Prior Learning Assessment Portfolio

Brady Underwood (08646866)
22823 SE 268th Pl
Maple Valley, WA, 98038
11/11/2024
425-358-1910
underwood.brady@gmail.com
CSCI 495 - Field Work/Practicum

Table of Contents:

Letter of Intent:

With this portfolio I intend to prove my learning outcomes in software design, implementation, and debugging and ability to explain technical information effectively to less advanced students.

Design - LocateCRM:

Introduction:

I aim to explain the design process for a “production-level” app from basic UML modeling to ultimately starting to write code. This paper will attempt to explain ideas to lower division CS students in a way that is digestible and informative while still remaining technical in nature.

Background:

I created a location-based customer relation management app called LocateCRM that runs on IOS, android, and the web. Other examples of CRM tools include hubspot, salesforce, or monday.com. The idea for this project was inspired by my brother Callan who started a door to door window washing company here in Bozeman. They needed a tool to mark what houses they have been to on a map, track leads, and keep track of jobs on a calendar. Upon discovering there were no good tools out there that weren’t hundreds of dollars a month they reached out to me to develop a similar app they could use. I decided to take on the task. In order to replicate the apps they were looking at I needed the following features: an interactive map, a database to store leads and markers, authentication, and a calendar service.

Design:

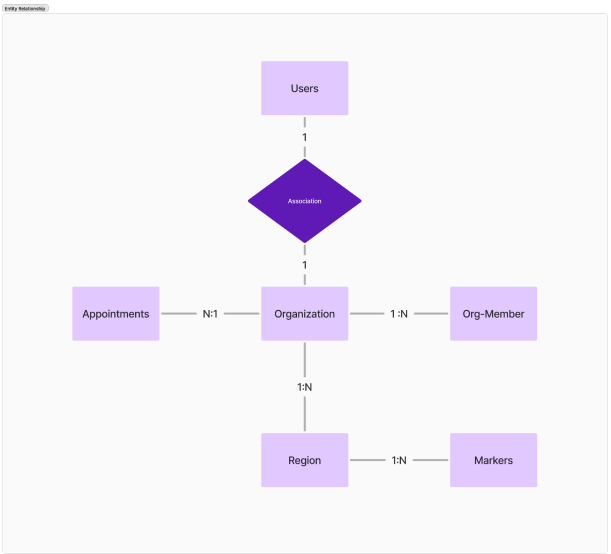
When designing this app I decided to use a “service-oriented architecture”. In a service oriented architecture you divide core features into unique services which contain individual business logic. When the frontend is created it then can call those services through providers. This follows the software design strategy of separation of concerns which encourages dividing a program into unique sections, each addressing a separate concern. The alternative approach is a design-first or monolithic approach which relies on a singular service with code designed around exactly what the front-end needs. I decided a service-oriented architecture was best for my case because I knew that each service would likely use vastly different API’s and libraries. Additionally, for redundancy, I didn’t want bugs from one service to accidentally influence another. In this architecture the flow of information through the app is then as follows:

Core business functionality, such as database’s, API’s, or authentication are written as services, divided into the following categories: Database, Authentication, Mapping, Scheduling, Payments. The functions from these services are then called by a set of provider classes on the frontend which verify the data model with strict-typed model classes. These providers are

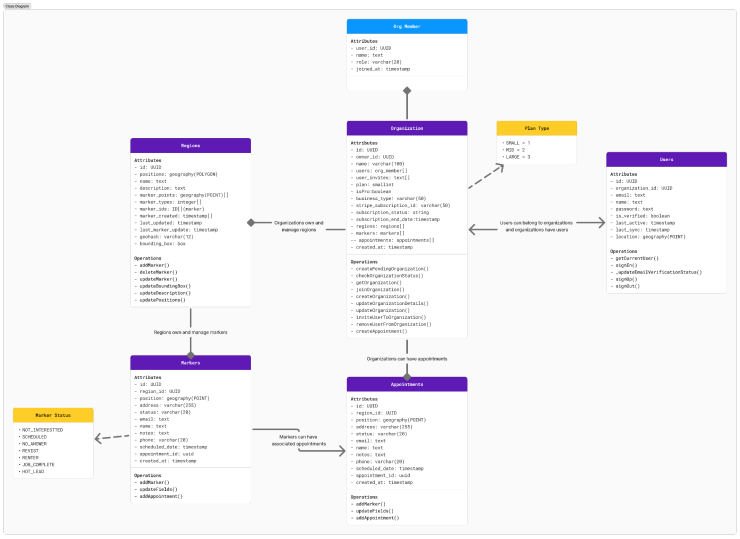
implementation agnostic in regards to the services they call. This means that the providers “provide” a set of public functions that the frontend can call without caring about the underlying database or authentication service, for example. This prevents bugs from propagating throughout the app when updating the backend service classes (for example if switching authentication from one library to another) because the frontend will only interact with the services through a provider which has a strict set of functions and data they provide. Finally, specific screens in the app asynchronously watch these provider classes, following the observer design-pattern. This means that if the state of the provider is updated, all its dependents (the "observers") are notified and updated automatically. This is especially useful when working with authentication states where you want to track updates to the user model throughout the app to allow or deny access to certain pages.

Knowing what type of architecture I wanted to use, I now had to understand what data the user would need, how to model and store it, and what the user-flow would look like. I will now go through each of the diagrams that contributed to the software design of the app, and ultimately influenced the decided tech-stack.

When first designing software I always like to have a solid understanding of what data I need to actually be interacting with. This means before creating classes and functions and files, just writing out exactly what data you want to have access to and store. When writing out data models, list the data needed without worrying about relationships or implementation. In this case data models for organizations, users, and appointments were created as lists.

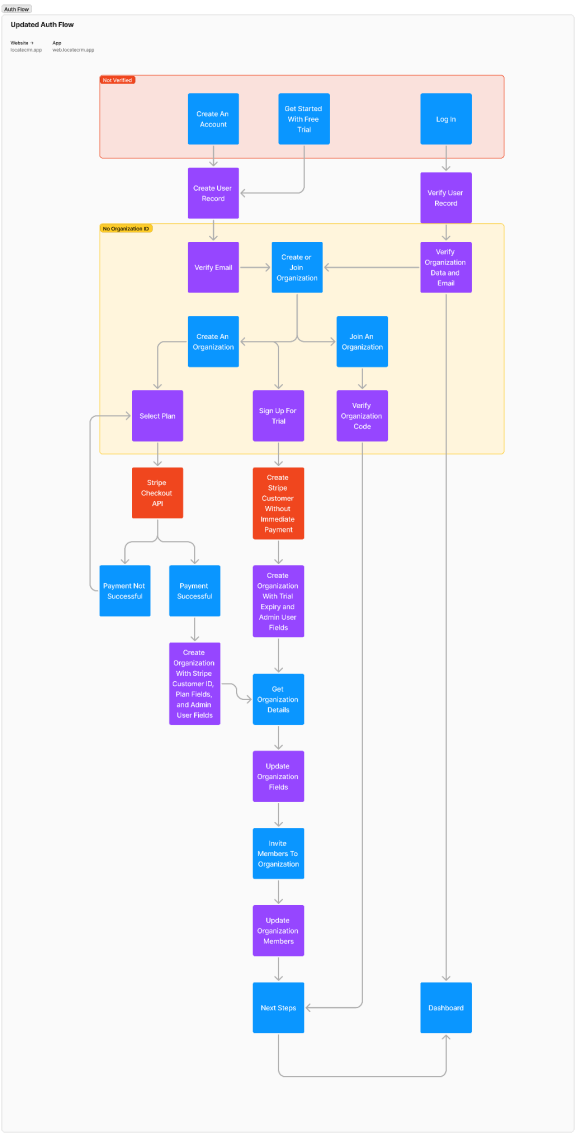


After listing the data models it is helpful to start to picture the relationships different data structures would have. The best diagrams for this analysis are entity relationship diagrams. Here, connections are mapped out between the key data for the app.



After knowing the relationships I created class diagrams to think about how a real implementation of this architecture could look like in code. Class diagrams are the most useful once a good understanding of the project and implementation are already complete because it dictates how the code is written.

Lastly, once what data is needed and the relationships between them are understood one can then start thinking about how the UI/UX design incorporates. The most helpful tools for that are user-flow diagrams and UX designing.



This user-flow diagram outlines the entire authentication process for the user. As shown in the data model above, the user is linked to an organization which is then then linked to a payment plan. This diagram starts at the login/signup screen and then follows the progression of creating an organization, payment through a 3rd party like stripe and then finalizing organization details like inviting members, verifying subscription, and customizing fields based on business type. The red box represents the pages a user is able to access if they don't have an authed user, and the yellow represents the pages a user is able to access if they have an account but haven't setup an organization yet. The blue boxes represent front-end pages, while the purple and red boxes represent backend-service calls that need to be made before the user can flow to the next page. This diagram was important in understanding exactly what pages to implement.

Conclusions:

Based on the diagrams above I was able to more easily decide the tech stack and layout of the codebase with a comprehensive understanding on what data structures and services were needed. I am using Flutter for the mobile and web app because of its thorough cross-platform support. This was based on user-requirements which needed access from both a browser and through an app on their phone. It also has a strong selection of libraries it integrates with such as stripe for payments or any database or authentication. I ended up going with supabase for both the authentication and database service which uses postgreSQL under the hood. I chose this based on the data modeling of markers and regions because postgreSQL has a plugin “postGIS” which extends the capabilities to better handle geospatial data. Most of the classes would store some component of geospatial data I would need to query. It also allows for easy support of partitioning and inheritance which are crucial when querying thousands of markers and customer data. Beyond design constraints, it is also cheaper and similar to setup than options like AWS or Firebase, is open-source and is able to be self-hosted. I ended up choosing Mapbox for the mapping service based on the entity-relationship diagrams because it allows for the best flexibility with displaying regions and markers. Furthermore, it easily integrates with flutter and also is much cheaper than any alternatives.

Implementation - AdafruitGFX:

Introduction:

The goal of this paper is to describe the implementation process for AdafruitGFX, a visual editing tool (like microsoft paint) that translates designs directly into code compatible with Arduinos and other microcontrollers.

Background:

In my freetime I have always been interested in working with microcontrollers and microcomputers to automate and improve things around the house. Unfortunately, as someone who cares about the design of things I constantly ran into a problem, any display I wanted to connect to an Arduino and use had a separate graphics library and dimensions and even worse there was no easy way to test how display code would look unless you waited for it to compile.

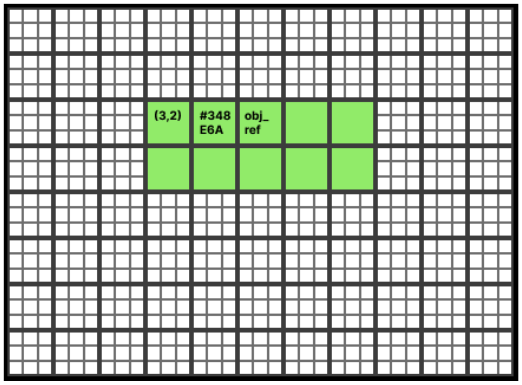
Then, for every slight adjustment you wanted to test out you would have to recompile all over again. If you ever made the decision to change displays you would have to write the code all over again accounting for the new dimensions, not knowing if everything would fit or look good. That's where I came up with the idea for the Adafruit Visual Editor, which is based at its core on the AdafruitGFX library written in C and is the most popular library for displaying graphics on small displays. This tool would allow users, such as myself, to test how different shapes and text looked on different sized displays instantly and also generate boilerplate code for the users to copy into their editor of choice (including the unique library for each display). No more headaches about size, formatting, or waiting minutes just to get an idea on how something would look. In order to accomplish this I had to utilize everything I learned in my data structures and algorithms class at MSU to write reliable data structures for the shape objects and algorithms for the drawing of shapes.

Implementation:

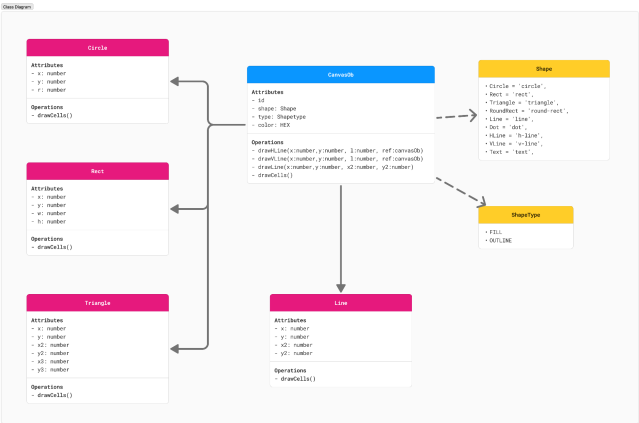
The front-end library used was Sveltekit because it allows for writing native Typescript and HTML out of the box with the benefit of a component-based architecture. Component-based architecture is the methodology behind all front-end libraries and allows developers to divide a website up into reusable "components" which can then be implemented and reimplemented anywhere needed throughout the app without having to rewrite code. Components can be as simple as a Button or a Title, or as complicated as an Accordion dropdown or in the case of this app the Canvas is represented as a component. This also allows for a separation of concerns and simple organization of code, meaning any simple logic can be contained within the component file.

However, for this project, the library chosen for the front-end does not matter. This project could easily be reproduced in any library including just simple HTML and Javascript so for the remainder of this paper I will focus on the implementation of the actual data structures and algorithms needed.

The basis for the implementation of this project was inspired by prior work done by the team at Adafruit who created the most popular graphics library for Arduinos, Raspberry Pi's, or any other microcontroller. This library was originally written in C and is heavily documented online so reverse engineering the data structures was just a matter of putting in time. In my web app, the



canvas users interact with is a “virtual canvas” built on top of the actual canvas that HTML provides natively for drawing shapes. The virtual canvas consists of an array of “Cells” which are made up of the following parameters; x cord, y cord, size(w=h), color, and a reference in memory to any object that occupies the cell. These cells are then spread over the top of the HTML Canvas and act as the basis for any Adafruit GFX shape drawing. When a shape is drawn, the shape is stored in memory and the cells are updated to reference the object if the object is drawn over the respective cells. To display to the user the canvas simply just redraws the list of cells that have been updated with the corresponding color. To create code is as simple as taking the list of objects stored in memory, which the classes themselves are based around the Adafruit library, and then creating the respective line of code. Each shape is a class with parameters and inherits from a parent CanvasOb class which provides simple utility functions for drawing lines and parameters for defining shape and fill type.



For example, one shape that was implemented was a triangle. The triangle has 6 parameters on top of the CanvasOb class which are: x1, x2, y1, y2, x3, and y3. In code the triangle class extends the object and calls super in order to inherit its base parameters and methods. Triangle also implements a method drawCells (like every other shape) that defines the algorithm for what cells are painted in and updated.

A feature that was crucial for a paint replicate is undo and redo buttons. To accomplish this a Linked List was used to constantly keep track of the canvas state and store old canvas states.

Another interesting implementation is a bitmap text algorithm that stores the text style in bytes. This works by representing each character by a 5x7 grid of pixels. Each pixel in that grid is either ON (1) or OFF (0). When converting these 1’s and 0’s to on pixels and off pixels this

creates a simple bitmap pattern for each letter. By reverse engineering the code from the original C library, the visual editor is able to show a full font system.

Conclusions:

The implementation of AdafruitGFX is largely based on a core understanding of basic data structures and algorithms, as demonstrated above.

Debugging - Red Trade

Abstract:

The goal of this paper is to describe the debugging process for Red Trade, a financial visualization startup with the goal of providing advanced and customizable financial analytics to everyday retail investors

Background:

This project was an ongoing project from the years 2020 to 2023 constantly evolving as I learned more about financial visualization and software. It was ultimately written in Sveltekit for the front-end, node.js for the backend, and used data provided by TD Ameritrade's developer API. Throughout the process I had to do extensive debugging with both the API and my own backend code. In this paper I will describe the process I used when attempting to debug parts of the app and the API that was provided.

Debugging:

Debugging for full stack applications is complicated because there are separate debugging tools required for each part of the tech stack. On the front-end one may use .., …, or .., while on the backend or

Bibliography:

https://www.educative.io/answers/ssr-vs-csr-vs-isr-vs-ssg

https://aws.amazon.com/compare/the-difference-between-soa-microservices/