

Quick Quiz Grader

Colorado School of Mines Field Session 2018

Mark Beaty, William Brown, Michelle Hulongbayan, Tanner Jones
Dr. Paone

Table of Contents

Introduction	2
Requirements	2
Functional Requirements	
Non-Functional Requirements	
System Architecture	3
Overall Design	
The Home Page	
The Grade Page	
The Rubric Page	
The Help Page	
Back End Summary	
The Backend: Authenticating a User and why that's hard	
The Backend: Storing SQL and Injection for Beginners	
The Backend: Writing an API to AJAX about	
Technical Design	9
The Rubric Section of the Grade Page	
Saga of the API	
Design Decisions	11
LTI app vs Standalone app	
Programming Languages	
Web Page Design	
Results	11
Unimplemented features	
Testing	
Future Work	
Lessons Learned	
Appendices	13
Appendix A: Initial Diagram of Functionality	
Appendix B: Initial Layouts	

Introduction

Client and Current Product

Our client, Dr Paone, is a professor at Colorado School of Mines in the Computer Science Department who teaches CSCI-261 (Programming Concepts) and CSCI-445 (Web Programming). He uses Canvas quizzes a lot in those classes and noticed a lack of efficiency in the currently implemented grading methods of Canvas.

Canvas has a pair of existing graders, both of which are flawed. Moderate This Quiz is a bare-bones option for grading. It allows teachers to grade quizzes by the individual submission of the student. To navigate between students, the program forces a user to navigate back to the quiz, select the application again and select the next student. It does, however, allow a grader to grant extra attempts on a quiz fairly easily, but that is the only major benefit of using the application. SpeedGrader is another such application that allows grading per student showing their entire response to the quiz in one window. When grading in this application, there are long load times to switch between the student's responses. This leads to many teachers like Dr. Paone having to spend exorbitant amounts of time grading quizzes. In addition, a rubric is allowed to be parsed into the SpeedGrader but this rubric is only shown with no usable features embedded. Comments can be easily attached from this grader for all of the responses to the quiz as well as each individual question. There are a lot of options in the application for selecting how the student responses are sorted, however, the application has the major impediment of grading by student.

Product Vision

Dr. Paone's goal was to create either an external application or an integrated extension of Canvas to help provide faculty with a faster grading process. The application would allow instructors and graders to grade free-response and fill-in-the-blank quiz questions more efficiently through use of an old feature found in Blackboard. This would be accomplished by replicating the previously used Blackboard Learning Management System (LMS) where graders could view every response to a single question and grade all students at once which is a feature that does not occur in the existing graders. In addition, he wanted a rubric that included click events to give basic feedback and populate the grade field based on specific values attributed to the rubric.

Requirements

Functional Requirements:

The project has a list of functional requirements that the app needs to accomplish:

- Ability to select a class and section
- Ability to select a quiz and question for grading
- Auto grading with use of text filters listed below
 - Remove whitespace
 - Remove capital letters
- Highlight multiple searched phrases through a visual/syntactic highlighting feature embedded in the source code (not just Ctrl+F)
- Help page

- Display and allow the application of the rubric for the quiz through element-selection grading and attach a comment including the rubric to that individual response
- The ability to shift through entries using Tab and Enter keystrokes to allow for quick data entry
- The push and pull of grade data through use of the Canvas API

Non-Functional Requirements

The nonfunctional requirements were for ease of use of the end user and to make it so it is easier for people to look at.

- Accessible to those with color blindness
- User Friendly Graphics and Font
 - Scalable with web page zoom
- Quick entry to move between fields
 - Mice or keyboard interface

System Architecture

Overall Design

The overall design is as an LTI app that launches within Canvas. The LTI app was named the Quick Quiz Grader. The app has multiple pages that are used to create the app. The blue navigation bar is consistent through all of the pages. The bar has a home and a help button which link to their respective pages. This bar can be shown in Figure 1.



Figure 1: The Navigation Bar

There is a lot that it took to interact between the GUI and the back end. The back end includes the API calls to canvas, the SQL databases, and the AJAX calls. Figure 2 shows these interactions.

Figure 2: Interactions between Backend and GUI

The Home Page

The home page is where most of our functionality takes place. This page's functionality is detrimental to the rest of the application functioning properly. The first drop down under the Quiz title is used to select a quiz. These quizzes are populated from the course that the application is opened in. The Section selection section then reveals various checkboxes that allow the user to select the different sections they wish to grade, allowing for individual TAs to ensure they only grade their selection of students. The Question drop down is also populated after selecting a quiz with all the different questions of the quiz, allowing a user to specify a starting point if the quiz has already been partially graded. The Home page is shown in Figure 4.

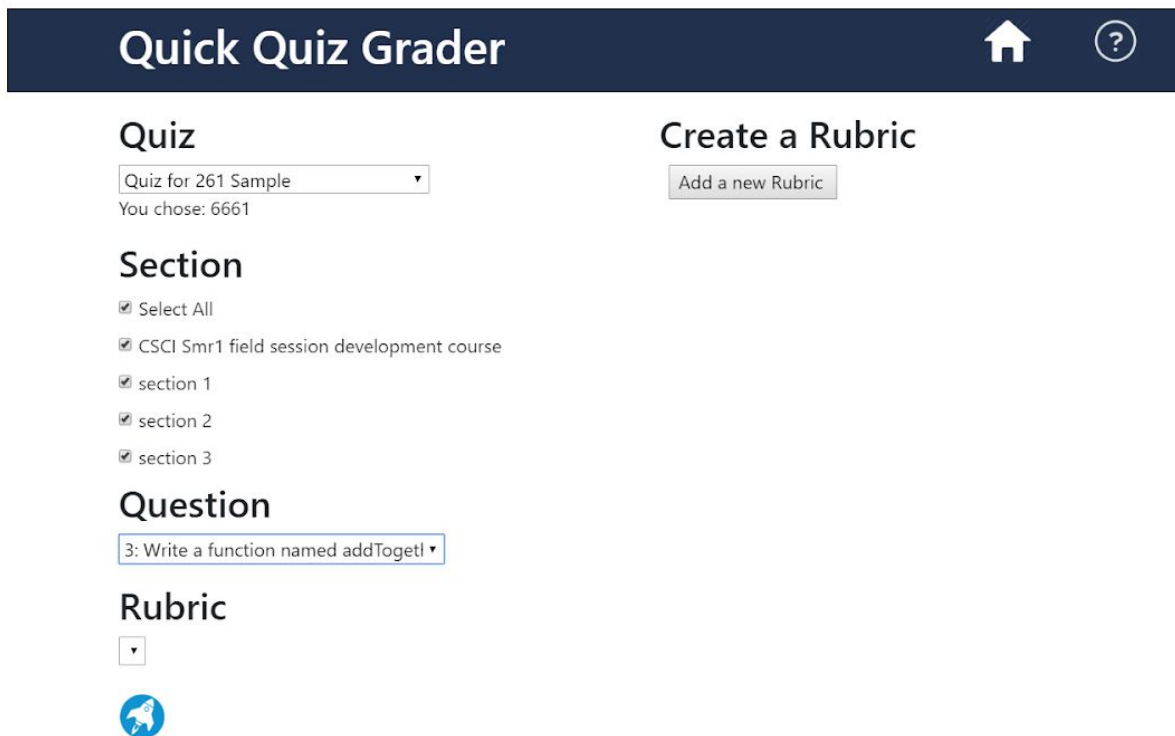


Figure 4: The Home Page launched through Canvas.

The Grade Page

The grade page is the meat of this program's functionality. At the top of the page resides the title bar along with the home button and help button which route you to the respective pages. Below the blue title bar we get into the fun stuff.

The quiz's name is displayed in this section as well as a bar of buttons that allow a user to select which question they are grading. This is made in part to mimic the current functionality of the SpeedGrader mentioned earlier in this paper, as it is a functionality that a lot of faculty enjoy.

Under the quiz name resides the question name, giving a reference for where the user is in the quiz and allowing them to better grade questions that do not have a rubric element attached to them. The essential feature of our grader sits below the question name. It's an array of student responses with information populated through api and ajax calls.

The left hand side of the response container holds the student name, the center holds the student's response, and the right hand side holds the grade of the student. The grade box of a response is a changeable field that can be altered from inside the box itself or through use of the right hand side.

On the right hand, near the top of the window, are a couple of buttons used to implement a couple secondary features for our client. Highlight keyword acts as a Ctrl+F command that stays on the page to assist graders that are looking for specific hot words to grade off of. Jump to is a button that allows a user to enter an index of a response and simply highlight it, making jumping around the responses or returning to a response much more simple for the user. Under the buttons on the right is the rubric section. In Figure 5.0, we showcase what happens if a question of a quiz does not have a rubric associated with it. The page, on load, determines if the question has a rubric or not and either displays it like in Figure 5.1, or displays the error message seen below.

Under the rubric is the three buttons for working with the state of a highlighted response. Revert works like a nuke button. It allows the user to clear all comments, grades, and rubric states from the right hand side, giving a fresh start to the grader. The save button works as one would expect, saving the state of the response. Save and next implements save, but also moves to the next response in the list and updates the comment box and grade on the right hand side accordingly. This takes us down to the comment box. This is populated automatically by the rubric but also gives the user a chance to add comments to each individual response. The clear button below it is used to clear just the comment field. The grade box at the bottom of the right hand side is used to grade the highlighted response and manually enter a grade. This can also be populated by the rubric if one exists and is used.

The grade page is shown in figure 5.0 and 5.1

Quick Quiz Grader Home ?

Quiz for 261 Sample << 1 2 3 4 >>

Complete the missing line of code to print out the value of x, ending with a new line.

0. 10873	<input type="text" value="cout << x;"/>	Grade: <input type="text" value="8.0"/> / 1	Highlight Keyword Jump to... Rubric not found Rubric Revert Save Save&Ne <input type="text"/> Clear Grade: <input type="text" value="8.0"/> / 1
1. 10875	<input type="text" value="cout << x << endl;"/>	Grade: <input type="text" value="4.0"/> / 1	
2. 10874	<input type="text" value="Select x from main where x='5'"/>	Grade: <input type="text" value="0.0"/> / 1	

Figure 5.0: The Grade Page without rubric

Quick Quiz Grader Home ?

Quiz Name << 0 1 2 3 4 5 6 7 8 9 >>

Question name

A	<input type="text" value="Alpha"/>	Grade: <input type="text" value="0"/> / 20	Highlight Keyword Jump to... Autograde and Filter Rubric <table border="1"> <tr><td>a</td><td>b</td><td>c</td><td>d</td></tr> <tr><td>e</td><td>f</td><td>g</td><td>h</td></tr> <tr><td>i</td><td>j</td><td>k</td><td>l</td></tr> <tr><td>m</td><td>n</td><td>o</td><td>p</td></tr> <tr><td>q</td><td>r</td><td>s</td><td>t</td></tr> </table> Revert Save Save&Next <input type="text"/> Clear Grade: <input type="text" value="0"/> / 20	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t
a	b	c		d																			
e	f	g		h																			
i	j	k		l																			
m	n	o		p																			
q	r	s		t																			
B	<input type="text" value="Beta"/>	Grade: <input type="text" value="0"/> / 20																					
C	<input type="text" value="Charlie"/>	Grade: <input type="text" value="0"/> / 20																					
D	<input type="text" value="Delta"/>	Grade: <input type="text" value="0"/> / 20																					
E	<input type="text" value="Echo"/>	Grade: <input type="text" value="0"/> / 20																					
F	<input type="text" value="Foxtrot"/>	Grade: <input type="text" value="0"/> / 20																					
A1	<input type="text" value="Alpha"/>	Grade: <input type="text" value="0"/> / 20																					
B1	<input type="text" value="Beta"/>	Grade: <input type="text" value="0"/> / 20																					
C1	<input type="text" value="Charlie"/>	Grade: <input type="text" value="0"/> / 20																					



2018 - Colorado School of Mines: Computer Science Field Session Project: Quick Quiz Grader

Figure 5.1: The Grade Page with rubric

The Rubric Page

The rubric page allows user to create a rubric to help accelerate and improve the consistency grading across multiple responses. The rubric can be named and saved to be reused in the future. Currently, the rubric supports up to 100 questions with between 0 and 10 pieces of criteria per question and up to 11 point options earned for each criteria. This allows for custom rubrics with enough detail to provide a large coverage of all potential responses. The options are laid out as columns, with each criteria being represented as rows in the table.

The number of questions can be changed “dynamically.” Once a number is inputted, it shows the number of questions inputted. There’s currently a limitation of 100 questions on the web page, which should be a reasonable number for any quiz hosted on Canvas. Each question has the option to add it’s own rubric. These rubrics have a number of criteria and a number of options per criteria. Each criteria becomes a row in the grid, with each row having a column for every option per criteria. An example of the Rubric page is shown in Figure 6.

Quick Quiz Grader  

Rubric

Rubric Name:

Number of Questions:

Question 1

Question 2

Number of Criteria

Number of Options per Criteria

Criteria Name	Points <input type="text" value="2"/>	Points <input type="text" value="0"/>
Mentioned Alpha	Comment	Comment
	Did Mention	Alpha

Criteria Name	Points <input type="text" value="1"/>	Points <input type="text" value="0"/>
Grammer	Comment	Comment
	Good	Bad Grammer

2018 - Colorado School of Mines: Computer Science Field Session Project: Quick Quiz Grader

Figure 6: The Rubric Page

The Help Page

The help page gives crucial information about how to use the application, including information on the limitations of the Rubrics. This page also gives information on how to use the grading page, describing the basics of the page’s functionality. The Help Page is shown in Figure 7.

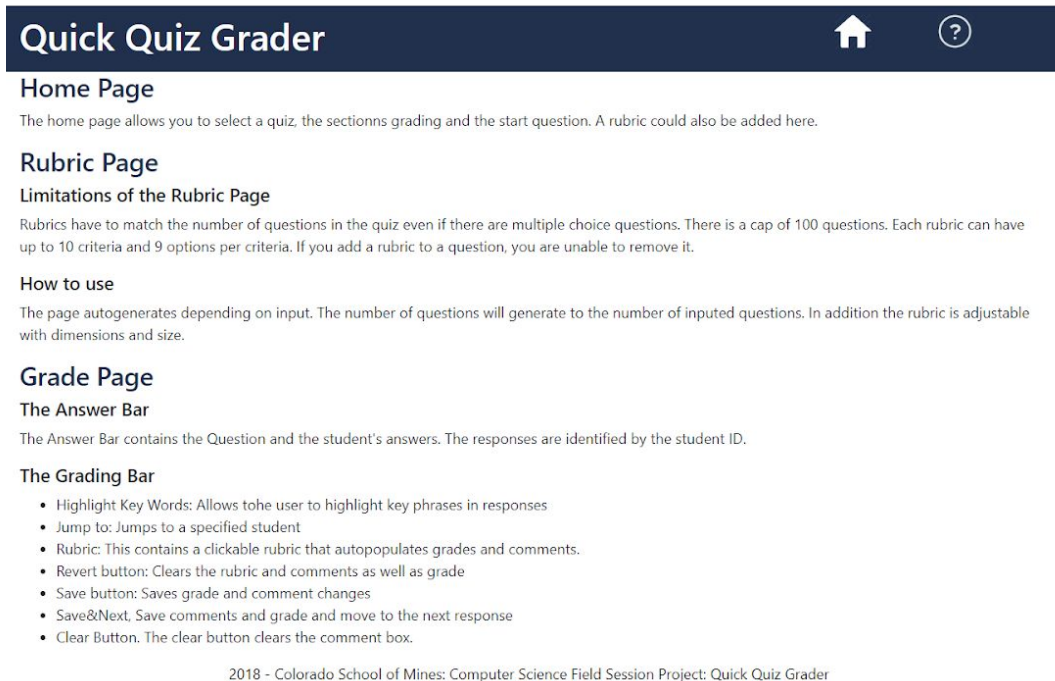


Figure 7: The Help Page

Back end summary

Like all modern web application, the application uses a Model-View-Controller schema for the core codebase. In this case, our application utilizes Microsoft's ASP.Net MVC to create a web application server. This design allows for all aspects of authentication and API calls within Canvas to be accessed remotely by the user's machine, making our application accessible to anyone who has access to the course our application is deployed to.

All API calls are normal https requests against the school's canvas domain. This includes the initial launch request, the entire OAuth2 authentication workflow, and every request for information call made while the user accesses our application. This also includes our put request to post grades back for a student's submission.

All web applications will make return calls to their home server, and ours is no exception. All pages make AJAX calls back to a custom written Restful API, not unlike that which we use to communicate with Canvas. This API makes extensive use of local storage, so a SQL database is attached for storage of access tokens, session data, and quiz data.

Quiz data was an interesting use of our SQL database, mainly because of the unforeseen requirement of needing to store anything at all. Due to failings in the Canvas API, student responses for a question are downloaded as a quiz report CSV file, which is then parsed and sanitized for semi-temporary database storage. All loads of the grade page and every one of our API calls leverage this database, exchanging information to serve a usable interface to the user.

Figure 8 shows the complete interaction map of our application with our database, the user's web browser, and Mines' Canvas instance.

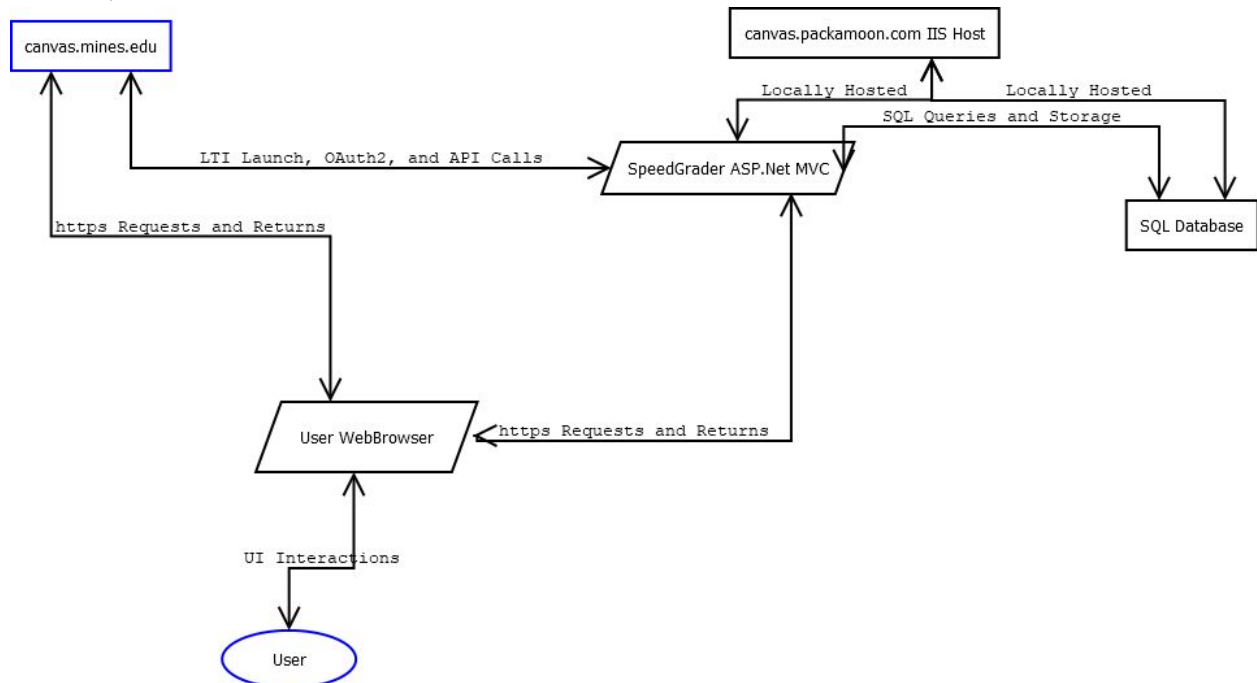


Figure 8: Description of the interactions hosted in the backend

The Backend: Authenticating a User and why that's hard

Canvas uses a modified OAuth2 workflow to establish user access and authentication. At the heart of this process is a Developer Key, a generated key and id value in Canvas that allows an application to request for a user's permission to masquerade as them for all API calls. If permission is given, the application receives a launch token to then query the Canvas OAuth API to retrieve an access token and a refresh token. The access token is good for 1 hour, allowing the application access to every API call the user has permission to execute. The refresh token is used to requery Canvas after a token expires to retrieve a new access token.

While the initial authentication process is only executed when the LTI application is launched, refreshing the token must be done before every call. This is done solely to prevent the application from using a bad access token, since a grading session can easily last longer than an hour.

The Backend: Storing SQL and Injection for Beginners

SQL is the most prevalent database language, and for good reason. From a wide variety of database server options to a large list of libraries to access a database for every language you can imagine, it's incredibly easy to use. All LTI applications must store and track a session state and authentication token package, and SQL makes the most sense for persistent managed storage.

Our application makes extensive use of SQL to store session information, access tokens, quiz submissions, and in-application created rubrics. A single database is stored on an instance of

SQL Server 2016, with tables for everything we could need. Access tokens and Session states each have their own table, with rubrics also having their own table. Each one has a small variety of an identifying string and a bit of relevant information. Some store everything in a json literal, while others have separate fields for everything.

Each quiz gets its own table, filled with every question, every relevant detail of every question, every student's response to every question, their associated grades, feedback, and all of the IDs needed to complete any and every API call we need. This forms the largest storage and IO requirement, due to the fact it replaces an estimated 4 different queries per page load against Canvas for information retrieval.

The application has a class of wrapper functions that complete all the required SQL queries, and do some basic parsing of the response to return only the relevant information the query required. This also allows easy re-use of all queries, regardless of where in the application the user is at. This also provides an easy space to sanitize inputs, making all user-enterable strings safe for storage and retrieval in a SQL database. This last part is important, since it's incredibly easy to create a response that is an accidental SQL injection, and as such just as easy to actually inject SQL. With sanitation, our database will actually store any code trying to act as a SQL injection, and simply treat it as normal text.

The Backend: Writing an API to AJAX about

Web applications often use AJAX calls in javascript to add functionality and user intractability without having to use page refreshes to submit and retrieve new information. These calls are only usable with a functional web API, typically a Restful API, to leverage against. Since our application aimed to remove all page reloads once a question was selected, a custom web API had to be written. While it only consists of a few possible calls, new calls are easily added to support any new functionality required.

The three current calls are responsible for a few critical features, primarily: getting a list of questions for a quiz; saving a new grade and feedback for a student's submission. The question list generation call is one of the most complex functions in the entire codebase. This call is responsible for retrieval of a quiz's statistics report, which is the only known way in the Canvas API to get a student's full submission text. It is also responsible for parsing said CSV file, storing said responses in the application's SQL database, the returning of a list of questions and IDs to allow user selection of a question, and the retrieval and association of a unique submission ID with every single student's response to every single question. This last bit is important for the second call.

The grade submission call is, in comparison, much simpler. It's given a student's grade for a question in a quiz with a session id to reference. From this, it must get the student's submission id from the SQL database, then put the updated grade and feedback comment back into the database, as well as leveraging one of the only working API calls for Canvas to add the new grade and feedback to that student's grade. This call is referenced a lot, with nearly every action done in the grading page activating an AJAX call to resubmit the student's score.

Technical Design

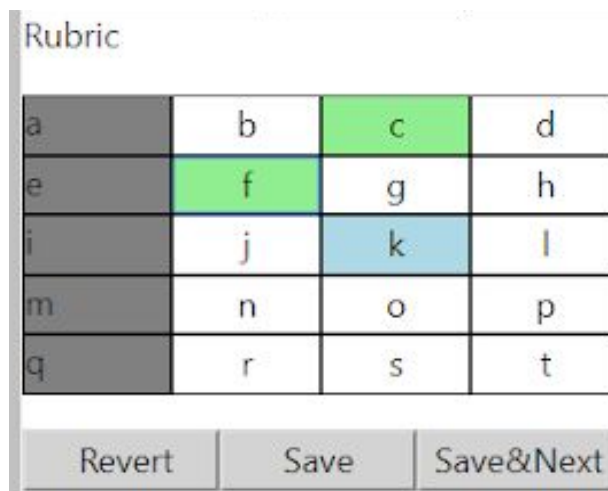
The Rubric Section of the Grade Page

The rubric section of the grade page is one of the most interesting features of the website. This feature's goal was to provide a quick and easy way of grading free-response questions, with a basic comment generation based on the criteria a response met. This feature is considered optional for grading, and Figure 9 shows the message if a rubric is not selected while grading a particular quiz.



Figure 9: Rubric not found Message

The rubric is read into the web page from a stored json literal of a map. The first column holds the criteria title for a row, with the rest of the row showing the potential point values based on their response. A value will become green if that option is selected. In addition, any unselected white cell will become blue on hover. Selecting one of the options adds to the grades and comments, depending on which option was selected. Figure 10 shows an example rubric.



Rubric			
a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p
q	r	s	t

Below the table is a horizontal bar containing three buttons: "Revert", "Save", and "Save&Next".

Figure 10: Rubric Grid on Grade page

Saga of the API

The foundation of every Canvas LTI Application is the Canvas API, a well-documented Restful API that allows easy access to all of the information Canvas as a Learning Management System maintains and serves. From quiz submission to a user's name, there is a call for anything you could want, at least according to the documentation.

Despite the large and thorough documentation, there is a lot to be desired. For example, Canvas offers a built-in Rubric tool for quizzes and assignments, but the Rubric API offers nothing of value. While the documentation is clear about the lack of information, the documentation is not always as helpful. Our application is a quiz grader, so a crucial piece of information is the student's actual submission for an answer. Canvas offers a beautiful Quiz Submission API, making grading and retrieving of submissions nice and easy. Unfortunately, the API documentation is incorrect, and a quiz submission object includes no actual submission. Work-arounds exist, such as creating a student analysis quiz report and then parsing the resulting CSV file, but that incurs additional storage and processing requirements, not to mention the long wait times on generating the quiz reports and then parsing it into a usable format.

The Canvas API is incredibly easy to interface with. All calls are standard http calls, whether it's a POST, GET, or PUT call. Each call includes the user's access token and a small JSON object of all information being passed. Nearly all calls return a JSON object, including a variety of parameters and status conditions. These JSON objects can be parsed to retrieve the desired information. Figure 11 shows an API call executed in Postman, along with the returned JSON object. This API call is used for updating grades and comments for a particular submission.

When Canvas launches our LTI application, we are given an initial launch request with a myriad of information. Much of this is incredibly useful, from OAuth validation tokens to contextual information about where our application was launched, including the course id and the user's role in aforementioned course. This information exchange is a one-time deal, and this information is never provided again. Given its relevance for every single API call that will be made for the life of a user's session, this entire object is stored and retrieved via a generated state Guid on the initial launch, before the request for an access token. This state id is then passed along in the application, and is the only piece that ties a user to their unique session.

Through the aforementioned and previously explained OAuth2 workflow, an access token for a user is generated and retrieved via API calls against Canvas. This token is passed in the authentication header of every API request against Canvas, and is incredibly powerful. As such, like the session data, the access token and associated refresh token are stored for later access, and are retrieved at all point in our application. However, due to the security concerns with such a token, this token is only ever sent to the stored Canvas domain the initial launch request originated from. The user never sees their own token, and cannot retrieve it from our web application.

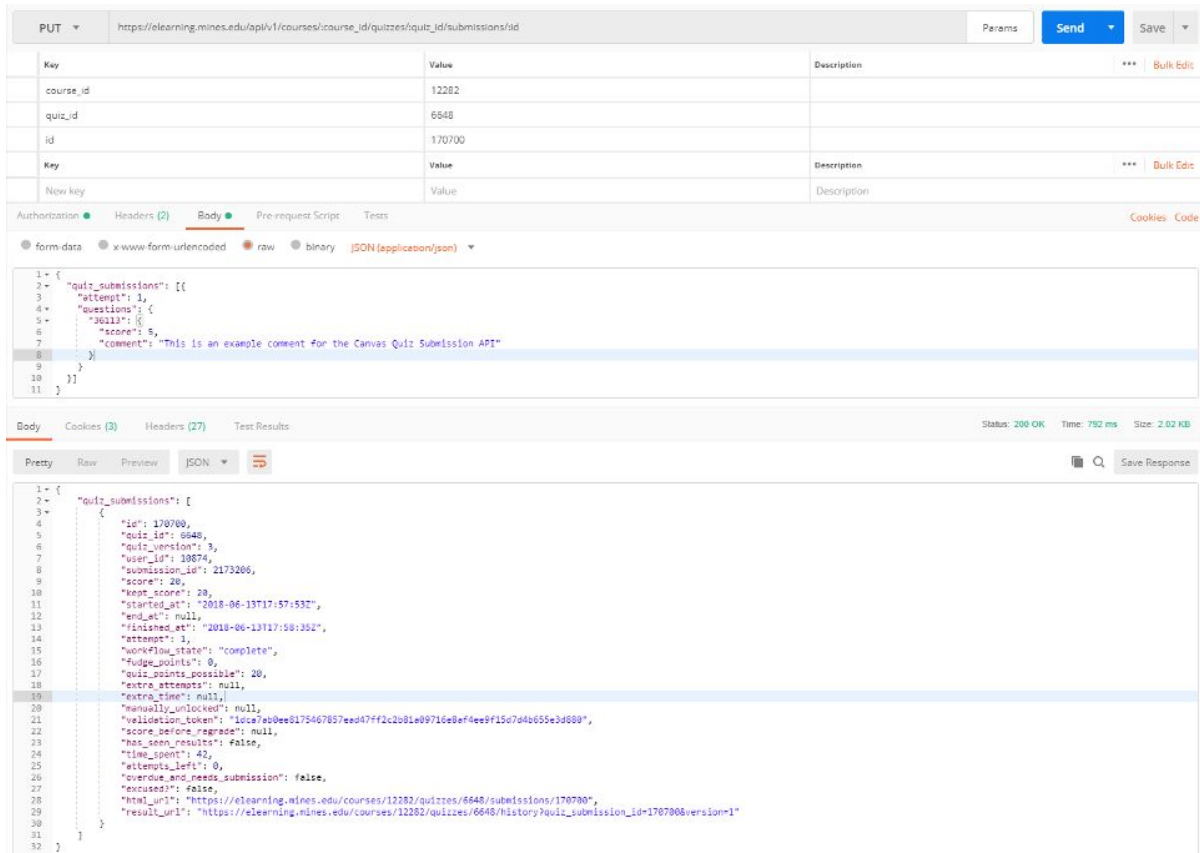


Figure 11: Example of an API call in Postman

Design Decisions

LTI app vs Standalone app

When the project was originally pitched, Dr. Paone was looking for an external application he would run on his machine. Through some cursory research, our team discovered the possibility of creating an LTI application, which would allow for a much larger potential user base due to the lower technical skill requirement needed to use the application. An LTI application also simplifies a lot of the OAuth workflow, and the contextual information from an application launch simplifies our user interface.

Programming Languages

While web applications like LTI apps use languages like HTML, CSS, and javascript for the in-browser interface, the backend web server can be written in a variety of languages, including Ruby, PHP, and ASP.Net. In our case, we chose ASP.Net, which is built on the .Net framework, and is compatible with nearly all C# .Net libraries. Like most web framework languages, ASP.Net utilizes an MVC schema, with a model defining a view and a controller to define the model. The view is pushed to a user's browser, while the controller executing all background logic including Canvas API calls. Between the readily available tutorials for all aspects of our LTI app, including OAuth authentication, to the team's familiarity with C#, it was the obvious choice for us.

Web Page Design

There were a lot of design decisions that took place with the appearance of certain elements. One of the recommendations given to us was to replicate the appearance of Canvas and the already existing SpeedGrader app on Canvas. Our visual design was a lot easier since we had something to use as a base. Most elements on the webpage were contained in divs that were set to a specific percentage width. This allowed the web page to easily scale.

Results

Unimplemented Features

There were quite a few features that were not implemented in our design. First, retrieval of previous feedback comments simply lack a documented Canvas API call to return them. Quiz reports lack them, and the quiz submissions API lacks anything useful. Secondly, coloring of a quiz name based on completion status is not implemented due to the aforementioned lack of useful API calls as well as a lack of time on our part. We had a large laundry list of secondary features that were not implemented as well, but it is too numerous to fit in the context of this document.

Testing

Our project performs all of the critical functionality our client wanted. The app is a user-friendly application with straight-forward controls with a few key features to streamline the grading process. Our project works with all major web browsers, with validation done in the latest build of Firefox, Chrome, Safari, and Microsoft Edge, and handles all known edge cases we have thrown at it. In development for the front end, a hard coded set of values were used to ensure that the interface was working as expected. Once the back end was operational, the application was run through Canvas and quizzes were developed. These quizzes were similar to actual quizzes that would be deployed by the client, with reasonable responses a student could give.

Future work

The application is fully functional, but has a large amount of options for extension. This includes sorting of the students based on various criteria, including their section, group, or name. Responses could be sorted based on grading status, and quizzes could be sorted in a similar manner.

Lessons learned

Never trust API documentation. No matter how complete it looks, bad documentation exists and is far more dangerous to a project than the lack of documentation. Any and all API calls should be tested before development starts, primarily so that any potential workarounds can be planned with the project, instead of the weekend after the project is due.

When working with ASP.Net's Razor pages, all variables and commands that need to be executed in the user's browser need to be javascript, and not C#. The Razor pages allow the use of C# to allow custom generation of a page based on a model, but are only run server-side for the initial page load, and are never seen in-browser.

Appendices

Appendix A: Initial Diagram of Functionality

This diagram was the initial diagram that helped guide the development of the app. This helped guide the expected flow and use of the app.

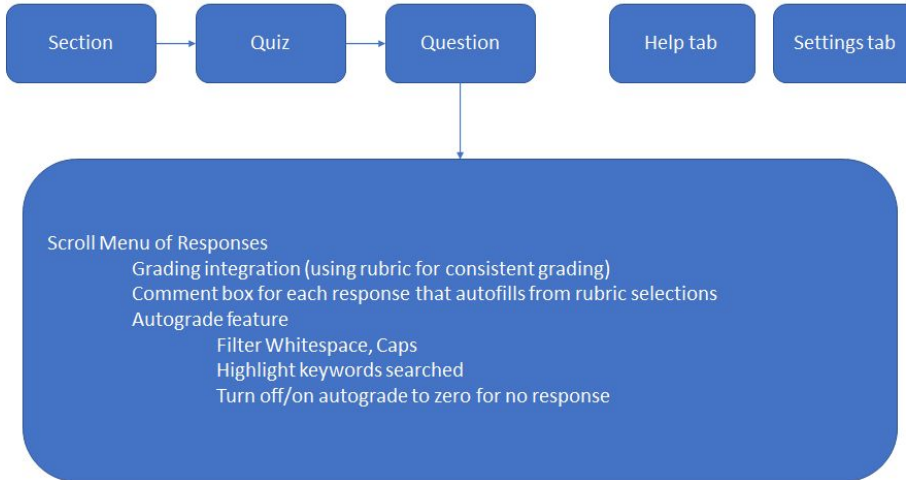


Figure 12: Initial App Flow

Appendix B: Initial Layouts

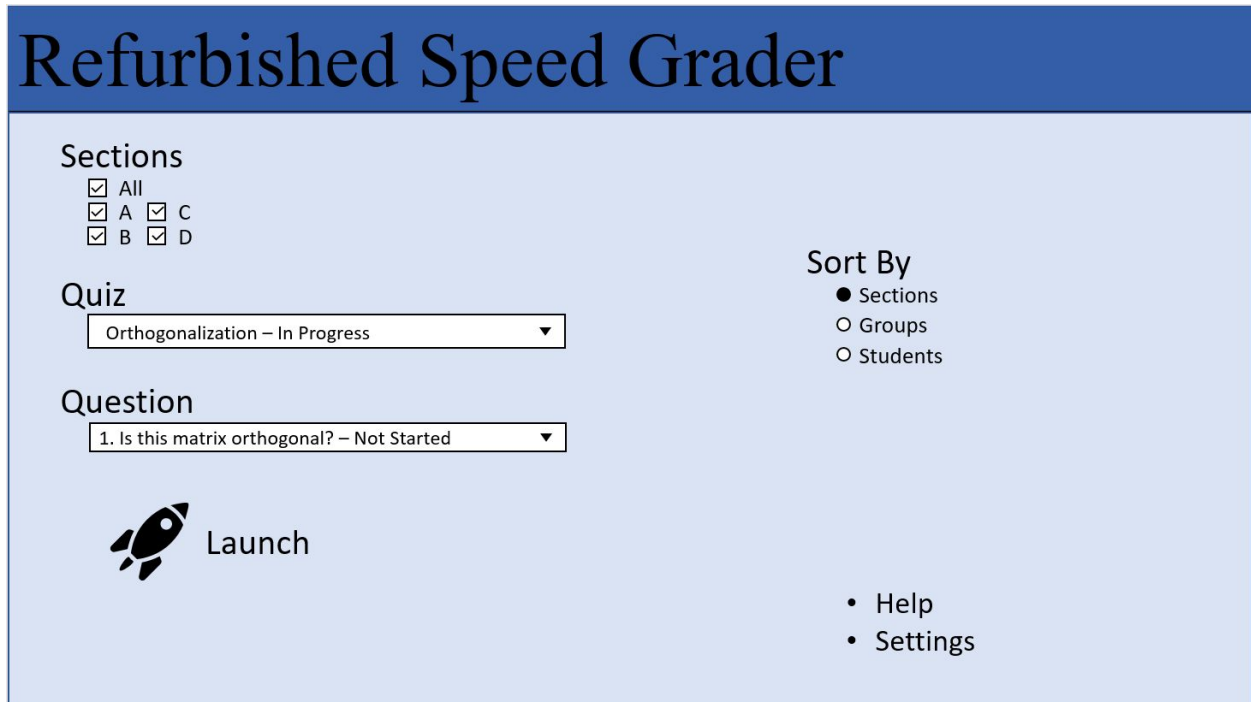


Figure 13: Initial Home Page Design

Orthogonalization ◀ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ▶

Is this matrix orthogonal? [Highlight Keyword](#) | [Jump to...](#) | [Autograde and Filter](#)




<p> Anonymous</p> <p>No, because the properties of an orthogonal matrix do not coincide with the matrix given</p> <p>25 / 25</p>	+5 Orthogonal Matrix	+0 Didn't use term Orthogonal Matrix
	+10 No	+0 Answer wasn't No
	+5 Reasonable explanation	+0 No reasonable explanation
<p> Anonymous</p> <p>Yes</p> <p>0 / 25</p>	<p>Revert Save Save & Next</p>	
<p> Anonymous</p> <p>No, because it looks funny</p> <p>10 / 25</p>	<p>+5 Orthogonal Matrix +10 No +5 Reasonable Explanation ----- Dude, nice Clear</p> <p>Grade: 25 / 25</p>	

Figure 14: Initial Grade page Layout