Developer Documentation

Application Functionality

Currently, the application is still very much in a development state and is not currently functional enough to be considered production ready. However, the application components that are finished are coded up to production standards as best as possible, and a testing suite is included as well. Because of how the client application is currently coded in React, unit testing is not very effective, so we have included a written test plan for this iteration for the client specifically.

The application currently only supports three question types: multiple choice, true or false, and short answer. Users cannot add these questions to the application currently, although more can be added manually by the developer to the database. Additionally, only one exam is currently supported, again because questions cannot be added by the end user and must be done manually. The application also only supports one user currently. Because of these facts, many of the Canvas identifiers for properties such as UserID and CanvasExamID are hard coded, although they do use the real values from Canvas, and the server is set up to process the values from the client as though they are not hard coded. This means that it will be relatively simple to replace them with dynamically received values in the client in a future sprint.

When the user loads the application outside of Canvas, the React client app is returned, and when the client renders itself, it makes a GET request to the server to retrieve the list of questions and renders the questions dynamically based on what is returned. After this, it makes another GET request to the server to see if any responses have already been submitted and renders those responses if there are any.

The user can set their answers to the provided questions and click submit, and they will receive feedback on if their answer was able to be submitted or not. The server will insert or update their answers in the database if they are valid and send a response notifying the client of whether the submission was valid or not.

Development Environment Setup

The development environment for the application outside of Canvas 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 and run the application, 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 "**server**" 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, then run the command "**npm run build**" to create a build of the client application
12. To start the application, start an instance of **WSL**, navigate to the "**server**" folder within the "**CodingExam642**" repository, and run the command "**npm start**" to start the server. The application will be running at "**localhost:9000**" and can be accessed through the web browser.

To have the application run through Canvas, additional setup is required. The application function itself is identical, but it is accessed through a POST request from Canvas as opposed to a GET request.

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. Start the server using the instructions from the above section.
4. After the application has been started, open a separate WSL client, enter the command "**ngrok http 9000**", and copy the link labelled "**Forwarding**".
5. 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.
    a. For testing this application specifically, there is already an external app named "**CodingExam Noah App**" that can be edited, leaving the Key and Secret but replacing the Launch URL.
6. After this has been done, create a new assignment with a submission type of "**External App**" and enter the URL generated by ngrok.
    a. For testing this application specifically, there is already an assignment created named "**CodingExam Noah Assignment**" that can be edited, replacing the External Tool URL with the new URL generated by ngrok.
7. Because there is a restriction with free ngrok accounts, the app must be loaded outside of Canvas first to clear a security warning before it can be displayed in Canvas. Paste the URL generated by ngrok into your web browser and click the "**Visit Site**" button. The security warning should then be cleared, and the app should now load in Canvas when visiting the assignment page.

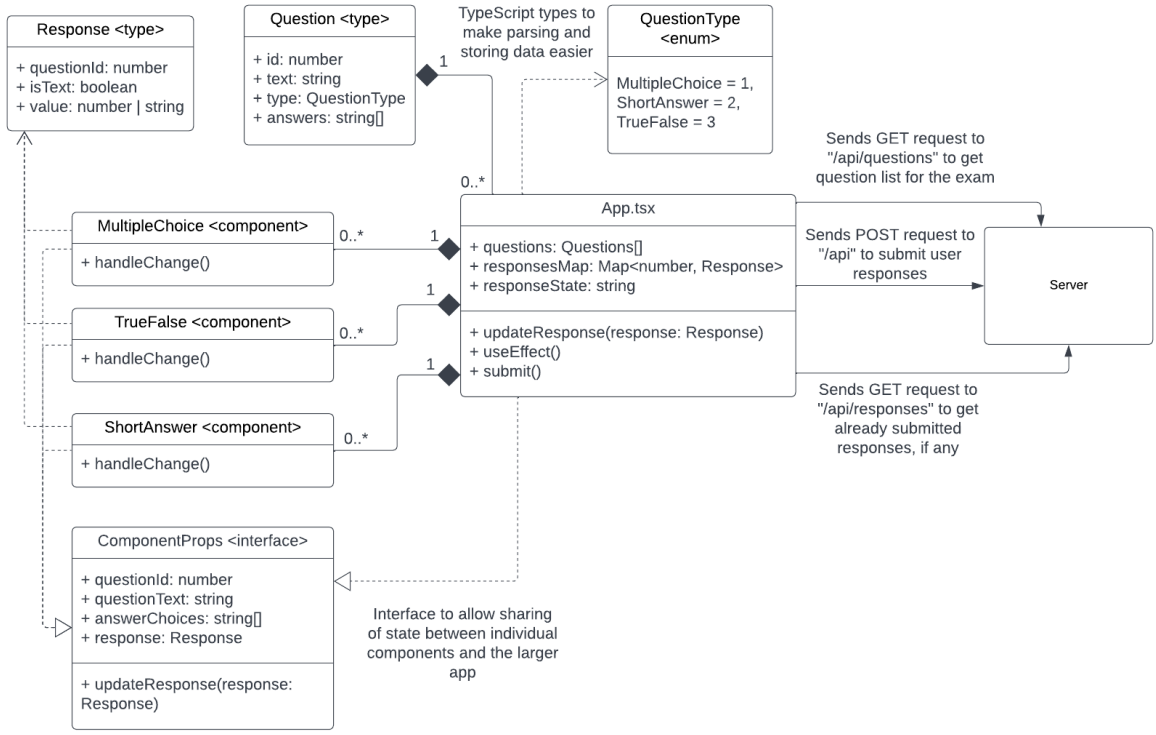
Client

The client application is a React JavaScript application, with the main body of the application housed in the "**App.tsx**" file. The client also consists of three different React components, named "**multipleChoice.tsx**", "**trueFalse.tsx**", and "**shortAnswer.tsx**" for the multiple choice, true or false, and short answer questions respectively. Each component has props to keep track of its own state as well as inform the parent App for its state. The props come in the form of "**ComponentProps**", which define a common interface for each component to implement. The parent App has a "**questions**"

state to keep track of the questions retrieved from the server and a "**responsesMap**" state to keep track of the responses that have been entered by the user. These states are both backed by the "**Question**" and "**Response**" types through TypeScript, both of which are detailed in the UML diagram in this section.

When the app is rendered, it makes a GET request to the "**/api/questions**" endpoint of the server with the internal Canvas assignment ID as a header. The response from this request contains a list of questions that the application then renders dynamically. After this, the app also makes a request to the "**/api/responses**" endpoint of the server with the internal Canvas user and assignment IDs as headers, which will return a list of the responses to the questions if the user has already submitted a response. The app will then set the state of the components accordingly based on the responses.

After the user has filled out their responses, the client app sends a POST request to the "**/api**" endpoint of the server with the internal Canvas user ID as a header on the request and the responses as a JSON object in the body of the request. Currently, the internal Canvas IDs are hard coded in the client application, as only one user and assignment is supported, but these will eventually be replaced with a dynamic system in a future sprint.



Server

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 9000. When a request is made to the main endpoint, the server routes the request through a request router and returns the "**index.html**" file built from the client application.

The server contains two main routes: an LTI route and an API route, provided by the "**lti.js**" and "**api.js**" files respectively. The LTI route accepts both POST and GET requests to the "**/**" endpoint and is used to launch the main application. The POST request that the LTI endpoint receives is

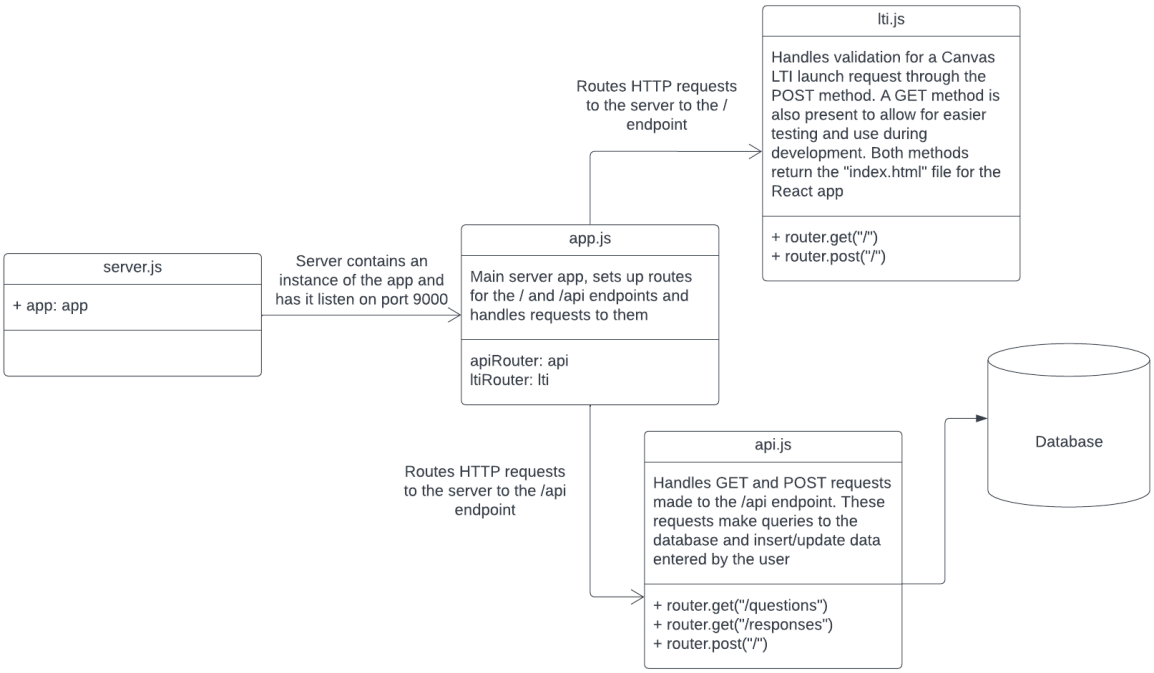
intended to be sent from an LTI launch request, likely originating from Canvas. Because of this, the LTI endpoint uses the "**ims-lti**" library, which allows for it to validate the request using a shared secret, and if the request is valid, send back the client application. Since the application is still in the development stage, the client key and secret are hard coded in the server application, but they will be moved to a database as we expand the application further in future sprints.

A GET request to the LTI endpoint will simply send back the client application. This was done to assist in developing the application and will be removed as we develop closer to a production build.

The API route accepts both GET and POST requests from the client at the "**/api**" base endpoint. A GET request to "**/api/questions**" occurs when the application is started for the first time, and it expects a request containing the internal ID of the assignment in Canvas as a header. It will then look up the questions for that exam in the database and return that data to the client for it to render.

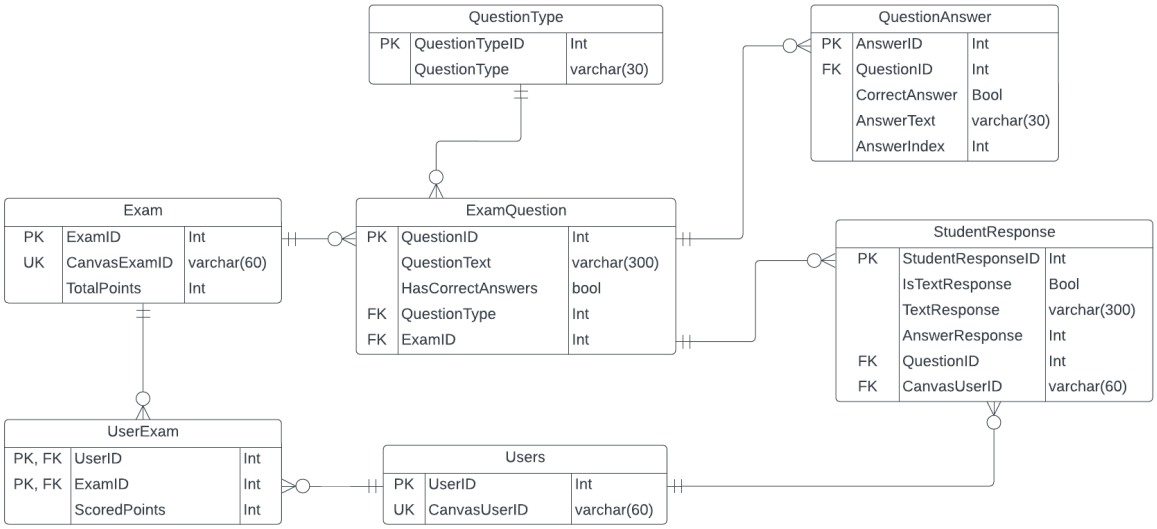
A POST request to "**/api**" occurs when the user on the front end submits their answers for the exam page they are on, and it expects a request containing the internal Canvas ID of the user as a header as well as the IDs of the questions answered and their responses. The server will then either add a row to the "**StudentResponse**" table of the database with their response information or update an existing response to a question if the user has already answered it. Finally, it will send a response to the client letting it know it was a valid submission.

A diagram of the server is shown below to give a general idea of how the different files interact with each other. Since this is a JavaScript application, it does not follow the typical object oriented design pattern, but this diagram should help to describe its functionality and how the different components communicate and work with each other.



Database

The current database 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-choice question. It has an “**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 for grading purposes.

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, and a "**CanvasUserID**" column that is a foreign key referencing the "**Users**" table. These represent the question it is a response to and the user that the response comes from. It also has "**IsTextResponse**" and "**TextResponse**" columns to denote if it is a text response, and if it is, what the text response is. An "**AnswerResponse**" column is also used to indicate the index of the student's response.