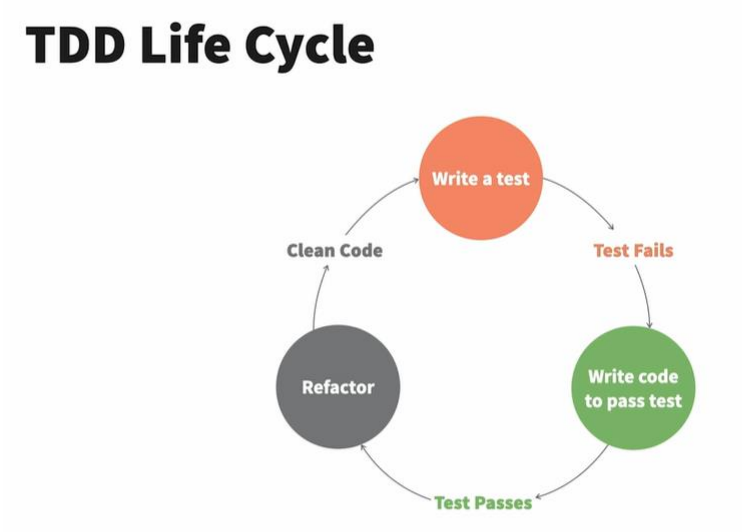
Test-Driven Development (TDD)



Integration Testing Approaches

- Big bang approach
- Incremental approach

In **integration testing**, there are two common approaches to integrating and testing the components or modules of a system: the **Big Bang Approach** and the **Incremental Approach**. Here's an explanation of both:

1. Big Bang Approach

The **Big Bang Approach** to integration testing is when all modules or components of the system are integrated together at once, and then testing is performed on the entire system as a whole.

Characteristics:

- **All components are combined at the same time**: No intermediate testing is done until everything is developed and integrated.
- **Testing occurs only after full integration**: Once the integration is complete, testing begins to see if the components work together as expected.
- **Time-efficient for development but risky**: Since there is no step-by-step integration, this approach can seem quicker in the initial phases, but it poses significant risks.

Advantages:

- **Simple to execute**: There is no need to plan the order of integration.
- **Fast integration**: Developers focus on building the entire system first before testing.

Disadvantages:

- **Difficult to isolate issues**: If an error occurs, it can be challenging to determine which component is causing the issue.
- **Risk of critical failures**: If the components don't work well together, testing and debugging can be complicated, leading to significant delays.

- **Testing is delayed**: Since no testing happens until everything is integrated, bugs may accumulate and become harder to trace and fix.

Best for:

- Projects where all components are relatively independent of each other or where the system is relatively simple.

---

2. Incremental Approach

The **Incremental Approach** to integration testing is when components or modules are integrated one at a time, and testing is done as each new module is integrated. This allows testing to occur incrementally, catching issues as they arise.

Types of Incremental Approaches:

1. **Top-Down Integration**: Modules are integrated starting from the top of the module hierarchy (typically the main module) and proceed downward, one at a time. Stubs are used to simulate lower modules that haven't been integrated yet.
2. **Bottom-Up Integration**: Modules are integrated starting from the lower levels of the hierarchy and moving upwards. Drivers (dummy modules) are used to simulate higher-level modules that haven't been integrated yet.
3. **Hybrid (or Sandwich) Integration**: Combines both top-down and bottom-up approaches, with testing happening in both directions simultaneously.

Characteristics:

- **Step-by-step integration**: Components are added and tested incrementally.
- **Early testing**: Issues are detected early in the integration process, reducing the risk of finding complex bugs later on.
- **Isolated testing**: Makes it easier to isolate and debug specific issues in individual components.

Advantages:

- **Easier to identify and fix bugs**: Errors can be isolated more easily as testing happens after each module is integrated.
- **Less risk of system-wide failure**: Since the system is tested incrementally, there's less risk of encountering large, complex issues.
- **Continuous progress and feedback**: Developers get continuous feedback about the integration process.

Disadvantages:

- **Requires more time and effort**: The incremental process of adding and testing modules can take more time and requires careful planning.
- **Requires the use of stubs and drivers**: If certain components are not ready, testers may have to use stubs or drivers to simulate missing parts, which can complicate the process.

Best for:

- Large or complex systems where components are dependent on each other.
- Systems where early defect detection and isolation are critical.

Summary Comparison:

Aspect	Big Bang Approach	Incremental Approach
Integration Timing	All at once	One module at a time
Testing Timing	After full integration	After each integration step
Error Isolation	Difficult to isolate issues	Easier to isolate issues
Complexity	Simpler to plan	Requires more careful planning
Risk	Higher risk of large failures	Lower risk, as errors are caught early
Usage of Stubs/Drivers	Not required	Required when some components are not available

Both approaches have their own advantages and drawbacks, and the choice between them depends on the nature of the system being tested and the project requirements.

Drivers and Stubs

Drivers and **Stubs** are used in **incremental integration testing** to simulate missing components during the testing of individual modules or parts of a system. They allow for partial testing when some modules are not yet developed or integrated. These are especially useful in **Top-Down** or **Bottom-Up Integration Approaches**.

1. Drivers:
A **Driver** is a piece of code that simulates a calling module or higher-level component that is not yet developed. Drivers are used in **Bottom-Up Integration Testing**, where lower-level modules are integrated and tested first, and the higher-level modules that would normally call them are not yet available.

Purpose:
- **To call the module being tested**: The driver provides the necessary inputs and calls the lower-level module for testing.
- **Simulates control**: The driver mimics the behavior of the parent module (which would call the module being tested), providing the necessary environment for the test.

Example:
Suppose we are testing a function that calculates the total sales in a store, but the main program that calls this function is not yet developed. We create a driver to simulate the main program's behaviour by providing the necessary inputs and invoking the function to test its functionality.

When to use:
- **Bottom-Up Integration**: When lower-level modules are ready, but the higher-level modules that should call them are not.

---

2. Stubs:
A **Stub** is a piece of code that simulates a called module or lower-level component that is not yet developed. Stubs are used in **Top-Down Integration Testing**, where higher-level modules

are tested first, and the lower-level modules that the higher-level module depends on are not yet available.

Purpose:

- **To simulate a lower-level module's response**: The stub provides dummy data or a fixed response when the module being tested calls it.
- **Provides a temporary placeholder**: The stub helps in testing higher-level modules by simulating the behavior of lower-level modules that are still in development.

Example:
Suppose you have developed a function that processes customer orders but haven't yet developed the function to retrieve customer details from a database. You can create a stub to simulate the response from the database, returning mock customer details so that you can test the order processing function.

When to use:

- **Top-Down Integration**: When higher-level modules are ready but lower-level modules that they depend on are not.

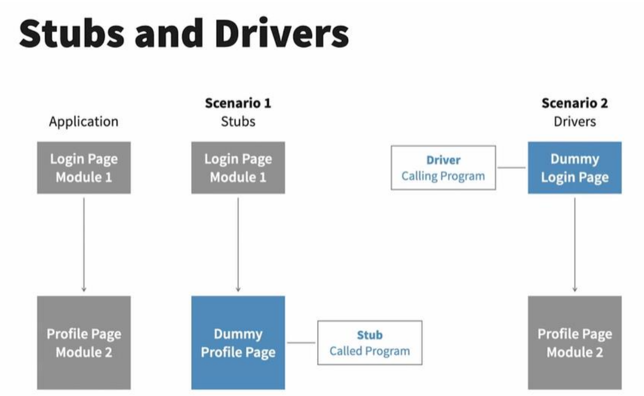
Comparison of Drivers and Stubs:

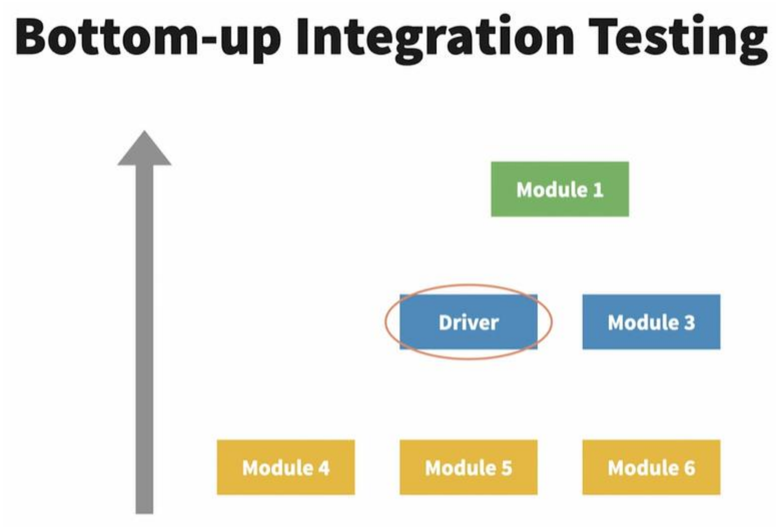
Aspect	Driver	Stub
Purpose	Simulates a higher-level calling module	Simulates a lower-level called module
Used In	Bottom-Up Integration Testing	Top-Down Integration Testing
Role	Invokes the module being tested	Is invoked by the module being tested
Example Usage	Testing lower modules without higher-level modules	Testing higher modules without lower-level modules
Dependency	Simulates the caller	Simulates the callee

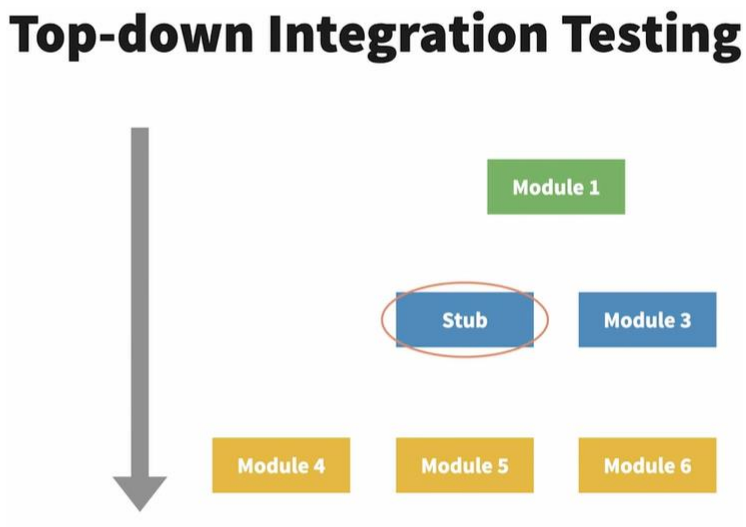
Summary:

- **Drivers** are temporary components that simulate **higher-level modules** and are used to test **lower-level modules**.
- **Stubs** are temporary components that simulate **lower-level modules** and are used to test **higher-level modules**.

Both are essential in incremental testing strategies to allow testing even when all components are not yet available.







Top-down and **Bottom-up** are two popular approaches used in **incremental integration testing**, where components are integrated and tested step by step. Here's a detailed comparison and explanation of both approaches:

---

1. Top-Down Integration Testing

In the **Top-Down Integration Testing** approach, the system is developed and tested starting from the **topmost module (main module)** in the hierarchy, progressively integrating and testing lower-level modules step by step.

Key Characteristics:

- **Starts with the top-level module**: The process begins with the highest-level module in the system hierarchy, usually the "main" function or controller.
- **Uses stubs for missing lower-level modules**: Since lower-level modules may not yet be developed, **stubs** are used as placeholders to simulate their behavior.
- **Progressively replaces stubs with actual modules**: As lower modules are developed, they replace the stubs, and integration testing continues down the hierarchy.
- **Test-driven by control flow**: Testing follows the control flow of the program, starting at the top and moving downward.

Advantages:

- **Early discovery of design issues**: Since testing starts at the top, architectural and control flow issues can be detected early.
- **Demonstrates system-level functionality early**: Early integration of high-level modules allows testers to see the system's behavior early in the process.
- **No need for drivers**: The higher-level modules are already in place to invoke lower-level modules as they are integrated.

Disadvantages:

- **Requires many stubs**: If lower-level modules aren't ready, many stubs must be created to simulate them.
- **Lower-level testing is delayed**: Lower modules may not be tested thoroughly until later in the process, as they are integrated only after the higher modules.
- **Time-consuming to create and maintain stubs**: Developing accurate stubs can be complex and adds overhead.

Best Suited For:

- Systems where the high-level functionality is critical and should be tested first, such as user interfaces or control systems.

---

2. Bottom-Up Integration Testing

In the **Bottom-Up Integration Testing** approach, testing begins with the **lowest-level modules** (i.e., the foundational components), and progressively higher-level modules are integrated and tested.

Key Characteristics:

- **Starts with the lowest-level modules**: Testing begins at the base of the module hierarchy (utility or foundational modules).
- **Uses drivers for missing higher-level modules**: Since the higher-level modules may not be ready, **drivers** are used to simulate the behavior of the calling (higher) modules.
- **Progressively integrates higher modules**: Once the lower modules are integrated and tested, higher modules are gradually added and tested.
- **Test-driven by data flow**: The approach generally follows the data flow, starting from the data-processing modules and moving upward.

Advantages:

- **Thorough testing of lower-level modules**: Since lower modules are tested first, they are thoroughly verified before being integrated into higher-level modules.
- **No need for stubs**: There's no need to simulate lower-level modules, as they are developed and tested first.
- **Easier debugging**: Testing lower modules first makes it easier to isolate and fix issues, as foundational components are tested before integration with higher modules.

Disadvantages:

- **Drivers required**: Test drivers need to be created to simulate higher-level modules, which adds overhead.

- **Late discovery of system-level issues**: Since higher-level modules and the overall control flow of the system are tested later, system-level issues may be discovered late in the process.
- **Top-level functionality is delayed**: The full system functionality, especially at the user interface or control levels, is tested late in the process.

Best Suited For:

- Systems where foundational or utility modules are critical, and their correctness is paramount, such as database or backend-heavy systems.

Comparison of Top-Down vs Bottom-Up Integration Testing:

Aspect	Top-Down Integration Testing	Bottom-Up Integration Testing
Integration Flow	Starts from the top-level module, moving downward	Starts from the bottom-level module, moving upward
Testing Approach	Focuses on testing the control flow early	Focuses on testing the data flow and core functionality first
Test Tools Used	Requires stubs for lower-level modules	Requires drivers for higher-level modules
Discovery of Issues	Detects system-level issues early	Detects foundational module issues early
Focus on System	System functionality is tested early on	Lower-level module functionality is tested first
Advantages	- Early testing of high-level control and UI - No drivers needed	- Early validation of foundational modules - No stubs needed
Disadvantages	- Stubs required for lower modules - Late testing of foundational modules	- Drivers required for higher modules - Late testing of system-level control
Best For	Systems where high-level functionality (e.g., UI, control logic) is critical	Systems where lower-level functionality (e.g., data handling, utilities) is critical

Summary:

- **Top-Down Integration**: Starts with high-level modules and integrates downward. It's ideal for systems where the control flow or user interface is critical, but it requires **stubs** to simulate lower modules.
- **Bottom-Up Integration**: Starts with low-level modules and integrates upward. It's ideal for systems where foundational components are critical, but it requires **drivers** to simulate higher modules.

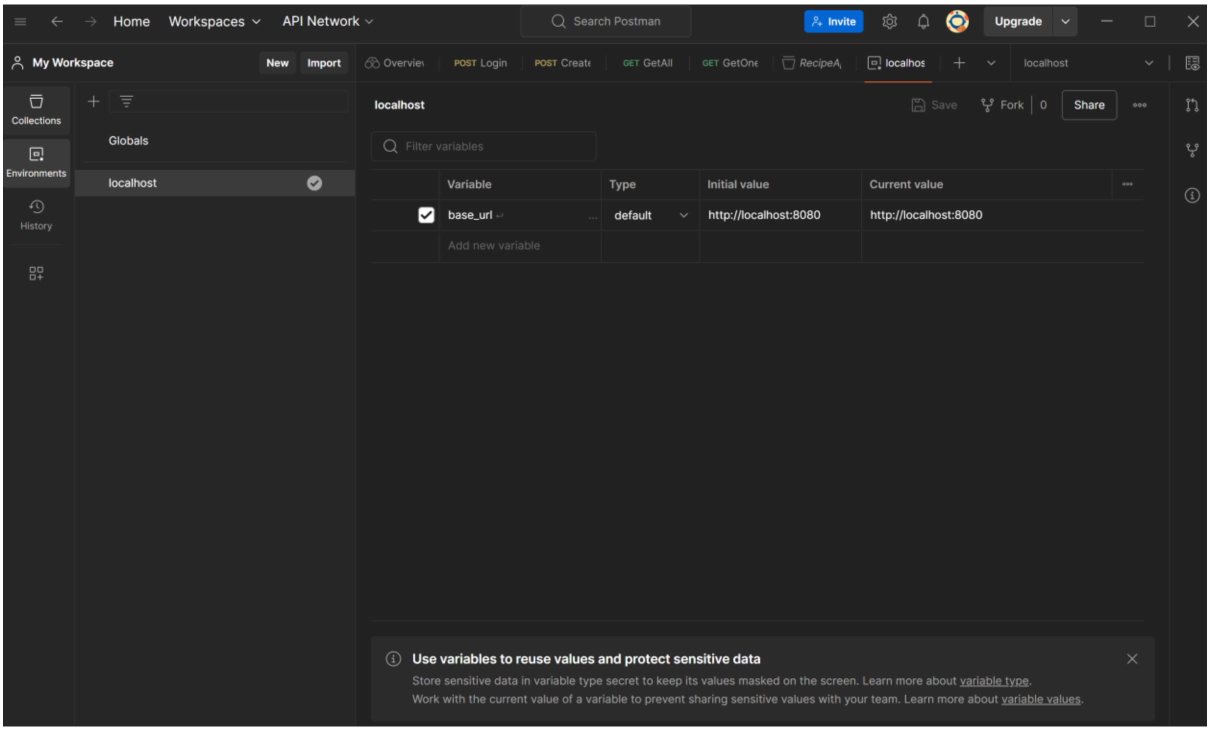
Choosing between **Top-Down** and **Bottom-Up** depends on the system's architecture and what parts of the system are most critical to test first.

Postman

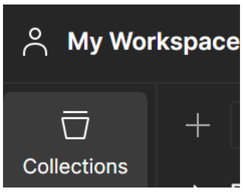
- npm install (Terminal) to install all the dependencies
- Create **.env** file
- npm run seed (Terminal) to seed some dummy data
- npm run start (Terminal) to run the project

- Postman Desktop (if we use localhost, we need this. Otherwise, can use web Postman).
    a. https://www.postman.com/downloads/

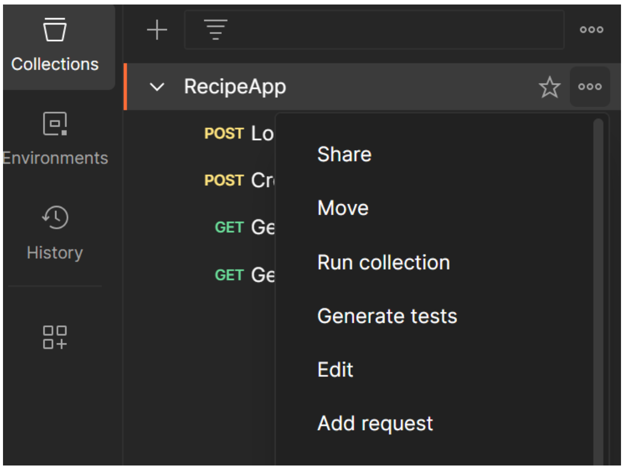
- Create a new environment



- Create a new collection

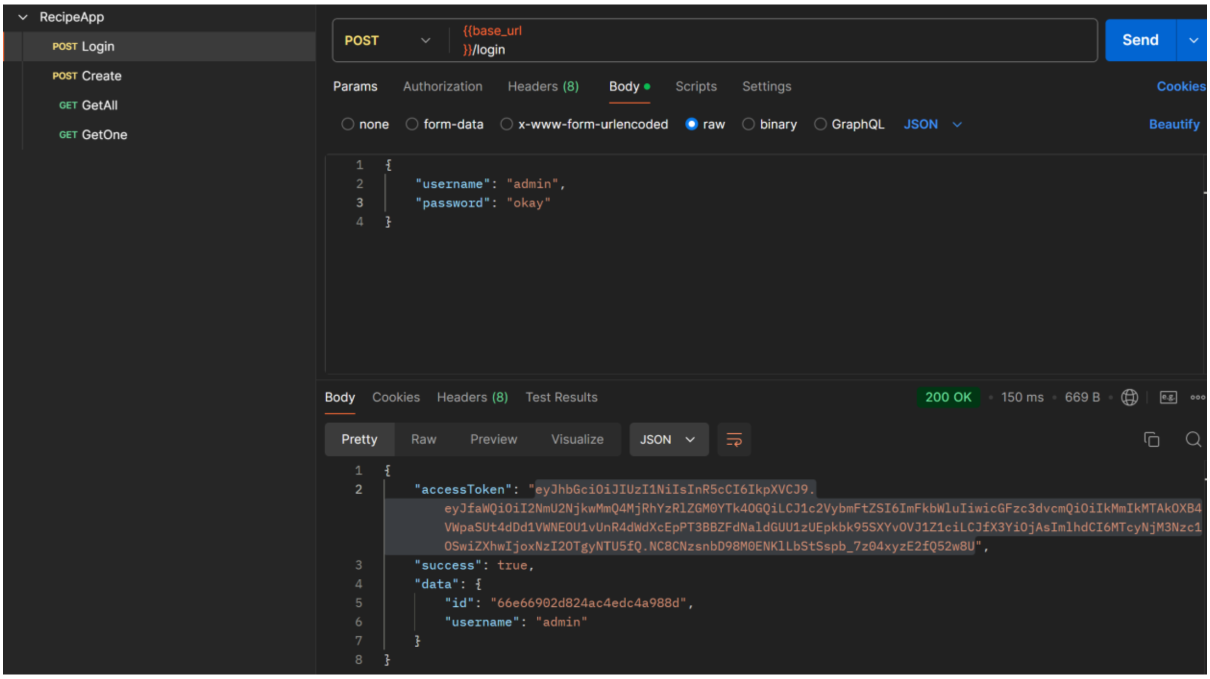


- Create a new request

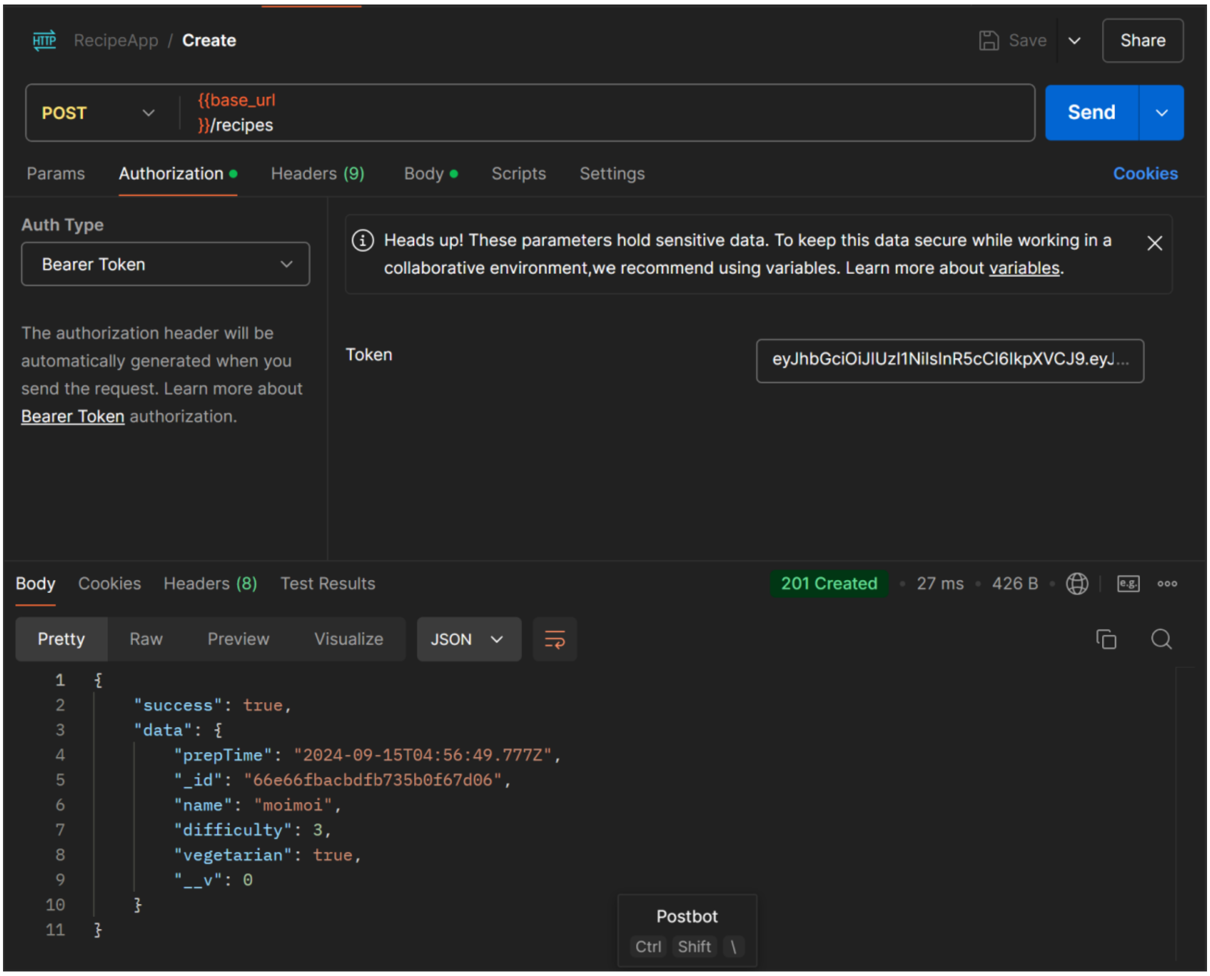


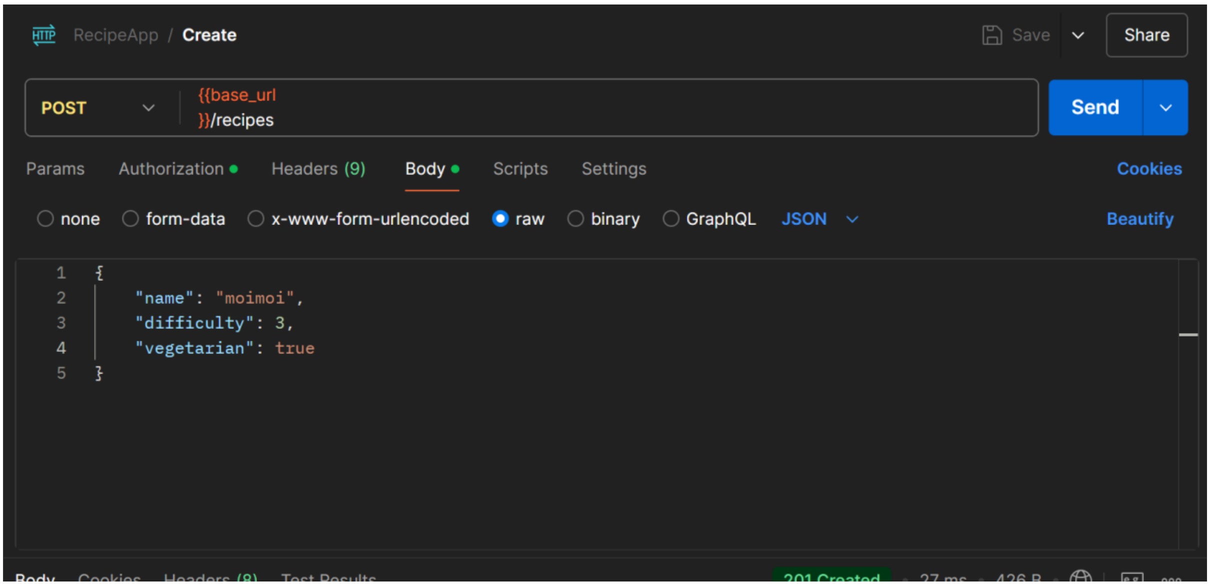
- https://reqres.in/

    a. Login endpoint. Since I was using seed, the default username = admin and password = okay.

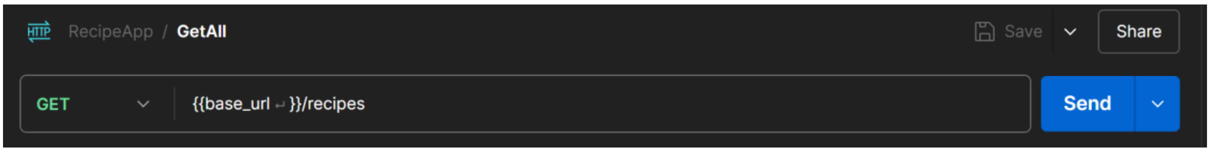


b. Create endpoint. Need AccessToken field (Get from Login endpoint).

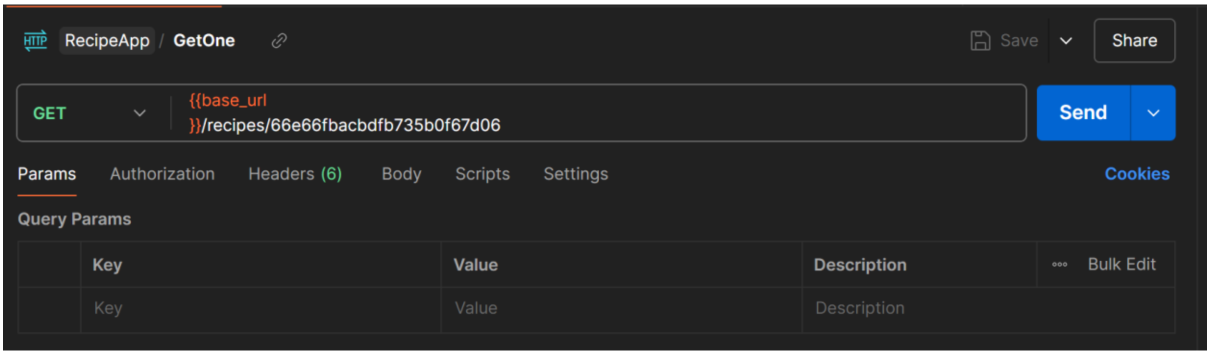




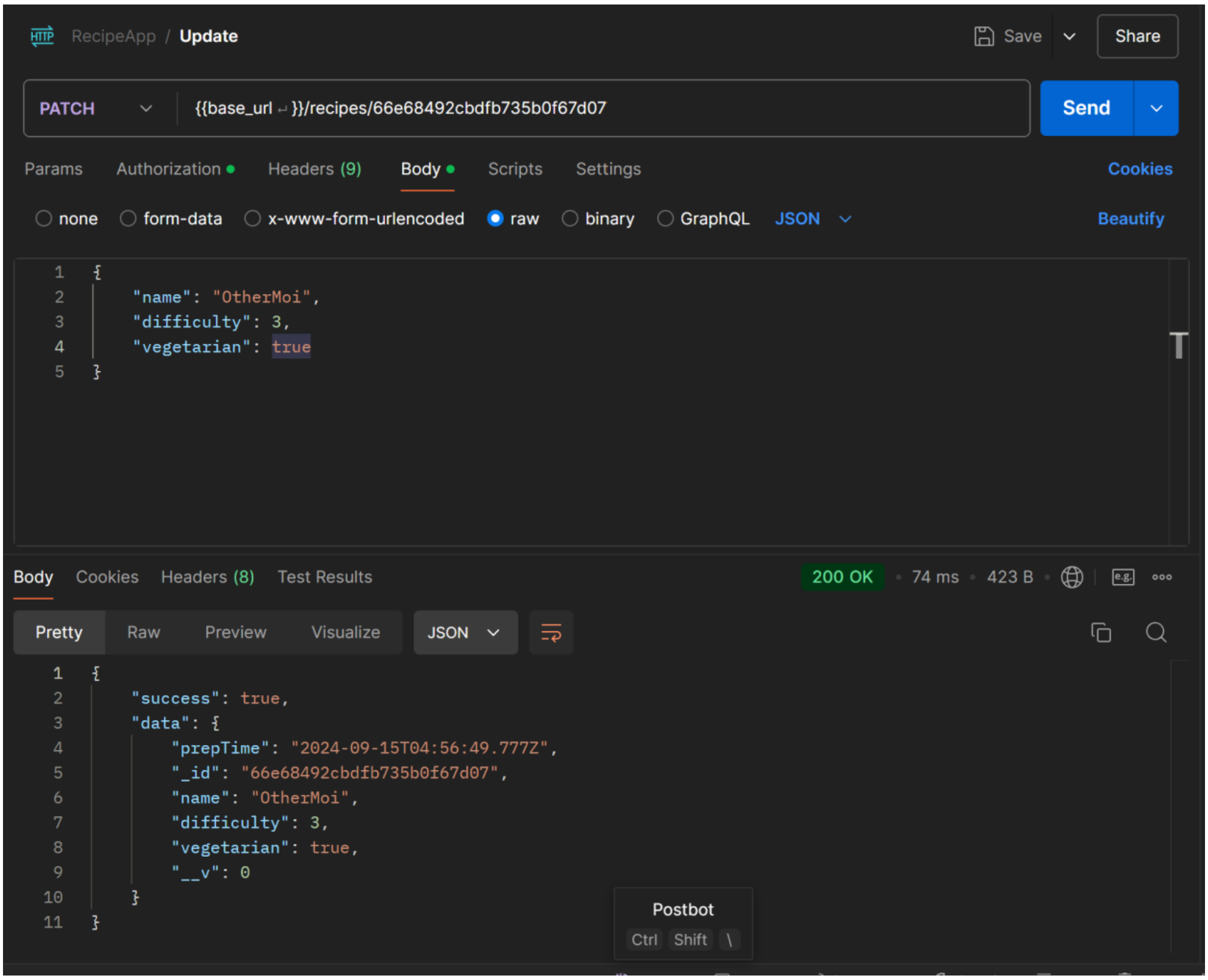
c. GetAll



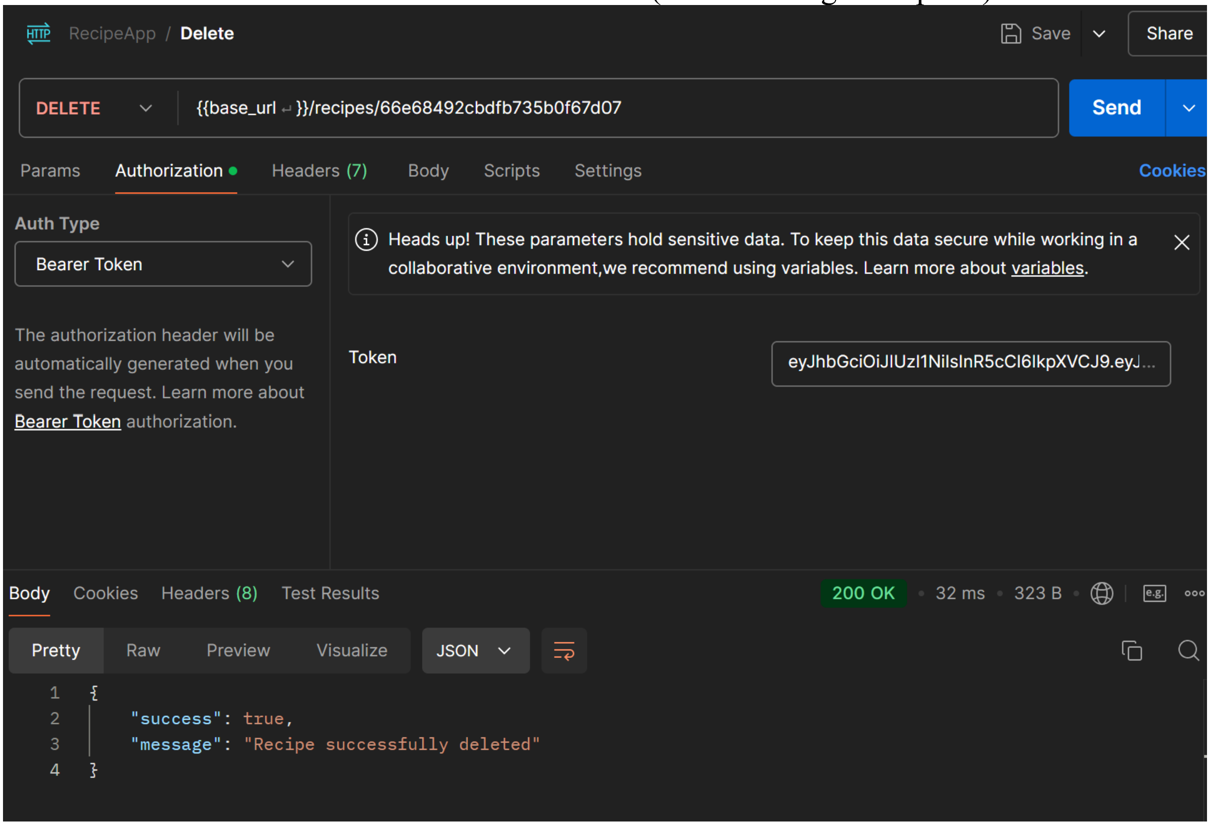
d. GetOne. Need the _id field from GetAll request.



e. PATCH (Partial update) vs PUT (Entire update). Need AccessToken field (Get from Login endpoint).



f. DELETE. Need AccessToken field (Get from Login endpoint).



Introduction to Jest

- **Jest:** a testing library created by Facebook to help test JavaScript code.

Setup Jest for testing RESTful API

- HTTP Requests
  - GET
  - POST
  - PATCH
  - PUT
  - DELETE

- Install Jest and supertest
  - npm install --save-dev jest supertest
  - Add the codes below in the package.json file

```
"jest":{
  "testEnvironment": "node"
},
```

  - npm i -D cross-env
  - Add the codes below in the package.json file

"test": "cross-env DATABASE_URI=mongodb://localhost:27017/recipe_app_test jest --collectCoverge --forceExit --detectOpenHandles"

The script you have provided is a configuration for running tests in a Node.js project, specifically using the **jest** testing framework along with some additional environment setup using **cross-env**. Let's break down each part of the command:

1. "test":
This is the name of the script in your package.json. It specifies what happens when you run npm test or yarn test. In this case, it will execute the command that follows.

2. cross-env DATABASE_URI=mongodb://localhost:27017/recipe_app_test

- cross-env: This is a utility that allows you to set environment variables in a way that works across different operating systems (Windows, macOS, Linux). Normally, setting environment variables can differ between OS, so cross-env makes this consistent.
- DATABASE_URI=mongodb://localhost:27017/recipe_app_test: This is the environment variable being set. It tells the script to use a MongoDB database located at localhost on port 27017, specifically targeting the recipe_app_test database. This variable would be used by your application to connect to this MongoDB instance during tests.

3. jest
This is the command to run the jest testing framework, which is a popular testing tool for JavaScript and Node.js projects. It handles unit tests, integration tests, and can generate test coverage reports.

4. --collectCoverage
This flag instructs jest to collect test coverage information during the test run. It generates a report that shows which parts of your codebase are covered by tests and which are not. This is helpful for identifying untested areas of your application.

5. --forceExit
This flag forces jest to exit after all tests are done, even if there are asynchronous operations still running in the background. This is useful if you have long-running processes (such as open database connections) that might prevent the tests from finishing properly.

6. --detectOpenHandles
This flag helps to detect open handles (such as open database connections or pending timers) that might be preventing jest from finishing execution. If jest hangs or doesn't exit properly, this flag will help track down what might be causing that.

Summary
This script sets up the environment with a MongoDB URI, runs tests using jest, collects test coverage, and ensures that the tests exit properly even if there are open handles or background processes. This script is likely used for running tests in a Node.js application that interacts with a MongoDB database during the test process.

  - Refactor index.js (Outer layer) [Compare with index copy.js]
  - Move config variable and app.listen method to new server.js file

The statement require('dotenv'); in your code is used to load environment variables from a .env file into your Node.js application.

Test the Login endpoint

Jest Global Methods

- beforeAll() and afterAll()
- describe()
- test() or it()
- expect()

Matcher Methods

- toEqual()
- objectContaining()