Canvas Quick Quiz Grader

Field Session Summer 2019

Andrew Bukowski
Shaine Carroll-Frey
Clara Larson
Elliott McCabe

Client: Colorado School of Mines, Dr. Jeffrey Paone

Tuesday, June 18, 2019

I. Introduction

The Canvas Quick Quiz Grader was conceptualized by Dr. Jeffrey Paone, a professor in the Computer Science Department at the Colorado School of Mines (CSM) and the client of this project. Currently, CSM uses Canvas as its Learning Management System (LMS). As an LMS, Canvas is used by students, staff, and faculty across campus to facilitate student learning. Canvas allows professors to keep an online gradebook, administer online quizzes, provide students with course materials, and more. When CSM switched its LMS from Blackboard to Canvas, certain functionality was lost in the move. One major component that Blackboard has that Canvas does not is the ability to grade online quizzes by question. That is, a grader can look at all student responses to, say, question number one, all on one web page with Blackboard. This is especially useful in grading free response and fill-in-the-blank questions. However, the current online quiz grader, the Canvas SpeedGrader™, only gives graders the option to grade quizzes by student. This is also an option with Blackboard, where each student's entire quiz is graded at once, similar to how hardcopy quizzes are graded by hand.

In order to expedite Canvas online quiz grading, the functionality to grade quizzes by question is desired on Canvas. Thus, the Canvas Quick Quiz Grader Learning Tools Interoperability (LTI) application was created. This application has several features, but its main goal is to provide graders with the option to grade by question. It also gives graders the option to create interactive rubrics to grade with, to regrade a quiz given a new correct fill-in-the-blank response, and to anonymize students while grading. With these and more features, the Canvas Quick Quiz Grader will, ideally, be deployed to CCIT and integrated into Canvas as an LTI app to be used on the CSM campus.

II. Requirements

There were several important functional and nonfunctional requirements specified by the client. There were three main goals that we aimed to complete throughout this project related to the functional requirements. First, the ability to grade quizzes by question, the main feature, is the most important functional requirement. Essentially, this means that graders can grade one question at a time, rather than one student at a time. The question is displayed at the top of the page, and all student responses, as well as grading information, are listed below. In addition to this, the ability to anonymize students while grading was a desired feature. For example, when grading a section comprised of n students, student names are replaced with "Student 1," "Student 2," and so on through “Student n” and student responses are randomly shuffled to ensure anonymity.

The second functional requirement is the auto-regrade feature. When grading fill-in-the-blank questions, a grader may find that a student input a correct answer that was marked wrong because of varying spacing or punctuation. This does not necessarily make the student’s answer incorrect, but the answer varies from the correct answer options stored in Canvas, so it is automatically marked incorrect. In normal quiz grading, professors can add a new correct answer option to Canvas and manually regrade student by student to ensure that all students get correct credit. However, the second requirement of the Canvas Quick Quiz Grader is that it has an auto-regrade feature. This feature will automatically regrade all the questions of the selected quiz for the students in the selected sections. So, for example, if a student answered “yes,” but the correct answer stored in Canvas is “Yes,” all the professor will need to do to ensure that the student gets credit is add “yes” to the answer bank in Canvas, then run the quiz through the auto-regrader.

The third functional requirement is the interactive rubric feature. This will expedite grading by allowing graders to simply click on a rubric option to both assign points to a student and leave them a comment. This can be done for each free response question. The rubric is shown on the right side of the grading page, where it can be interacted with to grade each student’s response. The grader can create a rubric for each quiz they want to grade with the Canvas Quick Quiz Grader, and can edit and delete rubrics as necessary. When grading is complete, the grade information is sent to Canvas and posted in the gradebook automatically.

The non-functional requirements, while less important to the client, are still important to handle and follow. Considering that the end goal of this project is to integrate the application with Canvas for CSM professors to use, the project must adhere to guidelines from several sources, namely the CSM IT Department—Computing, Communications, and Information Technologies (CCIT). One thing that CCIT requires is that the C# component of the project runs on a .NET framework. They also require that the application is Windows-based, since Windows is the OS that is used on the CSM campus. The Family Educational Rights and Privacy Act (FERPA) is a law that requires that student data is kept private, except for certain authorized parties. So, we need to ensure that no sensitive student information, such as student grades, is accessible to unauthorized parties. Finally, Canvas uses OAuth2 authentication, so this application needs to follow this trend in order to work correctly with Canvas when integrated as an LTI application.

III. System Architecture

The architecture of the system follows a model-view-controller (MVC) structure. *Figure 1* shows the layout of the MVC structure and how each different class communicates with one another, with an outside entity like Canvas, or to the final end user view at the webpage. The main webpage is used to allow the end user to select a given quiz and to limit grading to specific sections. This data is then sent back to the homeController and it gathers the needed report from Canvas and filters data to fit the requirements that were entered.The main architecture of the implementation follows the Model-View-Controller architecture pattern for user interfaces and web applications. However, because of the dependence on updating an external site there is an external section to represent this functionality.

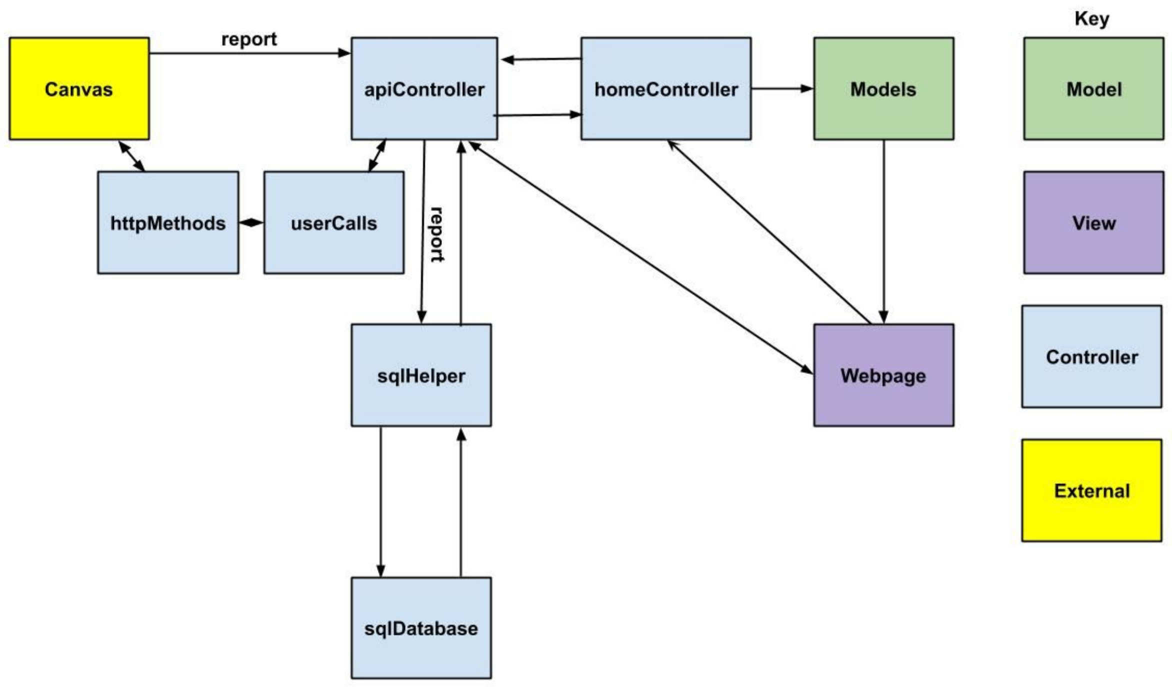

Figure 1: Project MVC design structure

Components of Architecture:

Model:

The role of this component of the architecture is to store data from the controller, and then pass that data to the view where the user can see and interact with it. This is accomplished by creating a new object of a Model class in the controller and filling in the needed data within that class. This object is then returned to the webpage where the user can access the data as needed.

View:

The view is what is displayed to the end user. This is done using HTML and JavaScript to load all of the data from the model correctly. This component gets all of its needed data from the model assigned to that page. The view is also used to adjust what the controller does generally through the use of forms and by submitting them or by also using AJAX calls.

Controller:

This component is where the bulk of the data is processed at. This involves both retrieving the data from Canvas and then parsing and modifying the data to store within a SQL server. The controller is also tasked with updating scores to Canvas. Specifically, apiController updates the score in Canvas whenever it gets called from the view. For example, whenever a user updates a student's grade in the application, an API call is made to update that student's score in Canvas as well. Within the autograder this is all done automatically with the calls being sent as the data is being processed and later shown on screen with the results via a model object.

The apiController is the controller that handles API calls to Canvas as well as calls from the web page to perform updates to the web page when more data is needed. For example, once a quiz is selected, more data is needed from Canvas. So, the view sends a call to the apiController to gather the needed info where it is then returned directly to the view and the web page is updated.

The homeController is the controller that sets up the page as it is loading and gets all the needed information into the model for the view to use. Each page has its own function to setup a model to use for the page. When a user is directed to a page it calls its specific function to get its needed model where the necessary information is stored.

External:

This component is not normally in a Model-View-Controller setup. Rather it is included to represent the end goal of this implementation to receive and modify data from a separate site. It is separated from the controller due to the limited access to Canvas and the inability to change much about it. In this case to get quiz data from Canvas and to show that data in a more meaningful fashion, and to then update pieces of the data on Canvas. In comparison to the other pieces of the architecture this is a small piece as this code cannot be modified. Rather, the only modification of the external site that is possible is through API calls, which simply retrieve or update data that is already on Canvas.

IV. Technical Design

One interesting aspect of the final design was when most of the waiting would occur. In the initial setup there were two occurrences in which a user would have to wait for backend processing to be done. The first was immediately after a quiz was selected to load all the student data into the local system to be used and then finally display the questions for that quiz. The other occurrence was with filtering out the students. The wait time here happened where multiple API calls to Canvas were needed to filter out sections that weren't selected and to link the student name to the Canvas id. This design was changed such that both waiting times would be performed after the form was submitted and the next page was launched. This allowed for the user to quickly choose their options instead of waiting around for the needed data to be downloaded from Canvas. This also addressed the issue of data persistence as the data would only actually be downloaded once the user needed it. Preventing the ability of a misclick to download data to the server that was no longer needed. This new flow is shown in *Figure 2* where the download report now comes after the launch rather than the select question.

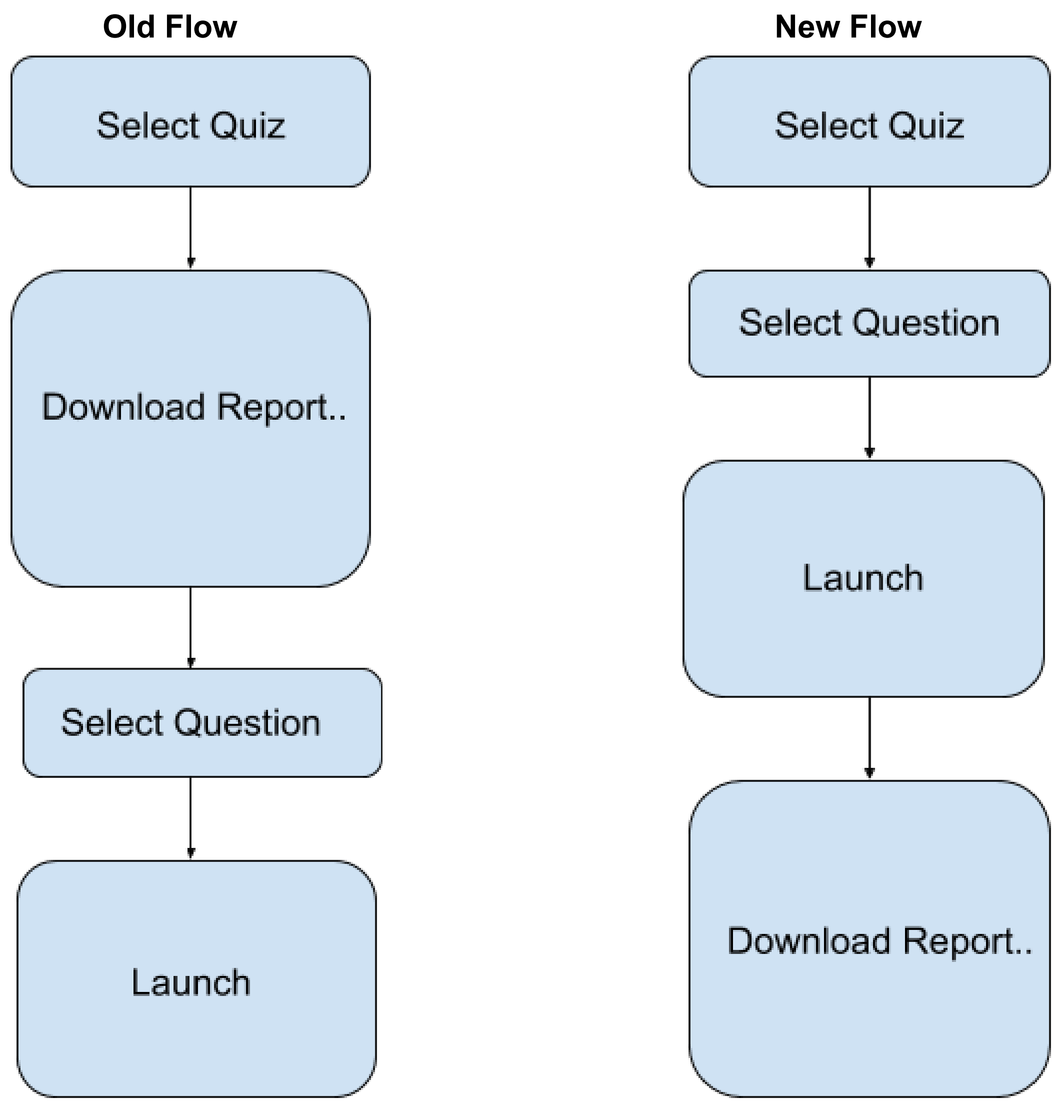

Figure 2: Backend flow design changes

Another interesting aspect of the final design is the specific API calls that were used. In the instance of getting the students in a specific section, two different API calls had drastically different performance. The initial one that was used was the enrollment API call that contained a large amount of unnecessary data which resulted in the call taking a longer time to perform. This call could take so much time that a single section containing only 40 students could take one and a half seconds. While seemingly not very much time given how many sections there are, this amount of time would eventually add up. So instead a different API call was chosen. This API call was instead used to get the specific section and by adding a parameter to the call could also gather all the students in that section. The amount of data included with each student was much smaller and contained everything that was needed as only roughly a fifth the amount of JSON argument objects were included. By changing this call around such that less data was transferred there was a 5-fold decrease in time taken from 1.5 seconds down to 0.3 seconds per each section. The change in API call is shown in *Figure 3* where the new call results in a faster response time.

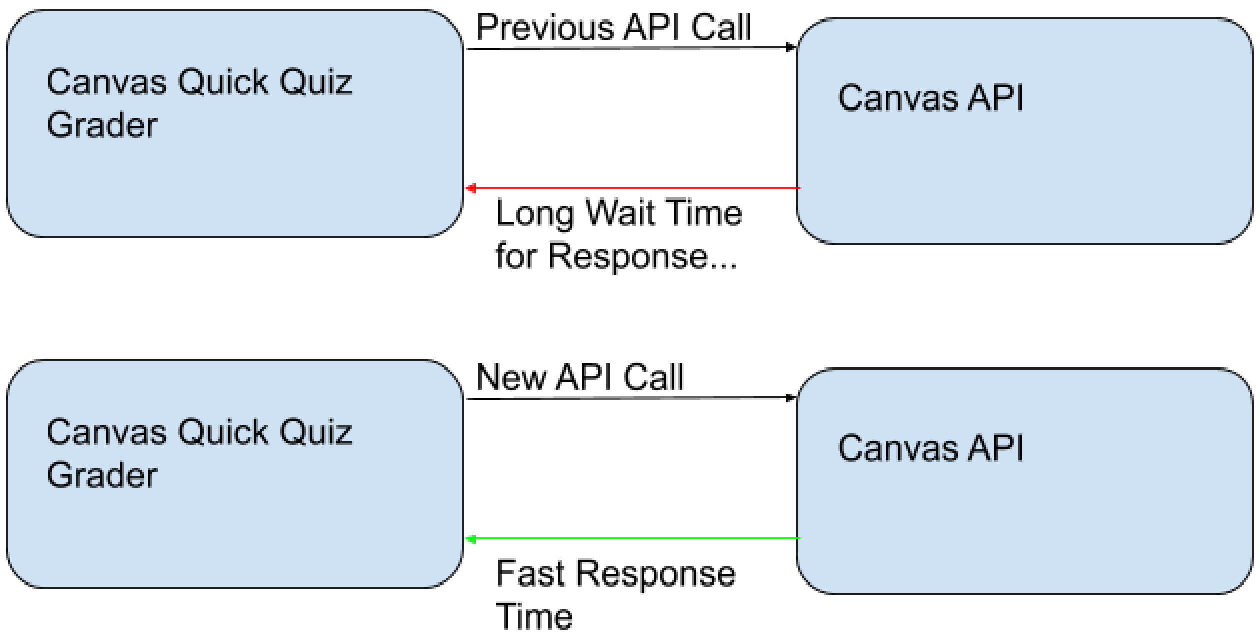

Figure 3: API call changes

V. Quality Assurance

Testing

Testing for this project was very rudimentary compared to testing approaches used in today's practices. That is, automated testing and unit testing were not used during the undertaking of this project. Instead, application, user interface, and user acceptance testing were used. Application testing included manually testing the application in order to try and find bugs that an every-day user might come into contact with and then solve them before the project is pushed to the user. Some examples of this are changing a student’s score and comment in the program and then checking to see if the score and comment were changed in the Canvas gradebook. Another example of application testing is creating different kinds of rubrics with different amounts of criteria, options, names, and even points per option.

For user interface testing it was important to make sure that the project was appealing to the user and not an eyesore. It was also important to make sure that the interface worked as intended. To make sure the user interface worked as intended all of the following were tested:

- Size, position, width, height of elements
- Font readability
- Error messages and confirmation prompts
- Different screen resolutions
- Zooming
- Alignment of elements
- Colors of fonts, images, elements
- Spelling
- Scrollbars
- Ease of use

These elements were tested by manualing changing an aspect of each respect and then relaunching the application to make sure that everything looked correct.

The last type of testing, user acceptance testing, took a broader approach. To test this, it was asked if the system can support real-world scenarios. This meant to solve any problems that could come up from the following prompts:

- Can the user use the software?
- Is it really what they asked for?
- Do they have trouble using it?
- Does it behave exactly as anticipated?

Complying to these prompts meant to constantly stay in contact with the client and make sure that the vision they had for this software was met every step of the way. This was done by attending meetings with the client once or twice every week and displaying what had been worked on and if anything needed to be adjusted to their liking. Should the client not enjoy what was presented, they would work with the team to make sure that the project lived up to their expectations.

VI. Results

Unimplemented Features

Unfortunately, due to time constraints, not all of the desired features of the Canvas Quick Quiz Grader were able to be implemented. The most notable feature which wasn't implemented was the final integration into Canvas. The final product is hosted locally and pulls data from Canvas using API calls, but is not actually hosted on the Mines Canvas site. As such, a professor who wishes to use the application must go through the attached documentation (Appendix A) in order to set up the necessary environment in order to run the program. Smaller features not implemented include CSS and HTML5 validation, which validates that all pages in the application adhere to CSS and HTML5 conventions and that everything runs properly.

Lessons Learned

At the conclusion of this project, all members of the group have become better developers and have gained more experience working in an Agile environment, as well as technical knowledge of the .NET web framework. The biggest takeaways we had as a group are:

1. In-sprint bugs will happen, so take them into account when planning for a smoother development process. These were difficult for us to anticipate, especially since most of our group was very inexperienced with the skills that this project required. Having this experience has taught us to plan for in-sprint bugs.
2. Agile is the dominant methodology in software development for a good reason, make sure to utilize it properly. It was the biggest reason that we were able to deliver a product that made Dr. Paone happy. Getting constant client feedback helped us to work on things as efficiently as possible.
3. Leaving code in a clean state with proper documentation is of utmost importance, as it’s very likely someone will have to maintain and/or expand on that code later. This will reduce the time that future teams need to spend digging into your code and understanding what is going on. The team that worked on this project before us did not leave good code or documentation, so we made sure to clean it up and leave it better than we found it.

Future Work

As the final product is a local program and not an actual LTI application hosted on the Canvas site, the first and most important extension to our project would be adding it to the Mines Canvas site for any professor to use. After that, making the application available as a Canvas extension to all universities currently using Canvas would further extend the product.
Another piece that could be improved for this project is the UI. Currently, the pages have a lot of white space and some pages take on the appearance of a 2000’s era webpage. The white space would be fixed however, if the application is applied as an LTI application within Canvas.

VII. Appendix

Appendix A: Product Installation Instructions

This is a guide to get the Canvas quick quiz grader setup to run on a local machine and to access a live Canvas course. This guide assumes that you have downloaded or cloned the git repository to your local machine.

Requirements:

- Windows machine
- SQL server

SQL Server setup:

- Download and install sql server. This was done using the evaluation 2017 edition of SQL server.
- Next create a database within SQL named oauth2test
- Next there are three .sql files within the repository. Run all three on the database.

This is all that needs to be done directly on the database

Connection String:

If you know how to create a connection string for the database you may skip this step.

- Otherwise start by creating a new file named test.udl.
- Open up this file and navigate to the connection tab if not already there. From there click on the drop down and select the server you want to use. This step may not be necessary if only one SQL server is installed.
- Next make sure that windows nt integrated security is selected. This may eventually change to using a specific login and username once stuff is figured out
- Next click on the dropdown and select oauth2test
- Click on the test connection and if it succeeds the connect string has been generated.
- To view the connection string simply open up the .udl file in a text editor. The connection string will be everything past the first semicolon on the third line

Config Setup:

- Now that you have the connection string copy that to the web.config file located in the base directory of the repository.
- If you are going to run this locally you will place it under the dev option. Otherwise place it in the oauth2 section.
- Next go to the global.config located within the config folder. Change the mode how this site will be handled. Likely you will want it to be local mode.
- If you choose local mode you will need to go to Canvas and generate an access token.
- This is done under the account settings tab on the left of Canvas and then located within the settings tab.
- Copy this access token into the localMode.config under personal token. Also enter the courseID for the course that this grader will access.

- Fill in the userID for the instructor and the host url that the Canvas instance is run under. Do not include http or https with it. The host url should look like “Canvas.institution.edu” or “elearning.institution.edu”

Folder Setup:

- Within your C drive create a folder named qqg_temp_data directly under the C drive. This will be the location that reports are downloaded to and will be deleted from.

Launch:

- Using visual studio you can run the application locally and test it through an internal IIS.
- Make sure to launch from a file that does not require any fields to be filled out in order to reach, such as Grade.cshtml, performReGrade.cshtml, or editRubric.cshtml.
- Instead, a safe file to launch from would be either one of the controllers.
- Otherwise you will have to deploy it somewhere and run it there.