COSC 310 Software Engineering

MILESTONE 5
Final Report & Delivery

Due April 12, 2024

gitGoblins (L01)

Eric Harrison
Ian Steyn
Joshua Ward
Chase Winslow

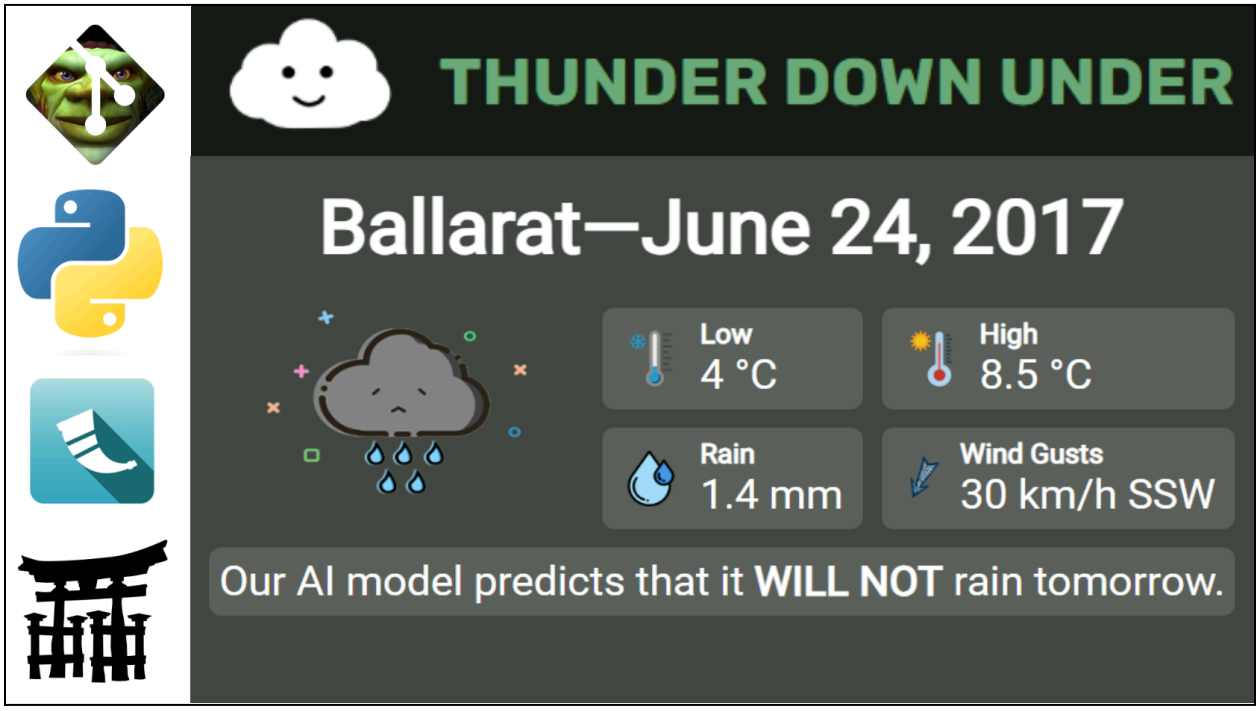
Links:
GitHub Repo
GitHub Dashboard

Table of Contents

Introduction to the Project

Video Walkthrough

A full walkthrough of the project, including a look at all of the features and some information on the project's structure, database, testing, and management practices can be found here.



Installation and Hand-over Documents

Please see the file `installation_guide.md` in our GitHub repository to learn how to install, run, test and navigate our application.

Updated Requirements Document

Note: Updated sections are highlighted in green

Project Description

- Project Type: IoT Sensor/Monitor Dashboard
- Dataset: Rain in Australia

We are developing a web-based dashboard for Australian weather data. The dataset we are using contains meteorological sensor data from dozens of weather stations all over Australia, measured between 2007 and 2017. It includes data for rainfall, sunshine, temperature, wind, clouds, humidity and pressure.

This is a student project built for the purpose of learning, and as such our application only "pretends" the data is live, by allowing users to select a date and location, and then giving them "current" information for their selected location. This information is displayed in various interactive forms: a general weather summary, a map with graphic overlays, and four graphs which compare past and current data. The app also predicts whether it will rain the next day based on current weather conditions.

The app is directed towards Australian locals, to help inform them about the weather in their local area. To access the app, users must register with our login system. They can then also receive optional email alerts about severe weather conditions in their location.

User Groups

- General Australian citizens should be able to check on weather information for their home cities as well as the surrounding area.
- Meteorologists want to be able to see different weather visualizations and details to help them perform their jobs to the best of their abilities.
- Foreigners travelling to Australia should be able to view the weather for different locations and dates so they can plan the best possible vacation.
- Administrators will use the software to manage the weather database and provide important information to the users.

User Requirements

- **User Dashboard.** Users can access different parts of the system from a dashboard which contains four pages: Home page, Weather Summary, Graphs, and Map
- **Select Location.** Users can use the map or the Weather Summary's filtering to select any location in Australia, to then view weather data from the closest available weather station.
- **See Data Summary.** Users can access a summary of the most recent weather data for the selected location.
    - **See Rain Forecast.** The summary includes a prediction of whether it will rain the next day.
- **Visualize Data.** Users can visualize current and past weather data.
    - **Map.** Users can choose from different graphic overlays on an interactive map.
    - **Graph 1:** Compare Past Temperature Data. Users can select a single location to see a graph of its temperature data over a specified time.
    - **Graph 2:** Compare Current Data. Users can select multiple locations to see a graph comparing the current weather data of those locations.
    - **Graph 3:** Compare Past Wind Data. See user requirements. Users can select a single location to see a graph of its wind data over a specified time.
    - **Graph 4:** Compare Past Rainfall Data. See user requirements. Users can select a single location to see a graph of its rainfall data over a specified time.
- **Filter Data.** For the daily summary and all three visualizations, users can choose which types of weather data they want to see (e.g. rain, humidity, or wind).
- **Login & Register.** Users must register with a valid email address to be able to log in to the system.
- **Receive Alerts.** Users can sign up to receive email alerts warning them of current severe weather conditions in their location.
- **Dark Mode.** Users can switch between light mode and dark mode.

System Requirements

Functional Requirements

- **Dashboard.** Acts as a home/start page for users. Provides access to all other parts of the system: data summary, visualizations, and location search.
- **Location Selection.** There are two location selection interfaces: the Weather Summary's filtering, or clicking on a point on the map. Data is provided for the closest weather station to the selected location.
- **Date Selection.** Since we are not actually using live IoT data, the system must provide a way for the user to select the "current" date. The system pretends that the data is live.
- **Data Summary.** This contains an easy-to-read overview of the most current available weather data for the selected location, including temperature, rainfall, humidity, wind direction & speed, pressure and cloud cover.
    - **Rain forecast.** The system runs a machine learning algorithm on past weather data to predict whether it will rain the next day.
- **Visualizations.**
    - **Map.** The system provides functionality to zoom in and out, select locations, and choose from different graphic overlays on the interactive map. The overlays are icons attached to each location, with some attribute (colour, number, shape etc.) that tells you something about the data.
    - **Graph 1:** Compare Past Temperature Data. See user requirements.
    - **Graph 2:** Compare Current Data. See user requirements.
    - **Graph 3:** Compare Past Wind Data. See user requirements.
    - **Graph 4:** Compare Past Rainfall Data. See user requirements.
- **Filtering Data.** See user requirements.
- **Login & Registration System.** The system provides a way for users to register a profile with their email. This profile must be used to access the system.
- **Alert System.** Email notifications regarding current severe weather conditions or system issues are sent to users who have opted in to the alert system.
- **Database.** The system will get data from the dataset specified in our description, but this will be implemented using a database.
- **Administrator login.** There is a special administrator login. An administrator can:
    - Manually modify faulty data.
    - Manually send notifications through the alert system, regarding regional system issues.
- **Dark Mode.** The system offers both a light and a dark mode that users can switch between.

Non-functional Requirements

Product Requirements

- **Web App.** The system must be a web application.
- **Good Performance.** Search results should be returned within a maximum of 3 seconds.

Organizational Requirements

- **Programming Tools.** The system must be developed using Python and Python-adjacent libraries and APIs. This is because TA/instructor support is available for these tools.
- **Development Tools.** The system must be developed with appropriate reliance on the following team-based tools: Git, Github, and Google Drive.
- **Due Dates.** The various components of the system must be completed by the milestones specified in COSC 310, with the final due date being April 12.

Miscellaneous

- **User Information.** The system will not collect any user information, except for their email for the notification system.
- **Data limitations.** The system is limited to Australian weather data from 2008-2017. Domain Requirements (Standards)

Domain Requirements

- **Accessibility.** The system will implement at least basic web accessibility features, as specified in WCAG2 (Web Content Accessibility Guidelines). Most relevant:
    - 1. Perceivable. Especially:
        - 1.1 Text alternatives.
        - 1.4 Distinguishable.
    - 2.4 Navigable.

Status of the Software Implementation

Delivery of Requirements

List of Initial Requirements Met

How many of your initial requirements that your team set out to deliver did you actually deliver (a checklist/table would help to summarize)?

System and User Requirements:

- ☑ **Dashboard:** Users have a home page and can access different parts of the system from a navbar.
- ☑ **Location Selection**: Users can select a location on the map or on the Weather Summary page's dropdown selector.
- ☑ **Data Summary:** This contains an easy-to-read overview of the available weather data for the selected location and date.
- ☑ **Rain forecast:** The system runs a machine learning algorithm on past weather data to predict whether it will rain the next day.
- ☑ **Map**: The system provides functionality to zoom in and out, select locations, and choose from different graphic overlays on the interactive map. The overlays are icons attached to each location, with a weather-related icon that describes the current weather at the given station.
- ☑ **Graphs:**
    - ☑ **Graph 1, 3, 4:** Users can select a single location to see a graph of its wind, rain, or temperature data over time.
    - ☐ **Graph 2:** [unimplemented] Users can select multiple locations to see a graph comparing the current weather data of those locations.
- ☑ **Filtering Data:** For the graphs, users can choose which types of weather data they want to see (e.g. rain, temperature, or wind). They can also filter specific traces on the graphs. For all the feature pages, users can choose a city and date or date range.
- ☑ **Login & Registration System:** The system provides a way for users to register a profile with their email.
    - ☐ **Password recovery:** [unimplemented] Users can change their password via an email confirmation.
- ☑ **Alert System**: Email notifications regarding current severe weather conditions or system issues are sent to users who have opted into the alert system.
- ☑ **Database:** The system populates a database with our dataset.
- ☑ **Administrator login**: An administrator can:
    - ☑ Manually modify temperature data.
    - ☑ Manually send notifications through the alert system, regarding regional system issues.

Non-functional and Domain Requirements

- ☑ **Web App**: The system is a web application.
- ☑ **Good Performance:** Results are returned within a few seconds..
- ☑ **Accessibility.** The system has good contrast, all images have alt text, and buttons and form inputs are keyboard focusable.

Commentary on Meeting Requirements

We didn't deliver on every initial requirement, but this was mostly due to us changing our plans based on what made more sense for the actual project (rather than us failing to implement the things we'd planned on implementing). Throughout the project, we made decisions to expand on some features and move on from others.

During the planning process, we were given a list of requirements that our system had to have, and we were forced to update our initial plans. After updating our plans with the newly required features we felt our website did have all the requirements for this project. In general, as we worked on our project, we made small changes to our requirements so features could be better implemented. With these changes, we focused on our updated features still achieving every aspect of the required features of the system. As a result, our final project most certainly did capture sufficient details needed.

Additionally, we added a feature that wasn't in our initial requirements: **DARK MODE**!

Unimplemented Features

Current Day Comparison or "Multi-location" Graph
We decided to not implement a multi-location graph, which would have allowed users to compare weather data from two or more locations on the same day. This feature was not implemented mainly due to time; the team chose to spend the last 2 weeks polishing a high-quality final product instead of adding new features. We also already had over 3 visualizations, so it just made sense to cut.

Password Recovery
We were originally planning on using the email framework to allow users to recover lost passwords through email. However, as we got closer to the deadline for the project we realized we didn't have the time to fully complete this feature, so we decided against implementing it.

Partially Working Features

The project finished with no *partially* working features. This is due to our group prioritizing our time management, so we ensured that for any features we started working on we would be able to finish entirely.

Backlog

How many tasks are left in the backlog?

There are no tasks left in the backlog! This is due to the fact that we completed all of our main features. There were a few features and issues that were in our GitHub that we did not complete, but that is because we removed them due to the features no longer fitting into our application. As we worked on our system and discovered more about our process and software, it evolved and changed and some of our initial ideas for the application no longer made sense to add, so they were removed. Unimplemented features were closed as 'not planned' issues on our GitHub dashboard, weeks before delivery was due.

System Architecture

What is the architecture of the system? What are the key components? What design patterns did you use in the implementation?

System Overview

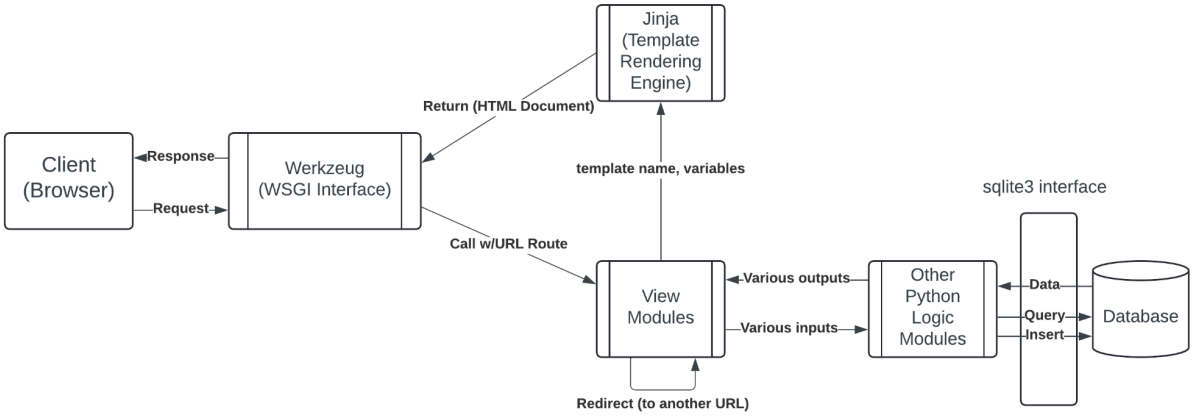
Most of our back-end code logic is written in Python, and we used the Flask framework to glue all the other tools together and run an actual application. The front end is "vanilla" CSS/HTML/JavaScript; that is, we wrote custom classes and functions instead of using a front-end framework.

Key components

- **Database** - a local SQLite database that needs to be initialized before you run the app for the first time
- **Flask application** - an instance of this is created by the app factory when the app is run, which configures it, connects it to the correct database, and runs it on a local server
    - **Werkzeug WSGI interface** - handles client requests and responses and translates them into Python-usable objects and function calls.
    - **Templates and the Jinja rendering engine** - HTML templates that contain some Jinja logic; rendered as actual HTML documents when called from the view functions
    - **Python View Modules** - Modules that contain view functions, which run when their corresponding URL is accessed. Includes `views.py`, `auth.py`, `admin.py` and `settings.py`
    - **Python Logic Modules** - Modules that contain various backend logic functions, called by the views. Includes `queries.py`, `graphs.py`, `predictions.py` and others.

Understanding the Architecture: Data Flow

The best way to understand the overall system architecture is to understand how data moves between the key components. Below is a simplified data flow diagram for the application.



Internally, JavaScript functions cause the browser to have visual changes, new URL accesses, and form input submissions. The latter two are what will cause an HTTP request to the application. Werkzeug turns the request into an object that can be handled in Python, and it is then routed to a view function based on the accessed URL.

The relevant view functions typically extract some information from the HTTP request and call various functions from other modules with that information as inputs. However, they may also redirect to another view function, either because of the authentication process or because the “view” is really just an in-between step for other views.

The other logic modules do lots of different things, but most of them interact with the database via the SQLite3 interface (whether directly or indirectly), either to insert new data or to query data that is needed to perform an action. Sometimes the action performed terminates here (such as when notifications are sent by `notifications.py`), but much of the time these modules return something useful to the view function (such as the `graph_html` object from `graphs.py`).

In any case, the user must be left with something to see, so any process will eventually redirect to a view function that calls `render_template`. This asks the Jinja rendering engine to turn an HTML template into an actual HTML document, often using variables that the view function got from the other Python modules. This document gets passed back to Werkzeug, which passes it back to the client as a response, where it is displayed as a web page in the browser.

Other Major Tools

This section details some of our features, the major tools/libraries they used and how they fit into the system architecture.

- **Predictions—scikit-learn:** When the database is initialized, a prediction model is also initialized and trained on our dataset, using a `RandomForest` machine learning algorithm. Given a date and city, the model predicts whether it will rain the next day based on that date's weather data.
- **Graphs—Plotly:** Graph logic is all written in the back-end Python module `graphs.py`. Graphs are typically retrieved as a string of HTML that can be passed directly to the Jinja template, where they are rendered as interactive SVGs.
    - **Pandas:** Both scikit-learn and Plotly functions require Pandas `DataFrame` objects as inputs. See the Adapter design pattern below.
- **Notifications—Flask-mail**: A Flask-mail `mail` object is initialized with the app in the app factory. The back-end Python module `notification.py` then uses this object to send emails to users subscribed to a particular city. All emails are formatted according to an HTML template.
- **Authentication—Flask-login and Bcrypt:** Flask-Login is a Flask extension that provides user authentication and user session management. Bcrypt, used in `auth.py`, is a password-hashing function that provides a secure way to hash passwords by incorporating salt and cost parameters. These parameters protect from brute-force attacks on our user login.
- **Map—Leaflet.js:** Uniquely, the map is handled completely in the front end, since Leaflet is a JavaScript module. The map is shown by binding a leaflet element directly into an HTML file. Leaflet is then able to add layers and clickable markers to this element, allowing the map to be interactive.

System Design

Design Patterns

- **Factory:** The app instance is initialized with an app factory called `create_app` in `__init__.py`. The reason this needs to be a factory instead of just a constructor is because different contexts require different configurations for the app—for example, we use a different database for testing the app than for running it normally.
- **Facade:** Many of our Python logic modules are essentially facades for more complex libraries; they help us use only what we need. Examples:
    - `queries.py` is a facade for database queries (more on this in 'Design Flaws').
    - `graphs.py` contains facades for `plotly.express`, which is itself a wrapper module for the lower-level `plotly.graph_objects` module. `WeatherGraph` objects are wrappers for Plotly `Figure` objects which initialize specific figures with the settings and data we need.
- **Adapter**: The chief examples of this pattern are in `predictions.py` and `graphs.py`, since they have functions to convert SQLite3 queries in Pandas dataframes.

Design patterns, cont'd

- **Decorator:** There are two main uses of decorators in the app:
    - *Requiring Login:* Certain view functions have a `@login_required` decorator. This forces a redirect to the login page if that view receives a request and no user is logged in.
    - *URL Routing:* Each view function has a decorator in this form: `@some_blueprint.route('some_url')`. This is how the app is able to direct a request to the appropriate view function.

Design Strengths

- **Naming conventions.** We decided on naming conventions for our files and Python code early on and we stuck by it really well (and if anyone didn't, it came up in reviews).
    - Files: `file_name.extension`
    - Packages: `packageName`
    - variables: `variable_name`
    - Functions:`function_name()`
    - classes: `ClassName`
    - constants: `CONSTANT_NAME`
- **Modularity and Single Responsibility Principle.** We worked hard to move large sections of code out of the view functions and into separate modules with their own responsibilities. There is some room for improvement here, but overall modules are well-organized.
- **See section 'Code Reuse' below**

Design Flaws

Most of our major design flaws are a direct result of this being the first time we've worked on such a project. We believe we planned well based on what we knew, but a lot of planning didn't happen simply because of a lack of knowledge and experience. We were designing and implementing the system at the same time as we were learning dozens of new tools, libraries, formats and processes. This led to many flaws in our design paradigms. However, we don't regret our process. There is still so much to learn, but this project has given us an idea of some of the things we *didn't know* we didn't know, and that will help us improve our design in the future.

Below are some specific case studies of design flaws in our system, and how we worked around them or would fix them if we were to do it again.

Front-end stuff in the Back-end: Graphs and dark mode
While the fact that the graphs module returns a simple HTML string is convenient, it made it hard to put graphs into dark mode (since that feature is all handled in the front end). We solved this by always passing both a dark and light mode graph to the front end, then having a JavaScript function decide which to hide and which to reveal.

Encapsulation problems: Dictionary Parameters
Many functions and templates expect specific dictionaries with specific keys as parameters. This was often chosen to eliminate the 'long parameter list' code smell, which improves readability. However, the issue with this is that it requires different modules, as well as the front and back end, to have intimate knowledge of each other. Proper documentation in functions and templates makes this easier to achieve, but still—ideally we would use custom objects with getters and setters instead of dictionaries.

Coupling Issues: Database Interactions
Because we had various levels of experience working with SQL, and only learned about database models and the model-view-controller design pattern late into the project, we had some trouble settling on a consistent way for view functions to interact with the database.

We were already using SQLite3 to be able to interact with the SQL database in Python, but we tried to add another layer of abstraction with the `queries` module. The idea was that any SQLite3 query scripts would be written exclusively in Python functions in this module, which other Python modules could then call. This was especially helpful because the results of SQLite3 queries often needed a few more lines of code to extract useful Python variables. For example, the weather summary page requires all the information for the weather of a certain day and location, so we wrote a method called `get_weather_data` that returns a dictionary full of weather data.

However, there were multiple problems with this approach:

- If the query had to be altered, it needed to be done inside `queries.py.` This happened often, removing the point of encapsulating it there in the first place.
- There was no similar abstraction layer for inserting information into the database, leading to insert scripts/functions being scattered across various modules, including ones where they didn't belong.

This led to the following weird module coupling:

- Views interact with the database in 3 different ways: abstractly via queries.py, indirectly via other Python logic modules, or directly via SQLite3 scripts.
- Most logic modules interact with the database in 2 different ways: abstractly via queries.py or directly via SQLite3 scripts

If we were to change how we did this in the future, we would plan a consistent way of interacting with the database from the start; most likely through a model.

Code Reuse

The Good

The team was able to achieve a high level of reusability in most of our code.

- HTML templates use Jinja template inheritance as well as Jinja macros (which are kind of like functions) so that we don't have to rewrite a ton of HTML boilerplate
- Many of our functions are tooled to be used throughout the system in different contexts, and many are used more than once. An example of this is our `queries.get_weather_data()` function that returns a dictionary containing most of the weather info from the database, and is re-used in a few places in our code.
- Some modules use abstract classes and class inheritance. For example, in `graphs.py`, the `WeatherGraph` abstract class defines the order of operations for initializing a graph, since they are similar for all graph types/
- We even have reuse in our test code, such as the test user authentication functions in `auth_actions.py`

The Bad

An area where we were unable to reuse a lot of our code was with buttons and form submissions. Our team's unfamiliarity with the required skills led to divergent software designs and practices. To retrieve the user's selection from a dropdown/form, some parts of our app rely on JavaScript and URL arguments, while other parts use HTML form submissions. This means that we have multiple modules that have selectors and submit buttons that all use unique implementations for the same task.

If our team had more experience before starting the project or had more time towards the end to clean up our program, we could have refactored the code, centralized the selectors for use on all of the pages, and avoided the individual implementation and the repeated code smell that currently exists in the software.

Known Bugs

There are only two known bugs in the system.

1. **Inconsistent city name spacing.** Some selectors and headers display two-word cities (e.g. "Wagga Wagga") without a space (e.g. "WaggaWagga"). This is a result of the various location selection methods mentioned above. The immediate solution would be to use functions like `add_space` to add a space, but the deeper problem and solution are described above in 'Code Reuse'.
2. **Double click for graph filtering not registered.** This bug only occurs on some machines. When viewing the graphs, it is normally possible to double-click on a trace colour in the legend to isolate that trace on the graph. However, on slower computers, this doesn't seem to work. We think that it has to do with the small time required between clicks for it to register as a double click, and are unsure how to fix it as we didn't implement the feature ourselves (it is built into Plotly graphs).

Team Reflections

Project Management

How did your project management work for the team? What was the hardest thing, and what would you do the same/differently the next time you plan to complete a project like this?

The Good

Overall, our team worked really well together. We maintained a consistent pace throughout the project lifecycle that led to a successful, complete project. We had healthy practices when it came to collaboration and GitHub project tracking. We made good use of our Kanban board, using it to keep track of our issue backlog, what was currently being worked on, and organizing what needed to be done by date, issue type, and milestone. We used GitHub pull requests extensively as a way to communicate and document changes, progress, and issues in our project. We also kept to a very consistent meeting schedule, both during lab time and outside of class hours, to check in with each other, to ensure that our issues were being completed on time, and to help each other out.

The Hardest Part

The hardest part was the lack of written feedback. We got almost no feedback on our practices until over halfway into the project, and by the end of the project, we still had important milestones ungraded.

What We Would Change

There were a few things that our group would have done differently and a few setbacks that hampered our progress.

- **Number of members.** In terms of setbacks, our group only had 4 members, as opposed to 5, meaning that we had fewer hours dedicated to the project, and we each had to put in more hours to get our software where we needed it.
- **Divergent Learning.** If we could redo the project, we would have accounted for our divergent learning earlier on by coming to group decisions on what tools to use (and how to use them) *before* implementing them. During the course of the project, everyone was learning how to use virtually every tool and language as we were implementing them. This led to each person finding different solutions for similar problems, so our project has duplicate code and inconsistent problem-solving in our architecture.
- **Coding/learning the tools earlier.** Even though it seems like we had a pretty good timeline compared to some groups, we think we should have started implementing some of our code earlier, especially the front-end code. We vastly underestimated the time and effort it would take to change and adjust the front end, so we would have allocated more time to account for it.

Requirements

Do you feel that your initial requirements were sufficiently detailed for this project? Which requirements did you miss or overlook?

We feel that our initial requirements provided a really good backlog of issues for us to tackle as we implemented features. It was notable that we rarely had to change issues due to poor initial requirements. With that said, there were still some requirements we originally missed or overlooked. For example, our original filtering requirements could've been much more precise, so we would have a better understanding of what the filtering should do. The requirements also overlooked the implementation of dark mode, which wound up being a pivotal feature of the application.

Planning

What did you miss in your initial planning for the project (beyond just the requirements)? Would you (as a team) deal with testing differently in the future?

As a group, one major thing that we missed in our initial planning was deciding which frameworks to use for all the major system processes. If we had planned ahead for the entire project, then each team member would've been able to learn how all of the frameworks work before starting, instead of learning throughout the process.

In terms of testing, if we had planned the testing framework along with the rest of the architecture from the start, it would have saved us some time figuring things out during the process. In the future, we would definitely implement a front-end testing framework like Selenium (as we found front-end testing difficult with pytest). We also could have been better about developing the software with a test-driven approach, by writing more tests before functions.

Efforts

If you were to estimate the efforts required for this project again, what would you consider? (Really I am asking the team to reflect on the difference between what you thought it would take to complete the project vs what it actually took to deliver it).

The biggest thing our team would need to consider is the additional time required to learn how to do everything. Almost every aspect of the development process involved using some software/framework that was brand new to the entire group. As a result, the project required a much larger amount of effort than we initially prepared for. In the future, we could've helped to mitigate this problem by dedicating the first few weeks of the course to learning the languages and frameworks we planned on using throughout the project.

The other aspect of the project that took much more effort than expected was everything surrounding the front end. We spent the majority of our time on the project working on the back end and didn't leave time to make sure the application looked good without having to do crunch-time coding. We discovered that CSS and HTML can be very finicky, meaning that even after we understood how they worked the front-end programming still took much longer than we expected.

Learning

What did your team do that you feel is unique or something that the team is especially proud of (was there a big learning moment that the team had in terms of gaining knowledge of a new concept/process that was implemented).

In terms of the development process, our group is very proud of how we used Git and GitHub. While there was a learning curve at the start of the course, by the end of it we were effectively using branches, issues, pull requests, code reviews, and the Kanban dashboard to ensure the project went as smoothly as possible.

In terms of features, our group is extremely proud of our interactive dark mode. Not only does it create a much better user interface and engagement, but it's also a very unique feature that allows our group to stand out when compared to the rest of the class. And come on, don't tell us you don't want to click that button on and off 42 times in a row.

Throughout the course, there were so many enormous learning efforts required for the entire team to gain knowledge in order to complete the project. This included tools for data handling, project management, testing, many libraries for the features, new programming paradigms, code editing tools, and even entire coding languages like Python, JavaScript, Jinja, CSS, and HTML.

[END OF REPORT]

[END OF PROJECT]