Code Judge

Version 1.F

Karthik Mohan Raj
Paul Schade (Scrum Master)
Laren Spear
Kathan Vyas (Product Owner)

May 9, 2022

Contents

1 Introduction

CodeJudge is a site for Texas A&M instructors to assign automatically-judged programming problems to their students. The application is written in Ruby on Rails and makes use of a free API from `https://www.glot.io` to compile and run the code. For the 39 supported languages, the API will accept the source code passed through an HTTP request, read from stdin, and return the results. See the glot.io API documentation for more information.

The entire application lives in a Docker container, which should hopefully make porting it to a variety of hosting services easy. Currently, the application is hosted on Heroku, and is not incurring any costs. Due to the Docker container, local development is also simple to get up and running. We hope that this documentation allows you to make improvements to CodeJudge as quickly as possible.

2 Getting Started

2.1 Local Development

Because CodeJudge is in a Docker container, starting up the application is very simple. First, make sure Docker is installed and running on your machine. After cloning the repository, place the `master.key` file in the `config` directory. After that, navigate to the `codejudge` directory and initialize the container by running the shell command

```
docker-compose build
```

If this command returns an error, make sure Docker is running. This step may take a while if it's your first time building the container, but in later iterations Docker caches many of the components it needs.

Once the container is built, run the shell command

```
docker-compose run web rails db:create db:schema:load db:seed
```

This command initializes the database, loads the schema from schema.rb, then seeds the database with fake data as specified in seed.rb. You can find a comprehensive overview of the database commands in appendix .1.

Once the database has been built and seeded, run the shell command

```
docker-compose up
```

If you want to recreate the container from scratch later, run

```
docker-compose down -v
```

Environment variables are handled with the Figaro gem. You'll need to set your environment variables in the `application.yml` file. You will need to set your domain as an environment variable as well as your glot.io API key. Other variables are:

- RAILS_MASTER_KEY & SECRET_KEY_BASE: have to be set in Heroku under Settings $\rightarrow$ Config Vars
- RACK_ENV & RAILS_ENV: set to production
- RAILS_SERVE_STATIC_FILES: set to enabled
- GLOT_KEY: get by registering for the glot.io API
- DOMAIN: your app domain, e.g. appname.herokuapp.com
- REDIS_URL_SIDEKIQ: has to be the same as the REDIS_URL that is set as soon as you add the Redis Add-On to Heroku
- DISABLE_DATABASE_ENVIRONMENT_CHECK: set to 1

2.2 Deploying to Heroku

First, download and install the Heroku CLI.

Run the command

```
heroku login
```

Once you're logged in, you can log into the Heroku container registry with

```
heroku container:login
```

To deploy your container, you must be in the directory with your Docker image have already built the image. If it is the first time deploying to heroku you have to add the Heroku Postgres and Heroku Redis Add-On first. To push the container to Heroku, run

```
heroku container:push web sidekiq --recursive
```

Then, to deploy your changes to the production site, run

```
heroku container:release web sidekiq
```

Again, if it is the first time deploying, the sidekiq dyno should now show up under the Resources tab on Heroku. If so, go ahead and enable it.

Since the Heroku database is the production database you do not have to create it first. All you have to do after you pushed the container is running migrations on the production database. If it is the first time, do

```
heroku run rails db:schema:load db:seed
```

For future deployment, just do

```
heroku run rails db:migrate
```

And you're done!

3 Explaining the Code

3.1 The Docker Image

The Docker image is already written, so you will hopefully only have to do some minor configuration. The main files that contain the Docker logic are the `Dockerfile`, and the `docker-compose.yml` files. Currently, the `docker-compose.yml` defines the 'database' image – running postgres 11.5 – and the 'web' image – which runs the puma server on `localhost:3000`. The `Dockerfile` contains the setup commands that are run on the 'web' container. You can find an overview over the most important docker-related commands in appendix .2. We encourage you to set up shorter aliases for the main commands since some of them are very long.

3.2 The Database

We have included our database diagram in Figure 1. The Lucidchart link is here. The tables mostly follow along with the classes. One important thing to remember is that an attempt has many test cases through both problems and scores. This is useful for when a problem is currently being graded, because whether a test case was passed or failed cannot be stored in the problems table.

3.3 The Classes and Methods

3.3.1 User

Only the site admin can create, update, and delete users. The login and registration page handle the creation of users. There is no user-facing account

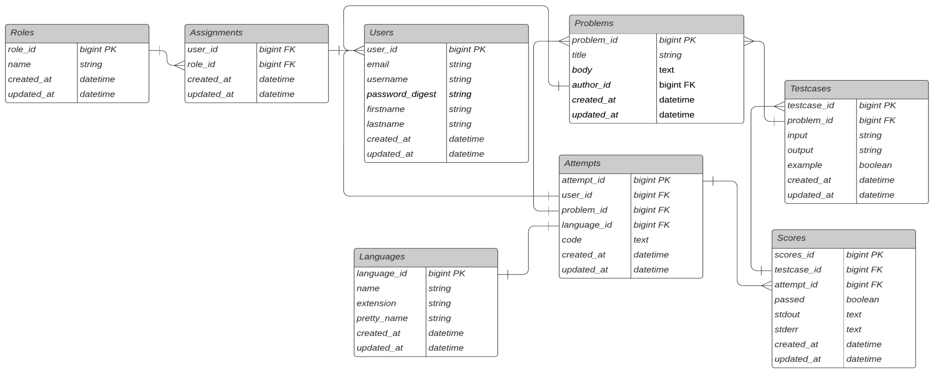


Figure 1: Illustration of the CodeJudge database

deletion functionality.

3.3.2 Problem

An instructor or TA and create, read, update, and delete a problem. Permissions are limited to the user who created the problem. Students can only read problems.

3.3.3 Attempt

A student, instructor, or TA can create an attempt, as well as read their own attempts. Instructors and TAs have the ability to read the attempt of any student, but not any other TAs or instructors. Attempts cannot be updated or deleted except by the site admin. Updates are meant to be immutable records of code submission.

3.3.4 Testcase

The author of a problem (the instructor or TA) can create any number of test cases for that problem, and has the ability to set them as visible or invisible to students. Instructors can see all of the test cases for every problem, including those created by others, but cannot update or delete them unless they are the author of that problem.

3.3.5 Language

The language list is populated from the glot.io API and manually seeded. However, the administrator still has the power to add and remove languages. In order for Glot to run, the file extension has to be specified, which is stored in the database. Some languages, like Python and Ruby, will happily run whatever file you serve them regardless of extension. That is not the case for languages like C++.

3.3.6 Score

When an attempt is created, the test cases are packaged up and sent off to glot.io to grade. When they come back, their output is compared to the expected output, and the results (pass or fail) are logged in Score, along with the stdout and stderr from the test case. This leaves open the future ability to display this information to students rather than just a binary pass/fail. There is otherwise no direct access to the scores table, nor should there need to be.

3.4 Security Policies

Security in CodeJudge is handled through the Pundit Ruby gem. Pundit allows you to set permissions per method that is called on an object. This is contained in the `app/policies` folder and comes with a generator, that helps setting up the policies. More information can be found in the Pundit documentation.

The corresponding files that define the policies can be found in the `app/policies` folder. A policy acts like a middleware when a controller method is called. Therefore, a policy defines the same methods like a controller (index, new, create, etc.) but every method just returns a boolean value whether the user is allowed to access that resource or not. Usually you want to check the role of the currently logged-in user. To invoke the policy-check all you have to do is call the authorize method in your controller method. It automatically checks the corresponding policy and returns from the controller method if the policy returns false. As of right now, only basic user role checks are included in the policies. Users may have multiple roles. We leave it to future developers to exploit this functionality.

Pundit also makes it easy to show or hide certain UI elements depending on the defined policy. All you have to do is wrapping the UI element in this code snippet (example relates to the problem policy)

```
<% if policy(:problem) %>
    ...
<% end %>
```

If you want to check for special policy methods you can do

```
<% if policy(:problem).<<method-name>>? %>
    ...
<% end %>
```

3.5 Testing

Testing is handled through standard RSpec as well as Cucumber, which is a testing framework for Ruby on Rails that interacts with the browser in the same way a user would.

As of now, we have defined a few Cucumber feature files and some step definitions. You can run the cucumber tests using the following command (NOTE: there may be an error message from SampleCov after you run this but it can be ignored, we are not sure how to fix it)

```
docker-compose run web cucumber
```

We added the SampleCov metric to our repository but it was a hassle to work with on the Docker container. To learn more about this, you can read the documentation on codecov's website. The workaround we made to actually get results was: running the above command, then downloading and running the codecov uploader tool on your local machine.

3.6 Styling

For the styling we used the SCSS precompiler that allows nesting CSS rules. On top of that, we implemented a style system that makes it easy to theme the application. The themes can be found in the `app/assets/stylesheets/themes` folder. A theme is essentially a SCSS map with the attribute identifiers as keys and the corresponding colors as values. So lets say you have a SCSS rule for a button where you want to specify the background-color attribute. To make it themeable, instead of writing

```
.button {
    background-color: #500000;
}
```

you should write

```
.button {
    @include theme(background-color, button-background);
}
```

Then, in your theme file you would have a key-value pair like this

```
$light: (
    button-background: color(maroon);
);
```

The `color()` function has to be imported in every theme. It grabs the specified color from the color definition in `app/assets/stylesheets/themes/_colors.scss`. The `theme()` mixin in `app/assets/stylesheets/themes/_mixins.scss` takes care of the style generation. All you have to do now is set a class, named after your theme, on the body element of the page. So if you named your theme `light` then you would have to put the `.light` class on the body element. Besides that, you write SCSS as usual. A light theme is already included. Go ahead and look at it for reference. If you apply this technique in all your SCSS files you are able to introduce new themes easily by just defining a new theme file and importing it in `app/assets/stylesheets/_colors.scss`.

The `app/assets/stylesheets/application.scss` file is the main SCSS file that imports all other SCSS files, gets compiled to CSS, and shipped to the client. If you want to include third-party stylesheets, import them in there.

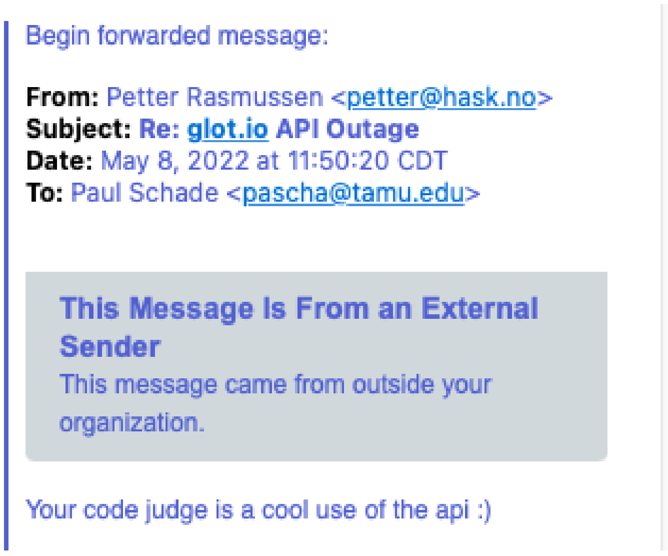
3.7 Glot.io

The grading portion of the app is covered by glot.io, created by Petter Rasmussen. When the API went down, he very graciously fixed it within 30 minutes. If Glot goes down again, you should email him. This is his hobby project, so try to be nice. The good news is he likes CodeJudge and even registered for the site (Figure 2)!

In the future, it would be a great idea to use his provided Docker containers for each language to run the code so we aren't flooding his API. However, that will be quite the undertaking, and we didn't have time for it this semester.

3.8 Google Sign In

The Google Sign In allows users to use their existing Google account to register and log into our application. For users that registered this way we do not store a password digest. To set up the Google Sign In, create a Google Account, and go to Google Cloud Console. You can follow this Tutorial to obtain your keys. You will get a *client_id* and a *client_secret* which you will have to put in the `credentials.yml` file. You will find the command for that in Appendix .2.


Begin forwarded message:

From: Petter Rasmussen <petter@hask.no>
Subject: **Re: glot.io API Outage**
Date: May 8, 2022 at 11:50:20 CDT
To: Paul Schade <pascha@tamu.edu>

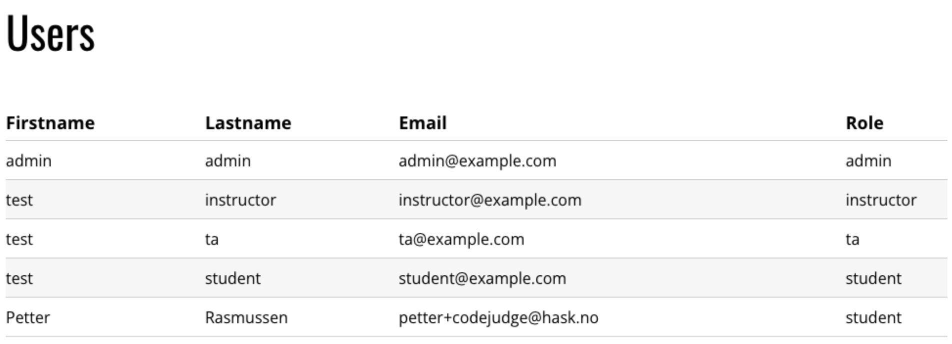
This Message Is From an External Sender
This message came from outside your organization.

Your code judge is a cool use of the api :)

Figure 2: Message from the creator of the glot.io API

4 User Stories

In this section, we will describe all the user stories we designed for this project and explain any changes we made. Section 4.1 details the completed user stories that covered the application's features – more detail of the implementation is given in section 5. Section 4.2 covers the completed story points for the project presentation, demo, and report. Finally, section 4.3 explains the user stories that were left unprioritized and uncompleted due to time constraints.


Users

Firstname	Lastname	Email	Role
admin	admin	admin@example.com	admin
test	instructor	instructor@example.com	instructor
test	ta	ta@example.com	ta
test	student	student@example.com	student
Petter	Rasmussen	petter+codejudge@hask.no	student

Figure 3: Portion of the Users table in the database

4.1 Completed user stories

1. Log in as instructor (1 point):
   This was the first user story we completed and required a user table and log-in logic.

2. Show list of problems (1 point):
   This user story required a problems table and application logic to query the table and present it to the user.

3. Add a problem as an instructor (1 point):
   This user story was changed to 'add, edit, and remove ...' in iteration 2. However, we made a lo-fi UI mockup for this in iteration 0 and it is shown in Figure 4.

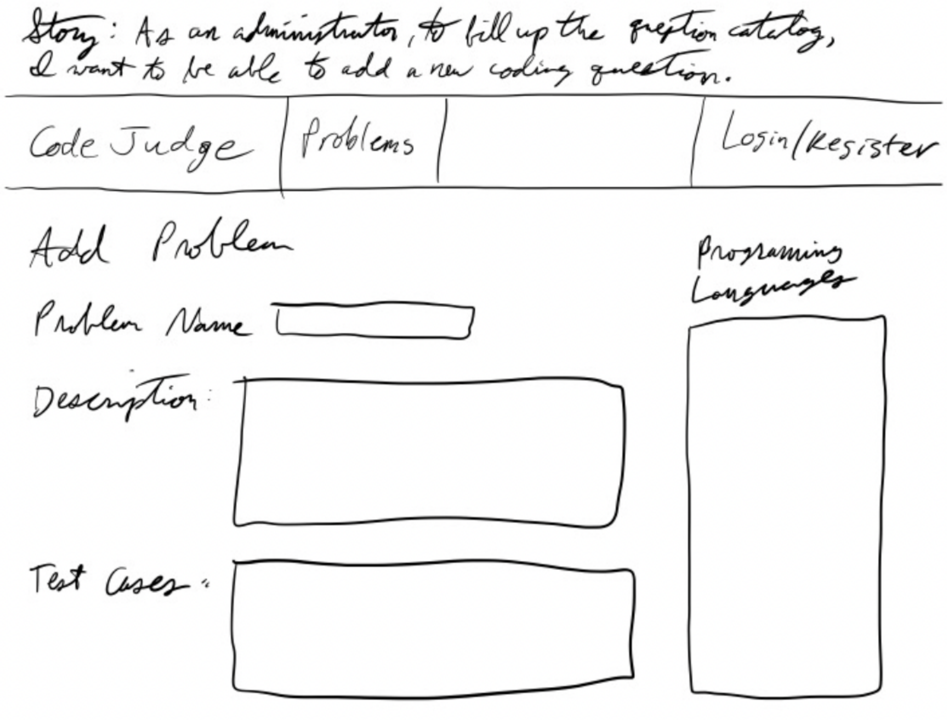

Story: As an administrator, to fill up the question catalog, I want to be able to add a new coding question.
Code Judge
Problems
Login/Register
Add Problem
Problem Name
Description:
Test Cases:
Programming Languages

Figure 4: Mockup of the 'Add a problem as instructor' story

4. Log in as a user (1 point):
   This user story required a role table and proper role assignment.

5. Role authentication (1 point):
   We added this user story in iteration 3 to ensure all privileges are as-

signed and checked correctly. For example, students cannot add, edit, or delete problems and instructors cannot see the users page.

6. Account creation (1 point):
   This user story was added in iteration 4 and covered sign-up logic. Users can sign-up using the provided form or with their Google account and users are assigned the student role by default.

7. Submitting code (3 points):
   A mockup of this user story is shown in Figure 5.

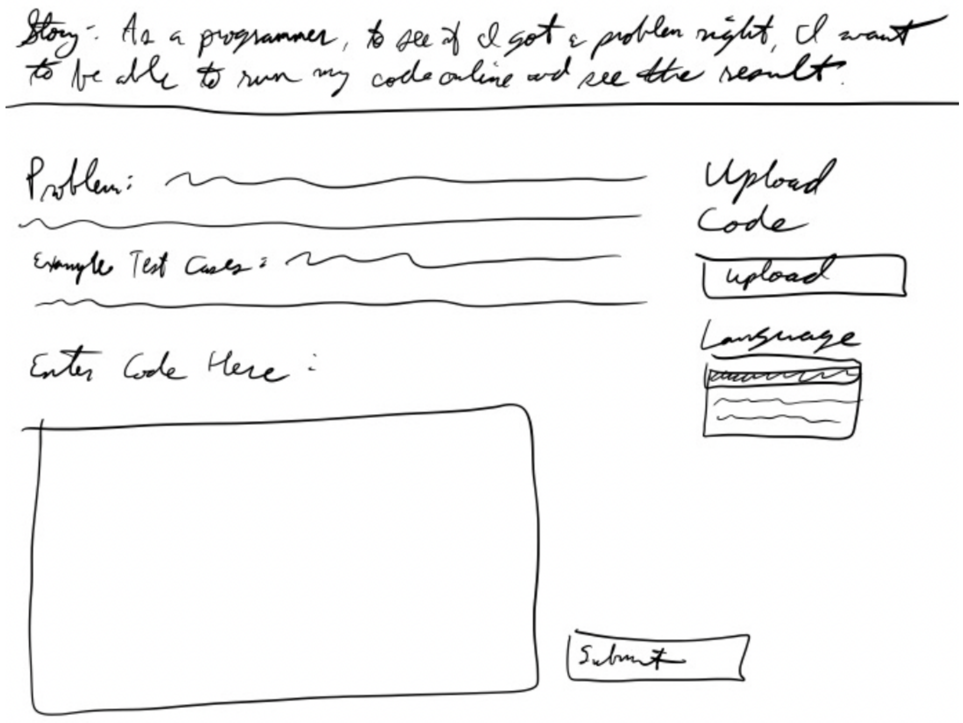

Story: As a programmer, to see if I got a problem right, I want to be able to run my code online and see the result.
Problem:
Example Test Cases:
Enter Code Here:
Upload Code
upload
Language
Submit

Figure 5: Mockup of the ‘Submitting and running student code’ story

8. Integration with the glot.io API (3 points):
   We spent a lot of time extracting information from the file and making sure glot could handle it.

9. Adding test cases to problems (3 points):
   This user story was added in iteration 4 and required a testcases table. It is also illustrated in Figure 4.

10. View past submissions (2 points):
    We also added this user story in iteration 4 and it required an attempts table as well as some changes to the problem view.
11. Show results of test cases in real time as they come in (2 points):
    This uses messaging via Javascript to update the cards in real time.
12. Code syntax highlighting (1 point):
    Using the Pygments library, we were able to provide syntax highlighted code when viewing an attempt
13. Display list of allowed languages (2 points):
    This came from seeding the database with the languages that glot.io supports.
14. Dark mode (1 point):
    Paul quickly added this while we waited for Petter to fix glot.io.

4.2 Completed story points

The following stories were assigned 2 points each since they are big parts of the project and have all been completed.

1. Presentation:
   This covered the slide deck and the 6 and a half minute video presentation (link to video in appendix .3).
2. Demo:
   This story covered the 6 and a half minute video demo walking through major use cases of our application (link to video in appendix .3).
3. Report:
   The current document which gives a overview of the project for a team that follows on from our application.

4.3 Iceboxed user stories

This section explains the user stories that were not prioritized or assigned points. These stories were created in the first couple iterations and were later iceboxed due to the fast-approaching deadline (see section 5). This could be used as inspiration for additional features by a following team.

1. Tag system for coding questions:
   A mockup of this story is given in Figure 6. Instructors can add tags to a problem, and users can sort the problems list by these tags.

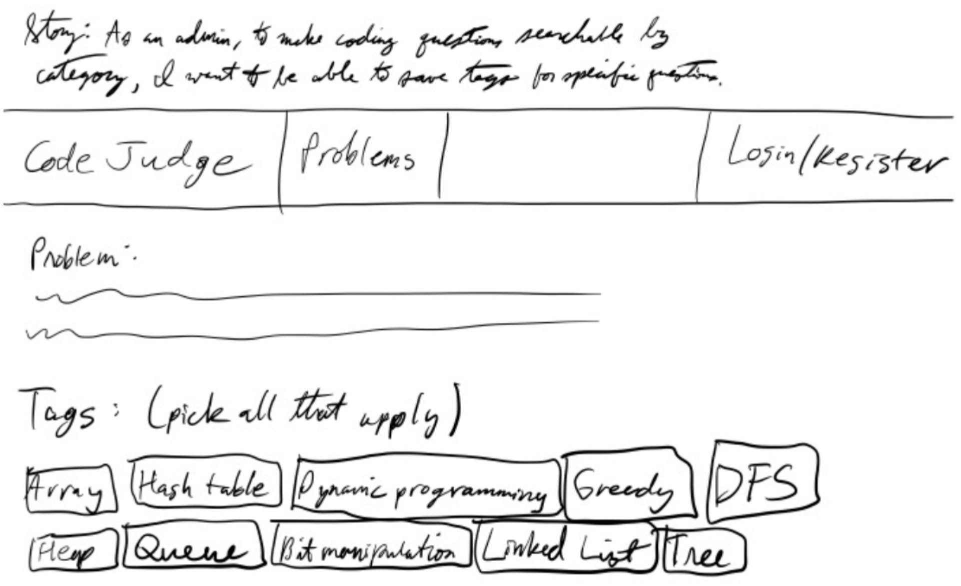

Story: As an admin, to make coding questions searchable by category, I want to be able to save tags for specific questions.
Code Judge
Problems
Login/Register
Problem:
Tags: (pick all that apply)
Array
Hash table
Dynamic programming
Greedy
DFS
Heap
Queue
Bit manipulation
Linked List
Tree

Figure 6: Mockup of the 'Tag system' story

2. Points leaderboard:
   This user story would require logic to rank users and show a sorted list either per problem or using an overall score.

3. Problem suggestion by students:
   This would require completely new logic to add problems but could be a useful feature for students.

4. Profile page:
   To see attempt history, rankings, and other information.

5. Allow instructor to add other instructors:
   We thought about using a specific sign up link or code to assign different roles at account creation.

6. Class groups:
   This user story covers instructors using the application to assign coding problems to a specific set of students (i.e., it is only visible to the users in the group). We also thought about allowing instructors to add other instructors and TAs to a class group.

5 Iterations & Meetings

The following section lists the meeting summaries as well as a breakdown of what was completed in each iteration of the project.

5.1 Iteration 0 + Meeting 0

In this initial iteration, we had a quick meeting with Dr. Ritchey to get an idea of what the requirements were. After this, we created 10 user stories (stories 1-4,7 in section 4.1 and stories 1-5 in section 4.3) which were added to the Pivotal Tracker and set up the GitHub repository.

5.2 Iteration 1

We attempted to complete the first three user stories in section 4.1; however, we ran into issues with running the application on different machines. Thus, we spent the rest of the iteration Dockerizing the application (which also came with issues with connecting to the database and deploying to Heroku). We also added a cucumber feature file to test the 'Log in as instructor' user story. Due to spring break, we couldn't get much user-facing work done and we didn't have a meeting with the client this iteration.

5.3 Iteration 2 + Meeting 1

We met Dr. Ritchey for the first meeting on March 23 2022 for our first meeting. We discussed the features of the product and understand the customer requirements. The initial database schema was discussed and inputs from the customer were taken into consideration for the change in schema.We demonstrated the user stories and featrues we will be covering in the second iteration and displayed a short demo of what we covered in the first iteration. In second iteration, we completed the first three user stories in section 4.1. The database was seeded with an initial instructor user and feedback was provided at login for success and failure. We also added full CRUD functionality the problems. However, at this point, the application did not save the logged in user and all pages could be navigated to without authentication. We added cucumber feature files for the other two user stories and implemented the step definitions for the first feature file. This required chrome and chrome-driver to be installed in the Docker container which took a lot of time to install due to version differences in the packages.

5.4 Iteration 3 + Meeting 2

We met with Dr. Ritchey on the 6th of April 2022 and demoed the functionality available at the end of iteration 2. He noted that, since we were getting close to the final iteration, we should focus on implementing the core functionality of the application. After this meeting, we decided to move the stories in section 4.3 to the icebox and focused on making sure the roles were checked properly. As such, we completed story 4 as well as creating and completing story 5 in section 4.1. We also rewrote our database from a `structure.sql` to a `schema.rb` with migrations due to issues we had with updating the production database. Additionally, we set up a theming system in SCSS that supports the traditional Texas A&M style but can easily be extended to other themes like colorblind themes, high contrast themes etc. to make the app more accessible in the future.

5.5 Iteration 4 + Meeting 3

For this iteration, we met with the client on the 19th of April 2022 and showcased our application's functionality at the end of iteration 3. Dr. Ritchey noted that we did not have a sign up page and recommended we leverage existing technologies to run student submitted code. We noticed that we didn't have user stories to cover signing up for an account or adding test cases to the problems so we updated the Pivotal Tracker. This iteration we focused on the last user stories in section 4.1. Users can sign up using the provided form or with their Google account and are assigned the student role by default; as of now, we are yet to figure out a way to limit the Google sign on to 'tamu.edu' emails or how to have instructors assign other roles. Additionally, we made some slight modifications to the styling of the page showing the list of problems and also integrated the SimpleCov metric.

After iteration 4, our app really came together as a complete product. We added code syntax highlighting, asynchronous code grading, cards with each test case on the attempt page, a dark mode, and a functional list of allowed languages with all the ones glot supports. The admin can also now change roles assigned to users.

6 Ideas for Future Features

- Support for Firefox and Safari (support is currently limited to Chrome, Edge, and Opera)
- Display stdout and stderr to users

- In-browser editor, such as Microsoft's Monaco.
- Canvas integration
- Instructors can limit the allowed languages and set custom time and memory limits
- Advent of Code-style problems, with unique answers determined based off of the user's id.
- Adding functionality for groups
- Figure out how to allow instructors to assign privileged roles
- Restrict the google sign on to 'tamu.edu' emails

Appendices

.1 Database Commands

- `db:migrate`
  runs (single) migrations that have not run yet
- `db:create`
  creates the database
- `db:drop`
  deletes the database
- `db:schema:load`
  creates tables and columns within the existing database following schema.rb. This will delete existing data
- `db:setup`
  does db:create, db:schema:load, db:seed
- `db:reset`
  does db:drop, db:setup
- `db:migrate:reset`
  does db:drop, db:create, db:migrate

.2 Docker Commands

- `docker-compose build`
  builds the container from scratch but uses caching to speed things up. Use the –no-cache flag to build entirely from scratch
- `docker-compose up`
  starts the containers
- `docker-compose down`
  stops the containers
- `docker-compose run --rm web <<command>>`
  runs command on the docker instance

The nano text editor is installed on the docker instance. If you ever need to edit the credentials.yml file on the docker instance, run the following:

```
docker-compose --rm -e EDITOR=nano web rails credentials:edit
```

.3 Project Links

- Pivotal Tracker
- Heroku Deployment
- GitHub Repository
- Presentation Link
- Demo Link