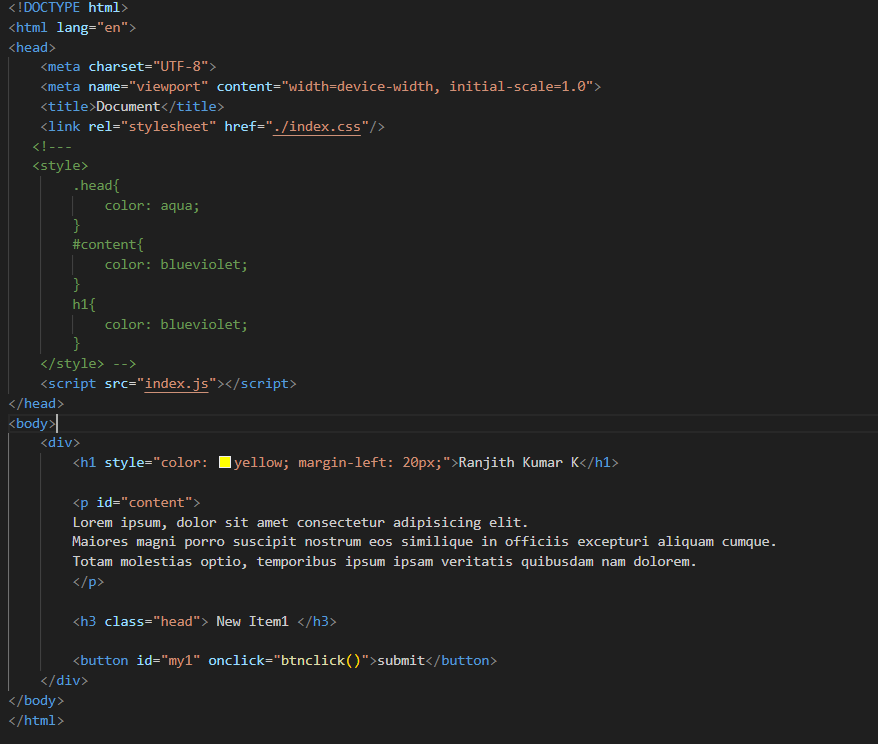
**Playwright with TypeScript**

**Day1**

HTML: **Hyper text markup language**

* HTML is the structure of the page
* HTML consists of series of elements.



This HTML document is a simple webpage with the following key components:

* Standard HTML5 structure with DOCTYPE declaration and language attribute
* Head section containing:
  + Metadata for character encoding and responsive viewport
  + Page title
  + External CSS link to "index.css"
  + Commented-out internal CSS styles that would have applied different colors
  + JavaScript link to "index.js"
* Body content including:
  + A main div container
  + H1 heading with inline styling (yellow color and left margin)
  + Paragraph with Lorem ipsum placeholder text and ID "content"
  + H3 heading with class "head"
  + Button with ID "my1" that calls a JavaScript function "btnclick()" when clicked

**Java Script Basics:**

**Function Definition and Event Handling**

javascript

function btnclick(){

alert('Hi team click here')

}

This defines a function named btnclick that displays an alert dialog with the message "Hi team click here" when called. This function would typically be connected to a button's click event in HTML.

**Variable Declaration**

javascript

let a=10;

let b=5;

const c =90;

* let is used to declare variables that can be reassigned later
* const is used to declare constants that cannot be reassigned
* Three variables are created: a (value 10), b (value 5), and c (constant value 90) javascript

**How to print the Values different formats**

*/\**

*console.log("sum of ",a+b);*

*console.log("Sum of "+a+" and "+b+" is "+(a+b));*

*console.log(`Sum of ${a} and ${b} is ${a+b}`);*

*\*/*

These commented lines demonstrate three ways :

* Using comma separation
* Using string concatenation with +
* Using template literals with backticks and ${}

**Boolean Variable**

let val = true;

Declares a boolean variable val with the value true.

**Array Operations**

let arr = ["hello",48,true,3.0]

Creates an array with mixed data types (string, number, boolean, float).

**Array manipulation methods:**

javascript

arr.push("jira"); *// Adds "jira" to the end*

arr.push(50); *// Adds 50 to the end*

arr.pop(); *// Removes the last element (50)*

**Array Methods with Arrow Functions**

arr.filter((e)=> Number.isInteger(e))

.map((e)=> e\*10)

.forEach((e) => console.log(e));

This chain of methods:

1. Filters the array to keep only integer elements
2. Multiplies each remaining element by 10
3. Prints each result to the console

**String Reversal Function**

javascript

function reverse(a) {

console.log(a.split('').reverse().join(''));

}

reverse("today"); *// Outputs: "yadot"*

**Recursive Function**

javascript

function rec(a){

if(a==10){

return;

}

console.log(a)

rec(a+1);

}

rec(1);

**Local Storage Form Handling Summary**

This code demonstrates a two-part process for handling form data using HTML, JavaScript, and localStorage:

**The store() Function**

This function handles form submission and data storage:

1. Gets a reference to a form with ID "registrationForm"
2. Adds a submit event listener to the form
3. Prevents the default form submission behavior
4. Creates an object containing form values (name, role, age)
5. Stores this data in the browser's localStorage after converting it to JSON
6. Redirects the user to "index1.html" after saving

**CODE:**

function store(){

const form = document.getElementById("registrationForm");

form.addEventListener("submit", function(event) {

event.preventDefault();

const formData = {

name: form.name.value,

role: form.role.value,

Age: form.age.value

};

localStorage.setItem("formData", JSON.stringify(formData));

window.location.href = "index1.html";

});

}

**The display() Function**

This function retrieves and displays previously stored form data:

1. Gets a reference to an element with ID "output"
2. Retrieves the stored form data from localStorage
3. If data exists:
   * Parses the JSON string back into an object
   * Displays the data in formatted HTML
4. If no data exists:
   * Shows a "No data found" message

function display(){

const output = document.getElementById("output");

const storedData = localStorage.getItem("formData");

if (storedData) {

const data = JSON.parse(storedData);

output.innerHTML = `

<p><strong>Name:</strong> ${data.name}</p>

<p><strong>Role:</strong> ${data.role}</p>

<p><strong>Age:</strong> ${data.Age}</p>

`;

} else {

output.textContent = "No data found.";

}

}

**Day2:**

**JS Events:**

In JavaScript, events are actions or occurrences that happen in the browser that the browser or the user can respond to, Events are essential for making web pages interactive like when a user clicks a button.

**Button Click Event:**

This provide if you click event display the current data and time.

<!---Button Clicked event-->

   <button

   onclick="this.innerHTML = Date()">

   The time is :</button>

**Mouse entered and leave event:**

<!--- Mouse enter / mouse leave event-->

   <button

   onmouseenter="this.innerHTML='Mouse Entered'"

   onmouseleave="this.innerHTML='Mouse Leave'">Mouse</button>

This event provide hover the mouse on the button is display the text.

**Key Pressing:**

This Event working based on enter the text into inside the text box it will show the alert.

<!--- Key pressing Event-->

   <input type="text" onkeydown="pressing()"/>

JS 🡪

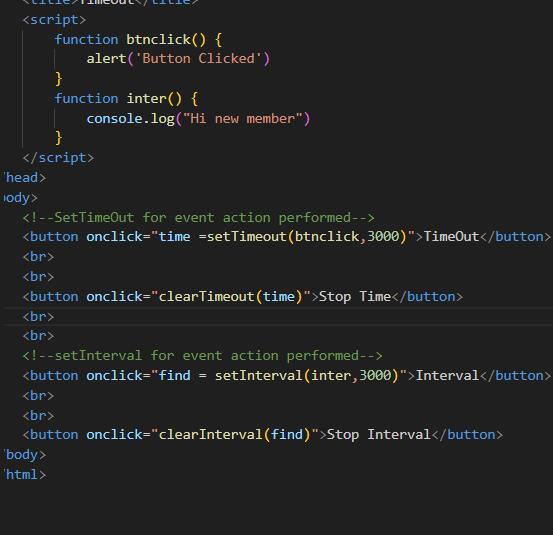
function pressing(){

   alert("clicked");

}

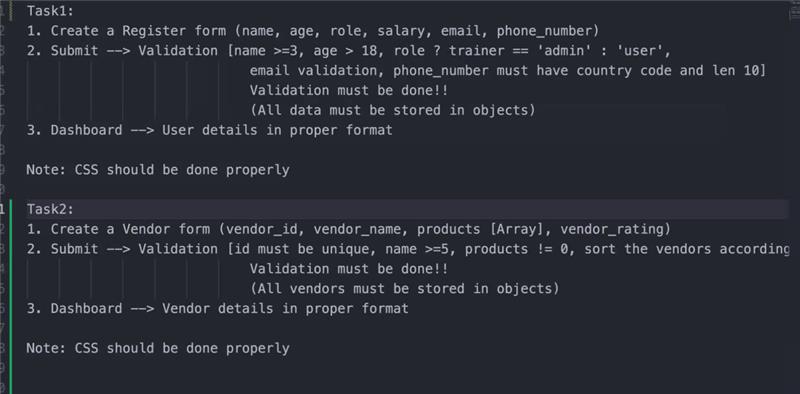
**Set Timeout & Set Interval Event:**

* Timeout used for performing some action specific interval of time and you want to stop the time out action is possible.
* Interval Specific interval you can perform some action in the console or alert.



**Create the Form Add some Functionality:**

**Task 1 and Task 2:**



**Promises in JS:**

JavaScript is single-threaded and often needs to do async operations like:

* Fetching data from a server
* Reading a file
* Waiting for a timer

**Asynchronous Function:**

async function catdis(params) {

   const url = "[https://cataas.com/cat?width=200;height=200;json=true";](https://cataas.com/cat?width=200;height=200;json=true%22;)

   const cat = await fetch(url).then(res => res.json()).then(data => data.url);

   document.getElementById("catid").src =cat

}

**Day3:**

**TypeScript:**

Install typescript and run commands.

* npm -v
* npm init -y
* npm install typescript –save-dev
* npm tsc init
* npx tsc index.ts
* node index.js

**Define:**

TypeScript is the superset of java script and typescript add the syntax on top of JS.

**Basic Function Creation:**

function dis1() {

    let name: string = "Ranjith";

    console.log(name);

}

function dis2(){

    const age:number = 10000;

    console.log(age)

}

dis1()

dis2()

**Create the Interface and class store the data and displays:**

// Interface

interface emp{

    user\_id: number;

    name: string;

    role? : any;

}

const user1:emp={

    user\_id:2,

    name:"rash",

    role:"EC"

}

const user2:emp={

    user\_id:2,

    name:"rash"

}

console.log(user1);

console.log(user2);

This code refers the add the Interface and add the data into the class based on the type.

**Task:**  
you need to create a vendor object with types and then display all the values by sorting on the basis of their id

**Sort the array and Optional Parameters:**

* In typescript how the object and type value are created and handle the functions using foreach loop, mention optional param, give the type (any).

interface emp{

    user\_id: number;

    name: string;

    role? : any;

}

class compnay1 implements emp{

user\_id: number;

name: string;

role?: any;

constructor(user\_id:number, name:string, role:any){

this.user\_id=user\_id;

this.name=name;

this.role=this.role;

}

}

function showcompany1(){

const emp1:emp = new compnay1(1,"dhurv","trainer");

console.log(emp1);

}

function showcompany2(){

const emp2:emp = new compnay1(10,"karthik","employee");

console.log(emp2);

}

showcompany1()

showcompany2()

const database1:compnay1[] = [

new compnay1(2,"user1","dep1"),

new compnay1(2,"user1","dep1"),

new compnay1(2,"user1","dep1")

]

const sort1 = database1.sort((e1,e2)=>e1.user\_id>e2.user\_id ? -1: 0)

for (const vendor of sort1) {

console.log(`ID: ${vendor.user\_id}, Name: ${vendor.name}, Role: ${vendor.role != undefined ? vendor.name + " " : vendor.name}`);

}

**Promise to Get the response from API:**

* Get the data from url in json format data and display the json data using Promise<Response>.

function getFacts(url: string): Promise<Response> {

    return fetch(url).then(res => res.json());

getFacts("https://cataas.com/cat?width=200;height=200;json=true").then(data => console.log(data));

**Day 4:**

1. B) Microsoft

The typescript developed by Microsoft, It's open source high level programming language and TypeScript used to develop JavaScript applications for both client-side and server-side.

2. C) Node

TypeScript can technically be used with other backend technologies that support JavaScript, TS is subset of Java script library.

3. C) Java Script

The primary purpose of the TypeScript compiler tsc is to transpile TypeScript code into plain JavaScript code that can be understood and executed by web browsers and Node.js.

4. C) extends

The keyword to inherit the class using extends keyword.

5. B) var x = "string";

This one is incorrect because the syntax is JS.

6. B) var x:number= 999;

This one is refer the type of x is number store the value 999 this syntax for TypeScript.

7. B) .ts

TypeScript source code files typically use the .ts file extension.

8. tsc filename.ts

This command used for run the typescript file using Typescript Compiler.

9. B) tsc filename -w

-w flag tells the TypeScript compiler to watch the specified file or the entire project if a tsconfig.json is present for changes and automatically recompile when a change is detected.

10. super()

The super() keyword is used within the constructor of the child class to invoke the constructor of the immediate parent class.

**Day 5:**

**Key of:**

Keyof is used to store the different type of value in same variable and return the values, get the data from type Point using generic function.

type Point = { name: string; age: number };

type a = keyof Point;

let x:a;

x = "name";

x = "age";

interface Person{

    name: string;

    age: number;

    address: string;

    phone: string;

}

const person: Person = {

    name: "John Doe",

    age: 30,

    address: "123 Main St",

    phone: "123-456-7890"

};

interface Employee{

    name: string;

    department: string;

    salary: number;

}

const newemp: Employee = {

    name: "Jane Smith",

    department: "Engineering",

    salary: 80000

}

//generic function to get data from object

function getData<T,K extends keyof T>(obj : T, key : K){

    return obj[key];

}

console.log(getData(person, "name"));

console.log(getData(person, "age"));

console.log(getData(person, "address"));

console.log(getData(newemp, "name"));

console.log(getData(newemp, "department"));

console.log(getData(newemp, "salary"));

**Reduce Function:**

This one is used for simplifying the code and easy to understand and reduce the code size.

//Reduce Function

function sum1(...numbers: number[]):number{

    return numbers.reduce((t, num)=>t+num,0);

}

function addstring(...names: string[]):string{

    return names.reduce((t, name)=>t+name, "");

}

const result = sum1(1,2,3,4,5);

console.log(result);

const result1 = addstring("John", "Doe", "Smith");

console.log(result1);

**Method overloading:**

In typescript overloaded method same as java and pass the different type of values into the same method, I want pass number or string same method can be used return the values.

interface employee{

    details:{

        (name : string):string;

        (age : number) :string;

    }

}

const emp: employee={

    details: (name: string | number): string => {

        if (typeof name === "string") {

            return `Name is ${name}`;

        } else {

            return `Age is ${name}`;

        }

    }

}

console.log(emp.details("John Doe"));

console.log(emp.details(30));

interface EcomApp {

    products: {

        (id: number): string;

        (name: string): string;

    };

}

const product1: EcomApp = {

    products: (idOrName: number | string): string => {

        if (typeof idOrName === "number") {

            return `ID of product is ${idOrName}`;

        } else {

            return `Product name is ${idOrName}`;

        }

    }

};

console.log(product1.products(101));

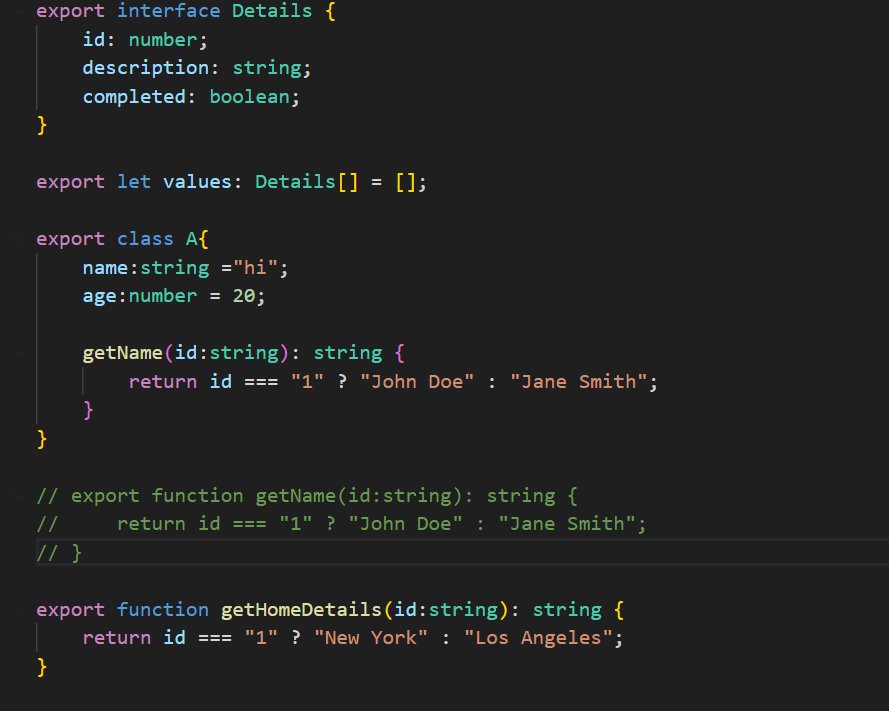
console.log(product1.products("Laptop"));

**Modules import & export Classes, Functions:**

Access the method and class level variables one class to another class using import/export keyword.

**Class1 File:**

In this Class1 getname() and getHomeDetails () two functions used in another class Class2 using import export keyword to access the method from another class this class have interface and Details list can add product.



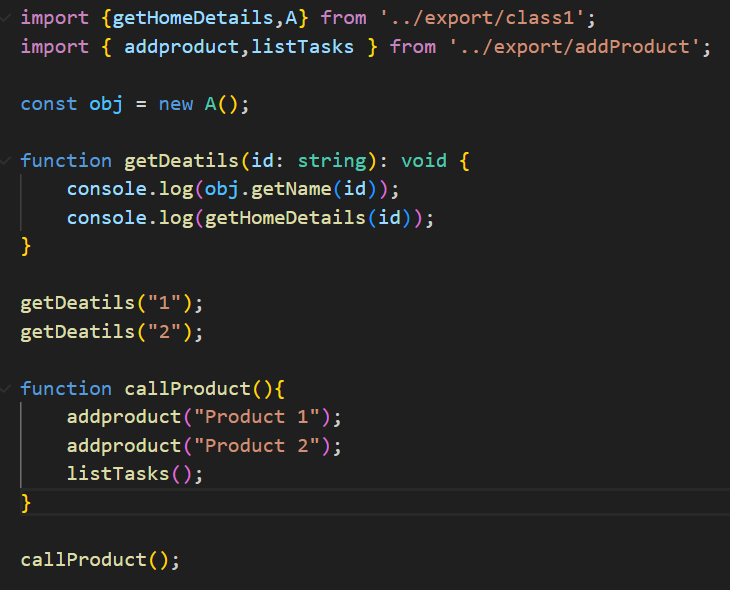
**Addproduct.ts File:**

This class have two function addproduct() and listTasks() two functions to access inside the class2 call this functions.

****

**Class 2 File:**

The Class2 main class to call all functionalities to import statement access the method from Class1 and use inside the Class2 we can create the obj for that class and to access the getName() function .



**Day 6:**

**Playwright with TypeScript:**

**Features:**

* Auto wait
* Cross browser Compatibility – Chrome, Firefox, Edge, Webkit.
* Multi-platform – Mac OS, Windows, Linux
* Multilingual Flexibility – Supports Java, Python, Java Script, C#, .NET

**Advance Feature:**

* Tracing and Debugging – take screenshot, video recording
* Network Interception
* Browser Context Management
* Codegen Tool
* Java Script – Asynchronous this one does not execute the order wise.
* **await – used to wait each action**

**Installation of Playwright:**

* npm install playwright@latest

**Run the test:**

* npx playwright test

**Run the test headed mode:**

* npx playwright test –headed

**Run the test UI mode:**

* npx playwright test –ui

**Get the report in html page:**

* npx playwright show-report
* **Test report generate in 9323 port**

**Import the playwright and expect in .ts file**

import {test, expect} from "@playwright/test";

* **Navigate to the website**

****

* **Assertion in Playwright:**



* **Different type of assertion:**

await expect(locator).toBeChecked();

await expect(locator).not.toBeChecked();

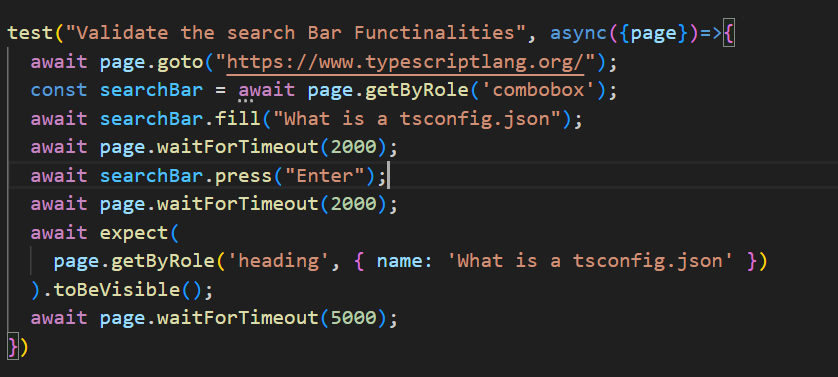
await expect(locator).toBeDisabled();

await expect(locator).toBeEnabled();

await expect(locator).toBeEditable();

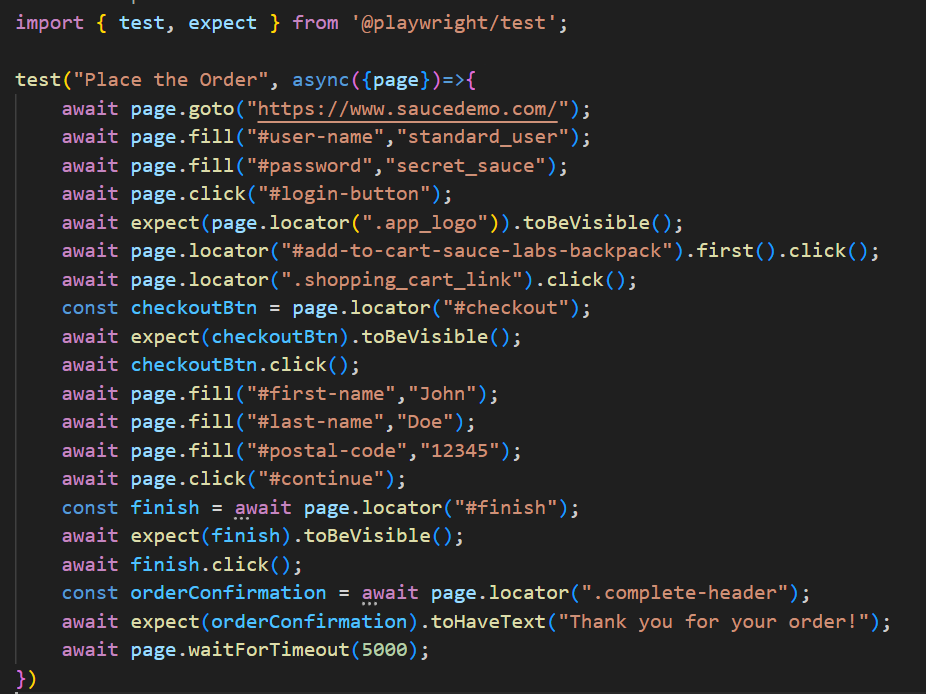
await expect(locator).toBeEmpty();

**Example TEST 1:**

****

This test checks if the search bar on the TypeScript website works correctly. It opens the website and finds the search bar. Then it types "What is a tsconfig.json" into the search bar. After waiting a bit, it presses Enter to search. It then checks if a heading with that text appears on the page. This means the search worked as expected.

**Example TEST 2:**

****

This test script automates the process of placing an order on the SauceDemo website. It starts by opening the website and logging in with a standard user account. Once logged in, the presence of the site logo confirms a successful login. The script then adds a backpack item to the cart and navigates to the shopping cart page. After clicking the checkout button, it fills in the required user information including first name, last name, and postal code. It proceeds by clicking the continue button and then the finish button to complete the order. Finally, it verifies the success of the order by checking for the confirmation message Thank you for your order.

**Day 7:**

**Run the test with title:**

* npm playwright test -g “test title”

**Run the Failed testcase:**

This only run the last failed testcase.

* npx playwright test --last-failed

**Trace mode:**

Trace used for creating the trace view on report we can see what the report and step by step visual report.

* npx playwright test --project chromium --trace on

**Test Ignore:** playwright.config.ts

Ignore the specific test folder or testcase.

testIgnore: '\*tests/\*github.spec.ts',

**Test Match:**

This is used run the specific file test cases match the file.

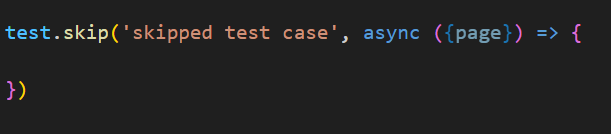


**Run the test specific browser:**

* npx playwright test --project chromium

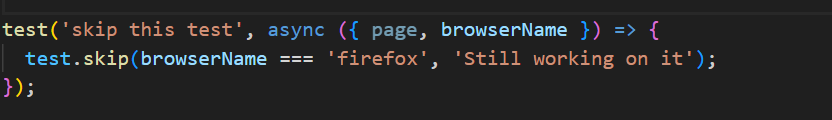
**Test skip:**

This used for skip the specific test case.



**Conditionally skip a test**[**​**](https://playwright.dev/docs/test-annotations#conditionally-skip-a-test):

You can skip certain test based on the condition.



**Take screenshot:**

// used for taking screenshot on fail testcases

    screenshot: 'only-on-failure',

// used for offline test some condition

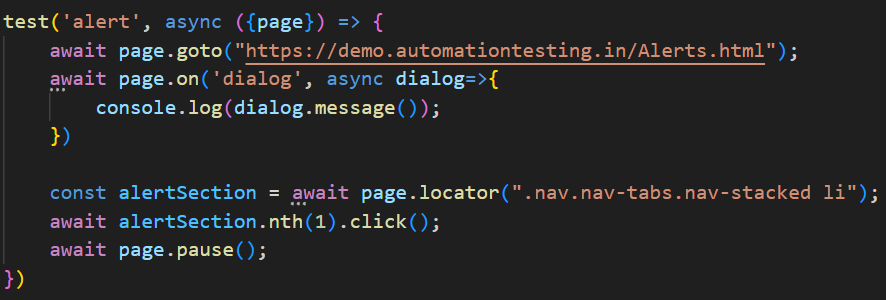
    offline: false,

**Set Base URL:**

Set the base URL each test.

* baseURL: 'https://playwright.dev/',

**Handle the Alerts:**

****

**Set the API credential:**

httpCredentials: {

username: 'user',

password: 'pass'

},

Ignore the httperrors on testing the testcase.

* ignoreHTTPSErrors: true,

**Emulation options:**

colorScheme: 'dark',

geolocation: {longitude: 12.343535, latitude: 45.56575},

locale: 'en-GB',

permissions: ['geolocation'],

timezoneId: '',

viewport: {width: 1280, height: 720}\*/

**Day 8:**

**Fixtures:**

**Two types:**

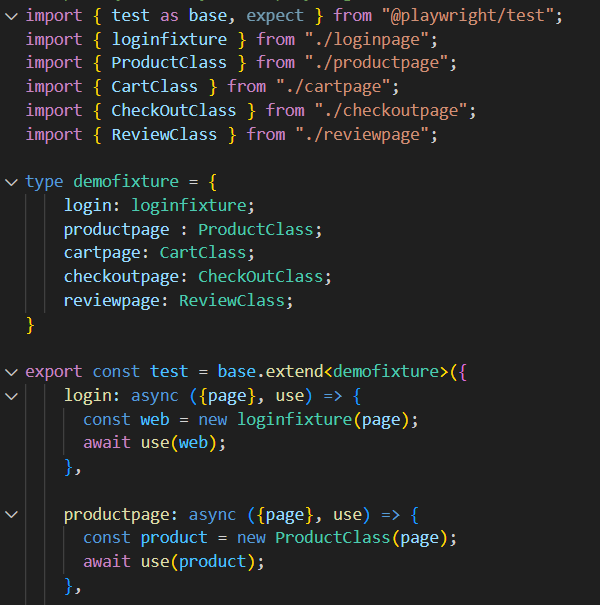
* Custom
* Bulit In – page, context, browser

**Custom Fixture:**

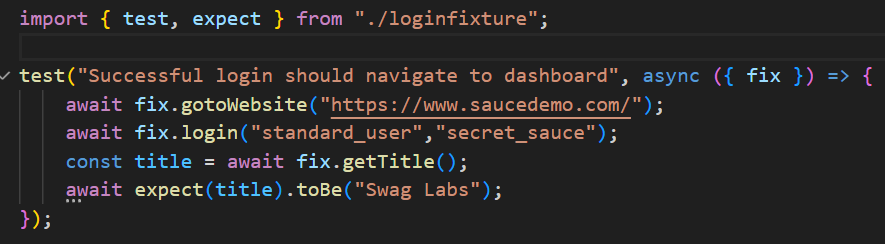
Create the class for the login Functionality



**Fixture:**

****

**Test file example.spec.ts :**



* A fixture is a way to define reusable test setup.
* In Playwright and similar frameworks, it simplifies initializing and sharing test state.
* Custom fixtures = powerful, DRY, and clean tests.

**Run the Test in parallel:**

In playwright.config.ts file set the parallelism, it will test the testcase fully parallel mode.

* fullyParallel: true,

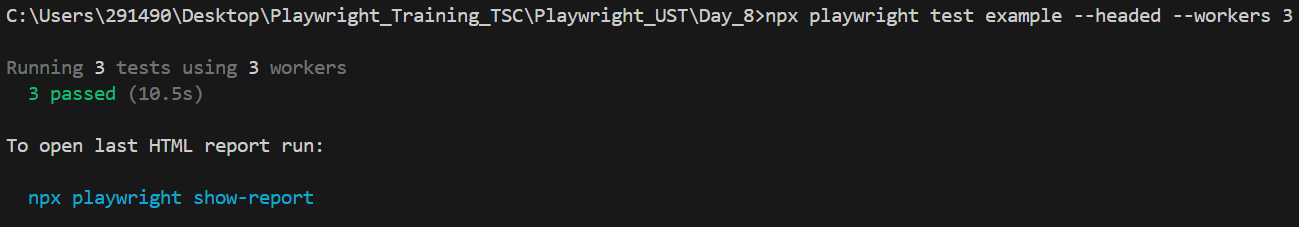
**Assign the Workers for Test Parallel:**

Run the test parallel mode in typescript file

* test.describe.configure({ mode: 'parallel' }) // serial

**Assign the Workers and run the test in CMD:**

* npx playwright test example --headed --workers 3



**Day 9:**

**Write the 10 Test Cases**

1. Verify User on Seller Page
   * Navigate to the Amazon Best Sellers homepage.
   * Validate that the user successfully lands on the Seller Page.
2. Verify User on Product Page
   * Click on the Beauty product ranked first.
   * Confirm the user is correctly redirected to the Product Page.
3. Verify Navigation Across Departments
   * Check that the user can navigate through each department from the sidebar and land on the correct department page.
4. Verify 'All' Button Navigation to Sign-In Page
   * Click the 'All' button.
   * Validate that the user is redirected to the Sign-In page.
5. Verify Pincode Entry Functionality
   * Click on the location bar.
   * Enter a valid pincode (641402).
   * Validate that the pincode is accepted correctly.
6. Verify Navigation Through Slide Bar
   * Interact with and navigate using the slide bar.
   * Validate successful slide interactions.
7. Validate Dropdown Selection
   * Select the "Beauty" option from a dropdown menu.
   * Ensure the correct selection is made.
8. Verify Search Functionality
   * Click on the search bar.
   * Input the search term "iphone".
   * Click the search icon.
   * Confirm the search is performed successfully.



**Day 10: Test Cases**

**Test: has title**

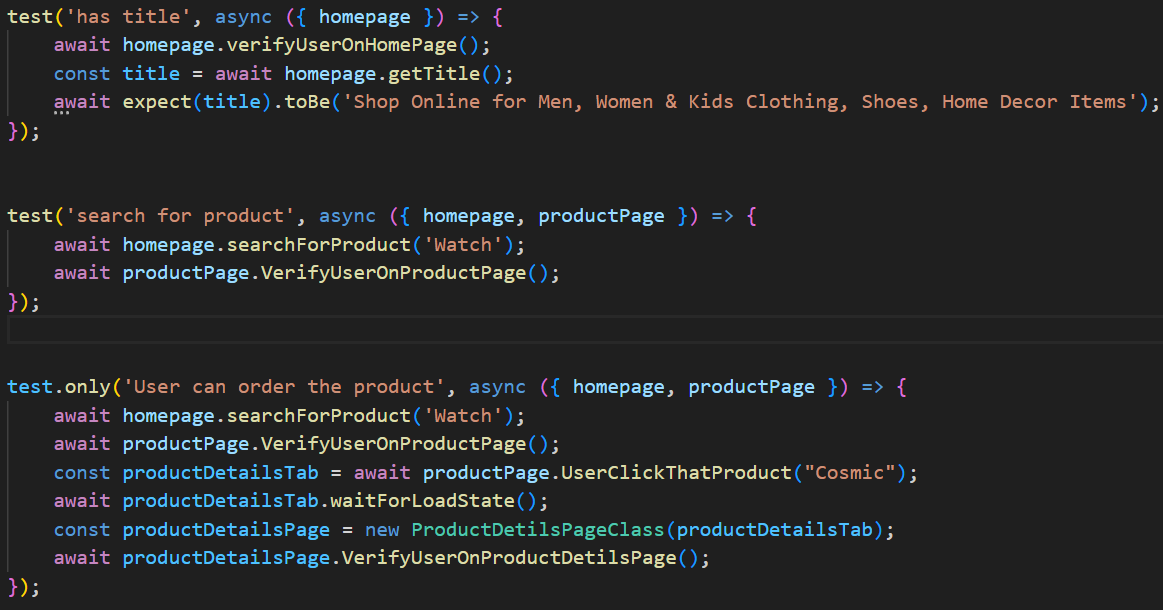
This test verifies that the user is correctly navigated to the homepage.  
It calls a method to ensure the homepage is loaded.  
Then it retrieves the page title using a helper method.  
An assertion checks the page title matches the expected store name.  
This ensures the homepage UI loaded with correct brand

**Test: search for product**

This test checks the product search functionality.  
It searches for "Watch" using the homepage's search bar.  
After submitting the query, it waits for navigation to the product page.  
It verifies the user is correctly taken to the product listing page.  
This confirms the search feature is working as expected.

**Test: User can order the product**

This test validates the full user journey from search to product detail.  
It searches for a "Watch" and ensures the product page is visible.  
Then it clicks on a specific product ("Cosmic"), which opens in a new tab.  
The new tab is passed to the product details page class for verification.  
It confirms the user has landed on the correct product detail page.

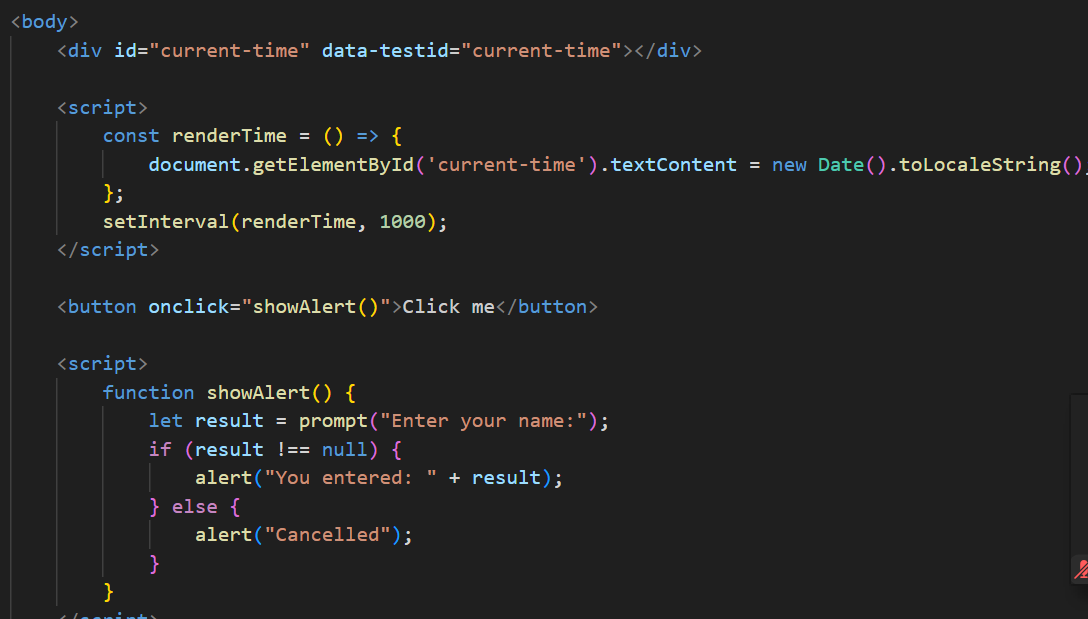
****

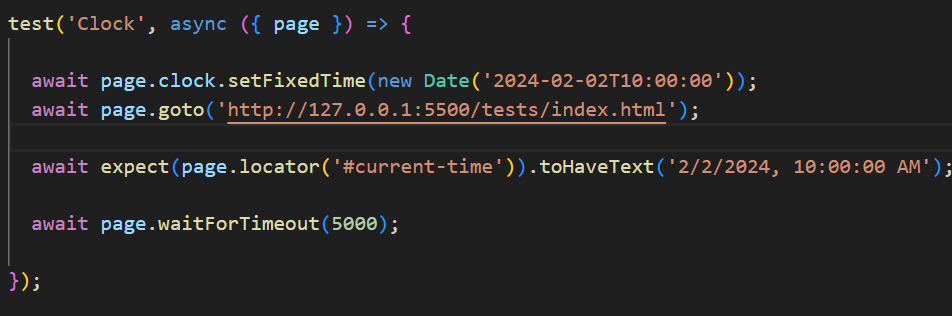
**Day11:**

**Clock:**

Get the local time and date validate using local host server.

**HTML PAGE:**





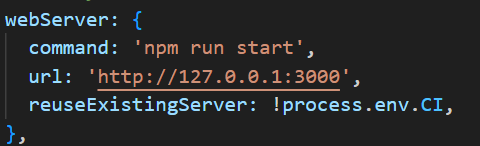
**HANDLE ALERT:**

* Alert will be handle by using dialog in playwright and accept, cancel, fill the Input on that dialog.

****

**Host local System projects and test:**

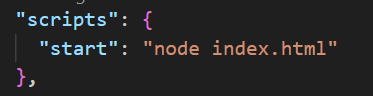
* playwright.config.ts



* Set the base URL also localhost <https://127.0.0.1.3000>



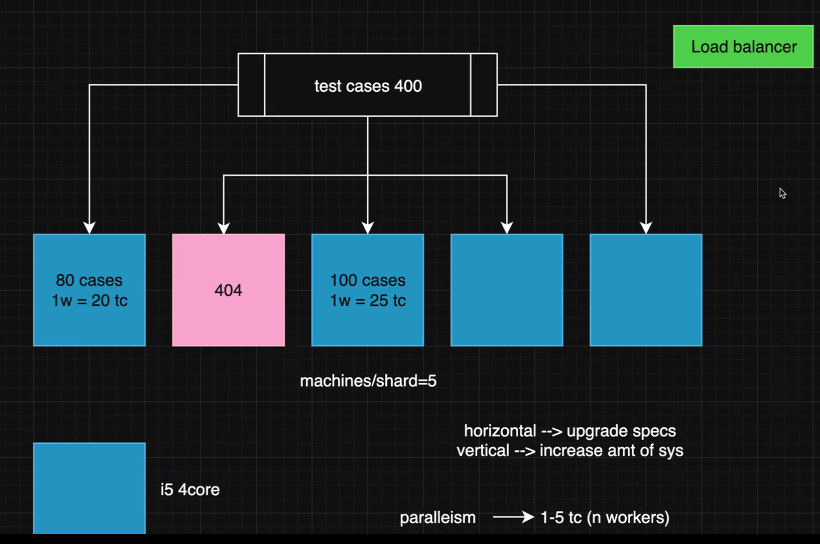
* package.json specify the script to run



**Shading:**

The shading is increasing the system efficiency divide the testcases based on the shading it’s a vertical scaling each shading have memory and core processors.

Playwright executes test files in parallel to make the most efficient use of your machine's CPU cores. To further boost parallelism and speed up test execution, you can distribute tests across multiple machines. This approach is known as sharding in Playwright. Sharding allows large test suites to be split and run concurrently on separate environments, significantly reducing total execution time.



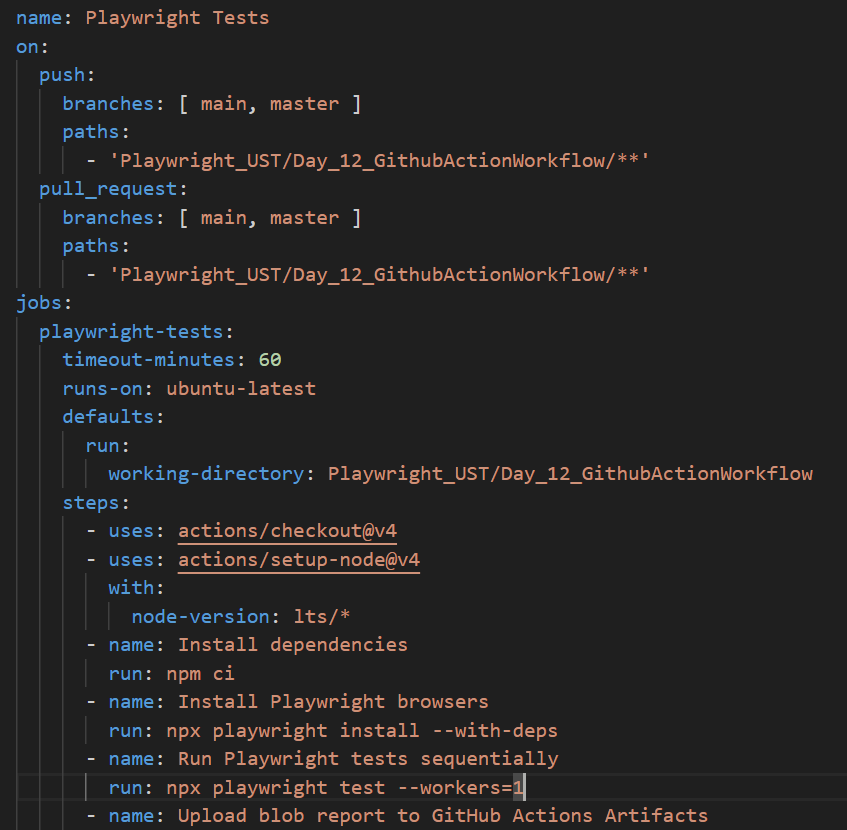
**Day 12:**

**Git Actions & Workflow**

GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline. You can create workflows that build and test every pull request to your repository or deploy merged pull requests to production.

The github workflow is provide the CI for testing once you push the code into the git that will automatically run the workflow and show the pass or fail results.

* Create the github action file .yml inside the project file



Run the GitHub Actions:

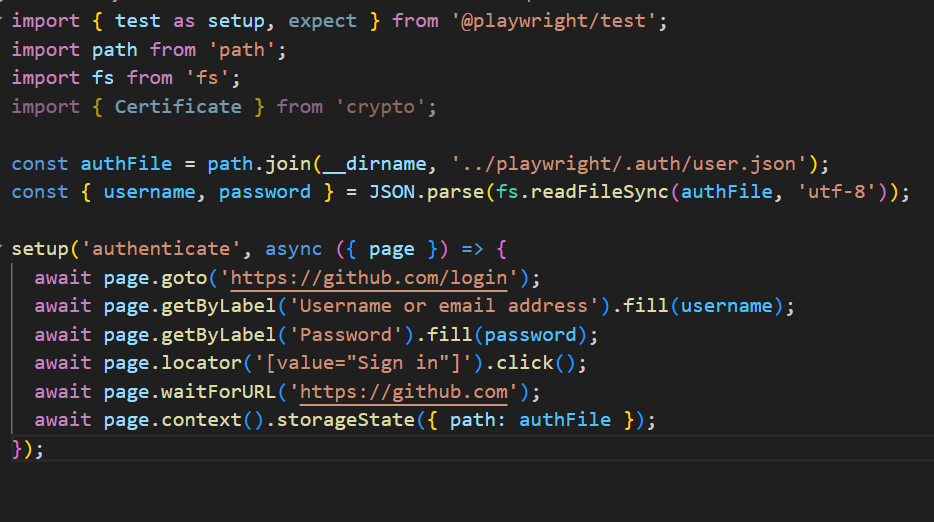
* Git add .
* Git commit -m “message”
* Git push origin master

Once push the code into Github that will automatically trigger the workflow. Inside the jobs specify the version of OS and running directory mention what are the dependencies to install versions of the Node and test running command generate the report.

**Playwright Authentication:**

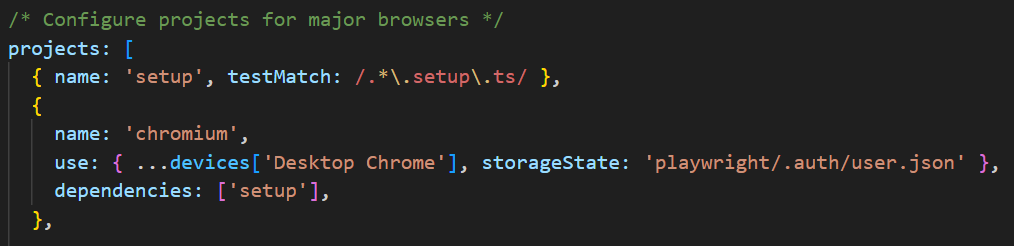
The set the authentication for your website and default run the authentication before run you test that will provide the security of your data and set before login to that website with valid credential.

**Auth.spec.ts**



The data username and password will be store in the format of json file into user.json.

**Set the playwright.config.json file as**



After the authentication successful the Github will login cookies are saved next time going to the github.com the user already login, So easy to test the website.

****

**Day 14 & 15:**

**Playwright with API Testing:**

1. GET /objects Test

This test sends a GET request to retrieve all available objects from the public API.  
It checks whether the response is successful using expect(res.ok()).toBeTruthy().  
Then it parses the response body as JSON using res.json().  
Although it doesn’t log or assert the contents, it ensures the API endpoint is reachable and responding.  
It acts as a health check or baseline verification for the objects collection endpoint.

2. POST /objects Test – Create an Object

This test creates a new object by sending a POST request with a JSON payload.  
The payload describes an Apple MacBook Pro with properties like year, price, CPU model, and disk size.  
It confirms that the request was successful and extracts the returned object’s id.  
This id is stored in a global variable for use in later tests.  
It validates the API’s object creation functionality and sets up state for further CRUD operations.

3. GET /objects/:id Test – Retrieve Created Object

Using the id from the previous test, this test retrieves that specific object.  
It sends a GET request to /objects/:id and verifies the response is successful.  
Then it parses and logs the object’s data to the console for visibility.  
This step ensures the object was properly saved and can be accessed individually.  
It serves as a verification of the data integrity after object creation.

4. PUT /objects/:id Test – Update Object

This test updates the object’s data using a PUT request to /objects/:id.  
The updated payload renames the object to "Dell Laptop" and tweaks CPU model to "Intel Core i11".  
After a successful update, it checks that the returned name is correctly updated.  
It also logs the full response for confirmation of the new object state.  
This confirms that object updates are correctly applied and reflected by the API.

5. DELETE /objects/:id Test – Delete Object

The final test deletes the previously created object using its id.  
After a successful deletion, it immediately attempts to retrieve the same object.  
The test expects the retrieval attempt to fail (expect(resget.ok()).toBeFalsy()), confirming deletion.  
This ensures the API is correctly removing resources and not leaving stale data.  
It completes the full lifecycle test of the object: create, read, update, and delete (CRUD).



****

**Email Verification API :**

Test 1: Email ID Verification  
This test sends a GET request to the email verification API using the provided email address as a path parameter. It includes the required API key in the request headers to authenticate the request. Upon receiving the response, the test checks for a successful status code (200) and confirms that the request was processed correctly. It then verifies that the email returned in the response body matches the email that was submitted. This test ensures that the basic endpoint for direct email validation is functional and returning accurate data.

Test 2: Check Email ID  
In this test, the email verification API is accessed via the /check endpoint, passing the email address as a query parameter. Like the first test, it includes the API key in the headers to authorize the request. After sending the request, the test validates that the server responds with a status code of 200, indicating success. It then logs the JSON response for manual inspection, ensuring that the email is processed correctly by this alternate endpoint. This test confirms that the query-parameter-based verification route works as expected and returns useful diagnostic information about the email address.



**Sentimental Analysis API**

Test 1: Positive Sentiment Analysis  
This test verifies that the sentiment analysis API correctly identifies a positive statement. It sends a POST request with the plain text "I love playwright" to the sentiment analysis endpoint, including the required apikey and content type in the headers. The response is expected to have a successful status code (200) and a body indicating a sentiment classification. After parsing the JSON response, the test checks that the detected sentiment is "positive". This ensures that the API is capable of recognizing favorable language in input text.

Test 2: Negative Sentiment Analysis  
This test case ensures that the sentiment analysis API can accurately detect negative expressions. It sends a POST request with the text "I hate you", along with necessary headers for API key and content type. Once the response is received, the test verifies that it is successful and that the returned sentiment is correctly marked as "negative". The test helps confirm that the API is effective in distinguishing unfavorable or hostile language. It also reinforces confidence in the accuracy of the sentiment classification model.

