Developer Documentation

Application Functionality

Currently, the application is still very much in a development state and does not have full functionality. 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 for the server, which are located under **/server/tests**. These tests can be run by navigating to the **server** directory and running the command **npm test**. Note that before the server tests are run, the test server entry point must have been started at least once using the command **node serverTestEntry.js**. This allows for the database to be setup before the tests are run. Additionally, for the GUI components, testing plans for both the Student and Instructor views have been included.

The application currently only supports five question types: multiple choice, true or false, short answer, code editor, and Parson's Problems. Instructor users can add questions to an exam in the application as well as partially edit them. Multiple exams/assignments are supported by the application, and the application also can create new exams/assignments. However, it will only do so when running through Canvas on the HTTP proxy, since it uses the real internal Canvas ID of the assignment when creating it in the database. Because of this, the recommended way to create an exam currently is still by manually adding it to the database, although it can be done through Canvas as well. The application supports multiple users through JSON Web Tokens, which are encoded with the information needed to identify who they are, what exam they are on, and what their Canvas role is. To avoid needing to run the Canvas setup process every time a test is needed, a development page is used to select between the Instructor and Learner modes when not running in production. Additionally, to aid in client development, both student and instructor tokens are hard coded and commented out in the client, and they can be uncommented when needed.

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 check what role the current user is. If the user is a student, the application will retrieve the list of questions and render 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. If the student has already submitted the exam, the questions will be disabled and feedback boxes that contain any instructor feedback that was left will appear below each question.

If the role returned by the server for the current user is an instructor, the application will make another GET request to the server to get a list of the students who have submitted responses for the current assignment and display them. When the instructor clicks on a student, the application will do a GET request to get the questions for the exam and a GET request to get the specific student's responses to the exam. Feedback boxes will also be rendered below each question where an instructor can write feedback for the student. Once the instructor has entered their desired feedback, they can click the submit feedback button to send the feedback to the server and return to the list of students. Instructors can also switch to the create exam view, which will allow for them to add questions to an exam. At this time, instructors can only add and edit questions within the application, not remove them. The view will show the current questions in the exam

when creating new questions. Instructors are also able to grade submissions through the grading view of the application. Currently, the grading view will display the individual responses of the students for each question, and the Instructor can see the list of questions on the right side of the view. A feedback box is also located by each submission, allowing for the instructor to either view existing feedback on the question or leave new feedback. Instructors can enter the student's score for the question, and once the Instructor has graded everyone's submissions, they can submit the grades to Canvas, where the server will then calculate the student's grade and update their Canvas assignment score accordingly. The multiple choice and true false questions also support automatic grading, meaning that when opening the grading view, those question types will already have scores of either full or no points for the student based on the correct answer to the question and the answer the student submitted. These scores can be changed by the instructor as well.

To change roles in the application when in development mode and not running in Canvas, you can return to the **localhost:9000** page and select which role you would like to view the application as. Additionally, you can click the "Retake Exam" button to retake the exam as a student after submitting it. To change roles within Canvas, you can use the Student View button to switch between the Instructor and Student roles within Canvas itself, which will also change the view in the application. Note that if grading is tested when doing this, do not click the "Reset Student" button and then try to submit grades to the application. This will create a new "Test Student" within Canvas and overwrite the other one, which will cause grading to begin throwing error messages, as the values of the old test student no longer exist.

Development Environment Setup

Since the application is dockerized, the development setup is relatively simple. 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 with docker, the following steps must be completed.

1. Make sure that **Docker** is installed.
    a. Note that for running Ubuntu through WSL2, Docker Desktop for Windows will also need to be installed.
2. Navigate to the main folder of the repository and run the command **docker compose up --build** if you are running it for the first time.
    a. If you are not running the container for the first time, you can instead run **docker compose up**.
3. Docker should create a container for the Postgres instance and a container for the main application.
4. The application can now be accessed through **localhost:9000.**

To run the application in development mode outside of docker, you will need to do the following:

1. Run **npm install** in both the client and server directories to install the required packages.
2. Install **postgresql** to your environment and run the **ResetDatabaseScript.sql** file to create the database.
3. Navigate to the client folder and run **npm run build** to build the client.
4. Navigate to the server folder and run **npm start** to run the database migrations and start the server.

5. The application can now be accessed through **localhost:9000**.

When the application is accessed through the above address, a developer page will be accessible that allows for the user to pick between the Learner and Instructor views To have the application run through Canvas, additional setup is required.

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** that can be edited, leaving the Key and Secret but replacing the Launch URL.
    b. The Key used is “Codekey”, and the Secret used is “Keysecret”.
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 Assignment 1** 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 five different React components, named **multipleChoice.tsx**, **trueFalse.tsx**, **shortAnswer.tsx**, **feedbackBox.tsx**, **codingAnswer.tsx**, and **parsonsProblem.tsx** for the multiple choice, true or false, short answer, feedback, coding answer, and Parson’s Problem questions respectively. The Parson’s Problem component is also dependent on the **parsonsColumn.tsx** and **columnItem.tsx** components for column and item functionality. 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. A **gradingGrid.tsx** component is used to display the student responses when grading. There is also a **confidenceRating** component that will eventually be used by users to leave their confidence rating for questions, but this is a component provided by Dr. Bean for use in a future sprint. This component is not currently used in any way.

Each component is given a unique id that allows it to interact with the Redux store. The Redux store contains eight major components: **questionIds, questionsMap, submissionsMap, feedbackIds, feedbackMap, nextQuestionId,** and **token**. **questionIds** holds an array of unique ids for

each question in the exam, and **questionsMap** maps each of these ids to the actual Question object that holds its information. **submissionsMap** maps each student's **canvasUserId** (the userId used to interact with Canvas) to map of their submissions, which maps **questionIds** to **Submission** objects. This way, each view can access Submissions on a per user per question basis and data retrieval is very easy to handle. **feedbackIds** holds an array of unique ids for each feedback the instructor has submitted for the student, and **feedbackMap** maps each of these ids to their respect Feedback object in the store. **nextQuestionId** holds an incrementing integer that is used to determine what the next question id should be when creating an exam. **Token** stores the token passed to the client from the server.

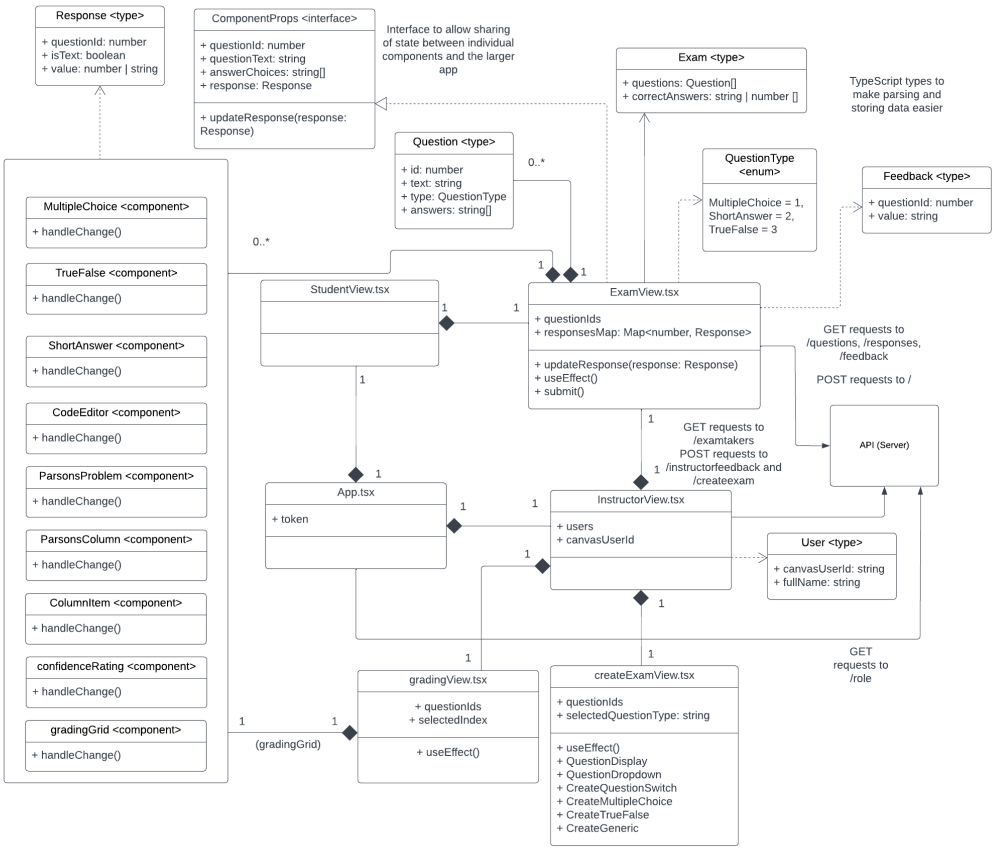
When the app is rendered, it stores the token that it receives as a part of the **index.html** file and makes a GET request to the **/api/role** endpoint of the server with the token to determine if the user is a Student or an Instructor. If the client is a Student, the app will then render the **StudentView**, and if the client is an Instructor, it will render the **InstructorView**. The **StudentView** will simply render an **ExamView**. This will request will also tell the client if it should display the points possible for the exam questions. Within **ExamView**, a request to **/api/questions** is made with the client's JWT token as a header to get the questions for the exam. Next, a GET request to **/api/submissions** is made with the client's JWT token as a header to get the responses that the student has already submitted, if there are any. Finally, a GET request to **/api/feedback** is made with the client's JWT to get any instructor feedback for the user. Note that at this time, the **ExamView** loaded for a student will not display this feedback until the exam has been submitted. Upon reload, the exam will be disabled and feedback will be loaded.

For the **InstructorView**, when it is initially rendered, a GET request to **/api/instructor/examtakers** with the client's JWT token is made to get the list of students who have taken the exam. When a name is clicked, an **ExamView** is rendered almost identically as in the **StudentView** but with the question components disabled and feedback boxes rendered below each question. When the user clicks the **Submit Feedback** button, a POST request to **/api/instructor/feedback** is made with the client's JWT token and the userID of the student the feedback is for as headers and the feedback itself as a JSON object in the body of the request.

The instructor is also able to create an exam, where they can select which questions they want and their respective answers. After adding all the questions, a POST request to **/api/instructor/createexam** is made using their token and passing the list of questions. Additionally, the same view can be used to edit or delete the questions for an existing exam.

When grading exam submissions, the application will load the **gradingView** within the **InstructorView**. This view contains the list of questions for the current exam as well as a **gradingGrid** component. This component is a grid with four columns, one for the student's name, one for their submission, one for the feedback for that submission, and one for their current grade on the question. When grades are entered in this view, they are automatically sent to the server and updated within the database. The grid will contain a row for every response from a student for the question. The Instructor can also click the "Grade" button on the page to submit the grades that are currently entered in the database to Canvas.

Since the application is a JavaScript React application, a UML diagram does not directly apply to the application. However, a diagram has been included below to give a better sense of the client application, its components, and how they interact with each other.



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 through docker. 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. This **index.html** file is modified to also contain the client's token before it is sent, allowing the client to identify themselves with their requests.

The server contains three main routes: an LTI route, an API route, and an Instructor route, provided by the **lti.js**, **api.js**, and **instructor.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. The key and secret for the LTI validation are a part of the docker file. These variables must be moved to a separate .env file if they are to be used outside of the docker image.

A GET will bring up a special developer screen. This screen will give the user the option to load either the Learner view or the Instructor view. Depending on their selection, the application will encode the appropriate token into the **index.html** file and send it to the client. This was done to assist in development, as switching between views previously required stopping the application and modifying the code to send a different token. Additionally, you can click the "Retake Exam" link to take the exam again as the same student for development purposes. For both the GET and POST endpoints, a JWT token is generated using the assignmentID, userID, role, fullName, familyName, givenName, and email values provided by the LTI launch request, and the JWT is encoded in the

index.html file as a window property so the client application can access the token once it is rendered. Before the file is returned, the route will also insert a row into the Exam table if the user is an Instructor and the exam has not been opened before, and it will also insert a row into the **exams_users** table if a student is opening the exam for the first time. These are done through the **createExam** and **findOrCreateUser** functions respectively.

The API route accepts both GET and POST requests from the client at the **/api** base endpoint. The router is designed to handle requests from both the Instructor and Student views, although Instructors may not use some of them. There is also a middleware function, **router.use**, which enforces that every request must come with a valid authentication token as a header. If there is no token, a 403 error will be returned, otherwise the function will return the data within the token.

A GET request to **/api/questions** occurs when the application requests the questions for the exam from the serer. It will use the decoded the JWT to get the assignmentID, look up the questions for that exam in the database, and return that data to the client for it to render. This endpoint is used by both the Instructor and Student views.

A GET request to **/api/submissions** serves to return responses to a question for a given user and exam combination, and it occurs just after the **/api/questions** endpoint has been called by a client. It will use the information from the decoded JWT to get the userID, assignmentID, and role of the client and the database is queried to retrieve the responses for the specified student's exam, using the userID from the token if the client is a student or the userID from the header if the client is an instructor. It will then return the responses to the client to be rendered. This endpoint is used by both the Instructor and Student views.

A GET request to **/api/role** serves to let the client know if it is a Student or an Instructor in the application. It will use the role within the JWT token to determine the user's role before sending that role back to the client. It will also make a query to the database to see if the user has taken the currently loaded exam and send that in the request as well. The endpoint is used by both the Instructor and Student views.

A POST request to **/api** occurs when a Student client submits their answers for the exam page they are on, and their responses as the body of the request. The server will use the userID of the student from the decoded JWT and either add a row to the **student_responses** table of the database with their response information or update an existing response to a question if the user has already answered it. After this has been done, the endpoint will call the autoGrade helper function, which will grade any questions that have correct answers and can be automatically graded. Finally, it will send a response to the client letting it know it was a valid submission. This endpoint is only used by the Student view.

A GET request to **/api/confidence** occurs when the confidence ratings for a student's response are requested by the client. This will get the list of confidence ratings for a student's submissions for an exam using information from the decoded JWT and return them to the client. If the user is an Instructor, it will use the userID header instead of the one from the token. The endpoint will be used by both the student and instructor views. However, it is not currently used within the project and is for a planned feature in the future.

A GET request to **/api/feedback** occurs when a client loads a student's exam after feedback has been left. If the client is an Instructor, the request must contain the userID of the student they are retrieving feedback for as a header. The sever will then use the userID, examID, and role of the

client and query the **student_responses** table in the database to retrieve the feedback for the specified user and exam, using the userID from the token if the client is a student or the userID from the header if the client is an instructor. It will then return the feedback to the client. This endpoint is used by both the Instructor and Student views.

The Instructor route accepts both GET and POST requests from the client at the **/api/instructor** base endpoint. The router is designed to handle requests that only an Instructor should be able to make. In addition to the **router.use** middleware function, the Instructor endpoint also contains the **InstructorOnly** middleware function. This function will return an invalid request if the role within the user's JWT does not decode to be an Instructor. All endpoints mentioned from this point forward will pass through the **InstructorOnly** middleware function and are only used by the Instructor view.

A GET request to **/api/instructor/examtakers** occurs when the Instructor view is loaded and returns a list of students who have taken the exam the Instructor has selected. It uses the examID of the exam and the client's role from the decoded JWT. If the client's role is not an Instructor, the endpoint will return an invalid request and do nothing else. If the client's role is an Instructor, the endpoint will query the **exams_users** table of the database for the list of students and return the list to the client.

A GET request to **/api/instructor/allsubmissions** occurs when the Grade view within the Instructor view is loaded and returns all the submission from students for an exam. It expects a request. It will query the database to retrieve these rows and then return the results as a Submissions typescript object.

A POST request to **/api/instructor/feedback** occurs when the Instructor submits the feedback left for a student, and it expects a request containing the userID of the student feedback is being left for as a header. If the client is not an Instructor, an invalid request will be returned, and the endpoint will do nothing else. If the client is an Instructor a query is made to the database to update rows in the **student_responses** table with the feedback left by the Instructor.

A GET request to **/api/instructor/newquestionid** occurs when a new question is created by the Instructor. It simply returns the next valid question ID from the database to avoid any data being overwritten as questions are created.

A POST request to **/api/instructor/createexam** occurs when the Instructor has created the questions for a new exam or edited the questions for an existing exam, and it expects a request containing the questions and answers for the exam as a JSON string in the body. The JWT will be decoded to get the internal Canvas ID of the assignment as well as the client's role. If the client is not an Instructor, an invalid request response will be returned. Otherwise, the server will iterate through each question in the body of the request and parse it to get the correct information to be inserted into the database. To get if the question has correct answers, it will check the question type and set a local property to keep track of that. The server will also make a query to the database to get the database examID of the assignment using the internal Canvas ID. Finally, the server will insert the question into the database and any of the questions answers, setting the correct answer in the database using the index of the answer. If the question already exists in the database, it will update the values instead, allowing the user to edit the questions for an existing exam.

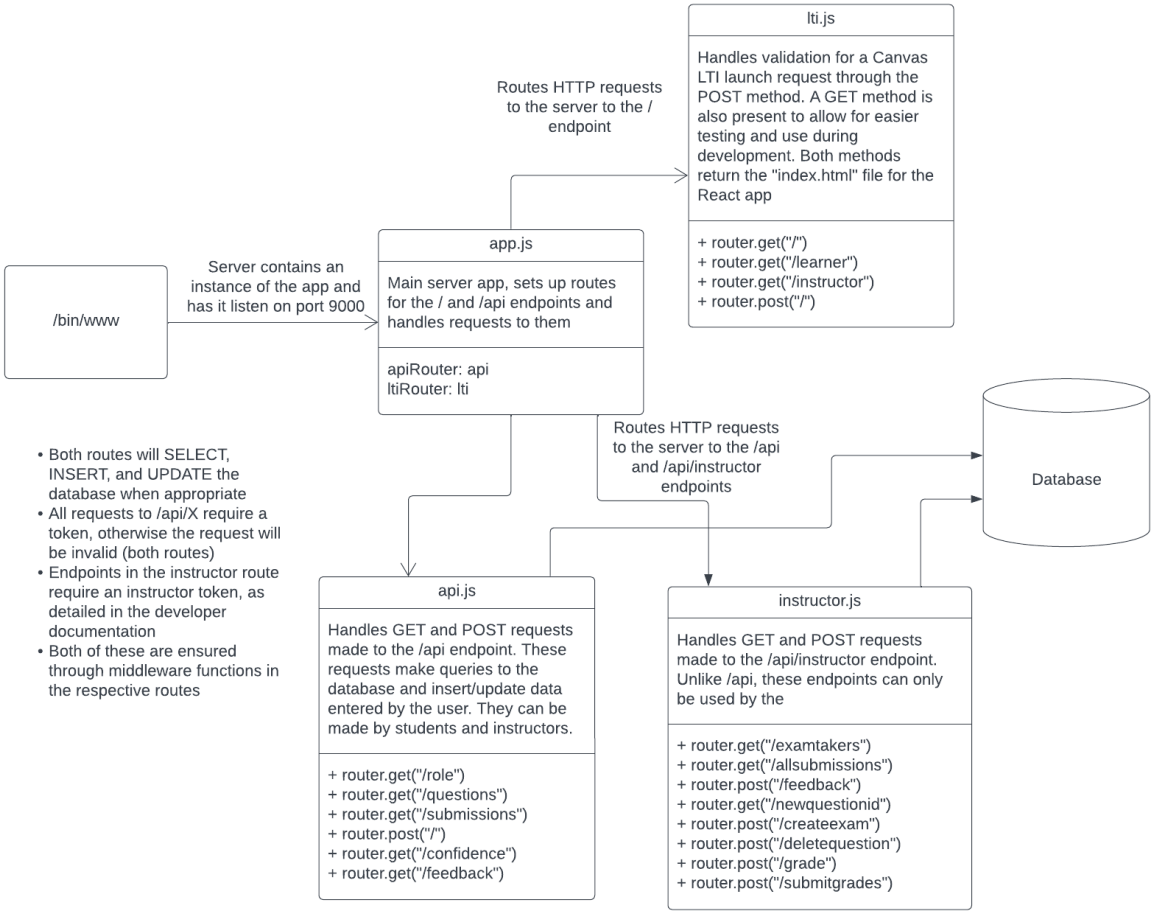
A POST request to **/api/instructor/deletequestion** occurs when the Instructor deletes a question when editing the exam. It expects a request containing the questionID of the question being deleted. The JWT will be decoded to get the internal Canvas ID of the assignment as well as

the client's role. If the client is not an Instructor, an invalid request response will be returned. Otherwise, the server will mark the question as deleted with a soft delete, which is done by marking the **is_deleted** column as true. This will prevent the **/api/questions** endpoint from returning that question when getting the list of questions.

A POST request to **/api/instructor/grade** occurs when the Instructor submits a grade for a particular student's question response after grading it. It expects the userID of the student the grade is being submitted for, the questionID of the question, and the grade for the student's response in the body of the request. It will then update the **scored_points** field in the **student_responses** table with their score for the response.

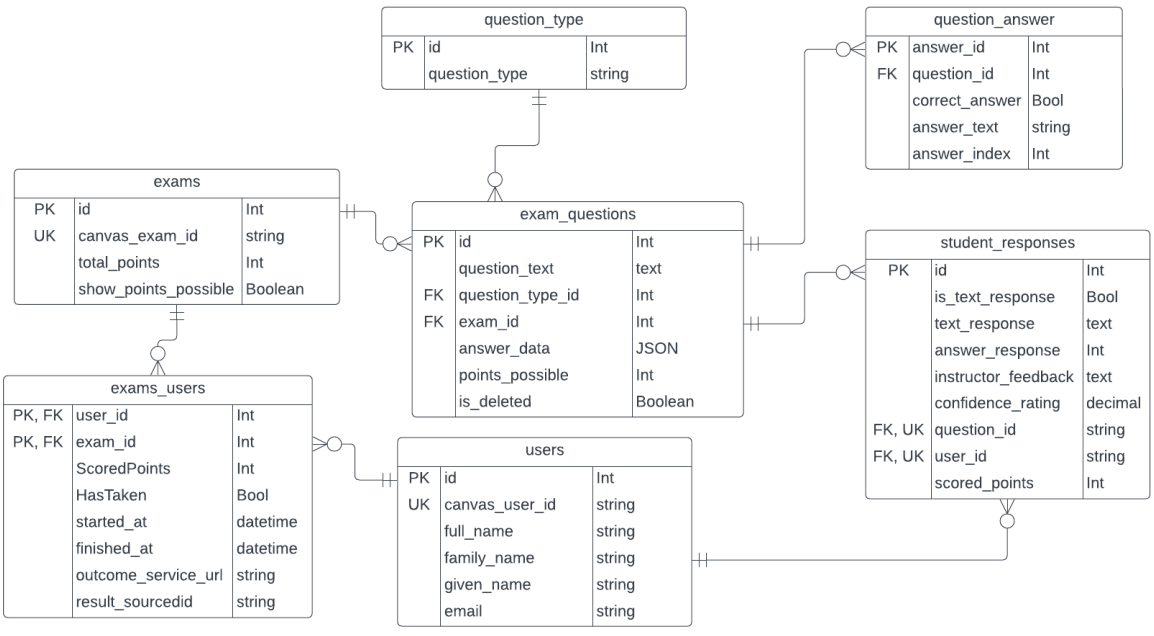
A POST request to **/api/instructor/submitgrades** occurs when the Instructor syncs the scores within the application to the Canvas gradebook. First, the endpoint will call the **updateGrades** function, which will update the **ScoredPoints** and **total_points** fields used to calculate the overall assignment grade. This function will aggregate the **scored_points** values from the **student_responses** table to update the **ScoredPoints** field in the **users_exams** table for the appropriate rows. It will also aggregate the **points_possible** column on the **exam_questions** table to update the **total_points** column on the **exams** table in case the instructor has adjusted the total point value for the exam. After this, the endpoint will get the internal ID and total number of points for the assignment. Next, it will get all the rows within the **exams_users** table that have taken the exam. It will then iterate through those rows, create a percentage grade for the student using the **ScoredPoints** and **total_points** fields, and then submit that grade to Canvas using the **outcome_service_url** and **result_sourcedid** values from the **exams_users** rows. This is done by encoding the grade and **result_sourcedid** values into an XML message and sending them via a POST request to the **outcome_service_url**. The server will then send a message to the client to inform them if the submission was successful or not.

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 an **id** column that is a primary key and generated by an identity property, and a **canvas_user_id** column that is a unique key and represents the internal ID from Canvas of the user. The remaining columns are used to store additional identifying information for each user.

The **exams** table represents an exam or assignment created for the application. It has the column **id** that is a primary key and generated by an identity property. The column **canvas_exam_id** that is a unique key and represents the internal ID of the assignment in Canvas, and the **total_points** column represents the total number of points possible for the exam. The **show_points_possible** column is a Boolean value that is used to determine if the exam should show the number of points possible for each question.

Since users can take multiple exams, and an exam has multiple users take it, the joint table **exams_users** is used to represent the unique combinations of user and exam. This table has columns **user_id** and **exam_id**, which are both primary keys and foreign keys to the **users** and **exams** tables respectively. The **ScoredPoints** column represents the number of points scored by the user on an exam, and the **HasTaken** column is a Boolean value that indicates whether a student has already taken an exam and has a default constraint to be false. The **started_at** and **finished_at** columns are datetime values that represent the times the student started and finished the exam respectively. The **outcome_service_url** is the URL provided by the LTI launch request that is used to post grades to the LTI provider, and the **result_sourcedid** column is an identifier given by the LTI launch request that represents the unique combination of user and grade within the LTI provider, in this case Canvas.

The **question_type** table is a reference table that has one row for each of the available question types, like an enum.

The **exam_questions** table represents each of the questions on a created exam. It has an **id** column that is a primary key generated from an identity property, a **question_text** column that contains the text of a question, and the **answer_data** column stores any additional info about the question in the form of a JSON string. The **points_possible** column represents the point value of the individual question. For foreign keys, the columns **question_type_id** and **exam_id** reference the **question_type** and **exams** tables respectively. The reference to **question_type_id** column allows you to determine what type of question a row represents, and the **exam_id** column allows you to determine what exam the question belongs to. The **is_deleted** column is used for soft deletions of questions, and it is set to false by default. When a question is deleted, the column is set to true.

The **question_answer** table represents the available answers for a question in the **exam_questions** table, such as for a true/false or multiple-choice question. It has an **id** column as a primary key generated by an identity property, a **question_id** column that is a foreign key referencing the **exam_questions** table. This represents the question it is a potential answer for. It also has an **answer_text** column representing the text of the answer, an **answer_index** column for the index where the answer is displayed, and a **correct_answer** Boolean column representing whether the answer is correct for grading purposes.

The **student_responses** table represents a student's response to a given question for an exam. It has an **id** column as a primary key generated by an identity property, a **question_id** column that is a foreign key referencing the **exam_questions** table, and a **user_id** column that is a foreign key referencing the **users** table. The two columns also form a unique constraint together. These

rows represent the question it is a response to and the user that the response comes from. It also has **is_text_response** and **text_response** columns to denote if it is a text response, and if it is, what the text response is. An **answer_response** column is also used to indicate the index of the student's response, and the **instructor_feedback** column is used to store feedback left by the instructor for a response. The **scored_points** column is used to denote the student's score on the particular response. The **confidence_rating** column will be used to store a student's confidence in their response's correctness, which is a feature we plan to add in a future sprint.