Project Milestone 3

Design

CS 3704
Spring 2024

Prepared by
TimeTeam: Nicholas Hess, Yasir Hassan, Thomas Tran, Tim Vadney, Gio Romero-Ruiz

4/8/2024

Table of Contents

High-level Design

Event-Driven Design

An event-driven architectural pattern would be the most suitable.

This pattern promotes the production, detection, consumption of, and reaction to events.

Pros:

- Anonymous handlers of events
- Support reuse and evolution, new features easy to add

Cons:

- Components have no control over the order of execution

Our project focuses on scheduling a meeting and deciding the best meeting time for a group of people. Various events occur at the same time, whether that be users indicating their availability, changes in availability, scheduling conflicts, and new additions or removals from the meeting team. An event-driven architecture will allow the system to react to these events in real time and update the list of best meeting times and the overall meeting schedule accordingly. Requirements for meeting scheduling may evolve and using this pattern allows the project to be scalable and flexible. It is the ideal choice for building a system that efficiently schedules meetings for large groups of people.

Low-level Design

Creation Patterns

This low-level design pattern aligns nicely with TimeScape as it offers ways flexibility and reusability in building out the web application. For instance, some of its offerings can be implemented in the following ways:

- Abstract Factory: since the web application is aimed at users being able to create meetings on an on-demand basis, being able to build out all of the necessary classes that's involved in such a task makes it reusable and convenient to do from a back-end perspective
- Object Pool: meetings that are archived or have passed could be viewed as no longer necessary from a maintenance perspective—keeping track of them in a pool when the meeting hasn't happened, and releasing it afterwards makes our application flexible and efficient
- Prototype: for users looking to quickly boot up meetings with/without configuration, having a prototype or template instance to build off of creates a nice abstraction to the application
- Singleton: since the web application aims to serve multiple users simultaneously, using a singleton for the server or manager class that communicates all the updates, data, and information to things like the database is necessary—having one instance prevents the need for solving concurrency-related issues

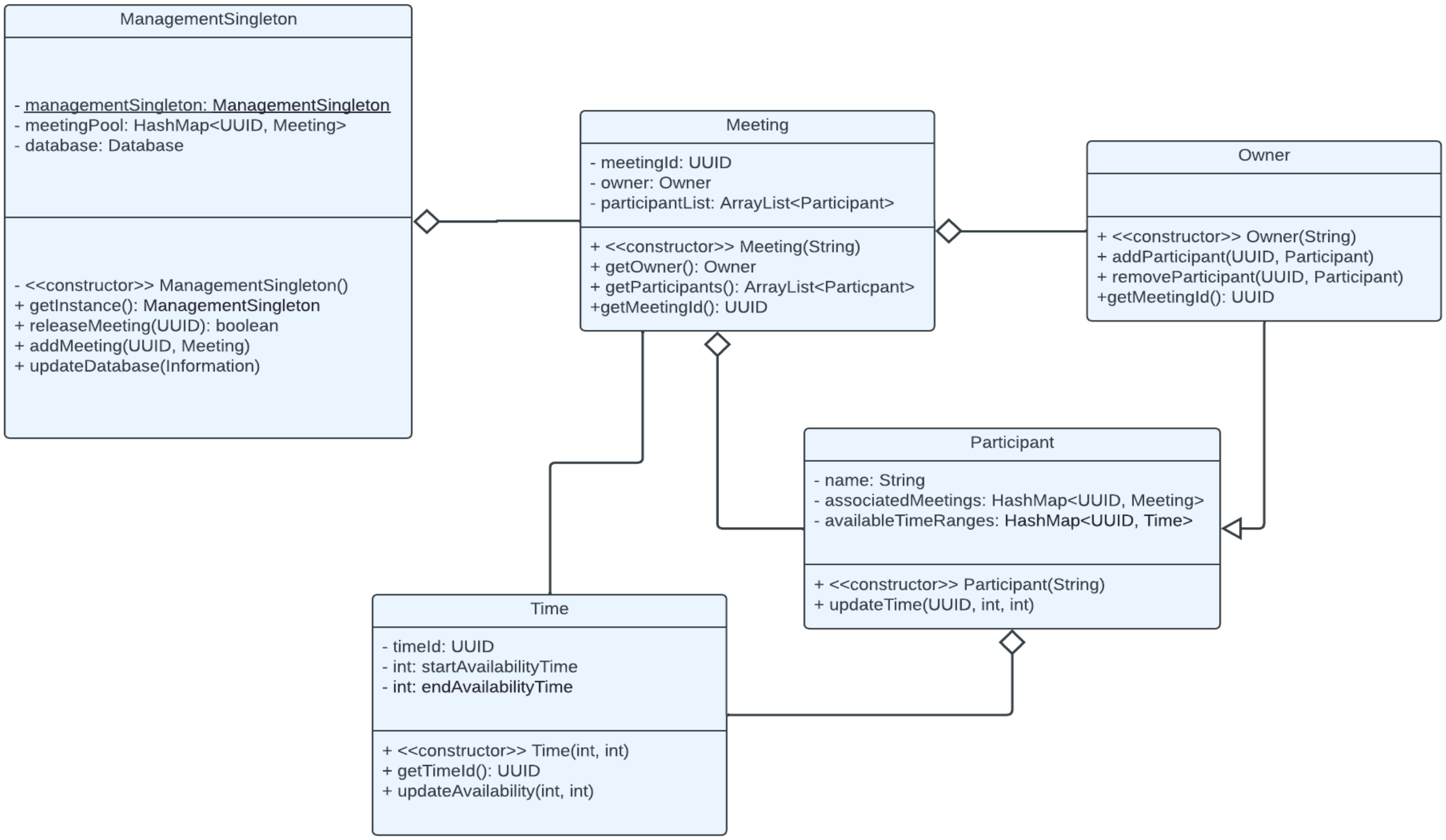
```
// Singleton class
public class ManagementSingleton
     private static ManagementSingleton managementSingleton;
     private HashMap<UUID, Meeting> meetingPool;
     private // Database instance/connection object

private ManagementSingleton()
{
     meetingPool = new HashMap<UUID, Meeting>();
     database = new Database();
}

public static ManagementSingleton getInstance()
{
     if(managementSingleton == null)
          managementSingleton = new ManagementSingleton();
     return managementSingleton;
}

public boolean releaseMeeting(UUID meetingId)
{
     return meetingPool.remove(meetingId);
}

public void addMeeting(UUID meetingId, Meeting meeting)
{
     meetingPool.put(meetingId, meeting);
}

public void updateDatabase(Information info) // abstraction for
information/database scheme
{
     database.update(info);
}
```

```
// can be viewed as the AbstractFactory—one stop shop for setup
public class Meeting
     private UUID meetingId;
     private Owner owner;
     private ArrayList<Participant> participantList;

public Meeting(String owner)
{
     meetingId = new UUID(); // added in the backend to the
singleton
     owner = new Owner(owner);
     participantList = new ArrayList<Participant>();
}

public Owner getOwner()
{
     return owner;
}

public ArrayList<Participant> getParticipants()
{
     return participantList;
}

public UUID getMeetingId()
{
     return meetingId;
}
```

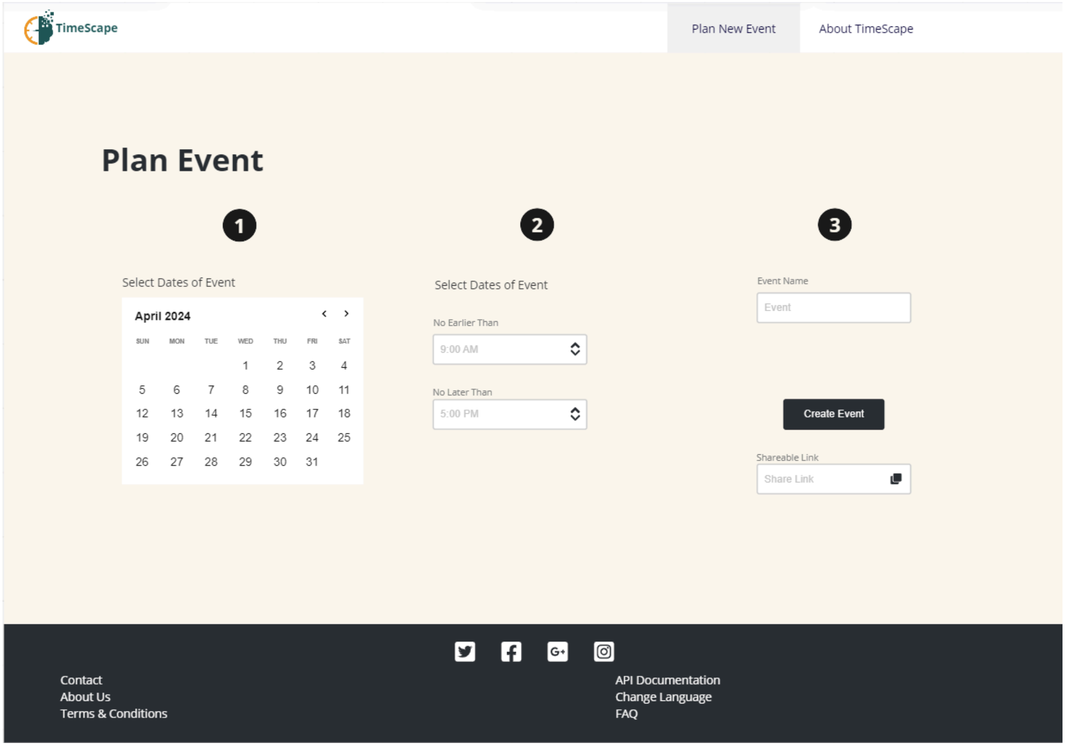
```
// Owner class—also a Participant, but has administrator access to
the Meeting (can add/remove people)
public class Owner extends Participant

    public Owner(String name)
    {
        super(name);
    }

    public void addParticipant(UUID meetingId, Participant
participant)
    {
        // assume that the meetingId is valid

associatedMeetings.get(meetingId).getParticipants().add(participant);
    }

    public void removeParticipant(UUID meetingId, Participant
participant)
    {
        // assume that the meetingId is valid
        return
associatedMeetings.get(meetingId).getParticipants().remove(participan
t);
    }
```

```
// Participant class
public class Participant
    private String name
    private HashMap<UUID, Meeting> associatedMeetings
    private HashMap<UUID, Time> availableTimeRanges

    public Participant(String name)
    {
        name = name;
        associatedMeetings = new HashMap<UUID, Meeting>();
        availableTimeRanges = new HashMap<UUID, Time>();
    }

    public void updateTime(UUID timeId, int start, int end)
    {
        // if timeId not inside of the availableTimeRanges
            Time time = new Time(start, end);
            availableTimeRanges.put(time.getTimeId(), time);
        // else if it has already been created
            Time time = availableTimeRanges.get(timeId);
            time.updateAvailability(start, end);
    }
```

```
// Time class
public class Time
     private UUID timeId;
     private int startAvailabilityTime;
     private int endAvailabilityTime;

     public Time(int start, int end)
     {
          timeId = new UUID();
          startAvailabilityTime = start;
          endAvailabilityTime = end;
     }

     public getTimeId()
     {
          return timeId;
     }

     public updateAvailability(int start, int end)
     {
          startAvailabilityTime = start;
          endAvailabilityTime = end;
     }
```


ManagementSingleton
- managementSingleton: ManagementSingleton
- meetingPool: HashMap<UUID, Meeting>
- database: Database
- <<constructor>> ManagementSingleton()
+ getInstance(): ManagementSingleton
+ releaseMeeting(UUID): boolean
+ addMeeting(UUID, Meeting)
+ updateDatabase(Information)
Meeting
- meetingId: UUID
- owner: Owner
- participantList: ArrayList<Participant>
+ <<constructor>> Meeting(String)
+ getOwner(): Owner
+ getParticipants(): ArrayList<Particpant>
+getMeetingId(): UUID
Owner
+ <<constructor>> Owner(String)
+ addParticipant(UUID, Participant)
+ removeParticipant(UUID, Participant)
+getMeetingId(): UUID
Participant
- name: String
- associatedMeetings: HashMap<UUID, Meeting>
- availableTimeRanges: HashMap<UUID, Time>
+ <<constructor>> Participant(String)
+ updateTime(UUID, int, int)
Time
- timeId: UUID
- int: startAvailabilityTime
- int: endAvailabilityTime
+ <<constructor>> Time(int, int)
+ getTimeId(): UUID
+ updateAvailability(int, int)

Mock User Interface - Wireframe



TimeScape

Plan New Event

About TimeScape

Plan Event

1

Select Dates of Event

April 2024 ‹ ›

SUN	MON	TUE	WED	THU	FRI	SAT
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

2

Select Dates of Event

No Earlier Than

9:00 AM

No Later Than

5:00 PM

3

Event Name

Event

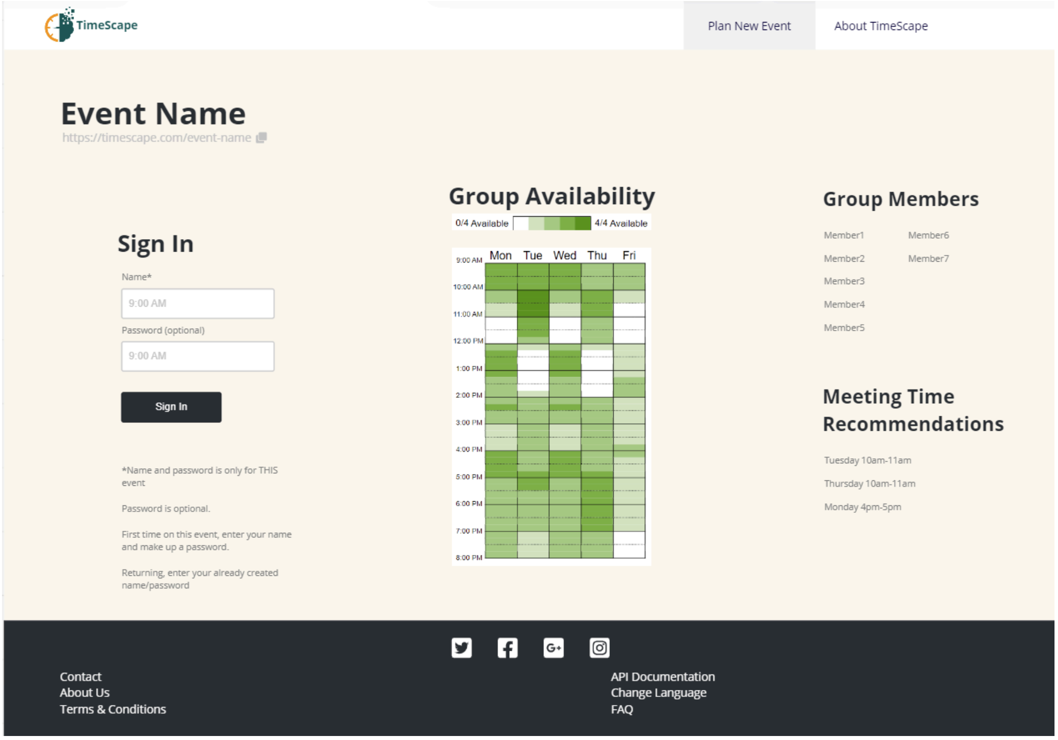
Create Event

Shareable Link

Share Link

Contact
About Us
Terms & Conditions

API Documentation
Change Language
FAQ



TimeScape

Plan New Event

About TimeScape

Event Name

https://timescape.com/event-name

Sign In

Name*

9:00 AM

Password (optional)

9:00 AM

Sign In

*Name and password is only for THIS event

Password is optional.

First time on this event, enter your name and make up a password.

Returning, enter your already created name/password

Group Availability

Group Members

Member1
Member2
Member3
Member4
Member5
Member6
Member7

Meeting Time Recommendations

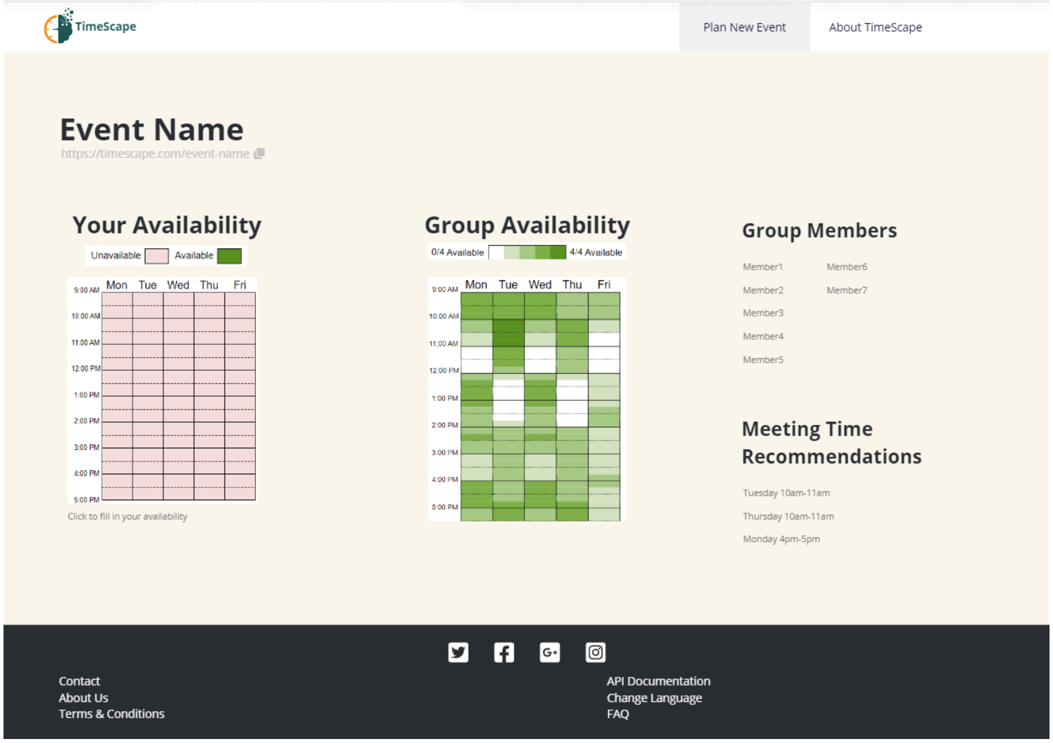
Tuesday 10am-11am

Thursday 10am-11am

Monday 4pm-5pm

Contact
About Us
Terms & Conditions

API Documentation
Change Language
FAQ



Event Name

https://timescape.com/event-name

Your Availability

Click to fill in your availability

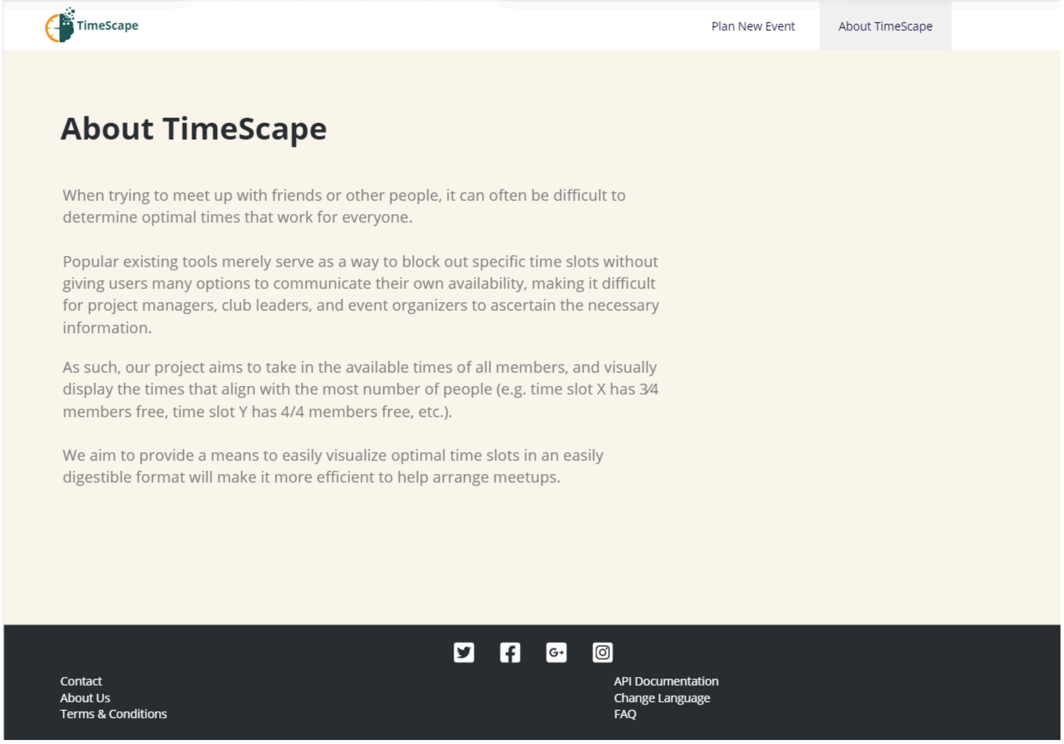
Group Availability

Group Members

Member1
Member2
Member3
Member4
Member5
Member6
Member7

Meeting Time Recommendations

Tuesday 10am-11am
Thursday 10am-11am
Monday 4pm-5pm



About TimeScape

When trying to meet up with friends or other people, it can often be difficult to determine optimal times that work for everyone.

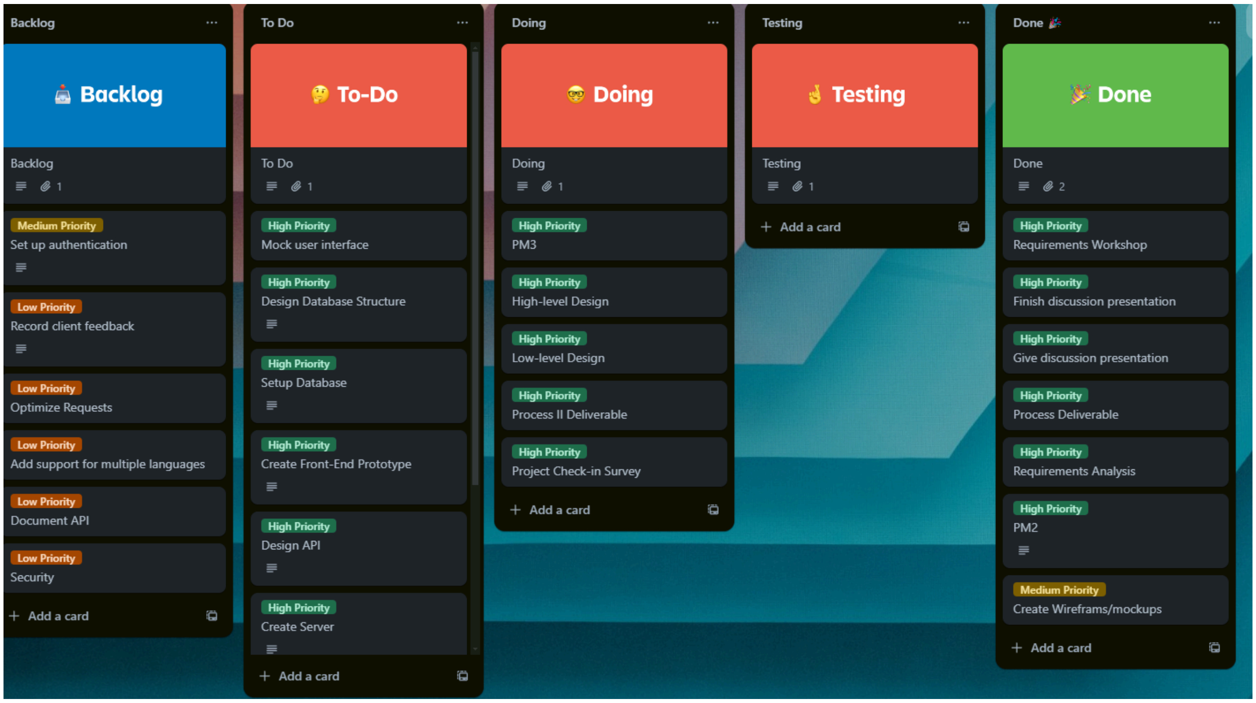
Popular existing tools merely serve as a way to block out specific time slots without giving users many options to communicate their own availability, making it difficult for project managers, club leaders, and event organizers to ascertain the necessary information.

As such, our project aims to take in the available times of all members, and visually display the times that align with the most number of people (e.g. time slot X has 3⁄4 members free, time slot Y has 4/4 members free, etc.).

We aim to provide a means to easily visualize optimal time slots in an easily digestible format will make it more efficient to help arrange meetups.

The first frame of our wireframe is the landing page where users can create a TimeScape meeting event table. They choose the times and dates of the availability they wish to plan the event in, give the event a name, then create the event. Then they are given the option to copy a shareable link to the event for meeting members to join and add their availability. When a member joins the event, they will be prompted to sign in to this event. Upon signing in they input their availability and that will update the Group Availability table. The section titled “Meeting Time Recommendations” is a section provided by us which will provide the event with times and dates of most availability. We also have an About TimeScape tab that allows users who are curious to understand more about TimeScape.

Process II Deliverable



Backlog ···

Backlog

Backlog

1

- Medium Priority
  Set up authentication
- Low Priority
  Record client feedback
- Low Priority
  Optimize Requests
- Low Priority
  Add support for multiple languages
- Low Priority
  Document API
- Low Priority
  Security

+ Add a card

To Do ···

To-Do

To Do

1

- High Priority
  Mock user interface
- High Priority
  Design Database Structure
- High Priority
  Setup Database
- High Priority
  Create Front-End Prototype
- High Priority
  Design API
- High Priority
  Create Server

+ Add a card

Doing ···

Doing

Doing

1

- High Priority
  PM3
- High Priority
  High-level Design
- High Priority
  Low-level Design
- High Priority
  Process II Deliverable
- High Priority
  Project Check-in Survey

+ Add a card

Testing ···

Testing

Testing

1

+ Add a card

Done ···

Done

Done

2

- High Priority
  Requirements Workshop
- High Priority
  Finish discussion presentation
- High Priority
  Give discussion presentation
- High Priority
  Process Deliverable
- High Priority
  Requirements Analysis
- High Priority
  PM2
- Medium Priority
  Create Wireframs/mockups

+ Add a card

Here is the Trello board that we have been using for the past few weeks to efficiently manage our tasks. As mentioned previously we split up our tasks into three different kinds of priorities. We have high priority tasks which in this case are the completion of PM3 including the deliverables for PM3. This is of high importance because we are prioritizing upcoming deadlines to satisfy our stakeholders. The medium priority tasks are the ones we know that come after the high priority. We need Login/Authentication, and a good design once we get our backend up and running, so it is important for us to think about those tasks while we are working on the ones before. Low priority tasks are more so tasks that our application could live without. Ideally we would have those features, but they are not of utmost importance. These features would likely be added later in the project's lifetime. By breaking up our tasks into these priority levels, we are ensuring that our efforts are focused on meeting immediate requirements while also keeping an eye on the roadmap for future improvements. This approach enables us to maintain a structured and efficient development process.