

# CodeAnnotator

Henry Feild

Thesis II

Endicott College

Spring 2019

# Table of Contents

<b>Table of Contents</b>	<b>1</b>
<b>Abstract</b>	<b>2</b>
<b>1. Introduction &amp; Background</b>	<b>3</b>
<b>3. Client Requirements, Features, and Assumptions</b>	<b>9</b>
<b>4. User Interface Design</b>	<b>11</b>
<b>5. System Architecture and Design</b>	<b>12</b>
<b>6. Data Design</b>	<b>13</b>
<b>7. System Analysis &amp; Testing</b>	<b>14</b>
<b>8. Conclusions</b>	<b>15</b>

# Abstract

Providing detailed feedback to students about programming assignments is challenging in a computer science class. Source code is meant to be viewed and manipulated on a screen while feedback is often more easily provided by handwritten annotations—a series of circles, lines, scratched out text, and comments overlaid on the code itself. In this thesis project, I survey and evaluate the current options with regard to providing feedback on source code before describing a novel custom solution I created: a web application called CodeAnnotator.

# 1. Introduction & Background

An important requirement of teaching others how to program is providing feedback on the source code of their programs. For my programming classes, I often comment on students' formatting style, logical errors, inaccurate or inefficient lines of code, and documentation. For example, consider the C++ program in Figure 1, which runs on the command line and allows the user to provide any three of the variables  $x$ ,  $y$ ,  $m$ , and  $b$  and solve for the remaining variable using the formula for a point on a line,  $y = m \times x + b$ . There are several errors that prevent this code from compiling and running and several conventions that are violated that impact the readability and maintainability of the code. For instance, line 6 contains a declaration for a variable named `z`, which is used to store the name of the variable the user would like to solve for. The name `z` does not convey that, however. A better name would be `userInput` or `varToSolveFor`.

Providing this feedback in a written form is important because it allows both the instructor and the student to keep a record of the feedback for future reference. Based on my personal experience, a good method for providing written feedback is one where the comments: are easy for the instructor to make, can be distributed to multiple people (in the case of group work), line up to the original code, and are easily accessible to all parties.

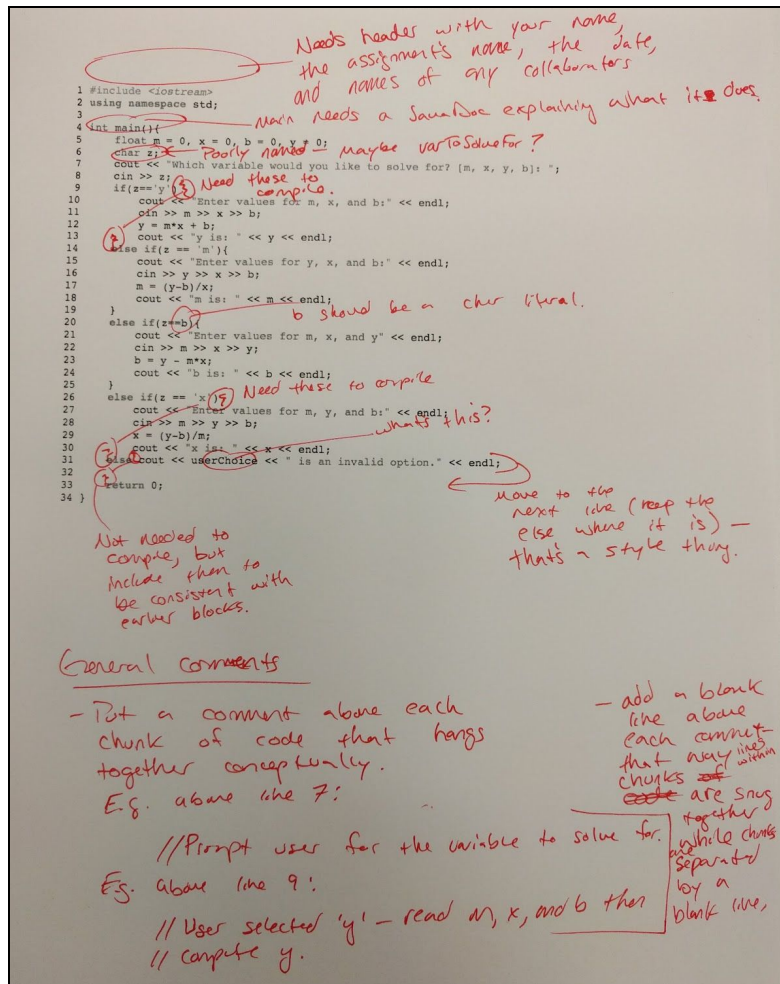
```

1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      float m = 0, x = 0, b = 0, y = 0;
6      char z;
7      cout << "Which variable would you like to solve for? [m, x, y, b]: ";
8      cin >> z;
9      if(z=='y')
10         cout << "Enter values for m, x, and b:" << endl;
11         cin >> m >> x >> b;
12         y = m*x + b;
13         cout << "y is: " << y << endl;
14     else if(z == 'm'){
15         cout << "Enter values for y, x, and b:" << endl;
16         cin >> y >> x >> b;
17         m = (y-b)/x;
18         cout << "m is: " << m << endl;
19     }
20     else if(z==b){
21         cout << "Enter values for m, x, and y" << endl;
22         cin >> m >> x >> y;
23         b = y - m*x;
24         cout << "b is: " << b << endl;
25     }
26     else if(z == 'x')
27         cout << "Enter values for m, y, and b:" << endl;
28         cin >> m >> y >> b;
29         x = (y-b)/m;
30         cout << "x is: " << x << endl;
31     else cout << userChoice << " is an invalid option." << endl;
32
33     return 0;
34 }

```

**Figure 1.** The C++ source code for a program that computes the missing value in the line formula.

There are many ways to communicate written feedback on code, but all of them have substantial drawbacks that make them less than ideal for use in a programming class. In this section, I describe two categories and related work that falls into each and why those related works are lacking for my purpose.



**Figure 2.** Handwritten comments on a printout of the code.

The first category is handwritten feedback and there are a variety of physical and digital methods for this. The simplest is to print code out onto paper, annotate the paper copy, and hand that back to the student. Oddly, the first step of that—printing—is the most challenging. Unlike word processor documents, source code is not created with an 8.5"x11" piece of paper in mind. It is for consumption of other programmers, who view it through a source code editor with monospace font (reducing the amount of text that can fit on a line), specialized syntax highlighting, hyperlinks, and specific line wrapping rules or vertical scrolling. Once the code is printed, annotations can be made. However, disseminating those annotations require delivering them physically, making copies (in the event of distributing to a team), or scanning comments



effort. An example of this can be found in Figure 3 (right). In all cases, annotations can be difficult to make or read because of repeated errors, poor legibility (either due to handwriting or overlapping text), and source code blocks spanning multiple pages.



**Figure 4.** A screenshot of Crucible. Note that source code versioning information is present. Alternative code is not supported.

A second category consists of tools for performing code reviews, such as Gerrit Code Review,<sup>4</sup> Review Assistant,<sup>5</sup> and Crucible.<sup>6</sup> A code review is a common procedure for software development teams. Anytime a developer puts forth new or modified code, it must be reviewed by a peer developer to ensure accuracy, conformance to the team's style guidelines, and a variety of other characteristics. In some ways, this is analogous to a professor providing feedback to students. However, many of these tools expect version control integration (e.g., Git) and are

<sup>4</sup>Gerrit Code Review, <https://www.gerritcodereview.com/>

<sup>5</sup> Review Assistant, <https://www.devart.com/review-assistant/>

<sup>6</sup> Crucible, <https://www.atlassian.com/software/crucible>



setup to allow back and forth code changes. These extra features make the interfaces complicated and difficult to use for the purpose of providing simple feedback to a student.

Consider Figure 4, which shows the interface for Crucible. A portion of code is displayed after an update. Code highlighted in red was removed and code in green was added in the update. A conversation between the updater and the reviewer is also shown. This is more information than is needed and is more likely to confuse students (especially in an introductory class).

The lack of suitable methods for easily providing written feedback on code is the motivation for CodeAnnotator, a web application I developed to address this problem. Unlike the paper and PDF solutions, CodeAnnotator allows code to reside in its original plain text form with appropriate syntax highlighting. Unlike code review solutions, CodeAnnotator only exposes features that are relevant to providing feedback to students. CodeAnnotator allows the same comment to be attributed to multiple points in the code. The professor (or whoever is annotating the code) can also supply alternate code for any portion of the code, providing matching formatting. CodeAnnotator is open source<sup>7</sup> and freely available online.<sup>8</sup> Files and projects can be shared publicly (anyone with the link can see comments) or privately to select users. I describe the complete list of features as well as the system design, data design, and interface in the following sections.

The remainder of this thesis is set up as follows. In Section 2, I list the requirements that CrowdLogger must satisfy and the assumptions I made when developing the application. In Sections 3–5, I describe the user interface, the system components, and the data model. In Section 6, I describe how I ensured it works as intended, and parting conclusions follow in Section 8.

---

<sup>7</sup> CodeAnnotator GitHub page with source code: <https://github.com/hafeild/code-annotator>

<sup>8</sup> Live CodeAnnotator instance: <https://annotate.feild.org>

### 3. Client Requirements, Features, and Assumptions

In this section, I describe the requirements a satisfactory solution to the problem of providing programming feedback to students must provide. I indicate which of these requirements are implemented in CodeAnnotator and additional features CodeAnnotator possesses that extend its functionality. Finally, I list assumptions that are made in fulfilling these requirements, some of which limit CodeAnnotator's effectiveness when those assumptions are violated.

A satisfactory solution to the problem of providing feedback on code to students should meet the following requirements:

#### Requirements

- providing feedback requires few steps
- feedback is easy for the students and the instructor to see
- feedback can be provided to a group of students
- code can be represented the same way it was submitted (e.g., with syntax highlighting and not line wrapped)
- the same comment can be easily associated with multiple places in the code
- it is easy to provide code snippets to demonstrate how a piece of code should be altered

CodeAnnotator includes features to meet these requirements as well as additional ones:

#### Additional Features

- user accounts
- projects: users can add code to projects, and projects can be shared with other users
- link-based sharing: projects can also be shared with non-users via link sharing

- multiple links can be created for the same project, and links can be revoked at any point preventing sharing after a link has been distributed
- web based and cross-browser tested, so anyone can access the application
- plain text files can also be annotated
  - any unsupported programming language is treated as plain text (i.e., no syntax highlighting)

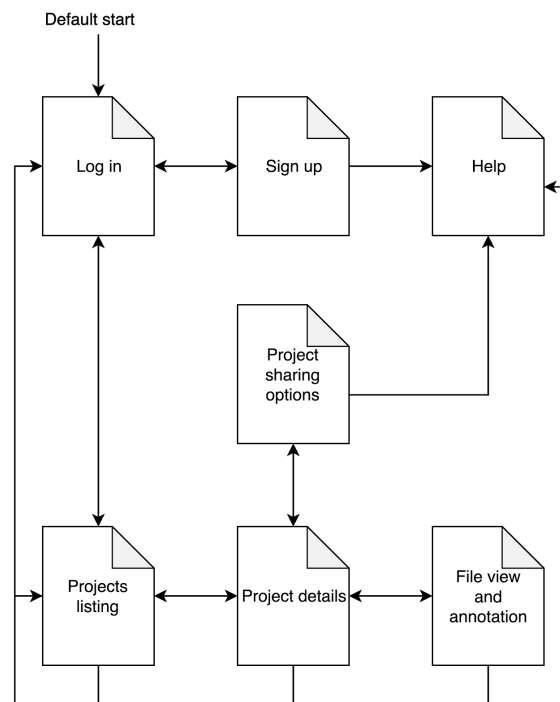
CodeAnnotator makes a number of assumptions, as described below:

#### Assumptions

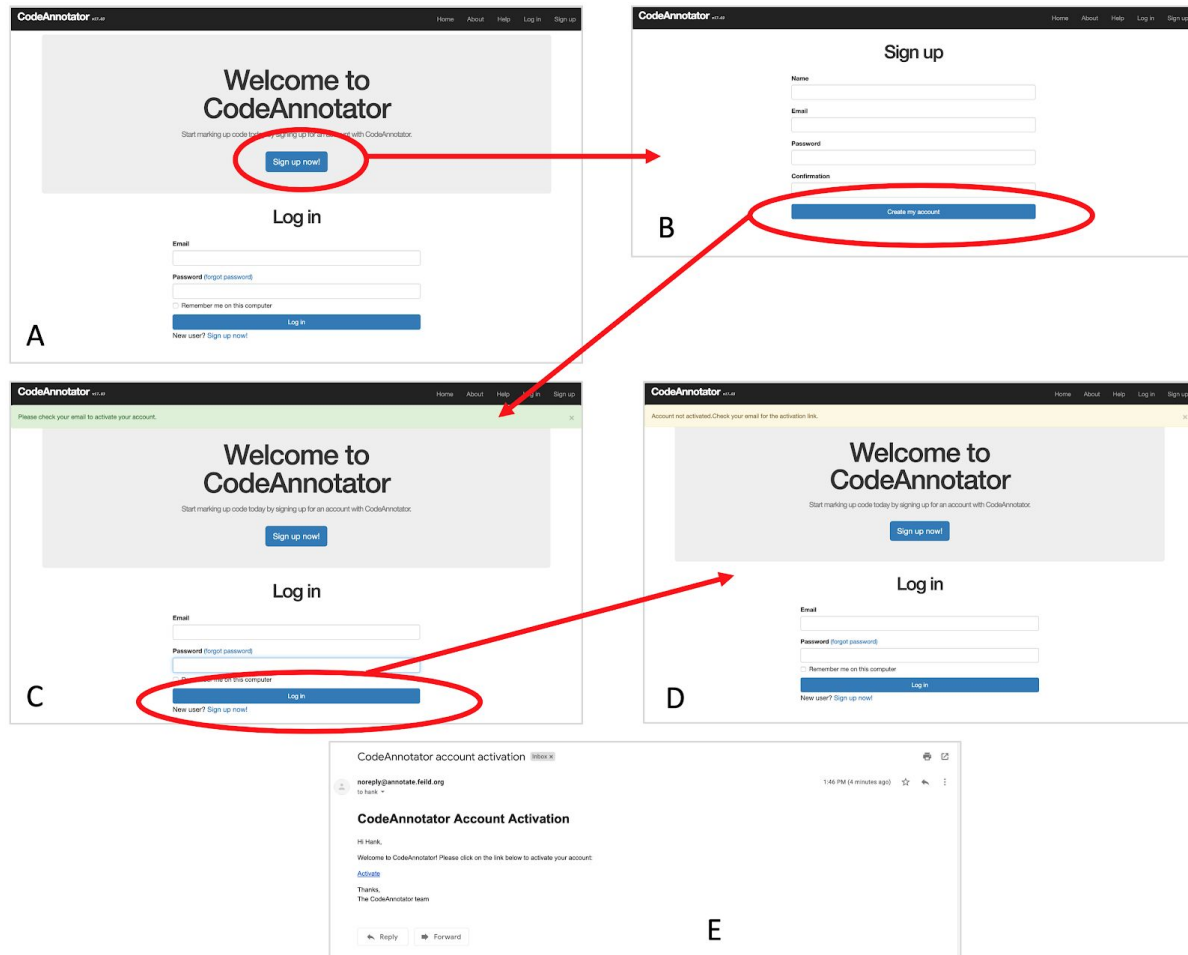
- users will have internet access
  - feedback cannot be downloaded or printed for offline access
- instructors will only want syntax highlighting for certain programming languages (the ones supported by the library)
- source code files use specific extensions corresponding to the language of the program
  - e.g, a C++ source code file must end in .cpp in order to activate C++ syntax highlighting

## 4. User Interface Design

The user interface of CrowdLogger consists of several web pages, as mapped in Figure 5 and shown in Figures 6–10. The log in, sign up, and help pages are accessible by anyone (no user account is required). A logged in user must log out to access the log in and sing up pages. The projects listing page is the root page for any logged in user—a logged in user navigating to the CodeAnnotator homepage will be redirected to this page. The project details page display the files for a specific project. The project details page also embeds the annotations for a selected project file and project sharing management (these are depicted as separate pages in Figure 5). In order to view a project details page, a user must either be logged in and have permissions, or must access it via a public link.



**Figure 5.** The site map for CodeAnnotator.



**Figure 6.** The signup (A, B), login (C) screens and email verification (E). Note that logging in without verifying the associated email address will raise an error (D).

## Sign up process

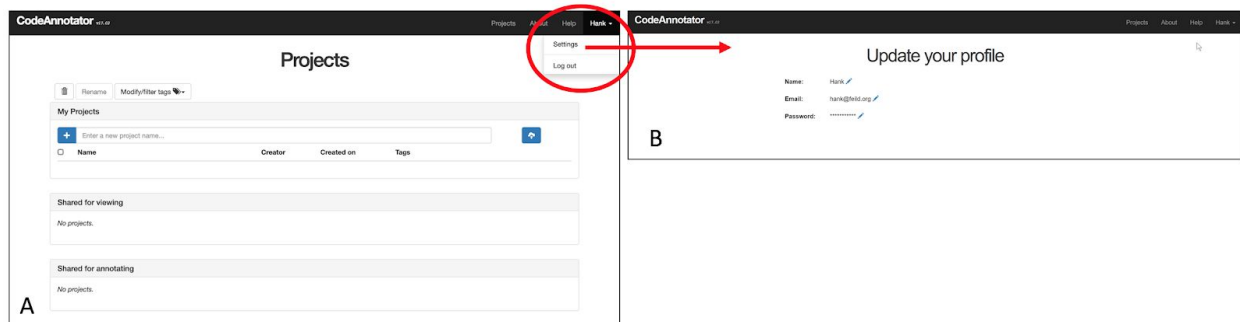
When a new user arrives to the CodeAnnotator site, they must click on a "Sign up now!" button (Figure 6A). When signing up, they supply a name, a unique email (no two user accounts may share an email address), and a password (Figure 6B). Upon submission, if the email has been taken, their name is left blank, or their password is too short, an error message will appear. If successful, the user will automatically be sent an email with a link to activate their account (Figure 6E) and then redirected to the homepage. A toast will appear at the top of the page informing them "Please check your email to activate your account." (Figure 6C). If the user

attempts to log in prior to activating their account, they will be redirected to the home page with a warning toast appearing stating "Account not activated. Check your email for the activation link." (Figure 6D). Once the user activates their account by clicking on the URL in the email, they will be logged in and directed to their projects page, with a one-time toast that says "Account activated!" (Figure 7A).

Logged out users can log in via the CodeAnnotator home page, which features a log in form. Logged in user may log out at any time by clicking the "Log out" button in the menu bar that is displayed at the top of every page.

Although not shown, a user who has forgotten their password or failed to activate their account may request a password reset. This causes an email to be sent with a link to a page where they can reset their password. The link cannot be used multiple times. Once their password is reset, the user is logged into their account and directed to the projects page.

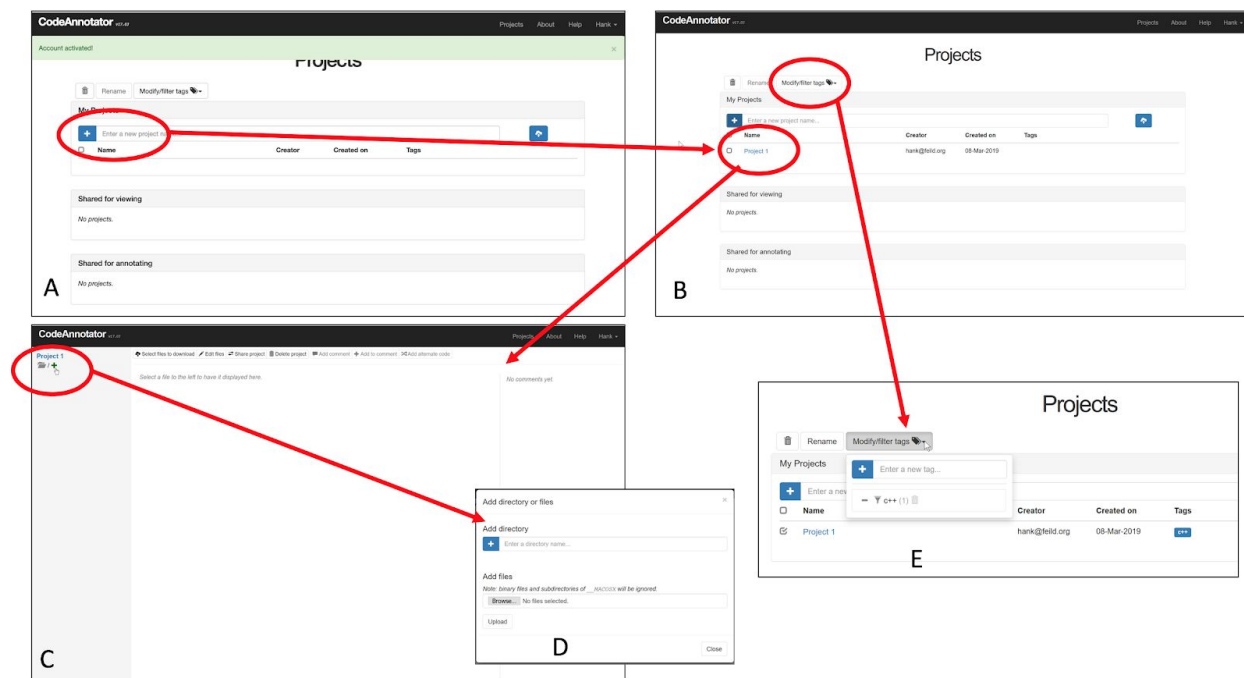
Logged in users may update their name, email address, or password by clicking their name on the menu bar and selecting "Settings" (Figure 7). Changing an email triggers a confirmation email to be sent to the user with a link the user must follow, much like the account activation link works.



**Figure 7.** How to edit user credentials.


## Project listing


CodeAnnotator requires that files belong to a project. Projects may be shared with other users for editing, viewing, or annotating, or with non-users for viewing. Editing gives users the full rights as the original creator of the project—they may add or modify files, sharing permissions, or make annotations. Viewing limits a user to only examining the files and annotations in a projects, but they may not add or modify files, annotations, or change permissions. Annotating prevents a user from adding or modifying files or permissions, but does allow annotations of the files to be made. This is a good option for users that are students who want to share their code with an instructor for feedback, or an instructor who want to share a student's project with a teaching assistant or grader.



**Figure 8.** The projects page (A), adding new projects (A->B), adding a tag to a project (B->E), opening a project (B->C) and uploading source code files to a project (C->D).

Once logged in, the user will be brought to a listing of their projects. They can see which projects they have edit rights to, are shared with them for viewing, and are shared with

them for annotating. A new user will see a section for each of these, but no projects (Figure 8A). To create new projects, the user has three options: a) make a single, empty project, b) upload one or more files into a new project or c) bulk upload several projects from a Zip file. Option (a) is depicted in Figure 8A and B. The user must provide a name for the project in the text field at the top of the "My projects" section of the projects listing page and then click the  to the left. This will cause a new project with the given name to appear under the "My projects" section.

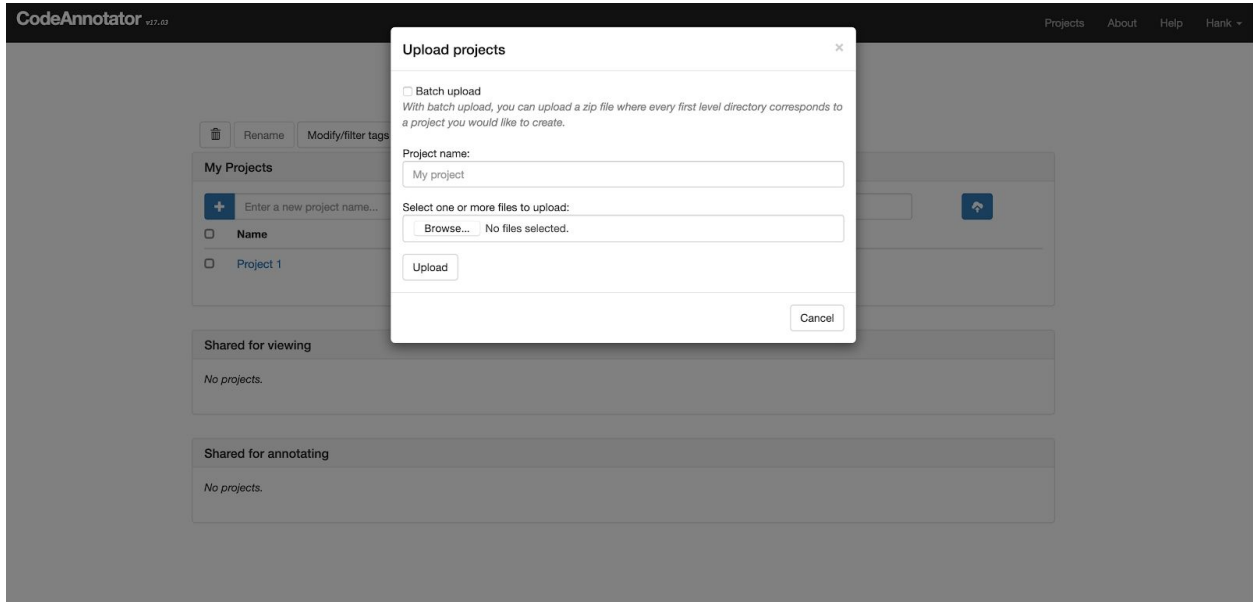
Clicking on the  button at the top right of the "My projects" section causes a modal to appear where the user may carry out options (b) or (c) (Figure 9). To perform option (c), the user must provide a project name, then select one or more files (Zip or source code files), then click "Upload". A project with the given name will then appear in the "My projects" section of the projects listing page.

To perform option (c), the user must check the box next to "Batch upload" in the modal. Bulk upload allows the user to upload a Zip file wherein every first-level folder will be associated with a project, and all sub-folders and files will belong to that project. The user may additionally select an option "Update" (Figure 10). If checked, existing projects that match the first-level folder names will be updated with the associated files; otherwise, new projects (potentially with duplicate names of existing projects) will be created.

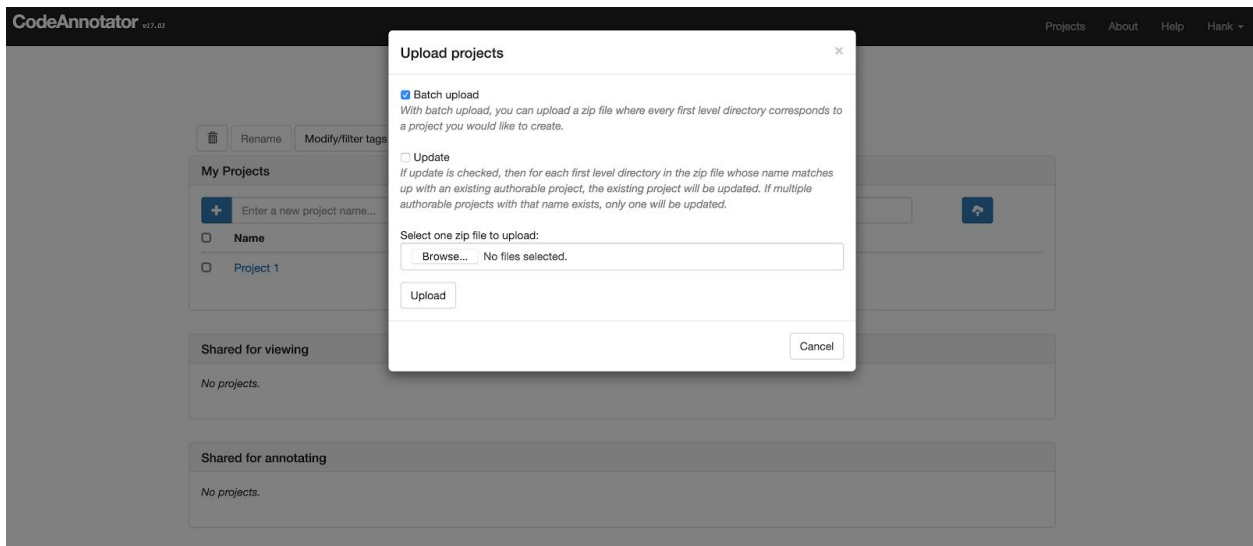
The update option is particularly useful when adding new programs to student projects. For example, after collecting a second programming assignment from students in a class, they can be trivially added to each student's CodeAnnotator project through the batch upload and update feature.

Projects may also be tagged (e.g., with a class number or semester) (Figure 8 B and E). Projects can be further filtered by the presence of one or more tags.





**Figure 9.** The bulk upload modal.

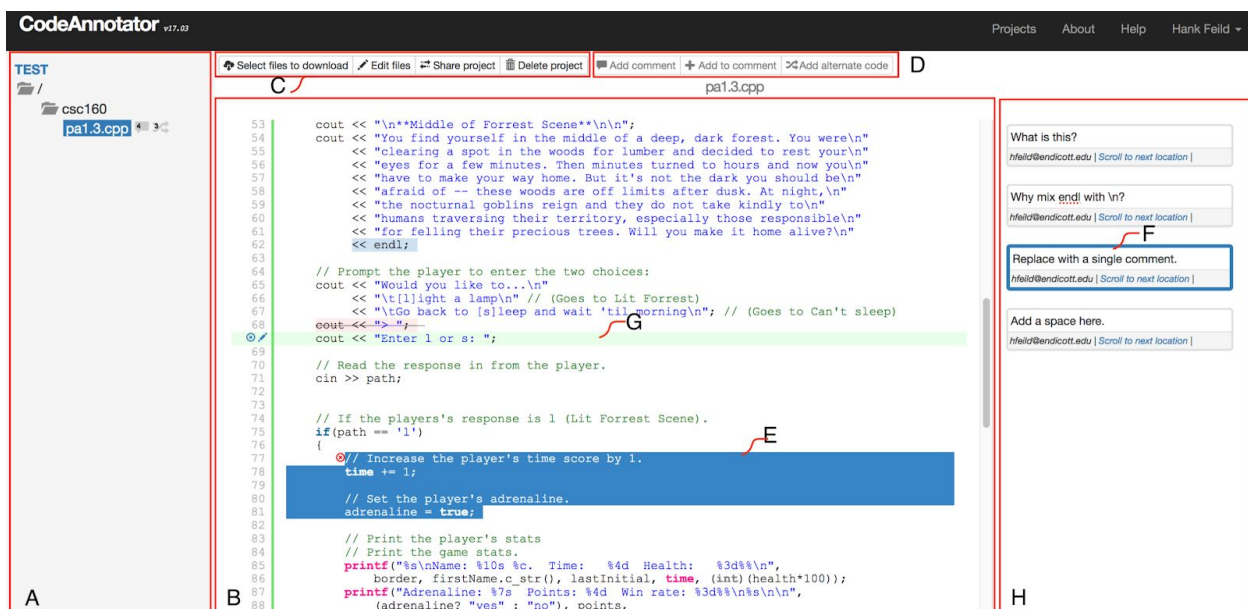


**Figure 10.** The bulk upload modal when the "Batch upload" option is checked.

## Project view

Once a project is clicked, the view changes to the project view (Figure 8C). The project view allows files to be added and modified (Figure 8C and D), annotations to be added (Figure 11E, F, and G), and projects to be shared (Figure 12). The file manipulation supported are: add new files, add new directories, move files between directories, rename files, download files, and

remove files and directories. Adding files and directories is done through the file navigation pane (Figure 11A); hovering over an existing directory name (the root directory, "/", is always present) brings up a modal box, as shown in Figure 8D. To add a directory, the user must fill in the "Directory" text field. To add files, the user must select one or more files to upload, including Zip files (with directory structure kept in tact). To download files, the "Select files to download" from the project management bar (Figure 11C), which then exposes checkboxes next to each file and directory name in the file navigation pane and a "Download selected files" button to appear in the project management bar—this must be pressed to after selected files to initiate the download. Selecting a single file results in a plain text download. Selecting multiple files or a directory downloads a Zip file named "project.zip". The "Edit files" button in the project management bar causes a rename button and move handle to appear next to each directory and file in the file navigation pane and the "Edit files" button is converted to a "Done editing files" button. Pressing the latter exits edit mode.



**Figure 11.** The project view with file navigation pane (A), file pane (B), comments pane (H), project management bar (C) and source code annotation bar (D). Also shown are code comments (E and F), and alternative code (G).

Clicking on a filename in the file navigation pane selects the file and makes it appear in the file pane (Figure 11B). Syntax highlighting is applied based on the file extension. A total of 25 common programming languages are supported; all non-supported files are displayed without syntax highlighting. Line numbers are displayed to the left of each line and vertical and horizontal scrolling is provided as needed. No line wrapping is performed, thus preserving the original formatting. Comments associated with the file are displayed in the comments pane to the right (Figure 11H).

### **Annotating files**

To annotate a selection of code or text, the user must first make the selection by pressing down on the left mouse button where the highlighting should start, dragging to where it should end, and then lifting up on the mouse (like in most other applications). Once the code is highlighted, a number of options become available in the annotation bar (Figure 11D): "Add comment" and "Add to comment". The former will create a new comment box (Figure 11F) in the comments pane (Figure 11H). The comment text can be added and is automatically saved after a few seconds (unsaved comment text appears red, then briefly flashes green when saved before finally turning black). The latter option will trigger a modal to appear with all of the comments made on any file in the project. The user selects a comment from the list to associate with the selected text. In this way, a comment may be associated with multiple locations within and across multiple files. This is helpful when pointing out an issue that is repeated several times (e.g., improper indentation or a misspelled variable). Clicking on a comment location in the file pane will highlight both the location and the comment in the comment pane. If multiple comment locations overlap, subsequent clicks on the text will rotate through the constituent locations and highlight the corresponding location and comment accordingly. To see the comment location

that corresponds to a given comment in the comment pane, the user can click the "Scroll to next location" link. Subsequent clicks will highlight each of the locations in the current file in turn, scrolling the file pane as necessary. Comments and comment locations may be deleted. A comment with no existing locations is automatically deleted.

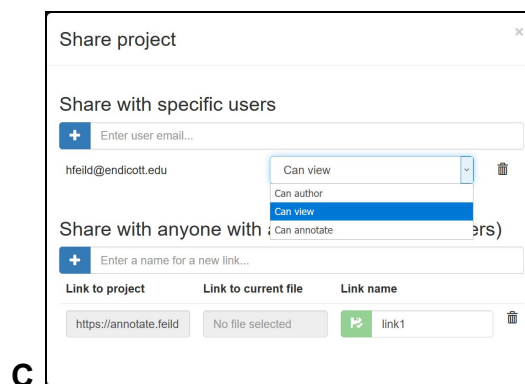
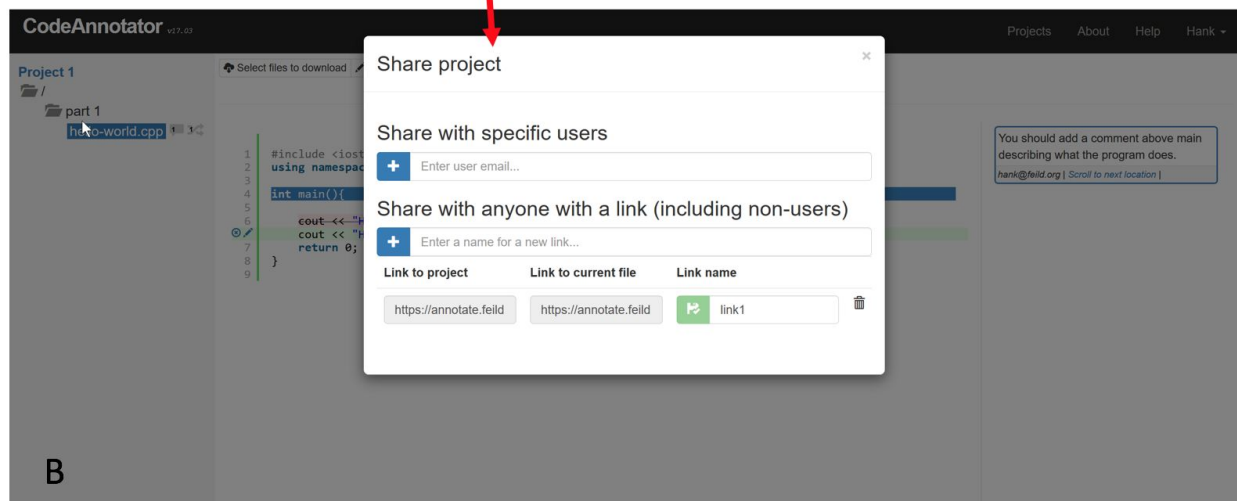
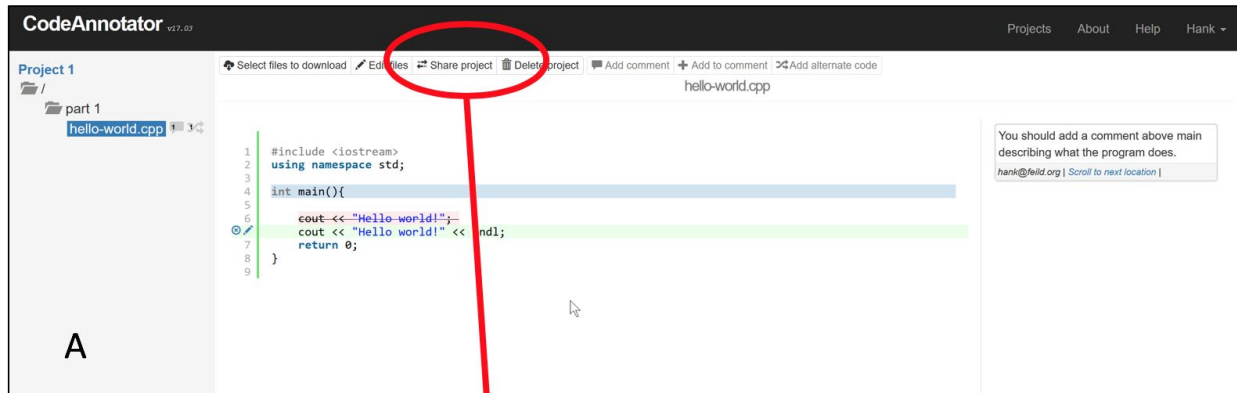
To add alternative code (e.g., to provide a correction), the user must highlight a segment of code or text just as they would to add a comment, then press the "Add alternate code" button in the annotation bar. This will trigger a modal to appear in which the user may provide one or more lines of alternative code. Once submitted, the original code will be highlighted in red with a strikethrough, and the alternate code will appear highlighted in green (Figure 11G). An alternate code annotation may be edited or deleted altogether via buttons that appear next to the annotation.

## **Sharing**

As mentioned, projects may be shared with other users for viewing, annotating, or authoring, or may be shared publicly for viewing. To manage these options, the user may select the "Share project" button in the project management bar (Figure 12A). This causes a modal to appear (Figure 12B). To share with a user, the user's email address must be entered. The permission level for the user may then be selected (middle of Figure 12C). Permissions may be revoked at any time.

To share a link that someone without an account may use to view a project, the user must supply a name for the link (this can be updated later). More than one may be created so that each may be distributed to separate individuals or groups and then revoked later if required. The "Link to project" box contains the link to the project with no files opened. To send a link with the

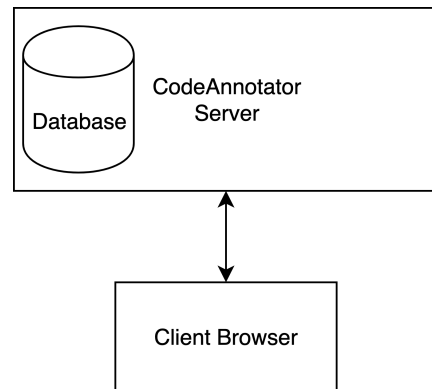
currently displayed file automatically loaded, the user can copy the "Link to current file". This still gives those with the link access to the whole project.



**Figure 12.** How to share a project with another user or via a link.

## 5. System Architecture and Design

I designed CodeAnnotator as a web application following a classic client-server architecture with a single server and a single on-server database (Figure 13). I expect this configuration is well suited to my intended usage case: thousands of simultaneous users. Keeping the server and database centralized eliminates complicated load management and database access. To support increased loads, such management would be necessary.



**Figure 13.** The basic architecture of CodeAnnotator.

The server side is implemented using the Ruby on Rails (Rails) web application framework.<sup>9</sup> Rails is a powerful and fairly simple to use model-view-controller (MVC) framework with representational state transfer (REST)<sup>10</sup> support. There are several other MVC or MVC-like frameworks that I could have employed instead of Rails, such as Laravel<sup>11</sup> (PHP), Django<sup>12</sup> (Python), or Sails<sup>13</sup> (JavaScript); any of these would have been reasonable substitutes.

---

<sup>9</sup> Rails v4.2.2: <https://rubyonrails.org>

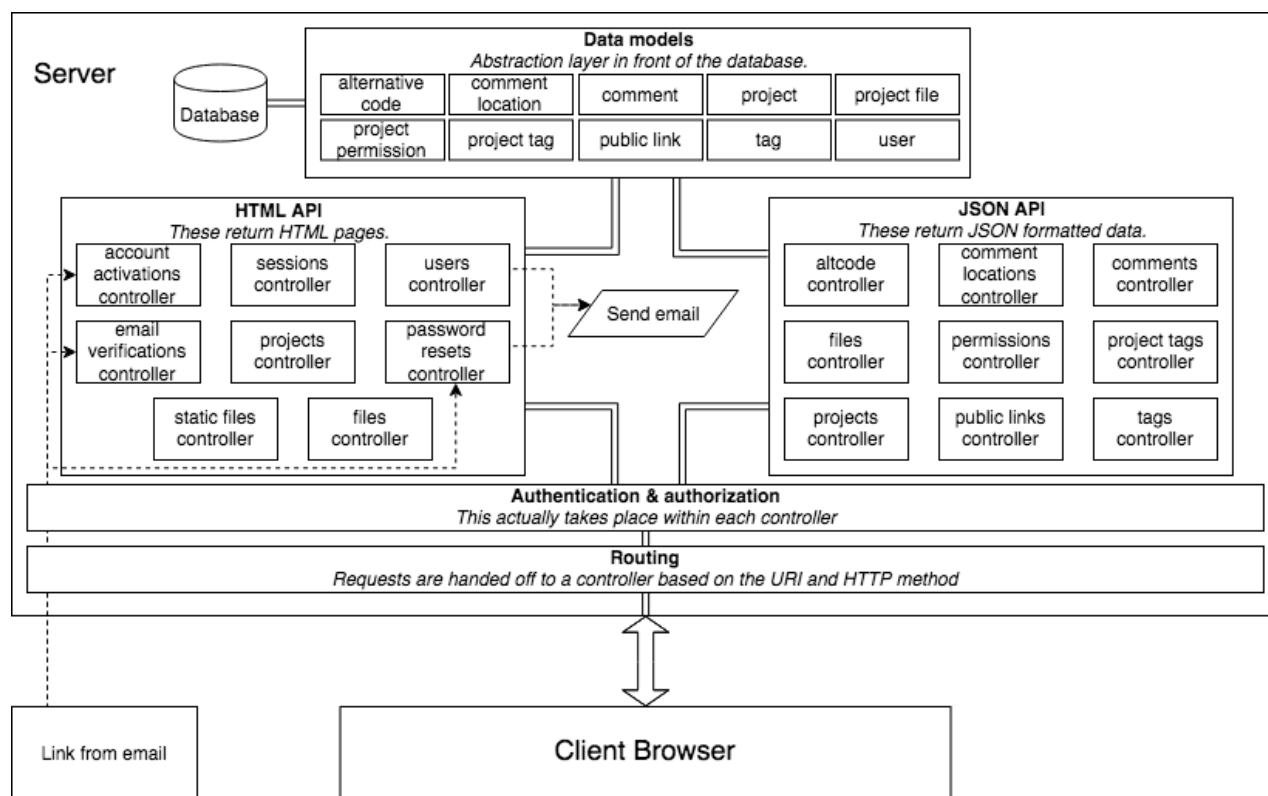
<sup>10</sup> Fielding, Roy Thomas (2000). "Chapter 5: Representational State Transfer (REST)". Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine.

<sup>11</sup> Laravel: <https://laravel.com/>

<sup>12</sup> Django: <https://www.djangoproject.com>

<sup>13</sup> Sails: <https://sailsjs.com/>

I decided to use a framework over implementing everything from scratch to avoid scalability issues, especially related to interfacing with the data layer. As part of MVC, Rails offers partially automated creation of model classes over database tables. Rails automatically takes care of updating the database based on changes to a given model. In addition, representing relationships such as "has and belongs to many" between models requires a single line of code in Rails, rather than several lengthy and bug-prone functions if I were to implement it from scratch. The models and controllers in Rails allow for easy authentication and authorization checking as well as data verification. Rails allows partial view templates to be created and pieced together and supports multiple output types, including HTML and JSON.



**Figure 14.** A more detailed diagram of the major logical components on the server side of CodeAnnotator.

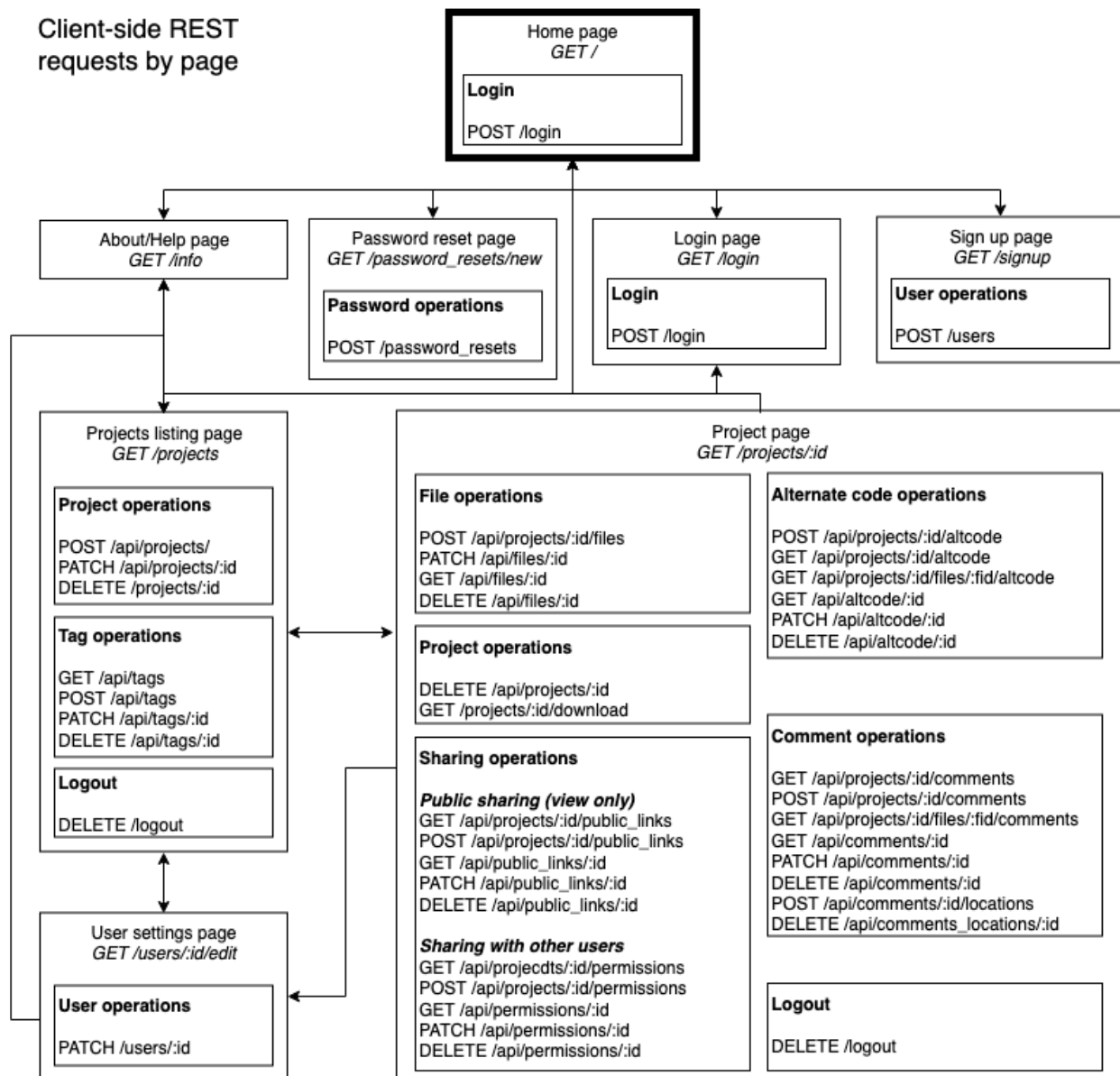
Figure 14 shows the major components broken down into models and controllers. Controllers are further divided between those that produce web pages (the HTML application

programming interface (API) on the left) and those that produce data for behind-the-scenes requests (the JSON API on the right). The latter are accessed from client side code (Figure 15) through asynchronous JavaScript and XML (AJAX) requests, though the data format is in JavaScript Object Notation (JSON). Figure 14 also provides some information on how requests are served. It shows that all requests are routed to the proper controller; this is based on the universal resource identifier (URI) (e.g., `/api/projects`) and the HTTP method (e.g., POST). It is important to note that URIs that begin in `/api` are routed to the JSON API while those that do not are routed to the HTML API. Each controller takes care of authentication (is a user who they say they are?) and authorization (is a user allowed to do what they are trying to do?), though a few controllers do not require such checks, namely the static files and the public links controllers.

In the HTML API, most requests come from links on the CodeAnnotator website. However, there are three controllers that handle links that originate from emails sent out by the server. The first is the account activation controller. This handles links generated by the users controller when a new user account is created. It relies on a randomly generated, one-time-use token, which is embedded in the URL, and as such, does not require that the user enter their credentials for authentication and authorization. The second is the email verification controller. This handles links generated by the users controller when an existing user changes their email address while logged in. It works similarly to the account activation controller in that a token is used for authentication and authorization. The third is the password resets controller, which both generates emails and handles the embedded URLs. When a user requests a password reset, presumably because they forgot their password, an email is sent with a link to a change password page served by the password resets controller. The link includes a one-use token and expires after two hours. Since the email is tied to an account, a malicious user has no way of



using their own email address to change the password of another account; they would have to compromise the target user's email account to take advantage of the system (in which case, the target user has much larger problems).



**Figure 15.** The major components of each CodeAnnotator page and the REST requests involved. Page links are shown with arrows. The home page is shown with a bolded border.

The other takeaways from Figure 14 are that there are many models (ten total) and almost as many JSON controllers (more than HTML controllers). This demonstrates the complexity of the system and why using a framework such as Rails was warranted.

The client-side relies on JavaScript and jQuery<sup>14</sup> for programming and Bootstrap<sup>15</sup> for styling and user experience.

Figure 15 shows all the REST requests made by the client side and the page of the CodeAnnotator from which the request is made. These are further organized into logical groupings, such as file operations and logout. All users, whether registered or not, start at the root page at the top (indicated by a bolded border). From there, a number of static pages can be requested (the next layer of the diagram), from which only simple REST calls are made to the HTML API.

Once logged in, the projects listing (GET /projects) and project (GET /projects/:id) pages show where the bulk of the requests are made. The projects listing page uses AJAX to create and manipulate projects and create and apply tags to those projects. The projects page handles file management, project permissions, file viewing, and annotations through comments and alternative code. All of this is done through AJAX requests rather than whole page requests as the other would substantially limit the desktop-feel of the user interface.

Tables 1 and 2 show details about the RESTful interface, including the URI, method, parameters, and, for the JSON API, the values returned.

URI	Request Method	Parameters	Description
/, /home	GET		Returns the homepage (Fig. 6A)
/signup	GET		Returns the signup page (Fig. 6B)

---

<sup>14</sup> jQuery v1: <http://jquery.com/>

<sup>15</sup> Bootstrap v3: <https://getbootstrap.com/docs/3.3/>

/users	POST	name, email, password, password_confirmation	Creates a new user
/users/:id	PATCH		Updates user
	DELETE		Removes user
/users/:id/edit	GET		Returns a user's settings page (Fig. 7B)
/login	GET		Returns login page
	POST	email, password	Logs a user in once authenticated
/logout	DELETE		Logs a user out
/projects	GET		Returns the projects page (Fig. 7A)
	POST	name, batch, update, files	Creates a new project, or, if 'update' is checked, updates existing
/projects/:id	GET		Returns the project page (Fig. 8C)
/projects/:id/files	POST	files, directory_id (optional)	Adds the provided file(s) to the project, optionally within a specific folder within the project
/projects/:id/download	GET		Returns a Zip file of the files in the project
/projects/public/:link_uuid	GET		Returns the the project page (Fig. 8C) corresponding to the :link_uuid
/projects/public/:link_uuid/download	GET		Returns a Zip file of the files in the project corresponding to the :link_uuid
/email_verifications/edit/:id	GET	email	Verifies a new email address, returns projects page (Fig. 7A)
/account_activations/edit/:id	GET	email	Activates an account, returns projects page (Fig. 7A)
/password_resets/new	GET		Returns password reset form
/password_resets	POST	email	Creates a new password reset token and emails the user a link to a password reset page
/password_resets/:id/edit	GET		Returns a change password form for the user
/password_resets/:id	POST	password, password_confirmation	Changes the user's password, redirects to projects page (Fig. 7A)

**Table 1.** The CodeAnnotator HTML REST interface—returns/redirects to CodeAnnotator pages. Resource identifier placeholders are indicated by a leading ':':

Endpoint	Request Method	Parameters	Returned JSON	Description
/users	POST	name, email, password, password_confirmation		Creates a new user
/users/:id	PUT			Updates user
	DELETE			Removes user
/sessions	POST	email, password		Logs a user in once authenticated
/sessions/:id	DELETE			Logs a user out
/api/projects	GET		projects: (list of projects) id name can_view can_author can_annotate	Returns metadata for every project that belongs to the logged in user.
	POST	name, files, batch, update		
/api/projects/:id	PATCH	name		
	GET			Returns metadata about the project.
	DELETE			Removes the project.
/api/projects/:pid/tags	GET			Returns all tags associated with the project with id :pid.
	POST	...		Creates a new tag and associates it with the project with id :pid; returns the id of the newly created tag.
/api/projects/:pid/tags/:tid	POST			Adds the existing tag with id :tid to the project with id :pid.
	DELETE			Disassociates the tag with id :tid from the project with id :pid.
/api/tags	GET		id (integer), text (string), projects (array of objects): id (integer), creator_email (string), created_on (timestamp), name (string)	Returns all tags for the logged in user.
	POST	text -- the tag text		Adds a new tag.
/api/tags/:id	GET			
	PATCH	text -- the tag text		Updates the tag with id

				:id with the provided text.
	DELETE		success (boolean) error (string, only if there's an error)	Deletes the given tag.

**Table 2.** The CodeAnnotator JSON REST API endpoint descriptions. Resource identifier placeholders are indicated by a leading ':':

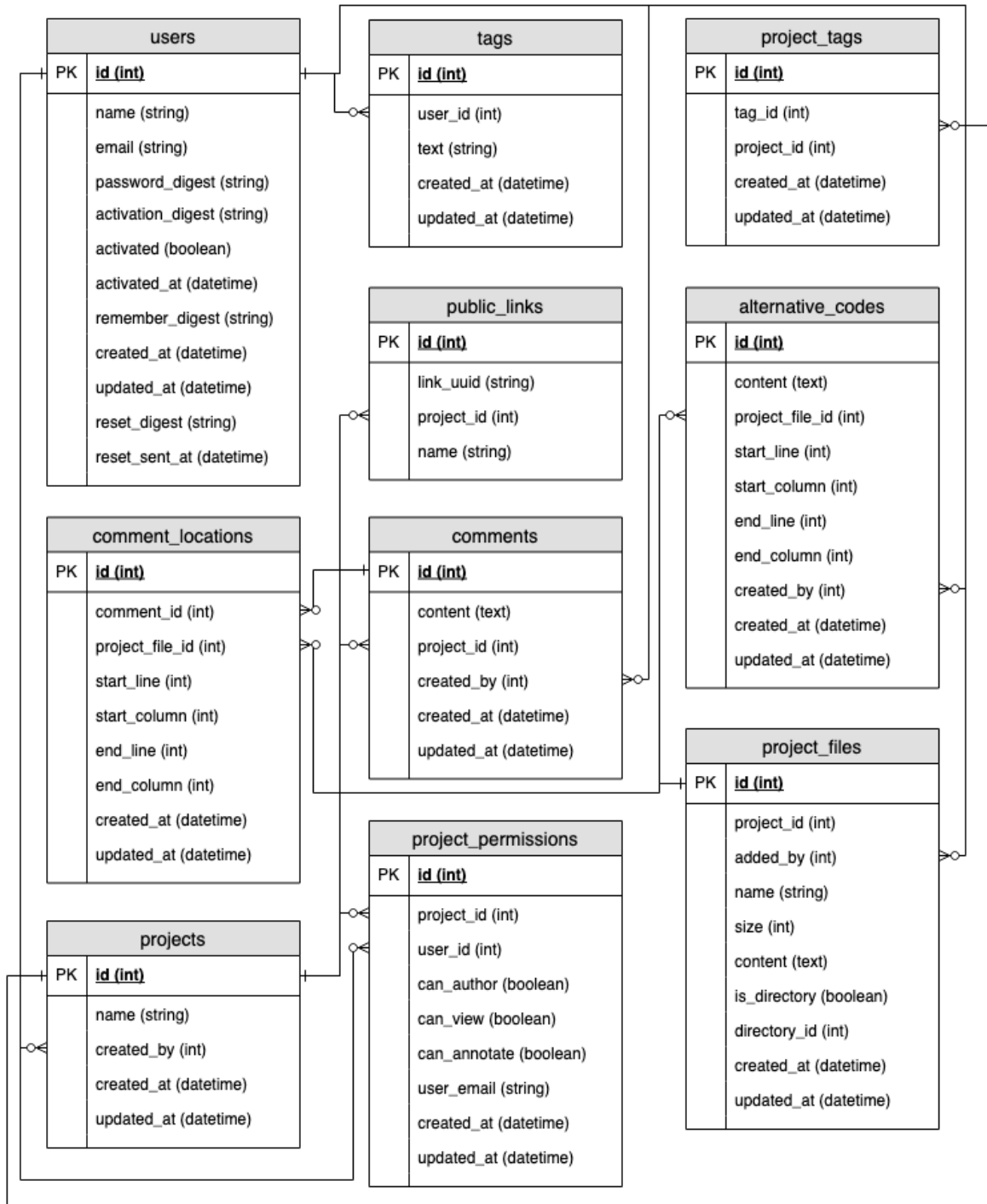
## 6. Data Design

CodeAnnotator uses a relational database on the server to store all data, as shown in Figure 14. Specifically, a PostgreSQL database. A relational database is more appropriate than a NoSQL database for CodeAnnotator for a few reasons. First, the data is easily represented relationally, as depicted in Figure 17. Rails' has support for many complicated model relationships, such as has-a, belongs-to, and has-and-belongs-to-many, and searches and aggregations over those relations. Second, the data is highly interconnected, making a document-based representation difficult to design.

The tables can be divided roughly into these categories: users (`users`), tags (`tags` and `project_tags`), projects (`projects` and `project_files`), annotations (`comments`, `comment_locations`, and `alternative_codes`), and sharing (`public_links` and `project_permissions`). The `users` table keeps track of everything about a user, including tokens for account activation and password resets. It is also the most connected table: every table except `comment_locations`, `public_links`, and `project_tags` are associated with a user.

The tags tables are divided into `tags` and `project_tags`. The former represent a stand alone tag, which encompasses the creator and the text. A user can only see and manipulate their own tags, which is why they are associated with a user. The `project_tags` table is a join table and serves to represent which tags have been applied by the user to which project. They are associated with a user via the tag.

Projects have a name and a creator (captured in the `projects` table), as well as files (in the `project_files` table). The `projects_files` table not only associates files with a project,



**Figure 17.** The relational database schema for CodeAnnotator.

but files with each other. Namely, a file may be designated as a directory or regular file. Every file (regular or directory) must belong to a directory (i.e., another entry in the `project_files`

table). Since files may be uploaded by anyone with author permissions, the user who added a file is also tracked in the `project_files` table.

Annotations consist of comments and alternative code. Since the same comment may be applied to multiple locations within the same or across different files in a project, I split the data into the comment text and the location. The comment text is stored in the `comments` table, along with the project it belongs in and the user who created it (since comments can be added by anyone with author or annotator permissions). The locations are stored in the `comment_locations` table. They include information about the file and the starting and ending positions of the location. This allows the location to be highlighted in the browser. Alternative code is only applied at a single location, so the text of the alternative code and the location are both captured in the same table, `alternative_code`. The file and user who added it are also captured.

The final category of data is sharing. Projects can be shared two ways: specific permissions (author, annotate, view) can be granted to other CodeAnnotator users, or a public view-only link can be produced so that anyone, even those without a CodeAnnotator account, can view a project and the annotations contained within. The user-level permissions are held in the `project_permissions` table. The exact permissions are set through a number of boolean fields; someone who can author can also view and annotate. A user who can view only has the other two permissions set to false. A user who can annotate has the `can_annotate` and `can_view` fields set to true and `can_author` to false. Public links are managed in the `public_links` table and are not associated with a user.

This data *could* have been represented in a NoSQL manner; however, that would have required more instrumentation to extract and update the data within Rails, and the data is easily represented in a relational format.



## 7. System Analysis & Testing

An important part of developing a system is ensuring that it meets the requirements. We can consider this from many different perspectives, but the two I considered two: do the system components work as intended, and is the user experience (UX) comfortable and convenient.

To ensure that the components work, I wrote automated model, controller, and integration tests with in Rails. The model tests pass if the data is validated, stored and retrieved correctly. The controller tests pass if the controllers implement handle requests properly: they need to check the incoming parameters, carry out the requested operations, and return the expected response. Finally, integration tests pass if use cases can be executed end to end, e.g., creating a project, adding a file, and annotating it. This also tests that the views produced are correct.

While the integration tests consider some aspects of the views and UI, they do not render any of the pages returned by the system, which also means the JavaScript and other dynamic components of the pages are not tested. There are automated tests suites for UI testing, but not ones that are easily integrated into Rails. Instead, I relied on the automated tests described above to ensure the basics worked as expected, then did manual testing to check the UI and UX worked as expected.

In addition, I have used CodeAnnotator in classes to provide feedback to students over several semesters. By using the application to achieve the end it was developed for, I am able to test that the UI, UX, and server-side components work as expected. Though there a many small tweaks I would make, a major one is that the comments in the comment pane appear

disconnected from their locations. Including lines between locations and comments would help with this, though may be unsightly.

## 8. Conclusions

In this thesis, I described the design and implementation of a web application named CodeAnnotator for annotating code with comments and alternative code to use as feedback. The project is open source<sup>16</sup> and freely available online.<sup>17</sup>

While CodeAnnotator does meet the current set of requirements as outlined in Section 2, there are still several improvements that could be made. For example, the system does not handle concurrent modifications to a project; two people commenting or uploading files to the same project will not see the changes the other has made until they refresh the page. This could cause unexpected and confusing (to the user) situations, such as when one user deletes a file while another is annotating that same file. Synchronizing concurrent sessions by updating components modified in other sessions and presenting real-time warning banners to communicate important changes would help in this regard.

Another helpful feature would be a print view or save-as-PDF. This would allow the code and annotations to be printed out, emailed, or saved for posterity. Finally, there are always ways to improve the UX of any application, and CodeAnnotator is no exception. Future work should consider the pain points of using the application and how to address those points. For example, aligning comment locations in a file to the comments they correspond to in the comment pane.

---

<sup>16</sup> CodeAnnotator GitHub page with source code: <https://github.com/hafeild/code-annotator>

<sup>17</sup> Live CodeAnnotator instance: <https://annotate.feild.org>