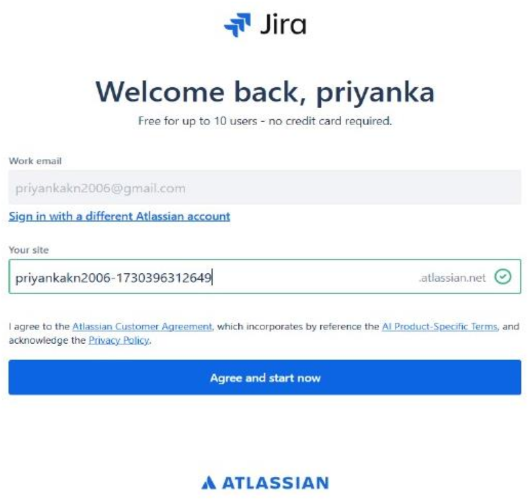
TABLE OF CONTENT:

1. How to create project plan and product backlog for project and User story creation using Jira Software.

Step 1: Log in to Jira

1. **Open your browser** and search for "Jira Login."
2. **Log in with your Gmail account** or use your existing Jira credentials.


Jira

Welcome back, priyanka

Free for up to 10 users - no credit card required.

Work email: priyankakn2006@gmail.com

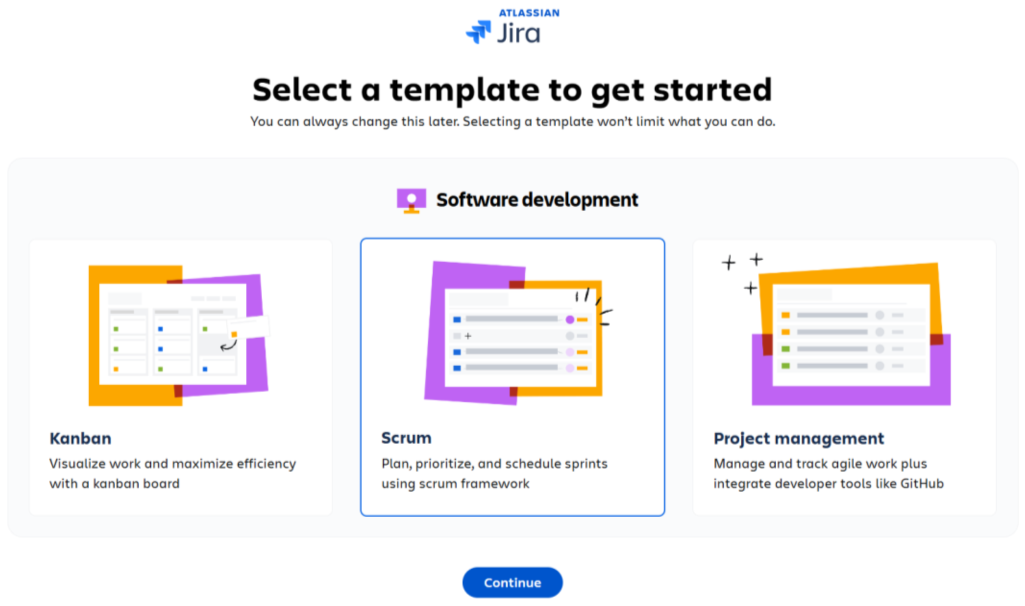
Sign in with a different Atlassian account

Your site: priyankakn2006-1730396312649 .atlassian.net

I agree to the Atlassian Customer Agreement, which incorporates by reference the AI Product-Specific Terms, and acknowledge the Privacy Policy.

Step 2: Create a New Project

1. Once logged in, **select a template** to start your project.


ATLASSIAN Jira

Select a template to get started

You can always change this later. Selecting a template won't limit what you can do.

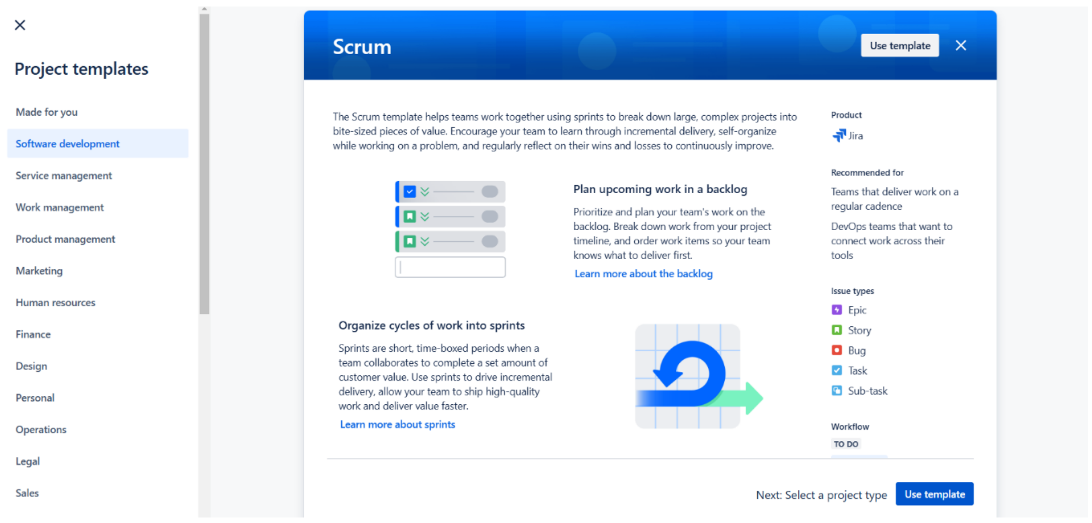
Software development

Kanban: Visualize work and maximize efficiency with a kanban board

Scrum: Plan, prioritize, and schedule sprints using scrum framework

Project management: Manage and track agile work plus integrate developer tools like GitHub

2. From the top menu bar, go to the **Project** tab and select **Create Project**.
3. Choose **Software Development** as the project template.
4. Click on **Use Template**.



Project templates

- Made for you
- Software development
- Service management
- Work management
- Product management
- Marketing
- Human resources
- Finance
- Design
- Personal
- Operations
- Legal
- Sales

Scrum

The Scrum template helps teams work together using sprints to break down large, complex projects into bite-sized pieces of value. Encourage your team to learn through incremental delivery, self-organize while working on a problem, and regularly reflect on their wins and losses to continuously improve.

Plan upcoming work in a backlog

Prioritize and plan your team's work on the backlog. Break down work from your project timeline, and order work items so your team knows what to deliver first.

Learn more about the backlog

Organize cycles of work into sprints

Sprints are short, time-boxed periods when a team collaborates to complete a set amount of customer value. Use sprints to drive incremental delivery, allow your team to ship high-quality work and deliver value faster.

Learn more about sprints

Product: Jira

Recommended for: Teams that deliver work on a regular cadence; DevOps teams that want to connect work across their tools

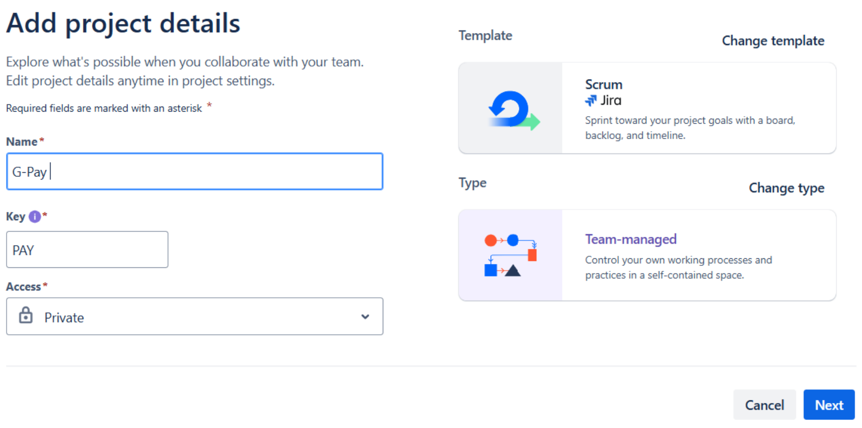
Issue types: Epic, Story, Bug, Task, Sub-task

Workflow: TO DO

Next: Select a project type — Use template

5. Select **Team-managed project** as the project type and give your project a **name**.

6. Click on **Next** to proceed.



Add project details

Explore what's possible when you collaborate with your team. Edit project details anytime in project settings.

Required fields are marked with an asterisk *

Name*: G-Pay

Key*: PAY

Access*: Private

Template — Change template

Scrum
Jira
Sprint toward your project goals with a board, backlog, and timeline.

Type — Change type

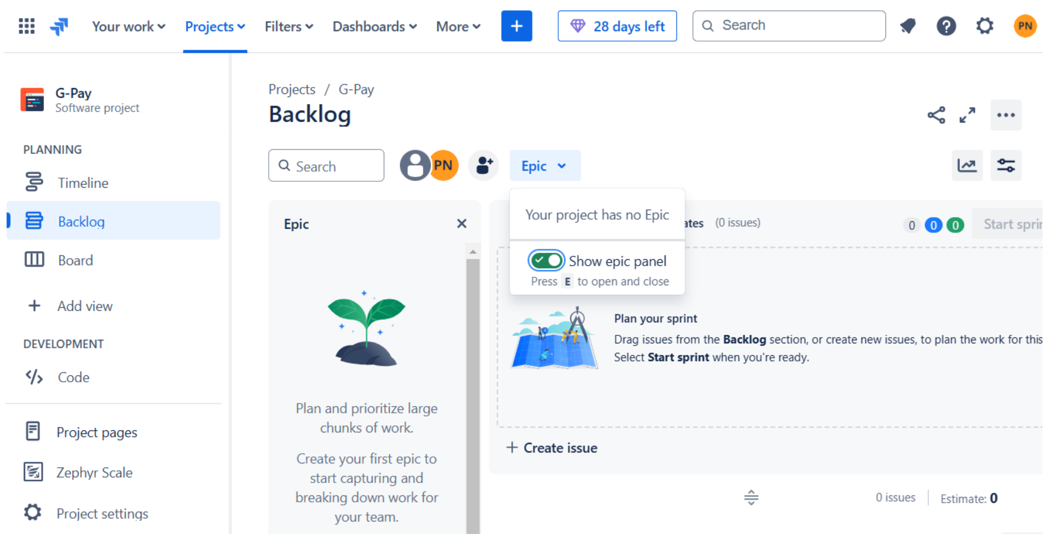
Team-managed
Control your own working processes and practices in a self-contained space.

Cancel — Next

Step 3: Set Up the Product Backlog

1. Go to the **Backlog** view in your project.

2. **Enable Epics** by clicking on the Epic panel in the backlog view.



Your project has no Epic

Show epic panel
Press E to open and close

Epic

Plan and prioritize large chunks of work.

Create your first epic to start capturing and breaking down work for your team.

(0 issues) 0 0 0 Start sprint

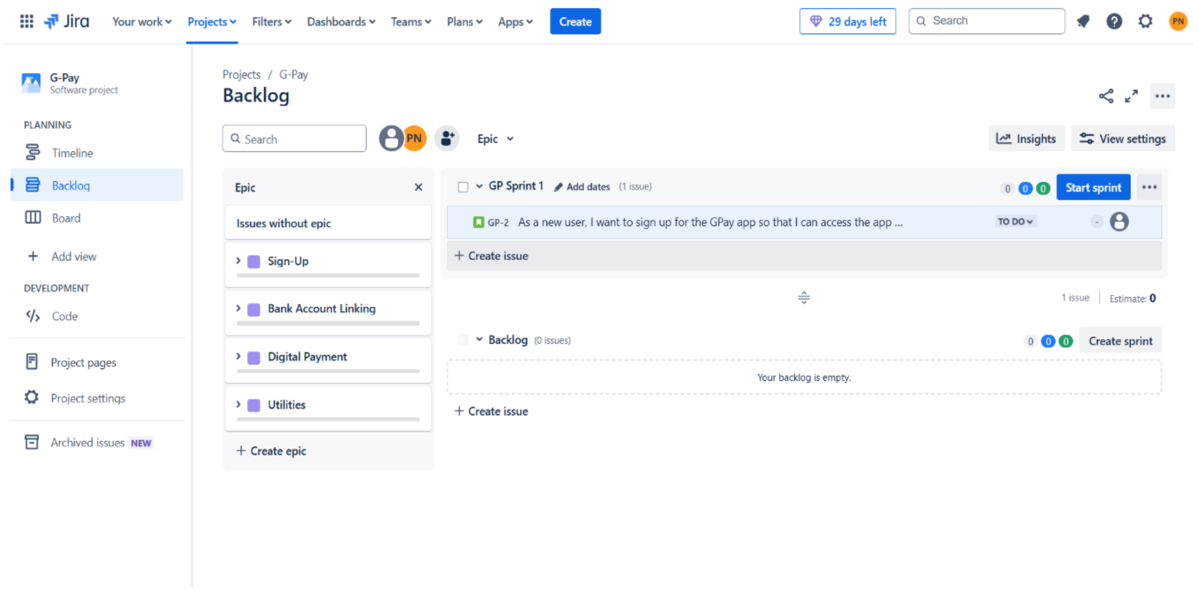
Plan your sprint

Drag issues from the **Backlog** section, or create new issues, to plan the work for this sprint. Select **Start sprint** when you're ready.

+ Create issue

0 issues | Estimate: **0**

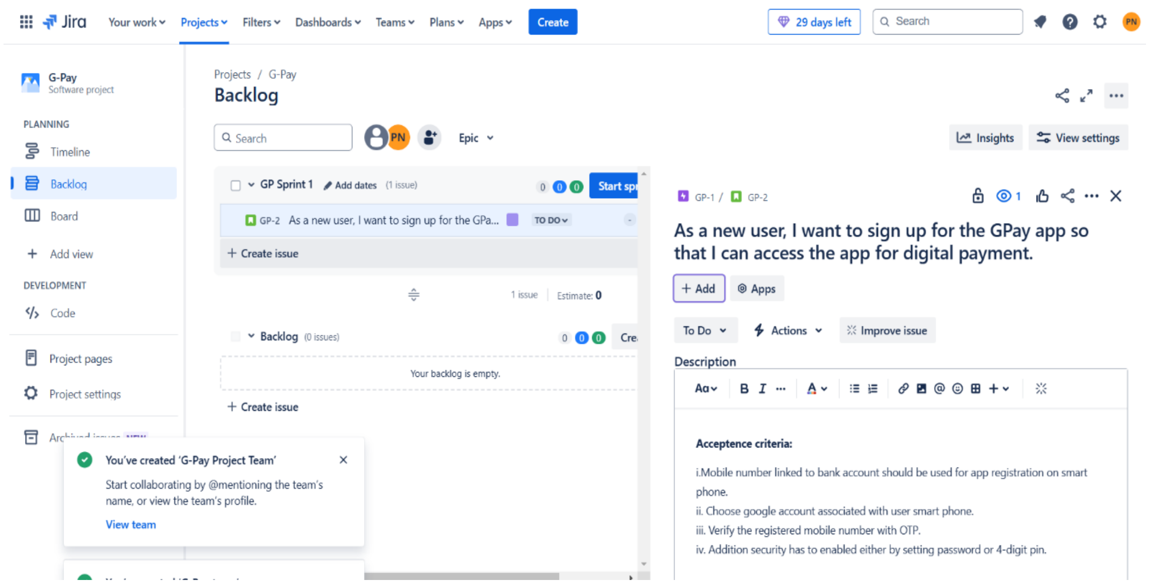
3. **Create an Epic**:
   - From the top menu, select **Issues**, then **Create Issue**.
   - Choose **Epic** as the issue type, and write a summary for the user story.
4. **Create User Stories and Related Issues**:
   - For each epic, create related issues such as **User Story**, **Acceptance Criteria**, and **Tasks** by selecting appropriate issue types.
   - For example, create issues for tasks like **Bank Account Linking**, **Digital Payment**, and **Utilities**.



GP-2 As a new user, I want to sign up for the GPay app so that I can access the app ...

Your backlog is empty.

5. Once issues are created, **drag and drop** them to arrange the order of tasks in the backlog.



GP-2 As a new user, I want to sign up for the GPa...

Your backlog is empty.

You've created 'G-Pay Project Team'

Start collaborating by @mentioning the team's name, or view the team's profile.

As a new user, I want to sign up for the GPay app so that I can access the app for digital payment.

Acceptance criteria:

i.Mobile number linked to bank account should be used for app registration on smart phone.

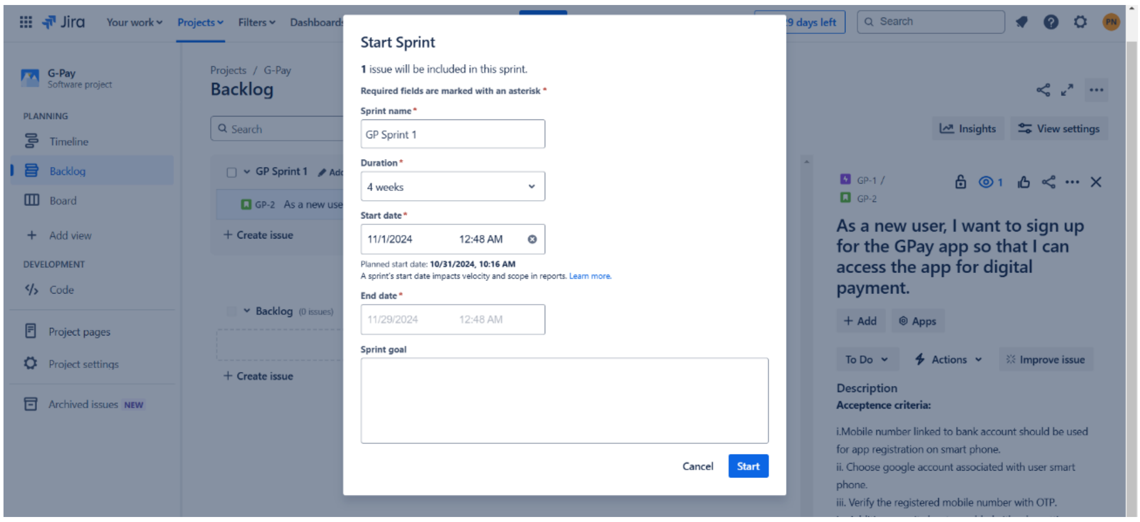
ii. Choose google account associated with user smart phone.

iii. Verify the registered mobile number with OTP.

iv. Addition security has to enabled either by setting password or 4-digit pin.

Step 4: Start a Sprint

1. Select **Sprint Planning** from the backlog view.
2. Drag issues you want to include into the **Sprint** section.
3. Set the **Start Date** and **End Date** for the sprint.
4. Click **Start Sprint** to begin tracking.



Start Sprint

1 issue will be included in this sprint.

Required fields are marked with an asterisk *

Sprint name *
GP Sprint 1

Duration *
4 weeks

Start date *
11/1/2024 12:48 AM

Planned start date: **10/31/2024, 10:16 AM**
A sprint's start date impacts velocity and scope in reports. Learn more.

End date *
11/29/2024 12:48 AM

Sprint goal

Cancel Start

GP-1 / GP-2

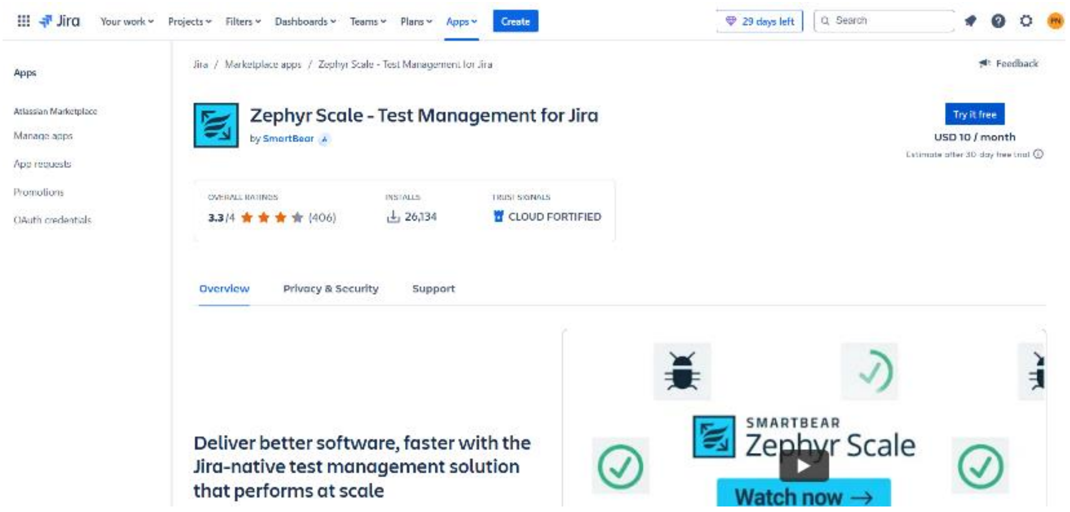
As a new user, I want to sign up for the GPay app so that I can access the app for digital payment.

Description
Acceptence criteria:

i.Mobile number linked to bank account should be used for app registration on smart phone.
ii. Choose google account associated with user smart phone.
iii. Verify the registered mobile number with OTP.

Step 5: Add Zephyr Scale for Test Case Management

1. From the top menu bar, go to the **Apps** tab and select **Explore Apps**.
2. **Search for Zephyr Scale** and install it (choose "Try for Free" if available).
3. **Enable Zephyr Scale** within your project once installed.



Zephyr Scale - Test Management for Jira
by SmartBear

USD 10 / month
Estimate after 30 day free trial

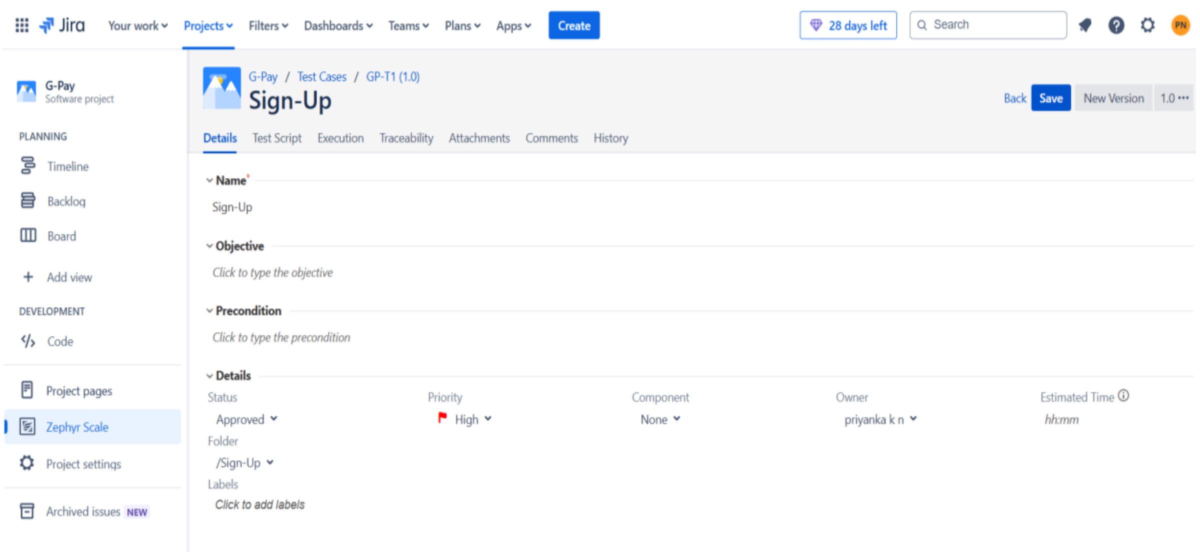
OVERALL RATINGS 3.3/4 (406)
INSTALLS 26,134
TRUST SIGNALS CLOUD FORTIFIED

Deliver better software, faster with the Jira-native test management solution that performs at scale

Step 6: Create Test Cases in Zephyr Scale

1. Go to **Zephyr Scale** and create a new test case.

2. **Name the Test Case** based on the issue (e.g., **Sign-Up**).
3. Add details for the test case:
   - **Objective**: Describe the purpose of the test.
   - **Preconditions** (optional): List any setup required before testing.
   - **Priority**: Set according to the importance of the test.



Sign-Up

Name*
Sign-Up

Objective
Click to type the objective

Precondition
Click to type the precondition

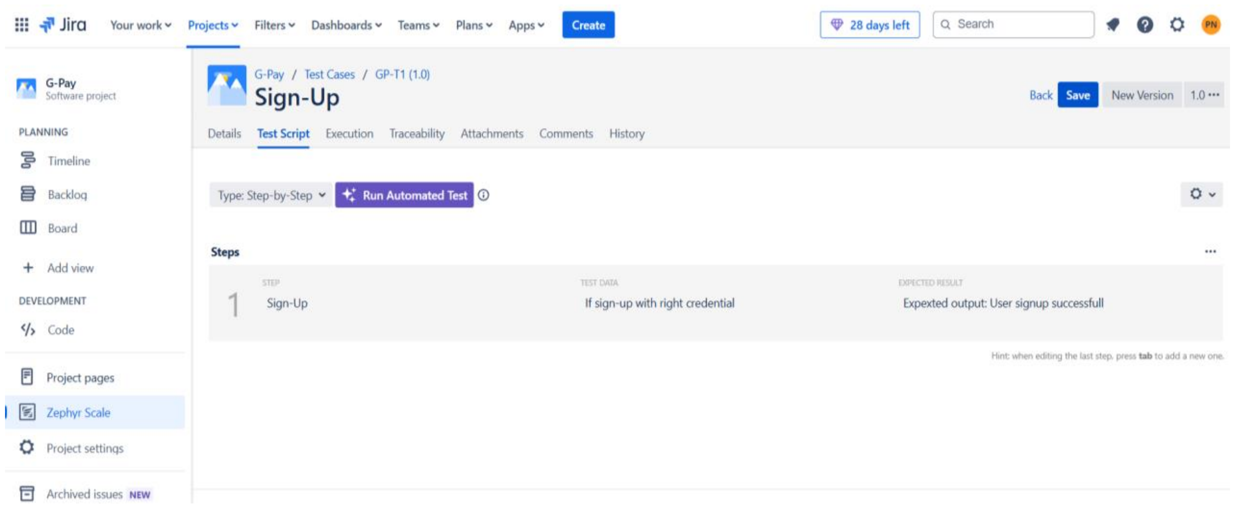
Details

Status	Priority	Component	Owner	Estimated Time
Approved	High	None	priyanka k n	hh:mm

Folder
/Sign-Up

Labels
Click to add labels

4. Go to the **Test Script** section to add steps:
   - **Description**: Describe each step (e.g., "Enter Valid Registration Details").
   - **Test Data**: List data inputs, such as valid first name, last name, email, and password.
   - **Expected Result**: Describe the expected outcome (e.g., "User is redirected to the login page").
5. **Save** the test case.



Sign-Up

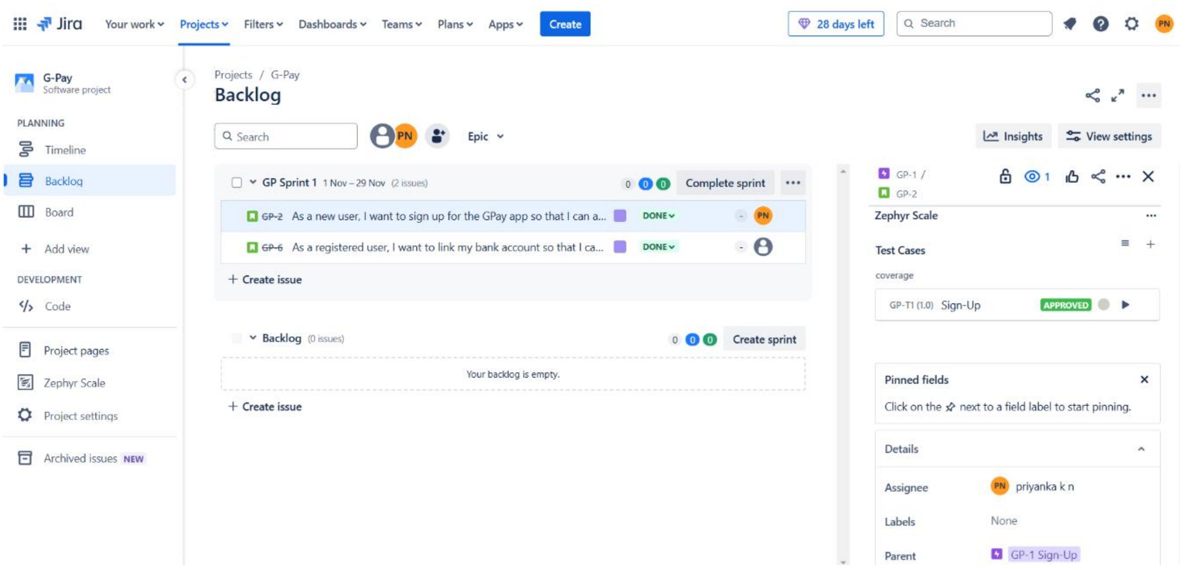
Steps

	STEP	TEST DATA	EXPECTED RESULT
1	Sign-Up	If sign-up with right credential	Expexted output: User signup successfull

Hint: when editing the last step, press **tab** to add a new one.

Step 7: Link Test Cases to Issues

1. Go back to the **Backlog** and select an issue.
2. Under **Zephyr Scale**, select **Create a New Test Case** or **Add an Existing Test Case**.
3. Link the relevant test cases to each issue, then **Add** them to the issue.



GP Sprint 1 1 Nov – 29 Nov (2 issues)

GP-2 As a new user, I want to sign up for the GPay app so that I can a... DONE

GP-6 As a registered user, I want to link my bank account so that I ca... DONE

Backlog (0 issues)

Your backlog is empty.

GP-1 / GP-2

coverage

GP-T1 (1.0) Sign-Up APPROVED

Click on the next to a field label to start pinning.

Assignee: priyanka k n

Labels: None

Parent: GP-1 Sign-Up

2. Create UI/UX design - for created user stories (wire framing).

Step 1: Set Up the Frame

3. Open **Figma** and create a new design file.
4. Select the **Frame Tool (F)**, choose the iPhone Mini preset, and set its dimensions to **375 x 812**.
5. Rename the frame to "Google Pay Wireframe" for easy organization.

Step 2: Add the Top Header Section

1. At the top of the frame, create a **header** with a white background.
2. **Open Iconify Plugin**:
   - Add the **QR code scan icon** on the left side.
   - Add the **profile icon** on the right side.
3. Place the **G Pay** logo text in the center of the header.
4. Below the header, add small icons of people in circular shapes to represent the user connections in the app.

Step 3: Main Illustration

1. In the middle of the screen, create an area for the **main illustration** showing people interacting (you can create placeholder shapes or draw the graphic using Figma shapes or you can add images).
2. Center this illustration and place it below the header area to mimic the original design.

Step 4: People Section

1. Below the illustration, add a **label** for the "People" section.
2. Add a **horizontal scrollable row of circular icons** representing contacts.
   - Use Iconify to add placeholder icons or colored circles with text initials (e.g., "S," "R," etc.) for each contact.
   - Label each icon with a name below the circle.
3. Add a **Show More** button to the right of the contacts row.

Step 5: Divider Line

1. Draw a horizontal line below the "People" section to separate it from the next section, "Businesses and Bills."

Step 6: Businesses and Bills Section

1. Below the divider, add a label titled **"Businesses and Bills"**.

2. Add another **horizontal scrollable row of business logos** (e.g., MakeMyTrip, redBus, Tata Sky, etc.). Use the Iconify plugin to find placeholder icons for each business or create circle placeholders with text.
3. Include an **Explore icon** on the right side of the row for additional businesses.

Step 7: New Payment Button

1. Add a large button in the middle of the screen with the label **"+ New Payment"**.
2. Use a rounded rectangle with centered text inside it.

Step 8: Additional Options

1. Add another **Show More** button below the business icons row for additional options.
2. Align and space these elements to maintain visual balance.

Step 9: Review and Group Elements

1. Check alignment and spacing for consistency.
2. Group related items (e.g., header icons, people icons, businesses row) to keep the design organized within Figma.

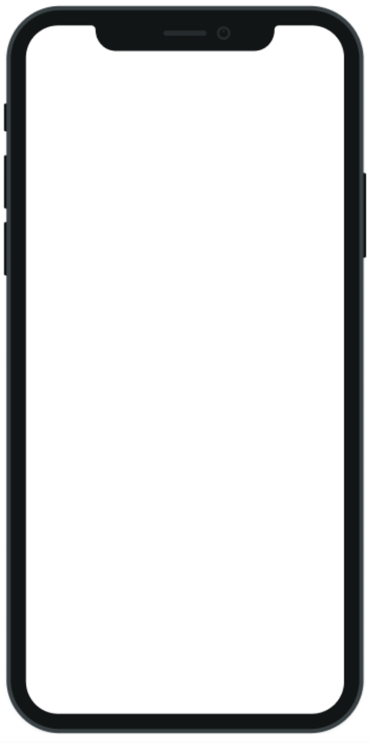
Step 10: Final Review and Save

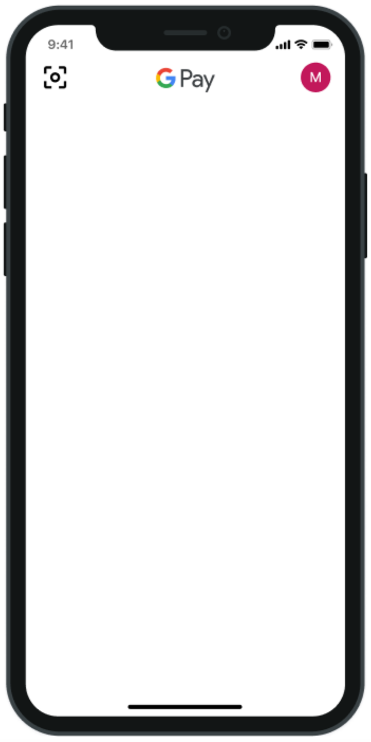
1. Review the layout on Figma, ensuring it resembles the Google Pay app.
2. Save the file and export if needed.

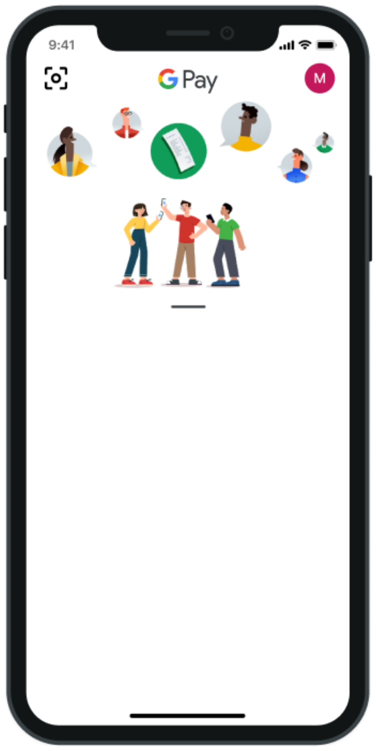
Step 11: Preview or Present the Wireframe

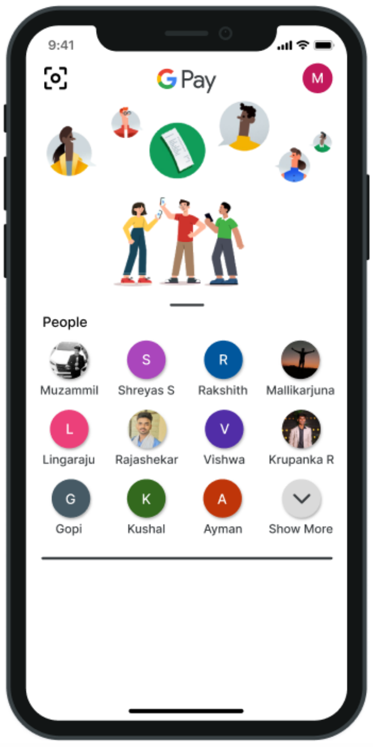
1. In Figma, click on the **Play** icon located in the top-right corner of the screen. This opens the **Prototype** view, allowing you to see a live preview of your design.
2. Your wireframe will open in a new tab, simulating how it would look on the **iPhone Mini**.
3. To share this preview with others, click **Share Prototype** in the prototype view, then copy the link provided by Figma.
4. You can also adjust the device frame in the Prototype settings to show it on a device screen.

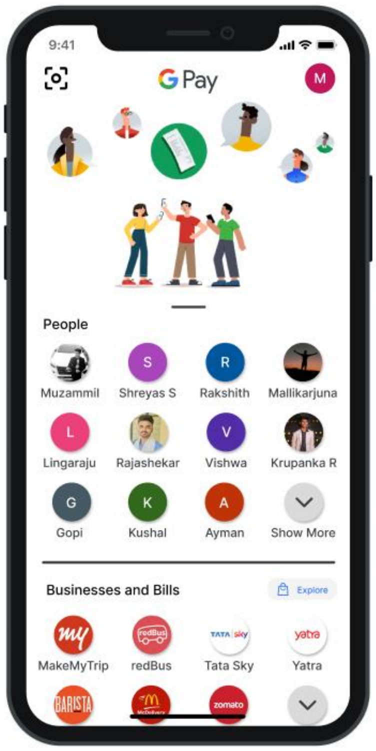
Step-by-Step Images for Wireframe Creation:

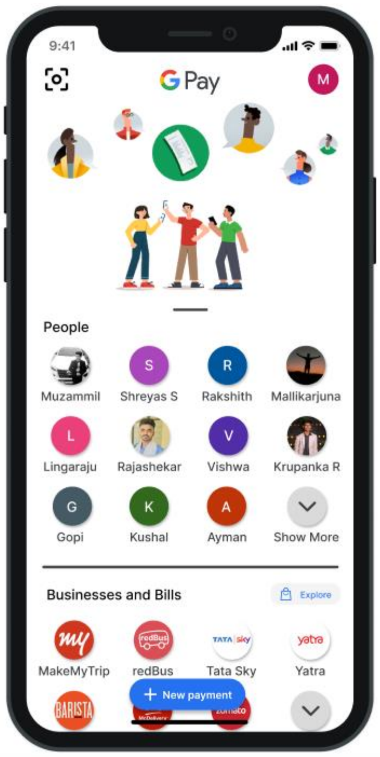












3. GitHub Commands.

Sl. no.	Commands	Example	Description
1.	*git --version*		To display the version of the git downloaded on your PC
2.	*git config --global user.name < "Username" >* *git config --global user.email < "valid-email" >*	git config --global user.name "riya123" git config –global user.email "riya2517@gmail.com"	GitHub uses the email address and username set in your local Git configuration to associate commits pushed from the command line with your account on GitHub.com.
3.	*git config --list*		To view what changes were made during the configuration.
4.	*git clone < link of a repository from Git hub >*	Git clone https://github.com/riya123/project.git	To clone a repository from remote machine (Github) to local machine (PC)
5.	*git status*		To display the status of the code or a file • Untracked- new files that git hasn't tracked yet. • Modified- changed. • Staged- file is ready to be committed. • Unmodified- unchanged.
6.	*git init*		To initialize a new, empty repository.
7.	*git add < file name >* *or* *git add <.>*	git add f1.html git add .	To add a new or changed files in your working directory to gitstaging area. (Use "." to add all the files that's in your working directory to the git staging area.)
8.	*git commit -m "message"*	git commit -m "this is my firstcommit"	To commit the changes that is made in a file. (-m is an option to specify the message that should be displayed)

9.	*git push origin main*		To upload local repository (PC) content to remote repository (Git hub).
10.	*git remote add origin <link of the repository form git hub>*	git remote add origin https://github.com/riya123/repo.git	To add a new remote repository with the name of origin.
11.	*git remote -v*		To verify that the remote repository actually exists.
12.	*git branch*		To check which branch we are on currently.
13.	*git branch -M <name>*	git branch -M main	To rename a branch
14.	*git push -u origin main*		-u : To set upstream, meaning that the next time you want to push something into git hub, you just have to type "git push" instead of typing the full command every time. It specifies that you want To work on "origin main"for a long time.
15.	*git checkout <branch name>*	git checkout sub1	To navigate/get inside of a branch.
16.	*git checkout -b <new branch name>*	git checkout -b sub2	To create a new branch.
17.	*git branch -d <branch name>*	git branch -d sub1	To delete a branch
18.	*git diff <branch name>*	git diff sub2	To compare commits, files, branches and more
19.	*git pull origin main*		To fetch and download content from remote repository and update the local repository to match the content.
20.	*git merge <branch name>*	git merge sub3	To merge 2 branches together.
21.	*git reset <file name>*	git reset sub4	To undo the changes after adding the changes that were done to a file
22.	*git reset HEAD~1*		To undo the changes by 1 step/commit which has already been committed.
23.	*git log*		Shows the commit history of the current active branch. (Commit hash can be

			copiedfrom here)
24.	*git reset <commit hash>*	git reset 56142346434sdf64645sfd4	Undoing committed changes by many commits. Multiple commits can be undo-ed by this. (But the undone changes will be just in git but not visible in VS code)
25.	*git reset --hard <commit hash>*	git reset --hard 56142346434sdf64645sfd4	The undone changes will bevisible in VS code

4. Create form validation using JavaScript.

Step 1: Create the HTML Form (Form.html)

1. **Create a new file** called Form.html.

Form.html

```
<!DOCTYPE html>
<html>

<head>
    <title>Registration Form</title>
    <link rel="stylesheet" href="Style.css">
</head>

<body>
    <h1>Registration Form</h1>
    <form onsubmit="return validateForm()">
        <label for="name">Name</label>
        <input type="text" id="name" /><br><br>

        <label for="email">Email ID</label>
        <input type="email" id="email" /><br><br>

        <label for="address">Address</label>
        <input type="text" id="address" /><br><br>

        <input type="submit" value="Submit" />
    </form>

    <script>
        function validateForm() {
            // Get the values of each input field
            var name = document.getElementById("name").value;
            var email = document.getElementById("email").value;
            var address = document.getElementById("address").value;

            // Check if any field is empty
            if (name === "" || email === "" || address === "") {
                alert("Fill the required fields before submitting!");
                return false; // Prevent form submission if validation fails
            }

            alert("Registration successful!"); // Success message
        }
    </script>
</body>
</html>
```

Step 3: Create the External CSS File (Style.css)

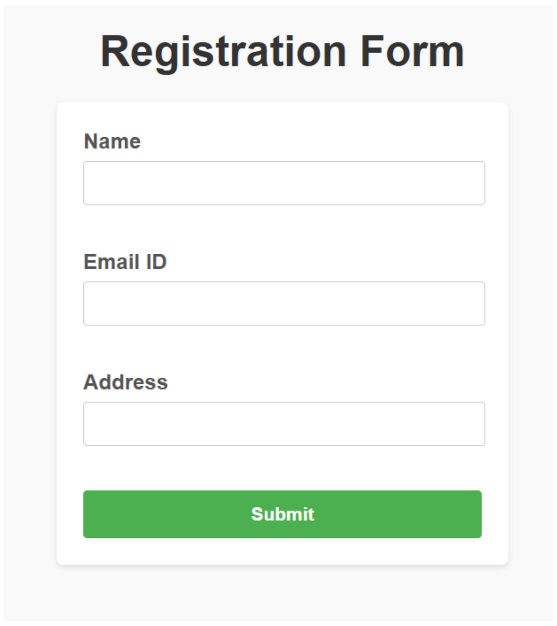
1. **Create a CSS file** named Style.css.

Style.css

```
/* Body styling */
body {
   font-family: Arial, sans-serif;
   background-color: #f9f9f9;
   display: flex;
   flex-direction: column;
   align-items: center;
   justify-content: center;
   min-height: 100vh;
   margin: 0;
   padding: 0;
}

/* Form container styling */
form {
   background-color: #ffffff;
   padding: 20px;
   border-radius: 5px;
   box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
   max-width: 300px;
   width: 100%;
}

/* Headings */
h1 {
   color: #333;
   text-align: center;
   margin-bottom: 20px;
}

/* Labels and input fields */
label {
   display: block;
   margin-bottom: 5px;
   font-weight: bold;
```

```
    color: #555;
}

input[type="text"],
input[type="email"] {
    width: 95%;
    padding: 8px;
    margin-bottom: 15px;
    border: 1px solid #ccc;
    border-radius: 3px;
    font-size: 14px;
}

/* Submit button */
input[type="submit"] {
    width: 100%;
    padding: 10px;
    background-color: #4CAF50;
    border: none;
    border-radius: 3px;
    color: #ffffff;
    font-weight: bold;
    cursor: pointer;
    font-size: 14px;
}
```

Output:



5. Demonstrate While Loop and For Loop using TypeScript.

Step 1: Install Node.js

TypeScript is built on Node.js, so you need to have it installed on your computer.

1. **Download Node.js**:
   - Go to the Node.js website.
   - Download the LTS (Long Term Support) version for your operating system.
2. **Install Node.js**:
   - Run the installer and follow the on-screen instructions.
   - During installation, make sure to check the box that says "Add to PATH" to ensure Node.js is accessible from the command line.
3. **Verify Installation**:
   - Open your terminal or command prompt and run:

```
node -v
npm -v
```

   - You should see the versions of Node.js and npm (Node Package Manager) displayed, indicating they were installed successfully.

Step 2: Install TypeScript

Now that you have Node.js installed, you can install TypeScript globally.

1. **Open the Terminal**:
   - You can use the integrated terminal in VS Code or any command line interface (Command Prompt, PowerShell, Terminal).
2. **Run the following command**:

```
npm install -g typescript
```

   - The -g flag installs TypeScript globally, making it available in any project on your machine.
3. **Verify TypeScript Installation**:
   - After installation, you can verify that TypeScript is installed by running:

```
tsc -v
```

- You should see the version of TypeScript displayed.

Step 3: Set Up Visual Studio Code

1. **Open VS Code**.
2. **Create a New Folder**:
   - Create a new folder for your TypeScript projects (e.g., typescript-loops).
   - Open this folder in VS Code by selecting File → Open Folder....
3. **Create TypeScript Files**:
   - In the Explorer panel on the left, create two new files:
     - ❖ whileLoop.ts
     - ❖ forLoop.ts

Step 4: Write the TypeScript Code

1. For **whileLoop.ts**, add the following code:

```
let count: number = 1;

while (count <= 5) {
  console.log(`While Loop Count: ${count}`);
  count++;
}
```

2. For **forLoop.ts**, add the following code:

```
for (let i: number = 1; i <= 5; i++) {
  console.log(`For Loop Count: ${i}`);
}
```

Step 5: Compile TypeScript to JavaScript

TypeScript needs to be compiled to JavaScript before it can be run. You can do this using the TypeScript Compiler (tsc).

1. **Open the Terminal in VS Code**:
   - You can do this by clicking on View → Terminal.

2. **Compile the TypeScript Files**:
   - To compile both files, run:

     ```
     tsc whileLoop.ts forLoop.ts
     ```

   - This command will generate two JavaScript files: whileLoop.js and forLoop.js.

Step 6: Run the Compiled JavaScript Files

You can run the generated JavaScript files using Node.js.

1. **Run whileLoop.js**:

   ```
   node whileLoop.js
   ```

2. **Run forLoop.js**:

   ```
   node forLoop.js
   ```

6. Create a single page application (Registration form) using React JS.

Step 1: Set Up Your React Project

1. **Create a new React project** by running:
   - **npx create-react-app registration-form**
2. **Navigate to the project directory**:
   - cd registration-form

Step 2: Create the Signup.js Component

1. In the src folder, create a file named Signup.js.

<u>Signup.js</u>

```
import React, { useState } from 'react';

export default function Signup() {
  const [name, setName] = useState("");
  const [id, setId] = useState("");
  const [password, setPassword] = useState("");
  const [submit, setSubmit] = useState(false);

  function handleName(e) {
    setName(e.target.value);
  }

  function handleId(e) {
    setId(e.target.value);
  }

  function handlePassword(e) {
    setPassword(e.target.value);
  }

  function handleSubmit(e) {
    e.preventDefault();
    if (name === '' || id === '' || password === '') {
      alert("Please enter all the fields");
    } else {
      setSubmit(true);
      clearAll();
    }
  }
```

```
    function clearAll() {
      setName("");
      setId("");
      setPassword("");
    }

    function successMessage() {
      if (submit) {
        return (
          <div>
            <p id="p">User signed up successfully</p>
          </div>
        );
      }
    }

    return (
      <div>
        <div className="form">
          <form onSubmit={handleSubmit}>
            <div id="d1">
              <p>Signup</p>
            </div>

            <label htmlFor="UserName">User Name</label>
              <input type="text" id="name" name="UserName" value={name}
              onChange={handleName} />

            <label htmlFor="EmailId">Email Id</label>
              <input type="email" id="id" name="EmailId" value={id}
             onChange={handleId} />

            <label htmlFor="Password">Password</label>
              <input type="password" id="psw" name="Password" value={password}
              onChange={handlePassword} />

            <button type="submit" id="btn">Create Account</button>
          </form>
        </div>
        <div id="success">
          {successMessage()}
        </div>
      </div>
    );
  }
```

Step 3: Modify App.js to Display the Signup Component

1. In the src folder, open or create App.js.
2. Import the Signup component, and use it in the App function.

App.js

```
import React from 'react';
import './App.css';
import Signup from './Signup';

export default function App() {
   return (
      <div>
         <Signup />
      </div>
   );
}
```

Step 4: Add Styling in App.css

1. In the src folder, open or create the App.css file.

App.css

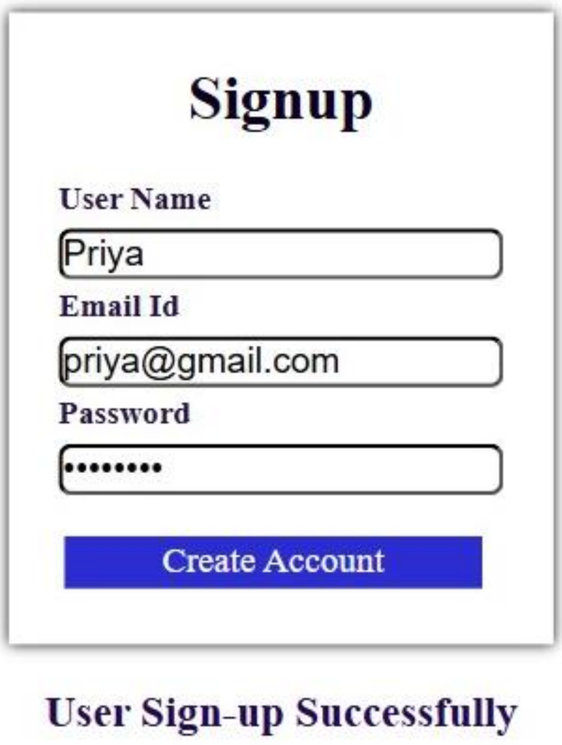
```
body {
   padding: 0;
   margin: 0;
   background-color: white;
}

.form {
   padding: 16px;
   margin: 16px;
   border-style: solid;
   border-color: white;
   background-color: white;
   box-shadow: 0px 0px 5px 0px black;
}

label {
   display: block;
   font-family: 'Times New Roman', Times, serif;
   color: rgb(35, 18, 65);
   font-size: 12px;
   font-weight: 550;
```

```
    line-height: 20px;
  }

  input {
    border-color: black;
    border-radius: 5px;
    font-family: 'Times New Roman', Times, serif;
    background-color: rgb(244, 240, 240);
  }

  #btn {
    margin-top: 15px;
    color: white;
    border-style: double;
    border-color: #2b2dd1;
    background-color: #2b2dd1;
    font-family: 'Franklin Gothic Medium', 'Arial Narrow', Arial, sans-serif;
    padding: 10px 20px;
    cursor: pointer;
  }

  div #d1 {
    text-align: center;
    color: #080939;
    font-family: 'Times New Roman', Times, serif;
    font-weight: 850;
    margin-bottom: 15px;
    font-size: 23px;
  }

  div #success p {
    text-align: center;
    padding-right: 0px;
    color: rgb(34, 8, 86);
    font-family: 'Times New Roman', Times, serif;
    font-weight: 600;
  }
```

Step 5: Run the Application

1. In the terminal, run the following command to start the application:
   - **npm start**
2. Open your browser and go to http://localhost:3000 to view your register form.

Output:



Priya

priya@gmail.com

••••••••

User Sign-up Successfully

7. Create a form to add a new product detail to the product catalogue using React.

Step 1: Set Up Your React Project

1. Create a new React project by running:
   - **npx create-react-app product-catalog**
2. Navigate to the project directory:
   - **cd product-catalog**

Step 2: Create the ProductCatalog.js Component

1. In the src folder, create a file named ProductCatalog.js.

ProductCatalog.js

```
import React, { useState } from 'react';

export default function ProductCatalog() {
   const [pname, setPname] = useState("");
   const [pprice, setPprice] = useState("");
   const [pdescription, setPdescription] = useState("");
   const [submit, setSubmit] = useState(false);

   function handlePname(e) {
      setPname(e.target.value);
   }

   function handlePprice(e) {
      setPprice(e.target.value);
   }

   function handlePdescription(e) {
      setPdescription(e.target.value);
   }

   function handleSubmit(e) {
      e.preventDefault();
      console.log("Product Name:", pname);
      console.log("Product Price:", pprice);
      console.log("Product Description:", pdescription);
      console.log(`Product ${pname} added`);

      setSubmit(true);
      clearAll();
   }
```

```
    function clearAll() {
        setPname("");
        setPprice("");
        setPdescription("");
    }

    function successMessage() {
        if (submit) {
            return (
                <div>
                    <p id="p">Product added successfully!</p>
                </div>
            );
        }
    }

    return (
        <>
            <h1>Product Catalog</h1>
            <div className="form">
                <form onSubmit={handleSubmit}>
                    <label htmlFor="ProductName">Product Name:</label>
                    <input type="text" id="name" name="ProductName" value={pname}
                    onChange={handlePname} required /><br />

                    <label htmlFor="ProductPrice">Product Price:</label>
                    <input type="number" id="price" name="ProductPrice" value={pprice}
                    onChange={handlePprice} required /><br />

                    <label htmlFor="ProductDescription">Product Description:</label>
                    <input type="text" id="description" name="ProductDescription"
                    value={pdescription} onChange={handlePdescription} required /><br />

                    <button id="btn" type="submit">Add Product</button>
                </form>
            </div>

            <div id="d1">
                {successMessage()}
            </div>
        </>
    );
}
```

Step 3: Modify App.js to Display the Product Catalog Form

1. In the src folder, open or create App.js.
2. Import the ProductCatalog component, and use it in the App function.

App.js

```
import React from 'react';
import './App.css';
import ProductCatalog from './ProductCatalog';

export default function App() {
   return (
      <div>
         <ProductCatalog />
      </div>
   );
}
```

Step 4: Add Styling in App.css

1. In the src folder, open or create the App.css file.

App.css

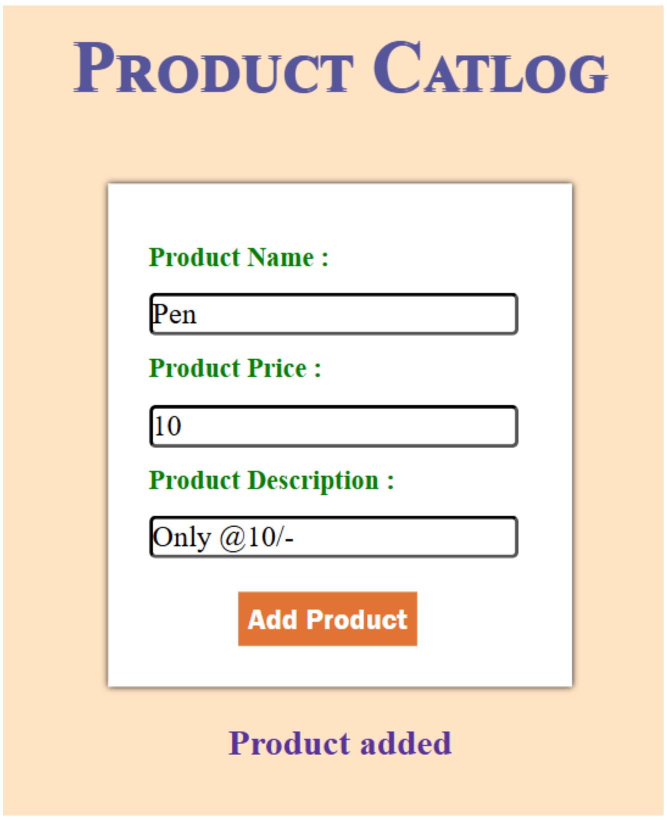
```
body {
   padding: 0;
   margin: 0;
   background-color: bisque;
}

h1 {
   text-align: center;
   font-variant: small-caps;
   text-shadow: 0px 1px;
   color: rgb(85, 85, 156);
   font-family: 'Times New Roman', Times, serif;
}

.form {
   padding: 16px;
   margin: 16px;
   border-style: solid;
   border-color: white;
   background-color: white;
   box-shadow: 0px 0px 3px 0px black;
}
```

```
label {
   display: block;
   font-family: 'Times New Roman', Times, serif;
   color: green;
   font-size: 12px;
   font-weight: 550;
   line-height: 30px;
}

input {
   border-color: black;
   border-radius: 3px;
   background-color: rgb(244, 240, 240);
   font-family: 'Times New Roman', Times, serif;
   padding: 5px;
   margin-bottom: 10px;
}

button {
   padding: 3px 20px;
   margin-top: 15px;
   color: white;
   border-style: double;
   border-color: #e17435;
   background-color: #e17435;
   font-family: 'Franklin Gothic Medium', 'Arial Narrow', Arial, sans-serif;
   cursor: pointer;
}

div #d1 {
   text-align: center;
   color: rgb(85, 50, 156);
   font-family: 'Times New Roman', Times, serif;
   font-weight: 750;
   margin-top: 15px;
}
```

Step 5: Run the Application

1. In the terminal, run the following command to start the application:
   - **npm start**
2. Open your browser and go to http://localhost:3000 to view your product catalog form.

Output:


PRODUCT CATLOG

Product Name :

Pen

Product Price :

10

Product Description :

Only @10/-

Product added

8. Implement programmatical navigation between different components using react Router.

Step 1: Install React Router

If you haven't already, install React Router by running the following command in your project directory:

- **npm install react-router-dom**

Step 2: Create Component Files

1. In your project's **src** directory, create three files for the components: Home.js, About.js, and Contact.js.

Home.js

```
import React from 'react';
function Home () {
  return (
    <div>
      <h2>Home Page</h2>
      <p>Welcome to the Home Page! </p>
    </div>
  );
}
  export default Home;
```

About.js

```
import React from 'react';
function About () {
  return (
    <div>
      <h2>About Page</h2>
      <p>This is the About Page. </p>
    </div>
  );
}
export default About;
```

Contact.js

```
import React from 'react';
function Contact () {
  return (
    <div>
```

```
      <h2>Contact Page</h2>
      <p>Contact us at example@example.com. </p>
    </div>
  );
}
export default Contact;
```

Step 3: Create the App.js File

1. In the **src** folder, create or edit App.js.
2. Import React, BrowserRouter, Route, Routes, and Link from react-router-dom, along with your component files.
3. Set up the router structure and navigation links in App.js as shown below:

App.js

```
import React from 'react';
import { BrowserRouter as Router, Route, Routes, Link } from 'react-router-dom';
import './App.css';
import Home from './Home';
import About from './About';
import Contact from './Contact';

function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li><Link to="/">Home</Link></li>
            <li><Link to="/About">About</Link></li>
            <li><Link to="/Contact">Contact</Link></li>
          </ul>
        </nav>

        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/About" element={<About />} />
          <Route path="/Contact" element={<Contact />} />
        </Routes>
      </div>
    </Router>
  );
}
export default App;
```

Step 4: Set Up Styling in App.css

1. In the **src** folder, create or edit the App.css file.

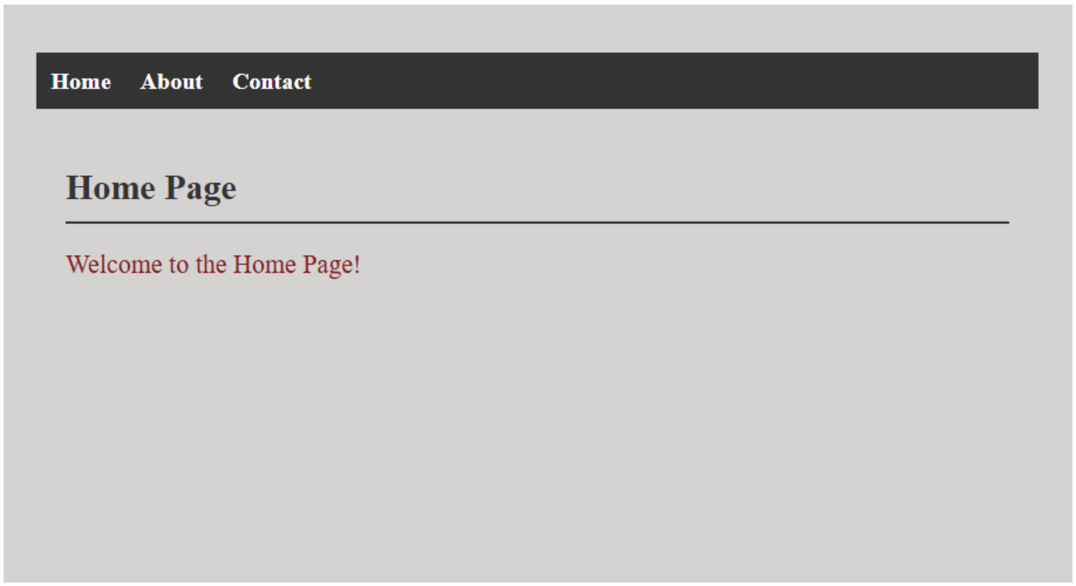
App.css

```
/* General styles */
body {
  font-family: 'Times New Roman', Times, serif;
  margin: 0;
  padding: 0;
  background-color: #d5d2d2;
}

h2 {
  color: #333;
}

p {
  font-size: 1.1em;
  text-align: left;
  color: #831e1e;
}

/* Navigation styles */
nav {
  background-color: #333;
  padding: 10px;
}

nav ul {
  list-style: none;
  padding: 0;
  margin: 0;
  display: flex;
}

nav ul li {
  margin-right: 20px;
}

nav ul li a {
  color: white;
  text-decoration: none;
  font-weight: bold;
}

nav ul li a:hover {
  text-decoration: underline;
}

/* Page content styles */
div {
  padding: 20px;
}

div h2 {
  border-bottom: 2px solid #333;
  padding-bottom: 10px;
  margin-bottom: 10px;
}

div p {
  line-height: 1.6;
}
```

Step 5: Run the Application

1. In the terminal, run the following command to start the development server:
   - **npm start**

<u>Output:</u>



Home Page

Welcome to the Home Page!

9. Create REST controller for CRUD operations using MySQL.

Step 1: Project Setup

1. **Initialize a Spring Boot Project**:

   - Go to Spring Initializr and create a new Maven project with the following dependencies:

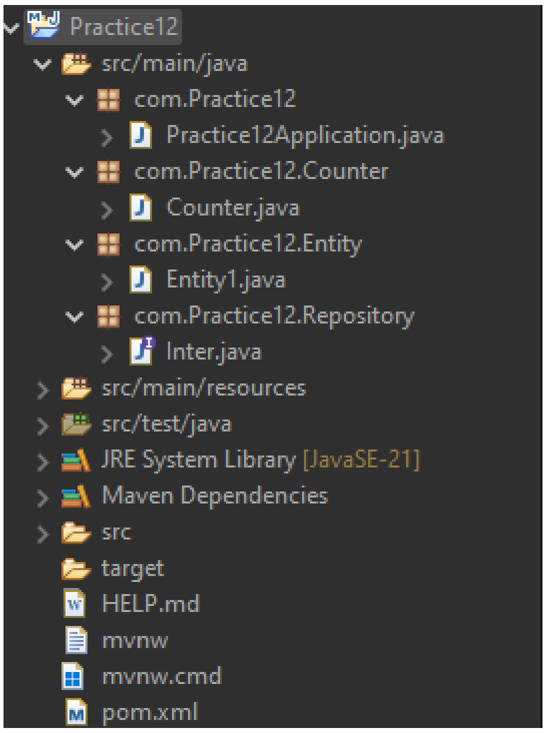
     ❖ Spring Web

     ❖ Spring Data JPA

     ❖ MySQL Driver

2. **Download and Import** the project into your preferred IDE.

Folder Structure:



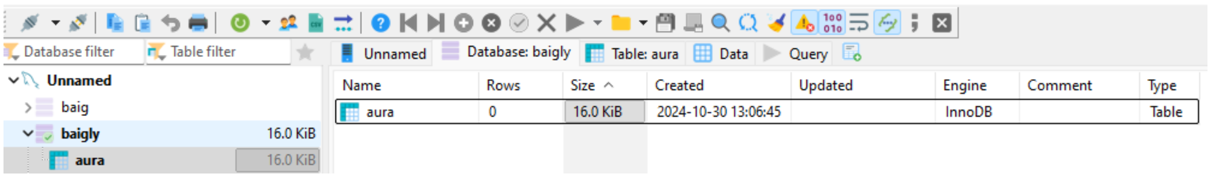
```
Practice12
  src/main/java
    com.Practice12
      Practice12Application.java
    com.Practice12.Counter
      Counter.java
    com.Practice12.Entity
      Entity1.java
    com.Practice12.Repository
      Inter.java
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-21]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

Step 2: Set Up MySQL Database or HeidiSQL

1. **Create the Database**:

   - Open MySQL and create a database named baigly or Anything else:

     ❖ CREATE DATABASE baigly;



Name	Rows	Size	Created	Updated	Engine	Comment	Type
aura	0	16.0 KiB	2024-10-30 13:06:45		InnoDB		Table

2. **Configure application.**:

   - Go to src/main/resources/application. and set the following to connect to MySQL:
     - ❖ spring.application.name=Practice12
     - ❖ spring.datasource.url=jdbc:mysql://localhost:3306/baigly
     - ❖ spring.datasource.username=root
     - ❖ spring.datasource.password=muzammilbaig@123
     - ❖ spring.jpa.hibernate.ddl-auto=update

Step 3: Create Entity Class

Create the entity class that maps to the database table. Here's the Entity1 class, which maps to the aura table in the baigly database

```
package com.Practice12.Entity;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name="aura")
public class Entity1 {
    @Id
    private int mid;
    private String mname;
    private String description;

    // Getters and Setters
    public int getMid() {
        return mid;
    }
    public void setMid(int mid) {
        this.mid = mid;
    }
    public String getMname() {
        return mname;
    }
    public void setMname(String mname) {
        this.mname = mname;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
}
```

Note: Ensure the description field is lowercase in the entity to avoid issues with case sensitivity.

Step 4: Create Repository Interface

Define the repository interface that will extend JpaRepository to provide CRUD operations.

```
package com.Practice12.Repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.Practice12.Entity.Entity1;

public interface Inter extends JpaRepository<Entity1, Integer>
{
}
```

Step 5: Create the REST Controller

The controller manages HTTP requests for CRUD operations.

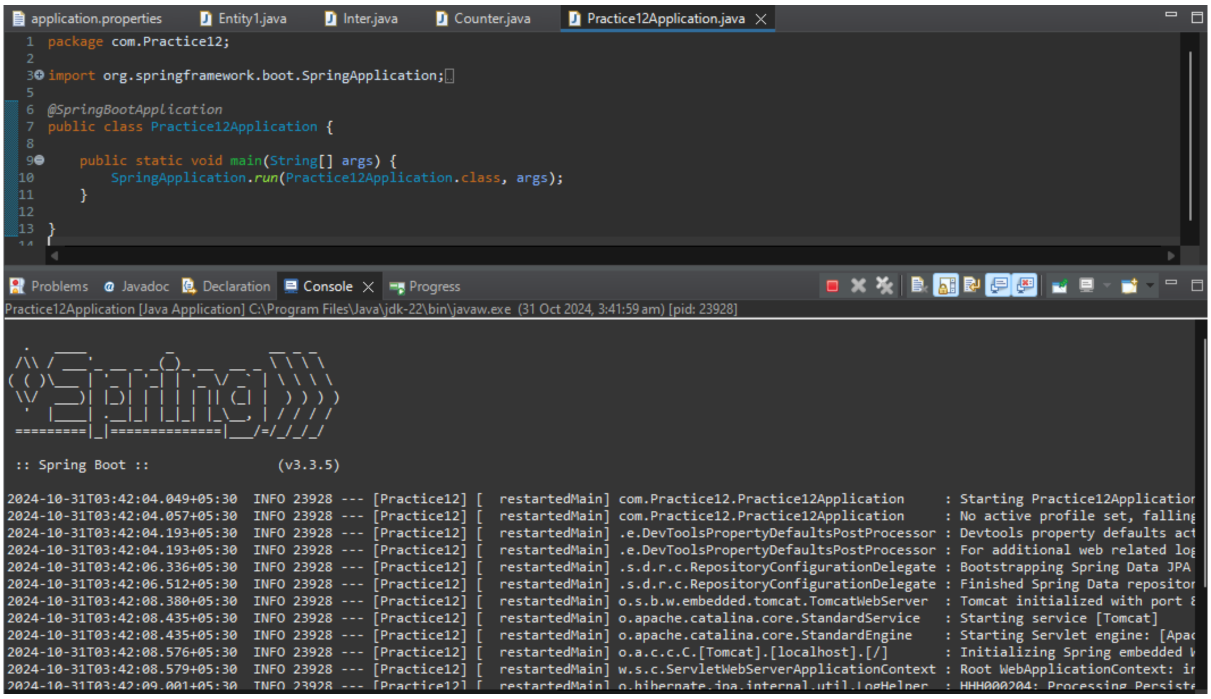
```
package com.Practice12.Counter;

import .util.List;
import .util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import com.Practice12.Entity.Entity1;
import com.Practice12.Repository.Inter;

@RestController
@RequestMapping("/api") // Optional prefix to group routes under /api
public class Counter {

    @Autowired
    private Inter obj;

    @GetMapping("/get")
    public List<Entity1> getAllEntities() {
        return obj.findAll();
    }

    @PostMapping("/post")
    public String addEntity(@RequestBody Entity1 e) {
        obj.save(e);
        return "Inserted";
    }

    @PutMapping("/put/{id}")
    public String updateEntity(@PathVariable int id, @RequestBody Entity1 d) {
        Optional<Entity1> optionalEntity = obj.findById(id);
        if (optionalEntity.isPresent()) {
            Entity1 ex = optionalEntity.get();
            ex.setMname(d.getMname());
            ex.setDescription(d.getDescription());
            obj.save(ex);
            return "Updated";
        }
        return "ID Not Found";
    }

    @DeleteMapping("/del/{id}")
    public String deleteEntity(@PathVariable int id) {
        if (obj.existsById(id)) {
            obj.deleteById(id);
            return "Deleted";
        }
        return "ID Not Found";
    }
}
```

Main/Default Method:

```
package com.Practice12;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Practice12Application {

        public static void main(String[] args) {
                SpringApplication.run(Practice12Application.class, args);
        }

}
```

Step 6: Run the Application

1. **Start MySQL**: Make sure your MySQL server is running.
2. **Run the Application**: In your IDE, run the main application class (e.g., Practice12Application.).



```java
package com.Practice12;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class Practice12Application {

    public static void main(String[] args) {
        SpringApplication.run(Practice12Application.class, args);
    }

}
```

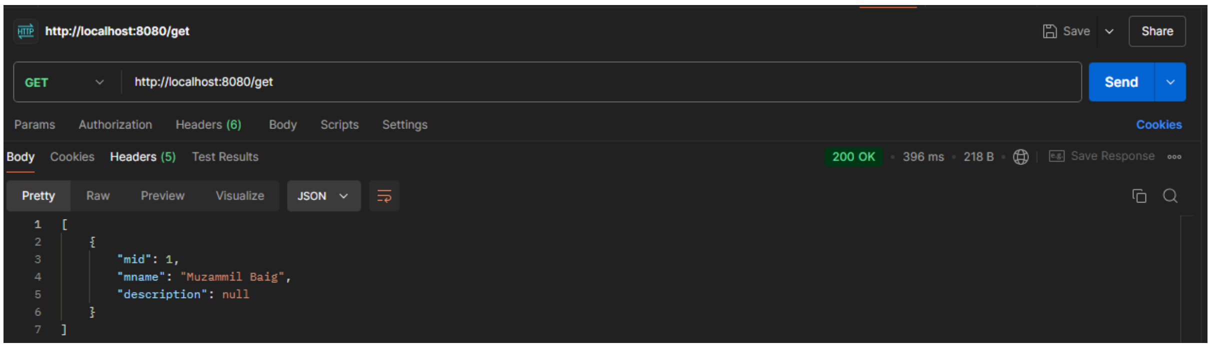
```
Practice12Application [Java Application] C:\Program Files\Java\jdk-22\bin\javaw.exe  (31 Oct 2024, 3:41:59 am) [pid: 23928]

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::                (v3.3.5)

2024-10-31T03:42:04.049+05:30  INFO 23928 --- [Practice12] [  restartedMain] com.Practice12.Practice12Application     : Starting Practice12Applicatior
2024-10-31T03:42:04.057+05:30  INFO 23928 --- [Practice12] [  restartedMain] com.Practice12.Practice12Application     : No active profile set, falling
2024-10-31T03:42:04.193+05:30  INFO 23928 --- [Practice12] [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults act
2024-10-31T03:42:04.193+05:30  INFO 23928 --- [Practice12] [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related log
2024-10-31T03:42:06.336+05:30  INFO 23928 --- [Practice12] [  restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA
2024-10-31T03:42:06.512+05:30  INFO 23928 --- [Practice12] [  restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repositor
2024-10-31T03:42:08.380+05:30  INFO 23928 --- [Practice12] [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8
2024-10-31T03:42:08.435+05:30  INFO 23928 --- [Practice12] [  restartedMain] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2024-10-31T03:42:08.435+05:30  INFO 23928 --- [Practice12] [  restartedMain] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apac
2024-10-31T03:42:08.576+05:30  INFO 23928 --- [Practice12] [  restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded W
2024-10-31T03:42:08.579+05:30  INFO 23928 --- [Practice12] [  restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: in
2024-10-31T03:42:09.001+05:30  INFO 23928 --- [Practice12] [  restartedMain] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing Persiste
```

Step 7: Test CRUD Operations Using Postman

Use the following endpoints in Postman to test the API. Ensure Content-Type is set to application/ for the POST and PUT requests.

1. Get All Entities

- **Method**: GET
- **URL**: http://localhost:8080/api/get



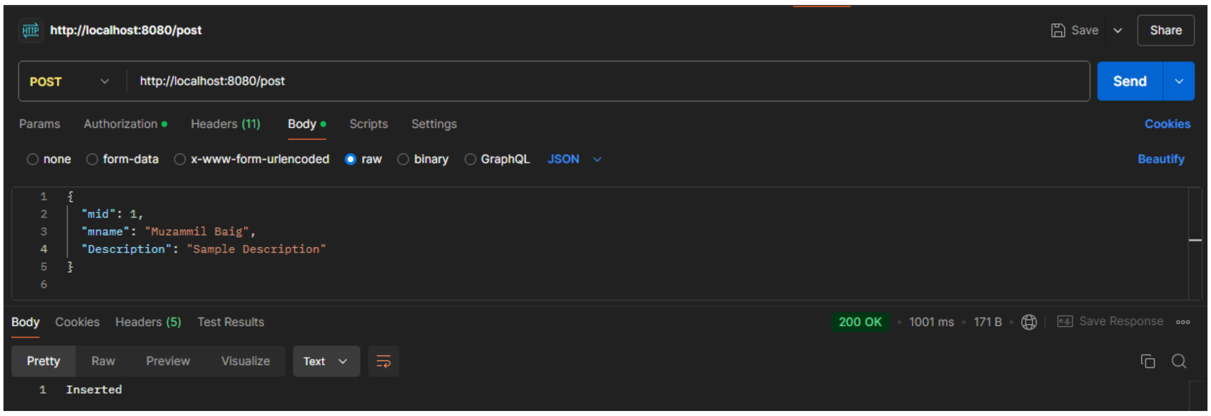
```json
[
    {
        "mid": 1,
        "mname": "Muzammil Baig",
        "description": null
    }
]
```

2. Add a New Entity

- **Method**: POST
- **URL**: http://localhost:8080/api/post

- **Body-raw**:

```
{
 "mid": 1,
 "mname": "Sample Name",
 "description": "Sample Description"
}
```



http://localhost:8080/post

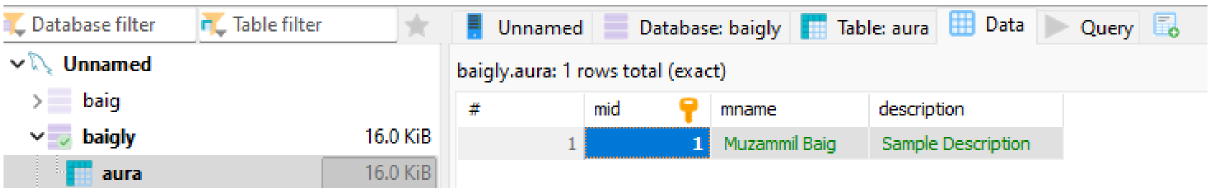
POST http://localhost:8080/post

```
{
  "mid": 1,
  "mname": "Muzammil Baig",
  "Description": "Sample Description"
}
```

200 OK · 1001 ms · 171 B

```
Inserted
```

Check in Database:



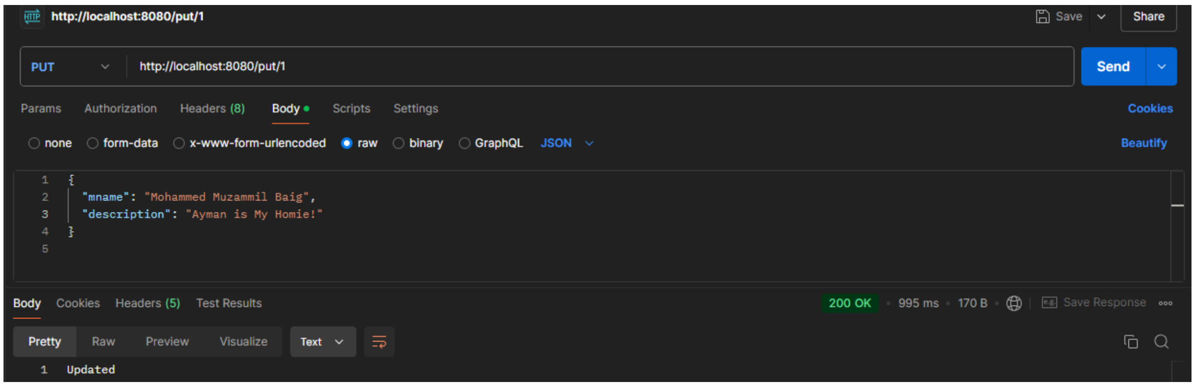
baigly.aura: 1 rows total (exact)

#	mid	mname	description
1	1	Muzammil Baig	Sample Description

3. Update an Existing Entity

- **Method**: PUT
- **URL**: http://localhost:8080/api/put/{id}
- Replace {id} with the mid value of the entity you want to update.
- **Body**:

```
{
 "mname": "Updated Name",
 "description": "Updated Description"
}
```



http://localhost:8080/put/1

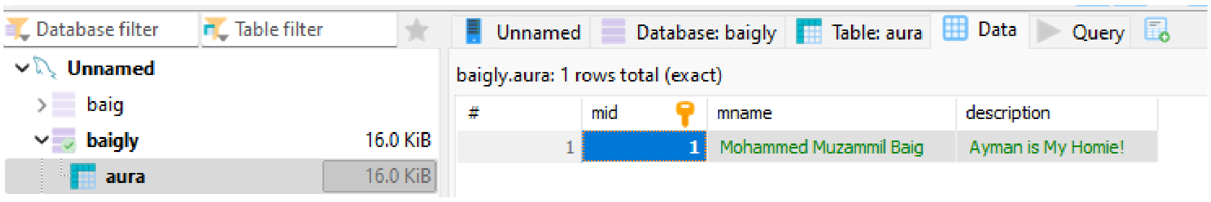
PUT http://localhost:8080/put/1

```
{
  "mname": "Mohammed Muzammil Baig",
  "description": "Ayman is My Homie!"
}
```

200 OK · 995 ms · 170 B

```
Updated
```

Check in Database:

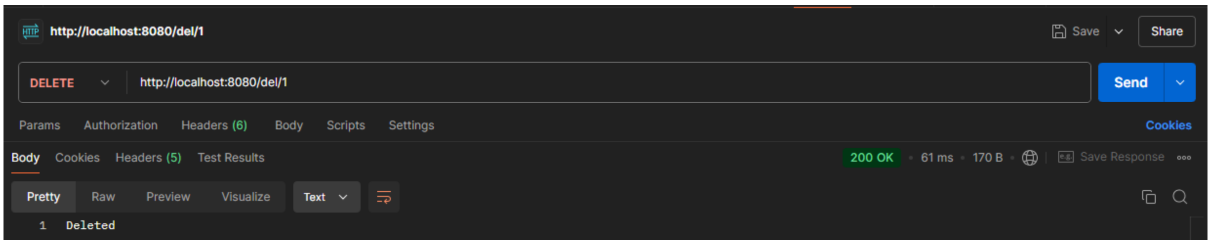


baigly.aura: 1 rows total (exact)

#	mid	mname	description
1	1	Mohammed Muzammil Baig	Ayman is My Homie!

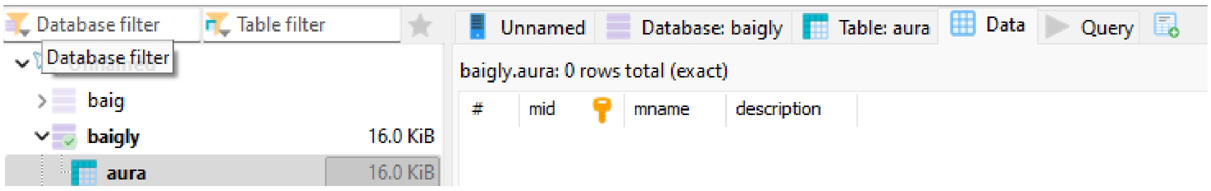
4. Delete an Entity

- **Method**: DELETE
- **URL**: http://localhost:8080/api/del/{id}
- Replace {id} with the mid value of the entity you want to delete.



```
1  Deleted
```

Databases:



baigly.aura: 0 rows total (exact)

#	mid	mname	description

NOTE: The Name Used in The Above Program are Random. If You Want You can Change the Name(e.g. Database_Name, Entity_Name, name, Description, Methods Name etc).

10. CRUD Operations on document using MongoDB.

1. Create and Drop Database

1.1 Create Database

MongoDB creates a database when you first **use** it and insert data into it. If the database doesn't exist, MongoDB will create it once data is inserted.

- ***use myDatabase*** **// Switch to or create 'myDatabase'**

<u>NOTE: MongoDB will only create the myDatabase once you insert a document into a collection in this database.</u>

1.2 Drop Database

To delete a database, use the dropDatabase() command:

- ***db.dropDatabase();***

<u>This will drop the current database you are connected to.</u>

---

2. Create and Drop Collections

2.1 Create Collection

MongoDB will automatically create a collection when you insert a document into it, but you can also explicitly create a collection with the createCollection() command.

- ***db.createCollection("myCollection");***

<u>This creates a new collection named myCollection.</u>

2.2 Drop Collection

To delete a collection, use the drop() method on that collection.

- ***db.myCollection.drop();***

<u>This will drop the myCollection from the database.</u>

---

3. CRUD Operations on Documents

3.1 Create (Insert Document)

To insert a new document, use insertOne() for one document or insertMany() for multiple documents.

- **Insert One Document**:

```
db.myCollection.insertOne({
  "name": "John",
  "age": 30,
  "city": "New York" });
```

- **Insert Multiple Documents**:

```
db.myCollection.insertMany([
  { "name": "Jane", "age": 28, "city": "London" },
  { "name": "Mark", "age": 35, "city": "San Francisco" }
]);
```

3.2 Read (Find Document)

Use the find() method to retrieve documents from a collection.

- **Find All Documents**:

```
db.myCollection.find().pretty();  // pretty() makes the output more readable
```

- **Find Documents with a Query**:

```
db.myCollection.find({ "city": "New York" }).pretty();
```

3.3 Update (Modify Document)

To modify documents, you can use the updateOne(), updateMany(), or replaceOne() commands.

- **Update One Document**:

```
db.myCollection.updateOne(
  { "name": "John" },             // Query to find the document
  { "$set": { "age": 31 } }       // Update action
);
```

- **Update Multiple Documents**:

```
db.myCollection.updateMany(
  { "city": "New York" },            // Query to find matching documents
  { "$set": { "city": "NYC" } }       // Update action
);
```

3.4 Delete (Remove Document)

To delete documents, you can use deleteOne() or deleteMany().

- **Delete One Document**:

  db.myCollection.deleteOne({ "name": "John" });

- **Delete Multiple Documents**:

  db.myCollection.deleteMany({ "city": "NYC" });

11. Create REST Controller for CRUD Operations using MongoDB.

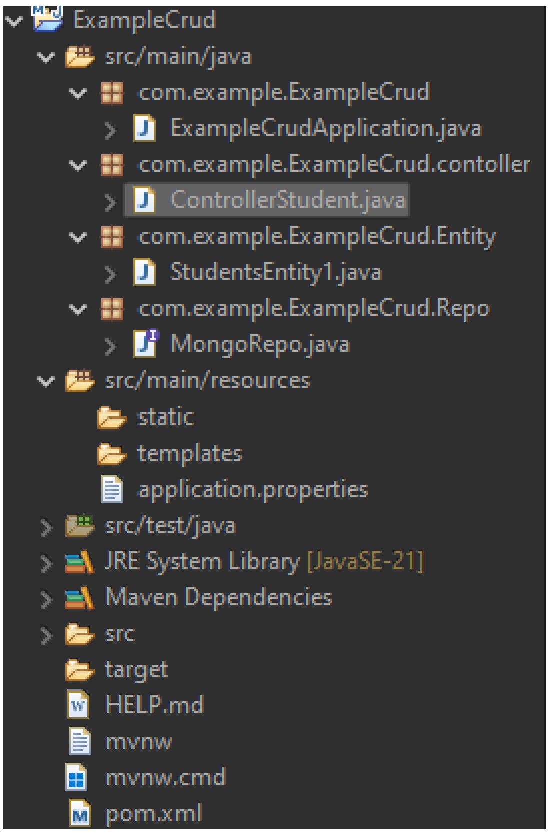
Prerequisites:

1. Install MongoDB (Ensure MongoDB is running on your system or in a Docker container).
2. Spring Boot Setup: Make sure you have spring-boot-starter-data-mongodb and spring-boot-starter-web dependencies in your project.

Step 1: Set Up Spring Boot Project

1. Go to Spring Initializr and set up a new project.
   - Project: Maven Project
   - Language:
   - Spring Boot: 3.0 or above
   - Dependencies: Spring Web, Spring Data MongoDB
2. Download and unzip the project. Open it in your preferred IDE (e.g., IntelliJ IDEA, Eclipse).

Folder Structure:



```
ExampleCrud
  src/main/java
    com.example.ExampleCrud
      ExampleCrudApplication.java
    com.example.ExampleCrud.contoller
      ControllerStudent.java
    com.example.ExampleCrud.Entity
      StudentsEntity1.java
    com.example.ExampleCrud.Repo
      MongoRepo.java
  src/main/resources
    static
    templates
    application.properties
  src/test/java
  JRE System Library [JavaSE-21]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

Step 2: Set Up MongoDB Configuration

In src/main/resources/application.properties, add the MongoDB configuration to connect to a local database:

- spring.application.name=ExampleCrud
- spring.data.mongodb.host=localhost
- spring.data.mongodb.port=27017
- spring.data.mongodb.database=ChildrenStudents

Ensure that MongoDB is running on the specified host and port.

Step 3: Define the Entity or DataClass(Because in MongoDB there is no Entity, its just the name to specify you can Change) Class

The entity class StudentsEntity1 should have getters, setters, and proper annotations. Make sure the fields match MongoDB standards. Your entity class should look like this:

```
package com.example.ExampleCrud.Entity;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Document(collection="CollectionStudents")
public class StudentsEntity1 {
    @Id
    private int srollno;
    private String sname;
    private int sage;
    private String address;
}
```

Step 4: Create the Repository Interface

Define a repository interface by extending MongoRepository. This will provide built-in methods for CRUD operations:

```
package com.example.ExampleCrud.Repo;

import org.springframework.data.mongodb.repository.MongoRepository;
import com.example.ExampleCrud.Entity.StudentsEntity1;

public interface MongoRepo extends MongoRepository<StudentsEntity1, Integer> {
}
```

Step 5: Create the REST Controller

The ControllerStudent class manages CRUD operations via endpoints.

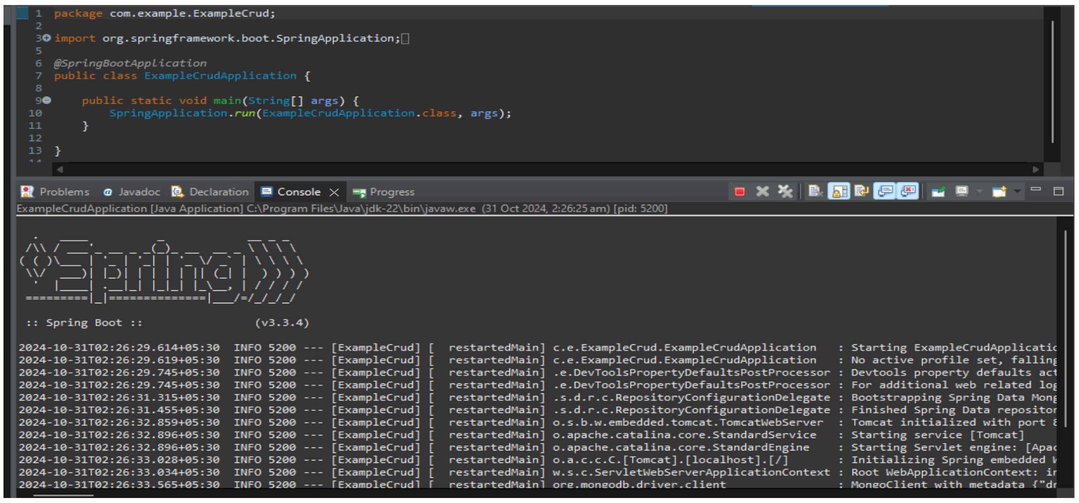
```
package com.example.ExampleCrud.controller;

import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import com.example.ExampleCrud.Entity.StudentsEntity1;
import com.example.ExampleCrud.Repo.MongoRepo;
@RestController
@RequestMapping("/students")
public class ControllerStudent {

    @Autowired
    private MongoRepo obj;

    @PostMapping("/add")
    public String addStudent(@RequestBody StudentsEntity1 e) {
        obj.save(e);
        return "Inserted Successfully";
    }

    @GetMapping("/getAll")
    public List<StudentsEntity1> getAllStudents() {
        return obj.findAll();
    }

    @GetMapping("/get/{id}")
    public StudentsEntity1 getStudentById(@PathVariable int id) {
        return obj.findById(id).orElse(null);
    }

    @DeleteMapping("/delete/{id}")
    public String deleteStudent(@PathVariable int id) {
        if (obj.existsById(id)) {
            obj.deleteById(id);
            return "Deleted Successfully";
        }
        return "ID Not Found!";
    }
    @PutMapping("/update/{id}")
    public String updateStudent(@PathVariable int id, @RequestBody StudentsEntity1 en) {
        Optional<StudentsEntity1> optionalStudent = obj.findById(id);
        if (optionalStudent.isPresent()) {
            StudentsEntity1 existingData = optionalStudent.get();
            existingData.setSname(en.getSname());
            existingData.setSage(en.getSage());
            existingData.setAddress(en.getAddress());
            obj.save(existingData);
            return "Updated Successfully";
        }
        return "ID Not Found!";
    }
}
```

Main/Default Functiom

```
package com.example.ExampleCrud;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ExampleCrudApplication {

        public static void main(String[] args) {
                SpringApplication.run(ExampleCrudApplication.class, args);
        }

}
```

Step 6: Run the Application

1. **Run MongoDB**: Make sure MongoDB is running on localhost:27017.
2. **Run the Application**: Start your Spring Boot application from the main class in the IDE (e.g., ExampleCrudApplication).



```
package com.example.ExampleCrud;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class ExampleCrudApplication {

    public static void main(String[] args) {
        SpringApplication.run(ExampleCrudApplication.class, args);
    }

}
```

```
ExampleCrudApplication [Java Application] C:\Program Files\Java\jdk-22\bin\javaw.exe  (31 Oct 2024, 2:26:25 am) [pid: 5200]

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::                (v3.3.4)

2024-10-31T02:26:29.614+05:30  INFO 5200 --- [ExampleCrud] [  restartedMain] c.e.ExampleCrud.ExampleCrudApplication   : Starting ExampleCrudApplicati
2024-10-31T02:26:29.619+05:30  INFO 5200 --- [ExampleCrud] [  restartedMain] c.e.ExampleCrud.ExampleCrudApplication   : No active profile set, falling
2024-10-31T02:26:29.745+05:30  INFO 5200 --- [ExampleCrud] [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults act
2024-10-31T02:26:29.745+05:30  INFO 5200 --- [ExampleCrud] [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related log
2024-10-31T02:26:31.315+05:30  INFO 5200 --- [ExampleCrud] [  restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data Mong
2024-10-31T02:26:31.455+05:30  INFO 5200 --- [ExampleCrud] [  restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repositor
2024-10-31T02:26:32.859+05:30  INFO 5200 --- [ExampleCrud] [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port 8
2024-10-31T02:26:32.896+05:30  INFO 5200 --- [ExampleCrud] [  restartedMain] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2024-10-31T02:26:32.896+05:30  INFO 5200 --- [ExampleCrud] [  restartedMain] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apac
2024-10-31T02:26:33.028+05:30  INFO 5200 --- [ExampleCrud] [  restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded W
2024-10-31T02:26:33.034+05:30  INFO 5200 --- [ExampleCrud] [  restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: in
2024-10-31T02:26:33.565+05:30  INFO 5200 --- [ExampleCrud] [  restartedMain] org.mongodb.driver.client                : MongoClient with metadata {"dr
```

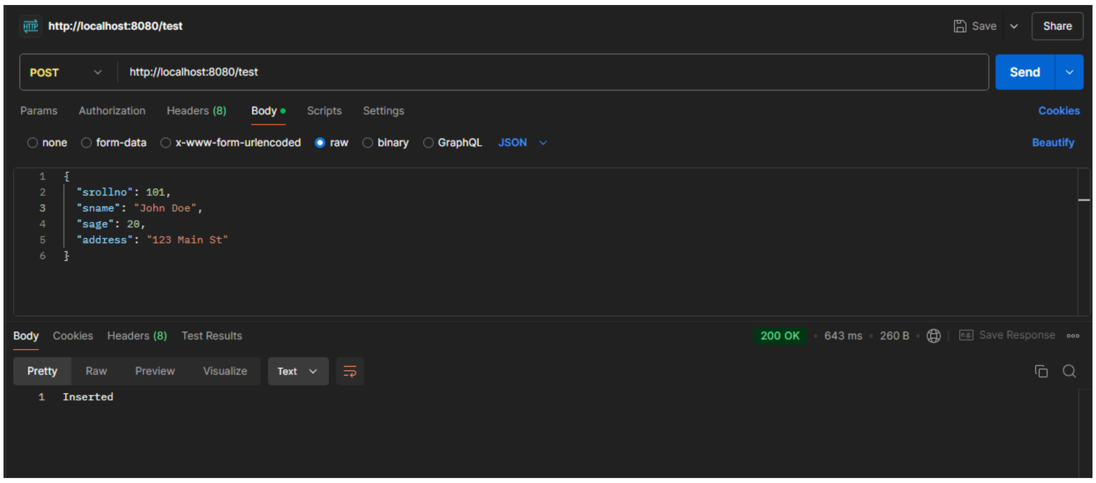
Step 7: Test CRUD Operations Using Postman

Use the following endpoints to perform CRUD operations. Make sure to set Content-Type to application/ in Postman.

1. Add a Student

- **Method**: POST
- **URL**: http://localhost:8080/students/add
- **Body-raw**:

```
{
 "srollno": 101,
 "sname": "John Doe",
 "sage": 20,
 "address": "123 Main St"
}
```



POST http://localhost:8080/test

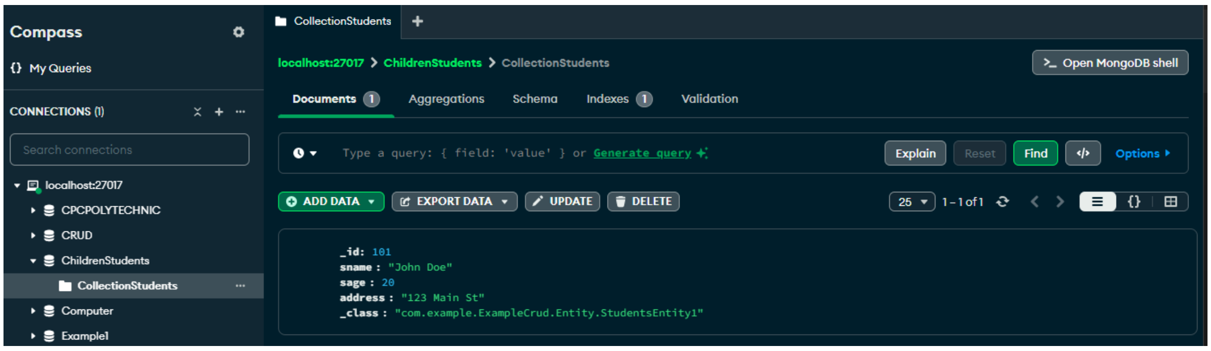
Body: raw, JSON

```
{
  "srollno": 101,
  "sname": "John Doe",
  "sage": 20,
  "address": "123 Main St"
}
```

200 OK · 643 ms · 260 B

```
Inserted
```

Check in Compass:

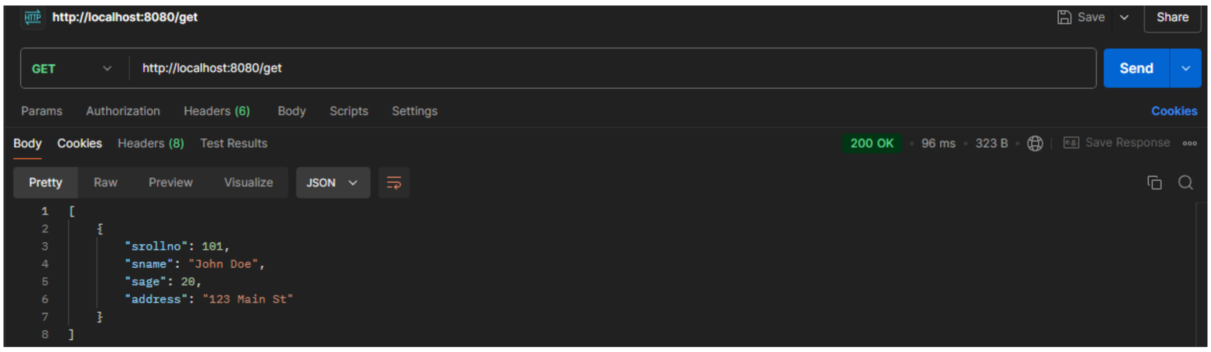


localhost:27017 > ChildrenStudents > CollectionStudents

```
_id: 101
sname : "John Doe"
sage : 20
address : "123 Main St"
_class : "com.example.ExampleCrud.Entity.StudentsEntity1"
```

2. Get All Students

- **Method**: GET
- **URL**: http://localhost:8080/students/getAll


http://localhost:8080/get

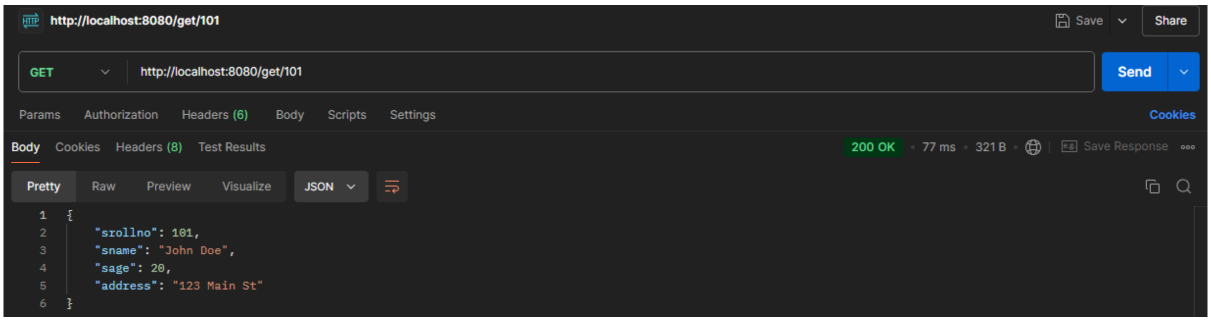
GET http://localhost:8080/get

200 OK · 96 ms · 323 B

```
[
    {
        "srollno": 101,
        "sname": "John Doe",
        "sage": 20,
        "address": "123 Main St"
    }
]
```

3. Get Student by ID

- **Method**: GET
- **URL**: http://localhost:8080/students/get/{id}
- Replace {id} with the student ID you want to retrieve (e.g., 101).


http://localhost:8080/get/101

GET http://localhost:8080/get/101

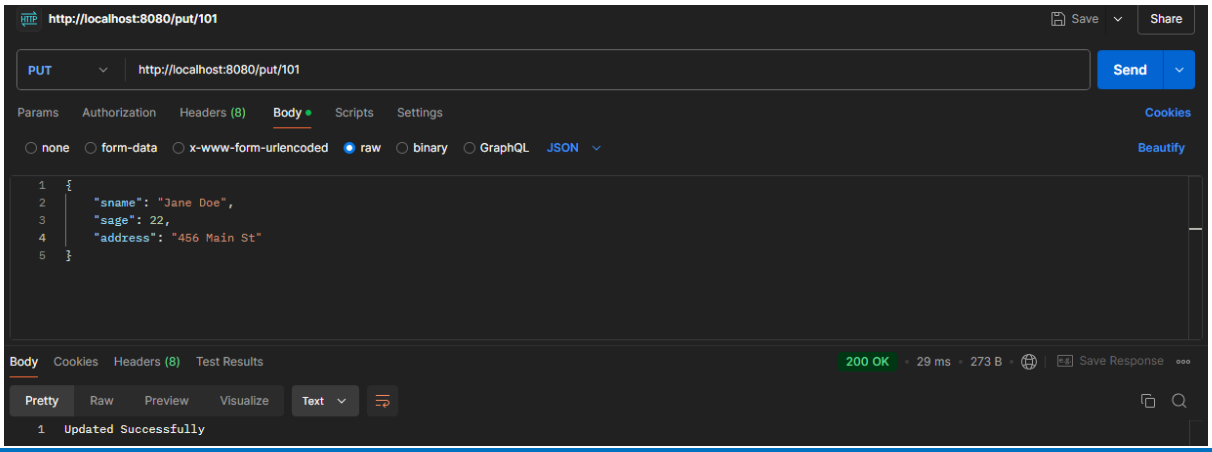
200 OK · 77 ms · 321 B

```
{
    "srollno": 101,
    "sname": "John Doe",
    "sage": 20,
    "address": "123 Main St"
}
```

4. Update a Student

- **Method**: PUT
- **URL**: http://localhost:8080/students/update/{id}
- Replace {id} with the student ID to update.
- **Body**:

```
{
  "sname": "Jane Doe",
  "sage": 22,
  "address": "456 Main St"
}
```


http://localhost:8080/put/101

PUT http://localhost:8080/put/101

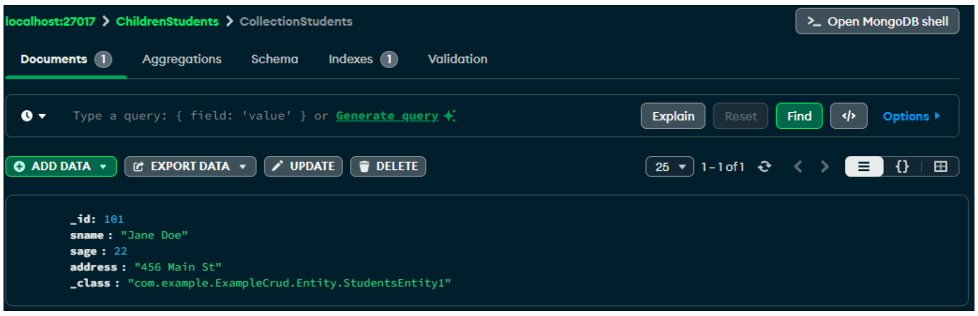
Body: raw JSON

```
{
    "sname": "Jane Doe",
    "sage": 22,
    "address": "456 Main St"
}
```

200 OK · 29 ms · 273 B

```
Updated Successfully
```

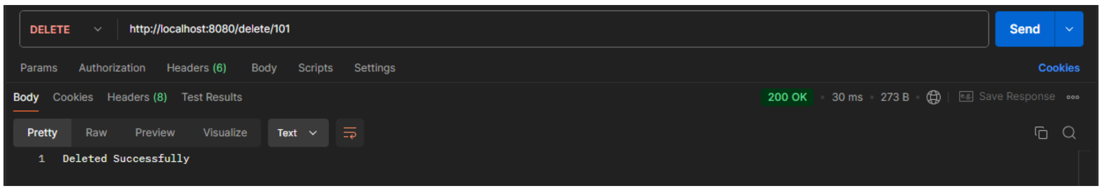
Check in Database:



```
_id: 101
sname : "Jane Doe"
sage : 22
address : "456 Main St"
_class : "com.example.ExampleCrud.Entity.StudentsEntity1"
```

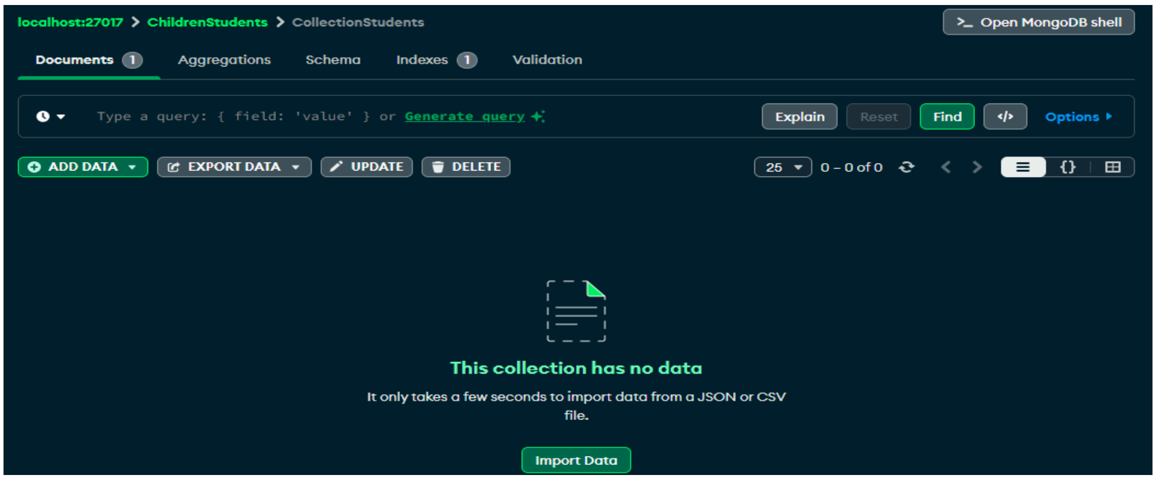
5. Delete a Student

- **Method**: DELETE
- **URL**: http://localhost:8080/students/delete/{id}
- Replace {id} with the student ID to delete.



```
Deleted Successfully
```

Check in Database:


localhost:27017 > ChildrenStudents > CollectionStudents
Open MongoDB shell
Documents 1
Aggregations
Schema
Indexes 1
Validation
Explain
Reset
Find
Options
ADD DATA
EXPORT DATA
UPDATE
DELETE
25
0 – 0 of 0
This collection has no data
It only takes a few seconds to import data from a JSON or CSV file.
Import Data