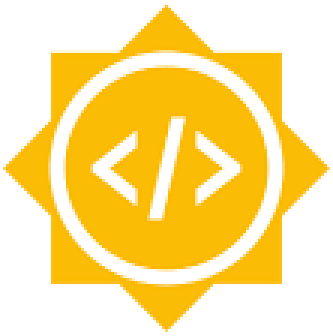


GSoC Proposal for

Global Alliance for Genomics and Health on

React preclinic and research data collection

SHUBHRAJYOTI DEY

INDIAN INSTITUTE OF TECHNOLOGY (BHU), VARANASI

Github : Shubhrajyoti-Dey-FrosTiK

Email : toshubhrajyotidey@gmail.com

Phone No : 7003026879 | LinkedIn

Timezone : (GMT + 05:30) Asia / Kolkata - IST

Table of Contents :

1. Personal Introduction

I am Shubhrajyoti Dey, a 2nd year undergrad at **Indian Institute of Technology (BHU), Varanasi**. I have been pursuing my web development for the past **1.5 years** and have completed **3** relevant internships regarding this field. I have worked on MERN stack applications and have made several projects in this field which can be found in my GitHub.

1.1 Work Experience

These are my responsibilities in my internship period.

1. **MentorPlus [Frontend Developer] (link) :** Developed the whole frontend of their web app using NextJS (SSR framework based on ReactJS) and improved the SSR site loading time by 32%. Implemented code splitting and React Lazy for dynamic import which decreased the overall .js chunk file size (on first load) by 31%. I also implemented PubNub and AgoraSDK and developed a real time interaction platform for the company.. Technologies used include : NextJS, PubNub, AgoraSDK.
2. **Akshary India Private Ltd [Full-Stack Developer] (link)** : Developed an inventory management service with the collaboration of the Government of India. This used a MERN stack and the backend was fully built on Typescript. Technologies used include : ReactJS , MongoDB, NodeJS, ExpressJS, Typescript.
3. **NiYO Solutions [Software Developer]** : Developed a payment auto-debit microservice using NestJS and Apache Kafka which is currently under sanction under Reserve Bank of India. Also developed an SMS collector service and deployed it using Kubernetes. Technologies used include : NestJS, Apache Kafka, Typescript, MongoDB.

1.2 Previous Projects

These are some of my previous projects :

1. **College Community (link) (Frontend) (Backend) :** A full fledged social media site using ReactJS as frontend and NodeJS and ExpressJS as the backend. MongoDB is used as a Database and Firebase Storage used as a file storage. All user sensitive credentials were stored using BCrypt encryption. Technologies Used: MongoDB, ReactJS, NodeJS, ExpressJS, Firebase, Typescript
2. **Entrepreneurship Cell IIT BHU website backend (Github) :** Developed the backend of the E-Cell IIT BHU official revamped website. Migrated the backend server from ExpressJS to NextJS using Next-Connect and built the REST APIs to implement authentication, login, CRUD, etc Technologies Used: NextJS, MongoDB, JWT, Multer, Next-Connect, JS.
3. **Sahayak bot (Github) :** This was a time when Covid was having a 2nd wave and the vaccines were just introduced to the common people. Thus no one was able to

book a vaccine dose slot as every slot was getting booked within a few minutes. So I implemented a discord bot using python which notifies subscribed users at a set interval of time about the availability of nearby vacant vaccine centers. Data was web scraped from the official government Co-Win website. Technologies Used: Python, Selenium

4. **Google Classroom Alternative (Deployed Link) (Github) :** Made a Google Classroom Clone Frontend in Flipr Hackathon. Implemented Google Login and authentication system using JWT and Auth0 in the project. Integrated Django REST as a backend for server-side operations. Technologies Used: ReactJS, Javascript, Axios, JWT.

1.3 Why do I wish to take part in GSoC 22 ?

I have been pursuing my web development journey from my 11th standard and have been very enthusiastic to know about new emerging technologies. In my second year of college I started to work in startups and service based companies in order to gain industrial exposure on how to scale code and make it efficient with such a big user base. It also taught me how to write clean readable code and collaborate with a team while contributing to a large codebase. From the start of this journey I had a special place for open source software as all the things I learnt are directly or indirectly related to open source. Be it npm or be it Mozilla browser, everything is open source. Now I want to come back to the roots and apply my knowledge to where I started from. For this open source journey I started by Hacktoberfest 21 and successfully completed it. I am now applying for GSoC 22 in order to know more about the open source community and to apply my knowledge to contribute something back to the open source community. By this opportunity I also want to enhance my current knowledge and learn about more efficient workflow strategies.

1.4 Why Global Alliance for Genomics and Health ?

I am an engineering student but I always had a fascination with biological studies. So I had been following GAGH for a while when I started looking at open source projects. So I always had a fascination to contribute to something which would be impactful to society. By this project I view it as a chance to contribute to something which will be the new normal in genomics data collection in the near future.Moreover this project also has the same tech-stacks as the requirements which I have my expertise. These are the reasons I am more inclined to contribute to the Global Alliance for Genomics and Health. I also expect to learn a lot of new technologies and genomics related studies during my GSoC time period.

2. Problem Statement Breakdown

There is a lot of genomics data which can be collected from patients which can be used for better diagnosis of a patient. Moreover the need for high quality, unambiguous, computable pedigree and family information is critical for scaling genomic analysis to larger, complex families. But the problem becomes on how to make the data portable as well as universal. The solution to this problem is a standardization of the data interface. This approach is deviced by GA4GH and genomics data standards are prepared.

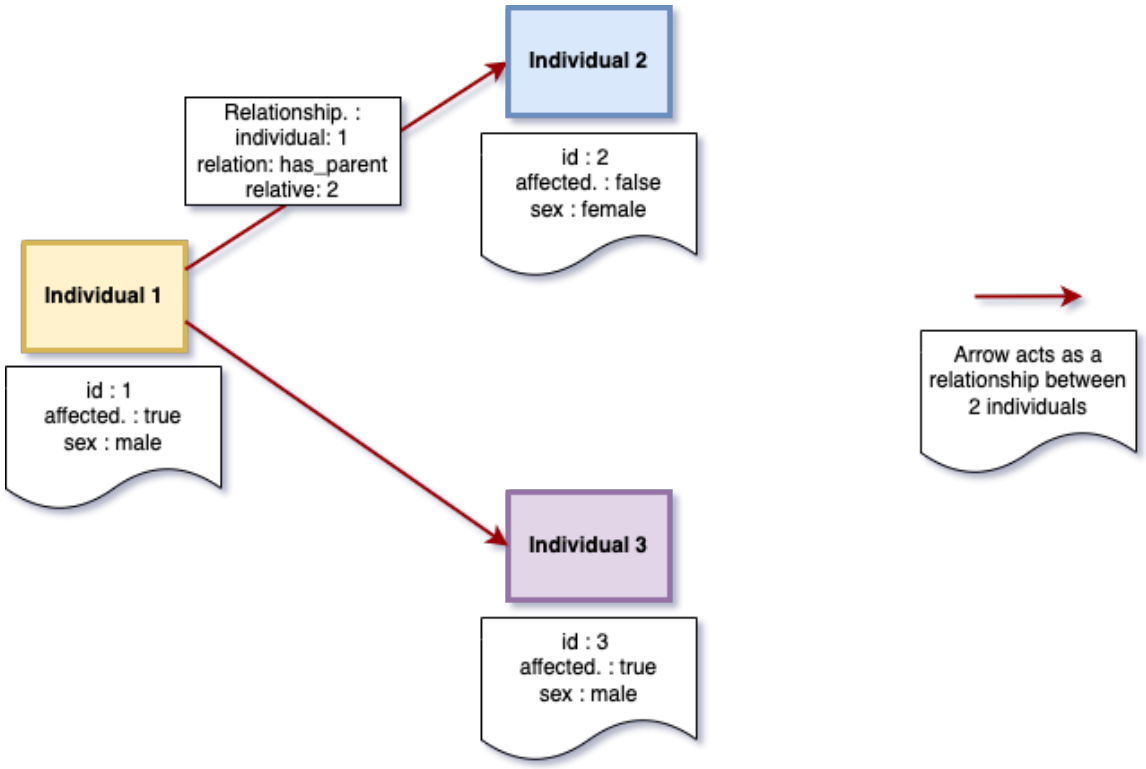
2.1 Types of Pedigree data representation techniques

Currently the pedigree data is majorly represented in one of the 2 ways :

1. **PED :** The PED format is a simple text file with 6 columns - IDs, a binary sex field, the phenotype (singular) and SNP genotypes. You can represent a basic parent-child trio, and that may cover a lot of use cases. However, you can't represent twins, things like adoption or donors, pregnancy, vital status, multiple phenotypes and data provenance. All of this type of data is important for genetic counseling and risk assessments where richer representations of relationships are valuable.
2. **The GA4GH Pedigree Model :** The GA4GH Pedigree Standard will natively incorporate PED to enable interoperability with legacy tools. It will be a superset of 6-column PED format Moreover optional genderless relationship vocabulary distinguishes biological and social relationships Graph structure allows specifying arbitrary relationships Easy-to-use within the context of other standards such as FHIR or Phenopackets Simplifies converting a genetic family history from one proband to another.

For this project we will use the 2nd method (The GA4GH Pedigree Model) due to reasons stated above. This will facilitate us to store the data of 2 or more individuals from a relational perspective in an efficient way. We will store the data of genetic family history such that it is easily accessible and is efficient enough to do computations on it if required. The project will aim at creating user customisable forms whose data regarding the family history will be internally stored in the GA4GH standards to maintain the uniformity of the data as well as to make it more portable to adapt with other external tools to perform any further computations. A brief on the GA4GH model is discussed below.

2.1 Brief on GA4GH Pedigree Model



The GA4GH Pedigree Standard is a graph-based model, a directed graph where nodes are the individuals and the arrows or edges represent relationships between pairs of those individuals. This is distinct from node information that corresponds to each individual separately. Separating these two things is important for extensibility and building out to more complex families, and for including more data elements than if you had something more individual-centric like PED. This is a very high level implementation diagram of the problem statement. It will be discussed in detail in the approach for the standardization section of the proposal.

3. Approach for Standardization

Our project aims at creating user customizable forms such that only the family history data needs to be standardized as all the other data will be customized by the user and cannot have a standard nature.

So the project will create a user customisable form. Let's divide the form into 2 parts :

1. **Family history section :** This part of the form will follow the GA4GH standards and will be stored in a customized JSON following a specific standardized schema.
2. **User customizable section :** This part of the form will be user customized and will be stored in raw non-standardized JSON.

3.1 Family History Section

This part of the form will be stored in a customized JSON format which will follow a specific schema following the GA4GH standards.
Defining some data types / classes : (These classes are taken from this reference)

1. **Concept** : This is a reference to a concept in an ontology/terminology/valueset.

```
class Identifier {
  id: String;
  label: String;
}
```

id : a CURIE associated with the concept (e.g., HP:0000118)
label : a human-readable label for the concept (e.g., "Phenotypic abnormality")

2. **Identifier :** This is an identifier for the individual in another system.

```
class Concept {
  id: String;
  label: String;
}
```

id : an external identifier such as a medical record number, participant id, or insurance number; URI/CURIE is preferred
label : a human-readable label for the identifier

3. **ID:** This is a string identifier for internal cross-referencing of individuals between the components of a Pedigree.

4. **Pedigree :** A clinical Pedigree is a curated selection of information about a family, including the individuals, relationships between them, and relevant health conditions.

```
class Pedigree {
  identifiers?: Array<Identifier>;
  proband?: ID;
  date?: Date;
  reason?: Concept;
}
```

identifiers: external identifiers for the family
proband: id of Individual that is the index case for the family, usually the first person referred to genetics or tested for the condition being investigated
date: the date the pedigree was collected or last updated, as ISO full or partial date, i.e. YYYY, YYYY-MM, or YYYY-MM-DD

reason: the reason for pedigree collection, especially a health condition of focus being investigated in the family; if any Individual has the affected property defined, it refers to this condition

5. **Individual:** This class is used to depict the info of an individual

   id: logical id
   sex: sex assigned at birth
   lifeStatus: presumed/accepted life status of an individual as of the pedigree collection date.
   name: name of the individual
   identifiers: external identifiers for individual
   gender: presumed or reported gender identity
   affected: whether or not the individual is affected by the condition being investigated in this pedigree

Below is an example of the above mentioned class. The sex, gender, lifeStatus can have more values in the enum. The `id` field in the enums of these fields can also change after finalizing the enum values.

```
const maleSex: Concept = { id: "Sex_1", label: "MALE" };
const femaleSex: Concept = { id: "Sex_2", label: "FEMALE" };
const otherSex: Concept = { id: "Sex_3", label: "MALE" };
const unknownSex: Concept = { id: "Sex_4", label: "UNKNOWN" };
enum Sex { maleSex, femaleSex, otherSex, unknownSex }

const alive: Concept = { id: "lifeStatus_1", label: "ALIVE" };
const deceased: Concept = { id: "lifeStatus_2", label: "DECEASED" };
const unborn: Concept = { id: "lifeStatus_3", label: "UNBORN" };
enum LifeStatus { alive, deceased, unborn }

const maleGender: Concept = { id: "Gender_1", label: "MALE" };
const femalesGender: Concept = { id: "Gender_2", label: "FEMALE" };
const nonBinaryGender: Concept = { id: "Gender_3", label: "NON-BINARY" };
const nonDisclosedGender: Concept = { id: "Gender_4", label: "NON-DISCLOSED" };
enum Gender { maleGender, femalesGender, nonBinaryGender, nonDisclosedGender }

class Individual {
  id: ID;
  sex: Sex;
  lifeStatus?: LifeStatus; // recommended
  name?: String;
  identifiers?: Array<Identifier>;
  gender?: Gender;
  affected?: Boolean;
}
```

6. **Relationship:** As described in the GA4GH Pedigree model there has to be a relation between two Individuals. This will be the structure.

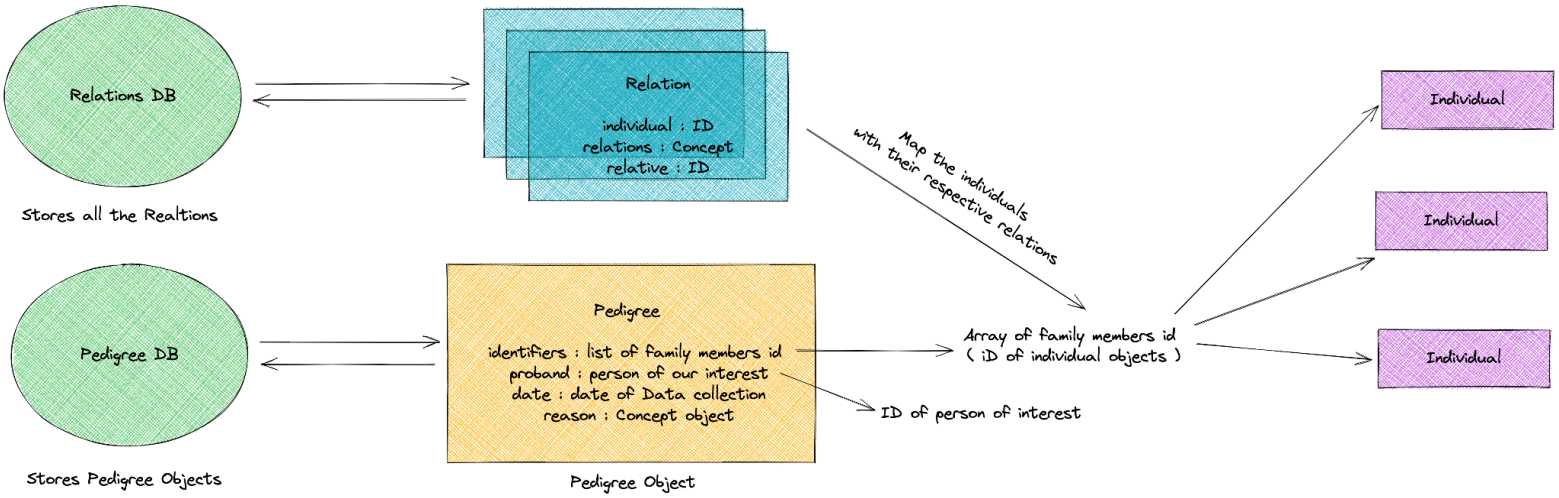
```
class Relationship {
  individual: ID;
  relation: Concept;
  relative: ID;
}
```

individual: identifier of the subject Individual; equivalent to the Biolink "Subject" and similar to the FHIR "Player"
relation: the relationship the individual has to the relative (e.g., if the individual is the relative's biological mother, then relation could be isBiologicalMother as per KIN terminology
relative: identifier of the relative Individual; equivalent to the Biolink "Object" and similar to the FHIR "Scoping Individual"

So now the standardization becomes :

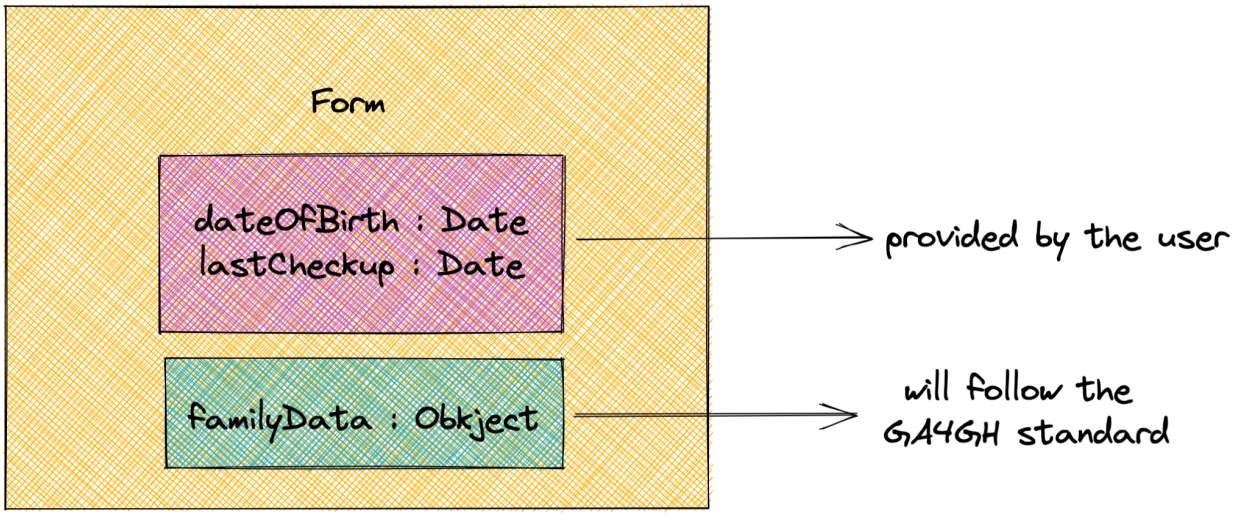
```
class Concept {
  id: String;
  label: String;
}

class Identifier {
  id: String;
  label: String;
}

class ID {
  id: String;
}

class Pedigree {
  identifiers?: Array<Identifier>;
  proband?: ID;
  date?: Date;
  reason?: Concept;
}

const maleSex: Concept = { id: "Sex_1", label: "MALE" };
const femaleSex: Concept = { id: "Sex_2", label: "FEMALE" };
const otherSex: Concept = { id: "Sex_3", label: "MALE" };
const unknownSex: Concept = { id: "Sex_4", label: "UNKNOWN" };
enum Sex { maleSex, femaleSex, otherSex, unknownSex }

const alive: Concept = { id: "lifeStatus_1", label: "ALIVE" };
const deceased: Concept = { id: "lifeStatus_2", label: "DECEASED" };
const unborn: Concept = { id: "lifeStatus_3", label: "UNBORN" };
enum LifeStatus { alive, deceased, unborn }

const maleGender: Concept = { id: "Gender_1", label: "MALE" };
const femalesGender: Concept = { id: "Gender_2", label: "FEMALE" };
const nonBinaryGender: Concept = { id: "Gender_3", label: "NON-BINARY" };
const nonDisclosedGender: Concept = { id: "Gender_4", label: "NON-DISCLOSED" };
enum Gender { maleGender, femalesGender, nonBinaryGender, nonDisclosedGender }

class Individual {
  id: ID;
  sex: Sex;
  lifeStatus?: LifeStatus; // recommended
  name?: String;
  identifiers?: Array<Identifier>;
  gender?: Gender;
  affected?: Boolean;
}

class Relationship {
  individual: ID;
  relation: Concept;
  relative: ID;
}
```

Moving further with the standardization; we know that pedigree is a family (diagrams) that represents family members and the relation between them and proband is the family member of interest. So we can represent the whole structure like this :



3.2 User Customizable Section

This section will be customized by the user and thus no standardization needs to be done. A basic JSON will be created on the basis of the user selection.
This is an example :



4. Technical Implementation

This section will focus on what should be the architecture of our webapp.

4.1 Tech-Stack and Libraries to be used

As we know that this is a full stack project we will need both a frontend and a backend to support all the operations. For this reason the below mentioned tech-stack is chosen :

Frontend :

1. **ReactJS :**
   ReactJS is a client-side rendered frontend framework which works on a component based architecture. This allows the code to stay clean even when the codebase becomes large. Moreover this also facilitates efficient reusability of code components which helps in maintaining the uniformity in the components on our webpage. ReactJS also has a very vast support of libraries which make it easier to create components without implementing it from scratch.
2. **Material UI :**
   Material UI is a frontend library which provides pre-built React components. This will make it easier for us to make the forms as it natively provides form components for textField, inputs etc.
3. **Axios :**
   Axios is a frontend library used to make API calls from the frontend to the backend server. It provides a vast variety of customizability and provides a clean structure for the API call for which it is chosen here.
4. **react-form-builder :**
   It is a complete react form builder that interfaces with a json endpoint to load and save generated forms.
5. **ContextAPI :**
   ContextAPI will be used to manage the state of the form
6. **Tailwind CSS :**
   Tailwind CSS is a frontend CSS library which makes it easier to style components with minimal classNames,

Unit Testing for Backend :

1. **ViTest**
   ViTest is a unit testing library based on Jest which simplifies the frontend unit testing.

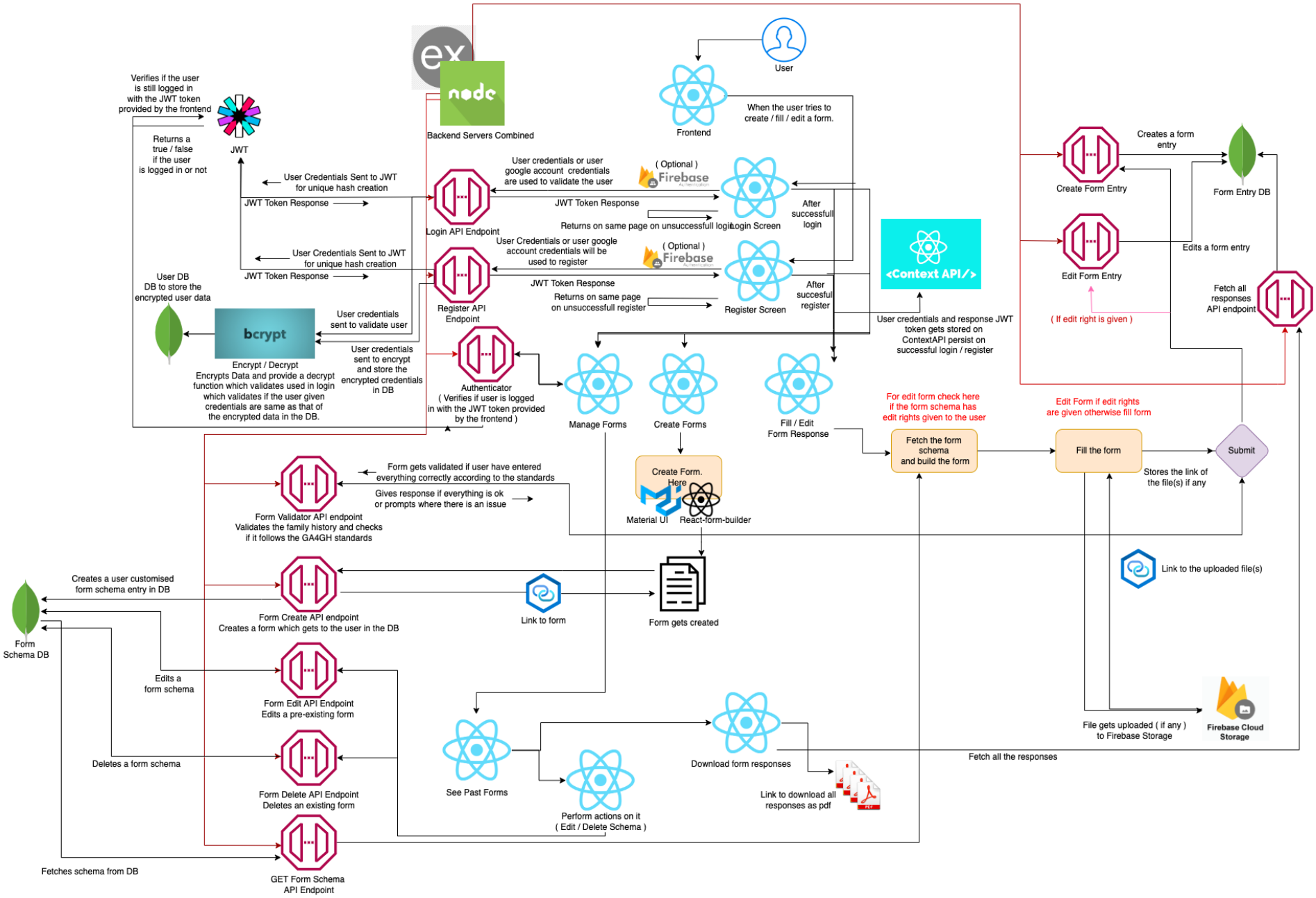
Backend :

1. **NodeJS :**
   NodeJS is a backend JS framework which can be used to create scalable backend servers to host API endpoints.
2. **Typescript :**
   Typescript is a typed language which makes it easier to make a reliable error-free server. Moreover being a typed language it will be easier to validate the incoming data to the GA4GH standards.
3. **ExpressJS :**
   ExpressJS will be used to create our server on top of NodeJS
4. **MongoDB :**
   As we are working with JSON data and as we are working with family histories which have lots of connections between each entry, MongoDB being a relational database storing BSON files is the best choice of our database. Moreover MongoDB also has aggregate functions which can perform pre computations on query request calls which make it faster than other databases.
5. **Firebase Storage :**
   As we are making user customizable forms, there will also be one option for file upload. In MongoDB we need to store files in a buffer or use GridFS for bigger files but both have their own limitations. For this reason we are choosing Firebase Storage as our storage option which will only store the file uploads of a particular user. This will enable us to store and download any file type of any file size as Firebase storage acts as a cloud storage rather than a conventional database. The approach will be upload the file to Firebase storage and then just store the link of the uploaded file in our MongoDB database
6. **Firebase Authentication :**
   This will be used to enable Google login in our website for greater ease in login for our user while providing the security of Google.
7. **JWT :**
   JWT will be used to authenticate users if they are logged in or not.
8. **BCryptJS :**
   BCryptJS will be used to encrypt the sensitive credentials of the user to maintain user privacy.

Unit Testing for Backend :

2. **Jest**
   Jest will be used for the unit testing of the backend.

4.2 Architecture Design



The above diagram represents the architecture design of our website. To take it a bit further and make it more scalable we will follow a **microservice architecture**.

4.3 Why microservice architecture ?

When we create a backend server one of the main aspects to consider is the scalability of the server and how much load it can take. Addressing this reason only we are opting for a microservice architecture. Here we basically break down our backend server into several servers which spread out the load among themselves.
These are the reasons of choosing a microservice architecture :

1. More Efficient: A microservice will only get triggered when a particular set of operations need to be performed unlike a conventional server which gets triggered when any API endpoint gets triggered. Moreover as microservices are designed to do a specific minor subset of the total work, the amount of computation is reduced for an individual microservice which makes it more scalable.
2. Efficient Error Handling: When the codebase becomes large, finding bugs becomes more and more tedious. Microservices solves this problem as a particular microservice addresses a very specific operation and thus error tracing becomes easier. Moreover as microservices are independent blocks of servers, malfunctioning of one server will not crash the whole server which is one issue in a conventional server.
3. Cost Efficient: As a microservice only gets triggered when a very specific service is needed the amount of irrelevant triggers is reduced which minimizes the cost of the server along with maintaining the scalability.

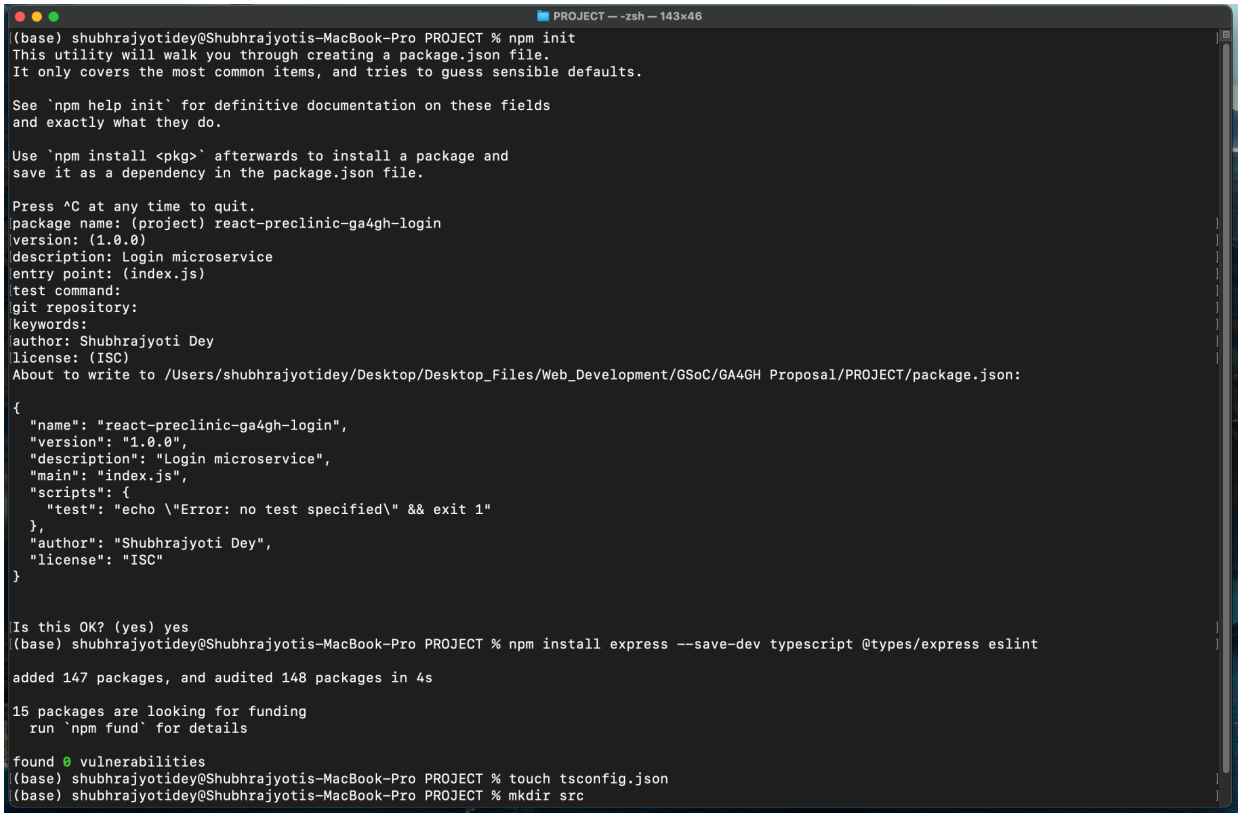
For our project these are the microservices we are going to implement:

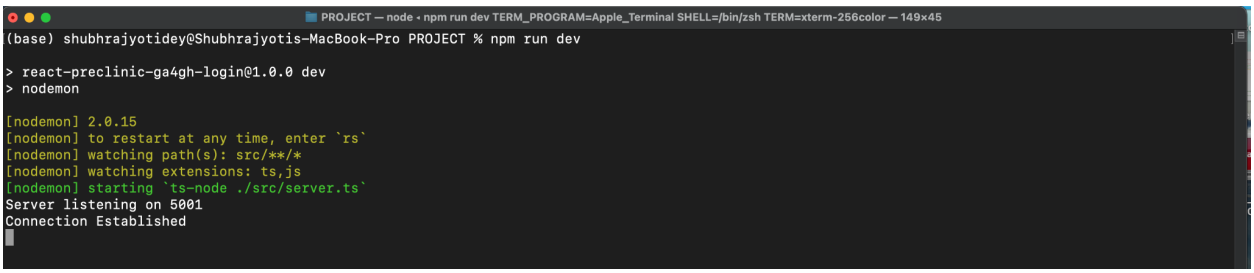
5. Project Setup

This section will focus on how we will get started with the project and the initial steps.

5.1 Setting up the backend

In the backend as discussed earlier there will be 4 backend servers (microservices) running at the same time. The backend will be written in Typescript entirely for its type strong nature and make our backend more predictable.
So these are the steps which will be followed :

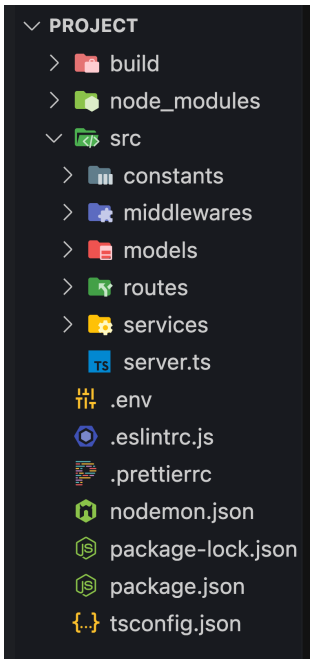




A sample server initialization code will be :

PROJECT - server.ts

```ts
import express from "express";
import cors from "cors";
import mongoose from "mongoose";
import dotenv from "dotenv";
import bodyParser from "body-parser";

/*-------- Initialization --------*/
const app = express();
dotenv.config();

app.use(express.urlencoded({ extended: true }));
app.use(bodyParser.urlencoded({ extended: true }));
app.use(express.json());
app.use(cors());

/**---------- Interceptor--------------- */

app.use((req: any, res: any, next: any) => {
  console.log("Request Received", req.originalUrl, req.method);
  next();
});


/**-------- DB Connect ------------------- */
mongoose
  .connect(
    process.env.MONGO_CONNECTION_URL! // connecting mongoose to mongoDb
  )
  .then(() => console.log("Connection Established"));

const PORT = process.env.PORT || 5001;

app.use("/", (request: any, response: any) => {
  // response.sendFile(path.join(__dirname,"client","build","index.html"))
  response.send({ message: "Specified Path is not Defined" });
});

app.listen(PORT, () => {
  console.log(`Server listening on ${PORT}`);
});
```



This can be the initial project structure :

constants : Consists of all the constants of our app.
middlewares : Consists of all the middlewares (eg : logged-in-validator)
models : Consists of the model schemas of MongoDB databases for schema validations etc. Also used in the standardization procedure
services : Consists of base functions which are used in various points over the codebase.
routes : Consists of all the API endpoints
server.js : Server is initialized here
.prettierrc : Used for auto-formatting our code

5.1.1 Models

This section will focus on the different types of models which will be used in all the above mentioned microservice

There will be 2 models in User microservice :

1. **User model :** Which stores the information of the user.
2. **Activity model :** It stores the activity of the user (eg user filled a form , user created a form etc)

There will be one model in the form schema microservice :

1. **FormSchema model :** This will store all the user customized form schema which will be provided by the frontend.

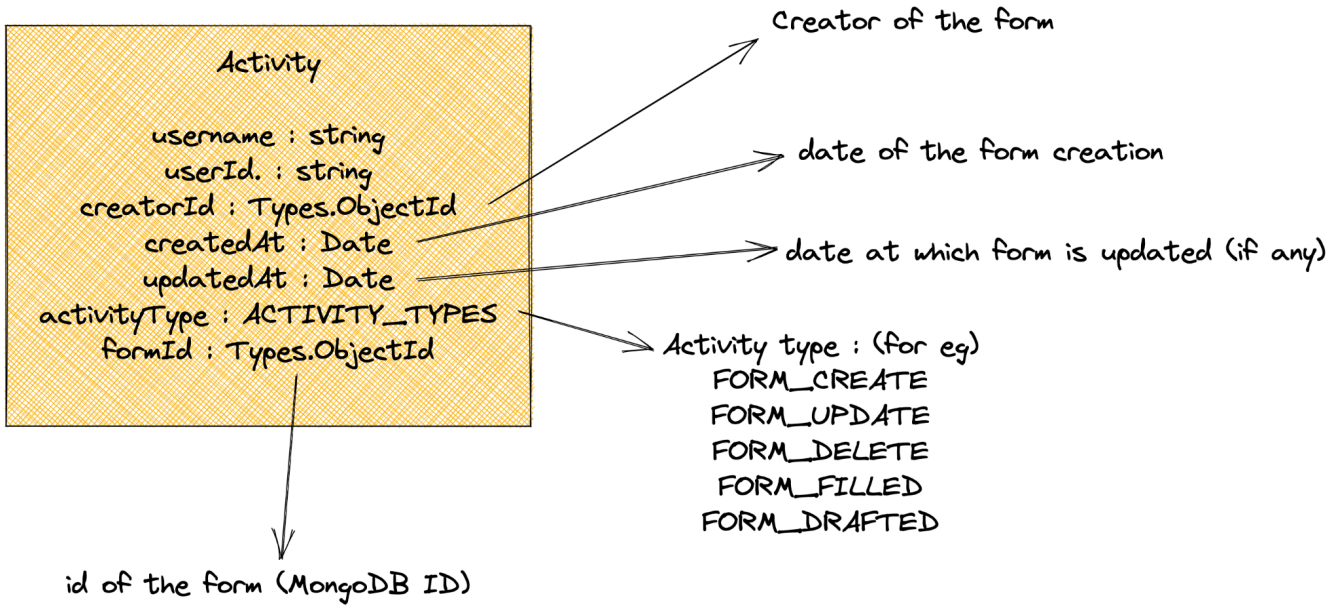
There will be one model in the form response microservice :

1. **FormResponse model :** This will store the response when the user fills the form.

There will be one model in the GA4GH standard validator microservice :

1. **GA4GH model :** Stores the latest GA4GH standard.

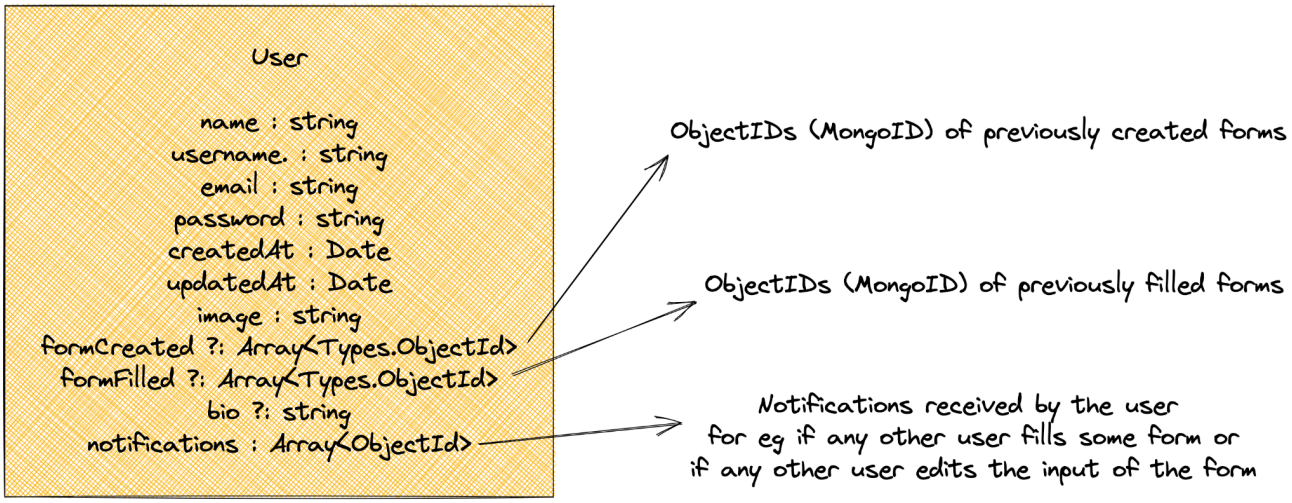
This is a activity model



PROJECT - Activity.model.ts

```
const schema = new Schema<Activity>(
  {
    username: { type: String, required: true },
    userId: { type: Schema.Types.ObjectId, ref: "User" },
    creatorId: { type: Schema.Types.ObjectId, ref: "User" },
    activityType: {
      type: String,
      required: true,
      enum: ACTIVITY_CONSTANTS_ARRAY,
    },
    createdAt: { type: Date, default: Date.now },
    updatedAt: { type: Date, default: Date.now },
    formId: { type: Schema.Types.ObjectId, ref: "FormSchema" },
  },
  {
    timestamps: true,
  }
);
```

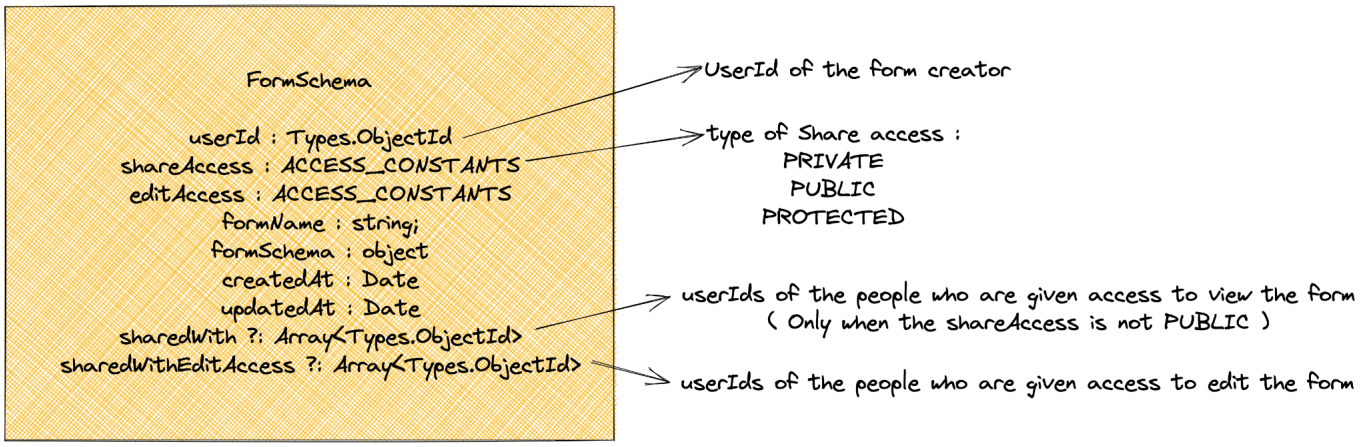
This is a user model :



PROJECT - User.model.ts

```
const schema = new Schema<User>(
  {
    name: { type: String, required: true },
    username: { type: String, required: true },
    email: { type: String, required: true },
    password: { type: String, required: true },
    createdAt: { type: Date, default: Date.now },
    updatedAt: { type: Date, default: Date.now },
    image: { type: String, required: false },
    formsCreated: [{ type: Schema.Types.ObjectId, ref: "User" }],
    formsFilled: { type: String, required: false },
    bio: { type: String, required: false },
    notifications: [{ type: Schema.Types.ObjectId, ref: "Activity" }],
  },
  {
    timestamps: true,
  }
);
```

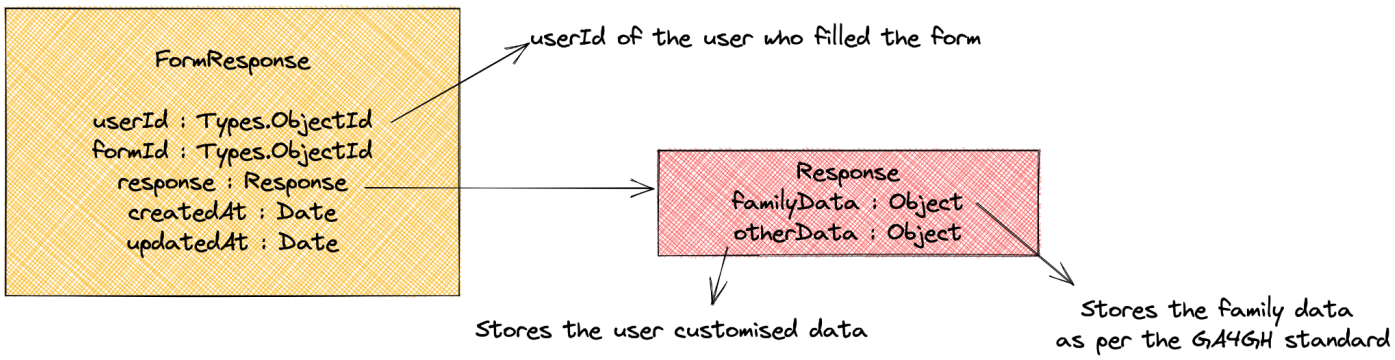
This is a form schema model :



PROJECT - FormSchema.model.ts

```
const schema = new Schema<FormSchema>(
  {
    userId: { type: Schema.Types.ObjectId, ref: "User" },
    shareAccess: {
      type: String,
      required: true,
      enum: ACCESS_CONSTANTS_ARRAY,
    },
    editAccess: {
      type: String,
      required: true,
      enum: ACCESS_CONSTANTS_ARRAY,
    },
    formName: { type: String, required: true },
    formSchema: { type: Object, required: true },
    createdAt: { type: Date, default: Date.now },
    updatedAt: { type: Date, default: Date.now },
    sharedWith: [{ type: Schema.Types.ObjectId, ref: "User" }],
    sharedWithEditAccess: [{ type: Schema.Types.ObjectId, ref: "User" }],
  },
  {
    timestamps: true,
  }
);
```

This is a form response model :



PROJECT - FormResponse.model.ts

```
const schema = new Schema<FormResponse>(
  {
    userId: { type: Schema.Types.ObjectId, ref: "User" },
    formId: { type: Schema.Types.ObjectId, ref: "FormSchema" },
    response: { type: Object, required: true },
    createdAt: { type: Date, default: Date.now },
    updatedAt: { type: Date, default: Date.now },
  },
  {
    timestamps: true,
  }
);
```

A brief over the GA4GH model is shared at the Approach to Standardization section and it will be modified after further discussion.

(All the above mentioned model parameters can be modified after discussion in the process of the development)

5.1.2 Middlewares

There will be some middlewares which will act as pre-validators before the request is passed on to the API Endpoint.
One such middleware is used in the User microservice to check if the user is logged in :

PROJECT - userLoggedIn.middleware.ts

```ts
import { NextFunction, Request, Response } from "express";

/*-------- Dto --------*/
import { AuthenticationService } from "../services/auth/authentication.service";

export class UserMiddleware {
  async loginStatus(
    request: Request,
    response: Response,
    next: NextFunction
  ): Promise<void> {
    const auth = new AuthenticationService();
    const token: string = request.headers.authorization || "";
    const authenticated = await auth.verifyToken(token);
    if (authenticated) {
      next();
    } else {
      response.send({
        error: "You are not logged in",
      });
    }
  }
}
```

This middleware will be used to verify if the user is logged in before every API endpoint is triggered except the login and the register API endpoint.

5.1.3 Services

These are the services which we will use.

1. **AuthenticationService :** This will manage all the auth in our backend. It communicates with JWT and generates / verifies the token for authentication purposes.

PROJECT - Authentication.service.ts

```
async generateToken(user: TokenGeneratorDto): Promise<string> {
   const token = jwt.sign({ username: user.username }, this.secret, {
     expiresIn: user.expiresIn,
   });
   return token;
 }

 async verifyToken(token: string): Promise<boolean> {
   try {
     jwt.verify(token, this.secret);
     return true;
   } catch {
     return false;
   }
 }
```

2. **EncryptionService** : This service will handle all the encryptions of the user data to keep user privacy in our app.

PROJECT - encryption.service.ts

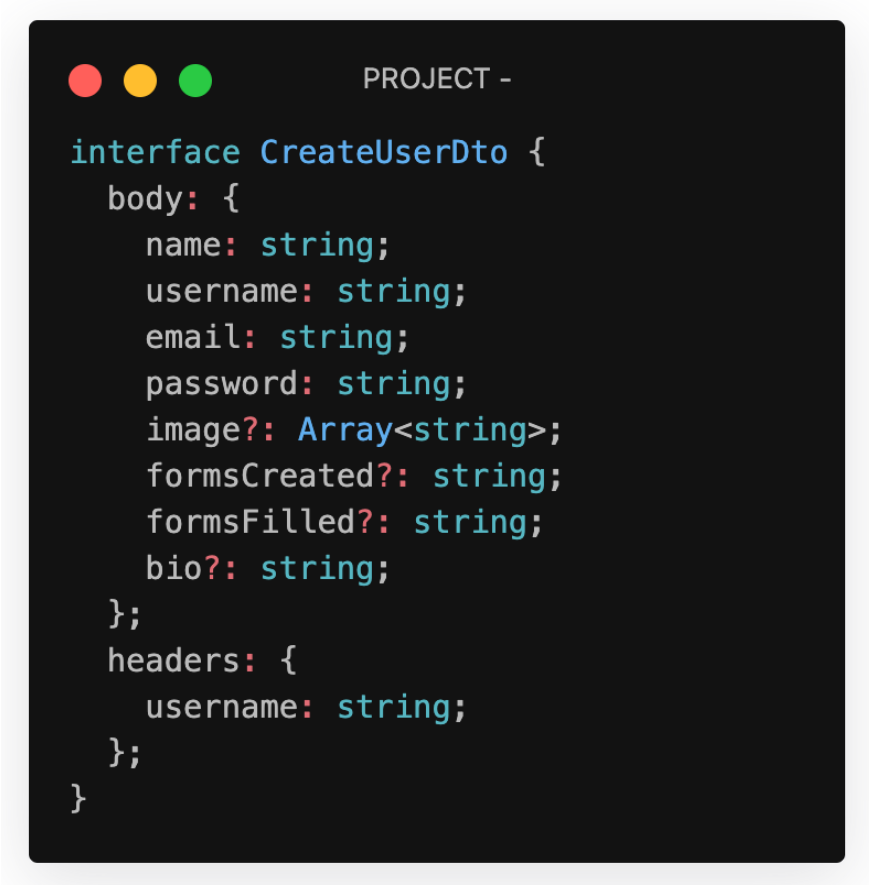
```
salt = Number(process.env.SALT_ROUNDS);

  async hash(password: string): Promise<string> {
    return bcrypt.hash(password, this.salt);
  }

  async compare(password: string, hashedPassword: string): Promise<boolean> {
    return bcrypt.compare(password, hashedPassword);
  }
```

This function `compare` will just return true if both the passwords (one provided by the user and the other is the hashed one) are equal and false otherwise.

5.1.4 Routes

There will be several API endpoints in the microservices which has been mentioned above in the Why Microservice Architecture ? section.
Each microservice will be designed to perform a specific function which is detailed before. There will be several endpoints in the microservices mentioned. One of them is mentioned below.

Create Account API Endpoint :
This will be responsible for creating an User entry in the DB when the user registers to our service.



First we will create a CreateAccountRequestDTO which will be a Typescript interface for the POST request

Next we will create a method that will perform the operations.

PROJECT - User.controller.ts

```ts
const createUser = async (request: CreateUserDto): Promise<ResponseDto> => {
  try {
    const email: Number = await db.count(UserModel, {
      email: request.body.email,
    });
    if (email) {
      return { message: "Email already exists" };
    }
    const username: Number = await db.count(UserModel, {
      username: request.body.username,
    });
    if (username) {
      return { message: "Username already exists" };
    }
    const userObject = {
      name: request.body.name,
      username: request.body.username,
      email: request.body.email,
      password: await es.hash(request.body.password),
      formsCreated: [],
      formsFilled: [],
      notifications: [],
      bio: request.body.bio || "",
    };
    const token = await auth.generateToken({
      username: request.body.username,
      expiresIn: request.body.expiresIn || "1d",
    });
    const user = await db.create(UserModel, userObject);
    return { message: "User Created", data: { user }, token };
  } catch (error: any) {
    return { error };
  }
};
```

Now we will call this class

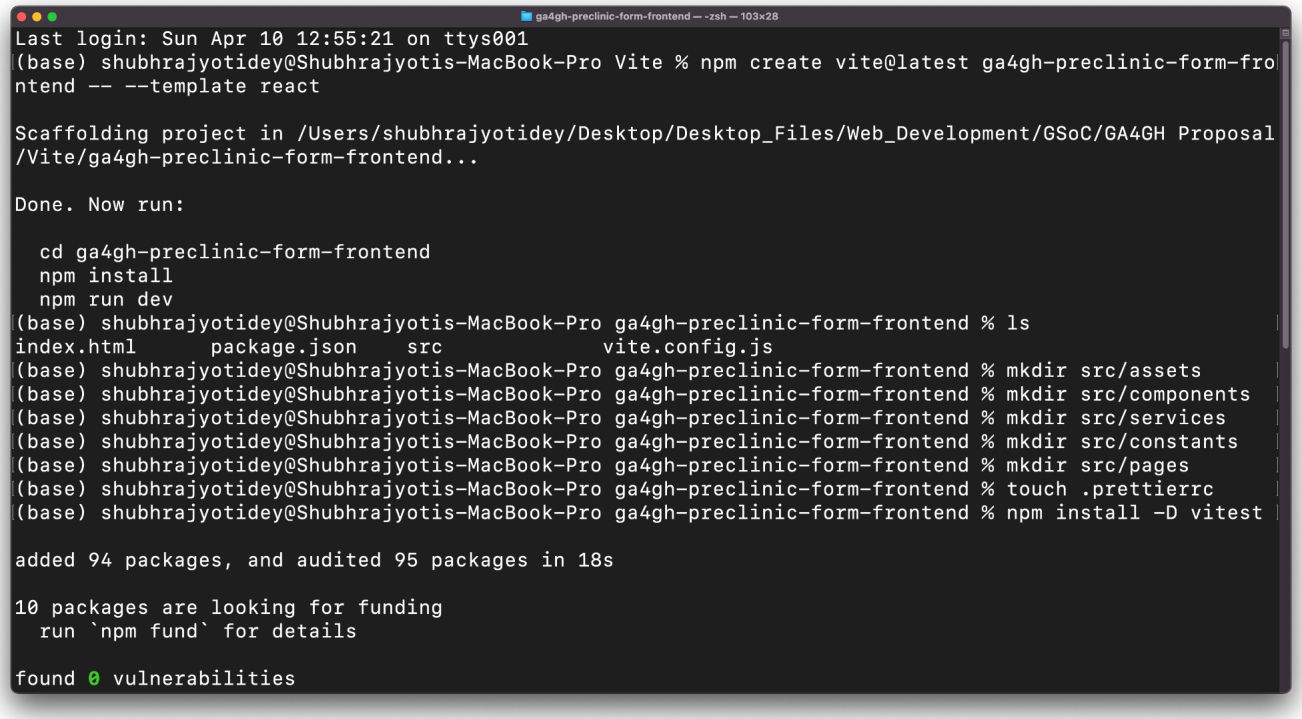
PROJECT - User.controller.ts

```ts
router.post("/", async (request: CreateUserDto, response: any) => {
  const newUser: ResponseDto = await createUser(request);
  response.send(newUser);
});
```

5.2 Setting up the frontend

This section will focus on setting up the frontend As mentioned earlier we will be using ReactJS for our frontend. For the setting up we will use Vite instead of normal React CRA due to 2 reasons :

1. Blazing fast speeds : Vite provides blazing fast speeds in creating / initializing / hot reloading servers which improves the overall efficiency in the development workflow.
2. Native Unit Testing : Vitest is a unit testing library which Vite natively supports. This will allow us to seamlessly implement unit testing in the frontend.

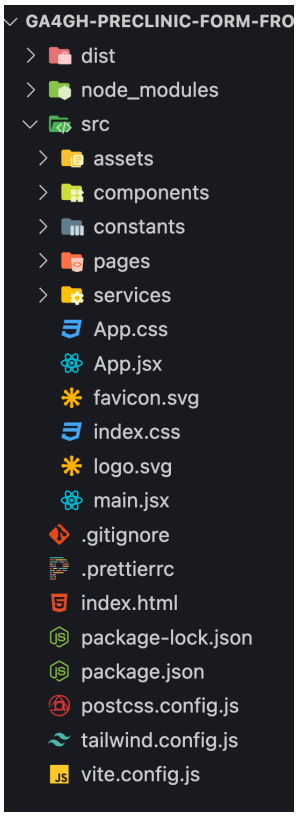


After this we will set up the tailwind css.

```
npm install -D tailwindcss
npx tailwindcss init
```

Now we have to configure the tailwind.config.js to track .jsx files.
To avail tailwind css in Vite we also need to make a postcss.config.js and configure tailwind css on it.

In the above process we create a basic boiler code for a starter React App we can follow.



This will be the directory structure :

assets : Consists of all the assets for our project like images, fonts etc
components : ReactJS is a component based framework. This directory will consist of all the components of our website.
constants : Consists of all the constants defined in our website.
pages : This will consist of all the pages .jsx files which will render the components as per the UI.
services : Consists of all the functions which will be used multiple times.
index.js : ContextAPI will be defined here.
App.js : Routing will be done here.

5.2.1 Services

As a frontend service one of the main functionalities will be handling a the file upload feature to our online cloud storage and returning the link to our DB (Which is MongoDB in this case)
We will first create StorageService which will manage the state of our app (Using React Context API).
The image upload service will access this storageService and will directly update the state of the app whenever one file gets uploaded to the cloud.

The FirebaseService() exposes one function which takes the file object as a parameter and returns the url of the path where the file is stored in the cloud

PROJECT - firebase.service.js

```
uploadToFirebaseStorage(file) {
    const baseFilePath = `/img/${user.username}/${date}-${file.name}`;
    const uploadTask = storage.ref(baseFilePath).put(file);
    uploadTask.on(
      "state_changed",
      (snapshot) => {
        const progress = Math.round(
          (snapshot.bytesTransferred / snapshot.totalBytes) * 100
        );
      },
      (error) => {
        console.log(error);
      },
      () => {
        storage
          .ref(baseFilePath)
          .getDownloadURL()
          .then((newUrl) => {
            urls.push(newUrl);
          });
      }
    );
    return urls;
  }
```

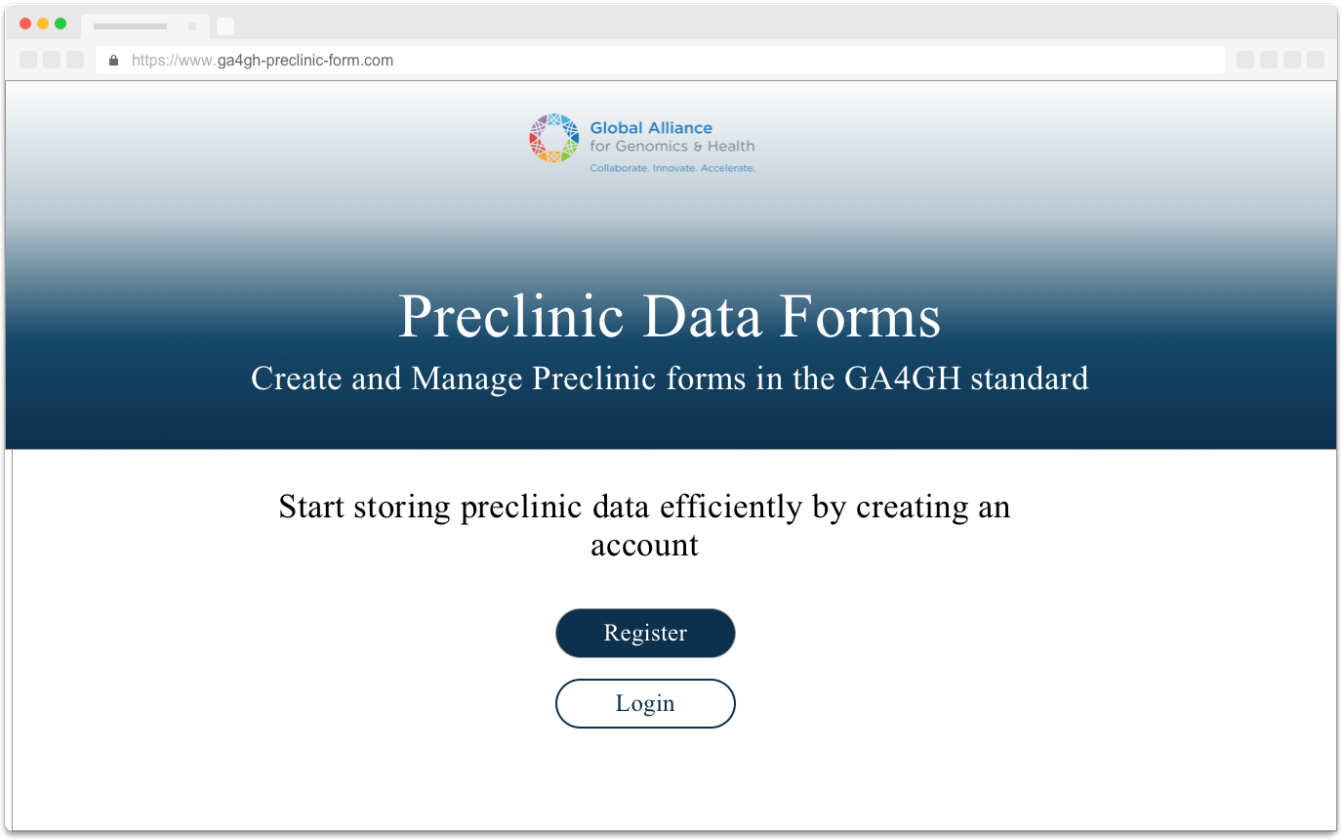
6. UI Mockups

This section will focus on UI mockups for our webapp.
In the section we have specified 4 UI screens :

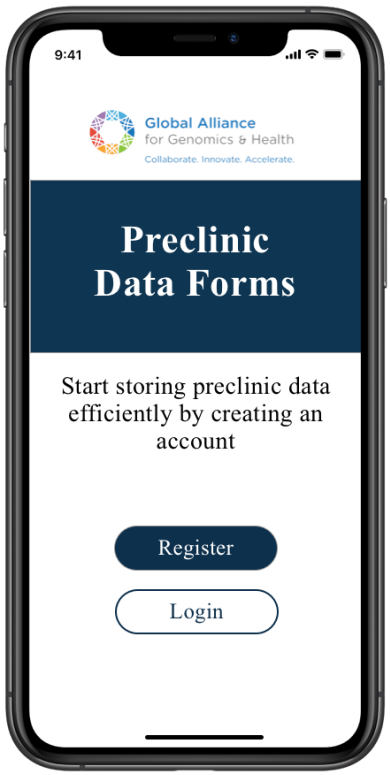
1. Homepage (Before Login) : This is the page which is shown to a user when he/she has not logged in. There are only 2 options for simplicity which are login and register respectively.
2. Login Screen : This is the login page where the user logins with the username/email and password. When the Google login is implemented a google login/register button will also be there in this page. Register page will share a similar UI as the login page and thus the UI is not shown.
3. Homepage (After Login) : This page is shown to the user when the user logs in to our website. The user is greeted with a create form option as well as the forms which the user has created / edited / filled. Here the user also has the option to go to the account page and change user details.
4. Form Create page : This page will be used to create the forms. User can edit the placement of the fields as well add custom validators to the fields.

6.1 Homepage (Before Login)

PC Version :

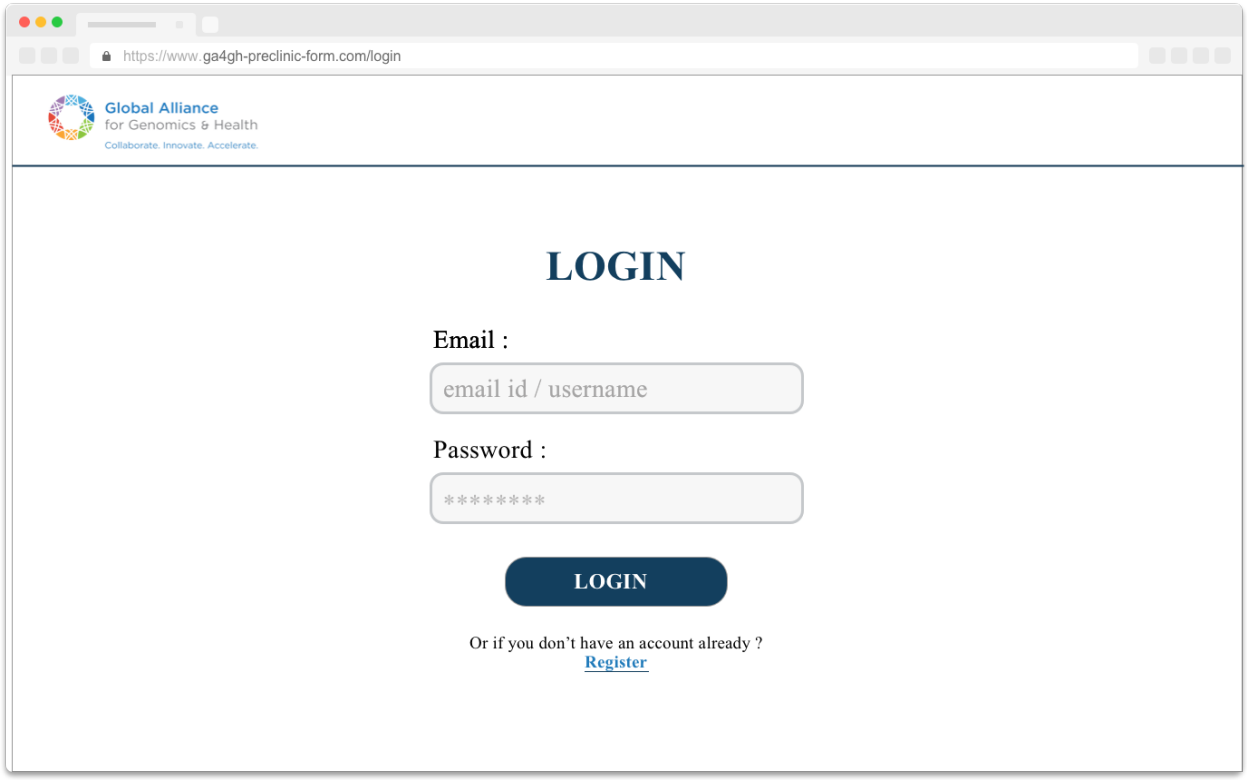


Mobile Version :

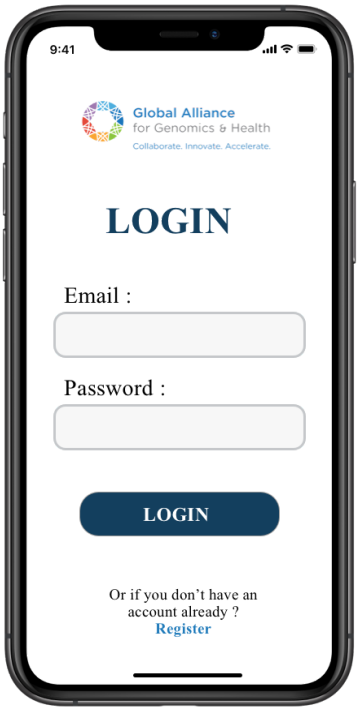


6.2 Login Screen

PC Version :

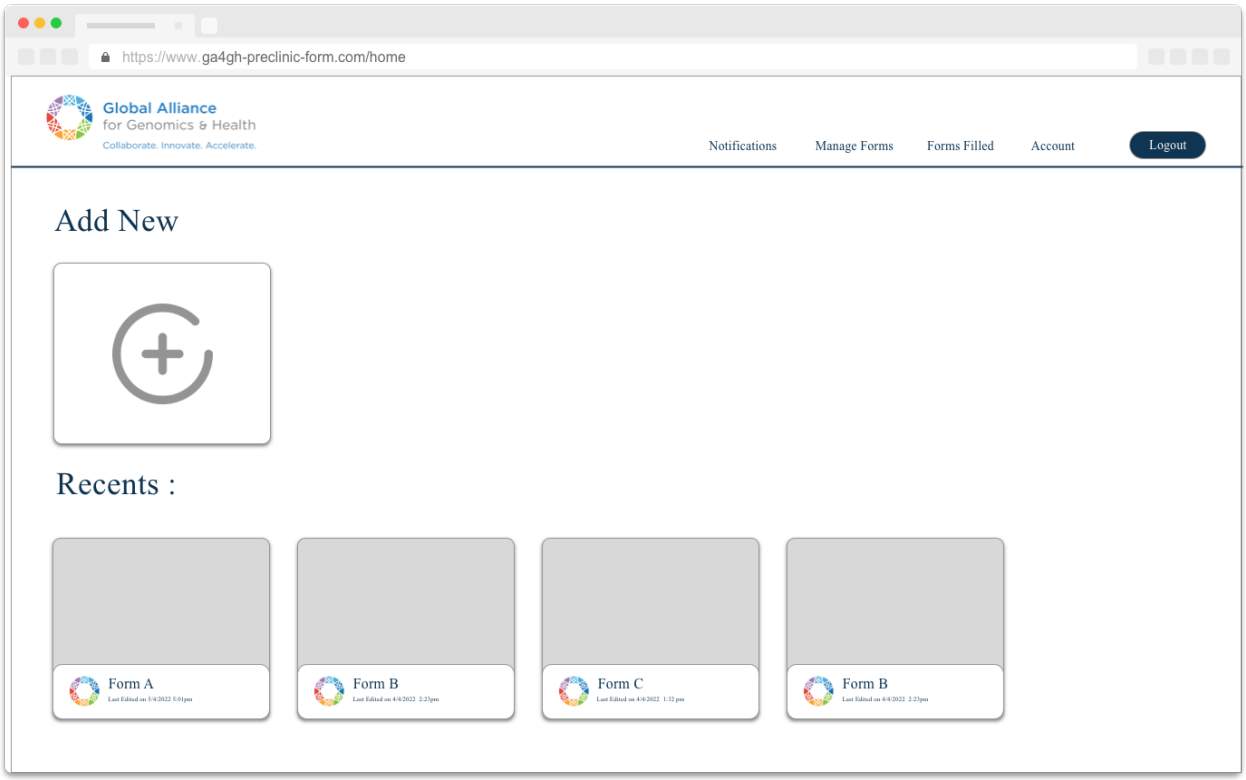


Mobile Version :

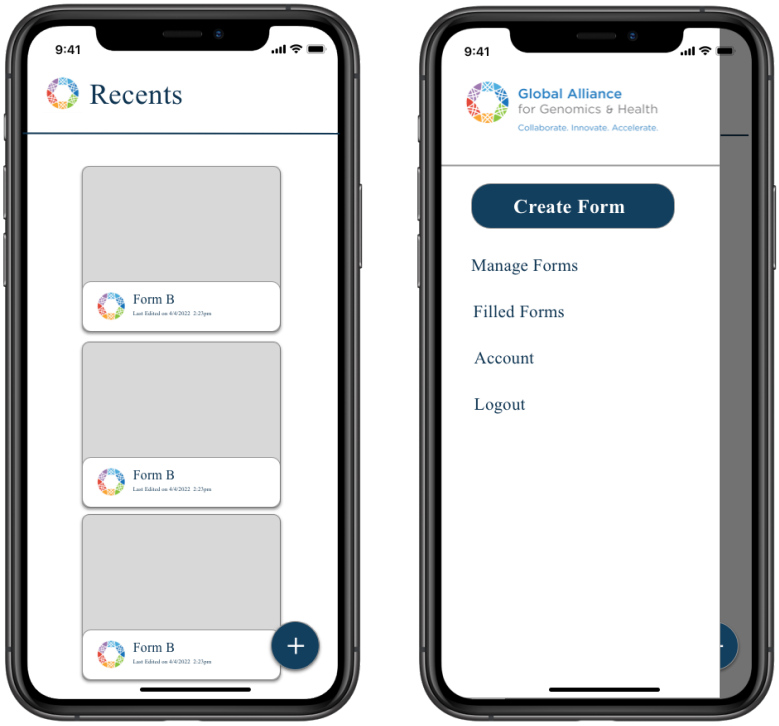


6.3 Homepage (After Login)

PC Version :

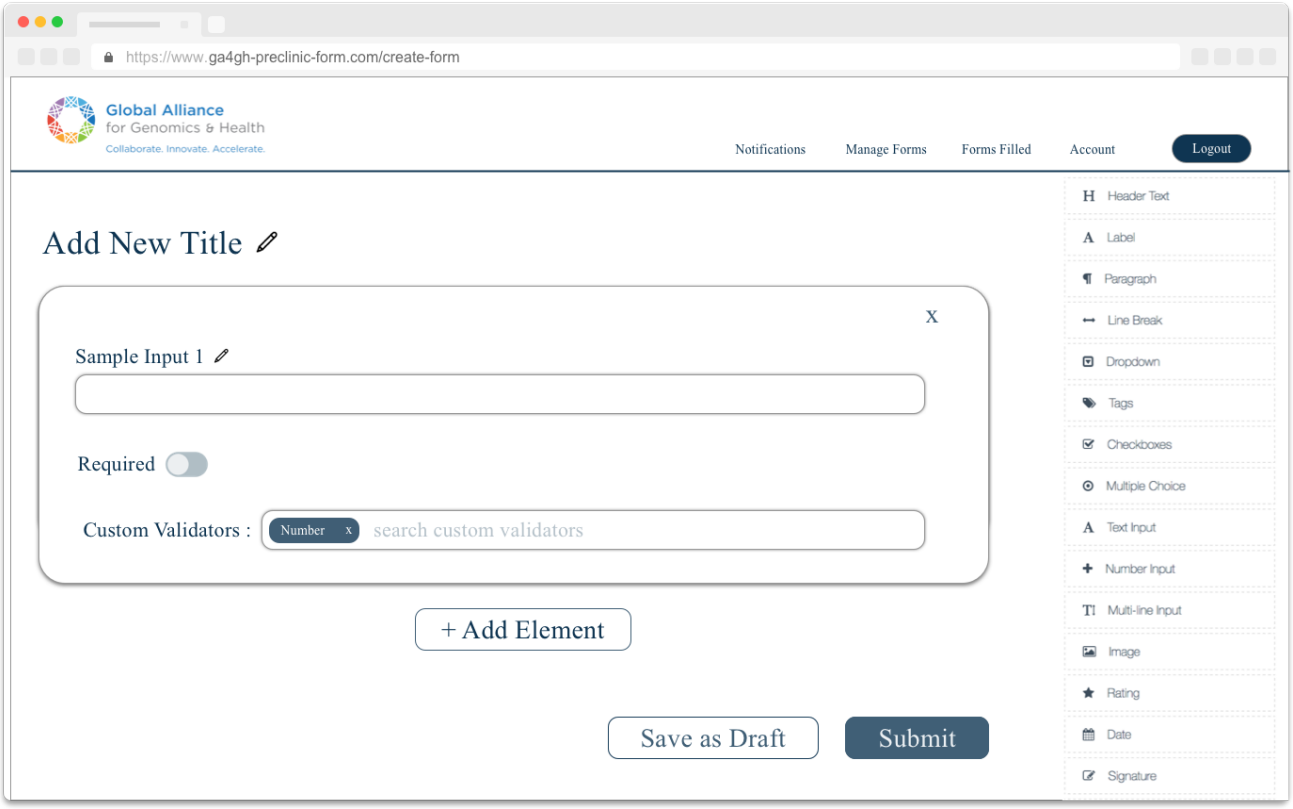


Mobile Version :



6.4 Create Forms

PC Version :



Mobile Version :

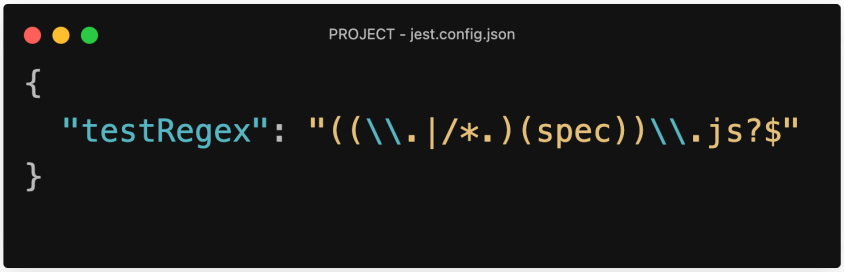
7. Unit Testing

Unit testing is one of the most important parts in the development lifecycle to ensure the consistency of our website. For this unit testing we will use Vitest. During the setup process we have already set up Vitest for our frontend.

For the backend :

```
npm install --save-dev jest
touch jest.config.json
```

Now we need to configure Jest.



8. CI / CD and Further Implementations

I will implement CI / CD using Github Actions which will enable us to automate deploy and test for bugs.
I will also provide a docker file for the project. This will enable :

1. Consistent Development Environments : This will make sure that future developers have the same development setup so that the development process is smooth and consistent.

9. Project Timeline

Community Bonding Phase	
May 20 - June 12,	1. Have an overall idea about the processes of standardization. 2. Work with my mentor to improve the proposed data structure for the GA4GH standardization. 3. Learn more about React JSON Schema Form Editor and React Form Builder v2 packages.

	4. Setup my workspace environment for the project.
Coding Period	
June 13 - June 27	1. Create a proper structure for the GA4GH validator. 2. Complete the coding after the finalization of the standard data structure(s). 3. Have the GA4GH validator up and running. 4. Complete unit testing for the microservice.
June 27 - July 7	1. Discuss the authentication workflow with the mentor and finalize the auth architecture. 2. Complete the coding part of the User microservice as proposed above. 3. Complete the unit testing of the user microservice.
July 8 - July 22	1. Discuss the from schema and the form response architecture and finalize it. 2. Complete the coding part of the form schema and the form response microservice. 3. Complete the unit testing of both the microservices.
July 23 - July 25	1. Discuss and work on feedback about any changes / improvements possible.
Mid Evaluations (At this point of time the backend will be completed)	
July 26 - August 1	1. Discuss and finalize the UI for homepage (before login), login and register page 2. Complete the coding part for the homepage (before login), login and register page 3. Complete unit testing for the homepage,login and register page
August 2 - August 20	1. Discuss and finalize the UI for the create form page and the homepage (after login) 2. Complete the coding part of the create form page. This will also include editing forms functionality. 3. Complete handling the user activity part (eg creating forms, editing forms, deleting forms, filing forms etc) which will act as a sort of a user history.

	4. Work on making the data exportable (in the form of .pdf or any other format) 5. Complete unit testing for all the above mentioned code.
August 21 - August 25	1. Discuss and finalize the UI for the accounts page. 2. Complete the coding part of the accounts page. 3. Complete unit testing for the accounts page
August 26 - September 2	1. Complete the documentation for the whole website.
September 3 - September 15	1. Work on any unimplemented feature or any improvement feedback.
Final Evaluation By this time a working full stack React preclinic application should be developed which exports family data as per the GA4GH standards	

10. Post GSoC

Post the GSoC period I would love to continue contributing to this project and implement more features which may not have been covered in GSoC in the short time interval. I will also try to provide consistent patches and bug fixes to the project. I will also try to contribute to the community by taking part in development activities other than the scope of this project also.

11. References

These are references I have taken to write my proposal :

1. https://www.project-redcap.org
2. https://github.com/GA4GH-Pedigree-Standard/pedigree
3. https://github.com/ginkgobioworks/react-json-schema-form-builder
4. https://github.com/kiho/react-form-builder#readme