University of Houston-Victoria

Department of Computer and Information Science

Group Project

Expense Tracker

Deployment Manual

COSC 6342 Software Engineering Project Management

Spring 2024

By

Aradhana Sharma (2250955)

Kapil Suryawanshi (2039409)

Shayan Khan (2244554)

Suraj Odedra (2329291)

Advisor: Dr. Amjad Nusayr

Director of UG Computer Science & Computer Information Systems  
Associate Professor of Computer Science

University of Houston Victoria

Table of Contents

[Abstract 4](#_Toc158808060)

[Development Environmet 4](#_Toc158808061)

[Visual Studio Code 4](#_Toc158808062)

[GitHub 9](#_Toc158808063)

[Git 15](#_Toc158808064)

[Node.js 27](#_Toc158808065)

[MySQL 33](#_Toc158808066)

[Install Postman 46](#_Toc158808067)

[Get all users 49](#_Toc158808068)

[Get a user by id 49](#_Toc158808069)

[Create user 50](#_Toc158808070)

[Update user 50](#_Toc158808071)

[Delete user 51](#_Toc158808072)

[Setup Git <->GitHub communication 51](#_Toc158808073)

[Setup Front End app 61](#_Toc158808074)

[Setup Back End App 63](#_Toc158808075)

[Production Environment 65](#_Toc158808076)

[Archived 65](#_Toc158808077)

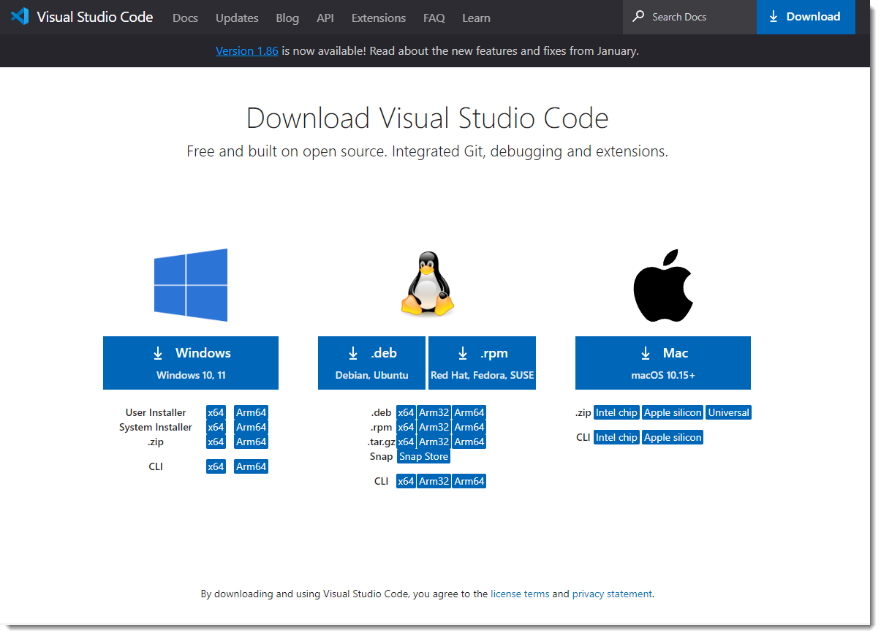
# Abstract

The document is intended to describe the setup of the software tools required for preparing the development environment. It also describes the deployment protocol to be followed to deploy the developed software.

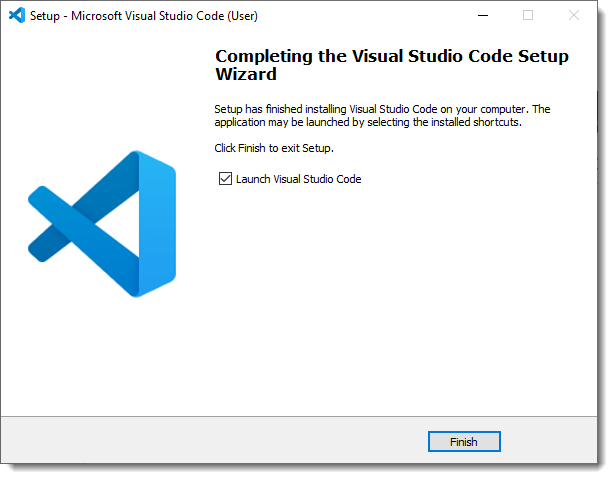
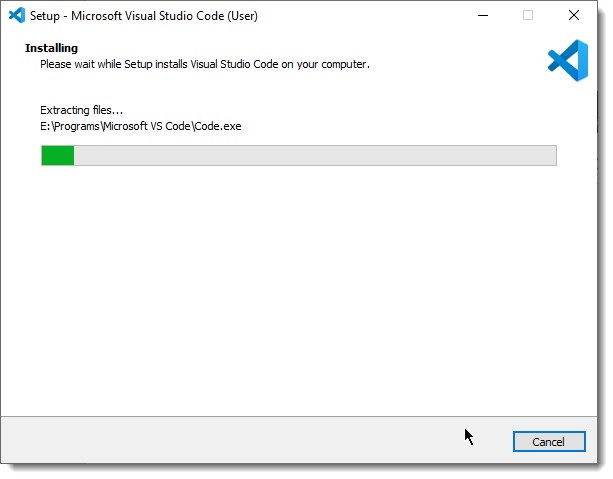
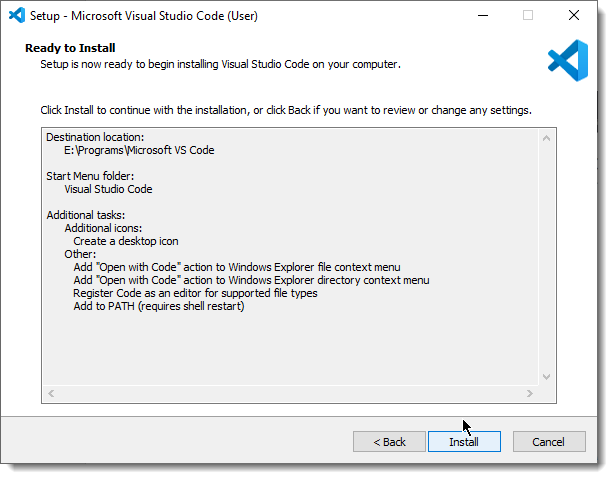
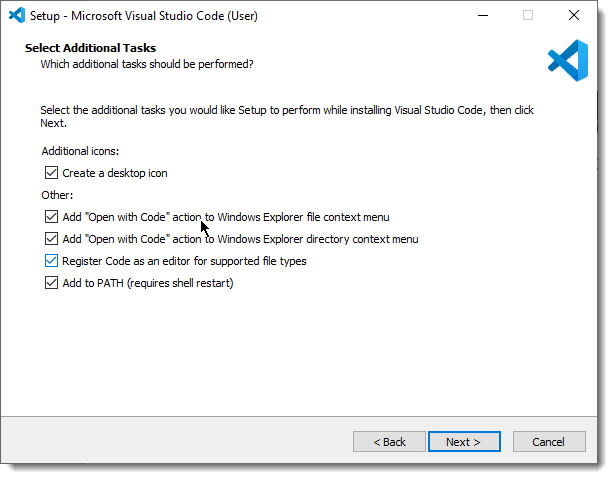
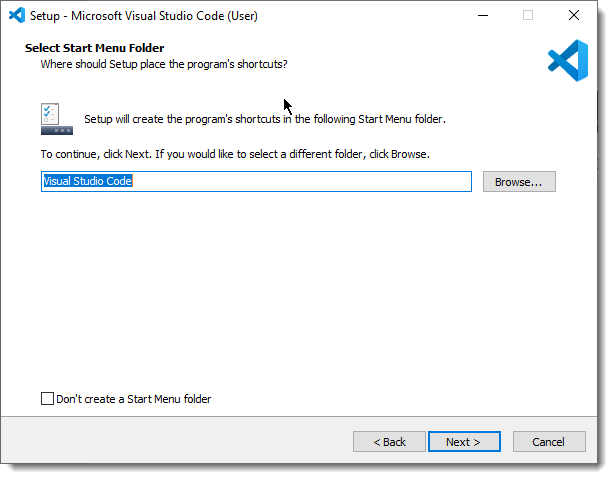
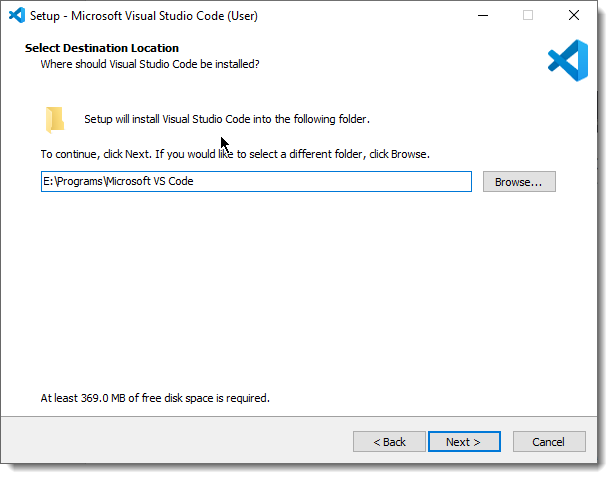
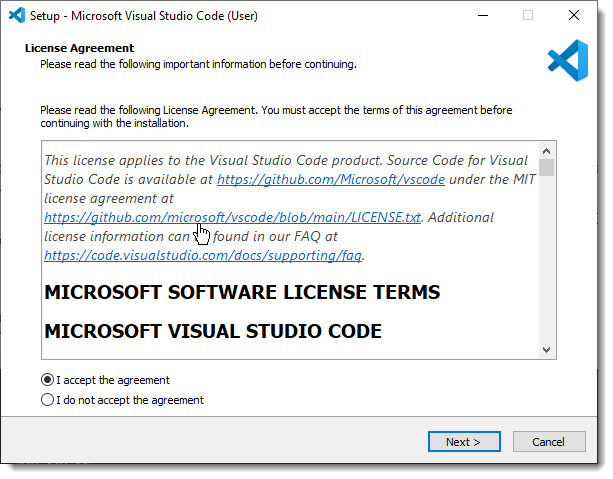
# Development Environmet

## Visual Studio Code

Navigate to <https://code.visualstudio.com/download>

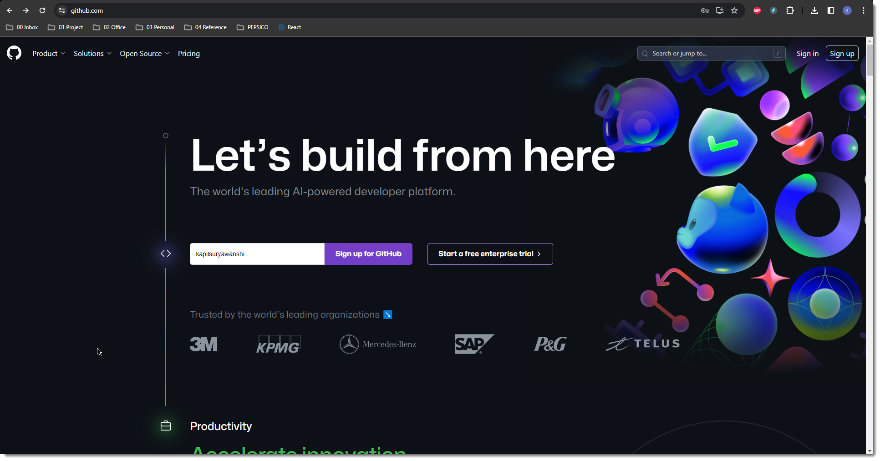


Download the required executable as per your operating system. Here I am downloading installer for Windows OS. Once downloaded double click the executable and start the installer. Following screens shows the options selected during a typical installation.

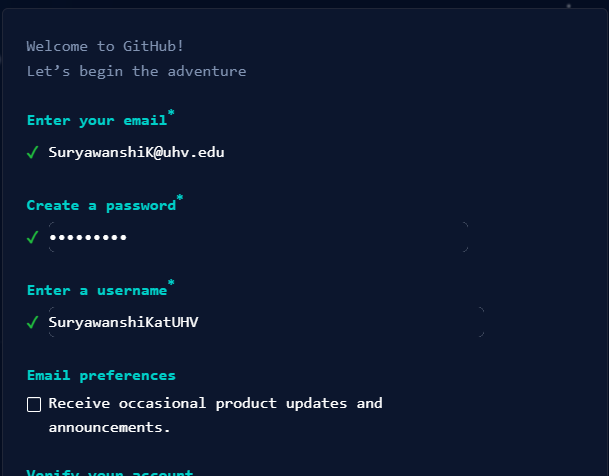


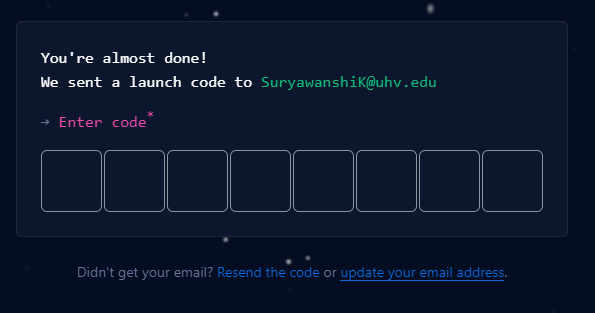
## GitHub

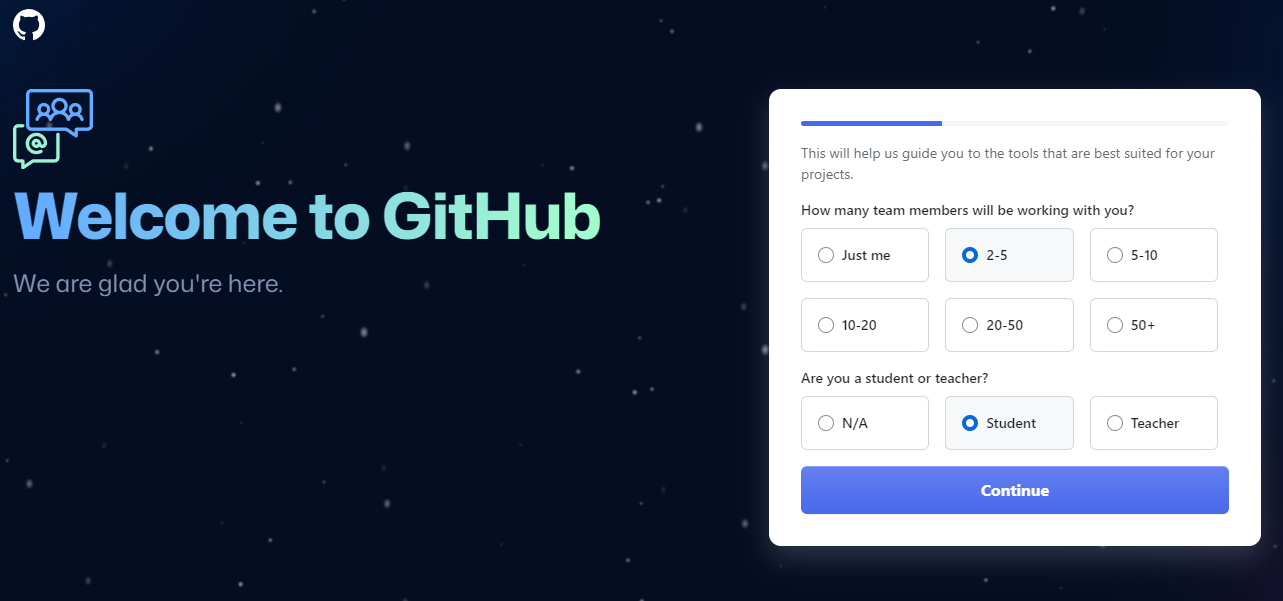
Navigate to <https://github.com/>

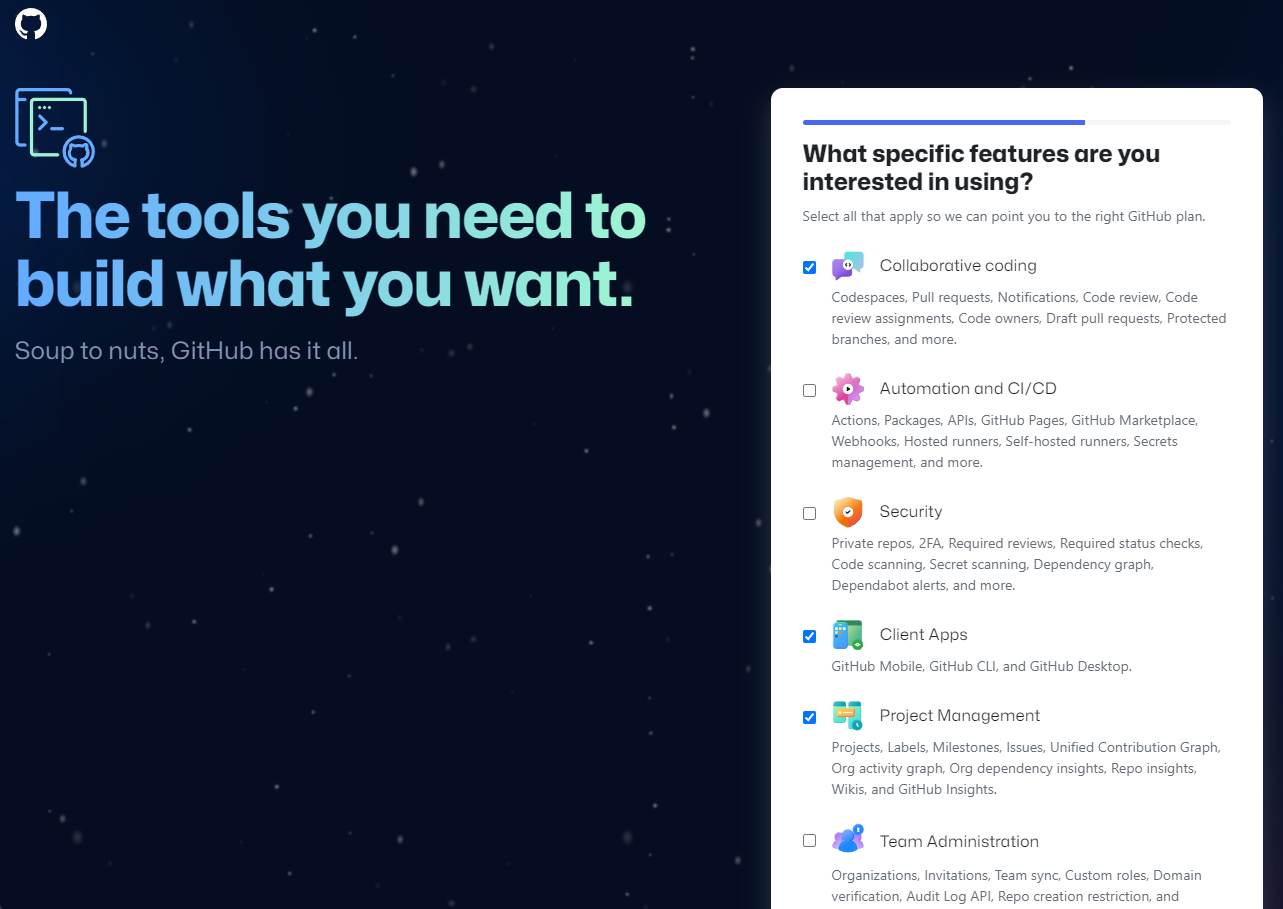


Click Sign Up. Create username and password.





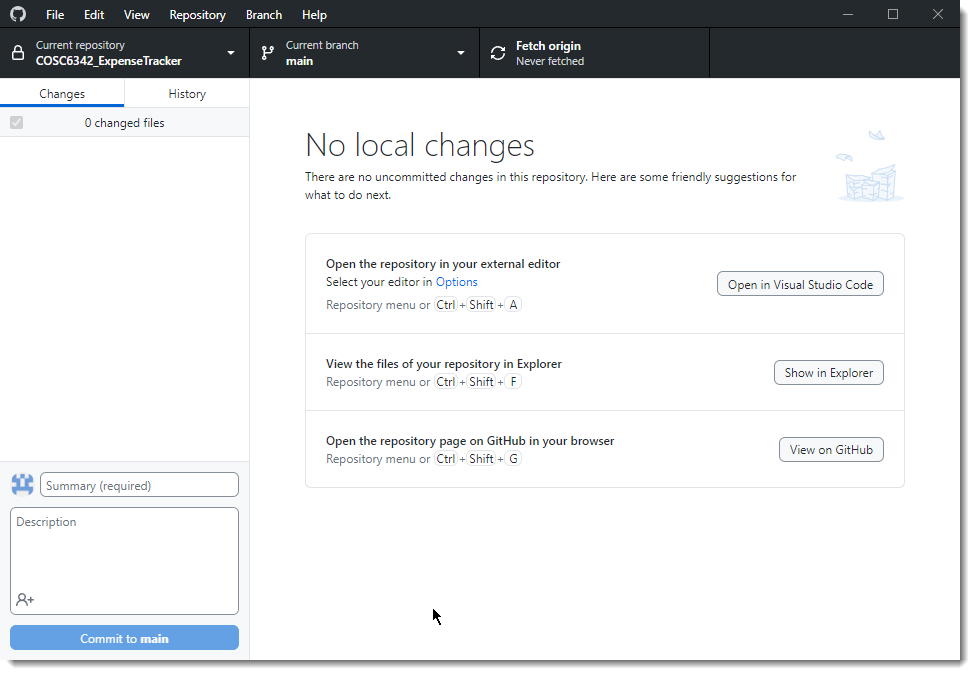
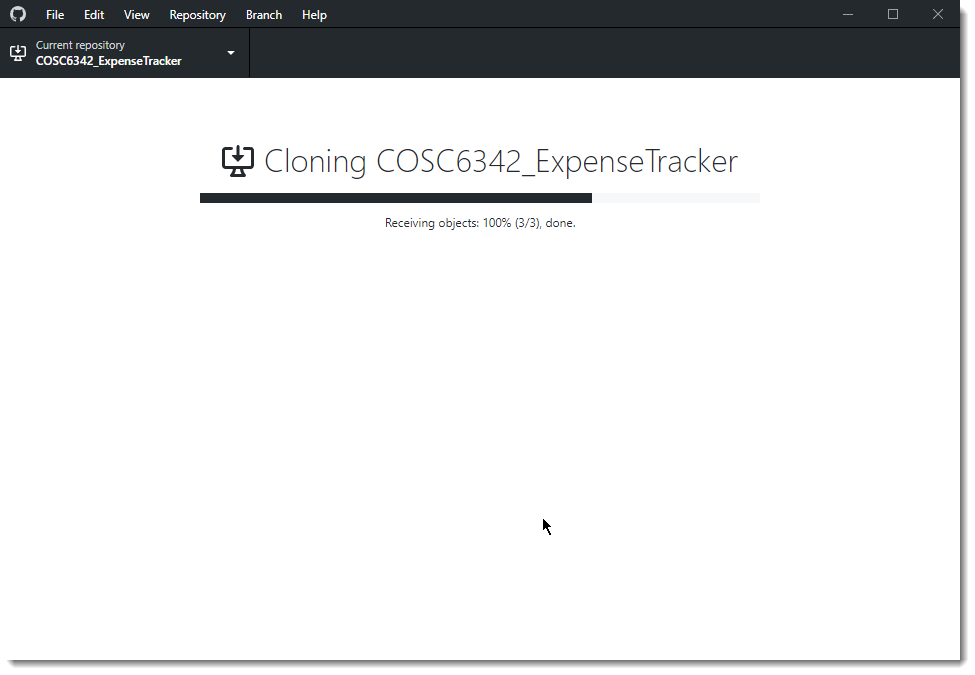
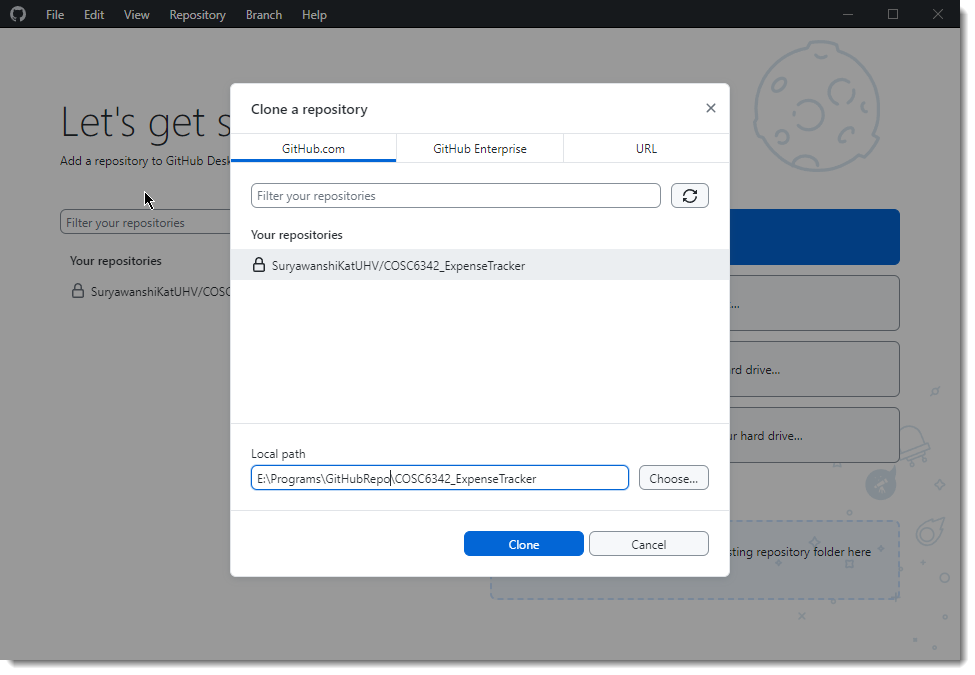
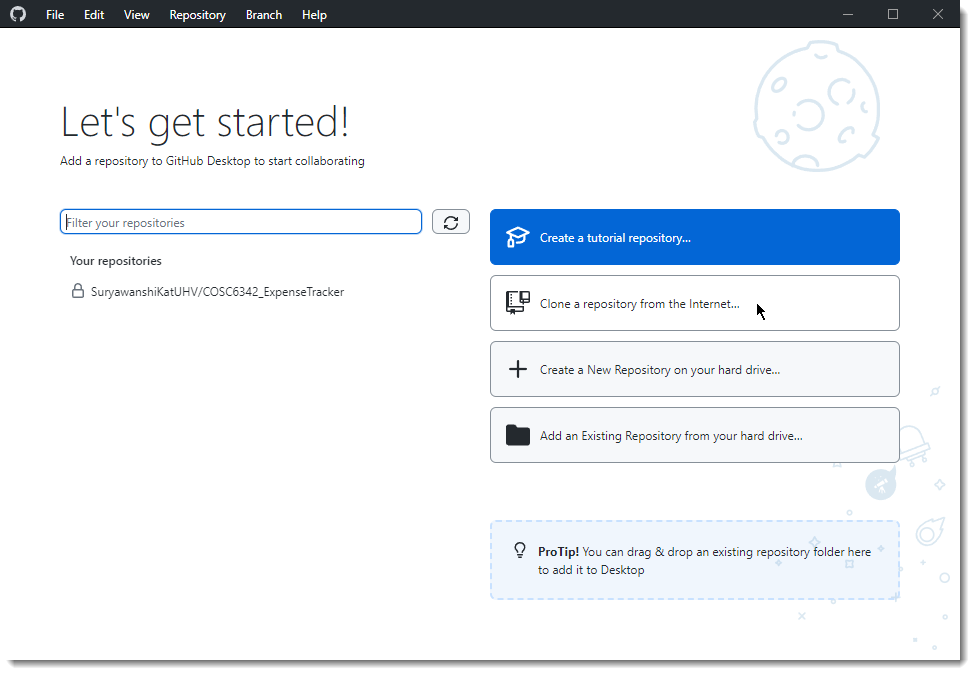
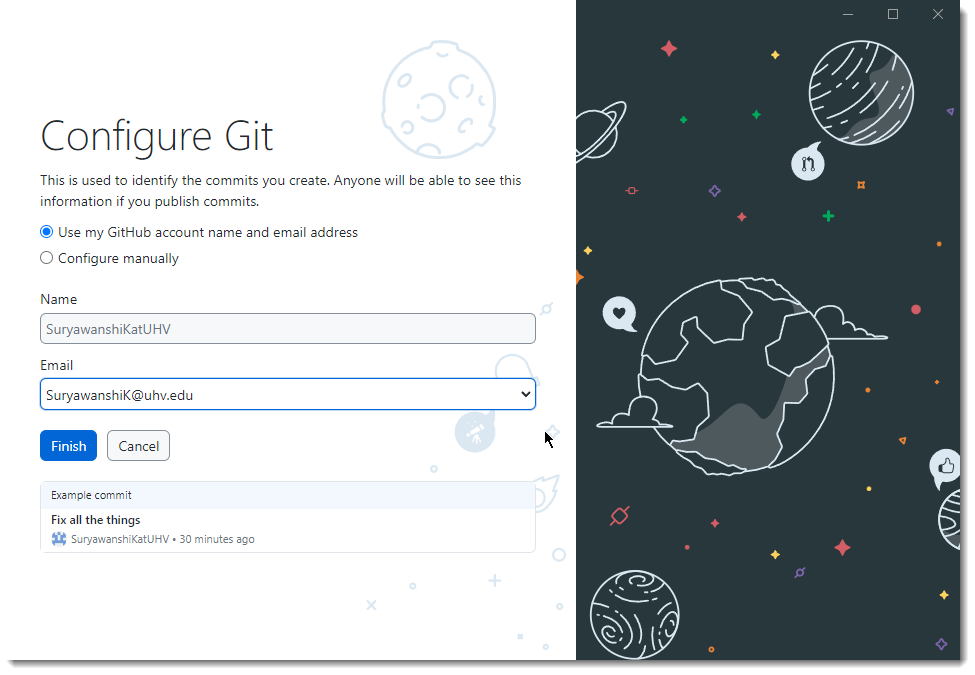
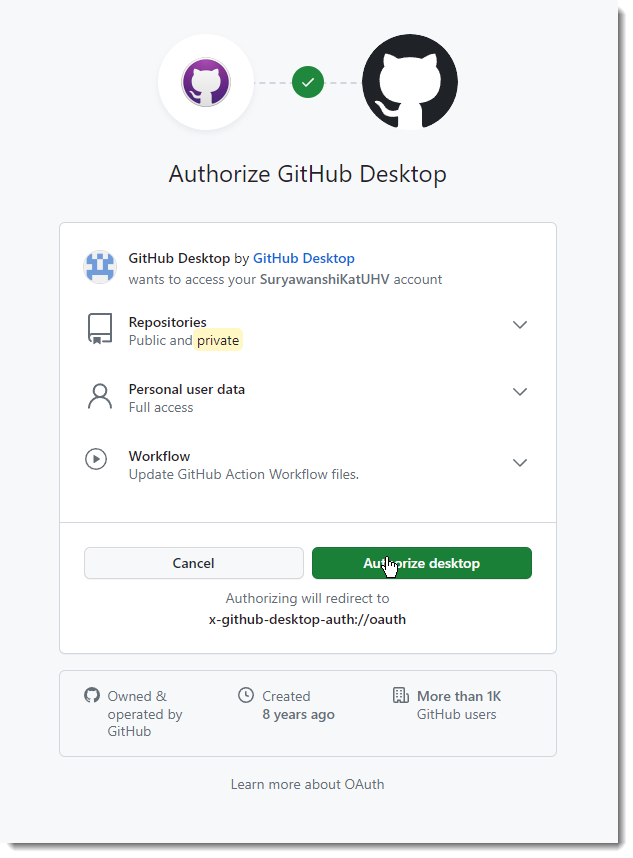
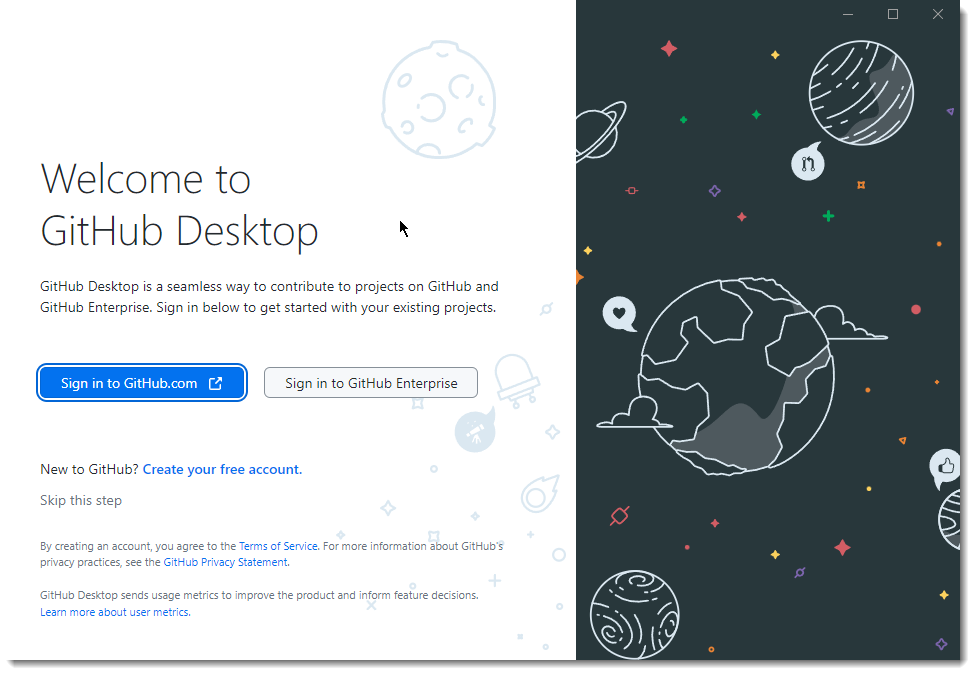




I have created a repository COSC6342\_ExpenseTracker

(Optional) Now download GitHub Desktop from <https://desktop.github.com/>

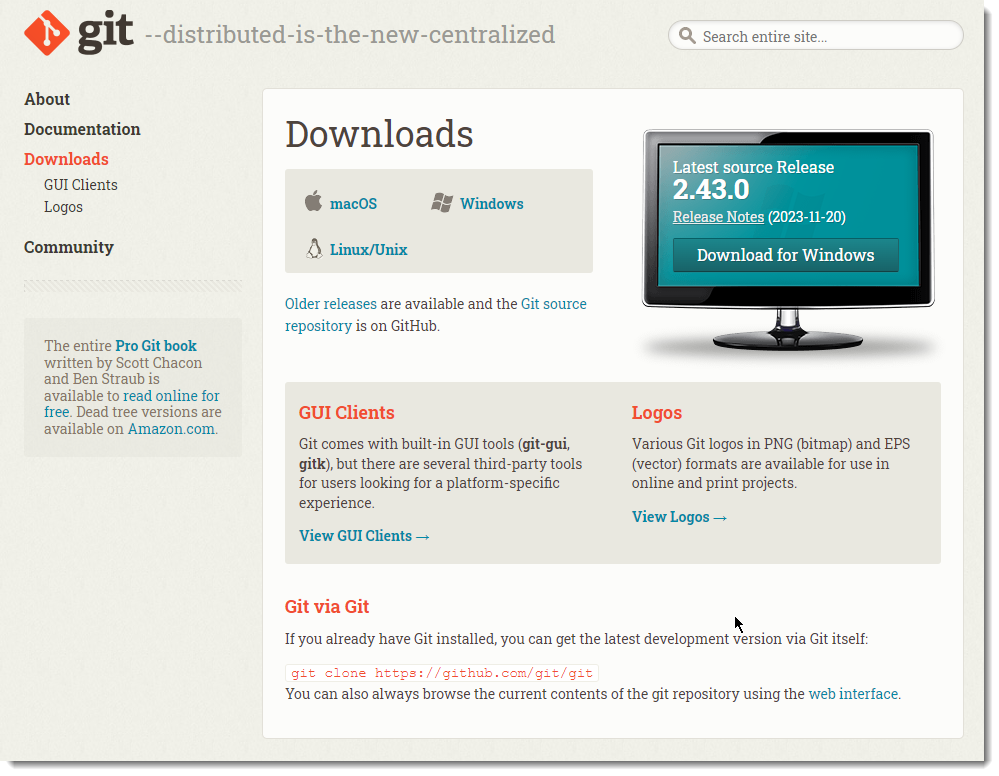




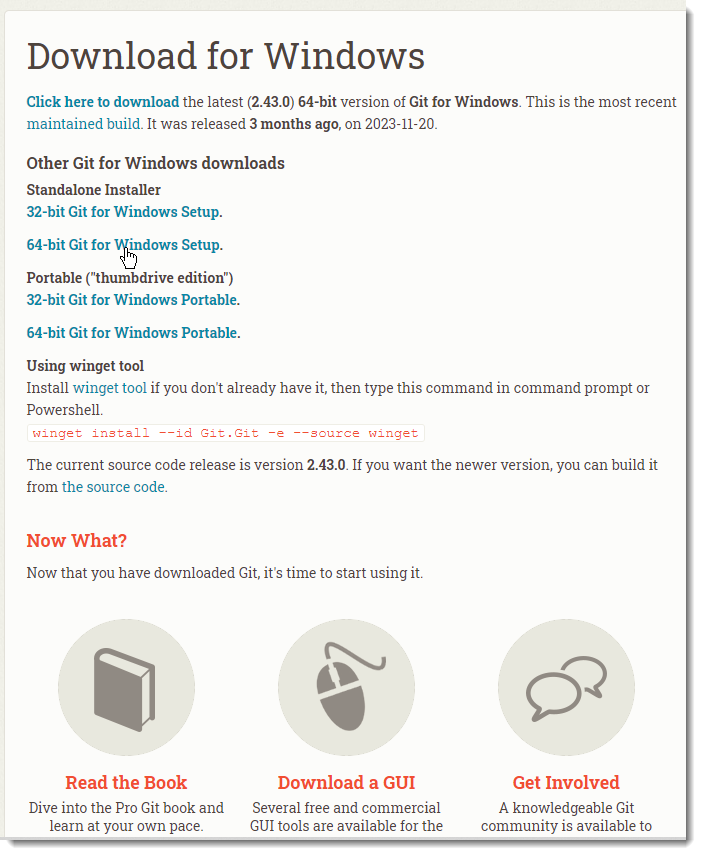
End

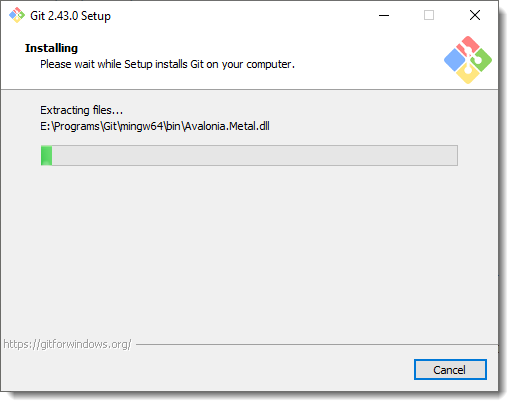
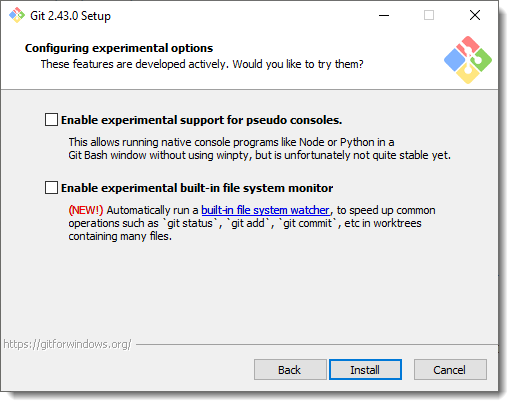
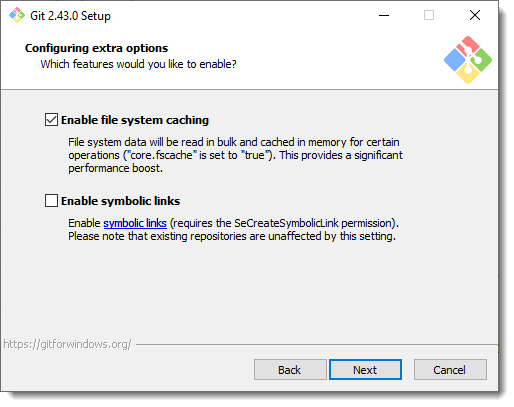
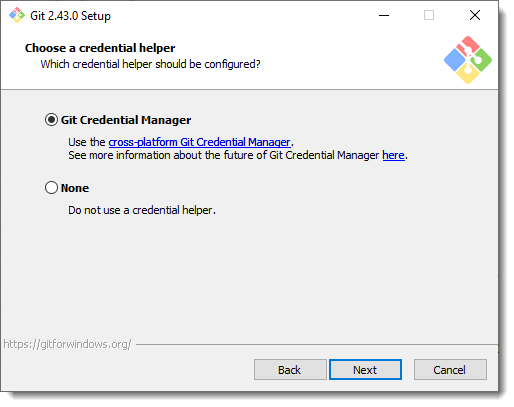
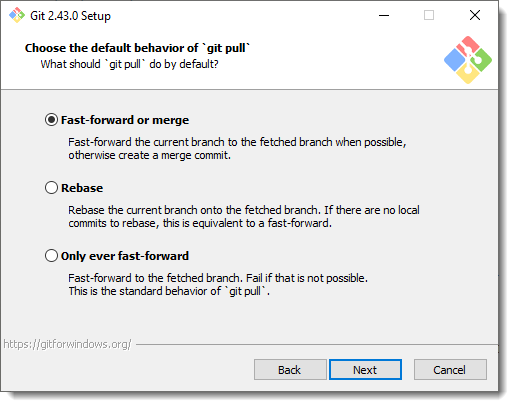
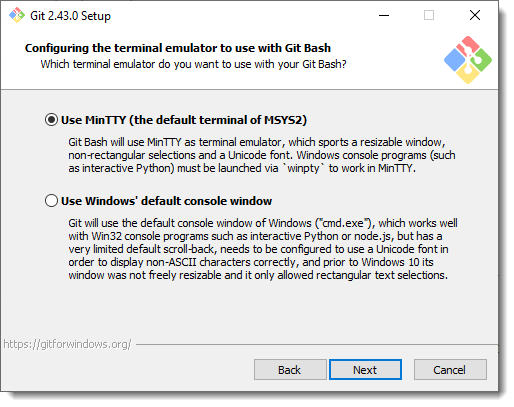
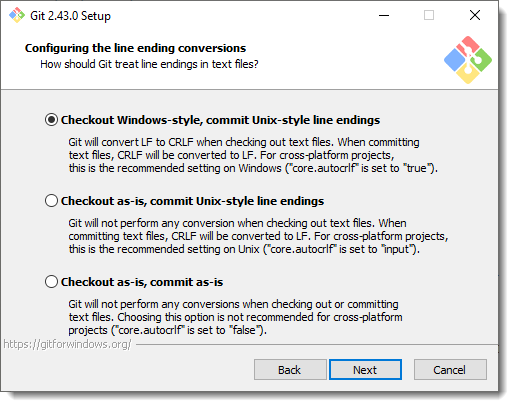
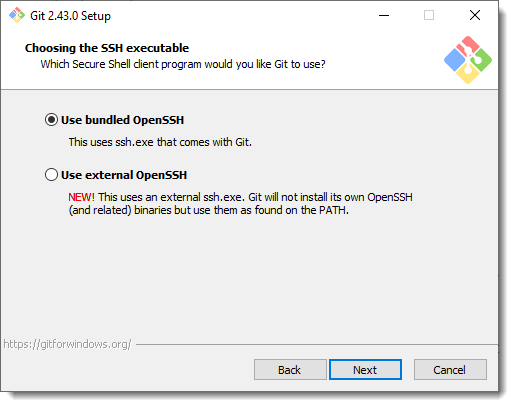
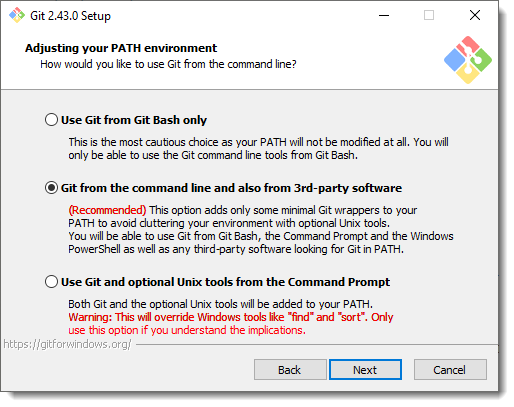
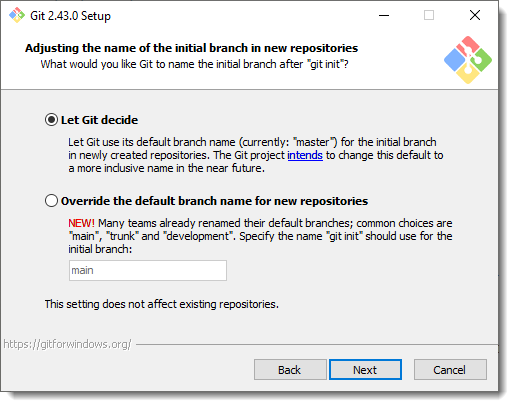
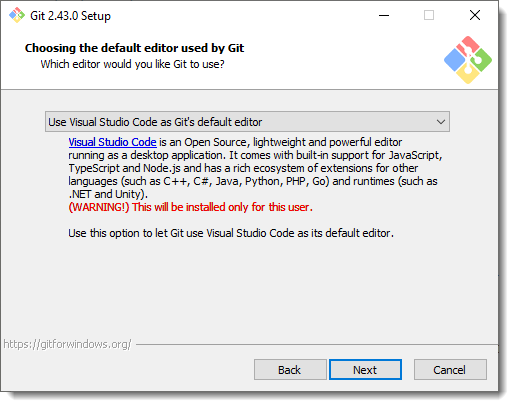
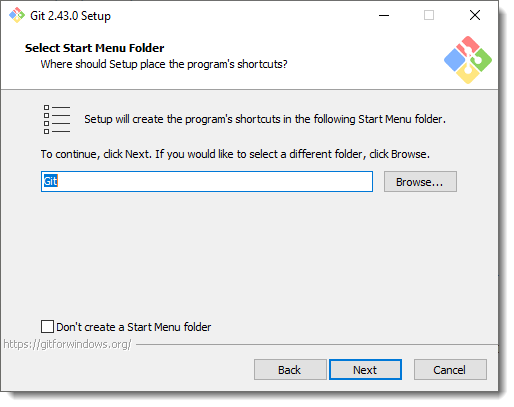
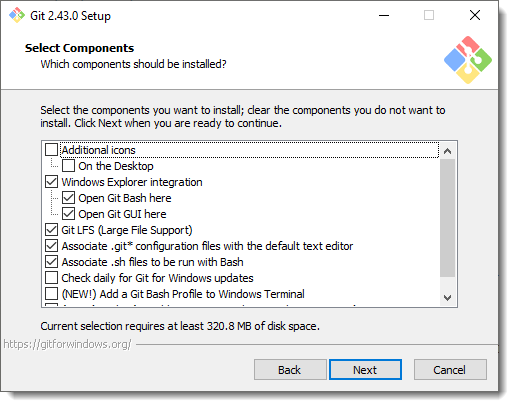
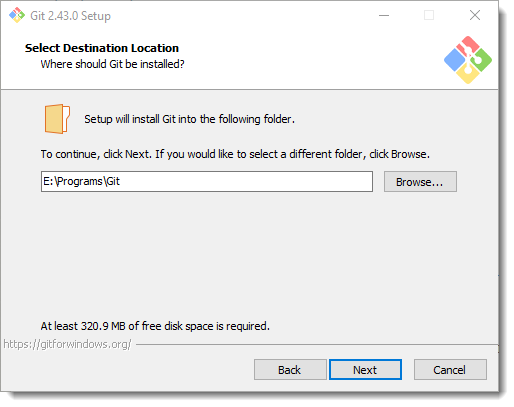
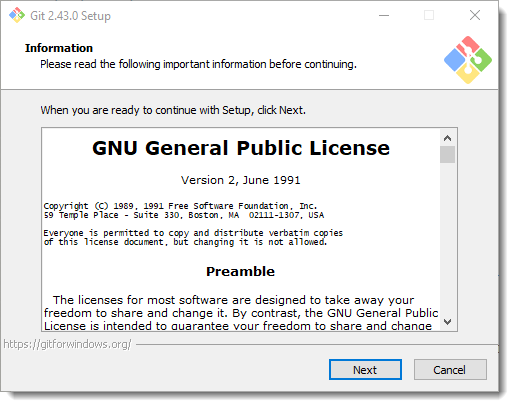
## Git

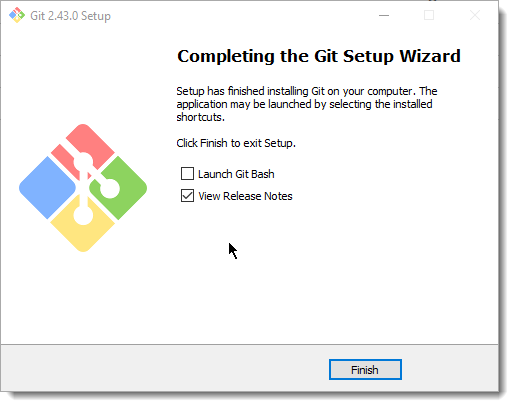
Navigate to <https://git-scm.com/downloads>



Download the required executable as per your operating system. Here I am downloading installer for Windows OS. Once downloaded double click the executable and start the installer. Following screens shows the options selected during a typical installation.







There are two ways to configure your username and email in Git settings: globally for all repositories on your computer, or locally for specific repositories. Here's how to do it for each:

* **Globally**:

1. Open your terminal.
2. Set your username: Use the command git config --global user.name "YOUR\_USERNAME". Replace YOUR\_USERNAME with your preferred username.
3. Set your email: Use the command git config --global user.email "YOUR\_EMAIL". Replace YOUR\_EMAIL with your actual email address.
4. Verify your settings: Use the command git config --global --list to check if your username and email are set correctly.



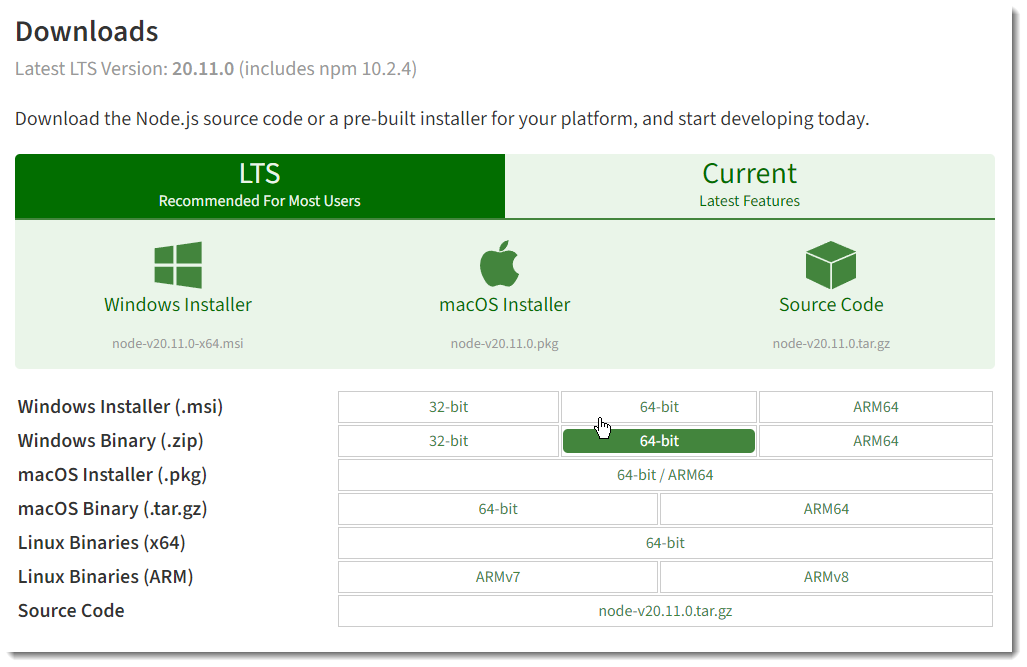
* **Locally:**

1. Navigate to the desired repository directory.
2. Set your username: Use the command git config user.name "YOUR\_USERNAME".
3. Set your email: Use the command git config user.email "YOUR\_EMAIL".
4. Verify your settings: Run git config --list to see the local configuration for that repository.

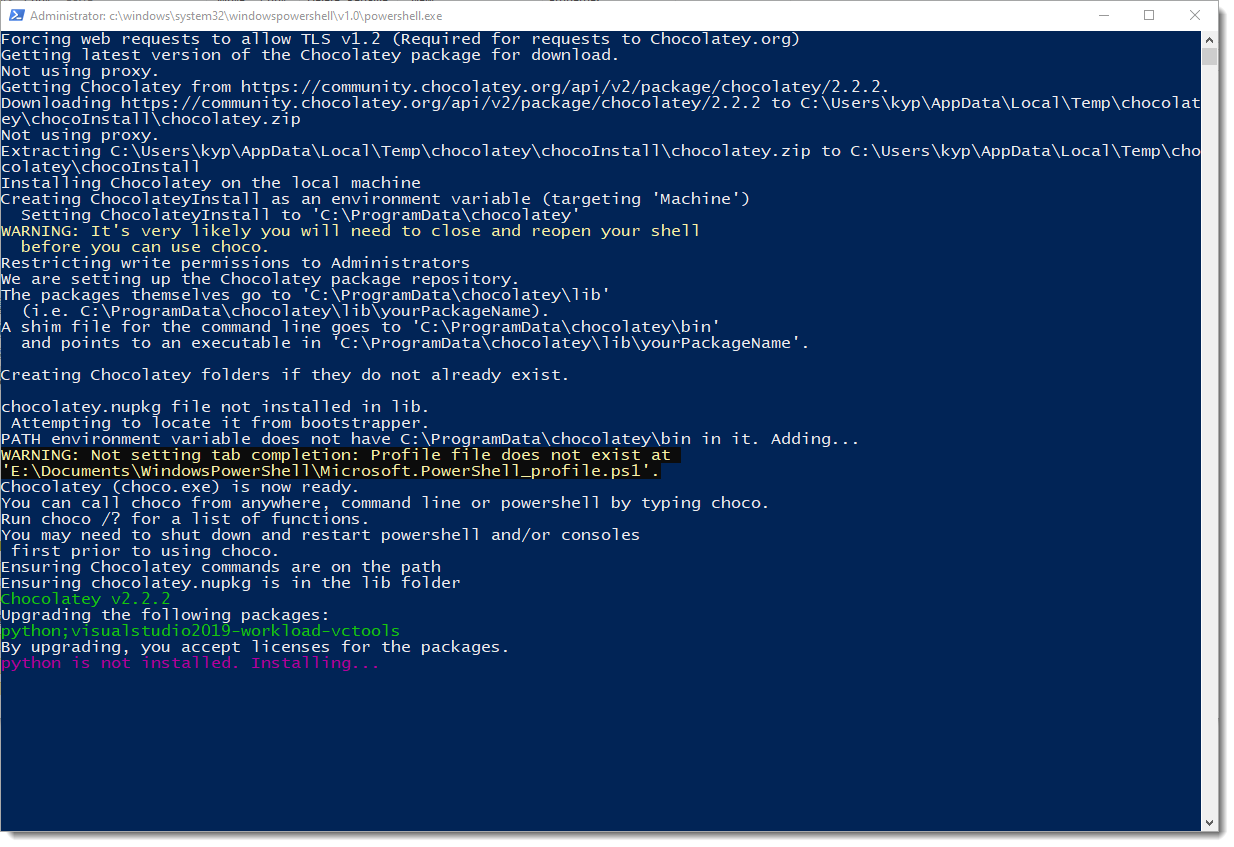
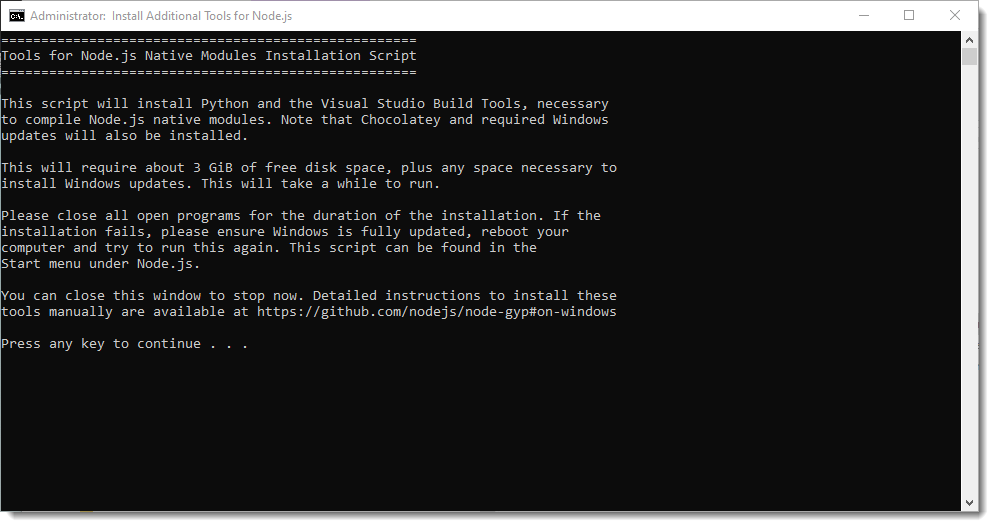
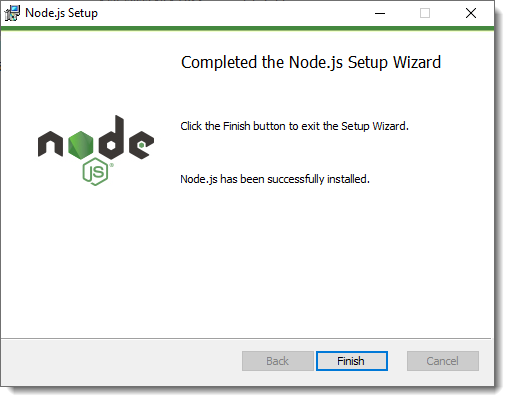
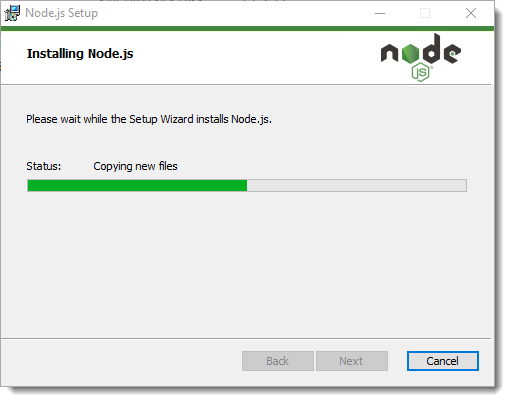
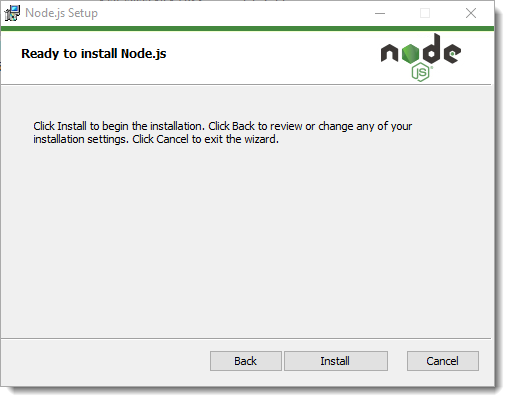
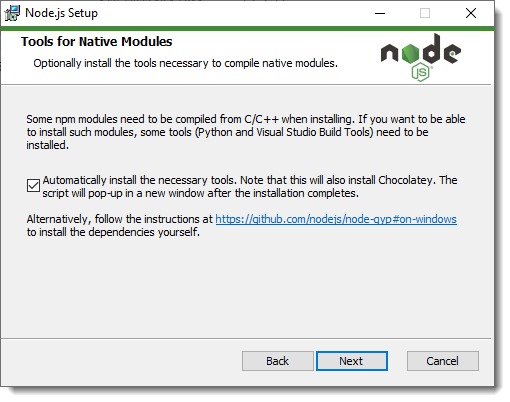
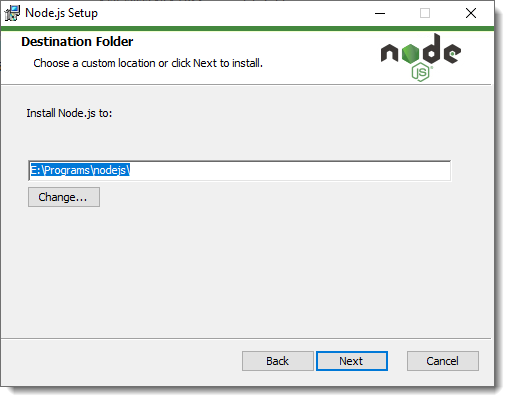
* **Additional tips:**
* You can use quotes around your username and email if they contain spaces.
* Consider using a separate email address for Git commits if you want to keep your primary email private.
* Some platforms like GitHub may have additional settings for configuring your username and email.

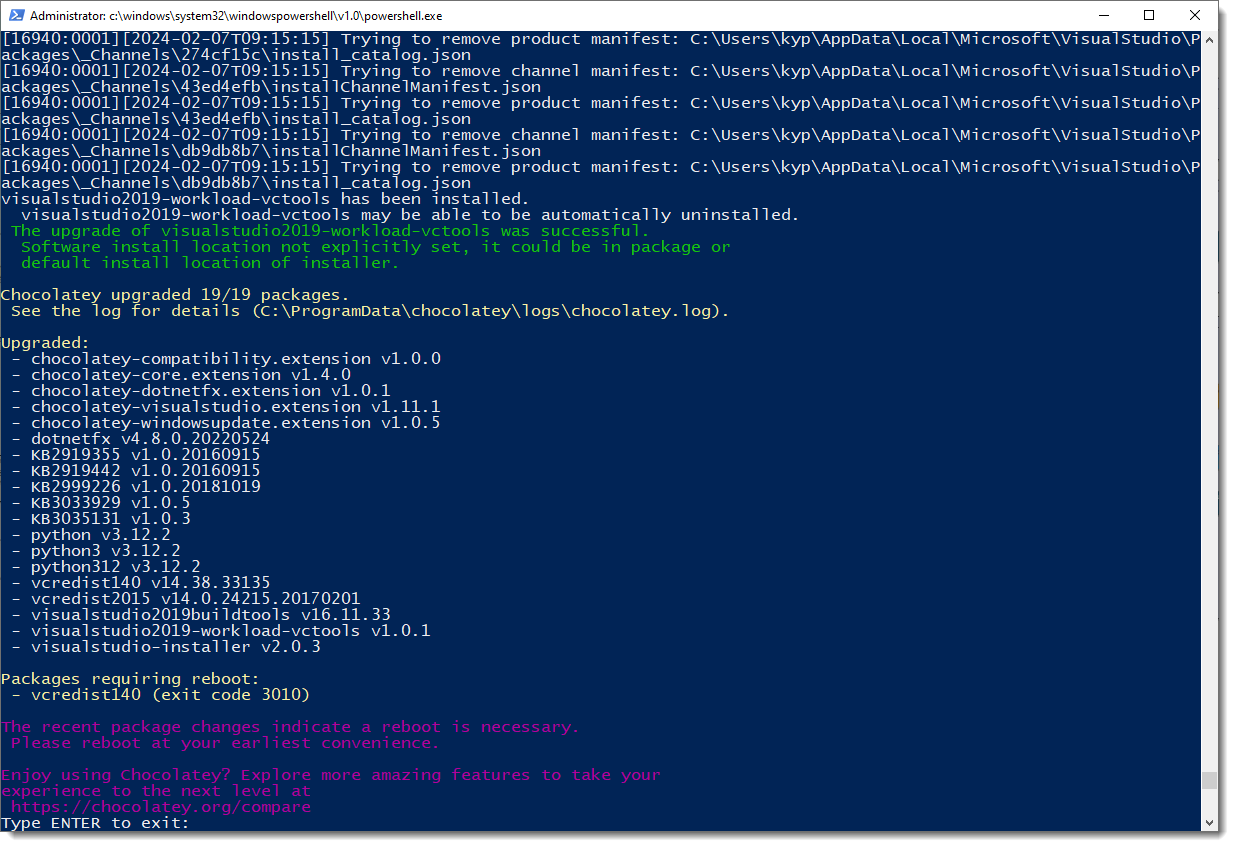
## Node.js

Navigate to <https://nodejs.org/en/download>



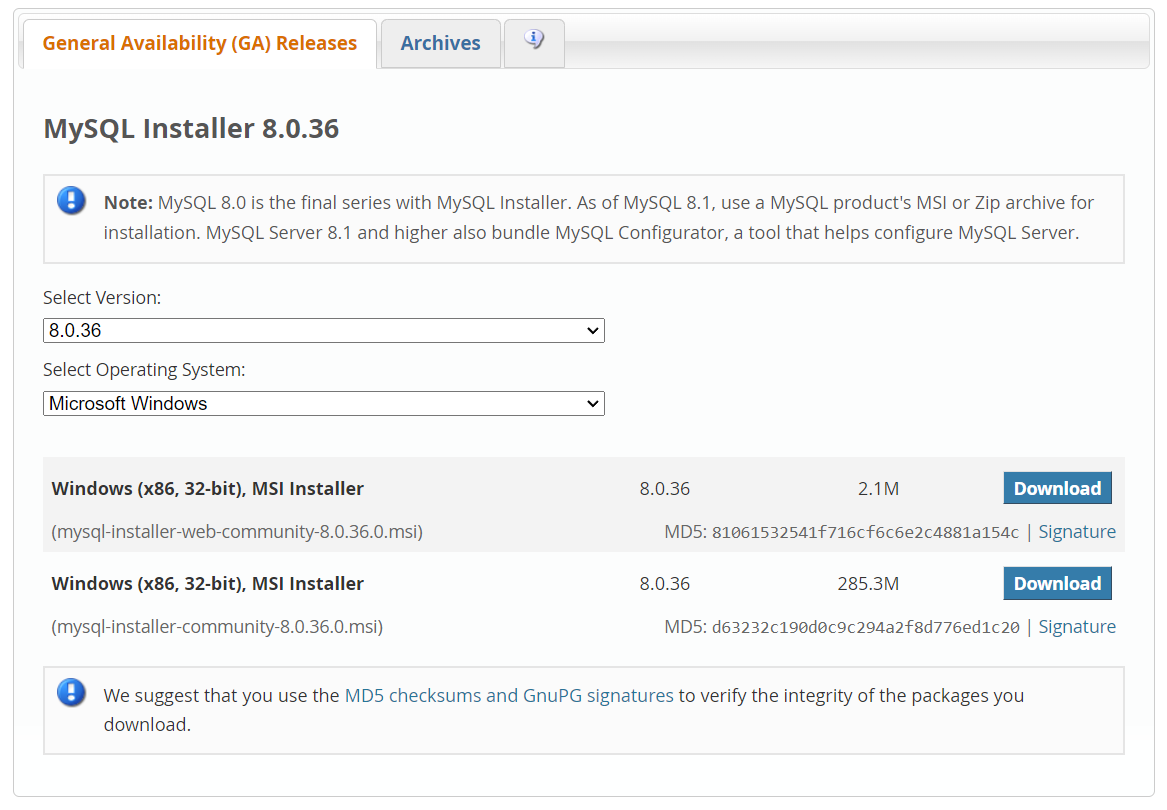
Download the required executable as per your operating system. Here I am downloading installer for Windows OS. Once downloaded double click the executable and start the installer. Following screens shows the options selected during a typical installation.

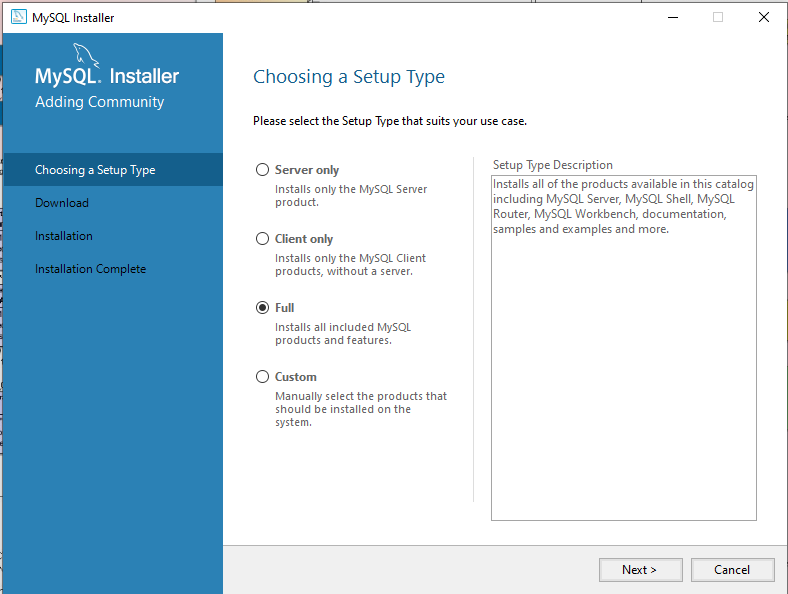


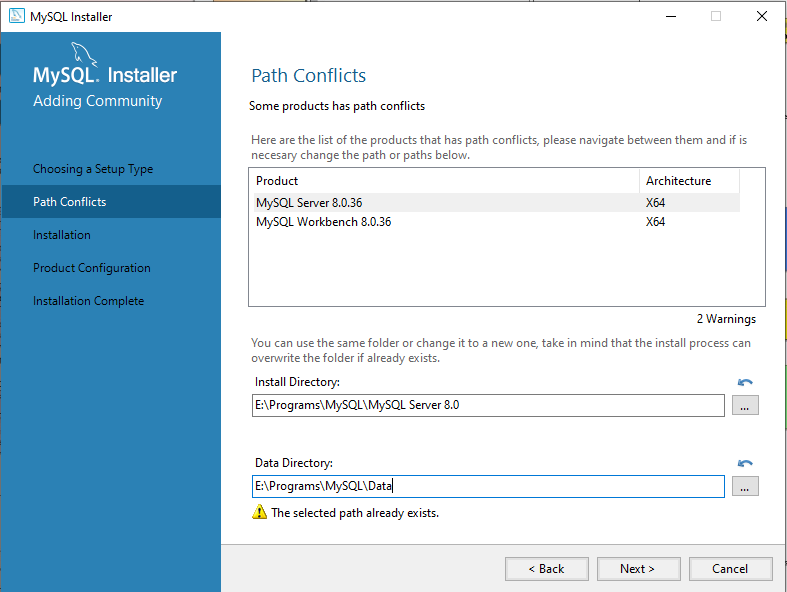


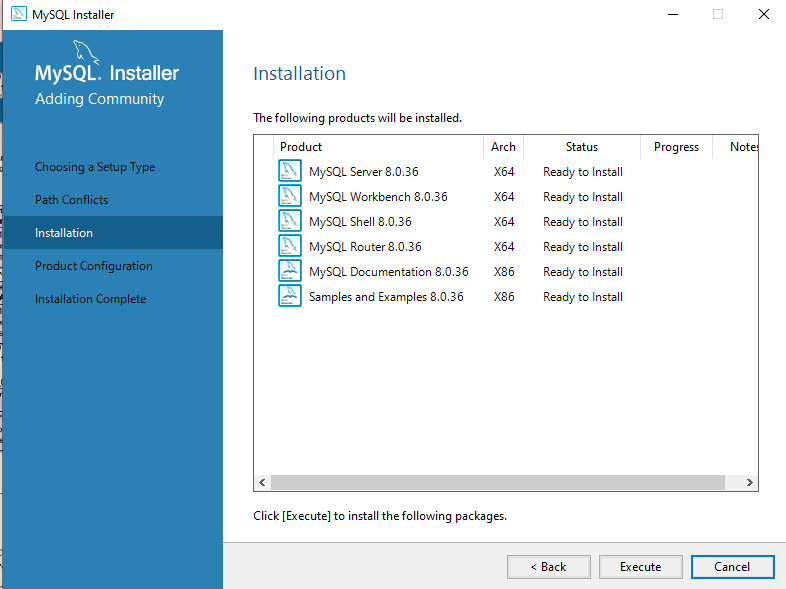
## MySQL

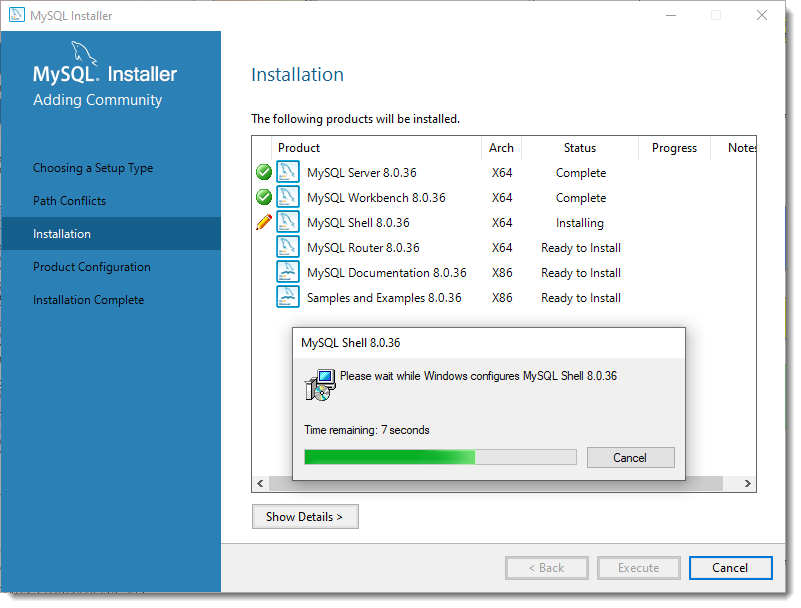
Navigate to <https://dev.mysql.com/downloads/installer/> and download the installer for your platform.

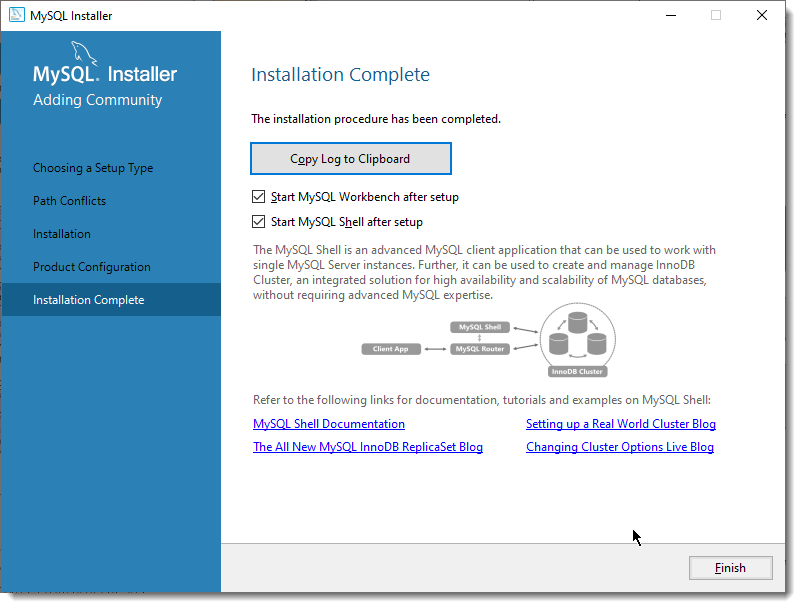
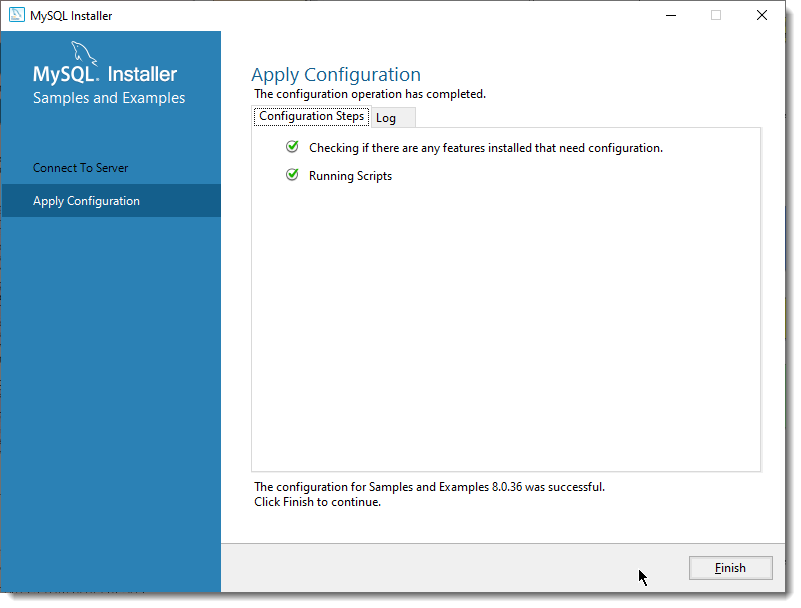
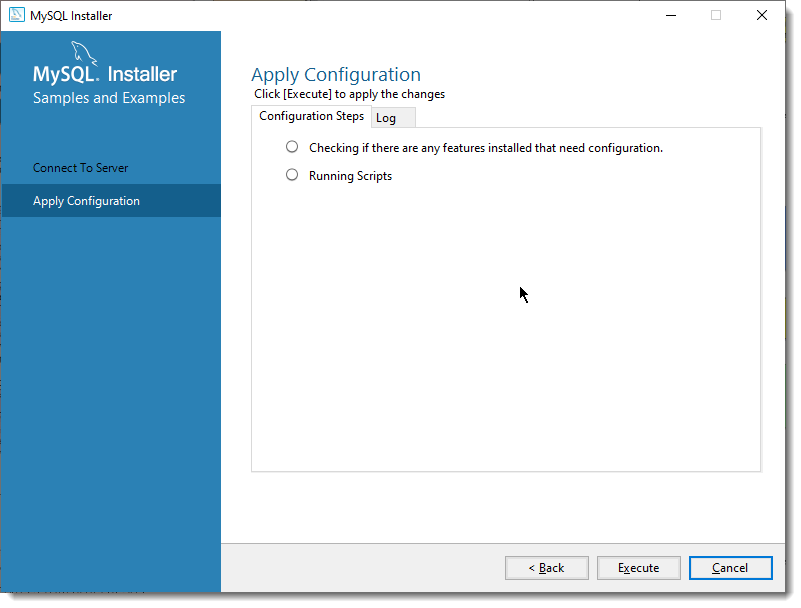
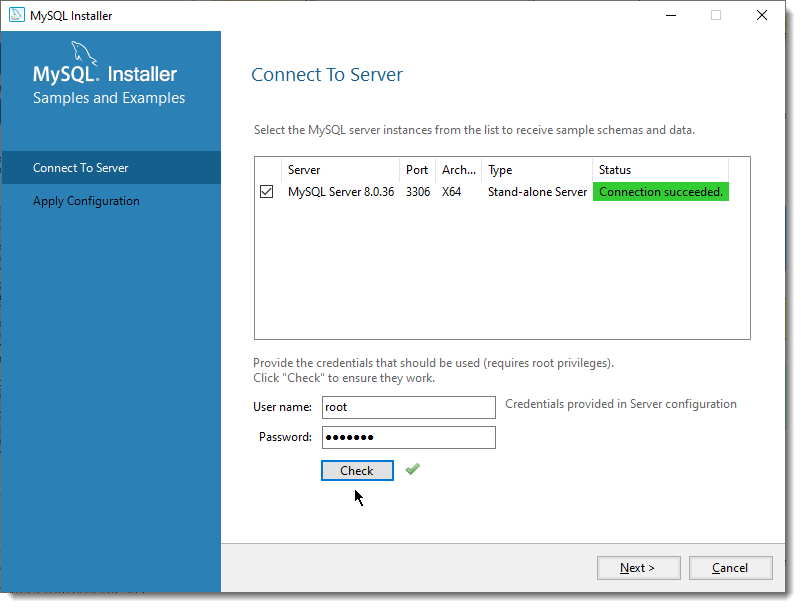
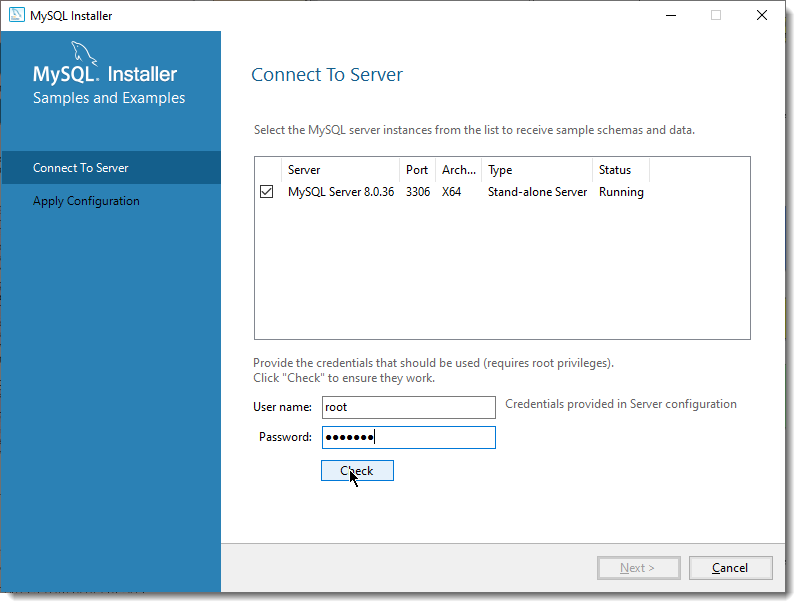
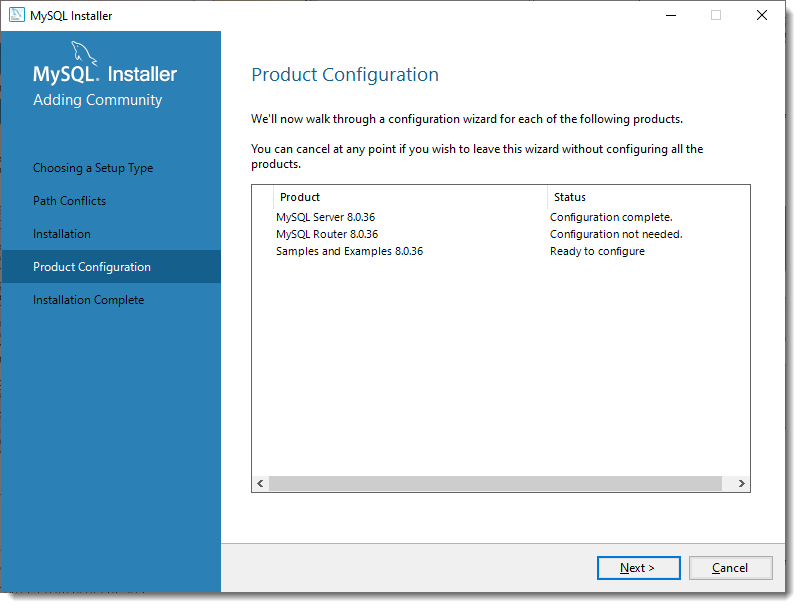
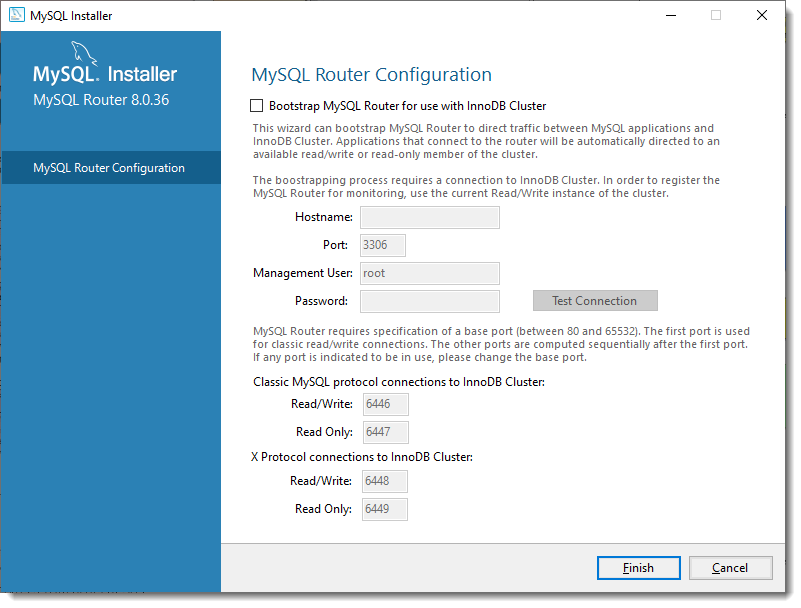
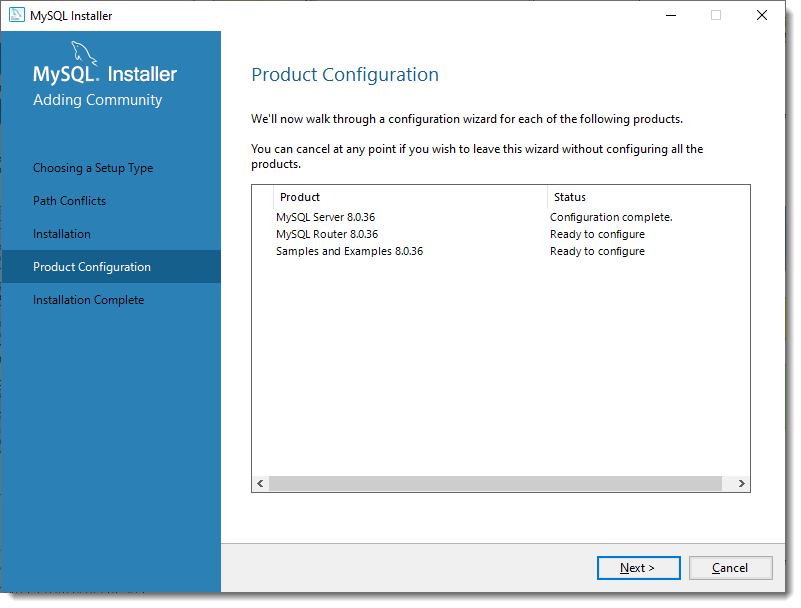
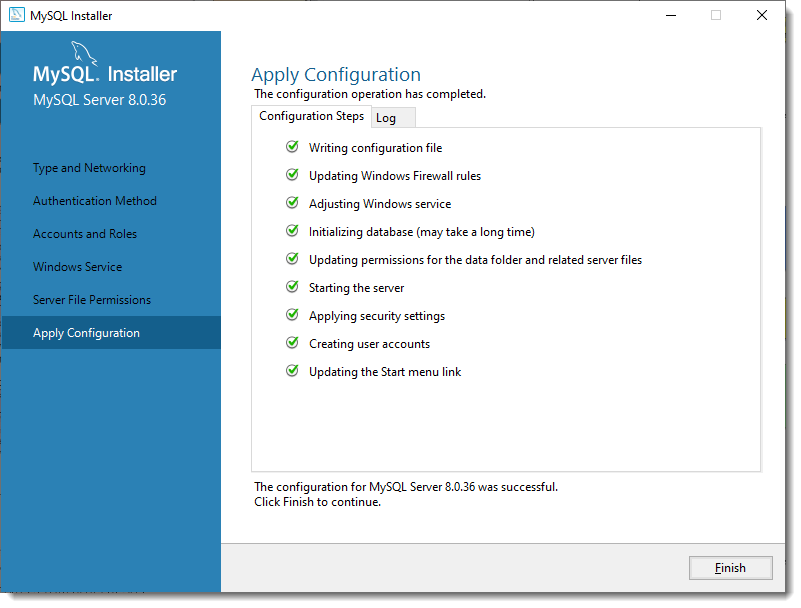
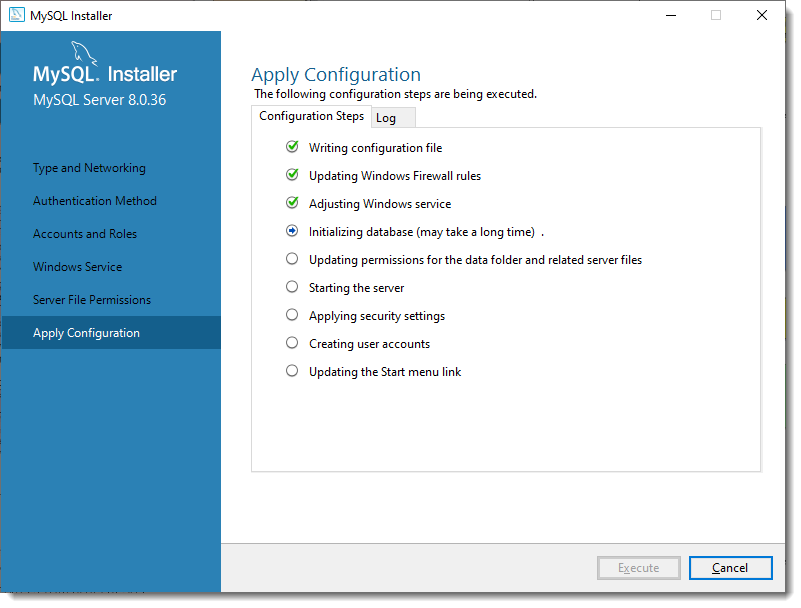
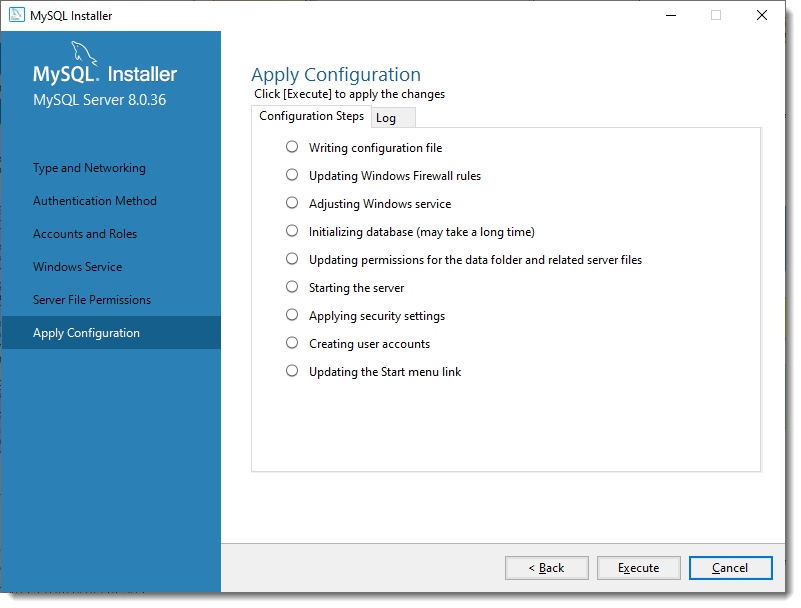
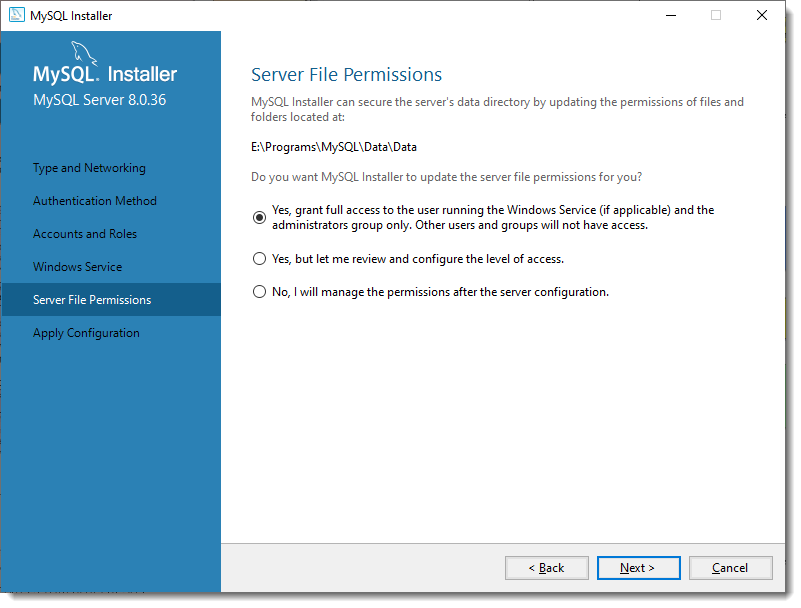
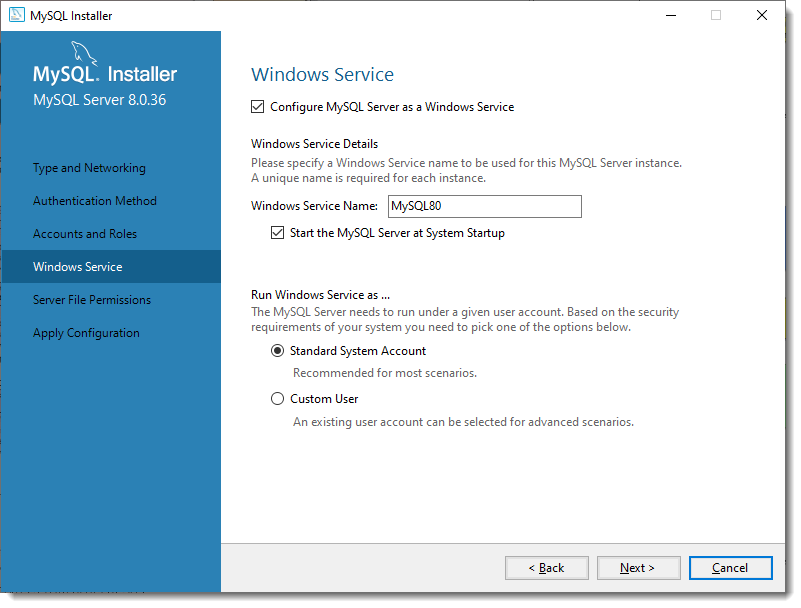
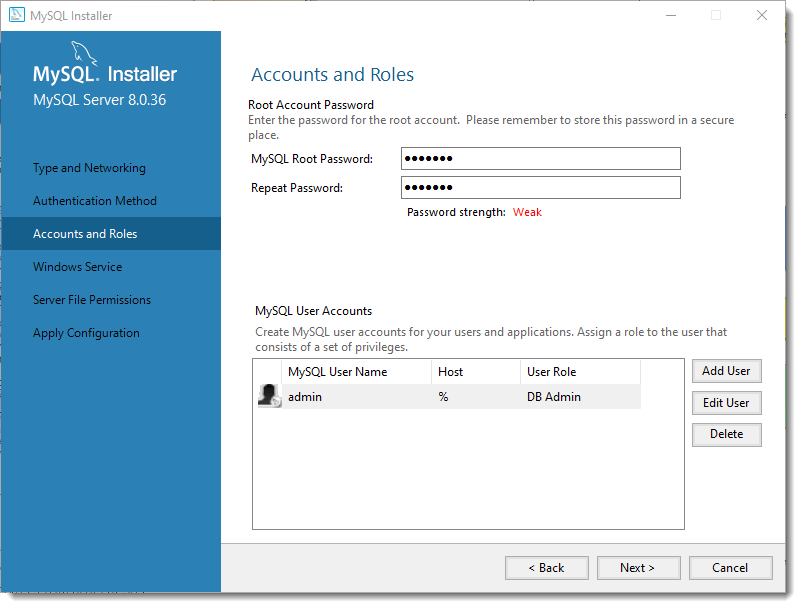
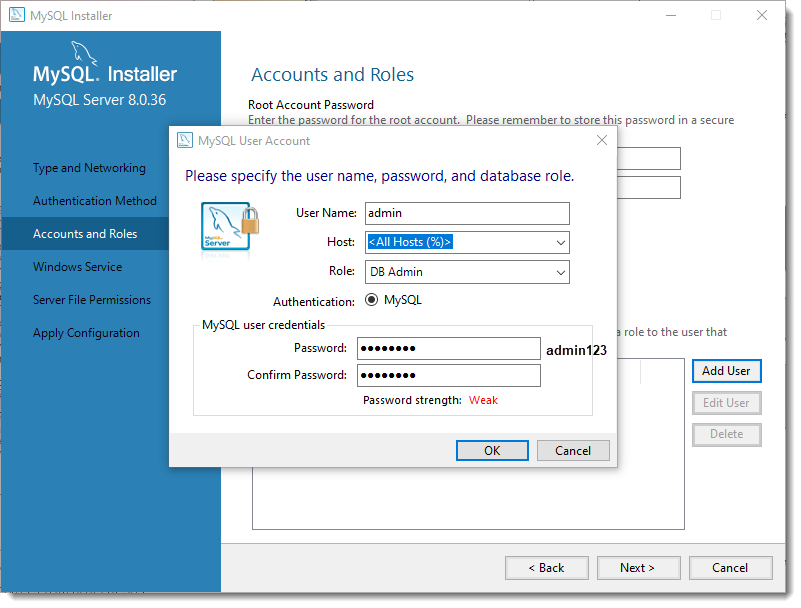
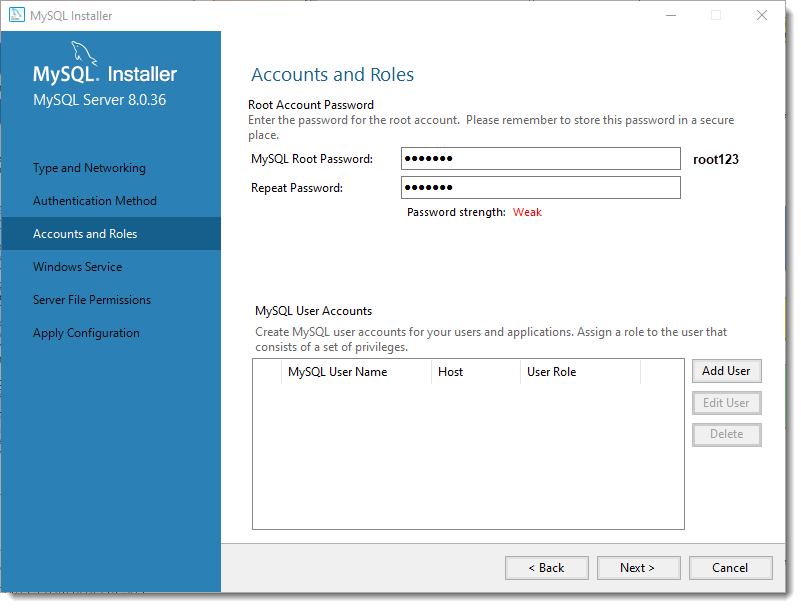
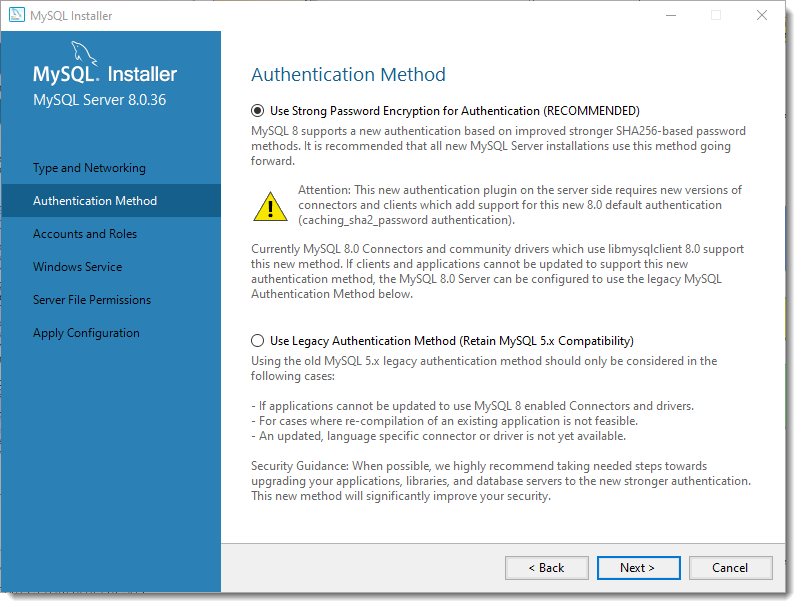
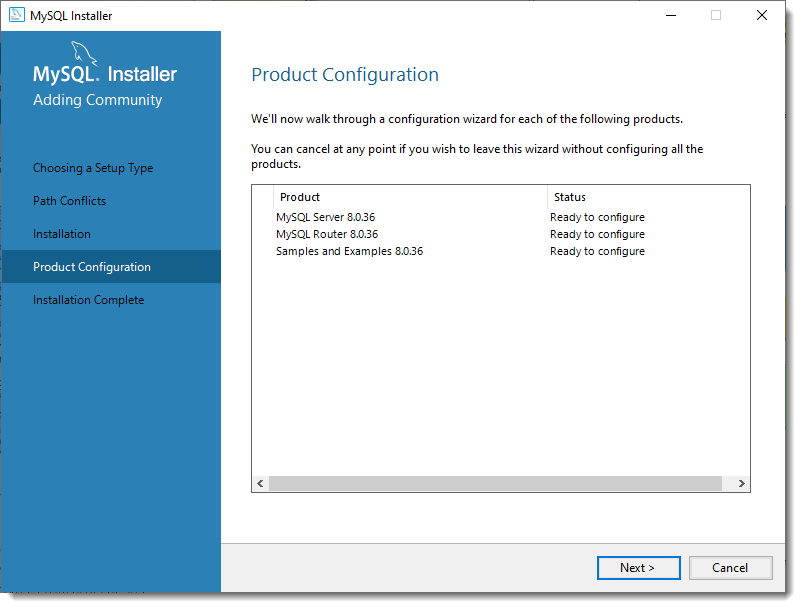
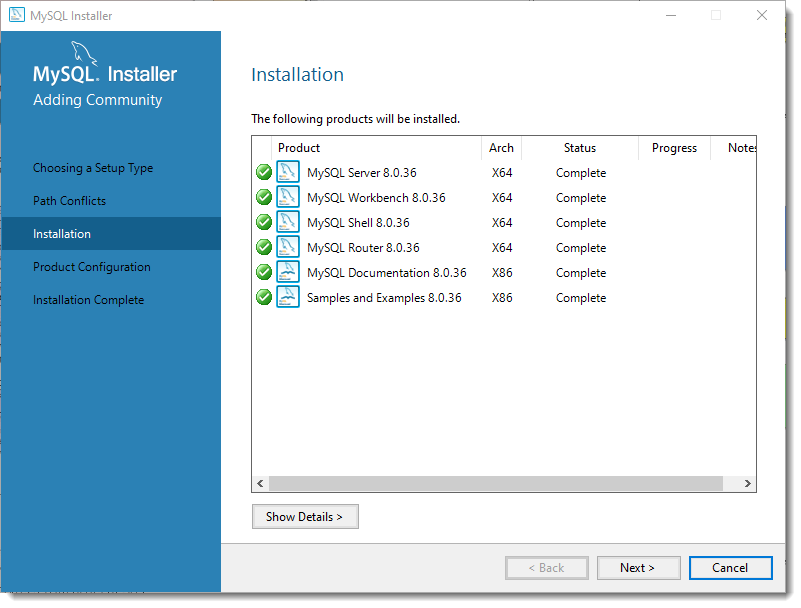








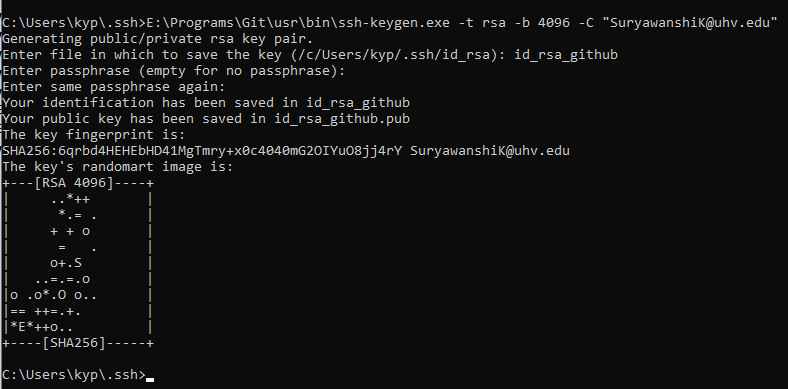


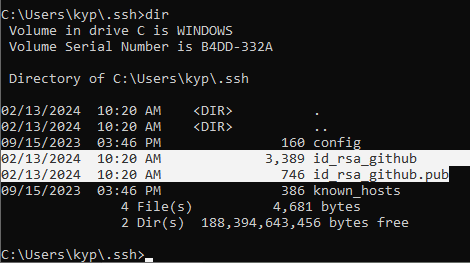


## Setup Git <->GitHub communication

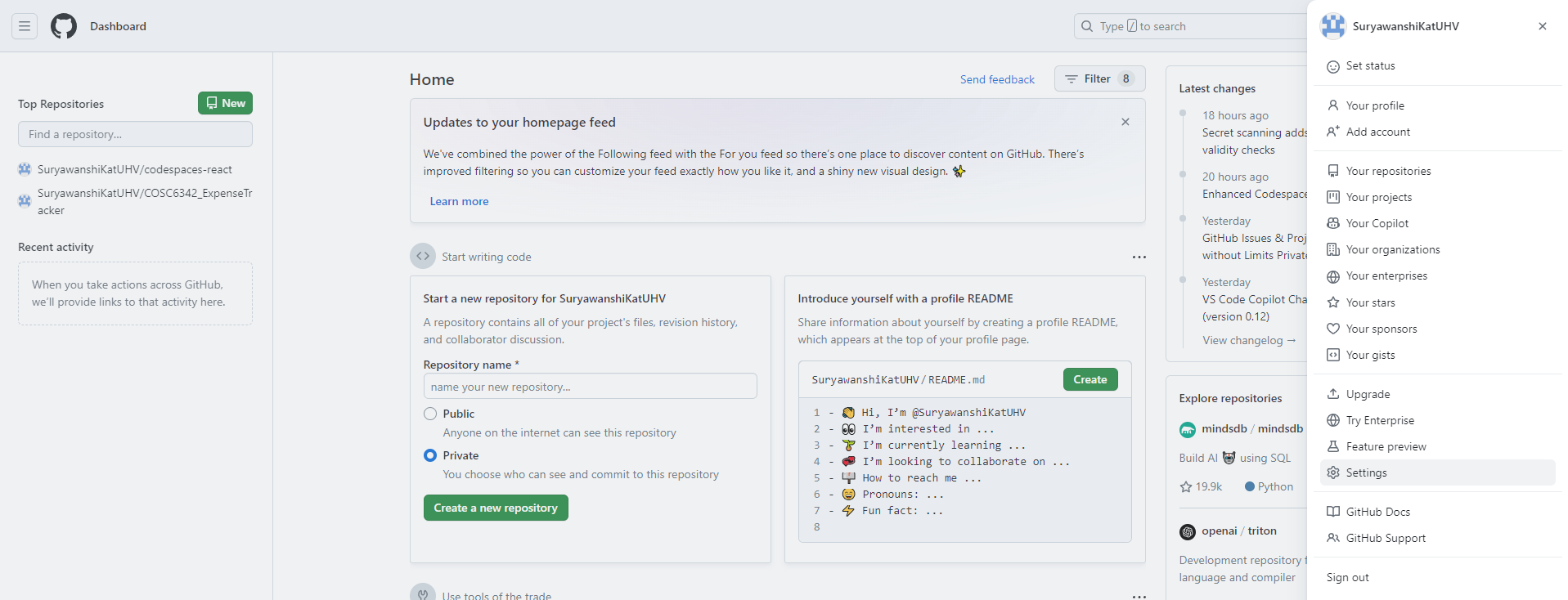
To push/pull code from GitHub to local Git repository we need to configure secure connection using private-public key pair.

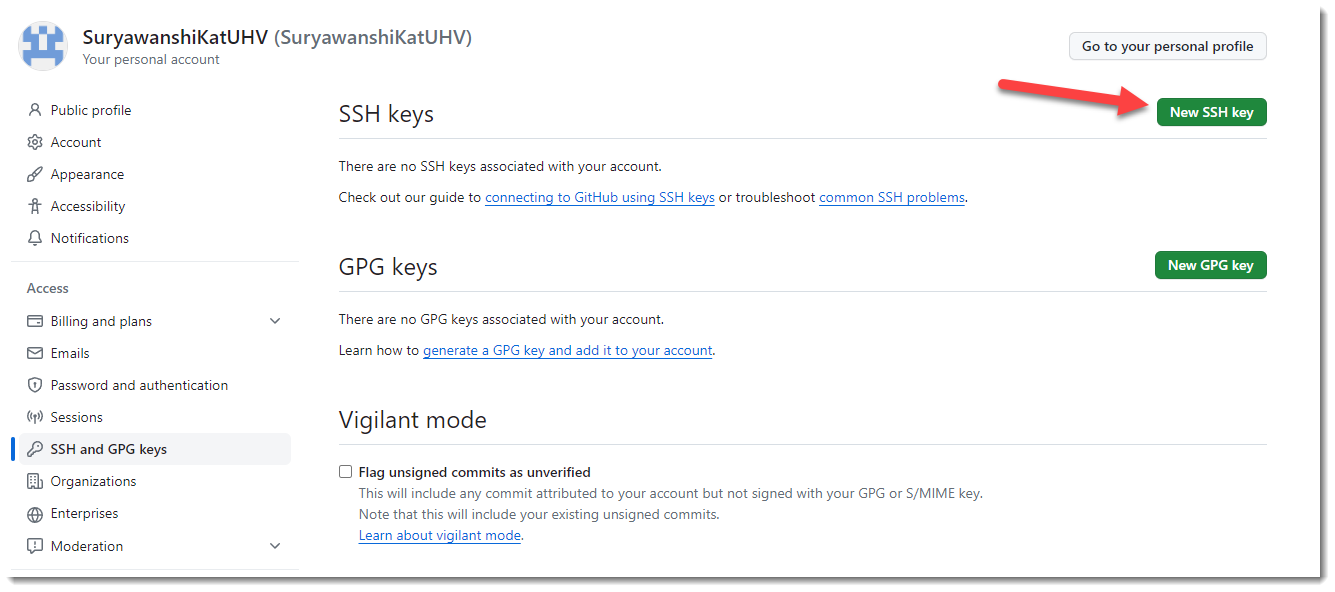
Navigate to the user profile directory and create a ssh key. For this I have used ssh-keygen application from Git installation.



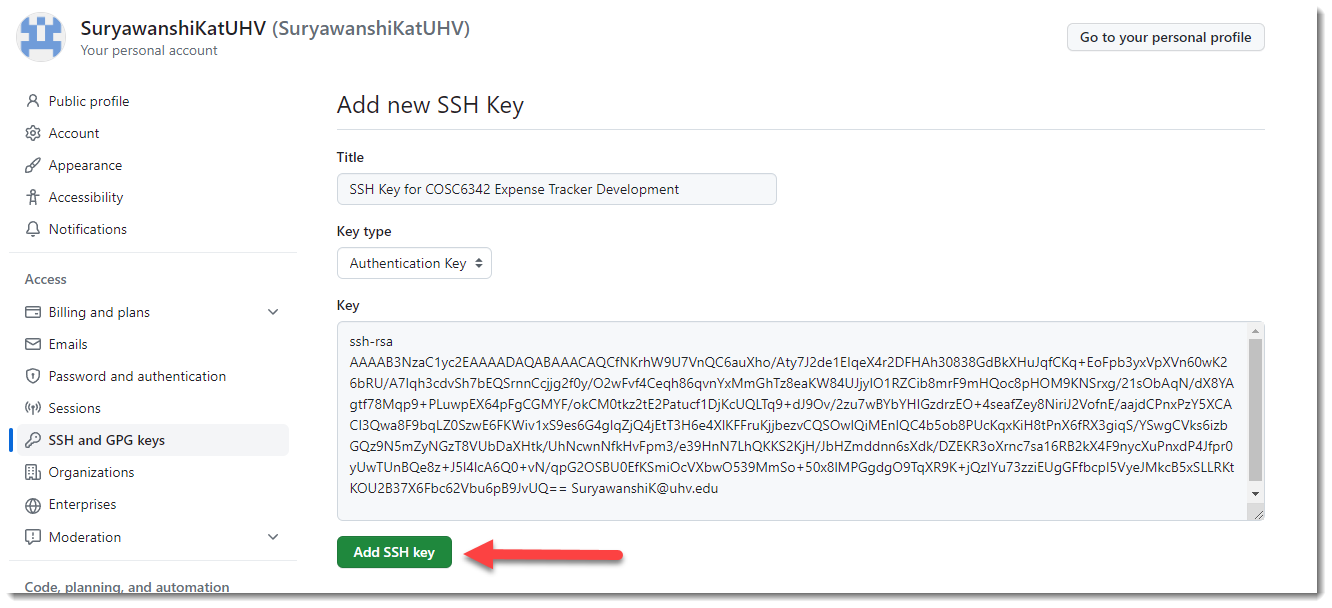


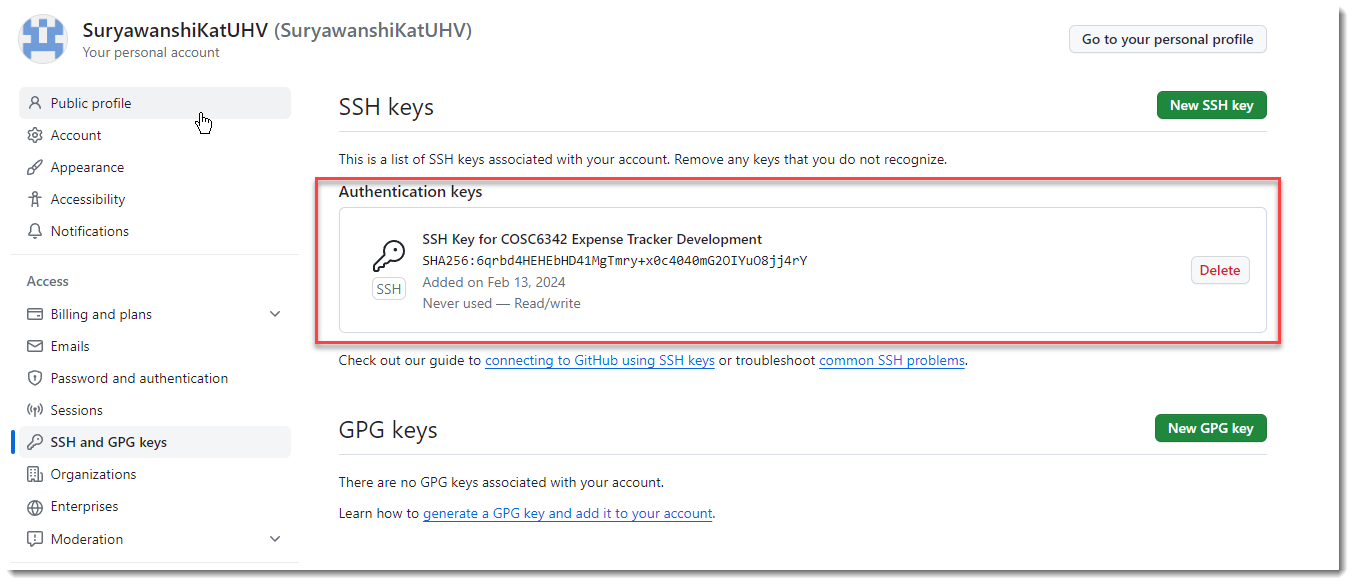
Now add the public key in Github settings.





Now copy the contents from .pub key file into new key created below. Give any title you like.



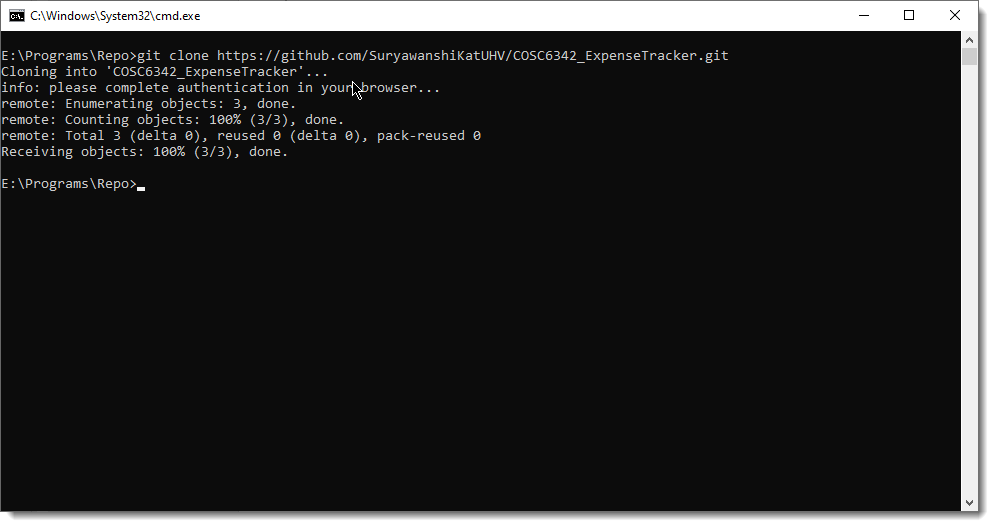
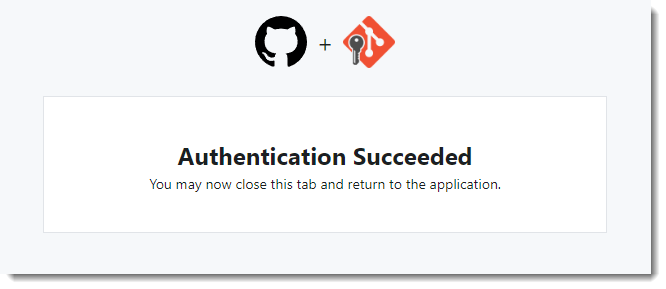
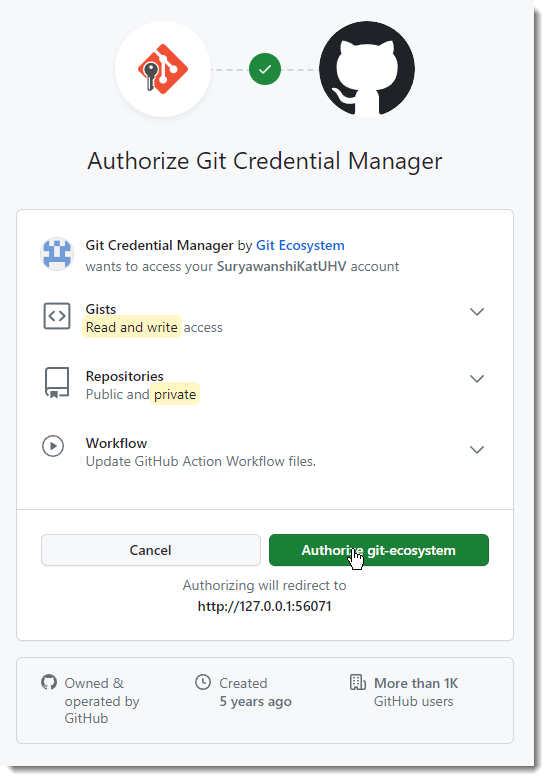


Once this is done now let’s clone and pull the repository from github into local directory to work on.

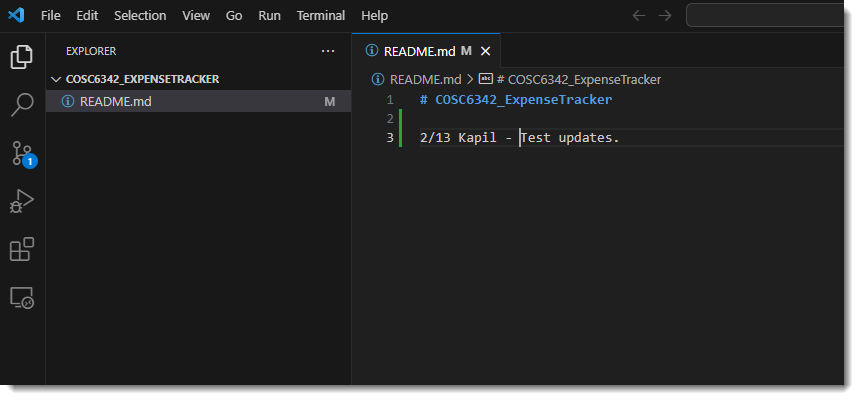
Copy the clone url for the remote repository from github.

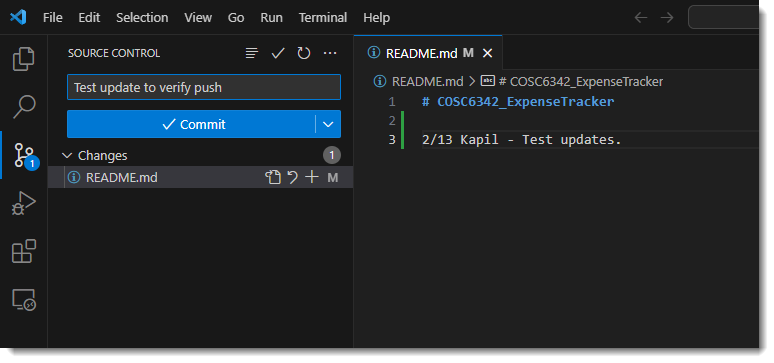


Create a local folder and open a command prompt in this folder, then clone this repo.

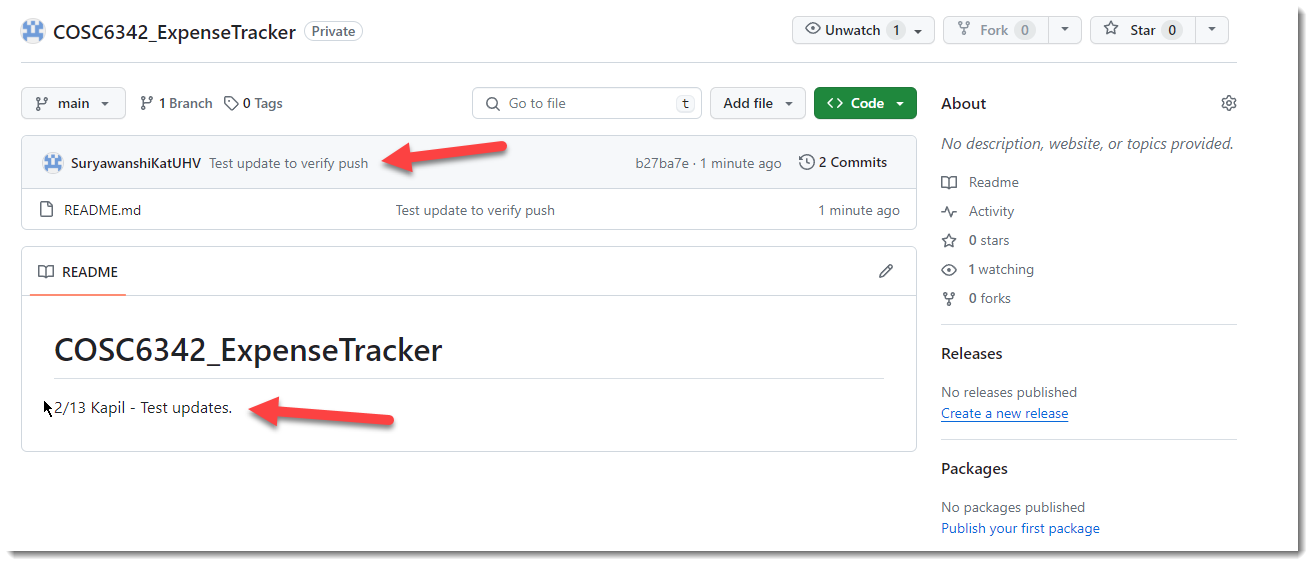


Now open the local repo folder into Visual Studio Code.

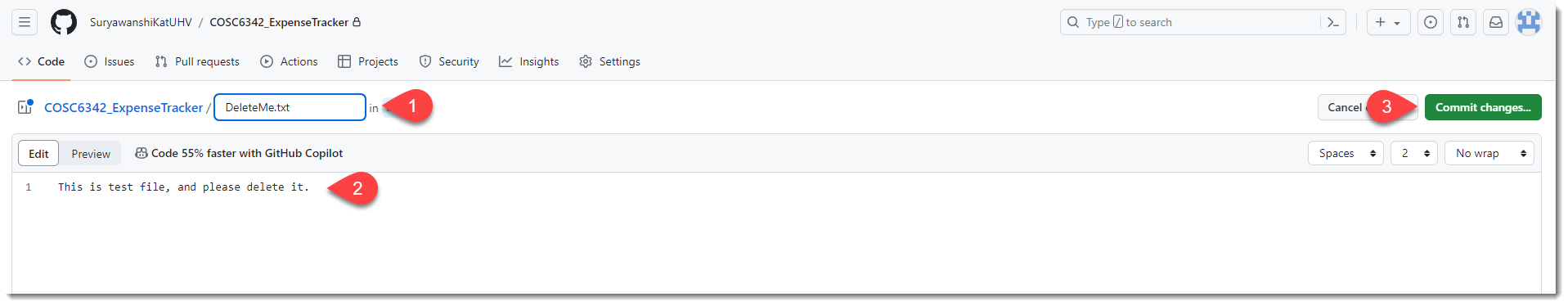
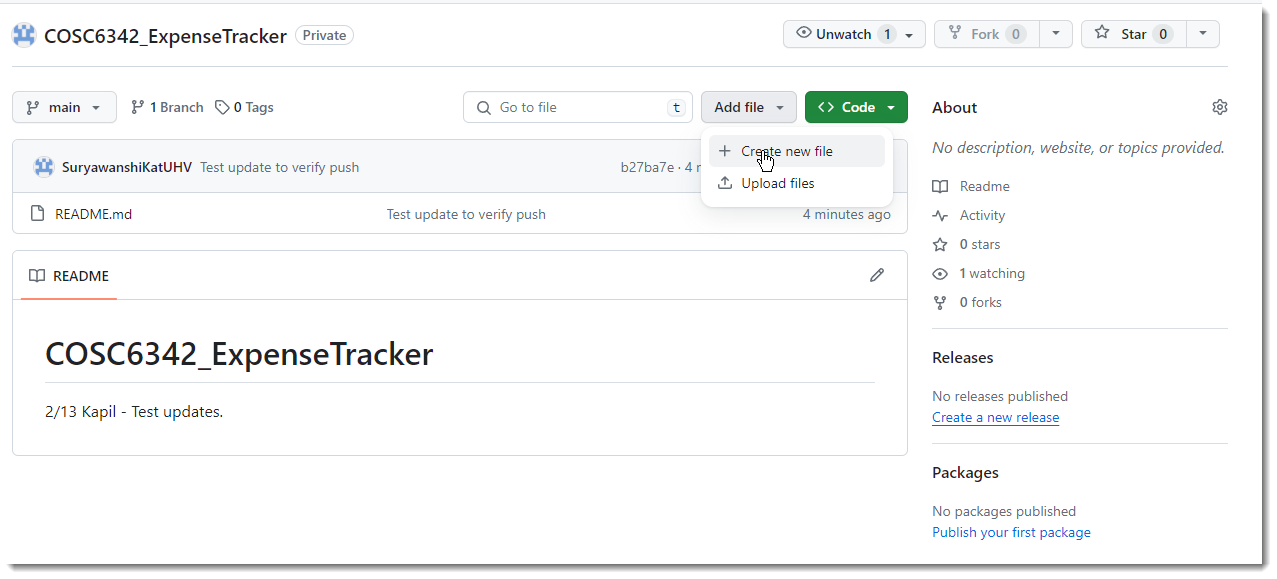




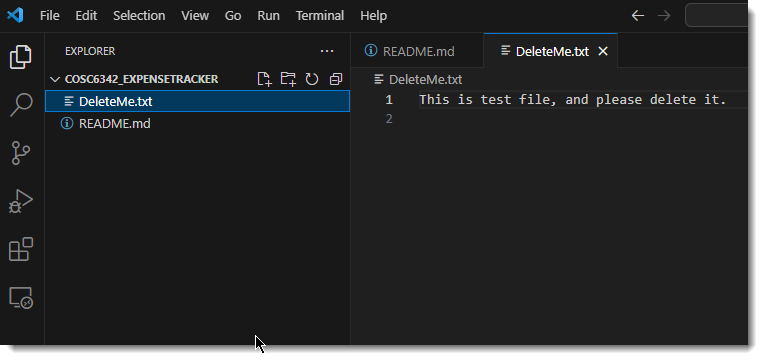
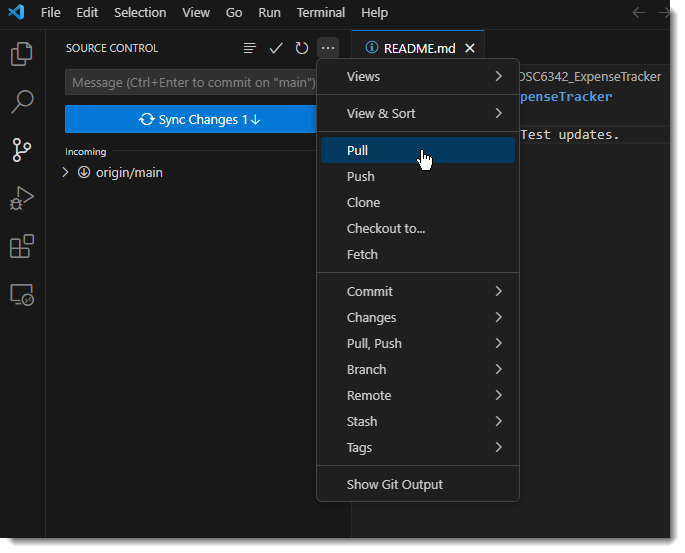
Verify that the changes are pushed into the remote repo by checking the files from Git hub.



Now creating new file in GitHub and verifying it in Visual Studio Code by pulling the remote changes.

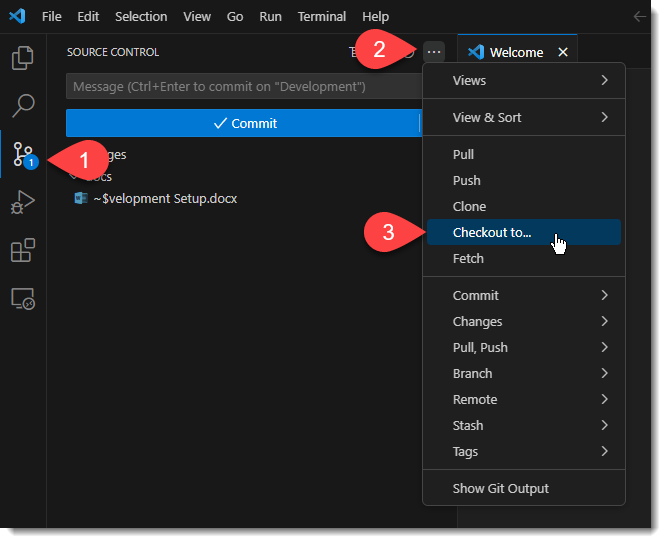


Pull changes in Visual Studio Code and confirm.

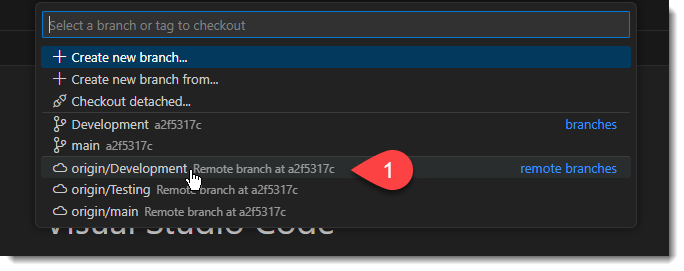


## Working with Git Development branch

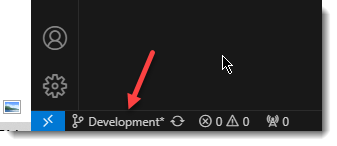
As usual open the Visual Studio Code and Sync the Git repository.



Then checkout the Development branch as shown.

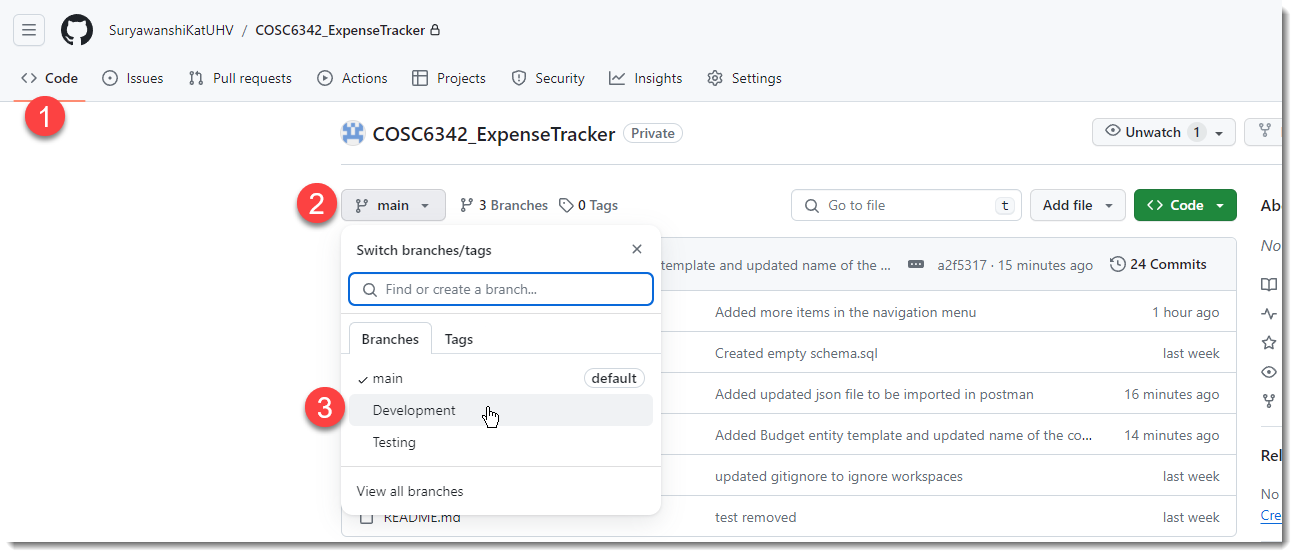


When the VSCode has synced the Development branch then the name of the branch will be shown in the lower status bar.



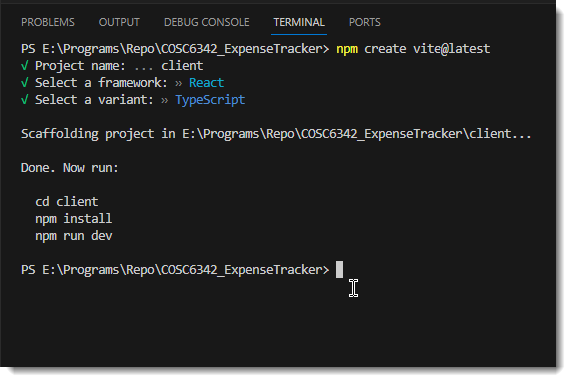
Here onwards whatever changes you commit and push will be in this branch.

To review the contents from GitHub online portal, accordingly select the Development branch.

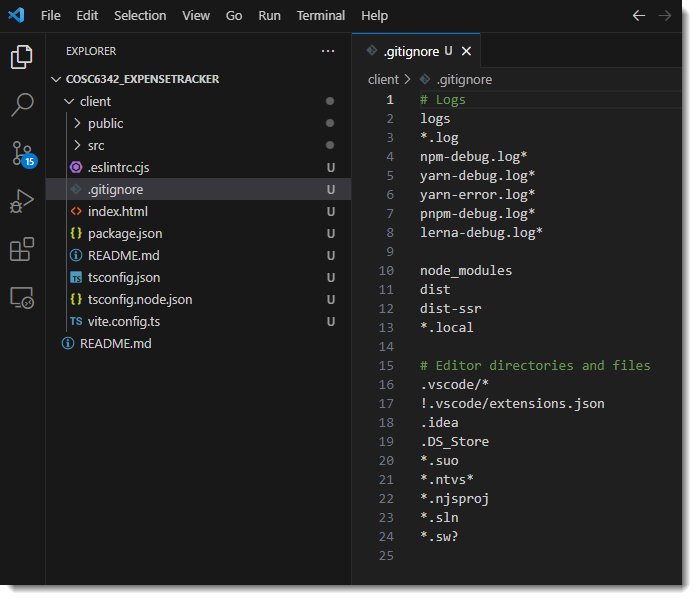


## Setup Front End app

From Visual Studio Code create a react app as below. (I have created the app and pushed it to the Github repo, so you don’t need to to these steps. Only pull the latest code into your local repo.)

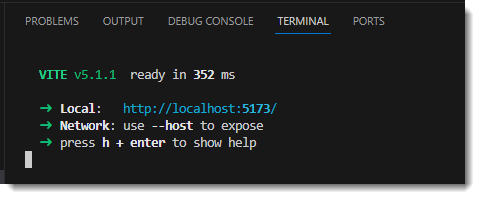


This created a template project for us. Note that it also have a file .gitignore which contains all the files which should not be part of the repo as these are runtime file.



When you would pull the code for the first time, you will need to install all the dependencies locally by executing “npm install” in VSCode terminal inside “client” folder.

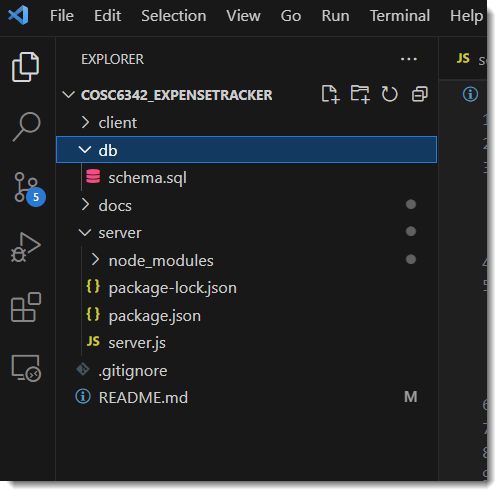
After this is done, execute “npm run dev” to start the server which can host the application in browser as below.





## Setup Back End App

Pull from Visual Studio Code which will sync server and db directories in the local repository.

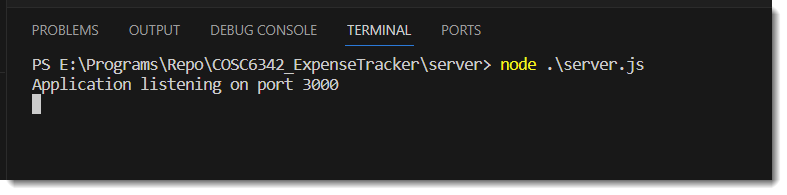


Db directory would hold the SQL script that will be used to create schema and sample data in database.

Docs directory holds documents (non-source code files) related to this application.

Server directory would hold the backend app. At this time I have server.js file created with sample endpoint services using ExpressJS module.

To run the server app, you would navigate to server directory and execute “npm install” to install related packages. Then from the terminal execute “node server.js” to start the server.



Test this server from browser by navigating to any of the following url

<http://localhost:3000/api/users/>

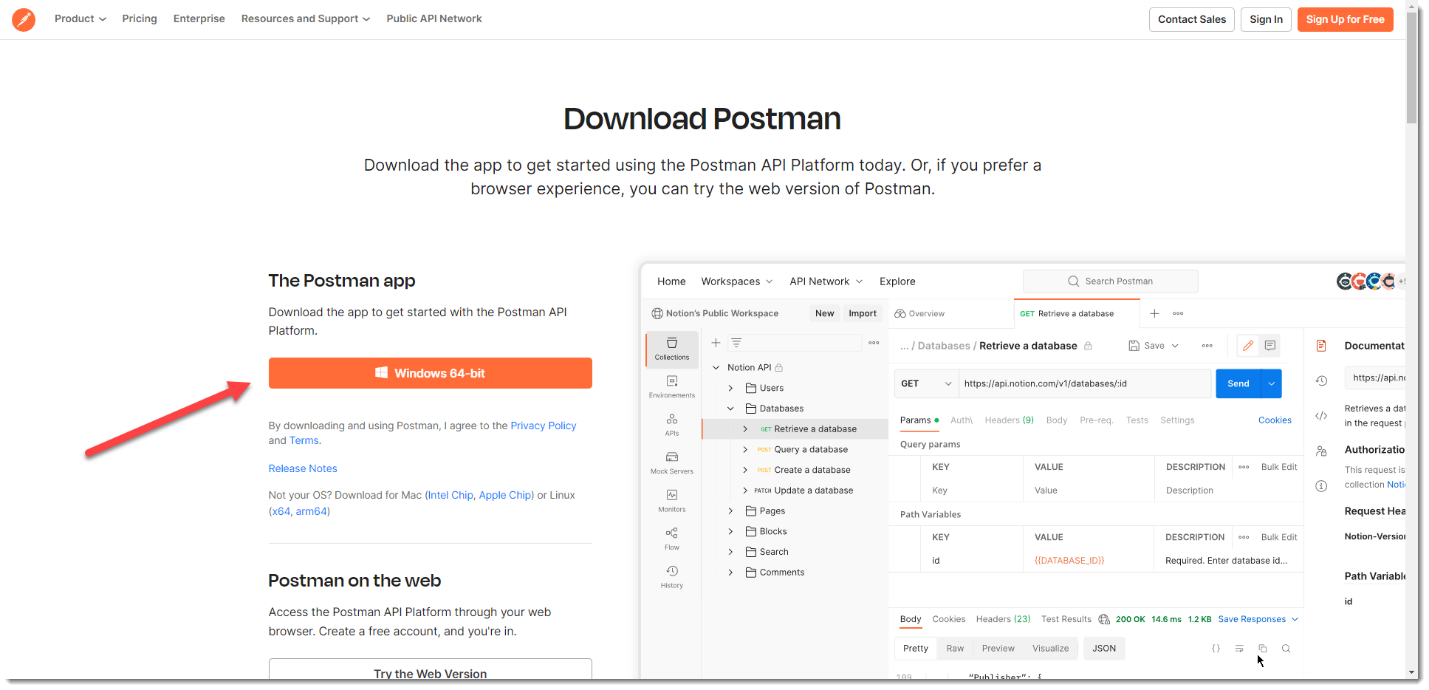
<http://localhost:3000/api/categories/>

<http://localhost:3000/api/expenses/>

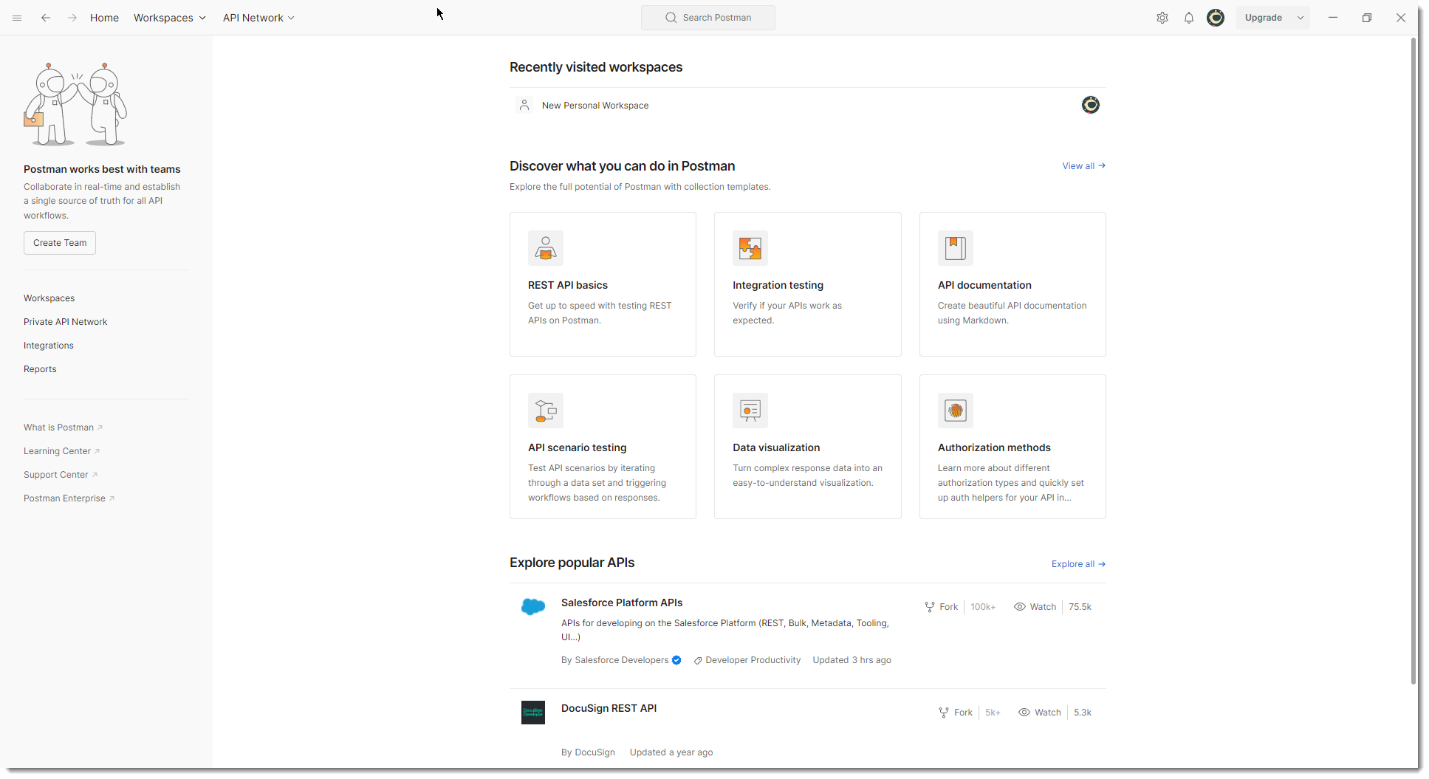
<http://localhost:3000/api/incomes/>

## Install Postman

Navigate to <https://www.postman.com/downloads/>



Download the executable and install the application.

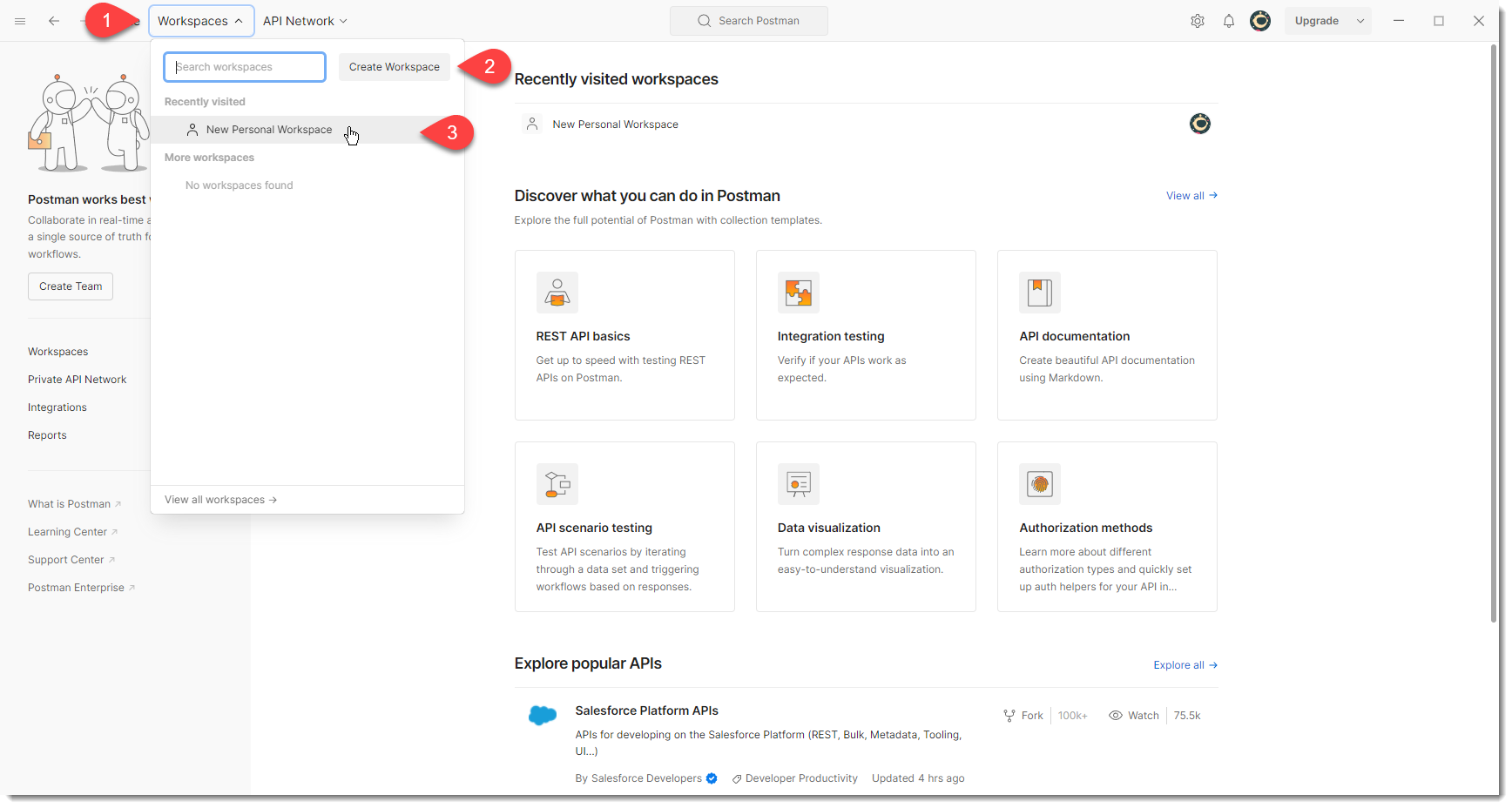


We shall use this tool to test the REST web services that we shall develop in this project.

Download the attached json file below.



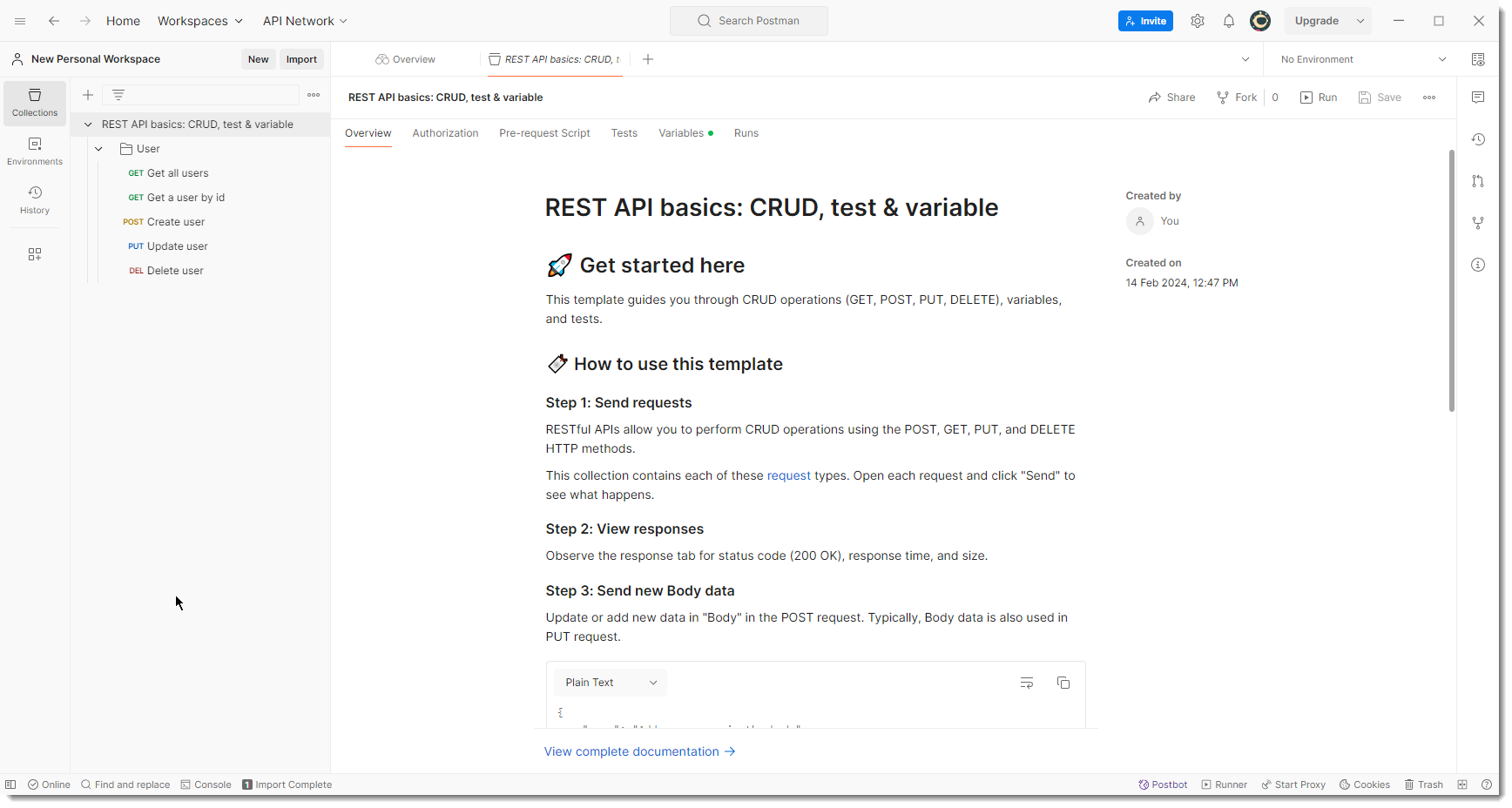
Now create and open a new workspace in Postman as shown below.



Now click import and import this json file in this workspace.

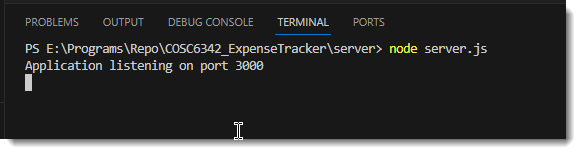


This will import a basic configuration which will be useful to test the user web service in our project.



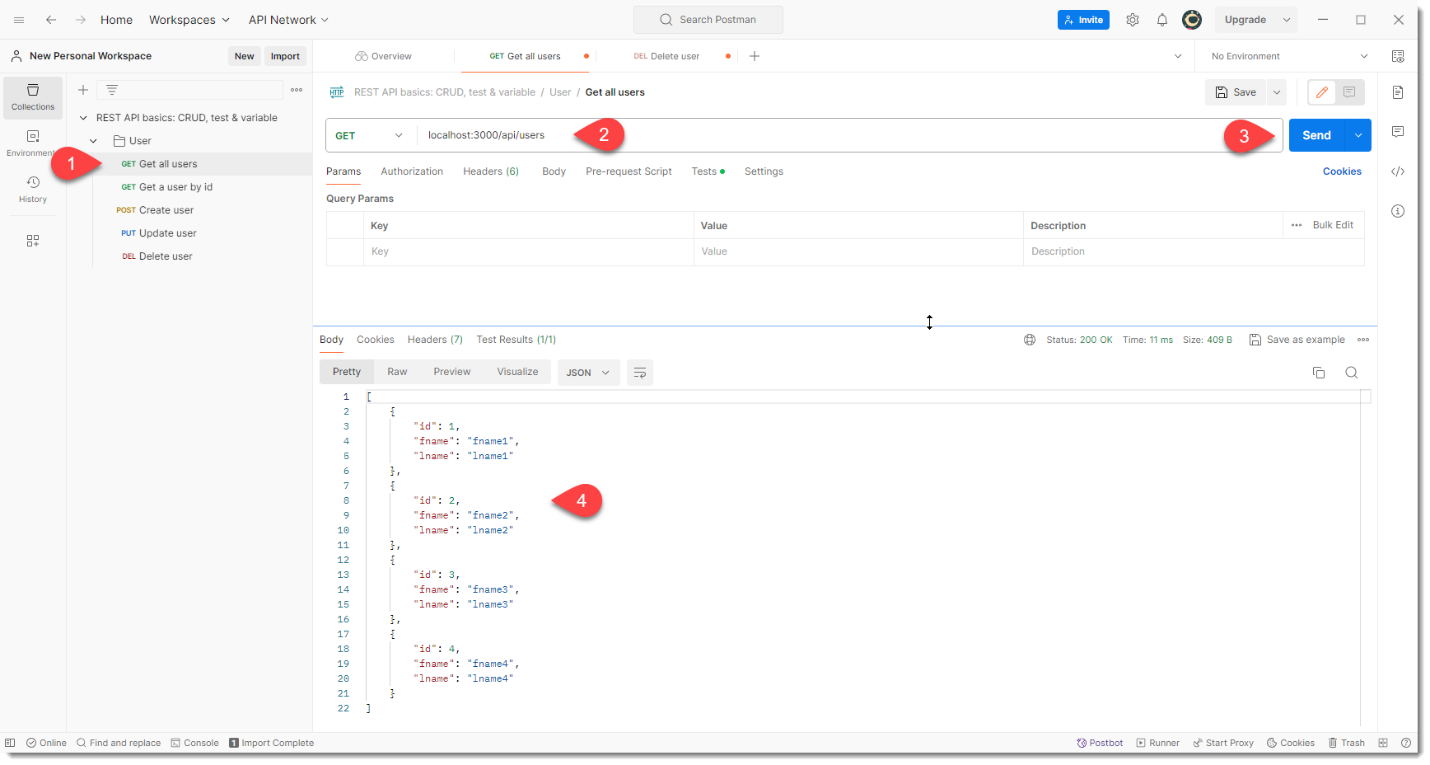
Note: I was expecting that this json file will have all the parameters and URLs I used to test the user web service, but it does not have. Therefore use the following instructions to test each of the CRUD operation.

Pull latest code from Github and then from terminal navigate inside server directory and then start the server.

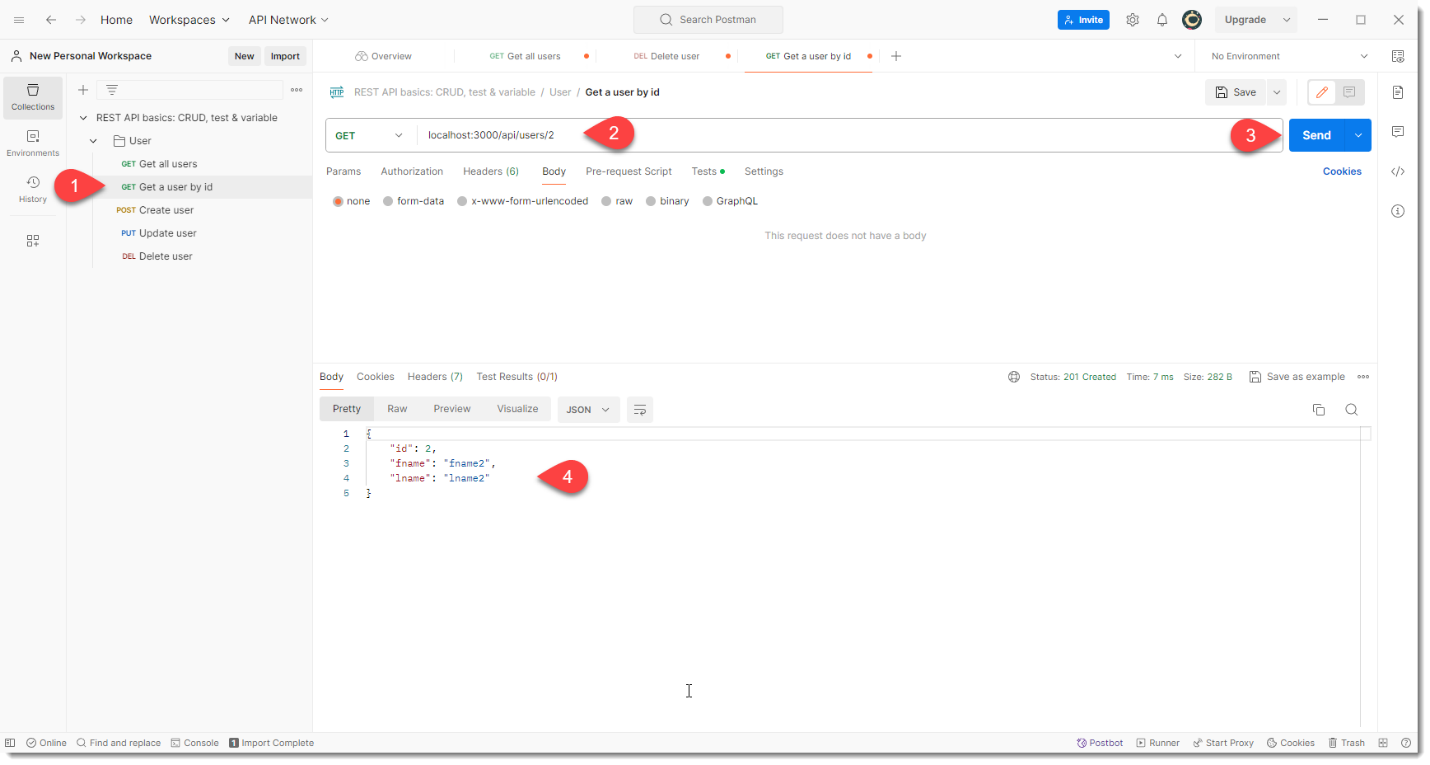


Now follow the rest of the sections to test each of the user web service end points.

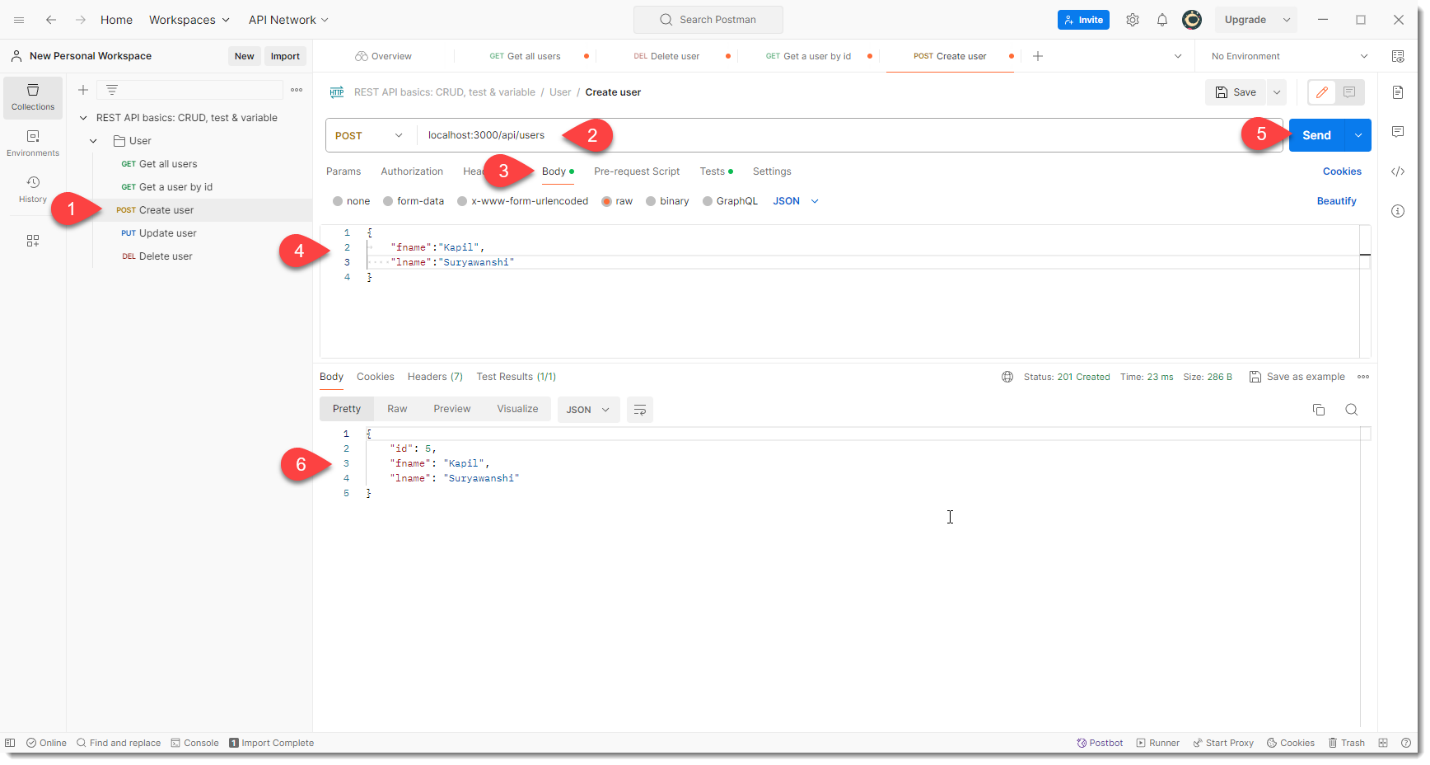
### Get all users



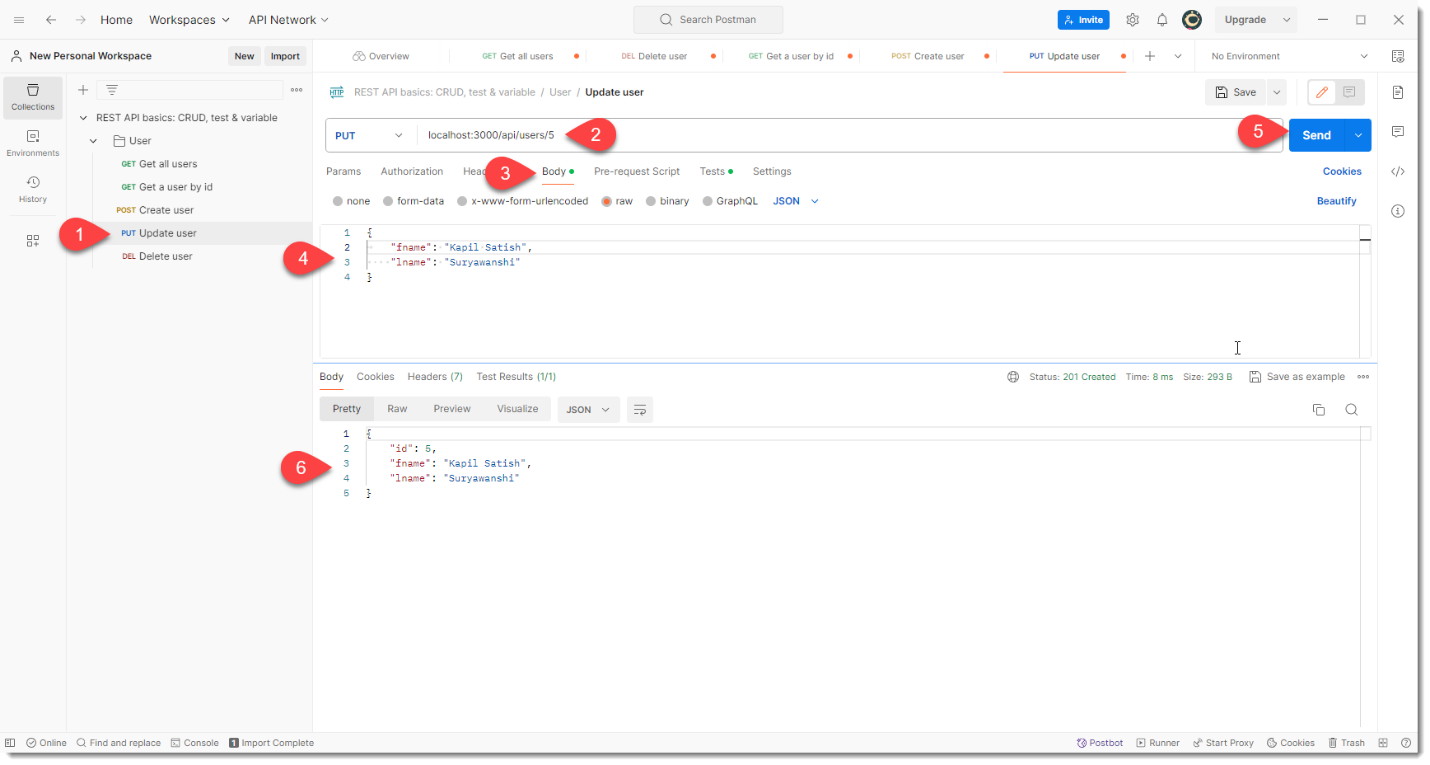
### Get a user by id



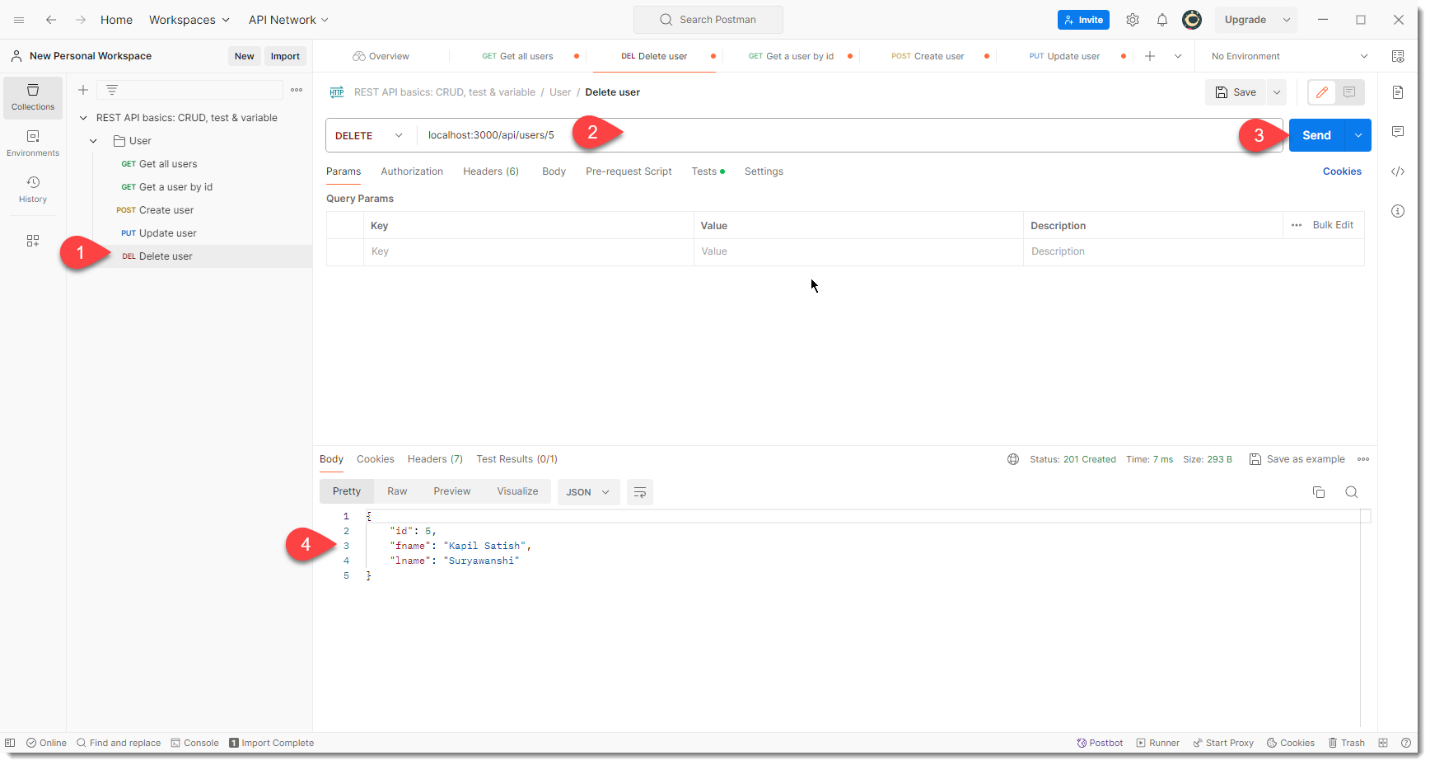
### Create user



### Update user



### Delete user



Now test the “Get all users” case again to verify that this record is deleted.

## About Jest and Supertest for automated unit testing

I had used JUnit for automated unit testing in Java. Which is similar framework for automated unit testing in JavaScript?

In JavaScript, Jest is a popular framework for automated unit testing that is similar to JUnit in Java. Jest is a powerful and flexible testing framework that provides features such as snapshot testing, mocking, and code coverage analysis. It is also easy to set up and use, making it a great choice for testing JavaScript applications and libraries.

Some other popular JavaScript testing frameworks include Mocha, Jasmine, and Chai. These frameworks also provide similar functionality to JUnit and are widely used in the JavaScript community. Ultimately, the choice of which testing framework to use will depend on your specific needs and preferences.

KS

Ok. Now help me to choose one of these framework. My need is to test the expressJs webservice endpoints of my application backend.

For testing Express.js web service endpoints, Jest and Mocha are both great choices.

Jest is a more modern and feature-rich framework, and it provides a lot of useful features out of the box, such as snapshot testing, code coverage analysis, and easy mocking. It also has a large and active community, which means that you can find a lot of resources and help online.

Mocha, on the other hand, is a more established and mature framework that has been around for a longer time. It is highly customizable and flexible, and it provides a lot of options for configuring your tests. Mocha also has a large ecosystem of plugins and extensions that you can use to enhance its functionality.

Both Jest and Mocha can be used with Supertest, which is a library for testing HTTP servers. Supertest allows you to make HTTP requests to your Express.js endpoints and assert the responses.

Ultimately, the choice between Jest and Mocha will depend on your personal preferences and the specific needs of your project. If you prefer a more modern and feature-rich framework, Jest might be the better choice. If you prefer a more established and customizable framework, Mocha might be the better choice. You can also try both frameworks and see which one works better for you.

KS

ok. Give me an example unit test code for in both Jest and Mocha, for the following scenario. You are testing a GET endpoint /api/categories. This would provide response as array of javascript object. Each javascript object in array will have properties id, title and description. You need to invoke this endpoint and ensure 1) that it will return 10 objects 2) each object has title and description.

Here's an example unit test code in Jest for the given scenario:

const request = require('supertest');

const app = require('../app');

describe('GET /api/categories', () => {

it('should return an array of 10 categories', async () => {

const res = await request(app).get('/api/categories');

expect(res.statusCode).toBe(200);

expect(Array.isArray(res.body)).toBe(true);

expect(res.body.length).toBe(10);

res.body.forEach(category => {

expect(category).toHaveProperty('id');

expect(category).toHaveProperty('title');

expect(category).toHaveProperty('description');

});

});

});

And here's an example unit test code in Mocha with Chai for the same scenario:

const chai = require('chai');

const chaiHttp = require('chai-http');

const app = require('../app');

chai.use(chaiHttp);

const expect = chai.expect;

describe('GET /api/categories', () => {

it('should return an array of 10 categories', (done) => {

chai.request(app)

.get('/api/categories')

.end((err, res) => {

expect(res).to.have.status(200);

expect(res.body).to.be.an('array');

expect(res.body).to.have.lengthOf(10);

res.body.forEach(category => {

expect(category).to.have.property('id');

expect(category).to.have.property('title');

expect(category).to.have.property('description');

});

done();

});

});

});

Both of these examples use Supertest to make an HTTP request to the **/api/categories** endpoint, and then use assertions to check that the response contains an array of 10 categories, each with the properties **id**, **title**, and **description**. The Jest example uses the **expect** function provided by Jest, while the Mocha example uses the **expect** function provided by Chai.

KS

I like the Jest and supertest combination.

Great! Jest and Supertest are both popular and powerful tools for testing web applications, and they work well together.

Here's a more detailed breakdown of how the Jest and Supertest combination works:

* Jest is a testing framework that provides a simple and intuitive way to write and run tests. It includes features like code coverage analysis, snapshot testing, and mocking, and it has a large and active community.
* Supertest is a library that allows you to make HTTP requests to your web application and test the responses. It works by wrapping your Express.js server and providing a simple API for making requests and asserting the responses.
* To use Jest and Supertest together, you first need to import the Supertest library and your Express.js app in your test file. Then, you can use the **request** function provided by Supertest to make HTTP requests to your app.
* In your test, you can use the **expect** function provided by Jest to make assertions about the response. For example, you can check that the response has the correct status code, headers, and body. You can also use Jest's snapshot testing feature to automatically generate and compare snapshots of the response.
* Finally, you can run your tests using the **jest** command in your terminal. Jest will automatically discover and run all the tests in your project, and it will provide detailed output and code coverage analysis.

Here's an example of how to use Jest and Supertest to test a simple Express.js app:

const request = require('supertest');

const app = require('../app');

describe('GET /', () => {

it('should return a 200 status code', async () => {

const res = await request(app).get('/');

expect(res.statusCode).toBe(200);

});

it('should return a welcome message', async () => {

const res = await request(app).get('/');

expect(res.text).toBe('Welcome to my app!');

});

});

In this example, we're testing a simple GET endpoint that returns a welcome message. We use Supertest to make a GET request to the endpoint, and then we use Jest to make assertions about the response. We check that the response has a 200 status code and that the body contains the correct message.

Overall, the Jest and Supertest combination provides a powerful and flexible way to test your web application. It allows you to easily write and run tests, and it provides a lot of useful features for testing HTTP requests and responses.

KS

How to execute these tests using jest command line in terminal, give me an example of the command and the sample output that jest would produce.

To execute Jest tests using the command line, you can use the **jest** command in your terminal. Here's an example command:

jest --config jest.config.js

In this example, we're using the **--config** flag to specify the path to our Jest configuration file. This file contains options and settings for Jest, such as the test file patterns, the test environment, and the code coverage threshold.

When you run this command, Jest will automatically discover and run all the tests in your project that match the specified file patterns. By default, Jest looks for files with the **.test.js** or **.spec.js** extension in the current directory and its subdirectories.

Here's an example of the output that Jest might produce:

PASS \_\_tests\_\_/app.test.js

GET /

✓ should return a 200 status code (4 ms)

✓ should return a welcome message (1 ms)

Test Suites: 1 passed, 1 total

Tests: 2 passed, 2 total

Snapshots: 0 total

Time: 1.234 s, estimated 2 s

Ran all test suites.

In this example, Jest has found and run a single test file, **\_\_tests\_\_/app.test.js**. This file contains two tests for a simple GET endpoint that returns a welcome message. Both tests have passed, and Jest has printed a summary of the test results.

The output includes information about the test suites and tests that were run, the number of snapshots that were taken, and the total time taken to run the tests. Jest also provides detailed output for each test, including the test name, the test status, and the time taken to run the test.

Overall, Jest provides a powerful and flexible command-line interface for running tests. It allows you to easily run tests, view test results, and analyze code coverage.

KS

How to install jest in my project?

To install Jest in your project, you can use npm (Node Package Manager) or yarn (Yarn Package Manager). Here are the steps to install Jest using npm:

1. Open your terminal and navigate to your project directory.
2. Run the following command to create a new **package.json** file if you don't already have one:

npm init -y

1. Run the following command to install Jest as a development dependency:

npm install --save-dev jest

1. Create a new file named **jest.config.js** in the root directory of your project. This file will contain the configuration options for Jest. Here's an example configuration file:

module.exports = {

testEnvironment: 'node',

testMatch: ['<rootDir>/\_\_tests\_\_/\*\*/\*.test.js'],

verbose: true,

};

In this example, we're specifying that we want to use the Node.js environment for our tests, that our test files are located in the **\_\_tests\_\_** directory, and that we want to enable verbose output.

1. Create a new directory named **\_\_tests\_\_** in the root directory of your project. This directory will contain your test files.
2. Create a new test file in the **\_\_tests\_\_** directory. The file should have a **.test.js** extension. Here's an example test file:

const request = require('supertest');

const app = require('../app');

describe('GET /', () => {

it('should return a 200 status code', async () => {

const res = await request(app).get('/');

expect(res.statusCode).toBe(200);

});

it('should return a welcome message', async () => {

const res = await request(app).get('/');

expect(res.text).toBe('Welcome to my app!');

});

});

In this example, we're testing a simple GET endpoint that returns a welcome message. We use Supertest to make a GET request to the endpoint, and then we use Jest to make assertions about the response.

1. Run the following command to execute your tests:

npm test

By default, npm will look for a **test** script in your **package.json** file. If you don't have a **test** script, npm will run the **jest** command.

That's it! You've successfully installed Jest and created and executed your first test. You can now continue to write and run tests for your project.

KS

What is the line in code below doing? const app = require('../app');

The line **const app = require('../app');** is importing an Express.js application instance from a module located in a file named **app.js** in the parent directory of the current file.

In Express.js, an application instance is created by calling the **express()** function. This instance is then used to define routes, middleware, and other settings for the application.

By importing the application instance into a test file, you can use it to make HTTP requests to the application using a library like Supertest. This allows you to test the application's endpoints and ensure that they are working as expected.

Here's an example of how the **app** instance might be used in a test file:

const request = require('supertest');

const app = require('../app');

describe('GET /', () => {

it('should return a 200 status code', async () => {

const res = await request(app).get('/');

expect(res.statusCode).toBe(200);

});

it('should return a welcome message', async () => {

const res = await request(app).get('/');

expect(res.text).toBe('Welcome to my app!');

});

});

In this example, we're using the **app** instance to make a GET request to the root endpoint of the application. We then use Jest to make assertions about the response, such as checking that the status code is 200 and that the response body contains a welcome message.

KS

What should I modify in my server.js file so that the jest will have app instance available to execute the tests? ``` require('dotenv').config(); // Loads the environment from .env file const express = require("express"); const app = express(); // Enable json processing app.use(express.json()); // Import routes const userRouter = require("./routers/userRouter"); const categoryRouter = require("./routers/categoryRouter"); const budgetRouter = require("./routers/budgetRouter"); const transactionRouter = require("./routers/transactionRouter"); const groupRouter = require("./routers/groupRouter"); const groupTransactionRouter = require("./routers/groupTransactionRouter"); // Use routes app.get("/", (req, res) => { res.send("Hello from the server"); }); app.use("/api/users", userRouter); app.use("/api/categories", categoryRouter); app.use("/api/budgets", budgetRouter); app.use("/api/transactions", transactionRouter); app.use("/api/groups", groupRouter); app.use("/api/groupTransactions", groupTransactionRouter); // Finally start the server const port = process.env.SERVER\_PORT || 3000; app.listen(port, () => { console.log(`Application listening on port ${port}`); }); ```

To make the **app** instance available for testing with Jest, you need to export it from your **server.js** file. Here's an updated version of your **server.js** file that exports the **app** instance:

require('dotenv').config(); // Loads the environment from .env file

const express = require("express");

const app = express();

// Enable json processing

app.use(express.json());

// Import routes

const userRouter = require("./routers/userRouter");

const categoryRouter = require("./routers/categoryRouter");

const budgetRouter = require("./routers/budgetRouter");

const transactionRouter = require("./routers/transactionRouter");

const groupRouter = require("./routers/groupRouter");

const groupTransactionRouter = require("./routers/groupTransactionRouter");

// Use routes

app.get("/", (req, res) => {

res.send("Hello from the server");

});

app.use("/api/users", userRouter);

app.use("/api/categories", categoryRouter);

app.use("/api/budgets", budgetRouter);

app.use("/api/transactions", transactionRouter);

app.use("/api/groups", groupRouter);

app.use("/api/groupTransactions", groupTransactionRouter);

// Export the app instance

module.exports = app;

// Finally start the server

const port = process.env.SERVER\_PORT || 3000;

app.listen(port, () => {

console.log(`Application listening on port ${port}`);

});

By exporting the **app** instance, you can now import it into your test files using the **require** function. For example, in your test file, you can import the **app** instance like this:

const request = require('supertest');

const app = require('../server'); // Import the app instance from server.js

describe('GET /api/categories', () => {

it('should return an array of 10 categories', async () => {

const res = await request(app).get('/api/categories');

expect(res.statusCode).toBe(200);

expect(Array.isArray(res.body)).toBe(true);

expect(res.body.length).toBe(10);

res.body.forEach(category => {

expect(category).toHaveProperty('id');

expect(category).toHaveProperty('title');

expect(category).toHaveProperty('description');

});

});

});

In this example, we're importing the **app** instance from **server.js** and using it to make a GET request to the **/api/categories** endpoint. We then use Jest to make assertions about the response, such as checking that the status code is 200 and that the response body contains an array of 10 categories, each with the properties **id**, **title**, and **description**.

KS

I got following error while running the tests: ``` PS E:\Programs\Repo\COSC6342\_ExpenseTracker\server> npm test npm ERR! Missing script: "test" npm ERR! npm ERR! To see a list of scripts, run: npm ERR! npm run npm ERR! A complete log of this run can be found in: C:\Users\kyp\AppData\Local\npm-cache\\_logs\2024-04-04T17\_35\_24\_789Z-debug-0.log PS E:\Programs\Repo\COSC6342\_ExpenseTracker\server> npm run PS E:\Programs\Repo\COSC6342\_ExpenseTracker\server> ```

The error message **Missing script: "test"** indicates that there is no **test** script defined in your **package.json** file. The **npm test** command runs the script defined in the **test** property of the **scripts** object in your **package.json** file.

To fix this error, you need to add a **test** script to your **package.json** file. Here's an example **package.json** file with a **test** script that runs Jest:

{

"name": "my-app",

"version": "1.0.0",

"description": "",

"main": "server.js",

"scripts": {

"test": "jest"

},

"keywords": [],

"author": "",

"license": "ISC",

"dependencies": {

"dotenv": "^10.0.0",

"express": "^4.17.3",

"mongoose": "^6.2.7"

},

"devDependencies": {

"jest": "^27.5.1",

"supertest": "^6.2.2"

}

}

In this example, the **test** script is defined as **"jest"**, which runs the Jest test runner.

After you've added the **test** script to your **package.json** file, you can run your tests using the **npm test** command.

KS

before invoking an API for testing, the code must login and obtains a login token, then add this token into the GET request header as Authorization key.

To test APIs that require authentication, you can use Jest and Supertest to perform the login and obtain the authentication token before making the API request. Here's an example of how you can do this:

1. Create a **beforeAll** block in your test file to perform the login and obtain the authentication token. This block will run once before all the tests in the file are executed.

let token;

beforeAll(async () => {

const loginResponse = await request(app)

.post('/api/auth/login')

.send({

email: 'test@example.com',

password: 'testpassword'

});

token = loginResponse.body.token;

});

In this example, we're using Supertest to make a POST request to the **/api/auth/login** endpoint to perform the login. We're sending the login credentials in the request body. The login response contains the authentication token, which we're storing in a variable named **token**.

1. In your test, add the authentication token to the request header using the **set** method provided by Supertest.

it('should return an array of 10 categories', async () => {

const res = await request(app)

.get('/api/categories')

.set('Authorization', `Bearer ${token}`)

.expect('Content-Type', /json/)

.expect(200);

expect(Array.isArray(res.body)).toBe(true);

expect(res.body.length).toBe(10);

res.body.forEach(category => {

expect(category).toHaveProperty('id');

expect(category).toHaveProperty('title');

expect(category).toHaveProperty('description');

});

});

In this example, we're using Supertest to make a GET request to the **/api/categories** endpoint. We're adding the authentication token to the request header using the **set** method. We're also using the **expect** method to set the expected content type and status code. Finally, we're using Jest to make assertions about the response.

By performing the login and obtaining the authentication token before making the API request, you can test APIs that require authentication.

KS

Can I do without supertest module?

Yes, you can test your Express.js application without using the Supertest module. However, Supertest provides a convenient way to test HTTP requests and responses in a simple and concise manner.

If you don't want to use Supertest, you can use the built-in **request** module in Node.js to make HTTP requests to your Express.js application. Here's an example of how you can do this:

const request = require('request');

const app = require('../app');

it('should return an array of 10 categories', (done) => {

const options = {

url: 'http://localhost:3000/api/categories',

method: 'GET',

headers: {

'Authorization': `Bearer ${token}`

}

};

request(options, (error, response, body) => {

expect(response.statusCode).toBe(200);

expect(response.headers['content-type']).toMatch(/json/);

const categories = JSON.parse(body);

expect(Array.isArray(categories)).toBe(true);

expect(categories.length).toBe(10);

categories.forEach(category => {

expect(category).toHaveProperty('id');

expect(category).toHaveProperty('title');

expect(category).toHaveProperty('description');

});

done();

});

});

In this example, we're using the **request** module to make a GET request to the **/api/categories** endpoint. We're passing the authentication token in the request header using the **headers** property. We're also using the **done** callback to signal to Jest that the test is asynchronous.

The response from the server is passed to a callback function, where we can make assertions about the response using Jest. We're using the **expect** function to check the status code, content type, and properties of the response body.

While it's possible to test your Express.js application without using Supertest, Supertest provides a more convenient and expressive way to test HTTP requests and responses. It's worth considering using Supertest to simplify your tests and make them easier to read and maintain.

KS

Jest always gives the following message: ``` Jest has detected the following 1 open handle potentially keeping Jest from exiting: ● TCPWRAP 26 | async function httpPost(url, payload) { 27 | return await request(app) > 28 | .post(url) | ^ 29 | .set('Content-Type', 'application/json') 30 | .set('Authorization', `Bearer ${login\_token}`) 31 | .send(payload); at Test.serverAddress (node\_modules/supertest/lib/test.js:48:35) at new Test (node\_modules/supertest/lib/test.js:34:14) at Object.obj.<computed> [as post] (node\_modules/supertest/index.js:43:18) at post (\_\_tests\_\_/common.js:28:10) at Object.httpPost [as login] (\_\_tests\_\_/common.js:11:28) at Object.login (\_\_tests\_\_/category.test.js:4:16) ``` and then the process does not terminate after executing all the test. How do I terminate the process.

This error message indicates that there is an open handle that is preventing Jest from exiting. In this case, it appears that the open handle is related to a Supertest request.

To fix this issue, you need to make sure that all Supertest requests are properly closed after they are executed. You can do this by calling the **end** method on the Supertest request object, or by using the **afterAll** block in your test file to close any open handles.

Here's an example of how you can use the **afterAll** block to close any open handles:

let server;

beforeAll(() => {

server = require('../app');

});

afterAll(() => {

server.close();

});

describe('GET /api/categories', () => {

// your tests here

});

In this example, we're starting the Express.js server before running the tests using the **beforeAll** block. We're storing the server instance in a variable named **server**. After all the tests have completed, we're using the **afterAll** block to close the server instance using the **close** method.

By closing the server instance after the tests have completed, you can ensure that all open handles are properly closed and that Jest can exit correctly.

If you're still seeing the open handle error after closing the server instance, you may need to investigate other potential causes of the open handle. For example, you may need to close any open database connections or file handles that are used in your tests.

KS

the object returned by express() i.e. server, does not have close method.

You're correct that the object returned by **express()** does not have a **close** method. Instead, you need to close the server instance that is created by calling **app.listen()**.

Here's an updated example that shows how to close the server instance:

let server;

beforeAll(() => {

const app = require('../app');

server = app.listen(3000);

});

afterAll(() => {

server.close();

});

describe('GET /api/categories', () => {

// your tests here

});

In this example, we're starting the Express.js server before running the tests using the **beforeAll** block. We're storing the server instance in a variable named **server**. After all the tests have completed, we're using the **afterAll** block to close the server instance using the **close** method.

By closing the server instance after the tests have completed, you can ensure that all open handles are properly closed and that Jest can exit correctly.

KS

I still have this issue. After calling server.close(), the Jest application does not exit. ``` console.log afterAll() completed. at Object.log (\_\_tests\_\_/category.test.js:30:11) PASS \_\_tests\_\_/category.test.js GET /api/categories √ should return an array of categories (41 ms) Test Suites: 1 passed, 1 total Tests: 1 passed, 1 total Snapshots: 0 total Time: 1.824 s, estimated 2 s Ran all test suites. Jest has detected the following 1 open handle potentially keeping Jest from exiting: ● TCPWRAP 26 | async function httpPost(url, payload) { 27 | return await request(app) > 28 | .post(url) | ^ 29 | .set('Content-Type', 'application/json') 30 | .set('Authorization', `Bearer ${login\_token}`) 31 | .send(payload); at Test.serverAddress (node\_modules/supertest/lib/test.js:48:35) at new Test (node\_modules/supertest/lib/test.js:34:14) at Object.obj.<computed> [as post] (node\_modules/supertest/index.js:43:18) at post (\_\_tests\_\_/common.js:28:10) at Object.httpPost [as login] (\_\_tests\_\_/common.js:11:28) at Object.login (\_\_tests\_\_/category.test.js:8:16) ```

If you're still seeing open handles after calling **server.close()**, it's possible that there are other asynchronous operations that are keeping the process from exiting. One common cause of this issue is unresolved promises.

To ensure that all promises are resolved before Jest exits, you can use the **--detectOpenHandles** option when running Jest. This option will print a warning for any open handles that are detected, and will also wait for all promises to be resolved before exiting.

Here's an example of how to use the **--detectOpenHandles** option:

jest --detectOpenHandles

If there are any open handles that are preventing Jest from exiting, this option will print a warning with information about the handle. You can then investigate the cause of the handle and take appropriate action to close it.

In some cases, you may need to manually resolve any pending promises before calling **server.close()**. You can do this by using the **Promise.all** method to wait for all promises to be resolved before calling **server.close()**.

Here's an example of how to use **Promise.all** to wait for all promises to be resolved:

let server;

beforeAll(() => {

const app = require('../app');

server = app.listen(3000);

});

afterAll(async () => {

// wait for all promises to be resolved

await Promise.all(settledPromises);

server.close();

});

describe('GET /api/categories', () => {

// your tests here

});

In this example, we're using an array named **settledPromises** to keep track of all pending promises. In your test code, you can push any promises that are created into this array. After all the tests have completed, we're using **Promise.all** to wait for all promises to be resolved before calling **server.close()**.

By ensuring that all promises are resolved before calling **server.close()**, you can avoid issues with open handles and ensure that Jest exits correctly.

# Production Environment

# Archived