LocalOT

https://github.com/atcasimir/3354-TeamOne

Delegation of Tasks:

a. Aidan Casimir
   i. Project Scheduling
b. Lance Miller
   i. Test Plan
c. Josh Sebastian
   i. Comparison with Similar/Competing Products
d. Kayla Burks
   i. Conclusion
e. Satish Parajuli
   i. Objectives
   ii. References
f. Venkatasai Gudisa
   i. Presentation Slides
g. Azlaan shafi
   i. Cost,Effort, Pricing Estimation

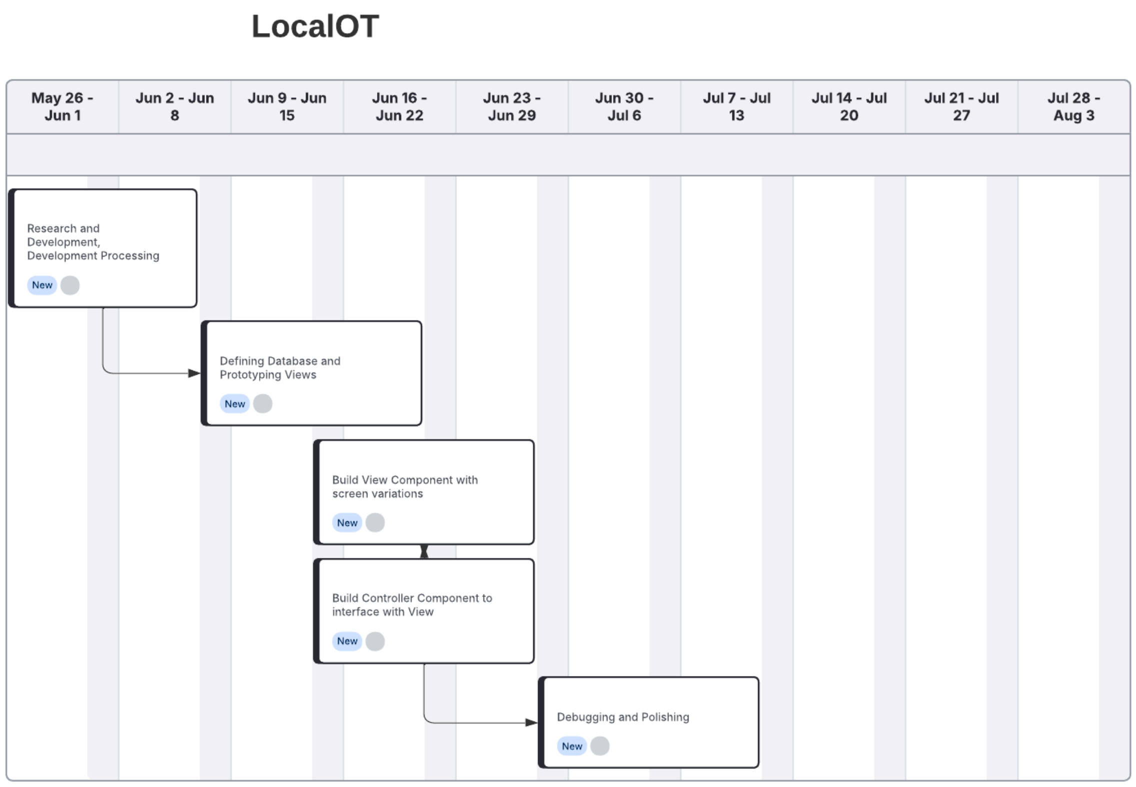
Project Scheduling:

To provide an example start and end date, May 26th would be the start and July 11th would be an approximated ending date, This would provide about 6 weeks more or less for the project to be completed in. Using the person months calculated in the following cost section, 4.3875 person months, aka roughly 4 people working full time for a month would be the estimated time to complete.

With this time frame in mind, the activities/tasks were divided up on a roughly weekly basis with R&D and Processing Planning being given about 1-2 weeks for proper completion. The 3 following activities would be in building the database (Model), screen/look and feel (View), and back-end interactivity (Controller). We structured the semantics of the project timeline around the Architecture Model, with the separate activities focusing on completing the different components. There are no major requirements that are dependent on previous tasks. The only tasks that must be completed in a specific order are the planning steps first, then all of the project implementation, followed by the debugging.

The Gantt Chart below does include weekends, and we did not account for weekends either in our person-months calculations, therefore in a weekend free schedule were to be considered, time would need to be shifted based on specified weekends and holidays. This current timeline does not account for weekends.

The number of working hours assumes an 8 hour work day, i.e. 9-5 or 8-4. This considers lunch breaks as part of the 8 hour work day. So realistically only about 7 hours of work would be completed every day on the clock.



<u>Cost, Effort, Pricing Estimation</u>:

We decided to go with the Application Composition Estimation Model for figuring out the cost of implementing this project. This model seemed to be the best at figuring out the total cost, because it was very easy for us to put our plan from the calendar into the points needed to calculate this formula. We calculated the total cost by figuring out what screens and reports we were going to use, and then measuring the difficulty of implementing each. In total this came out to 27 Object Points. We determined that around 35% of our code was reusable so we calculated the New Object POints as 17.55. Then assuming we had the lowest available developer experience with a productivity value of 4, we determined that it would take 4.3875 Person-Months to complete our project. That means around 4 people working full time for a month to complete our project. Below is the mathematical breakdown of this cost.

- i. Application Composition Estimation Model
    1. Screens:
        - a. Home calendar - 5
            - i. Month view - 3
            - ii. Week View - 1
            - iii. Day View - 1
        - b. Adding Events - 5
        - c. User settings - 5
    2. Reports:
        - a. Event scheduling output - 6
        - b. Event alterations confirmation - 6
    3. Object Points = 3 * 5 + 2 * 6 = 27
    4. New Object Points = [(27) * (100 - 35 (% reuse))/100] = 17.55
    5. Developers Experience % Capability aka PROD = 4
    6. Effort = 17.55 / 4 = 4.3875 Person-Months

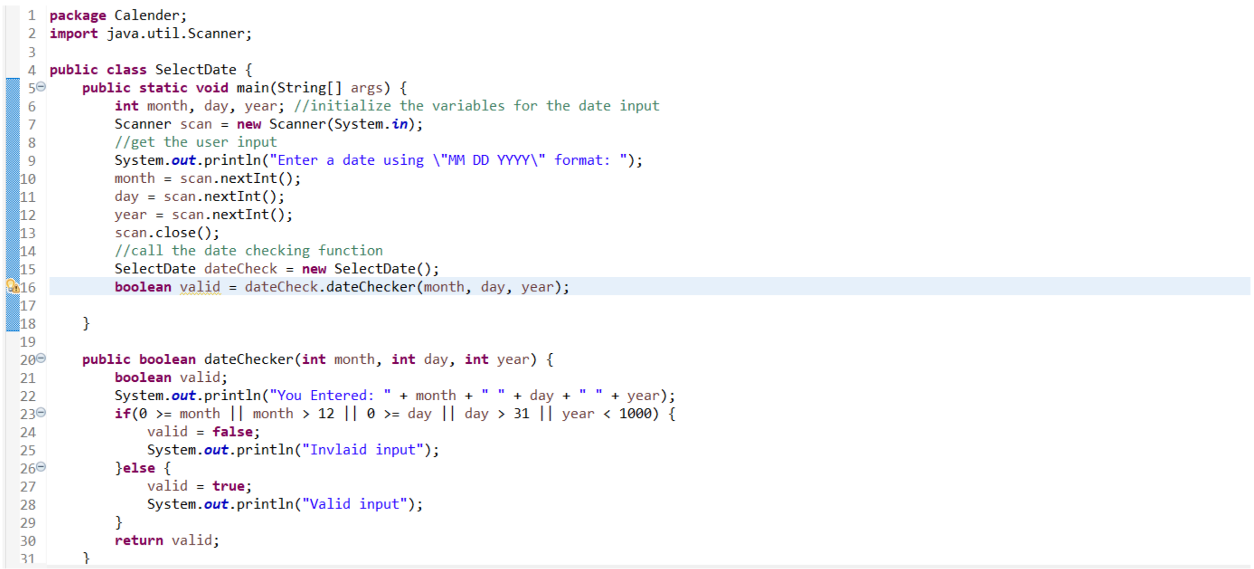
There is no estimated cost of hardware products. Everything will be locally run on the user's computer, so there is no need to have servers. For software, we only use completely free open source software such as SQLite, so there is no cost.

For this project, we would like to have 3 people work on it for about 4-6 weeks. We would ideally have one senior developer and 2 junior developers or interns working on it. Assuming a competitive working salary of $85,000/year for the senior dev and $20/hour for the interns, we can calculate that the total maximum cost for a 6-week project would be $14,607.69.

<u>Test Plan</u>:

- a. Describe the test plan for testing a minimum of one unit of your software:
    - For our test plan we will be testing the entering a date function of our app.
    - We will test this function using unit testing, specifically the java junit in this case.
    - These tests will have a positive test and a negative test to show it functions as intended.

b. As evidence, write a code for one unit (a method for example) of your software in a programming language of your choice, then use an automated testing tool (such as JUnit for a Java unit) to test your unit and present results:

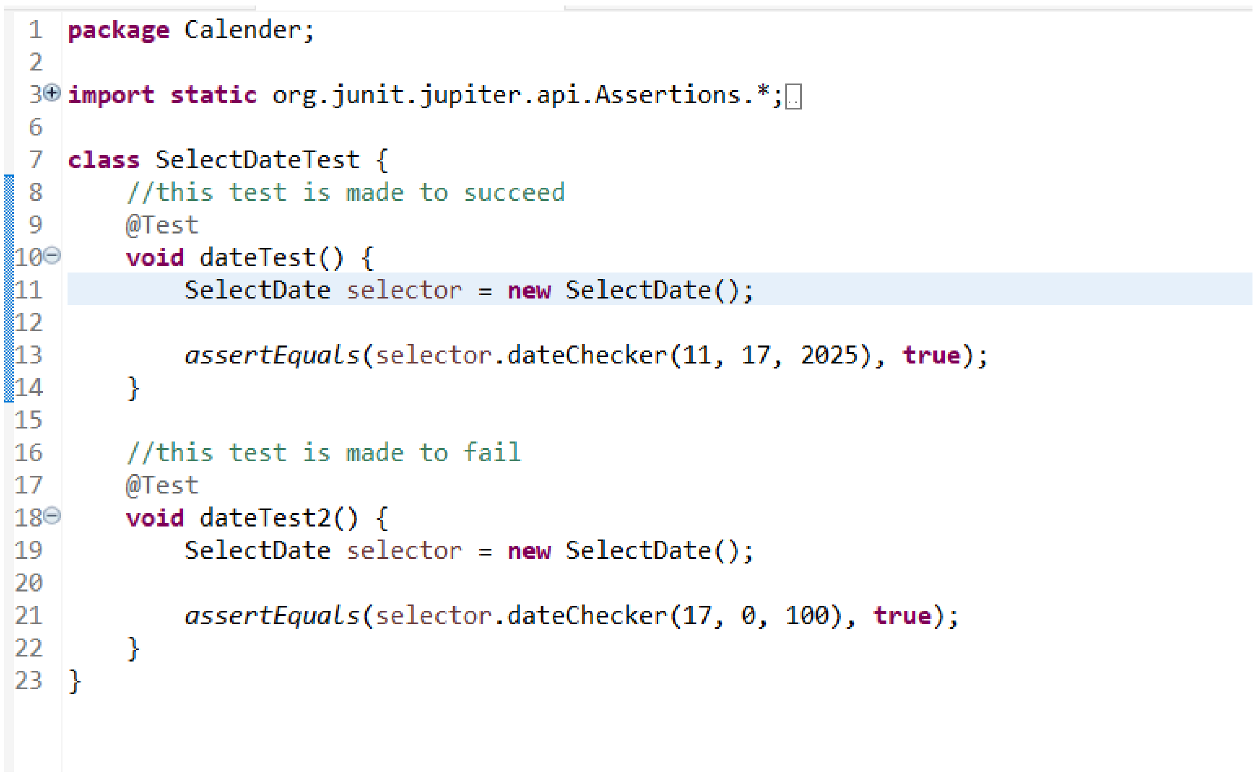


```java
package Calender;
import java.util.Scanner;

public class SelectDate {
    public static void main(String[] args) {
        int month, day, year; //initialize the variables for the date input
        Scanner scan = new Scanner(System.in);
        //get the user input
        System.out.println("Enter a date using \"MM DD YYYY\" format: ");
        month = scan.nextInt();
        day = scan.nextInt();
        year = scan.nextInt();
        scan.close();
        //call the date checking function
        SelectDate dateCheck = new SelectDate();
        boolean valid = dateCheck.dateChecker(month, day, year);

    }

    public boolean dateChecker(int month, int day, int year) {
        boolean valid;
        System.out.println("You Entered: " + month + " " + day + " " + year);
        if(0 >= month || month > 12 || 0 >= day || day > 31 || year < 1000) {
            valid = false;
            System.out.println("Invlaid input");
        }else {
            valid = true;
            System.out.println("Valid input");
        }
        return valid;
    }
```



```java
package Calender;

import static org.junit.jupiter.api.Assertions.*;

class SelectDateTest {
    //this test is made to succeed
    @Test
    void dateTest() {
        SelectDate selector = new SelectDate();

        assertEquals(selector.dateChecker(11, 17, 2025), true);
    }

    //this test is made to fail
    @Test
    void dateTest2() {
        SelectDate selector = new SelectDate();

        assertEquals(selector.dateChecker(17, 0, 100), true);
    }
}
```

c. Clearly define what test case(s) are provided for testing purposes and what results are obtained. (Ch 8):

- We designed a test case designed to see if the function returns true when given a proper date.

- We designed a test that will fail on purpose since it will give the code an impossible date.

<u>Comparison with Similar/Competing Products</u>:

a. Google Calendar [2]:
    i. **Strengths:** Unbeatable integration with the Google ecosystem (Gmail, Google Meet, Maps). Excellent search functionality and AI-driven features like "Find a Time." It excels in collaboration and sharing.
    ii. **Weaknesses:** While it has location features, it lacks the explicit proximity-based alerts and the high-level of user data control you are planning.
    iii. **Architectural Insight:** It operates on a massive, highly distributed, proprietary client-server architecture, far more complex than a simple MVC.
b. Microsoft Calendar [3]:
    i. **Strengths:** Best for corporate environments. Excellent for managing meeting rooms, complex organizational scheduling, and integration with email/Teams. The desktop and mobile apps offer robust offline capabilities.
    ii. **Weaknesses:** Can be overly complex for a simple personal user. Integration outside of the Microsoft ecosystem is less seamless than Google.
c. Apple Calendar [4]:
    i. **Strengths:** Seamless user experience and integration with Apple's devices and services (Siri, Mail, Maps). Strong emphasis on local device privacy.
    ii. **Weaknesses:** Heavily tied to the Apple ecosystem; non-Apple users cannot easily adopt it. Its features are generally more basic than Google or Outlook.

Feature	**LocalOT**	**Google Calendar**	**Microsoft Outlook Calendar**	**Apple Calendar**
Architectural Pattern	MVC	Distributed, proprietary (client-server)	Distributed, proprietary (client-server)	Distributed, proprietary (client-server)

Views	Daily, Weekly, Monthly	Day, Week, Month, Year, Schedule	Day, Work Week, Week, Month	Day, Week, Month, Year
Conflict Detection	Yes	Yes	Yes	Yes
Location Data	Detailed (Address, Map Hyperlink, Proximity Alerts)	Yes (Address/Map link)	Yes (Address/Map link)	Yes (Address/Map link)
Hyperlink Embedding	Yes (Location/Event Description)	Yes (In description)	Yes (In description)	Yes (In description)
Third-Party Integration	External Holiday API	Extensive (Gmail, Maps, Meet, etc.)	Extensive (Teams, Exchange, etc.)	Moderate (Mail, Maps)
Offline Mode	Planned (Environmental Req.)	Limited/Cached Access	Yes (Desktop/Mobile Apps)	Yes (Desktop/Mobile Apps)
Ethical/Security Focus	Moderate (Standard TOS/Privacy)	Moderate (Standard TOS/Privacy)	Moderate (Standard TOS/Privacy)	Strong (Strong emphasis on user privacy)

Development	HTML/Java or Python/MySQL	Proprietary Stacks	Proprietary Stacks	Proprietary Stacks

In summary, while the project's basic functional requirements match those of competitors, our advantage lies in superior QoL features (proximity alerts) and a commitment to performance (e.g., "Each calendar view must load within 3 seconds") that could appeal to users that want an alternative to the feature-loaded main competitors.

<u>Conclusion</u>:

During the initial stages of the project, our team intended to implement LocalOT in order to show its completion of the functions listed. Over time, however, we decided against implementation due to time constraints. This change led to a redelegation of tasks because we no longer, for instance, had to use UI tools or design a relational database as mentioned in the project proposal.

In addition to this change, we revised the requirements to use the correct verbs 'must' and 'shall' as was brought to our attention in the feedback from the first deliverable. We also made the requirements atomic with each having a single, testable behavior. For example, the functional requirement "A user shall be able to add/delete event categories and choose a color to differentiate event categories," was divided into separate requirements. This was done in order to adhere to the proper rules regarding writing requirements.

In adapting to these changes and completing the deliverables, our team was able to create a thorough project outline. By detailing necessities such as project scheduling and cost, we made the development of LocalOT clear and comprehensible; this makes future development not only possible, but entirely achievable.

<u>References</u>:

a. Please include properly cited references in IEEE paper referencing format. (You may see a referencing example in the sample IEEE paper in URL:

https://ieee-dataport.org/sites/default/files/analysis/27/IEEE%20Citation%20Guidelines.pdf

b. It means that your references should be numbered, and these numbers be properly cited in your project report.
   [1] GeeksforGeeks, "Application Composition Estimation Model (COCOMO II | Stage 1) Software Engineering," *GeeksforGeeks*, Mar. 14, 2019. https://www.geeksforgeeks.org/software-engineering/software-engineering-application-composition-estimation-model-cocomo-ii-stage-1/

   [2] Google LLC, "Google Calendar (Current version)," Mobile app, Google Play Store/Apple App Store, [Online]. Available: https://calendar.google.com/.

   [3]"Shareable Online Calendar App | Microsoft 365," *Microsoft.com*, 2024. https://www.microsoft.com/en-us/microsoft-365/outlook/calendar-app

   [4] "Calendar," *App Store*. https://apps.apple.com/us/app/calendar/id1108185179

---*Deliverable #1 Below*---

<u>Delegation of Tasks</u>:

- Aidan Casimir
  - Git Repository Creation and adding all group members and TAs
  - Architectural Design
  - Deliverable 1 report drafting
- Lance Miller
  - Sequence Diagram
- Josh Sebastian
  - Worked on Calendar Layout/UI Drafting
  - Use case diagram
  - README commit
- Kayla Burks
  - Requirements Lists
- Satish Parajuli
  - Backend Development, Database Research
  - Project Scope commit

- ❖ Venkatasai Gudisa
  - ➢ Spearheaded Sequence, Class, and Software Process Model Diagram Development
- ❖ Azlaan shafi
  - ➢ Relational Database Development

Assumptions: No assumptions required.

Addressing of Proposal Feedback:

The feedback is quoted as verbatim here:
“”“

Good start.

The proposed implementation could have been more precise:

"location data" - refers to location of events or location of the user for some functionality like weather?
"options to embedded hyperlinks" - functionality to embed hyperlinks in the event description? Or maybe as event names?

Deliverable #2 tasks for Venkatasai Gudisa missing.
The required change can be done in the team's Deliverable #1 report. If there are no such tasks, please indicate this.
”””

- ❖ Location Data Specification
  - ➢ The ‘location data’ referred to in the project proposal was referring to the feature of allowing the user to manually include location specific data such as an event address, hyperlink to map directional route, and priority time-based alerts relying on current location proximity to approaching events.
  - ➢ These are all quality of life (QoL) features that provide a more seamless and customizable experience for the user. The location data opt-in request would explicitly state that no data is to be sold or track, and only accessed for the purpose of proximity-based reminders
- ❖ “Options to embedded hyperlinks”
  - ➢ This is already covered in the above section as it relates to location data.
- ❖ “Deliverable #2 tasks for Venkatasai Gudisa missing”

- ➢ A few messages were sent about the missing task declaration, but no response was received before the proposal due date. The group did not want to assume their tasks and was therefore left blank.
- ➢ The tasks which would have been written will be stated here:
  Venkatasai Gudisa: Scheduling along with Aidan

Software Process Model:

For this project, we chose to use an Incremental Software Process Model as it fit our schedule of building key features in separate iterations. Since each team member worked on different modules, this model supported parallel development and avoided any blockages. It also provided flexibility to refine requirements based on feedback as this new information could be taken into account on the fly rather than restarting the whole process. By delivering working increments that were tested and improved over time, we believe that the Incremental model will increase progress visibility and reduce overall project risk, making it ideal for our use case.

Software Requirements:

- ❖ **Functional Requirements**:
  - ➢ The system must allow users to add an event.
  - ➢ The system must allow users to edit an existing event.
  - ➢ The system must allow users to delete an event.
  - ➢ The system must detect scheduling conflicts when a new event is added.
  - ➢ The system must display a daily calendar view.
  - ➢ The system must display a weekly and monthly calendar view.
  - ➢ The system must highlight holidays and weekends in distinct colors.

- ❖ **Non-Functional Requirements**:
  1. Usability Requirements:
     a. The system should be learnable by new users within 30 minutes.
     b. The system must support keyboard navigation.
     c. The system must support screen readers.
  2. Performance Requirements:
     a. Each calendar view must load within 3 seconds.
     b. A change to an event must appear within 1 second.
     c. The system should start up within 8 seconds.
  3. Space Requirements
     a. The system must need 100 MB or less of local storage.
     b. Each event must consume 1 MB for every 1000 events.

4. Dependability Requirements:
    a. All data must be saved automatically in the event that an unexpected event occurs.
    b. The system should fail no more than 1% of the operational time.
5. Security Requirements:
    a. The system must require a unique username for each user.
    b. The system must require a password for each user.
    c. The system must encrypt all stored user data.
    d. The system must lock a user account after five failed login attempts.
6. Environmental Requirements:
    a. The system must work on the major browsers including Google Chrome, Microsoft Edge, and Apple Safari.
    b. The system must need an internet connection for syncing.
    c. The system should have an offline mode with autosave capabilities.
7. Operational Requirements
    a. The system must save all data when being shut down.
    b. The system should be operational within 3 seconds of being opened.
8. Development Requirements:
    a. The system should be made using HTML for the frontend.
    b. The system should be made using Java or Python for the backend.
    c. The system database should be made using MySQL.
    d. The team should use tools such as GitHub and VS Code when developing the software.
9. Ethical Requirements:
    a. The system must not share any data without user consent.
    b. The system must allow users to delete all of their data at any time.
    c. The interface design must be accessible by supporting screen readers and keyboard
10. Accounting Requirements:
    a. The system must record any log-ins and actions made by users.
    b. The system must provide users an account of their own event edit history.
11. Safety/Security Requirements:
    a. The system must comply with standard security protocols.
    b. The system must lock out users after five failed log-in attempts.
    c. The system must comply with relevant laws and regulations concerning data and user protection.

<u>Use Case Diagram</u>:

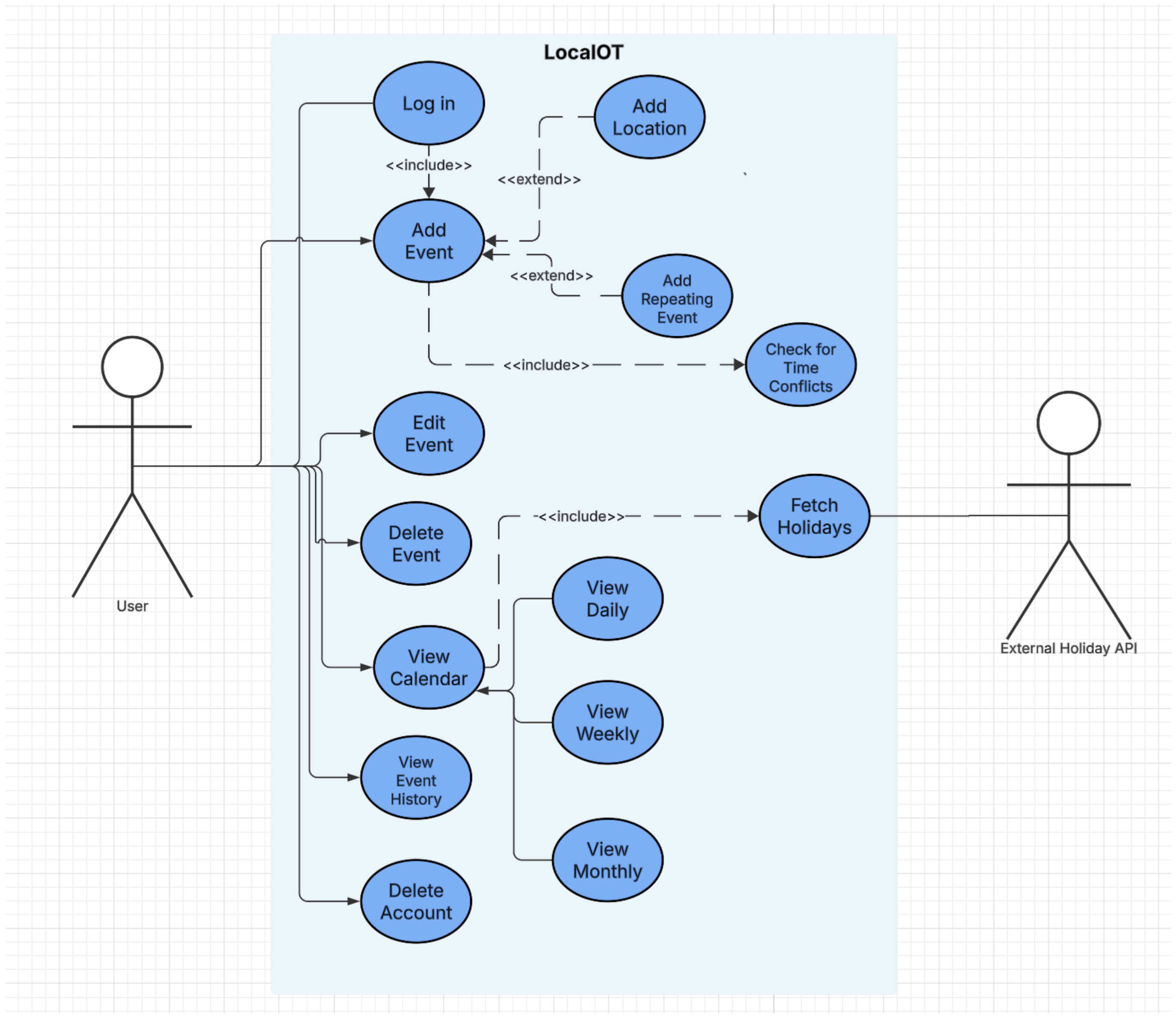


Fig. 1: Use Case Diagram

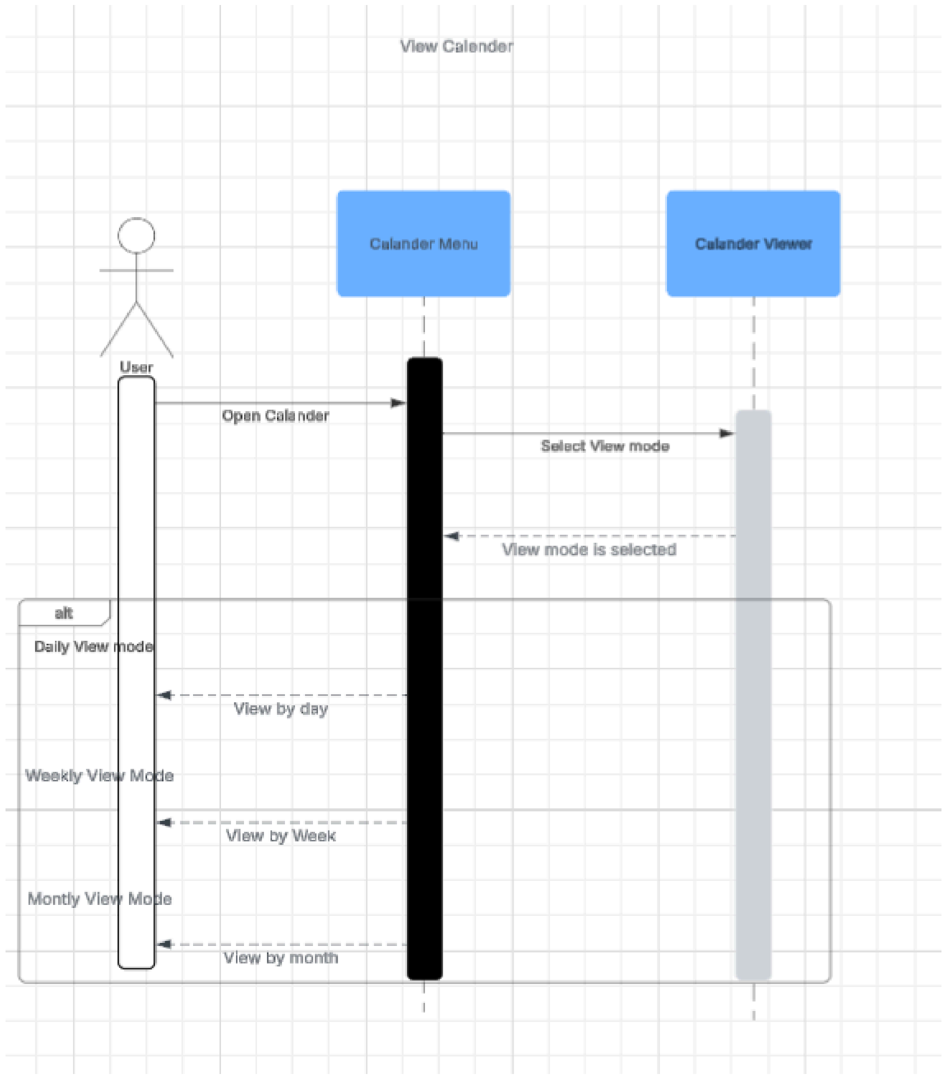
There is one primary human actor, the User, and one secondary system actor: an External Holiday API.

The User can perform all fundamental calendar operations, such as Log In, View Calendar, and manage events (which includes Add, Edit, and Delete Event). The diagram shows relationships to specify functionality:

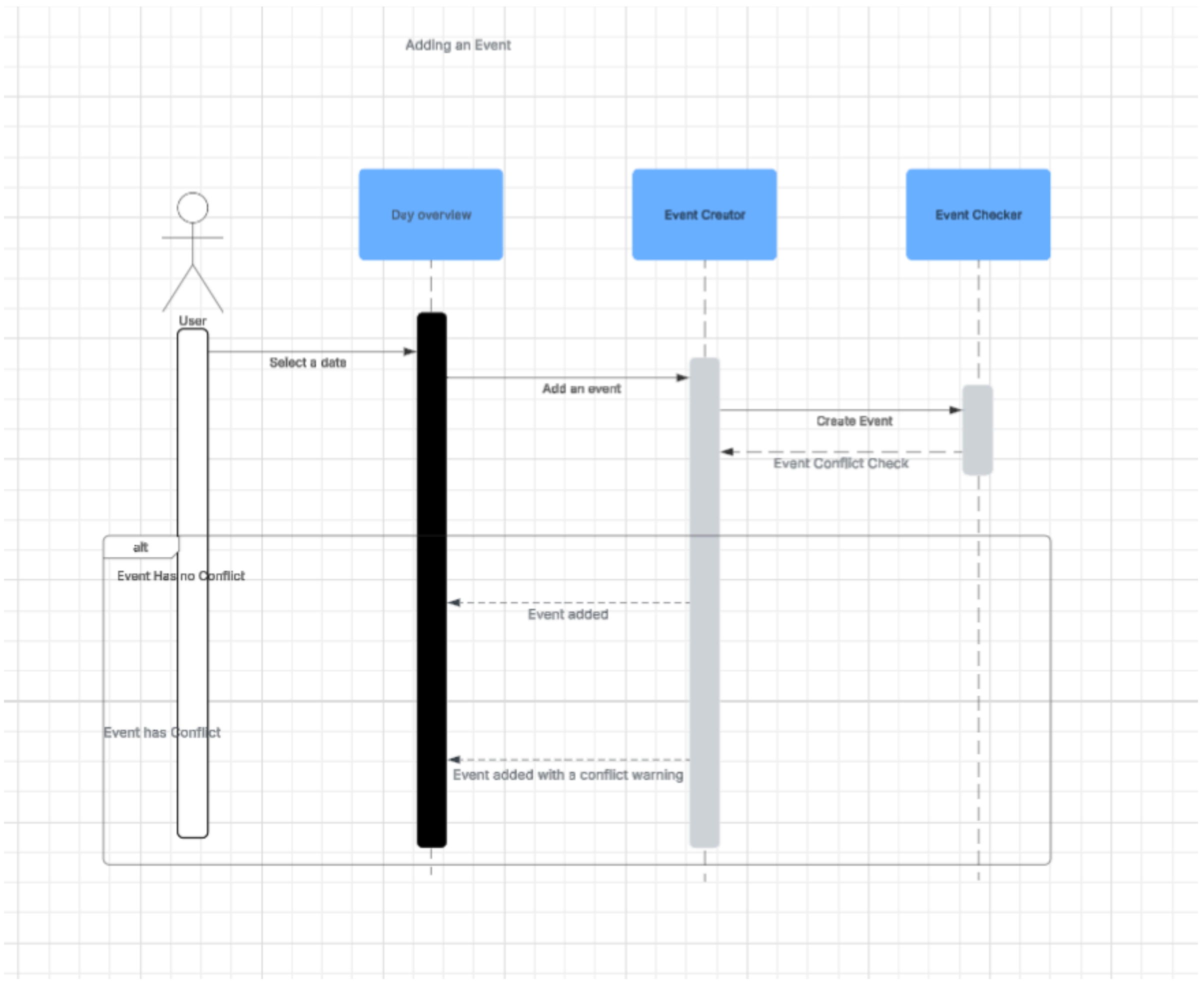
- Generalization is used to show that the View Daily, View Weekly, and View Monthly use cases are all specializations of the general View Calendar function.

- <<include>> relationships demonstrate required behaviors. For example, Add Event always includes Check for Time Conflicts, and View Calendar includes Fetch Holidays (which interacts with the External Holiday API).
- <<extend>> relationships show optional features. The Add Event use case can be extended by Add Location and Add Repeating Event, representing the optional features.
- External system interactions are shown by the Holiday API use case, which is triggered by the system, interacts with the Fetch Holidays service, and delivers information to the user.

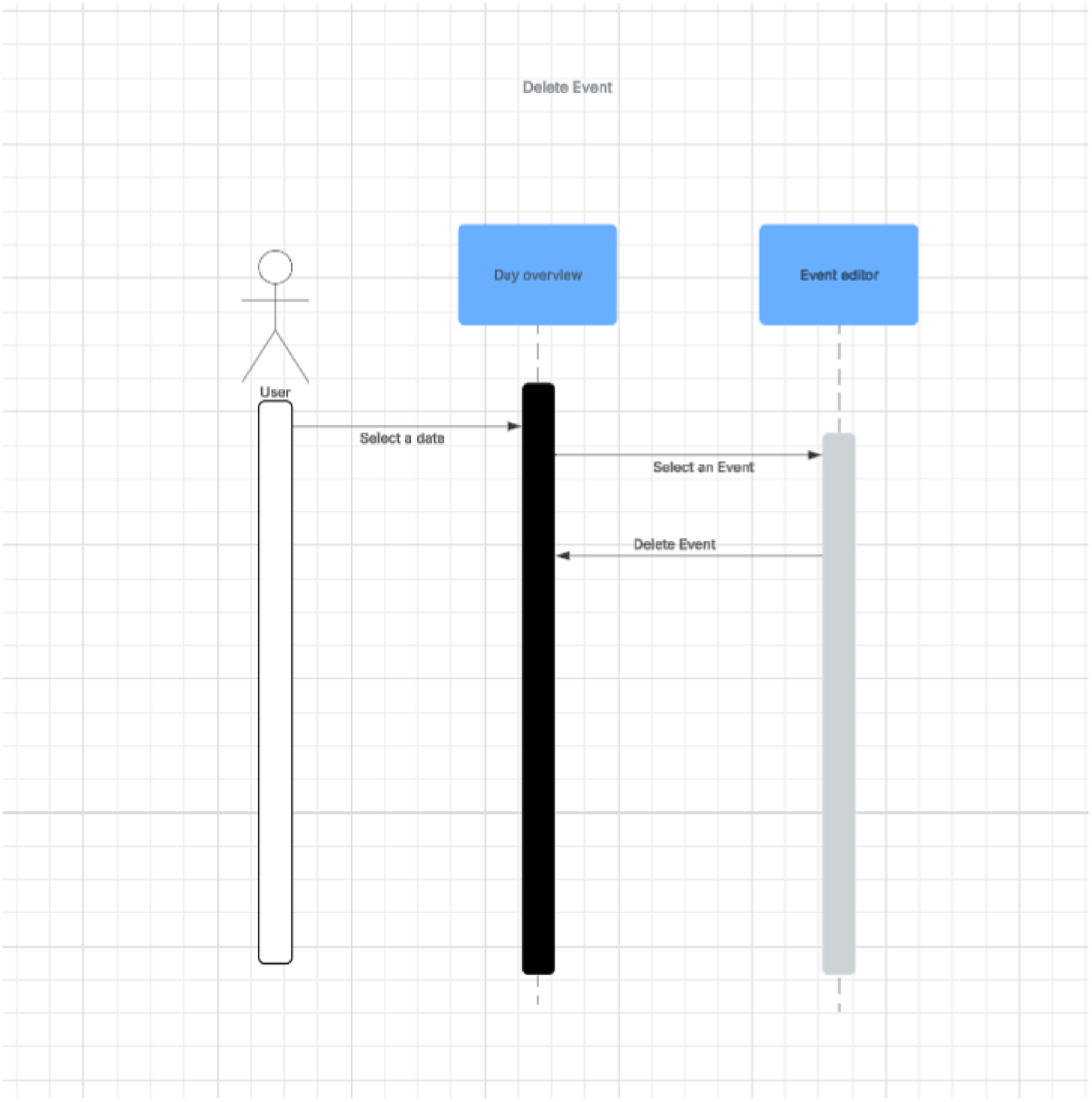
Sequence Diagram:



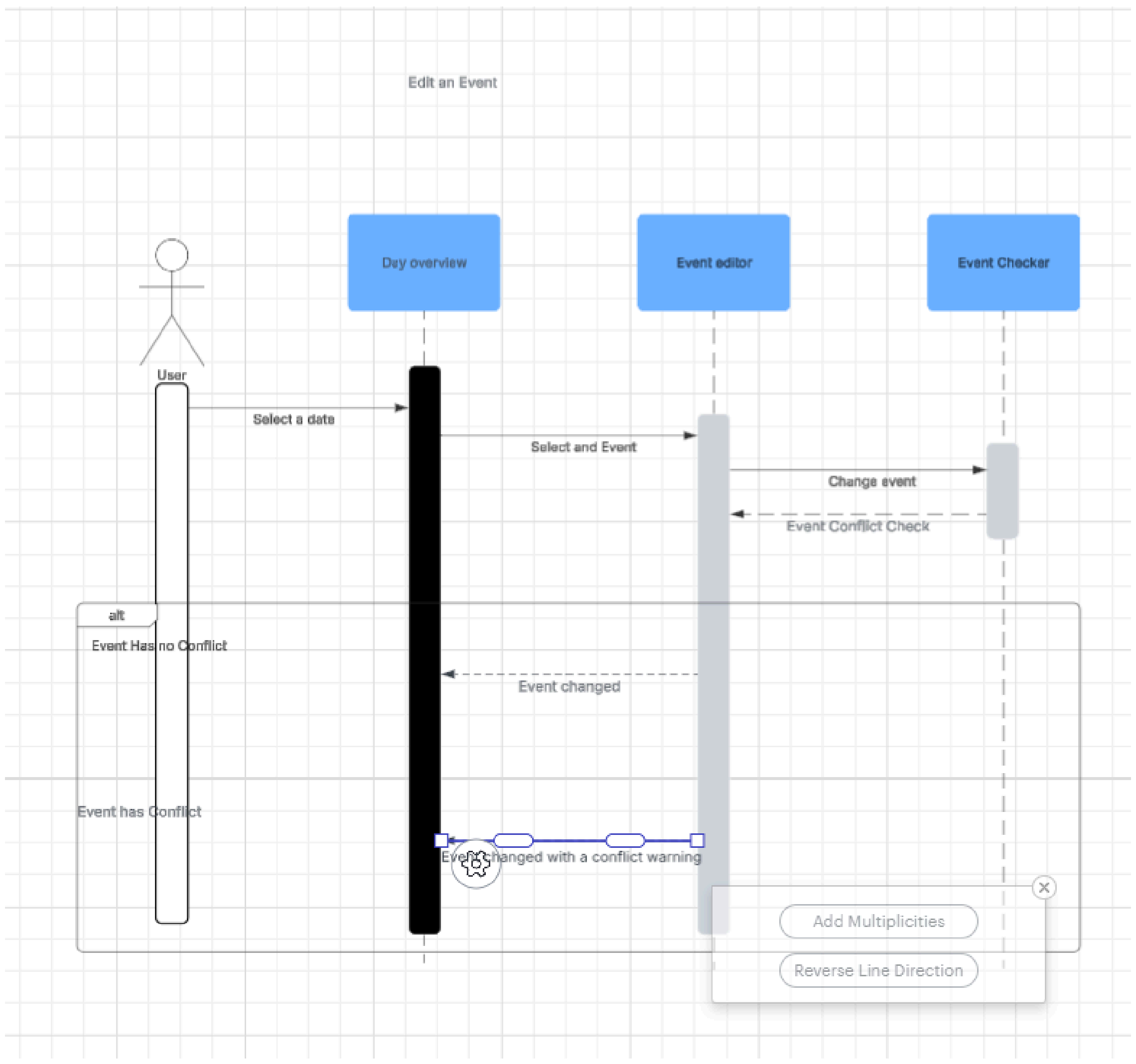
Sequence Diagram (View Calendar) (made using Lucidchart)



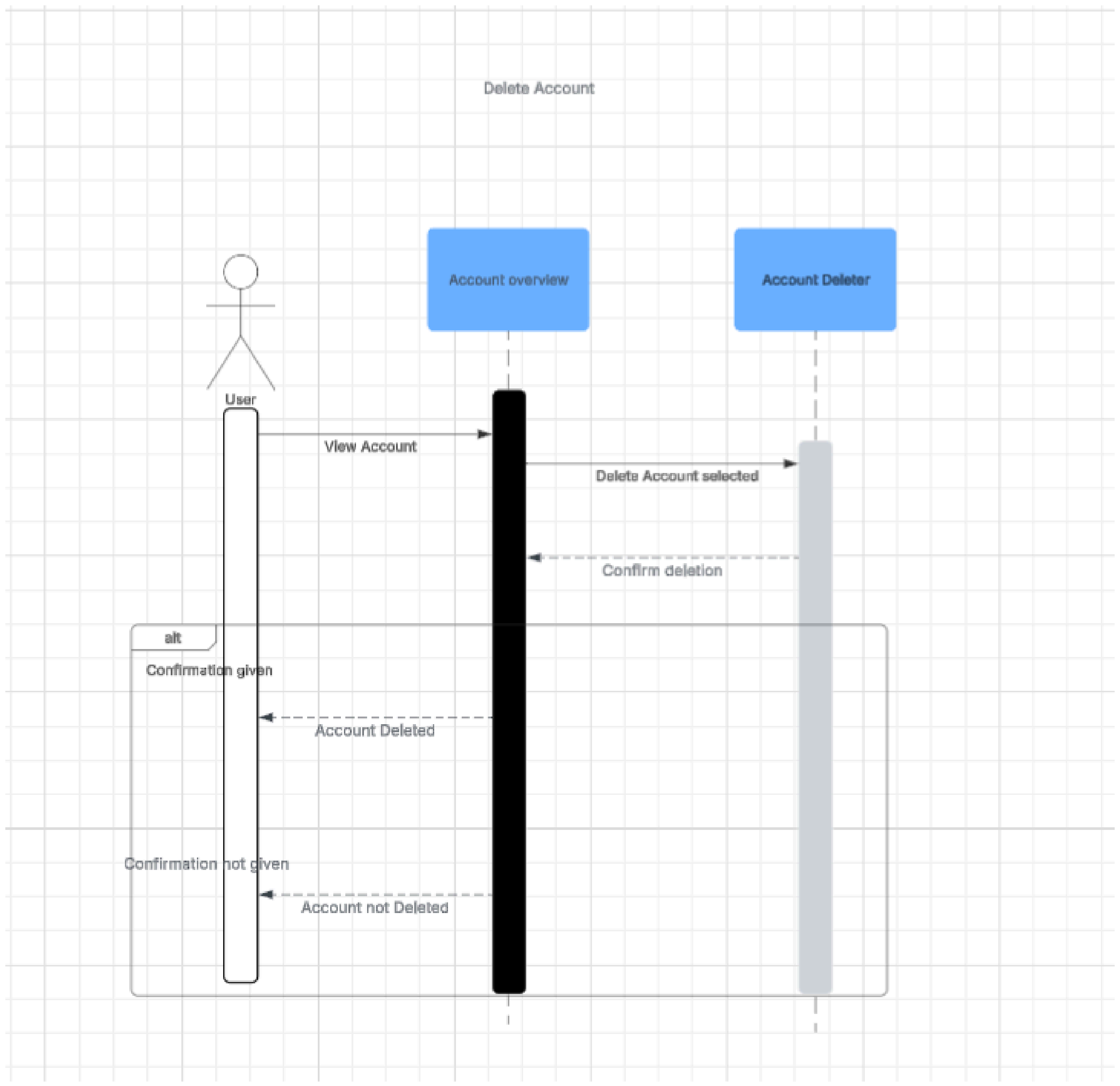
Sequence Diagram (Add Event) (made using Lucidchart)



Sequence Diagram (Delete Event) (made using Lucidchart)



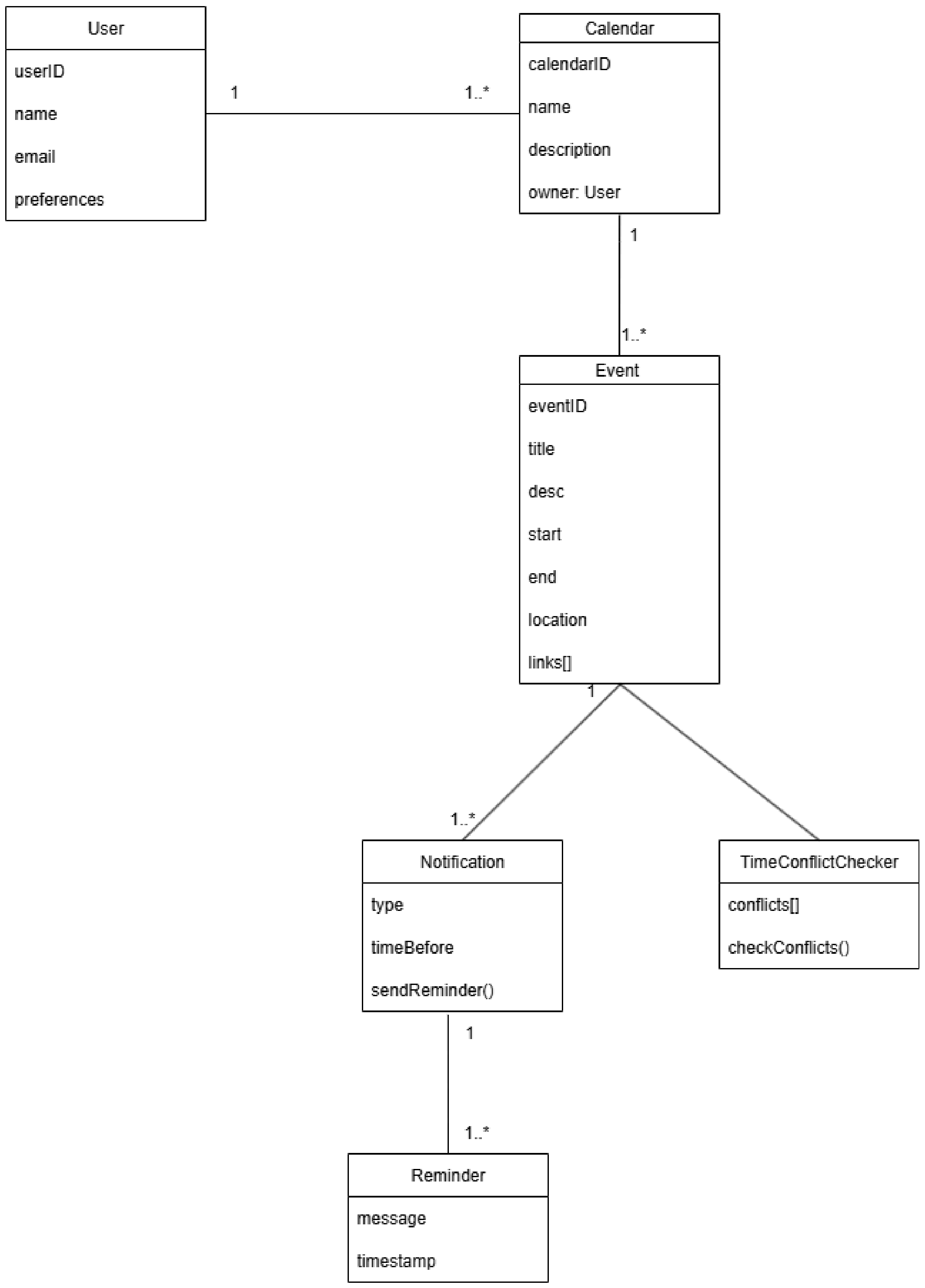
Sequence Diagram (Edit Event) (made using Lucidchart)



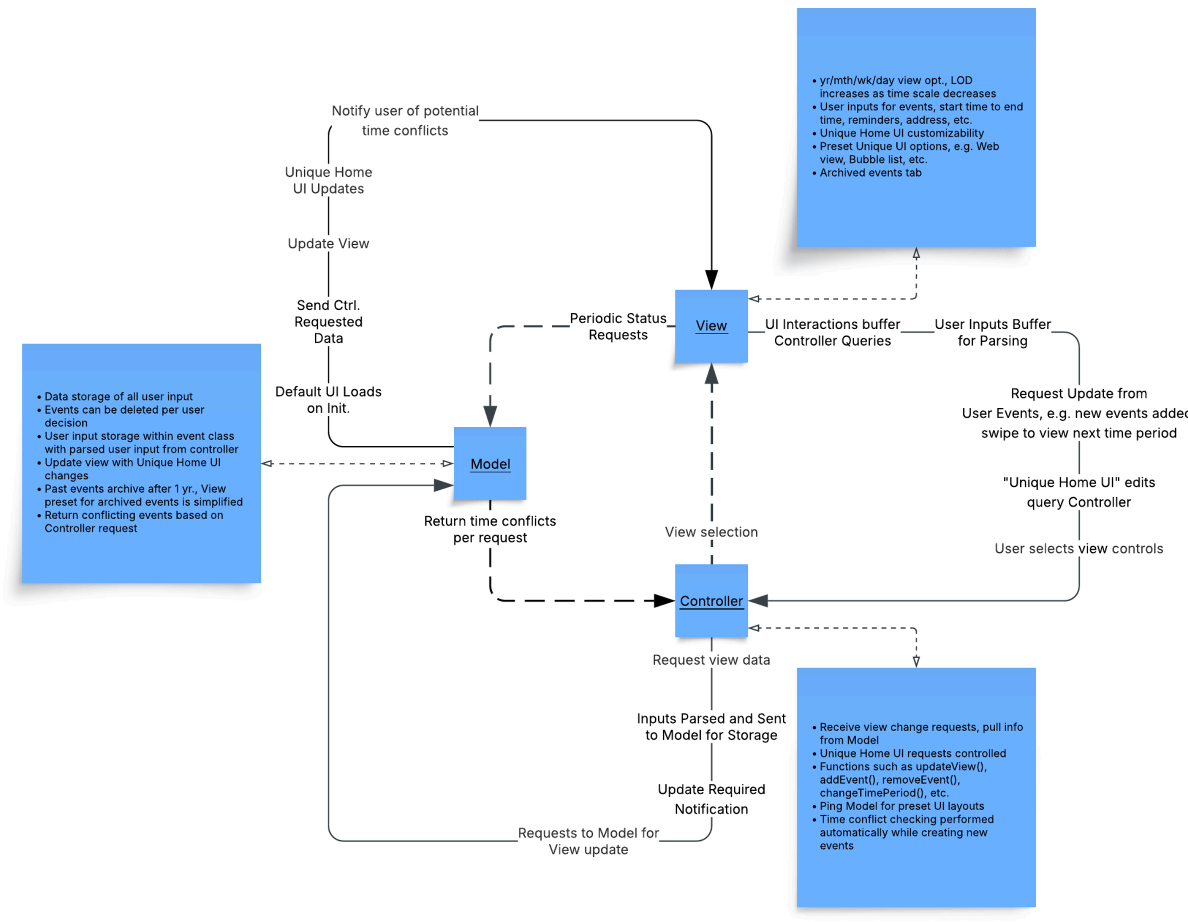
Sequence Diagram (Delete Account) (made using Lucidchart)

The above sequence diagrams show the way systems and menus will be used to allow the user to do things like login to an account where they can then view the calendar, make/delete/edit events, and delete their account.

Class Diagram:



<u>Architectural Design</u>:


• yr/mth/wk/day view opt., LOD increases as time scale decreases
• User inputs for events, start time to end time, reminders, address, etc.
• Unique Home UI customizability
• Preset Unique UI options, e.g. Web view, Bubble list, etc.
• Archived events tab
Notify user of potential time conflicts
Unique Home UI Updates
Update View
Send Ctrl. Requested Data
Default UI Loads on Init.
Periodic Status Requests
View
UI Interactions buffer Controller Queries
User Inputs Buffer for Parsing
Request Update from User Events, e.g. new events added, swipe to view next time period
"Unique Home UI" edits query Controller
User selects view controls
• Data storage of all user input
• Events can be deleted per user decision
• User input storage within event class with parsed user input from controller
• Update view with Unique Home UI changes
• Past events archive after 1 yr., View preset for archived events is simplified
• Return conflicting events based on Controller request
Model
Return time conflicts per request
View selection
Controller
Request view data
Inputs Parsed and Sent to Model for Storage
Update Required Notification
Requests to Model for View update
• Receive view change requests, pull info from Model
• Unique Home UI requests controlled
• Functions such as updateView(), addEvent(), removeEvent(), changeTimePeriod(), etc.
• Ping Model for preset UI layouts
• Time conflict checking performed automatically while creating new events

Architectural Pattern Diagram (made using Lucidchart)

The MVC pattern was chosen due to the nature of the requirement for distinct control over the front and back end of the calendar app. After initial research and review, the team deliberated over the decision between MVC and Layered, eventually reaching a conclusion that MVC would be the most fitting.

By separating the calendars functionality into 3 distinct components allowed me to construct an abstract view of the interdependencies between data storage, user input query control, and UI look and feel.

- The View component is solely responsible for displaying all relevant information to the user. The concepts of presentation, layout, design, etc. all focus on the front end look of the app.
- The Controller component deals with all interconnecting action based functionality between the View and Model components. These functions create the feeling of

interactivity with the app and allow the user to have a modular and seamless experience.
- ❖ The Model component is the data storage abstraction, dealing with database management, storing user input, providing updates to the View per request, etc.

All 3 components work together to provide their individual services in a cohesive and well structured manner:

- ❖ The View component, serving as the pure front end, loads the default UI from the Model at initialization. There are a multitude of buttons which provide the user with choices: time period selection, add an event, remove an event, swipe right or left (to view next or previous months/weeks/days), etc. Each of these interactions causes an event which is listened for by the controller.
- ❖ The controller receives these event queries and performs the corresponding function methods to begin processing the user inputted data. A multitude of different input types are taken by the controller, e.g. eventName, startTime, endTime, eventReminder, eventLocation, etc.
- ❖ When the user has completed entering their data, the model is then pinged to begin sorting and storing the new event data according to relational factors like time conflicts (checked during data parsing prior to controller processing), existing event time period relativity, etc.

The interactions between M, V, and C form a looping pattern in which Model loads init UI, View queries controller for user input, Controller requests Model to update View and so on. While most interactions follow in a single direction, some events result in “backwards” communication to another component. These events typically occur when a component has a deeper embedded feature that is automated to allow the app to function more efficiently. For example, the View will ping the Model for any status updates on a set period. In return, the Model will update the view for any background updates that require a subsequent UI update.