Each type of test serves a different purpose, and can catch different types of bugs, so you'll likely want to use a mix of all three types. The purpose of unit tests is to have tests that run quickly so that you can get fast feedback on your changes and validate a variety of different permutations to build up confidence that the basic building blocks of your code (the individual units) work as expected. But just because individual units work correctly in isolation doesn't mean that they will work correctly when combined, so you need integration tests to ensure the basic building blocks fit together correctly. And just because different parts of your system work correctly doesn't mean they will work correctly when deployed in the real world, so you need end-to-end tests to validate that your code behaves as expected in conditions similar to production.

Let's now go through how to write each type of test for Terraform code.

Unit Tests

To understand how to write unit tests for Terraform code, it's helpful to first look at what it takes to write unit tests for a general-purpose programming language such as Ruby. Take a look again at the Ruby web server code:

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    case request.path
    when "/"
      response.status = 200
      response['Content-Type'] = 'text/plain'
      response.body = 'Hello, World'
    when "/api"
      response.status = 201
      response['Content-Type'] = 'application/json'
      response.body = '{"foo":"bar"}'
    else
      response.status = 404
      response['Content-Type'] = 'text/plain'
      response.body = 'Not Found'
    end
  end
end
```

Writing a unit test that calls the `do_GET` method directly is tricky, as you'd have to either instantiate real `WebServer`, `request`, and `response` objects, or create test doubles of them, both of which require a fair bit of work. When you find it difficult to write unit tests, that's often a code smell and indicates that the code needs to be refactored. One way to refactor this Ruby code to make unit testing easier is to extract the "handlers"—that is, the code that handles the `/`, `/api`, and not found paths—into its own `Handlers` class:

```
class Handlers
  def handle(path)
    case path
    when "/"
      [200, 'text/plain', 'Hello, World']
    when "/api"
      [201, 'application/json', '{"foo":"bar"}']
    else
      [404, 'text/plain', 'Not Found']
    end
  end
end
```

There are two key properties to notice about this new `Handlers` class:

Simple values as inputs

> The `Handlers` class does not depend on `HTTPServer`, `HTTPRequest`, or `HTTPResponse`. Instead, all of its inputs are simple values, such as the `path` of the URL, which is a string.

Simple values as outputs

> Instead of setting values on a mutable `HTTPResponse` object (a side effect), the methods in the `Handlers` class return the HTTP response as a simple value (an array that contains the HTTP status code, content type, and body).

Code that takes in simple values as inputs and returns simple values as outputs is typically easier to understand, update, and test. Let's first update

the `WebServer` class to use the new `Handlers` class to respond to requests:

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    handlers = Handlers.new
    status_code, content_type, body =
handlers.handle(request.path)

    response.status = status_code
    response['Content-Type'] = content_type
    response.body = body
  end
end
```

This code calls the `handle` method of the `Handlers` class and sends back the status code, content type, and body returned by that method as an HTTP response. As you can see, using the `Handlers` class is clean and simple. This same property will make testing easy, too. Here are three unit tests that check each endpoint in the `Handlers` class:

```
class TestWebServer < Test::Unit::TestCase
  def initialize(test_method_name)
    super(test_method_name)
    @handlers = Handlers.new
  end

  def test_unit_hello
    status_code, content_type, body = @handlers.handle("/")
    assert_equal(200, status_code)
    assert_equal('text/plain', content_type)
    assert_equal('Hello, World', body)
  end

  def test_unit_api
    status_code, content_type, body = @handlers.handle("/api")
    assert_equal(201, status_code)
    assert_equal('application/json', content_type)
    assert_equal('{"foo":"bar"}', body)
  end

  def test_unit_404
    status_code, content_type, body = @handlers.handle("/invalid-
```