

# **Developing Software Infrastructure that Empowers Teachers to Create Custom Content on a Computer Aided Learning Platform**



**WPI**



*Gabriel Aponte  
Andrew Bonaventura  
James Kajon  
Ioannis Kyriazis*

*Advised by Neil Heffernan*

# MQP: Soft. Eng. Capstone

A Major Qualifying Project  
submitted to the Faculty of  
WORCESTER POLYTECHNIC INSTITUTE  
in partial fulfilment of the requirements for the  
degree of Bachelor of Science

By

Gabriel Aponte  
Andrew Bonaventura  
James Kajon  
Ioannis Kyriazis

Advisor: Neil Heffernan

Organizations: Worcester Polytechnic Institute and the ASSISTments Foundation

March 18<sup>th</sup>, 2021

*This report represents work of WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its web site without editorial or peer review. For more information about the projects program at WPI, see <http://www.wpi.edu/Academics/Projects>*

WORCESTER POLYTECHNIC INSTITUTE

# Abstract

Online learning software, such as ASSISTments, has become a major tool within the pockets of educators. Our project attempts to improve the Content Builder component that ASSISTments is planning to add into their latest system. We achieved this goal by implementing data persistence for user created problems and ensuring that the user interface was fluid and simple to use. Our team also implemented collaboration features by giving users the ability to import content created by other users. To improve the security of the Content Builder, we also identified various software techniques that can be used to prevent cyberattacks and protect user information. Ultimately, we provided the ASSISTments Team with a new iteration of the Content Builder that is more robust and intuitive to use.

# Acknowledgements

We would like to thank the entire ASSISTments Team for facilitating this project and welcoming us into their organization. Specifically, we would like to thank David Magid, Joan Wong, and Christopher John Donnelly for providing helpful clarifications on system processes and offering technical guidance to our team throughout the project.

Secondly, we would like to thank Cristina Heffernan for introducing us to the ASSISTments Platform and teaching us about the significance of The ASSISTments Foundation.

We would also like to acknowledge our project advisor, Neil Heffernan, for his continual support, ideas, and motivation to fully investigate the potential of our project.

Finally, we would like to give a special thank you to our technical lead, Ashish Gurung, who provided crucial feedback on the development of our project and always made time to answer our questions and support our work in any possible way.

# Executive Summary

The age of internet technologies has led to tremendous growth within the education sector due to the development of online learning tools. One such tool that has pioneered the age of digital learning is ASSISTments. This free-to-use K-12 mathematics focused system, first developed in 2003, provides educators with the ability to put their classroom online and track their students' progress on homework and assignments. The first version of this system, known as ASSISTments 1.0, included the Content Builder component that allowed teachers to fully customize and create their own homework and problem sets. It was in this way that ASSISTments was known for putting “teachers in the driver seat” of digital education.

After gaining a lot of support from both the press and the US Department of Education, the ASSISTments Foundation has received around 50 million dollars' worth of funding for educational research (ASSISTments, 2020). The ASSISTments Team is now using this funding to develop ASSISTments 2.0, a new version of the learning tool being built with modern-day internet technologies. The 2.0 system is currently available even though some features, including the new Content Builder (Builder 2.0), are still in development and have not been released. Builder 2.0 is being developed with more current software technologies such as the Google Web Toolkit (GWT) and a combination of Java and JavaScript programming.

Our team was tasked with continuing the development of a Minimum Viable Product for the ASSISTments 2.0 Content Builder component so that educators could create, save, and modify their own customizable problems. With the accomplishment of this goal, our team delivered a new iteration of Builder 2.0 that provides both a more intuitive user experience, and multiple features that enhance and simplify the content creation process. By leaving the ASSISTments Team with a more robust Content Builder, we aided ASSISTments in its mission to empower teachers through its educational software.

## Objectives and Implementation

To achieve our goal, the team established five key objectives that were completed throughout the project implementation;

- 1) To implement RESTful API calls that connect the front-end and back-end of Builder 2.0 to keep problems up-to-date.
- 2) To enhance the user experience by improving and condensing the Builder 2.0 user interface.
- 3) To provide users with the ability to import existing content into their projects.
- 4) To automate the development build process via scripting.
- 5) To explore potential security implementations that can protect user information and prevent cyberattacks.

## Implementation of Objective 1

The first objective focused on creating a link between the front-end and back-end of Builder 2.0. When our project began, Builder 2.0 was unable to save any modifications made to problems within the user interface (UI). Without this functionality, problem data would not be saved to the ASSISTments database and users would have to recreate content every time they visited the site. To implement data persistence for problems, our team developed three RESTful API endpoints within the back-end of Builder 2.0. These endpoints send data to the ASSISTments software development kit (SDK) which holds all of the functionality for updating the databases. On the front-end, a new singleton Java class called *MakeAsyncCall* was created. It contains all the functions that make requests to the endpoints within the back-end. To make sure that these endpoints were automatically saving user inputted data, our team implemented various UI event handlers that call the functions within *MakeAsyncCall*. These event handlers were added to all the problem UI elements, shown in [Figure 1](#), so that any changes made by the user would be synced to the database in real time.

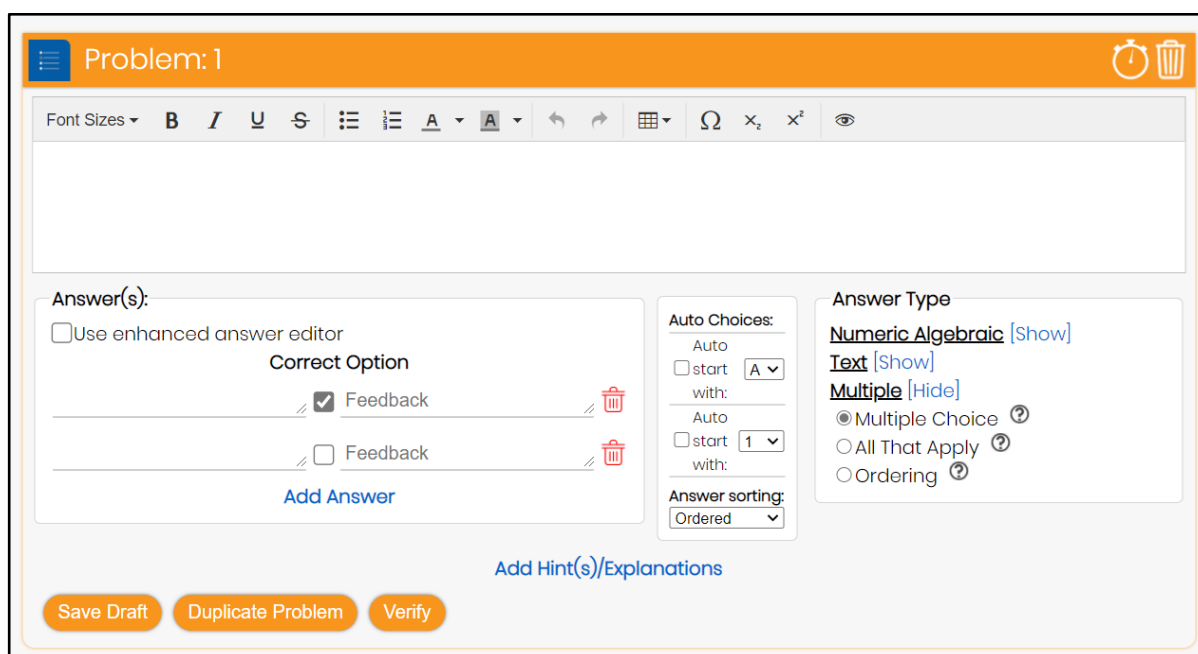
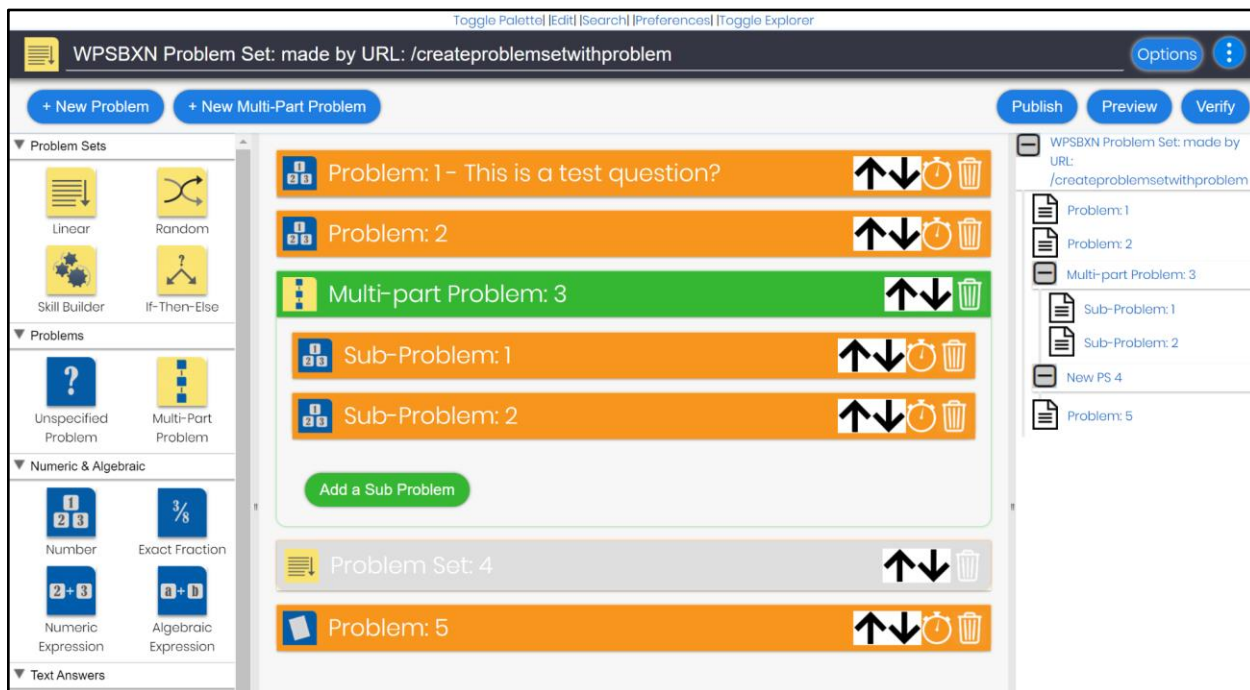
The screenshot displays the 'Problem: 1' editor in Builder 2.0. At the top is an orange header bar with the title 'Problem: 1' and icons for a clock and trash. Below the header is a rich text editor toolbar with options for font size, bold, italic, underline, strikethrough, bulleted list, numbered list, text color, background color, undo, redo, table, link, unlink, and a visual editor icon. The main content area is a large white box for the problem text. Below this are three panels: 1. 'Answer(s):' panel with a checkbox for 'Use enhanced answer editor', a 'Correct Option' section with a feedback checkbox, and an 'Add Answer' button. 2. 'Auto Choices:' panel with 'Auto' and 'start with' dropdowns, and an 'Answer sorting:' dropdown set to 'Ordered'. 3. 'Answer Type' panel with options for 'Numeric Algebraic', 'Text', and 'Multiple', each with a 'Show' or 'Hide' link, and radio buttons for 'Multiple Choice', 'All That Apply', and 'Ordering'. At the bottom are three orange buttons: 'Save Draft', 'Duplicate Problem', and 'Verify'. A link 'Add Hint(s)/Explanations' is also present.

Figure 1: A Problem in the Builder 2.0 UI

## Implementation of Objective 2

The second objective consisted of modifying the Builder 2.0 UI to remove unnecessary and redundant features, and to make the experience more user friendly. We started by analyzing the three separate UI layouts that were available to users. Our team found that the slight differences between these UI layouts could be condensed into a singular view. By removing the functionality to switch between UIs and ensuring that all Builder 2.0 features were accessible to the user, a more fluid UI was created. This new UI layout contains various toggle elements that

give the user complete control over what features are visible to them at a certain time. *Figure 2* provides a glimpse at the new UI when all of the functionalities are enabled.

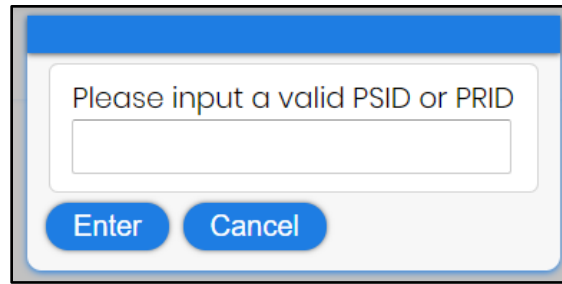


*Figure 2: The Builder 2.0 UI when all Features are Visible*

The UI was also updated to reduce confusion and frustration. Originally, Builder 2.0 allowed users to reorder the content within their projects by dragging and dropping problems and problem sets into new locations. This feature was both clunky and, at times, unresponsive. A simpler and more responsive solution was implemented through the addition of up and down arrow buttons (*Figure 2*). Users can interact with these buttons to manipulate the location of their content. In addition, Builder 2.0 contained a view that attempted to display user created content in the form of a “tree”. This ‘Tree View’ was redundant, did not provide any helpful functionality and contained many technical issues. As such, the ‘Tree View’ was removed to further improve the fluidity of the UI. Lastly, our team improved the consistency of the Builder 2.0 aesthetic by developing custom popup windows that mimic the color scheme and style of the rest of the website. The custom windows were then implemented into the Builder 2.0 UI to display alerts, confirmations, and user input boxes.

### *Implementation of Objective 3*

The third objective entailed the development of an import content feature that allows users to utilize problems and problem sets that already exist within the ASSISTments database. This feature was implemented first with the addition of the ‘+ Import Content’ button. When clicked, it will display a custom popup window that asks users to enter the unique ID associated with the problem or problem set they would like to import (*Figure 3*).



*Figure 3: Import Content Popup Window*

Our team then implemented two functions into the front-end that use these IDs to pull the correct data from the database. This was done by having the two functions send a request to the RESTful API endpoints responsible for accessing the SDK functions that retrieve problem and problem set information. The SDK finds the content linked to the inputted ID within the database and then sends the information back to the front-end. Lastly, our team implemented functionality that displays the imported problem and problem set data into the Builder 2.0 UI. With this functionality, users are able to reuse content they have already created and build off of content created by other users.

#### *Implementation of Objective 4*

The fourth objective focused on providing more efficiency to the Builder 2.0 development process. Developing and testing Builder 2.0 functionality requires rebuilding a development version of the website anytime a change is made to the code-base. The build process follows the order of; updating all project dependencies, separately building each of the three projects within Builder 2.0, loading resources onto a test server and finally, starting the test server and accessing the Builder 2.0 website. This process could be quite tedious as it required the developer to manually trigger each step of the build process. This meant that our team members and other developers working on Builder 2.0 had to sit and wait at their workstations and execute each step in the correct order. If any errors were made, the process would have to be restarted. To solve these inefficiencies, our team created a batch script that, upon being run, will automatically update all the project dependencies, and build all three of the projects within Builder 2.0. After running the script, developers will only need to refresh the server resources and restart the Tomcat server, which takes the least amount of time compared to the other steps. This build script provides developers with a more effective and less mundane process to follow when building a development version of the Builder 2.0 website.

#### *Implementation of Objective 5*

The final objective consisted of analyzing Builder 2.0 for potential security vulnerabilities. To fulfill both this objective and the cybersecurity component of his degree, Ioannis conducted a security audit on Builder 2.0. He found that Builder 2.0 is currently susceptible to both SQL injection and JavaScript injection attacks. These types of attacks can occur when a user inputs malicious SQL statements or JavaScript code into any of UI elements



that trigger requests to the back-end server via RESTful API endpoints. With injected SQL statements, attackers can potentially delete information stored within the ASSISTments database since these endpoints link to SDK functionality that manipulates the database. With JavaScript injection, attackers can potentially steal user login data or use their computers to mine cryptocurrency. The audit also identifies the precautions that SDK currently takes against these types of attacks. In terms of security enhancements, the audit details additional security implementations that can be put in place to further mitigate these types of attacks.

## **Discussion and Conclusion**

The new iteration of Builder 2.0 that our team developed provides a more robust and intuitive system compared to its predecessor. Builder 2.0 now facilitates automatic data persistence and provides its users with the means to collaborate with each other. All of the new features implemented by our team are augmented by the new UI layout that delivers a more fluid and simpler to understand experience to the users. In addition, our team provided the ASSISTments Team with a build script that will improve the efficiency of all future work. Lastly, the audit conducted on the website provides future developers with a clear path on how to improve the security of Builder 2.0 so that its users are protected from potential cyberattacks. With our implementations and recommendations, ASSISTments is much closer to the completion of a Minimum Viable Product for the new Content Builder. Once the system is released to the public, Builder 2.0 will empower teachers to collaborate and create custom content that will enhance the learning of all their students.

# Authorship

The following report was written in a collaborative manner by all four group members: Gabriel Aponte (GA), Andrew Bonaventura (AB), James Kajon (JK), and Ioannis Kyriazis (IK)

	Author(s)	Editor(s)
<b>Introduction</b>	GA	All
<b>Background</b>		
ASSISTments Builder 1.0	JK	All
ASSISTments Builder 2.0	All	All
The Accordion View	IK	All
Multi-Part Problems	IK	All
If-Then-Else Problems	AB	All
Linear and Random Problem Sets	JK	All
Auto-Saving User Data	GA	All
Builder 2.0 Project Setup	AB, IK	All
Common	IK	All
Service	IK	All
Client	IK	All
SDK 3.0	IK	All
Apache Maven	AB	All
Apache Tomcat	AB	All
GWT and GWT-Jackson	AB	All
Java to JavaScript Translation	AB	All
Serialization	AB	All
Binding Tag Annotations	AB	All
UI Creation with GWT	GA	All
Using XML to Design a UI	GA	All

Manipulating UI Elements in Code	GA	All
Utilizing TinyMCE with GWT	GA	All
<b>Implementation</b>		
Saving Problems: Front-End	GA, AB, JK	All
Linking Problems in the UI to the Database	GA	All
Adding Saving Service API Calls to the UI	GA	All
MakeAsyncCall Class	AB	All
Testing New Functionality	JK	All
Saving Problems: Back-End	AB	All
API Endpoint: /problemservice/save	AB	All
API Endpoint: /createproblemsetwithproblem	AB	All
API Endpoint: /tutoringservice/save	AB	All
Saving Problems: SDK Errors	GA, AB, JK	All
V1 Certificate Error	JK	All
DOA Null Parameter Error	GA	All
Owner_ID Null Error	GA	All
Answer SQL Calls Null	GA	All
Error Location Validation Process	AB	All
UI Decisions and Changes	AB, JK, IK	All
Condensing the UI Layout	GA, JK	All
Removal of Tree View	AB, JK	All
Removal of Drag and Drop	GA, AB	All
Custom Popup Windows	GA	All
Importing Content	GA, AB	All
Importing Problem Sets	AB	All
Importing Individual Problems	GA	All
Automated Build Process	IK	All

Security Audit of Builder 2.0	IK	All
SQL Injection	IK	All
JavaScript Injection	IK	All
<b>Discussion and Analysis of Results</b>		
Project Overview	AB	All
Project Impact	GA	All
Future Work	GA, AB	All
Fixing the Saving Answers SDK Error	GA	All
Adding Data Persistence for Problem Sets	GA	All
Fixing the SDK Bugs Affecting Importing Content	AB	All
Cleaning Up the User Interface	GA	All
Adding Restrictions Based on User Authority	GA	All
Preventing Malicious Injections	GA	All
<b>Conclusion</b>		

# Table of Contents

Abstract .....	i
Acknowledgements .....	ii
Executive Summary .....	iii
Authorship .....	viii
List of Figures .....	xiii
1. Introduction.....	1
2. Background.....	3
2.1 ASSISTments Builder 1.0 .....	3
2.2 ASSISTments Builder 2.0.....	5
2.2.1 The Accordion View.....	6
2.2.2 Multi-Part Problems.....	7
2.2.3 If-Then-Else Problems .....	9
2.2.4 Linear and Random Problem Sets .....	10
2.2.5 Auto-Saving User Data .....	10
2.3 Builder 2.0 Project Setup .....	11
2.3.1 Common.....	11
2.3.2 Service.....	12
2.3.3 Client.....	13
2.3.4 SDK 3.0.....	14
2.3.5 Apache Maven.....	15
2.3.6 Apache Tomcat.....	15
2.4 GWT and GWT-Jackson .....	16
2.4.1 Java to JavaScript Translation.....	17
2.4.2 Serialization.....	17
2.4.3 Binding Tag Annotations .....	18
2.5 UI Creation with GWT .....	19
2.5.1 Using XML to Design a UI.....	19
2.5.2 Manipulating UI Elements in Code .....	22
2.5.3 Utilizing Tiny MCE with GWT .....	25
3. Implementation .....	28
3.1 Saving Problems: Front-End.....	28
3.1.1 Linking Problems in the UI to the Database .....	29
3.1.2 Adding Saving Service API Calls to the UI.....	30
3.1.3 MakeAsyncCall Class .....	32
3.1.4 Testing New Functionality .....	36
3.2 Saving Problems: Back-End .....	37
3.2.1 API Endpoint: /problemservice/save .....	38
3.2.2 API Endpoint: /createproblemsetwithproblem.....	39
3.2.3 API Endpoint: /tutoringservice/save .....	40
3.3 Saving Problems: SDK Errors .....	41

3.3.1 V1 Certificate Error .....	41
3.3.2 DOA Null Parameter Error .....	42
3.3.3 Owner_ID Null Error .....	42
3.3.4 Answer SQL Calls Null .....	43
3.3.5 Error Location Validation Process .....	43
3.4 UI Decisions and Changes .....	44
3.4.1 Condensing the UI Layout .....	44
3.4.2 Removing the Tree View .....	52
3.4.3 Removing Drag-and-Drop .....	54
3.4.4 Custom Popup Windows.....	57
3.5 Importing Content .....	61
3.5.1 Importing Problem Sets .....	62
3.5.2 Importing Individual Problems .....	64
3.5 Automated Build Process .....	66
3.6 Security Audit of Builder 2.0.....	67
3.6.1 SQL Injection .....	67
3.6.2 JavaScript Injection .....	68
4. Discussion and Analysis of Results .....	70
4.1 Project Overview.....	70
4.2 Project Impact .....	72
4.3 Future Work .....	73
4.3.1 Fixing the Saving Answers SDK Error.....	73
4.3.2 Adding Data Persistence for Problem Sets .....	73
4.3.3 Fixing the SDK Bugs Affecting Importing Content.....	73
4.3.4 Cleaning Up the User Interface .....	74
4.3.5 Adding Restrictions Based on User Authority .....	74
4.3.6 Preventing Malicious Injections .....	75
5. Conclusion .....	76
References .....	77
Appendix A: List of UI Elements with MakeAsyncCall().....	78

# List of Figures

Figure 1: Builder 1.0's Problem Set Creation Page.....	4
Figure 2: Builder 1.0's Problem Set Edit View.....	4
Figure 3: Saving Problems and Answers in Builder 1.0.....	5
Figure 4: The Accordion View with Multiple Problems in Builder 2.0.....	6
Figure 5: Collapsed Problems in Builder 2.0.....	7
Figure 6: An Expanded Problem in Builder 2.0.....	7
Figure 7: A Multi-Part Problem with a Single Sub-Problem in Builder 2.0.....	8
Figure 8: A Multi-Part Problem with Two Sub-Problems in Builder 2.0.....	8
Figure 9: Example of an If-Then-Else Problem Set Inside a Problem Set.....	9
Figure 10: The Structure of an If-Then-Else Problem Set.....	9
Figure 11: Setting the Problem Set Ordering During Creation.....	10
Figure 12: Serialization and Deserialization Process (GeeksForGeeks, 2019).....	17
Figure 13: Rules for a JavaBean (baeldung, 2020).....	18
Figure 14: Example of Binding Annotations in Java.....	19
Figure 15: AddAnswerView XML File.....	20
Figure 16: Builder 2.0's UI for Adding Answers.....	22
Figure 17: Builder 2.0's UI for Adding Answers.....	22
Figure 18: AddAnswerView.java Constructor.....	23
Figure 19: Declaring GWT UI elements in Java Code.....	23
Figure 20: Example of a GWT Built-In Event Handler Function (GWT, n.d.).....	24
Figure 21: Example of a Custom Event Handler Function (GWT, n.d.).....	25
Figure 22: TinyMCEView XML File.....	26
Figure 23: Snippet of AddAnswerMCEView XML File.....	26
Figure 24: Builder 2.0's TinyMCE UI for Adding Answers.....	27
Figure 25: GWT Jackson PROBLEM_ID Serialization Error.....	29
Figure 26: The makeAsyncCall() for AddAnswerView.java.....	30
Figure 27: AddAnswerMCEView.java BlurEvent Handler.....	31
Figure 28: AddAnswerView.java ValueChangeEvent Handler.....	31
Figure 29: AddAnswerView.java Custom Event Handler.....	32
Figure 30: Singleton Setup for MakeAsyncCall.java.....	32
Figure 31: The onBlur() Function in ProblemEditorPanelView.java.....	33
Figure 32: Extracting a needsToBeSaved Object From a problemBuilder Object.....	34

Figure 33: The Three HashMaps Used for Filtering in MakeAsyncCall.java .....	34
Figure 34: Snippet of the For-Loop Structure within filterDuplicates() .....	35
Figure 35: The ANSWER Case Comparison within filterDuplicates() .....	35
Figure 36: JavaScript Promise Structure with Requestor Class and Container Object .....	36
Figure 37: Extracting Content from a needsToBeSaved Object .....	38
Figure 38: Example of Dependency Injection for the ProblemBuildManagerImpl .....	38
Figure 39: Using Factory Classes to Generate a Problem Set and Problem .....	39
Figure 40: Setting the Question and Name of the Problem Builder .....	39
Figure 41: Wrapping problemSet and problemSetID into an SPSJsonContainer .....	40
Figure 42: V1 Certificate Error.....	41
Figure 43: DOA Null Parameter Error.....	42
Figure 44: Owner_ID Null Error .....	42
Figure 45: Manually Setting the Owner_ID Field .....	43
Figure 46: Answer SQL Calls Null Error.....	43
Figure 47: Preferences Popup Window with the Three Original UI Options .....	45
Figure 48: Builder 2.0's Minimalist UI.....	45
Figure 49: Builder 2.0's Default UI.....	46
Figure 50: Builder 2.0's Advanced UI.....	47
Figure 51: Making the Palette and Explorer Visible to the User.....	47
Figure 52: Conditional Statements for using userLevel to Switch UI Layouts.....	48
Figure 53: Placeholder Toggle Buttons for the Palette and Explorer .....	49
Figure 54: The togglePalette() and toggleExplorer() Functions.....	49
Figure 55: The Condensed UI with Both the Palette and Explorer Open .....	50
Figure 56: The Condensed UI with the Palette Closed and the Explorer Open .....	50
Figure 57: The Condensed UI with the Palette Open and the Explorer Closed .....	51
Figure 58: The Condensed UI with Both the Palette and Explorer Closed.....	51
Figure 59: Advanced Settings Options in the Preferences Popup Window .....	52
Figure 60: Advanced Builder Options Enabled (Left) and Disabled (Right).....	52
Figure 61: Builder 2.0's Tree View .....	53
Figure 62: Conditional Statement for Drawing Either the Tree or Accordion View.....	54
Figure 63: The Various Headers for Problems and Problem Sets .....	55
Figure 64: The onDragStart() Event Handler that Disables Dragging AccordionContent .....	56
Figure 65: New Up Arrow and Down Arrow Buttons for Content Reordering .....	56
Figure 66: Non-customizable Window Alert for Problem Sets with No Title.....	57
Figure 67: Non-customizable Window Confirmation for Deleting Problem Sets.....	57



Figure 68: PopupWindowView XML UI Layout File.....	58
Figure 69: Setting up the Custom Popup Window DialogBox .....	59
Figure 70: New Alert Popup Window for Problem Sets with No Title .....	60
Figure 71: A Runnable Being Passed into a Custom Confirmation Popup Window .....	60
Figure 72: A Runnable Being Passed into a Custom Confirmation Popup Window .....	61
Figure 73: The Toolbar with the New Import Content Button.....	61
Figure 74: Import Content Popup Window .....	62
Figure 75: Example of an Imported Problem Set .....	62
Figure 76: Using a Resulting ProblemSet to Obtain a ProblemSetBuilder.....	63
Figure 77: Creating an Imported Problem Set with the Current Problem Set as the Parent .....	63
Figure 78: The For-Loop that Adds Problems and Problem Sets to the Imported Problem Set ...	64
Figure 79: Parsing for Content Keys instead of PRIDs .....	64
Figure 80: Snippet of the onDone() Function Within importPostRequestForSPR() .....	65
Figure 81: “Proof of Concept” For an Imported Problem.....	66
Figure 82: Batch File that Automatically Builds the Builder 2.0 Project .....	67

# 1. Introduction

The age of internet technologies has led to tremendous growth in all aspects of life, especially education. Learning, whether it be for elementary school students or college students, has been integrated with a vast number of online tools. One such tool that pioneered the transition from using physical paper in the classroom to digital screens is ASSISTments. This K-12 mathematics focused system was developed in 2003 by Neil and Cristina Heffernan. As math teachers themselves, they saw first-hand how classrooms would benefit tremendously from more “effective student feedback and data-driven insights” (ASSISTments, 2020). The original ASSISTments online learning tool and app, commonly known as ASSISTments 1.0, provides educators with the ability to put their classroom online and track their students’ progress on homework and assignments. The 1.0 system also has the Content Builder, which is a component that allows teachers to fully customize and create their own homework and problem sets. It was in this way that ASSISTments was known for putting “teachers in the driver seat” in terms of digital education since they do not have to rely solely on pre-made coursework that the platform offers.

After much praise from the press and the US Department of Education, ASSISTments was given the chance to scale up, and expand its capabilities and reach. With the establishment of the non-profit organization known as the ASSISTments Foundation in 2019, the free-to-use education platform has received around 50 million dollars’ worth of funding for educational research (ASSISTments, 2020). The ASSISTments Team is now working on the development of ASSISTments 2.0, a new version of the tool being built from the ground up on modern-day internet technologies. While parts of the 2.0 system are available now, some features such as the Content Builder are still in development. With the help of a recent 8 million dollar Education Innovation and Research (EIR) grant, the ASSISTments team can enhance this essential tool so that teachers can more efficiently craft their own educational materials.

Before the start of our project, the new version of the builder component (Builder 2.0), was still in early development and had not been deployed to the public. The Content Builder is currently being rebuilt to use more current software technologies such as the Google Web Toolkit (GWT) and a combination of Java and JavaScript programming. To create Builder 2.0 in such a way, the project has to rely on three main components: a front-end user interface, a back-end service system and its own custom software development kit (SDK).

When our team began working on Builder 2.0, the user interface (UI) was nearly completed. The formatting, color scheme, and interactive elements were all setup and usable. The UI only needed a few additions to its functionality, some bug fixes and a more condensed layout. The back-end side of the project held a couple of initial RESTful API calls that would be used to save problems created by users to the ASSISTments server-based databases. However, these calls had not been fully completed nor implemented into the front-end of the system, which

meant that users were unable to save their problems. As for the SDK, this component of the system was still being developed. Since Builder 2.0 relies on the SDK for connecting to the databases, development on a basic functional version of Builder 2.0 was halted. Upon the completion of the SDK, our team's main task was to fully connect the back-end services to the front-end user interface so that teachers would be able to save their custom coursework. Additionally, our team was tasked to improve upon the user experience and explore potential cybersecurity measures for the Builder 2.0 site.

The goal of this project was to further the development of a Minimum Viable Product for the ASSISTments 2.0 Content Builder component so that educators could create, save, and modify their own customizable problems. We accomplished this goal by completing the following objectives;

- 1) To implement RESTful API calls that connect the front-end and back-end of Builder 2.0 to keep problems up-to-date.
- 2) To enhance the user experience by improving and condensing the Builder 2.0 user interface.
- 3) To provide users with the ability to import existing content into their projects.
- 4) To automate the development build process via scripting.
- 5) To explore potential security implementations that can protect user information and prevent cyberattacks.

By helping to integrate a usable Content Builder into the ASSISTments 2.0 system, we aided the ASSISTments Team in their goal to keep teachers in control of the classroom.

## 2. Background

To understand the significance of this project's software development, this chapter covers four topics that make up the foundations of ASSISTments 2.0's new builder component. The first topic analyzes the functionality of the original Content Builder included in ASSISTments 1.0. The second section details the functionality being added to the new builder component for ASSISTments 2.0. The third topic provides an overview of how the underlying code of Builder 2.0 is organized and managed. The next section discusses the software and programming methodologies that are used to develop Builder 2.0. The final section describes how the UI for Builder 2.0 is created and organized. In addition, this chapter represents the research and investigation of Builder 2.0 that our team conducted during the first few weeks of the project. The background information detailed below provided our team with a proper understanding of the Builder 2.0 project, which informed the implementations we carried out during development.

### 2.1 ASSISTments Builder 1.0

The builder component in ASSISTments 1.0 (Builder 1.0) was added so teachers could make sets of problems to assign to their students. As seen in [Figure 1](#), an ASSISTments 1.0 problem set can be made up of multiple problems. These problems can be of any ten different answer types.

Four of these answer types have to do with interpreting mathematical answers. *Numeric* type answers are for problems with number answers. Its main feature is that it can be answered in multiple different forms. For example, 10, 10.0, and  $20/2$  would all be accepted if the answer were 10. *Exact Fraction* type answers are for answers which must match the answer exactly and without simplification. *Numeric Expression* type answers can be answered by a mathematical expression using various operators on numbers. A user inputted expression is compared to the correct answer a teacher provides to find equivalence. *Algebraic Expression* type answers are similar to *Numeric Expressions* answers, but they can have variables within the expression.

There are also three answer types that require written responses. *Exact Match (case sensitive)* needs to be answered by text that precisely matches the correct answer inputted by a teacher. *Exact Match (ignore case)* is the same except it will not distinguish answers by case. *Ungraded Open Response* does not have a correct answer associated with it. This type of answer requires a teacher to manually read and assign a grade to a student's inputted answer.

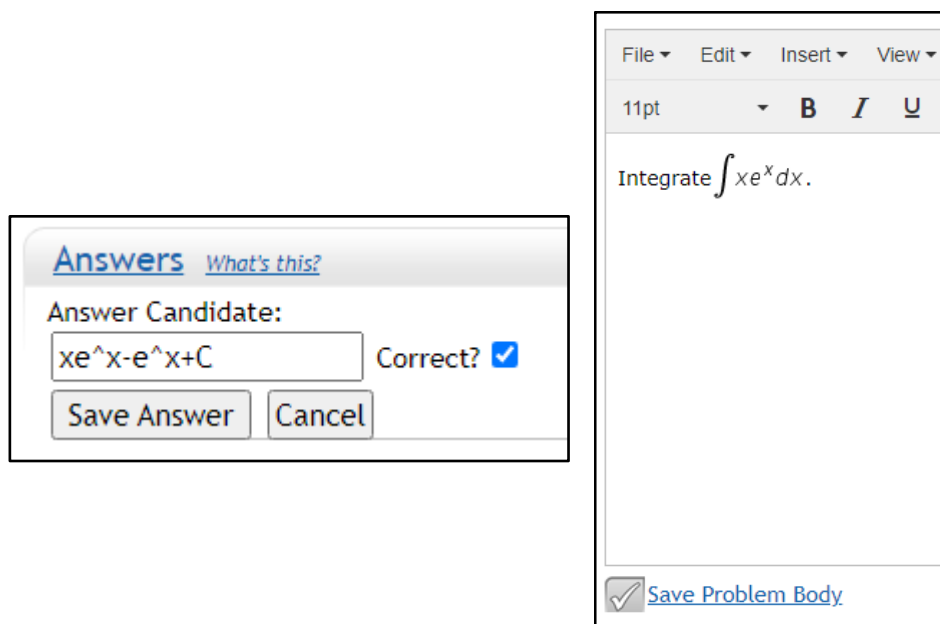
The final three types require student interaction with multiple predefined answers provided by a teacher. *Multiple Choice* is selecting the one correct answer from many possible answers. *Check All That Apply* is the same as multiple choice but there can be more than one correct answer. The last type, *Ordering*, involves ranking multiple answer choices in the correct order.

Figure 1: Builder 1.0's Problem Set Creation Page

The creation of a new problem set in Builder 1.0 begins with five empty problems that default to the *Numeric* type. There are text boxes to fill out the question, hint, and answer (Figure 1). After filling in some questions, a user can click a button to create the problem set. To add more problems, a user can press the 'Write New Problems Using the Builder' button, located in the edit view of the problem set (Figure 2). Doing so will add an additional five problems to be filled out.

Figure 2: Builder 1.0's Problem Set Edit View

Lastly, Builder 1.0 saves user data onto the ASSISTments servers. This functionality allows users to revisit their problem sets and modify them at their convenience. Builder 1.0 provides this feature with various save buttons. Once a user feels like their work is finished, or they would like to save a problem set that is a work-in-progress, they simply press these save buttons and the data gets updated on the servers. [Figure 3](#) shows an example of the saving functionality by detailing the buttons that control saving individual problems and answers.



*Figure 3: Saving Problems and Answers in Builder 1.0*

## 2.2 ASSISTments Builder 2.0

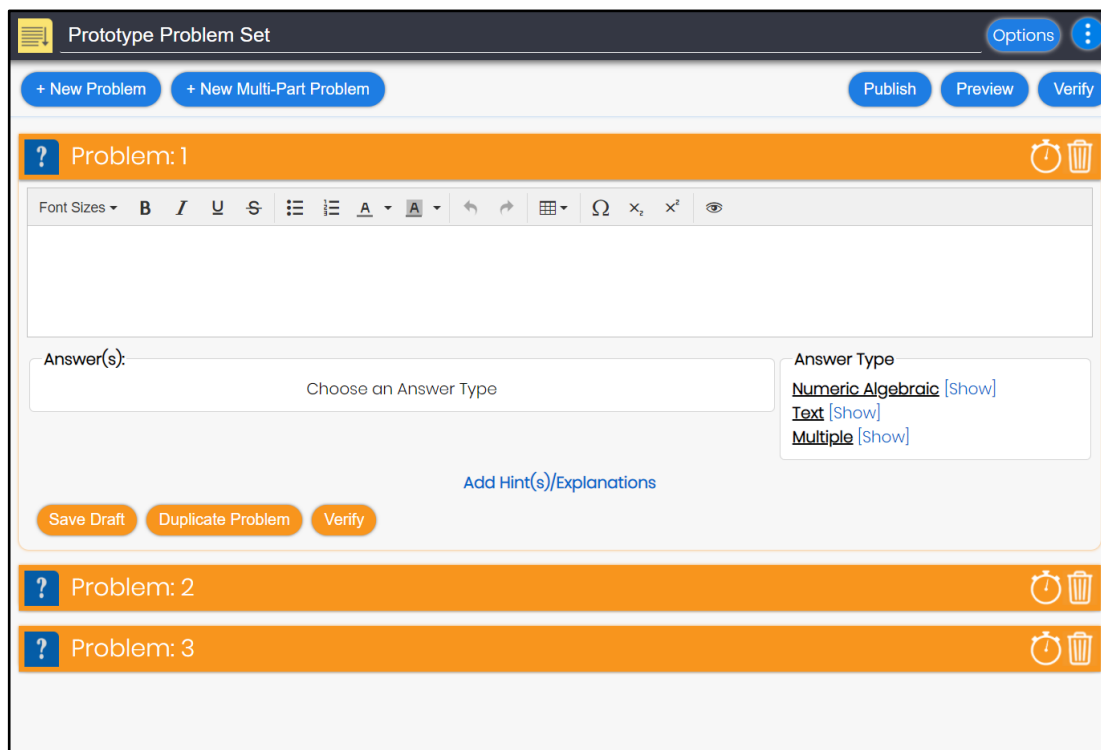
While Builder 1.0 was revolutionary for its time, it was lacking some intuitive features. These shortcomings are being addressed with the development of Builder 1.0's successor, Builder 2.0. The new content builder includes a fresh look for the UI while also being designed to be more intuitive and user friendly. With the inclusion of an 'Accordion View', creating and managing problems within the UI has become more robust. In addition, Builder 2.0 has added the 'Multi-Part' problem type and the 'If-Then-Else' problem set type which offer more customization to the user in terms of content creation. Problem sets can also be set up in 'Linear' and 'Random' fashions which allows teachers to utilize Builder 2.0 as both common classwork and individual assessments. Lastly, Builder 2.0 features auto-saving functionality which ensures that user data will be saved in real time and eliminates the overhead of the original manual saving options. While Builder 2.0 had not been publicly released by the date of publishing this paper, this section will go into detail about the new features and functionality that are being worked on during Builder 2.0's development. As Builder 2.0 is currently a work-in-progress, the features described below may look and function differently from the public release.

## 2.2.1 The Accordion View

The UI for Builder 1.0 limited users to only manipulating five problems at a time. The new UI for Builder 2.0 rectifies this limitation with the introduction of the ‘Accordion View’. This layout provides two new key features to make the user experience more intuitive:

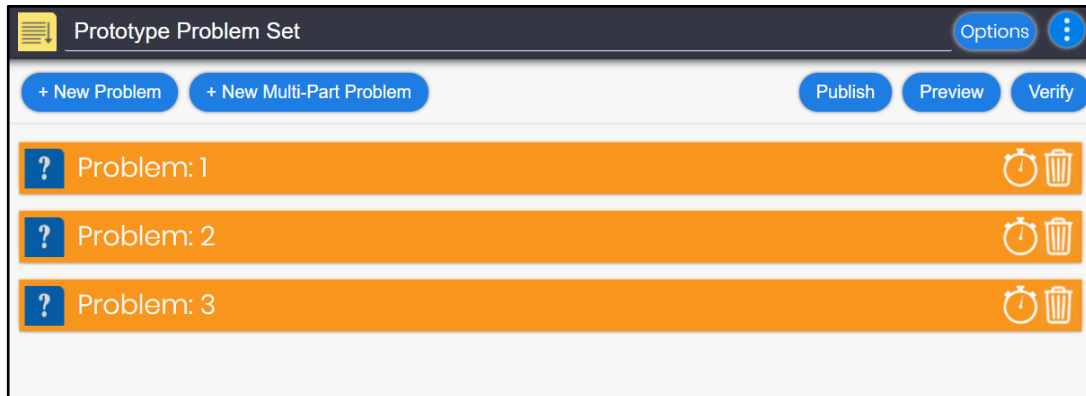
- 1) Viewing and editing all problems at once in a scrollable fashion, and
- 2) Collapsing problems for easier navigation.

Apart from providing a more fluid user experience, having all the problems in a set viewable at the same time allows teachers to easily create content that builds off of each other. For example, teachers can now copy similar content from one problem to another problem rather than having to manually search for content they already created. *Figure 4* shows the setup of the ‘Accordion View’ on a problem set with multiple problems.



*Figure 4: The Accordion View with Multiple Problems in Builder 2.0*

Since the ‘Accordion View’ is a scrollable page, the UI has the potential to get cluttered with the addition of many problems. To ensure that the UI is still easily navigable, the ‘Accordion View’ also allows users to collapse problems when they are not actively working on them by clicking a problem’s orange title bar. When collapsed, all the input fields for a specific problem are hidden and only the problem title bar will display in the UI (*Figure 5*).



*Figure 5: Collapsed Problems in Builder 2.0*

To expand the problem back to its original layout, the user clicks on the problem title bar again. Once expanded, a user has access to all of the input fields for modifying a Builder 2.0 problem. As seen in [Figure 6](#), an expanded problem offers the ability to add a question body, input answers, specify the type of ordering, add hints or explanations, and change the answer type.

*Figure 6: An Expanded Problem in Builder 2.0*

### 2.2.2 Multi-Part Problems

Builder 1.0 had no way of representing problems with multiple parts to it. Teachers need this feature because sometimes a problem is more complex than just one question. Each part of these problems requires different question text and accept different answers. It is not intuitive to separate each part of these problems into its own problem, so Builder 2.0 implements the 'Multi-Part' problem type. This feature allows teachers to easily create a problem with multiple parts



into new and existing problem sets. An example of a blank multi-part problem is detailed in *Figure 7*.

The screenshot shows the 'Multi-part Problem: 3' interface. At the top is a green header bar with a list icon and a trash icon. Below it is an orange bar for 'Sub-Problem: 1' with a question mark icon, a timer icon, and a trash icon. The main area contains a rich text editor with a toolbar (Font Sizes, Bold, Italic, Underline, Strikethrough, Bulleted List, Numbered List, Text Color, Background Color, Undo, Redo, Table, Link, Unlink, Math, Omega, Subscript, Superscript, Hide) and a large text input field. Below the text field is an 'Answer(s):' section with a 'Choose an Answer Type' dropdown. To the right is an 'Answer Type' panel with links for 'Numeric Algebraic [Show]', 'Text [Show]', and 'Multiple [Show]'. Below these are buttons for 'Save Draft', 'Duplicate Problem', and 'Verify'. A blue link 'Add Hint(s)/Explanations' is also present. At the bottom is a green button labeled 'Add a Sub Problem'.

*Figure 7: A Multi-Part Problem with a Single Sub-Problem in Builder 2.0*

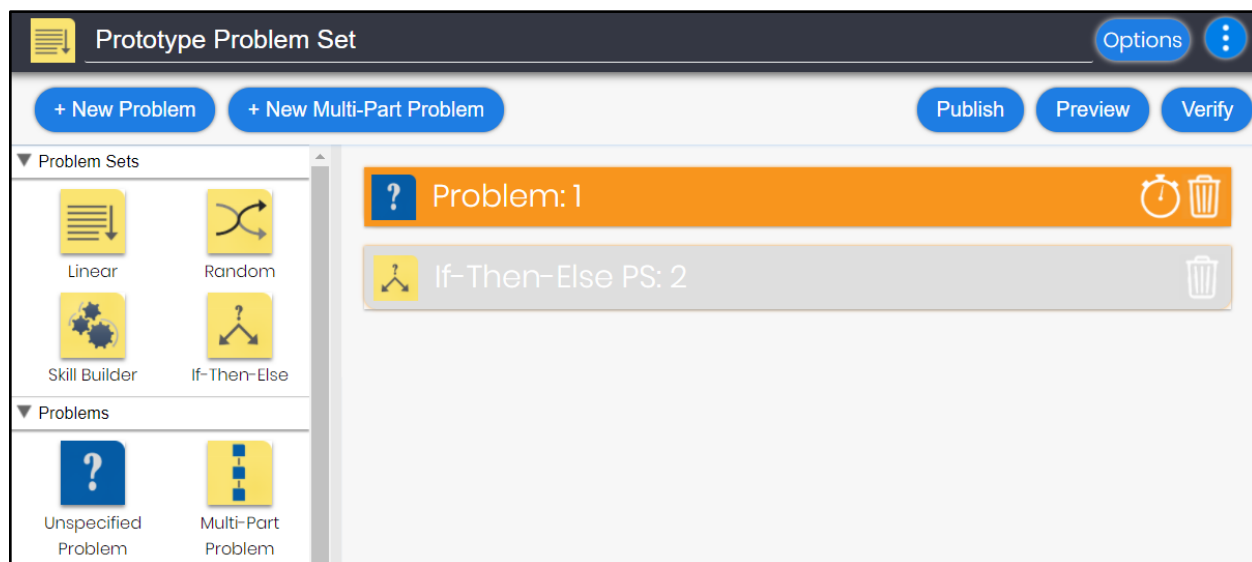
Once a new multi-part problem is created, the user can add more sub-problems by clicking on the green ‘Add a Sub-Problem’ button. *Figure 8* shows a second sub-problem added to the multi-part problem.

This screenshot is similar to Figure 7 but shows two sub-problems. The top section is identical to 'Sub-Problem: 1'. Below it is a new orange bar for 'Sub-Problem: 2' with a question mark icon, a timer icon, and a trash icon. The 'Add a Sub Problem' button is no longer visible at the bottom.

*Figure 8: A Multi-Part Problem with Two Sub-Problems in Builder 2.0*

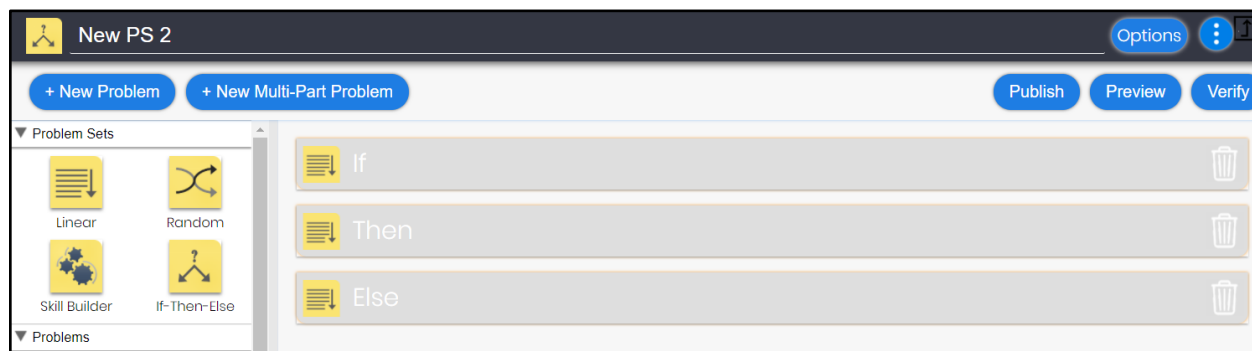
### 2.2.3 If-Then-Else Problems

One of the upgrades for Builder 2.0 was the inclusion of ‘If-Then-Else’ problem sets. These problem sets were initially included as part of the Advanced UI view since it is more complex than other problem set types. These ‘If-Then-Else’ problem sets give teachers more control in customizing the student’s experience when going through a problem set. An ‘If-Then-Else’ problem set displays as a separate sub-problem set added to the initial problem set as seen in *Figure 9*.



*Figure 9: Example of an If-Then-Else Problem Set Inside a Problem Set*

The purpose of the ‘If-Then-Else’ problem set is to let teachers create a different control flow based on the correctness of student's answers. It is set up such that teachers can add a problem into the ‘If’ block that will act as control flow for a problem added into the ‘Then’ and ‘Else’ blocks as seen in *Figure 10*.



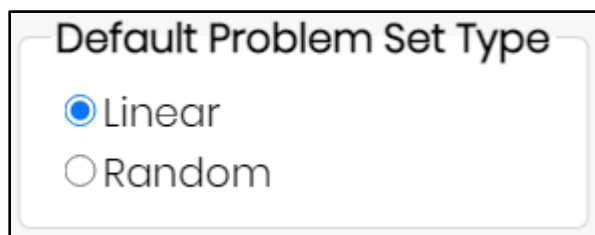
*Figure 10: The Structure of an If-Then-Else Problem Set*

If the student is able to answer the question in the ‘If’ block correctly, they will see the problem in the ‘Then’ block and not the problem in the ‘Else’ block. If they get the question wrong, they will see the problem in the ‘Else’ block and skip over the ‘Then’ block. This allows teachers to

both reinforce problems that are not answered correctly and provide more challenging questions when a student is doing well with the standard subject matter.

#### 2.2.4 Linear and Random Problem Sets

Another upgrade included in Builder 2.0 is the addition of the ‘Random’ problem set type. In Builder 1.0, when completing an assignment, the problems were in the same order every time. This type of ordering is provided by the ‘Linear’ problem set type. Now, there is the option to make problems randomly ordered. The ordering can be set on problem set creation (*Figure 11*) or when assigning problem sets to students.



*Figure 11: Setting the Problem Set Ordering During Creation*

This feature is key for when students are doing assignments in class. A random problem set can vastly reduce cheating since students are not answering the same question at the same time. When used in conjunction with turning off feedback, a random problem set assignment can be used as a test or exam.

#### 2.2.5 Auto-Saving User Data

One of the more prominent shortcomings of Builder 1.0 dealt with users saving the content they built. Builder 1.0 relied on manual saving. Once a user was satisfied with their progress, they had to manually click a save button that would then update all of their work within the ASSISTments databases. There were three main factors that made this process unreliable:

- 1) Internet outages,
- 2) Computer crashes, and
- 3) Users forgetting to save their progress at the end of a session.

All three of these potential external issues would cause the user to lose all of their progress since the last successful save. To mitigate the data loss presented by these factors, Builder 2.0 no longer relies on the user to save their work manually. Instead, every change a user makes is automatically synced with the ASSISTments database. For example, adding a new problem into the problem set or changing the title of the problem set will be saved automatically and immediately after the user makes the addition or edit. Builder 2.0 provides this functionality by linking each UI element that contains user inputted data to event-handler functions and RESTful API calls to the ASSISTments servers. The process for how the auto-saving functionality for problems was implemented is detailed within the Implementation section of this paper.

## 2.3 Builder 2.0 Project Setup

The outdated ASSISTments 1.0 system is now being rebuilt using modern technologies. As such, ASSISTments 2.0 is being developed with enhanced functionality that is based on the older system, such as the Content Builder.

The underlying code-base of Builder 2.0 is made up of three packages: Common, Service, and Client. Common includes code that is relevant to both the Service and Client projects. Service contains the server code that is invoked when users interact with Builder 2.0. Client includes the code that is used to create the user interface for Builder 2.0 and is solely executed by the user's web browser.

Additionally, Builder 2.0's code utilizes ASSISTments' own 'SDK 3.0'. This external project is the back-bone of the new Content Builder project as it provides the functionality needed to communicate with the ASSISTments servers. The project also makes use of the third-party software tools Apache Maven and Apache Tomcat for package management and hosting the Builder 2.0 web server.

This section will go into detail about the hierarchy of computer code that makes up Builder 2.0's project packages as well as the roles that the SDK, Apache Maven and Apache Tomcat fill in managing and running the project.

### 2.3.1 Common

The Common package defines the data structures used by both the Client and Service packages. The Client and Service packages use these data structures to send problems and problems sets back and forth to each other. For example, to retrieve an existing problem set from the ASSISTments database, the problem set needs to be sent to the front-end in a specific format. The format that the problem set data is stored in before the server sends it to the front-end is called an *SPSJsonContainer*. The structure *SPSJsonContainer* is defined below along with the other data structures that are used for sending information to and from the front-end and the back-end.

#### **NeedsToBeSavedPRContainer**

This data structure is used to hold a *NeedsToBeSaved* object for a problem. Since Builder 2.0 automatically saves the updates a user makes to their problem, the new data must be sent to the server for storage. The data type that stores the edits made to the problem is the *NeedsToBeSavedPRContainer*. This container object makes it easier for the server to parse data and update the problems in the database.

### **NeedsToBeSavedPSContainer**

This data structure holds a *NeedsToBeSaved* object for a problem set that contains all the problems the user is editing. When the user makes changes to the problem set, the data sent to the server takes the format of this data structure.

### **NeedsToBeSavedTContainer**

This data structure holds tutoring parameters which make up the hints and explanations for a problem and its answers.

### **SPSJsonContainer**

When a user fetches a problem set, the server sends the problem set to the front-end in this format. This *SPSJsonContainer* includes an *ISimplifiedProblemSet* which contains information about the problem set and a list of the problems within the problem set.

### **SPRJsonContainer**

To send the problems in the problem set, the server sends *SPRJsonContainers* to the front-end. The *SPRJsonContainer* includes an *ISimplifiedProblem* which contains all the information about the problem. For example, the type of question, the type of answers, the question text, and the answer text are all properties of the problem that are stored in this data structure.

### **ProblemInfoResponse**

This data structure contains some identifying properties of a problem and problem set and holds the success status of a saving operation. This data structure is created by the server and sent to the front-end.

## **2.3.2 Service**

The Service package contains the code that executes when the front-end requests or sends data to the server. The front-end executes these connections by sending a particular web request to Service in the form of a URL. This protocol of sending a web request to a server, and the server sending a different response back based on the data that the front-end sent, is called a RESTful API. Each URL that the server is listening on for front-end requests is called an endpoint. This section will define what each endpoint of the RESTful API does in this package.

**/createproblemset:** Takes in a problem set ID as a string and creates a problem set. It does not send any data back to the front-end client.

**/getproblemset:** Takes in a problem set ID as a string and sends an *SPSJsonContainer* of the problem set data to the front-end client.

**/getproblem:** Takes in a content key of a problem as a string and returns an *SPRJsonContainer* to the front-end client.

**/problemsetservice/save:** This endpoint is triggered when a problem set needs to be saved. It receives a *NeedsToBeSavedPSContainer* and returns a *ProblemInfoResponse* that indicates the status of the save request.

**/problemsetservice/verify:** This endpoint receives a *NeedsToBeSavedPSContainer* when the user wants to verify their problem. After the server verifies the problem, it sends back a *ProblemInfoResponse* that indicates the status of the verify request.

**/problemsetservice/publish:** This endpoint is triggered when a problem set needs to be published. The server receives a *NeedsToBeSavedPSContainer*, publishes the problem set, and sends the user a *ProblemInfoResponse* that indicates the status of the publish request.

**/problemservice/save:** This endpoint is triggered when a problem needs to be saved. The server receives a *NeedsToBeSavedPRContainer*, updates the old problem, and returns a *ProblemInfoResponse* that indicates the status of the save request.

**/problemservice/verify:** This endpoint is triggered when a problem needs to be verified. This functions similarly to ‘/problemsetservice/verify’ but it uses a *NeedsToBeSavedPRContainer* instead of a *NeedsToBeSavedPsContainer*.

**/problemservice/publish:** This endpoint is triggered when a problem needs to be published. This functions similarly to /problemsetservice/save but it uses a *NeedsToBeSavedPRContainer* instead of a *NeedsToBeSavedPSContainer*.

**/tutoringservice/save:** This endpoint is triggered when the hints of a problem need to be saved. The server receives a *NeedsToBeSavedTContainer* and sends back a *ProblemInfoResponse* that indicates the status of the save request. This endpoint was added by our team during development.

**/createproblemsetwithproblem:** Creates a brand new problem set with a problem inside of it, and then sends an *SPSJsonContainer* of the problem set data to the front-end client. This endpoint was added by our team during development.

### 2.3.3 Client

The Client package includes code that executes on a user’s web browser after it is converted to HTML, CSS, and JavaScript. All the UI elements and interactivity for Builder 2.0 is defined in this package. Three sub-packages are contained within Client to keep the code-base organized. The Animations sub-package contains the code that controls animating the UI elements on the user’s screen. The Views sub-package defines the components in Builder 2.0 that the user interacts with and inputs data into. Finally, the Client sub-package combines both packages to display the Builder 2.0 UI on the user’s screen. The three packages are described below in more detail.

## Animations

The code stored here handles the animations that display in the user's web browser when they access Builder 2.0. There are two animations defined in this package. The first animation causes a problem to fade onto the screen when the user creates it. The second animation scrolls the user to the problem that they click on when navigating the problems through the problem set explorer.

## Views

The code stored in this package defines all of the components that are to be used in the Client package. For example, these components include the elements controlling how the user inputs problem answers and how the user inputs question text.

## Client

This sub-package is where the front-end code is put together. It uses code defined in the Views package and Animations package. This package also defines some of its own data types that are used for parsing problems and displaying them in the UI. When the user navigates to the Builder 2.0 website, this code assembles all the components from the Views package and provides the functionality that allows the user to create and edit problem sets. Two vital functions are defined in this package. The first is *postRequestForSPS()*, which accepts two parameters. The first is a URL which is expected to be an endpoint of the Builder 2.0 RESTful API. The second parameter is a key that allows the server to keep track of which problem set is being manipulated. When this function is called with a valid URL and key, it sends the server a request to obtain a problem set. When the server fetches the problem set, it sends the client an *SPSJsonContainer* in response. As discussed in [Section 2.2.1](#), this *SPSJsonContainer* includes content keys for every problem within that problem set. For each of these content keys, *postRequestForSPS()* executes a function called *postRequestForSPR()*. This functions very similarly to the function that calls it except that it deals with problems and not problem sets. When the server sends the front-end an *SPRJsonContainer* for each problem in the problem set, this function parses the container and updates the UI so that the user can see all the problems contained within the problem set.

### 2.3.4 SDK 3.0

An SDK is computer code that allows developers to interact with servers in a safe way. This code is heavily abstracted so that the developers do not need to know the inner workings of the servers to implement any intended functionality. For example, saving an object to a database may take many lines of code which vary depending on which type of database is used, the version of that database, and the operating system the database is running on. It is not feasible for a developer to know every database and how to store custom objects to them. Instead, organizations, such as ASSISTments, develop SDKs that abstracts saving objects to a database.

The ASSISTments SDK is the underlying code that every component of the ASSISTments platform is built off of. In this enormous code-base, many internal processes are abstracted to make the code more readable and less repetitive. An important component defined in the SDK is the *getNeedsToBeSaved()* function. When called on by a problem or problem set, this function gets the fields that the user modified. The function returns the data structure containing the modified fields, and then sends it to the server so that the data the user entered gets saved. This data structure only contains the modified fields to increase performance. Sending the unmodified fields would cause the database to override old data with the same exact data, which wastes computation time.

### 2.3.5 Apache Maven

Apache Maven is a tool for building and managing any Java-based project (ASF, 2021). Developers do not write every functionality that they use in their project from scratch, so it is necessary to find and use different packages and sub-projects that can fill in missing functionalities. As developers find more and more packages that their project depends on, it becomes increasingly difficult to manage all of these dependencies. Maven automatically manages these dependencies for the developer through the use of a Project Object Model or POM. The POM is a document containing metadata and configurations for how a developer might want Maven to act when managing specific packages (Dilshan, 2018). Overall, Maven strives toward four objectives in its package management service (ASF, 2021):

- 1) Making the build process easy,
- 2) Providing a uniform build system,
- 3) Providing quality information, and
- 4) Encouraging better development practices.

The first objective is the core purpose mentioned above that shields the developer from managing potentially massive dependency trees and packages by automating a lot of the process. The second objective uses the POM as a standard for defining the metadata of packages so that all packages are defined in the same standard format. The third objective also makes use of the POM format to show quality information about a package such as the dependencies that a package needs and a change log of the package. The final objective is an ideal that Maven aims for by claiming that the Maven workflow will serve as a guideline for implementing good software practices. An example of this would be how Maven's directory structure can serve as a guideline for a base project's directory structure (ASF, 2021).

### 2.3.6 Apache Tomcat

Apache Tomcat is a web server and Servlet container for Java code (Fol, 2020). A web server or Servlet container is a Java class that is set up to intercept HTTP requests coming from a client and respond with the appropriate data and information. Tomcat is considered lightweight when compared to other web servers as it does not provide the full feature set from Java



Enterprise Edition, which is the standard for Java servers. Tomcat still proves satisfactory for many applications as it generally provides all of the features needed by those applications (Fol, 2020). Tomcat is set up to run as an independent platform tool that will run by itself on startup. To do so, applications need to deploy specific resources to the server. The deployment is done by copying .WAR (Web Application Resource) files to a \$CATALINA\_BASE/webapps/ directory. This directory is also where the Catalina component is located and is the actual Servlet container housing the implementation for the server. Another component called Coyote handles the HTTP communication and forwards the requests to the Catalina component (Fol, 2020). The main purpose for using Tomcat in an application is to facilitate the client server relationship that is standard for web applications.

## 2.4 GWT and GWT-Jackson

To effectively deliver a modern software solution that is uniform in its development, it is common practice to make use of already established software development tools and frameworks. These tools lower the amount of effort required by the programmer as they automate common software practices. These tools have also shown that they mesh well with E-Learning Ecosystems like ASSISTments. Bottom layer Web 2.0 technologies make websites more usable and convenient by facilitating the “subscription, access, propagation, reuse and compilation of small chunks of content” (Gutl et. al., 2008). These web technologies have also proved to be extremely scalable in terms of software development. This means that they are modular enough to be able to support both low and high traffic development.

The specific technologies used by the ASSISTments’ Builder 2.0 are the Google Web Toolkit (GWT) and GWT-Jackson. These systems provide the advantages of new web technologies detailed above. GWT is a general-purpose web development toolkit for building and optimizing complex browser-based applications (GWT, n.d.). This means that GWT provides many useful functions and utilities for writing software hosted through web browsers such as Google Chrome or Microsoft Edge. GWT’s goal is to enable productive development of web applications, such as ASSISTments’ Builder 2.0, through abstraction by providing functions and processes that can be utilized without explicit knowledge of lower-level browser web development (Ibid.). GWT also has a built-in compiler for translating its native Java code-base to the JavaScript language that is commonly used in web applications.

Additionally, Builder 2.0 makes use of another tool that builds off of GWT called GWT-Jackson, which is a public open-source add-on. GWT-Jackson is a JavaScript Object Notation (JSON) parser for GWT that allows for more customization of the serialization/deserialization process. JSON is a uniform data structure that many software applications use to send information between their outward client and server interfaces. The GWT-Jackson parser further automates the process of serializing and deserializing these data structures.

### 2.4.1 Java to JavaScript Translation

Java is the development framework language that ASSISTments chose to use when developing Builder 2.0. However, Builder 2.0 is ultimately meant to be easily accessed from any browser which lends itself better to the JavaScript language. JavaScript allows for efficient web development through its strong compatibility with Hypertext Markup Language (HTML). HTML is the standard for documents and applications designed to be displayed within a web browser, and it is commonly assisted by scripting languages such as JavaScript. GWT is able to mitigate the development language mismatch through its internal cross-compiler that translates Java source code to equivalent JavaScript code. This translation makes it possible for developers with more experience and comfortability using Java to write web applications that will end up using standard JavaScript (GWT, n.d.). In addition, GWT makes use of eXtensible Markup Language (XML) for webpage design as it links with Java code easily. These XML layout files are converted into HTML during compile time so that the UI designs will function properly with the generated JavaScript.

### 2.4.2 Serialization

Web applications tend to follow the standard client server model for providing feedback to its users. The client side for a web application is the digital document or page that the user sees as well as the underlying code that handles user interaction. However, websites need to be able to store the information that they take from users so that the data can perform its purpose. This process is generally done through server source code which performs more complex calculations as well as database management. Before the data taken from the user is sent to the back-end server, the data undergoes the process of serialization. As seen in [Figure 12](#), serialization is the process of converting the state of an object into a byte stream (GeeksForGeeks, 2019).

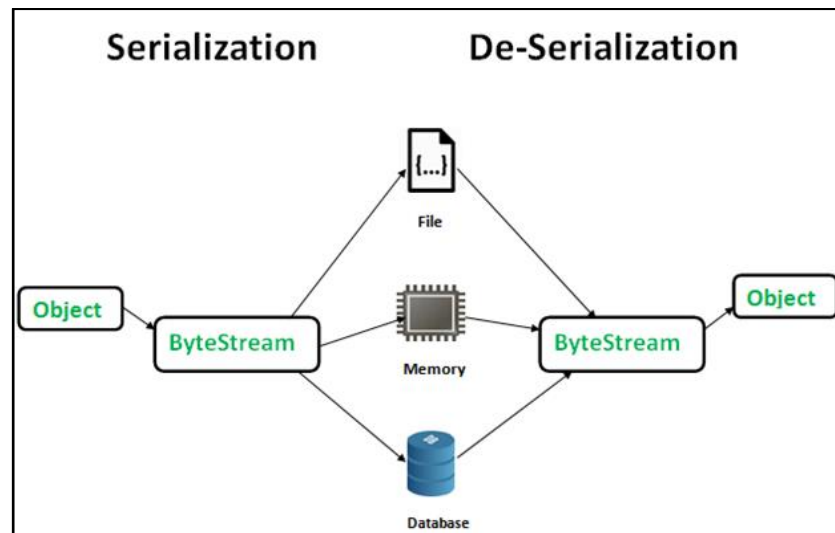


Figure 12: Serialization and Deserialization Process ([GeeksForGeeks](#), 2019)

Serialization is important to use when transferring data from the client to the server because the client and server may be written using different formats or source languages. Serialization converts the data from the client's format to a uniform standard. Once the data reaches the server code, there needs to be functionality that can convert the serialized standard byte stream into its own format. As shown in [Figure 12](#), this process is known as deserialization, and it is essentially the reversing of the byte code back to its original form in memory (GeeksForGeeks, 2019).

GWT and GWT-Jackson have their own form of the Serialization library that they make use of through a JavaBean. All that needs to be done to create this JavaBean is to build a Java class that adheres to the rules detailed in [Figure 13](#). The class does not necessarily need to do any calculations as the most important part of the class is that it can accurately model all of the data that is being passed to the server. GWT-Jackson allows for customization of the built in GWT serialization methods through the use of binding tag annotations.

### 3.1. What Is a *JavaBean*?

A *JavaBean* is still a POJO but introduces a strict set of rules around how we implement it:

- Access levels – our properties are private and we expose getters and setters
- Method names – our getters and setters follow the *getX* and *setX* convention (in the case of a boolean, *isX* can be used for a getter)
- Default Constructor – a no-argument constructor must be present so an instance can be created without providing arguments. for example during deserialization
- Serializable – implementing the *Serializable* interface allows us to store the state

*Figure 13: Rules for a JavaBean (baeldung, 2020)*

## 2.4.3 Binding Tag Annotations

Annotations within Java allow developers to have more customization in how their source code is compiled in addition to providing more runtime processing (Oracle, 2020). Annotations are how GWT-Jackson customizes key serialization features. As seen in [Figure 14](#), there are annotations linked to both a method and its arguments. These annotations are the keywords preceded by an '@' symbol. The effect of these annotations is that they are giving the built-in GWT serialization process important metadata to adhere to. In the case of [Figure 14](#), the metadata is the various fields that will bind together during serialization.

```
@JsonCreator
public ProblemInfoJsonSPS(@JsonProperty("sps") ISimplifiedProblemSet sps,
    @JsonProperty("respo") String response)
{
    this.sps = sps;
    this.response = response;
}
```

```
public ProblemInfoJsonSPS(@JsonProperty("respo") String response)
{
    this.response = response;
}
```

*Figure 14: Example of Binding Annotations in Java*

There are also additional general annotations that appear at the top of class structures that do not link to specific methods or properties defined inside of the class. These are usually additional rules for the compiler to follow when serializing, such as what to do when the serializer receives a JSON input that has no properties bound to it, and what types of values are allowed in the serialized byte stream.

## 2.5 UI Creation with GWT

ASSISTments 2.0 is an online tool and is therefore accessed by visiting its associated webpage. A webpage needs to be designed in a format that is readable by a computer and its web browsers. Known as markup languages, these formats utilize human-readable tags that represent elements to be processed by the computer and displayed on a webpage (Christensson, 2011). Webpages are typically designed and displayed using Hyper Text Markup Language, or HTML. In fact, about 91.4% of all websites utilize some version of HTML as its primary markup language (Q-Success, 2020). The most common way to create the design of a webpage with HTML is to simply write a .html file. However, since Builder 2.0 is built with GWT, it utilizes a different markup language that automatically gets converted into HTML upon compiling the project's code-base.

### 2.5.1 Using XML to Design a UI

Since GWT is a web toolkit that contains widgets and extensions that are separate from basic HTML, a web-designer cannot properly access these features by writing a straightforward .html file. Therefore, GWT makes use of eXtensible Markup Language, or XML, for designing a user interface. XML is a “metalanguage” that is used to create application specific markup languages (Christensson, 2006). This flexibility of XML powers the GWT UiBinder framework, which allows web-designers to create HTML websites with GWT specific widgets (GWT, n.d.).

The UiBinder framework is very similar to an HTML file since GWT XML files are created with the same three key parts:

- 1) An initial document reference that defines what the file is,
- 2) A section for adding styling parameters, and
- 3) A hierarchy that defines the layout of each UI element and widget.

For reference, [Figure 15](#) details the XML file that defines the layout for the UI elements within Builder 2.0 that control adding answers to problems. The XML file begins with the lines:

```
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
  <ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
    xmlns:g="urn:import:com.google.gwt.user.client.ui">
```

The first line starts with *!DOCTYPE* and declares that the XML file will be utilizing GWT's UiBinder framework. This is done by providing a *SYSTEM* reference with the GWT Document Type Definition (DTD) URL that is shown above and simply labeling it as *ui:UiBinder*. Doing so ensures that any HTML entities are properly configured and imported inside of the XML UI design (GWT, n.d.).

Since XML is a markup language that uses tags, the second line declares the *ui:UiBinder* tag at the top of the hierarchy so that GWT's UiBinder widget is set as the root element for the UI layout. Inside the *ui:UiBinder* tag, there are two namespaces declared using *xmlns*; *ui* and *g*. These namespaces are needed so that any UI elements used inside of the file are found by the compiler and loaded into the webpage.

```
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
  xmlns:g="urn:import:com.google.gwt.user.client.ui">
  <ui:style>
    .important {
      font-weight: bold;
    }
    .spacing {
      margin: 4px 2px 2px 2px;
      width: 94%;
    }
    .answer-hover-effect {
      margin: 3px 0px;
    }
    .answer-hover-effect:hover {
      box-shadow: 0px 0px 3px #ddd;
    }
  </ui:style>
  <g:HorizontalPanel width="100%" addStyleNames="{style.answer-hover-effect}">
    <g:Cell horizontalAlignment="ALIGN_CENTER" verticalAlignment="ALIGN_MIDDLE">
      <g:TextArea ui:field="inputAnswer" addStyleNames="{style.spacing}" visibleLines="1"></g:TextArea>
    </g:Cell>
    <g:Cell horizontalAlignment="ALIGN_CENTER" verticalAlignment="ALIGN_MIDDLE">
      <g:CheckBox ui:field="correctCheck"></g:CheckBox>
    </g:Cell>
    <g:cell horizontalAlignment="ALIGN_CENTER" verticalAlignment="ALIGN_MIDDLE" >
      <g:TextArea ui:field="answerExplanation" addStyleNames="{style.spacing}" visibleLines="1"></g:TextArea>
    </g:cell>
    <g:Cell horizontalAlignment="ALIGN_CENTER" verticalAlignment="ALIGN_MIDDLE">
      <g:Image ui:field="removeAnswer"/>
    </g:Cell>
  </g:HorizontalPanel>
</ui:UiBinder>
```

Figure 15: AddAnswerView XML File

The next section of the XML layout details specific stylings that will be applied to elements inside of the file:

```
<ui:style> ...  
    .spacing {  
        margin: 4px 2px 2px 2px;  
        width: 94%;  
    } ...  
</ui:style>
```

Any form of styling specific to an XML layout file is denoted by the tag *ui:style*. Inside this tag, an infinite amount of styling effects can be declared. The style shown above is named *spacing* and when applied, will modify the margins and width of a UI element. The styling used by GWT is industry standard as it utilizes Cascading Style Sheets, or CSS. For in depth details on the types of styling that CSS offers, refer to the [CSS Reference](#) provided by W3Schools.

The last part of GWT's XML layout files detail the hierarchy of all the elements and widgets that will be displayed on the webpage. As shown in [Figure 15](#), the section of Builder 2.0's UI that deals with adding answers to a problem section is contained inside of a *HorizontalPanel*. That panel contains four cells that hold the *TextAreas*, *CheckBox*, and *Image* that makes up the user interface for inputting answers. These elements contain properties that detail how it should be styled and formatted. For example, the cell containing a *TextArea* for inputting answers is written as:

```
<g:Cell horizontalAlignment="ALIGN_CENTER" verticalAlignment="ALIGN_MIDDLE">  
    <g:TextArea ui:field="inputAnswer" addStyleNames="{style.spacing}"  
        visibleLines="1"></g:TextArea>  
</g:Cell>
```

The cell is given two formatting specifications of *horizontalAlignment* and *verticalAlignment*. Neither of these CSS options are being handled by the style section of the XML document detailed earlier. Instead, these two properties are using *ALIGN\_CENTER* and *ALIGN\_MIDDLE*, which are CSS configurations declared inside of an external CSS file. For details on how to link external CSS files to a GWT XML file, refer to the GWT Documentation on [CSS functionality](#).

As for the *TextArea*, it has the properties of *ui:field*, *addStyleNames* and *visibleLines*. *ui:Field* is used to give this element a unique name. *addStyleNames* applies a CSS style that is defined inside of the style tag of the XML file. In this case, the *TextArea* is given the *spacing* styling detailed earlier. Lastly, *visibleLines* is set to '1', which denotes that the *TextArea* will only be one line long. For details on all of GWT's UI elements and their properties, refer to the GWT Documentation on [User Interfaces](#).

Once a GWT based project is compiled and loaded inside of a web browser, the XML files are converted to HTML and the website is displayed. For reference, [Figure 16](#) shows the UI for adding answers to a problem on the Builder 2.0 webpage.

Answer(s):	Correct Option	
Answer 1	<input checked="" type="checkbox"/>	Feedback: True
	<input type="checkbox"/>	Feedback: False
	<input type="checkbox"/>	Feedback: False

[Add Answer](#)

Figure 16: Builder 2.0's UI for Adding Answers

## 2.5.2 Manipulating UI Elements in Code

GWT's XML UI creation is only capable of designing a webpage and does not provide any way of adding functionality to the UI elements. There are two essential steps needed to manipulate a GWT UI. The first is to simply add unique *ui:field* tags to each element that will be manipulated. For example, the Image element in Figure 15 has its *ui:field* property set to *removeAnswer*, which indicates this UI element will be used to delete an answer. The second step is to build a UI controller by creating a Java class that correlates with the XML UI layout file. As seen in Figure 17, a Java controller class for Builder 2.0's *AddAnswerView* is created with its first few lines implementing the UiBinder Framework.

```
public class AddAnswerView extends Composite
{
    private static AddAnswerViewUiBinder uiBinder = GWT.create(AddAnswerViewUiBinder.class);

    interface AddAnswerViewUiBinder extends UiBinder<Widget, AddAnswerView> {}
```

Figure 17: Builder 2.0's UI for Adding Answers

First, a static *uiBinder* variable for the specific UI to be manipulated must be created (GWT, n.d.). In the case of *AddAnswerView*, the *uiBinder* variable is defined by:

```
static AddAnswerViewUiBinder uiBinder = GWT.create(AddAnswerViewUiBinder.class);
```

In addition, a UiBinder interface also needs to be declared to properly configure the Java class (GWT, n.d.). The interface should be titled according to the specific UI being manipulated, and it must extend the UiBinder framework with the widget class and the current Java class defined as its generic types. The UiBinder interface for *AddAnswerView* is defined by:

```
interface AddAnswerViewUiBinder extends UiBinder<Widget, AddAnswerView> {}
```

Now that the Java class has been properly set up for utilizing the UiBinder framework, the next step is to initialize the UI elements and widgets that will be modified. First, the constructor of the Java class must initialize the UiBinder so that the XML file and the Java class

are linked. [Figure 18](#) shows a snippet of the constructor for *AddAnswerView* that performs this action.

```
public AddAnswerView(AnswerTypeLayoutView attached)
{
    initWidget(uiBinder.createAndBindUi(this));
}
```

*Figure 18: AddAnswerView.java Constructor*

After linking the controlling Java class to its XML layout file, the UI elements can now be declared and handled by Java code. Declaration of the UI elements is done by utilizing the GWT binding tag `@UiField` as seen in [Figure 19](#). This tag notifies the Java controller that the proceeding variable will be linked to an element inside of the XML layout file.

```
@UiField
Image removeAnswer;
@UiField
TextArea inputAnswer;
@UiField
CheckBox correctCheck;
@UiField
TextArea answerExplanation;
```

*Figure 19: Declaring GWT UI elements in Java Code*

Lastly, the declared UI elements can now be controlled by the Java controller class. Adding controlling functionality to a UI element is done in two ways:

- 1) Utilizing GWT's built-in UI event handler functions, and
- 2) Implementing custom event handler functions.

Both of these methods involve the use of event handler functions, which execute specific functionality whenever a user interacts with a UI element (GWT, n.d.).

The GWT built-in event handler functions cover a wide range of possible interactions with a UI element and requires three steps for utilization:

- 1) Calling one of GWT's specific event handler attachment functions on a UI element,
- 2) Declaring a new event handler function inside of the attachment function with proper parameters, and
- 3) Providing functionality to the new event handler function.

All three steps can be seen in [Figure 20](#), which details the implementation of a button click event while using a GWT built-in event handler function.



```

public class MyFoo extends Composite {
    Button button = new Button();

    public MyFoo() {
        button.addClickHandler(new ClickHandler() {
            public void onClick(ClickEvent event) {
                handleClick();
            }
        });
        initWidget(button);
    }

    void handleClick() {
        Window.alert("Hello, AJAX");
    }
}

```

Figure 20: Example of a GWT Built-In Event Handler Function ([GWT](#), n.d.)

After the button is declared and initialized, the GWT built-in event handler attachment function, `addClickHandler()` is called on the button UI element. Inside of that attachment function, a new `ClickHandler()` function is created and labeled as `onClick()`. The `onClick()` function is also properly set up for handling click events with the parameter type of `ClickEvent` being passed into its function header. Lastly, the `onClick()` function is given functionality as it calls `handleClick()`. The end result is when a user clicks the button in the UI, an alert will appear on screen that reads “Hello, AJAX”. For details on all of GWT’s built-in event handler functions, refer to the GWT Documentation on [User Interfaces](#).

Custom event handler functions require less code for setup and provide greater flexibility for function naming and handling UI events (GWT, n.d.). This type of UI element manipulation requires only two steps:

- 1) Utilizing the `@UiHandler` binding tag with an element's unique identifier, and
- 2) Writing a custom event handler function with the proper parameters.

[Figure 21](#) details these two simple steps by showing how a button click event can be manipulated with a custom event handler.

```

public class MyFoo extends Composite {
    @UiField Button button;

    public MyFoo() {
        initWidget(button);
    }

    @UiHandler("button")
    void handleClick(ClickEvent e) {
        Window.alert("Hello, AJAX");
    }
}

```

Figure 21: Example of a Custom Event Handler Function (GWT, n.d.)

After the declaration and initialization of the button, the custom event handler is first setup with the `@UiHandler` tag. The tag takes in the *ui:field* unique identifier “button”, which refers to the button UI element declared in the Java controller and the associated XML UI layout file. Doing so lets the Java controller know that the proceeding functionality will be executed only on that specific UI element. The second and final part of the custom event handler is completed with the declaration of the function *handleClick()*. It takes in the proper `ClickEvent` parameter and contains the needed functionality for displaying the alert message “Hello, AJAX”.

### 2.5.3 Utilizing Tiny MCE with GWT

GWT is also versatile in the sense that it allows third-party UI libraries to be implemented into projects (GWT, n.d.). Builder 2.0 takes advantage of this with its implementation of TinyMCE4. TinyMCE4 is a JavaScript powered library that converts a basic HTML TextArea into a fully customizable rich-text editor (TTI, 2021). Rich-text editors such as TinyMCE are UI widgets that allow users to edit inputted text with various options such as font type, font size and font color. These widgets also provide users with the ability to link media such as images, videos, and audio to the text. For Builder 2.0, the addition of these text editing functionalities means that teachers can create more dynamic and complex questions for their students. For a full list of the options that TinyMCE offers, refer to the [TinyMCE Documentation](#).

GWT makes the implementation of additional libraries such as TinyMCE fairly easy. For TinyMCE, a new XML layout file called TinyMCEView needs to be created. Since TinyMCE is an extension of a basic HTML TextArea, the XML file only needs to include a TextArea ([Figure 22](#)).

```

<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
             xmlns:g="urn:import:com.google.gwt.user.client.ui">
  <g:HTMLPanel>
    <g:TextArea ui:field="ta" width="485px" height="110px" ></g:TextArea>
  </g:HTMLPanel>
</ui:UiBinder>

```

*Figure 22: TinyMCEView XML File*

The TextArea is then modified and customized into a TinyMCE text editor through its own Java controller class. The steps to create a Java controller class are detailed in [Section 2.4.2](#). Within the TinyMCE controller class, developers can add text editing options and fully customize TinyMCE for their specific needs. The controller class is also where developers can link external CSS files to the TinyMCE so that its appearance is customized. For more information on how to implement these customizations inside of a Java controller class, refer to the [TinyMCE Documentation](#).

Lastly, the TinyMCE text editor can be implemented into webpages by using the TinyMCE tag. For example, the UI for adding answers to problems also has a TinyMCE variant. A snippet of the XML UI layout file for this variation can be seen in [Figure 23](#).

```

<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
             xmlns:g="urn:import:com.google.gwt.user.client.ui"
             xmlns:c="urn:import:org.assistments.contentbuilder.views">
  <ui:style>
    .important {
      font-weight: bold;
    }
    .spacing {
      margin: 4px 2px 2px 2px;
    }
  </ui:style>
  <g:HorizontalPanel>
    <g:Cell horizontalAlignment="ALIGN_CENTER" verticalAlignment="ALIGN_MIDDLE" height="90px">
      <c:TinyMCEView ui:field="inputAnswer"/>
    </g:Cell>
  ...

```

*Figure 23: Snippet of AddAnswerMCEView XML File*

In this variation, a new namespace, known as *c*, has been added and it references the directory location of *TinyMCEView*. In addition, the *TextArea* tag for *inputAnswer* has been replaced with the *TinyMCEView* tag that is being referenced with namespace *c*. With these changes, *inputAnswer* will now display the TinyMCE TextArea that is being converted from the basic HTML TextArea defined in the TinyMCEView XML File. For reference, [Figure 24](#) shows the TinyMCE UI for adding answers to a problem on Builder 2.0's website.

Answer(s):

☒ Use enhanced answer editor


Correct Option

Font Sizes ▾ **B** *I* U ~~S~~


- ☰
- ☷

A ▾ A ▾ ↶ ↷ 


 ▾

Ω  $x_2$   $x^2$  

Answer 1: TinyMCE

☒ Feedback: True 

[Add Answer](#)

*Figure 24: Builder 2.0's TinyMCE UI for Adding Answers*

## 3. Implementation

The goal of this project was to further the development of a Minimum Viable Product for the ASSISTments 2.0 Content Builder component so that educators could create, save, and modify their own customizable problems.

Our measurable objectives were;

- 1) To implement RESTful API calls that connect the front and back ends of Builder 2.0 to keep problems up-to-date.
- 2) To enhance the user experience by improving and condensing the Builder 2.0 user interface.
- 3) To provide users with the ability to import existing content into their projects.
- 4) To automate the development build process via scripting.
- 5) To explore potential security implementations that can protect user information and prevent cyberattacks.

To accomplish these objectives, our team worked directly with the Lead Software Engineer on the ASSISTments 2.0 Builder Project, Ashish Gurung, for a seven-month period that began on August 31<sup>st</sup>, 2020 and ended on March 18<sup>th</sup>, 2021. The project work was conducted remotely by each team member in which we used online teleconference and version control software to manage our work. The features we implemented into the Builder 2.0 project enhanced the user experience of the website, allowed users to collaborate with each other and provided a means for automatically saving data to the ASSISTments servers. The development described below outlines the functionality that was implemented to achieve the goal and objectives.

### 3.1 Saving Problems: Front-End

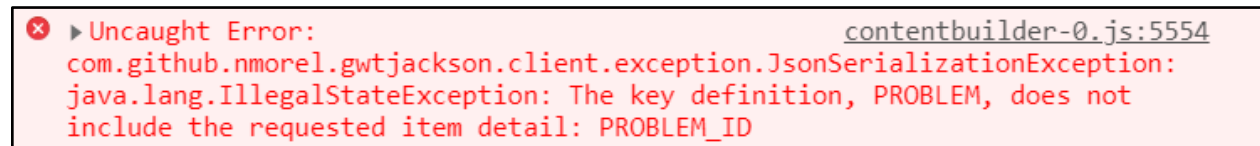
Upon beginning this project, the UI portion of Builder 2.0's front-end was mainly completed. All of the UI elements such as TextBoxes, TinyMCE editors, RadioButtons and CheckBoxes were positioned and set up for their intended functionality. However, the UI was not connected to any back-end services. Therefore, any problems created within the UI would only persist for the duration that the webpage was open. The UI elements had still not been given functionality that would save a user's progress on making or editing problems within a problem set which meant that there was no data persistence in Builder 2.0 at all. Our team implemented data persistence for problems on the front-end by making three modifications to the Client portion of the Builder 2.0 code-base:

- 1) Ensuring that problems generated in the UI were properly linked to problems saved in the ASSISTments database,
- 2) Making calls to the saving service RESTful API for each element in the UI, and
- 3) Removing duplicate code and streamlining calls to the saving service RESTful API.

Additionally, we utilized software engineering testing practices to ensure that the new front-end functionality was working as intended and without error.

### 3.1.1 Linking Problems in the UI to the Database

The process of implementing persistent data into Builder 2.0 began with figuring out a bug in the Builder 2.0 front-end code. The ASSISTments Team had set up the front-end to connect with the back-end with two key functions: *postRequestForSPS()* and *postRequestForSPR()*. Respectively, these functions are used to obtain data for a problem set (SPS) and its internal problems (SPR). Upon retrieval of the data from the database, the UI would be updated to display that problem set and its internal problems. While the initial retrieval and display functioned as intended, any attempts to edit the question text of a problem would cause a GWT Jackson serialization error to be thrown (*Figure 25*).



*Figure 25: GWT Jackson PROBLEM\_ID Serialization Error*

Since the question text of a problem was one of the few elements in the UI that was linked with a saving service RESTful API call, it was clear that attempting to update a problem in the database was not functional.

To track down the issue causing this bug, we had to figure out what could cause such a serialization error. After reviewing the requirements of the GWT-Jackson JSON serialization process, the problem was narrowed down to a parameter failing to serialize since it had a null value. With that information, we wrote debugging code to see the value of each parameter that needed to be serialized before a saving service RESTful API request could be made. We found that every time the front-end system would try and update a problem, the content key for the problem was null. After consulting with Ashish on our findings, we learned that the type *PROBLEM\_ID* refers to a problem's content key. Ashish also informed us that a problem's content key is used to look up and modify a specific problem inside of the database. Therefore, the proper serialization of the type *PROBLEM\_ID* was vital for maintaining a connection between the front-end UI and the back-end services that save user data.

To implement a fix, we had to look at the process of generating problems in the UI based on data retrieved from the database. Once *postRequestForSPS()* pulled a problem set's data from the database, it would call *postRequestForSPR()* for each problem found inside of the problem set. Inside of *postRequestForSPR()*, the individual problems would be drawn into the UI, but these UI problems were never assigned a content key. Without a valid content key, serialization would fail because there was no unique key to associate the problems displayed in the UI with the problems stored in the database. Since the method header for *postRequestForSPR()* took in

the correct content key as a parameter, the bug fix ended up being an addition of one line of code to *postRequestForSPR()*;

```
problemBuilder.setContentKey(contentKey);
```

Now, every time a problem set is loaded from the ASSISTments database, its internal problems will be properly linked so that modifications can be made without error.

### 3.1.2 Adding Saving Service API Calls to the UI

After implementing the proper serialization that allowed problems in the UI to correlate with the database, our team then needed to add problem saving functionality to the rest of the UI. To do this we needed to link all the UI elements relating to problems within the project to a back-end service call. The Java function to do so had already been written. Known as *makeAsyncCall()*, this function requires two parameters to be passed into its header:

- 1) A URL extension for Builder 2.0's saving service RESTful API, and
- 2) A default GWT requestor that allows GWT projects to make API requests.

When called upon, the *makeAsyncCall()* method will call the API methods associated with the passed in URL extension and perform a saving action. For example, in the case of *AddAnswerView*, it utilized the URL extension *ContentBuilderService/problemservice/save* which indicates that the problem associated with the current answer will be saved. [Figure 26](#) illustrates the method header and the parameter values passed into *makeAsyncCall()* inside of the *AddAnswerView* Java controller class.

```
private String saveProblemUrl = "/ContentBuilderService/problemservice/save";  
private static final Requestor requestorSave = GWT.create(Requestor.class);  
  
private void makeAsyncCall(String url, Requestor requestor)  
{
```

*Figure 26: The makeAsyncCall() for AddAnswerView.java*

Our team needed to make sure the *makeAsyncCall()* method was being called by every UI element within Builder 2.0. To do so, we made use of both GWT's built-in and custom event handler functions. The two built-in event handler functions that were implemented are:

- 1) *BlurEvent* Handler: Executes when the UI in focus becomes unfocused, and
- 2) *ValueChangeEvent* Handler: Executes when the value of a UI element changes.

The *BlurEvent* handler was implemented by adding *onBlur()* calls to the *TinyMCE* and title bar UI elements. The *ValueChangeEvent* handlers were implemented by adding *onValueChanged()* calls to all of the basic *TextAreas* that did not utilize the *TinyMCE* editor. Both of these handlers were needed so that when a user types a new value into a text editor, the value would also be updated in the ASSISTments database. This ensured that answers, questions, explanations, hints, and problem titles would be saved every time a user made a change to them.

The implementation of these handlers required two steps:

- 1) Creating the event handler method and calling *makeAsyncCall()* within it, and
- 2) Utilizing GWT's built-in attachment functions to link the handler method to a UI element.

For UI handlers that would only be used on one element in the current Java controller class, these two steps were able to be done together. An example can be seen with the *BlurEvent* handler that was added to the TinyMCE editor for *AddAnswerMCEView* (Figure 27). However, to eliminate code redundancy, these steps were separated when an event handler was needed for more than one UI element. Figure 28 demonstrates the separation by detailing how the two basic *TextAreas* in *AddAnswerView* utilize the same *ValueChangeEvent* handler.

```
@Override
public void onInitialize(TinyMCEView view)
{
    view.addBlurHandler(new BlurHandler()
    {
        @Override
        public void onBlur(BlurEvent event)
        {
            GWT.log("onblur triggered from the TinyMCE:jasvaside view : AddMCEAnswerView " +
                inputAnswer.getText());
            attachedTo.notifyChange();
            makeAsyncCall(saveProblemUrl, requestorSave);
        }
    });
}
```

Figure 27: *AddAnswerMCEView.java* *BlurEvent* Handler

```
ValueChangeHandler<String> valChangedHandler = new ValueChangeHandler<String>()
{
    @Override
    public void onValueChange(ValueChangeEvent<String> event)
    {
        attachedTo.notifyChange();
        makeAsyncCall(saveProblemUrl, requestorSave);
    }
};

answerExplanation.addValueChangeHandler(valChangedHandler);
inputAnswer.addValueChangeHandler(valChangedHandler);
```

Figure 28: *AddAnswerView.java* *ValueChangeEvent* Handler

Our team also needed to add *makeAsyncCall()* within the custom event handlers that had been setup within the Builder 2.0 project. These event handlers are denoted by the GWT *@UiHandler* binding tag that precede a method declaration. The Builder 2.0 front-end uses these custom events on one-action UI elements, such as Buttons, Images, CheckBoxes and



RadioButtons. Since the UI had already been created, these custom event handlers were already added to the code-base. However, they lacked a method call to *makeAsyncCall()*. This meant that UI actions such as deleting an answer or changing the problem type were not being saved. To implement this functionality, a call to *makeAsyncCall()* was added to the majority of custom event handlers. [Figure 29](#) shows the *makeAsyncCall()* added to the delete image in *AddAnswerView*, which now ensures that a deleted answer is updated in the ASSISTments database.

```
@UiHandler("removeAnswer")
void onClick(ClickEvent e)
{
    this.removeFromParent();
    attachedTo.notifyChange();
    makeAsyncCall(saveProblemUrl, requestorSave);
}
```

*Figure 29: AddAnswerView.java Custom Event Handler*

For a list of all the UI elements given *makeAsyncCall()* implementations, refer to [Appendix A](#).

### 3.1.3 MakeAsyncCall Class

The *makeAsyncCall()* functionality was very common across many of the UI views as it was the main method used for taking data inputted in the front-end and saving it in the back-end database. Builder 2.0 was originally set up with a separate instance of *makeAsyncCall()* written in each Java controller class responsible for manipulating the UI elements that made up problems. Our team found that these instances of *makeAsyncCall()* were extremely similarly if not exactly the same. So, we decided to move this method and its functionality into its own class within the client directory instead of having code repeated across many classes. This would reduce the technical debt that could accumulate in the project as any change in the functionality for *makeAsyncCall()* would only need to be made in one place.

The class we created to contain this functionality is *MakeAsyncCall.java*. It is designed as a singleton class so that any view can access it while ensuring its contents persist across multiple executions of the client. Its persistence is created by having a static instance of itself defined as an internal property and then providing other classes a method to get that instance ([Figure 30](#)).

```
public static MakeAsyncCall getInstance()
{
    if (instance == null)
    {
        instance = new MakeAsyncCall();
    }
    return instance;
}
```

*Figure 30: Singleton Setup for MakeAsyncCall.java*

By converting the *makeAsyncCall()* function into a Java class, it gave us the ability to implement versions of its main function by utilizing the software development practice of overloading. The overloaded functions we implemented allow *MakeAsyncCall.java* to differentiate between saving problems, problem sets, and tutorings. Classes that need to save a specific type of content can simply call the associated overloaded function labeled *call()* within *MakeAsyncCall.java*.

The persistence of the singleton class design is important to an additional modification that was added to the saving functionality. This modification was the implementation of the *filterDuplicates()* function. Since the main purpose *MakeAsyncCall.java* is to save user inputs to the back-end database, there is a possibility that the code-base will run into saving issues if any UI fields are modified again before their previous modifications were saved. The *filterDuplicates()* method mitigates this issue by keeping its own version of the changes within the class instance. Whenever a saving call is made, *filterDuplicates()* compares the newly modified fields to the changes that were made during previous saving calls. If a match is found, *filterDuplicates()* will block the duplicate data from being sent to the back-end. This lowers the amount of data being sent to the back-end and ensures that automatic saving will be efficient and error free. The *filterDuplicates()* method also has multiple overloaded versions to account for differences between problems, problem sets, and tutorings.

As described in [Section 3.1.2](#), the *makeAsyncCall()* functionality was added to every UI element that handled problem data. Now that all the functionality for this function was moved into the *MakeAsyncCall.java* singleton class, the calls to the outdated function had to be replaced by calls to *MakeAsyncCall.java*. To do this, our team replaced every call to *makeAsyncCall()* with:

```
MakeAsyncCall.getInstance().call();
```

An example of this updated implementation can be seen in [Figure 31](#) which details the *BlurEvent* handler located within *ProblemEditorPanelView.java*. The *onBlur()* method triggers the saving of the current problem by getting the instance of *MakeAsyncCall.java* and executing the function *call()* with the *problemBuilder* variable passed into it.

```
@Override
public void onBlur(BlurEvent event)
{
    GWT.log("onblur triggered from the TinyMCE:jasvside view :
           problemEditorPanelView" + questionText.getText());
    updateProblemTitle();

    problemBuilder.setQuestion(questionText.getText());
    MakeAsyncCall.getInstance().call(problemBuilder);
}
```

*Figure 31: The onBlur() Function in ProblemEditorPanelView.java*

The first step of the call function is to move into the *filterDuplicates()* method for problems. The first step of the *filterDuplicates()* function is to make sure that we can get a *needsToBeSaved* object out of the *problemBuilder* since it will ultimately be passed to the service side to finish the save. This is done within a try catch block to ensure that any potential builder exceptions do not crash the website (*Figure 32*).

```
if (problemBuilder != null)
{
    try
    {
        needsToBeSaved = problemBuilder.getNeedsToBeSaved();
    }
    catch (BuilderException e)
    {
        GWT.log("Builder Exception occurred");
        e.printStackTrace();
    }
}
```

*Figure 32: Extracting a needsToBeSaved Object From a problemBuilder Object*

With the initial *needsToBeSaved* object created, *filterDuplicates()* will now filter out duplicate fields that have not yet been saved since the last method call. This required the use of storage tools that were capable of comparing similar data structures. Our team decided to use HashMaps to complete this functionality. For example, there is a HashMap that holds *problemBuilders* and their associated *needsToBeSaved* objects which contain the list of the modified fields. *Figure 33* details the three HashMaps used for each potential saving type. The first HashMap is used for problems while the additional two are used for problem sets and tutorings.

```
private static Map<IProblemBuilder<?>,
    Set<NeedsToBeSaved<ISimplifiedProblem, SimplifiedProblemFieldType>>> problemMap =
    new HashMap<IProblemBuilder<?>, Set<NeedsToBeSaved<ISimplifiedProblem,
        SimplifiedProblemFieldType>>>>();

private static Map<ISimpleProblemSetBuilder, Set<SimplifiedProblemFieldType>> problemSetMap =
    new HashMap<ISimpleProblemSetBuilder, Set<SimplifiedProblemFieldType>>>();

private static Map<ITutoringBuilder<?, ?>,
    Set<NeedsToBeSaved<? extends ISimplifiedTutoring, SimplifiedTutoringFieldType>>> tutoringMap =
    new HashMap<ITutoringBuilder<?, ?>,
        Set<NeedsToBeSaved<? extends ISimplifiedTutoring, SimplifiedTutoringFieldType>>>>();
```

*Figure 33: The Three HashMaps Used for Filtering in MakeAsyncCall.java*

These HashMaps are populated with a new *needsToBeSaved* object for the passed in builder each time the *filterDuplicates()* method is called, which allows the class to keep track of the data from previous save attempts. From here, the *filterDuplicates()* method needs to compare the current *needsToBeSaved* object's *modifiedFields* with *modifiedFields* from previous saving calls while also keeping track of which *modifiedFields* have not been saved yet. This process is

handled by a for-loop structure that checks each *SimplifiedProblemFieldType* and determines if its associated content has already been queued for saving (*Figure 34*). If it has been, that specific *modifiedField* is removed.

```
Set<SimplifiedProblemFieldType> newModifiedFields = new HashSet<>(needsToBeSaved.getModifiedFields());
for (NeedsToBeSaved<ISimplifiedProblem, SimplifiedProblemFieldType> ntbs : prevNTBSs)
{
    for (SimplifiedProblemFieldType sps : needsToBeSaved.getModifiedFields())
    {
        switch (sps)
        {
            case NAME:
                if (needsToBeSaved.getContentToSave().getName().equals(
                    ntbs.getContentToSave().getName()))
                {
                    newModifiedFields.remove(SimplifiedProblemFieldType.NAME);
                }
                break;
        }
    }
}
```

*Figure 34: Snippet of the For-Loop Structure within filterDuplicates()*

While there is a different case for each *SimplifiedProblemFieldsType*, many of them function similarly to the *NAME* case shown in *Figure 34*. The one exception is the *ANSWER* enumeration which has a special comparison since it is a list of values rather than just one value. The code for the *ANSWER* case is detailed in *Figure 35*.

```
case ANSWERS:
    List<SimpleAnswer> oldAnswers = ntbs.getContentToSave().getAnswers();
    for (SimpleAnswer newAnswer : needsToBeSaved.getContentToSave().getAnswers())
    {
        if (oldAnswers.contains(newAnswer))
        {
            oldAnswers.remove(newAnswer);
            needsToBeSaved.getContentToSave().getAnswers().remove(newAnswer);
        }
    }
    if (needsToBeSaved.getContentToSave().getAnswers().size() == 0)
    {
        newModifiedFields.remove(SimplifiedProblemFieldType.ANSWERS);
    }
    break;
```

*Figure 35: The ANSWER Case Comparison within filterDuplicates()*

After the comparisons are completed, the function returns the true list of *modifiedFields* in a new *needsToBeSaved* object and continues execution back in the main *call()* function.

With the new *needsToBeSaved* object that accounts for unconfirmed saves, the *call()* function now only needs to send the object to the service side so that it can perform the save. The transfer is done by creating a *NeedsToBeSavedProblemContainer* that wraps the *needsToBeSaved* object so that it can be serialized when the request is made to the server. The

request is handled by making use of the *GWT.create()* method to make a JavaScript post request which also comes with a promise for when the request returns back, as seen in [Figure 36](#).

```
NeedsToBeSavedPSContainer problemSetDTO = new NeedsToBeSavedPSContainer(
    "Post request for save from problem.", needsToBeSaved);
try
{
    requestor.req(saveProblemSetUrl).payload(problemSetDTO).post(ProblemInfoResponse.class)
        .then(new DoneCallback<ProblemInfoResponse>()
        {
            @Override
            public void onDone(ProblemInfoResponse result)
            {
                if (result != null)
                {
                    try
                    {
                        builder.updateFromServerStatus(result.getCbs());
                        ContentBuilder.getInstance().updateactivePS();
                    }
                    catch (BuilderException e)
                    {
                        Window.alert("Response was received but the content build status update
                                    failed.");
                    }
                }
                Window.alert("Response : " + result.getResponse());
            }
        })
}
```

Figure 36: JavaScript Promise Structure with Requestor Class and Container Object

The JavaScript promise structure created by GWT allowed our team to start code execution through the *onDone()* function. This function gets triggered once the back-end server acknowledges the request and successfully returns a *ProblemInfoResponse* container object back to the *call()* method. The returned *ProblemInfoResponse* is used to extract the *ContentBuildStatus* which includes relevant information for the front-end version of the *problemBuilder*. The *ContentBuildStatus* is then used to update the *problemBuilder* structure with the new information from the completed save. The UI is also alerted to these changes by calling the *updateactivePS()* method located within *ContentBuilder.java*. Finally, the true *needsToBeSaved* object is removed from the *problemMap* HashMap now that its *modifiedFields* have successfully been saved. In the case that the request is unsuccessful, the promise structure will trigger the *onFail()* function instead of *onDone()* and log any error information.

### 3.1.4 Testing New Functionality

After adding any functionality, we strived to robustly test the feature to ensure that our changes worked as intended and did not have unintended consequences. On the back-end, we were also able to write test cases to test any modifications our team made at a more granular

level. On the front-end, these tests involved running the system and manually testing all of the features.

For the back-end functionalities that *MakeAsyncCall.java* implemented, we wrote the test case *MakeAsyncCallTest*. This JUnit test case tests the full process of saving a problem while also conducting tests on filtering duplicate data. These tests save a problem contained within a problem set by calling the *call()* function within *MakeAsyncCall.java* and then checking the internal state of the instance. Since the internal state is stored within properties private of the class, we used the reflection technique to access them. The saving functionalities of *MakeAsyncCall.java* were also tested by triggering the event handlers associated with the UI elements that make up the problems displayed within the Builder 2.0 website.

The testing of the front-end involved manual testing instead of JUnit testing. These types of tests required our team to interact with UI elements and observe how the system responded. For example, to test the features of *onBlur()* and *onValueChange()* making RESTful API calls, our team had to boot up an instance of the Builder 2.0 website. Once it was loaded, we had to input values into the problems displayed in the UI so that the event handlers would trigger. To confirm the tests, we checked the server logs to make sure the appropriate RESTful API calls were executed. While this testing was a bit tedious, it was necessary to ensure that we did not create any bugs within the UI.

## 3.2 Saving Problems: Back-End

Upon starting this project, the server-side of the client-server interaction set up for Builder 2.0 was mostly a shell without functionality. Initially, there was only enough functionality to display a simple problem set upon connecting to the website as well as some skeletal functions for future functionality such as saving, verifying, and publishing problems and problem sets. In setting out to further develop a Minimum Viable Product, our team focused on completing the problem saving functionality as well as implementing additional methods for creating blank problem sets and loading already existing problem sets. This process was completed by updating and adding a few RESTful API endpoints that the front-end needed for making requests to the back-end.

Our team developed three endpoints throughout the project that were responsible for keeping data up to date in the ASSISTments database:

- 1) `‘/problemservice/save’`: Responsible for saving a problem to the database,
- 2) `‘/createproblemsetwithproblem’`: Responsible for creating a new problem set within the database and returning it to the front-end, and
- 3) `‘/tutoringservice/save’`: Responsible for saving tutorings to the database

Each of these endpoints have a function bound to them that will execute when the front-end searches for these specific routes. The sections below detail the work our team completed to develop each endpoint.

### 3.2.1 API Endpoint: /problemservice/save

The ‘/problemservice/save’ route is requested by the front-end any time one of the UI event handler functions triggers the saving of a problem. Upon execution of the function bound to the route, the endpoint will attempt to take the modified parts of the problem that are passed in from the front-end and save them to the correct problem in the database. Our team finished the implementation of this functionality since ‘/problemservice/save’ only contained parts of the required code when the project began. The functionality we added starts with the extraction of the data passed in from the front-end that needs to be saved into the database (*Figure 37*).

```
NeedsToBeSaved<ISimplifiedProblem, SimplifiedProblemFieldType> ntbs = payloadDTO.getNtbs();
ISimplifiedContent content = ntbs.getContentToSave();
ISimplifiedProblem problem = ntbs.getProblemToSave();
```

*Figure 37: Extracting Content from a needsToBeSaved Object*

If there are no issues with the extraction, the function will make type checks to ensure that the front-end request includes the information needed to perform a save. This is done by checking if the *needsToBeSaved* object can be cast between *ISimplifiedContent* and *ISimplifiedProblem* as well as if the *modifiedFields* contained in the *needsToBeSaved* object are an instance of the *SimplifiedProblemFieldType* enumeration.

With those checks in place, the code then attempts to save the content into the back-end database. The *prBuildMgr* variable defined at the top of the *BuilderService.java* is an implementation of *ProblemBuilderManagerImpl*. The *@Autowired* annotation tag is associated with *prBuildMgr* to facilitate dependency injection (*Figure 38*). Utilizing dependency injection here allows *BuilderService.java* to access the *ProblemBuildManagerImpl* class without inherently relying on *ProblemBuildManagerImpl* being inside of the project. This is important because it can save time if the *ProblemBuildManagerImpl* needs to be changed out for another class. Also, this manager handles database interactions by providing a method called *saveProblem()* that takes in a *needsToBeSaved* object’s *content* value and *modifiedFields* value.

```
@RestController
public class BuilderService
{
    private @Autowired ProblemBuildManagerImpl problemBuildManagerImpl;
    private @Autowired ProblemSetBuildManagerImpl psBuildMgr;
    private @Autowired ProblemBuildManagerImpl prBuildMgr;
    private @Autowired TutoringBuildManagerImpl tBuildMgr;
    private @Autowired SharedHelperContentKeysImpl contentKeyToCeri;
```

*Figure 38: Example of Dependency Injection for the ProblemBuildManagerImpl*

The *ProblemBuilderManager* is set up to return a response of type *ContentBuildStatus* when saving methods are called from it. The *ContentBuildStatus* is then used to determine if the database was able to save the inputted problem data correctly. The saving method was placed



within a try-catch-block to ensure that Builder 2.0 does not crash if an exception is thrown by *ContentBuildStatus*. If the *ContentBuildStatus* object does not cause an exception, then the ‘/problemservice/save’ functionality will finish execution by returning a *ProblemInfoResponse* object that contains the *ContentBuildStatus* to the front-end. The front-end then utilizes that *ContentBuildStatus* to validate that the save was completed successfully.

### 3.2.2 API Endpoint: /createproblemsetwithproblem

The ‘/createproblemsetwithproblem’ endpoint is requested by the front-end whenever a new instance of Builder 2.0 is loaded. The function bound to this route creates a new problem set within the database and inserts a blank problem into it. This new problem set and its internal problem are then sent to the front-end so that the content can be drawn into the UI. This endpoint did not exist before the start of the project and was added by our team. The endpoint accomplishes its task by first making use of the *ProblemSetBuilderProvider* and *ProblemBuilderProvider* factory classes ([Figure 39](#)). In this figure, the factory classes are used to create a linear problem set and a single choice problem.

```
ISimpleProblemSetBuilder psBuilder = ProblemSetBuilderProvider.getProblemSetBuilder(  
    VisibleProblemSetType.LINEAR_COMPLETE_ALL);  
  
IProblemBuilder<?> prBuilder = ProblemBuilderProvider.getProblemBuilder(  
    VisibleProblemType.CHOOSE_ONE);
```

*Figure 39: Using Factory Classes to Generate a Problem Set and Problem*

The endpoint function then creates a null *needsToBeSaved* object for both the problem and the problem set. It also creates a null *ContentBuildStatus* so that it can make use of the saving methods used by the *ProblemBuilderManagerImpl* and *ProblemSetBuildManagerImpl* which is a similar process to the saving method used by the ‘/problemservice/save’ route.

Once the objects are defined, the function attempts to populate them by utilizing getter, setter, and adder functions within a try-catch-block. The try-catch-block is used here so that any potential builder exceptions do not crash Builder 2.0. It was also important to control the order in which the overall problem set was being populated since building a full problem set needs to start off by building the innermost areas first. After setting up the correct order, the endpoint function sets the question text and problem name of the *ProblemBuilder* variable, *prBuilder*, using the built-in setter functions ([Figure 40](#)).

```
try  
{  
    prNTBS = prBuilder.getNeedsToBeSaved();  
    prBuilder.setQuestion("This is a question?");  
    prBuilder.setName("New Problem");
```

*Figure 40: Setting the Question and Name of the Problem Builder*



The *prBuilder* is then added to the end of the *ProblemSetBuilder* variable, *psBuilder*. Now that the *ProblemBuilder* has been populated with data and inserted into the *ProblemSetBuilder*, the problem can be saved to the database as being linked to this particular problem set. This saving is facilitated by making use of the *needsToBeSaved* object for the *ProblemBuilder*, as was done in the ‘/problemservice/save’ route. This function uses the same method in the *ProblemBuildManagerImpl* to persist the save. However, the difference with this route is that problem sets can potentially have more than one problem saved in it at a time. To account for this difference the function for this endpoint runs a for-loop over the list of problems contained within the problem set. It also makes a call to *updateFromServerStatus()* on the *ContentBuildStatus* from the saving method to notify observing classes that the save is taking place.

With the new problem successfully saved into the database, *ProblemSetBuilder* can be saved as well. The function executes this saving by following a similar process to the one described above for the problem. The key differences in this saving functionality are that the for-loop is removed since there is only one problem set, and that the *problemSetID* variable is extracted out from the *ContentBuildStatus* returned from the saving call. The endpoint function finishes execution by wrapping the newly created *problemSet* and *problemSetID* inside of a *SPSJsonContainer* (Figure 41). This container object is then returned to the front-end, and the new problem set and problem are drawn into the UI.

```
ISimplifiedProblemSet problemSet = psNTBS.getContentToSave();
SPSJsonContainer response = null;
if (problemSet != null)
{
    problemSet.setName(problemSetID + " Problem Set: made by URL: /createproblemsetwithproblem ");
    response = new SPSJsonContainer(problemSet, problemSetID);
}
else
{
    response = new SPSJsonContainer(problemSetID);
}
return response;
```

Figure 41: Wrapping *problemSet* and *problemSetID* into an *SPSJsonContainer*

### 3.2.3 API Endpoint: /tutoringservice/save

The ‘/tutoringservice/save’ route is requested by the front-end any time one of the UI event handler functions triggers the saving of a tutoring. This endpoint did not exist before the start of the project and was added by our team. The function bound to this endpoint is very similar to the function that executes the ‘/problemservice/save’ route, as there are only a few differences. The fields that a tutoring problem needs to save are almost identical to that of a regular problem. However, tutorings use different structures to process their extra functionalities. The endpoint function utilizes the same general process for making a save in the database as a

‘/problemservice/save’ does. First it sets up a *needsToBeSaved* object and declares a null *ContentBuildStatus* for the saving method to return into. It then executes the same type checks with the exception that *modifiedFieldList* now consists of fields from the *SimplifiedTutoringFieldType* enumeration instead of the *SimplifiedFieldType* enumeration that is used when saving a regular problem. The saving process for a tutoring problem then uses the save method from the *TutoringBuildManager* which requires the *needsToBeSaved* object’s content and *modifiedFields* as parameters. The endpoint function finishes execution by returning a *ProblemInfoResponse* object containing the *ContentBuildStatus* to the front-end. The front-end then utilizes that *ContentBuildStatus* to validate that the save was completed successfully.

### 3.3 Saving Problems: SDK Errors

Throughout our development of adding data persistence to Builder 2.0, we ran into a number of bugs. Some of which were simple fixes that we could implement, but others were rooted inside of the underlying SDK that Builder 2.0 is built off of. SDK 3.0’s development was handled by a team of engineers at ASSISTments separate from our project. Upon finding these errors, we would write test cases to ensure that the errors were indeed being caused by the SDK, and then give our findings to the ASSISTments Team. The sections below detail the major SDK bugs that were identified by our team and passed onto the SDK 3.0 developers at ASSISTments.

#### 3.3.1 V1 Certificate Error

Our first roadblock during development was a bug dealing with improper certificate versionings. When attempting to save any type of problem, the SDK would return an error detailing that the problem version was “not a V1 CERI” and thus could not be saved ([Figure 42](#)).

```
org.assistments.domain.exception.NotFoundException: External Reference: Not a V1 CERI: "WPRBQF"  
at org.assistments.service.manager.content.migration.impl.MigrationHelperCommonImpl.getLegacyCeriInfo(MigrationHelperCommonImpl.java:129)  
at org.assistments.service.manager.content.migration.impl.MigrationHelperCommonImpl.getCeriInfo(MigrationHelperCommonImpl.java:213)  
at org.assistments.contentbuilderservice.BuilderService.fetchProblemFromCeri(BuilderService.java:380)
```

*Figure 42: V1 Certificate Error*

A “V1 CERI” refers to a version one certificate for identifying problems in the database. The database that held all of Builder 1.0’s data is separate from Builder 2.0. With Builder 2.0, a new database was created to hold the newer problems that were marked with V2 certificates rather than V1 certificates. However, whenever the SDK interacted with the databases, if it could not find a matching certificate inside of the V2 database it would fall back to the V1 database. This in turn would cause the error as the V1 database expected to look for a “V1 CERI” and not a “V2 CERI”. The SDK developers took these findings and implemented a fix for the certificate versioning error.

Close to the end of the project, our team discovered another instance of the V1 certification error. As of the date of publishing this paper, the new instance of this error still exists. For more information, refer to [Section 3.5.2](#).

### 3.3.2 DOA Null Parameter Error

Once the SDK team fixed the first instance of the V1 certificate bug, the saving service API calls were now properly attempting to manipulate ASSISTments 2.0 data. Our team went forward with testing our data persistence features and stumbled upon a new bug that again blocked our progress. This time, the saving service request was failing due to an issue with sending data to the ASSISTments database. The error thrown in the developer console revealed that the DOA, Direct Oracle Access, variable had a null parameter dealing with the *VersionRow* of a problem (*Figure 43*).

```
java.lang.NullPointerException
    at org.assistments.service.manager.content.builders.impl.ProblemBuildManagerHelper.newProblemVersionRow(ProblemBuildManagerHelper.java:410)
    at org.assistments.service.manager.content.builders.impl.ProblemBuildManagerHelper.updateProblem(ProblemBuildManagerHelper.java:260)
    at org.assistments.service.manager.content.builders.impl.ProblemBuildManagerHelper.persistSimplifiedProblem(ProblemBuildManagerHelper.java:110)
    at org.assistments.service.manager.content.builders.impl.ProblemBuildManagerImpl.saveProblem_aroundBody0(ProblemBuildManagerImpl.java:160)
    at org.assistments.service.manager.content.builders.impl.ProblemBuildManagerImpl$AjcClosure1.run(ProblemBuildManagerImpl.java:1)
    at org.aspectj.runtime.reflect.JoinPointImpl.proceed(JoinPointImpl.java:167)
    at org.assistments.service.security.authorization.aspect.ServiceAuthorizationAspect.hasOneOfTheseAuthorities(ServiceAuthorizationAspect.java:89)
    at org.assistments.service.security.authorization.aspect.ServiceAuthorizationAspect.interceptionAdvice(ServiceAuthorizationAspect.java:194)
    at org.assistments.service.manager.content.builders.impl.ProblemBuildManagerImpl.saveProblem(ProblemBuildManagerImpl.java:128)
    at org.assistments.contentbuilderservice.BuilderService.problemSavePostRequest(BuilderService.java:691)
```

*Figure 43: DOA Null Parameter Error*

The stack trace of this error revealed that manipulations with the DOA were handled inside of the SDK. As such, the bug was handed off to the SDK 3.0 team. Their investigation revealed that the bug dealt with a data publication issue and a fix was implemented.

### 3.3.3 Owner\_ID Null Error

Following the implemented fix for the null DOA parameter, our team revealed another bug dealing with the database interactions needed for data persistence. In this case, the *owner\_id* field associated with problems stored in the ASSISTments database was null. *Figure 44* details the database error that was thrown.

```
Caused by: org.postgresql.util.PSQLException: ERROR: null value in column "owner_id" violates not-null constraint
Detail: Failing row contains (232, null, 2, 1, New Problem, <p>This is a test question?sqwfwqf54y45yu54y</p>, , 2020-1
    at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl.java:2553)
    at org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.java:2285)
    at org.postgresql.core.v3.QueryExecutorImpl.execute(QueryExecutorImpl.java:323)
    at org.postgresql.jdbc.PgStatement.executeInternal(PgStatement.java:473)
    at org.postgresql.jdbc.PgStatement.execute(PgStatement.java:393)
    at org.postgresql.jdbc.PgPreparedStatement.executeWithFlags(PgPreparedStatement.java:164)
    at org.postgresql.jdbc.PgPreparedStatement.executeUpdate(PgPreparedStatement.java:130)
    at org.apache.tomcat.dbcp.dbcp2.DelegatingPreparedStatement.executeUpdate(DelegatingPreparedStatement.java:136)
    at org.apache.tomcat.dbcp.dbcp2.DelegatingPreparedStatement.executeUpdate(DelegatingPreparedStatement.java:136)
    at org.springframework.jdbc.core.JdbcTemplate.lambda$update$0(JdbcTemplate.java:867)
    at org.springframework.jdbc.core.JdbcTemplate.execute(JdbcTemplate.java:617)
    ... 53 more
```

*Figure 44: Owner\_ID Null Error*

The *owner\_id* field is meant to specify which ASSISTments user created the current problem. Since Builder 2.0 was in early development, the user account system had not been integrated as of yet. Therefore, the *owner\_id* field in our development environment would always be null. While other engineers at ASSISTments worked on fixing this issue, our team was able to continue development by hard-coding a “dummy value” of ‘2’ for every problem’s

`owner_id` field. We implemented the temporary work around by utilizing the function `setOwnerDbid()` inside of the RESTful API endpoint `‘/problemservice/save’` (*Figure 45*).

```
@PostMapping(value = "/problemservice/save")
public ProblemInfoResponse problemSavePostRequest(@RequestBody NeedsToBeSavedPRContainer
    payloadDTO)
{
    NeedsToBeSaved<ISimplifiedProblem, SimplifiedProblemFieldType> ntbs = payloadDTO.getNtbs();

    ntbs.getContentToSave().setOwnerDbid(2);
}
```

*Figure 45: Manually Setting the Owner\_ID Field*

### 3.3.4 Answer SQL Calls Null

After our team implemented the temporary work around to solve the `owner_id` field being null, the RESTful API calls for saving problems finally began to work. We were now able to update a problem’s question text and successfully save that modification. Upon reloading the Builder 2.0 site with that specific problem set, the problem would display with the new additions. Therefore, data persistence for problems had been implemented. However, we quickly realized that there was still one more error blocking full completion of this task. While saving the question text worked without flaw, any attempt to add, delete or modify answers would result in the saving RESTful API call to fail. This failure was due to an incorrect fetch to the SQL database that stores answers (*Figure 46*). Anytime Builder 2.0 attempted to retrieve or save data to the answers section of the SQL database, the result came back as null because the SQL database could not find any reference to the answers, as if their storage did not exist. This error was brought to the SDK team and was still being worked on by the end of this project. Once this error is fixed, all problem data should save correctly and data persistence for problems will function as intended.

```
null
org.assistments.domain.exception.BuilderException:
org.assistments.domain.exception.NotFoundException: No query results: core.answers : SELECT
* FROM core.answers WHERE (answer_set_id = 20) ORDER BY position ASC
```

*Figure 46: Answer SQL Calls Null Error*

### 3.3.5 Error Location Validation Process

The issues found above were all caused by some underlying error within some function located inside of the ASSISTments SDK. The issues often occurred when attempting to save problem data since this functionality requires modifying the database where the problem data is stored. The scope of this project focused solely on the Content Builder repository within the greater ASSISTments software library, so making modifications to the SDK in order to fix these errors was not in our jurisdiction. However, before handing over any issue we encountered, we created and ran tests that verified these issues were indeed stemming from the SDK.

Our team adopted a process for determining whether an issue was caused by our repository's source code or the SDK through the use of JUnit testing. The process started off by using the debugger and writing print-out statements to identify which lines of code inside of Builder 2.0's code-base were throwing the errors. A new testing class for that specific part of the code was then created within the test folder of the directory that the issue was located in. Within these testing classes, JUnit test cases were created to simulate the intended functionality our team was trying to implement. If these test cases failed due to the same errors that arose during development, then the simulation was working properly. The simulated test was then copied into the testing directory on the SDK repository to see if the error persisted there. If the tests ran and failed due to the same error, then it was confirmed that the root cause of the issues were located within the SDK. Our team would then report the issues to Ashish who would notify the developers working on the SDK of the bugs. Conversely, if the test cases passed without error within the SDK repository, then the SDK was not the cause of the issues. In these cases, our team had to review the code we wrote during development to track down the bugs we created. All instances of bugs created by our team were fixed without issue.

## 3.4 UI Decisions and Changes

To prepare for the release of the Minimal Viable Product, our team looked at possible ways to improve the UI. When we started our project, Builder 2.0 had three UI layouts that were very similar in functionality. We discovered a way to simplify and condense these layouts to make a more fluid experience. Builder 2.0 also had an additional way to visualize a problem set, known as the 'Tree View', which we decided to remove due to it being both complex and redundant. In addition, throughout our development of Builder 2.0, we discovered many bugs in the drag-and-drop feature. We decided to replace drag-and-drop with buttons to ensure a less complicated user experience. Lastly, the Builder 2.0 UI was relying on an non-customizable HTML library for displaying popup windows. So, our team implemented a custom popup window functionality that allows Builder 2.0 to display alerts boxes, confirmation boxes and user input boxes that follow the layout and color scheme of the rest of the site.

### 3.4.1 Condensing the UI Layout

At the start of this project, Builder 2.0 had three different UI layouts: Minimalist, Default, and Advanced. In order to switch between the three, users had to open the preferences window and manually choose which UI layout they wanted to use ([Figure 47](#)). There was also code that would restrict what UIs a user could use with the use of a *userLevel* variable.

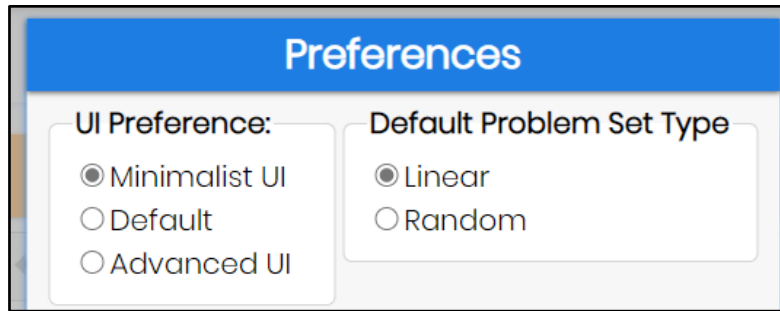


Figure 47: Preferences Popup Window with the Three Original UI Options

The Minimalist UI only allowed users to add basic problems and multi-part problems with their respective buttons. With this UI layout, users were unable to choose the answer type of a problem until a new basic problem was added into the UI. In addition, this UI did not allow users to add additional problem sets into the current problem set. Figure 48 details the Minimalist UI.

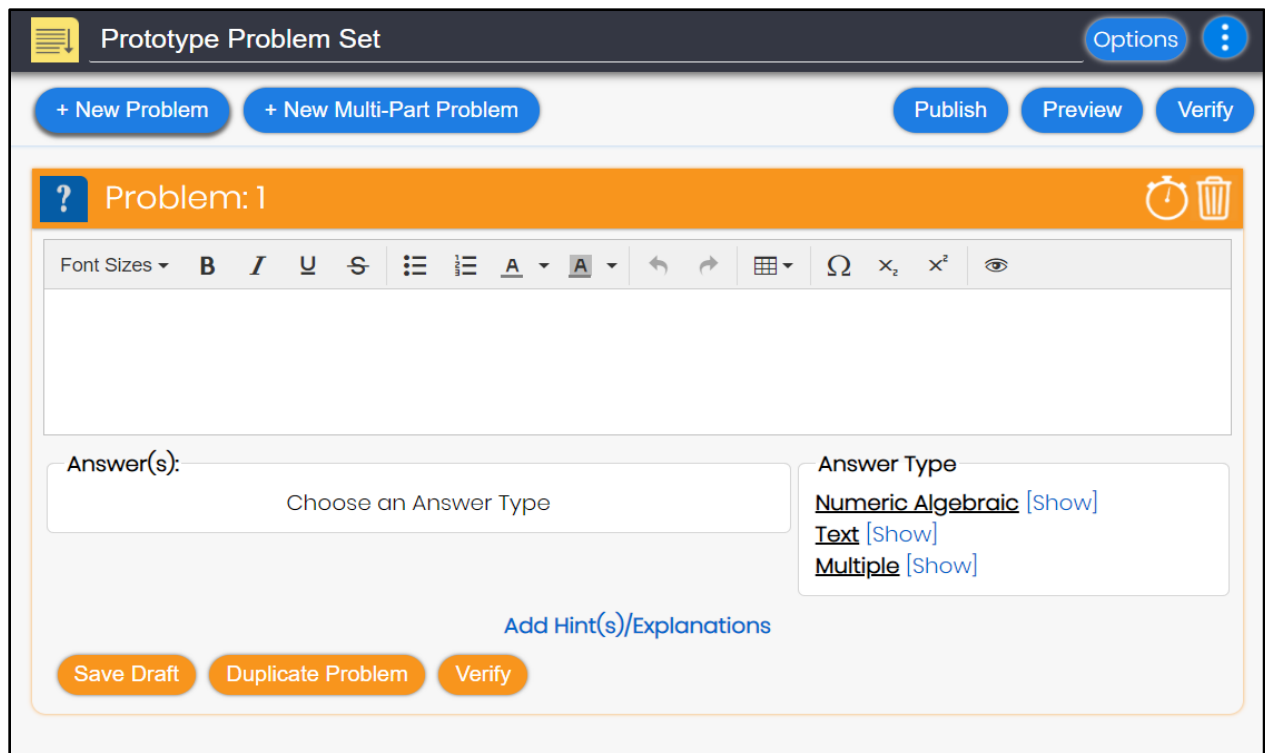
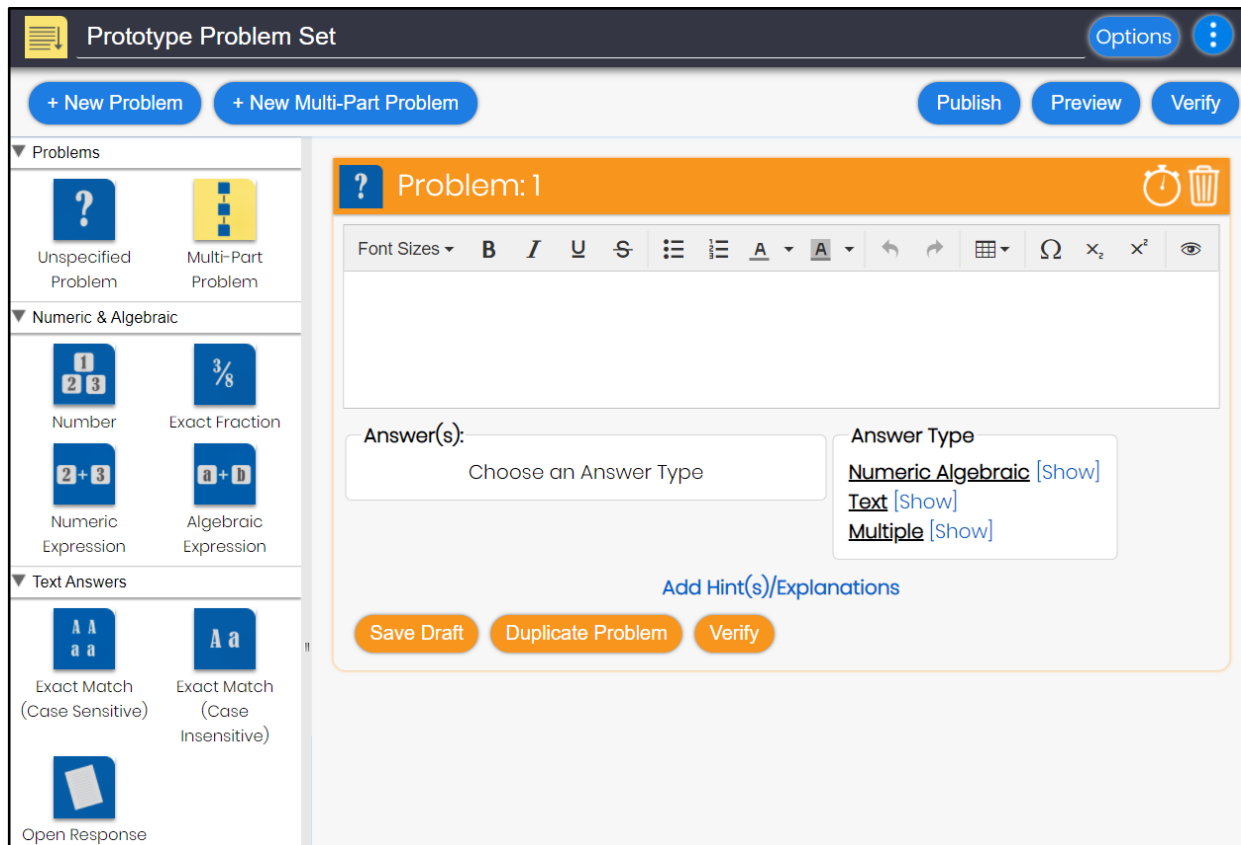


Figure 48: Builder 2.0's Minimalist UI

The Default UI built off of the Minimalist UI by adding an additional menu on the left hand side of the screen. This menu is known as the palette and contains draggable icons that represent all of the problem types a user can choose from. By either double clicking or dragging these icons, users can add problems of that specific type to the current problem set. This variation of the UI still did not allow users to add additional problem sets into the current problem set. Figure 49 details the Default UI.



*Figure 49: Builder 2.0's Default UI*

Finally, the Advanced UI built off of the Default UI in two major ways. First, the palette on the left hand side now included icons for problem sets. This meant that users would now be able to add additional problem sets into the current problem set. The inclusion of these icons also gave users access to the unique problem set types of 'Skill Builder' and 'If-Then-Else'. Secondly, the Advanced UI included a right hand side menu that displays a concise and easily readable view of the hierarchy for all the problems and problem sets inside of the users' current builder project. This menu is known as the explorer. [Figure 50](#) details the Advanced UI.



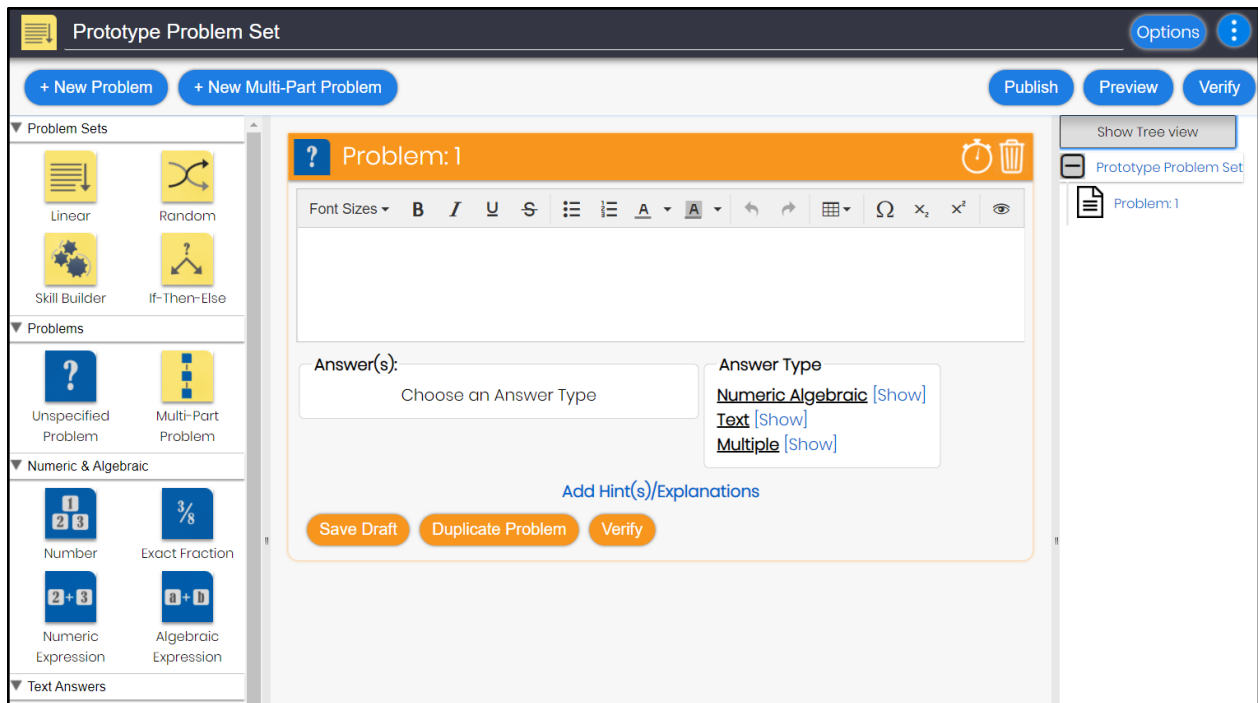


Figure 50: Builder 2.0's Advanced UI

The only difference between these three UI layouts were the two side menus that provided more functionality to the user. Our team believed that users should not need to switch between three different UIs to access the ability to choose specific problem types and utilize additional problem sets. So, we decided to condense the UI into one singular layout that gives users access to all features. To ensure that users still have a choice for what functionality displays on their screen, we made the two side menus collapsible and resizable. With this improved layout, users will have a much easier way to switch between the functionalities they want to use, and they can also control the amount of space they have to view their problems and problem sets.

To implement this change we first had to make sure that both the palette and the explorer side menus were always enabled. Since the UI is initialized and displayed within the `onModuleLoad()` function located in `ContentBuilder.java`, that is where we started. In this function we found the method that was responsible for hiding both the palette and the explorer from the user, `setWidgetHidden()`. We modified these method calls by replacing a true Boolean with a false Boolean to ensure that the two side menus would be visible whenever a user navigated to the Builder 2.0 site (Figure 51).

```
innerPanel.setWidgetHidden(palette, false);
innerPanel.setWidgetHidden(topExplorer, false);
```

Figure 51: Making the Palette and Explorer Visible to the User



Now that the two side menus were present in the UI at all times, we had to remove the functionality that would disable the side menus. The disabling functionality was controlled by the `RadioButtons` for choosing which UI layout to view ([Figure 47](#)) and the `userLevel` variable within the `PreferencesView`. We began by stripping out the three `RadioButtons` that switched between the Minimalist UI, the Default UI and the Advanced UI. This required removing the elements from the associated XML UI layout file, removing their declarations within the associated Java controller class, and removing their `onClick()` event handlers. These event handlers would set the `userLevel` to either “novice”, “beginner” or “advanced”. Once a user saved their new preferences, the `updatePreferences()` function would be called and the new `userLevel` would be passed in. Within `updatePreferences()`, there were conditional statements for each `userLevel` value ([Figure 52](#)). In these conditions, the palette and explorer would be either enabled or disabled based on the `userLevel`.

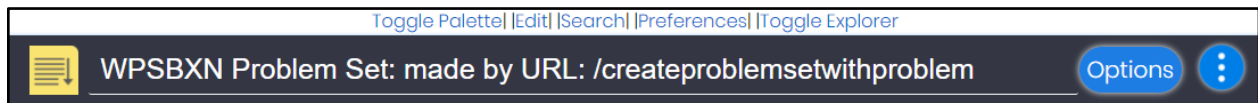
```
if (userLevel.equals("novice"))
{
    ...
    innerPanel.setWidgetHidden(topExplorer, true);
    innerPanel.setWidgetHidden(palette, true);
    ...
}
else if (userLevel.equals("beginner"))
{
    ...
    innerPanel.setWidgetHidden(topExplorer, true);
    innerPanel.setWidgetHidden(palette, false);
    ...
}
else
{
    ...
    innerPanel.setWidgetHidden(topExplorer, false);
    innerPanel.setWidgetHidden(palette, false);
    ...
}
```

*Figure 52: Conditional Statements for using userLevel to Switch UI Layouts*

By removing these conditions, we had successfully removed all instances of disabling the two menus that made up the three original UI layouts. After conversing with Ashish, our team decided to leave the rest of the `userLevel` functionality as it may be used in the future to restrict more advanced features and beta features.

Now that the UI was condensed and the three UI layout options were removed, we had to ensure that the side menus could be easily manipulated by the user. The resizing component of each menu was automatically taken care of by GWT. Since the menus are a part of `innerPanel`, which is a GWT `SplitLayoutPanel`, they are automatically given drag bars that allow the user to resize them. Our team still wanted to add more flexibility in how users could manipulate these

menus. To do so, we added toggle buttons that open and close the palette and explorer which gives the user a fast and simple way to control what is displayed on their screen. Since the decisions about where to add these new buttons would be decided by the ASSISTments UI Team, we added placeholder buttons to *HeaderOptionsView*. This view is currently used to hold placeholder UI elements that will be better incorporated by the UI Team at a later date. As seen in [Figure 53](#), the placeholder buttons we added are labeled as “Toggle Palette” and “Toggle Explorer”.



*Figure 53: Placeholder Toggle Buttons for the Palette and Explorer*

Clicking these buttons will trigger two new functions that we added into *ContentBuilder.java*: *togglePalette()* and *toggleExplorer()*. As seen in [Figure 54](#), these functions close the menus by setting their size to ‘0’ and open the menus by setting their size to ‘250’.

```
public void togglePalette()
{
    if (innerPanel.getWidgetSize(palette) > 0)
    {
        innerPanel.setWidgetSize(palette, 0);
    }
    else
    {
        innerPanel.setWidgetSize(palette, 250);
    }
}

public void toggleExplorer()
{
    if (innerPanel.getWidgetSize(topExplorer) > 0)
    {
        innerPanel.setWidgetSize(topExplorer, 0);
    }
    else
    {
        innerPanel.setWidgetSize(topExplorer, 250);
    }
}
```

*Figure 54: The togglePalette() and toggleExplorer() Functions*

With the toggle button functionality fully implemented, our task for condensing the three UI layouts into one fluid layout was complete. Figures [Figure 55](#), [Figure 56](#), [Figure 57](#) and [Figure 58](#) show the new UI layout with all variations of the menus being opened and closed.

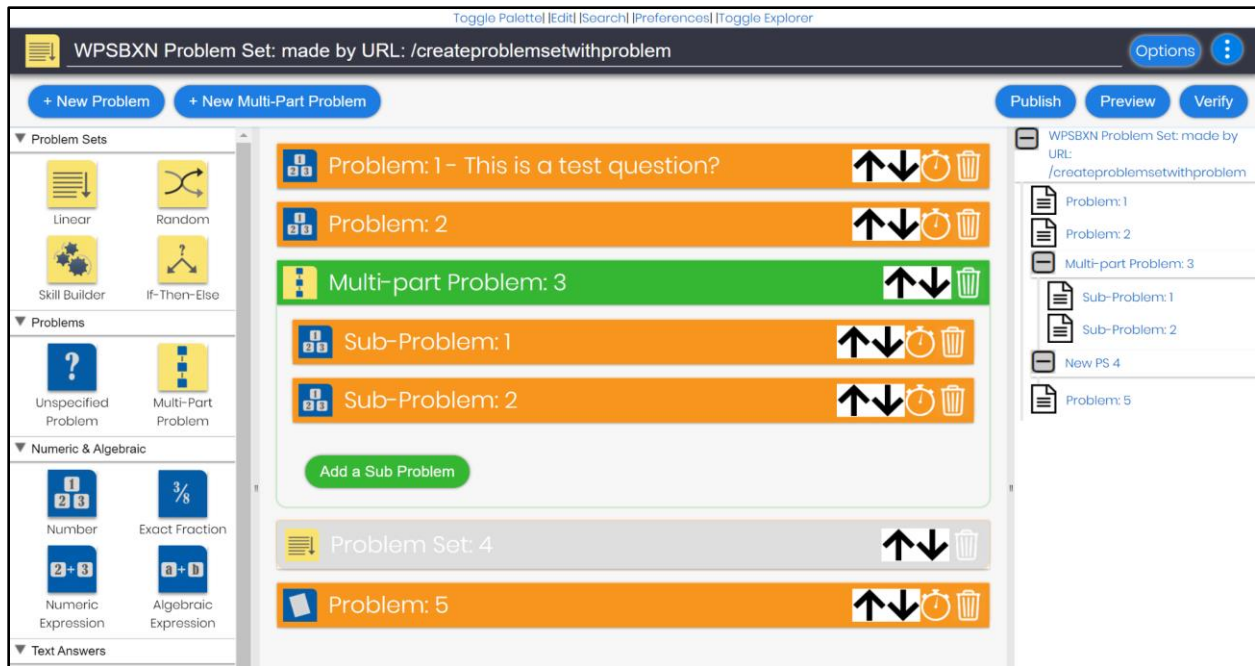


Figure 55: The Condensed UI with Both the Palette and Explorer Open

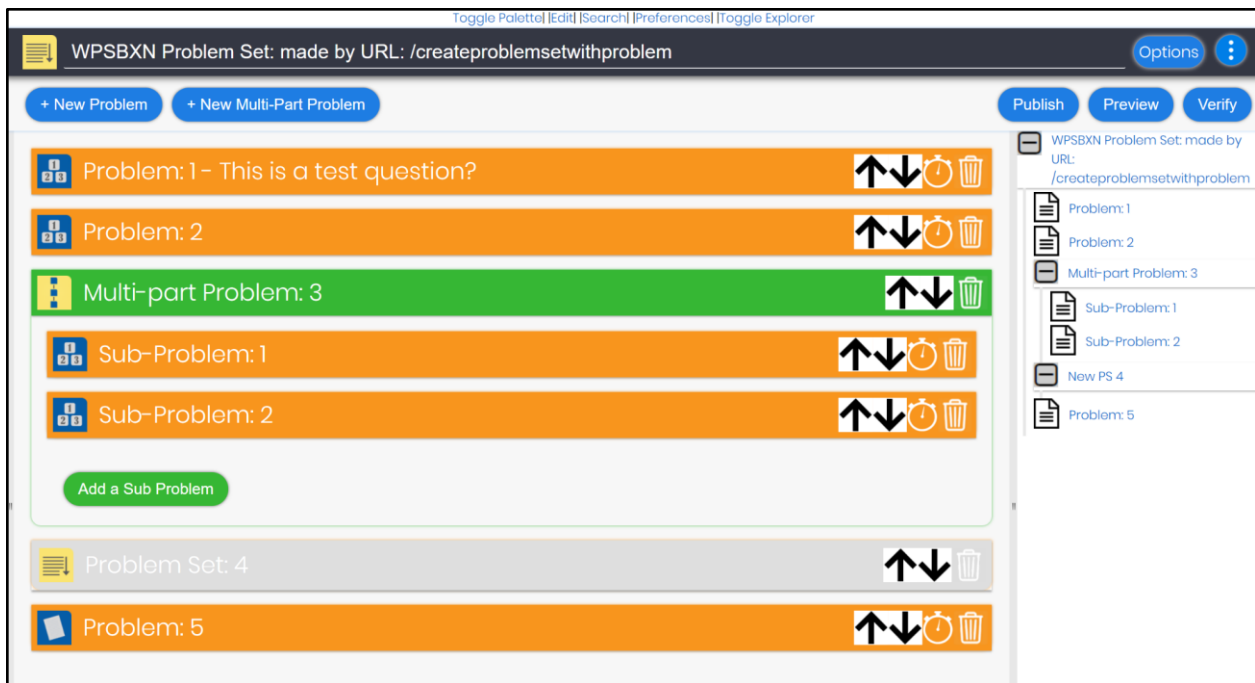


Figure 56: The Condensed UI with the Palette Closed and the Explorer Open

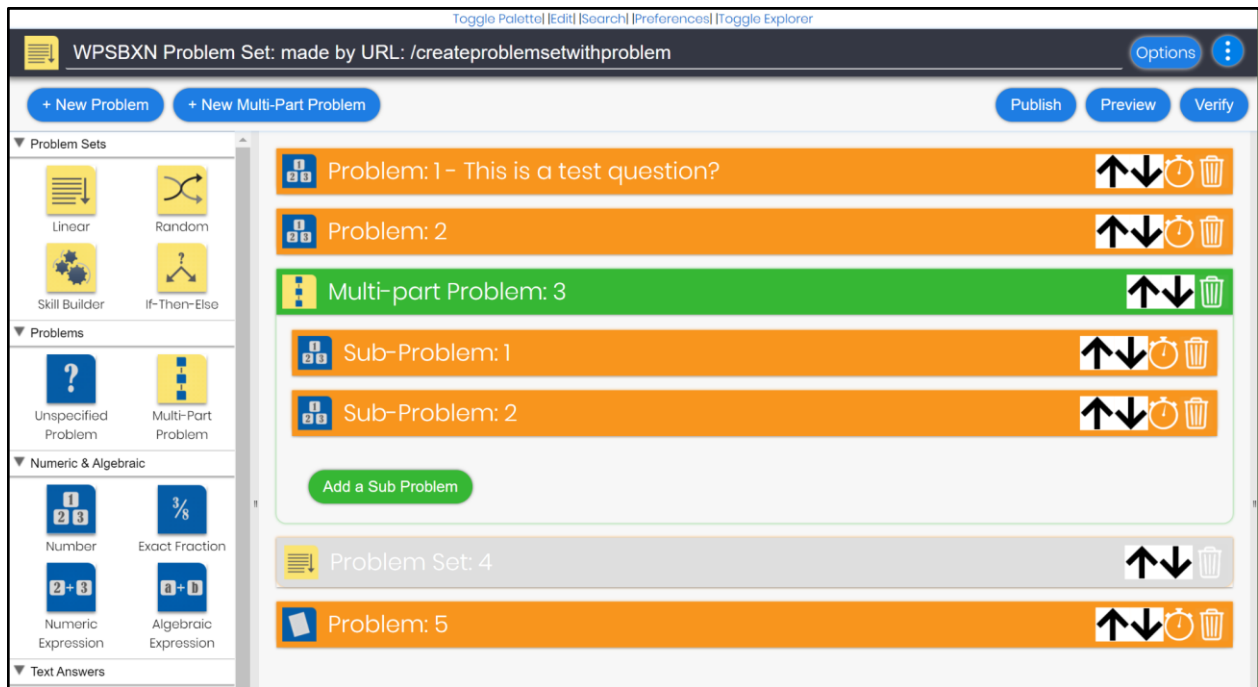


Figure 57: The Condensed UI with the Palette Open and the Explorer Closed

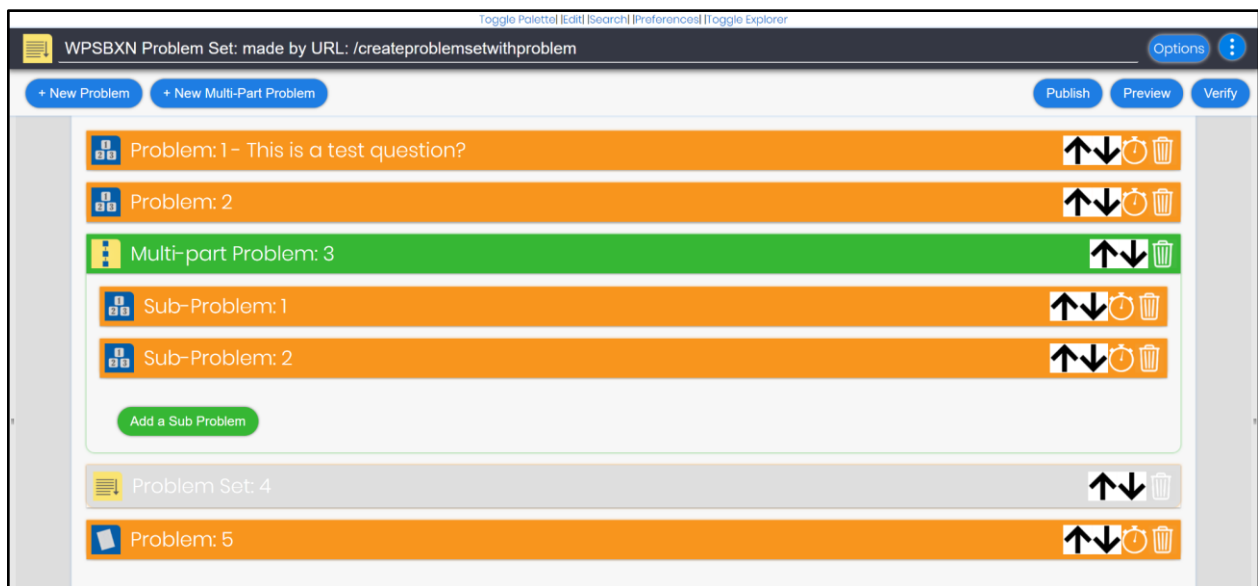
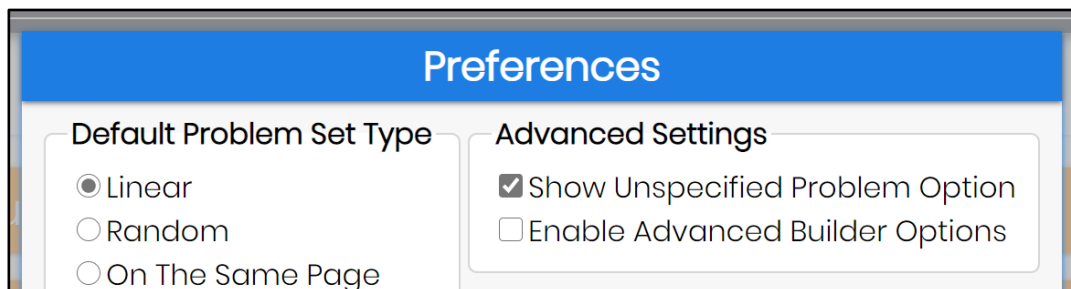


Figure 58: The Condensed UI with Both the Palette and Explorer Closed

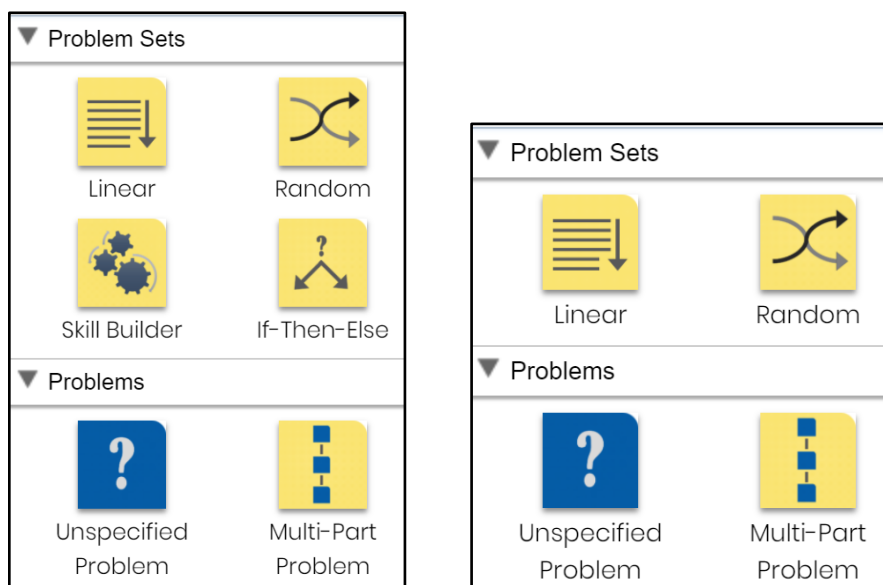
After completion of the new UI layout, it came to our team's attention that ASSISTments wanted some of the more advanced builder options to be "opt-in" by the user. Problem set types such as 'Skill Builder' and 'If-Then-Else' are more complicated than the rest and are not going to be used by the average user. To adhere to these requirements, our team added an Advanced Settings section to the preferences popup window. As seen in [Figure 59](#), this section includes a CheckBox for enabling 'Advanced Builder Options'. Our team also moved the 'Show

Unspecified Problem Option’ CheckBox into this section since it was originally not included inside of any section, which made it look out of place.



*Figure 59: Advanced Settings Options in the Preferences Popup Window*

If a user decides to “opt-in” and check ‘Enable Advanced Builder Options’, the ‘Skill Builder’ and ‘If-Then-Else’ problem set types will appear within the palette containing all of the problem and problem set icons a user can choose from. If the user does not have this option checked, these advanced problem types will not be available to them (*Figure 60*).



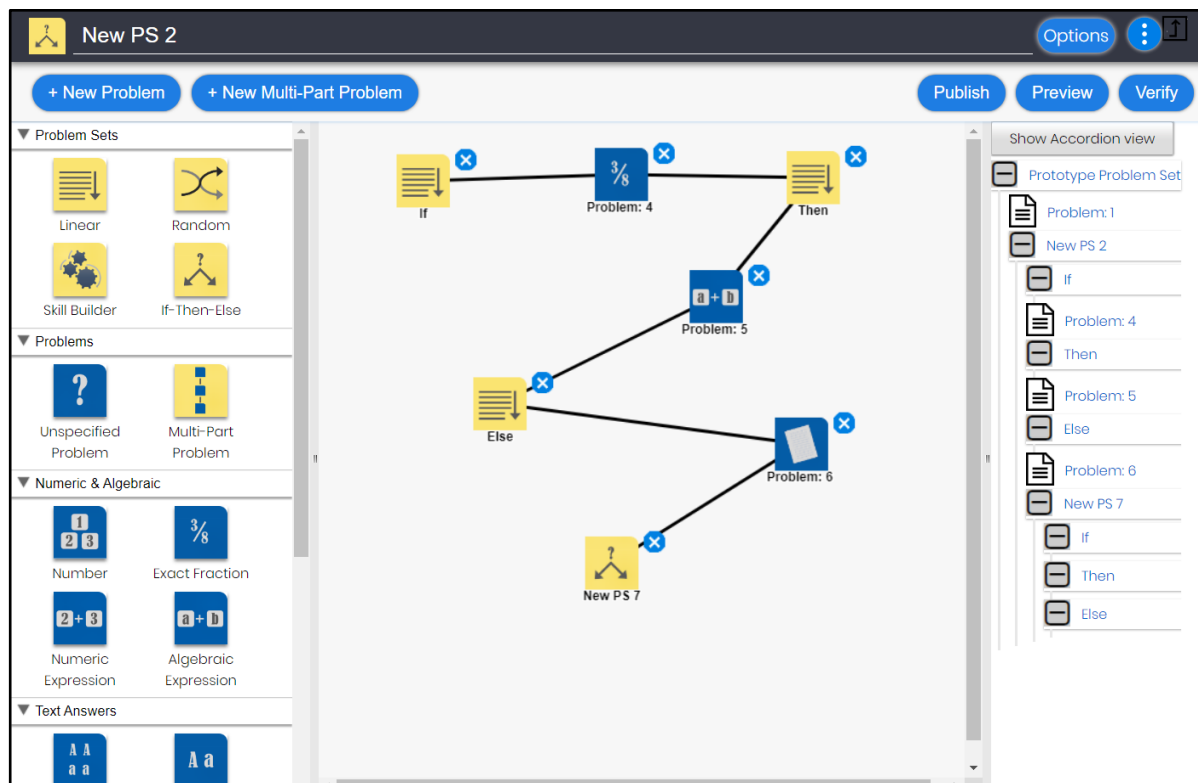
*Figure 60: Advanced Builder Options Enabled (Left) and Disabled (Right)*

This feature of enabling specific problem and problem set types will also be useful in the future if the ASSISTments Team decides to revisit utilizing *userLevel*. With the configurations our team implemented, future developers will be able to easily disable and enable problem and problem set types based on the authority a user is given.

### 3.4.2 Removing the Tree View

In addition to the ‘Accordion View’ that displayed all the content within a problem set, there was also the ‘Tree View’. This view attempted to visualize all of the content within a problem set in the form of a “tree” by connecting all of the problem and problem set icons with

black lines (*Figure 61*). When the user double clicked on an icon, the ‘Accordion View’ would open up to that problem or problem set. We identified several issues with this view that lead to our decision to remove it.



*Figure 61: Builder 2.0's Tree View*

One problem with the ‘Tree View’ was that the icons would be placed on top of each other. This meant that most of the icons were hidden and the user would have to manually drag each one off of each other before they could see all the icons. Without manually organizing this view, the user would be unable to make any sense of the structure of a problem set. An additional flaw with the ‘Tree View’ was that it was not really a “tree”. There was no visualization for how content can “branch” off from each other via layered problem sets and ‘If’ ‘Then’ ‘Else’ blocks. This ‘Tree View’ was more of a connect the dots type graph than anything else. Apart from the buggy and unhelpful visualization that this view provided, the ‘Tree View’ did not provide any new functionality that was not present elsewhere within Builder 2.0. With this analysis, our team decided to remove the ‘Tree View’ since it was not intuitive to use, it did not provide any new functionality, and it would have needed a lot of additional development before it could be considered user friendly.

Removing the ‘Tree View’ required two major changes to the code. Originally, both the ‘Accordion View’ and ‘Tree View’ were loaded into a ViewPane based on settings chosen by the user. Therefore, we needed to remove the code that created the ‘Tree View’ as well as the code that would allow the user to switch between the two views. Our team found and either modified

or removed the three functions located in *ContentBuilder.java* that took care of the above functionality:

- 1) *updatePreferences()*: Controlled the switching between views,
- 2) *openTreeView()*: Called by functions that wanted to display the ‘Tree View’, and
- 3) *drawTreeCanvas()*: Controlled how the ‘Tree View’ was displayed on the front-end.

The *updatePreferences()* function makes changes to the UI based on any options that the user chooses inside of the preferences popup window. It also used to rely on *userLevel* to determine what could be displayed in the UI. For example, the values for *userLevel* were controlling which of the three UI layouts a user started off with. Therefore, *userLevel* was also responsible for controlling if a user could access the ‘Tree View’ since the ‘Tree View’ was only available in the Advanced UI. As described in [Section 3.4.1](#) the UI was condensed and the *userLevel* implementations were removed along with the code it used to enable the ‘Tree View’. The *updatePreferences()* function also kept track of a *currentView* variable that was set to a passed in *ViewPane*. Since that *ViewPane* can now only be an ‘Accordion View’, the parameter was removed from the signature of the function. There was also functionality for opening and drawing the different views onto the front-end canvas, which was removed as well ([Figure 62](#)).

```
if (currentView.equals("viewtree"))
{
    openTreeView(currentPS);
}
else
{
    openProblemSet(currentPS);
}
```

Figure 62: Conditional Statement for Drawing Either the Tree or Accordion View

The *openTreeView()* function was called whenever the view was switched to the ‘Tree View’, such as in *updatePreferences()*. The function *openTreeView()* was used to set up the front-end to display the ‘Tree View’ by clearing the *centerPanel* and adding a *treeCanvas* to the *treePanel*. The *openTreeView()* function then called the *drawTreeCanvas()* function which handled drawing shapes and lines to visually create the ‘Tree View’ and display it to the user. This function was also called whenever a change was made to the tree structure so that the ‘Tree View’ was redrawn to show those changes in the UI. Since *drawTreeCanvas()* controls functionality that directly affects the front-end behavior of Builder 2.0, it was decided to leave this function inside of the code-base in case it is ever needed after the completion of the Minimum Viable Product. However, all calls to this function were removed and the *openTreeView()* function was removed completely.

### 3.4.3 Removing Drag-and-Drop

Another UI decision made by our team was the removal of the drag-and-drop functionality for reordering the content inside of a problem set. After some testing and



discussion, we decided that the drag-and-drop functionality that was previously incorporated into the ‘Accordion View’ had some issues that would cause it to work erroneously. For example, if a problem set only had one problem inside of it, attempting to drag that problem would cause the UI to stall and prevent the user from doing anything else. In addition, dragged problems would sometimes lose their title. Taking the scope of the project into consideration and our conversations with Ashish, we decided that drag-and-drop was not a core functionality needed for the Minimum Viable Product. That conversation also illustrated that the end users, being the teachers who are making problem sets for their classes, may not find drag-and-drop to be very intuitive and could lead to confusion.

The actual removal of drag-and-drop for reordering content in a problem set was not done by simply removing all code that dealt with dragging objects. The front-end of Builder 2.0 also utilizes drag-and-drop for adding specific problem and problem set types into the UI. In addition, the front-end relies on a data structure called *dragData* to determine which problem and problem set types need to be drawn into the UI. If our team simply removed all drag-and-drop code, these functionalities would break, and the UI for Builder 2.0 would be unusable. So, our team had to track down and remove the code that was only specific to reordering content with drag-and-drop. This code was found within *ContentBuilder.java* as it controls a majority of the UI interactions. More specifically, the *AccordionContent* instances that represent both problems and problem sets were being manipulated inside of this class. These instances of *AccordionContent* contain a *DisclosurePanel*, which is a GWT widget that allows other elements within it to be collapsed and un-collapsed when a user clicks on its header. These *DisclosurePanel* headers are the orange, green or gray title bars that are present on each type of *AccordionContent* (Figure 63). Since these headers were the draggable part of problems and problem sets, we had found the proper GWT element to remove drag-and-drop code from.

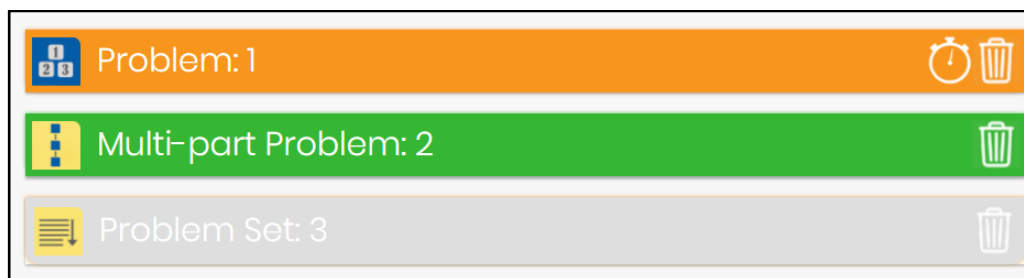


Figure 63: The Various Headers for Problems and Problem Sets

The *DisclosurePanels* for each problem and problem set were being manipulated through a function called *addReorderingHandler()*. Inside of this method, every *DisclosurePannel* was having the two built-in GWT event handlers specific to manipulating drag-and-drop functionality added to them: *onDragStart()* and *onDragEnd()*. Our team removed all of the functionality inside of these event handlers so that dragging the headers of problems and problem sets would no longer reorder any of the content. Even though the reordering drag-and-drop functionality was removed, our team realized that users could still pick up and drag problems and problem sets all

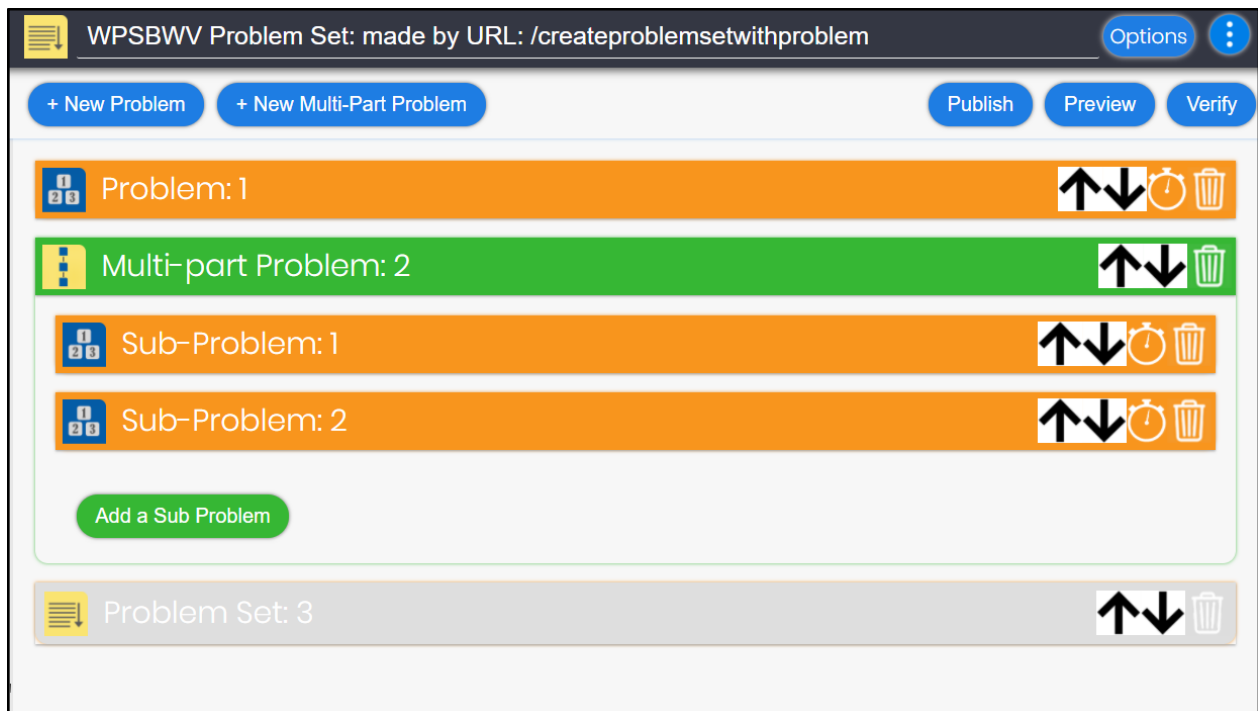


over the screen. To remove this unnecessary action, we had to disable the default dragging functionality that was causing *AccordionContent* to still be picked up by the user. This was done by adding a new *onDragStart()* event to the constructor of *AccordionContent*. As seen in [Figure 64](#), this new event handler includes the line *e.preventDefault()*. Any use of this line of code ensures that the default GWT functionality for a specific UI event will not be executed. In this case, it disables the functionality of picking up problems and problem sets when a user attempts to drag them by holding down the right mouse button on *AccordionContent* headers.

```
this.addDomHandler(new DragStartHandler()
{
    public void onDragStart(DragStartEvent e)
    {
        e.preventDefault();
    }
}, DragStartEvent.getType());
```

*Figure 64: The onDragStart() Event Handler that Disables Dragging AccordionContent*

Although the drag-and-drop functionality was removed from *AccordionContent*, its core functionality of reordering problems and problem sets within a parent problem set still needed to be available to the users. As a replacement, our team added ‘Up Arrow’ and ‘Down Arrow’ buttons to the headers of problems and problem sets ([Figure 65](#)).



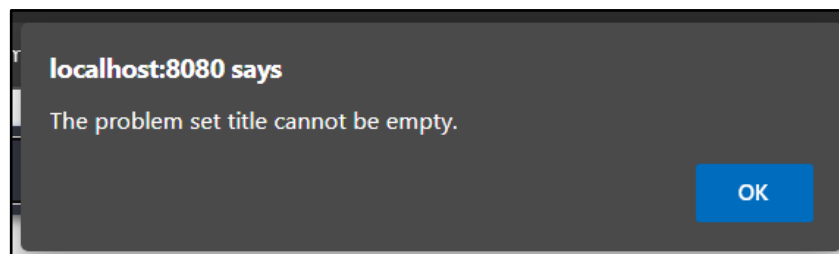
*Figure 65: New Up Arrow and Down Arrow Buttons for Content Reordering*

These buttons were implemented by placing their respective .png image files into the XML UI layout files associated with problems, multi-part problems and problem sets. Within their

respective controller classes, these buttons have an *onClick()* function assigned to them that causes them to go into the *contentList*, which is an *ArrayList* that contains the *ICBContent* for every problem, multi-part problem and problem set contained within the parent problem set. This *contentList* is used to draw the contents of a problem set into the ‘Accordion View’. The *onClick()* function will evaluate which direction it will be attempting to move the current problem in *contentList* and then proceed to make the swap operation. After the swap operation is completed, the drawing of the UI is refreshed and the two *AccoridonContents* will appear as swapped in the Builder 2.0 UI. On a final note, the current up and down arrow images are placeholders that will be replaced with custom arrow images by the ASSISTments UI Team at a later date.

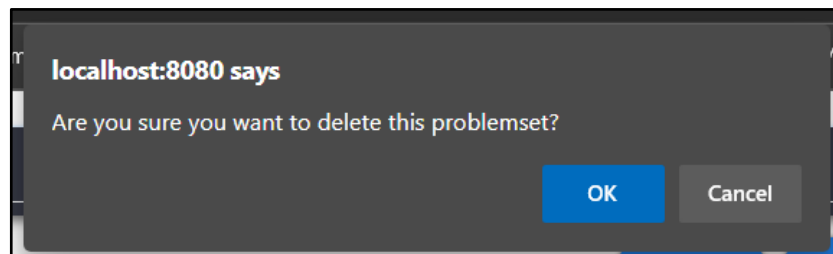
### 3.4.4 Custom Popup Windows

As aforementioned, the majority of the UI for Builder 2.0 had already been designed and implemented. The color scheme, font choices, and UI element formats had already been set and applied to almost every aspect of the website. However, there was one piece of the UI that stood out from the aesthetic of Builder 2.0. Both alerts and confirmation popup windows were being displayed by using the basic HTML library known as *Window*. These popup windows cannot be customized and look extremely out of place on websites that follow a custom design, such as Builder 2.0. [Figure 66](#) provides an example of how Builder 2.0 implemented alerts from the *Window* library. This alert would display anytime a user tried to completely remove the title of a problem set.



*Figure 66: Non-customizable Window Alert for Problem Sets with No Title*

Another example of these popup windows being used in Builder 2.0 was when a user would try to delete a multi-part problem or a problem set. Upon attempting to do so, a confirmation box from the *Window* library would display and ask the user to confirm the deletion ([Figure 67](#)).



*Figure 67: Non-customizable Window Confirmation for Deleting Problem Sets*

In addition, future development of Builder 2.0 would need popup windows asking for user input. It was originally planned to use the Window library for this functionality as well. However, just like the alert and confirmation popups that this library provides, the input popup windows would be non-customizable.

To ensure that the entire feel of the UI was fluid and appealing to the user, our team implemented a custom popup window that could handle alerts, confirmations, and user inputs. To do so, we first had to create a new XML UI layout file (*PopupWindowView.ui.xml*) and link it to a new Java controller class (*PopupWindowView.java*). [Figure 68](#) details *PopupWindowView.ui.xml* and shows the four UI elements that are to be used by the custom popup window:

- 1) A label to display a message to the user,
- 2) A TextArea for users to input information,
- 3) A close button to hide the popup window, and
- 4) An action button to execute any additional code.

The UI layout has the same styling as the preferences popup window that was already designed.

```
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
    xmlns:g="urn:import:com.google.gwt.user.client.ui">
    <ui:style>
        ...
    </ui:style>
    <g:VerticalPanel>
        <g:Label addStyleNames="{style.ps-prop-title}">&#160;</g:Label>
        <g:VerticalPanel>
            <g:HorizontalPanel width="100%">
                <g:HTMLPanel>
                    <fieldset>
                        <g:Label ui:field="label"></g:Label>
                        <g:TextBox ui:field="input" width="95%"></g:TextBox>
                    </fieldset>
                </g:HTMLPanel>
            </g:HorizontalPanel>
            <g:HorizontalPanel>
                <g:cell verticalAlignment="ALIGN_BOTTOM" horizontalAlignment="ALIGN_RIGHT">
                    <g:Button ui:field="actionBtn" text="ActionBtn"
                        addStyleNames="{style.builder-custom-button}"/>
                </g:cell>
                <g:cell verticalAlignment="ALIGN_BOTTOM" horizontalAlignment="ALIGN_RIGHT">
                    <g:Button ui:field="closeBtn" text="CloseBtn"
                        addStyleNames="{style.builder-custom-button}"/>
                </g:cell>
            </g:HorizontalPanel>
        </g:VerticalPanel>
    </g:VerticalPanel>
</ui:UiBinder>
```

Figure 68: *PopupWindowView XML UI Layout File*

With the UI and its elements declared and initialized within the XML UI layout file and the Java controller class, the next step was to make the newly created UI appear as a popup window. To do so, we utilized GWT's DialogBox widget. This element allows developers to inject a UI layout into the GWT popup window functionality. Inside of *ContentBuilder.java*, we declared a new DialogBox called *popup* and initialized an instance of *PopupWindowView.java* called *popupWindowView*. We then injected *popupWindowView* into *popup* by using the internal DialogBox function, *add()*. *Figure 69* details the injection as well as the styling settings that were applied to the DialogBox that mimic the aesthetic of *PreferencesView.java*.

```
popup.add(popupWindowView);
popup.addStyleName("settingsDialogue");
popup.setGlassEnabled(true);
popup.setGlassStyleName("dialog-block-bg");
```

*Figure 69: Setting up the Custom Popup Window DialogBox*

Once the DialogBox was set up, we had to implement a simple way to display the alert, confirmation, and user input variations of *PopupWindowView*. Our team created three individual functions inside of *PopupWindowView.java* to do so.

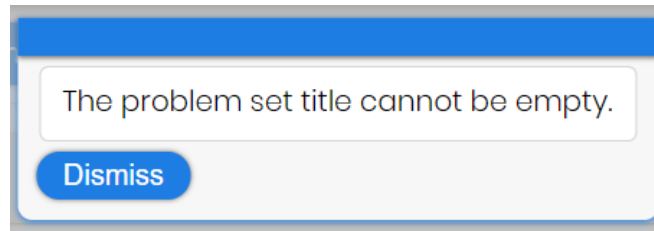
- 1) *displayAlertBox(String message)*: Displays the DialogBox *popup* located within *ContentBuilder.java*. Sets the label to the inputted message string. Hides the input TextArea and action button. Defaults the close button text to 'Cancel'.
- 2) *displayConfirmBox(String message, Runnable action)*: Displays the DialogBox *popup* located within *ContentBuilder.java*. Sets the label to the inputted message string. Makes the action button visible and links its *onClick()* event to the passed in Java runnable. Hides the input TextArea. Defaults the action button text to 'Confirm' and the close button text to 'Cancel'.
- 3) *displayInputBox(String message, Runnable action)*: Displays the DialogBox *popup* located within *ContentBuilder.java*. Sets the label to the inputted message string. Makes the action button visible and links its *onClick()* event to the passed in Java runnable. Makes the input TextArea visible. Defaults the action button text to 'Enter' and the close button text to 'Cancel'.

*PopupWindowView.java* also ensures that the close button always hides the popup window from the users view while also resetting all of the UI elements and class variables. Lastly, the class contains a function that allows developers to retrieve any text entered by the user in the input TextArea, as well as functions to easily change the text of the action button and the close button.

Once the custom popup windows were complete, we replaced all production uses of the Window library. For example, our team utilized *displayAlertBox()* to replace the Window library alert that warned the user that a problem set title cannot be empty. The change was simply replacing the *Window.alert()* calls inside of *BuilderHeaderView.java* with;

```
ContentBuilder.getInstance().popupWindowView.displayAlertBox("The problem set title cannot  
be empty.");
```

The result can be seen in [Figure 70](#) which details the new custom popup alert that follows the same UI design and color scheme as the rest of Builder 2.0.



*Figure 70: New Alert Popup Window for Problem Sets with No Title*

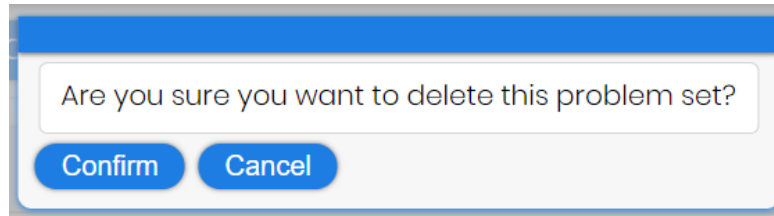
As detailed earlier within the list of functions, both *displayConfirmBox()* and *displayInputBox()* require Java runnables to be passed in. This is because both of these types of popup windows will trigger some sort of action when the user clicks the action button. To make these action buttons do something, they need to trigger some code on click. Any developer working on Builder 2.0 can specify what code they want to run when these action buttons are clicked by creating a runnable and then passing it into either *displayConfirmBox()* or *displayInputBox()*.

An example of utilizing runnables can be seen with the custom popup window implementation that our team used to replace the problem set deletion confirmation that was originally handled by the Window library. This implementation is located within the *deletePSContent()* function held within *ContentBuilder.java* ([Figure 71](#)).

```
public void deletePSContent(AccordionContent content, boolean showAlert)
{
    if (showAlert)
    {
        Runnable action = new Runnable()
        {
            public void run()
            {
                deletePSContent(content);
            }
        };
        popupWindowView.displayConfirmBox("Are you sure you want to delete this problem set?",
            action);
    }
    else
    {
        deletePSContent(content);
    }
}
```

*Figure 71: A Runnable Being Passed into a Custom Confirmation Popup Window*

In this instance, a runnable is created that executes the deletion of a problem set upon being run. That runnable is then passed into the `displayConfirmBox()` function which displays the *popup* `DialogBox` with the intended confirmation message ([Figure 72](#)). If the user clicks the action button labeled as “Confirm”, the runnable will execute, and the problem set will be deleted.



*Figure 72: A Runnable Being Passed into a Custom Confirmation Popup Window*

With `PopupWindowView.java`, Builder 2.0 now has the ability to display alert and confirmation popup windows without obscuring the aesthetic of the UI. `PopWindowView.java` can also handle user inputs and the first instance of that functionality is detailed in the next section.

### 3.5 Importing Content

Another feature implemented by our team was the ability to import existing problems and problem sets into the current problem set. This allows teachers to reuse old problems and problem sets so that they do not have to spend time creating the same content all over again. It also allows them to use publicly available problems and problem sets to further increase productivity. To use this functionality, a user must know the unique ID of the problem or problem set that they wish to import. Then, within the problem set they want to import content to, they need to click on the newly added ‘+ Import Content’ button that is located in the upper right-hand corner of the UI next to the ‘+ New Multi-Part Problem’ button ([Figure 73](#)).

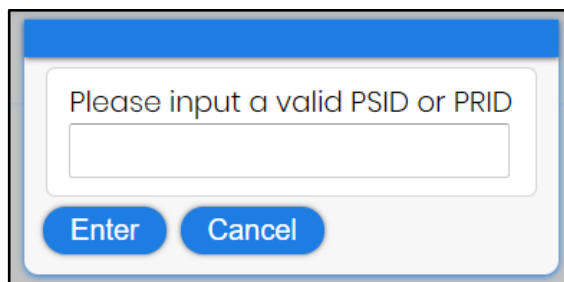


*Figure 73: The Toolbar with the New Import Content Button*

This button was added to the UI by modifying the `BuilderHeaderView` XML UI layout file and its Java controller class. Once the user clicks the button, the `displayImportWindow()` function located within `ContentBuilder.Java` will be triggered. This function is responsible for setting up a user input popup window. First, the function creates a Java runnable called *action* that will parse the user inputted text to find a valid PSID or PRID. These IDs are the unique identifiers associated with each individual problem and problem set stored within the ASSISTments database. Valid problem set IDs (PSIDs) begin with “WPS” and valid problem IDs (PRIDs) begin with “WPR”. If a valid ID is found, the runnable will execute the proper import function. After creation of the runnable, `displayImportWindow()` executes the line;

```
popupWindowView.displayInputBox("Please input a valid PSID or PRID", action);
```

Doing so displays the new custom input popup window which asks the user to enter the PSID or PRID of the content they would like to import and assigns the runnable *action* to the ‘Enter’ button *onClick()* event (*Figure 74*).

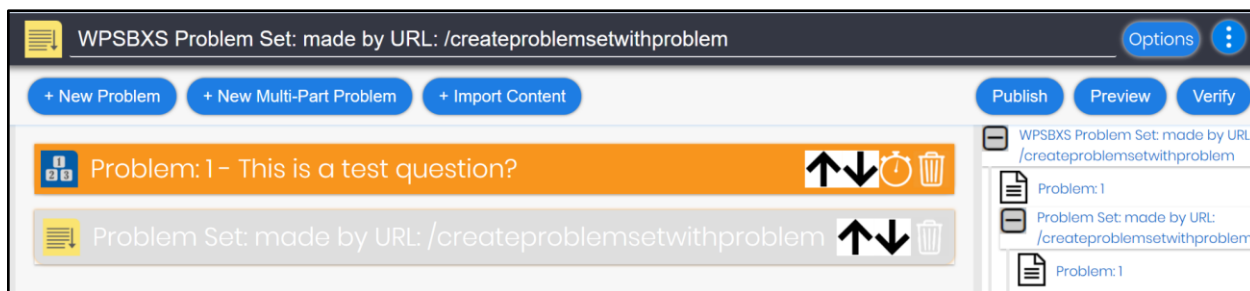


*Figure 74: Import Content Popup Window*

Once the user enters a valid ID, the runnable *action* will trigger, and the corresponding content will be imported and displayed in the UI. The implementation of retrieving already existing content from the ASSISTments database and importing it into the current problem set is detailed in the sections below.

### 3.5.1 Importing Problem Sets

When a valid PSID is found upon execution of the runnable *action* declared within *displayImportWindow()*, the *importPostRequestForSPS()* function will be called. This function is similar to the *postRequestForSPS()* function which finds an existing problem set in the database and returns its contents to be displayed within the front-end. The main difference in *importPostRequestForSPS()* is that it creates a brand new problem set and adds the found problem set content into it. Once all the content is added, that newly created problem set is added into the current problem set that the user is working inside of. *postRequestForSPS()* does not create a new problem set as it is only capable of adding the found problem set content directly into the current problem set. This distinction is further explained within *Figure 75* where a problem set (seen with the grey bar) is imported and added to the current problem set. In addition, the explorer on the right side of the figure shows that the problem imported alongside the problem set is linked to the newly created problem set rather than the current problem set.



*Figure 75: Example of an Imported Problem Set*



The *importPostRequestForSPS()* function works by first making a request to the back-end. This request sends the user inputted PSID to the ‘/getproblemset’ RESTful endpoint in order to retrieve the contents of the problem set, such as the problems and problem sets contained within it. When that endpoint finishes retrieving the data, the found content is sent back to the front-end. On a successful retrieval, the *activeSPS* variable, which points to the current problem set the user is working in, will be set to the problem set returned by the request. This is shown in [Figure 76](#) where the variable *result* is the contents returned by the back-end request. This figure also shows how the *activeSPS* variable is used to retrieve the *problemSetBuilder*.

```
activeSPS = result.getSps();
updatePS(activeSPS);

problemSetBuilder = ProblemSetBuilderProvider.getProblemSetBuilder(activeSPS);
```

*Figure 76: Using a Resulting ProblemSet to Obtain a ProblemSetBuilder*

The *activeSPS* variable is used again to figure out if the imported problem set is linear or random so that the *contentType* field is set to correctly. By setting the correct type, the front-end will be able to load the proper image into the problem set title bar. *importPostRequestForSPS()* then creates a new *VerticalPanel* to hold the contents of the imported problem set. The next step turns the imported contents into a new problem set with the current problem set applied as its parent through the *BuilderProblemSet* constructor ([Figure 77](#)).

```
BuilderProblemSet newProblemSet = new BuilderProblemSet(activeSPS.getName(),
    contentType, imgData, currentPS);
```

*Figure 77: Creating an Imported Problem Set with the Current Problem Set as the Parent*

Our team made sure that the current problem set knew that a new problem set was being added to it as a child. We did this, by using the *addContent()* function and passing in the newly created *BuilderProblemSet*. The *newProblemSet* is then used to extract its *AccordionContent* and *PSContentExplorer* so that its *AccordionContent* can be added to the *VerticalPanel* created earlier. The *PSContentExplorer* is then added as a subset of the current problem set’s explorer. For reference, a problem set’s explorer can be seen with the right hand side menu in [Figure 56](#).

Finally, *importPostRequestForSPS()* adds the content of each problem within the imported problem set to the newly created *BuilderProblemSet*. This is done by executing a for-loop over all of the *MemberContentKeys* stored within *activeSPS*. Each iteration of the loop identifies if the current *contentKey* is a problem or problem set. The loop then calls the appropriate *postRequest* method for the found type and adds the content to the newly created *BuilderProblemSet* ([Figure 78](#)). On a final note, the code for adding a problem set is not yet implemented as it was not in the scope of this project.

```
memberContentKeys = activeSPS.getMemberContentKeys();
ContentKey contentKey;
```



```

for (int i = 0; i < memberContentKeys.size(); i++)
{
    contentKey = memberContentKeys.get(i);
    GWT.log(contentKey.getContentKeyDefnType().toString());
    switch (contentKey.getContentKeyDefnType())
    {
        case PROBLEM:
            GWT.log("It was a problem.");
            postRequestForSPR(simpleProblemUrl, contentKey, problemEditorContainer,
                newProblemSet);
            break;
        case PROBLEM_SET:
            GWT.log("It was a problem set.");
            break;
        default:
            GWT.log("The encountered memberContentKey was neither a Problem nor a Problem Set");
    }
}

```

Figure 78: The For-Loop that Adds Problems and Problem Sets to the Imported Problem Set

### 3.5.2 Importing Individual Problems

The process for importing individual problems is very similar to that of problem sets. The main difference lies in needing to use PRIDs to retrieve the existing content. When our team attempted to pull problem information from the database with a singular PRID, an SDK error that we thought had been fixed revealed itself again. Attempting to use PRIDs to obtain problem data would throw the V1 certification error that is detailed in [Section 3.3.1](#). This error was indeed fixed, but only for manipulating problems by their full content key rather than their PSID. We also tried using the SDK methods that convert a PSID into a full content key, but these functions also threw errors. Since these bugs still needed to be fixed, our team was tasked with providing a “proof of concept” for importing problems. We achieved this “proof of concept” by requiring users to input full problem content keys instead of PRIDs. This use of content keys will be replaced by PSIDs once the new rendition of the V1 certification error and the functions for converting PRIDs into content keys are fixed within the SDK. Since content keys are not intended to be known by the user, these bugs will need to be fixed before Builder 2.0 is publicly released so that PRIDs can be used without error.

Our team implemented the “proof of concept” by modifying the runnable *action* declared within *displayImportWindow()* to look for content keys rather than PRIDs. Since content keys begin with “C-1”, we replaced the instance of “WPR” within the string parser section of the runnable with “C-1” ([Figure 79](#)).

```

else if (id.substring(0, 3).equals("C-1"))
{
    importPostRequestForSPR(simpleProblemUrl, id);
}

```

Figure 79: Parsing for Content Keys instead of PRIDs

If a valid problem content key is found upon execution of the runnable *action*, the *importPostRequestForSPR()* function will be called with the content key passed in as a parameter. Once called, *importPostRequestForSPR()* makes a request to the back-end and sends the content key to the ‘/getproblem’ RESTful API endpoint. This endpoint then processes the content key and retrieves the associated problem data if it exists. Upon successful retrieval, the problem data is sent back to the front-end and the *onDone()* method within *importPostRequestForSPR()* gets executed. The *result* variable returned to *onDone()* by the back-end contains all of the problem data such as the question text and the answers to the question (*Figure 80*).

```
@Override
public void onDone(SPRJsonContainer result)
{
    if (result != null)
    {
        ISimplifiedProblem spr = result.getProblem();
        IProblemBuilder<?> problemBuilder = ProblemBuilderProvider.getProblemBuilder(spr);
        problemSetBuilder.addProblemBuilderToEnd(problemBuilder);
        VerticalPanel target = editorPanels.get(currentTab);
        dragData = new ArrayList<String>(Arrays.asList(ImagePaths.unspecifiedProblem,
            "Unspecified Problem", ICBContent.CBContentType.PUNSPECIFIED.toString()));
        String imgData = dragData.get(0);

        String questionNum = Integer.toString(currentPS.getExplorer().getProblemCount() + 1);
        BuilderProblem newProblem = new BuilderProblem("Problem: " + questionNum + " ",
            problemBuilder.getProblemType(), imgData, problemBuilder);
        ...
    }
    ...
}
```

*Figure 80: Snippet of the onDone() Function Within importPostRequestForSPR()*

To complete the “proof of concept”, our team displayed the imported problem in the UI by implementing the same functionality as *postRequestForSPR()*. The only difference we had to account for was the setting of a content key. As detailed in *Section 3.1.1*, problems created with *postRequestForSPR()* require a content key to be given to the problem drawn in the UI so that it is properly linked to the database. In this case, the imported problem already has a content key linked to it, so that functionality was not necessary. Once *importPostRequestForSPR()* manipulates the imported problem data, a new problem will be displayed in the UI. An example of an imported problem can be seen in *Figure 81* where the first question was imported into the same problem set and displayed as a new problem.

The screenshot displays a web application interface for testing problem import. It features two orange headers for "Problem: 1 - Test importing a problem" and "Problem: 2 - Test importing a problem", each with navigation icons. Below the headers is a rich text editor with a toolbar containing options like font size, bold, italic, underline, strikethrough, list, link, unlink, table, and math symbols. The editor contains the text "Test importing a problem".

Below the editor is an "Answer(s):" section with a "Correct Option" checkbox, a "Feedback: True" checkbox, and an "Add Answer" button. To the right is an "Answer Type" panel with a dropdown menu showing "Numeric Algebraic [Hide]". Below the dropdown are radio buttons for "Number", "Exact Fraction", "Numeric Expression", and "Algebraic Expression". There are also links for "Text [Show]" and "Multiple [Show]".

At the bottom of the interface are three buttons: "Save Draft", "Duplicate Problem", and "Verify". A link "Add Hint(s)/Explanations" is also present.

*Figure 81: “Proof of Concept” For an Imported Problem*

Once the V1 certification error and the functions for converting PRIDs into content keys are fixed within the SDK, the code written by our team will need to be updated. Future developers will need to re-add parsing for PRIDs with “WPR”. Once that is completed, they will need to modify `importPostRequestForSPR()` so that it sends the user inputted PRID to the RESTful API endpoint instead of a full problem content key.

### 3.5 Automated Build Process

To interact with the development version of Builder 2.0 within a web browser, all three projects need to be built: Common, Client, and Service. However, before a developer can start building the projects, they must update all the dependencies managed by Maven. Once the dependencies are loaded and the three projects are built, the Tomcat server resources need to be refreshed. Finally, the Tomcat server then needs to be restarted. Once the project and server are ready for interaction, the developer navigates to the URL that the server console directs them to. While the copying and pasting of the URL into a web browser is very quick, the process of updating the dependencies, building each project, refreshing the server resources, and restarting the server takes much longer and requires the programmer to manually initiate each step. In addition, if a developer made a mistake in executing the build process order, they would have to restart the entire process. To solve these inefficiencies, our team created a Windows batch script that updates all the maven dependencies and builds all three projects automatically. After running the script, developers will only need to refresh the server resources and restart the

Tomcat server, which takes the least amount of time compared to the other steps. This script helps streamline the tedious and time consuming process of getting a development version of Builder 2.0 up and running. [Figure 82](#) contains the batch file that automates building the project.

```
@echo off
:: set tomcat directory to the bin folder within your tomcat installation
set tomcatDirectory=C:/Users/Example/Desktop/apache-tomcat-9.0.37-windows-x64/apache-tomcat-9.0.37/bin
set /P id="Enter the level of building (0 = common, 1 = client, 2 = service): "
if %id% == 0 cd common & mvn clean install & cd ../client & mvn clean install & cd ../service & mvn clean install -DskipTests
if %id% == 1 cd client & mvn clean install & cd ../service & mvn clean install -DskipTests
if %id% == 2 cd service & mvn clean install -DskipTests
cd ..
pause
```

*Figure 82: Batch File that Automatically Builds the Builder 2.0 Project*

While this batch script is made for the Windows operating system, the code can easily be ported to Linux if needed. None of our team members used Linux to develop Builder 2.0 so there was no need to convert it at the time.

## 3.6 Security Audit of Builder 2.0

The last objective of this project was to explore potential security implementations that can protect users' information and prevent cyberattacks within Builder 2.0. To fulfil both this objective and the requirements for the cybersecurity concentration component of his bachelor's degree, Ioannis completed a security audit on the ASSISTments 2.0 Content Builder. This audit focuses on how Builder 2.0 can protect its users from exploitive variations of SQL Injection and JavaScript Injection.

### 3.6.1 SQL Injection

Structured Query Language (SQL) is a programming language for databases. When a database is installed and running, a program can store, modify, and delete objects in that database by sending the database SQL statements. When a website stores information that a user enters, the user may input SQL and a vulnerable website's database will be compromised when it tries to interpret the data that the user provided. For example, when a teacher inputs a problem into the ASSISTments 2.0 Content Builder, they are allowed to put SQL statements in the textbox. Once the data is sent to the database server, the statements that the teacher entered may be interpreted by the database and cause data leaks or data loss. To mitigate this, we can use a technique called SQL Sanitization. It is named this way because a developer must "sanitize" every inlet of data that a user can send to the server. This way, whatever text the user enters is interpreted as text instead of an SQL command.

## **Mitigating SQL Injection in ASSISTment's 2.0 Content Builder**

One way to mitigate SQL injection is to check whether the user inputs any SQL and prevent the user from submitting the malicious text. The benefit of this strategy is that the server does not need to receive and send any data back. This puts less strain on the server because the server will not receive the requests that include invalid inputs. A drawback to this strategy is that the user can easily bypass the checks and send whatever text they want to the server.

To patch the drawback, we must mitigate SQL injections on the server side so that whatever malicious SQL bypasses the first wall will be blocked once it reaches the server. To do this, we use “prepared statements”. Prepared statements mitigate SQL injection attacks by using types when storing data. This mitigates an SQL injection attack because the programmer defines the SQL statement with specific types of data. This way, whatever the user sends will be interpreted as the programmer intends it to be. For example, the programmer can define that they only accept strings. Then, whatever the user sends to the server will be interpreted as a string and not an SQL statement. The ASSISTments SDK uses prepared statements to access the database, so any attempt to inject SQL is thwarted by the SDK.

### **3.6.2 JavaScript Injection**

JavaScript is a programming language that web browsers interpret when the website loads. This code makes websites responsive and run on a user's web browser. This is a major potential vulnerability because whatever JavaScript code is on the webpage will run on a user's device. For example, the ASSISTments 2.0 Content Builder uses JavaScript to make changes when a user presses buttons on the website. In other words, while HTML and CSS make up the visual aspect of websites, JavaScript is the mechanical aspect of websites that make websites function. Attackers can leverage JavaScript injection to execute a multitude of attacks.

#### **Consequences of JavaScript Injection**

An interesting JavaScript injection attack that makes money for the attacker is when the attacker injects JavaScript code that mines a cryptocurrency using the victims' computational resources. This attack is popular because it sends an anonymous payment to the attacker, so it is difficult to track them down.

An attacker could also inject JavaScript that enumerates all the cookies that the user has that are associated with the vulnerable website and send them to himself. Cookies are used to store authentication tokens. With an authentication token, the attacker can claim to be the user visiting the vulnerable website. Now that the attacker is using a stolen identity, they can pretend to be the real user and look at sensitive information that only the legitimate user has permission to see. The implications of this attack vary depending on which platform is vulnerable to this JavaScript injection attack. If it is a bank site, then important financial information will be leaked to the attacker. If it is the ASSISTments 2.0 Content Builder, grades and associated homework problems will be compromised.

## **Mitigating JavaScript Injection in ASSISTment's 2.0 Content Builder**

Mitigating JavaScript injection is similar to mitigating SQL injection as it comes in two parts. The first part would be to detect the JavaScript injection on the client side and prevent it from reaching the server. This mitigation has the same drawbacks though, as a malicious client could bypass the client side detection method. Thus, we resort to a server side protection as well. The protection method on both sides will check for any '<script>' tags then delete them and any text in between them.

The mitigation method proposed in the previous paragraph only works if we want no user-generated JavaScript to run on the site. One of the goals of the ASSISTments 2.0 Content Builder includes allowing teachers to submit their own JavaScript and allow it to run for educational purposes. This creates the challenge of detecting malicious JavaScript, i.e. JavaScript that takes up resources by mining cryptocurrency or cookie-stealing JavaScript.

## 4. Discussion and Analysis of Results

This section provides an overview of the new iteration of the ASSISTments 2.0 Problem Builder component that resulted from our team's work. The new iteration includes data persistence for user created problems, various improvements to the user interface and the ability to import content. In addition to the new software, we provided improvements to the development process and an audit detailing potential implementations for improving the security of Builder 2.0. Following the overview, our team discusses the impact that our work had on both educators and the ASSISTments Team. Lastly, this section details the future work that needs to be completed on Builder 2.0 before the component can be publicly released.

### 4.1 Project Overview

Before our team began work on Builder 2.0, the system was split into pieces. The UI and the aesthetic for the site had been established and implemented. The back-end architecture that could facilitate a RESTful API service had been built. And the SDK was being developed to accommodate the newer data structures designed for ASSISTments 2.0. All of these aspects needed for the new Content Builder were under development, but there were hardly any connections between them that could facilitate a fully functional education tool. The objectives that our team completed in order to accomplish our goal ended up putting a majority of these pieces together.

We began our development by modifying the front-end of Builder 2.0 so that the problems displayed in the UI would be linked to problems stored within the ASSISTments database. This modification required manipulating RESTful API endpoints so that problem data was retrieved from the database, sent back to the front-end and applied to the problems drawn into the UI. To ensure that problem modifications made by the user would be automatically saved to the database, we had to explore all of the UI layouts within Builder 2.0. After identifying the key elements that made up problems within these views, our team implemented various UI event handlers that trigger the saving API services whenever a user makes a modification. To further streamline the development process and improve the code-base, our team compiled all the recurring saving functionality into a new singleton Java class called *MakeAsyncCall.java*. In addition to executing the saving service API calls, this class also filters out any duplicate problem data to ensure efficiency and mitigate any potential saving errors. Finally, test cases were created to make sure all of the added functionality worked as intended.

The back-end systems of Builder 2.0 were also modified. In particular, our team further developed the RESTful API endpoints that the front-end makes requests to. The endpoint `‘/problemservice/save’` is responsible for saving problem content to the database and is used by all the UI event handlers that call *MakeAsyncCal.java*. Originally, this endpoint had a skeleton outline for the saving functionality but was unable to save anything to the database. Our team

updated its functionality so that the endpoint properly manipulates the new SDK and saves any problem data sent to it. In addition to saving problems, fresh instances of the Builder 2.0 website needed to automatically display a new problem set and problem that were both connected to the database. We implemented this functionality by writing a new endpoint called `‘/createproblemsetwithproblem’`. This endpoint creates a blank problem and inserts it into a new problem set within the database. Both the problem set and its problem are then passed to the front-end and loaded into the UI. This endpoint is responsible for giving users a starting point when creating new instances of Builder 2.0. The final endpoint our team developed is `‘/tutoringservice/save’`, which performs actions similar to `‘/problemservice/save’`. In this case, the endpoint saves additional tutoring specific data that is not a part of a standard problem. This endpoint also facilitates the saving of tutoring problems added to the front-end.

After our team implemented the new front-end and back-end functionalities, we made improvements to the UI to make it more fluid and user friendly. We examined each of the three original UI layouts for commonalities and found that the slight differences in functionality did not warrant separation. Our team condensed these three UIs into one singular layout that utilizes toggle buttons for displaying the various functionalities within the UI. Builder 2.0 also had an entirely separate view dedicated to showing problem sets in the form of a “tree”. This ‘Tree View’ was not fully implemented and had a few visual bugs. Our team determined that the ‘Tree View’ did not provide any useful functionality and decided to remove it rather than spending our resources trying to fix it. We also found that using drag-and-drop to reorder problems and problem sets within the UI caused many visual and technical bugs. To prevent issues and avoid user confusion, drag-and-drop was removed, and its functionality was replaced by adding up and down buttons on all the content within a problem set. Lastly, custom popup windows were added to the UI which replaced the default HTML alert and confirmation boxes that were non-customizable.

The implementation for an import content feature was also added. This functionality allows users to take already existing problems and problem sets and add them to their own problem sets. This feature works by taking a user inputted problem set ID or problem ID and then querying the database to find and load the proper content into the user’s current problem set. The functionality for problem sets was fully implemented without issue. However, our team encountered a couple SDK errors when attempting to fully import existing problems into a problem set. Our team instead added a “proof of concept” for importing problems to demonstrate that the implementation will work once the SDK errors are resolved in the future.

Our team also worked on aspects of the project that were outside of making modifications to the Builder 2.0 code-base. After spending many hours manually triggering each step required to build the development version of Builder 2.0, we decided to automate the tedious process. A batch script was written that handles building a majority of the project. This change made it so our team and any future developers would not have to waste time by waiting for each build step to complete. In addition, our team looked into potential security risks with the current state of



Builder 2.0. We then provided recommendations to the ASSISTments Team on various security features that can be implemented in the future that will protect the website and its users from cyberattacks.

## 4.2 Project Impact

The work our team completed to further the development of Builder 2.0 improved the overall functionality and user experience of the system while also paving the way for future development. Once Builder 2.0 is released to the public, our work will help give educators an easy-to-use and intuitive tool that empowers them to create educational content to enhance their students' learning.

The addition of autosaving problems directly to the ASSISTments database removes the need for users to manually save their data. As such, the process of creating new problems is now much easier for the user as they do not need to worry about making sure their modifications are being saved. Every single change that a user makes to a problem will be updated within the database system in real time. This auto saving and data persistence functionality ensures that users will not lose any progress on their custom content when extenuating circumstances occur, such as internet outages or computer crashes. By removing the potential for data loss, the new saving abilities for Builder 2.0 create a safer and more appealing experience for its users.

The new condensed UI layout provides users with all of Builder 2.0's functionality in a simpler to understand and streamlined fashion. Rather than having to switch between different UI layouts to access certain features, users can now easily toggle the components of Builder 2.0 that they want to use. The ability to toggle the visibility of these features also gives complete control to the user in terms of how the space within their web browser is used. In addition, the removal and replacement of dysfunctional, unnecessary, and out of place UI features ensure that users will not get frustrated or confused while using the website. Overall, Builder 2.0 now presents the user with a much more enjoyable and fluid experience.

The reuse of previously created content is another Builder 2.0 feature that gives more power to its users. By being able to import existing problems and problem sets, users are now able to reuse content they have already made. This reduces the amount of time it can take to create a new problem set since users no longer need to recreate content that is the same or similar to previous content they have created. Furthermore, the importing content feature allows users to collaborate with each other. To do so, users can share the problem IDs and problem set IDs of their own custom content with other users so that they can import and use that content as well. For example, educators within the same field of study will be able to share their content and build off of each other. Collaboration is a key component to education and is now facilitated within Builder 2.0.

For the matter of future development on Builder 2.0, the process that developers need to follow in order to run a development version of the website has been automated. Developers will

now be able to run a script that takes care of the majority of the build process without needing any manual interactions. While this automation may seem insignificant compared to the actual development of the system, it provides more efficiency for the developer and removes the potential of making a mistake in terms of building the code-base in the proper order. More development will also be needed to ensure that Builder 2.0 is secure. With our team's research and analysis of various security implementations, future developers have a starting point for building protections to keep the users of Builder 2.0 safe.

## 4.3 Future Work

Before Builder 2.0 can be released publicly, the Minimum Viable Product needs to be completed. Our team has identified six key additions and improvements that will need to be implemented before Builder 2.0 can reach its first release state.

### 4.3.1 Fixing the Saving Answers SDK Error

The first main objective for our team was to ensure that problems were saving to the ASSISTments database in real time. While we did correctly implement the saving functionality for problems, we ran into an issue where all aspects of a problem were being saved except for their answers. This is due to an error in the SDK where answers are not being properly located and stored in the SQL database. More details on this error can be found in [Section 3.3.4](#). Once this error is fixed, data persistence for problems will be fully completed.

### 4.3.2 Adding Data Persistence for Problem Sets

While our team implemented the automatic saving of problems created by the user, the saving of problem sets still needs to be implemented. We did not implement this functionality as it was out of the scope of our project. As of the date of publishing this paper, the functionality for adding new content into a problem set is not properly updating the ASSISTments database. This means that even though adding problems, multi-part problems and problem sets into the current problem set is displayed within the UI, they are never added to the associated problem set in the back-end. Therefore, these types of modifications to a problem set do not persist. To add this functionality, future developers will need to modify, and potentially add, the RESTful API endpoints that deal with saving problem set data. These endpoints will then need to be linked to all the UI elements that make up the way users add, remove, and modify content within a problem set.

### 4.3.3 Fixing the SDK Bugs Affecting Importing Content

Coming into the final weeks of the project, our team was requested to create a feature for importing existing content into the problem set a user was currently working in. We were unable to completely finish this feature by the end of the project due to a couple of errors that appeared

in the SDK. These SDK errors center around being unable to convert a problem ID into its associated content key as well as issues with database manipulation. More details on these errors can be found in [Section 3.5.2](#). Because of these issues, our team completed a “proof of concept” for the problem side of importing content. This temporary version of importing problems utilizes content keys rather than problems IDs to fetch existing problem data. In addition, the SDK is not functioning properly in terms of saving both imported problems and problem sets. Rather than creating new instances of the imported content in the database, the imported versions of problems and problem sets are pointing to their original versions. This means that any modifications made to imported content will also modify the original content. The intended functionality is that imported content will be separated from the original versions so that users can modify the imported content without affecting the original content. Before the feature of importing problems and problem sets can be finalized, further investigation of these errors will need to be conducted within SDK 3.0 so that fixes can be implemented.

#### 4.3.4 Cleaning Up the User Interface

The Builder 2.0 UI will also need some more work before the website can go live. First, the ASSISTments UI Team will need to address the placeholder UI elements that are present within the website. The placeholder images for the up and down arrows buttons our team implemented to facilitate the reordering of content need to be replaced with images that fit the aesthetic of the UI. The toggle buttons for the palette and explorer side menus also need to be updated to match the UI. These two toggle buttons also need to be moved to a new location. Our team recommends attaching these buttons to the upper corners of their respective menus. Lastly, the placeholder UI elements that were placed within the *OptionsHeaderView* before our team started development need to be addressed. Once all of the placeholders are replaced, future developers will need to conduct an extensive review of the UI to identify any lingering bugs and implement fixes.

#### 4.3.5 Adding Restrictions Based on User Authority

The three UI layouts that Builder 2.0 originally used to separate different functionalities were also linked to an initial implementation of restricting features based on the “level” of a user. As detailed in [Section 3.4.1](#), this *userLevel* functionality was removed when our team condensed the UI since having three different UIs was not a good design. Our team then implemented toggles that allow users to enable and disable the more advanced features present in Builder 2.0. As of the date of publishing this paper, there were no features present within Builder 2.0 that needed to be hidden from certain users. However, in the event that certain features do need to be restricted to users with specified authority, the *userLevel* functionality can be reimplemented. Future developers will be able to set up these types of user authority restrictions by utilizing the new infrastructure our team created to enable and disable specific features.

### 4.3.6 Preventing Malicious Injections

As detailed in [Section 3.6](#), Ioannis conducted a security audit on the Builder 2.0 website. He found that Builder 2.0 is currently susceptible to harmful cyberattacks from SQL Injection and JavaScript Injection. Within the audit, he detailed various ways that Builder 2.0's front-end UI and the back-end server can be modified to prevent these attacks from compromising user data and the website itself. Future developers will need to implement these protections before the Builder 2.0 can be released to the public. In addition, our team was informed that there are plans to add an “externally run” feature into Builder 2.0. This functionality would allow users to safely inject JavaScript code to further customize their problem sets. This is a feature that could potentially be restricted to authorized users only ([Section 4.3.5](#)). When this feature is implemented, developers will need to ensure that the injected JavaScript is only processed if there is no malicious code within it.

## 5. Conclusion

To accomplish our goal of further developing the Minimum Viable Product for the ASSISTments 2.0 Content Builder component, our team successfully completed five objectives. The first ensured that user created problems were always staying up-to-date within the ASSISTments databases. This objective was accomplished by implementing RESTful API calls into the back-end of Builder 2.0 and linking them to the UI components in the front-end. The next completed objective resulted in a more condensed and fluid user experience through the removal of unnecessary and complex features, and the addition of custom popup windows. Once the UI had been updated, our team utilized the RESTful API calls and custom popup windows to implement features that allow users to import existing content into their projects. While implementing these features, our team also improved the development process through the creation of a script that automatically builds various aspects of the Builder 2.0 code-base. The last completed objective focused on identifying various ways to improve user security and mitigate potential cyberattacks within Builder 2.0. Through these methods, our team delivered a new iteration of the ASSISTments 2.0 Content Builder component that is more robust and intuitive than its predecessor. With this new version of Builder 2.0, the ASSISTments Team is much closer to completing a Minimum Viable Product. Upon the public release of the system, teachers will be empowered to collaborate and create custom content that will further educate and impact all of their students.

# References

- The Apache Software Foundation (ASF). (2021, January 15). *Apache Maven Project - Introduction*. Retrieved from <https://maven.apache.org/what-is-maven.html>
- ASSISTments. *About Us*. (2020). Retrieved from <https://new.assistments.org/about>
- Christensson, P. (2006). *XML Definition*. Retrieved from <https://techterms.com>
- Christensson, P. (2011, June 1). *Markup Language Definition*. Retrieved from <https://techterms.com>
- Dilshan, D. (2018, August 16). *Do you know MAVEN ? A Dependency Manager or a Build tool ? What is POM ?* Retrieved from <https://medium.com/@dulajdilshan/do-you-know-maven-a-dependency-manager-or-a-build-tool-what-is-pom-bd7dd8b43e80>
- Fol, P. (2020, August 19). *Java Basics: What Is Apache Tomcat?* Retrieved from <https://www.jrebel.com/blog/what-is-apache-tomcat>
- GeeksForGeeks. (2019, October 10). *Serialization and Deserialization in Java with Example*. Retrieved from <https://www.geeksforgeeks.org/serialization-in-java/>
- Google Web Toolkit (GWT). (n.d.). *Build User Interfaces*. Retrieved from <http://www.gwtproject.org/doc/latest/DevGuideUiBinder.html>
- Google Web Toolkit (GWT). (n.d.). *Compiling and Debugging*. Retrieved from <http://www.gwtproject.org/doc/latest/DevGuideCompilingAndDebugging.html>
- Google Web Toolkit (GWT). (n.d.). *Overview*. Retrieved from <http://www.gwtproject.org/overview.html>
- Gutl, C., & Chang, V. (2008). *The use of Web 2.0 Technologies and Services to support E-Learning Ecosystem to develop more effective Learning Environments*. Retrieved from [https://espace.curtin.edu.au/bitstream/handle/20.500.11937/12467/128535\\_12222\\_V%20C%20ICDEM%202008.pdf?sequence=2](https://espace.curtin.edu.au/bitstream/handle/20.500.11937/12467/128535_12222_V%20C%20ICDEM%202008.pdf?sequence=2)
- Oracle. (2020). *Lesson: Annotations*. Retrieved from <https://docs.oracle.com/javase/tutorial/java/annotations/>
- Q-Success. (2020). *Usage statistics of HTML for websites*. W3 Techs: Web Technology Surveys. Retrieved from <https://w3techs.com/technologies/details/ml-html>
- Tiny Technologies, Inc (TTI). (2021). *TinyMCE / Documentation*. Tiny. Retrieved from <https://www.tiny.cloud/docs/>

## Appendix A: List of UI Elements with MakeAsyncCall()

### Java Controller Class

Element Name ( <i>UiField</i> tag)	Type of Event Handler	Handler Method Name
------------------------------------	-----------------------	---------------------

#### AddAnswerView.java

removeAnswer	Custom	@UiHandler("removeAnswer") void onClick(ClickEvent e)
correctCheck	Custom	@UiHandler("correctCheck") void onCorrectClick(ClickEvent e)
inputAnswer	Built-in	onValueChange()
answerExplanation	Built-in	onValueChange()

#### AddHintView.java

removeHint	Custom	@UiHandler("removeHint") void onClick(ClickEvent e)
TinyMCEView view	Built-in	onBlur()

#### AddMCEAnswerView.java

removeAnswer	Custom	@UiHandler("removeAnswer") void onClick(ClickEvent e)
correctCheck	Custom	@UiHandler("correctCheck") void onCorrectClick(ClickEvent e)
removeHint	Custom	@UiHandler("removeHint") void onClick(ClickEvent e)
TinyMCEView view	Built-in	onBlur()
answerExplanation	Built-in	onValueChange()

#### AnswerMainPanelView.java

addAnswer	Custom	@UiHandler("addAnswer") void onClick(ClickEvent e)
autoL	Custom	@UiHandler("autoL") void onAutoL(ClickEvent e)
autoN	Custom	@UiHandler("autoN") void onAutoN(ClickEvent e)
order	Built-in	onChange()
autoLChoices	Built-in	onChange()

autoNChoices	Built-in	onChange()
--------------	----------	------------

#### AnswerTypeInfoView.java

number	Custom	@UiHandler("number") void onClickNumber(ClickEvent e)
exactf	Custom	@UiHandler("exactf") void onClickExactf(ClickEvent e)
nume	Custom	@UiHandler("nume") void onClickNume(ClickEvent e)
alge	Custom	@UiHandler("alge") void onClickAlge(ClickEvent e)
exactcs	Custom	@UiHandler("exactcs") void onClickExactcs(ClickEvent e)
exactci	Custom	@UiHandler("exactci") void onClickExactci(ClickEvent e)
multc	Custom	@UiHandler("multc") void onClickMultc(ClickEvent e)
allta	Custom	@UiHandler("allta") void onClickAllta(ClickEvent e)
order	Custom	@UiHandler("order") void onClickOrder(ClickEvent e)
openr	Custom	@UiHandler("openr") void onClickOpenr(ClickEvent e)

#### BuilderHeaderView.java

psTitle	Built-in	onValueChange()
---------	----------	-----------------

#### HintExplanationView.java

addHint	Custom	@UiHandler("addHint") void onHintClick(ClickEvent e)
TinyMCEView view	Built-in	onBlur()

#### ProblemEditorPanelView.java

TinyMCEView view	Built-in	onBlur()
------------------	----------	----------