1 Code Under Test

The code under test throughout this document is a miniature note taking API. It has only 2 resources, namely, notebook and note. All CRUD operations are implemented for each of the 2 resources. The notes controller has some endpoints that do their work in the background and communicate with external API.

You can find the code under test here

2 Tackled Problems

In this section we describe the problems that we covered and the solutions we implemented to address those problems.

2.1 How to test an endpoint?

2.1.1 Problem definition

The heart of every API is a collection of endpoints that the API responds to and serve. An endpoint is a communication channel through which an API receives HTTP requests and exposes its resources to an API consumer. Testing an endpoint would require the ability to send fake requests to this specific endpoint with certain parameters and also be able to consume the response sent back by the endpoint.

2.1.2 Solution

Rspec offers the ability to send fake requests to an endpoint that are routed inside the application exactly the same way a legit request be routed. It also offers the ability to send a specific type of request, e.g. GET request, along with the needed parameters and request headers.

The following code snippet shows how we tested a GET endpoint that should respond with a list of all requested resources. We can notice at line 8 a GET request is sent on the url defined at line 2. Success is validated in lines 10 and 11 by expecting the status code to be 200 which is the success status code of a GET request and expecting the body to be as expected, in this case an empty array.

```
1   context '#index' do
2       let!(:url) { '/notebooks' }
3
4       context 'When there are no notebooks' do
5         let!(:expected_response) { [] }
6
7           it 'should return an empty array' do
8               get url #Sends GET request to url /notebooks
9
10              expect(response.status).to eq 200
11              expect(response.body).to eq expected_response.to_json
12          end
13      end
14  end
```

Listing 1: Testing a GET endpoint

2.2 How to validate a large endpoint response?

2.2.1 Problem definition

As observed in Listing 1, one method of validating that an endpoint is working properly, is to validate that it returns the expected response. In the previous example, the expected response was just an empty array, but what if the expected response is a few records each consisting of many attributes?

2.2.2 Solution

Instead of writing the expected response inside the test scripts which will take up unnecessary space and render the test scripts unreadable, Rspec allows storing values in files, usually called

`fixtures` that can be loaded when needed while a test is running.

The following code shows how can we modify the example in `Listing` 1 to load the expected response from a file located in the specified directory. In line 8 the file is loaded, parsed and assigned to the expected response variable for later use.

```
1   context 'When there are some notebooks' do
2     before do
3       Notebook.create(title: "Notebook1", description: "Notebooks1 does work1")
4       Notebook.create(title: "Notebook2", description: "Notebooks1 does work2")
5       Notebook.create(title: "Notebook3", description: "Notebooks1 does work3")
6     end
7
8     let!(:expected_response) do
9       JSON.parse(File.read("#{Rails.root}/spec/fixtures/controllers/notebooks/
    database_has_some_values.json"))
10    end
11
12    it 'should return an array of all records in the DB' do
13      get url
14
15      expect(response.status).to eq 200
16      expect(response.body).to eq expected_response.to_json
17    end
18  end
```

Listing 2: Using fixtures while testing endpoints

And the fixture file would contain the expected result as follows:

```
1   [
2     {
3       "id": 1,
4       "title": "Notebook1",
5       "description": "Notebooks1 does work1"
6     },
7     {
8       "id": 2,
9       "title": "Notebook2",
10      "description": "Notebooks1 does work2"
11    },
12    {
13      "id": 3,
14      "title": "Notebook3",
15      "description": "Notebooks1 does work3"
16    }
17  ]
```

Listing 3: Fixture used in `Listing` 2

2.3 How to validate data persistence in databases?

2.3.1 Problem definition

In `Listing` 1 and 2, we were testing `GET` endpoints whose job is usually returning data stored in a database. But what if we are to test other types of endpoints, such as, `POST` or `DELETE`? Those types of endpoints usually insert or delete records stored in a database. So how to validate that records have been inserted in or deleted from a database?

2.3.2 Solution

`Rspec` expectation `gem` provides a handful of methods that allow for expecting change. Those can be used to expect a change in the number of records in a database before or after inserting or deleting a record.

The following example shows a test for a `DELETE` endpoint that given an `id` deletes the associated record. In line 9 we expect that the size of the `Notebook` table will be reduced by 1 as a result of deleting a record and in line 15 we expect that querying the database for the deleted record would raise an exception of type `ActiveRecord::RecordNotFound`.

```
1   context 'delete request' do
2       it 'should return 202' do
3         delete url
4
5         expect(response.status).to eq 202
```

```
6         end
7
8         it 'should decrease the size of the notebook relation by 1' do
9           expect{ delete url }.to change{ Notebook.count }.from(Notebook.count).to(
    Notebook.count - 1)
10        end
11
12        it 'should delete the requested record' do
13          delete url
14
15          expect { Notebook.find(id) }.to raise_error(ActiveRecord::RecordNotFound)
16        end
17    end
```

Listing 4: Validating data persistence in databases.

2.4 How to use test hooks?

2.4.1 Problem definition

While writing large test suites, it's often the case that there will be common steps that need to run at a specific time, either `before/after` the entire test suite or `before/after` each individual test, etc.

2.4.2 Solution

`Rspec` offers support for `Test Hooks` which allow executing some code at specific events.
The following example shows how a `before` test hook is used to populated the database with data before running a test. In line 2, a `before` test hook is defined that inserts 2 records in the database before the test at line 7 runs.

```
1     context 'When params are correct' do
2         before do
3           nb1.notes.create(title: "Note1", country: "Egypt")
4           nb2.notes.create(title: "Note2", country: "U.S.")
5         end
6
7         it 'should return an array of all records in the DB' do
8           get url
9
10          #Do validations here
11        end
12    end
```

Listing 5: Using test hooks.

Test hooks can be used in so many different ways. Another common usage of a test hook is to clean the database before running each test so that a test would run in an isolated environment from any previous tests that may have run before it and altered the database state. The following example shows how this is done using a `gem` called `database-cleaner`. In line 1 an `around` test hook is defined which runs around each test. In line 2, the database is cleaned before running the test in line 3.

```
1     config.around(:each) do |example|
2         DatabaseCleaner.cleaning do
3           example.run
4         end
5     end
```

Listing 6: Using test hooks.

2.5 How to stub external API requests?

2.5.1 Problem definition

In many cases, an `API` might need to communicate with another `API` for any reason which would require an internet connection and that the called `API` is up and running. But what would happen if the test ran in an offline environment? Or what if the called `API` was down or changed it's response? That should not affect the test outcome.

2.5.2 Solution

We implemented an endpoint that creates resources based on a set of parameters, one of which is an `IP` address that we use to populate the `Country` attribute for that resource. For this task, we communicate with an external geo-location `API` that given an `IP` address, returns the `Country` from which this address originates.
`Rspec` provides a `gem` called `webmock` which enables an application to disable all external communications and instead stub certain requests with predetermined responses.

The following example shows how to stub an external request using a `before` test hook as described in section 2.5. In line 3 the stubbed `URL` is defined and in line 4, the expected status code and body are defined.

```
1    context 'stub geo-locator request' do
2        before do
3          stub_request(:get, "http://ip-api.com/json/156.204.128.187").
4          to_return(status: 200, body: "expected body")
5        end
6    end
```

Listing 7: Stubbing external API requests.

2.6 How to test background jobs?

2.6.1 Problem definition

Examples in Listing 1, 2 and 4 all test endpoints that do their work in the `foreground`, in other words, they do all the computations in the main application thread causing other activities to block waiting. Often times, those computations might take a long time or be of a sheer volume, and applications need to be responsive, i.e. receive and serve other requests while some previous computations are still being executed. For that use case, background jobs are used to enqueue jobs for deferred execution. This type of configuration requires special methods of validating that a job was enqueued and executed.

2.6.2 Solution

We implemented a controller that performs its work in the background using a messaging queue system called `Sidekiq`. `Rspec` and `Sidekiq` offer a `gem` called `rspec-sidekiq` which enables `Rspec` to test background jobs that use `Sidekiq`.

The following example shows a test for a `POST` endpoint that inserts a record into the database in a background job. In line 6 we validate that the size of the `noota::notes-creator` queue, which is responsible for queuing those jobs to a worker, has increased by 1 and thus enqueued 1 job. Also in line 13 we validate that job parameters are correct and as expected. In line 19 we run the job using the `NotesCreator` worker and do the needed validations to ensure that a record have been inserted as we have learned in section 2.3.

```
1    context 'When params are correct' do
2        let!(:expected_response) { { message: 'success' } }
3
4        it 'should enqueue a job in the noota::notes-creator queue' do
5          expect { post url, params: params }.to change(
6            Sidekiq::Queues['noota::notes-creator'], :size
7          ).by(1)
8        end
9
10       it 'Should enqueue the job with the correct args' do
11         post url, params: params
12
13         expect(
14           JSON.parse(Sidekiq::Queues['noota::notes-creator'].first.
     with_indifferent_access['args'].first
15           ).symbolize_keys
16         ).to eq(expected_job_args)
17
18       it 'should add the the requested note' do
19           NotesCreator.new.perform(args.to_json)
20
21           #do validations here
22       end
```

```
23      end
```

Listing 8: Testing background jobs.

2.7 How to evaluate how good is a test suite?

2.7.1 Problem definition

It's important to be able to evaluate how comprehensive a test suite is. Having a good idea about the quality of a test suite helps setting the level of confidence and pinpointing possible areas of improvement.

2.7.2 Solution

There are a lot of methods to evaluate the quality of a test suite. One of the most commonly used metric in the industry is `Code Coverage`. `Rspec` works with a `gem` called `SimpleCov` that generates detailed code coverage reports, highlighting the total coverage, how many covered lines, what lines are not covered, how many hits per line and lots of other insightful statistics that help developers improve a test suite.