midway through my final report for the software project. I won't lie, it has been incredibly more time-consuming than I had initially thought. When I met with Glen, one important thing with regards to the report was to not refer to myself as "I did this", but rather "This was done", for example. In that respect, my previous technical report was useless, because it was absolutely littered with the former. It was only skim-read, but the contents themselves were not salvageable even if paraphrased to the correct format.

On the bright side, I think the new document is a vast improvement. I was aiming to have it done today because I really need to focus on finishing the project code, hosting it, and doing the poster. I underestimated just how little time is left, but it's still doable. Glen also advised me to send my report with at least two days advance, so that's another reason to have it done. There are still a couple of blockades to do with testing. I can't complete the document without testing and customer testing. For the former, it means I have to create test cases with PHPUnit, which comes pre-packaged with Laravel. It's just irritable, because it means I have to go develop them then come back to the document. I'm going to keep them relatively simple and not go too overboard. Doing a self-penetration test is totally out of the scope at this point.

As of writing, I'm thinking of hosting via AWS. It seems the most modern and reliable way to do so. The reason I say that, is because I tried via Blacknight, and it was the most convoluted setup I've ever seen despite seemingly great reviews of them. For the project showcase, I want to have it hosted and bring along some vista print faux business cards, since I heard one 4th year did that last year and it was a hit. I plan to have them matching my poster. That's all for the moment. Hopefully next entry I'm talking about code again.

MAY 13TH, 2018

Today is the day the project is due. I did not have time for any update entries, but since this is an inclusion of the final technical report, I thought I'd briefly discuss the things I managed to do in the last couple of days. I did host the project via AWS. It's still not what I would call a comprehensive process, but a big improvement over Blacknight. I've included the domain in a read me file within the project. The biggest reason for this was to illustrate my use of an SSL cert to enforce HTTPS, but also to provide ease-of-access to the project for whoever must go through the code, since it would require having Laravel set-up on their machine. Notably, I was able to migrate the projects tables over to AWS via a YAML command inclusive of seeded data.

I'm going to briefly discuss the project code as the report is finished bar including this last journal entry, and it's going to sound like rabble. I don't have it all in my head what I had still left to do since last time, but roughly I would say the search functionality and profile tabs. These are finished, but sort of cascaded into miniature sub-tasks that I ended up doing in addition. For example, if a user has no liked content or comments on their profile, is the page just blank? So I created some icons and custom messages for whenever there is a case whereby a query does not retrieve anything. The profile tabs also created another task, and that was that if a user revoked a like on something within the likes tab, AJAX would update it with no score and it looked bizarre. The reason for this is because the query is retrieving things the user *has* liked, so when the score is updated, that post is no longer part of the query. I had to create a new partial specifically for posts within that tab. I realised very late that I was allowing a lot of different things to go through form submissions that could make the sites content look absolutely atrocious, such as allowing special characters in usernames or post titles. This in itself snowballed into reinforcing every instance of validation in each controller as well as providing the JavaScript validation to match it. I had to use regular expressions for a lot of very particular instances. For example, in user settings, users can provide a URL. I had to create a regular expression that would exactly fit the specification of the input that could be classified as a valid URL. Anyway, I wanted to keep this one short but still illustrate the some of the process from the past week. Time to submit Reflux!

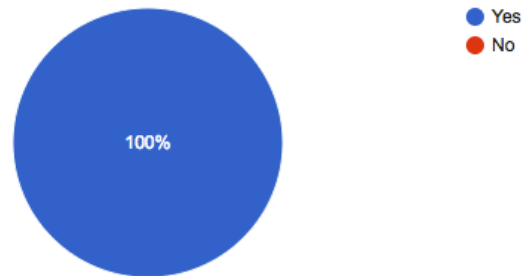
OTHER MATERIAL USED

REQUIREMENTS ELICITATION SURVEY

QUESTION 1

Would you use a service like this?

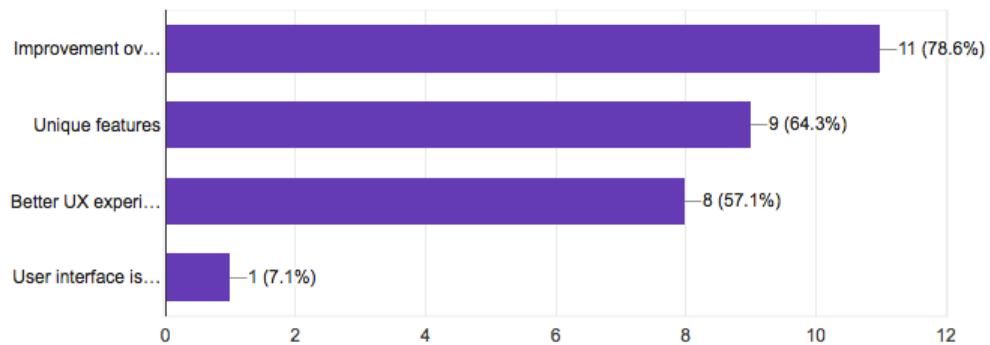
14 responses



QUESTION 2

What would make you use this system over a competitor?

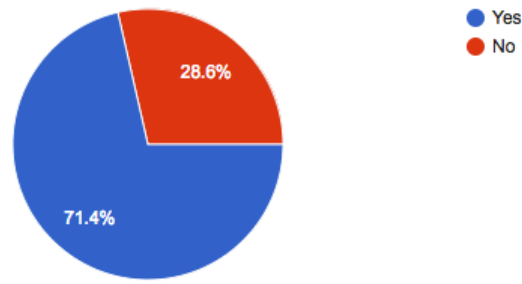
14 responses



QUESTION 3

Are social aspects important to you? (e.g. Commenting system, Rating system, Private messaging)

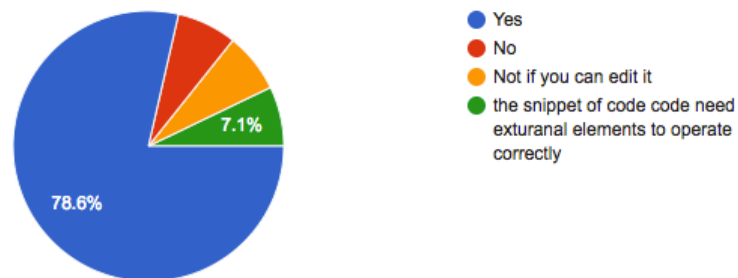
14 responses



QUESTION 4

Does code validation matter for a service like this? (i.e. Checking for errors in the code)

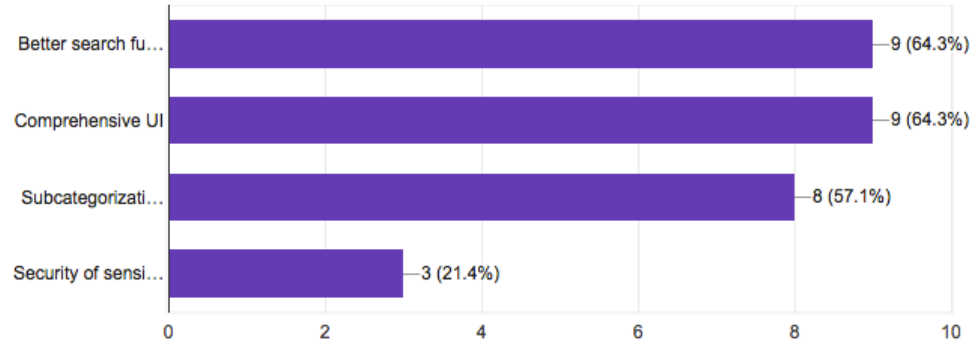
14 responses



QUESTION 5

What would be the biggest improvement over existing services?

14 responses



QUESTION 6

Any other thoughts or ideas for such a system? (Not compulsory)

2 responses

Nice project, if it's possible to integrate a local server to see server side stuff run - would be cool

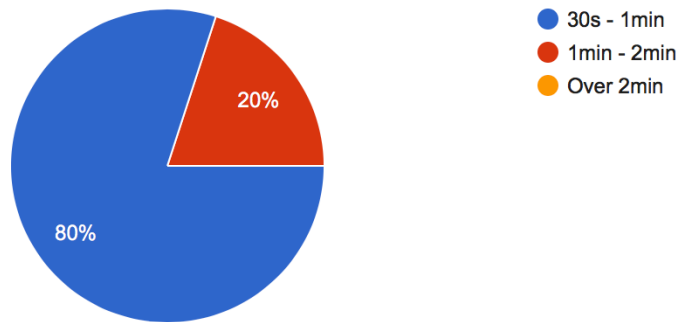
seems to be a "dolled up" version of stack overflow but i like the idea and think there is a lot of potential for something like this to take off, have you thought about using any SCM within this web app which could allow users to access the whole project to aid in debugging. i wish you all the best with this project and id love to see an update in future

CUSTOMER TESTING SURVEY

QUESTION 1

How long did it take to register an account?

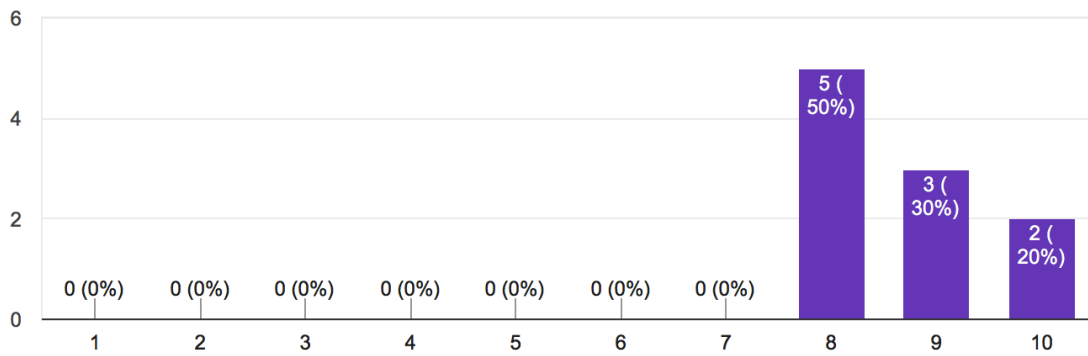
10 responses



QUESTION 2

On a scale of 1 to 10, how comprehensive was the user interface?

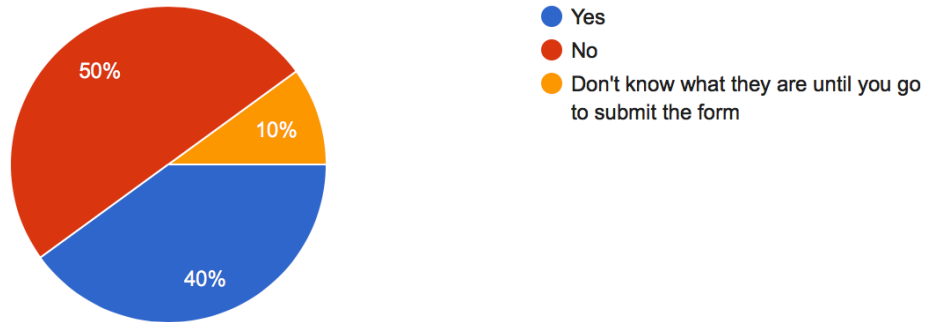
10 responses



QUESTION 3

Did enforced validation impede using the application? (e.g. max character count)

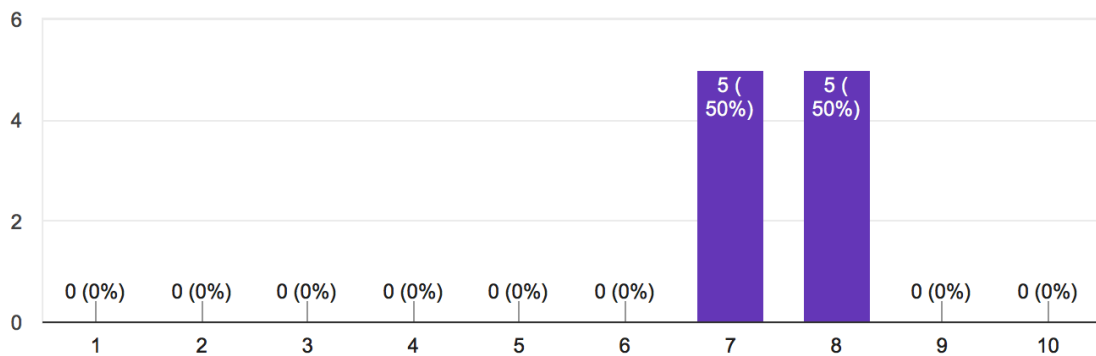
10 responses



QUESTION 4

On a scale of 1 to 10, how well does the system communicate these enforcements?

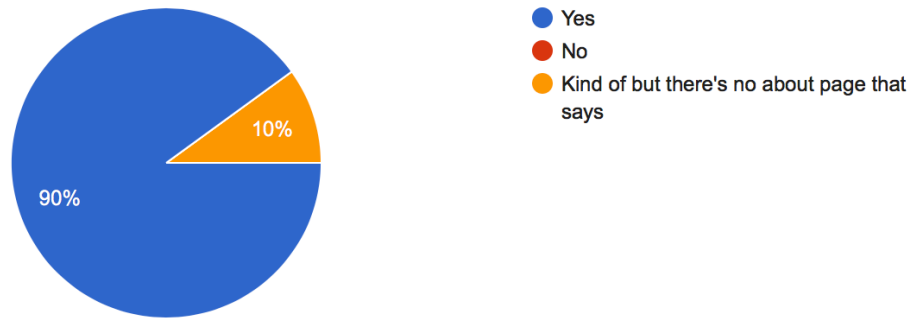
10 responses



QUESTION 5

Is it clear what the intent of the system is?

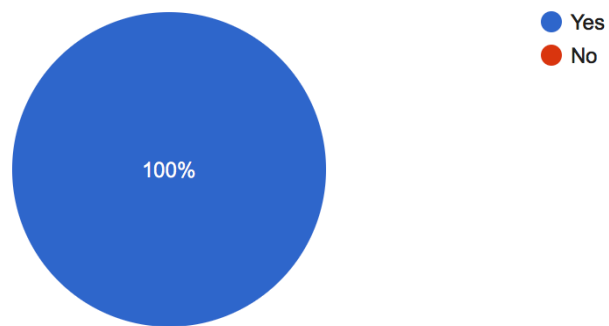
10 responses



QUESTION 6

Does the system invoke the sense that it is secure?

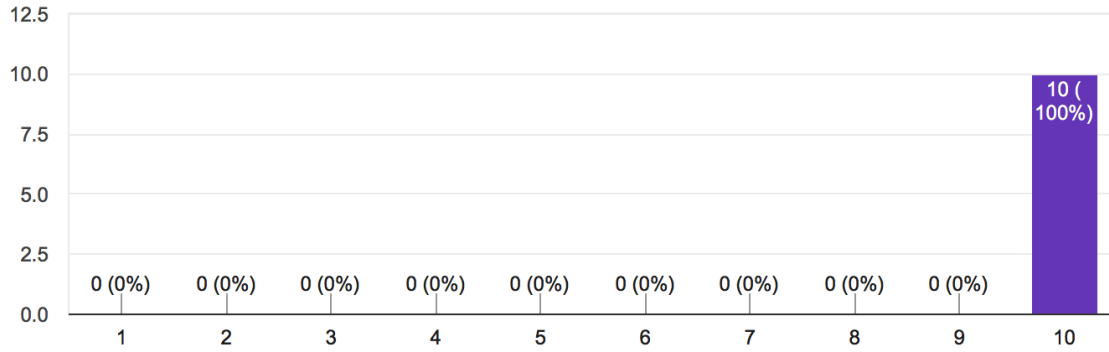
10 responses



QUESTION 7

On a scale of 1 to 10, do you agree or disagree that security is important, even if it is not necessary for the application to work?

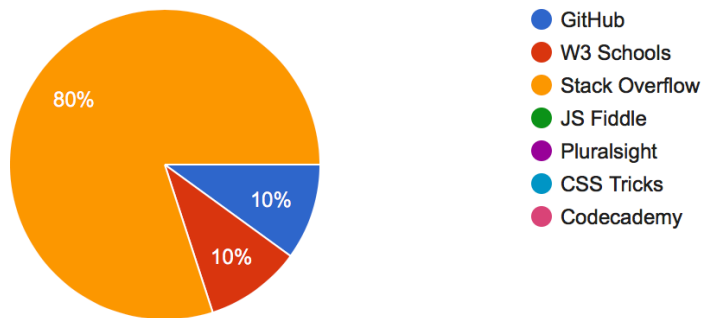
10 responses



QUESTION 8

Which of the following is the system most like:

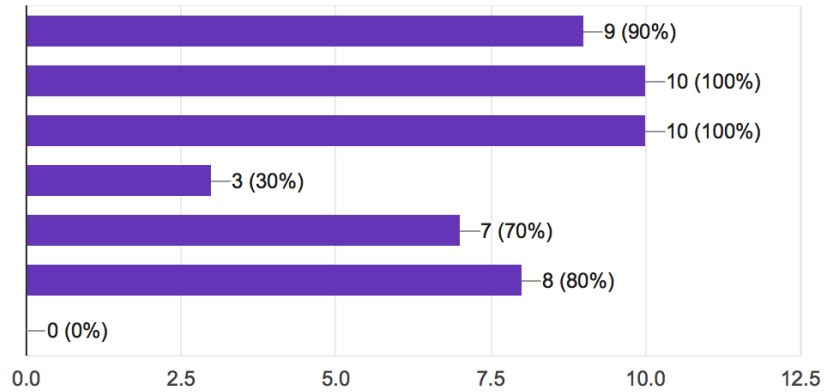
10 responses



QUESTION 9

What aspects of the system - if any - are improved comparatively to your selection above?

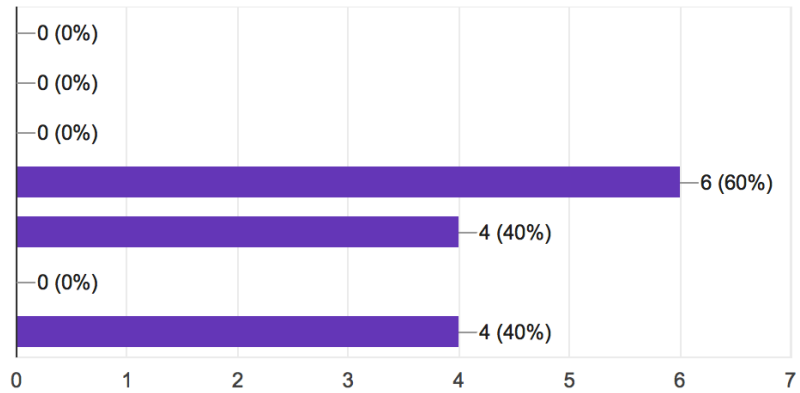
10 responses



QUESTION 10

What aspects of the system - if any - could be improved?

10 responses

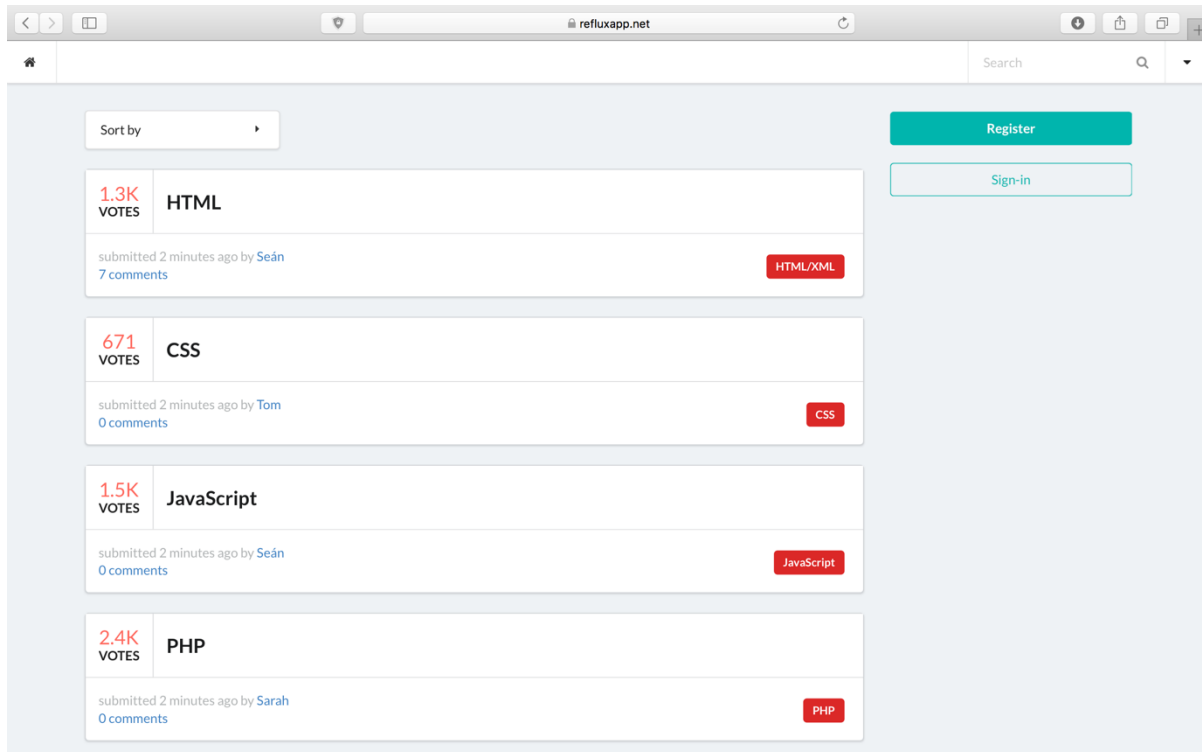


USER MANUAL

ACCESSING THE PROJECT

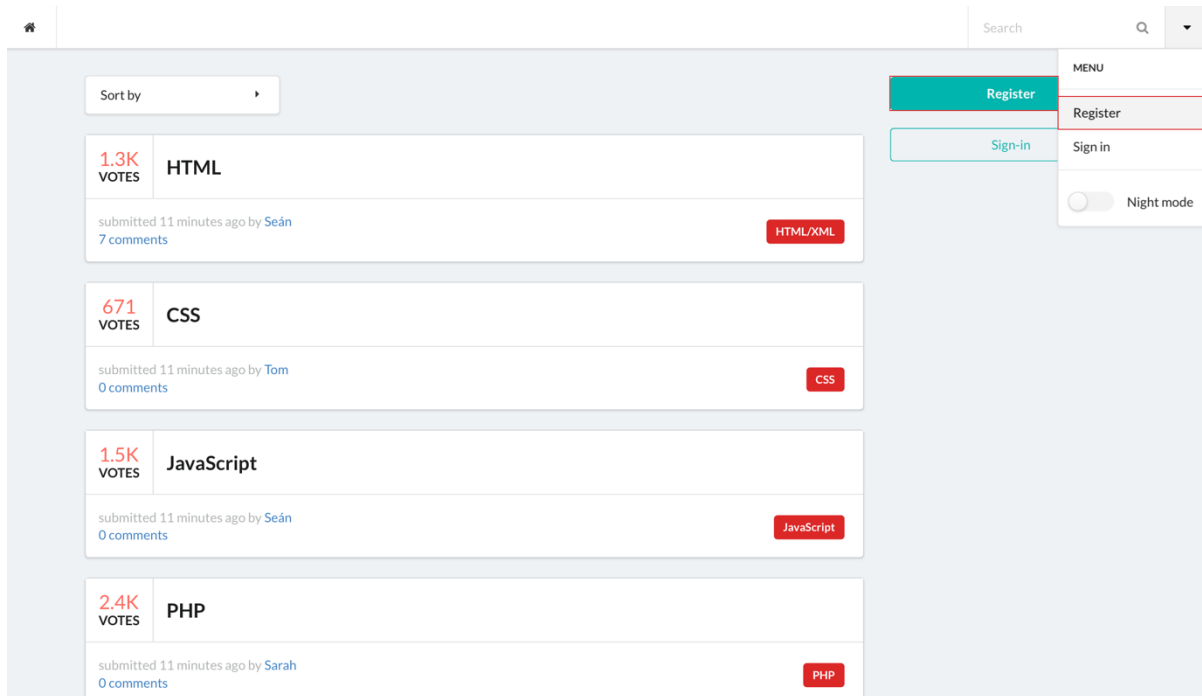
The project can be accessed locally if you have Laravel on your machine, otherwise it can be found at the following URL:

❖ URL: <https://refluxapp.net>



CREATING AN ACCOUNT

To begin the account creation process, navigate to the registration view via one of the following elements:



From here, enter in the details for a new account and register. Note the validation requirements in place as illustrated:

The registration form is titled "Register an account" and includes the following fields:

- Username: JohnDoe
- Email: johndoe@example.com
- Password: [Redacted]
- Confirm Password: [Redacted]

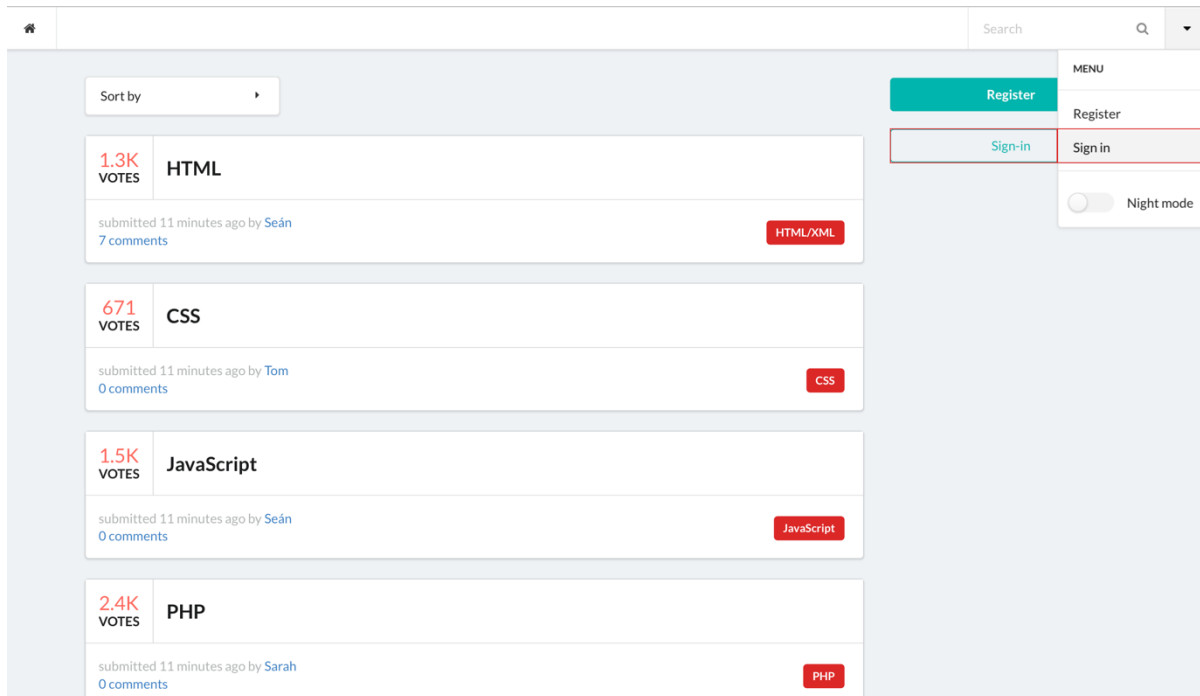
A red "Register" button is located below the password fields. Below the form is a link: "Have an account? [Log-in](#)".

Validation requirements are listed in a red box:

- Username must be **one word** and contain **no special characters** or **spaces**
- Please enter a valid e-mail
- Your password must be **at least 10 characters**, and include a **lowercase**, **uppercase**, **numeric**, and **special character**
- Passwords do not match

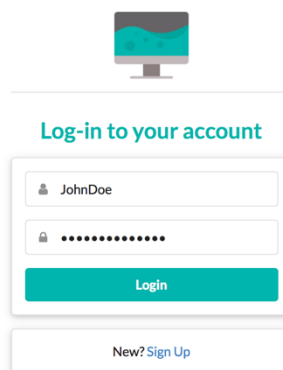
LOGGING IN TO AN ACCOUNT

To login to an account, navigate to the login view via one of the following elements:



The screenshot shows a web application interface. At the top, there is a search bar and a home icon. Below the search bar, there is a 'Sort by' dropdown menu. The main content area displays a list of programming topics, each with a vote count, the topic name, submission details, and a comment count. The topics are: HTML (1.3K votes, submitted 11 minutes ago by Seán, 7 comments), CSS (671 votes, submitted 11 minutes ago by Tom, 0 comments), JavaScript (1.5K votes, submitted 11 minutes ago by Seán, 0 comments), and PHP (2.4K votes, submitted 11 minutes ago by Sarah, 0 comments). On the right side, there is a user menu with options for 'Register' and 'Sign in', and a 'Night mode' toggle switch.

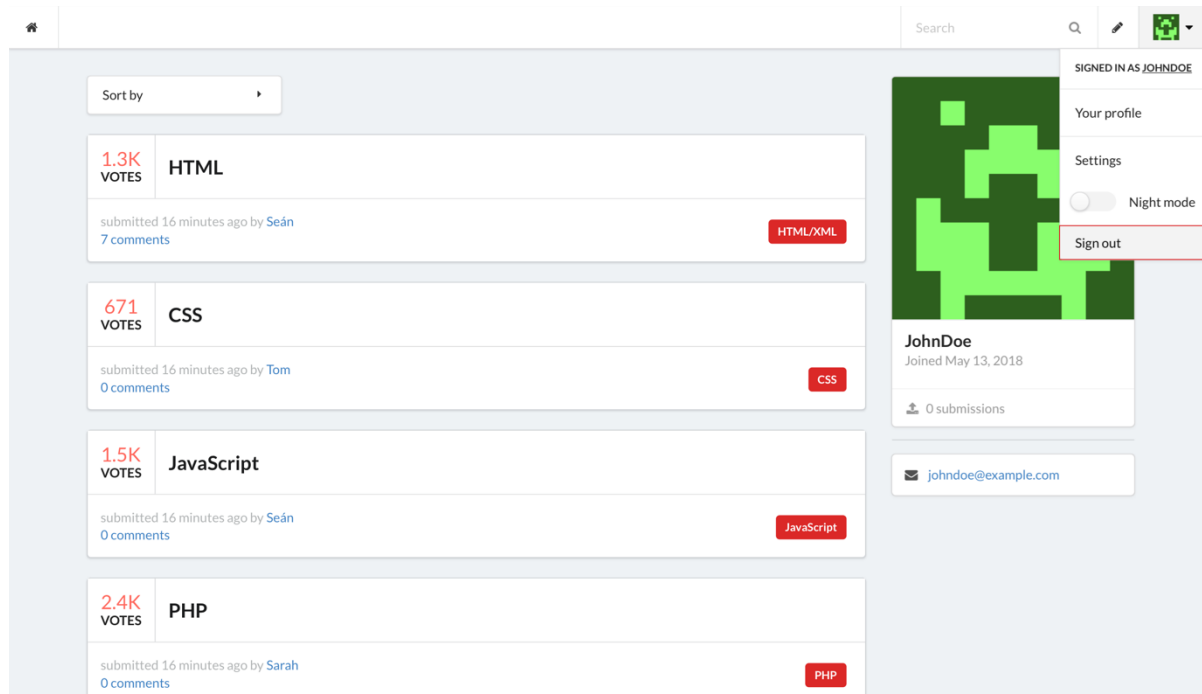
From here, enter the details of an existing account and login:



The screenshot shows a login form titled 'Log-in to your account'. It features a teal monitor icon at the top. Below the title, there are two input fields: the first for the username 'JohnDoe' and the second for the password, represented by a series of dots. A teal 'Login' button is positioned below the password field. At the bottom of the form, there is a link that says 'New? Sign Up'.

LOGGING OUT OF AN ACCOUNT

To logout of an account, access the option via the dropdown menu of the navigation bar:

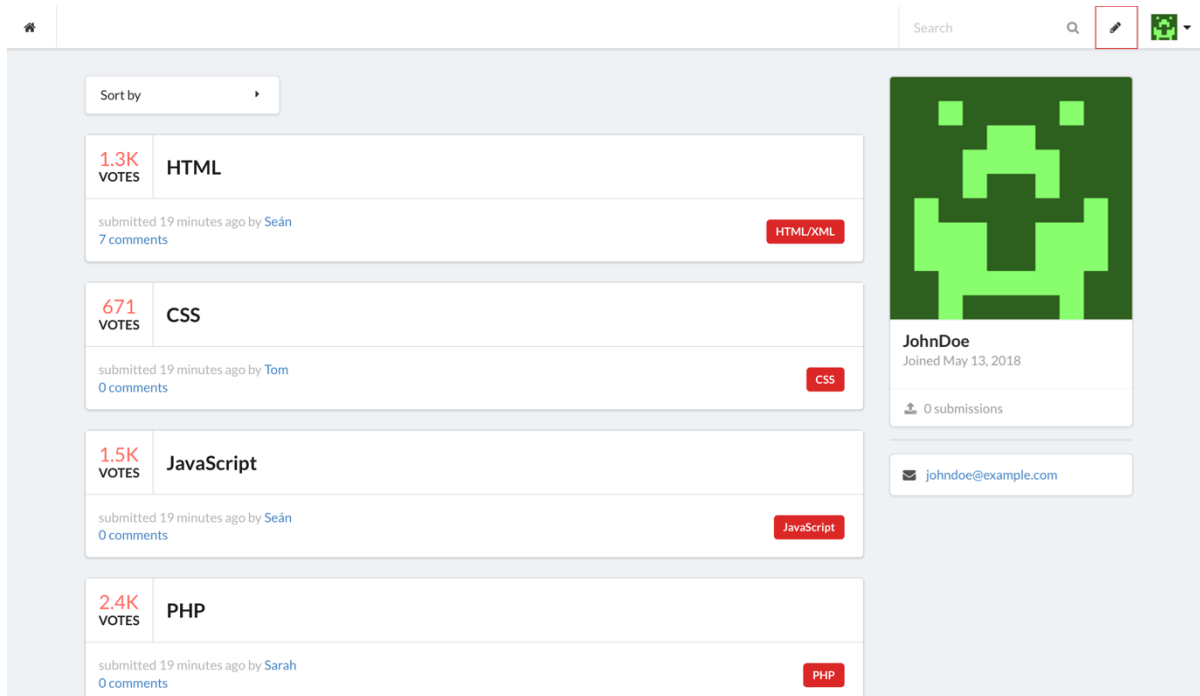


The screenshot displays a web application interface. At the top right, there is a search bar and a user profile icon. The user profile dropdown menu is open, showing the following options: "SIGNED IN AS JOHNDOE", "Your profile", "Settings", "Night mode" (with a toggle switch), and "Sign out" (highlighted with a red border). Below the navigation bar, there is a "Sort by" dropdown menu. The main content area features a list of programming topics, each with a vote count, a title, submission details, and a comment count. The topics are: HTML (1.3K votes, 7 comments), CSS (671 votes, 0 comments), JavaScript (1.5K votes, 0 comments), and PHP (2.4K votes, 0 comments). Each topic has a red button with the language name. The user profile information on the right includes the name "JohnDoe", the join date "Joined May 13, 2018", "0 submissions", and the email address "johndoe@example.com".

Topic	Votes	Comments	Submitted by	Language
HTML	1.3K	7	Seán	HTML/XML
CSS	671	0	Tom	CSS
JavaScript	1.5K	0	Seán	JavaScript
PHP	2.4K	0	Sarah	PHP

CREATING A POST

To create a post – while logged in – access the option via the navigation bar:



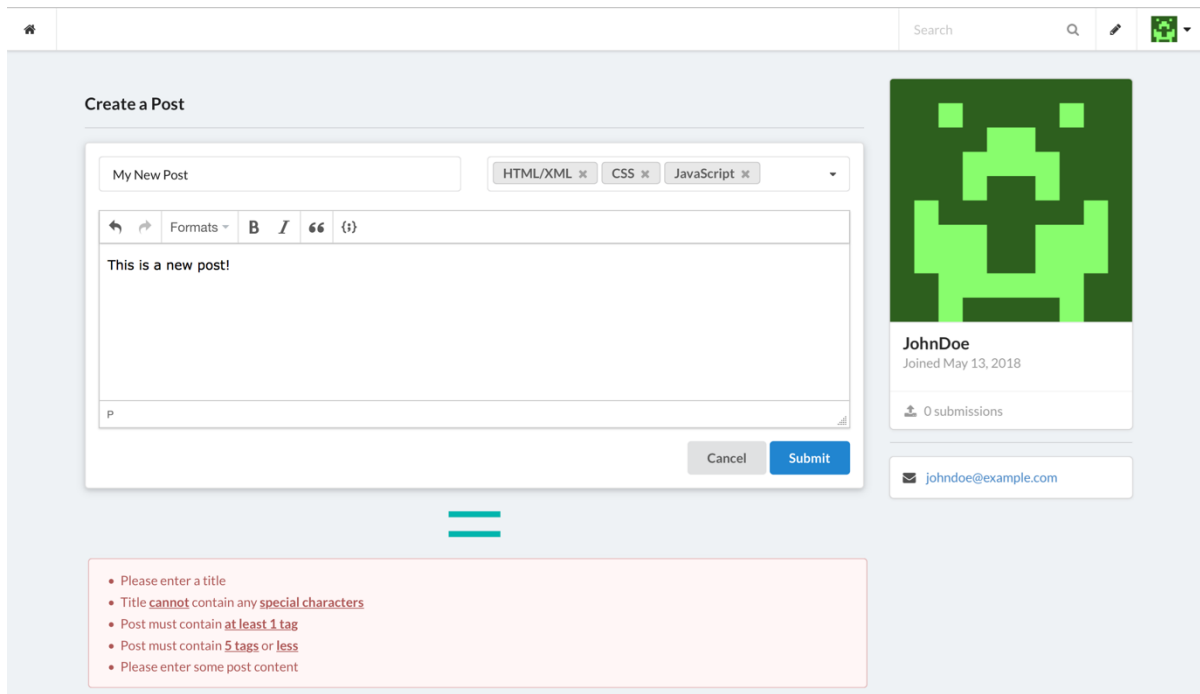
The screenshot shows a forum interface. At the top right, there is a navigation bar with a search icon, a pencil icon, and a user profile icon. Below the navigation bar, there is a 'Sort by' dropdown menu. The main content area displays a list of posts:

- 1.3K VOTES** **HTML**
submitted 19 minutes ago by [Seán](#)
7 comments HTML/XML
- 671 VOTES** **CSS**
submitted 19 minutes ago by [Tom](#)
0 comments CSS
- 1.5K VOTES** **JavaScript**
submitted 19 minutes ago by [Seán](#)
0 comments JavaScript
- 2.4K VOTES** **PHP**
submitted 19 minutes ago by [Sarah](#)
0 comments PHP

On the right side, there is a user profile for **JohnDoe**, who joined on May 13, 2018. The profile shows 0 submissions and an email address: johndoe@example.com.

TEXT POST

From here, enter the details for a new post and submit. Note the validation requirements in place as illustrated:

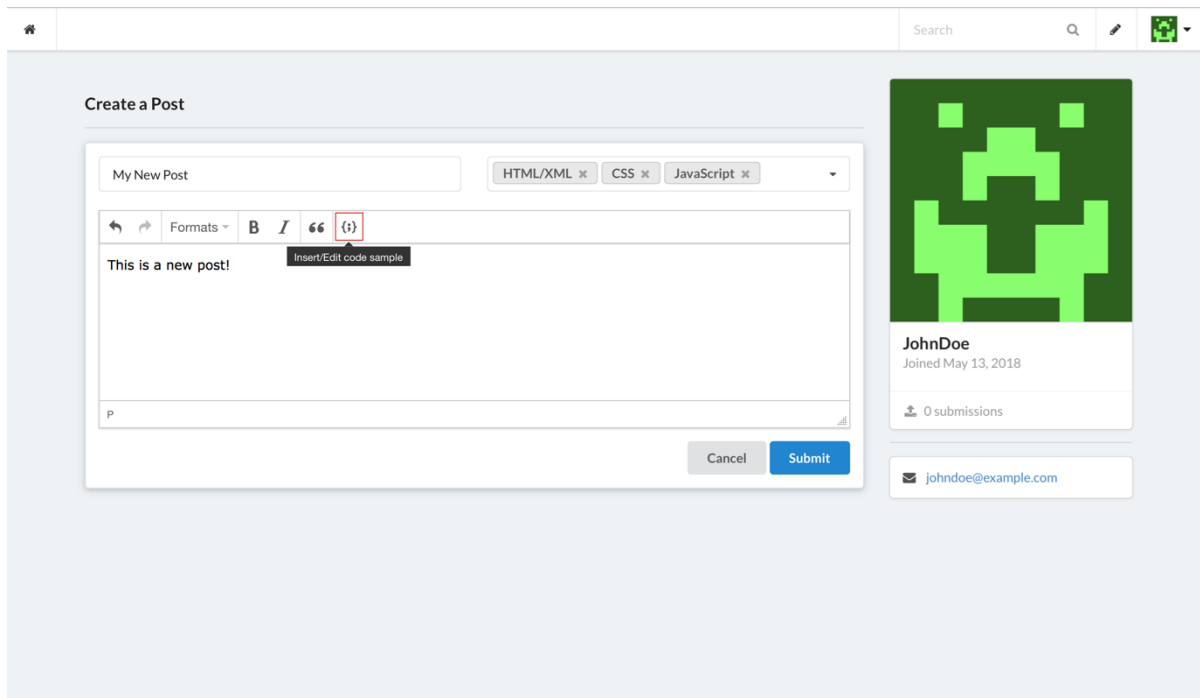


The screenshot shows the 'Create a Post' form. The title field contains 'My New Post'. The tags field shows 'HTML/XML', 'CSS', and 'JavaScript'. The text area contains 'This is a new post!'. The form has 'Cancel' and 'Submit' buttons. Below the form, there is a validation message box with the following requirements:

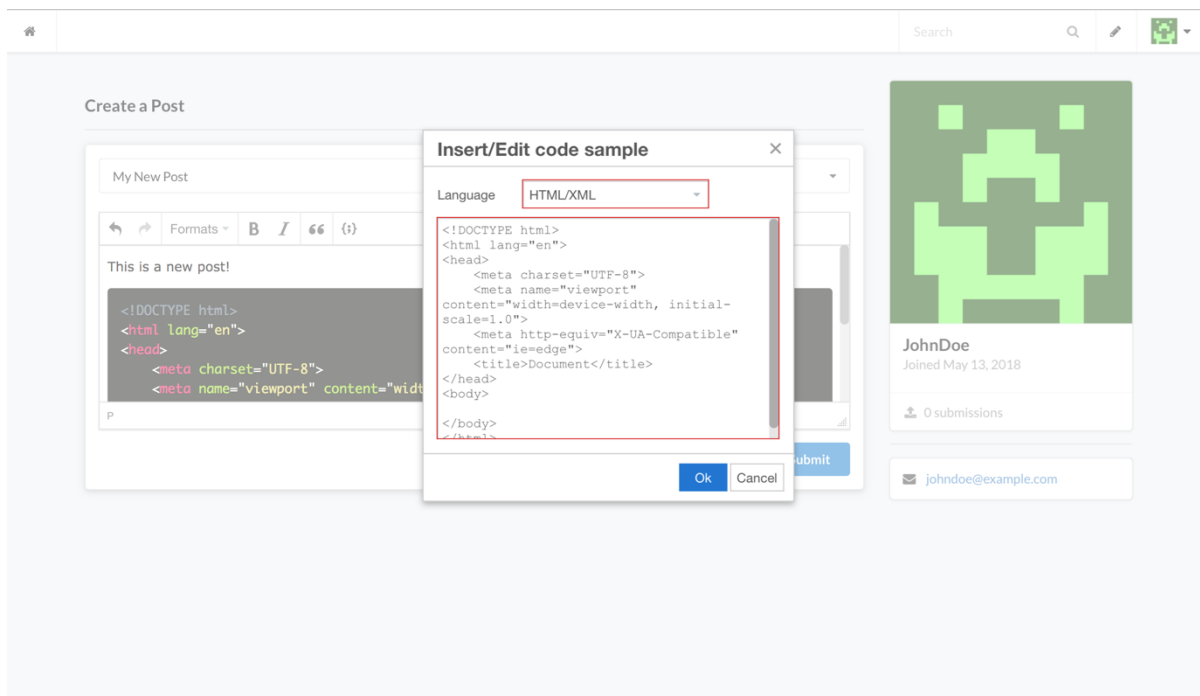
- Please enter a title
- Title **cannot** contain any **special characters**
- Post must contain **at least 1 tag**
- Post must contain **5 tags or less**
- Please enter some post content

CODE POST

To insert one or many blocks of code into a post, access the option from within the post bodies text editor:



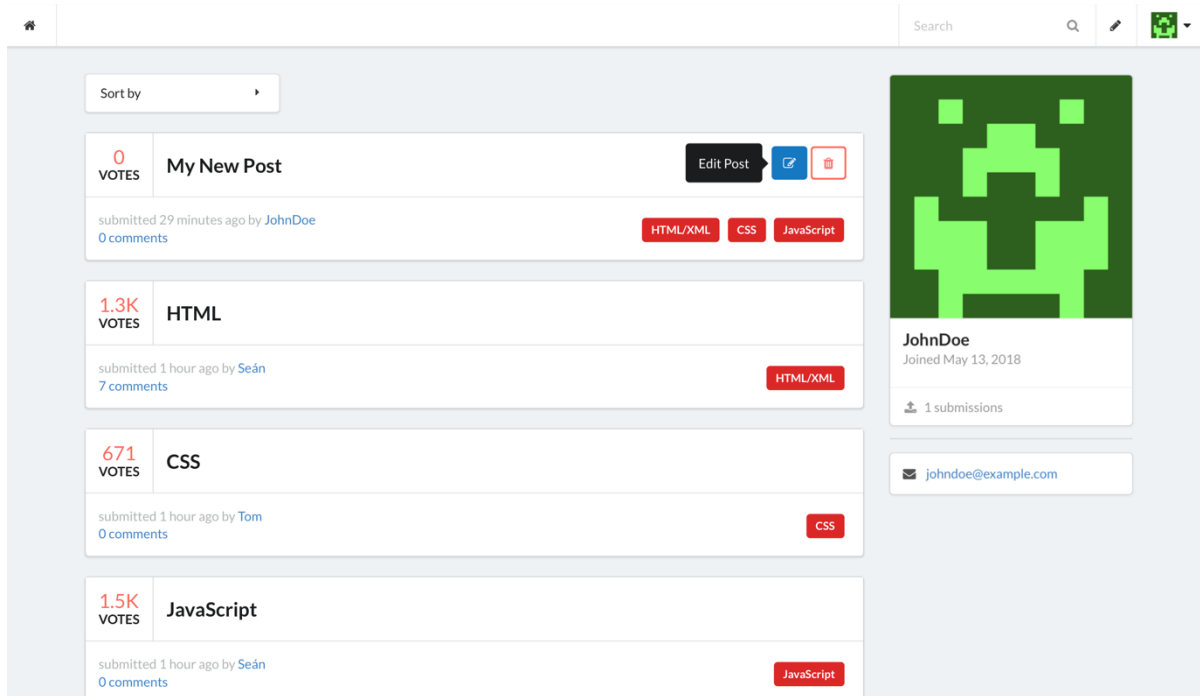
From here, select a language via the dropdown and paste a code block relevant to the language selected:



As illustrated beneath the modal, the code block will appear as such within the text editor upon confirming.

EDITING A POST

To edit an existing post, access the post edit view via the following element encompassed by posts owned by a specific user:

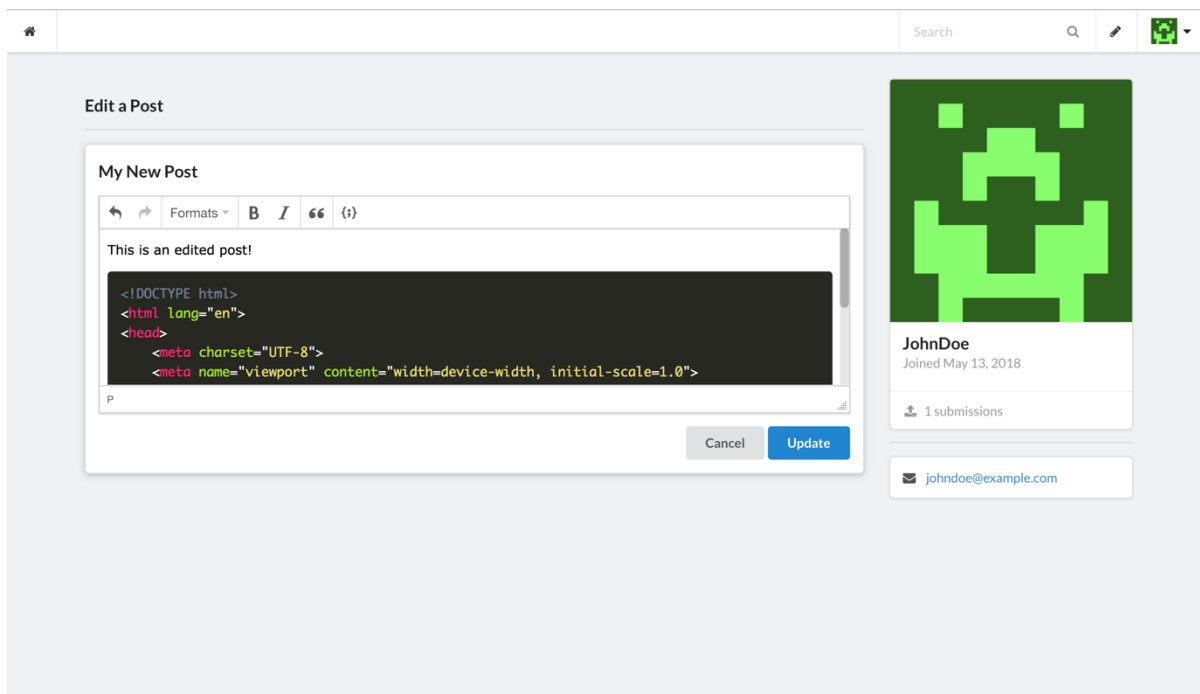


The screenshot shows a user's post feed. At the top, there is a search bar and a home icon. Below the search bar is a 'Sort by' dropdown menu. The main content area displays a list of posts:

- My New Post**: 0 VOTES, submitted 29 minutes ago by JohnDoe, 0 comments. Tags: HTML/XML, CSS, JavaScript. An 'Edit Post' button is visible.
- HTML**: 1.3K VOTES, submitted 1 hour ago by Seán, 7 comments. Tag: HTML/XML.
- CSS**: 671 VOTES, submitted 1 hour ago by Tom, 0 comments. Tag: CSS.
- JavaScript**: 1.5K VOTES, submitted 1 hour ago by Seán, 0 comments. Tag: JavaScript.

On the right side, there is a user profile sidebar for **JohnDoe**, who joined in May 2018 and has 1 submission. The email address is johndoe@example.com.

From here, modify the body of the post or leave it as is. The title or tags cannot not be changed, as this would allow the entire context of the original post to be changed:



The screenshot shows the 'Edit a Post' interface. The title is 'My New Post'. The editor has a toolbar with 'Formats', 'B', 'I', 'Quote', and 'Link' icons. The text area contains the message 'This is an edited post!' followed by a code block:

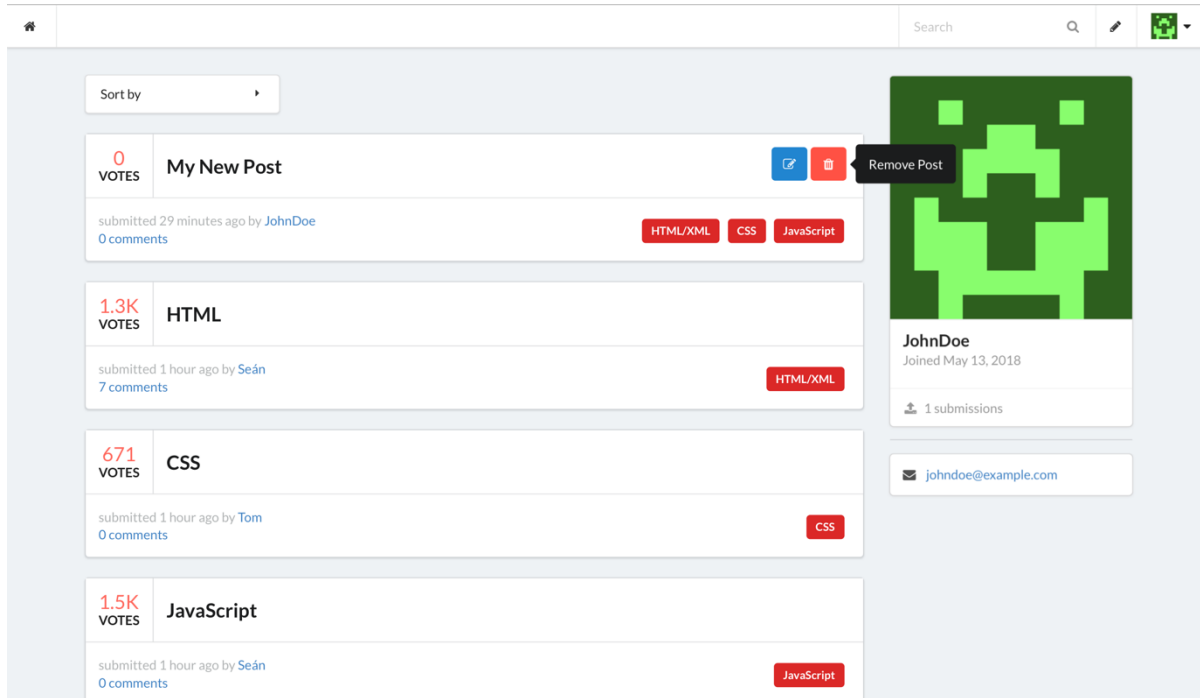
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">

```

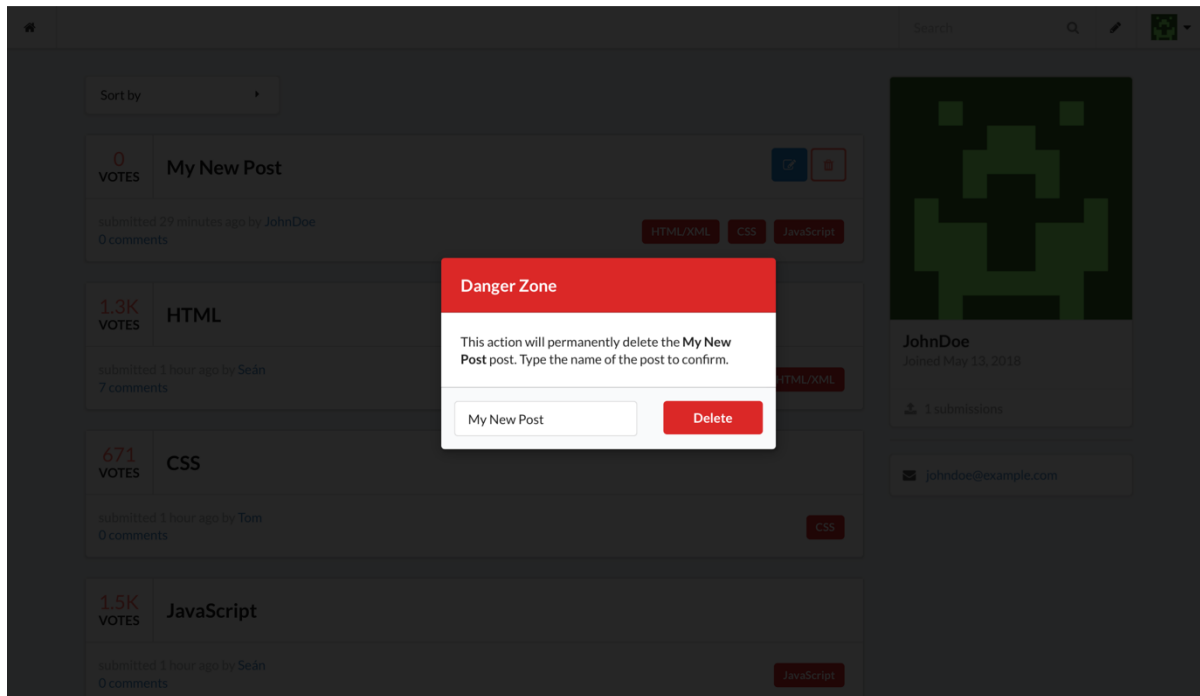
At the bottom of the editor are 'Cancel' and 'Update' buttons. The right sidebar shows the user profile for JohnDoe, with 1 submission and email johndoe@example.com.

REMOVING A POST

To remove an existing post, access the post deletion modal via the following element encompassed by posts owned by a specific user:



From here, type the name of the post to confirm the intention of its deletion via the presented popup modal:



SEARCHING

To search for a post, access the search input via the navigation bar:

The screenshot shows a search results page. At the top right, a search bar contains the text "Title | Tags | User". Below the search bar, there is a "Sort by" dropdown menu. The main content area displays four search results:

- My New Post**: 0 VOTES, submitted 33 minutes ago by JohnDoe, 0 comments. Tags: HTML/XML, CSS, JavaScript.
- HTML**: 1.3K VOTES, submitted 1 hour ago by Seán, 7 comments. Tag: HTML/XML.
- CSS**: 671 VOTES, submitted 1 hour ago by Tom, 0 comments. Tag: CSS.
- JavaScript**: 1.5K VOTES, submitted 1 hour ago by Seán, 0 comments. Tag: JavaScript.

The right sidebar shows the profile of **JohnDoe**, who joined on May 13, 2018, has 1 submission, and an email address of johndoe@example.com.

As illustrated, the three criteria for the search query are returning posts that match or partially match a posts title, tags, or author. From here, enter the details for one of the aforementioned and hit enter:

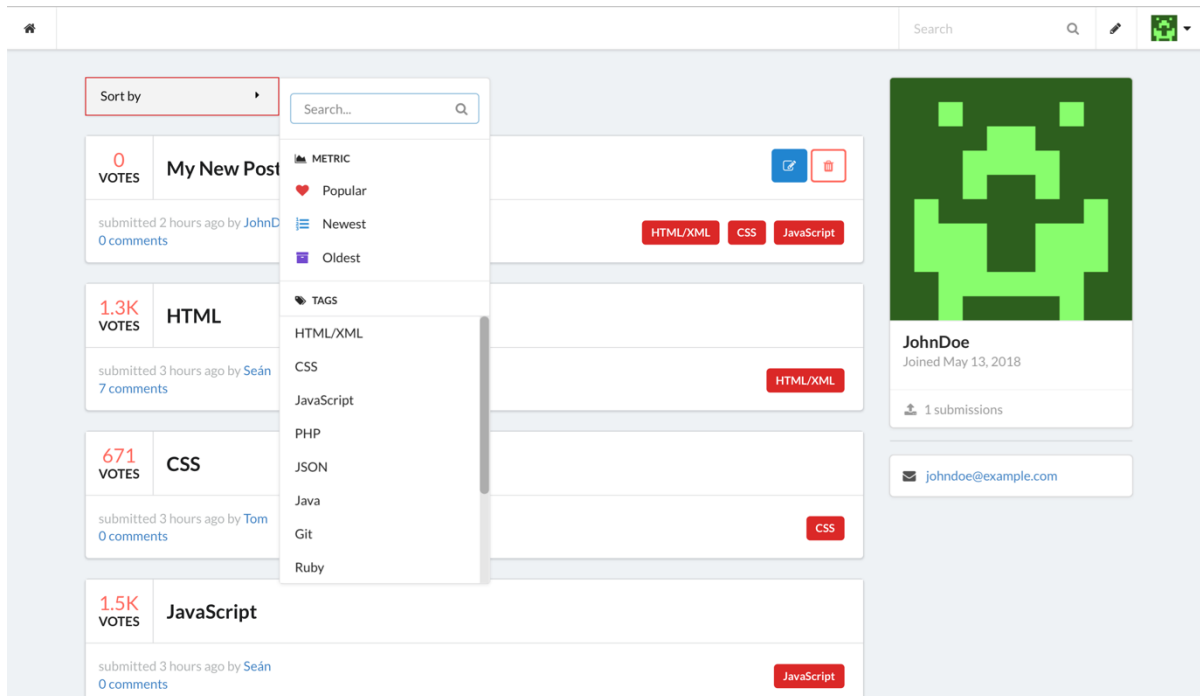
The screenshot shows the same search results page, but the search bar now contains the text "new po". The search results are filtered to show only the first result:

- My New Post**: 0 VOTES, submitted 46 minutes ago by JohnDoe, 0 comments. Tags: HTML/XML, CSS, JavaScript.

The right sidebar remains the same, showing the profile of **JohnDoe**.

FILTERING

To filter posts, select the following element above the list of all posts:



The screenshot shows a web application interface with a list of posts on the left and a user profile on the right. The posts are:

- My New Post**: 0 VOTES, submitted 2 hours ago by JohnDoe, 0 comments. Tags: HTML/XML, CSS, JavaScript.
- HTML**: 1.3K VOTES, submitted 3 hours ago by Seán, 7 comments. Tag: HTML/XML.
- CSS**: 671 VOTES, submitted 3 hours ago by Tom, 0 comments. Tag: CSS.
- JavaScript**: 1.5K VOTES, submitted 3 hours ago by Seán, 0 comments. Tag: JavaScript.

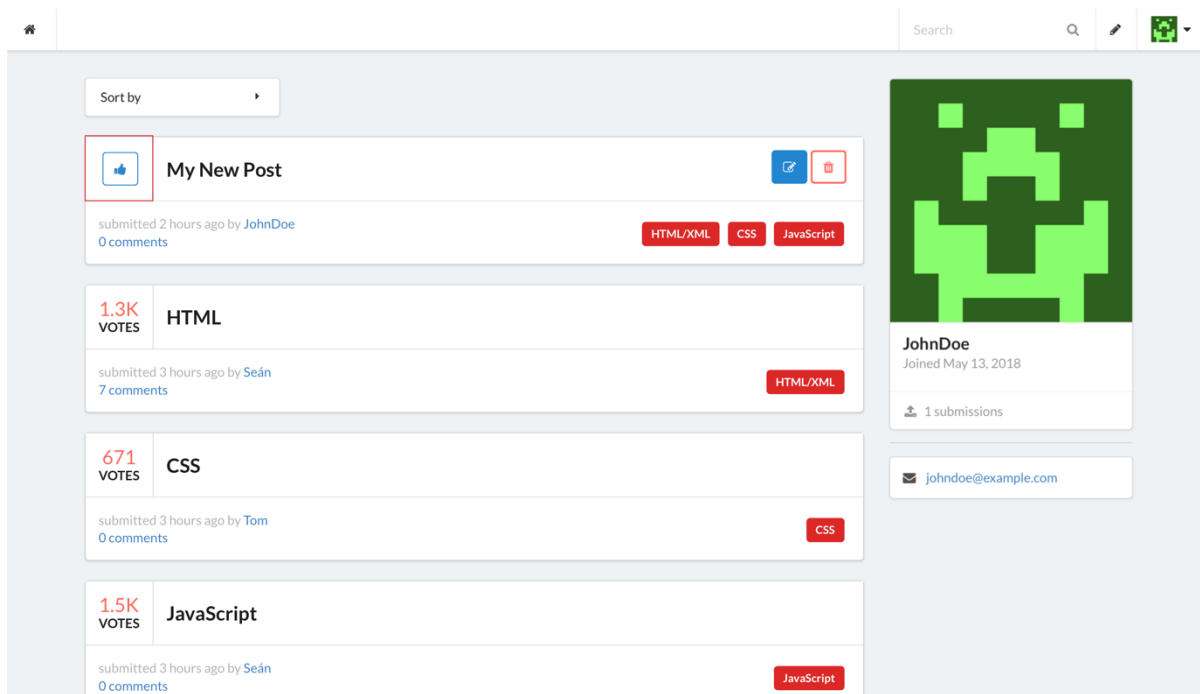
A dropdown menu is open under the 'Sort by' button, showing the following options:

- METRIC
 - Popular
 - Newest
 - Oldest
- TAGS
 - HTML/XML
 - CSS
 - JavaScript
 - PHP
 - JSON
 - Java
 - Git
 - Ruby

From here, select an option manually, or search for one and hit enter when the desired option is found. Note that all example posts were created at the same time, thusly to test metrics such as newest and oldest, you must first add a post to notice those filters in effect.

LIKING/UNLIKING A POST

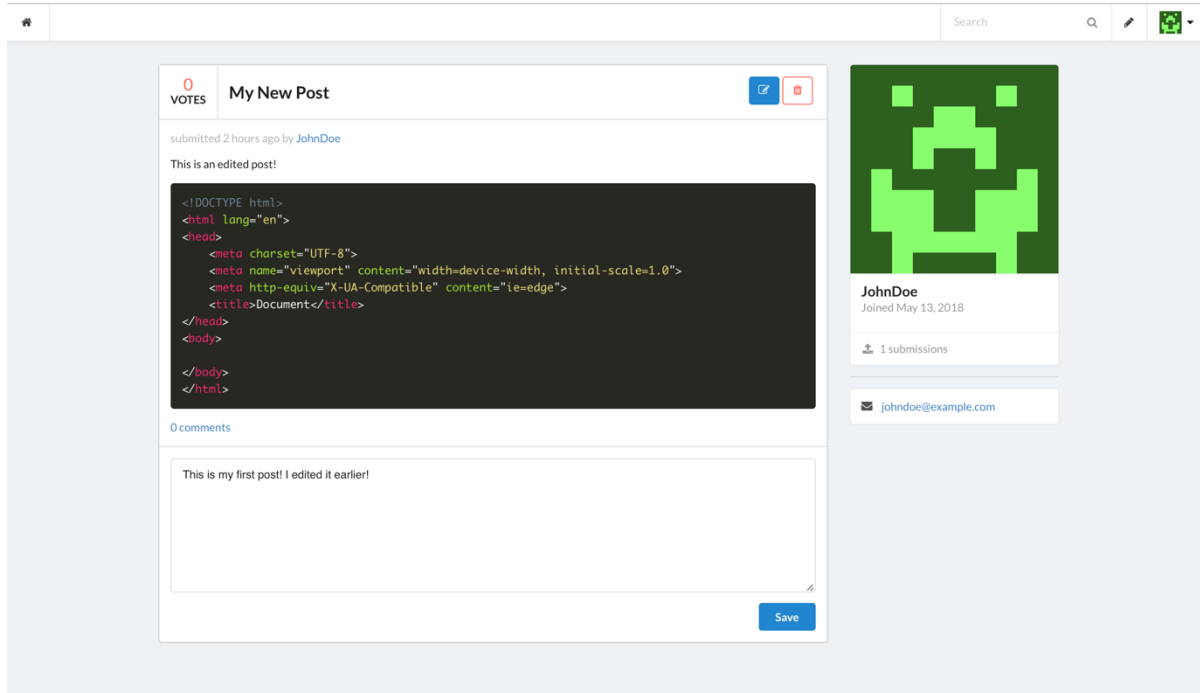
To like/unlike a post, hover over a given posts score to reveal the element and select it:



The screenshot shows the same web application interface as above, but with the 'Like' button (a blue thumbs-up icon) on the 'My New Post' card highlighted with a red box. The rest of the interface, including the list of posts and the user profile, remains the same.

COMMETING/REPLYING

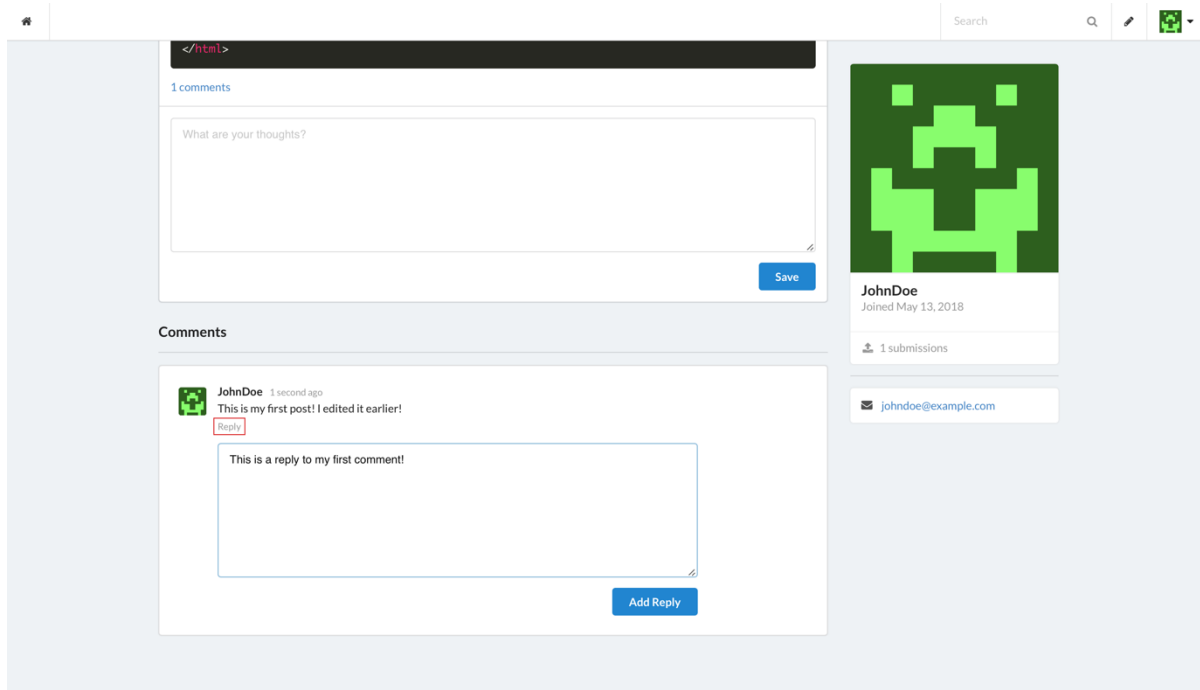
To comment on a post, select one of the posts from the list of all posts to navigate to a dedicated post page:



The screenshot shows a web interface for a post titled "My New Post" with 0 votes. The post was submitted 2 hours ago by JohnDoe. The main content is a code editor displaying HTML boilerplate code. Below the code is a comment box containing the text "This is my first post! I edited it earlier!" and a "Save" button. On the right side, there is a user profile for JohnDoe, joined May 13, 2018, with 1 submission and an email address johndoe@example.com.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
</body>
</html>
```

From here, enter a comment and hit the save button to create a comment on the post in question:

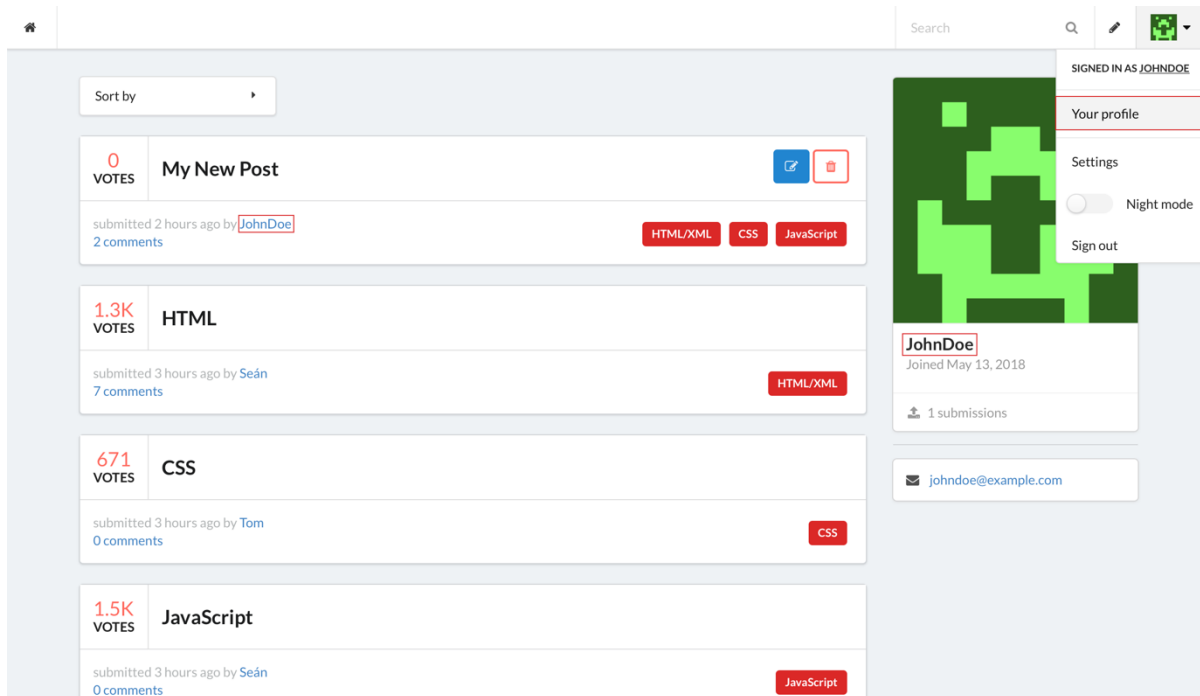


The screenshot shows the same post page, but now with a comment box containing the text "What are your thoughts?" and a "Save" button. Below the comment box is a "Comments" section. The first comment is from JohnDoe, posted 1 second ago, with the text "This is my first post! I edited it earlier!". A "Reply" button is visible next to the comment. Below the reply button is a text input field containing "This is a reply to my first comment!" and an "Add Reply" button.

This creates a new comment on the post. Comments can also be replied to by selecting the element as illustrated.

USER PROFILES

To access your own or another users profile, select one of the following identifying elements related to users:

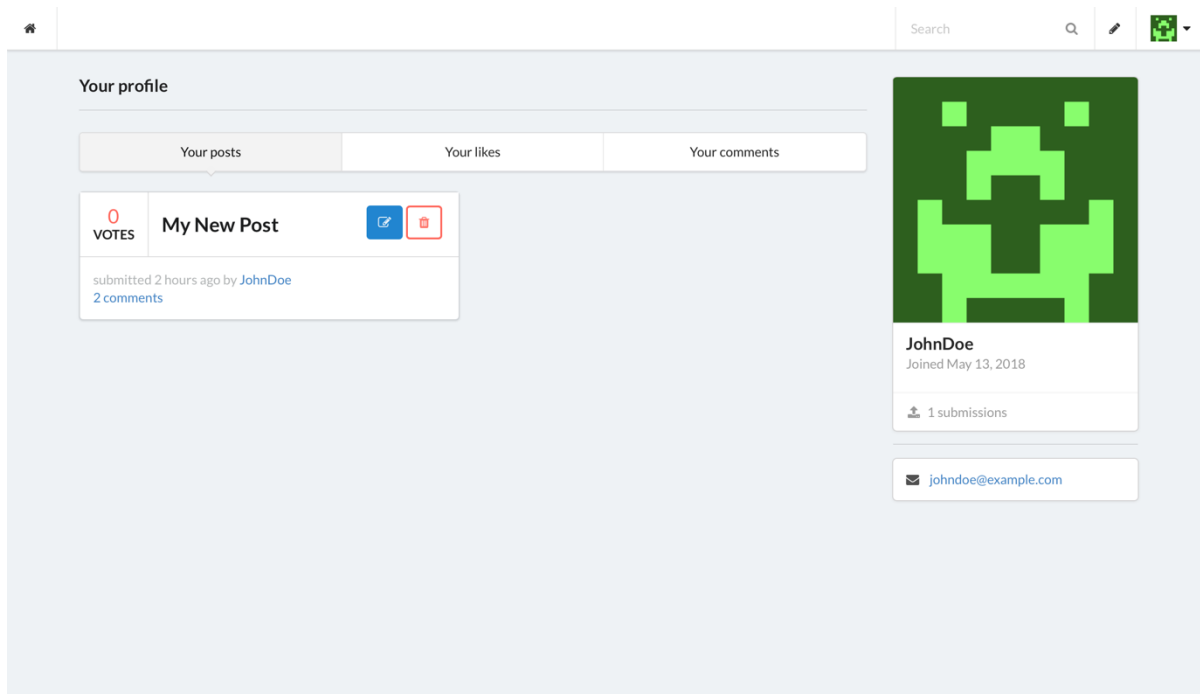


The screenshot shows a user profile page. On the left, there is a list of posts with the following details:

- My New Post**: 0 VOTES, submitted 2 hours ago by [JohnDoe](#), 2 comments. Tags: HTML/XML, CSS, JavaScript.
- HTML**: 1.3K VOTES, submitted 3 hours ago by [Seán](#), 7 comments. Tag: HTML/XML.
- CSS**: 671 VOTES, submitted 3 hours ago by [Tom](#), 0 comments. Tag: CSS.
- JavaScript**: 1.5K VOTES, submitted 3 hours ago by [Seán](#), 0 comments. Tag: JavaScript.

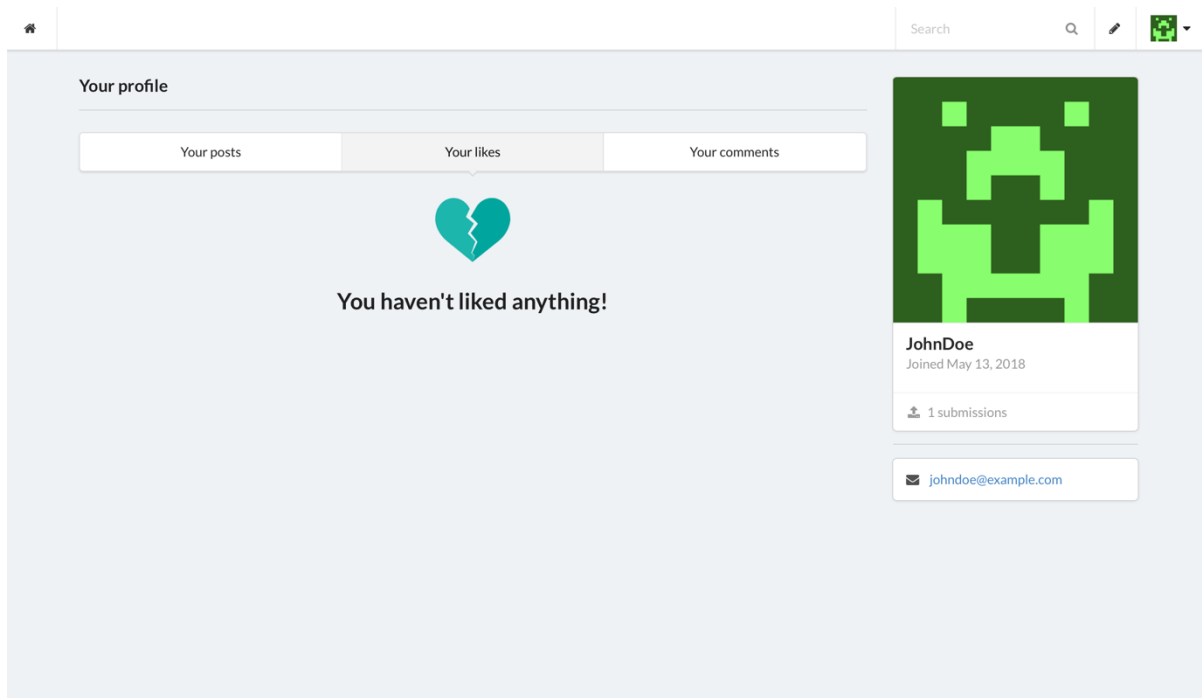
On the right, there is a user profile sidebar for **JohnDoe**, who joined on May 13, 2018, and has 1 submission. The email address is johndoe@example.com. A dropdown menu is open, showing options: SIGNED IN AS JOHNDOE, Your profile (highlighted), Settings, Night mode (toggle), and Sign out.

From here, the a users submissions, liked posts, and comments history can be accessed:

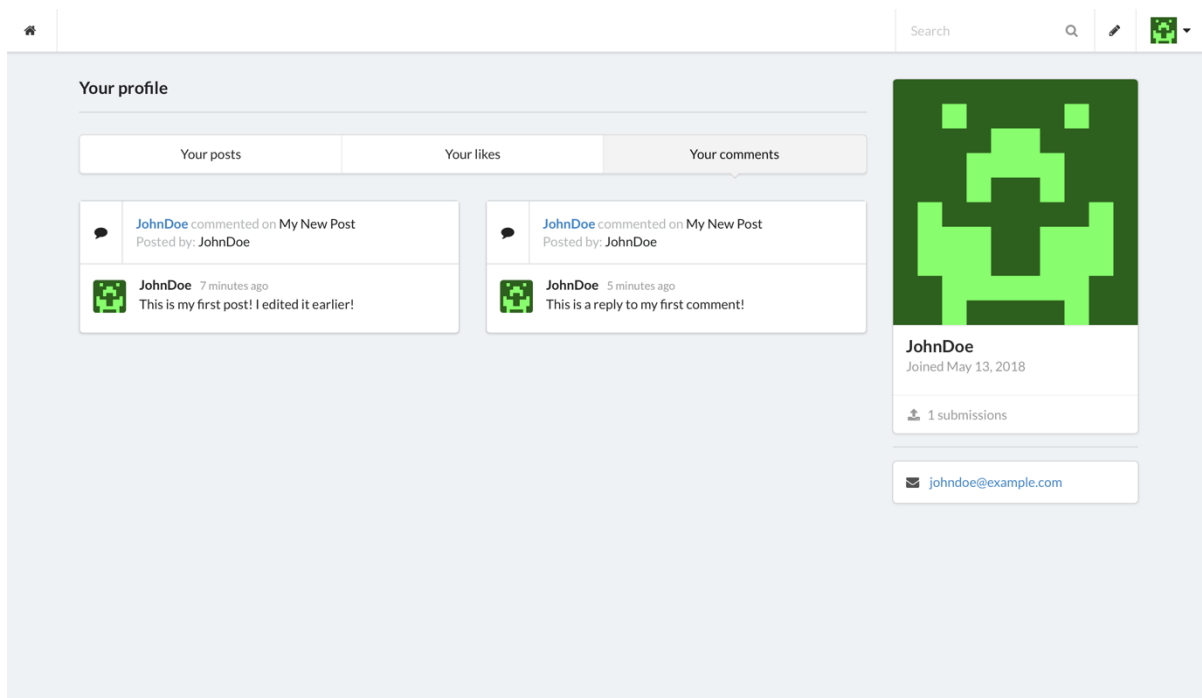


The screenshot shows the 'Your profile' section of the user profile page. It features three tabs: 'Your posts', 'Your likes', and 'Your comments'. The 'Your posts' tab is active, showing the same 'My New Post' as in the previous screenshot. The user profile sidebar on the right is also visible, showing the user's name, join date, and email address.

Posts belonging to the authenticated user on their own profile can be edited and removed.



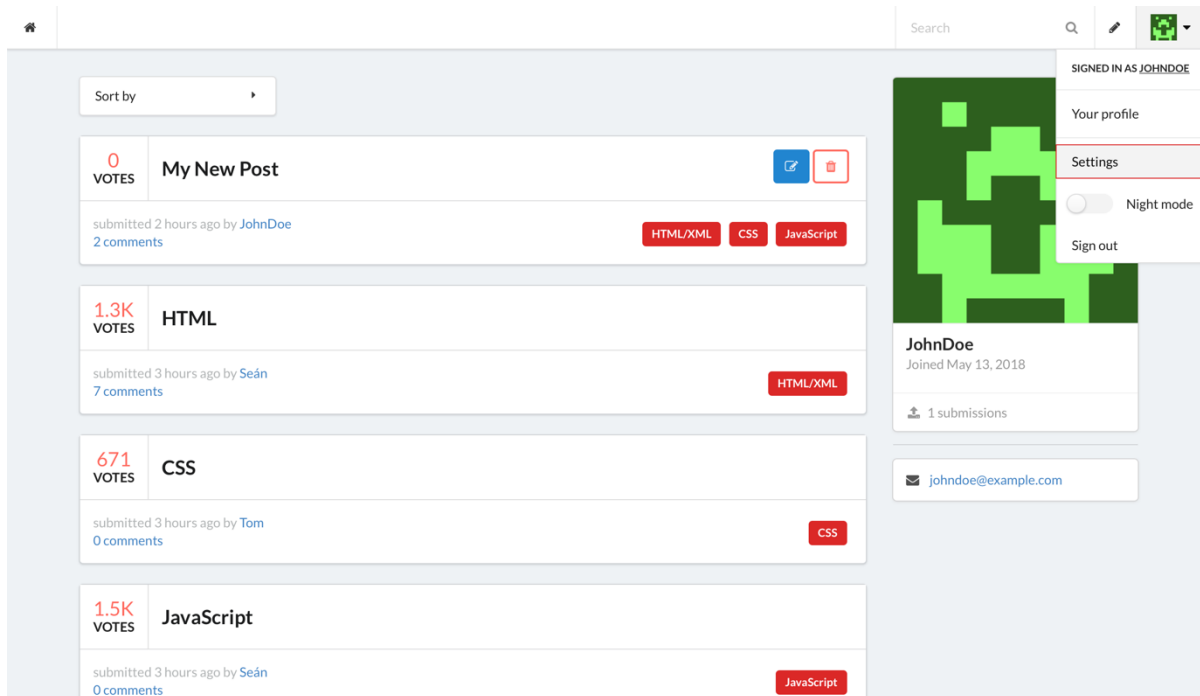
Sometimes, a user may not have any submissions, likes, or comments, and this is communicated as such. In addition to the aforementioned, liked content can be removed from here.



The comments tab provides a user's comment history inclusive of the comment itself, the post it was submitted on, and the author of the post.

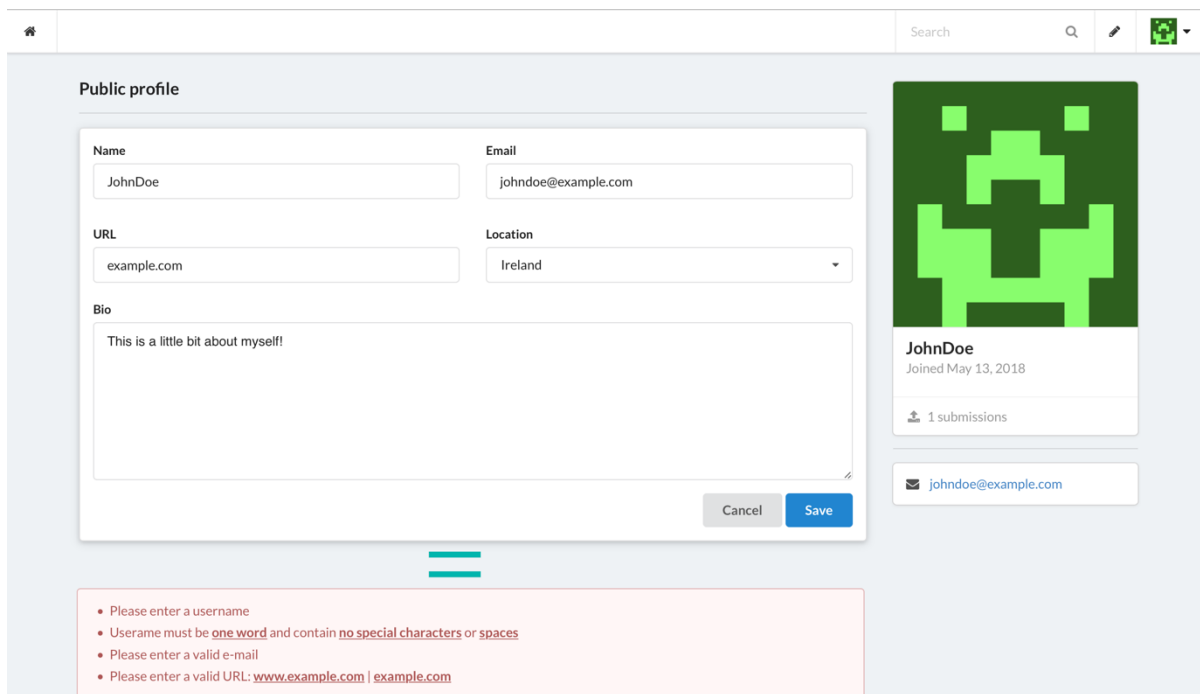
USER SETTINGS

To access user settings, select the following element from the dropdown encompassed by the navigation bar:



The screenshot shows a user's dashboard with a navigation bar at the top. The navigation bar includes a search icon, a pencil icon, and a user profile icon. A dropdown menu is open, showing the user's name "JOHNDOE" and options for "Your profile", "Settings" (highlighted), "Night mode", and "Sign out". Below the navigation bar, there is a "Sort by" dropdown and a list of posts. The first post is "My New Post" with 0 votes, submitted 2 hours ago by JohnDoe, with 2 comments and tags for HTML/XML, CSS, and JavaScript. The second post is "HTML" with 1.3K votes, submitted 3 hours ago by Seán, with 7 comments and a tag for HTML/XML. The third post is "CSS" with 671 votes, submitted 3 hours ago by Tom, with 0 comments and a tag for CSS. The fourth post is "JavaScript" with 1.5K votes, submitted 3 hours ago by Seán, with 0 comments and a tag for JavaScript. On the right side, there is a user profile card for JohnDoe, joined May 13, 2018, with 1 submission and an email address johndoe@example.com.

From here, a user's details can be changed inclusive of optional identifying information, such as an affiliated URL, location, and bio. Note, a name and email cannot be blank nor can they equal the details of an existing user. Note the validation requirements in place as illustrated:



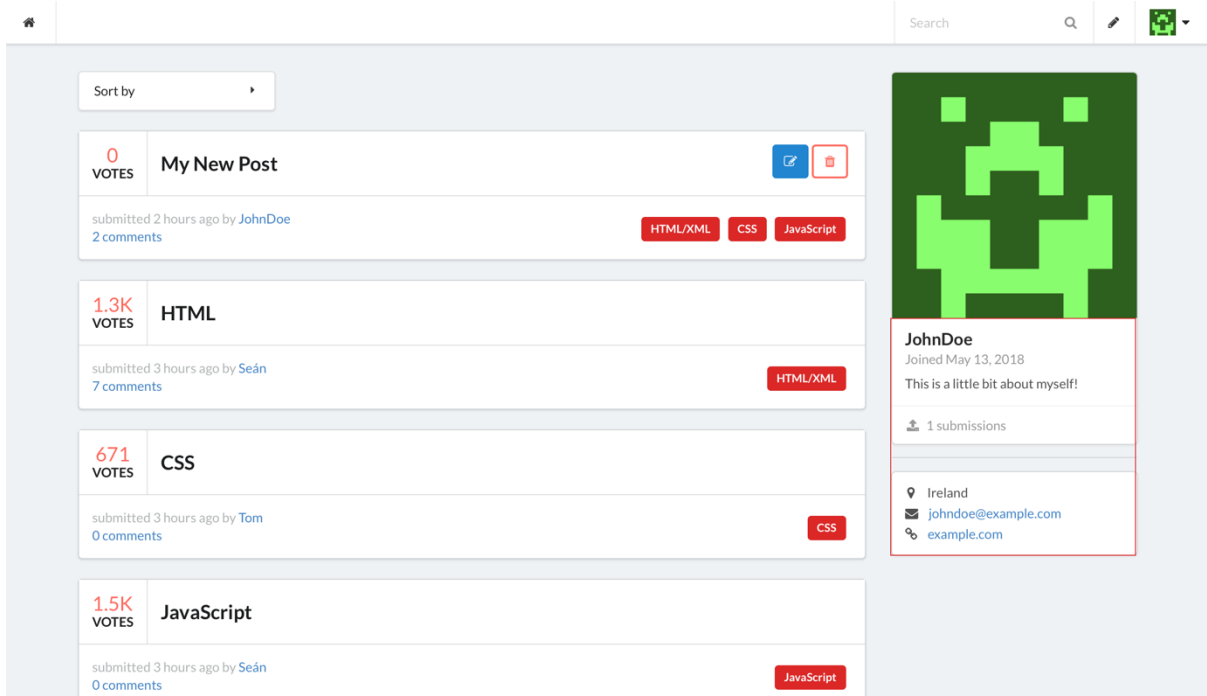
The screenshot shows the "Public profile" settings form. The form has the following fields:

- Name: JohnDoe
- Email: johndoe@example.com
- URL: example.com
- Location: Ireland
- Bio: This is a little bit about myself!

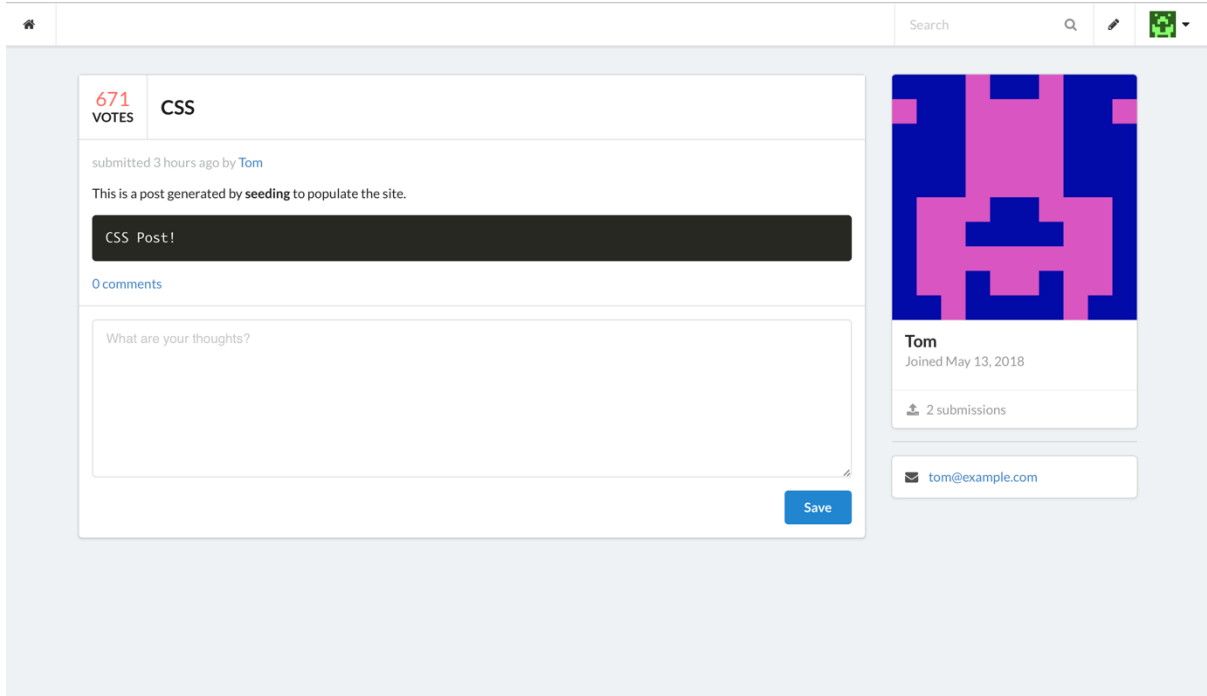
At the bottom of the form, there are "Cancel" and "Save" buttons. Below the form, there is a validation requirements list:

- Please enter a username
- Username must be one word and contain no special characters or spaces
- Please enter a valid e-mail
- Please enter a valid URL: www.example.com | example.com

Changes are reflected in the authenticated users card:



In another users post or profile, *their* user card is shown instead. Take for instance, the CSS post:



Within Tom's post, information about him is displayed instead of the authenticated users card. This is inclusive of the details as changed above.

NIGHT MODE

To access night mode, select the element within the dropdown menu encompassed by the navigation bar:

