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 for both the client and the server. Server tests are located under **/server/tests**, and client tests are located under **/client/components/tests** and **/client/slices/tests**. These tests can be run by navigating to either the **server** or **client** directories and running the command **npm test**.

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, but they are only able to add questions, not edit them. Additionally, the add question view does not currently show any existing questions on an exam when adding to it. Multiple exams are supported by the application, and the application also can create new exams. 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. 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, both student and instructor tokens are hard coded and commented out in both the server and 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 questions to an exam, not edit or remove existing ones inside the database. Additionally, the create exam view does not yet show any questions that are already a part of the exam.

To change roles in the application when not running in Canvas, open the file **lti.js**, which is located at **/server/routes/lti.js.** Within the **router.get()** function and inside the **token1** variable, change the variable **roles** to be either **"Instructor"** or **"Learner"**, and then restart the server. This will cause the server to return the appropriate token when accessed through the non-Canvas endpoint. To use the create exam view in a more realistic state, change the **token1** variable in the **fs.readFile** statement to be **token2**.

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 that can be served from the server.
12. To start the application, start an instance of **WSL**, navigate to the **server** folder within the **CodingExam** 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** 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 Exam 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.

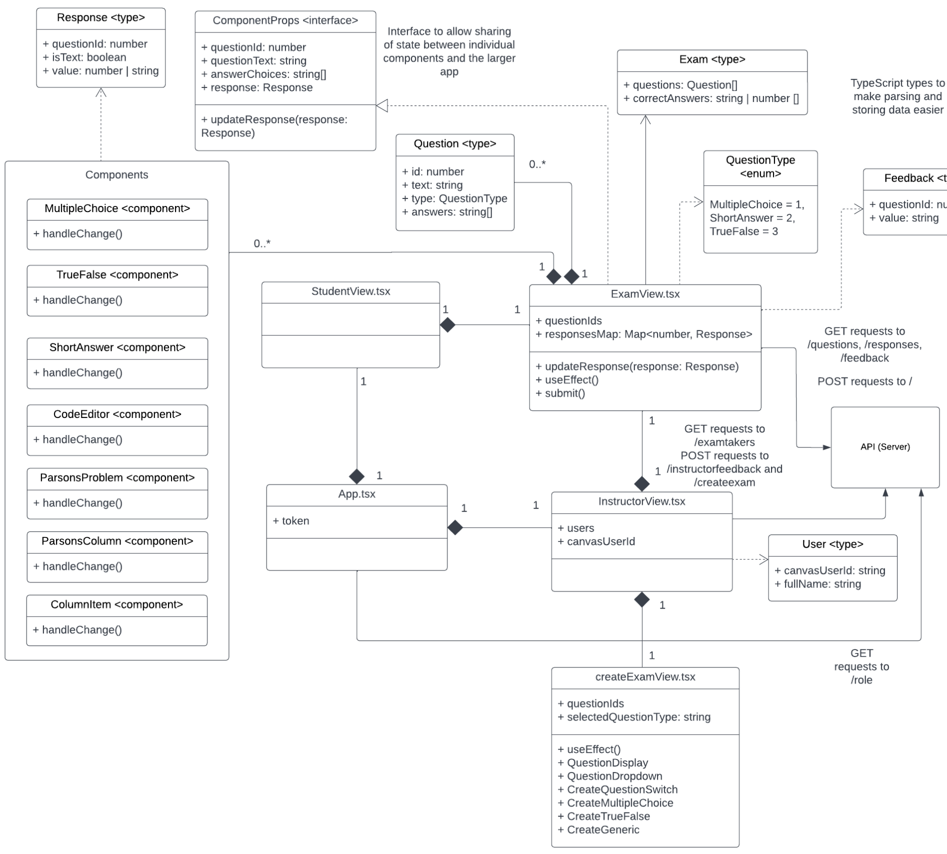
Each component is given a unique id that allows it to interact with the Redux store. The Redux store contains eight major components: **questionIds, questionsMap, responseIds, responsesMap, feedbackIds, feedbackMap, nextQuestionId,** and **token**. **questionIds** holds an array of unique ids for each question in the exam, and **questionsMap** maps each of these ids to the actual Question object that holds its information. **responseIds** holds an array of unique ids for each response the student has given, and **responsesMap** maps each of these ids to the actual Response object in the store. **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**. 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/responses** 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/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/instructorfeedback** 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/createexam** is made using their token and passing the list of questions.

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. 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 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. Code to generate instructor and learner tokens is present in this endpoint, but they use hardcoded userID, examID, and role values as those are sourced from Canvas. For both the GET and POST endpoints, a JWT token is generated using the userID, examID, and role values, whether they originate from Canvas or the hardcoded values, 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 UserExam table if a student is opening the exam for the first time.

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 client's JWT as a header. It will then decode the JWT to get the examID, 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/responses** 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 expects a request containing the client's JWT as a header, and if the client is an Instructor, the request will also need to contain the userID of the student whose responses are being loaded. The JWT is then decoded to get the userID, examID, 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 expects a request containing the client's JWT as a header, and it will decode the JWT to get the client'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 it expects a request containing the client's JWT as a header and their responses as the body. The server will then decode the JWT to get the userID of the student and 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.

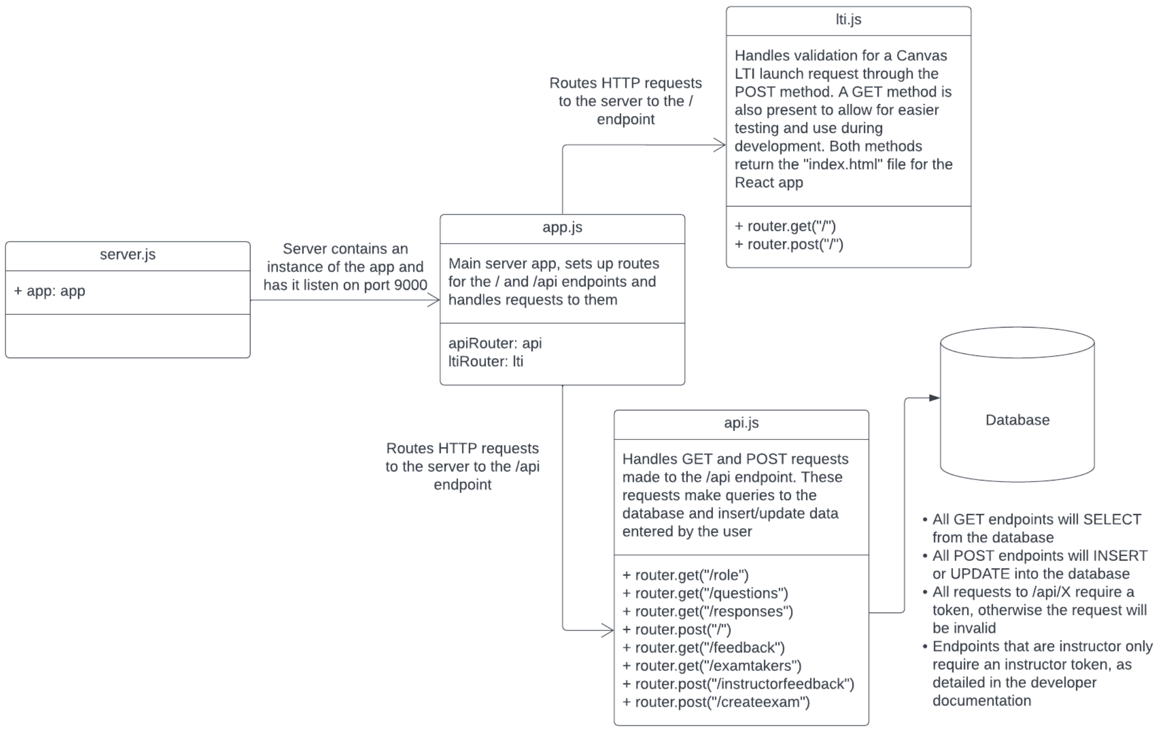
A GET request to **/api/feedback** occurs when a client loads a student's exam after feedback has been left, and it expects a request containing the client's JWT as a header. If the client is an Instructor, they will also need to send the userID of the student they are retrieving feedback for. The server will then decode the JWT to get the userID, examID, and role of the client and query the **StudentResponse** 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 will be used by both the Instructor and Student views, but it is currently only used by the Instructor view.

A GET request to **/api/examtakers** occurs when the Instructor view is loaded and returns a list of students who have taken the exam the Instructor has selected. It expects a request containing the client's JWT as a header, which it will then decode to get the examID of the exam and the client's role. 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 **UserExam** table of the database for the list of students and return the list to the client. This endpoint is only used by the Instructor view.

A POST request to **/api/instructorfeedback** occurs when the Instructor submits the feedback left for a student, and it expects a request containing the client's JWT and the userID of the student as headers. The JWT will then be decoded to get the client's role, and 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 **StudentResponse** table with the feedback left by the Instructor. The endpoint is only used by the Instructor view.

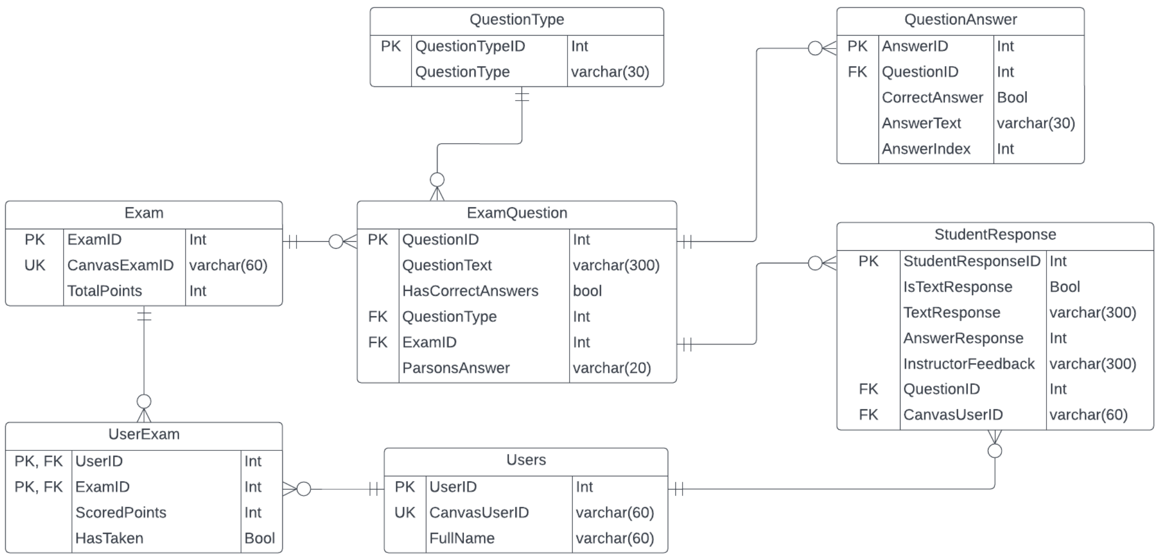
A POST request to **/api/createexam** occurs when the Instructor has created the questions for a new exam, and it expects a request containing the client's JWT as a header and the questions and answers for the exam as a JSON string in the body. The JWT will be decoded to get the internal Canvas ID of 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. This endpoint is only used by the Instructor View.

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.


lti.js
Handles validation for a Canvas LTI launch request through the POST method. A GET method is also present to allow for easier testing and use during development. Both methods return the "index.html" file for the React app
+ router.get("/")
+ router.post("/")
Routes HTTP requests to the server to the / endpoint
server.js
+ app: app
Server contains an instance of the app and has it listen on port 9000
app.js
Main server app, sets up routes for the / and /api endpoints and handles requests to them
apiRouter: api
ltiRouter: lti
Database
api.js
Handles GET and POST requests made to the /api endpoint. These requests make queries to the database and insert/update data entered by the user
+ router.get("/role")
+ router.get("/questions")
+ router.get("/responses")
+ router.post("/")
+ router.get("/feedback")
+ router.get("/examtakers")
+ router.post("/instructorfeedback")
+ router.post("/createexam")
Routes HTTP requests to the server to the /api endpoint
• All GET endpoints will SELECT from the database
• All POST endpoints will INSERT or UPDATE into the database
• All requests to /api/X require a token, otherwise the request will be invalid
• Endpoints that are instructor only require an instructor token, as detailed in the developer documentation

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. Finally, a **FullName** column is used to store the first and last name 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 **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 **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, **ParsonsAnswer** represents the correct order of a Parson's Problem 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 I**sTextResponse** 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, and the **InstructorFeedback** column is used to store feedback left by the instructor for a response.