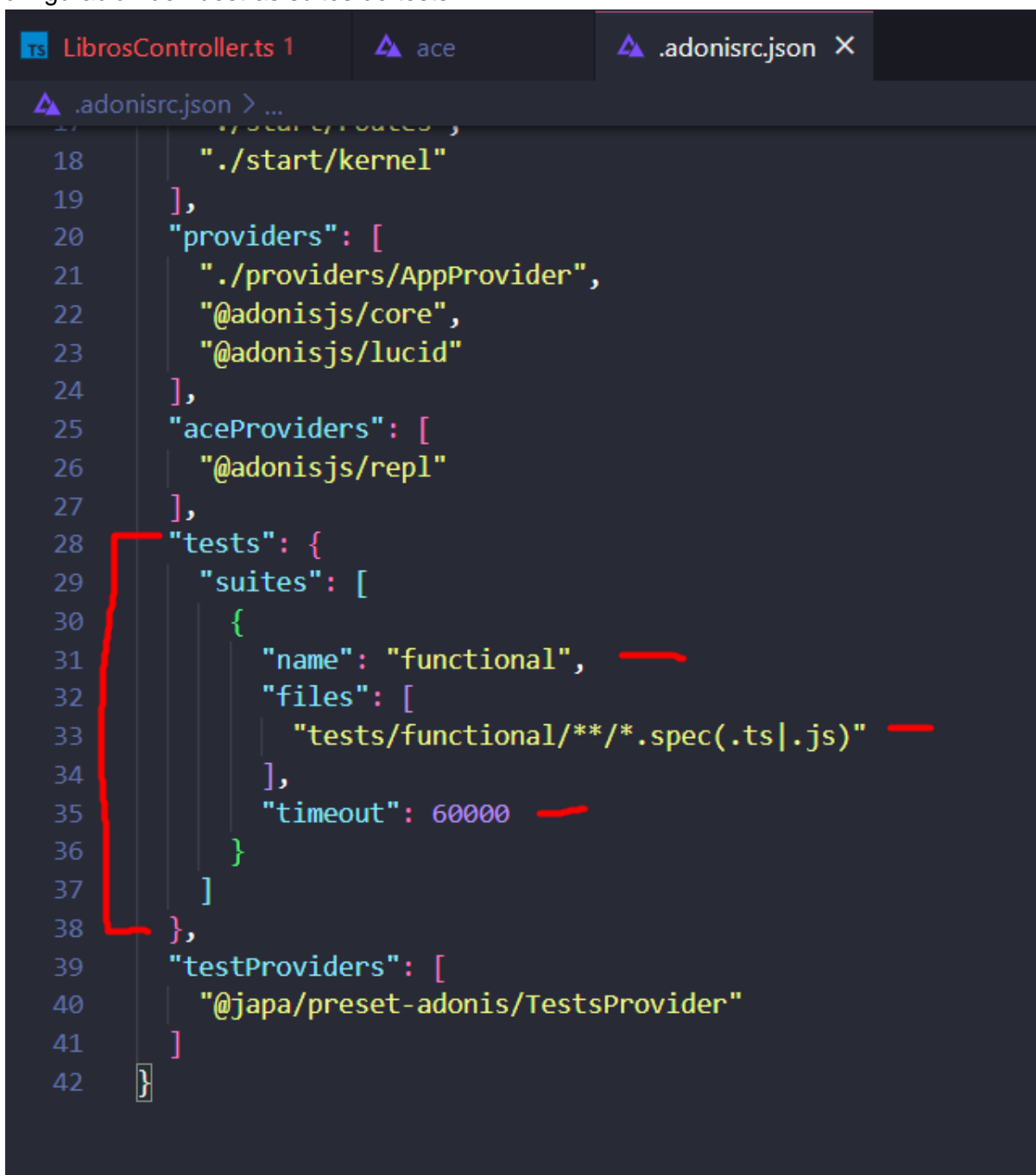


GUIA DE PRUEBAS UNITARIAS

1. Recientemente el framework de adonis ya viene integrado con un framework de testing para Node.js llamado Japa. Este framework viene con todas las herramientas necesarias para probar las aplicaciones de Backend.

En la documentación que podemos encontrar en el siguiente enlace <https://japa.dev/docs> se pueden encontrar los detalles de todos los métodos y plugins que se pueden usar para realizar nuestros tests.

2. Uno de los archivos de configuración más importantes es el **.adonisrc.json** donde está la configuración de nuestras suites de tests.

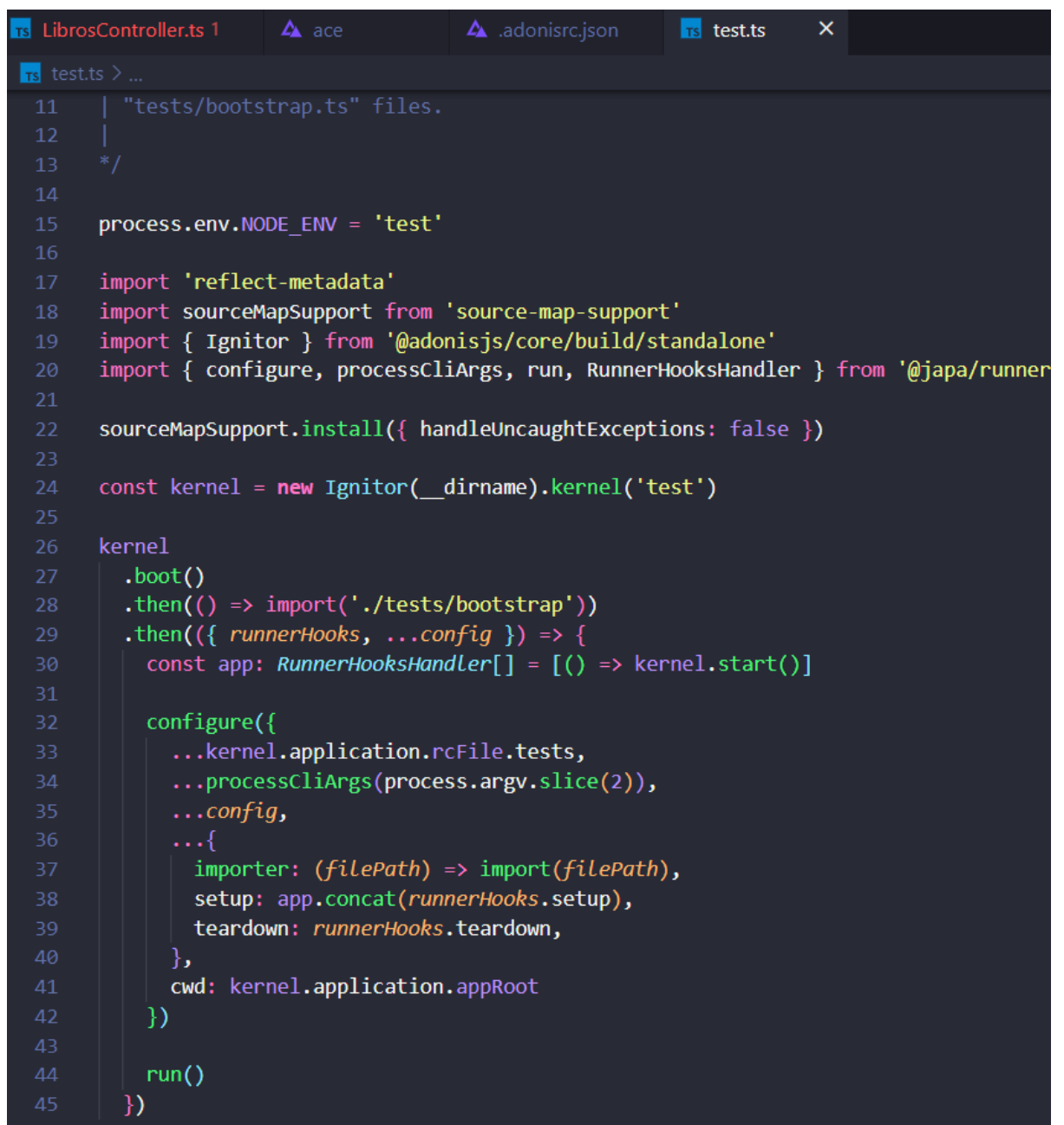


```
17 // Start, Routes,
18   "./start/kernel"
19 ],
20 "providers": [
21   "./providers/AppProvider",
22   "@adonisjs/core",
23   "@adonisjs/lucid"
24 ],
25 "aceProviders": [
26   "@adonisjs/repl"
27 ],
28 "tests": {
29   "suites": [
30     {
31       "name": "functional",
32       "files": [
33         "tests/functional/**/*.spec(.ts|.js)"
34       ],
35       "timeout": 60000
36     }
37   ]
38 },
39 "testProviders": [
40   "@japa/preset-adonis/TestsProvider"
41 ]
42 }
```

El arreglo de suites contiene la información de cada suite que configuremos, cada suite tiene las propiedades de “name” que es el nombre de la suite, “files” que es una expresión que determina en que parte de nuestro proyecto vamos a buscar los archivos de tests y el “timeout” que es el tiempo máximo que puede tardar un test en completarse de lo contrario va a fallar por tardanza.

Nótese que la extensión de los archivos de prueba debe ser .spec, según dice la expresión de “files” si se desea que tengan otra extensión como .test debe cambiarse en esa propiedad la expresión.

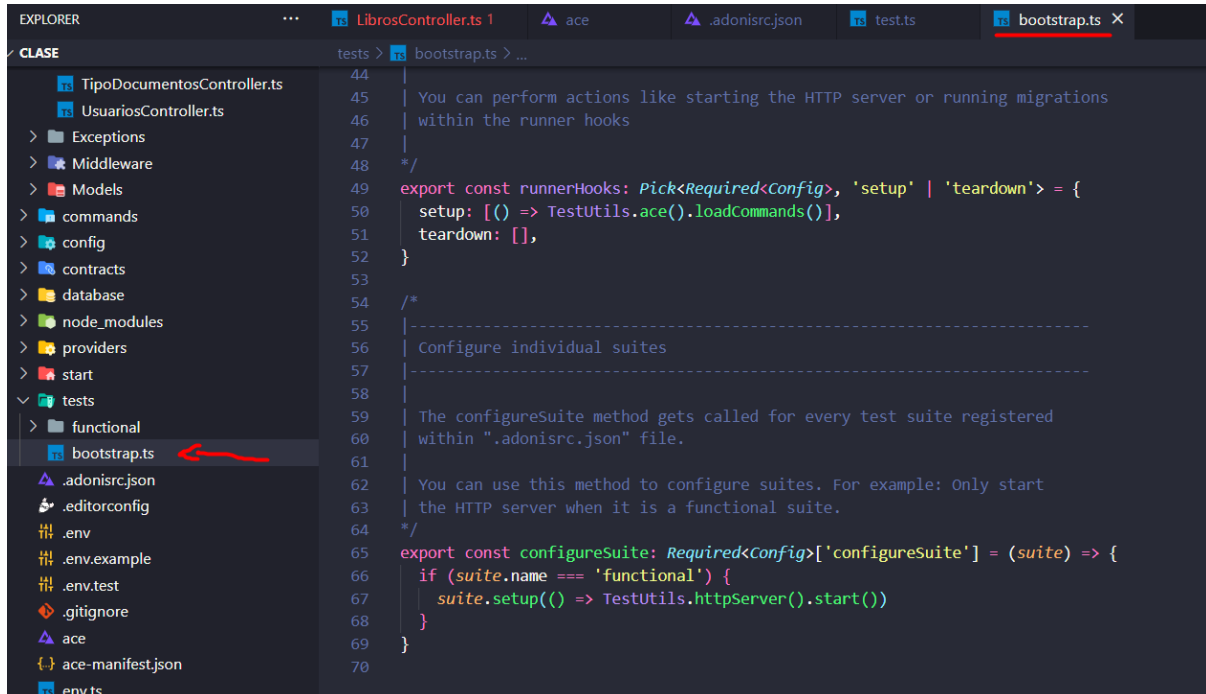
3. Otro archivo de configuración importante es el test.ts en la raíz del proyecto, aquí encontramos el nombre que tiene que tomar el archivo .env para que pueda ser tenido en cuenta por el framework de pruebas.



```
test.ts > ...
11 | "tests/bootstrap.ts" files.
12 |
13 */
14
15 process.env.NODE_ENV = 'test'
16
17 import 'reflect-metadata'
18 import sourceMapSupport from 'source-map-support'
19 import { Ignitor } from '@adonisjs/core/build/standalone'
20 import { configure, processCliArgs, run, RunnerHooksHandler } from '@japa/runner'
21
22 sourceMapSupport.install({ handleUncaughtExceptions: false })
23
24 const kernel = new Ignitor(__dirname).kernel('test')
25
26 kernel
27   .boot()
28   .then(() => import('./tests/bootstrap'))
29   .then(({ runnerHooks, ...config }) => {
30     const app: RunnerHooksHandler[] = [() => kernel.start()]
31
32     configure({
33       ...kernel.application.rcFile.tests,
34       ...processCliArgs(process.argv.slice(2)),
35       ...config,
36       ...{
37         importer: (filePath) => import(filePath),
38         setup: app.concat(runnerHooks.setup),
39         teardown: runnerHooks.teardown,
40       },
41       cwd: kernel.application.appRoot
42     })
43
44     run()
45   })
```

Si observamos la línea 15 de la imagen anterior dice que el nombre que tiene que tomar el archivo .env es test, por lo que el archivo de environments que se debe crear para las pruebas debe ser ".env.test" este valor se puede cambiar según se desee en esa misma línea. En este caso lo que hacemos es un nuevo archivo con la extensión .env.test en la raíz del proyecto y le copiamos toda la información del archivo .env.

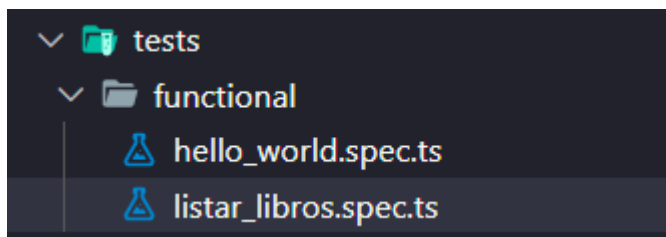
4. El último archivo de configuración que se debe tener en cuenta es el bootstrap.ts que se encuentra dentro de la carpeta tests.



```
44 |  
45 | You can perform actions like starting the HTTP server or running migrations  
46 | within the runner hooks  
47 |  
48 | */  
49 | export const runnerHooks: Pick<Required<Config>, 'setup' | 'teardown'> = {  
50 |   setup: [() => TestUtils.ace().loadCommands()],  
51 |   teardown: [],  
52 | }  
53 |  
54 | /*  
55 | -----  
56 | Configure individual suites  
57 | -----  
58 |  
59 | The configureSuite method gets called for every test suite registered  
60 | within ".adonisrc.json" file.  
61 |  
62 | You can use this method to configure suites. For example: Only start  
63 | the HTTP server when it is a functional suite.  
64 | */  
65 | export const configureSuite: Required<Config>['configureSuite'] = (suite) => {  
66 |   if (suite.name === 'functional') {  
67 |     suite.setup(() => TestUtils.httpServer().start())  
68 |   }  
69 | }  
70 |
```

Aquí en la línea 66 hay un condicional que indica que si el nombre de la suite que se ejecuta es la de "functional" se levanta el servidor para que se puedan hacer las peticiones http. Si se tiene una suite distinta que requiere un servidor web se debe modificar el condicional para que levante el servidor http cuando el nombre de la suite sea el correspondiente.

5. para crear un test simplemente creamos un archivo con la extensión que se indica en la suite y redactamos nuestro test con la siguiente estructura



```
tests  
└─ functional  
   ├── hello_world.spec.ts  
   └── listar_libros.spec.ts
```

```

1  import { test } from '@japa/runner'
2  import { obtenerTokenAutorizacion } from './TestAuths'
3
4  test('listar libros', async ({client, assert}) => {
5      const token = await obtenerTokenAutorizacion()
6      const response = await client.get('api/libros/listar')
7          .header('Authorization', `Bearer ${token}`)
8      response.assertStatus(200)
9      assert.isArray(response.body())
10 })
11

```

importamos la dependencia test de @japa/runner que es un función y la llamamos, la función recibe dos parámetros, el nombre del test y la función que es el test como tal, esta última recibe dos parámetros, el “client” y el “assert”; el “client” es un cliente http con el que podemos consultar el api que hemos creado y el “assert” es un objeto que contiene funciones que me permiten probar los resultados obtenidos. Para más detalle de cómo funcionan estos dos objetos, debemos leer la documentación de “Japa”.

En este caso estamos consultando nuestro endpoint de consultar libros, llamando a su ruta con el client y luego nos aseguramos de que la respuesta fue de status 200 (línea 8) y que el cuerpo de la respuesta es un arreglo (línea 9).

También creamos el archivo TestAuths.ts, el cual tendrá la función llamar el token por medio de la función login, enviando los parámetros de un usuario registrado.

```

1  import axios from "axios";
2  import Env from "@ioc:Adonis/Core/Env";
3
4  export async function obtenerTokenAutorizacion(): Promise<string>{
5      let endpoint = "/api/login"
6      let body = {
7          correo: "usuario01@gmail.com",
8          contrasena: "12345"
9      }
10     let axiosResponse = await axios.post(`${Env.get("PATH_APP") + endpoint}`, body)
11     return axiosResponse.data["token"]
12 }

```

En script de package.json debemos agregar:

```

"test": "node -r @adonisjs/assembler/build/register japaFile.ts",
"coverage": "nyc npm run test"

```

- una vez escrito el test lo corremos con “node ace test”. Debemos tener en cuenta que para correr el test , la base de datos del proyecto debe estar activa.

```
C:\Windows\System32\cmd.e x + v
* listar libros (1s)
FAILED
Tests : 1 Failed (1)
Time : 1s

* listar libros

Assertion Error: expected 400 to equal 200

- Expected - 1
+ Received + 1

- 200
+ 400

at Object.executor D:/clase/tests/functional/listar_libros.spec.ts:7
2|
3| test('listar libros', async ({ client, assert }) => {
4|   const token = 'aqui va el token de autorización'
5|   const response = await client.get('/api/libro/listar')
6|     .header('Authorization', `Bearer ${token}`)
7|     .response().assertStatus(200)
8|     .assert.isArray(response.body())
9| })
10|
```

En la imagen anterior se ejecuta el test listar libros y este falla porque no se proporciona un token real, por lo tanto, el status de la respuesta no coincide con el esperado. Después que el token si sea correcto, se recibe entonces un status 200, y la función deberá pasar la prueba unitaria.

7. para sacar la cobertura de mis tests necesitamos instalar una librería llamada “nyc” con el comando “npm i nyc -D”
8. Corremos el comando “npx nyc node ace test”, esto va a generar los archivos de cobertura.
9. Luego, corremos el comando “npx nyc report --reporter=lcov” nótese que --reporter se escribe con doble guion al inicio, esto va a generar el archivo “lcov.info” dentro de la carpeta coverage, que luego se utilizará para sacar el informe de cobertura con Sonarqube.
10. En la raíz del proyecto creamos un archivo sonar-project.properties con todas las configuraciones de sonar y exclusiones.
11. Por último corremos el escaner de calidad de código de Sonarqube.