# SYSTEM DEVELOPMENT METHODOLOGY

## Introduction

This chapter focuses on the description of the development process, methodologies adopted, and design issues related to the Research Data Dashboard for the Faculty of Computing. The implementation involves designing the front-end interface, setting up the backend, managing the database, and coding the functionalities to meet the system requirements. Key components of the implementation are presented, including code snippets of important functions and explanation of system architecture for the overall view of development.

The testing procedures are presented in this chapter as well that are used in making sure that the functionalities work correctly, and the errors are well validated.

## Development Environment

The Faculty of Computing's Research Data Dashboard is a data-dependent system reliant on data extraction from target websites, followed by data analysis. The MERN Stack employed in this project involves MongoDB, Express.js, React, and Node.js. Among the libraries of Node.js, Puppeteer is preferred as the most suitable tool because of its usefulness in the extraction of data. Extracted data are stored in MongoDB in JSON format, which is further accessed by the backend and served to the frontend.
The design of the backend part will be based on Node.js using the Express framework. It will be implemented on the server side, providing necessary APIs. The React-based frontend will fetch values from it and plot, using the library React Chart.js to create interactive charts - bar charts, pie charts, doughnut charts, and line charts, depending on the requirements.

The frontend and backend are on two different ports; still, they use libraries like Axios on them to handle API requests and middleware like CORS to make the communication smooth between them.

## Implementation

The implementation stage means putting into code the system design and architecture mapped out through the requirement analysis such that the system can work and perform the intended goals set out for it. After setting up the backend and frontend, the development of the system commences. The front-end contains numerous React components that are spread across different pages as a means to show different types of information. Many of these components have JavaScript logic that processes the data. The back end is basically the core of the Puppeteer scripts which are the living vital source of data for the whole system. These scripts form the basis of the Research Data Dashboard for the Faculty of Computing.

### View Implementation

The frontend interfaces are displayed in this section that includes the statistical diagrams like pie chart, bar chart generated and the login page.

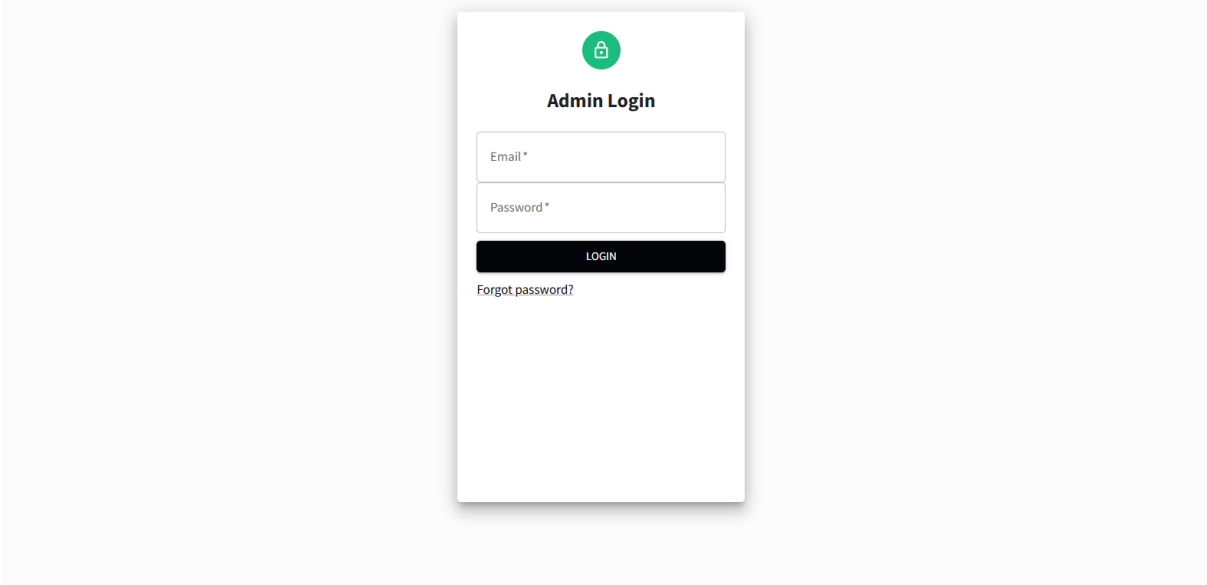


Figure 5.1: Login Page

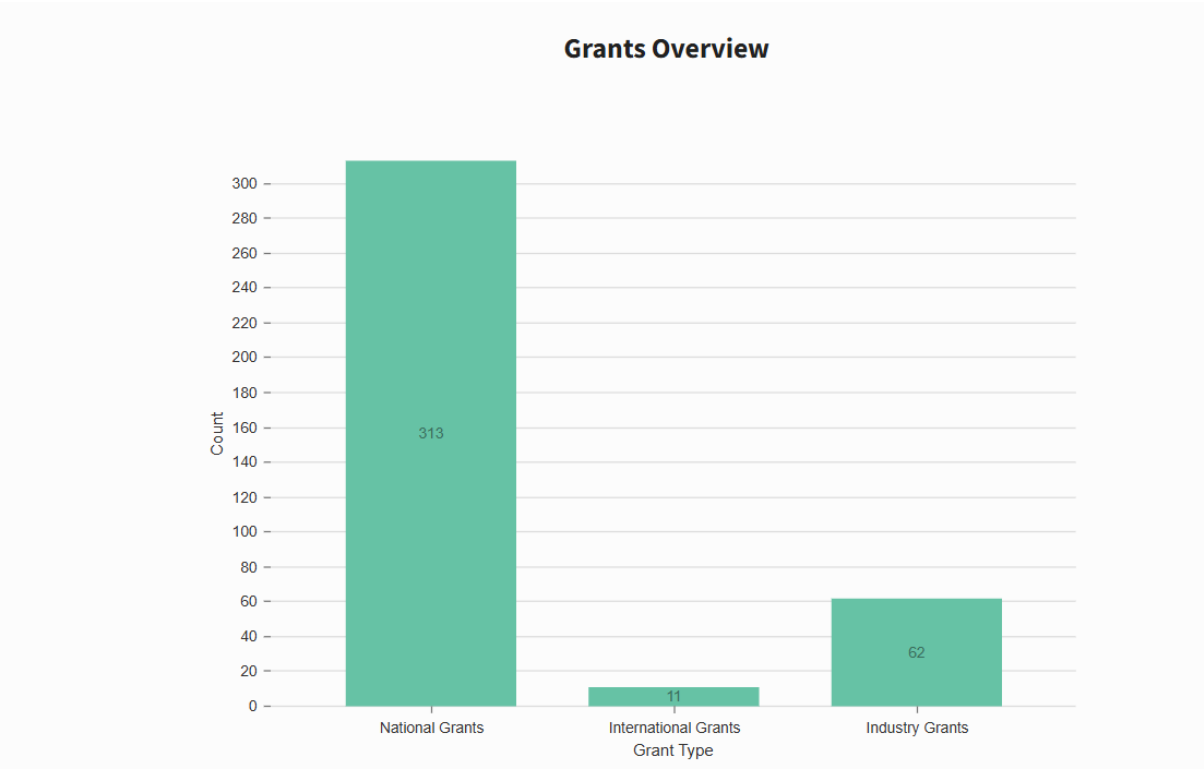


Figure 5.2: Grants Bar Chart

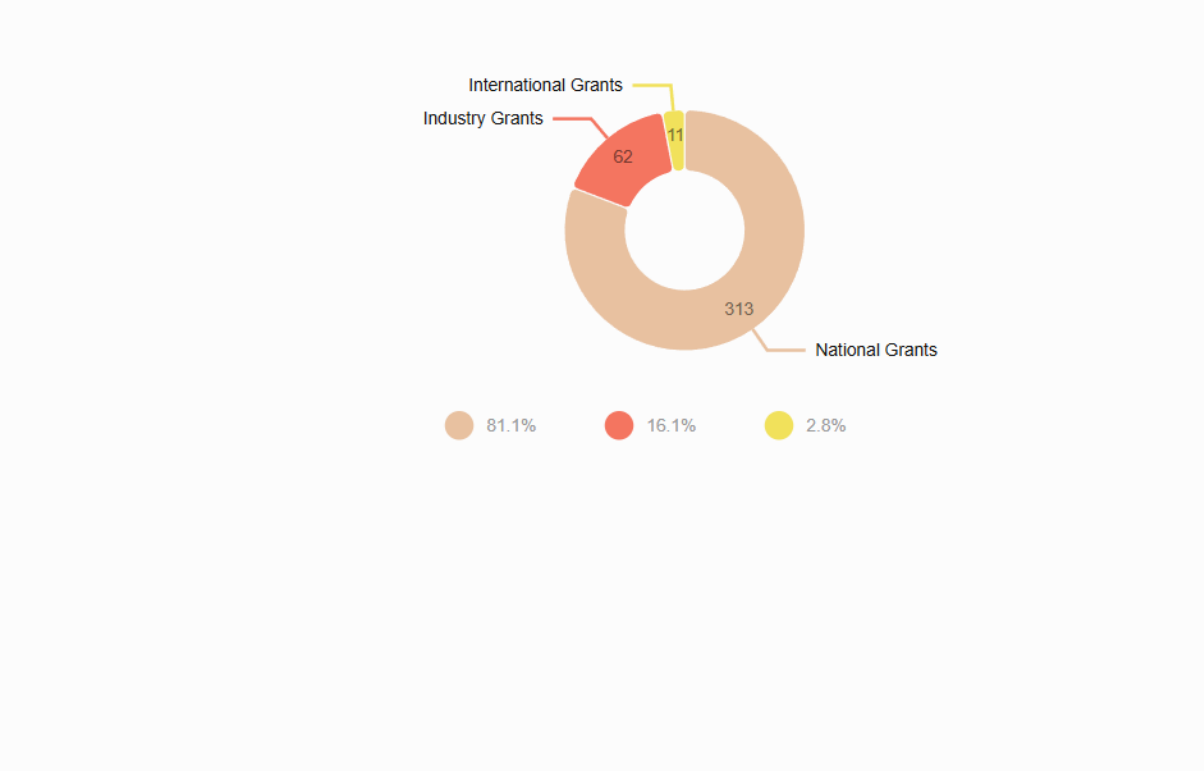


Figure 5.3: Grants Doughnut Chart

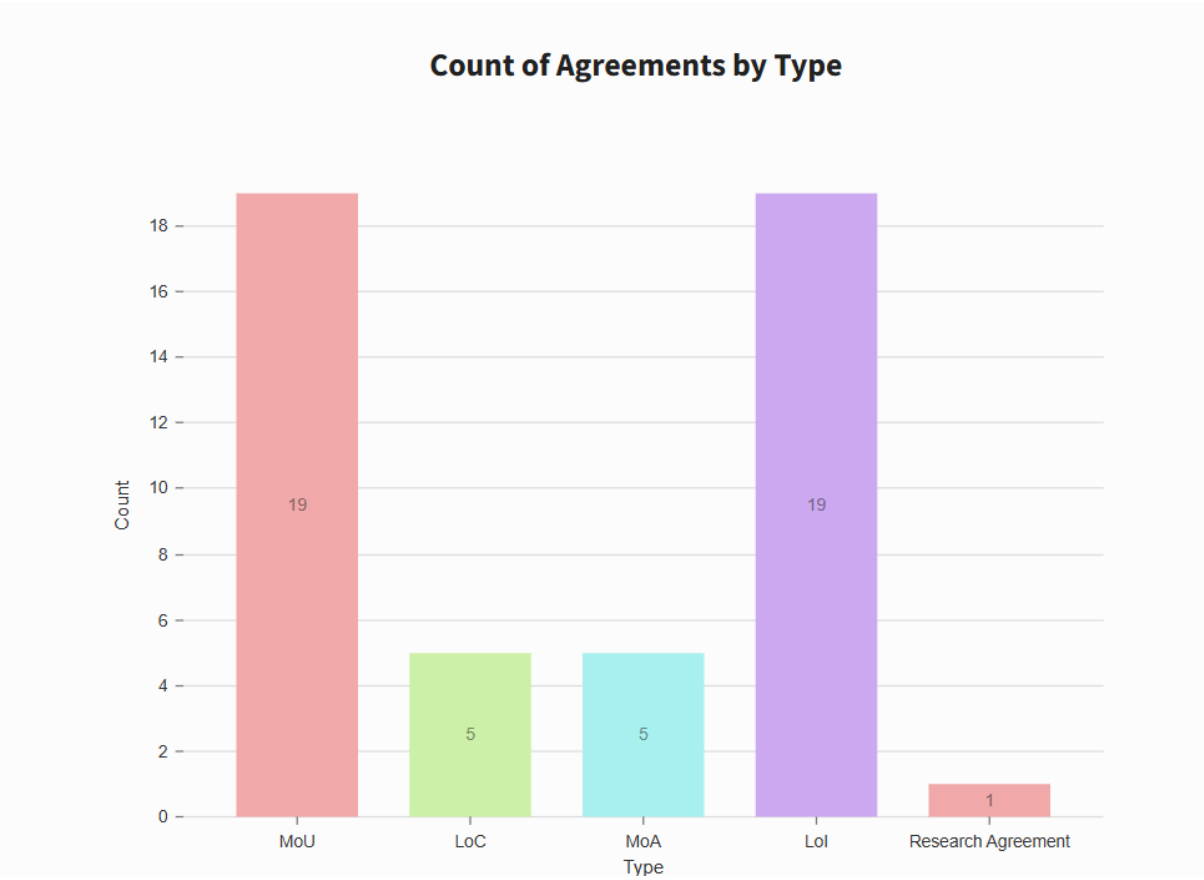


Figure 5.4: Networking data BarChart

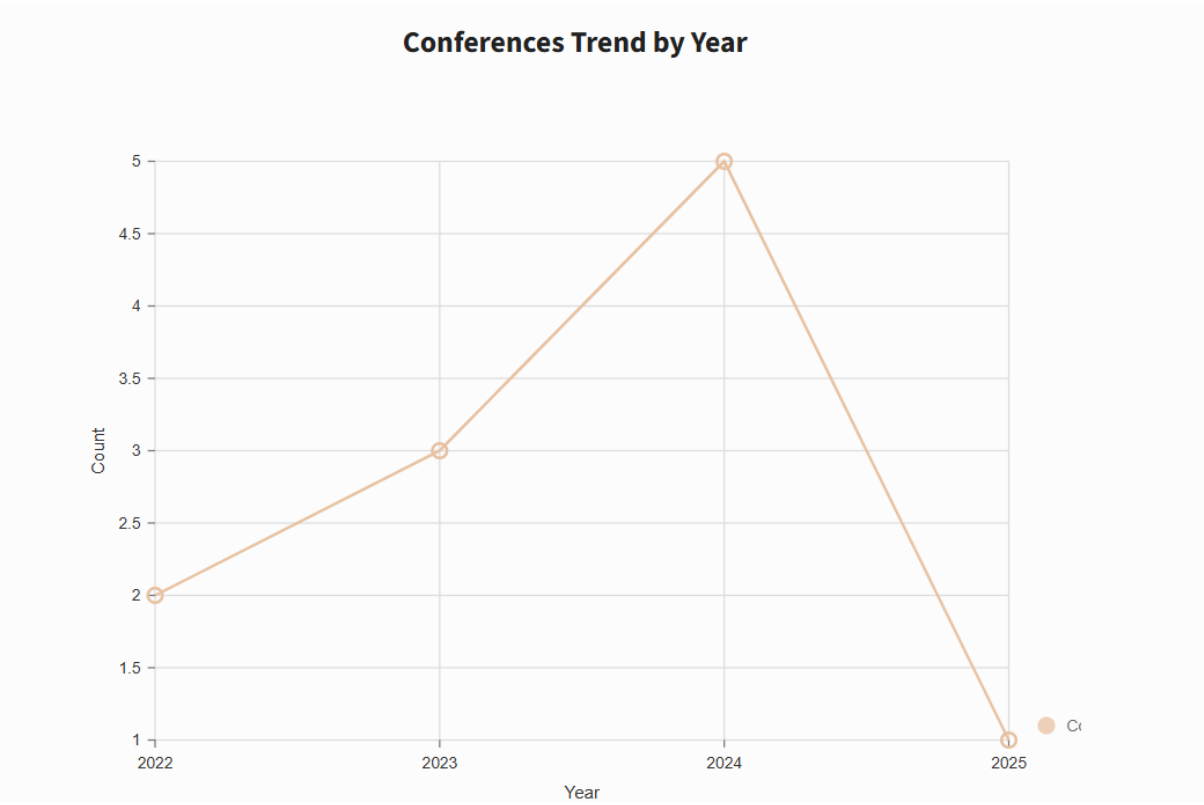


Figure 5.5: Conferences Data Line Chart

## Coding Implementation

This part includes the structure of the code in the system. The database integration, backend setup also including the components used in the frontend will be discussed in this section.

```
// Middleware
app.use(cors());
app.use(express.json());

// MongoDB Connection
mongoose
  .connect("mongodb://127.0.0.1:27017/utm_scholars", {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  })
  .then(() => console.log("Connected to MongoDB: utm_scholars"))
  .catch((err) => console.error("MongoDB connection error:", err));
```

Figure 5.6: MongoDB Connection setup

The api endpoints were called to display the data from the database.

```
app.get("/api/grants", async (req, res) => {
  try {
    const grantsData = await Scholar.aggregate([
      {
        $addFields: {
          NATIONAL_GRANTS_INT: { $toInt: "$NATIONAL_GRANTS" },
          INTERNATIONAL_GRANTS_INT: { $toInt: "$INTERNATIONAL_GRANTS" },
          INDUSTRY_GRANTS_INT: { $toInt: "$INDUSTRY_GRANTS" },
        },
      },
      {
        $group: {
          _id: null,
          totalNationalGrants: { $sum: "$NATIONAL_GRANTS_INT" },
          totalInternationalGrants: { $sum: "$INTERNATIONAL_GRANTS_INT" },
          totalIndustryGrants: { $sum: "$INDUSTRY_GRANTS_INT" },
        },
      },
    ]);

    const {
      totalNationalGrants = 0,
      totalInternationalGrants = 0,
      totalIndustryGrants = 0,
    } = grantsData[0] || {};

    res.json({
      nationalGrants: totalNationalGrants,
      internationalGrants: totalInternationalGrants,
      industryGrants: totalIndustryGrants,
    });
  } catch (err) {
    console.error("Error fetching grants data:", err);
    res.status(500).send("Error fetching grants data");
  }
```

Figure 5.7: Api endpoint for Grants

```
// 4. Training Projects Endpoint (training_2024 data)
app.get("/api/trainingProjects", async (req, res) => {
  try {
    const projects = await TrainingProject.find({});
    res.json(projects);
  } catch (err) {
    console.error("Error fetching training projects:", err);
    res.status(500).send("Error fetching training projects");
  }
});
app.post("/api/trainingProjects", async (req, res) => {
  try {
    console.log("Request body:", req.body); // Debug incoming data
    const newProject = new TrainingProject({
      Bil: Number(req.body.Bil),
      KetuaProjek: String(req.body.KetuaProjek),
      Vot: Number(req.body.Vot),
      TajukProjek: String(req.body.TajukProjek),
      Klien: String(req.body.Klien),
      KosProjek: Number(req.body.KosProjek),
      Year: Number(req.body.Year), // Ensure Year is a number
    });

    const savedProject = await newProject.save();
    console.log("Saved project:", savedProject); // Debug saved data
    res.status(201).json(savedProject);
  } catch (err) {
    console.error("Error saving training project:", err);
    res.status(500).send("Error saving training project");
  }
});
```

Figure 5.8: Api endpoints for Training Data

Here's an example of the frontend react component of the sidebar.

```
const Sidebar = () => {
  const theme = useTheme();
  const colors = tokens(theme.palette.mode);
  const [isCollapsed, setIsCollapsed] = useState(false);
  const [selected, setSelected] = useState("Dashboard");

  return (
    <Box
      sx={{
        "& .pro-sidebar-inner": {
          background: `${colors.primary[400]} !important`,
        },
        "& .pro-icon-wrapper": {
          backgroundColor: "transparent !important",
        },
        "& .pro-inner-item": {
          padding: "5px 35px 5px 20px !important",
        },
        "& .pro-inner-item:hover": {
          color: "#868dfb !important",
        },
        "& .pro-menu-item.active": {
          color: "#6870fa !important",
        },
      }}
    >
      <ProSidebar
        collapsed={isCollapsed}
        style={{
```

Figure 5.9: Frontend Component of Sidebar

```
import React, { useEffect, useState } from "react";
import { ResponsiveBar } from "@nivo/bar";

const BarChart = () => {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchGrantsData = async () => {
      try {
        const response = await fetch("http://localhost:5000/api/grants");
        if (!response.ok) throw new Error(`Error: ${response.statusText}`);
        const jsonData = await response.json();
        // Map API response to chart data format
        const chartData = [
          { type: "National Grants", count: jsonData.nationalGrants },
          { type: "International Grants", count: jsonData.internationalGrants },
          { type: "Industry Grants", count: jsonData.industryGrants },
        ];
        setData(chartData);
      } catch (err) {
        console.error("Error fetching grants data:", err);
        setError("Failed to load chart data.");
      } finally {
        setLoading(false);
      }
    };

    fetchGrantsData();
  }, []);

  if (loading) return <p>Loading chart...</p>;
  if (error) return <p>{error}</p>;

  return (
```

Figure 5.10: Frontend Component for Bar Chart

**Architecture Implementation**

The architecture followed by the Dashboard is the Model-View Controller architecture. A model holds the database models of the system. The controller holds the logic and enables the communication between the front-end and back-end while the views are the React components.

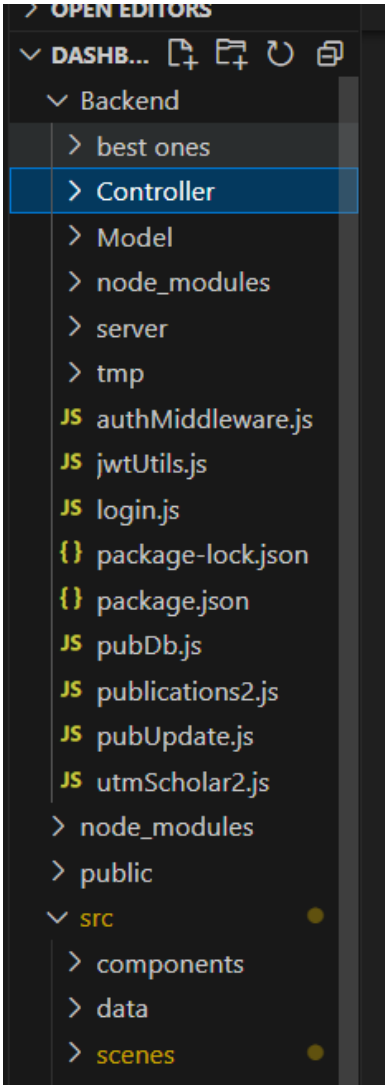


Figure 5.11: Architecture Implementation

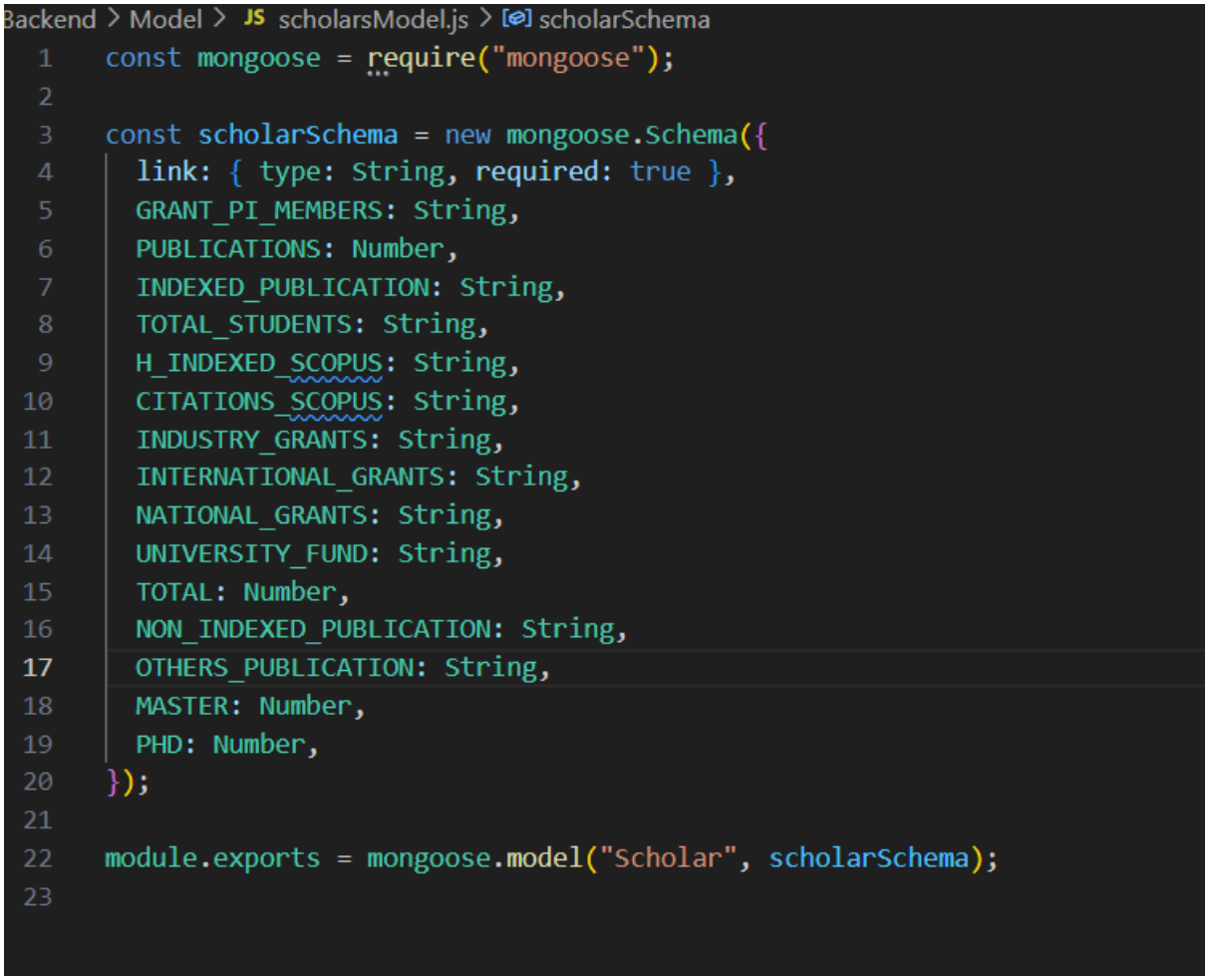


Figure 5.12: Scholars Model for database

```
const authService = require("../login");

async function login(req, res) {
  try {
    const { email, password } = req.body;

    // Verify the admin login
    const adminUser = await authService.verifyAdmin(email, password);

    // Return a success response with the admin's info
    res.json({
      message: "Login successful",
      admin: {
        id: adminUser._id,
        email: adminUser.email,
        name: adminUser.name,
      },
    });
  } catch (error) {
    res.status(401).json({ message: error.message }); // Return error messages directly
  }
}

module.exports = { login };
```

Figure 5.13: Login Auth Controller

**System Testing**

The immediate vital activity after the implementation of the Research Grant Finder system would be the testing of the system for whether the system works as it was required and designed. Tests can be performed in a number of ways: black box testing, white box testing, and user acceptance testing. Each of them has different advantages and helps in making the system robust. Research Grant Finder shall use Black Box Testing and User Acceptance Testing during the testing process.

**Black Box Testing**

Black-box testing is normally a software testing approach that focuses on assessment functionality based on application behavior. It doesn't require much use of the system code. It always involves investigations of input and output values of the software system regarding every test case. The objective of this testing is to ascertain whether the system acts as documented.

**Test Case Design**

Test TC002 for module <**Authentication & View**>: **<View Dashboard Analytics>**

Test execution steps:

| No. | Action | Test data | Expected result | Actual Result | Pass/Fail |
|---|---|---|---|---|---|
| 1. | Dashboard is visible | - | Home Page is displayed | Dashboard is displayed | Pass |
| 2 | User clicks on the Publication metrics button. | - | Button works properly and user is taken to publication metrics page. | Publication Button works | Pass |
| 3. | User can view the dashboard analytics | - | All the data are displayed properly. | Data is displayed | Pass |

**Test TC006 for Module Publication: Generate H-Indexed Publication**

Test execution steps:

| No. | Action | Input data | Expected result | Actual Result | Pass/Fail |
|---|---|---|---|---|---|
| 1. | User is taken to the homepage | - | Home Page is displayed | Displayed Properly | Pass |
| 2 | User clicks on the Citations metrics button. | - | Button works properly and user is taken to Citations metrics page. | Button is working | Pass |
| 3. | User can view the generated H-Index Publication data from the database | H-Index Publication Data | Data has been successfully retrieved from database. | Data Displayed | pass |

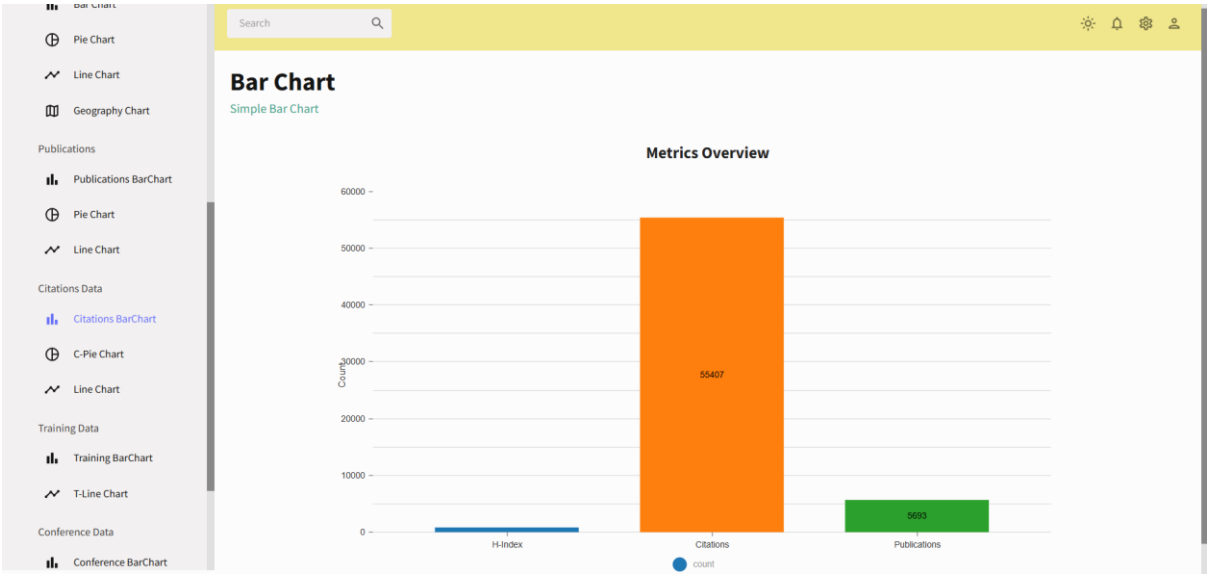


Figure 5.14: Citations Bar Chart

## User Acceptance Testing

User Acceptance Testing is one form of testing wherein the stakeholder goes on to review himself in the system functionalities to make sure they are all up to specifications. In this GRF project, the stakeholder needed to do UAT for the functionality of scraping.

### Scraping Functionality

| No. | Function | Description | Performed By |
|---|---|---|---|
| 1 | Scraping the data from the website | This function ensures the scraping of the desired data | Admin |