**12 Testing and validating APIs**

This chapter covers

* Generating automatic tests for REST APIs using Dredd and Schemathesis
* Writing Dredd hooks to customize the behavior of your Dredd test suite
* Using property-based testing to test APIs
* Leveraging OpenAPI links to enhance your Schemathesis test suite
* Testing GraphQL APIs with Schemathesis

This chapter teaches you how to test and validate API implementations. Thus far, we’ve learned to design and build APIs to drive integrations between microservices. Along the way, we did some manual tests to ensure our implementations exhibited the correct behavior. However, those tests were minimal, and most importantly, they were purely manual and therefore not repeatable in an automated fashion.

In this chapter, we learn how to run an exhaustive test suite against our API implementations using tools such as Dredd and Schemathesis, tools for API testing that are part of every API developer’s tool kit. Both Dredd and Schemathesis work by looking at the API specification and automatically generating tests against our API server. For an API developer, this is very handy because it means you can focus your efforts on building your APIs instead of testing them.

By using tools such as Dredd and Schemathesis, you can save time and energy while resting assured that the implementation you’re delivering is correct. You can run Dredd and Schemathesis in combination, or you can choose one of them. As you’ll see, Dredd runs a more basic test suite that is very useful in the early stages of your API development cycle, while Schemathesis runs a robust test suite that is useful before you release your APIs to production.

To illustrate how we test REST APIs, we’ll use the orders API, which we implemented in chapters 2 and 6. To illustrate how we test GraphQL APIs, we’ll use the products API, which we implemented in chapter 10. As a recap, both APIs are part of CoffeeMesh, the fictional on-demand coffee delivery platform that we’re building in this book. The orders API is the interface to the orders service, which manages customers’ orders, while the products API is the interface to the products service, which manages the catalogue of products CoffeeMesh offers.

The code for this chapter is available in GitHub, under the folder named ch12. In section 12.1, we set up the folder structure and the environments to work on this chapter’s examples, so make sure you go through that section if you want to follow along with the examples in this chapter.

**12.1 Setting up the environment for API testing**

In this section, we set up the environment to follow along with the examples in this chapter. Let’s start by setting up the folder structure. Create a new folder called ch12 and cd into it. Within this folder, we’ll copy the orders API and the products API. To keep things simple in this chapter, we use the implementation of the orders API as we left it in chapter 6. Chapter 6 contains a full implementation of the orders API, but it lacks a real database and integration with other services (those features were added in chapter 7). Since the goal of this chapter is to learn how to test APIs, the implementation in chapter 6 is sufficient and will help us stay focused, as we won’t have to set up the database and run additional services. In real life, you’d want to test the API layer in isolation and run integration tests, including on the database. See the README.md file under the ch12/orders folder in the GitHub repository for this chapter for instructions on running the tests against the state of the application after chapters 7 and 11.

Within the ch12 folder, copy the implementation of the orders API from ch06/orders by running the following command:

$ cp -r ../ch06/orders orders

cd into ch12/orders and run the following command to install the dependencies:

$ pipenv install --dev

Don’t forget to include the --dev flag when you run pipenv install, which tells pipenv to install both production and development dependencies. In this chapter, we’ll use development packages to test the orders API. To run the tests, we’ll need pytest, dredd\_hooks, and schemathesis, which you can install with the following command:

$ pipenv install --dev dredd\_hooks pytest schemathesis

To run the tests, we’ll use a slightly modified version of the orders API specification without the bearerAuth security scheme, which you can find under the ch12/orders/ oas.yaml file in the GitHub repository for this book. In this chapter, we’ll focus on testing that the API implementation complies with the API specification, namely, ensuring the API uses the right schemas, the right status codes, and so on. API security testing is a whole different topic, and for that I recommend chapter 11 of Mark Winteringham’s *Testing Web APIs* (Manning, 2022) and Corey J. Ball’s *Hacking APIs* (No Starch Press, 2022).

Let’s now copy the implementation of the products API from chapter 10. Go back to the top level of the ch12 directory by running cd .. and then execute the following command:

$ cp -r ../ch10 products

cd into ch12/products and run pipenv install --dev to install the dependencies. We’ll use pytest and schemathesis to test the products API, which you can install by running the following command:

$ pipenv install pytest schemathesis

We’re now all set up to start testing the APIs. We’ll start our journey by learning about the Dredd API testing framework.

**12.2 Testing REST APIs with Dredd**

This section explains what Dredd is and how we use it to test REST APIs. Dredd is an API testing framework that automatically generates tests to validate the behavior of our API server. It generates tests by parsing the API specification and learning from it how the API is expected to work. Using Dredd is very helpful during development because it means we can focus our efforts on building the API while Dredd ensures that our work is going in the right direction. Dredd was released by Apiary in 2017 as the first tool of its kind (<http://mng.bz/5maq>), and ever since it’s been part of every API developer’s essential tool kit.

In this section, we’ll learn how Dredd works by using it to validate the implementation of the orders API. We’ll start by first running a basic test suite against the API, and then we’ll explore more advanced features of the framework.

**12.2.1 What is Dredd?**

Before we start working with Dredd, let’s take a moment to understand what Dredd is and how it works. Dredd is an API testing framework. As shown in figure 12.1, Dredd works by parsing the API specification and discovering the available URL paths and the HTTP methods they accept.

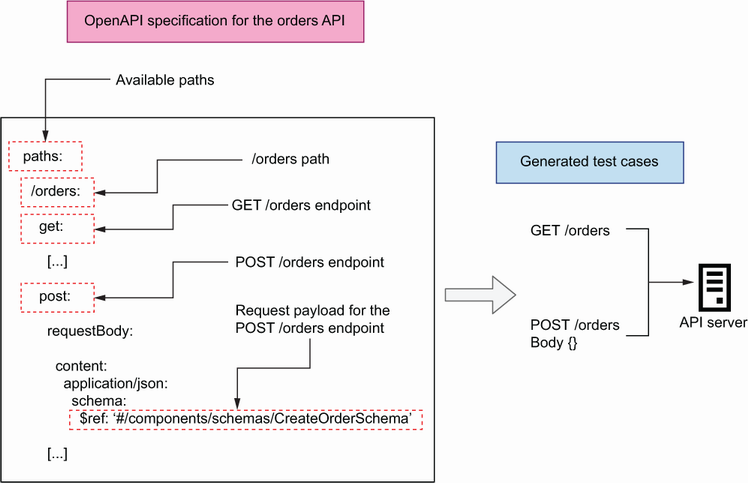


Figure 12.1 Dredd works by parsing the API specification, discovering the available endpoints, and launching tests for each endpoint.

To test the API, Dredd sends requests to each of the endpoints defined in the API specification with the expected payloads, if any, as well as any query parameters accepted by the endpoint. Finally, it checks whether the responses the API receives conform to the schemas declared in the API specification and whether they carry the expected status codes.

Now that we understand how Dredd works, let’s start using it! The next section explains how to install Dredd.

**12.2.2 Installing and running Dredd’s default test suite**

In this section, we install Dredd and run its default test suite against the orders API. cd into ch12/orders and run pipenv shell to activate the environment. Dredd is an npm package, which means you need to have a Node.js runtime available in your machine, as well as a package management tool for JavaScript, such as npm or Yarn. To install Dredd with npm, run the following command from your ch12/orders directory:

$ npm install dredd

This will install Dredd under a folder called node\_modules/. Once the installation is complete, we can start using Dredd to test the API. Dredd comes with a CLI that is available under the following directory: node\_modules/.bin/dredd. The Dredd CLI exposes optional arguments that give us great flexibility in how we want to run our tests. We’ll make use of some of those arguments later in this section. For now, let’s execute the simplest Dredd command to run a test:

$ ./node\_modules/.bin/dredd oas.yaml http://127.0.0.1:8000 --server \

"uvicorn orders.app:app"

The first argument for the Dredd CLI is the path to the API specification file, while the second argument represents the base URL of the API server. With the --server option, we tell Dredd which command needs to be used to start the orders API server. If you run this command now, you’ll get a few warnings from Dredd with the following message (the ellipsis omits the path to the API specification file, which will be different in your machine):

warn: [...] (Orders API > /orders/{order\_id}/cancel > Cancels an order >

➥ 200 > application/json): Ambiguous URI parameter in template:

➥ /orders/{order\_id}/cancel

No example value for required parameter in API description document:

➥ order\_id

Dredd is complaining because we haven’t provided an example of the URL parameter order\_id, which is required in some of the URL paths. Dredd complains about the missing example because it’s unable to generate random values from the specification. To address Dredd’s complaint, we add an example of the order\_id parameter in each URL where it’s used. For example, for the /orders/{order\_id} URL path, we make the modification shown in listing 12.1 (the ellipses represent omitted code). The /orders/{order\_id}/pay and the /orders/{order\_id}/cancel URLs also contain descriptions of the order\_id parameter, so add examples to them as well. Dredd will use the exact value provided in the examples to test the API.

Listing 12.1 Adding examples for the order\_id URL path parameter

# file: orders/oas.yaml

[...]

/orders/{order\_id}:

parameters:

- in: path

name: order\_id

required: true

schema:

type: string

**example: d222e7a3-6afb-463a-9709-38eb70cc670d**  ①

get:

[...]

① We add an example for the order\_id URL parameter.

Once we’ve added examples for the order\_id parameter, we can run the Dredd CLI again. This time, the tests suite runs without problems, and you’ll get a result like this:

complete: 7 passing, 5 failing, 0 errors, 0 skipped, 12 total

complete: Tests took 90ms

INFO: Shutting down

INFO: Finished server process [23593]

This summary tells us that Dredd ran 18 tests, of which 7 passed and 11 failed. The full outcome of the test is too long to reproduce here, but if you scroll up in the terminal, you’ll see that the failing tests are on endpoints that target specific resources:

* GET, PUT, and DELETE /orders/{order\_id}
* POST /orders/{order\_id}/pay
* POST /orders/{order\_id}/cancel

Dredd runs three tests for each of those endpoints, and it expects to obtain one successful response per endpoint. However, in the previous execution, Dredd only obtained 404 responses, which means the server couldn’t find the resources Dredd requested. Dredd is using the ID we provided as an example in listing 12.1 when testing those endpoints. To address this problem, we could add a hardcoded order with that ID to our in-memory list of orders (we’d add it to the database if we were using one for the tests). As we’ll see in the next section, however, a better approach is to use Dredd hooks.

There’s also a failing test for the POST /orders endpoint in which Dredd expects a 422 response. The failed tests for 422 responses happen because Dredd doesn’t know how to create tests that generate those responses, and Dredd hooks will also help us address this problem.

**12.2.3 Customizing Dredd’s test suite with hooks**

Dredd’s default behavior can be limited. As we’ve seen in section 12.2.1, Dredd doesn’t know how to handle endpoints with URL path parameters, such as order\_id in the /orders/{order\_id} URL. Dredd doesn’t know how to produce a random resource ID, and if we provide an example, it expects the sample ID to be present in the system during the execution of the test suite. This expectation is unhelpful, since it means our API is only testable when it’s in a certain state—when certain resources or fixtures have been loaded into the database.

**DEFINITION** In software testing, *fixtures* are the preconditions required to run a test. Typically, fixtures are data that we load into a database for testing, but they can also include configuration, directories and files, or infrastructure resources.

Instead of using fixtures, we can take a better approach by using Dredd hooks. This section explains what Dredd hooks are and how we use them. Dredd hooks are scripts that allow us to customize Dredd’s behavior during the execution of the test suite. Using Dredd hooks, we can create resources for use during the test, save their IDs, and clean them up after finishing the test.

Dredd hooks allow us to trigger actions before and after the whole test suite, and before and after each endpoint-specific test. They are useful for stateful tests that involve creating resources and performing operations on them. For example, we can use hooks to place an order using the POST /orders endpoint, save the ID of the order, and reuse the ID to perform operations on the order, such as payments and cancellations, with other endpoints. Using this approach, we can test that the POST /orders endpoint fulfills its job of creating a resource, and we can test other endpoints with a real resource. As illustrated in figures 12.2, 12.3, and 12.4, we’ll create the following hooks with these steps:

1. After the POST /orders test, we use a hook to save the ID returned by the server for the newly created order.
2. Before the GET, PUT, and DELETE /orders/{order\_id} tests, we use hooks to tell Dredd to use the ID from the order created at point (1). These endpoints are used to retrieve the details of the order (GET), to update the order (PUT), and to remove the order from the server (DELETE). Therefore, after running the DELETE /orders/{order\_id} test, the order will no longer exist in the server.
3. Before the POST /orders/{order\_id}/pay and the POST /orders/ {order\_ id}/cancel endpoints, we use hooks to create new orders for use in these tests. We won’t be able to reuse the ID from point (1), since the DELETE /orders/ {order\_id} test from point (2) deletes the order from the server.
4. For the 422 responses, we need a strategy that generates a 422 response from the server. We’ll use two approaches: for the POST /orders endpoint, we’ll send an invalid payload, while for the remaining endpoints, we’ll modify the order’s URI and include an invalid identifier.

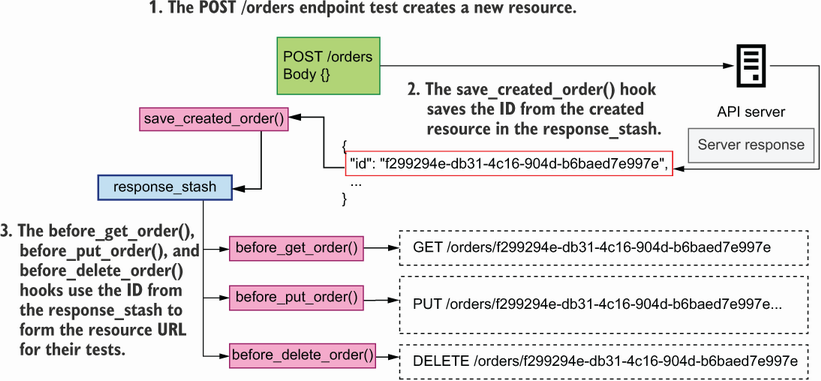


Figure 12.2 After the POST /orders endpoint test, the save\_created\_order()hook saves the ID from the server response body in the response\_stash. The before\_get\_order(), before\_put\_order(), and before\_delete\_order() hooks use the ID from response\_stash to form their resource URLs.

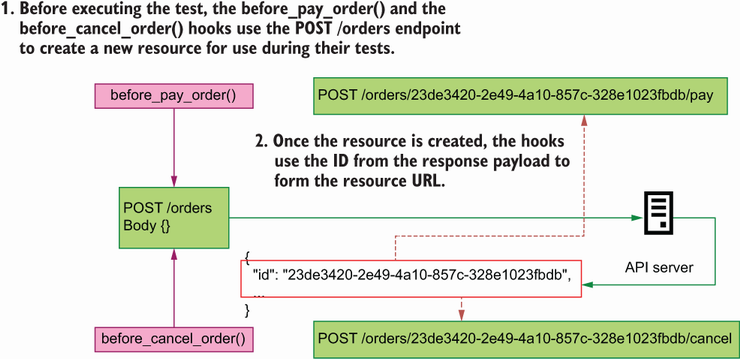


Figure 12.3 Before executing the test, the before\_pay\_order() and the before\_cancel\_ order() hooks use the POST /orders endpoint to place a new order and use the ID from the response payload form their resource URLs.

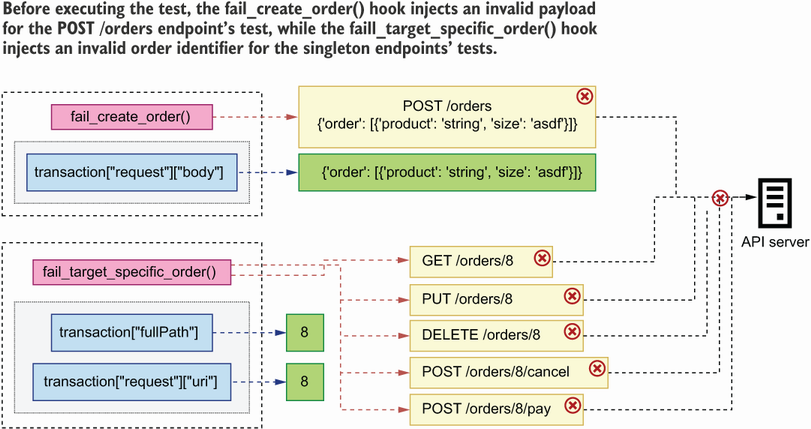


Figure 12.4 The fail\_create\_order() and the fail\_target\_specific\_order() hooks inject invalid payloads and invalid order identifiers to trigger a 422 response from the server.

Using Dredd hooks to save the ID of a created resource

Now that we know what we want to do, let’s write our hooks! First, if you haven’t done it yet, cd into ch12/orders and activate the virtual environment by running pipenv shell. Create a file called orders/hooks.py, where we’ll write our hooks. Although Dredd is an npm package, we can write our hooks in Python by using the dredd-hooks library. In section 12.1, we set up the environments for this chapter, so dredd-hooks has already been installed.

To understand how Dredd hooks work, let’s look at one of them in detail. Listing 12.2 shows the implementation of an after hook for the POST /orders endpoint. This code goes into the orders/hooks.py file. We first declare a variable called response\_ stash, which we’ll use to store data from the POST /orders request. dredd-hooks provides decorator functions, such as dredd\_hooks.before() and dredd\_hooks.after(), that allow us to bind a function to a specific operation. dredd-hooks’ decorators accept an argument, which represents the path to the specific operation that we want to bind the hook to. As you can see in figure 12.5, in Dredd, an operation is defined as a URL endpoint with its response status code and its content-encoding format. In listing 12.2, we bind the save\_created\_order() hook to the 201 response of the POST /orders endpoint.

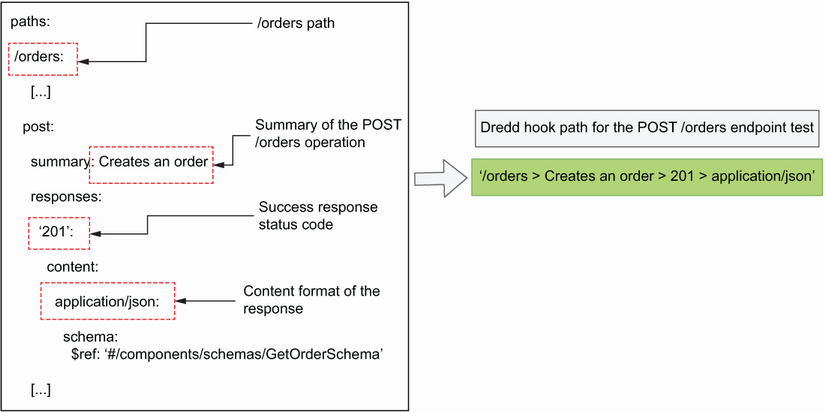


Figure 12.5 To form the path for a specific operation in a Dredd hook, you use the URL path with the operation’s summary, response status code, and content encoding of the response.

**DEFINING OPERATION PATHS IN DREDD HOOKS** When defining the path for an operation using dredd-hooks, you can’t use HTTP methods as part of the operation path; that is, the following syntax won’t work: /orders > post > 201 > application/json. Instead, we use other properties of the POST endpoint, such as summary or operationId, as in the following example: /orders > Creates an order > 201 > application/json.

Dredd hooks take an argument that represents the transaction Dredd performed during the test. The argument comes in the form of a dictionary. In listing 12.2, we name the hook’s argument transaction. Since our goal in the save\_created\_ order() hook is to fetch the ID of the created order, we inspect the payload returned by the POST /orders endpoint, which can be found under transaction['real'] ['body']. Since our API returns JSON payloads, we load its contents using Python’s json library. Once we get hold of the order’s ID, we save it for later use in our global state dictionary, which we named response\_stash.

Listing 12.2 Implementation of an after hook for the POST /orders endpoint

# file: orders/hooks.py

import json

import dredd\_hooks ①

response\_stash = {} ②

@dredd\_hooks.after('/orders > Creates an order > 201 > application/json') ③

def save\_created\_order(transaction):

response\_payload = transaction['real']['body'] ④

order\_id = json.loads(response\_payload)['id'] ⑤

response\_stash['created\_order\_id'] = order\_id ⑥

① We import the dredd\_hooks library.

② We create a global object to store and manage the state of the test suite.

③ We create a hook to be triggered after the POST /orders endpoint test.

④ We access the response payload from the POST /orders endpoint.

⑤ We load the response using Python’s json library and retrieve the order’s ID.

⑥ We store the order ID in our global response\_stash object.

Using hooks to make Dredd use custom URLs

Now that we know how to save the ID of the order created in a POST request, let’s see how we use the ID to form the order’s resource URL. Listing 12.3 shows how we build hooks for the order resource endpoints. The code shown in listing 12.3 goes into the orders/hooks.py file. The code from listing 12.2 is omitted using ellipses, while the new additions are shown in bold.

To specify which URL Dredd should use when testing the /orders/{order\_id} path, we need to modify the transaction payload. In particular, we need to modify the transaction’s fullPath and its request’s uri properties and make sure they point to the right URL. To form the URL, we access the order’s ID from the response\_stash dictionary.

Listing 12.3 Using before hooks to tell Dredd which URL to use

# file: orders/hooks.py

import json

import dredd\_hooks

response\_stash = {}

[...]

**@dredd\_hooks.before(**

**'/orders/{order\_id} > Returns the details of a specific order > 200 > '**

**'application/json'**

**)**  ①

**def before\_get\_order(transaction):**

**transaction["fullPath"] = (**

**"/orders/" + response\_stash["created\_order\_id"]** ②

**)**

**transaction['request']['uri'] = (**

**'/orders/' + response\_stash['created\_order\_id']**

**)**

**@dredd\_hooks.before(**

**'/orders/{order\_id} > Replaces an existing order > 200 > '**

**'application/json'**

**)**

**def before\_put\_order(transaction):**

**transaction[**'**fullPath**'**] = (**

**'/orders/' + response\_stash['created\_order\_id']**

**)**

**transaction['request']['uri'] = (**

**'/orders/' + response\_stash['created\_order\_id']**

**)**

**@dredd\_hooks.before('/orders/{order\_id} > Deletes an existing order > 204')**

**def before\_delete\_order(transaction):**

**transaction['fullPath'] = (**

**'/orders/' + response\_stash['created\_order\_id']**

**)**

**transaction['request']['uri'] = (**

**'/orders/' + response\_stash['created\_order\_id']**

**)**

① We create a hook to be triggered before the GET /orders/{order\_id} endpoint test.

② We change the GET /orders/{order\_id} endpoint test’s URL to include the ID of the order we created earlier.

Using Dredd hooks to create resources before a test

The DELETE /orders/{order\_id} endpoint deletes the order from the database, so we can’t use the same order ID to test the /orders/{order\_id}/pay and /orders/{order\_id}/cancel endpoints. Instead, we’ll use hooks to create new orders before testing those endpoints. Listing 12.4 shows how we accomplish that. The code in listing 12.4 goes into the orders/hooks.py file. The new code is shown in bold, while the code from previous listings is omitted with ellipses.

To create new orders, we’ll call the POST /orders endpoint using the requests library, which makes it easy to make HTTP requests. To launch a POST request, we use requests’ post() function, passing in the target URL for the request and the JSON payload required to create an order. In this case, we hardcode the server base URL to http://127.0.0.1:8000, but you may want to make this value configurable if you want to be able to run the test suite in different environments. Once we’ve created the order, we fetch its ID from the response payload and use the ID to modify the transaction’s fullPath and its request’s uri properties.

Listing 12.4 Using before hooks to create resources before a test

# file: orders/hooks.py

import json

import dredd\_hooks

**import requests**  ①

response\_stash = {}

[...]

**@dredd\_hooks.before(**

**'/orders/{order\_id}/pay > Processes payment for an order > 200 > '**

**'application/json'**

**)**

**def before\_pay\_order(transaction):**

**response = requests.post(**  ②

**"http://127.0.0.1:8000/orders",**

**json={**

**"order": [{"product": "string", "size": "small", "quantity":1}]**

**},**

**)**

**id\_ = response.json()['id']**  ③

**transaction['fullPath'] = '/orders/' + id\_ + '/pay'**  ④

**transaction['request']['uri'] = '/orders/' + id\_ + '/pay'**

**@dredd\_hooks.before(**

**'/orders/{order\_id}/cancel > Cancels an order > 200 > application/json'**

**)**

**def before\_cancel\_order(transaction):**

**response = requests.post(**

**"http://127.0.0.1:8000/orders",**

**json={**

**"order": [{"product": "string", "size": "small", "quantity":1}]**

**},**

**)**

**id\_ = response.json()['id']**

**transaction['fullPath'] = '/orders/' + id\_ + '/cancel'**

**transaction['request']['uri'] = '/orders/' + id\_ + '/cancel'**

① We import the requests library.

② We place a new order.

③ We fetch the newly created order’s ID.

④ We change the POST /orders/{order\_id}/pay endpoint test’s URL by to include the ID of the order we created earlier.

Using hooks to generate 422 responses

Some of the endpoints in the orders API accept request payloads or URL path parameters. If an API client sends an invalid payload or uses an invalid URL path parameter, the API responds with a 422 response. As we saw earlier, Dredd doesn’t know how to generate 422 responses from the server, so we’ll create hooks for that.

As you can see in listing 12.5, we only need two functions:

* fail\_create\_order() intercepts the request for the POST /orders endpoint before it reaches the server, and it modifies its payload with an invalid value for the size property.
* fail\_target\_specific\_order() modifies the order’s URI with an invalid identifier. Since we know that Dredd fires this test using the example ID we provided in the API specification, we simply need to replace that ID with an invalid value. The type of the order\_id path parameter is a UUID, so by replacing it with an integer, the server will respond with a 422 status code.

These hooks are a good opportunity to test how the server behaves with different types of payloads and parameters, and if you need to, you can create specific tests for each endpoint for more comprehensive test coverage.

Listing 12.5 Generating 422 responses with Dredd hooks

# file: orders/hooks.py

**@dredd\_hooks.before('/orders > Creates an order > 422 > application/json')**

**def fail\_create\_order(transaction):**

**transaction["request"]["body"] = json.dumps(**

**{"order": [{"product": "string", "size": "asdf"}]}**

**)**

**@dredd\_hooks.before(**

**"/orders/{order\_id} > Returns the details of a specific order > 422 > "**

**"application/json"**

**)**

**@dredd\_hooks.before(**

**"/orders/{order\_id}/cancel > Cancels an order > 422 > application/json"**

**)**

**@dredd\_hooks.before(**

**"/orders/{order\_id}/pay > Processes payment for an order > 422 > "**

**"application/json"**

**)**

**@dredd\_hooks.before(**

**"/orders/{order\_id} > Replaces an existing order > 422 > "**

**"application/json"**

**)**

**@dredd\_hooks.before(**

**"/orders/{order\_id} > Deletes an existing order > 422 > "**

**"application/json"**

**)**

**def fail\_target\_specific\_order(transaction):**

**transaction["fullPath"] = transaction["fullPath"].replace(**

**"d222e7a3-6afb-463a-9709-38eb70cc670d", "8"**

**)**

**transaction["request"]["uri"] = transaction["request"]["uri"].replace(**

**"d222e7a3-6afb-463a-9709-38eb70cc670d", "8"**

**)**

Running Dredd with custom hooks

Now that we have Dredd hooks to make sure that each URL is correctly formed, we can run the Dredd test suite again. The following command shows how to run Dredd using a hooks file:

$ ./node\_modules/.bin/dredd oas.yaml http://127.0.0.1:8000 --server \

"uvicorn orders.app:app" --hookfiles=./hooks.py --language=python

As you can see, we simply need to pass the path to our hooks file using the --hookfiles flag. We also need to specify the language in which the hooks are written by using the --language flag. If you run the command now, you’ll see now that all tests pass.

**12.2.4 Using Dredd in your API testing strategy**

Dredd is a fantastic tool for testing API implementations, but its test suite is limited. Dredd only tests the happy path of each endpoint. For example, to test the POST /orders endpoint, Dredd sends only a valid payload to the endpoint and expects it to be processed correctly. It doesn’t send malformed payloads, so by using Dredd alone, we don’t know how the server reacts in those situations. This is fine when we’re in the early stage development of our service and we don’t want to be carried away by the API layer.

However, before we release our code, we must ensure it works as expected in all situations, and to run tests that go beyond the happy path, we need to use a different library: schemathesis. We’ll learn about Schemathesis in section 12.4, but before we do that, we need to understand the core approach to testing that Schemathesis uses: property-based testing. That’s the topic of our next section, so move on to learn more about it!

**12.3 Introduction to property-based testing**

This section explains what property-based testing is, how it works, and how it helps us write more exhaustive tests for our APIs. Along the way, you’ll also lean about Python’s excellent property-based testing library, hypothesis. As you’ll see, property-based testing helps us create robust test suites for APIs, allowing us to easily generate hundreds of test cases with multiple combinations of properties and types. This section paves the way for the upcoming sections of this chapter, where we’ll learn about Schemathesis, an API testing framework that uses property-based testing.

**12.3.1 What is property-based testing?**

As you can see in figure 12.6, property-based testing is a testing strategy in which we feed test data to our code and design our tests to make claims about the properties of the result of running our code.[**1**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/12.htm#pgfId-1081073) Typically, a property-based framework generates test cases for us given a set of conditions that we define.

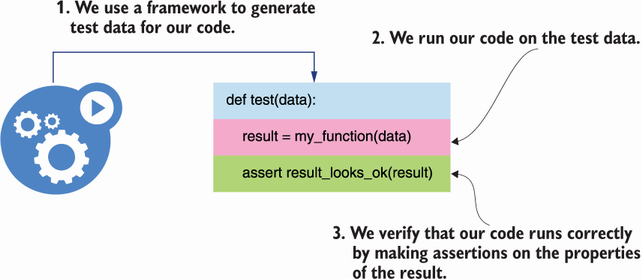


Figure 12.6 In property-based testing, we use a framework to generate test cases for our functions, and we make assertions on the result of running our code on such cases.

**DEFINITION** Property-based testing is an approach to testing in which we make claims about the properties of the return value of our functions or methods. Instead of manually writing lots of different tests with various inputs, we let a framework generate the inputs for us, and we define how we expect our code to handle them. In Python, an excellent library for property-based testing is Hypothesis (<https://github.com/HypothesisWorks/hypothesis>).

**12.3.2 The traditional approach to API testing**

Let’s say we want to test our POST /orders endpoint to ensure it only accepts valid payloads. As you can see from the OpenAPI specification for the orders API under the ch012/orders/oas.yaml file, a valid payload for the POST /orders endpoint contains a key named order, which represents an array of ordered items. Each item has two required keys: product and size.

Listing 12.6 Schema for the POST /orders endpoint’s request payload

# file: orders/oas.yaml

components:

schemas:

OrderItemSchema:

type: object

additionalProperties: false

required:

- product

- size

properties:

product:

type: string

size:

type: string

enum:

- small

- medium

- big

quantity:

type: integer

format: int64

default: 1

minimum: 1

CreateOrderSchema:

type: object

additionalProperties: false

required:

- order

properties:

order:

type: array

minItems: 1

items:

$ref: '#/components/schemas/OrderItemSchema'

In a traditional approach, we’d write various payloads manually, then submit them to the POST /orders endpoint and write the expected result for each payload. Listing 12.7 illustrates how we test the POST /orders endpoint with two different payloads. If you want to try out the code in listing 12.7, create a file called orders/test.py and run the tests with the following command: pytest test.py.

In listing 12.7, we define two test cases: one with an invalid payload missing the required size property of an order item and another with a valid payload. In both cases, we use FastAPI’s test client to send the payloads to our API server, and we test the server’s behavior by checking the status code from the response. We expect the response for an invalid payload to carry the 422 status code (Unprocessable Entity), and the response for the valid payload to carry the 201 status code (Created). FastAPI uses pydantic to validate our payloads, and it automatically generates a 422 response for malformed payloads. Therefore, this test serves to validate that our pydantic models are correctly implemented.

Listing 12.7 Testing the POST /orders endpoint with different payloads

# file: orders/test.py

from fastapi.testclient import TestClient ①

from orders.app import app

test\_client = TestClient(app=app) ②

def test\_create\_order\_fails(): ③

bad\_payload = {

'order': [{'product': 'coffee'}] ④

}

response = test\_client.post('/orders', json=bad\_payload) ⑤

assert response.status\_code == 422 ⑥

def test\_create\_order\_succeeds():

good\_payload = {

'order': [{'product': 'coffee', 'size': 'big'}] ⑦

}

response = test\_client.post('/orders', json=good\_payload)

assert response.status\_code == 201 ⑧

① We import FastAPI’s TestClient class.

② We instantiate the test client.

③ We create a test.

④ We define a bad payload for the POST /orders endpoint.

⑤ We test the payload.

⑥ We confirm that the response status code is 422.

⑦ We define a valid payload for the POST /orders endpoint.

⑧ We confirm that the response status code is 201.

**12.3.3 Property-based testing with Hypothesis**

The testing strategy in listing 12.7, where we write all the test cases manually, is a common approach to API testing. The problem with this approach is that it’s quite limited unless we’re willing to spend many hours writing exhaustive test suites. The test suite in listing 12.7 is far from complete: it’s not testing what happens if the size property contains an invalid value, or if the quantity property is present with a negative value, or if the list of order items is empty.

For a more comprehensive approach to API testing, we want to be able to use a framework that can generate all possible types of payloads and test them against our API server. This is exactly what property-based testing allows us to do. In Python, we can run property-based tests with the help of the excellent hypothesis library.

Hypothesis uses the concept of strategy to generate test data. For example, if we want to generate random integers, we use Hypothesis’s integers() strategy, and if we want to generate text data, we use Hypothesis’s text() strategy. Hypothesis’s strategies expose a method called example() that you can use to get an idea of the values they produce. You can get a feeling of how Hypothesis’s strategies work by playing with them in a Python shell (since Hypothesis produces random values, you’ll see different results in your shell):

>>> from hypothesis import strategies as st

>>> st.integers().example()

0

>>> st.text().example()

'r'

As you can see in figure 12.7, Hypothesis also allows us to combine various strategies using the pipe operator (|). For example, we can define a strategy that produces either integers or text:

>>> strategy = st.integers() | st.text()

>>> strategy.example()

-2781

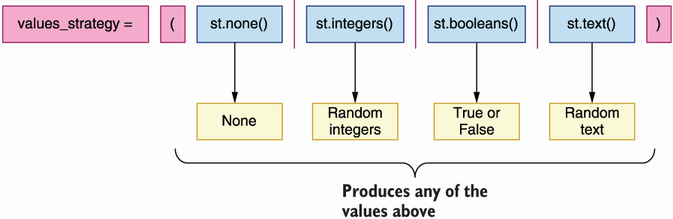


Figure 12.7 We can combine various Hypothesis strategies into one. The resulting strategy will produce a value from any of the combined strategies at random.

To test the POST /orders endpoint with Hypothesis, we want to define a strategy that produces dictionaries with random values. To work with dictionaries, we can use either Hypothesis’s dictionaries() or fixed\_dictionaries() strategies. For example, if we want to generate a dictionary with two keys, such as product and size, where each key can be either an integer or a text, we’d use the following declaration:

>>> strategy = st.fixed\_dictionaries(

{

"product": st.integers() | st.text(),

"size": st.integers() | st.text(),

}

)

>>> strategy.example()

{'product': -7958791642907854994, 'size': 16875}

**12.3.4 Using Hypothesis to test a REST API endpoint**

Let’s put all of this together to create an actual test for the POST /orders endpoint. First, let’s define a strategy for all the values that a property in our payload can take. We’ll keep it simple for illustration purposes and assume properties can only be null, Booleans, text, or integers:

>>> values\_strategy = (

st.none() |

st.booleans() |

st.text() |

st.integers()

)

Now, let’s define a strategy for the schema that represents an order item. To keep it simple, we use a fixed dictionary with valid keys, that is, product, size, and quantity. Since the size property can only take on values from an enumeration whose choices are small, medium, or big, we define a strategy that allows Hypothesis to choose a value either from that enumeration or from the values\_strategy strategy that we defined earlier:

>>> order\_item\_strategy = st.fixed\_dictionaries(

{

"product": values\_strategy,

"size": st.one\_of(st.sampled\_from(("small", "medium", "big")))

| values\_strategy,

"quantity": values\_strategy,

}

)

Finally, as you can see in figure 12.8, we put all of this together in a strategy for the CreateOrderSchema schema. From listing 12.4, we know that CreateOrderSchema requires a property called order, whose value is a list of order items. Using Hypothesis, we can define a strategy that generates payloads to test the CreateOrderSchema schema like this:

>>> strategy = st.fixed\_dictionaries({

'order': st.lists(order\_item\_strategy)

})

>>> strategy.example()

{'order': [{'product': None, 'size': 'small', 'quantity': None}]}

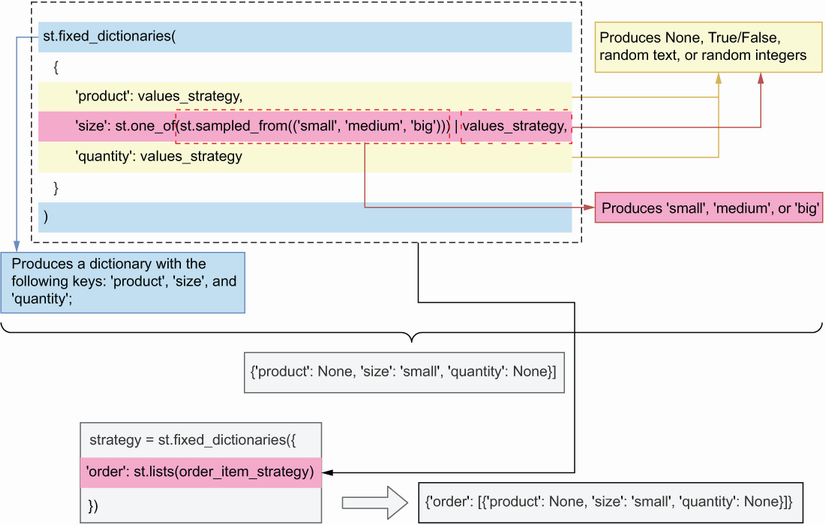


Figure 12.8 By combining Hypothesis’s fixed\_dictionaries() strategy with the lists() strategy and the values\_strategy, we can produce payloads that resemble the CreateOrderSchema schema.

We’re now ready to rewrite our test suite from listing 12.6 into a more generic and comprehensive test for the POST /orders endpoint. Listing 12.7 shows how we inject Hypothesis strategies into a test function. The code in listing 12.7 goes into the orders/test.py file. I’ve omitted the definitions of some variables in listing 12.7, such as values\_strategy and order\_item\_strategy, since we already came across them in the previous examples.

The testing strategy in listing 12.8 uses the jsonschema library to validate the payloads generated by Hypothesis. To validate payloads with the jsonschema library, we first load the OpenAPI specification for the orders API, which lives under ch012/ orders/oas.yaml. We read the file contents using pathlib’s Path().read\_text() method, and we parse them using Python’s yaml library. To check whether a payload is valid, we create a utility function called is\_valid\_payload(), which returns True if the payload is valid and, otherwise, returns False.

We validate the payload using jsonschema’s validate() function, which requires two arguments: the payload that we want to validate and the schema that we want to validate against. Since CreateOrderSchema contains a reference to another schema within the API specification, namely, the OrderItemSchema schema, we also provide a resolver, which jsonschema can use to resolve references to other schemas within the document. jsonschema’s validate() function raises a ValidationError if the payload is invalid, so we call it within a try/except block, and we return True or False depending on the result.

To inject data into our test functions, Hypothesis provides the given() decorator, which takes a Hypothesis strategy as an argument and uses it to feed test cases to our test function. If the payloads are valid, we expect our API to return a response with the 201 status code, while for bad payloads we expect a 422 status code.

Listing 12.8 Using hypothesis to run property-based tests against an API

# file: orders/test.py

from pathlib import Path

import hypothesis.strategies as st

import jsonschema

import yaml

from fastapi.testclient import TestClient

from hypothesis import given, Verbosity, settings

from jsonschema import ValidationError, RefResolver

from orders.app import app

orders\_api\_spec = yaml.full\_load(

(Path(\_\_file\_\_).parent / 'oas.yaml').read\_text() ①

)

create\_order\_schema = ( orders\_api\_spec['components']['schemas']['CreateOrderSchema'] ②

)

def is\_valid\_payload(payload, schema): ③

try:

jsonschema.validate(

payload, schema=schema,

resolver=RefResolver('', orders\_api\_spec) ④

)

except ValidationError:

return False

else:

return True

test\_client = TestClient(app=app) ⑤

values\_strategy = [...]

order\_item\_strategy = [...]

strategy = [...]

@given(strategy) ⑥

def test(payload): ⑦

response = test\_client.post('/orders', json=payload) ⑧

if is\_valid\_payload(payload, create\_order\_schema): ⑨

assert response.status\_code == 201

else:

assert response.status\_code == 422

① We load the API specification.

② Pointer to the CreateOrderSchema schema

③ Helper function to determine whether a payload is valid

④ We validate a payload with jsonschema’s validate() function.

⑤ We instantiate the test client.

⑥ We feed the hypothesis strategies into our test function.

⑦ We capture each test case through the payload argument.

⑧ We send the payload to the POST /orders endpoint.

⑨ We assert the expected status code depending on whether the payload is valid.

As it turns out, Hypothesis is very suitable for generating datasets based on JSON Schema schemas, and there’s already a library that translates schemas into Hypothesis strategies, so you don’t have to do it yourself: hypothesis-jsonschema (<https://github.com/Zac-HD/hypothesis-jsonschema>). I strongly encourage you to look at this library before trying to generate your own Hypothesis strategies for testing web APIs. Now that we understand what property-based testing is and how Hypothesis works, we’re ready to learn about Schemathesis, which is the topic of our next section!

**12.4 Testing REST APIs with Schemathesis**

This section introduces Schemathesis and explains how it works and how we use it to test REST APIs. Schemathesis is an API testing framework that uses property-based testing to validate our APIs. It uses the hypothesis library under the hood, and thanks to its approach, it’s capable of running a more exhaustive test suite than Dredd. Once you’re getting ready to release your APIs to production, I recommend you test them with Schemathesis to make sure you cover all edge cases.

**12.4.1 Running Schemathesis’s default test suite**

In this section, we’ll get familiar with Schemathesis by running its default test suite. Since we already installed our dependencies in section 12.1, all we need to do is cd into the orders folder and activate our environment by running pipenv shell. In contrast with Dredd, Schemathesis requires you to have your API server running before you run your test suite. You can start the server by opening a new terminal window and running the server there or by starting the server and pushing it to the background with the following command:

$ uvicorn orders.app:app &

The & symbol pushes the process to the background. Then you can run Schemathesis with the following command:

$ schemathesis run oas.yaml --base-url=http://localhost:8000 \

--hypothesis-database=none

Hypothesis, the library that Schemathesis uses to generate test cases, creates a folder called .hypothesis/ where it caches some of its tests. In my experience, Hypothesis’s cache sometimes causes misleading results in subsequent test executions, so until this is fixed, my recommendation is to avoid caching the tests. We set the --hypothesis-database flag to none so that Schemathesis doesn’t cache test cases.

After executing the command, you’ll see that Schemathesis runs around 700 tests against the API, testing all possible combinations of parameters, types, and formats. All tests should pass correctly. Once Schemathesis has finished, you can bring the Uvicorn process to the foreground by running the fg command, and stop it if you wish. (I’m sure know you know, but remember that to stop a process you use the Ctrl-C key combination).

**12.4.2 Using links to enhance Schemathesis’ test suite**

The test suite we just ran with Schemathesis has one major limitation: it doesn’t test whether the POST /orders endpoint is creating orders correctly nor if we can perform the expected operations, such as payments and cancellations, on an order. It’s simply launching independent and unrelated requests to each of the endpoints in the orders API. To check whether we are creating resources correctly, we need to enhance our API specification with links. As you can see in figure 12.9, in the OpenAPI standard, links are declarations that allow us to describe the relationships between different endpoints.[**2**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/12.htm#pgfId-1081343)

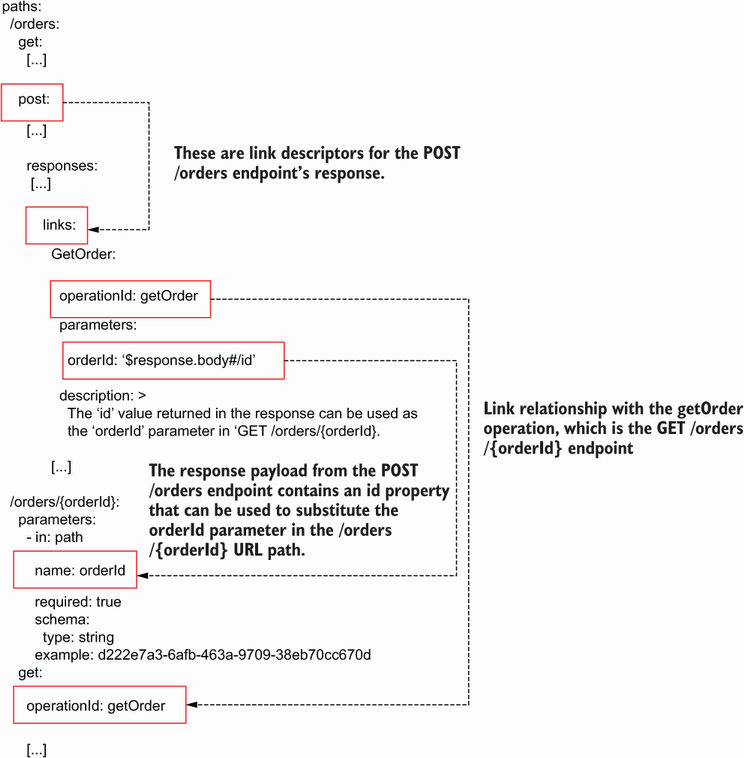


Figure 12.9 In OpenAPI, we can use links to describe the relationships between endpoints. For example, the POST /orders response contains an id property that we can use to replace the order\_id parameter in the /orders/{order\_id} URL.

For example, using links, we can specify that the POST /orders endpoint returns a payload with an ID, and that we can use that ID to form the resource URL of the order just created under the GET /orders/{order\_id} endpoint. We use operation IDs to describe the relationships between our endpoints. As we learned in chapter 5 (section 5.3), operation IDs are unique identifiers for each endpoint in the API. Listing 12.9 shows how we enhance the orders API with a link that describes the relationship between the POST /orders endpoint and the GET /orders/{order\_id} endpoint. For the full list of links, please see the ch12/orders/oas\_with\_links.yaml file in the GitHub repository for this book. Ellipses are used to hide parts of the code that are not relevant to the example, and newly added code is in bold.

In listing 12.9, we name the link between the POST /orders and the GET /order/{order\_id} endpoints GetOrder. GetOrder’s operationId property identifies the endpoint this link refers to (getOrder). The GET /order/{order\_id} endpoint has an URL parameter named order\_id, and GetOrder’s parameters property tells us that the response body from the POST /orders endpoint contains an id property, which we can use to replace order\_id in the GET /order/{order\_id} endpoint.

Listing 12.9 Adding links to create relationships between endpoints in OpenAPI

# file: orders/oas.yaml

paths:

/orders:

get:

[...]

post:

operationId: createOrder

summary: Creates an order

requestBody:

required: true

content:

application/json:

schema:

$ref: '#/components/schemas/CreateOrderSchema'

responses:

'201':

description: A JSON representation of the created order

content:

application/json:

schema:

$ref: '#/components/schemas/GetOrderSchema'

**links:**  ①

**GetOrder:**

**operationId: getOrder**  ②

**parameters:**

**order\_id: '$response.body#/id'**  ③

**description: >**  ④

**The `id` value returned in the response can be used as**

**the `order\_id` parameter in `GET /orders/{order\_id}`**

[...]

/orders/{order\_id}:

[...]

get:

operationId: getOrder

[...]

① We add links to the POST /orders endpoint.

② We define a link with the GET /orders/{order\_id} endpoint.

③ The order\_id URL parameter in the getOrder endpoint can be replaced with the response payload’s id property.

④ We explain how this link works.

We can now run Schemathesis and take advantage of our links by running the following command:

$ schemathesis run oas\_with\_link.yaml --base-url=http://localhost:8000 \

--stateful=links

The --stateful=links flag instructs Schemathesis to look for links in our documentation and use them to run tests on the resources created through the POST /orders endpoint. If you run Schemathesis now, you’ll see that it runs well over a thousand tests against the API. Since Schemathesis generates random tests, the exact number of tests cases may differ from time to time. Listing 12.10 shows the output of the Schemathesis test suite after running it with the --stateful parameter set to links. The listing omits the first few lines of the test suite as they contain only system-specific metadata. Notice that some of the tests appear nested within the POST /orders endpoint (the lines starting with the -> symbol). Nested tests are tests that leverage links from our API documentation. If the tests on the POST /orders endpoint’s links pass, we can rest assured that our resources are being created correctly.

Listing 12.10 Output of a Schemathesis test suite

[...]

Base URL: http://localhost:8000 ①

Specification version: Open API 3.0.3 ②

Workers: 1 ③

Collected API operations: 7 ④

GET /orders . [ 14%] ⑤

POST /orders . [ 28%]

-> GET /orders/{order\_id} . [ 37%] ⑥

-> PUT /orders/{order\_id} . [ 44%]

-> DELETE /orders/{order\_id} . [ 50%]

-> POST /orders/{order\_id}/cancel . [ 54%]

-> POST /orders/{order\_id}/pay . [ 58%]

GET /orders/{order\_id} . [ 66%]

PUT /orders/{order\_id} . [ 75%]

DELETE /orders/{order\_id} . [ 83%]

POST /orders/{order\_id}/pay . [ 91%]

POST /orders/{order\_id}/cancel . [100%]

================================ SUMMARY ==================================

Performed checks:

not\_a\_server\_error 1200 / 1200 passed PASSED ⑦

========================== 12 passed in 57.57s ============================

① The server’s base URL

② The version of OpenAPI used by our server

③ Number of processes running the test suite in parallel

④ Number of operations defined in the API specification

⑤ Test for the GET /orders endpoint

⑥ Test for the GET /orders/{order\_id} endpoint linked to the POST /orders endpoint test

⑦ The test suite runs 1,200 tests, and all of them pass.

The output from the previous test says that our API passed all checks in the not\_a\_ server\_error category. By default, Schemathesis only checks that the API doesn’t raise server errors, but it can be configured to also verify that our API uses the right status codes, content types, headers, and schemas as documented in the API specification. To apply all these checks, we use the --checks flag and we set it to all:

$ schemathesis run oas\_with\_link.yaml --base-url=http://localhost:8000 \

--hypothesis-database=none --stateful=links --checks=all

As you can see, this time Schemathesis runs over a thousand test cases per check:

================================ SUMMARY ==================================

Performed checks:

not\_a\_server\_error 1200 / 1200 passed PASSED

status\_code\_conformance 1200 / 1200 passed PASSED

content\_type\_conformance 1200 / 1200 passed PASSED

response\_headers\_conformance 1200 / 1200 passed PASSED

response\_schema\_conformance 1200 / 1200 passed PASSED

========================== 12 passed in 70.54s ============================

In some cases, Schemathesis may complain that it takes too long to generate test cases. You can suppress that warning by using the --hypothesis-suppress-health-check=too\_slow flag. By running the whole set of Schemathesis checks against your API, you can be certain that it works as expected and complies with the API specification. If you’d like to extend the tests with additional custom payloads or scenarios, you can do that as well. Since schemathesis is a Python library, it’s very easy to add additional custom tests. Check the documentation for examples on how to do that (<http://mng.bz/69Q5>).

This concludes our journey through testing REST APIs. It’s now time to move on to the world of GraphQL API testing, which is the topic of the next section!

**12.5 Testing GraphQL APIs**

This section explains how we test and validate GraphQL APIs so that we can ensure they work as expected before we release them to production. We’ll use the products API, which we implemented in chapter 10, as a guiding example. To work through the examples in this section, cd into ch12/products and activate the environment by running pipenv shell.

In sections 12.2 and 12.4, we learned about Dredd and Schemathesis, which automatically generate tests for REST APIs based on the API specification. For GraphQL, there’s less support for automatic test generation. In particular, Dredd doesn’t support GraphQL APIs, while Schemathesis only provides partial support. However, this is an active area of development, so expect to see increasing support for automatic GraphQL testing in the future.

**12.5.1 Testing GraphQL APIs with Schemathesis**

This section explains how we use Schemathesis to test and validate a GraphQL API. As we explained in section 12.4, Schemathesis is an API testing framework that uses an approach known as property-based testing to validate our APIs. Schemathesis can be used to test both REST and GraphQL APIs. In both cases, as you can see in figure 12.10, Schemathesis looks at the API specification to learn about its endpoints and schemas, and to decide which tests to run.

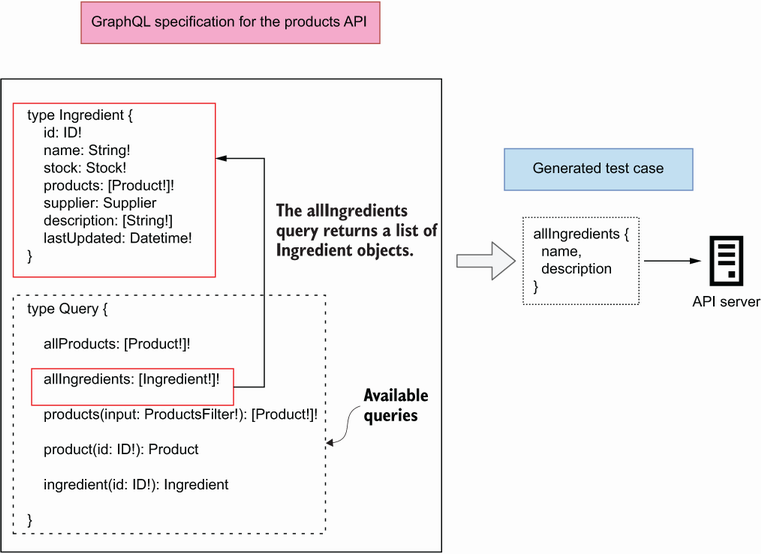


Figure 12.10 Schemathesis parses a GraphQL API specification in search of available operations and generates query documents with both valid and invalid parameters and selection sets to test the server’s response.

To generate tests for a GraphQL API, Schemathesis uses hypothesis-graphql (<http://mng.bz/o5Pj>), a library that generates Hypothesis strategies from a GraphQL schema. Before we run our test, we need to start the GraphQL API server. You can do that in a different terminal window, or you can run the process in the background with the following command:

$ uvicorn server:server &

The & symbol pushes the Uvicorn process to the background. To test a GraphQL API with Schemathesis, we simply need to give it the URL where our API specification is hosted. In our case, the GraphQL API is hosted under the following URL: http://127.0.0.1:8000/graphql. Armed with this information, we can now run our tests:

$ schemathesis run --hypothesis-deadline=None http://127.0.0.1:8000/graphql

The --hypothesis-deadline=None flag instructs Schemathesis to avoid timing the requests. This is useful in cases where our queries may be slow, which sometimes happens with GraphQL APIs. The following shows the output of the test suite, omitting the first few lines that contain platform-specific metadata. As you can see, Schemathesis tests all of the queries and mutations exposed by the products API, generating a very solid battery of tests: 1,100 test cases!

Listing 12.11 Output of a Schemathesis test suite for a GraphQL API

[...]

Schema location: http://127.0.0.1:8000/graphql

Base URL: http://127.0.0.1:8000/graphql

Specification version: GraphQL

Workers: 1

Collected API operations: 11

Query.allProducts . [ 9%]

Query.allIngredients . [ 18%]

Query.products . [ 27%]

Query.product . [ 36%]

Query.ingredient . [ 45%]

Mutation.addSupplier . [ 54%]

Mutation.addIngredient . [ 63%]

Mutation.addProduct . [ 72%]

Mutation.updateProduct . [ 81%]

Mutation.deleteProduct . [ 90%]

Mutation.updateStock . [100%]

================================ SUMMARY ==================================

Performed checks:

not\_a\_server\_error. 1100 / 1100 passed PASSED

========================== 11 passed in 36.82s ============================

After running the Schemathesis test suite against the products API, we can be certain that our queries and mutations work as expected. You can further customize your tests to make sure the application works correctly under certain conditions. To learn how to add custom tests cases, check out Schemathesis’ excellent documentation (<https://schemathesis.readthedocs.io/en/stable/>).

**12.6 Designing your API testing strategy**

You’ve learned a lot in this chapter. You’ve learned to use frameworks such as Dredd and Schemathesis, which run automated test suites against your APIs based on the API documentation. You’ve also learned about property-based testing and how to use Hypothesis to automatically generate test cases to test your REST and GraphQL APIs.

As we saw in section 12.2, Dredd runs a simple test suite against your APIs. Dredd only tests the happy path: it makes sure your API accepts the expected payloads and responds with the expected payloads. It doesn’t test what happens when the wrong payloads are sent to your server.

Dredd’s testing strategy is useful in the early development stage of your API, when you want to be able to focus on the overall functionality of your application rather than get bogged down with specific corner cases of your API integration. However, before you release your APIs to production, you want to make sure your APIs are tested with Schemathesis. Schemathesis runs a more comprehensive test suite, which ensures that your API works exactly as expected.

I recommend you run Dredd and Schemathesis locally during development, and also in your continuous integration (CI) server before releasing your code. For an example of how you can incorporate Dredd and Schemathesis into your CI server, check out my talk, “API Development Workflows for Successful Integrations,” at Manning’s API Conference (August 3 2021, <https://youtu.be/SUKqmEX_uwg>).

Some of the technologies and skills that you’ve learned in this chapter are still very new and experimental, so you’ve got an edge in your team and in the job market. Use your new powers wisely!

**Summary**

* Dredd and Schemathesis are API testing tools that automatically generate validation tests for APIs from the documentation. This helps you to avoid the effort of writing tests manually and to focus on building your APIs and services.
* Dredd is a REST API testing framework. It runs a basic test suite against your API without covering edge cases, and therefore it’s convenient in the early stages of your API cycle.
* You can customize Dredd’s behavior by adding Dredd hooks to your tests. Although Dredd is an npm package, you can write your hooks in Python. Dredd hooks are useful for saving information from one test for reuse in another test, and for creating or deleting resources before and after each test.
* Schemathesis is a more generic API test framework that runs an exhaustive test suite against your APIs. Before releasing your APIs to production, you want to make sure you’ve tested them with Schemathesis. You can use Schemathesis to test both REST and GraphQL APIs.
* To test that your POST endpoints are creating resources correctly, you can enrich your OpenAPI specification with links and instruct Schemathesis to use them in its test suite. Links are properties that describe the relationship between different operations in an OpenAPI specification.
* Property-based testing is an approach in which you let a framework generate random test cases, and you validate the behavior of your code by making assertions about the properties of the test result. This approach saves you the time of having to write test cases manually. In Python, you can run property-based tests with the excellent hypothesis library.