


Unit Testing with Firebase Local Emulator Suite

December 1, 2021

Abstract

This document describes the unit testing using Jest framework in a Nx environment. It covers the configuration for Jest tests, installing firebase emulator tools, and sample code for testing a firebase service module. Some useful information to debug failing tests is also provided at the end.

Using this prototype and test workflow with emulator can help in writing simple tests for database interactions that do not depend on app's logic in a safe local environment without affecting cloud service billing costs.

Contents

Unit Testing

Unit Testing is also known as Component Testing, is a level of software testing where individual units or components of the software are tested. The purpose is to validate that each unit of the software performs as designed.

We use Jest, the open-source JavaScript Testing Framework to write unit tests. The codebase is maintained in a Nx monorepo.

Jest offers the following advantages:

- Designed with monorepos in mind; isolates important parts of a monorepo.
- Immersive watch mode providing near-instant feedback when developing tests.
- Ability to use Snapshot Testing to validate features.

In the next sections, we describe how to use Jest and Firebase Emulator for performing unit testing in a Nx monorepo environment.

Using Jest Test Framework

Nx provides a plugin for Jest which makes it easier to develop tests and improve the workflow.

If not already installed, install **`@nrwl/jest`** using your package manager

```
npm install --save-dev @nrwl/jest
```

Next, use the **`jest-project`** generator to scaffold a project (app / library).

```
nx g @nrwl/jest:jest-project --project=<project-name>
```

A typical Nx workspace will be setup like below:

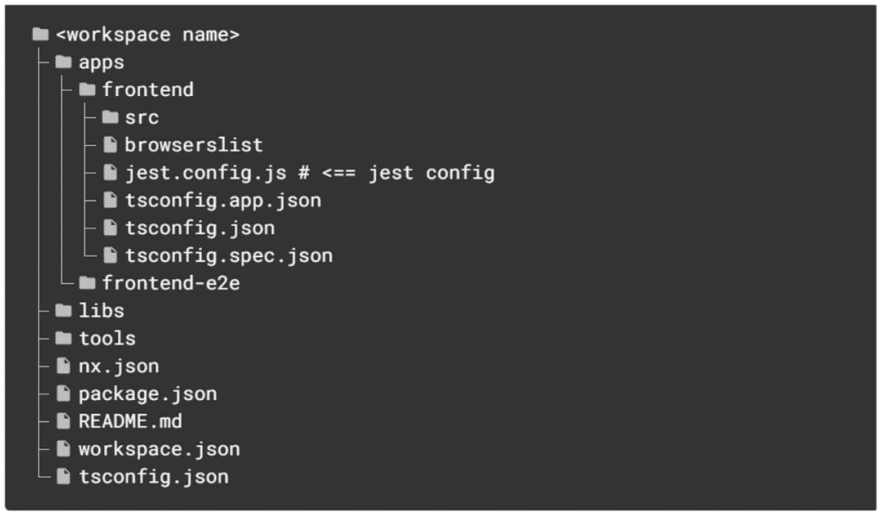

Figure 1. The top-level directory structure

The jest configuration file jest.config.js is stored in each project's root directory. It contains configuration for paths, timeout, etc. for customizing according to specific project needs.

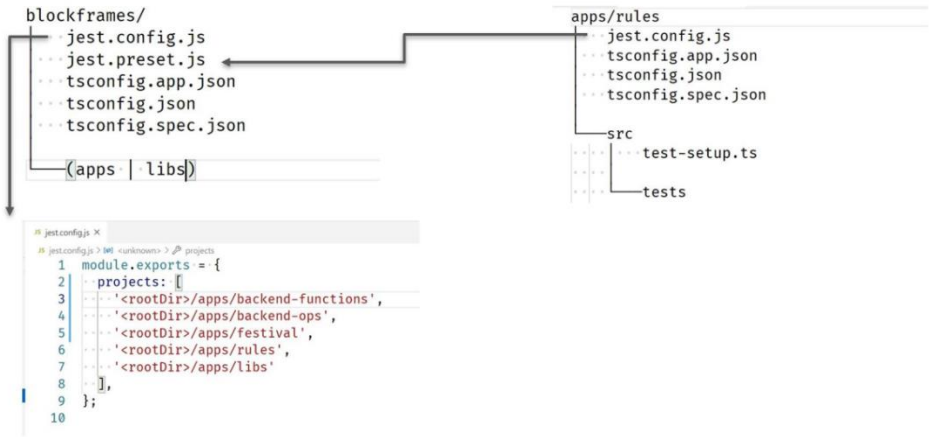

Figure 2. Jest configuration file

It is recommended to write many separate unit tests per component to get 100% code coverage. Tests should be reliable, trustworthy, fast, and maintainable.

Jest tests are organized in logical groups of tests (suites) in 'describe()' blocks. Other describe blocks can be further nested inside this as needed for logical separation. Within describe block, individual tests are specified inside the 'it' block.

Each test (it) block should test a specific functionality of code and tests should not be dependent on other test to have a precondition or database setup correctly.

To run Jest test for say frontend project via nx use.

```
nx test frontend --watch
```

`--watch` flag will rerun the tests whenever a file inside the frontend project is updated.

```
PASS apps/erpweb/functions/src/sample/sample.spec.ts
  Sample tests
    ✓ it should add lowercase name (85ms)
    ✓ it should add createdAt (14ms)
    ✓ it should increase companies count (12ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.507s, estimated 1s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

Figure 3. Example test run showing all tests passing

Using Firebase Local Emulator Suite

The Firebase Local Emulator Suite consists of individual service emulators that accurately mimic the behavior of Firebase services. This means you can connect your app directly to these emulators to perform integration testing or QA without touching production data.

By using a local emulator, we can avoid mocking firebase services like auth, database read/write, functions, pub-sub, etc. You can test not only the correctness of the component but also the integration with firestore without having to publish the app to firestore.

The Firebase Local Emulator Suite can be installed and configured for different prototype and test environments, anything from one-off prototyping sessions to production-scale continuous integration workflows.

Figure 4. Using the Firebase Local Emulator Suite

By using a local emulator, we can avoid mocking firebase services like auth, database read/write, functions, pub-sub, etc. You can test not only the correctness of the component but also the integration with firestore without having to publish the app to firestore.

To Install the firebase emulator, run the npm install CLI command from the shell or command prompt as below. Recommended version for firebase-tools is 8.4.3 or above.

```
npm install -g firebase-tools
```

Example unit test code for NotificationService module

The following code demonstrates how to write a unit test for the angular notification service module.

Note the usage of emulator setting **`'localhost:8080'`** inside **`TestBed.configureTestingModule.`** The firebase service access code remains similar to what you would write inside the application.

```
import { TestBed } from '@angular/core/testing';

import { NotificationService } from './notification.service';
import { NotificationStore } from './notification.store';
import { Notification } from './notification.model';
import { AngularFireModule } from '@angular/fire';
import { SETTINGS, AngularFirestoreModule, AngularFirestore } from
       '@angular/fire/firestore';
import { loadFirestoreRules, clearFirestoreData } from '@firebase/testing';
import { readFileSync } from 'fs';
import { Subject } from 'rxjs';

describe('Notifications Test Suite', () => {
  let service: NotificationService;
  let db: AngularFirestore;

  beforeEach(async () => {
    TestBed.configureTestingModule({
      imports: [
        AngularFireModule.initializeApp({ projectId: 'test' }),
        AngularFirestoreModule
      ],
      providers: [
        NotificationService,
        NotificationStore,
        // Use local emulator settings
        { provide: SETTINGS, useValue: { host: 'localhost:8080', ssl: false } }
      ],
    });
    db = TestBed.inject(AngularFirestore);
    service = TestBed.inject(NotificationService);

    await loadFirestoreRules({
      projectId: "test",
      rules: readFileSync('./firestore.test.rules', "utf8")
    });
  });

  afterEach(() => clearFirestoreData({ projectId: 'test' }));

  afterAll(() => db.firestore.disableNetwork());

  it('Should check notif service is created', () => {
    expect(service).toBeTruthy();
  })

  it('Should mark notifications as read', async () => {
    const notif = {
      id: '1',
      app: { isRead: false },
    };
    await db.doc('notifications/1').set(notif);
    await service.readNotification(notif);
    const doc = await db.doc('notifications/1').ref.get();
    const notification = doc.data() as Notification;
    expect(notification.app.isRead).toBeTruthy();
  });

});
```

Debugging failing tests

To debug a failing test, you may set a breakpoint in your code-editor to break in the test code when execution is reached.

Alternatively, the **`--inspect-brk`** flag for node can be used to leverage Chrome DevTools for debugging the failure.

```
node --inspect-brk ./node_modules/@nrwl/cli/bin/nx test
<project-name>
```

Summary

We have covered the entire steps of unit testing workflow using firebase local emulator suite from installation, configuration to writing tests. The changes required for Angular Testbed configuration for connecting with local emulators have been mentioned.

Sample code for testing an angular notification service is given. Information for debugging test failure is provided to correct the code to make the test work correctly according to expectations.

The configuration required for testing standalone modules or other types of modules like libraries is similar. Using the information provided, unit test code for other firebase services like auth, cloud functions, storage, and database rules can be written.