CS673 Software Engineering
Team 1 - UNO
Plan.ly
Software Design Document

Team Member	Role(s)	Signature	Date
George Wright	Team Leader	*George Wright*	3/9/2021
Aysha Zenab Kenza	Security Leader	*Aysha Zenab Kenza*	03/9/2021
Vibhu Bhatia	Backup Team Leader	*Vibhu Bhatia*	03/09/2021
Karen Sommer	Design and Implementation Leader	*Karen Sommer H.*	03/09/2021
Zach Schandorf-Lartey	Configuration Leader	*Zach Schandorf-Lartey*	3/9/2021
Chris Kulig	Requirements Leader	*Christopher Kulig*	3/9/21
Matt Dowding	Quality Assurance Leader	*Matthew Dowding*	3/9/2021

Revision history

Version	**Author**	**Date**	**Change**
0	Group	3/10/21	Initial release
1	Group	4/7/21	Updated component and class diagram per feedback comments
2	Karen Sommer	4/26/202	Add use cases and sequence

		1	diagrams

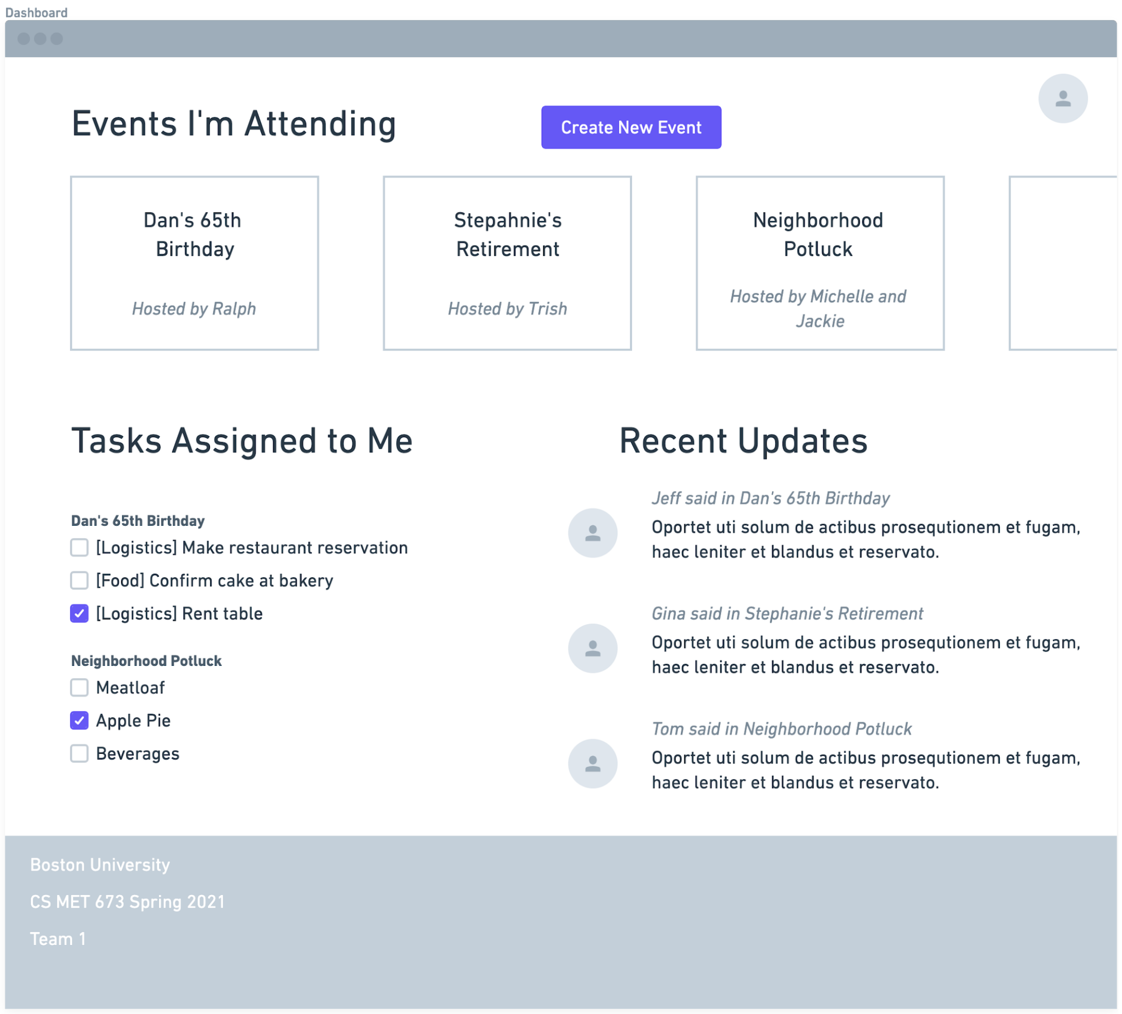
1. Introduction

This document provides the envisioned technical design for the Plan.ly application. At a high level, Plan.ly will use a segregated front- and back-end environment, with data persisted in a NoSQL environment. While the application's data is structured in nature, we have opted to use NoSQL rather than SQL given the agile nature of our development approach and the relative ease with which schema changes can be carried out in NoSQL in comparison to SQL.

The look and feel of our essential features (registering, logging in, creating events, managing to-do lists, and managing event invitees and ownership) is captured in the below wireframes.

Dashboard Wireframe

After signing in, users see the dashboard. The dashboard shows: all events the user is attending, all tasks the user has been assigned (across all events they are attending), and (in the future as part of a desirable story) recent updates from event feeds.



Dan's 65th Birthday
Hosted by Ralph

Stepahnie's Retirement
Hosted by Trish

Neighborhood Potluck
Hosted by Michelle and Jackie

Tasks Assigned to Me

Dan's 65th Birthday
- [ ] [Logistics] Make restaurant reservation
- [ ] [Food] Confirm cake at bakery
- [x] [Logistics] Rent table

Neighborhood Potluck
- [ ] Meatloaf
- [x] Apple Pie
- [ ] Beverages

Recent Updates

Jeff said in Dan's 65th Birthday
Oportet uti solum de actibus prosequtionem et fugam, haec leniter et blandus et reservato.

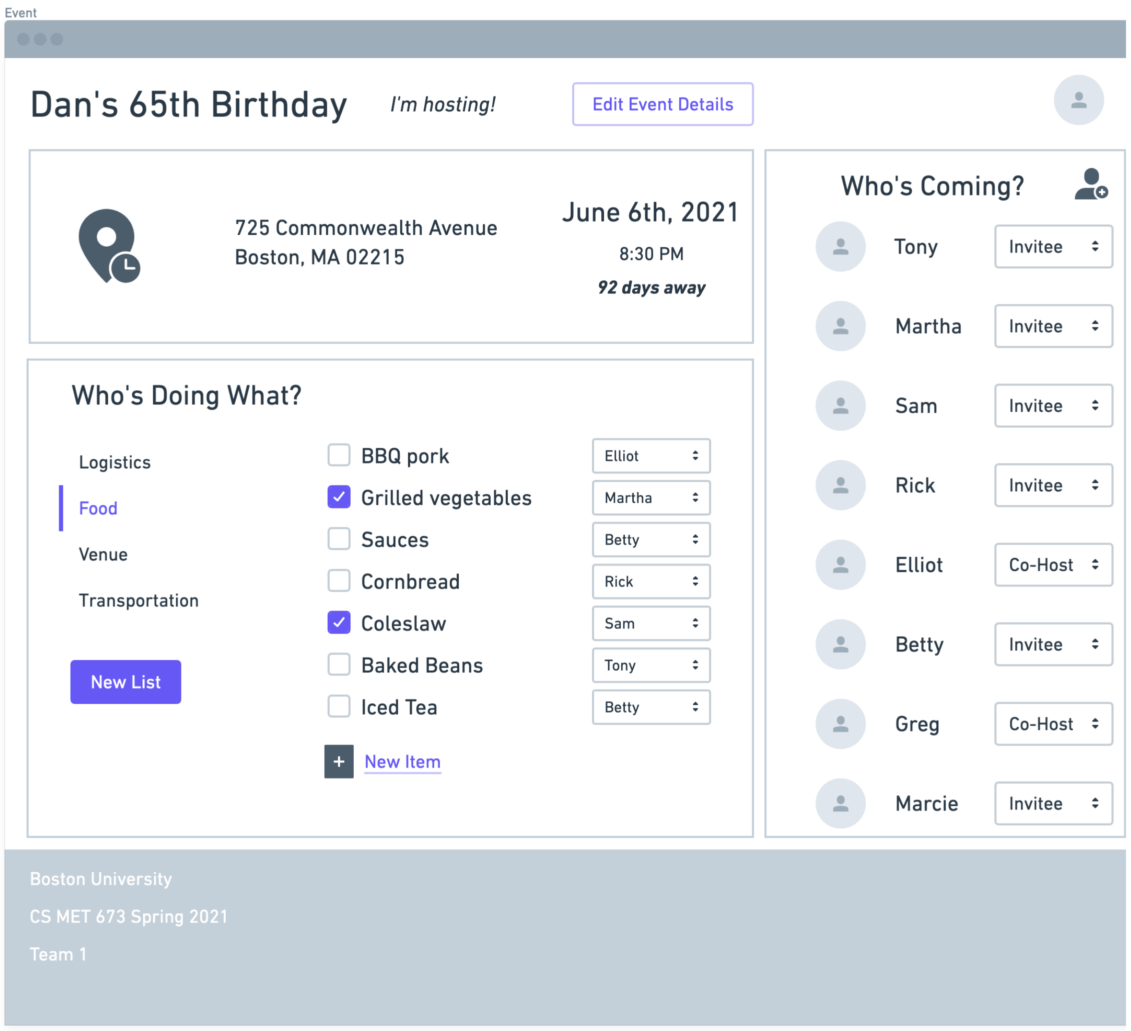
Gina said in Stephanie's Retirement
Oportet uti solum de actibus prosequtionem et fugam, haec leniter et blandus et reservato.

Tom said in Neighborhood Potluck
Oportet uti solum de actibus prosequtionem et fugam, haec leniter et blandus et reservato.

Boston University
CS MET 673 Spring 2021
Team 1

Event Wireframe

The event screen shows all details about an event. This includes the start date, address, all to-do lists (and contents), and all event invitees with their assigned role. Event owners will have an “Edit Event Details” button that will turn certain components of this page into form field elements that can be updated and saved back to the database.


Event

Dan's 65th Birthday *I'm hosting!* Edit Event Details

725 Commonwealth Avenue
Boston, MA 02215

June 6th, 2021
8:30 PM
92 days away

Who's Doing What?

Logistics
Food
Venue
Transportation

New List

☐ BBQ pork	Elliot
✓ Grilled vegetables	Martha
☐ Sauces	Betty
☐ Cornbread	Rick
✓ Coleslaw	Sam
☐ Baked Beans	Tony
☐ Iced Tea	Betty

+ New Item

Who's Coming?

Tony	Invitee
Martha	Invitee
Sam	Invitee
Rick	Invitee
Elliot	Co-Host
Betty	Invitee
Greg	Co-Host
Marcie	Invitee

Boston University
CS MET 673 Spring 2021
Team 1

2. Software Architecture

In this section, we have the software architecture and the class diagram of Plan.ly.
In image 1, we can see the different components of the system and how they interact.

Inside the client, the node is the **web browser component**, which gathers its data using an *HTTP request*. The view component responds to the request with the *HTTP response*. The Plan.ly node is the **view component,** which is written in React. The view component also initiates requests to send. Automated emails through the Django **email server component** using *SMTP*.
Inside the Plan.ly node, there are also two more components: the **template**

component, written in React and responsible for communicating with the view component, and the **model component**, which communicates both with the view and with the Mongo database using *djongo*.

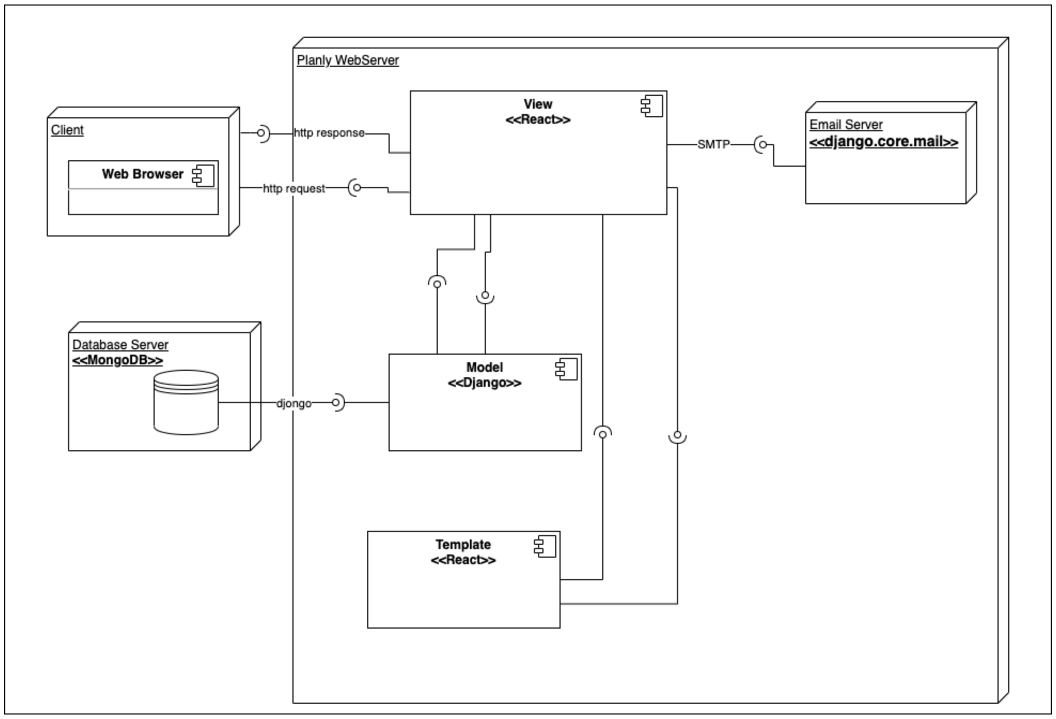


Image 1. Component diagram

Image 2 is the class diagram of our system.
Based on the user stories, a class diagram was constructed consisting of components for Plan.ly, User, EventInvitee, Event, Task, and ToDoList, the relationships between them. Inside of every class are the methods and attributes. Note that relationships are annotated with their cardinality.

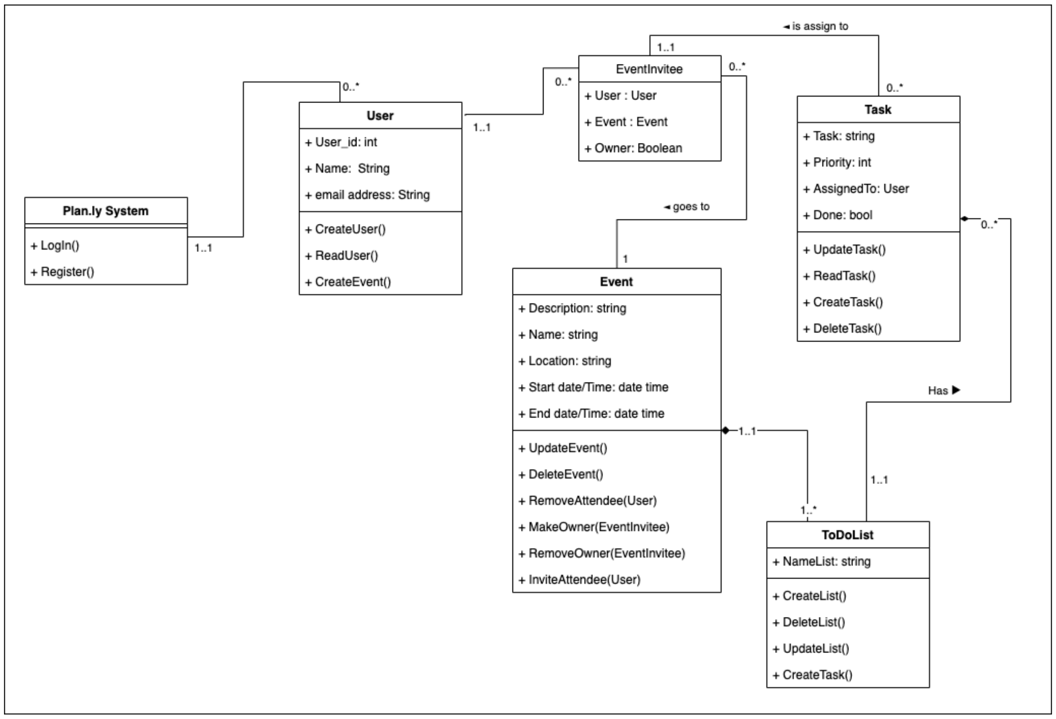


Image 2. Class diagram

3. Database Design

In this section, we describe the database we used in our system, the alternatives we considered, the advantages of our approach, and finally, the database structure of our app.

We are using MongoDB for our application. Our application backend is constructed with Django, which typically works best with SQL databases. However, after some discussions amongst our group, we decided to go with a NoSQL approach due to its simplicity and avoidance of expensive joins and grouping operations that may be needed when our application scales up. NoSQL also helps in reducing the redundancy in storing data. If one schema is supposed to contain instances of another schema (i.e., object nesting), we can just store them with the id, and while returning the objects, those ids are populated with the entire record. The data is also passed around in JSON format, making it universal for transfer, in our case, from python backend to javascript frontend. The connection between Django and MongoDB is optimized using the Djongo Python package. Some of the classes look.

4. Security

Cross-Site Request Forgery (CSRF) tokens are unique, secret, and randomly generated values that are generated by a server-side application and sent to the client. When a request is made from the client, the server checks to see if the request includes the expected CSRF token and if it does validates it. If the token is missing or invalid, the request is rejected. Django implements it in all form submissions to prevent CSRF attacks, and we implemented it on our front end so that the data sent in the post requests from our React frontend to our Django backend are secure.

Django comes with a user authentication system. It handles user accounts, groups, permissions, and cookie-based user sessions.

The auth system consists of:

- Users
- Permissions: Binary (yes/no) flags designating whether a user may perform a certain task.
- Groups: A generic way of applying labels and permissions to more than one user.
- A configurable password hashing system
- Forms and view tools for logging in users or restricting content
- A pluggable backend system

JSON Web Token (JWT) is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA** or **ECDSA**.

Authorization: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign-On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

JSON Web Tokens consist of three parts separated by dots (.), which are:

- Header: The header *typically* consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.
- Payload: The payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: *registered*, *public*, and *private* claims.
- Signature: The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

Therefore, a JWT typically looks like the following: xxxxx.yyyyy.zzzzz

5. Patterns

The design patterns are a general repeatable solution to a commonly occurring problem in a software design.

a. Architectural pattern

We selected Django as our web framework. Django architecture is based on MVC (Model View Controller) pattern [1]. The MVC pattern is oriented to Web applications. We decided to keep this architectural pattern because:

i. Django provides us all the tools to follow this pattern.
ii. It is easy to manage change because components are independent of one another. As such, modifications to one component do not typically impact others.
iii. We can track and see the three components in our code, making it easy for multiple developers to work on different aspects of the same application simultaneously.

It is important to note that in Django, this pattern is instead called the MTV pattern (Model-View-Template). Here is a quick explanation of each component.

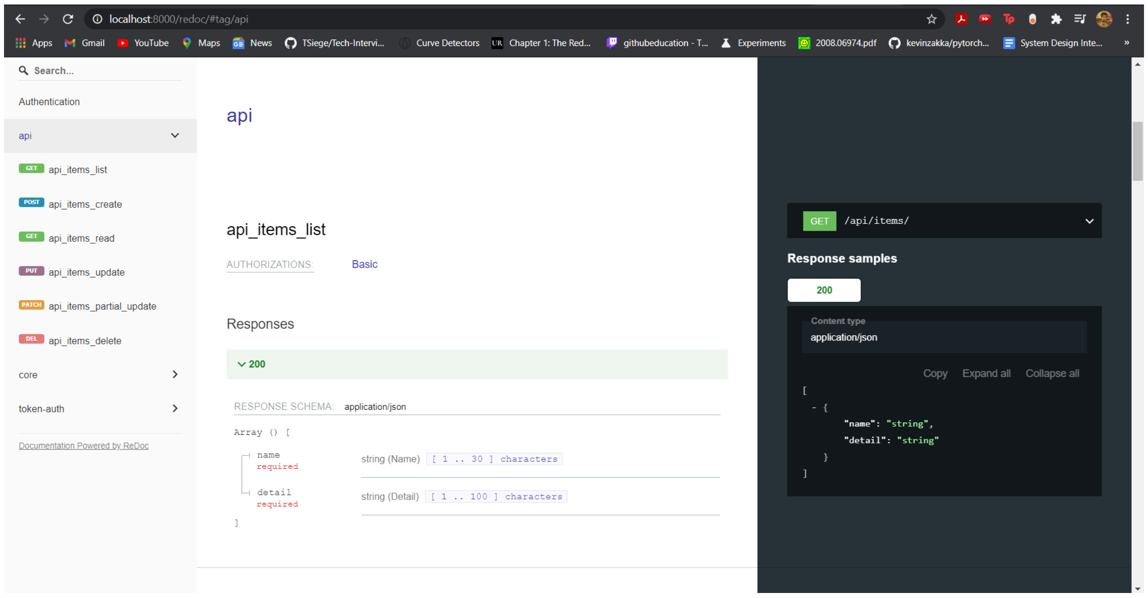
i. Model: it is the component that implements the logic for the application's data domain
ii. Template: this component contains the UI logic
iii. View: this component handles the user interaction and selects a view according to the model.

b. **Design Patterns**

Design patterns represent some of the best practices adopted by experienced object-oriented software developers. Our class diagram shows multiple uses of aggregation-based relationships. We needed to explicitly show the strong association between Event and toDoList and between ToDoList and Tasks. However, given the smaller size of our project, there isn't any design pattern (Creational, Structural or Behavioral patterns) in use.

6. Classes and Methods

For the backend, we are using a framework called drf-yasg, which generates real OpenAPI 2.0 specifications from a Django Rest Framework API. We are using Redoc to generate our API documentation which can be looked at by accessing http://localhost:8000/docs in a local build. Once generated, the API looks something like this:



api

api_items_list

AUTHORIZATIONS: Basic

Responses

200

RESPONSE SCHEMA: application/json

```
Array () [
    name      string (Name)    [ 1 .. 30 ] characters
    required
    detail    string (Detail)  [ 1 .. 100 ] characters
    required
]
```

GET /api/items/

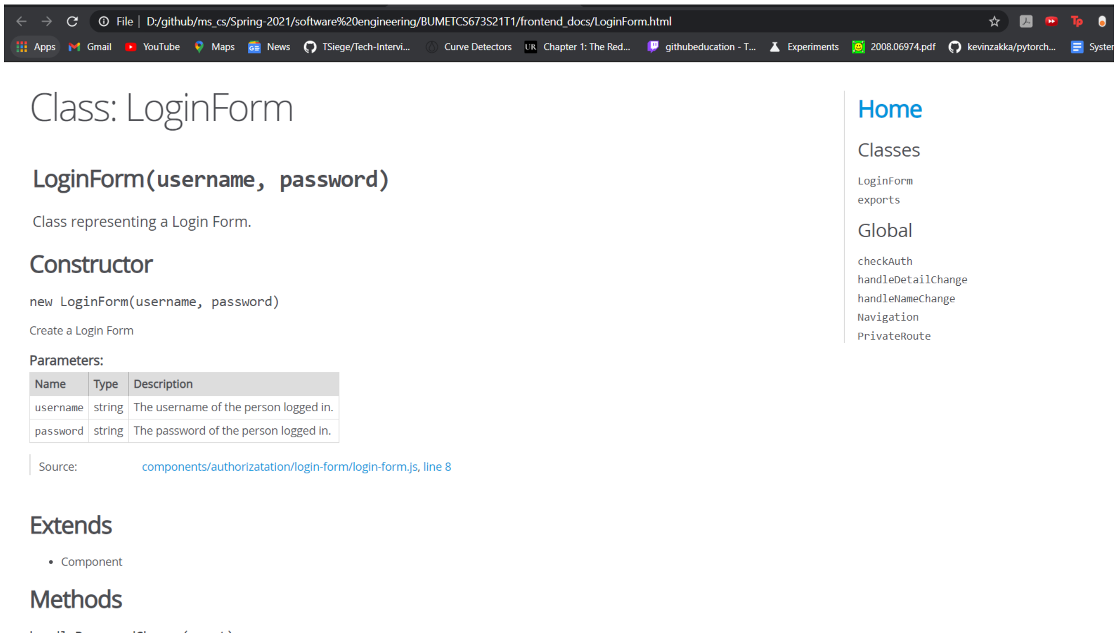
Response samples

200

Content type
application/json

```
[
  - {
        "name": "string",
        "detail": "string"
    }
]
```

For the front-end, we are using jsdoc for generating our documentation. We decided to use this resource because it sets a standard for annotating our code, making it easy to understand, in addition to automatically generating our documentation. A sample output looks something like this:



Class: LoginForm

LoginForm(username, password)

Class representing a Login Form.

Constructor

```
new LoginForm(username, password)
```

Create a Login Form

Parameters:

Name	Type	Description
`username`	string	The username of the person logged in.
`password`	string	The password of the person logged in.

Source: components/authorizatation/login-form/login-form.js, line 8

Extends

- Component

Methods

Home

Classes

LoginForm
exports

Global

checkAuth
handleDetailChange
handleNameChange
Navigation
PrivateRoute

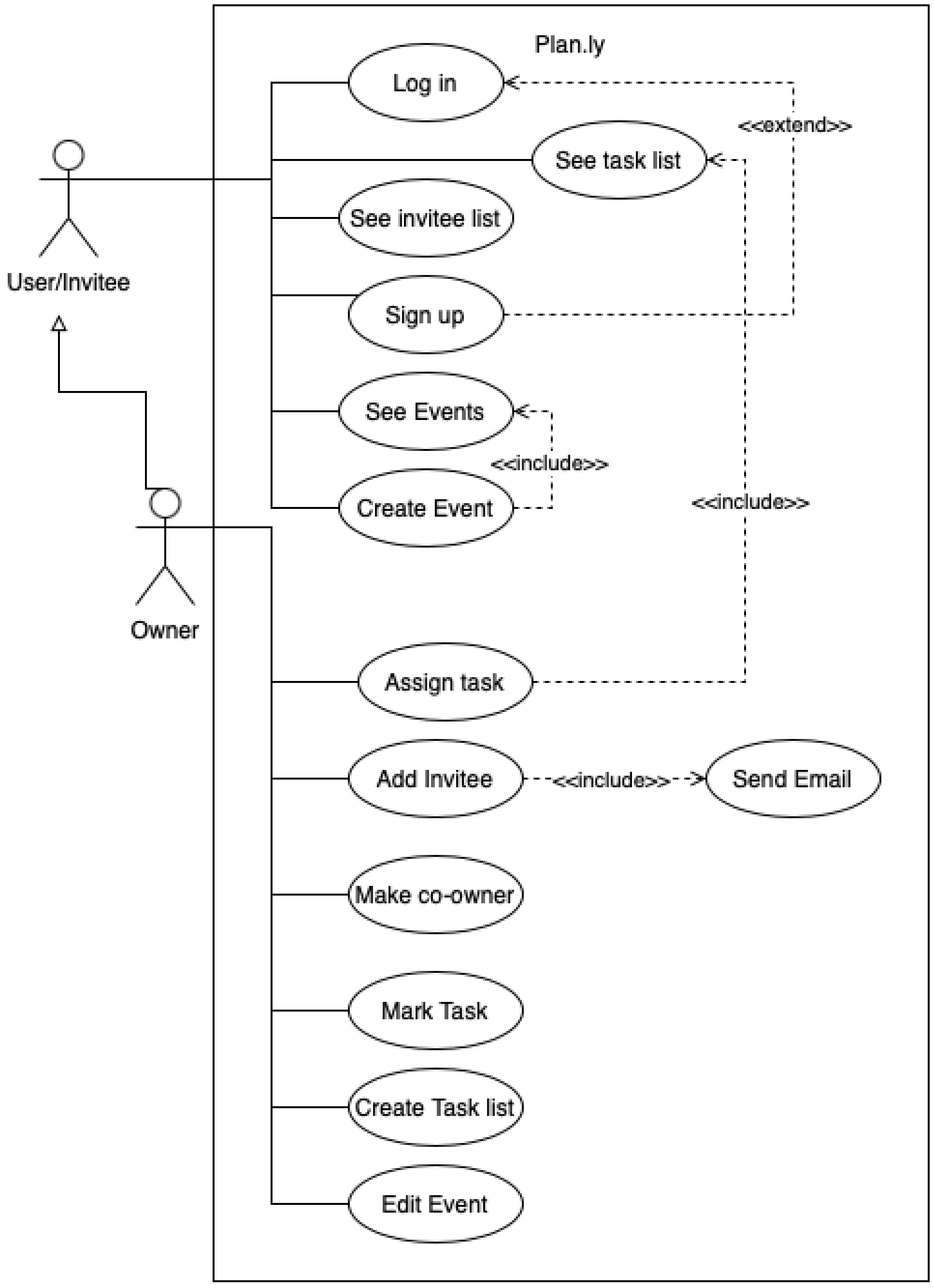
7. Diagram

This section contains two types of diagrams. These diagrams include the actual implementation made. We added only two descriptions for the use case because the use cases are similar to the user stories. We chose two important stories for the sequence diagram to show how the inside process/transactions are.

a. Use case

i. Diagram

The use case diagram below shows the functional requirements of Planly. Here we use the inheritance relationship between actors. “**User/Invitee**” can log in, log out, edit user profile, view events, create an event, see task list and see invitees. “**Owner**” inherits from “**User/Invitee**”, thus can perform all above use cases that “**User/Invitee**” can do. Besides, “**Owner**” can also assign tasks, add invitees, mark tasks, edit an event, create a task list and make an invitee a co-owner.



[Use case diagram]

ii. Description

Name:	Add an Event	**Initiating actor:**	User/Invitee
Entry condition:	→ The user is logged into Planly → The user is on the dashboard page	**Exit condition:**	→ The user becomes an Owner of the created event. → The user receives display information about the error.
The flow of events: <Basic Flow>			
User/Invitee Steps		**Planly steps**	
1. Clicks on Create event 3. Input information about the event		2. Displays the pop up for input information 4. Validates inserted information 5. Create a new event and save it into the database 6. Displays the dashboard page with the new event	
The flow of events: <Alternative Flow>			
User/Invitee Steps		**Planly steps**	
3. Input information about the event [Invalid information]		[Invalid information] Displays the failure.	

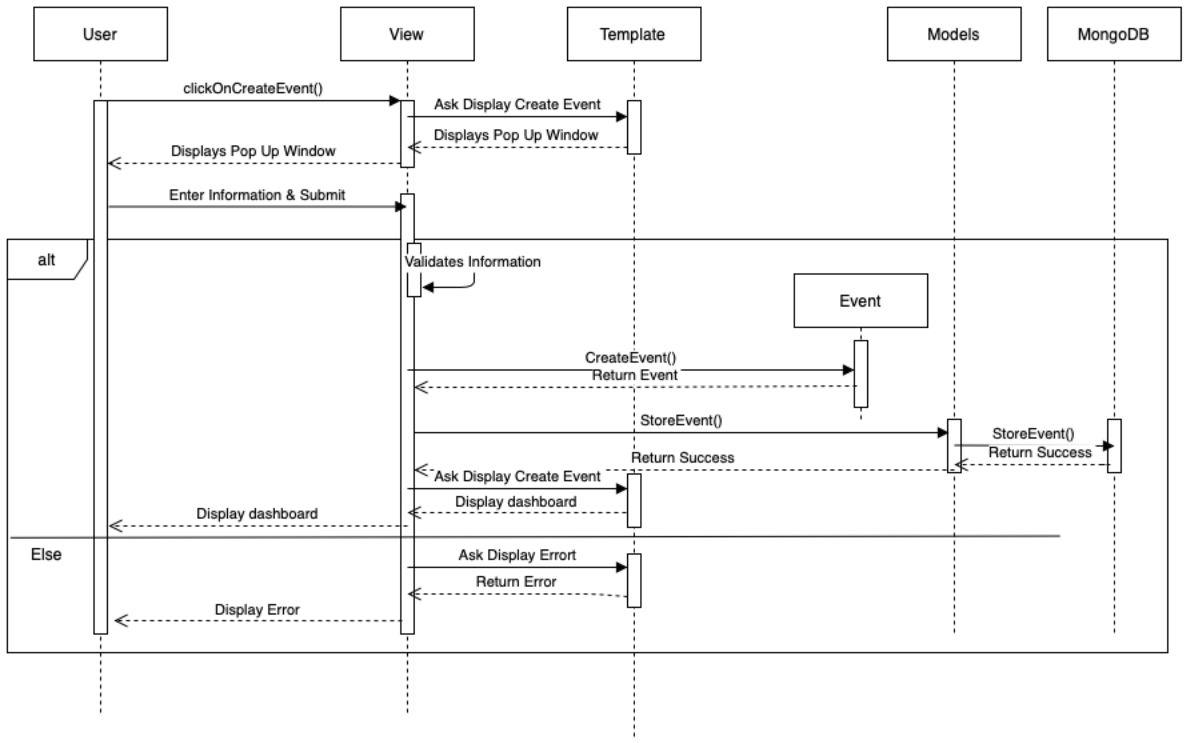
Name:	Add an Invitee	**Initiating actor:**	Owner
Entry condition:	→ The owner is logged into Planly. → The user is on the event page. → The invitee has to have an account with Planly.	**Exit condition:**	→The invitees receive an email

The flow of events: <Basic Flow>	
Owner Steps	**Planly steps**
1. Clicks on Add Invitee 3. Input email of invitee 4. Click on send invitee	2. Displays the pop up for input email 5. Create an event invitee and store it in the database 6. Sends email to invitees 7. Displays event page with the list of the names of invitees.
The flow of events: <Alternative Flow>	
User/Invitee Steps	**Planly steps**
3. Input email of invitee [Invalid email]	[Invalid information] Displays if the invitee has been invited or doesn't exist in Planly database.

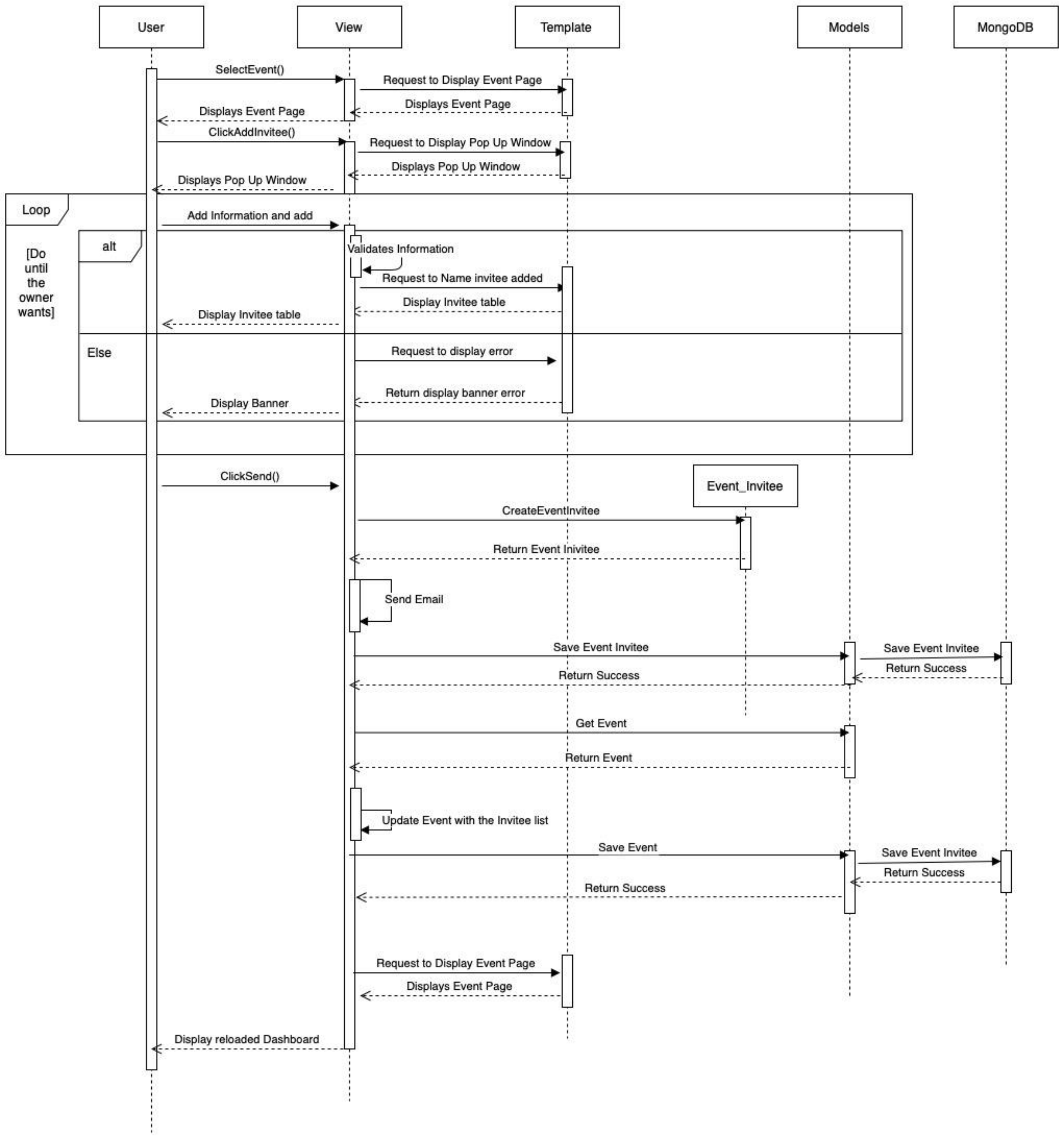
b. Sequence

Sequence diagrams are used to model the dynamic behavior between objects. Here are two sequence diagrams for the above AddInvitee and CreateEvent use cases.

Create an event: It starts with clicking the create event button by the user/invitee, then displays the create event form. After the form is filled, the view will create an event object and add it to the database. After it is done, the view will request the template to display the dashboard page to indicate success.



Add an invitee: It starts with clicking the add invitee button by the owner, then displays the add invitee form. After the email is filled, the view validates the email added. Then the owner will click on send invitation; the view will create an event invitee object and add it to the database. Also, it will update the events invitee list, and it will send an email to invitees. After it is done, the view will request the template to display the event page with the invitees' names to indicate success.



8. References

[1] https://data-flair.training/blogs/django-architecture/
[2] https://docs.djangoproject.com/en/3.1/topics/auth/
[3] https://jwt.io/introduction

9. Glossary

[1] UI: User interface

[2] MVC: Model–View–Controller
[3] HTTP: Hypertext Transfer Protocol
[4] CSRF: Cross-Site Request Forgery
[5] HMAC: Hash-based message authentication code
[6] RSA: Rivest–Shamir–Adleman
[7] ECDSA: Elliptic Curve Digital Signature Algorithm