Lab: Automated Testing

Lab for the "Software Engineering and DevOps" course @ SoftUni.

We are given a simple JS app that lets the user add tasks, mark them as completed, delete them and filter them according to their status.

We should use Playwright and write some code to test the UI of this app.

1. Install Playwright

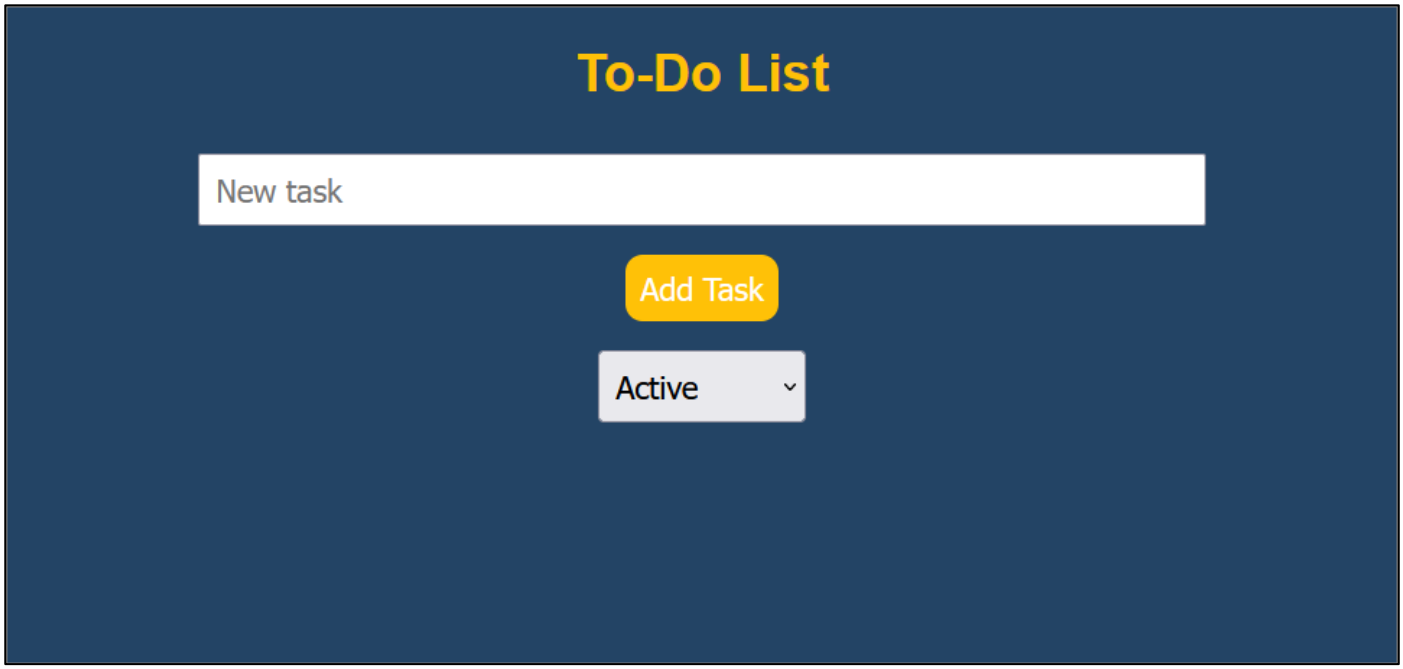
Open the To-Do app directory in Visual Studio Code and open a terminal. Install **Playwright**, using this command:

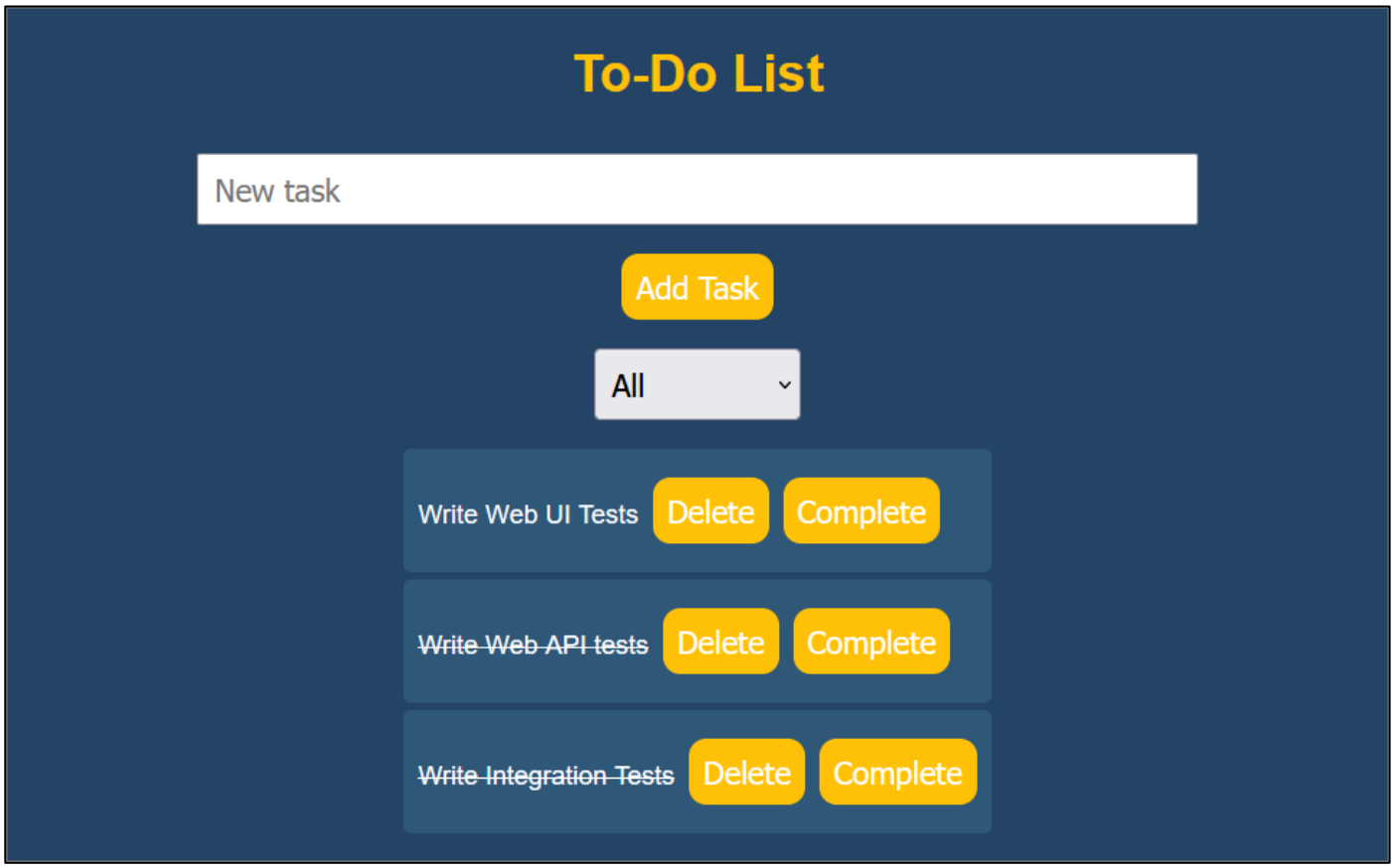
```
npm install -D @playwright/test
```

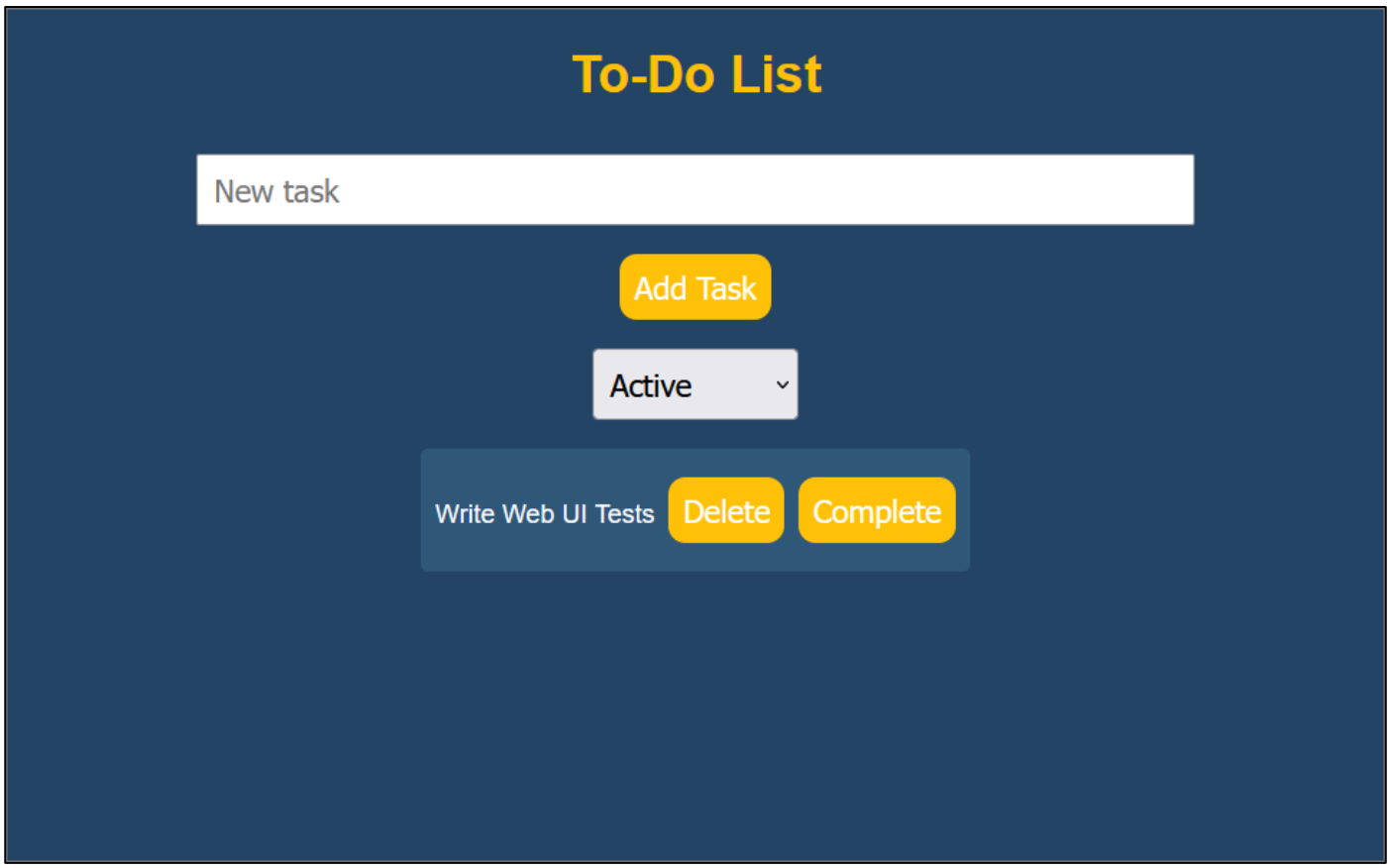
2. Write tests

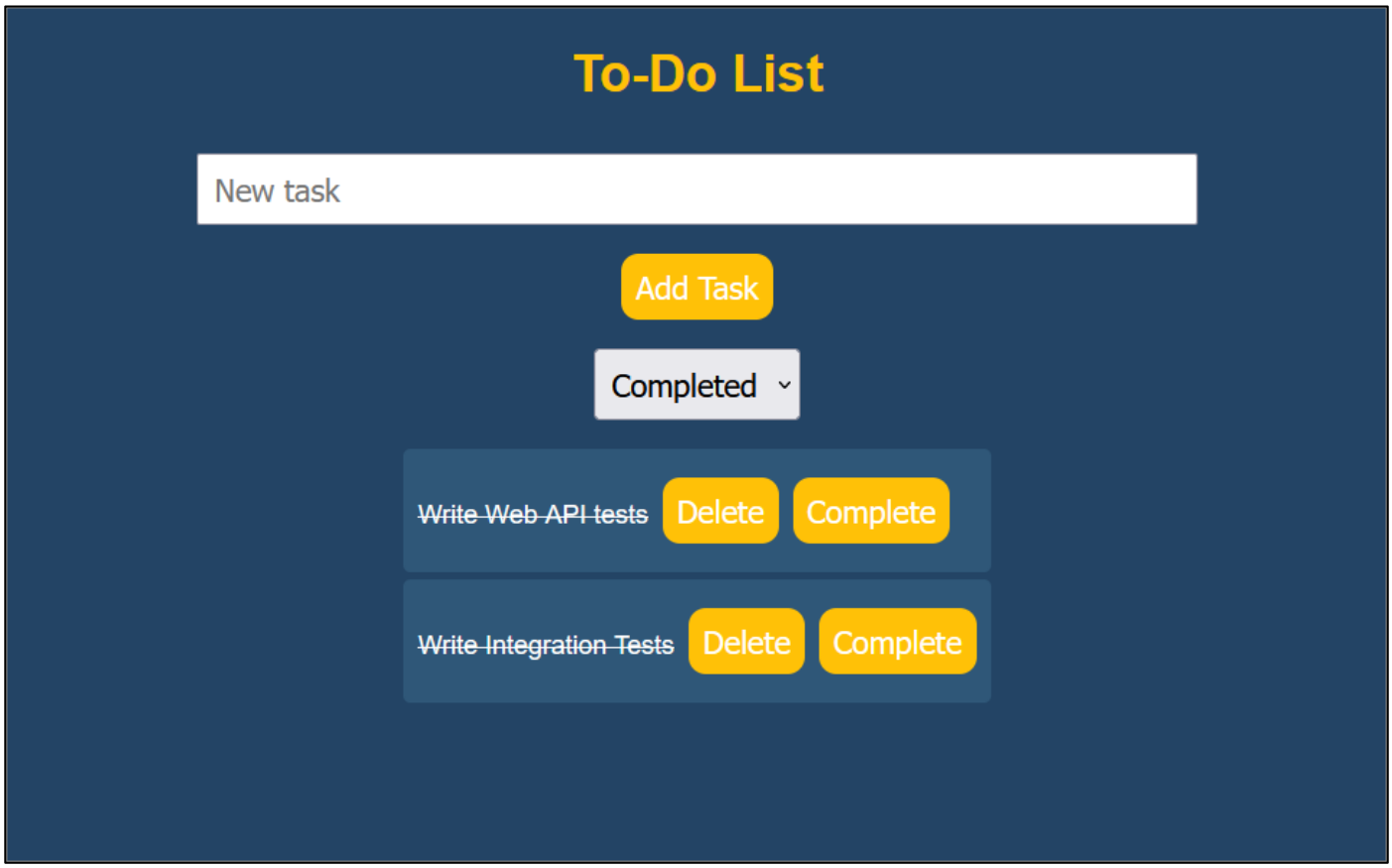
We are given a simple JS app that lets the user add tasks, mark them as completed, delete them and filter them according to their status.

We should use Playwright and write some code to test the UI of this app.









Before we start, we should create a new folder in the project directory and name it "**`tests`**" and create a file **`todo.tests.js`**. This file will hold the test code.

Now let's write the tests.

Test 1: Test If a User Can Add a Task

The purpose of this test is to ensure that the application **correctly adds a new task** when a **user types into the task input** and **clicks** the **`[Add Task]`** button.

To **import the test and expect functions** from the **Playwright Test library**, we use the following code:

```
const { test, expect } = require('@playwright/test');
```

After that, we **define** the **test**. It has a **description `('user can add a task')`** and an **asynchronous function** that **runs** the test. The **page object** is **automatically** injected by **Playwright Test**, and it represents a **new browser page**.

```
test('user can add a task', async ({ page }) => {
```

Then we navigate to the **To-Do** app. Note that the **port** may be **different** for you as it depends on the way the app has been started:

```
  await page.goto('http://localhost:5500/');
```

Then we fill the task input with the text **`'Test Task'`**:

```
  await page.fill('#task-input', 'Test Task');
```

And we click the **`[Add Task]`** button, which should add the new task:

```
  await page.click('#add-task');
```

Now we have to get the text content of the first element with the class **`.task`**. After the **`[Add Task]`** button is clicked, this should be the task that was just added:

```
  const taskText = await page.textContent('.task');
```

Finally comes the assertion. It checks that the text of the added task includes **`'Test Task'`**. If it does, the test passes. If it doesn't, the test fails:

```
  expect(taskText).toContain('Test Task');
});
```

The code should look like this:

```
// Verify if a user can add a task
const { test, expect } = require('@playwright/test');

test('user can add a task', async ({ page }) => {
  await page.goto('http://localhost:5500/');
  await page.fill('#task-input', 'Test Task');
  await page.click('#add-task');
  const taskText = await page.textContent('.task');
  expect(taskText).toContain('Test Task');
});
```

Test 2: Test If a User Can Delete a Task

This test checks whether the application correctly **deletes a task** when the **user clicks the `[Delete]`** button of a task.

The **first** part of this test is the same as the first test – it **navigates to the app** and **adds a new task**:

```
test('user can delete a task', async ({ page }) => {
  // Add a task
  await page.goto('http://localhost:5500/');
  await page.fill('#task-input', 'Test Task');
  await page.click('#add-task');
```

Then we add a **line that clicks** the **`[Delete]`** button of the task. The **task** should be **removed** from the list:

```
  // Delete the task
  await page.click('.task .delete-task');
```

Finally, we check that the text of the first task does not include **`'Test Task'`**. If the task was correctly deleted, its text should no longer be present in the list, and the test should pass:

```
  const tasks = await page.$$eval('.task',
    tasks => tasks.map(task => task.textContent));
  expect(tasks).not.toContain('Test Task');
});
```

The code for this test should look like this:

```
// Verify if a user can delete a task
test('user can delete a task', async ({ page }) => {
  // Add a task
  await page.goto('http://localhost:5500/');
  await page.fill('#task-input', 'Test Task');
  await page.click('#add-task');

  // Delete the task
  await page.click('.task .delete-task');

  const tasks = await page.$$eval('.task',
    tasks => tasks.map(task => task.textContent));
  expect(tasks).not.toContain('Test Task');
});
```

Test 3: Test If a User Can Mark a Task as Complete

This test checks whether the application correctly **marks a task as complete** when the user **clicks the checkbox of a task**.

The **first** part of this test is the same as the first test – it **navigates to the app** and **adds a new task**:

```
// Verify if a user can delete a task
test('user can delete a task', async ({ page }) => {
  // Add a task
  await page.goto('http://localhost:5500/');
  await page.fill('#task-input', 'Test Task');
  await page.click('#add-task');
```

We add a line that clicks on the **`[Complete]`** button of the task. The task should be marked as complete:

```
  // Mark the task as complete
  await page.click('.task .task-complete');
```

We then write a line that finds the first element with the class **`.task.completed`**. If a task was marked as complete, it should have this class:

```
const completedTask = await page.$('.task.completed');
```

Finally, we check that **`completedTask`** is not null. If a task was correctly marked as complete, **`completedTask`** should be an element, and the test should pass:

```
  expect(completedTask).not.toBeNull();
});
```

Finally, the code should look like this:

```
// Verify if a user can mark a task as complete
test('user can mark a task as complete', async ({ page }) => {
  // Add a task
  await page.goto('http://localhost:5500/');
  await page.fill('#task-input', 'Test Task');
  await page.click('#add-task');

  // Mark the task as complete
  await page.click('.task .task-complete');
  const completedTask = await page.$('.task.completed');
  expect(completedTask).not.toBeNull();
});
```

Test 4: Test If a User Can Filter Tasks

This final test checks whether the application **correctly filters tasks based on their status**.

The **first** part of this test is the same as the first test – it **navigates to the app** and **adds a new task**:

```
test('user can filter tasks', async ({ page }) => {
  // Add a task
  await page.goto('http://localhost:5500/');
  await page.fill('#task-input', 'Test Task');
  await page.click('#add-task');
```

We should change the selected option of the filter to **`'Completed'`**. The list should now only show completed tasks:

```
  // Mark the task as complete
  await page.click('.task .task-complete');
```

After that we should find the first task that is not marked as complete. If the filter is working correctly, there should be no such tasks:

```
  // Filter tasks
  await page.selectOption('#filter', 'Completed');
```

The last step is to check that **`incompleteTask`** is **`null`**. If the filter correctly shows only completed tasks, **`incompleteTask`** should be **`null`**, and the test should pass.

```
  const incompleteTask = await page.$('.task:not(.completed)');
  expect(incompleteTask).toBeNull();
});
```

The code should finally look like this:

```
// Verify if a user can filter
test('user can filter tasks', async ({ page }) => {
  // Add a task
  await page.goto('http://localhost:5500/');
  await page.fill('#task-input', 'Test Task');
  await page.click('#add-task');

  // Mark the task as complete
  await page.click('.task .task-complete');

  // Filter tasks
  await page.selectOption('#filter', 'Completed');
  const incompleteTask = await page.$('.task:not(.completed)');
  expect(incompleteTask).toBeNull();
});
```

Now that we have written the test, let's run them and check if everything works properly.

3. Run tests

Now that we have written all of the test, we can run them, using this command:

```
npx playwright test
```

We should see that all of the tests pass:

```
PS D:\SoftUni\to-do-app> npx playwright test

Running 4 tests using 1 worker

  ✓  1 tests\todo.test.js:3:1 › user can add a task (2.0s)
  ✓  2 tests\todo.test.js:11:1 › user can delete a task (510ms)
  ✓  3 tests\todo.test.js:21:1 › user can mark a task as complete (493ms)
  ✓  4 tests\todo.test.js:31:1 › user can filter tasks (518ms)

  4 passed (5.5s)
```