expected. This is about as fast of a feedback loop as you can get with infrastructure in AWS, and it should give you a lot of confidence that your code works as expected.

Dependency injection

Let's now see what it would take to add a unit test for some slightly more complicated code. Going back to the Ruby web server example once more, consider what would happen if you needed to add a new `/web-service` endpoint that made HTTP calls to an external dependency:

```
class Handlers
  def handle(path)
    case path
    when "/"
      [200, 'text/plain', 'Hello, World']
    when "/api"
      [201, 'application/json', '{"foo":"bar"}']
    when "/web-service"
      # New endpoint that calls a web service
      uri = URI("http://www.example.org")
      response = Net::HTTP.get_response(uri)
      [response.code.to_i, response['Content-Type'],
response.body]
    else
      [404, 'text/plain', 'Not Found']
    end
  end
end
```

The updated `Handlers` class now handles the `/web-service` URL by making an HTTP GET to `example.org` and proxying the response. When you `curl` this endpoint, you get the following:

```
$ curl localhost:8000/web-service

<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    <-- (...) -->
</head>
```

```
<body>
<div>
    <h1>Example Domain</h1>
    <p>
      This domain is established to be used for illustrative
      examples in documents. You may use this domain in
      examples without prior coordination or asking for
permission.
    </p>
    <!-- (...) -->
</div>
</body>
</html>
```

How would you add a unit test for this new method? If you tried to test the code as is, your unit tests would be subject to the behavior of an external dependency (in this case, `example.org`). This has a number of downsides:

- If that dependency has an outage, your tests will fail, even though there's nothing wrong with your code.
- If that dependency changed its behavior from time to time (e.g., returned a different response body), your tests would fail from time to time, and you'd need to constantly keep updating the test code, even though there's nothing wrong with the implementation.
- If that dependency were slow, your tests would be slow, which negates one of the main benefits of unit tests, the fast feedback loop.
- If you wanted to test that your code handles various corner cases based on how that dependency behaves (e.g., does the code handle redirects?), you'd have no way to do it without control of that external dependency.

Although working with real dependencies might make sense for integration and end-to-end tests, with unit tests, you should try to minimize external dependencies as much as possible. The typical strategy for doing this is *dependency injection*, in which you make it possible to pass in (or "inject")

external dependencies from outside your code, rather than hardcoding them within your code.

For example, the `Handlers` class shouldn't need to deal with all of the details of how to call a web service. Instead, you can extract that logic into a separate `WebService` class:

```
class WebService
  def initialize(url)
    @uri = URI(url)
  end

  def proxy
    response = Net::HTTP.get_response(@uri)
    [response.code.to_i, response['Content-Type'], response.body]
  end
end
```

This class takes a URL as an input and exposes a `proxy` method to proxy the HTTP GET response from that URL. You can then update the `Handlers` class to take a `WebService` instance as an input and use that instance in the `web_service` method:

```
class Handlers
  def initialize(web_service)
    @web_service = web_service
  end

  def handle(path)
    case path
    when "/"
      [200, 'text/plain', 'Hello, World']
    when "/api"
      [201, 'application/json', '{"foo":"bar"}']
    when "/web-service"
      # New endpoint that calls a web service
      @web_service.proxy
    else
      [404, 'text/plain', 'Not Found']
    end
  end
end
```

Now, in your implementation code, you can inject a real `WebService` instance that makes HTTP calls to `example.org`:

```
class WebServer < WEBrick::HTTPServlet::AbstractServlet
  def do_GET(request, response)
    web_service = WebService.new("http://www.example.org")
    handlers = Handlers.new(web_service)

    status_code, content_type, body =
handlers.handle(request.path)

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

In your test code, you can create a mock version of the `WebService` class that allows you to specify a mock response to return:

```
class MockWebService
  def initialize(response)
    @response = response
  end

  def proxy
    @response
  end
end
```

And now you can create an instance of this `MockWebService` class and inject it into the `Handlers` class in your unit tests:

```
  def test_unit_web_service
    expected_status = 200
    expected_content_type = 'text/html'
    expected_body = 'mock example.org'
    mock_response = [expected_status, expected_content_type,
expected_body]

    mock_web_service = MockWebService.new(mock_response)
    handlers = Handlers.new(mock_web_service)
```

```
    status_code, content_type, body = handlers.handle("/web-
service")
    assert_equal(expected_status, status_code)
    assert_equal(expected_content_type, content_type)
    assert_equal(expected_body, body)
  end
```

Rerun the tests to make sure it all still works:

```
$ ruby web-server-test.rb
Loaded suite web-server-test
Started
...

Finished in 0.000645 seconds.
-------------------------------------------
4 tests, 12 assertions, 0 failures, 0 errors
100% passed
-------------------------------------------
```

Fantastic. Using dependency injection to minimize external dependencies allows you to write fast, reliable tests and check all the various corner cases. And since the three test cases you added earlier are still passing, you can be confident that your refactoring hasn't broken anything.

Let's now turn our attention back to Terraform and see what dependency injection looks like with Terraform modules, starting with the `hello-world-app` module. If you haven't already, the first step is to create an easy-to-deploy example for it in the *examples* folder:

```
provider "aws" {
  region = "us-east-2"
}

module "hello_world_app" {
  source = "../../../modules/services/hello-world-app"

  server_text = "Hello, World"
  environment = "example"

  db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
  db_remote_state_key    = "examples/terraform.tfstate"
```

```
  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
  ami                = data.aws_ami.ubuntu.id
}

data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-
server-*"]
  }
}
```

The dependency problem becomes apparent when you spot the parameters `db_remote_state_bucket` and `db_remote_state_key`: the `hello-world-app` module assumes that you've already deployed the `mysql` module and requires that you pass in the details of the S3 bucket where the `mysql` module is storing state using these two parameters. The goal here is to create a unit test for the `hello-world-app` module, and although a pure unit test with 0 external dependencies isn't possible with Terraform, it's still a good idea to minimize external dependencies whenever possible.

One of the first steps with minimizing dependencies is to make it clearer what dependencies your module has. A file-naming convention you might want to adopt is to move all of the data sources and resources that represent external dependencies into a separate *dependencies.tf* file. For example, here's what *modules/services/hello-world-app/dependencies.tf* would look like:

```
data "terraform_remote_state" "db" {
  backend = "s3"

  config = {
    bucket = var.db_remote_state_bucket
    key    = var.db_remote_state_key
```

```
    region = "us-east-2"
  }
}

data "aws_vpc" "default" {
  default = true
}

data "aws_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}
```

This convention makes it easier for users of your code to know, at a glance, what this code depends on in the outside world. In the case of the `hello-world-app` module, you can quickly see that it depends on a database, VPC, and subnets. So, how can you inject these dependencies from outside the module so that you can replace them at test time? You already know the answer to this: input variables.

For each of these dependencies, you should add a new input variable in *modules/services/hello-world-app/variables.tf*:

```
variable "vpc_id" {
  description = "The ID of the VPC to deploy into"
  type        = string
  default     = null
}

variable "subnet_ids" {
  description = "The IDs of the subnets to deploy into"
  type        = list(string)
  default     = null
}

variable "mysql_config" {
  description = "The config for the MySQL DB"
  type        = object({
    address = string
    port    = number
  })
```

```
  default     = null
}
```

There's now an input variable for the VPC ID, subnet IDs, and MySQL config. Each variable specifies a `default`, so they are *optional variables* that the user can set to something custom or omit to get the `default` value. The `default` for each variable is `null`.

Note that the `mysql_config` variable uses the `object` type constructor to create a nested type with `address` and `port` keys. This type is intentionally designed to match the output types of the `mysql` module:

```
output "address" {
  value       = aws_db_instance.example.address
  description = "Connect to the database at this endpoint"
}

output "port" {
  value       = aws_db_instance.example.port
  description = "The port the database is listening on"
}
```

One of the advantages of doing this is that, as soon as the refactor is complete, one of the ways you'll be able to use the `hello-world-app` and `mysql` modules together is as follows:

```
module "hello_world_app" {
  source = "../../../modules/services/hello-world-app"

  server_text            = "Hello, World"
  environment            = "example"

  # Pass all the outputs from the mysql module straight through!
  mysql_config = module.mysql

  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
  ami                = data.aws_ami.ubuntu.id
}
```

```
module "mysql" {
  source = "../../../modules/data-stores/mysql"

  db_name     = var.db_name
  db_username = var.db_username
  db_password = var.db_password
}
```

Because the `type` of `mysql_config` matches the type of the `mysql` module outputs, you can pass them all straight through in one line. And if the types are ever changed and no longer match, Terraform will give you an error right away so that you know to update them. This is not only function composition at work but also type-safe function composition.

But before that can work, you'll need to finish refactoring the code. Because the MySQL configuration can be passed in as an input, this means that the `db_remote_state_bucket` and `db_remote_state_key` variables should now be optional, so set their `default` values to `null`:

```
variable "db_remote_state_bucket" {
  description = "The name of the S3 bucket for the DB's Terraform
state"
  type        = string
  default     = null
}

variable "db_remote_state_key" {
  description = "The path in the S3 bucket for the DB's Terraform
state"
  type        = string
  default     = null
}
```

Next, use the `count` parameter to optionally create the three data sources in *modules/services/hello-world-app/dependencies.tf* based on whether the corresponding input variable is set to `null`:

```
data "terraform_remote_state" "db" {
  count = var.mysql_config == null ? 1 : 0

  backend = "s3"
```

```
    config = {
      bucket = var.db_remote_state_bucket
      key    = var.db_remote_state_key
      region = "us-east-2"
    }
  }

  data "aws_vpc" "default" {
    count   = var.vpc_id == null ? 1 : 0
    default = true
  }

  data "aws_subnets" "default" {
    count = var.subnet_ids == null ? 1 : 0
    filter {
      name   = "vpc-id"
      values = [data.aws_vpc.default.id]
    }
  }
```

Now you need to update any references to these data sources to conditionally use either the input variable or the data source. Let's capture these as local values:

```
  locals {
    mysql_config = (
      var.mysql_config == null
        ? data.terraform_remote_state.db[0].outputs
        : var.mysql_config
    )

    vpc_id = (
      var.vpc_id == null
        ? data.aws_vpc.default[0].id
        : var.vpc_id
    )

    subnet_ids = (
      var.subnet_ids == null
        ? data.aws_subnets.default[0].ids
        : var.subnet_ids
    )
  }
```

Note that because the data sources use the `count` parameters, they are now arrays, so any time you reference them, you need to use array lookup syntax (i.e., `[0]`). Go through the code, and anywhere you find a reference to one of these data sources, replace it with a reference to one of the local variables you just added instead. Start by updating the `aws_subnets` data source to use `local.vpc_id`:

```
data "aws_subnets" "default" {
  count = var.subnet_ids == null ? 1 : 0
  filter {
    name   = "vpc-id"
    values = [local.vpc_id]
  }
}
```

Then, set the `subnet_ids` parameter of the `alb` module to `local.subnet_ids`:

```
module "alb" {
  source = "../../networking/alb"

  alb_name   = "hello-world-${var.environment}"
  subnet_ids = local.subnet_ids
}
```

In the `asg` module, make the following updates: set the `subnet_ids` parameter to `local.subnet_ids`, and in the `user_data` variables, update `db_address` and `db_port` to read data from `local.mysql_config`.

```
module "asg" {
  source = "../../cluster/asg-rolling-deploy"

  cluster_name  = "hello-world-${var.environment}"
  ami           = var.ami
  instance_type = var.instance_type

  user_data = templatefile("${path.module}/user-data.sh", {
    server_port = var.server_port
    db_address  = local.mysql_config.address
```

```
    db_port     = local.mysql_config.port
    server_text = var.server_text
  })

  min_size           = var.min_size
  max_size           = var.max_size
  enable_autoscaling = var.enable_autoscaling

  subnet_ids        = local.subnet_ids
  target_group_arns = [aws_lb_target_group.asg.arn]
  health_check_type = "ELB"

  custom_tags = var.custom_tags
}
```

Finally, update the `vpc_id` parameter of the `aws_lb_target_group` to use `local.vpc_id`:

```
resource "aws_lb_target_group" "asg" {
  name     = "hello-world-${var.environment}"
  port     = var.server_port
  protocol = "HTTP"
  vpc_id   = local.vpc_id

  health_check {
    path                = "/"
    protocol            = "HTTP"
    matcher             = "200"
    interval            = 15
    timeout             = 3
    healthy_threshold   = 2
    unhealthy_threshold = 2
  }
}
```

With these updates, you can now choose to inject the VPC ID, subnet IDs, and/or MySQL config parameters into the `hello-world-app` module, or omit any of those parameters, and the module will use an appropriate data source to fetch those values by itself. Let's update the "Hello, World" app example to allow the MySQL config to be injected but omit the VPC ID and subnet ID parameters because using the default VPC is good enough

for testing. Add a new input variable to *examples/hello-world-app/variables.tf*:

```
variable "mysql_config" {
  description = "The config for the MySQL DB"

  type = object({
    address = string
    port    = number
  })

  default = {
    address = "mock-mysql-address"
    port    = 12345
  }
}
```

Pass this variable through to the `hello-world-app` module in *examples/hello-world-app/main.tf*:

```
module "hello_world_app" {
  source = "../../../modules/services/hello-world-app"

  server_text = "Hello, World"
  environment = "example"

  mysql_config = var.mysql_config

  instance_type      = "t2.micro"
  min_size           = 2
  max_size           = 2
  enable_autoscaling = false
  ami                = data.aws_ami.ubuntu.id
}
```

You can now set this `mysql_config` variable in a unit test to any value you want. Create a unit test in *test/hello_world_app_example_test.go* with the following contents:

```
func TestHelloWorldAppExample(t *testing.T) {
        opts := &terraform.Options{
                // You should update this relative path to point
at your
```

```
                // hello-world-app example directory!
                TerraformDir: "../examples/hello-world-
app/standalone",
        }

        // Clean up everything at the end of the test
        defer terraform.Destroy(t, opts)
        terraform.InitAndApply(t, opts)

        albDnsName := terraform.OutputRequired(t, opts,
"alb_dns_name")
        url := fmt.Sprintf("http://%s", albDnsName)

        maxRetries := 10
        timeBetweenRetries := 10 * time.Second

        http_helper.HttpGetWithRetryWithCustomValidation(
                t,
                url,
                nil,
                maxRetries,
                timeBetweenRetries,
                func(status int, body string) bool {
                        return status == 200 &&
                                strings.Contains(body, "Hello,
World")
                },
        )
}
```

This code is nearly identical to the unit test for the `alb` example, with only two differences:

- The `TerraformDir` setting is pointing to the `hello-world-app` example instead of the `alb` example (be sure to update the path as necessary for your filesystem).
- Instead of using `http_helper.HttpGetWithRetry` to check for a 404 response, the test is using the `http_helper.HttpGetWithRetryWithCustomValidation` method to check for a 200 response and a body that contains the text "Hello, World." That's because the User Data script of the `hello-`

`world-app` module returns a 200 OK response that includes not only the server text but also other text, including HTML.

There's just one new thing you'll need to add to this test—set the `mysql_config` variable:

```
        opts := &terraform.Options{
                // You should update this relative path to point
at your
                // hello-world-app example directory!
                TerraformDir: "../examples/hello-world-
app/standalone",

                Vars: map[string]interface{}{
                        "mysql_config": map[string]interface{}{
                                "address": "mock-value-for-test",
                                "port":    3306,
                        },
                },
        }
```

The `Vars` parameter in `terraform.Options` allows you to set variables in your Terraform code. This code is passing in some mock data for the `mysql_config` variable. Alternatively, you could set this value to anything you want: for example, you could fire up a small, in-memory database at test time and set the `address` to that database's IP.

Run this new test using `go test`, specifying the `-run` argument to run *just* this test (otherwise, Go's default behavior is to run all tests in the current folder, including the ALB example test you created earlier):

```
$ go test -v -timeout 30m -run TestHelloWorldAppExample

(...)

PASS
ok      terraform-up-and-running        204.113s
```

If all goes well, the test will run `terraform apply`, make repeated HTTP requests to the load balancer, and, as soon as it gets back the