Developer Documentation

Development Environment Setup

The development environment for the prototype non-LTI application can currently be set up on any operating system, but for the purposes of development, we recommend using Ubuntu through WSL. To set up the development environment, the following steps must be completed.

1. Make sure that **NPM** is installed (it comes by default with most Ubuntu installations).
2. Install **NVM** and the latest version of **Node** through **NVM** (the version used is v16.17, but the latest version should suffice).
3. Clone the git repository to your development environment.
4. Install **PostgreSQL** to your machine and perform the initial setup for it.
5. Navigate into the "**Database Files**" folder in the repository and run the command "sudo -u postgres psql" to open the psql command line tool. (Note: this may be different if not on Linux)
6. While in the psql tool, run the command "**CREATE DATABASE "CodingExam";**" to create the database.
7. Run the command "**\c CodingExam**" to connect to the database.
8. Run the command "**CREATE SCHEMA "CodingExam";**" to create the database schema.
9. Run the command "**\i 'CreateDatabase.sql';**" to create the database tables, application account, and insert data into them for the prototypes.
10. Navigate to the "**api**" folder and run the command "**npm install**" in the command line to install the libraries needed for the server.
11. Navigate to the "**client**" folder and run the command "**npm install**" in the command line to install the libraries needed for the client.
12. To start the application, start an instance of **WSL**, navigate to the "**api**" folder within the "**CodingExam642**" repository, and run the command "**npm start**" to start the server. Once the server has been started, open a second WSL client, navigate to the "**client**" folder, and run the command "**npm start**" to start the client.

To set up the development environment for the LTI application, steps 1 through 8 still need to be completed, but starting the application is slightly different, and a few more libraries must be installed.

1. Install the tool "**ngrok**" using instructions from their website (https://ngrok.com/download).
2. After "**ngrok**" is installed, make an account and configure the authtoken for your installation. Instructions on how to do this can be found at the following link after logging in (https://dashboard.ngrok.com/get-started/your-authtoken).
3. Navigate to the "**lti**" folder and run the command "**npm install**" to install the libraries needed for the LTI application.
4. To start the application, start an instance of WSL, navigate to the "**lti**" folder, and run the command "**npm start**".
5. After the application has been started, open a separate WSL client, enter the command "**ngrok http 3000**", and copy the link labelled "**Forwarding**".
6. Enter the Canvas test installation and then navigate to **Settings > Apps > View App Configurations > + App**. Enter the necessary information for the application, such as a name, Consumer Key, and Shared Secret. In the "**Launch URL**" box, enter the URL generated by ngrok followed by a **"/lti"**.

7. After this has been done, create a new assignment with a submission type of “**External App**” and enter the URL generated by ngrok followed by a “**/lti**” again. When opening this assignment, the LTI application should load.

Client

The prototype of the client mainly consists of the **App.tsx** file, a **React JavaScript** program that communicates with the server and runs on localhost port **3000**. At start-up, it queries the database for all the questions related to a given exam. Currently, we just have a single multiple-choice question. The user can then select an answer and submit, posting to the database their answer for the given question. They then receive message saying what answer they now have stored in the database. We found that the React front end worked well for a client application, and the Blueprint.js library that we used provided a unified

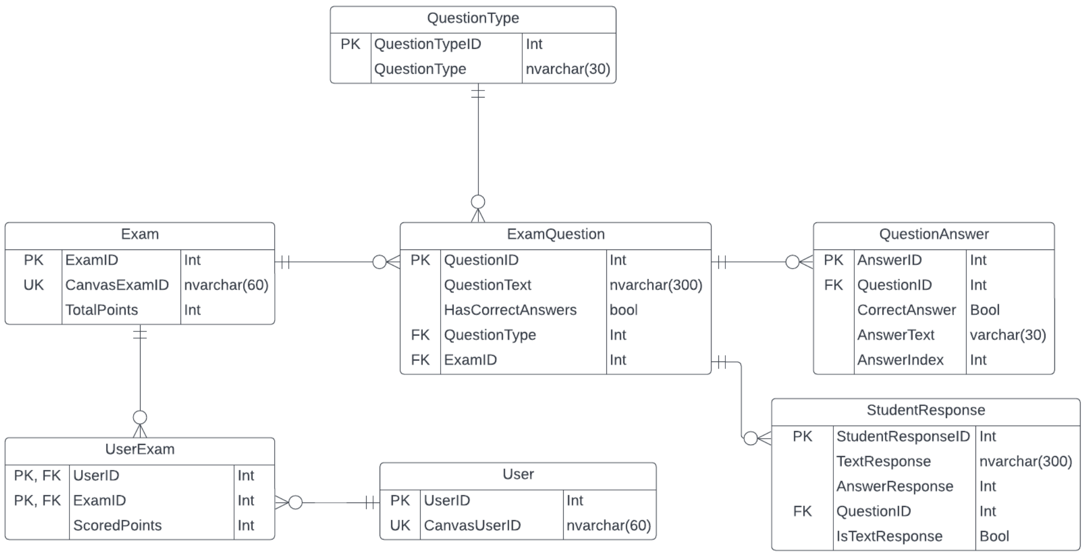
Server

The prototype of the server is an **Express JavaScript** application that performs request routing, resource fetching, and database interaction. The server by default runs on localhost on port 3000. The program contains two main routes currently for requests. A GET request to the “**/api/questions**” endpoint will return the questions for an exam as well as the available answers to those questions if there are any. For prototyping purposes, there is only

A POST request to “**/api/**” occurs when the user on the front end submits their answer for the multiple-choice question. When the server receives the request, adds a row to the “**StudentResponse**” table of the database. We found that this worked well for inserting response data, and we are planning on using this general structure in the initial production codebase next sprint. As of this sprint, the feature does not support recognizing different users, questions, exams, or courses, so the SQL statement inserting the row will remain the same aside from the answer selected by the front-end user.

Database

The prototype database’s structure is depicted in the following diagram.



The “**Users**” table represents all users that can log into the application, which primarily includes students and instructors for a course. It contains a “**UserID**” that is a primary key and generated by an identity property, and a “**CanvasUserID**” column that is a unique key and represents the internal ID from Canvas of the user.

The “**Exam**” table represents an exam or assignment created for the application. It has columns “**ExamID**” that is a primary key and generated by an identity property, “**CanvasExamID**” that is a unique key and represents the internal ID of the assignment in Canvas, and “**TotalPoints**” that represents the total number of points possible for the exam.

Since users can take multiple exams, and an exam has multiple users take it, the joint table “**UserExam**” is used to represent the unique combinations of user and exam. This table has columns “**UserID**” and “**ExamID**”, which are both primary keys and foreign keys to the “User” and “Exam” tables respectively. The “**ScoredPoints**” and “**ExamScorePercent**” columns represent the number of points scored by the user on an exam and the percentage score of the exam after it is graded.

The “**QuestionType**” table is a reference table that has one row for each of the available question types, similar to an enum.

The “**ExamQuestion**” table represents each of the questions on a created exam. It has a “**QuestionID**” column that is a primary key generated from an identity property, a “**QuestionText**” column that contains the text of a question, and a “**HasCorrectAnswers**” column to function as a Boolean. If the question has answers to choose from, it will be true, and if it is a written response question, it will be false. For foreign keys, the columns “**QuestionType**” and “**ExamID**” reference the “**QuestionType**” and “Exam” tables respectively. The reference to “**QuestionType**” column allows you to determine what type of question a row represents, and the “**ExamID**” column allows you to determine what exam the question belongs to.

The “**QuestionAnswer**” table represents the available answers for a question in the **ExamQuestion** table, such as for a true/false or multiple answer question. It has a “**AnswerID**” column as a primary key generated by an identity property, a “**QuestionID**” column that is a foreign key referencing the “**ExamQuestion**” table. This represents the question it is a potential answer for. It also has an “**AnswerText**” column representing the text of the answer, an “**AnswerIndex**” column for the index where the answer is displayed, and a “**CorrectAnswer**” Boolean column representing whether the answer is correct.

The “**StudentResponse**” table represents a student’s response to a given question for an exam. It has a “**StudentResponseID**” column as a primary key generated by an identity property, a “**QuestionID**” column that is a foreign key referencing the “**ExamQuestion**” table. This represents the question it is a response to. It also has “**IsTextResponse**” and “**TextResponse**” columns to denote if it is a text response, and what the text response is. A “**AnswerResponse**” column is also used to indicate the index of the student’s response.

Overall, this database design seemed to do well in our prototyping, and we will likely use its general structure in the production application