Extracted from:

Testing Elixir

Effective and Robust Testing for Elixir and its Ecosystem

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Testing Elixir

Effective and Robust Testing for Elixir and its Ecosystem

Andrea Leopardi and Jeffrey Matthias

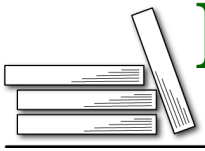
edited by Jacquelyn Carter

Testing Elixir

Effective and Robust Testing for Elixir and its Ecosystem

Andrea Leopardi
Jeffrey Matthias

The Pragmatic Bookshelf
Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Series Editor: Bruce A. Tate
Development Editor: Jacquelyn Carter
Copy Editor: Molly McBeath
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-782-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2021

Sending Real Requests

The simplest approach is to send the real request to the weather API in the tests and assert on the response sent back by the API. The advantage of this approach is that we test the real API directly, so if the API changes or if the code that interacts with it changes, the tests will possibly fail. However, there are some important disadvantages. First of all, our tests now depend on the availability of a third-party system, which we don't control. If the weather API goes down or we don't have access to the Internet, our tests will fail even if the code is correct, which means that our tests are brittle and not reproducible. Another disadvantage is that the API could change some returned data that the test relies on without breaking the contract. In that case, the test might start failing without signaling a real issue. The other main disadvantage of this approach is that making real HTTP requests can cause problems in many use cases. For example, the weather API we're using is rate-limited (as most APIs are in some way), which means that our tests could affect the rate limiting of the API without providing a service to the users.

The disadvantages of the real requests approach can be mitigated in some cases. For example, some third-party APIs provide a "staging" or "sandbox" API with the same interface as the real API but with different behaviour. For example, with a weather API like the one we're using, the sandbox API could always return the same weather data without actually talking to any forecasting service. This would significantly reduce the load on the weather API itself, allowing its developers to possibly lift the rate limiting in this environment. However, many third-party APIs don't provide anything like this, so we have to come up with ways to avoid making real requests in tests but still test the code that makes those requests. Let's see the two most common approaches next.

Building an Ad-hoc HTTP Server

The first approach we're going to examine is building an ad-hoc HTTP server we can send requests to that runs alongside our test suite. We'll have control over the server itself, so we'll be able to control its behaviour and send as many requests to it as we want during testing.

Let's go back to the weather API use case. The weather API exposes a `GET /data/2.5/forecast` endpoint that we hit from our application. This endpoint accepts two parameters in the query string: `q`, which is the *query*, and `APPID`, which identifies the credentials of our application. The endpoint returns a `200 OK` HTTP response with a JSON body containing information about the weather forecast. Let's build an HTTP server that mimics this API.

The Elixir ecosystem has established libraries to build HTTP servers. For now, we're going to use Plug.[2] Plug is a library that provides a common interface over different Erlang and Elixir web servers. A commonly used web server is Cowboy.[3] The first thing we'll need is to add Plug and Cowboy to our dependencies, which can be done by adding the :plug_cowboy dependency. We'll only add this dependency in the :test environment so that it won't be shipped with our application (assuming our application doesn't use Plug and Cowboy itself):

integration_tests/soggy_waffle_actual_integrations/mix_with_plug_cowboy.exs

```
defp deps do
  [
    # «other dependencies»
    {:plug_cowboy, ">= 0.0.0", only: :test}
  ]
end
```

Now let's define a Plug that exposes the endpoint. We'll use Plug.Router, which provides an intuitive DSL for writing simple HTTP endpoints:

integration_tests/soggy_waffle_actual_integrations/test/support/weather_api_test_router.exs

```
defmodule SoggyWaffle.WeatherAPITestRouter do
  use Plug.Router

  # We need to manually import the assertions since we're not
  # inside an ExUnit test case.
  import ExUnit.Assertions

  plug :match
  plug :dispatch
  plug :fetch_query_params

  get "/data/2.5/forecast" do
    params = conn.query_params

    assert is_binary(params["q"])
    assert is_binary(params["APPID"])

    forecast_data = %{
      "list" => [
        %{
          "dt" => DateTime.to_unix(DateTime.utc_now()),
          "weather" => [%{"id" => _thunderstorm = 231}]
        }
      ]
    }

    conn
    |> put_resp_content_type("application/json")
    |> send_resp(200, Jason.encode!(forecast_data))
```

2. https://github.com/elixir-plug/plug
3. https://github.com/ninenines/cowboy

```
  end
end
```

We have a server that exposes an endpoint that behaves exactly like the weather one but performs some assertions on the incoming request. We need to start the server. Do that in the setup callback of the SoggyWaffle.WeatherAPI test case:

integration_tests/soggy_waffle_actual_integrati ... st/soggy_waffle/weather_api_test_plug_cowboy.exs

```
setup do
  options = [
    scheme: :http,
    plug: SoggyWaffle.WeatherAPITestRouter,
    options: [port: 4040]
  ]

  start_supervised!({Plug.Cowboy, options})
  :ok
end
```

We started an HTTP server on port 4040 that we can use throughout the tests by hitting http://localhost:4040/data/2.5/forecast. However, the real weather API URL (https://api.openweathermap.org/data/2.5/forecast) is hard-coded in the SoggyWaffle.WeatherAPI module. We need to make that configurable. We can use the same approach we used when passing a module as an optional argument when we were dealing with doubles. Let's change SoggyWaffle.WeatherAPI:

integration_tests/soggy_waffle_actual_integrations/lib/soggy_waffle/weather_api.ex

```
defmodule SoggyWaffle.WeatherAPI do
  @default_base_url "https://api.openweathermap.org"

  @spec get_forecast(String.t(), String.t()) ::
          {:ok, map()} | {:error, reason :: term()}
➤ def get_forecast(city, base_url \\ @default_base_url)
➤     when is_binary(city) do
    app_id = SoggyWaffle.api_key()
    query_params = URI.encode_query(%{"q" => city, "APPID" => app_id})
➤   url = base_url <> "/data/2.5/forecast?" <> query_params

    case HTTPoison.get(url) do
      {:ok, %HTTPoison.Response{status_code: 200} = response} ->
        {:ok, Jason.decode!(response.body)}

      {:ok, %HTTPoison.Response{status_code: status_code}} ->
        {:error, {:status, status_code}}

      {:error, reason} ->
        {:error, reason}
    end
  end
end
```

Now we can add a test for SoggyWaffle.WeatherAPI that hits the ad-hoc test server:

integration_tests/soggy_waffle_actual_integrati … st/soggy_waffle/weather_api_test_plug_cowboy.exs

```
test "get_forecast/1 hits GET /data/2.5/forecast" do
  query = "losangeles"
  app_id = "MY_APP_ID"
  test_server_url = "http://localhost:4040"

  assert {:ok, body} =
           SoggyWaffle.WeatherAPI.get_forecast(
             "Los Angeles",
             test_server_url
           )

  assert %{"list" => [weather | _]} = body
  assert %{"dt" => _, "weather" => _} = weather
  # «potentially more assertions on the weather»
end
```

This test will hit the test server every time it's run and assert that the SoggyWaffle.WeatherAPI.get_forecast/1 function hits the correct endpoint. Writing our own server from scratch works fine, but there's room for improvement. For example, in our test server we're only asserting that the "q" and "APPID" parameters are strings, but we're not checking that they're the same strings as specified in the test. To do that, we would have to hard-code those strings in the test server code, which in turn means that we'd have to build new test servers to test different scenarios. There's a tool called Bypass that helps in this situation.

Bypass is a library that lets you define Plug-based servers on the fly with an API similar to the one provided by Mox that we saw in the previous sections. Let's see how we can use it to improve our tests.[4] First of all, add the library to your dependencies in place of :plug_cowboy:

4. https://github.com/PSPDFKit-labs/bypass

integration_tests/soggy_waffle_actual_integrations/mix_with_bypass.exs

```
defp deps do
  [
    # «other dependencies»
    {:bypass, ">= 0.0.0", only: :test}
  ]
end
```

Bypass provides Bypass.expect_once/4 to set up an expectation for a request. To use this function, we need to open a Bypass connection in our tests.

Do that in the setup callback:

integration_tests/soggy_waffle_actual_integrations/test/soggy_waffle/weather_api_test_bypass.exs

```
setup do
  bypass = Bypass.open()
  {:ok, bypass: bypass}
end
```

We return a bypass data structure from the test that will contain information like the port the server was started on. We'll pass this data structure around in tests through the test context, and we'll then pass it into the functions we invoke on the Bypass module so that they know how to interact with the test.

Now we can rewrite the test for get_forecast/1 using a Bypass expectation:

integration_tests/soggy_waffle_actual_integrations/test/soggy_waffle/weather_api_test_bypass.exs

```
test "get_forecast/1 hits GET /data/2.5/forecast", %{bypass: bypass} do
  query = "losangeles"
  app_id = "MY_APP_ID"
  test_server_url = "http://localhost:4040"

  forecast_data = %{
    "list" => [
      %{
        "dt" => DateTime.to_unix(DateTime.utc_now()) + _seconds = 60,
        "weather" => [%{"id" => _thunderstorm = 231}]
      }
    ]
  }

  Bypass.expect_once(bypass, "GET", "/data/2.5/forecast", fn conn ->
    conn = Plug.Conn.fetch_query_params(conn)

    assert conn.query_params["q"] == query
    assert conn.query_params["APPID"] == app_id

    conn
    |> Plug.Conn.put_resp_content_type("application/json")
    |> Plug.Conn.resp(200, Jason.encode!(forecast_data))
  end)
```

```
  assert {:ok, body} =
           SoggyWaffle.WeatherAPI.get_forecast(
             "Los Angeles",
             test_server_url
           )

  assert body == forecast_data
end
```

Bypass.expect_once/4 expects the specified request to be issued exactly once. The function passed to it takes a conn data structure (a Plug.Conn struct) that we can use to make assertions and to send a response. As you can see, this API is similar to what Mox provides and allows us to have fine-grained control over the test server and set different expectations in each test.

This "real requests" approach has the advantage of letting us send as many real HTTP requests as we want during testing so that we can exercise the code that interfaces with the real HTTP API as well as the HTTP client we're using. However, this approach has a disadvantage as well. When building the test server and setting request expectations, we're effectively copying what the third-party API does, and by doing so we're tying ourselves to a specific behaviour of that API. If the weather API were to change and we were only relying on test-server-based tests, we wouldn't notice the change when running the test suite. This is important to keep in mind, as there's no clear and straightforward solution for this problem. The only way around it is to periodically check that the weather API still behaves in the same way as the test server. We can do that either manually or by running the code against the real weather API once in a while.

In the next section, we'll see an alternative approach to the same problem that compromises on some things for the sake of making it easier to keep the tests up to date.

Recording Requests with Cassettes

So far, we've explored two alternatives for testing the interaction with a third-party API: issuing requests to the real API or building a test server to mimic the third-party API during testing. In this section we'll explore one last approach, which consists of recording and replaying requests through a library called ExVCR.[5]

The idea behind ExVCR is to issue a request to the real third-party API the first time and record the response into a file called a *cassette*. Then, when we

5. https://github.com/parroty/exvcr

need to make that same request to the third-party API, ExVCR will *replay* that request and return the response from the cassette without making any real HTTP calls. By now you probably get why it's called ExVCR: cassettes, recording, replaying.... It makes sense.

The way ExVCR works is by creating implicit mocks of widely used Erlang and Elixir HTTP clients, such as the built-in httpc or hackney.[6] [7] These mocks intercept requests and record them if there's no cassette for them, or replay the requests from the respective cassette. This approach is limiting in cases where you don't use one of the HTTP clients supported by ExVCR, since ExVCR won't work. However, many applications do use clients supported by ExVCR so it's still worth exploring.

Let's see how to change the SoggyWaffle.WeatherAPI test to make use of ExVCR. Start by adding :ex_vcr as a dependency:

integration_tests/soggy_waffle_actual_integrations/mix_with_ex_vcr.exs

```
defp deps do
  [
    # «other dependencies»
➤   {:ex_vcr, ">= 0.0.0", only: :test}
  ]
end
```

The get_forecast/1 function uses HTTPoison as its HTTP client and HTTPoison uses :hackney under the hood, so ExVCR will work. Now we need to call use ExVCR.Mock to make the ExVCR DSL available in our tests and we'll have to use the ExVCR.Adapter.Hackney adapter.

integration_tests/soggy_waffle_actual_integrations/test/soggy_waffle/weather_api_test_ex_vcr.exs

```
use ExVCR.Mock, adapter: ExVCR.Adapter.Hackney
```

ExVCR provides a use_cassette/2 macro that takes a cassette name and a block of code. Requests executed in the block of code are recorded to and replayed from the specified cassette. Let's rewrite the get_forecast/1 test to use use_cassette/2.

integration_tests/soggy_waffle_actual_integrations/test/soggy_waffle/weather_api_test_ex_vcr.exs

```
test "get_forecast/1 hits GET /data/2.5/forecast" do
  query = "losangeles"
  app_id = "MY_APP_ID"

  use_cassette "weather_api_successful_request" do
    assert {:ok, body} =
             SoggyWaffle.WeatherAPI.get_forecast("Los Angeles")
  end
```

6. http://erlang.org/doc/man/httpc.html
7. https://github.com/benoitc/hackney

```
  assert %{"list" => [weather | _]} = body
  assert %{"dt" => _, "weather" => _} = weather
  # «potentially more assertions on the weather»
end
```

The first time this test is run, the request is issued against the real weather API. After the weather API returns a response, that response is recorded into a cassette called weather_api_successful_request. We use a descriptive and unique name so that it won't conflict with other cassettes. When the same test is run again, no HTTP requests are made and the response recorded into the cassette is returned to the HTTP client.

This approach differs from a test server because it focuses less on asserting that the request is made correctly. The main goal of a cassette is to behave exactly like the real third-party service without having to write code to emulate that third-party service. The workflow is, in fact, simpler than the test server: we just wrap our code with use_cassette/2 and go on about our day. However, cassettes present a similar problem to the test server, which is that they can get out of sync with the actual API. The solution for cassettes is somewhat simpler though, since we only have to delete the stale cassette and rerun our tests in order to re-create an up-to-date cassette.

To push the idea of keeping cassettes up to date further, we can always force real requests to be made when running tests in a continuous integration (CI) server. This way, we'll avoid making real HTTP requests when developing on our local machine, but the CI server (which usually runs much less frequently) will make sure that the cassettes haven't gotten out-of-date. This approach heavily depends on what making a request to the real API implies. In the weather API example, making real requests to /data/2.5/forecast is feasible: if it's only done in CI then it's unlikely that we'll negatively affect our rate limiting. In other cases, making requests might cost money or break things, so making real requests on every CI run might not be ideal. Furthermore, we usually want CI to be reproducible and consistent between runs, and depending on the availability of an external API might not be feasible.

Our favorite use case for cassettes is an external service that allows you to set up and tear down resources through its API. For example, the weather API could expose endpoints to register and delete named queries. Now, if we wanted to test that we can query the forecast through a named query, we could create the named query, test the appropriate functions, and delete the named query all in the same test. In this use case, the cassette merely becomes a "cache" of HTTP requests. Even when not using cassettes (such as in CI),

the test would create and destroy all necessary resources to run the test, leaving the external service's state unchanged.

One important practice to keep in mind when working with ExVCR is to *never reuse cassettes*. If you have two tests that make the same request, you might be tempted to use the same cassette, but there's a good chance that at some point the request made in one of the two tests will change slightly and then things won't work anymore. If you use a different cassette in each test, you're guaranteed to not mess things up in this regard.

Let's recap what we discussed in this section and see which approach is best for different use cases.