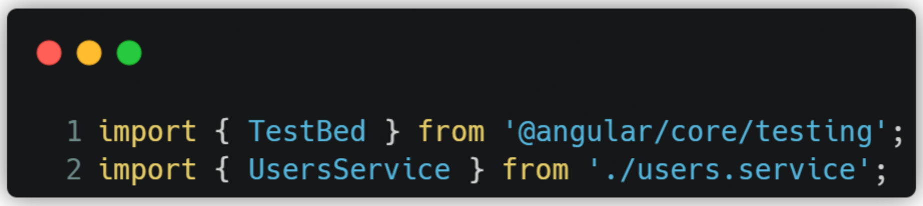
Unit Tests and Integration Tests

User Login

The following code snippets are written in JavaScript, and use the Angular framework for testing a service called `**UsersService**`

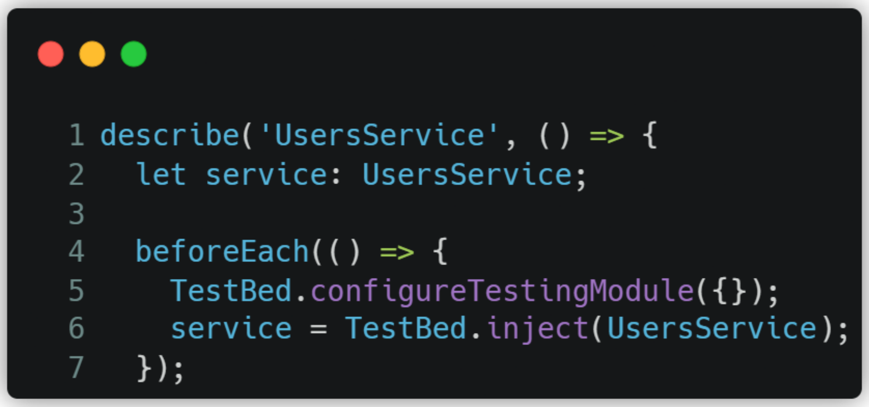
1. Importing Dependencies:



```
import { TestBed } from '@angular/core/testing';
import { UsersService } from './users.service';
```

In the first line, the `**TestBed**` module is imported from the Angular's `**core/testing**` package. This module provides utilities for configuring and creating a test module where Angular components, services, and other dependencies can be tested. The `**UsersService**` is imported from a separate file that contains the implementation of the service being tested.

2. Test Suite Initialization:



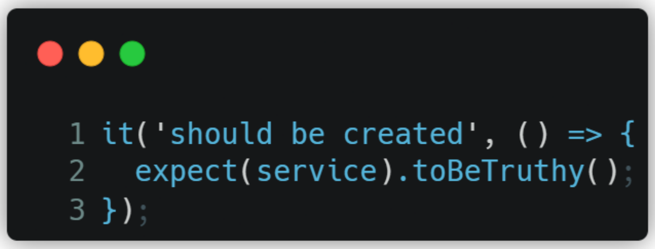
```
describe('UsersService', () => {
  let service: UsersService;

  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(UsersService);
  });
```

The `**describe**` function defines a test suite with the name `**UsersService**`. Inside the test suite, a variable `**service**` of type `**UsersService**` is declared to hold an instance of the service being tested.

The `**beforeEach**` function is a setup function that is executed before each test case in the suite. In this case, it configures the TestBed module by calling `**TestBed.configureTestingModule({})**`. The empty object passed to the `**configureTestingModule**` function indicates that no additional configurations or dependencies are required for this test.

The next line `**service = TestBed.inject(UsersService);**` retrieves an instance of the `**UsersService**`from the `**TestBed**`. This allows the tests to access and manipulate the service.

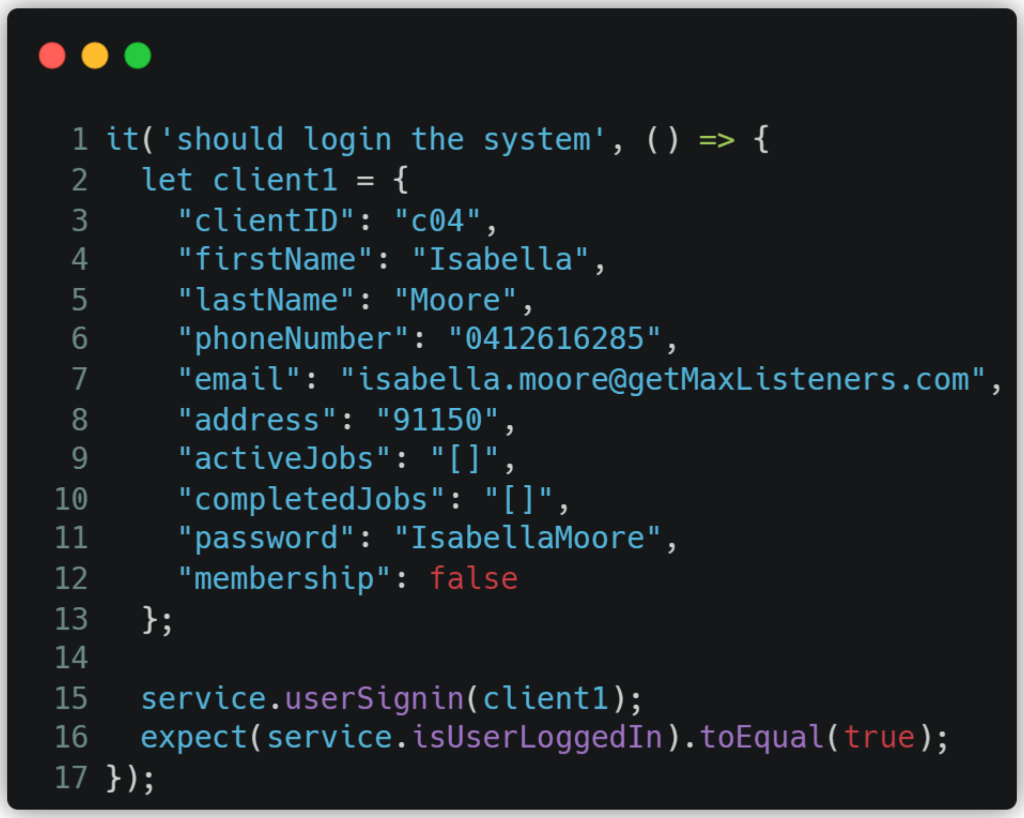
3. Unit Test: 'should be created'



```
it('should be created', () => {
  expect(service).toBeTruthy();
});
```

This is a unit test case that verifies whether the `**UsersService**` is created successfully. The `**it**` function defines an individual test case with a description. In this case, it checks if the `**service**` object is truthy (i.e., not null or undefined), indicating that the service was successfully instantiated. The `**expect**` statement with `**toBeTruthy()**` matcher verifies this condition.

4. Integration Test: 'should login the system'



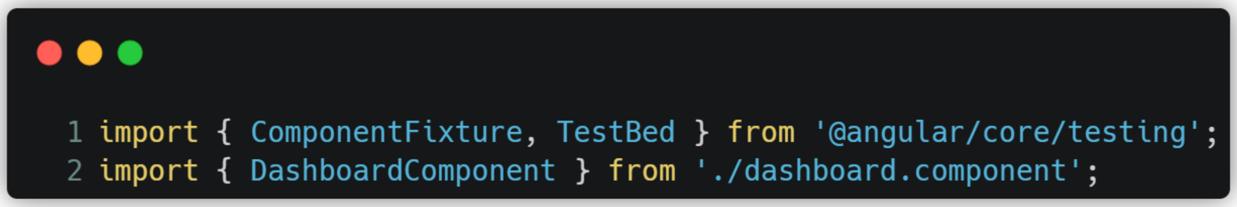
```
it('should login the system', () => {
  let client1 = {
    "clientID": "c04",
    "firstName": "Isabella",
    "lastName": "Moore",
    "phoneNumber": "0412616285",
    "email": "isabella.moore@getMaxListeners.com",
    "address": "91150",
    "activeJobs": "[]",
    "completedJobs": "[]",
    "password": "IsabellaMoore",
    "membership": false
  };

  service.userSignin(client1);
  expect(service.isUserLoggedIn).toEqual(true);
});
```

This is an integration test case that checks the behavior of the `**userSignin**` method in the `**UsersService**`. It simulates a user login by creating a `**client1**` object representing user data. The `**userSignin**` method of the service is then called with `**client1**` as an argument.

The `**expect**` statement with `**toEqual(true)**` verifies that the `**isUserLoggedIn**` property of the `**UsersService**`is set to `**true**` after calling `**userSignin**`. If the login is successful, the expectation will pass.

Adding Items to The Cart

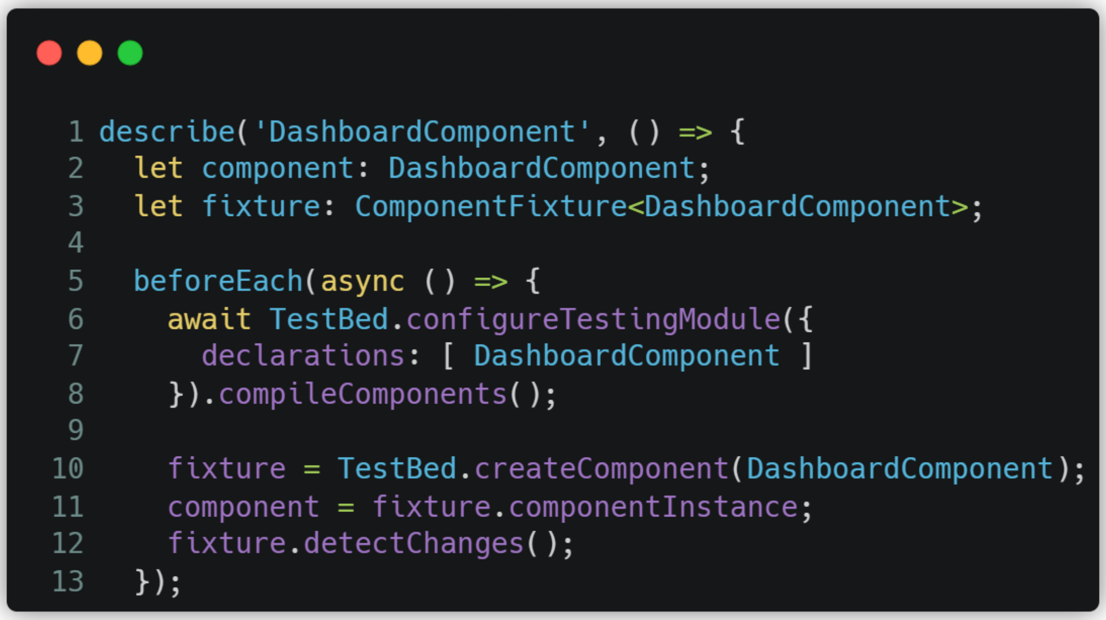
1. Importing Dependencies



```typescript
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { DashboardComponent } from './dashboard.component';
```

The necessary dependencies for testing Angular components, specifically `**ComponentFixture**` and `**TestBed**`, are imported. The `**DashboardComponent**` is imported from a separate file that contains the implementation of the component being tested.

2. Test Suite Initialization:



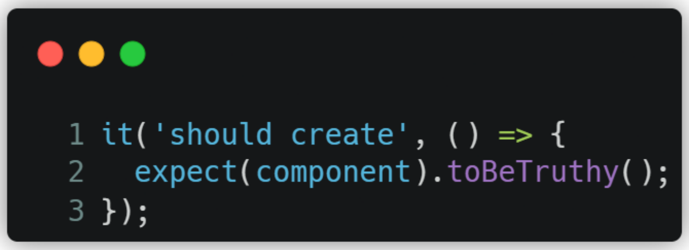
```typescript
describe('DashboardComponent', () => {
  let component: DashboardComponent;
  let fixture: ComponentFixture<DashboardComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ DashboardComponent ]
    }).compileComponents();

    fixture = TestBed.createComponent(DashboardComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
```

The `**describe**` function defines a test suite with the name **'DashboardComponent'**. Inside the test suite, two variables are declared: `**component**` to hold an instance of the DashboardComponent and `**fixture**` to hold a `**ComponentFixture**` instance for the component.

The `**beforeEach**` function is a setup function that is executed before each test case in the suite. In this case, it configures the `**TestBed**` module by calling `**TestBed.configureTestingModule({})**` and specifies the `**DashboardComponent**` as a declared component for testing.

The `**compileComponents**` function compiles the component's template and styles asynchronously. Then, `**TestBed.createComponent(DashboardComponent)**` creates an instance of the component, which is assigned to the `**fixture**` variable. Finally, `**fixture.componentInstance**` assigns the `**component**` instance to the component variable, and `**fixture.detectChanges()**` triggers change detection.

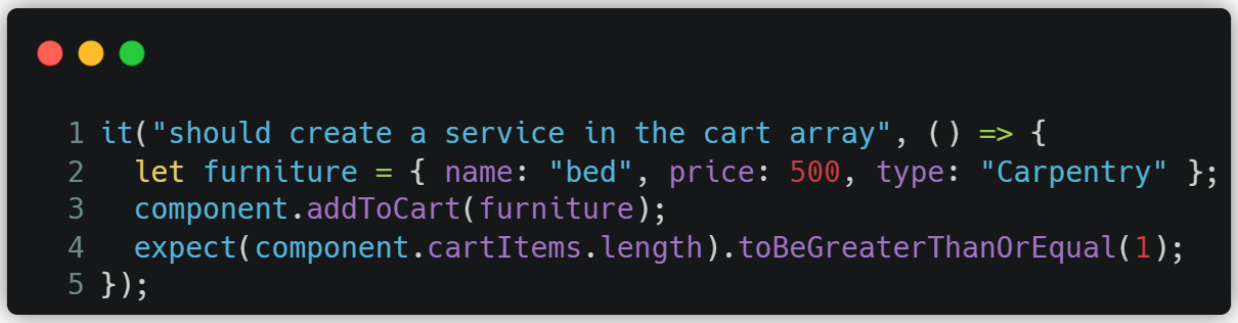
3. Unit Test: ‘should create’



```
it('should create', () => {
  expect(component).toBeTruthy();
});
```

This unit test case verifies whether the `**DashboardComponent**` is created successfully. The `**it**` function defines an individual test case with a description. In this case, it checks if the `**component**` object is truthy (i.e., not null or undefined), indicating that the component was successfully instantiated. The `**expect**` statement with `**toBeTruthy()**` matcher verifies this condition.

4. Integration Test: ‘should create a service in the cart array’



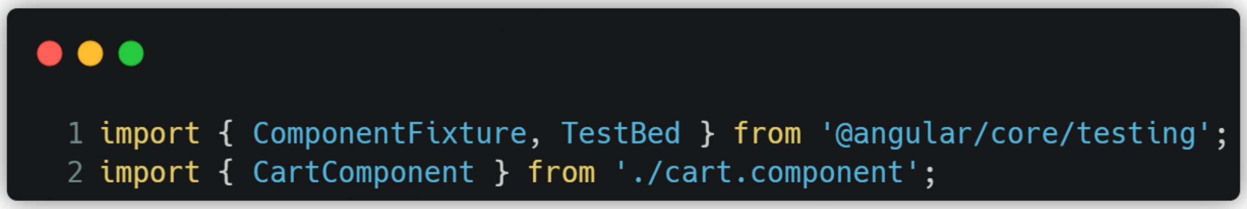
```
it("should create a service in the cart array", () => {
  let furniture = { name: "bed", price: 500, type: "Carpentry" };
  component.addToCart(furniture);
  expect(component.cartItems.length).toBeGreaterThanOrEqual(1);
});
```

This is an integration test case that checks the behavior of the `**addToCart**` method in the `**DashboardComponent**`. It simulates adding a service (`**furniture**` object) to the cart by calling the `**addToCart**` method of the component with the furniture object as an argument.

The `**expect**` statement with `**toBeGreaterThanOrEqual(1)**` matcher verifies that the length of the `**cartItems**` array in the component is greater than or equal to 1 after calling `**addToCart**`. If the service is successfully added to the cart, the expectation will pass.

Removing Items From The Cart

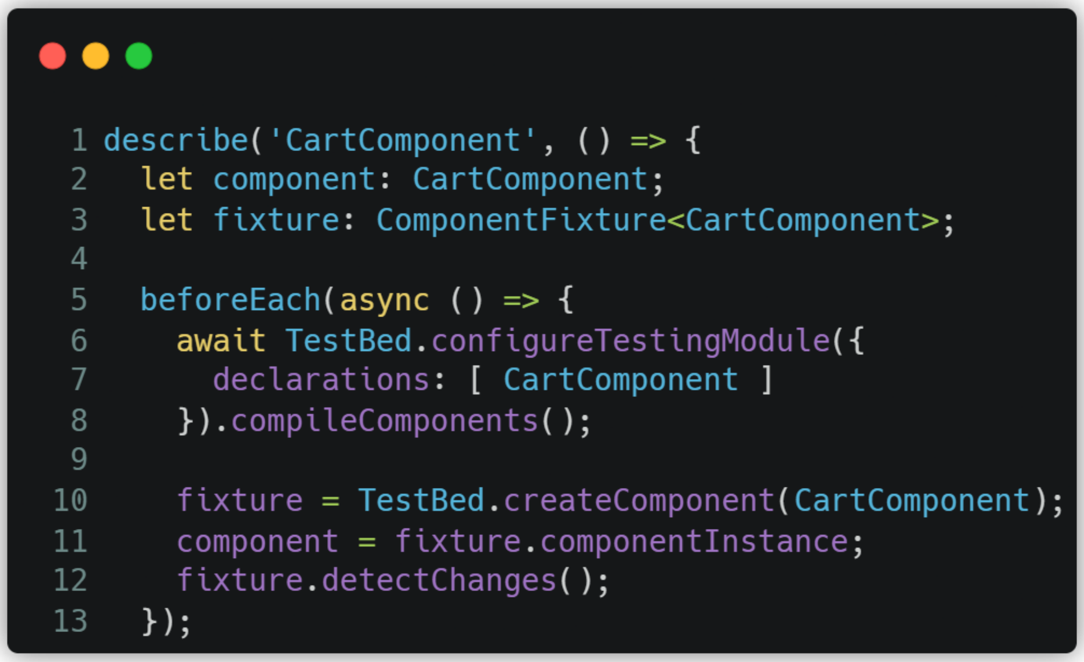
1. Importing Dependencies:



```typescript
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { CartComponent } from './cart.component';
```

The necessary dependencies for testing Angular components, specifically `**ComponentFixture**` and `**TestBed**`, are imported. The `**CartComponent**` is imported from a separate file that contains the implementation of the component being tested.

2. Test Suite Initialization:



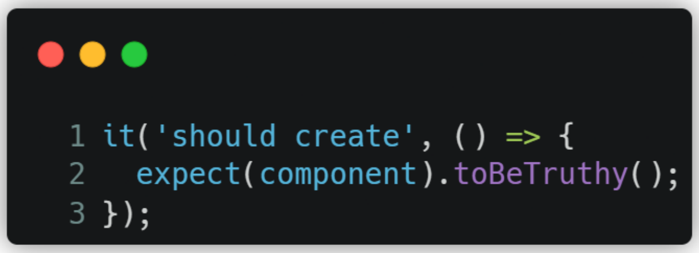
```typescript
describe('CartComponent', () => {
  let component: CartComponent;
  let fixture: ComponentFixture<CartComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [ CartComponent ]
    }).compileComponents();

    fixture = TestBed.createComponent(CartComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
```

The `**describe**` function defines a test suite with the name **'CartComponent'**. Inside the test suite, two variables are declared: `**component**` to hold an instance of the `**CartComponent**` and `**fixture**` to hold a `**ComponentFixture**` instance for the component.

The `**beforeEach**` function is a setup function that is executed before each test case in the suite. In this case, it configures the `**TestBed**` module by calling `**TestBed.configureTestingModule({})**` and specifying the `**CartComponent**` as a declared component for testing.

The `**compileComponents**` function compiles the component's template and styles asynchronously. Then, `**TestBed.createComponent(CartComponent)**` creates an instance of the component, which is assigned to the fixture variable. Finally, `**fixture.componentInstance**` assigns the `**component**` instance to the component variable, and `**fixture.detectChanges()**` triggers change detection.

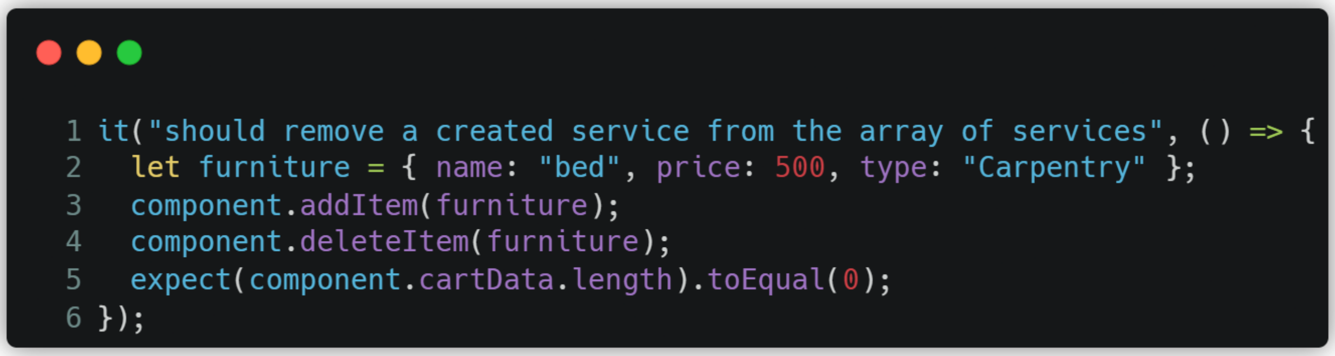
3. Unit Test: ‘should create’



```
it('should create', () => {
  expect(component).toBeTruthy();
});
```

This unit test case verifies whether the `**CartComponent**` is created successfully. The `**it**` function defines an individual test case with a description. In this case, it checks if the `**component**` object is truthy (i.e., not null or undefined), indicating that the component was successfully instantiated. The `**expect**` statement with `**toBeTruthy()**` matcher verifies this condition.

4. Integration Test: ‘should create a service in the cart array’



```
it("should remove a created service from the array of services", () => {
  let furniture = { name: "bed", price: 500, type: "Carpentry" };
  component.addItem(furniture);
  component.deleteItem(furniture);
  expect(component.cartData.length).toEqual(0);
});
```

This is an integration test case that checks the behavior of the `**deleteItem**` method in the `**CartComponent**`. It simulates adding a service (furniture object) to the cart by calling the `**addItem**` method and then removing it using the `**deleteItem**` method.

The `**expect**` statement with `**toEqual(0)**` matcher verifies that the `**cartData**` array in the component is empty after removing the service. If the service is successfully removed from the array, the expectation will pass.

Test Cases

Test Case ID: #TC001
Test Scenario: Client/Professional Login
Test Steps:

- The user navigates to the login page
- The user enters a registered email address as the username
- The user enters the registered password
- The user clicks 'Sign In'

Prerequisites:

- A registered account with a unique username and password.

Test Data: Legitimate username and password
Expected/Intended Results: Once the username and password are entered correctly, the user is successfully logged into their account.
Actual Results: As Expected
Test Status: Pass

Test Case ID: #TC002
Test Scenario: Client Creates Service Request
Test Steps:

- The client logs into their account
- The client navigates to the service request page
- The client fills in the necessary details for the service request
- The client submits the service request

Prerequisites:

- Client logged into their account

Test Data: Valid service request details
Expected/Intended Results: The service request is successfully created and submitted.
Actual Results: As Expected
Test Status: Pass

Test Case ID: #TC003
Test Scenario: Professional Accepts Service Request
Test Steps:

- The professional logs into their account
- The professional navigates to the service request section
- The professional views the available service requests
- The professional selects a service request to accept
- The professional clicks 'Accept'

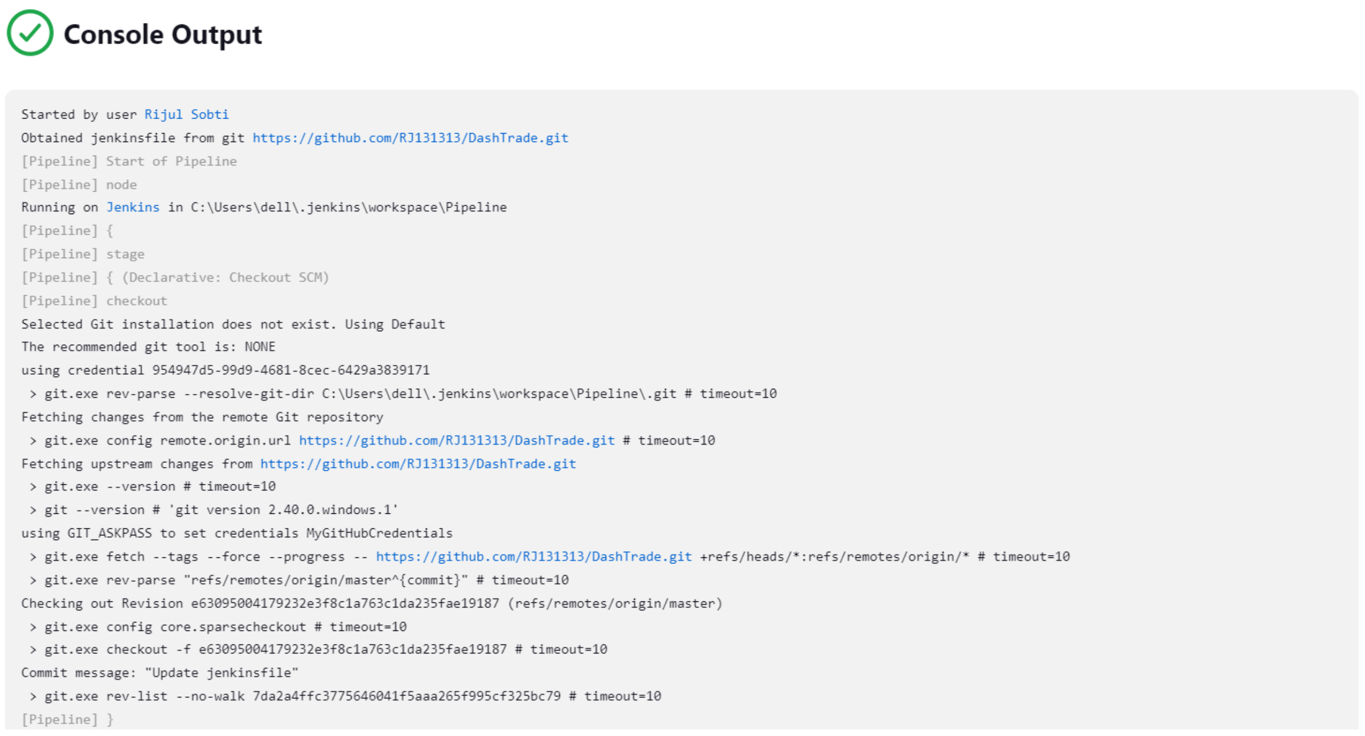
Prerequisites:

- Professional logged into their account
- Existing service requests

Test Data: Valid service request to accept
Expected/Intended Results: The selected service request is successfully accepted by the professional.
Actual Results: As Expected
Test Status: Pass

CI/CD

Jenkins is used alongside GitHub to enable continuous integration and deployment (CI/CD) for our project. It provides automated build and test capabilities, seamless integration with various development tools, scalability and distribution of workloads, and collaboration. By combining Jenkins with GitHub, we can automate essential tasks, improve software quality, streamline development and deployment processes, and foster efficient collaboration within our team.



Console Output

```
Started by user Rijul Sobti
Obtained jenkinsfile from git https://github.com/RJ131313/DashTrade.git
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in C:\Users\dell\.jenkins\workspace\Pipeline
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Declarative: Checkout SCM)
[Pipeline] checkout
Selected Git installation does not exist. Using Default
The recommended git tool is: NONE
using credential 954947d5-99d9-4681-8cec-6429a3839171
 > git.exe rev-parse --resolve-git-dir C:\Users\dell\.jenkins\workspace\Pipeline\.git # timeout=10
Fetching changes from the remote Git repository
 > git.exe config remote.origin.url https://github.com/RJ131313/DashTrade.git # timeout=10
Fetching upstream changes from https://github.com/RJ131313/DashTrade.git
 > git.exe --version # timeout=10
 > git --version # 'git version 2.40.0.windows.1'
using GIT_ASKPASS to set credentials MyGitHubCredentials
 > git.exe fetch --tags --force --progress -- https://github.com/RJ131313/DashTrade.git +refs/heads/*:refs/remotes/origin/* # timeout=10
 > git.exe rev-parse "refs/remotes/origin/master^{commit}" # timeout=10
Checking out Revision e63095004179232e3f8c1a763c1da235fae19187 (refs/remotes/origin/master)
 > git.exe config core.sparsecheckout # timeout=10
 > git.exe checkout -f e63095004179232e3f8c1a763c1da235fae19187 # timeout=10
Commit message: "Update jenkinsfile"
 > git.exe rev-list --no-walk 7da2a4ffc3775646041f5aaa265f995cf325bc79 # timeout=10
[Pipeline] }
```

The console output shows Jenkins linking with GitHub and establishing the pipeline.

Jenkins Pipeline Stages

Build stage:

The Build stage is responsible for building the project. It sets up the necessary environment and executes the build process. This stage includes tasks such as installing dependencies and compiling code. The script associated with this stage contains functions that perform the build-related tasks.

Test stage:

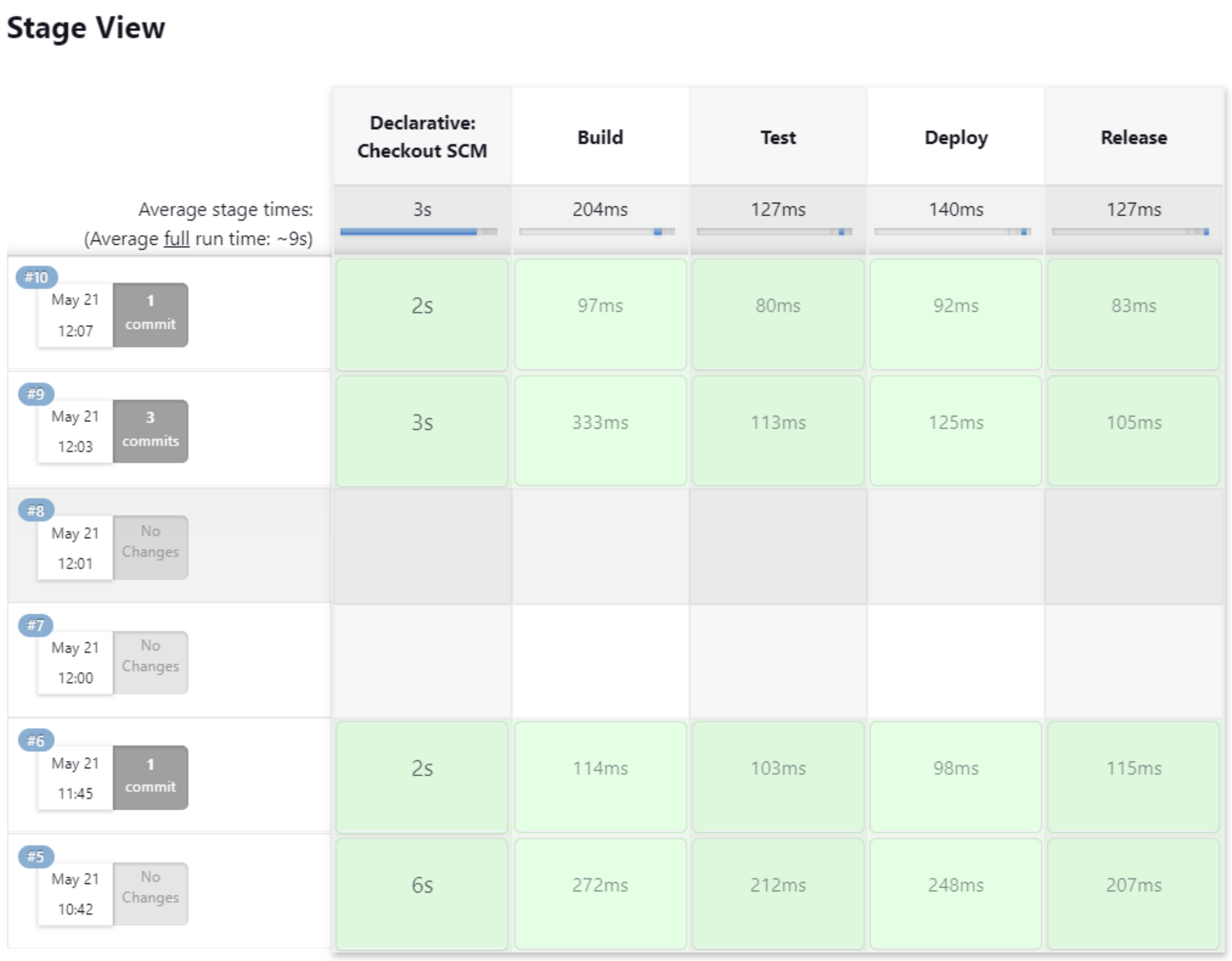
The Test stage focuses on testing the project. It runs various tests to ensure the quality and functionality of the codebase. This stage includes tasks such as running unit tests, integration tests, and other automated tests. The script associated with this stage contains functions that handle the test execution and generate test reports.

Deploy stage:

The Deploy stage is dedicated to deploying the project to the GitHub repository. It involves tasks such as transferring files and setting up the infrastructure required to make the software available. The script associated with this stage contains functions that facilitate the deployment process and handle any necessary configurations.

Release stage:

The Release stage is responsible for preparing and releasing the project. It performs tasks related to publishing the project to a package manager, generating release notes, and updating documentation. The script associated with this stage contains functions that handle the release-related tasks and ensure a smooth release process.



Stage View

	Declarative: Checkout SCM	Build	Test	Deploy	Release
Average stage times: (Average full run time: ~9s)	3s	204ms	127ms	140ms	127ms
#10 May 21 12:07 1 commit	2s	97ms	80ms	92ms	83ms
#9 May 21 12:03 3 commits	3s	333ms	113ms	125ms	105ms
#8 May 21 12:01 No Changes					
#7 May 21 12:00 No Changes					
#6 May 21 11:45 1 commit	2s	114ms	103ms	98ms	115ms
#5 May 21 10:42 No Changes	6s	272ms	212ms	248ms	207ms

This Stage View lets you visualize the progress of various stages of our pipeline in real-time.