LogRocket
Frontend Analytics

Testing TypeScript apps using Jest

October 14, 2021 · 8 min read

Testing, testing, 1, 2, 3... 🎤

Testing is an important aspect of software engineering. During testing, we determine the expectations and test cases for our application and check that they are met.

JavaScript has a lot of testing frameworks available that can be used across projects developed in React, Angular, Node.js, and TypeScript; but today, we will look at one testing framework in particular called Jest.

Jest is a simple, lightweight testing framework that provides a variety of testing capabilities to JavaScript and TypeScript projects. It provides functionality like assertions, mocking, spies, running tests in parallel, prioritizing failed tests, and coverage analysis. In this article, we will implement some of the functionality that Jest has to offer.

Prerequisites

To demonstrate Jest's testing capabilities, we will use the project from a previous LogRocket article titled, Using Sequelize with TypeScript.

To follow along, you will need to install:

- Git
- Node.js
- A JavaScript package manager (we'll use Yarn)
- An IDE or text editor of your choice, like Sublime Text or Visual Studio Code

Once we have all these installed, we will clone our TypeScript/Sequelize project by running the following:

```
$ git clone git@github.com:ibywaks/cookbook.git
```

Now that we've cloned our TypeScript project, we will add and configure our test framework.

Setting up our test dependencies

First, we will install our test libraries as development dependencies.

```
$ yarn add -D jest @types/jest ts-jest
```

Next, we will configure Jest for our app by creating a file called `jest.config.ts` and adding the following to it:

```
# jest.config.ts

import type {Config} from '@jest/types';
// Sync object
const config: Config.InitialOptions = {
  verbose: true,
  transform: {
  '^.+\\.tsx?$': 'ts-jest',
  },
};
export default config;
```

There are several other options for configuring Jest based on your preference, such as running `jest --init` which creates a `jest.config.js` file with default configuration or adding the configuration within `package.json`. Jest supports many configuration options, but there is no additional benefit to any of the configuration options you choose.

We will also add a Yarn command to run our tests in the script section of our `package.json` file.

```
# package.json

{
  "name": "cookbook",
  "version": "1.0.0",
  "description": "A simple recipe app using typescript and sequelize",
  "main": "dist/index.js",
  "scripts": {
    "test": "jest",
    ...
```

Now that we've configured our Jest library, we will define some structure for our tests.

Setting up our project test structure

For our project, we will focus on two types of functional tests:

- Integration tests — These tests focus on how our application's components interact with each other to achieve expected results, i.e., we test the outcomes of Services, Routes, and Data Access Layers (DALs)
- Unit tests — These tests focus on the actions of a specific component without paying attention to the other components it references, i.e., mocking external components so that we can test a component on its own

Functional tests focus on the functionality and results of an application. There are also non-functional tests that focus on aspects of our application like performance, load test, uptime, etc. — but these are not included within Jest's scope, so we will not cover them here.

Let's create our `tests` directory with the folders for integration tests and unit tests.

```
/tests
  /integration
  /unit
```

Implementing Jest's concepts and test functions

Now that we have created a structure for our tests, we will go through the available Jest concepts and how they can be used in our project.

Setup and teardown

This refers to the tasks we need to perform in order to get our app in the necessary state for testing, such as populating or clearing tables. Jest provides the ability to perform these tasks via functions like `beforeAll`, `beforeEach`, `afterAll`, and `afterEach`.

As the function names suggest, `beforeAll` and `afterAll` run some tasks before and after the other tests in a particular scope are run, while `beforeEach` and `afterEach` run tasks before and after each test within their scope.

“Scope” here refers to the portion of the test these functions have an influence on — for example, when `beforeAll` is called within a `describe` block, which defines a group of related tests, it applies to all of the tests inside the block.

Let’s test this out by writing tests for our `Ingredient` DAL, where we truncate our `Ingredient` table and create some `Ingredient` table entries for our tests that need them before running all the tests.

First, we will create a `dal` directory in `tests/integrations`, then add our test file `ingredient.test.ts`.

```
# tests/integration/dal/ingredient.test.ts

import {Ingredient} from '../../../src/db/models'
import * as ingredientDal from '../../../src/db/dal/ingredient'

const dbTeardown = async () => {
    await Ingredient.sequelize?.query("SET FOREIGN_KEY_CHECKS = 0")
    await Ingredient.truncate({force: true})
    await Ingredient.sequelize?.query("SET FOREIGN_KEY_CHECKS = 1")
}

describe('Ingredient DAL', () => {
    let ingredientId: number
    beforeAll(async () => {
        await dbTeardown()
        ;({id: ingredientId} = await Ingredient.create({
            name: 'Beans',
            slug: 'beans',
        }))
    })
    afterAll(async () => {
        await dbTeardown()
    })

    describe('Create method', () => {
```

In our Ingredient DAL test above, we defined the function `dbTeardown` , which we called in `beforeAll` and `afterAll` of the root `describe` . This means it will run before all of our tests are executed, and once more after all of our tests are executed.

We must consider that some of our tests will cover methods that run queries on our database. To prevent distorting production or development data, we must use a separate database solely for testing.

In this project, we do this by defining in our `.env` a variable called `TEST_DB_NAME` , which we pass to our Sequelize instance during test runs like this:

```
# src/db/config.ts

require('dotenv').config()

import { Dialect, Sequelize } from 'sequelize'

const isTest = process.env.NODE_ENV === 'test'

const dbName = isTest
                 ? process.env.TEST_DB_NAME as string
                 : process.env.DB_NAME as string
const dbUser = process.env.DB_USER as string
const dbHost = process.env.DB_HOST
const dbDriver = process.env.DB_DRIVER as Dialect
const dbPassword = process.env.DB_PASSWORD

const sequelizeConnection = new Sequelize(dbName, dbUser, dbPassword, {
  host: dbHost,
  dialect: dbDriver,
  logging: false,
})

export default sequelizeConnection
```

Note that Jest sets the `NODE_ENV` environment variable during test execution to have a value `'``test``'` .

More great articles from LogRocket:

- Don't miss a moment with The Replay, a curated newsletter from LogRocket
- Learn how LogRocket's Galileo cuts through the noise to proactively resolve issues in your app
- Use React's useEffect to optimize your application's performance
- Switch between multiple versions of Node
- Discover how to animate your React app with AnimXYZ
- Explore Tauri, a new framework for building binaries
- Compare NestJS vs. Express.js

Understanding matchers in Jest

Matchers are the functions of Jest that test the values produced in our test. Primarily, this refers to the functions we append to `expect()` , such as `toEqual` and `toBeNull` .

For example, in `ingredient.test.ts` , we wrote tests to cover the `findOrCreate` method where we expect it to return an existing entry with the same name without updating it.

```
# tests/integration/dal/ingredient.test.ts

describe('findOrCreate method', () => {
        beforeAll(async () => {
            await Ingredient.create({
                name: 'Brown Rice',
                slug: 'brown-rice'
            })
        })

        ...

        it('should return an existing entry where one with same name exists without updating it', async ()
=> {
            const payload = {
                name: 'Brown Rice',
                slug: 'brownrice',
                description: 'test'
            }
            await ingredientDal.findOrCreate(payload)
            const ingredientsFound = await Ingredient.findAll({where: {name: 'Brown Rice'}})

            expect(ingredientsFound.length).toEqual(1)
            expect(ingredientsFound[0].slug).toEqual('brown-rice')
            expect(ingredientsFound[0].description).toBeNull()
```

The `expect` function returns what Jest calls an expect object. We can call matchers on expect objects to assert that an expected value was achieved.

Testing routes and http requests with Jest and SuperTest

We can test the routes defined in our API using Jest and SuperTest. We will use these to test our `/recipes` routes. First, install SuperTest by running:

```
$ yarn add -D supertest @types/supertest
```

Next, create an instance of the SuperTest request agent to call our application's routes against. For this, we will create a helper file that creates a single SuperTest agent instance and shares it with all the tests that need it.

```
# src/index.ts

import express, { Application, Request, Response } from 'express'
import routes from './api/routes'
import dbInit from './db/init'

dbInit()

const port = process.env.PORT || 3000

export const get = () => {
    const app: Application = express()

    // Body parsing Middleware
    app.use(express.json());
    app.use(express.urlencoded({ extended: true }));

    app.get('/', async(req: Request, res: Response): Promise<Response> => {
        return res.status(200).send({ message: `Welcome to the cookbook API! \n Endpoints available at 
http://localhost:${port}/api/v1` })
    })

    app.use('/api/v1', routes)
    return app
}
```

SuperTest takes our Express application instance as a parameter, which we retrieve from our `src/index.ts` .

Now, we will create a folder called `routes` in `tests/integration` and add our test file, which we'll call `recipe.test.ts` .

```
# /tests/integration/routes/recipe.test.ts

import Recipe, { RecipeOutput } from "../../../src/db/models/Recipe"
import { request } from "../../helpers"

const dbTeardown = async () => {
    await Recipe.sequelize?.query("SET FOREIGN_KEY_CHECKS = 0")
    await Recipe.destroy({ cascade: true, truncate: true, force: true });
    await Recipe.sequelize?.query("SET FOREIGN_KEY_CHECKS = 1")
}

describe('Recipe routes', () => {
    let recipeId: number
    let recipe: RecipeOutput

    beforeAll(async () => {


 = await Promise.all([
            Recipe.create({title: 'Pesto pasta', slug: 'pesto-pasta'}),
            Recipe.create({title: 'Caesar salad', slug: 'caesar-salad'}),
        ])
        ;({id: recipeId} = recipe)
    })
```

Testing with mock objects

We've talked about unit tests that focus on testing a single component alone, ignoring the external files or services it calls. To do this effectively, we create mock objects to replace those external files and services.

Let's demonstrate this by writing tests for our review service while mocking the DAL methods it calls.

First, we create a `services` directory under our `unit tests` directory, and add our test file `review.test.ts` to it.

```
# /tests/unit/services/review.test.ts

import {publishReview} from '../../../src/db/services/ReviewService'
import {update as reviewDalUpdate} from '../../../src/db/dal/review'

const reviewId = 10

jest.mock('../../../src/db/dal/review', () => ({
    update: jest.fn(),
}))

const mockDate = new Date('10 Oct 2021').toISOString()
const dateSpy = jest.spyOn(global, 'Date')
    .mockImplementation(() => mockDate)

describe('Review Service', () => {
    afterAll(() => {
        dateSpy.mockRestore()
    })

    describe('Publish', () => {
        it('should accept a payload and call the review dal with it', async () => {
            await publishReview(reviewId)
            expect(reviewDalUpdate).toBeCalledTimes(1)
            expect(reviewDalUpdate).toHaveBeenCalledWith(reviewId, {isPublished: true, publishedOn: new
```

In this test, we test the `publish` method of the Review Service and mock the Review DAL's update method. Using mocks, we can confirm that the external methods are called the number of times and the ways we expect.

Spying and overwriting object attributes

With Jest, we can spy on and overwrite the implementation of an object's attribute. For example, in our Review Service unit test, we spy on the Date class, which is an attribute of Node's global object.

In our above test, we use `jest.spyOn` to overwrite the Date class and define a mock implementation of its constructor to return a particular value every time. Remember to apply `mockRestore` on classes you overwrite

Stop guessing why bugs happen with LogRocket

Get started for free

3 Replies to "Testing TypeScript apps using Jest"

Francis Upton Says:

Reply

February 5, 2022 at 10:07 pm

Very helpful, however there is a small bug in the RegExp that will cause .tsx files to be missed in the jest.config.ts file. The correct one is:

transform: {
'^.+\\.tsx?$': 'ts-jest',
},

Jennadanoy Says: Reply
February 7, 2022 at 11:00 am

Thank you for the catch! The post has been updated.

Philip Says: Reply
October 22, 2022 at 1:16 am

Additionally, that snipped has smart quotes rather than straight quotes, which will throw a syntax error in the file.

Leave a Reply

Our project's current test coverage needs further improvement, because

- We have not covered all our routes, services, or DALs in our tests (you will notice the number of files reported on is much smaller than the number of files we have)
- The files we covered in tests still have several statements, branches, functions, and lines uncovered

Conclusion

Now that we have added some tests to our API, we can run them using `yarn test` and we'll get an output like:

All the code from this article is available on my GitHub. Please share in the comments any cool finds and tricks from test libraries you've used in your own projects!

LogRocket: Full visibility into your web and mobile apps

LogRocket is a frontend application monitoring solution that lets you replay problems as if they happened in your own browser. Instead of guessing why errors happen, or asking users for screenshots and log dumps, LogRocket lets you replay the session to quickly understand what went wrong. It works perfectly with any app, regardless of framework, and has plugins to log additional context from Redux, Vuex, and @ngrx/store.

In addition to logging Redux actions and state, LogRocket records console logs, JavaScript errors, stacktraces, network requests/responses with headers + bodies, browser metadata, and custom logs. It also instruments the DOM to record the HTML and CSS on the page, recreating pixel-perfect videos of even the most complex single-page and mobile apps.

Try it for free.

Ibiyemi Adewakun Follow

Ibiyemi is a full-stack developer from Lagos. When she's not writing code, she likes to read, listen to music, and put cute outfits together.

#jest #typescript