TEST PLAN

Test Scope and Coverage

Given that the Todo List App is a task management solution designed to help users organize, prioritize, and monitor their tasks, its core functionality revolves around the execution of CRUD (Create, Read, Update, Delete) operations across multiple task categories. While the user interface is intentionally streamlined to enhance usability and productivity, it's critical from a QA perspective to ensure both functional integrity and user experience. Therefore, the test scenarios outlined below aim to validate the fundamental task flows and edge cases, data persistence, error handling, and UI responsiveness, ensuring the app behaves as expected and consistently under various usage conditions.

Test Scenarios

Number	Scenario	Purpose	Type	Priority
1	Create a new task in a list with an icon and a name	Verifies task creation	Functional	High
2	Edit an existing task	Ensures edit flow works correctly	Functional	High
3	Delete a task	Validates task removal	Functional	High
4	Create a new appointment inside the task created	Ensures new list creation and icon display	Functional	High
5	Error when creating a list without a name	Tests for form validation	UI/UX	Medium
6	Confirm tasks persist after app restart	Validates data persistence	Regression	High
7	Switch between lists and ensure the correct task count	Ensures proper navigation and data display	Functional	High
8	Measure app launch time (cold & warm start)	Performance benchmarking	Performance	Medium
9	Scroll through long task lists	Tests smoothness and responsiveness	Performance	Medium

10	Tap on each icon to verify the correct list filter	Ensures icon-list mapping is correct	Functional/UI	Medium

Test Data & Environments

Test Data requirements:

- At least two test lists with distinct icons;
- Multiple tasks per list (some marked complete or done);
- One list without tasks;
- Sample tasks with long descriptions and emojis to test edge cases
- Establish a DB for data persistence (Flask, AWS, microservices combined)

Test Environment:

- iOS 18 on simulator: iPhone 14 Pro or higher
- Xcode simulator
- Appium inspector

Automation Approach

The approach chosen is Code-first using Appium with XCTest/XCUITest backend. Appium supports cross-platform automation and can interact with native elements on iOS through the XCTest driver. Since the app uses SwiftUI and is iOS-only, Appium with direct XCUITest integration provides flexibility while enabling us to build scalable and maintainable tests in Java.
Here are some three initial test scenarios to be automated (will be implemented first):

Create a task flow:

1. Open the app
2. Tap the + icon to add a task
3. Type the task name in a text field (e.g, “Test1”)
4. Press return on the iPhone keyboard
5. Return to the main page
6. Assert that the task appears in the list

Create an appointment inside a task:

1. Go to the task created on the main page
2. When on the task page, click on the new reminder button
3. Check if a checkbox will be displayed on the left-hand side of the screen

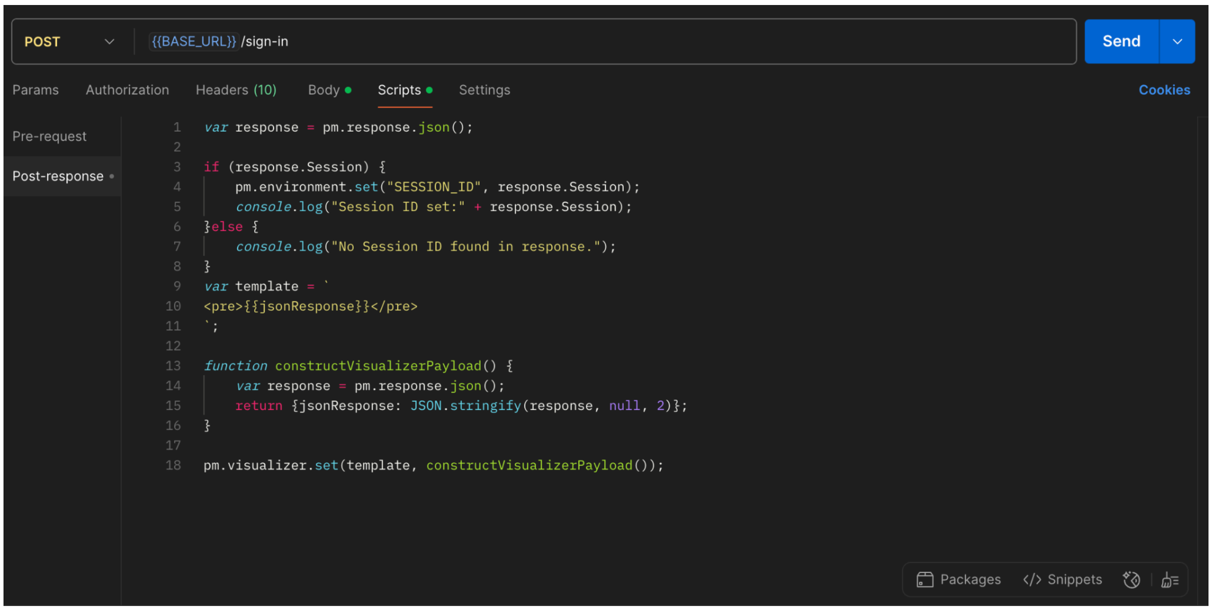
4. Click on the text field at the right side of the checkbox
5. Type the appointment required
6. Press return on the iPhone keyboard
7. Click on the back button, to return to the main page

Delete a task flow:

1. When on the main page, swipe left on any task name
2. Check if a red button “delete” will be displayed
3. Click on the red button
4. Check if the task has been deleted

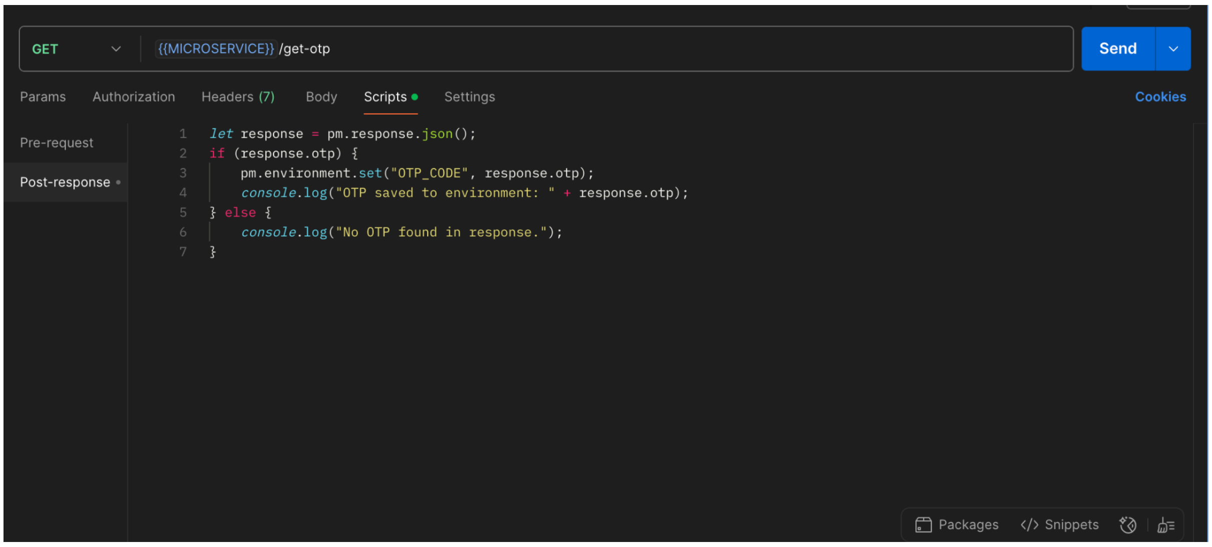
API Testing

Thinking in the long run, the app will be available, considering microservices integrated into the app functionality and some external services needed to guarantee security and performance. So, a hypothetical scenario is the following:


POST {{BASE_URL}} /sign-in

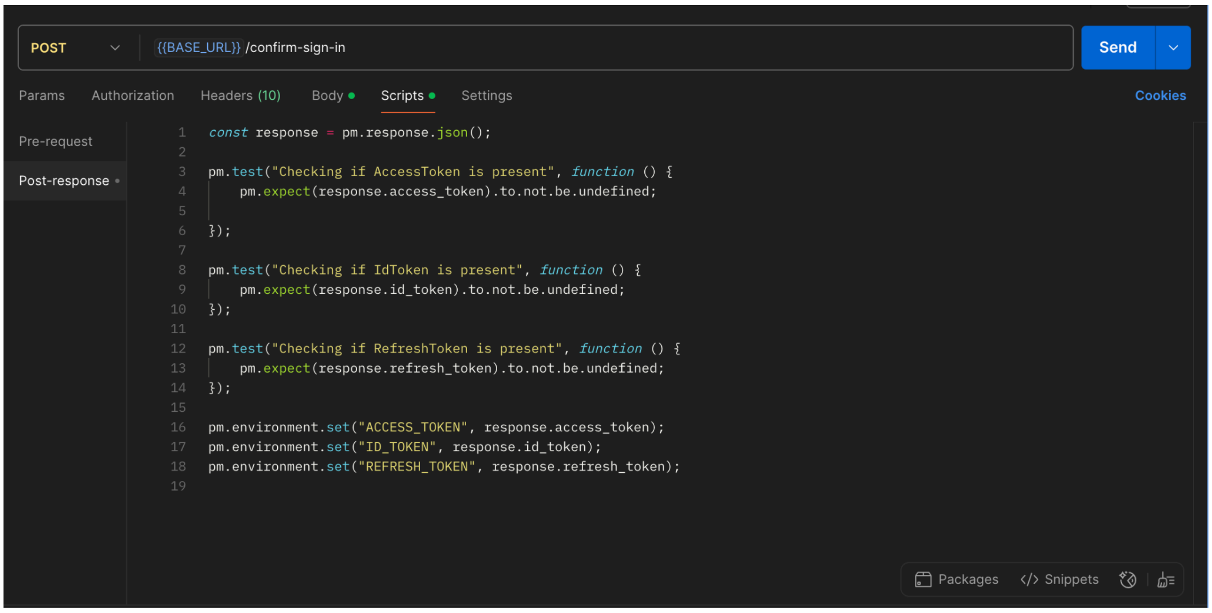
```
var response = pm.response.json();

if (response.Session) {
    pm.environment.set("SESSION_ID", response.Session);
    console.log("Session ID set:" + response.Session);
}else {
    console.log("No Session ID found in response.");
}
var template = `
<pre>{{jsonResponse}}</pre>
`;

function constructVisualizerPayload() {
    var response = pm.response.json();
    return {jsonResponse: JSON.stringify(response, null, 2)};
}

pm.visualizer.set(template, constructVisualizerPayload());
```

A service to provide a session ID for each user who will access and register their name (credentials)



```javascript
let response = pm.response.json();
if (response.otp) {
    pm.environment.set("OTP_CODE", response.otp);
    console.log("OTP saved to environment: " + response.otp);
} else {
    console.log("No OTP found in response.");
}
```

A future OTP authentication that will be possible to create environment variables and store those variables to be used for several different requests



```javascript
const response = pm.response.json();

pm.test("Checking if AccessToken is present", function () {
    pm.expect(response.access_token).to.not.be.undefined;

});

pm.test("Checking if IdToken is present", function () {
    pm.expect(response.id_token).to.not.be.undefined;
});

pm.test("Checking if RefreshToken is present", function () {
    pm.expect(response.refresh_token).to.not.be.undefined;
});

pm.environment.set("ACCESS_TOKEN", response.access_token);
pm.environment.set("ID_TOKEN", response.id_token);
pm.environment.set("REFRESH_TOKEN", response.refresh_token);

```

A confirm sign-in page to assert if we fetched all the required tokens for the Session ID