Student Code Online Review and Evaluation Developer Manual

A terminal program and web-application for use in Florida Tech's CSE department to facilitate the submission of code for professor created assignments.

Team Members:

Michael Komar - mkomar2021@my.fit.edu
Charlie Collins - ccollins2021@my.fit.edu
Logan Klaproth - lklaproth2021@my.fit.edu
Thomas Gingerelli - tgingerelli2021@my.fit.edu

Faculty Advisor / Client:

Dr. Raghuveer Mohan - rmohan@fit.edu
04/20/2025

Table of Contents

Table of contents -

Introduction

1.1 What is SCORE?

SCORE, or Student Code Online Review and Evaluation is a remote server application designed to assist students and professors at Florida Tech with the submission and grading process of code based assignments. SCORE allows professors to create courses and assignments, and for students to view assignments and submit their code solutions. The system allows professors to attach meaningful information to each test case in the testing suite, and the system returns that information to the student when they failed the test case. The main goal of SCORE is to make the code submission process not only easier, but also to create a platform that allows for more impactful feedback from professors to students.

1.2 Design intentions of SCORE

We designed SCORE with modularity in mind. To accomplish this, most of the functionalities of the application are written as python scripts. SCORE's interfaces just call these scripts as needed. The idea of developing this way is that more commands can easily be added, and existing commands can be changed, with little to no modifications to the interfaces or server itself. This is commonly referred to as model view separation, and we tried our best throughout development of this application to make things as easy to modify as possible.

1.3 Future works

As we concluded the development of this application, there were several features and ideas that we had that we unfortunately did not have time to implement.

Canvas Integration

Since SCORE is meant to be used in the classroom and has the ability to grade students' code, it seems rather obvious to include the ability to seamlessly import these grades into Canvas. While we never got to implement this feature, we did do research on it, and have several notes that may be helpful if you plan to develop this feature. 1.) Canvas is an open source application, so you should be able to create a local instance of it for development. 2.) You will somehow need to obtain the professor's api key to integrate with their canvas class. 3.) This link should contain useful information regarding the api endpoints: https://canvas.instructure.com/courses/785215.

MOSS Integration

MOSS is a winnowing algorithm used to find pairwise similarity between two codes. This algorithm could either be implemented for use with SCORE, or Stanford's MOSS server could be used. Either way, MOSS could be used to determine the similarities between submissions and report that to the professor. This could be furthered by using the pairwise similarities to cluster similar submissions to determine if code was written in a group.

Custom Test Cases

As of now, only the professor can attach test cases to an assignment, but what if a student comes up with a good test case that they would like to use? We think it would be a great idea to let students create and attach their own test cases to an assignment, which SCORE will test when that student submits. This could be furthered by allowing students to potentially share test cases, or even have "crowd sourced" test suites.

Florida Tech CAS authentication

SCORE use's google OAuth to validate users and sign them in. However, to bring SCORE more in line with how other Florida Tech applications work, there is value in swapping OAuth for Florida Tech's Central Authentication System (CAS).

AI Detection

With the rise of LLMs like ChatGPT, it has become easier than ever for a student to cheat on their coding assignments. To counteract this, we feel AI detection for the code submissions would be a valuable addition to SCORE.

Installation and Deployment

2.1 System Dependencies

Linux Environment

The CLI and Web application servers are written to be run on a linux machine. Ubuntu Server 24 was chosen for development purposes and the installation script utilizes APT and SYSTEMD for system dependency management.

Pre-Setup Requirements

- Pull the project directory from Github, placing the files in an easily accessible location.
- The following APT packages are required for the system to work properly:
    - NodeJS + NPM
    - Vite
    - Cargo + Rust

- Python3 + Pip3
- CURL
- Gnupg
- *MongoDB
- WGET
- Python3-docker
- Nginx
- Linux Snap

Please make sure to verify the installation of each dependency PRIOR to running the system. *MongoDB will not install through APT properly unless the mongodb public GPG key is imported and a list file is created.

2.2 Environment Setup

Dependency Verification

After pulling the repository from Github, verify that the above dependencies are installed by running the install.sh script located in the scripts directory. This will also install all of the required NodeJS packages for both the backend (Express) and the frontend (React). This will not however install the cargo (Rust) dependencies, as these get installed at compile and runtime for the rust server.

MongoDB Daemon

Verify that MongoDB is running in the background through:

"systemctl status mongod"

If it is not running, enable it to run on system startup and run it through:

"systemctl enable --now mongod"

Nginx Daemon

Nginx reverse proxy requires configuration of the web app domain endpoint. This can effectively be skipped if a reverse proxy is not desired, but this was chosen for simplicity and future proofing.

If Nginx is skipped, the IP address in the backend directory needs to be changed from "127.0.0.1" to "0.0.0.0" in order to listen on port 3000, which can also be modified.

Otherwise,

Verify that Nginx is running in the background through:

"systemctl status nginx"

If it is not running, enable it to run on system startup and run it through:

"systemctl enable --now nginx"

There are concise Nginx configuration guides available online, so the steps have been left out for brevity. Keep in mind that Linux Snap is installed for the purpose of streamlining the certbot HTTPS process.

2.3 Running the System

Web-App Compilation

The front-end of the web application is statically served by the backend. To handle this, in the “web_app” directory run “npm run build” in order to create a “dist” directory. This directory should then be copied into the “backend” directory. If a dist directory is already present in the “backend” directory, remove it prior to copying over the newly compiled one.

Server Startup

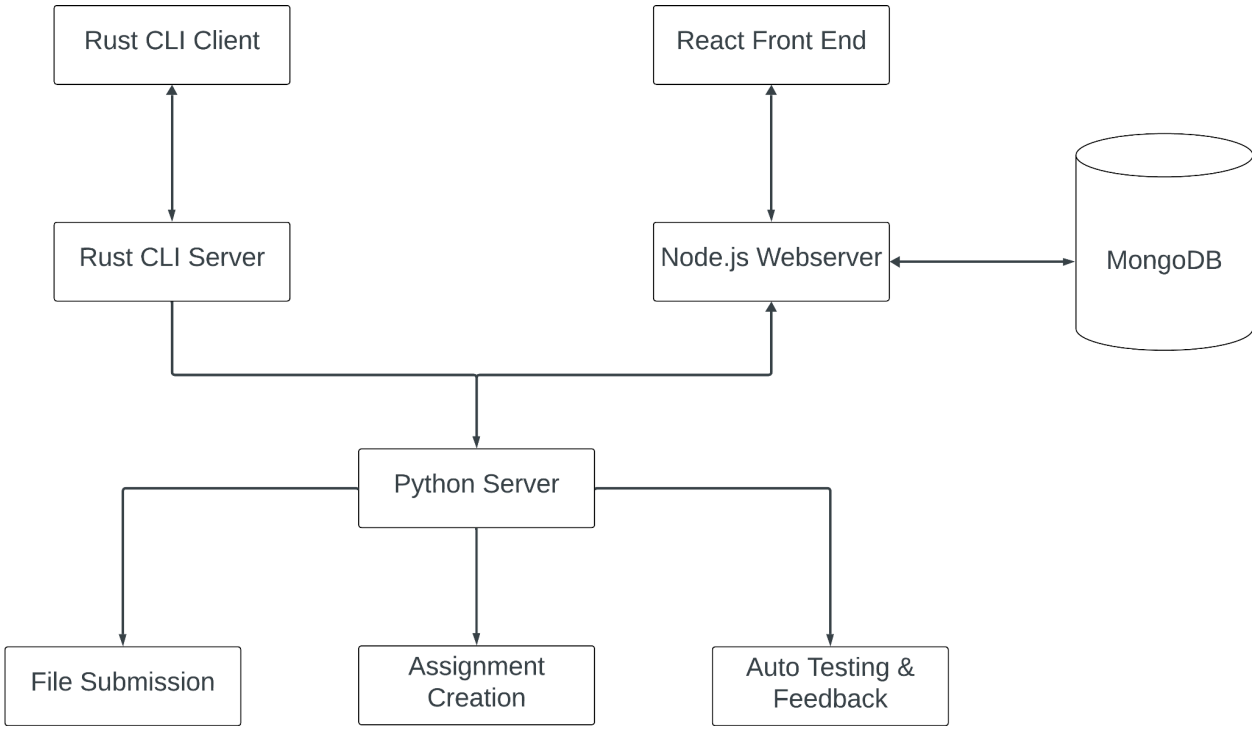
GNU Screen sessions are recommended to be used for the management of the multiple console views, but any multiplexer will do. In total, you will need three screen sessions: one for the file upload system, one for the web-app, and one for the rust server.

To startup the web-app, run “npm run test” in the “backend” directory within a multiplexer session. In order for file submissions to work, run the “ProjectScore/main.py” script in a separate multiplexer session using “python3 main.py”. Finally, to start up the rust server, use “cargo run” from within the “ProjectScore/score_server” directory.

In order to access the rust server, the score_client must be used, and for development purposes can also be run with “cargo run” from the “ProjectScore/score_client” directory, however the compiled binary located in the “ProjectScore/score_client/target” directory can be used as well.

Project Structure

3.1 System Diagram



3.2 Project File Structure

The SCORE repository contains all of the different components of the application, which we discuss later in this document. In the root of the repository, we have the following files: main.py: this is the auto testing and feedback portion of the server; RustListener.py: this manages the connection between the Rust portion of the server and Python portion; NodeListener.py: this manages the connection between the Node portion of the server and the Python portion.

Within the auto test directory, we have both auto_test_object.py, which handles the auto testing, and auto_feedback_object.py, which handles the grading of a tested submission.

The backend directory contains the Node server, which is called main.js. This also contains the packages.json for the server, and the build of the front end in the dist folder.

The web_app directory contains the React project. This consists of several components, all of which can be found within the src directory.

The score_client directory contains the front end Rust portion, and the score_server directory contains the Rust backend. The score_server directory also contains the commands directory, which houses all of the python commands, and the Classes directory which serves as the storage space for all of the classes. The basic structure of a class is an assignments directory, which will contain a directory for each assignment in that class. Each assignment directory contains an assignment description, a testcases.json, and a submissions directory. The submissions directory contains a folder for each student in the class, which contains the code that student submitted for that assignment.

3.3 Interfaces

SCORE contains two separate interfaces so that a student is free to use whichever they are more comfortable with.

3.3.1 Command Line

SCORE's command line interface is written in Rust. It consists of two directories, score_server, and score_client. They are both managed with cargo, so you must use the command *cargo run* within the directories to build and run the code. The server and client communicate with each other over a tcp connection. The Rust client is a synchronous client, so it blocks after a message is sent to the server until it receives a response. The Rust server is asynchronous and spawns a helper thread to handle each client connection.

3.3.2 Web Application

SCORE's web interface is built using the MERN (MongoDB, Express.js, React, Node.js) stack. The front end of the web app is built with React and lives within the web_app directory. To do a development build (only the front end) use the command *npm run dev.* To do a production build use the command *npm run build,* which will create a dist folder that must be copied to the backend.

The backend of the web app uses Node.js with Express.js for routing. This is housed in the main.js file in the backend directory. To run this file use the command *node main.js* from within the backend directory. This will run the app on the localhost using port 3000. This node backend is also responsible for doing all MongoDB operations, which are used throughout the application.

3.4 Auto Testing

SCORE's auto testing functionality is written in Python. Starting with the main.py file in the root directory. This python code uses a producer consumer thread synchronization pattern to manage the auto testing and feedback. To do this, it contains a mutex guarded list, which is given to the two listener files. These python files are responsible for listening for messages from the rust and node files. When they are sent a message containing a submission, they add it (produce) to the queue, and increment a semaphore. Main.py then contains a loop which will wait on the semaphore, and when it is not zero, meaning there is something to consume, it will spawn an auto test handler in a new thread to carry out the auto testing and feedback.

The actual auto testing is done using a docker container. Auto_test_object.py creates the docker image, copies the input files, then creates the docker container, and runs the submitted code for each test case. The results of this test get written to an output.json file and passed to auto_feedback_object.py. This file iterates through each output and compares it to the expected output in testcases.json. It will then write the results of this to details.json in the student's submission folder.

3.5 Inter Application Communication

Since SCORE has two different interfaces that a student can use to submit code, their interfaces need to be able to communicate to the python server when a file needs to be tested.

3.5.1 Rust to Python Communication

To facilitate communication between the rust interface and the python server, a tcp connection is opened and maintained in RustListener.py. The rust server maintains a separate thread that contains a queue, and whenever a rust thread is used to submit a file, it is added to the queue. Then, every 15 seconds, the rust thread will send all the submissions that are on the queue to the python server over a TCP connection. RustListener.py will then add this submission to the queue as described in section 3.4.

3.5.2 Node to Python Communication

To facilitate communication between the node server and the python server, an http server is opened and maintained in NodeListener.py. HTTP was used here over TCP, as node operates on a single thread, and we did not want a blocking TCP call.

While this differs from the rust to python communication, all other interactions remain the same.

Components

In this section we will provide high level details for the main components of the SCORE application.

4.1 Rust Client

The rust client provides the front end of SCORE's command line interface. The main portion of this code is a while loop. It begins by waiting for user input. This input is then matched to one of SCORE's commands. The rust client will then prompt the user for any additional details needed for the command, then sends the command to the server using TCP. If the inputted command is not a SCORE command, then it will be run as a normal shell command on the user's machine. This ensures that only the expected commands ever make it to the server.

The rust client also handles the command line portion of the google OAuth process. While this will be discussed in greater detail in a later section, the important note for the client is that it spawns a webserver on a separate thread, prior to beginning the main loop. This server is used to handle the redirect that is given to google during the OAuth process. Once the OAuth process is complete, the web server is shutdown, and the user's information is passed to the main thread.

4.2 Rust Server

The rust server provides the backend portion of SCORE's command line interface. Much like the client, the server is also mainly a while loop, however the loop exists within a thread that is generated for each client that is connected. This means that multiple clients can be connected at the same time and will just be handled by separate threads. When the server receives a command from the client over TCP, it will match that command to a SCORE command, run the corresponding python script, then construct and send a response back to the client. As previously mentioned, the server also maintains a queue of submitted files and will periodically send those submissions over to the python server to be tested.

4.3 React Front End

SCORE uses React for the front end of the web application. This means we broke the front end down into components with the entry point by main.jsx. These components use fetch requests in order to get information from the backend, and to run SCORE commands.

4.4 Node Backend

The backend of SCORE's web application uses Node.js along with Express.js for routing. We also use Expression sessions to handle the cookies so the user can persist. Certain endpoints in the node server correspond to SCORE commands. When these are called from the front end using a post request, node will run the corresponding python scripts and then construct an appropriate response.

As previously mentioned, the node server also handles all MongoDB operations. The node server constructs the schema and creates the table for both the users and courses. Then, there are endpoints for each CRUD operation for each table. These endpoints must be called using a post request, and can be called from not only the React frontend but also the python commands as needed. By having all MongoDB operations in a single place, this makes it much easier to maintain.

4.5 Auto Test

As previously mentioned, the auto test functionality is written in python, and uses Docker containers. We chose to use containers to limit the effect that the student's code can have on the server, and we chose Docker due to its familiarity and integration with docker hub. A new image is created for each submission to be tested. While this can be seen as inefficient, we chose to have that inefficiency for the sake of added security.

When the auto testing has been completed, all output is written to outputs.json. This file contains not only the output from the test, but also an identifier for each to indicate what test case it came from. These identifiers are used to ensure that the output is compared to the correct expected output.

4.6 Auto Feedback

The auto feedback component is run after the auto test has finished. Its purpose is to score the submission and store the results. The score is generated by comparing

the actual output from outputs.json with the expected output from testcases.json. Once the score has been calculated, it is added to the details.json under the most recent submission section. If the score is higher than the previous high score, then it replaces the score section of details.json.

4.7 OAuth

SCORE uses Google OAuth for its user authentication. This means SCORE will reach out to Google's OAuth server and request the user's profile information. If the user agrees, Google will send back a token. SCORE can then exchange this token for the user's email address, which we use as the primary key of the user in MongoDB. While this is the general flow of OAuth, some of the details differ between the interfaces.

4.7.1 Rust

The OAuth process in the Rust portion of the project uses the BasicClient class from the oauth2 crate. To this class, we pass in SCORE's clientId, which is stored as an environment variable, along with the authorization url, and the redirect url. The redirect is then handled on a web server that is running on a separate thread, as mentioned in a previous section. Once the user accepts our request with Google, a post request containing the token is sent to the redirect web server. This token is sent back to the main thread, and the server is shut down. The final part of this flow is to then trade this token for the email using Rust's http_client class.

4.7.2 Web Application

For the React portion of the web app, we used the npm package react-oauth/google. We used the GoogleOAuthProvider that this package provided to wrap our app with the client ID in main.jsx. Then in SignInPage.jsx, we used the GoogleLogin component to handle the front end of the sign in process.

Once the front end receives the OAuth token from the GoogleLogin package, it is sent via a post request to the node backend. In the backend, we use the official google-auth-library package made by Google. This allows us to exchange the token, which is in the form of a JWT, for the user's email.