Atlan - Backend Challenge

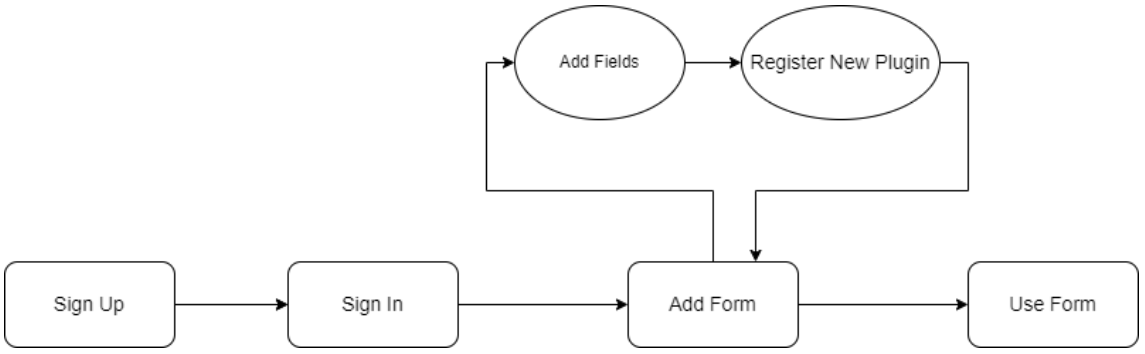
- Animesh Sinha, animeshksinha27@gmail.com

Problem Statement:

Design a data collection platform which supports offline data collection, validation, analysis, and multiple other post-submission business actions in a “plug-n-play” manner.
The addition of newer “actions” (plugins) over time should not affect the existing structure of the backend.

Design Schematic:

You are a user of this platform who needs to collect data using different forms that you can create once you sign up. You can create and manage multiple forms containing questions and add support for newer required features (plugins!) to process data collected through your clients’ responses.



This simple process can be followed to add support for a new plugin, without changing any pre-existing logic at the backend. *Thus leveraging our “plug-n-play” model.*
Note: Add your plugin to the list of PLUGINS in “./plugins/config.ts” for it to be recognized as valid.

Handling Plugins:

Introduction to Plugins:

I have provided the implementation for 3 different plugins: **Gsheets**, **IncomeValidation**, **EmailSender** (very similar implementation can also be followed for **SMSSender**.

We need to provide a layout for our plugins, so that it correctly identifies the fields in client responses.
Ques: How does the **IncomeValidation** plugin know which fields do monthly savings and income correspond to?
Ans: We specify this in the **PluginConfig** while creating class IncomeValidation.

```
export default class IncomeValidation extends FormPlugin {
    static pluginName: string = "Income Validation";
    static pluginDescription: string =
        "Plugin to validate form submissions, checking if monthly savings are more than monthly income.";

    static config: PluginConfig = {
        monthlyIncome: {
            name: "Monthly Income Field",
            description: "Field serving as the monthly income.",
            type: "field",
        },
        monthlySavings: {
            name: "Monthly Savings Field",
            description: "Field serving as the monthly savings.",
            type: "field",
        }
    };
```

All the plugins are classes which inherit from **FormPlugin**.

Sample Plugin:

```
export default class IncomeValidation extends FormPlugin {
    static pluginName: string = "Income Validation";
    static pluginDescription: string =
        "Plugin to validate form submissions, checking if monthly savings are more than monthly income.";

    static config: PluginConfig = {
        monthlyIncome: {
            name: "Monthly Income Field",
            description: "Field serving as the monthly income.",
            type: "field",
        },
        monthlySavings: {
            name: "Monthly Savings Field",
            description: "Field serving as the monthly savings.",
            type: "field",
        }
    };

    private readonly monthlySavings: string;
    private readonly monthlyIncome: string;

    constructor(definition: IIncomeValidationDefinition) {
        super(definition);
        if (!this.__form.formSchema[definition.monthlyIncome] || !this.__form.formSchema[definition.monthlySavings]) {
            throw new Error(
                "Fields specified for monthly savings and income are missing from the schema."
            );
        }
        this.monthlySavings = definition.monthlySavings;
        this.monthlyIncome = definition.monthlyIncome;
    }

    async onSave(formData: FormData): Promise<PluginReturn> {
```

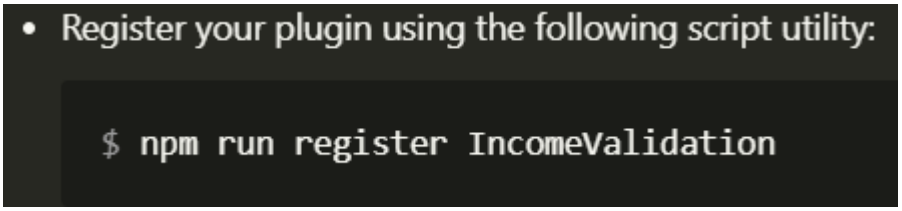
```
async onSave(formData: FormData): Promise<PluginReturn> {
    const monthlyIncome = formData["monthlyIncome"].value;
    const monthlySavings = formData["monthlySavings"].value;

    if (monthlyIncome && monthlySavings && monthlySavings > monthlyIncome) {
        // Flag the submission (you can customize this part based on your needs)
        const flagMessage = `Flagged submission: Monthly savings (${monthlySavings}) is more than monthly income (${monthlyIncome}).`;
        console.log(flagMessage);

        // Store the flagged submission in the logs
        await appendLogs("Income Validation", flagMessage);

        return {
            error: true,
            message: 'Validation failed. Monthly savings cannot be more than monthly income.',
        };
    }

    // No validation issues, continue with the save
    return {
        error: false,
        message: 'Validation passed. Form data saved successfully.',
    };
}
}
```

Registering new Plugins:

- Note (as mentioned earlier): Add your plugin to the list of **PLUGINS** in "./plugins/config.ts" for it to be recognized as valid.

```
// Register plugins here.
export const PLUGINS: {
    [key: string]: any;
} = {
    EmailSender: EmailSender,
    GSheet: GSheet,
    IncomeValidation: IncomeValidation,
};
```

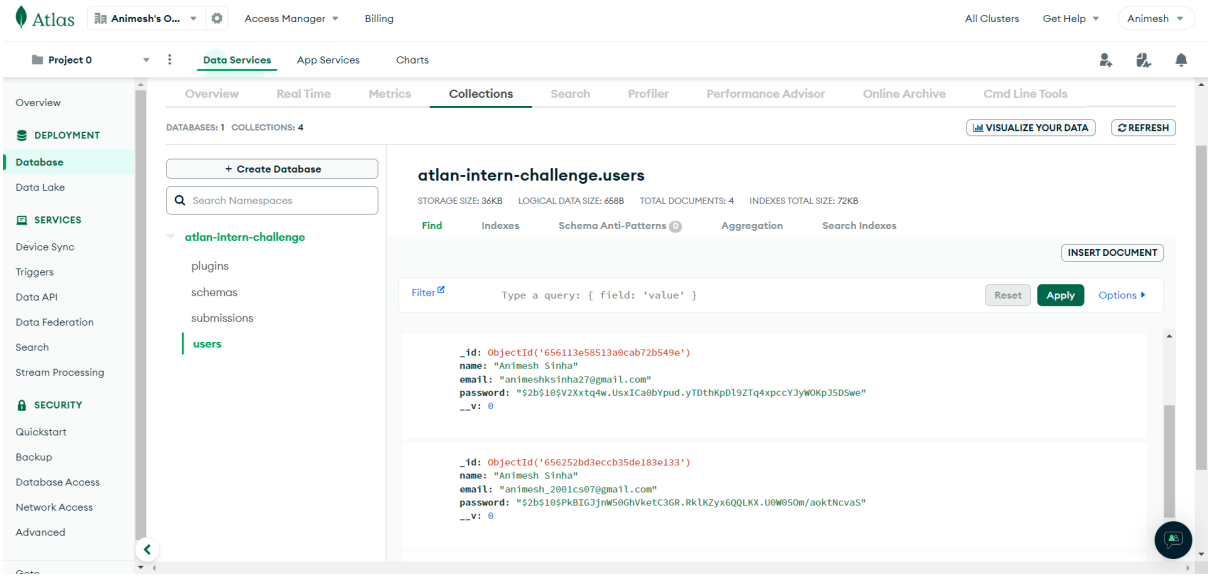
- Use the register script to register new plugins. Example: (See README.md)



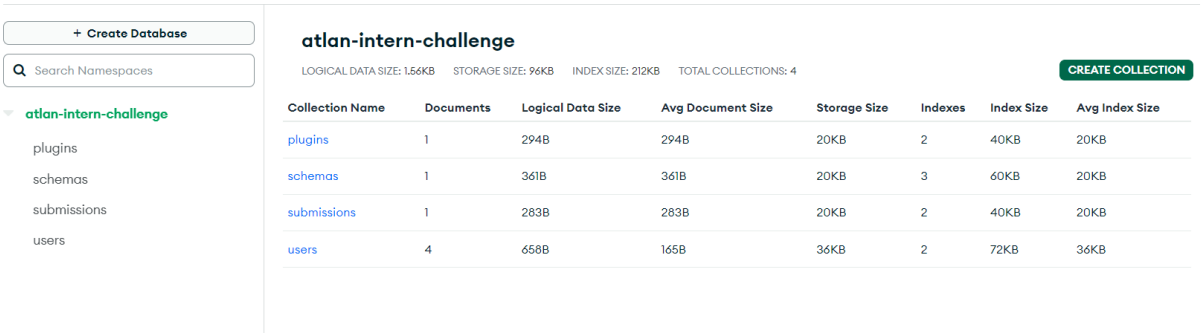
- When a new plugin is running, **onSave** event is triggered right before the form submission is saved to the database collection.

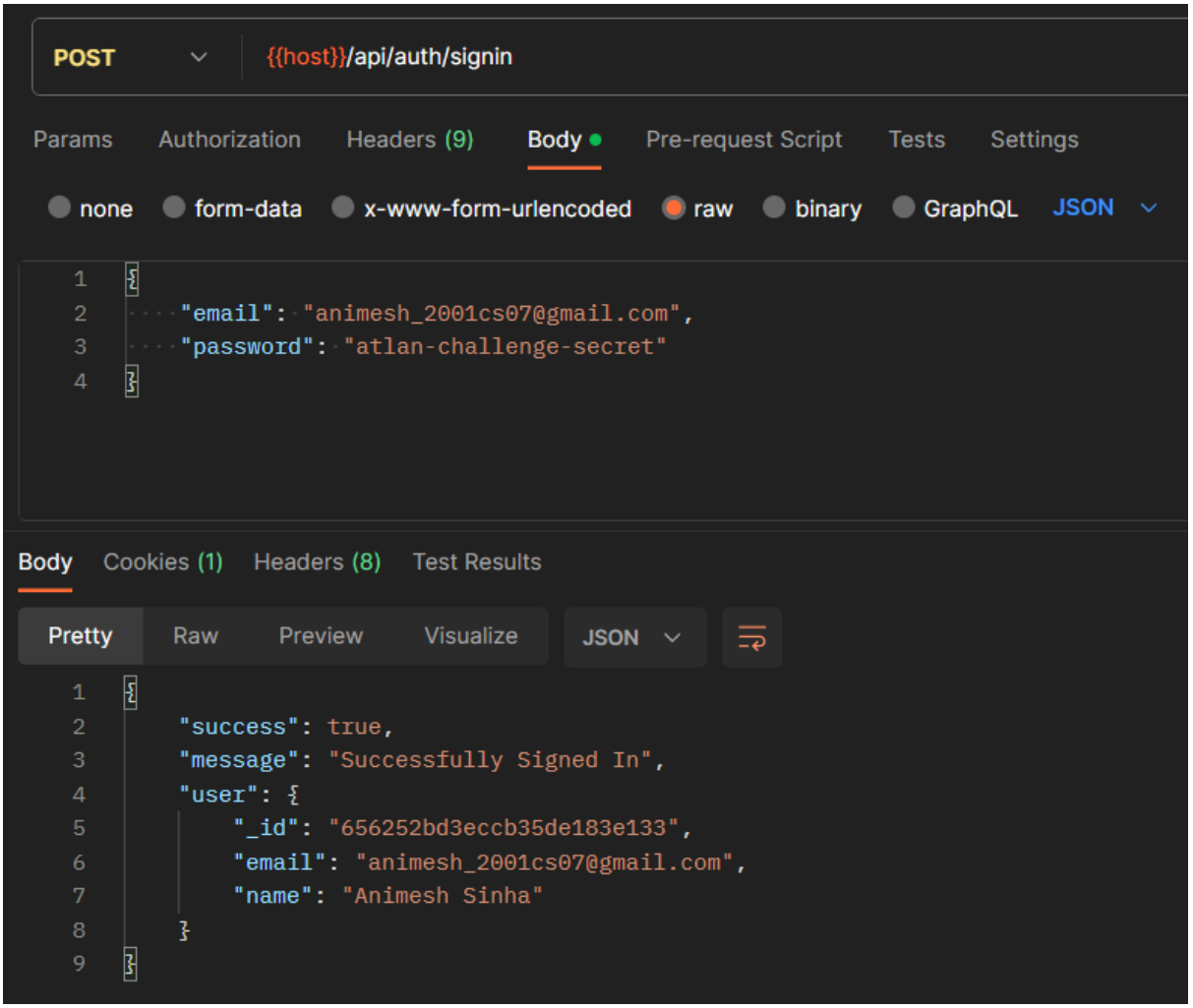
Mongo Database:

Our MongoDB consists of multiple collections: **plugins**, **schemas**, **submissions** and **users**.



You can start using database functions by deploying a MongoDB cluster (I am using MongoDB Atlas) and updating the Mongo URI, USER and PASSWORD in the environment (.env).





Note: You can send requests through POSTMAN (sample collection attached).

Log Manager:

Storing, updating and accessing logs is important in order to ensure system health. Thus, you can use the **logManager** present in "**./util/logManager.ts**". It provides functions to update logs (which has been done in most try catch blocks, but can also be added anywhere else if needed), clear logs, query logs by their **id** field, list all logs after a particular timestamp. The logs are stored in a JSON file "**./logs.json**". We can similarly implement other queries by log type or description.

Note: I have also provided the interface for logs to be directly stored in a collection in our database, which can be extended if necessary.

Preview README.md | logs.json | README.md | index.ts

logs.json > ...


```
    "description": "MONGODB_CONNECTION_SUCCESS",
    "body": "Connected to MONGO DB: mongodb+srv://cluster0.zpruc19.mongodb.net/?retryWrites=true&w=majority"
  },
  {
    "id": 9,
    "timestamp": "2023-11-25T23:23:51.679Z",
    "description": "MONGODB_CONNECTION_SUCCESS",
    "body": "Connected to MONGO DB: mongodb+srv://cluster0.zpruc19.mongodb.net/?retryWrites=true&w=majority"
  },
  {
    "id": 10,
    "timestamp": "2023-11-25T23:42:55.022Z",
    "description": "MONGODB_CONNECTION_SUCCESS",
    "body": "Connected to MONGO DB: mongodb+srv://cluster0.zpruc19.mongodb.net/?retryWrites=true&w=majority"
  },
  {
    "id": 11,
    "timestamp": "2023-11-26T10:59:19.520Z",
    "description": "MONGODB_CONNECTION_FAILURE",
    "body": "Connection to MONGO DB: mongodb+srv://cluster0.zpruc19.mongodb.net/?retryWrites=true&w=majority failed"
  },
  {
    "id": 12,
    "timestamp": "2023-11-26T12:32:26.666Z",
    "description": "MONGODB_CONNECTION_FAILURE",
    "body": "Connection to MONGO DB: mongodb+srv://cluster0.zpruc19.mongodb.net/?retryWrites=true&w=majority failed"
```


logManager functions:

```ts
const LOGPATH = path.join(".", "/logs.json");

// TODO : Shift the logs to Mongo DB directly

export const appendLogs = async (
    logDescription: string,
    logMsg: string
) => {
    try {
        // Read existing JSON content
        const existingData = await fs.readFile(LOGPATH, "utf-8").catch(() => '[]');
        const existingLogs = JSON.parse(existingData);

        console.log("Existing Logs:", existingLogs);

        // Calculate the next id as 1 + the last id in the existing logs
        const nextId = existingLogs.length > 0 ? existingLogs[existingLogs.length - 1].id + 1 : 1;

        const logData = {
            id: nextId,
            timestamp: new Date().toISOString(),
            description: logDescription,
            body: logMsg
        };
```

```
export const getLogsAfterTimestamp = async (
    timestamp: string
) => {
    try {
        // Read existing JSON content or initialize with an empty array if the file is empty or not valid JSON
        const existingData = await fs.readFile(LOGPATH, "utf-8").catch(() => '[]');
        const existingLogs = JSON.parse(existingData);

        // Filter logs based on the given timestamp
        const logsAfterTimestamp = existingLogs.filter((log: any) => log.timestamp >= timestamp);

        return logsAfterTimestamp;
    } catch (err) {
        console.error("Error while reading logs", err);
        throw err;
    }
};
```

Reasoning behind Approach:

- Keeping all the plugins independent and adding them to a common PLUGINS list helps keep pre-existing codebase unaffected by the addition of new plugins.
- Using MongoDB keeps the program and interacting with the database readable and user-friendly.
- We can also scale the database *both horizontally and vertically*, if needed. This will likely be needed as we keep adding new users, forms and plugins over time.
- Keeping an offline (or online) log manager running helps us keep track of system health and lets us easily deep dive in case of any failures.

Running the Platform:

- Please follow the README.md to get started with using this platform.
- You can also see and tinker with samples present in the attached POSTMAN collection.