

PolyFlowBuilder: An Intuitive Tool for Academic Planning at Cal Poly San Luis Obispo

A Senior Project Report
presented to
the Faculty of California Polytechnic State University
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Science in Computer Engineering

By
Duncan Thomas Applegarth
June 2023

© 2023 Duncan Applegarth

Table of Contents

LIST OF FIGURES	4
LIST OF TABLES	5
ABSTRACT	6
ACKNOWLEDGEMENTS	6
I. INTRODUCTION	7
PROBLEM STATEMENT	7
BIRTH OF THE ORIGINAL POLYFLOWBUILDER	9
THE NEED FOR A NEW POLYFLOWBUILDER	9
II. SCOPE OF WORK	10
PHASE I: DATA API	10
PHASE II: USER ACCOUNT MANAGEMENT	11
PHASE III: FLOWCHART MANAGEMENT	11
PHASE IV: FLOWCHART MANIPULATION	11
PHASE V: FLOWCHART UTILITIES	12
DEVELOPMENT TIMELINE	12
III. BACKGROUND	13
DEFINITIONS	13
WEB APPLICATION ANATOMY	14
FRONTEND COMPONENT FRAMEWORKS	15
BACKEND FRAMEWORKS	16
WEB APPLICATION FRAMEWORKS	16
DATA PERSISTENCE	16
ANATOMY OF A POLYFLOWBUILDER FLOWCHART	17
IV. APPLICATION DESIGN	19
DESIGN TENETS	19
DESIGN DECISIONS	19
APPLICATION ARCHITECTURE DESIGN	21
USER INTERFACE DESIGN	22
V. APPLICATION INTERFACE	24
APPLICATION INTERFACE OVERVIEW	24
FLOWCHART EDITOR INTERFACE	29
FLOWCHART VIEWER INTERFACE	29
FLOWCHART INFO PANEL INTERFACE	31
VI. POLYFLOWBUILDER DATA API	34
DATA SOURCES	34
DATA COLLECTION TECHNIQUES	35
DATA CLEANING TECHNIQUES	36
VII. USER ACCOUNT MANAGEMENT	38
CREATE ACCOUNT	39
DELETE ACCOUNT	40
LOGIN/LOGOUT	41
PASSWORD RESET	42
VIII. FLOWCHART MANAGEMENT	45
CREATE A NEW FLOWCHART	45

DELETING AN EXISTING FLOWCHART	46
SELECT A FLOWCHART TO EDIT	47
IX. FLOWCHART MANIPULATION	48
MODIFY FLOWCHART TERMS	49
MODIFY FLOWCHART METADATA	51
MODIFY FLOWCHART COURSE CONTENT	53
<i>Move Course(s)</i>	54
<i>Add New Course(s)</i>	54
<i>Course Selection(s)</i>	54
<i>Delete Course(s)</i>	55
<i>Change Color of Course(s)</i>	55
<i>Modify Course(s) Contents</i>	55
<i>Persisting Term Changes</i>	56
X. FLOWCHART UTILITIES	58
DUPLICATE FLOWCHART	58
EXPORT FLOWCHART AS PDF	59
XI. STATE MANAGEMENT	60
USER AUTHENTICATION	60
USER DATA SYNCING	61
FRONTEND STATE MANAGEMENT	63
XII. BACKEND	66
SERVING CONTENT EFFECTIVELY	66
BACKEND APIs	67
XIII. DATABASE	68
DATA TABLES	68
INTERACTING WITH THE DATABASE	69
XIV. EVALUATION	71
EVALUATION METRICS	71
EVALUATION RESULTS	72
<i>Feature Implementation</i>	72
<i>Feature Parity</i>	74
<i>End-to-End and Integration Tests</i>	74
<i>Unit Tests</i>	75
<i>User Feedback</i>	76
XV. CONCLUSIONS	78
CURRENT PROJECT STATE	78
PROJECT TAKEAWAYS	78
FUTURE WORK	79
REFERENCES	80

List of Figures

Figure 01. Sample Cal Poly PDF flowchart	7
Figure 02. Scheduler Builder	8
Figure 03. Sample PolyFlows Photo	8
Figure 04. Web Application Architecture High-Level Diagram	14
Figure 05. Common JavaScript Frontend Frameworks	15
Figure 06. Common JavaScript Backend Frameworks	16
Figure 07. Database Categorization Diagram	17
Figure 08. PolyFlowBuilder Architecture Diagram	21
Figure 09. Side-by-side Comparison of Old vs. New PolyFlowBuilder User Interfaces	22
Figure 10. Responsive User Interface Example	23
Figure 11. PolyFlowBuilder Home Page	25
Figure 12. PolyFlowBuilder About Page	25
Figure 13. PolyFlowBuilder Submit Feedback Page	26
Figure 14. PolyFlowBuilder Login Page	26
Figure 15. PolyFlowBuilder Registration Page	27
Figure 16. PolyFlowBuilder Forgot Password Page	27
Figure 17. PolyFlowBuilder Reset Password Page	28
Figure 18. PolyFlowBuilder Flowchart Editor Page	28
Figure 19. Flowchart Editor Decomposition	29
Figure 20. Flowchart Viewer Decomposition	29
Figure 21. Interacting with The Flowchart Viewer	31
Figure 22. Flowchart Info Panel Decomposition	32
Figure 23. Create Account Sequence Diagram	39
Figure 24. Delete Account Sequence Diagram	40
Figure 25. Login Sequence Diagram	41
Figure 26. Logout Sequence Diagram	42
Figure 27. Initiate Password Reset Sequence Diagram	43
Figure 28. Complete Password Reset Sequence Diagram	44
Figure 29. Create New Flowchart Modal	45
Figure 30. Create New Flowchart Sequence Diagram	46
Figure 31. Delete Existing Flowchart Sequence Diagram	47
Figure 32. Flowchart Actions Dropdown Menu	48
Figure 33. Add Flowchart Terms Modal	49
Figure 34. Add Flowchart Terms Sequence Diagram	50
Figure 35. Remove Flowchart Terms Modal	50
Figure 36. Remove Flowchart Terms Sequence Diagram	51
Figure 37. Edit Flowchart Properties Modal	52
Figure 38. Edit Flowchart Properties Sequence Diagram	53

Figure 39. Course Search Sequence Diagram	54
Figure 40. Course Color Selector Interface	55
Figure 41. Customize Course Modal	56
Figure 42. Persist Term Changes Sequence Diagram	57
Figure 43. Duplicate Flowchart Sequence Diagram	58
Figure 44. Export Flowchart as PDF Sequence Diagram	59
Figure 45. Session-Based vs Token-Based Authentication Diagram	61
Figure 46. Naïve User Data Update Sequence Diagram	62
Figure 47. Update Chunking Sequence Diagram	63
Figure 48. State Management via Component Props	64
Figure 49. State Management via Component Bindings	64
Figure 50. State Management via Custom Events	65
Figure 51. State Management via Svelte Stores	65
Figure 52. Web Rendering Techniques Comparison	67
Figure 53. Database Entity-Relationship Diagram	69
Figure 54. Explicit SQL Statement Code Example	70
Figure 55. Object-Oriented Database Interaction Code Example	70
Figure 56. End-to-end/Integration Test Runner	75
Figure 57. Unit Test Runner	76

List of Tables

Table 01. Pertinent Definitions	13
Table 02. PolyFlowBuilder Flowchart Components	17
Table 03. PolyFlowBuilder Design Decisions	19
Table 04. Web Application List of Pages	24
Table 05. Flowchart Viewer Components Description	30
Table 06. “Manage Flows” Tab Components Description	32
Table 07. “Add Courses” Tab Components Description	33
Table 08. Data API Sources	34
Table 09. Data Collection Techniques	35
Table 10. Data Cleaning Techniques	36
Table 11. User Account Management System Components	38
Table 12. PolyFlowBuilder Backend APIs	67
Table 13. Database Data Tables	68
Table 14. Senior Project Evaluation Criteria	71
Table 15. Feature Implementation Results	72
Table 16. User Testing Feedback	76

Abstract

PolyFlowBuilder is a web application that lets users create visually intuitive flowcharts to aid in academic planning at Cal Poly. These flowcharts can be customized in a variety of ways to accurately represent complex academic plans, such as double majors, minors, taking courses out-of-order, etc. The original version of PolyFlowBuilder, released Summer 2020, was not written for continued expansion and growth. Therefore, a complete rewrite was determined to be necessary to enable the project to grow in the future. This report details the process to completely rewrite the existing version of PolyFlowBuilder over the course of six months, using NodeJS, SvelteKit, TypeScript, MySQL, Prisma, and TailwindCSS + DaisyUI for the primary tech stack. The project was determined to be largely successful by a variety of holistic evaluation criteria, with the main limiting factor to complete success being time constraints. The rewritten version of PolyFlowBuilder will ensure the project's continued success.

Acknowledgements

Taking on such a monumental project in a short period of time is no easy task that I could not have done alone. This project represents a culmination of years of education and experience afforded through my education at Cal Poly. To this end, I would like to thank:

1. My advisor, Dr. Nico, for supporting this project's development and guiding me in the right direction to success.
2. The Cal Poly Computer Engineering Department and its faculty for giving me a high-quality education during my time here and teaching me the necessary skills to successfully accomplish a project of this scale.
3. My family and friends for supporting the project and pushing me to continue this project (especially when I felt unmotivated to do so).
4. Everyone who uses PolyFlowBuilder and contributes feedback to make the project better.

The initial and continued success of PolyFlowBuilder would not have been possible without the contributions of everyone mentioned above. Thank you all so much!

I. Introduction

Problem Statement

As a university student, planning one's academic schedule to fulfill all degree requirements in a timely fashion is no easy task. Students need to take the correct courses in the correct order over the course of several years, taking care that:

1. Course prerequisites are satisfied.
2. All other degree requirements are satisfied (USCP, GWR, GE, etc.).
3. Sections of courses are taught by faculty that work best for the student.
4. The course load for each term is well balanced.
5. Adequate degree progress is being made every term.
6. A variety of other considerations are met.

Things get more complicated if the student chooses to pursue additional educational goals, such as obtaining a minor or additional majors along with their primary degree coursework. Unique academic circumstances must also be considered that alter the standard timeline for course scheduling.

At Cal Poly, a variety of official and unofficial resources have been made available to students over the years to make this process more manageable. A non-exhaustive list of these resources is mentioned below:

1. Sample "flowcharts" are made available online [1] that visualize the standard progression of degree progress for each major and concentration in PDF form. This progress is shown with an intuitive list of courses to be taken each term during the student's time at Cal Poly.

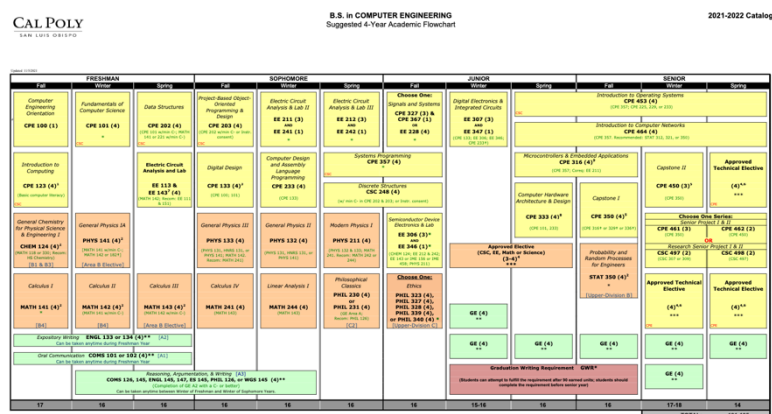


Figure 1: Sample PDF flowchart for the B.S. in Computer Engineering (2021-2022 catalog) at Cal Poly [2].

2. Publicly available Cal Poly course catalog information [3].
3. Internal resources available to current students such as Degree Progress Report, Schedule Builder, Degree Planner, Projected Course Schedules, and others.

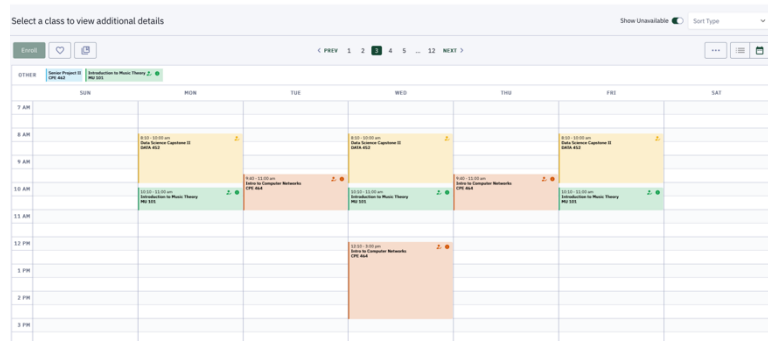


Figure 2: Schedule Builder, an internal tool that allows students to plan, visualize, and enroll in courses for a particular term.

4. External websites that have rating information for professors (e.g., Rate My Professor [4] and Polyratings [5])
5. Academic advisors

What is distinctly missing from this list is an intuitive visual tool that allows students to experiment and plan out their academic programs, by using the well-known format of the sample flowchart PDFs.

The concept of a tool to visualize course scheduling like the sample flowchart PDF templates is not new. In fact, a tool like this previously existed, created by Cal Poly students Connor Batch and Ian Meeder years ago, called PolyFlows. Archives of PolyFlows can be viewed on the Internet Archive “Wayback Machine” as early as 2015 [6]. Many students, me included, utilized this tool to visualize complex academic schedules.



Figure 3: Photo of my flowchart on PolyFlows before it went offline. Photo taken 6/1/2020.

However, PolyFlows suffered from the fate that a majority of student-run projects encounter: once the core maintainers of the project graduate, the project does not get maintained and it eventually goes offline. PolyFlows went offline ~Summer 2020, and a replacement was sorely needed.

Birth of The Original PolyFlowBuilder

To fill the gap of visualization tools for academic planning, “PolyFlowBuilder 1.0” [7] was created as a tool that aimed to achieve the same goals as PolyFlows but with more utilities, data, and long-term support. It ended up becoming my first full-stack web development project written before my sophomore year at Cal Poly, and it was a fantastic learning experience.

PolyFlowBuilder 1.0 first went live Summer 2020, and currently has over 3,000 users at the time of writing. These users include students, parents, faculty, advisors, and prospective students from all over the world. This broad user base shows the real need and usefulness of a tool such as PolyFlowBuilder for academic planning.

The Need for a New PolyFlowBuilder

Over time, as more features were developed and more people became dependent on PolyFlowBuilder, it became clear that the codebase was fragmented, primitive, and unsustainable for scale and further development. The original codebase was written very quickly and not intended for large-scale use. As an example, to further illustrate this point, all frontend logic of PolyFlowBuilder 1.0 is written in a single JavaScript file (as opposed to using a more scalable solution such as a component framework).

Ideas were tossed around about how to rectify these issues and continue development. In the end, the only solution to permanently solve these issues was to simply rewrite the entire project from the ground up. Care would have to be taken to ensure decisions were made with the long-term in mind, to make sure another rewrite-scale effort would not need to happen in the future.

This rewrite prioritizes longevity, reliability, and ease of maintenance, so that the project can continue to be of use to the thousands of people who use it. The long-term scope for this effort is to make PolyFlowBuilder a production-quality application that will allow it to live long after I graduate in the care of other Cal Poly students. Hopefully, this ensures that PolyFlowBuilder will not suffer from the same fate that many other student software projects at Cal Poly face.

Therefore, this senior project is dedicated to creating “PolyFlowBuilder 2.0” from the ground up, using modern software development practices and tooling. As such, the final deliverable is a rewritten PolyFlowBuilder codebase that has feature parity with the existing PolyFlowBuilder 1.0 website.

II. Scope of Work

To ensure that progress towards the final deliverable could be tracked adequately, a scope of work was developed. Therefore, the following specification was defined, divided into five distinct phases:

- I. Data API
- II. User Account Management
- III. Flowchart Management
- IV. Flowchart Manipulation
- V. Flowchart Utilities

These feature requirements were wholly defined by the current feature set in PolyFlowBuilder 1.0. The remainder of this chapter details these requirements.

Phase I: Data API

This phase includes collecting and organizing all data that will be consumed by PolyFlowBuilder, as well as making this data available as the “Data API”. This dataset includes data such as course information, prerequisite information, template flowcharts, etc.

The scope of work for this phase is:

1. To be able to access and store course data for all supported course catalogs (>2015). This includes:
 - a. standard metadata about each course (name, description, number of units, etc.).
 - b. whether a course fits into any categories, such as GE, USCP courses, or GWR courses.
 - c. requisites for a course (which includes prerequisites, corequisites, concurrent requisites, and recommended requisites).
2. To be able to access and store course data for all supported programs (catalog, major, and concentration combinations).
3. To be able to access and store template flowchart data, which represents the template flowcharts made available online [1].
4. All data mentioned above should be stored in the PolyFlowBuilder database and a consistent API interface must be available for the web application to consume these data.
5. Automated mechanisms should exist to collect all data mentioned above. Sources these data are collected from must be authoritative.

Phase II: User Account Management

This phase includes developing the various flows for account management on the platform.

The scope of work for this phase is:

1. To design, develop, build, and test the following user account flows:
 - a. Registration flow
 - b. Login/Logout flow
 - c. Delete account flow
 - d. Reset password flow

The work for each of these processes includes the frontend work as well as building all relevant backend APIs.

2. To design, develop, build, and test the APIs to access relevant user data.
3. All user data should be stored in the PolyFlowBuilder database and must be accessible via APIs for the frontend to consume.

Phase III: Flowchart Management

This phase includes implementing features for flowchart management.

The scope of work for this phase is:

1. To design, develop, build, and test the ability to create new flowcharts.
2. To design, develop, build, and test the ability to view all flowcharts that belong to the logged in user.
3. To design, develop, build, and test the ability to delete existing flowcharts that belong to the logged in user.

Each of these high-level features includes all frontend and backend work required.

Phase IV: Flowchart Manipulation

This phase includes implementing features related to manipulating a selected flowchart.

The scope of work for this phase is:

1. To design, develop, build, and test the ability to search for new courses to add to a selected flowchart.
2. To design, develop, build, and test the ability to add new terms to a selected flowchart.
3. To design, develop, build, and test the ability to delete existing terms from a selected flowchart.
4. To design, develop, build, and test the ability to manipulate the following flowchart metadata:
 - a. Flowchart name
 - b. Flowchart notes
 - c. Associated flowchart programs

5. To design, develop, build, and test the ability to manipulate courses within a selected flowchart:
 - a. Adding new courses (from course search)
 - b. Moving courses in a flowchart
 - c. Deleting existing courses in a flowchart
6. To design, develop, build, and test the ability to customize course(s) within a selected flowchart:
 - a. Changing the metadata for a particular course, which includes:
 - i. Course name (e.g., “MU101”)
 - ii. Course display name (e.g., “Introduction to Music Theory”)
 - iii. Course description
 - iv. Course units
 - b. Changing the course colors in the flowchart

Phase V: Flowchart Utilities

This phase includes implementing various flowchart utilities.

The scope of work for this phase is:

1. To design, develop, build, and test the ability to duplicate a selected flowchart.
2. To design, develop, build, and test the ability to export a selected flowchart as a PDF.
3. To design, develop, build, and test a suite of flowchart validation tools:
 - a. Total minimum units – this validates whether a flowchart has the minimum number of units to graduate (180).
 - b. Total upper-division minimum units – this validates whether a flowchart has the minimum number of upper-division units to graduate (60).
 - c. Class prerequisite validation – validate whether course requisites were met for each course.
 - d. GWR/USCP validation – validate whether a flowchart has courses that satisfy the GWR/USCP requirements to graduate.

Each phase of feature requirements was approached roughly in the order presented here.

Development Timeline

The planned timeline for this work was very volatile and unstructured due to the nature of uncertainty of the rewrite effort. However, in hindsight, each of the five phases of scope of work were roughly approached in the order presented, dividing the two quarters of work into five chunks. Phases IV and V took significantly longer than anticipated, with time constraints preventing the full completion of portions of these phases (see the **Evaluation** chapter).

III. Background

This chapter contains a variety of background topics that are relevant to the architecture and design of the PolyFlowBuilder project.

Definitions

This section includes a variety of definitions pertinent to the report content, shown in **Table 1**.

Table 1. Pertinent definitions

Term	Definition
Flowchart	A document that visualizes one's course progression to complete academic program(s) at Cal Poly.
Web Application	An interactive website that provides the user with functionality that would traditionally be seen in a standalone desktop application.
Frontend	The portion of an application that the users directly interact with. This includes things such as the GUI.
Backend	The portion of an application that users do not directly interact with. The frontend usually communicates with the backend to achieve expected functionality. This includes things such as APIs and servers.
Database	<p>The portion of an application that stores large volumes of data. Databases are highly optimized to store and query these large volumes of data efficiently.</p> <p>Examples of databases are MySQL [8], PostgreSQL [9], and MongoDB [10].</p>
Application Programming Interface (API)	<p>An exposed set of defined rules (interfaces) that allow different types of applications to communicate with each other in a standard way.</p> <p>As an example, the frontend of a web application communicates with the backend through a variety of developer-defined APIs.</p>

Object Relational Mapper (ORM)	A piece of software that allows the backend to interact with the database via object-oriented paradigms instead of issuing declarative database commands.
Router	The portion of the backend that determines where to send an incoming request from a client.
Academic Program	In the context of this report, this is either a (catalog, major, concentration), or a (catalog, minor) tuple.
HyperText Markup Language (HTML)	The markup language that gives all web pages their basic structure. HTML forms the “skeleton” of a web page.
JavaScript	The programming language that enables static HTML pages to become interactive.
Cascading Style Sheets (CSS)	The stylesheet language that gives all web pages their visual properties and styles.

Web Application Anatomy

A vast majority of web applications contain at least three core components, seen in **Figure 4**: the server, the client, and the database.

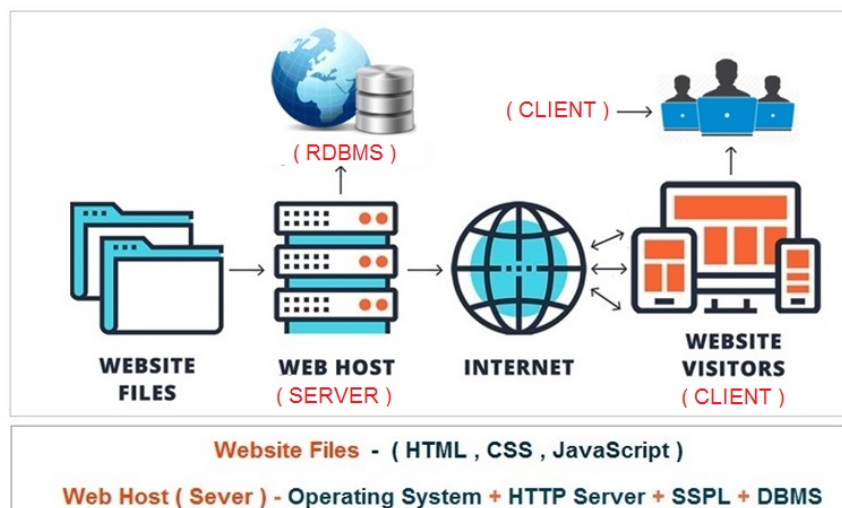


Figure 4: High-level diagram of a web application architecture [11]. RDBMS here stands for Relational Database Management System, which allows users to interact with the core database.

These components work together to achieve the following tasks:

1. Serve the frontend content to users on various web clients (desktops, phones, tables, etc.)
2. Handle web requests that come from these web clients to achieve various actions.
3. Fetch and persist data in the database.

With these fundamental components in place, any type of application can be built. As web applications scale, architectural upgrades may be necessary to keep performance and functionality at a level users expect. However, PolyFlowBuilder does not need these additional upgrades and is built with the standard frontend, backend, and database combination.

Frontend Component Frameworks

Interactive webpages became possible with the introduction of JavaScript in the mid-1990s. As website complexity increased and interactivity became more important, it was clear that websites would not scale well with developers writing plain scripts for their websites – this was the approach PolyFlowBuilder 1.0 took and is the reason why this rewrite effort is necessary.

This is when the idea of a “component framework” was introduced. These component frameworks allow the interactive portions of a webpage to be encapsulated into a single “component”, which acts as its own unit independent from other components. These components can be hierarchical in nature and allows developers to abstract high-level functionality very well.



Figure 5: Depiction of some of the top frontend JavaScript frameworks [12]. Note that jQuery is a standard JavaScript library instead of a component framework. All other frameworks pictured are frontend component frameworks.

The frontend of PolyFlowBuilder 2.0 uses a frontend component framework for these reasons. This enables the application to scale easily as the interfaces become more complex over time.

Backend Frameworks

There are a variety of backend frameworks that allow developers to create and run web servers that host APIs and other resources for the application frontend. Each framework is usually associated with a particular programming language, which in turn allows each of them to have their own ecosystem of tooling and software for developers to implement applications with.



Figure 6: Depiction of common backend frameworks used in the industry [13].

PolyFlowBuilder 1.0 uses ExpressJS, which is a backend framework written in JavaScript that runs on NodeJS, a backend JavaScript runtime. The advantage of JavaScript backend frameworks is that since the frontend is commonly written in JavaScript (and must eventually be converted to JavaScript to run in the browser), the entire application (minus the database) is using the same programming language, which unifies development efforts. Other frameworks, written in other languages, have their own ecosystems and integrations.

Web Application Frameworks

Usually, the frontend and backend are developed as two separate pieces of software to ship and are treated as such. However, there has recently been a shift to create frameworks that encompass both the frontend and backend development processes. These are called “web application frameworks” and unify the development experience substantially as the entire application is a single project instead of multiple smaller projects (for the frontend and backend). These include frameworks such as NextJS [14] (React [15]), GatsbyJS [16] (React [15]), NuxtJS [17] (Vue [18]), Angular [19] (Angular), SvelteKit [20] (Svelte [21]), and others.

PolyFlowBuilder 2.0 elected to use a full-stack web application framework instead of a separate frontend and backend system for this enhanced unity and improved developer experience.

Data Persistence

Arguably, the most important part of an application apart from its functionality is its data. Therefore, a robust data persistence system needs to be in place to handle the appropriate amount of traffic the application expects to see. These data persistence systems are usually databases.

There are many different types of databases depending on the types of data being stored/queried. The common types of databases can be seen in **Figure 7**.

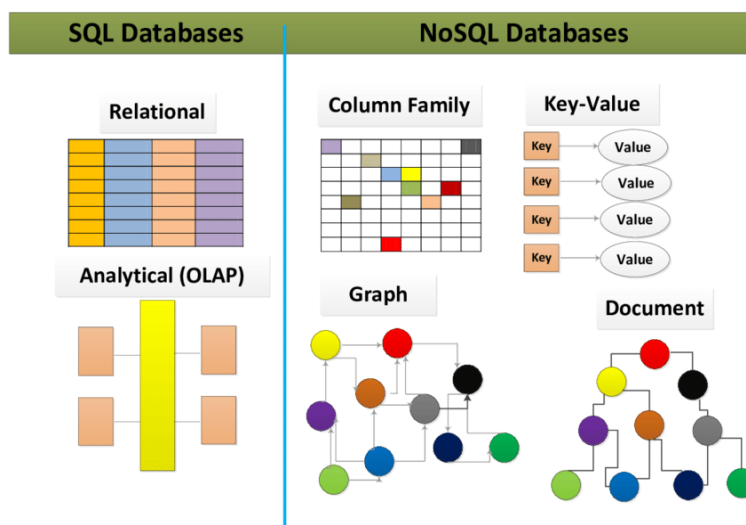


Figure 7: Common types of databases [22]. Databases are split into two categories: SQL databases, which are conventionally more structured and tabular in nature, and NoSQL databases, which are usually less structured and varied in shape.

The data used by PolyFlowBuilder is all tabular and relational in nature, so SQL databases are the natural choice.

Anatomy of a PolyFlowBuilder Flowchart

A flowchart is at the core of everything PolyFlowBuilder provides to users. Therefore, each flowchart contains a rich amount of information to encapsulate as much about the user's academic plan as possible. See **Table 2** for a high-level overview of the various pieces of metadata that make up a flowchart.

Table 2. Components of a PolyFlowBuilder flowchart.

Component	Single/Multiple	Allowed Values	Description/Notes
ID	Single	UUID	The unique identifier for this flowchart.
Owner ID	Single	UUID	The unique identifier for the user that created this flowchart.
Name	Single	Any string	Name of the flowchart.
Starting Year	Single	Year in YYYY format, starting from 2015	Starting year of the flowchart.
Programs	Multiple	Entries from list of available programs	The academic programs associated with the flowchart. These programs are

			<p>currently in the form of (catalog, major, concentration) tuples.</p> <p>Each major can only appear in a flowchart once, regardless of if the catalog or concentration are different.</p>
Notes	Single	Any string	User-defined notes about this flowchart.
Version	Single	Integer	Defines the version of the data model that the flowchart is using. The latest version, and the one associated with this rewrite, is the integer 7.
Term Data	Multiple	List of term information	<p>Each entry in the term data collection represents a term in the flowchart.</p> <p>Each term has the following pieces of information:</p> <ol style="list-style-type: none"> 1. Term index – uniquely identifies which term this is when combined with the starting year 2. Term units – the number of units in this term 3. Term courses – the individual courses contained in each term
Total unit count	Single	String	Records the total number of units in the flowchart. This is simply an aggregation of the units from each term.

IV. Application Design

This chapter details the various design aspects that went into choosing the tools and frameworks that would be used to build PolyFlowBuilder 2.0.

Design Tenets

This effort is being done with the end goal that this codebase should be production-grade and easy to maintain, as its future will eventually be in the hands of future Cal Poly students via a collaborative effort. Therefore, while making the design decisions that would dictate the direction and eventual success or failure of the project, I kept the following design tenets in mind:

1. **Code Quality and Robustness:** This codebase should not be “prototype quality” where spaghetti code is used to implement major features. All code should be developed while thinking about implications for the entire system. All new production-facing code must include tests to verify that the intended features are working (test-driven development), and all success and failure cases should be considered.
2. **Reliability:** All features implemented should be written to be reliable under any reasonable deployment scenario.
3. **Code and API Readability:** Going together with code quality, the written code and data flow throughout the application should be explicit and reasonably clear for other developers to understand.
4. **Performance:** Care should be taken to ensure that new code is reasonably optimized. With the speed of modern-day systems, however, the above three tenets take priority over this one (so code that is a marginally slower is acceptable if it is more readable/robust/reliable than other implementations).

Design Decisions

Before major development could begin, a variety of design decisions had to be made about the “tech stack” of the platform. These design decisions are documented in **Table 3**.

Table 3. Design decisions for the PolyFlowBuilder tech stack.

Application Component	Options Considered	Option Selected	Rationale
Application framework	NextJS [14] NuxtJS [17] SvelteKit [20]	SvelteKit [20]	This is the companion web framework for the frontend framework Svelte (see the next design decision). I am satisfied with the features it offers and its ease-of-use to create complex web applications with it. It is now also stable for

	Angular [19]		production use, whereas previously this was not the case.
Frontend component framework	React [15] Angular [19] Svelte [21] Vue [18]	Svelte [21]	Required by the selection of SvelteKit as the application framework. Svelte is a very easy frontend framework to pick up and has lots of “elegant” features built in. It is arguably more performant than other popular frameworks like React, Vue, or Angular due to Svelte’s lack of a browser runtime and shadow DOM.
Backend runtime	NodeJS [23]	NodeJS [23]	Required by the selection of SvelteKit as the application framework. NodeJS has been the backend runtime for PolyFlowBuilder since its inception, and it has scaled well. NodeJS is extremely popular due to not requiring knowledge of an additional programming language on the backend, and many developers are quite familiar with it. It is also quite performant and easy to setup for web applications like PolyFlowBuilder.
Primary Implementation Language	TypeScript [24], JavaScript [25]	TypeScript [24]	<p>Usually, the language that NodeJS and Svelte/SvelteKit use is JavaScript. However, since JavaScript is weakly typed, this is unfavorable for long-term reliability and robustness. Instead, TypeScript (which is strongly typed JavaScript) is the language chosen for the project so that type errors are caught sooner, and features can be built with more confidence.</p> <p>Another note to make is that the language(s) used for the frontend and backend are usually not the same. The explicit decision was made to have the frontend and backend language be the same for PolyFlowBuilder as it reduces the friction for onboarding and unifies the backend and frontend logic.</p>
Data Storage Type	Relational, Document	Relational	PolyFlowBuilder has always modeled its data in a relational fashion with tables because this model works well for the data that PolyFlowBuilder provides and consumes.

Database Engine	MySQL [8] PostgreSQL [9] MongoDB [10]	MySQL [8]	MySQL is a robust, free, and open-source RDBMS for storing relational data using SQL. It has been battle-tested and is more than adequate for the project requirements.
Database Communication Layer	Prisma [26], TypeORM [27], MikroORM [28]	Prisma [26]	<p>In the original version of PolyFlowBuilder, no object relational mapper was used to transform data stored in the SQL relational format to its JSON equivalent for application use – raw SQL queries and manual transforms were used.</p> <p>For this version of PolyFlowBuilder, reliability, robustness, and readability are preferred over immediate speed and performance, so the popular MySQL ORM, Prisma, was chosen to interact with the database. This allows for application logic to interact with the database in a much more object-oriented approach versus using issuing direct SQL queries to the database.</p>

Less significant design decisions were made throughout the course of the project, which will be detailed in their respective sections. The design decisions mentioned here are the ones that guided development for the entire project.

Application Architecture Design

Once the pertinent design decisions were made, the high-level architecture of the application was developed for the projected needs of the application. See **Figure 8** for a diagram of the application architecture.

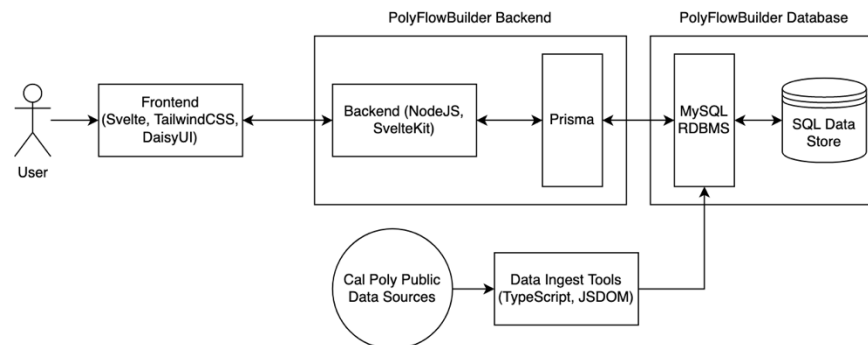


Figure 8: Architecture diagram of the PolyFlowBuilder application. It consists of a frontend, backend, database, and data ingest system for the Data API.

User Interface Design

One of the changes that will be the most apparent to users switching to the new version of PolyFlowBuilder are the user interface design language differences between the old and new versions. An example of the UI differences can be seen in **Figure 9**.

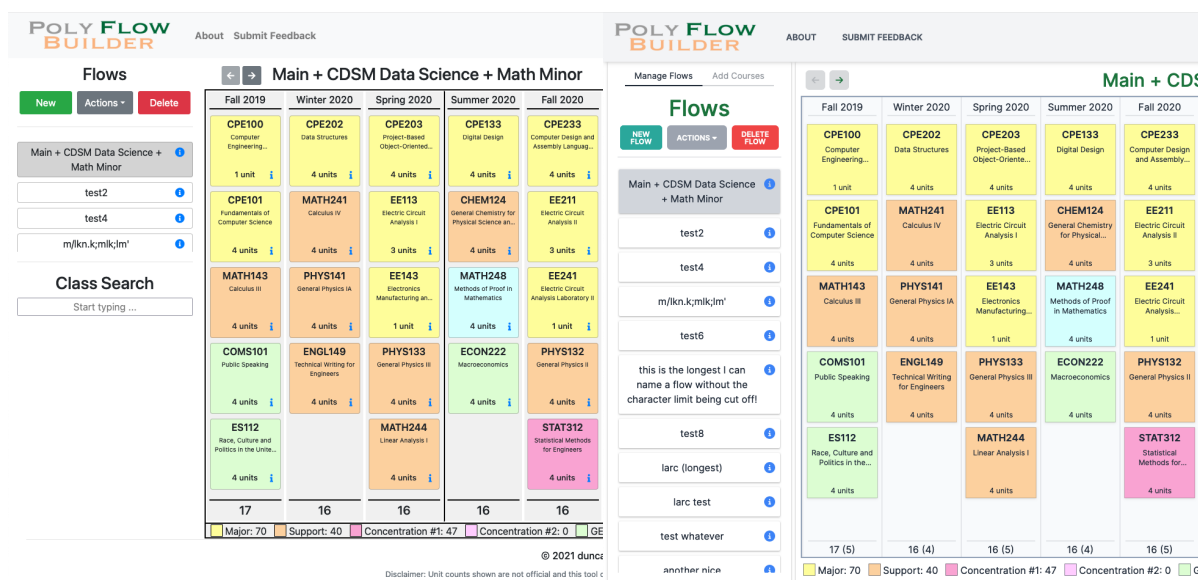


Figure 9: Side-by-side comparison of the existing UI (left) and the new UI (right) of the PolyFlowBuilder flowchart editor.

The differences in the UI/UX can be mainly attributed to a greater experience with web UI design as well as the use of UI component frameworks. In particular, the existing UI was built using the CSS library Bootstrap [29], whereas the new UI was built using the CSS library TailwindCSS [30] along with the TailwindCSS UI framework DaisyUI [31]. In both cases, custom styles were applied to the standard components to tailor to the exact UI that was required.

While using Bootstrap was simpler, the TailwindCSS + DaisyUI option selected for this rewrite allows for a variety of benefits, some of which are listed below:

1. It allows developers to create elegant interfaces easier, as one does not need to know specifics about user interface design (many of the nuances are abstracted away by these libraries).
2. The UI framework breaks up the interface pieces into components, which are all styled similarly and allow for a unified design language across the application.
3. The UI framework considers accessibility and responds to various accessibility features (such as “prefer reduced motion” with animations).
4. This option allows for responsive interfaces to be built easier. Responsive interfaces are ones that change to best accommodate the device that users access the application on.

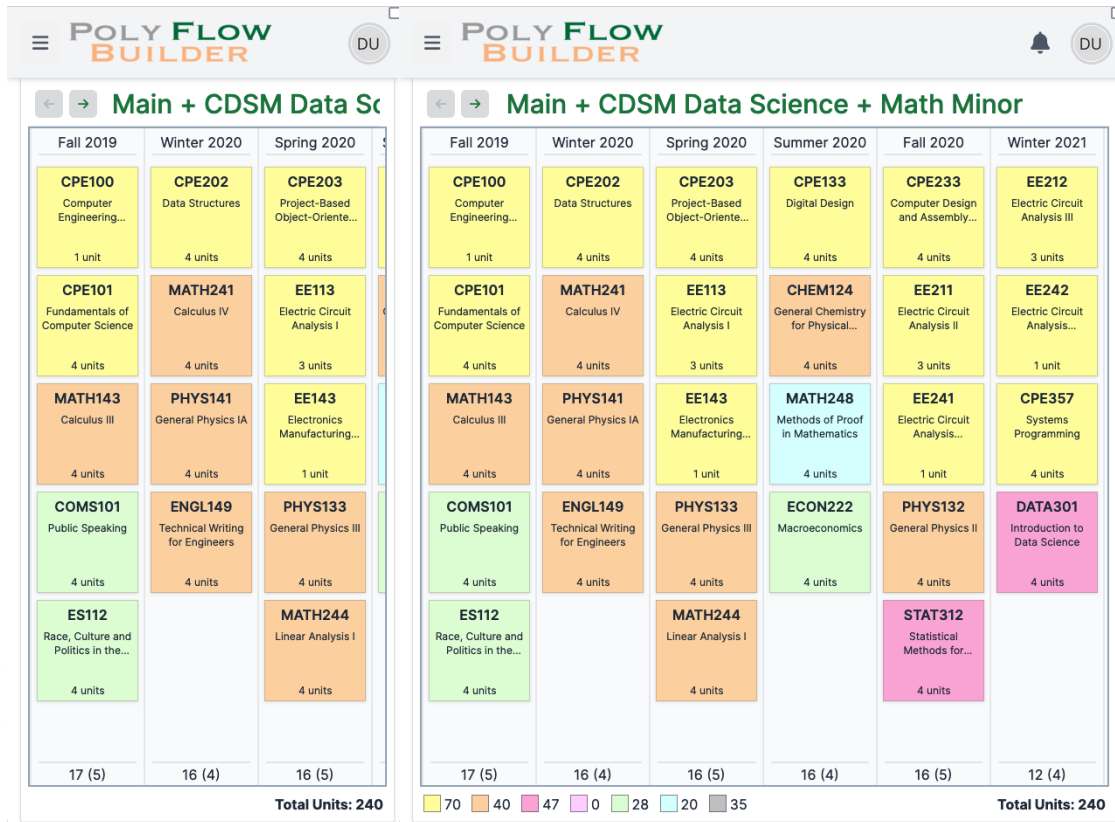


Figure 10: The main flowchart editor interface responding appropriately to different screen sizes. The interface on the left is meant for taller screens (e.g., phones), whereas the interface on the right is meant for screens that are larger than phones but smaller than a full desktop (e.g., tablets).

V. Application Interface

Application Interface Overview

The PolyFlowBuilder interface is a web application comprised of a collection of web pages.

These pages are documented in **Table 4**.

Table 4. List of webpages that make up the PolyFlowBuilder interface.

Page	URL	Access Control	Description
Homepage	/	Only viewable when signed out	The landing page when the user navigates to the PolyFlowBuilder website.
About	/about	None	This page describes what the project is, how it came to be, and some history behind the project.
Submit Feedback	/feedback	None	This page allows users to submit feature request, bug reports, and other feedback to the PolyFlowBuilder development team (currently just me) for review.
Register Account	/register	Only viewable when signed out	This page allows the user to create a new account on the PolyFlowBuilder platform.
Login to Account	/login	Only viewable when signed out	This page allows the user to log into their existing account on the PolyFlowBuilder platform.
Forgot Password	/forgotpassword	Only viewable when signed out	This page is where users navigate where they have forgotten their password. A password reset request is initiated here.
Reset Password	/resetpassword	Only viewable when signed out Only viewable if a valid token is presented	This page is inaccessible to users without an existing password reset request. If the user can view this page, the user can use it to reset their password.
Flowchart Editor	/flows	Only viewable when signed in	This page is where the main flowchart editor lives. Users can only access this page after authenticating on the login page.

Users can navigate between these pages by hyperlinks/buttons on other pages, or by directly entering them into the browser address bar. Some pages have access controls on them that allow

users to view them only if certain conditions are satisfied. For pages that are only viewable when signed out, users are redirected to the flowchart editor page if they are signed in. For pages that are only viewable when signed in, users are redirected to the login page if they are signed out.

See **Figure 11 – Figure 18** for how each of these pages look from the user perspective.

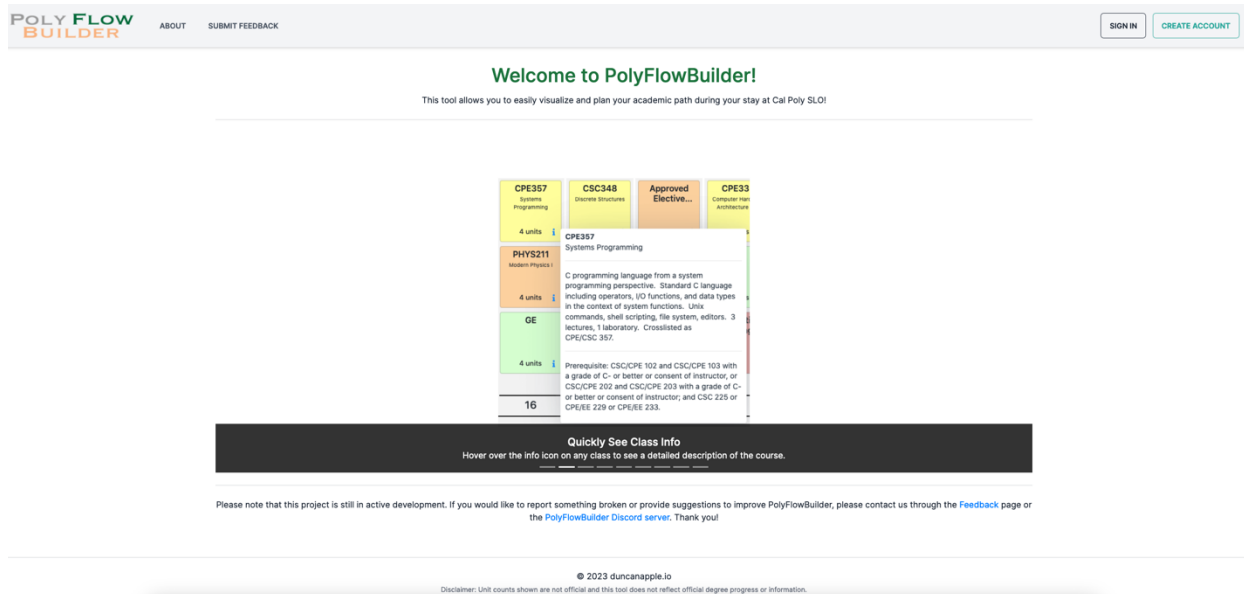


Figure 11: PolyFlowBuilder Home page. It includes a carousel of photos that represent features that PolyFlowBuilder provides to users, and links to PolyFlowBuilder-related resources.

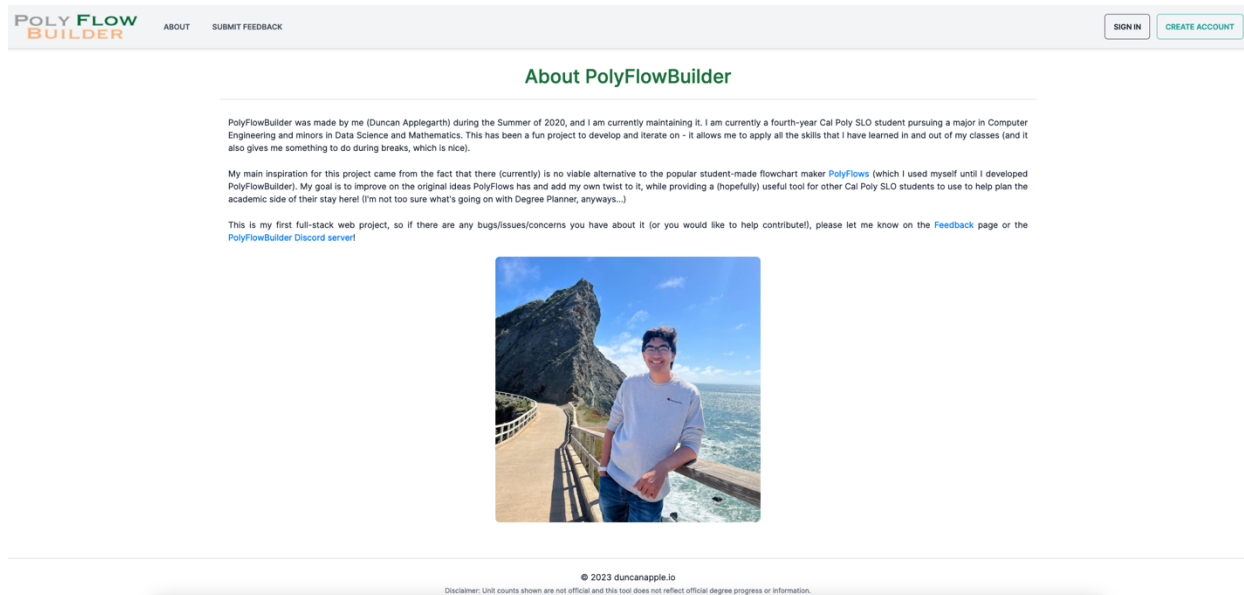


Figure 12: PolyFlowBuilder About page.

POLYFLOW BUILDER ABOUT SUBMIT FEEDBACK SIGN IN CREATE ACCOUNT

Submit Feedback

Thank you for taking the time to submit feedback to improve PolyFlowBuilder! Please fill out the following:

Subject

General Comment

Let us know what type of feedback you are submitting!
If you are submitting an issue, please make sure you have read the "Common Issues and Fixes" section within the PolyFlowBuilder CheatSheet (accessible via the blue question mark in the flow editor) before submitting a request.

Email

Email address

Let us know how to get in touch with you for a follow-up if necessary!

Feedback

Enter your feedback here!

SUBMIT FEEDBACK

© 2023 duncanapple.io
Disclaimer: Unit counts shown are not official and this tool does not reflect official degree progress or information.

Figure 13: PolyFlowBuilder Submit Feedback page. Users can provide a feedback subject and return e-mail address along with their feedback. Feedback submitted here is forwarded to the PolyFlowBuilder administrator’s e-mail for review and stored in the database for recordkeeping.

POLYFLOW BUILDER ABOUT SUBMIT FEEDBACK SIGN IN CREATE ACCOUNT

Sign In

Email address

Password

SIGN IN

[Forgot your password?](#) | [Create an account](#)

© 2023 duncanapple.io
Disclaimer: Unit counts shown are not official and this tool does not reflect official degree progress or information.

Figure 14: PolyFlowBuilder Login page.

POLYFLOW
BUILDER

ABOUT

SUBMIT FEEDBACK

SIGN INCREATE ACCOUNT

Create Account

Please fill out the following information to create a PolyFlowBuilder account.

Username

Email address

Password

Repeat Password

CREATE ACCOUNT

[Sign In](#)

© 2023 duncanapple.io

Disclaimer: Unit counts shown are not official and this tool does not reflect official degree progress or information.

Figure 15: PolyFlowBuilder Registration page.

POLYFLOW
BUILDER

ABOUT

SUBMIT FEEDBACK

SIGN INCREATE ACCOUNT

Request Password Reset

Enter your email address below. If we have it on file, we will send you an email to reset your password.

Email address

SUBMIT PASSWORD RESET REQUEST

[Sign In](#) | [Create an account](#)

© 2023 duncanapple.io

Disclaimer: Unit counts shown are not official and this tool does not reflect official degree progress or information.

Figure 16: PolyFlowBuilder Forgot Password page.

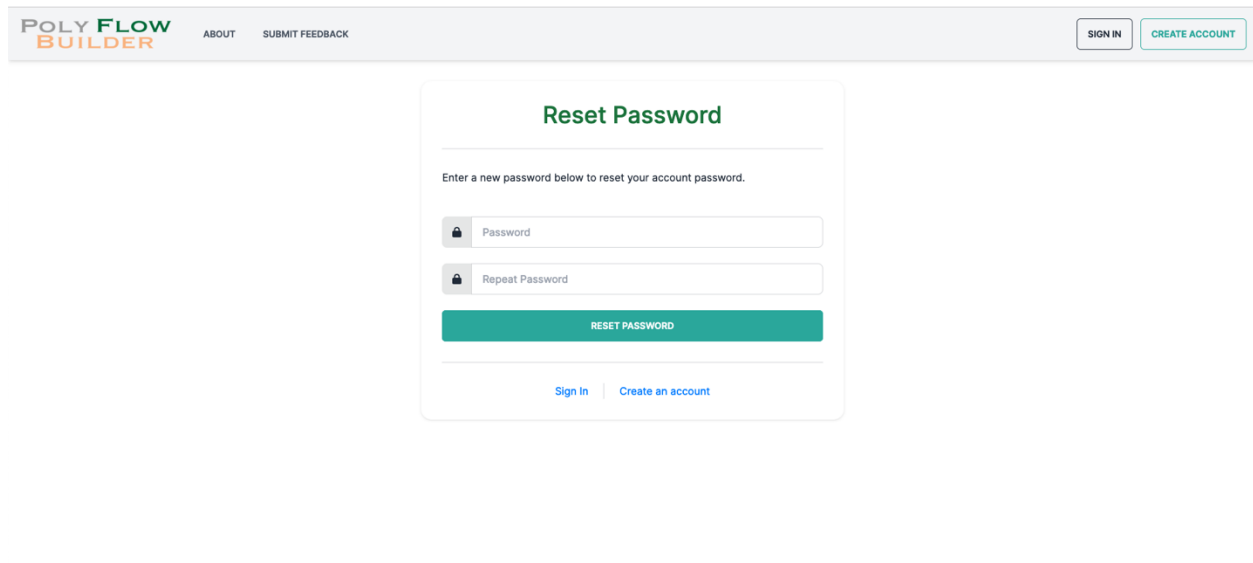


Figure 17: PolyFlowBuilder Reset Password page. This page can only be accessed if the user has a valid reset token (which comes from the reset password email the user receives).

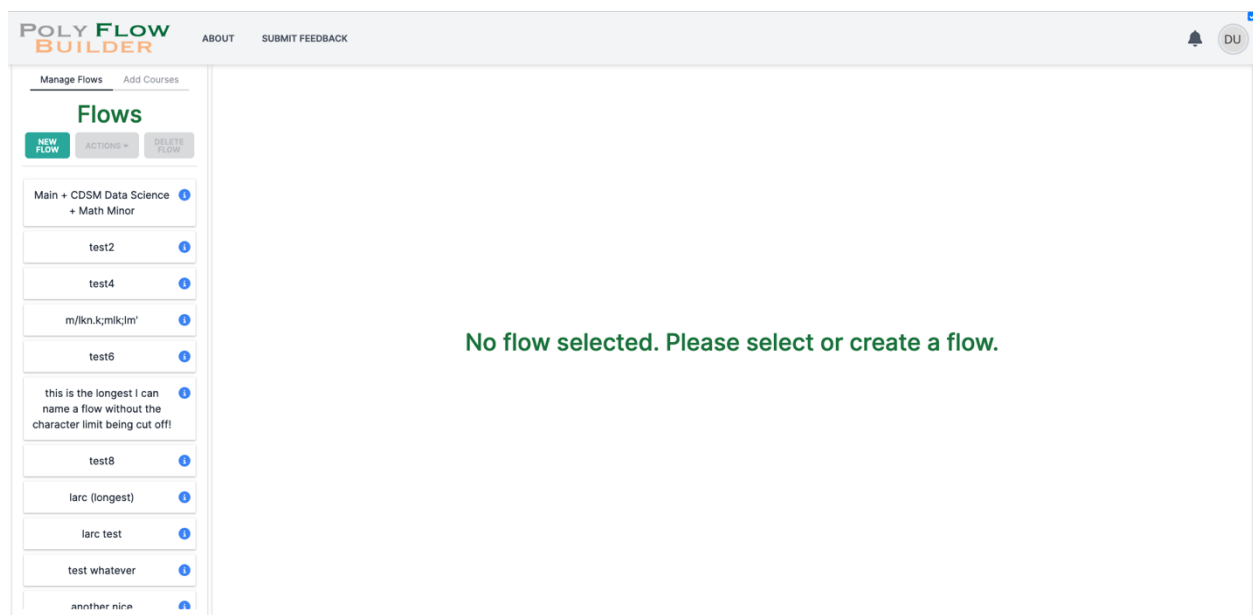


Figure 18: PolyFlowBuilder Flowchart Editor page. The flowcharts pictured here are from my personal PolyFlowBuilder account.

Flowchart Editor Interface

The flowchart editor is rich with features and information to allow users to customize their flowcharts effectively. The editor is made up of two major parts:

1. The flowchart info panel – this is the left pane of the UI where you can interact with and manipulate the flowchart being viewed.
2. The flowchart viewer – this is the portion of the UI where the flowchart is displayed.

These parts can be seen in **Figure 19**.

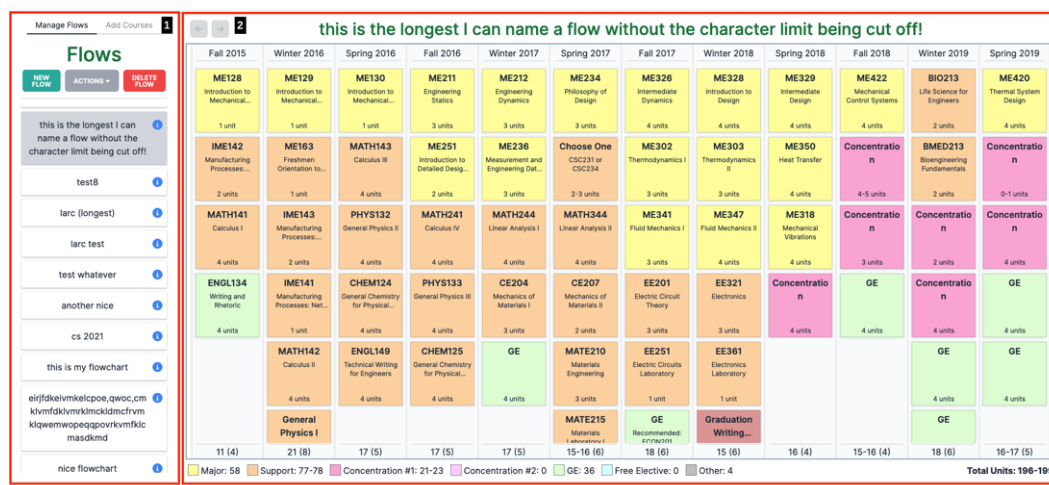


Figure 19: The flowchart editor broken down into its core components.

Flowchart Viewer Interface

See **Figure 20** and **Table 5** for a breakdown of the constituent pieces of the flowchart viewer and their purpose.

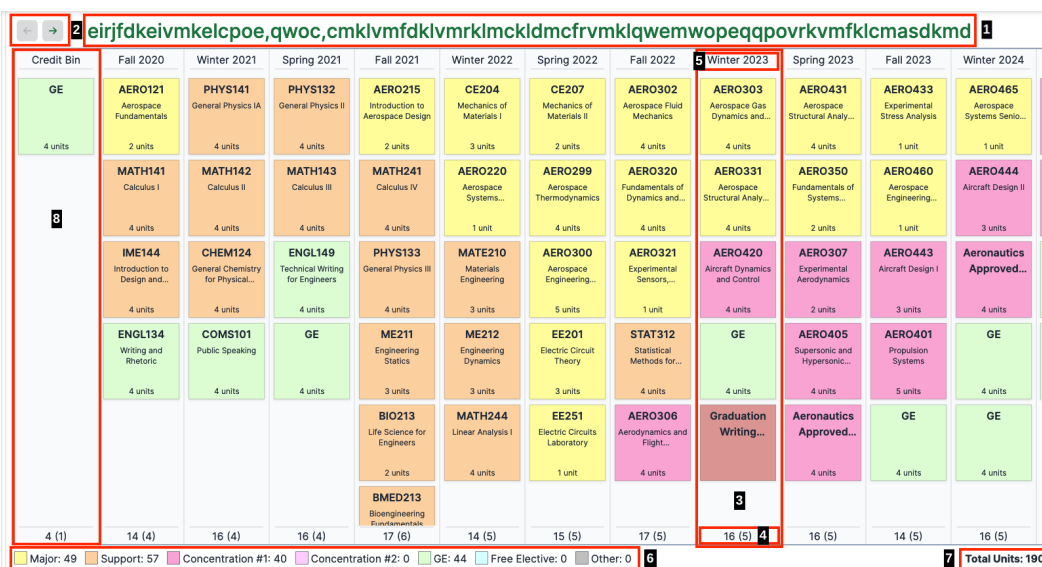


Figure 20: Flowchart viewer labeled by its constituent parts.

Table 5. Description of the various flowchart viewer components.

#	Component	Description
1	Name	Where the flowchart name is displayed.
2	Scroll buttons	Each scroll button is independently enabled when there is more content to the left/right of the current view. When the scroll button(s) are enabled, the user can press these buttons to scroll the flowchart horizontally in the left and right directions, respectively.
3	Term	The view for each term in the flowchart. Multiple terms are stacked horizontally next to each other as columns to create an intuitive chronological view of a user's academic progression.
4	Term unit count	The total number of units in the respective term. The number of courses in the term is also indicated by the number in parentheses.
5	Term name	The name of the respective term.
6	Unit counts (by category)	The number of units across the entire flowchart, grouped by category. Each category is a type of course that can be taken as part of someone's academic plan. These categories mirror the types present in the publicly available Cal Poly template flowcharts [1].
7	Total unit count	The total number of units in the entire flowchart.
8	Credit Bin	A special term that is meant to include courses that a user has already earned credit for before starting the academic plan shown in the flowchart. The visibility of this term can be toggled and is hidden if no courses are in it by default.

In the flowchart viewer, users can interact with each course in the displayed flowchart as follows:

1. Courses can be selected by clicking them (and deselected by clicking a second time).
2. Courses can be moved around in the flowchart by an intuitive drag-and-drop operation.
3. The user can view information about the course by hovering over it with their mouse.

See **Figure 21** for a visual of these three operations in a selected flowchart.

Fall 2020	Winter 2021	Spring 2021	Fall 2020	Winter 2021	Spring 2021	Fall 2021	Winter 2022	Spring 2022
AERO121 Aerospace Fundamentals 2 units	PHYS141 General Physics IA 4 units	PHYS132 General Physics II 4 units	AERO121 Aerospace Fundamentals 2 units	PHYS141 General Physics IA 4 units	PHYS132 General Physics II 4 units	AERO215 Introduction to Aerospace Design 2 units	CE204 Mechanics of Materials I 3 units	CE207 Mechanics of Materials II 2 units
MATH141 Calculus I 4 units	MATH142 Calculus II 4 units	MATH143 Calculus III 4 units	MATH141 Calculus I 4 units	MATH142 Calculus II 4 units	MATH142 Calculus II 4 units	MATH142 Calculus II 4 units Techniques of integration, applications to physics, transcendental functions. 4 lectures. Crosslisted as HNRS/MATH 142. Fulfills GE Area B4 (GE Area B1 for students on the 2019-20 or earlier catalogs); a grade of C- or better is required in one course in this GE area. 2020-21 or later catalog: GE Area B4 2019-20 or earlier catalog: GE Area B1 Prerequisite: MATH 141 with a grade of C- or better or consent of instructor. Terms Typically Offered (Dynamic): Summer, Fall, Winter, Spring		
IME144 Introduction to Design and... 4 units	CHEM124 General Chemistry for Physical... 4 units	ENGL149 Technical Writing for Engineers 4 units	IME144 Introduction to Design and... 4 units	CHEM124 General Chemistry for Physical... 4 units	CHEM124 General Chemistry for Physical... 4 units	AERO299 Aerospace Thermodynamics 4 units		
ENGL134 Writing and Rhetoric 4 units	COMS101 Public Speaking 4 units	GE 4 units	ENGL134 Writing and Rhetoric 4 units	COMS101 Public Speaking 4 units	COMS101 Public Speaking 4 units	AERO300 Aerospace Engineering... 5 units		
GE 4 units			ENGL134 Writing and Rhetoric 4 units	COMS101 Public Speaking 4 units	COMS101 Public Speaking 4 units	EE201 Electric Circuit Theory 3 units		
18 (5)	16 (4)	16 (4)	18 (5)	16 (4)	16 (4)	BIO213 Life Science for Engineers 2 units	MATH244 Linear Analysis I 4 units	EE251 Electric Circuits Laboratory 1 unit
						BMED213 Bioengineering Fundamentals 2 units		
						17 (6)	14 (5)	15 (5)

Figure 21: Left: A course being dragged around the selected flowchart. Right: The courses boxed in red are courses that are currently selected, and the user is hovering over the “MATH142” course with their mouse (not pictured) to reveal metadata about the highlighted course. This metadata includes course code, display name, description, additional metadata (GE information, prerequisites, etc.), and dynamic term typically offered information.

Flowchart Info Panel Interface

The flowchart info panel contains two tabs: the “Manage Flows” tab, and the “Add Courses” tab. The “Manage Flows” tab is the place where all operations on a user’s flowcharts are performed. The “Add Courses” tab is where the user can search for courses to add to the currently selected flowchart. See **Figure 22** for a visual of these tabs, **Table 6** for a description of the manage flows tab components, and **Table 7** for a description of the add courses tab components.

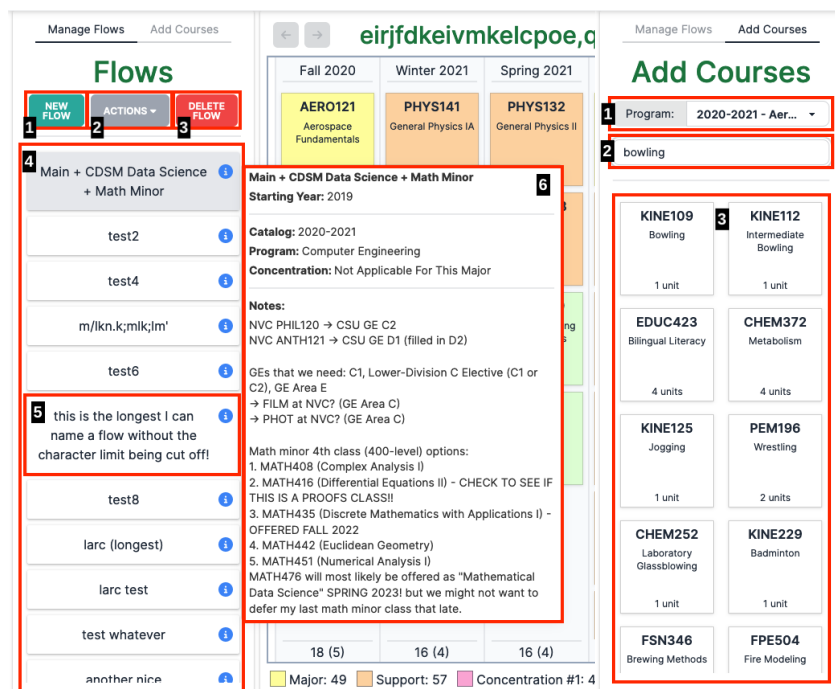


Figure 22: Left: the “Manage Flows” tab of the flowchart info panel annotated with its different components. Right: the “Add Courses” tab of the flowchart info panel annotated with its different components. This screenshot shows the search results for the query “bowling” in the 2020-2021 catalog, which is the associated catalog for the program in this flowchart.

Table 6. Components of the “Manage Flows” tab in the Flowchart Info Panel.

#	Component	Description
1	New Flow Button	Used to create a new flowchart. See the Flowchart Management section for more information.
2	Actions Button	Contains a dropdown with various actions that can be performed on a selected flowchart. See the Manipulating a Flowchart section for more information.
3	Delete Flow Button	Used to delete a selected flowchart. See the Flowchart Management section for more information.
4	Flowchart List	Area where all flowcharts created by this user are located.
5	Flowchart List Item	A single flowchart entry in the flowchart list. A user can do various things with this: <ol style="list-style-type: none"> 1. Select this flowchart by clicking on it. 2. Rearrange the flowchart list by drag-and-dropping this to another location in the flowchart list. 3. View flowchart information by hovering over the “information” icon in the bottom-right of this item (see entry #6 in this table).

6	Flowchart Information Tooltip	<p>A tooltip that displays the following flowchart information when activated:</p> <ol style="list-style-type: none"> 1. Name 2. Starting year 3. Academic program(s) 4. Notes
---	-------------------------------	--

Table 7. Components of the “Add Courses” tab in the Flowchart Info Panel.

#	Component	Description
1	Program Selector	Selector for which program the searched courses should be associated with. This selection also specifies which catalog the course search should occur with.
2	Search Query Box	Textbox for the search query.
3	Search Query Course Results	Area where the relevant courses to the search query are displayed. These courses can then be added to the selected flowchart by a drag-and-drop operation.

VI. PolyFlowBuilder Data API

One of the most important things PolyFlowBuilder needs to be useful to users is abundant, high-quality data related to all facets of academic scheduling. Unfortunately, acquiring this data is tedious and time consuming due to the lack of public Cal Poly APIs. This section details where the data was collected from, how it was collected, and how it was cleaned.

Data Sources

Over the course of development of the entire project (since Summer 2020), a variety of data has been scraped for PolyFlowBuilder use (accessible through the Data API). The rewritten version of PolyFlowBuilder uses the same dataset as the existing version. The sources for these data are detailed in **Table 8**.

Table 8. Data sources for PolyFlowBuilder's Data API.

Data Type	Source(s)	Notes
Valid Catalogs	n/a	This information was manually added to the Data API by including the catalogs that users were likely using at the time the original PolyFlowBuilder project started (≥ 2015 -2017).
Valid Start Years	n/a	This information was manually added to the Data API by including the starting years that users were likely using at the time the original PolyFlowBuilder project started (≥ 2015).
Academic Programs	Public Template Flowcharts [1]	The academic program information was scraped from here versus other publicly available sources because this source has the programs in the (catalog, major, concentration) format that PolyFlowBuilder uses.
Course Information	2015-2017 Course Catalog [32] 2017-2019 Course Catalog [33] 2019-2020 Course Catalog [34] 2020-2021 Course Catalog [35] 2021-2022 Course Catalog [36] 2022-2026 Course Catalog [3]	n/a
Course Requisite Data	From course information	This information was pulled from the existing course data that was scraped. Heavy data cleaning was required for these

		data (see the Data Cleaning Techniques section).
Course Category Information (GWR, USCP, GE, etc.)	From course information	This information was pulled from the existing course data that was scraped.
Term Typically Offered Data	Public Term Typically Offered Website [37]	n/a
Academic Program Template Flowcharts	Public Template Flowcharts [1]	This information was manually pulled from the PDFs made available on Cal Poly's website. Tedious data collection and cleaning was required (see the Data Collection Techniques and Data Cleaning Techniques sections).

Data Collection Techniques

To collect these data, several techniques were used:

1. HTML web scrapers: scripts were written to programmatically visit web pages and extract information from the HTML markup.
2. CSV parsing: scripts were written to download and parse CSV files.
3. Text scraping: scripts were written to pull relevant information from existing data sources.

Manual collection: data were collected by miscellaneous manual procedures.

See **Table 9** for the different ways data were collected.

Table 9. Data collection techniques for each data type.

Data Type	Collection Technique(s)	Notes
Valid Catalogs	Manual	n/a
Valid Start Years	Manual	n/a
Academic Programs	HTML web scraping	n/a
Course Information	HTML web scraping	n/a
Course Requisite Data	Text scraping	n/a
Course Category Information (GWR, USCP, GE, etc.)	Text scraping	n/a

Term Typically Offered Data	CSV parsing	n/a
Academic Program Template Flowcharts	HTML web scraping/manual	<p>The enumeration of all template flowcharts was done via HTML web scraping.</p> <p>The actual data collection for each template flowchart was done manually via a tedious process of inspecting and copying data from each template flowchart PDF into PolyFlowBuilder.</p>

Data Cleaning Techniques

In most cases, the collected raw data is not yet suitable for clean consumption by PolyFlowBuilder. Therefore, a variety of data cleaning techniques need to be employed to do things such as structuring the data and fixing anomalous observations. Due to the nature of these data, the data cleaning procedures were a combination of automated and manual techniques. See **Table 10** for how each data type was cleaned.

Table 10. Data cleaning techniques for each data type

Data Type	Cleaning Technique(s)	Notes
Valid Catalogs	n/a	Did not need to be cleaned
Valid Start Years	n/a	Did not need to be cleaned
Academic Programs	n/a	Did not need to be cleaned
Course Information	n/a	Did not need to be cleaned
Course Requisite Data	Automated/manual	<p>Several edge cases with these data were common enough to warrant automated scripts to correct them.</p> <p>For these data, tedious manual cleaning via inspection had to be done due to the large unstructured variation in textual input for course requisite information.</p>

Course Category Information (GWR, USCP, GE, etc.)	Automated/manual	Several edge cases with these data were common enough to warrant automated scripts to correct them. The remainder of these data were cleaned manually via inspection.
Term Typically Offered Data	n/a	Did not need to be cleaned
Academic Program Template Flowcharts	n/a	Did not need to be cleaned

Once these data are sourced, collected, and cleaned, they are ready to be consumed by the PolyFlowBuilder application.

VII. User Account Management

To allow PolyFlowBuilder to have personalized data for each user, a user account management system had to be implemented. This system encompasses all account-related operations someone can perform on the PolyFlowBuilder platform. The user interacts with this system through a variety of interfaces on the frontend, which in turn interacts with the user account management APIs. See **Table 11** for the components that make up the user account management system.

Table 11. Associated components for the user account management system.

Management Feature	Associated Webpages	Associated API Endpoints	Notes
Create a PolyFlowBuilder account	Register Account (/register)	/api/auth/register (POST)	Upon success, a new user record in the database is created. A user's password is hashed using the argon2 algorithm [38] before being stored for security.
Delete a PolyFlowBuilder account	Flowchart Editor (/flows)	/api/auth/login (DELETE)	Upon success, the requested user record is deleted in the database, along with all records associated with this user (e.g., flowcharts).
Log Into a PolyFlowBuilder Account	Login to Account (/login)	/api/auth/login (POST)	Upon success, a session record for this user is created in the database.
Log Out of a PolyFlowBuilder Account	Flowchart Editor (/flows)	/api/auth/login (DELETE)	Upon success, the associated session record for this user is deleted in the database.
Reset password for an existing PolyFlowBuilder account	Forgot Password (/forgotpassword) Reset Password (/resetpassword)	/api/auth/forgotpassword (POST) /api/auth/resetpassword (POST)	The reset password process has two steps: 1. Initiating the password reset request 2. Completing the password reset request

			Each step has its own associated page and API endpoint.
--	--	--	---

Create Account

Every user that wants to use PolyFlowBuilder to create customized flowcharts must first create an account on the platform. The process to create an account is described in **Figure 23**.

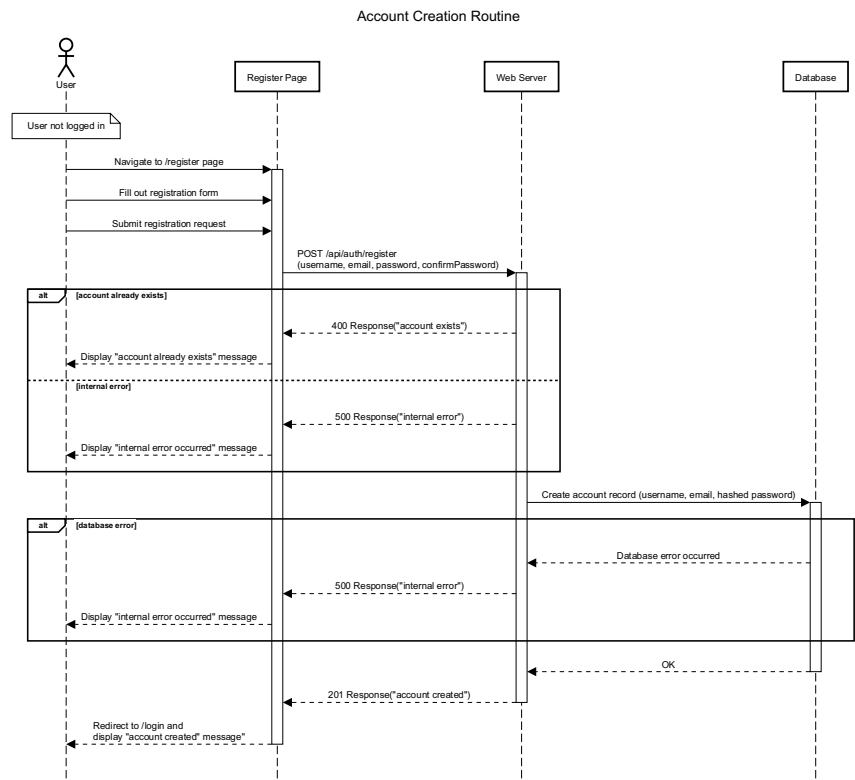


Figure 23: Sequence diagram for the account creation routine in PolyFlowBuilder. This diagram assumes the POST request sent to the web server from the frontend is formatted properly.

Delete Account

Users on the platform also can delete their account, along with all associated data, if they wish to. The process to delete an account is described in **Figure 24**.

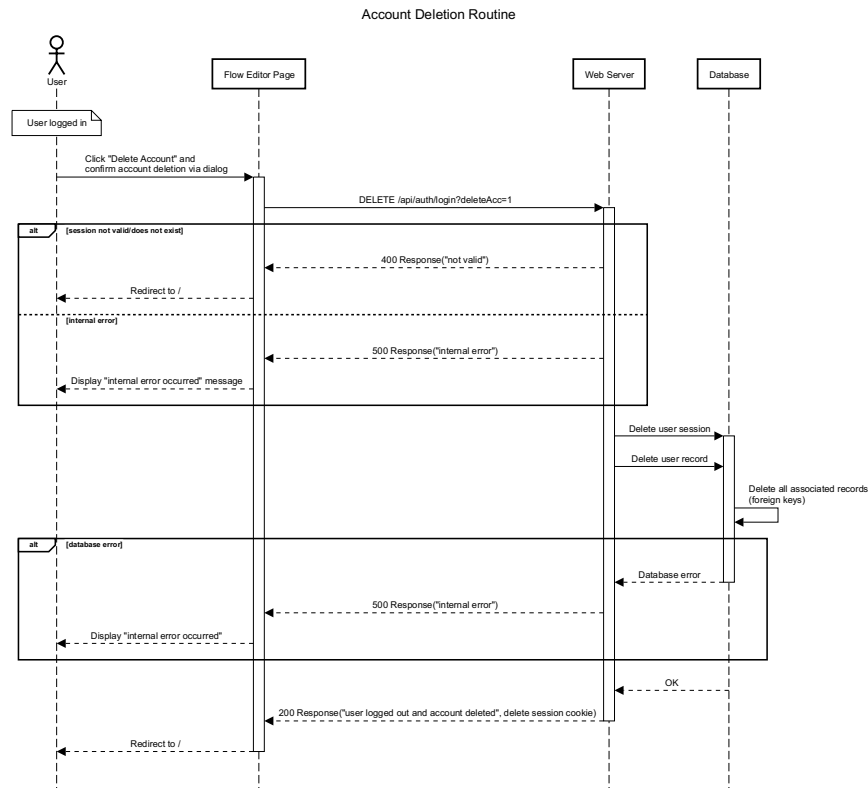


Figure 24: Sequence diagram for the account deletion routine in PolyFlowBuilder. This diagram assumes the DELETE request sent to the web server from the frontend is formatted properly and that the user is authenticated.

Login/Logout

For a user to use the account that they've created on the system, they need to be able to perform login and logout operations. **Figure 25** and **Figure 26** describe these two processes.

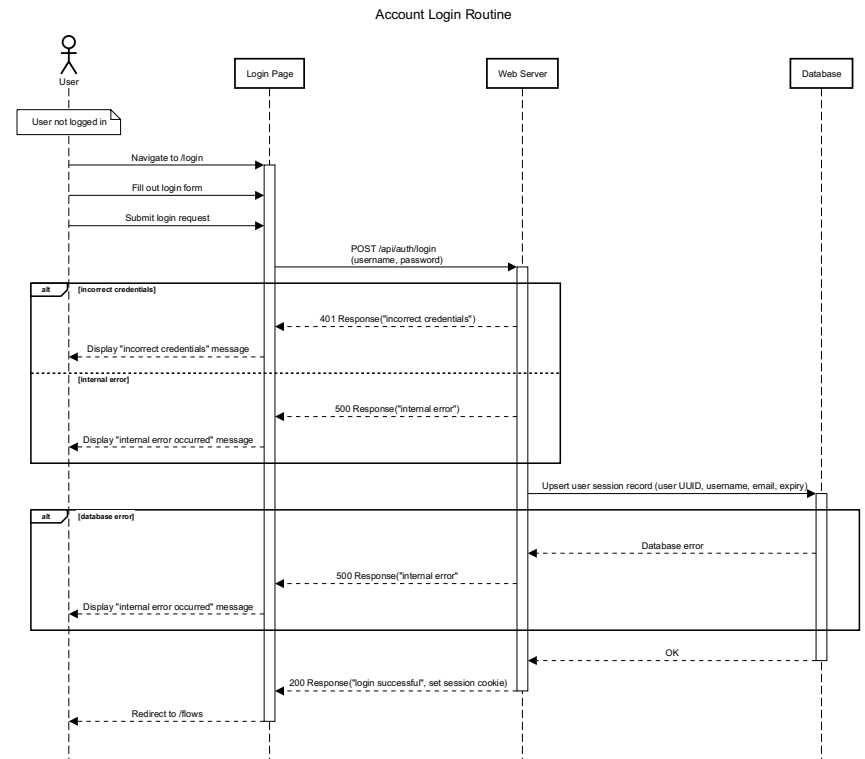


Figure 25: Sequence diagram for the login routine in PolyFlowBuilder. This diagram assumes the POST request sent to the web server from the frontend is formatted properly.

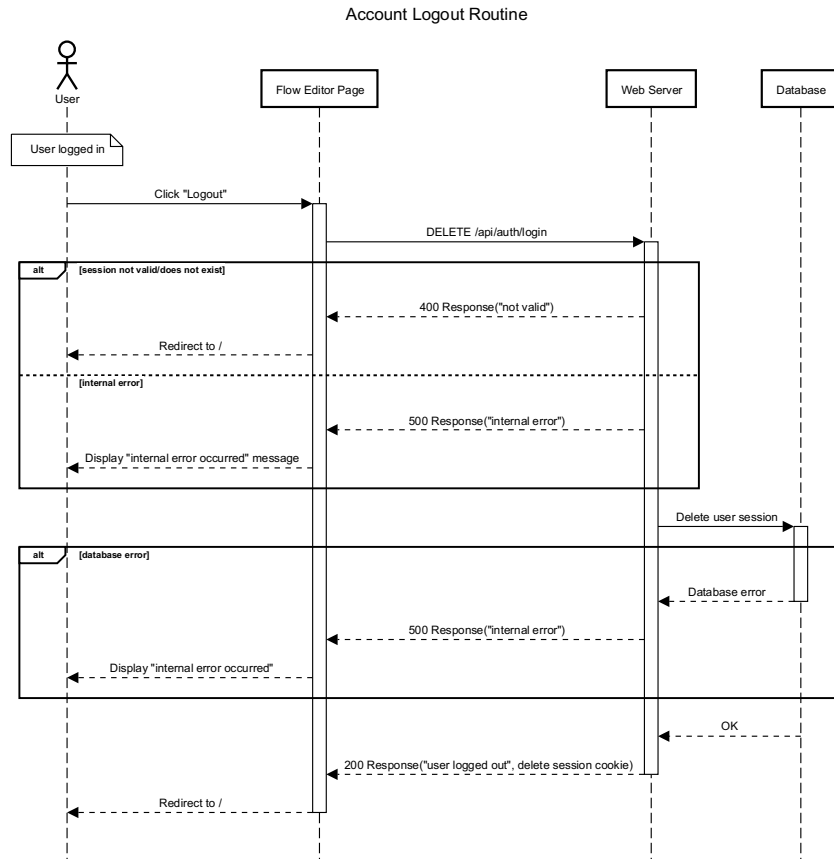


Figure 26: Sequence diagram for the logout routine in PolyFlowBuilder. This diagram assumes the DELETE request sent to the web server from the frontend is formatted properly and that the user is authenticated. Observe this routine is nearly identical to the account deletion routine seen in **Figure 24**.

Password Reset

If a user forgets the password to their account, they should be able to reset it in a secure and convenient manner. For PolyFlowBuilder, the password reset process looks as follows:

1. The user initiates a password reset by navigating to the appropriate page and submitting a request.
2. A password reset email gets sent to the user's email account with a unique password reset link. This link has a secure token associated with it to guarantee the user with access to the requested email account can reset the password.
3. The user opens the sent link, the token is verified, and they are taken to a page to enter a new password.
4. The user submits a new password, and their password is reset after the token is verified once more.
5. The user can now log into their account with the new password.

Figure 27 describes the first two steps of this process (initiating a password reset), and **Figure 28** describes the second two steps (completing the password reset).

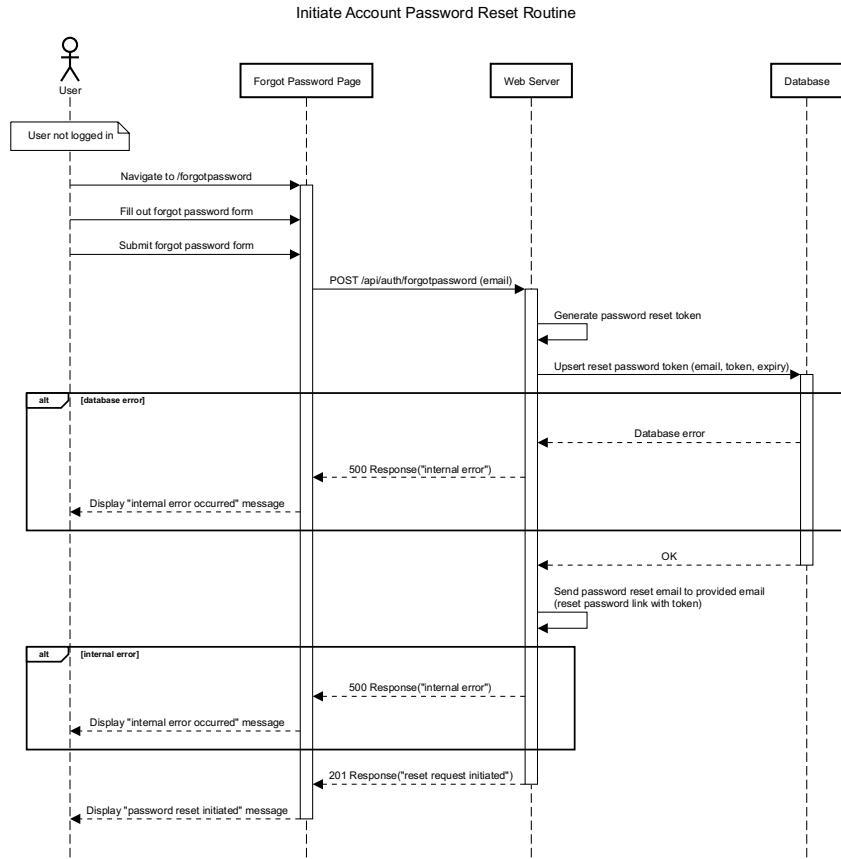


Figure 27: Sequence diagram for the initiate account password reset routine. This diagram assumes the API request is properly formatted.

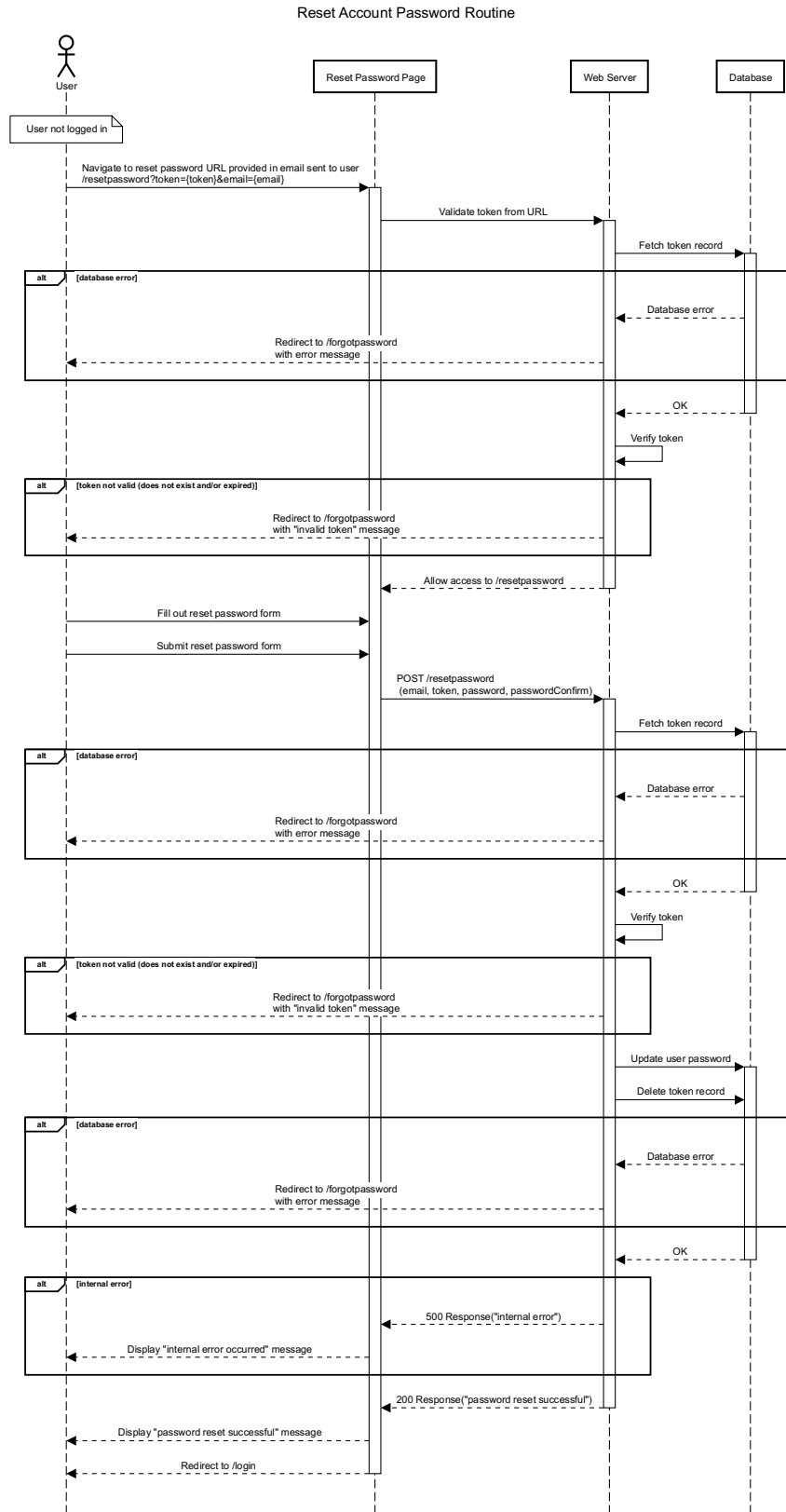


Figure 28: Sequence diagram for the reset account password routine. This diagram assumes the user has initiated a password reset and that the API requests are formatted properly. Observe the token is verified multiple times throughout this process.

VIII. Flowchart Management

To manage the flowcharts in account, the user can use the flowchart info panel controls. In particular, the user can:

1. Create a new flowchart.
2. Delete an existing flowchart.
3. Select a flowchart to edit.

This chapter describes these operations.

Create a New Flowchart

To create a new flowchart, the user clicks the “New Flow” button seen in **Figure 22**. This opens a modal to create a new flowchart, which is seen in **Figure 29**. See **Figure 30** for the corresponding sequence diagram.

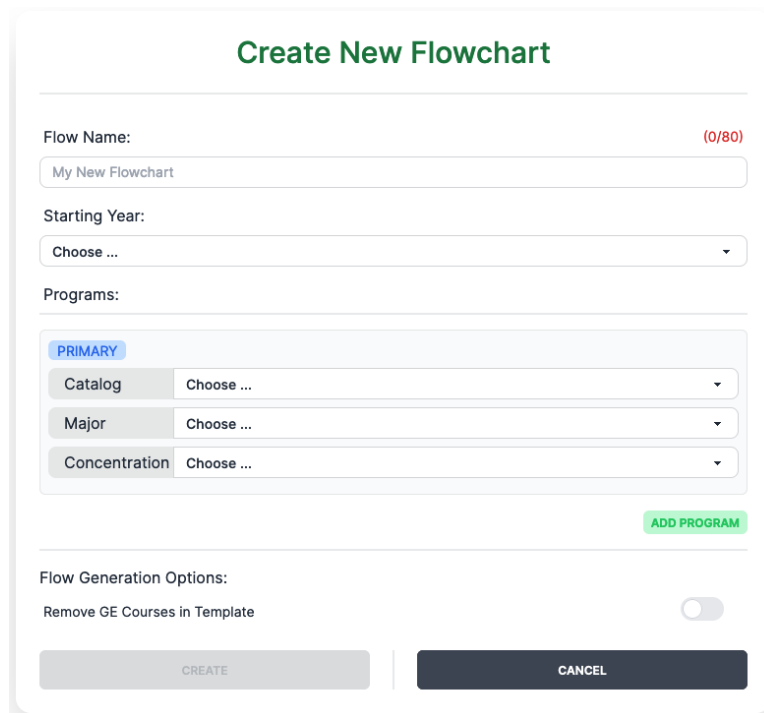
The image shows a modal window titled "Create New Flowchart" in green text. The modal has a white background with a light gray border. It contains several form fields: "Flow Name:" with a text input field containing "My New Flowchart" and a character count "(0/80)" in red; "Starting Year:" with a dropdown menu showing "Choose ..."; "Programs:" with a section header and three rows of dropdown menus labeled "Catalog", "Major", and "Concentration", each with a "Choose ..." option. There is a blue "PRIMARY" button above the first dropdown. Below the dropdowns is a green "ADD PROGRAM" button. At the bottom, there is a "Flow Generation Options:" section with a toggle switch for "Remove GE Courses in Template". At the very bottom are two buttons: "CREATE" (gray) and "CANCEL" (dark blue).

Figure 29: Modal to create a new flowchart. The user needs to specify the flowchart name, starting year, academic program(s), and a variety of generation options (the only one that exists currently is “remove GE courses in template”). The “Create” button will turn a solid green only if the submitted values are valid.

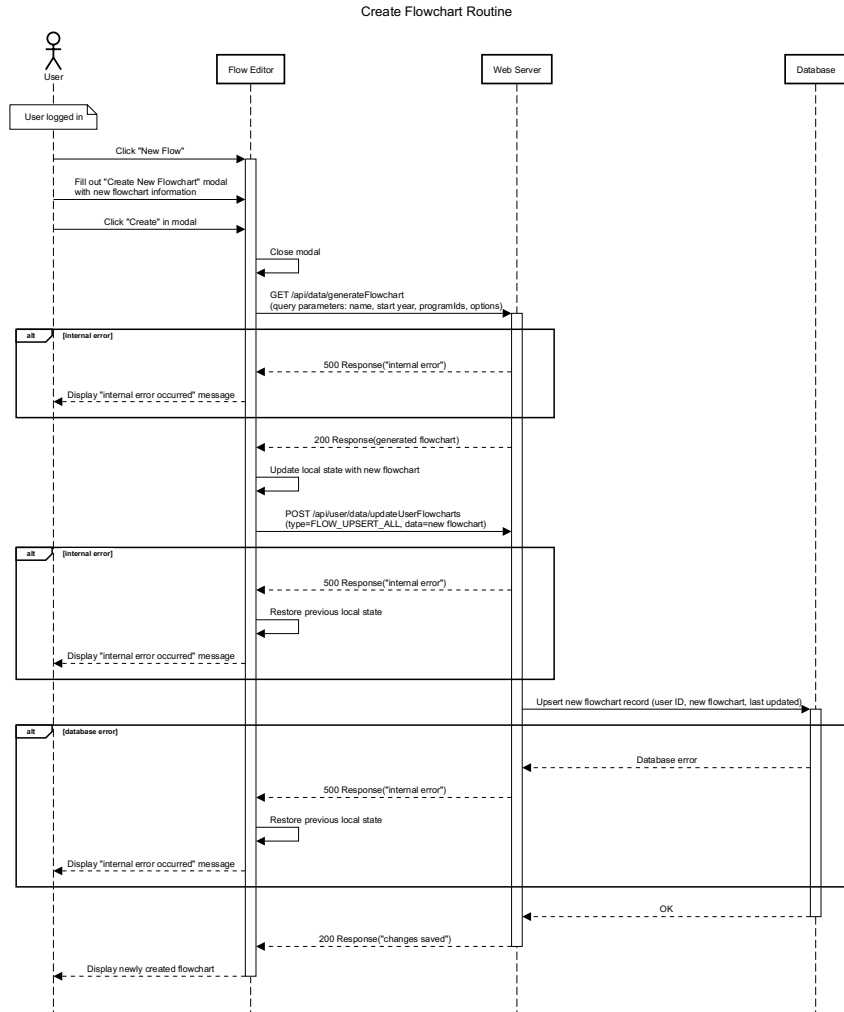


Figure 30: sequence diagram for the routine to create a new flowchart. This diagram assumes the GET and POST requests to the web server are correct from the frontend and that the user is authenticated.

Deleting an Existing Flowchart

To delete an existing flowchart, the user selects a flowchart from the flowchart list and clicks the “Delete Flow” button (see **Figure 22**). After a confirmation, the selected flowchart will be deleted permanently. See **Figure 31** for the corresponding sequence diagram.

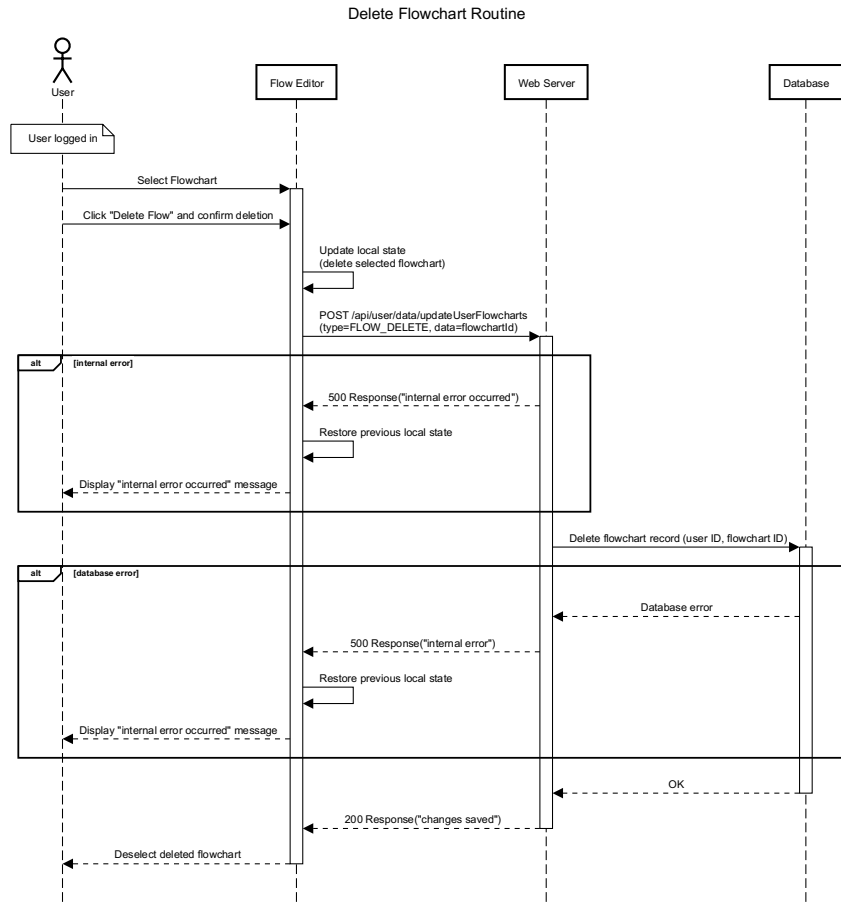


Figure 31: Sequence diagram for the routine to delete an existing flowchart. This diagram assumes the POST request to the web server is correct from the frontend and that the user is authenticated.

Select a Flowchart to Edit

To load a flowchart into the flow editor to view and manipulate, the user simply needs to click on the desired flowchart in the flowchart list (see **Figure 22**). Changes to a particular flowchart are described in the **Flowchart Manipulation** chapter.

IX. Flowchart Manipulation

Once a particular flowchart has been loaded into the flowchart editor, there are a myriad of actions a user can perform on it to modify its contents. These actions are:

1. Modifying the terms in the flowchart
2. Modifying flowchart metadata
3. Modifying flowchart course content
 - a. Viewing credit bin
 - b. Adding new course(s)
 - c. Deleting course(s)
 - d. Changing colors of course(s)
 - e. Modifying content of course(s)

These actions can be accessed by viewing the “Actions dropdown” in the Flowchart Info Panel, seen in **Figure 32**. The remainder of this section will describe these flowchart manipulation actions.

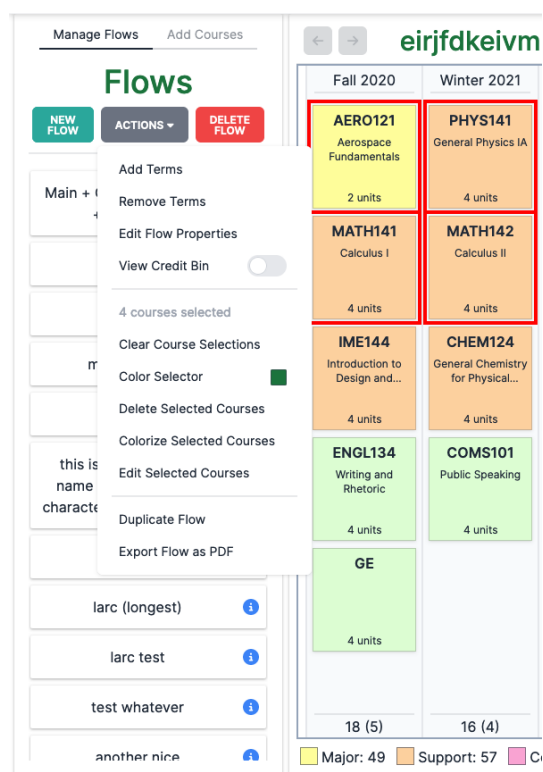


Figure 32: The flowchart Actions dropdown, which is split into three sections: the flowchart metadata section (top), the course manipulation (middle), and the flowchart utilities section (bottom). The flowchart utilities are described in the **Flowchart Utilities** chapter.

Modify Flowchart Terms

When a flowchart is first created, the number of terms in the flowchart are fixed and are 4-5 years with 3 terms each (Fall, Winter, Spring quarters), depending on the academic program(s) selected. If a user wants to modify the number of terms in the flowchart, they can do so with the “Add Terms” and “Remove Terms” actions (see **Figure 32**).

To add terms, a modal is displayed with the terms that can be added to the flowchart, seen in **Figure 33**. See **Figure 34** for the corresponding sequence diagram.

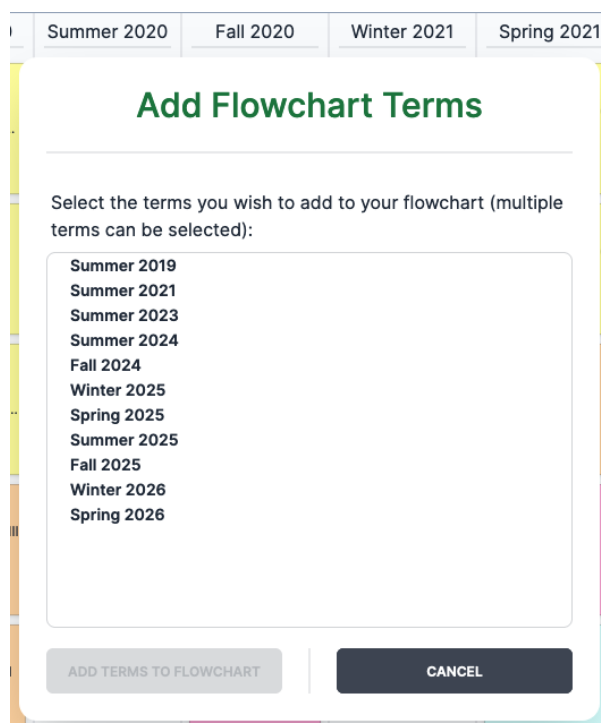


Figure 33: Modal to add new terms to a selected flowchart. Multiple terms can be selected simultaneously, and only the terms that can be added are shown in the selector. A user must select at least one term to add. In this version of PolyFlowBuilder, the latest term a user can add is the Spring term seven years after the flowchart’s starting year (so a flowchart can have up to 28 terms, not including the credit bin).

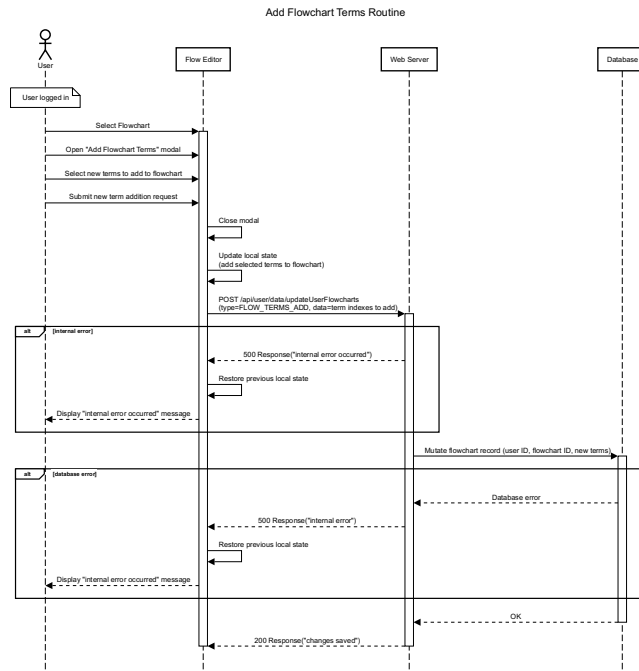


Figure 34: Sequence diagram for the routine to add terms to an existing flowchart. This diagram assumes the POST request to the web server is correct from the frontend and that the user is authenticated.

Similarly, to remove terms, a modal is displayed with the terms that can be removed from the flowchart, seen in **Figure 35**. See **Figure 36** for the corresponding sequence diagram.

Summer 2020
Fall 2020
Winter 2021
Spring 2021

Remove Flowchart Terms

Select the terms you wish to remove to your flowchart (multiple terms can be selected):

Fall 2019

Winter 2020

Spring 2020

Summer 2020

Fall 2020

Winter 2021

Spring 2021

Fall 2021

Winter 2022

Spring 2022

Summer 2022

Fall 2022

Winter 2023

Spring 2023

Fall 2023

REMOVE TERMS FROM FLOWCHART

CANCEL

Figure 35: Modal to remove existing terms from a selected flowchart. Multiple terms can be selected simultaneously, and only the terms that can be removed are shown in the selector. A user must select at least one term to remove.

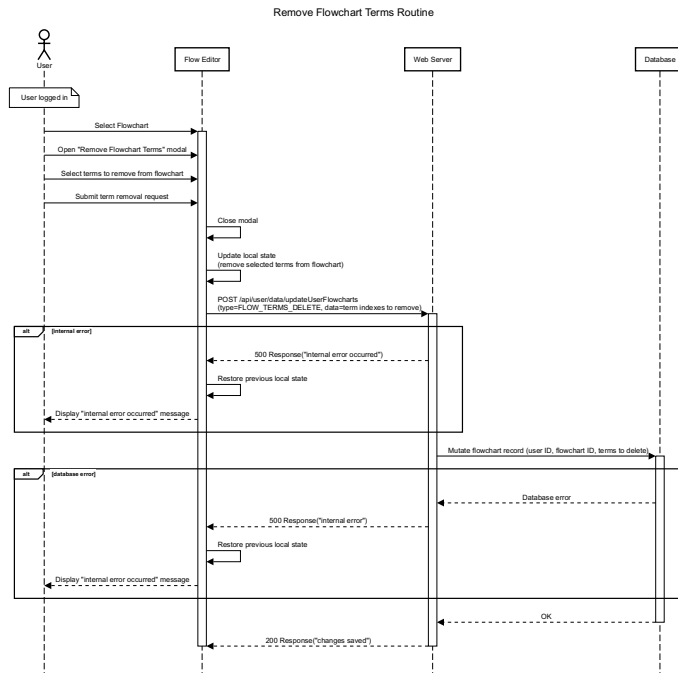


Figure 36: Sequence diagram for the routine to remove existing terms from a flowchart. This diagram assumes the POST request to the web server is correct from the frontend and that the user is authenticated.

Modify Flowchart Metadata

A user may want to modify certain pieces of metadata in their flowchart as it develops. To do this, they can use the “Edit Flow Properties” action (see **Figure 32**) on a selected flowchart. This action allows users to modify the following pieces of flowchart metadata:

1. Name
2. Starting year
3. Academic program(s)
4. Notes (likely the most common piece of metadata to change)

To edit these properties, a modal is displayed that allows the user to visualize and/or change these properties. This modal can be seen in **Figure 37**. See **Figure 38** for the corresponding sequence diagram.

Edit Flowchart Properties

Flow Name: (37/80)

Main + CDSM Data Science + Math Minor

Starting Year:

2019

Programs:

PRIMARY

Catalog

2020-2021

Major

Computer Engineering

Concentration

Not Applicable For This Major

ADD PROGRAM

Flow Notes: (635/1000)

NVC PHIL120 → CSU GE C2

NVC ANTH121 → CSU GE D1 (filled in D2)

GEs that we need: C1, Lower-Division C Elective (C1 or C2), GE Area E

→ FILM at NVC? (GE Area C)

SAVE CHANGES

CANCEL

Figure 37: Modal to view and/or edit user-accessible pieces of flowchart metadata. The user will only be able to save changes if a) there were changes made, and b) the changes are valid. Note that at the time of writing, the ability to modify the academic program(s) in the flowchart is not yet implemented (see the **Evaluation** chapter).

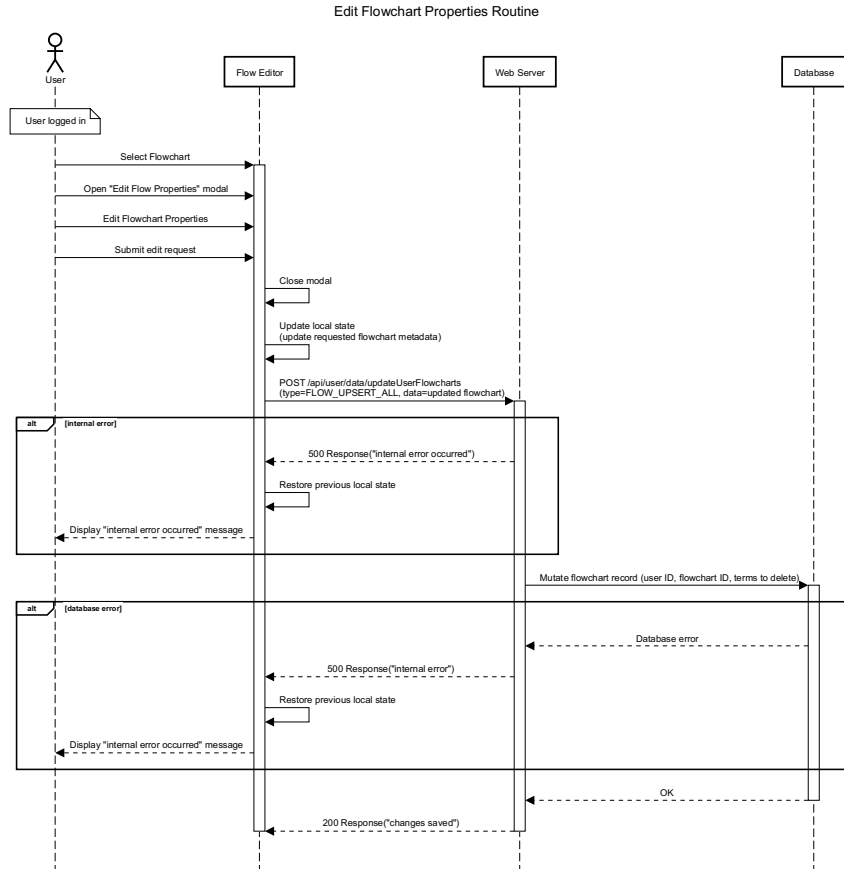


Figure 38: Sequence diagram for the routine to edit the metadata of a selected flowchart. This diagram assumes the POST request to the web server is correct from the frontend and that the user is authenticated. Observe that a `FLOW_UPSERT_ALL` update type is used versus something more specific that only includes fields for the updated properties. This is because the ability to change academic program(s) in a flowchart can mutate the entire flowchart.

Modify Flowchart Course Content

The most common thing that a user will change about their flowcharts are the courses that reside within it. There are several ways a user can modify a flowchart's course content:

1. Move course(s)
2. Add new course(s)
3. Select Course(s) to Modify
4. Delete course(s)
5. Change color of course(s)
6. Modify content of course(s)

These actions are described in the following subsections.

Move Course(s)

To move courses around in a flowchart, the user simply drags a course from its current position and drops it into its new position. If the user wants to move a course to the special Credit Bin term (which is meant to contain courses that the user has received credit for outside of Cal Poly), the Credit Bin must be visible in the flowchart viewer (the toggle can be seen in the Actions dropdown in **Figure 32**).

Add New Course(s)

To add new courses to a flowchart, a user must a) search for the courses to add, and b) drag-and-drop a course from the search results into the flowchart in a manner described in the previous subsection. See **Figure 22** for the course search interface, and **Figure 39** for the corresponding course search sequence diagram.

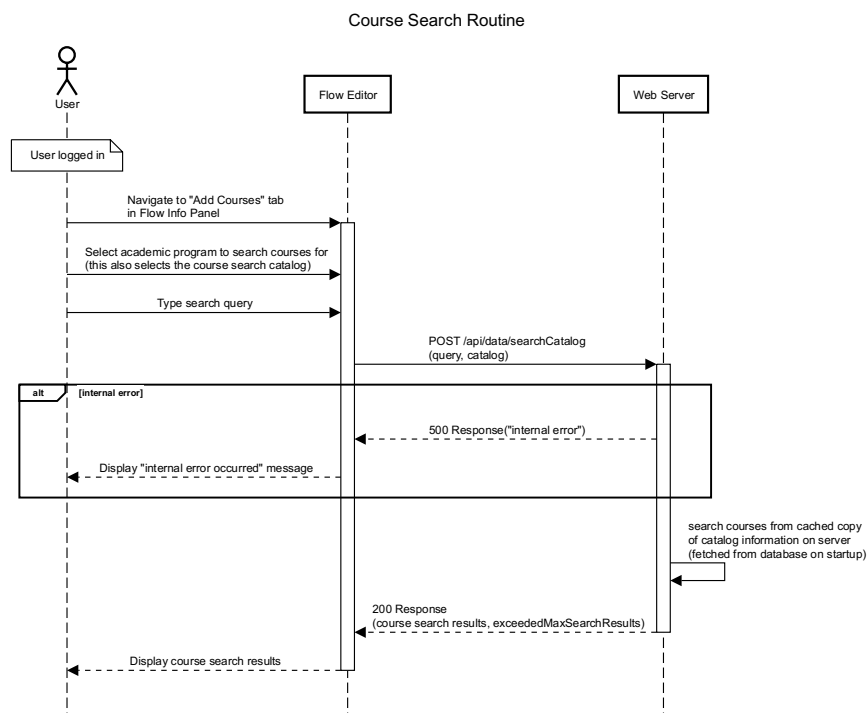


Figure 39: Sequence diagram for the routine to search for courses to add to a selected flowchart. This diagram assumes the POST request to the web server is correct from the frontend and that the user is authenticated. Observe that a copy of the course API data is cached on the web server at startup. This was a legacy design decision from the existing PolyFlowBuilder project to decrease search latency, but this will be redone to use an intermediary caching layer (e.g., Redis [39]) in a future update to ensure scalability.

Course Selection(s)

To delete, change the color, or customize the content of courses, the user must select one or more courses to change. See **Figure 21** for how to select courses, and **Figure 32** for the available actions that can be performed once courses are selected.

Delete Course(s)

To delete selected courses, the user simply clicks “Delete Selected Courses” in the Actions dropdown (see **Figure 32**). The course selections are cleared after this change has been persisted. See **Persisting Term Changes** for how this change is persisted.

Change Color of Course(s)

To change the color of selected courses, the user must first pick the new color of the selected courses by opening the Color Selector UI (see **Figure 40**), and then click “Colorize Selected Courses” in the Actions dropdown. The course selections are not cleared after this change has been persisted. See **Persisting Term Changes** for how this change is persisted.

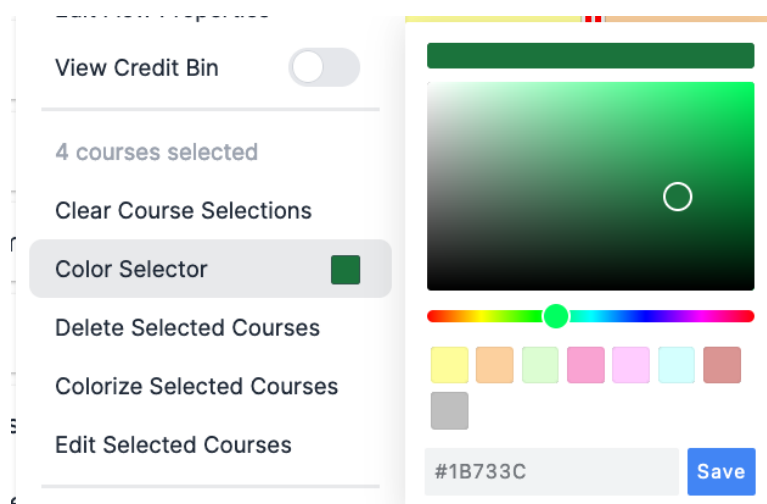


Figure 40: Color Selector UI in the Actions dropdown. A user can pick a color by selecting one of the preset colors (from left to right: major, support, general education, concentration #1, concentration #2, free elective, GWR, and other color categories), using the swatches and color picker, or by entering a #RRGGBB hex code for a color. The user then must click “Save” to save this color choice. The current color is shown in the box on the right of the “Color Selector” option in the Actions dropdown.

Modify Course(s) Contents

To modify the content of selected courses, the user must click the “Edit Selected Courses” option in the Actions dropdown (see **Figure 32**). This will then open the Customize Courses modal, which can be seen in **Figure 41**. See **Persisting Term Changes** for how this change is persisted.

The image shows two side-by-side screenshots of a software interface for customizing courses. The left screenshot is titled 'Customize Course' and shows fields for a single course: Course Type (Standard), Course Name (CPE101), Course Display Name (Fundamentals of Computer Science), Course Description (Basic principles of algorithmic problem solving...), and Course Units (4). The right screenshot is titled 'Customize Courses' and shows fields for multiple courses: Courses Type (Varied), Course Name (Multiple courses selected), Course Display Name (My Course Display Name), Course Description (Changes made here will apply to all selected courses.), and Course Units (My Course Units). Both modals have 'SAVE CHANGES' and 'CANCEL' buttons at the bottom.

Figure 41: The customize course modal. The Course Type specifies the type of course(s) that are selected: “Standard” means all selected courses are non-custom, “Custom” (not pictured) means all selected courses are custom, and “Varied” means there is a mix of Standard and Custom courses in the set of selected courses. If only one course is selected, the “Course Name”, “Course Display Name”, “Course Description”, and “Course Units” fields will populate with data from that course. If more than one course is selected, these fields will be a mix of empty and notes that there are multiple courses selected. The user will only be able to save course changes if a) changes were made, and b) the changes are valid.

Persisting Term Changes

To reduce the amount of data update types that had to be built out, all changes to a flowchart that change the contents of a particular term (e.g., changes to any courses in a term) are included in a single type of update, aptly named a TERM_MOD update. See **Figure 42** for the sequence diagram that corresponds to this type of update.

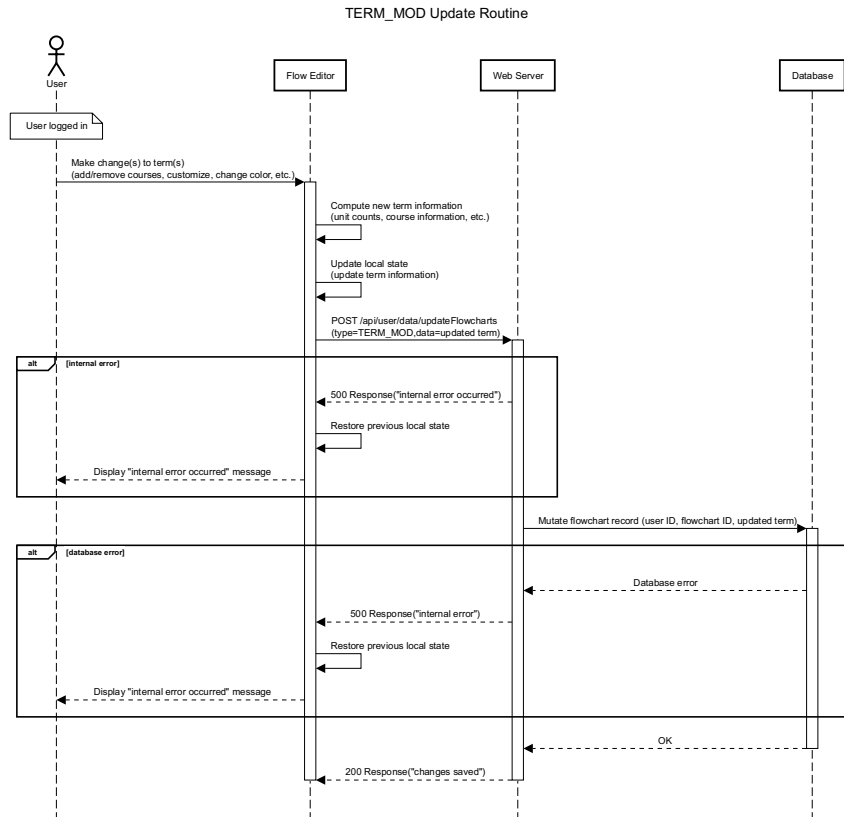


Figure 42: Sequence diagram for the routine to update term information for a selected flowchart. This diagram assumes the POST request to the web server is correct from the frontend and that the user is authenticated.

X. Flowchart Utilities

For any selected flowchart, there are utilities that exist to perform additional non-manipulative functionality. These utilities are:

1. Duplicate flowchart
2. Export flowchart as PDF

This chapter will describe these utilities and how they work.

Duplicate Flowchart

The duplicate flowchart utility is quite simple but powerful for creating snapshots of academic plans to compare against. When a user clicks “Duplicate Flow” in the Actions dropdown (see **Figure 32**), a flowchart is created at the bottom of the user’s flowchart list that is a duplicate of the original flowchart with the following changes:

1. The new flowchart name is “Copy of {original flowchart name}”
2. The new flowchart has a unique ID (does not share flowchart IDs with the original flowchart)

See **Figure 43** for a sequence diagram of this process.

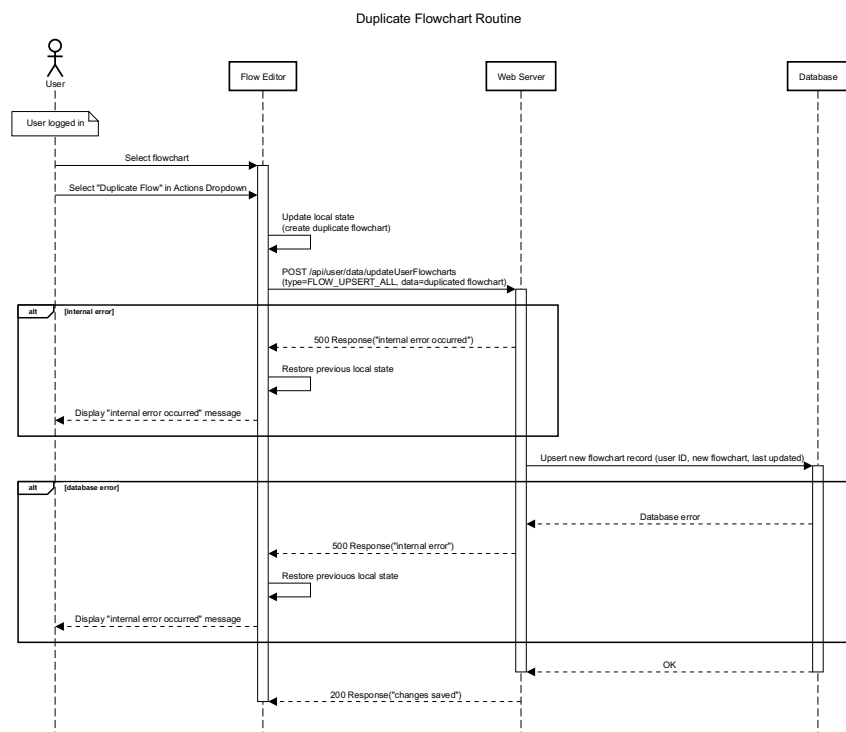


Figure 43: Sequence diagram for the routine to duplicate a selected flowchart. This diagram assumes the POST request to the web server is correct from the frontend and that the user is authenticated.

Export Flowchart as PDF

The export flowchart as PDF feature allows the user to download a PDF version of the currently selected flowchart. This utility is more complex due to the PDF rendering process, as this work is all done on the server. On the server, a special template called an Embedded JavaScript (EJS) [40] template is used to generate an HTML representation of the requested flowchart. After this HTML template is generated, it is loaded into a headless Google Chrome browser instance, Puppeteer [41]. This headless browser is spun up every time a PDF export operation occurs and is used to save the HTML template as a PDF. This PDF file is then sent back to the browser for the user to download.

Other PDF generation options were considered, with the primary alternative being to use JavaScript to manually create a PDF file from scratch. Initial efforts for this approach proved to be time consuming and brittle, so the Puppeteer option was selected instead. However, the Puppeteer option does consume extra resources as a dedicated headless browser needs to be spun up for every PDF export operation.

See **Figure 32** for the “Export Flow as PDF” option in the Actions dropdown and **Figure 44** for the corresponding sequence diagram.

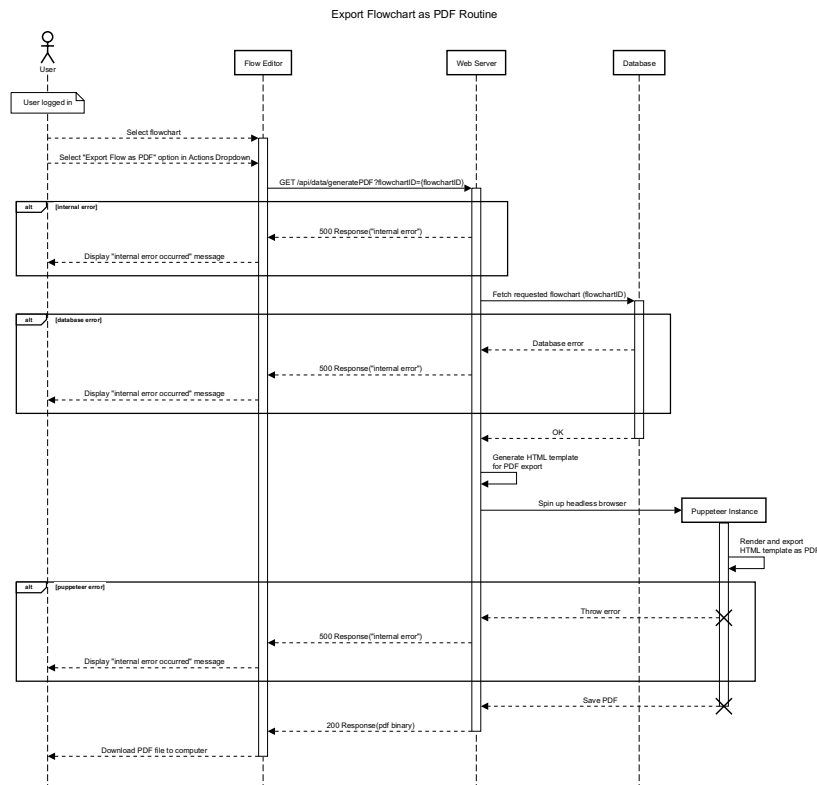


Figure 44: Sequence diagram for the routine to export a selected flowchart as a PDF. This diagram assumes the POST request to the web server is correct from the frontend, that the user is authenticated, and that the requested flowchart exists.

XI. State Management

This section details the various techniques that PolyFlowBuilder utilizes to maintain state across the application. The state management problems that to be addressed are:

1. User Authentication
2. Syncing user data between the frontend and backend
3. Maintaining frontend state

User Authentication

To have data associated with individual users, PolyFlowBuilder needs to authenticate users when they make various requests (and to determine authorization, but this is not a state management issue). There are two ways to authenticate users:

1. Session-based authentication
2. Token-based authentication

Session-based authentication is where the application keeps track of the current user session by storing a session record in a database that contains a session ID and user. This session ID is then saved as a cookie on the user's browser and sent to the server on each request. This ID is compared against the currently active sessions to determine which user made the request.

Token-based authentication is where a unique token (usually a JSON Web Token, or JWT) is stored on the user's browser, which is a small, encrypted payload that includes the user's identity and resources that user can authorize. This token is then sent to the server for every request, where it is decrypted and examined accordingly.

The big difference between these types of authentication schemas is that sessions store state on the server per active session, where token-based authenticate is stateless from the server perspective. However, there are security benefits to having information only located on the server to verify user identity (versus a stateless token, with JWTs in particular).

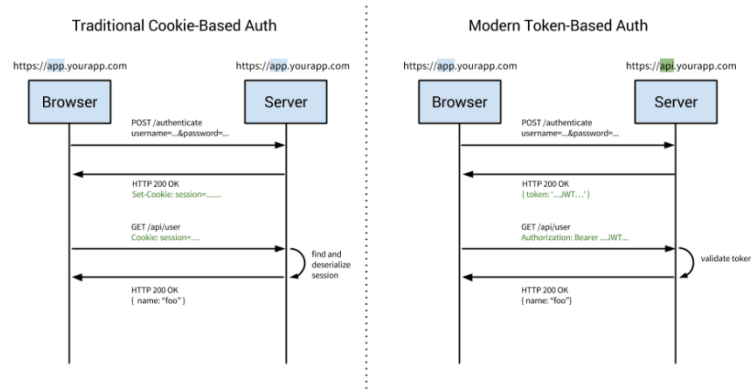


Figure 45: Sequence diagrams comparing traditional session-based (cookie) authentication versus modern token-based authentication [42].

PolyFlowBuilder uses session-based authentication as the mechanism to identify users when requests are made. Sessions on the server allows for arguably stronger security and the application is not (yet) at a scale where the overhead of a centralized database becomes a problem (which is where stateless authentication shows a clear advantage). Token-based authentication also introduces extra complexity that is currently unnecessary (e.g., refresh tokens, invalidating bad tokens, security issues with using JWTs as session tokens, etc.).

User Data Syncing

Another fundamental problem that needs to be addressed is how to ensure the user data on the frontend and backend are kept in sync with each other. The simplest but most naïve approach is to simply to overwrite all user data on the backend whenever a change to the user data is made in the frontend, illustrated in **Figure 46**.

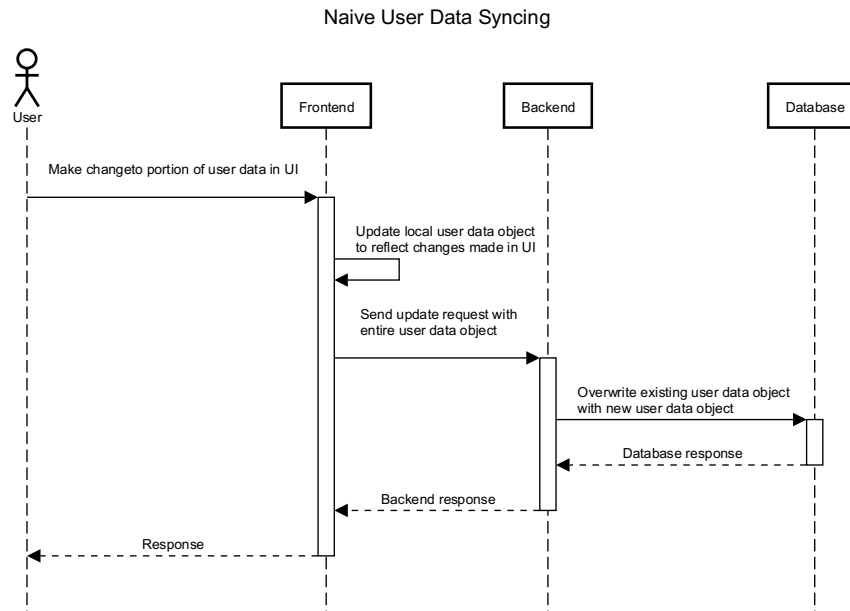


Figure 46: Sequence diagram representing the naïve way to update user data. Note that no failure cases are shown here. As the user data object grows, so do the size of the requests from the frontend to the backend.

The existing version of PolyFlowBuilder semi-follows this naïve approach, where an entire flowchart is overwritten in the database when any change is made in the frontend. This approach does not scale very well, especially as the amount of data associated with a user grows over time.

Therefore, to alleviate these inefficiencies, for operations that modify the user data (which are just flowcharts for the time being), a “chunking” or “update difference (diff)” system is introduced where only the changes that were made to the data are sent to the server. The frontend and backend then both update their current representations of the data independently from these changes to stay in sync. The version of PolyFlowBuilder detailed in this report utilizes the update chunking mechanism to keep user data synced when possible. **Figure 47** illustrates this concept.

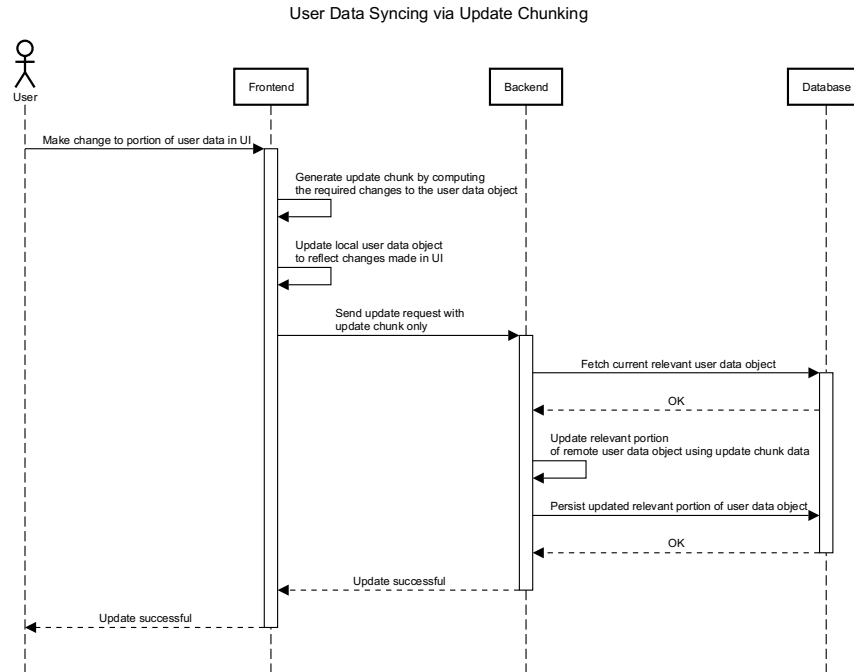


Figure 47: Sequence diagram representing the “update chunking” or “update difference” way to update user data. Observe that more work is done between the backend and database components to fetch the current version of user data. However, the requests from the frontend to the backend are much smaller, allowing for a snappier experience for users that might have slower Internet connections. Care must also be taken to ensure that the backend receives the updates in the correct order so that the data is updated properly. Note that no failure cases are shown here.

Observe that in both cases, the local copy of the data is always updated first before attempting to update the remote copy of the data. This is a form of opportunistic updating, and it allows for a better user experience as changes are seen immediately. If the server update fails, a message is sent back to the frontend to allow for the local change to be rolled back accordingly.

Frontend State Management

Due to the high degree of interactivity of the PolyFlowBuilder user interface, managing this state in the browser can become complicated if not done efficiently. Luckily, Svelte, the frontend framework that PolyFlowBuilder is built with, allows for state management to be elegant and straightforward between components. There are four different Svelte mechanisms that PolyFlowBuilder takes advantage of to ensure the efficient use of state:

1. Component properties (“props”): this allows for data to be passed from parent to child in a hierarchical fashion by allowing the parent component to mutate the “props” a child component exposes (see **Figure 48**).
2. Component property bindings: this allows for bidirectional data flow by enabling a parent component to pass data to a child component using props, but also enables the child’s changes to this prop to propagate back to the parent (see **Figure 49**).
3. Component events: this enables components to emit custom messages that its immediate parent can listen for and handle (see **Figure 50**).

4. Svelte stores: these are global state containers that publish updates to subscribers of the store when their internal values are modified, which are simple yet very powerful if used correctly (see **Figure 51**).

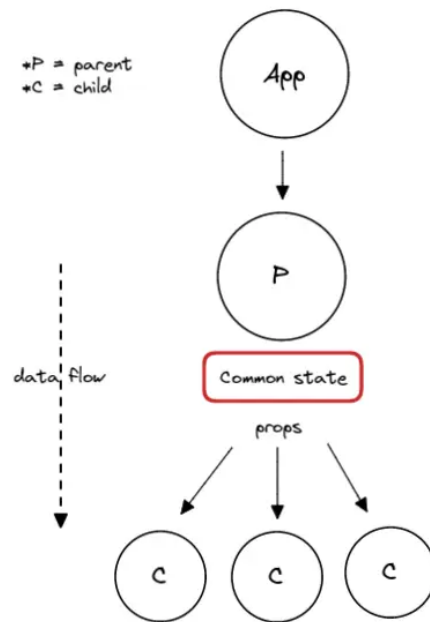


Figure 48: Propagating state from a parent component to child(ren) components using props [43]. Note that the data flow is one-way, from the parent to the child. Component props are used with things such as the PolyFlowBuilder course cards in the flow viewer.

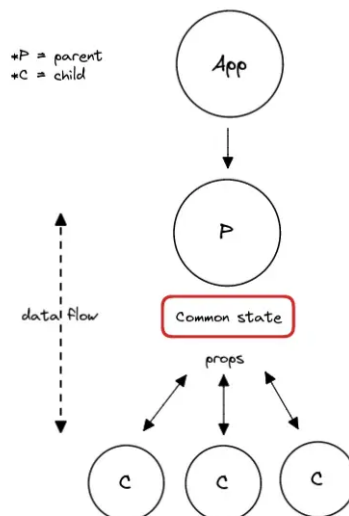


Figure 49: Allowing state in the parent to both be propagated and updated to/from child(ren) components using prop bindings [43]. Note that the data flow is bidirectional between the parent and child. A good use of prop bindings would be for textbox values – a parent can set the default value, but the child that contains the textbox can update the current value as well.

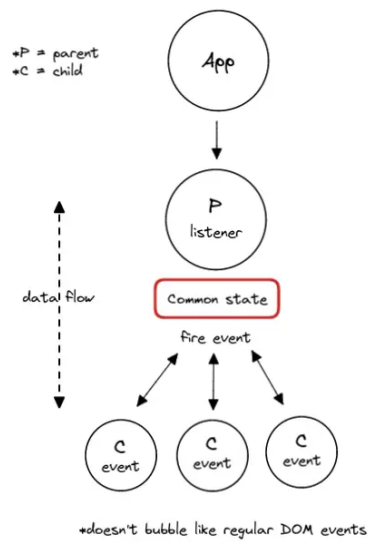


Figure 50: Using component custom events to send messages from a child component to its parent, where a listener intercepts and handles the message [43]. Used for interactions such as drag-and-drop and clicking.

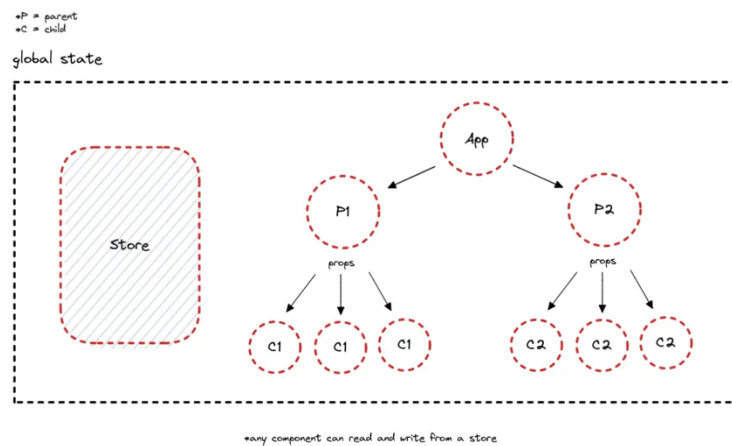


Figure 51: Using Svelte stores to store a plethora of state that multiple independent component trees need to interact with [43]. Svelte stores work using a publisher/subscriber model, which allows logic to run only when state is changed (as opposed to techniques such as polling).

XII. Backend

The PolyFlowBuilder backend is built using SvelteKit, PolyFlowBuilder’s application framework, which runs on the NodeJS runtime. The PolyFlowBuilder backend has two purposes:

1. Serve requests for content, such as frontend pages, pictures, etc.
2. Expose a set of APIs that the frontend/other applications can interact with to achieve specific functionality.

This chapter will describe how the PolyFlowBuilder backend is configured to meet these needs.

Serving Content Effectively

The backend is crucial to ensuring a snappy user experience. To this end, content should be served to the user from the backend as efficiently as possible to enable a fast time-to-interaction (TTI). This is straightforward to do with static content, as it does not change and is very fast to render. However, with modern web applications that have large degrees of dynamic content and interactivity, page loads and TTI can suffer due to the need for (potentially large) JavaScript bundles to load and execute in the browser. PolyFlowBuilder is one such application where this can be an issue.

Fortunately, there are techniques to mitigate these issues and to ensure a snappy experience for the user. These techniques are in the form of web rendering techniques, which are different methods for how to fetch and render content from the backend on the frontend.

The three categories of web rendering techniques [44] are:

1. Server-side rendering (SSR): load and generate an HTML page on the server with the dynamic content already added.
2. Client-side rendering (CSR): load the HTML and JavaScript from the server on the client and perform all dynamic processing and rendering on the client.
3. Something in between CSR and SSR: there are more sophisticated techniques that include advantages from both CSR and SSR techniques (e.g., hydration).

See **Figure 52** for a comparison between common web rendering techniques.

PolyFlowBuilder, through SvelteKit, uses server-side rendering to deliver the initial content (see the **Application Interface** chapter). The client then takes over to render and fetch additional pieces of content/data from the various backend APIs. This approach has two core benefits:

1. Because there is limited JavaScript dynamically updating the content of the page, search engines can better crawl and index these pages. This results in better search engine optimization (SEO) scores.
2. The TTI for webpages with dynamic content (e.g., the flowchart editor) is much faster and more consistent compared to the TTI for dynamic pages that need to be rendered on the client (especially as the complexity of the web application grows).

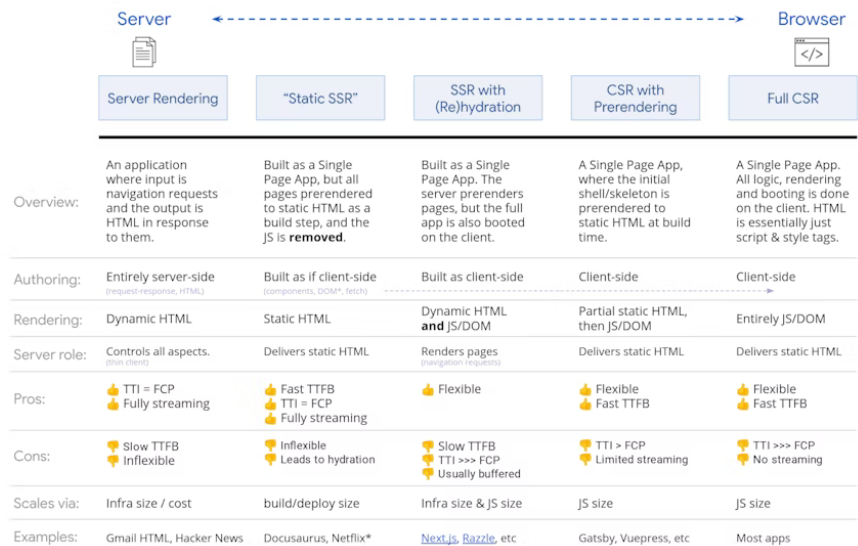


Figure 52: Comparison between five common rendering techniques to effectively serve content to users [44].

Backend APIs

All interactions that the frontend needs to have with the backend that don't involve content serving are done through the exposed APIs that the backend provides. In PolyFlowBuilder, all APIs are HTTP-based, but other protocols and techniques exist that are optimal depending on what the use case is. See **Table 12** for the APIs exposed by PolyFlowBuilder.

Table 12. PolyFlowBuilder backend APIs.

API	Endpoints	Notes
User account management	POST api/auth/register POST api/auth/login DELETE api/auth/login POST api/auth/forgotpassword POST api/auth/resetpassword	See the User Account Management chapter.
User flowchart data	GET /api/user/data/getUserFlowcharts POST /api/user/data/updateUserFlowcharts	APIs to interact with and manipulate user flowchart data.
Flowchart editor data	GET /api/data/getAvailableProgramData POST /api/data/searchCatalog	APIs to fetch data the flowchart editor requires.
Flowchart data utilities	GET /api/data/generateFlowchart GET /api/data/generatePDF	APIs to perform various data-related actions with flowcharts.

XIII. Database

The database is a critical component of the PolyFlowBuilder architecture, as all persistent data lives here. This includes all user data and all API data. Data in PolyFlowBuilder is inherently relational, so MySQL was chosen as the database for its ease of use, flexibility, and wide adoption.

Data Tables

The data required by PolyFlowBuilder is made up of the tables seen in **Table 13**. The entity-relationship diagram for these tables can be seen in **Figure 53**.

Table 13. Data tables to organize PolyFlowBuilder data.

Table Name	Description
User	Table for user account information is stored (user ID, username, email, password, etc.).
Token	Table for tokens storage (session tokens, password reset tokens).
FeedbackReport	Table for feedback reports from the Submit Feedback page.
Flowchart	Table for all flowcharts created by users.
Program	Table that stores data for all supported academic programs.
Catalog	Table that stores data for all supported catalogs (YYYY-YYYY format).
StartYear	Table that stores data for all supported start years (YYYY format).
TemplateFlowchart	Table that stores data for all template flowcharts. These template flowcharts have a 1:1 mapping to the public PDF template flowcharts [1].
Course	Table that stores data for all supported courses (all courses in all supported catalogs).
CourseRequisite	Table that stores data for the course requisites of all supported courses.
GECourse	Table that stores data for the course GE category of all supported courses.
TermTypicallyOffered	Table that stores data for the course term typically offered information of all supported courses [36].

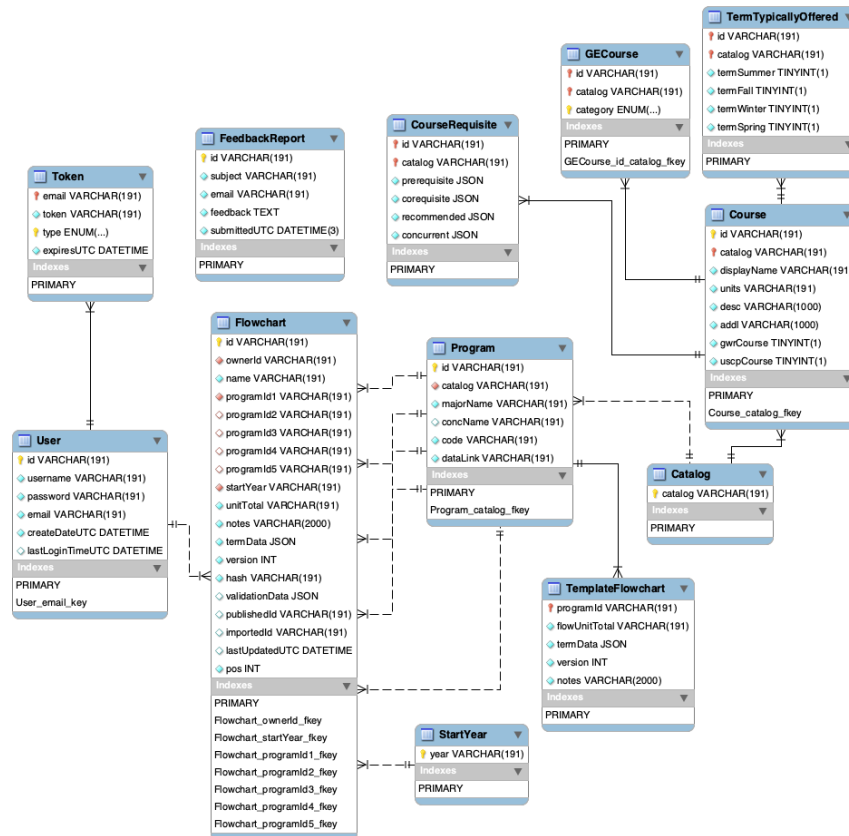


Figure 53: Entity-relationship (ER) diagram for the SQL tables used to store PolyFlowBuilder data.

Interacting with The Database

To interact with a MySQL database from the backend, three techniques can be used:

1. Use a minimal driver to issue raw SQL statements to the database.
2. Use a query builder to generate SQL statements using a more developer-friendly syntax.
3. Use an object relational mapper (ORM) to map records in SQL tables to language-native objects.

The first technique is the least abstracted and requires the most logic to integrate with an existing system, whereas the third technique is the most abstracted. However, using the minimal driver is arguably the most performant as you know exactly what queries you are issuing to the database. This differs from the other two techniques, which programmatically issue SQL statements to achieve the desired end behavior. If a poor query builder or ORM are used (or in a way that is not recommended), these SQL statements can quickly add up and become unoptimized, yielding a less performant experience.

The current version of PolyFlowBuilder used the minimal driver approach as performance and granularity of what the database did was imperative, and learning SQL was of priority. However, the design goals changed from the current version to the new version to prioritize code maintainability, readability, and reliability over strict performance.

Therefore, the new version of PolyFlowBuilder uses an ORM, Prisma [26], to interact with the database. This allows for developers to focus on what operations need to occur to the database to achieve the expected result, not how they are done. Additionally, various implementation concerns, such as SQL injection attacks, are abstracted away when not using a minimal driver.

See **Figure 54** and **Figure 55** for a code comparison between using a minimal driver and an ORM (Prisma) to interact with the SQL database.

```
function addUser({
  username: string,
  password: string,
  email: string,
  data: Object
}) {
  const sqlQuery = `INSERT INTO ${process.env.DB_TABLE_USERS} (username, password, email, data, createdAt) VALUES (?, ?, ?, ?, ?)`;
  // for timestamp, all accounts before this was added have timestamp of 2020-11-22 22:00:00
  const curDate = new Date(new Date().getTime() + -8 * 3600 * 1000)
    .toISOString()
    .slice(0, 19)
    .replace("T", " ");

  con.query(
    sqlQuery,
    [username, password, email, JSON.stringify(data), curDate],
    (err) => {
      if (err) throw err;
      logger.log('User [{username}] successfully added to main database');
    }
  );
}
```

Figure 54: Source code in the current version of PolyFlowBuilder to add a new user to the database from the backend (does not include existing user checks). Observe how an explicit SQL statement is created and parameterized before being issued to the database.

```
export async function createUser(registerData: {
  username: string;
  email: string;
  password: string;
}): Promise<string | null> {
  const user = await prisma.user.findUnique({
    where: {
      email: registerData.email
    }
  });

  if (user) {
    logger.info('New user attempted to register with existing email');
    return null;
  } else {
    const newUser = await prisma.user.create({
      data: {
        username: registerData.username,
        password: await argon2.hash(registerData.password, { type: argon2.argon2id }),
        email: registerData.email
      }
    });

    logger.info('User with email [{registerData.email}] successfully added to main database');
    return newUser.id;
  }
}
```

Figure 55: Source code in the rewritten version of PolyFlowBuilder to add a new user to the database from the backend. Observe how no explicit SQL statements are defined; instead, database operations are performed through function calls on the respective tables (wrapped by the Prisma ORM).

XIV. Evaluation

As seen in the previous chapters, a monumental amount of work went into developing and rewriting PolyFlowBuilder. Therefore, because this project is within the bounds of a senior project, proper evaluation metrics must be stated and measured to determine whether the project holistically was a “success”, “failure”, or somewhere in between. This chapter will describe the various criteria to determine project success.

Evaluation Metrics

The evaluation criteria for this project are detailed in **Table 14**.

Table 14. Criteria for evaluating senior project success.

Evaluation Metric	Acceptance Criteria
Feature implementation	All features mentioned in the Scope of Work section should be implemented and tested.
Feature parity with current live website	There should be feature parity between the flowchart functionality offered in the new version of PolyFlowBuilder developed for this project and the existing production website.
End-to-end tests	All end-to-end tests written to test the implemented features should be passing.
Integration tests	All integration tests written to test the implemented features should be passing.
Unit tests	All unit tests written to test the implemented features should be passing.
User feedback	Users that are beta testing the new website should be able to verify that no critical bugs exist in the current implementation of the project. If bugs are reported, these are addressed and either fixed before the project is complete, or a plan is put in place to fix the broken behavior after the project is finished.

These criteria ensure a holistic evaluation of the project’s many different systems and how they interact with each other, as well as how the user experience in the rewritten version of PolyFlowBuilder compares to the existing version.

Evaluation Results

This section describes the evaluation results, using the evaluation metrics detailed in the **Evaluation Metrics** section.

Feature Implementation

This metric evaluates to what degree the high-level features defined in the **Scope of Work** chapter were completed within the senior project timeframe.

Each feature has a unique “feature code” seen in **Table 15**, which is derived from the bullet number of a particular feature in the **Scope of Work** chapter. For example, code “I-1-a” is the feature defined in Phase I, item 1, subitem a: “standard metadata about each course (name, description, number of units, etc.)”.

The various levels of completion metric are defined as follows:

1. Complete (✓) – the feature is fully implemented according to the specification.
2. Feature incomplete (Δ) – the feature logic not fully implemented according to the specification.
3. Tests incomplete (☑) – the feature logic is fully implemented but not fully tested according to the specification.
4. Feature not implemented (✗) – the feature is not implemented.

Table 15. Feature implementation results.

Feature Code	Completion Metric	Feature Exists in PolyFlowBuilder 1.0	Notes
I-1-a	✓	Yes	n/a
I-1-b	✓	Yes	n/a
I-1-c	✓	Yes	n/a
I-2	✓	Yes	n/a
I-3	✓	Yes	n/a
I-4	✓	Partial	Data is currently cached on the local filesystem from the database on startup.
I-5	✓	Yes	n/a
II-1-a	✓	Yes	n/a
II-1-b	✓	Yes	n/a
II-1-c	✓	Yes	n/a
II-1-d	✓	Yes	n/a
II-2	✓	Yes	n/a
II-3	✓	Yes	n/a
III-1	✓	Yes	n/a

III-2	<input checked="" type="checkbox"/>	Yes	n/a
III-3	<input checked="" type="checkbox"/>	Yes	n/a
IV-1	<input checked="" type="checkbox"/>	Yes	n/a
IV-2	<input checked="" type="checkbox"/>	Yes	n/a
IV-3	<input checked="" type="checkbox"/>	Yes	n/a
IV-4-a	<input checked="" type="checkbox"/>	Yes	Feature was tested via manual inspection. Automated tests were not written due to time constraints.
IV-4-b	<input checked="" type="checkbox"/>	Yes	Feature was tested via manual inspection. Automated tests were not written due to time constraints.
IV-4-c	<input type="checkbox"/>	No	The UI to modify the associated flowchart programs exists, but the logic to persist these flowchart program modifications is not implemented due to time constraints. Additionally, this feature is not present in the current version of PolyFlowBuilder, so it was deprioritized in favor of features that contribute to existing feature parity.
IV-5-a	<input checked="" type="checkbox"/>	Yes	n/a
IV-5-b	<input checked="" type="checkbox"/>	Yes	n/a
IV-5-c	<input checked="" type="checkbox"/>	Yes	Feature was tested via manual inspection. Automated tests were not written due to time constraints.
IV-6-a-i	<input checked="" type="checkbox"/>	Yes	Feature was tested via manual inspection. Automated tests were not written due to time constraints.
IV-6-a-ii	<input checked="" type="checkbox"/>	Yes	Feature was tested via manual inspection. Automated tests were not written due to time constraints.
IV-6-a-iii	<input checked="" type="checkbox"/>	Yes	Feature was tested via manual inspection. Automated tests were not written due to time constraints.
IV-6-a-iv	<input checked="" type="checkbox"/>	Yes	Feature was tested via manual inspection. Automated tests were not written due to time constraints.
IV-6-b	<input checked="" type="checkbox"/>	Yes	Feature was tested via manual inspection. Automated tests were not written due to time constraints.

V-1	<input checked="" type="checkbox"/>	Yes	Feature was tested via manual inspection. Automated tests were not written due to time constraints.
V-2	<input checked="" type="checkbox"/>	Yes	Feature was tested via manual inspection. Automated tests were not written due to time constraints.
V-3-a	<input type="checkbox"/>	Yes	Feature could not be implemented due to time constraints.
V-3-b	<input type="checkbox"/>	Yes	Feature could not be implemented due to time constraints.
V-3-c	<input type="checkbox"/>	Yes	Feature could not be implemented due to time constraints.
V-3-d	<input type="checkbox"/>	Yes	Feature could not be implemented due to time constraints.

From the results seen in **Table 15**, this evaluation metric is considered mostly satisfied.

Feature Parity

From **Table 15**, we see that all features that contribute to feature parity with the existing version of PolyFlowBuilder have been implemented except for the flowchart validation system (requirements V-3-a through V-3-d).

Additionally, there are several high-level components of the new version of PolyFlowBuilder that do not exist in the current version, which are:

1. Support for multiple flowchart programs in a single flowchart
2. Automated tests for production-facing features

These features are part of the future roadmap for new features meant to increase user productivity. These features needed to be implemented at this stage of the project as they both heavily influence core functionality.

From these results, this evaluation metric is considered mostly satisfied.

End-to-End and Integration Tests

To increase the quality of code and application robustness at an application level, automated tests were used significantly. End-to-end and integration tests were written using the test runner Playwright [45]. To test the application on a more holistic level, Playwright will:

1. Create a production build of the application.
2. run a local instance of the production application.
3. run automated tests against the local instance.

In this version of PolyFlowBuilder, there are 184 unique end-to-end/integration tests that were written to test the features detailed in the **Scope of Work** and **Feature Integration** sections. See **Figure 56** for a visual of the automated Playwright test runner.

```
✓ 162 page/LoginPageTests.test.ts:75:3 › login page tests › correct credentials, check redirect and cookie (2.1s)
✓ 163 page/registerPageTests.test.ts:45:3 › registration page tests › register fails without confirm password (1.0s)
✓ 164 routine/createFlowRoutineTests.test.ts:152:3 › create flow routine tests › 401 case handled properly (2.0s)
✓ 165 page/resetPasswordPageTests.test.ts:92:3 › reset password page tests (token) › password and passwordConfirm dont match (1.3s)
✓ 166 page/flows/flowListTests.test.ts:246:3 › flow list tests › flow selection works (3.1s)
✓ 167 page/registerPageTests.test.ts:54:3 › registration page tests › register fails with invalid email address (type 1) (1.2s)
✓ 168 routine/resetPasswordRoutineTests.test.ts:34:3 › reset password routine tests › user initiates reset password request (1.5s)
✓ 169 page/resetPasswordPageTests.test.ts:110:3 › reset password page tests (token) › token gets deleted before reset can happen (1.1s)
✓ 170 page/LoginPageTests.test.ts:84:3 › login page tests › 500 case with correct credentials (988ms)
✓ 171 page/registerPageTests.test.ts:64:3 › registration page tests › register fails with invalid email address (type 2) (986ms)
[info] [DB/User] [2023-06-11 10:27:54]: User with email [pfb_test_flowsPage_modifyTermData_playwright@test.com] successfully deleted from master database
✓ 172 routine/createFlowRoutineTests.test.ts:220:3 › create flow routine tests › 400 case handled properly (1.9s)
✓ 173 routine/resetPasswordRoutineTests.test.ts:55:3 › reset password routine tests › user navigates to reset password link and resets password (1.3s)
[info] [DB/Token] [2023-06-11 10:27:54]: PASSWORD_RESET tokens expired for pfb_test_resetPasswordPage_playwright@test.com
✓ 174 page/resetPasswordPageTests.test.ts:128:3 › reset password page tests (token) › token expires before reset can happen (2.0s)
[info] [DB/User] [2023-06-11 10:27:54]: User with email [pfb_test_loginPage_playwright@test.com] successfully deleted from master database
✓ 175 page/registerPageTests.test.ts:79:3 › registration page tests › register fails with mismatched passwords (955ms)
[info] [DB/Token] [2023-06-11 10:27:55]: PASSWORD_RESET tokens expired for pfb_test_resetPasswordPage_playwright@test.com
[info] [DB/User] [2023-06-11 10:27:55]: User with email [pfb_test_flowsPage_flow_list_playwright@test.com] successfully deleted from master database
✓ 176 routine/resetPasswordRoutineTests.test.ts:98:3 › reset password routine tests › user cannot log in with old password (984ms)
✓ 177 page/registerPageTests.test.ts:97:3 › registration page tests › register fails with invalid email address (type 2) and mismatched passwords (911ms)
✓ 178 routine/createFlowRoutineTests.test.ts:288:3 › create flow routine tests › 500 case handled properly (1.8s)
[info] [DB/Token] [2023-06-11 10:27:56]: PASSWORD_RESET tokens expired for pfb_test_resetPasswordPage_playwright@test.com
✓ 179 routine/resetPasswordRoutineTests.test.ts:119:3 › reset password routine tests › user able to log in with new password (1.2s)
✓ 180 page/resetPasswordPageTests.test.ts:152:3 › reset password page tests (token) › 500 failure case (897ms)
✓ 181 page/registerPageTests.test.ts:117:3 › registration page tests › registration succeeds and redirects to login page (1.1s)
✓ 182 page/resetPasswordPageTests.test.ts:173:3 › reset password page tests (token) › password successfully reset (1.0s)
[info] [DB/User] [2023-06-11 10:27:57]: User with email [pfb_test_createFlowRoutine_playwright@test.com] successfully deleted from master database
[info] [DB/User] [2023-06-11 10:27:57]: User with email [pfb_test_resetPasswordRoutine_playwright@test.com] successfully deleted from master database
✓ 183 page/registerPageTests.test.ts:132:3 › registration page tests › registration with an existing email fails (859ms)
[info] [DB/User] [2023-06-11 10:27:58]: User with email [pfb_test_resetPasswordPage_playwright@test.com] successfully deleted from master database
✓ 184 page/registerPageTests.test.ts:147:3 › registration page tests › 500 case with valid registration data (832ms)
[info] [DB/User] [2023-06-11 10:27:59]: User with email [pfb_test_registerPage_playwright@test.com] successfully deleted from master database

Slow test file: page/Flows/FlowListTests.test.ts (20.3s)
Slow test file: page/Flows/modifyFlowTermDataTests.test.ts (16.2s)
Consider splitting slow test files to speed up parallel execution
184 passed (32.5s)
```

Figure 56: Playwright test runner executing all automated end-to-end and integration tests against a local instance of the production PolyFlowBuilder application.

As can be seen in **Figure 56**, all end-to-end and integration tests are passing successfully, so this evaluation metric is considered satisfied.

Unit Tests

To increase the quality of code and application robustness at the “unit” level (individual functions, components, etc.), automated tests were used significantly. Unit tests were written using the test runner Vitest [46] to ensure individual units of the application were working before testing their interactions with each other (in integration and end-to-end tests).

In this version of PolyFlowBuilder, there are 100 unique unit tests that were written to test the functionality of individual functions and components in the application. See **Figure 57** for a visual of the automated Vitest test runner.

```

✓ src/lib/common/util/FlowTermUtilCommon.test.ts (3)
✓ src/lib/components/Flows/FlowEditor/FlowEditorHeader.test.ts (4) 931ms
✓ src/lib/components/Flows/FlowEditor/CourseItem.test.ts (3) 1355ms
✓ src/lib/client/util/courseItemUtil.test.ts (1)
✓ src/lib/components/Flows/FlowEditor/TermContainer.test.ts (2) 362ms
✓ src/lib/common/util/courseDataUtilCommon.test.ts (7)
✓ src/lib/common/util/unitCounterUtilCommon.test.ts (15)
✓ src/lib/common/util/FlowDataUtilCommon.test.ts (9)
✓ src/lib/components/common/FlowPropertiesSelector/ProgramSelector.test.ts (10) 3398ms
✓ src/lib/client/util/unitCounterUtilClient.test.ts (3)
✓ src/lib/server/util/courseCacheUtil.test.ts (8)
✓ src/lib/components/Flows/FlowViewer.test.ts (3) 1162ms
✓ src/lib/server/util/FlowDataUtil.test.ts (2)
✓ src/lib/components/Flows/FlowEditor/FlowEditorFooter.test.ts (3)
✓ src/lib/components/Flows/FlowEditor/FlowEditor.test.ts (3) 831ms
✓ src/lib/components/Flows/FlowInfoPanel/FlowListItem.test.ts (2) 823ms
✓ src/lib/components/Flows/modals/NewFlowModal.test.ts (3) 655ms
✓ src/lib/components/common/FlowPropertiesSelector/Component.test.ts (16) 7205ms
✓ src/lib/components/common/FlowPropertiesSelector/UIWrapper.test.ts (3)

Test Files 19 passed (19)
Tests 100 passed (100)
Start at 10:37:49
Duration 29.71s (transform 2.06s, setup 5.51s, collect 68.22s, tests 17.36s, environment 14.57s, prepare 3.07s)

PASS Waiting for file changes...
press h to show help, press q to quit

```

Figure 57: Vitest test runner executing all automated unit tests against individual functions and components that make up the PolyFlowBuilder application.

As can be seen in **Figure 57**, all unit tests are passing successfully, so this evaluation metric is considered satisfied.

User Feedback

One of the most important evaluation metrics for an application that will eventually be used by an entire community of people is the feedback from them before the application is launched. When the project was close to feature-complete (in the scope of the senior project), various users of the existing PolyFlowBuilder platform were asked to compare their experiences with the rewritten version and to provide any feedback they had. A summary of this feedback is collected in **Table 16**.

Table 16. Summary of feedback received during project development.

Type	Feedback	Action
Bug (minor)	The list of flowcharts in the flowchart editor only load when the user first navigates to the flowchart editor. If the user navigates away from the flowchart editor and back to it, the list of flowcharts does not reload.	Need to perform root-cause analysis – this bug was reported at the end of the development cycle.
Bug (minimal)	The “New Flow”, “Actions”, and “Delete Flow” buttons are slightly misaligned in Google Chrome.	Need to perform root-cause analysis – this bug was reported at the end of the development cycle.

Suggestion	When multiple courses are selected, let a click in the flowchart that is not on a course deselect all courses.	Consider usefulness of suggestion and implement.
Suggestion	Color scheme of some components (action buttons, courses) to ensure accessibility to colorblind users.	Consider changing colors of impacted components to be more accessible and perform A/B testing.
Suggestion	Consider relocating “New Flow” action button at the bottom of the flowchart list and the “Delete Flow” action button next to each flowchart.	Consider impact to usability and perform A/B testing.
Comment	Course searching matches queries better in the newer version.	None
Comment	Selecting multiple terms in add/remove terms can be nonintuitive	Consider adding helper/guides to inform users how to select multiple terms.
Comment	New user interface is smoother, cleaner, less cluttered, and preferred overall to the existing interface.	None
Comment	Course selection interface and concept is preferred to existing interface.	None
Comment	Credit bin interface is very useful for storing courses that user has received credit for.	None
Comment	Cannot add programs that are not on the catalog(s) that a flowchart is loaded for.	Consider alternative use cases to see if this functionality needs to be changed.

As the feedback received from users throughout the development cycle of this project was overall positive with all bugs and issues having a clear path to resolution, this evaluation metric is considered satisfied.

XV. Conclusions

Current Project State

By the end of the senior project, a significant amount of work has been accomplished. This work is summarized in the following points:

1. A rewritten PolyFlowBuilder codebase exists and has parity with a vast majority of the features seen in the existing version.
2. The rewritten codebase has automated tests to cover a vast majority of important application functionality.
3. The rewritten codebase is built to a production-grade standard and is in a state where it can scale easily to additional features and complexity.
4. The rewritten codebase has been extensively evaluated through the scope of the senior project evaluation metrics (see the **Evaluation** chapter) and passed a majority of the criteria. Only the validation features and a handful of feature tests are missing, with everything else being fully satisfied.
5. The project is ready for further development but can run with no major issues in its current form as a replacement to the existing PolyFlowBuilder application.

From a holistic perspective, considering the amount of work done and what this work enables for the future of PolyFlowBuilder at Cal Poly, along with the success of the evaluation criteria, the senior project was a success.

Project Takeaways

With a project this large, there are many lessons to be learned in project management, planning, and software development. The largest takeaways for me are:

1. If something can go wrong, it eventually will go wrong.
2. Be realistic with timelines and how much work you can accomplish, especially if you have other commitments. I vastly underestimated the development time for some portions of the project, and as a result not everything was completed optimally (e.g., missing the validation suite of features and not everything is fully tested).
3. Small incremental work is often better than occasional chunks of large work. I found that if I worked on making small progress every day, I was able to accomplish more in the same time frame compared to if I did all that work in a single development session.
4. Always solicit user feedback. Some features that intuitively make sense to me may not make sense to others that use the platform. Other users may also have suggestions and other feedback that wasn't considered previously.
5. Developing software that is maintainable, reliable, robust, and scalable is hard! I did not always get things right which resulted in additional development time spent rewriting various components of systems to be more performant.

Future Work

Although the amount of work currently done for PolyFlowBuilder is significant, there is still a monumental amount of work to be done in the future. A non-exhaustive list of these future work tasks can be seen below:

1. Implement the flowchart validation suite of features and add automated tests for all missed items.
2. Perform more user testing to ensure changes are well accepted, rational, and intuitive.
3. Continue implementing support for multiple academic programs.
4. Begin implementing “shared marketplace” for flowcharts where flowcharts can be published and imported by other users.
5. Reach out to Cal Poly to inquire about access to more structured data for things like template flowcharts, advanced course metadata, etc.
6. Implement features to help users with the impending quarter-to-semester transition.
7. Implement curriculum sheet validation, to determine whether a collection of courses satisfies degree requirements.

The list goes on. The future work will eventually be carried out (hopefully) by other Cal Poly students to ensure that the project does not fizzle out due to my eventual departure from the project. The future is exciting for PolyFlowBuilder!

References

- [1] California Polytechnic State University, San Luis Obispo. “Degree Flowcharts and Curriculum Sheets”. Available: <https://flowcharts.calpoly.edu>. [Accessed June 2023]
- [2] California Polytechnic State University, San Luis Obispo. “B.S. in Computer Engineering”. Available: <https://flowcharts.calpoly.edu/downloads/mymap/21-22.52CPEBSU.pdf>. [Accessed June 2023]
- [3] California Polytechnic State University, San Luis Obispo. “2022-2026 Academic Catalog”. Available: <https://catalog.calpoly.edu>. [Accessed June 2023]
- [4] Altice USA News, Inc. “Rate My Professors”. Available: <https://ratemyprofessors.com>. [Accessed June 2023]
- [5] Polyratings. “Polyratings”. Available: <https://polyratings.dev>. [Accessed June 2023]
- [6] Batch, Meeder. “PolyFlows”. Available: <https://web.archive.org/web/20151206214231/http://polyflows.com/>. [Accessed June 2023]
- [7] Applegarth. “PolyFlowBuilder”. Available: <https://polyflowbuilder.duncanapple.io>. [Accessed June 2023]
- [8] Oracle. “MySQL”. Available: <https://mysql.com>. [Accessed June 2023]
- [9] The PostgreSQL Global Development Group. “PostgreSQL: The World’s Most Advanced Open Source Relational Database”. Available: <https://postgresql.org>. [Accessed June 2023]
- [10] MongoDB, Inc. “MongoDB: Build the next big thing”. Available: <https://mongodb.com>. [Accessed June 2023]
- [11] Learn Computer Science. “Full Stack Developer”. Available: <https://www.learncomputerscienceonline.com/full-stack-developer/>. [Accessed June 2023]
- [12] Raval. “Top 5 Frontend Frameworks to Look for Your Upcoming Web Project”. 29 March 2023. Available: <https://radixweb.com/blog/top-front-end-frameworks-for-web-development>. [Accessed June 2023]
- [13] Back4app. “Top 10 Backend Frameworks In 2023”. Available: <https://blog.back4app.com/backend-frameworks/>. [Accessed June 2023]

- [14] Vercel. “Next.JS: The React Framework for the Web”. Available: <https://nextjs.org>. [Accessed June 2023]
- [15] Meta Open Source. “React: The library for web and native user interfaces”. Available: <https://react.dev>. [Accessed June 2023]
- [16] Gatsby, Inc. “Gatsby: The Fastest Frontend for the Headless Web”. Available: <https://www.gatsbyjs.com/>. [Accessed June 2023]
- [17] Nuxt. “Nuxt: The Intuitive Web Framework”. Available: <https://nuxt.com>. [Accessed June 2023]
- [18] You. “Vue.js: The Progressive JavaScript Framework”. Available: <https://vuejs.org>. [Accessed June 2023]
- [19] Google. “Angular: Deliver web apps with confidence”. Available: <https://angular.io>. [Accessed June 2023]
- [20] SvelteKit. “SvelteKit: web development, streamlined”. Available: <https://kit.svelte.dev>. [Accessed June 2023]
- [21] Svelte. “Svelte: Cybernetically enhanced web apps”. Available: <https://svelte.dev>. [Accessed June 2023]
- [22] Khan et al. “SQL and NoSQL Databases Software architectures performance analysis and assessments – A systematic literature review”. September 2022. Available: <https://dx.doi.org/10.48550/arXiv.2209.06977>. [Accessed June 2023]
- [23] OpenJS Foundation. “Nodejs”. Available: <https://nodejs.org/en>. [Accessed June 2023]
- [24] Microsoft. “TypeScript”. Available: <https://www.typescriptlang.org/>. [Accessed June 2023]
- [25] The Mozilla Foundation. “What is JavaScript?”. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript. [Accessed June 2023]
- [26] Prisma Data, Inc. “Prisma: Next-generation Node.js and TypeScript ORM”. Available: <https://prisma.io>. [Accessed June 2023]
- [27] TypeORM. “TypeORM”. Available: <https://typeorm.io>. [Accessed June 2023]

[28] Adámek. “MikroORM: TypeScript ORM for Node.js, based on Data Mapper, Unit of Work and Identity Map patterns”. Available: <https://mikro-orm.io>. [Accessed June 2023]

[29] Bootstrap team. “Build fast, responsive sites with Bootstrap”. Available: <https://getbootstrap.com>. [Accessed June 2023]

[30] Tailwind CSS. “Rapidly build modern websites without ever learning your HTML”. Available: <https://tailwindcss.com>. [Accessed June 2023]

[31] Saadeghi. “DaisyUI. The most popular component library for Tailwind CSS”. Available: <https://daisyui.com>. [Accessed June 2023]

[32] California Polytechnic State University, San Luis Obispo. “2015-2017 Academic Catalog”. Available: <https://catalog.calpoly.edu/previouscatalogs/2015-2017/>. [Accessed June 2023]

[33] California Polytechnic State University, San Luis Obispo. “2017-2019 Academic Catalog”. Available: <https://catalog.calpoly.edu/previouscatalogs/2017-2019/>. [Accessed June 2023]

[34] California Polytechnic State University, San Luis Obispo. “2019-2020 Academic Catalog”. Available: <https://catalog.calpoly.edu/previouscatalogs/2019-2020/>. [Accessed June 2023]

[35] California Polytechnic State University, San Luis Obispo. “2020-2021 Academic Catalog”. Available: <https://catalog.calpoly.edu/previouscatalogs/2020-2021/>. [Accessed June 2023]

[36] California Polytechnic State University, San Luis Obispo. “2021-2022 Academic Catalog”. Available: <https://catalog.calpoly.edu/previouscatalogs/2021-2022/>. [Accessed June 2023]

[37] Office of the Registrar. “Term Typically Offered”. Available: <https://registrar.calpoly.edu/term-typically-offered>. [Accessed June 2023]

[38] Biryukov, Dinu, Khovratovich. “Argon2: the memory-hard function for password hashing and other applications”. Available: <https://www.password-hashing.net/argon2-specs.pdf>. [Accessed June 2023]

[39] Redis Ltd. “Redis: The open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker”. Available: <https://redis.io>. [Accessed June 2023]

[40] Eernisse. “EJS. Embedded JavaScript templating”. Available: <https://ejs.co>. [Accessed June 2023]

[41] Google. “Puppeteer”. Available: <https://pptr.dev/>. [Accessed June 2023]

[42] CodeSpot. “Token vs Session Authentication”. Available: <https://www.codespot.org/token-vs-session-authentication/>. [Accessed June 2023]

[43] Joy of Code. “Svelte State Management Guide”. Available: <https://joyofcode.xyz/svelte-state-management>. [Accessed June 2023]

[44] Google Developers. “Rendering on the Web”. Available: <https://web.dev/rendering-on-the-web/>. [Accessed June 2023]

[45] Microsoft. “Playwright. Playwright enables reliable end-to-end testing for modern web apps”. Available: <https://playwright.dev>. [Accessed June 2023]

[46] Fu, Capeletto, Vitest contributors. “Vitest: Blazing Fast Unit Test Framework”. Available: <https://vitest.dev/>. [Accessed June 2023]