GSoC 2023 Proposal

Talawa Admin: Improved Event and Feedback Management with Support for Stress Testing

Eshaan Aggarwal

Table of Contents

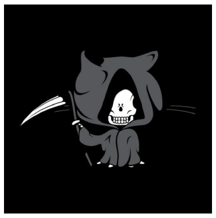
Talawa Admin: Improved Event and Feedback Management with Support for Stress Testing

GSoC 2023

Name: Eshaan Aggarwal
Major: Computer Science and Engineering
Degree: Bachelor of Technology
Year: Sophomore
University: Indian Institute of Technology, BHU (Varanasi)
GitHub Handle: @eshaanagg
Slack: Eshaan Aggarwal

Postal Address: 701, Mahagun Villa, Sector 4, Vaishali, Ghaziabad, UP (201010), India
Email: eshaan.aggarwal.cse21@itbhu.ac.in
Phone: (+91) 76786 48384
Resume: Link
Timezone: Indian Standard Time (UTC +5:30)

About Me



Hi! I am Eshaan Aggarwal, a Computer Science Engineering sophomore from IIT BHU, India. I am a full stack developer who has worked on multiple projects (as a part of my hobby or as the Tech Lead in the organization of our Institute fests). I love open-source development, and building scalable solutions for disadvantaged communities has motivated me to contribute to the Palisadoes Foundation.

You may know me better as EshaanAgg on GitHub!

My Contributions

I have been an active member of the Palisadoes Community for the last two months and have been actively engaged in discussions and reviews for all the proposed changes and functionalities.

I have **22+ merged PRs** in the Talawa API project, and **9 merged PRs** in Talawa Admin, along with some documentation PRs as well. I have also created over **14 issues in Talawa API** and **11 issues in Talawa Admin**.

I led the migration to the new **GraphQLYoga** package to replace the existing **Apollo Server**. This project came to a standstill due to the busy timeline of other contributors and mentors. Nevertheless, I hope my work in the same would provide an excellent starting point for the other contributors and help fulfill the same in GSoC ‘24.

Why GSoC and specifically the Talawa Project?

Being a technology enthusiast even before I entered college, I always enjoyed the time I spent debugging my code and PC. During my first year, I was amazed at how genuinely vast the world of Computer Science is, and there is so much for me to learn, which I genuinely enjoy. The idea of collaborating with tech enthusiasts all over the world and working with them to build amazing things attracted me to open source.

I have similar aspirations from Google Summer of Code. I want to work more in an open-source community where we have several people from different parts of the world working as a team and building projects collaboratively. The Talawa Project is an excellent opportunity for me to deep dive into the GraphQL, UI design and database management and scaling. I have always been keen to make my programs and applications more performant, fast and easy to understand, which are among the few sub-domains of Talawa ideology and principles as we the software built here to be accessible to all communities, irrespective of their access to technical resources and technology.

The project has great mentors who are always willing to provide the best possible aid and are very responsive, friendly and ready to share their knowledge. This has been a fantastic experience for me. Even during the contribution period, I have learnt soo much from the mentor’s in such a short spam, that the experience still feels too good to be true.

Background

Currently, both of the projects, Talawa-API and Talawa-Admin, provide support for basic functionality related to events (like creation, updation, and deletion of events for an organization). But this functionality still needs to be completed from a practical point of view. The system lacks some fundamental functionality required for event management (from both the organization's and the attendee's perspective). The purpose of this project includes the following:

1. Update the API models in a scalable and performant manner to provide support for new event management functionalities that are client agnostic (thus can be consumed on both the Talawa Admin and Talawa Mobile App)
2. Create all the relevant UI for these functionalities in the Talawa Admin.
3. Add a new feedback portal to the Talawa API and Talawa Admin to track the quality of organized events.
4. Introducing Artillery framework to the repository to introduce stress testing.

Design and Description of Work

This proposal involves working on the following sub-projects:

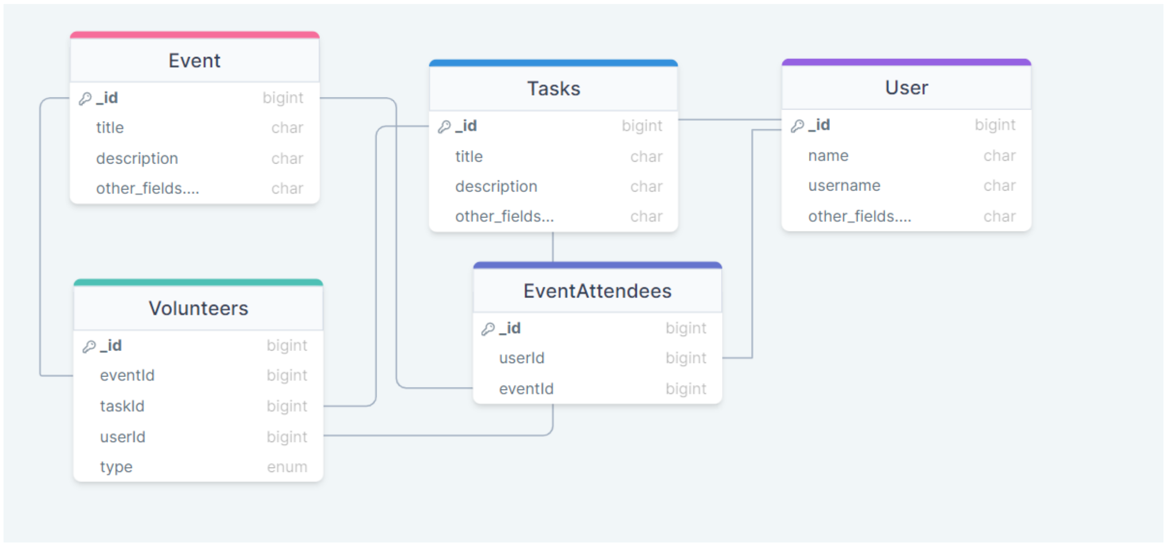
Volunteer Management

Volunteers are people attending an event and can be allotted some tasks required for the successful completion of the event. This step would involve introducing an organization and management system for volunteers:

1. *Change the Event model architecture.*

Currently, all the particular tasks for an event and all the event attendees are stored directly in the Event model itself, embedding documents in place (as an array). This approach is not ideal, as embedding documents in MongoDB should only be done when the fields are limited in number and not infinitely scalable, like in our case. The current model will be highly restrictive and would be a significant bottleneck in data retrieval and data upgradation for events.

As a general rule, for the kind of highly structured and relational data that we are dealing with, it makes more practical sense to create specific relational schemes (so that we can model relationships as in SQL), and it is an approach that is itself recommended by the core developers at MongoDB. Such relational schema work in a highly flexible way and provide granular yet efficient and performant control over the data. This is also the approach that we started migrating towards by introducing tags into the repository. It is described in great detail by Adarsh in the linked discussion.



Restructuring the models in the following way would not only allow us to use indexing of records for performance and pagination, but it would also allow us to make a simpler mental model of all the operations that can be performed. Events, Tasks, and Users are now treated as separate entities. This is important because some tasks and users might be shared with multiple events.

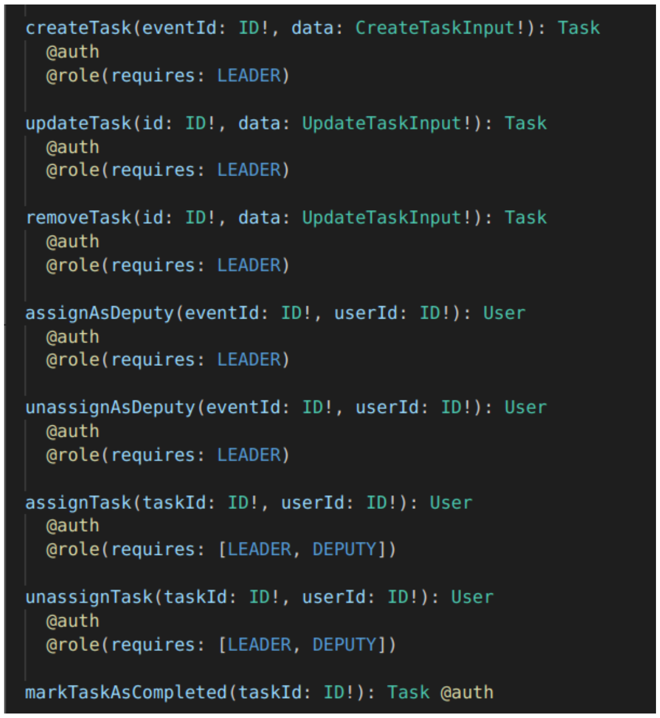
Now such a many-to-one relationship is clearly shown through the Volunteers table, where we can keep track of all the volunteers and assign them tasks. Continuing the present architecture of storing tasks in the event model would lead to data duplication and non-performant queries.

Each volunteer would also have a *role* field, which would be an ENUM used to describe the role of the volunteer concerning the event being talked about. The type can be `LEADER,` `DEPUTY,` and `VOLUNTEER.` These types would serve a two-way purpose:

1. Better classification of the volunteers participating in the organization of the event
2. Provide Role-based authorization and access to the various parts and functionalities of the event management system

As a preliminary implementation, we can ensure that only Leaders and Deputies can assign the various tasks related to an event to all the other volunteers, while the leaders can appoint deputies. All the volunteers would be able to view their tasks and would be able to mark them as completed upon completion. Leaders and deputies would be able to track the completion status of these tasks and thus assign new tasks.

The following mutations would be exposed to the client side with the relevant input types:



```
createTask(eventId: ID!, data: CreateTaskInput!): Task
  @auth
  @role(requires: LEADER)

updateTask(id: ID!, data: UpdateTaskInput!): Task
  @auth
  @role(requires: LEADER)

removeTask(id: ID!, data: UpdateTaskInput!): Task
  @auth
  @role(requires: LEADER)

assignAsDeputy(eventId: ID!, userId: ID!): User
  @auth
  @role(requires: LEADER)

unassignAsDeputy(eventId: ID!, userId: ID!): User
  @auth
  @role(requires: LEADER)

assignTask(taskId: ID!, userId: ID!): User
  @auth
  @role(requires: [LEADER, DEPUTY])

unassignTask(taskId: ID!, userId: ID!): User
  @auth
  @role(requires: [LEADER, DEPUTY])

markTaskAsCompleted(taskId: ID!): Task @auth
```

```
# Returns the list of tasks assigned to me
myTasks: [Task] @auth

# Returns the list of tasks completed by me
myCompletedTasks: [Task] @auth

# Get the leaders of an event
leaders(eventId: ID!): [User]

# Get the deputies of an event
depuites(eventId: ID!): [User]

# Get the volunteers of an event
volunteers(eventId: ID!): [User]
```

We would also expose the above queries to the frontend so that the client-side apps can consume the same to show various kinds of information to the users.

We would also add a new field called `**completed**` of the boolean type of the Volunteer model to track its completion status. This would also allow us to generate statistics for task completion, which can be displayed on the event dashboard (described in more depth later).

This field is added to the Volunteer model, as it makes sense to keep track of the completion status of a task only after it has been assigned to a user. Also, keeping the same separate from the actual Task model allows for the reusability of various tasks across different events created in the same parent organization.

The mongoose model for the **Volunteer Schema** would look like the following:

```
import { Schema, model, PopulatedDoc, Types, Document, models } from "mongoose";
import { Interface_Event } from "./Event";
import { Interface_User } from "./User";
import { Interface_Task } from "./Task";

export interface Interface_Volunteer {
 _id: Types.ObjectId;
 role: string;
 taskId: PopulatedDoc<Interface_Task & Document>;
 eventId: PopulatedDoc<Interface_Event & Document>;
 userId: PopulatedDoc<Interface_User & Document>;
 completed: boolean;
}

const volunteerSchema = new Schema({
 role: {
   type: String,
   required: true,
   enum: ["LEADER", "DEPUTY", "VOLUNTEER"],
   default: "VOLUNTEER",
 },
 eventId: {
   type: Schema.Types.ObjectId,
   ref: "Event",
   required: true,

},
userId: {
  type: Schema.Types.ObjectId,
  ref: "User",
  required: true,
},
taskId: {
  type: Schema.Types.ObjectId,
  ref: "Task",
  required: true,
 },
 completed: {
  type: Boolean,
  required: true,
  default: false,
 },
});

// We will also create an index here for faster database querying
volunteerSchema.index({
 userId: 1,
 taskId: 1,
 eventId: 1,
});

const VolunteerModel = () =>
 model<Interface_Volunteer>("Volunteer", volunteerSchema);

// This syntax is needed to prevent Mongoose OverwriteModelError while running tests.
export const Volunteer = (models.Volunteer || VolunteerModel()) as ReturnType<
 typeof VolunteerModel
>;
```

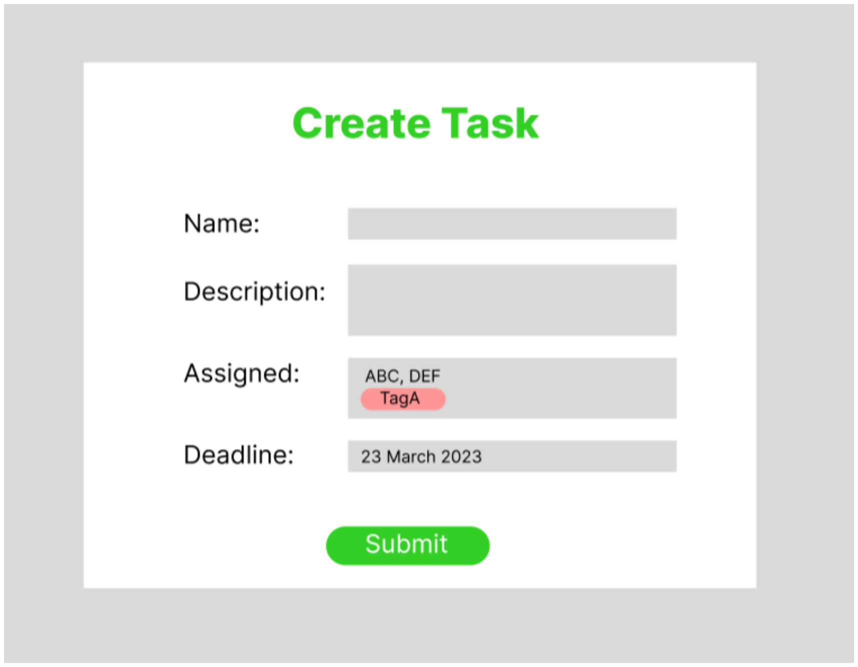
A very similar and minimal relational schema would also be created for the **EventAttendees**, which would look like the following:

```
import { Schema, model, PopulatedDoc, Types, Document, models } from "mongoose";
import { Interface_Event } from "./Event";
import { Interface_User } from "./User";

export interface Interface_EventAttendee {
 _id: Types.ObjectId;
 eventId: PopulatedDoc<Interface_Event & Document>;
 userId: PopulatedDoc<Interface_User & Document>;
}

const eventAttendeeSchema = new Schema({
eventId: {
  type: Schema.Types.ObjectId,

  ref: "Event",
  required: true,
},
userId: {
  type: Schema.Types.ObjectId,
  ref: "User",
  required: true,
},
});

// We will also create an index here for faster database querying
eventAttendeeSchema.index({
 userId: 1,
 eventId: 1,
});

const EventAttendeeModel = () =>
 model<Interface_EventAttendee>("EventAttendee", eventAttendeeSchema);

// This syntax is needed to prevent Mongoose OverwriteModelError while running tests.
export const EventAttendee = (models.EventAttendee ||
 EventAttendeeModel()) as ReturnType<typeof EventAttendeeModel>;
```

2. *Add screens to create tasks and assign volunteers to a task.*

Currently, the admin panel only allows us to create new events. We would develop similar frontend screens to allow for the addition of the basic CRUD (Create, Read, Update, and Delete) operations related to tasks for events.

This is the proposed wireframe for the creation of tasks. Similar wireframes for the updation and deletion of tasks would also be created. While the actual implementation of the proposed wireframe would be highly dependent on the related *Admin UI Redesign* GSoC project (as that project would be responsible for introducing new CSS frameworks such as Tailwind CSS or Windicss, the typical layouts and design principles), the proposed wireframe must be seen as an example to show the kind of functionalities that we would be exposed to the client/user on the Admin panel.



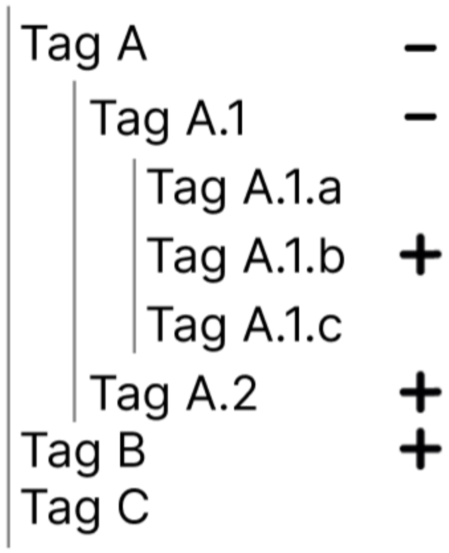
After the same is implemented, functionality would be implemented to assign and unassign volunteers to particular tasks. The functionality would use the previous section's queries and mutations exposed on the Tasks model.

Note that we would be providing functionality to assign individual users and users by tags, as it would make the task of management and characterization easier for the admins.

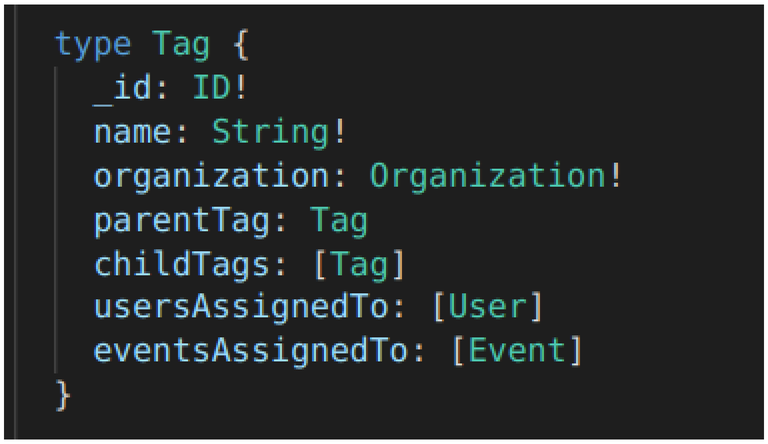
3. *Add support for tags.*

As a part of this issue, I have already worked extensively on implementing the backend to support tags (and their nesting in folders for easy classification) for all object types, like users. This step would deal with introducing a simple UI in the Admin panel, where the user (super

admin or admin) could bulk assign users (attendees and volunteers) to an event with the help of the tags that had been previously created.
As in the wireframe shown above, the UI would be developed so that multiple users can be directly added to the event with the help of tags. Having a clear and concise vision of how the frontend should be implemented would help me.



The display of tags would be done with the help of a nested architecture, where all the sub-tags of a tag can be loaded and minimized sequentially by clicking the relevant icons. This would be possible due to the already existing parentTags and childTags fields on the Tag Schema, which are already implemented in the backend.



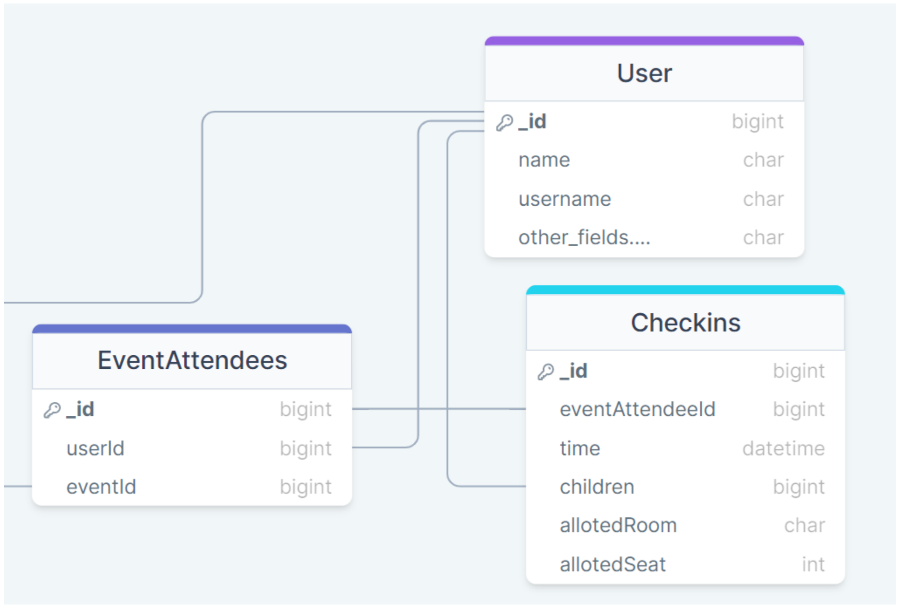
```
type Tag {
  _id: ID!
  name: String!
  organization: Organization!
  parentTag: Tag
  childTags: [Tag]
  usersAssignedTo: [User]
  eventsAssignedTo: [Event]
}
```

Check-ins

Another essential part of event management is to help track the proposed attendees' attendance and smooth facilitation. To implement this feature, this proposal would work on the following sub-projects:

1. *Introduce a new Checkins table.*

The Checkins table would store the actual instance of an Event Attendee checking into the event (i.e., attending the event). We would only be storing some primary data for the check-in, such as the actual time the check-in happened, the room, and the seat allotted to the particular user in the event, as the rest of the data can be simply fetched with the help of the relationships defined (due to our SQL like approach).



We have also introduced a field called **children**, which will be used to store document references to the children users attending the event with the parent. This is done to ensure that unique benefits can be given to such users, like priority entry and exit times, functionality to provide children and parents with neighboring seats, etc.

The mongoose model for the **CheckIn** table would look like this:

```
import { Schema, model, PopulatedDoc, Types, Document, models } from "mongoose";
import { Interface_EventAttendee } from "./EventAttendees";
import { Interface_User } from "./User";
```

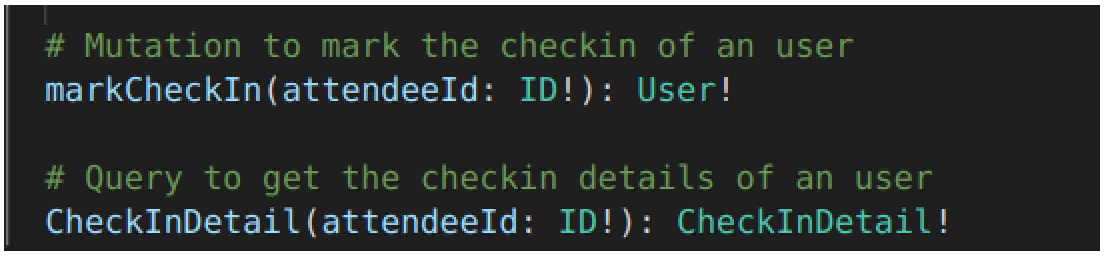
```
export interface Interface_CheckIn {
 _id: Types.ObjectId;
 eventAttendeeId: PopulatedDoc<Interface_EventAttendee & Document>;
 time: Date;
 children: Array<PopulatedDoc<Interface_User & Document>>;
 allotedRoom: string;
 allotedSeat: string;
}

const checkInSchema = new Schema({
 eventAttendeeId: {
   type: Schema.Types.ObjectId,
   ref: "EventAttendee",
   required: true,
 },
 children: [
   {
     type: Schema.Types.ObjectId,
     ref: "User",
     required: false,
   },
 ],
 time: {
   type: Date,
   required: true,
   default: Date.now,
 },
 allotedRoom: {
   type: String,
   required: false,
 },
 allotedSeat: {
   type: String,
   required: false,
 },
});

// We will also create an index here for faster database querying
checkInSchema.index({
 eventAttendeeId: 1,
});


const CheckInModel = () => model<Interface_CheckIn>("CheckIn", checkInSchema);

// This syntax is needed to prevent Mongoose OverwriteModelError while running tests.
```

```
export const CheckIn = (models.CheckInModel || CheckInModel()) as ReturnType<
 typeof CheckInModel
>;
```

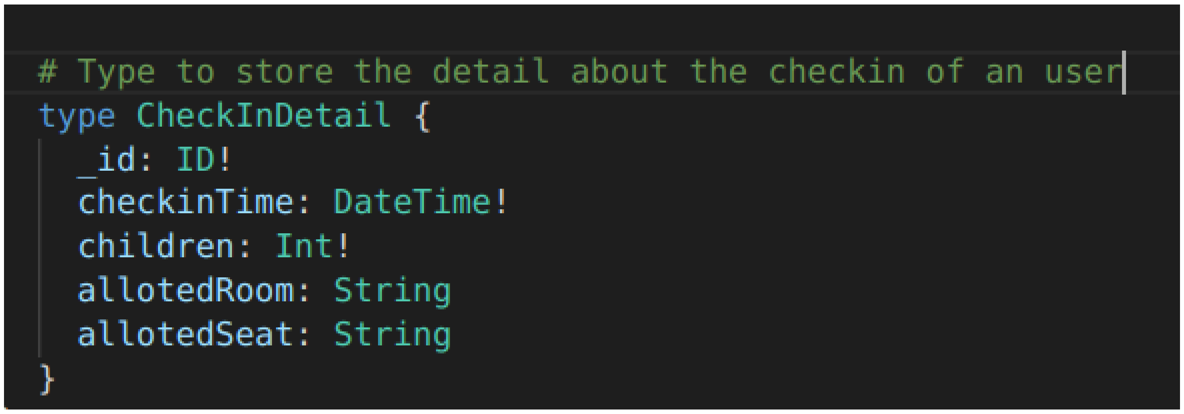
As evident from the schema here, we are storing the basic details only about the checkIn related information of the user. It must also be noted that instead of creating one more relational schema to keep all the information of the children of a particular event attendee, we are using the native embedding of documents to store this information.

The reason for the same is also simple. Since the number of children that any user would have would be minimal, limited (and in no way infinitely scalable), it makes much more sense to embed the documents directly (so that they can be populated in one database query itself) for faster retrieval.

The following additions would be made to the typeDefs schema as well to consume the newly introduced checkIn model:



```
# Mutation to mark the checkin of an user
markCheckIn(attendeeId: ID!): User!

# Query to get the checkin details of an user
CheckInDetail(attendeeId: ID!): CheckInDetail!
```



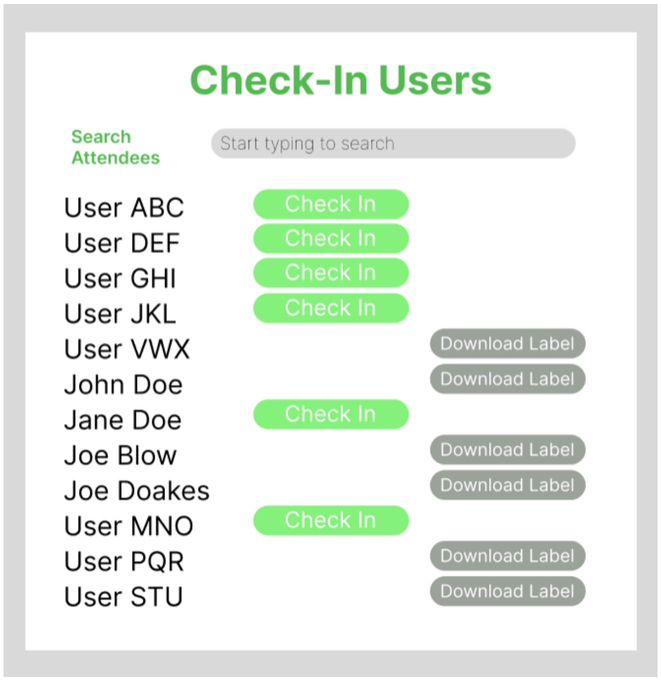
```
# Type to store the detail about the checkin of an user
type CheckInDetail {
  _id: ID!
  checkinTime: DateTime!
  children: Int!
  allotedRoom: String
  allotedSeat: String
}
```

2. *Implement UI for Check-In*

This step would involve creating a new screen on the Admin panel that would use the functionality introduced in the previous step to create a simple interface where the volunteers/admins can search through the list of attendees and allot their check-in.

First, we would introduce a check on the Event Detail Page itself, where if the user is assigned as a volunteer for the event, and the current system time is between the scheduled time of the event, then a custom button to mark the check-in's would be shown to the volunteer. This button would do the task of redirecting them to a new route, “**/checkin/:eventID**,” where the following screen would be shown to them:

Here is the proposed wireframe showing the same:



While marking the check-in of the user, the volunteer would be given two primary options:

1. A button to mark their check-in would be shown for users that still need to check-in. When the button is clicked, a database request is made to mark their check-in.

2. A “Download Label” button will be shown for the users that have already checked in. The custom sticky label would be generated as a pdf, which could be either printed or just shared with the user by means of any communication that they prefer.

   The purpose of this label is to print some basic information about the user, like their name and their seat information, so that not only the other people in the party can quickly identify and mingle with the attendee, but also the attendee himself doesn’t have to take the burden of remembering his seating information.

   For this generation of sticky labels, we would be using the open-source and typescript-enabled library pdfme! The library allows us to pre-define a template to inject the user's details to generate a pdf label in the browser, thus providing no additional load on the client.

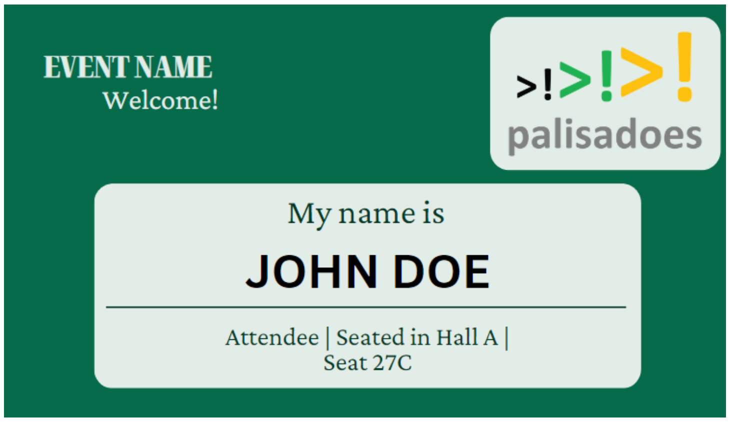
This library comes pre-baked with a designer, templating engine, and a generator so that a single take can accomplish all the configuration.

The implementation of the same would be achieved in the following steps:

1. Use the designer provided to develop a simple template for the cards. This step would be done on the developer's system; thus, no additional load would be added to the client side. Only the finally generated PDF template will be added to the Talawa Admin repository.

2. While designing the pdf, we would be creating multiple **domContainer's** and providing them with a specific name that can be used to access them and populate them with different values. On the client side, only the following minimal script would be required:

```
import { Template, generate } from "@pdfme/generator";

const template: Template = {
  basePdf, // Provide path to the template generated from
https://pdfme.com/template-design
  schemas: [
    {
      // All these values of x,y,height and width can be fetched from the site
      eventName: {
        type: "text",
        position: { x: 0, y: 0 },
        width: 10,
        height: 10,
      },
      userName: {
        type: "text",
        position: { x: 10, y: 10 },
        width: 10,
        height: 10,
      },
      role: {
        type: "text",
        position: { x: 20, y: 20 },
        width: 10,
        height: 10,
      },
      seat: {
        type: "text",
        position: { x: 20, y: 20 },
        width: 10,
        height: 10,
      },
    },
  ],
};

// These inputs would be dynamically fetched from the database

const inputs = [
  {
    eventName: "Test Event",
    userName: "John Doe",
    role: "Attendee",
    seat: "Seated in Hall A | Seat 27",
  },
];

generate({ template, inputs }).then((pdf) => {
  // Creates a pdf and open's it in the browser
  // Thus eliminating all needs of configuring some storage option
  const blob = new Blob([pdf.buffer], { type: "application/pdf" });
  window.open(URL.createObjectURL(blob));
});
```



This is an example of a straightforward name card that can be generated in this step. At this step itself, we could implement *unique name cards for children and parents together* to ensure a visible way to demarcate them. The labels generated for children would clearly state the contact numbers of the parents so that their safety can be provided while attending the event. Thus, these name cards would also serve as a much-needed security feature.

Event Dashboard Management

On the day of the event, we can make a new component to be rendered on the Admin panel that can be used to track various attendance-related statistics for the event, which will help the organizers gauge the impact the event had on their community.

All of the statistics would be calculated on the backend. It would be displayed on the front end with the help of the React-Chartjs-2 library, a custom typescript, and a React wrapper for the community's favorite ChartJS library. Using the same, we can display the following statistics on the event dashboard:

1. *Attendance*: Count and percentage of the attendees that have already checked into the event. The attendance statistics of the event's previous instances can also be fetched for recurring events. Then a comparative display of the progression of attendance would be displayed.

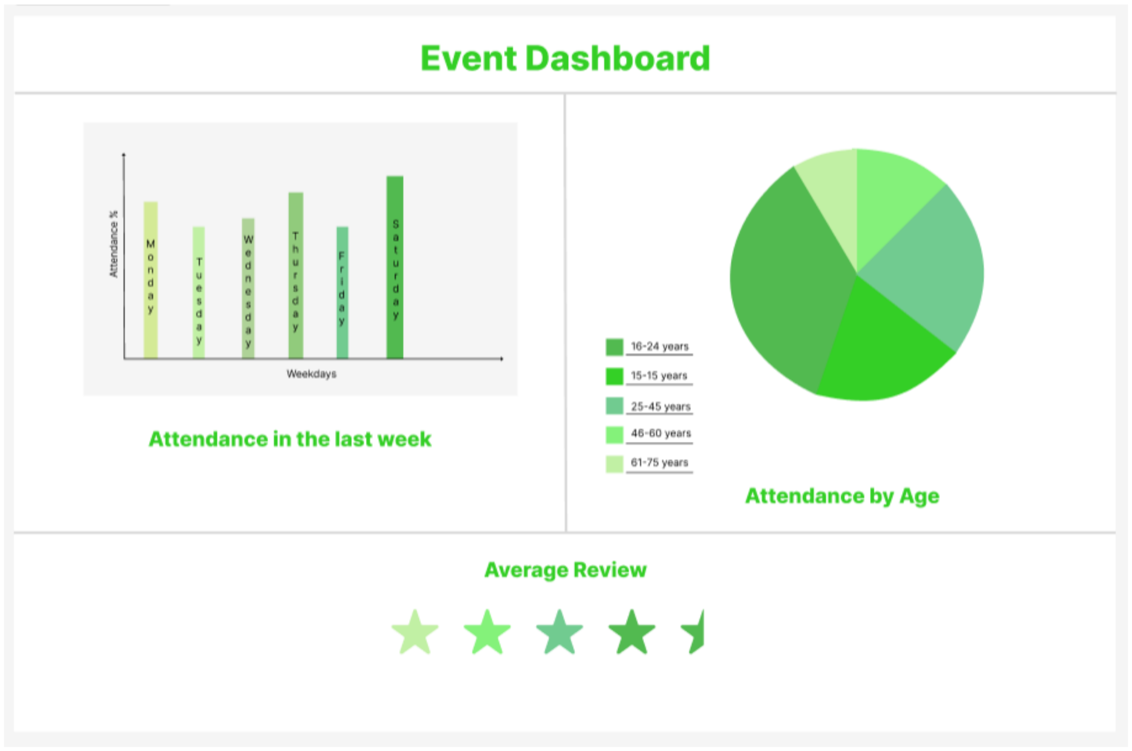
2. *Proportions:* We can use pie charts to display the composition of our attending audience by various fields, such as their age, gender, and location.

3. *Task Completion:* We can use various graphs to show statistics related to the percentage completion of the multiple tasks that were planned as a part of the event, and can even get detailed graphs for statistics pertaining to individual volunteers (like most tasks completed, the average time to complete the task, etc.).

4. *Feedback**:*** Various statistics (like average rating of the event, rating average by age, the most commonly used word in the reviews, etc.) can also be shown for the event. The details about the implementation of the same are covered in the next section.

Here is a sample script that shows both the server and client implementations to get the attendance percentage statistic for a particular event and plots a pie chart.

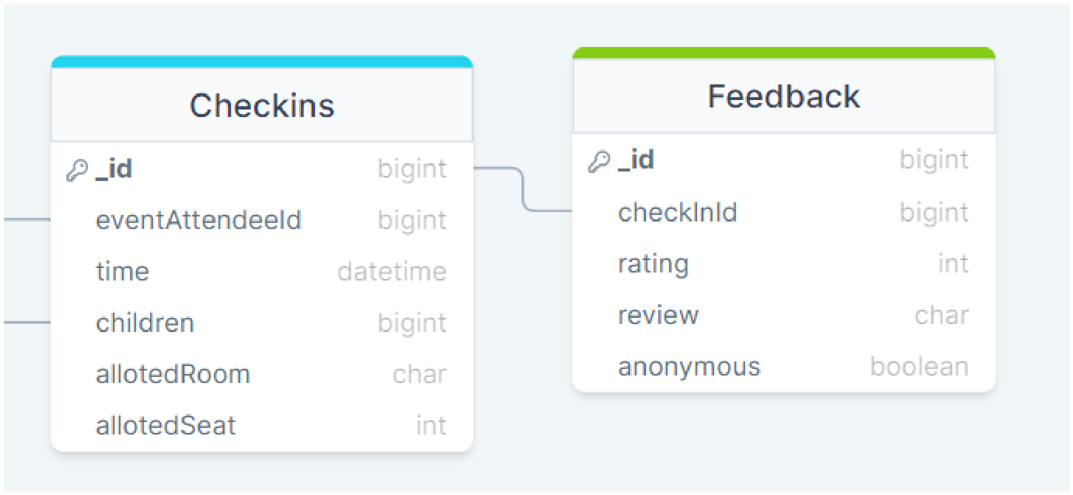
```
import { EventAttendee } from "./EventAttendees";
import { CheckIn } from "./CheckIn";

// The event for which the attendance stats have to be calculated
const EVENT_ID = "ABCD";

//Server-side code to generate the statistics which would
// be exposed via a query
const getAttendancePercentage = async () => {
 const attendees = await EventAttendee.find({
   eventId: EVENT_ID,
 })
   .select({
     _id: 1,
   })
   .lean();

 const checkInCount = await CheckIn.count({
   _id: {
     $in: attendees.map((attendee) => attendee._id),
   },
 });

 const attendancePercentage = checkInCount / attendees.length;
};

// Client side code
// React Code to display a pie chart of the attendance
import { Chart as ChartJS, ArcElement, Tooltip, Legend } from "chart.js";
import { Pie } from "react-chartjs-2";

ChartJS.register(ArcElement, Tooltip, Legend);
const attendance = await getAttendancePercentage();

const data = {
 labels: ["Present", "Absent"],
 datasets: [
   {
     label: "# of Votes",
     data: [attendance, 1 - attendance],
     // Green and red colour
     backgroundColor: ["rgba(255, 99, 132, 0.2)", "rgba(54, 162, 235, 0.2)"],
     borderColor: ["rgba(255, 99, 132, 1)", "rgba(54, 162, 235, 1)"],
     borderWidth: 1,
   },
 ],
};

export function App() {
 return <Pie data={data} />;
}
```



The same component can also be reused on the organization level to show the attendance statistics for all the events organized by a particular organization or the basis of a date range. In addition to all these statistics, all the event attendees would be shown on the user dashboard so that the volunteers could quickly look up their contact details.

Checkouts and Feedback

The API can also be extended to include a feedback model, which would work in close relation to the Checkin model introduced earlier. This would allow every event attendee who could participate in the event to provide feedback to the organizers.



The feedback can be either numeric: a rating (of 0-10) or textual: a review. Users would be given a choice to make their feedback anonymous so that they are not concerned about their privacy being violated and can provide genuine feedback. Since there would be a one-to-one relationship between check-in and feedback submitted, it would ensure that only honest feedback is submitted.

While the filling of the reviews would be done on the Talawa mobile application (as a notification issued after the event is completed using Firebase Cloud Messaging service), we can implement screens for the feedback section in the Admin panel as well, where the organizer of the event would be able to see all the submissions. The statistics from the same can also be displayed on the event dashboard.

The following would be the mongoose schema of the same:

```
import { Schema, model, PopulatedDoc, Types, Document, models } from "mongoose";
import { Interface_CheckIn } from "./CheckIn";

export interface Interface_Feedback {
 _id: Types.ObjectId;
 checkInId: PopulatedDoc<Interface_CheckIn & Document>;
 rating: number;
 review: string | null;
 anonymous: boolean;
}
```

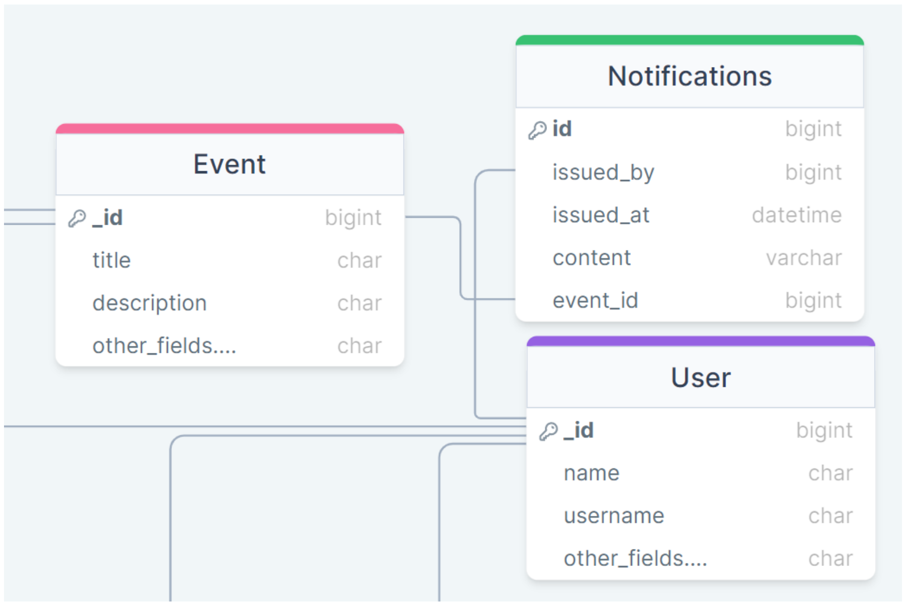
```
const feedbackSchema = new Schema({
 checkInId: {
   type: Schema.Types.ObjectId,
   ref: "CheckIn",
   required: true,
 },
 rating: {
   type: Number,
   required: true,
   default: 0,
   max: 10,
 },
 review: {
   type: String,
   required: false,
 },
 anonymous: {
   type: Boolean,
   required: true,
   default: false,
 },
});

// We will also create an index here for faster database querying
feedbackSchema.index({
 checkInId: 1,
});

const FeedbackModel = () =>
 model<Interface_Feedback>("Feedback", feedbackSchema);

// This syntax is needed to prevent Mongoose OverwriteModelError while running tests.
export const Feedback = (models.FeedbackModel || FeedbackModel()) as ReturnType<
 typeof FeedbackModel
>;
```

The same has been designed while keeping in mind to keep the system performant enough to quickly and efficiently generate the data to be displayed on the dashboard of the organization.

Event notifications and reminders

It might be necessary for the `Leaders` of an event to convey some information to the attendees of an event during the progression of the event. To implement this, we will create a notification panel on the event dashboard, which will only be accessible to the administrators of the organization of the event, and the event leaders.

The same can be used to send text messages as notifications to the users on the mobile app. They can compose the message and save the same using the Admin UI. This would request Talawa-API, where the same would be run through a complete communication script.



The proposed communication script would work in multiple steps:

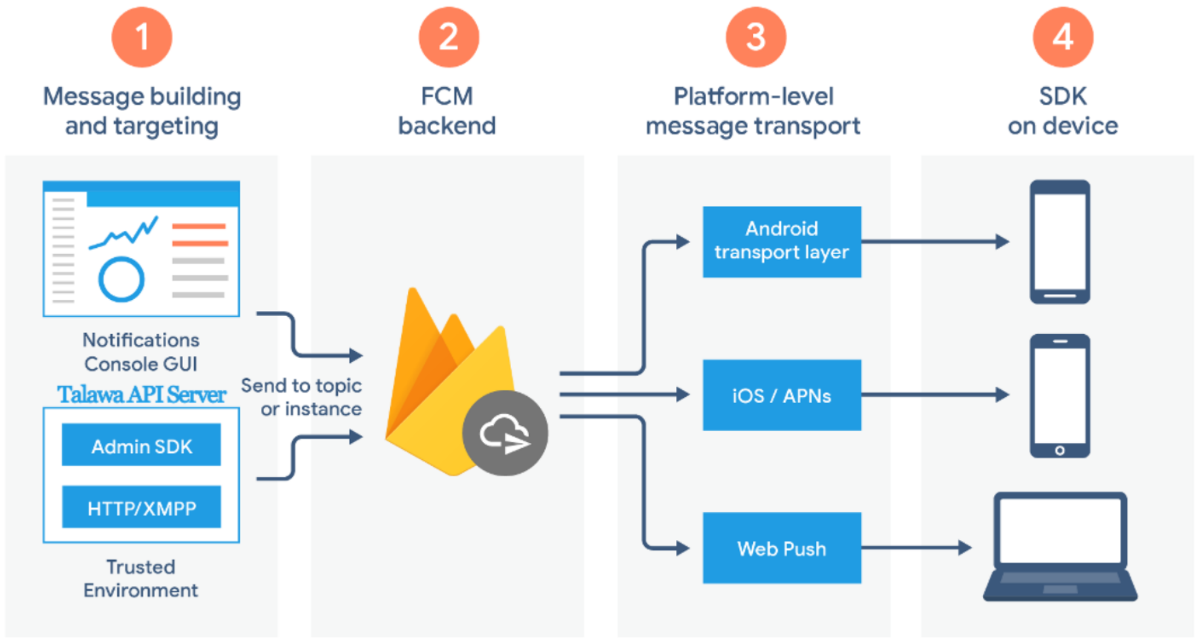
1. Details of the composed message will be run through bad-word filters (such as this) to ensure that the message meets the community guidelines. (As a future prospect, this can be later modified to use more sophisticated NLP models that can actually read and decode the tone of the message and determine if the same should be broadcasted to all of the users or not)

2. If no such violations are found, the message details would be stored in a new table called Notifications so that the same can be later used to keep track of all the communication during the event.

3. The backend would then use the Firebase’s Admin SDK for cloud messaging to send the message to all the clients (IOS and Android). This is perhaps the most crucial step, as this is where we would be mapping the users existing in the backend with their devices.

This FCM implementation primarily works in two components:

1. Creating a message/notification on the app server and using Firebase’s Admin SDK with cloud messaging to target and send these messages.
2. The Talawa-Mobile app would receive this message using the exposed transport vehicle by the FCM console.



While the Notifications Console GUI provided by Firebase can also be used to send notifications, we will primarily be using the Talawa API Server as a trusted environment, from where we would instantiate a request and then leverage platform-level message transport provided by the same to provide notifications to all types of users.

I have been in close contact with other active contributors on the Talawa repository about implementing the same. It is a common consensus that the FCM is the easiest and the most scalable way to introduce notifications to our project.

4. Use the already configured `nodemailer` along with SMTP to send email notifications to the users about the same.

Optional Step:-

This type of mobile notification system would be set up in the backend for the first time and thus would require great care while being implemented. However, once the same is done, we can further extend the functionality of the same to issue event reminders (the event is about to start) automatically for all the registered users, notifications for the preparation of events, etc.

Introduction of Stress Testing

The primary purpose of the introduction of load testing on the API side is to see if the proposed new features are practical and can be supported by the kind of hardware most communities can access. Having access to load tests (or a simple guide with configured frameworks) to simulate a large number of requests would allow future developers to understand better how useful the then-proposed features would be and if they fit the needs of the Palisadoes Community.

Here is a detailed guide on how the project would introduce load testing in the Talawa API:

- **Defining the objectives**:- The primary goals of load testing would be to ensure that the core functionalities of Talawa API are efficiently implemented. These core functionalities currently include (but are not limited to) the following:

    1. Users must be able to signup and log in through API.
    2. Users must be able to create, update and delete posts, and all the posts of an organization must be filtered and presented efficiently.
    3. Administrators of an organization must be able to create and update any number of events for the members of the organization.
    4. Tasks can be created for the events mentioned above, and attendees and volunteers must be able to be assigned to created events.
    5. The checkin and the feedback portal must be able to work an scale for all the created events.

The project aims to introduce and implement these into the Talawa API on a first-priority basis. At the same time, other core features can also be added with the help of other contributors if the scheduled time allows.

- **Choose a load testing tool**: There are several load testing tools available that can be used to test the API, such as Apache JMeter, Locust, and Gatling. We will use **Artillery** as the application of our choice for the following reasons:

1. It requires **ZERO** additional configuration as it can be installed just like any other npm package by including it in the package.json.
2. Artillery can simulate thousands of users concurrently, making it easier to replicate real-world usage patterns and test the application's ability to handle heavy traffic. It is **highly performant and lightweight**.
3. Artillery has **built-in support for GraphQL**, making it easy to test the performance of GraphQL APIs.
4. Artillery can be integrated with various monitoring tools like Graphite, InfluxDB, and Prometheus, allowing you to capture and analyze performance metrics during the test run.

   Note: The current project would not focus on integrating these monitoring tools, as they require additional configuration and a certain level of processing power on the user's PC. This can be treated as a future prospect.

5. Artillery provides **extensively detailed reports** that help identify performance bottlenecks, errors, and other issues. These reports can be used to fine-tune the application and improve its performance.
6. The artillery stress tests are written in **YML** and thus read like English and are self-descriptive in nature. Since artillery is a node framework, the syntax of mutations and queries being used in it is precisely the same as used by our front end, i.e., Talawa Admin and Talawa Mobile. Thus all the existing developers can directly start using the same and writing their own tests to identify the bottlenecks without the need to learn additional syntax.

- **Set up the load testing environment**: Just like we have configured a separate testing database, we will set up a different `Load Testing Database,` the default database connected to whenever the artillery tests are run via the command line.

  To accomplish the same, we will use the NODE_ENV flag, which can be set to `LOAD_TEST` whenever the following is being executed, and thus would connect to a specific LOAD_TESTING_DATABASE automatically, without any additional configuration required from the user. We also ensure the load testing database is cleared after every test run to prevent it from bloating with unnecessary data.

- **Create test scenarios**: This step would involve creating test scenarios that simulate real-world usage of the API. This includes simulating user behavior, such as registering, logging in, querying data, and posting data.

  As a sample, the following would be the test scenario to simulate SUPERADMIN's of an organization signing up and creating ten new organizations each, which they would be responsible for managing.

```yaml
config:
 target: "http://localhost:4000/graphql"
 # Describes the load testing phases
 phases:
   - duration: 60
     arrivalRate: 1
     name: Warm up
   - duration: 120
     arrivalRate: 5
     rampTo: 50
     name: Ramp up load
   - duration: 600
     arrivalRate: 50
     name: Sustained load

# Describes the various parts of the scenario being simulate
scenarios:
 - name: "SUPERADMIN sign's up and then creates organizations"
   flow:
     # Sign up the user and get it's access token
     - post:
         url: "/"
         json:
           # Send the singUp mutation query
           query: |
             mutation signUp($data: UserInput!) {
               signUp(data: $data) {
                 accessToken
                 user {
                   email
                 }
               }
             }
           # Provide variables to the mutation
           variables:
             data:
               firstName: "fname-{{ $randomString() }}"
               lastName: "lname-{{ $randomString() }}"
               email: "user-{{ $randomString() }}@palisadoes.org"
               password: "superSecretPassword123$"
               userType: "SUPERADMIN"
         # Store the received data in variables
         capture:
           json: "$.data.signUp.accessToken"
           as: "token"
```

```
    # Create a loop to create 10 organizations
  - loop:
      - post:
          url: "/"
          # Send the authorization header to authenticate user
          headers:
            Authorization: "Bearer {{ token }}"
          # Send the JSON data to create the organization
          json:
            query: |
              mutation createOrg($data: OrganizationInput!) {
                createOrganization(data: $data) {
                  _id
                }
              }
            variables:
              data:
                description: "Test Organization"
                name: "Organization {{ $loopCount + 1 }}"
                isPublic: true
                visibleInSearch: true
    count: 10

  # List all the organizations
  - post:
      url: "/"
      json:
        query: |
          query OrganizationsQuery() {
            organizations() {
              _id
              name
              description
              location
            }
          }
```

As can be seen from the above, a bare-bone YML file is all that needs to be stored in the repository to make it stress-testable. Artillery can be installed globally on the developers' computer and thus would be a **strictly opt-in** feature, which the developer can access and try if they want to analyze the effects of stress testing and make changes accordingly.

NOTE: All the entity relationship diagrams for the proposed changes can be viewed here.

Results for the Palisadoes Community

The Talawa project aims to help open-source communities manage their functioning efficiently. Events are an integral part of the same, as the community actually comes together during these events. This project aims to make managing all such events from the Talawa Admin completely functional and implement the design and architecture for the same in the Talawa API in a scalable and efficient manner.

In addition to the same, the project would also be introducing a modularised and self-contained communication script, which could be consumed at multiple other instances throughout the project. This communication would be necessary, as it would be the first instance of direct communication (in the form of notifications) between the Talawa mobile app and the Talawa API and Admin. The same can also be configured with other means of communication (other integrations such as Slack, Discord, etc., specific to the community) too later.

The project, in addition, would also introduce a feedback system with statistics and a display of statistics so that the communities can get an adequate representation of the impact of their events and how they can be improved. This is unique to this proposal and wasn’t included in the other project ideas.

As Palisadoes Foundation shows an affinity for long projects, upon the recommendation of mentor Dominin Mills sir, I have also included the idea of introducing stress testing into the repository so that the scalability of all the implemented features can be measured.

Deliverables and Scheduling

I would proceed with implementing the proposed changes in the order described above, as all the later functionalities would directly/indirectly depend on the previous ones for implementation/testing.

As the project involves working with both the backend architecture and front-end design, my time commitment would be divided as follows:

1. 40%: API redesign, optimization of the implemented models for querying and filtering, and implementation of the communication script
2. 60%: Admin Portal design, creating new components and screens that can utilize the features that the backend supports.

In addition to the same, I would spend approximately 65% of my time in the actual implementation of the same features, 20% in adding proper documentation and extensive and quality tests (to make sure all the features work as they are supposed to and do not break the functionality) and rest 15% in correcting the bugs that may be created due to the same (and in the coordination of efforts and active communication with Admin UI Redesign project, as these projects would go alongside).

The proposed project has been categorized into six sub-projects, all receiving equal time commitments of about 60 hours spread over three weeks (with an average dedication of 3-4 hours per day). This would lead to the project being completed in a timeline of 350 hours spread across 18-20 weeks.

Detailed Project Scheduling

Week 1-3 (May 29 - June 18)	● Restructuring all the pre-existing tables in the models ● Adding the proposed Volunteers and EventAttendees table ● Writing all the relevant mutations and queries for the same with tests ● UI screen development for creating tasks and assigning Volunteers
Week 4-6 (June 19 - July 9)	● Writing tests for the above screens ● Add support for tags to the UI of the admin panel ● Introduction of the Checkin’s schema ● Add relevant mutations and queries that would be required to use the same along with tests ● Implementing the UI screen for check-in with tests
Week 7-9 (July 10 - July 30)	● Work on implementing the basic designs for name tags and configuring the **pdfme** package into the repository ● Add tests to ensure that the name tags are generated correctly ● Work and research on devising efficient database queries that can be used to generate the various statistics as described in the Event Dashboard Management section of project
Week 10-12 (July 31 - August 20)	● Implement the database queries with tests and expose the same to the front end with simple queries ● Work on creating individual components for each of the statistics ● Add unit tests for each component ● Create the event dashboard screen where the different components are conditionally rendered based on the completion status of the event and its properties.
Submission for Final Evaluation of the standard coding period	
Week 13-17 (September 4 - October 4)	● Implement the complete feedback model ● Implement the mutations and queries related to the same in the API with tests ● Add database queries to create statistics from the feedback

	● Implement a component to show feedback statistics for the Admin panel and add the same to the main dashboard ● Write tests for the UI changes
Week 18-20 (October 5 - October 26)	● Configure the Firebase Admin SDK into the repository ● Implement the Notifications table and bad word filters into the Talawa API ● Implement the communication script and send the notifications to mobile devices through FCM ● Include nodemailer in the communication script ● Update the documentation about the same
Week 21-22 (October 27 - November 06)	● Add documentation about the Artillery framework and stress testing principles into Talawa Docs ● Configure a separate testing database for stress testing ● Implement the proposed stress tests
Project Completion and Final Submission	

Why Me?

I am an avid open-source contributor who is passionate about working with the Palisadoes Foundation. I am also a tech enthusiast who loves working with GraphQL, typescript, and React!

I am a responsible individual and a great team player and believe in the utmost importance of active communication. As for past experiences, I lead the tech team in Technex, Asia’s oldest techno-management fest conducted in IIT BHU, and thus know about working in large groups to achieve collaborative success. I am a gold medalist in the recent Inter IIT Tech Meet in the high preparation event, Grow Simplee.

Tech Stack:

Beginner	Intermediate	Advanced
YML, Django	GraphQL, Rust, React.js, Next.js, Bootstrap, MongoDB	HTML, CSS, JS, Typescript, Python, Vue.js, Nuxt, Node, SQL, Python

Community Engagement:
I have been an active member of the Palisadoes Foundation for above 2 months now, and have developed a deep understanding of all the philosophies and community values found to be of principal importance here.

I am also an active member of the community, as seen during the time of the introduction of GraphQL scalars into the API (which was a breaking change and led to many users facing errors with the Talawa-Admin project). I, along with other contributors, helped many fellow contributors resolve their issues and was in continuous contact with @literalEval to make all the relevant changes to the Talawa mobile project as well as to restore functionality.

I have contributed to the required projects on multiple instances and worked on various classes of issues, from improving developer experience (with the introduction of Husky) to implementing industry best practices (separation of testing database), from breaking changes (GraphQL scalars) to major feature introduction (introduction of tags for users).

Other Commitments:
I have no other commitments for the next 22 weeks of the GSoC application timeline. I my have institute exams after the semester completion, but the same are over in under a week. I am well versed in academics and can ensure that the same would pose no barrier to my dedication and my work schedule.