

This code will send a 200 response with the body “Hello, World” for the / URL, a 201 response with a JSON body for the /api URL, and a 404 for all other URLs. How would you manually test this code? The typical answer is to add a bit of code to run the web server on localhost:

```
# This will only run if this script was called directly from the
CLI, but
# not if it was required from another file
if __FILE__ == $0
  # Run the server on localhost at port 8000
  server = WEBrick::HTTPServer.new :Port => 8000
  server.mount '/', WebServer

  # Shut down the server on CTRL+C
  trap 'INT' do server.shutdown end

  # Start the server
  server.start
end
```

When you run this file from the CLI, it will start the web server on port 8000:

```
$ ruby web-server.rb
[2019-05-25 14:11:52] INFO WEBrick 1.3.1
[2019-05-25 14:11:52] INFO ruby 2.3.7 (2018-03-28)
[universal.x86_64-darwin17]
[2019-05-25 14:11:52] INFO WEBrick::HTTPServer#start: pid=19767
port=8000
```

You can test this server using a web browser or curl:

```
$ curl localhost:8000/
Hello, World
```

Manual Testing Basics

What is the equivalent of this sort of manual testing with Terraform code? For example, from the previous chapters, you already have Terraform code

for deploying an ALB. Here's a snippet from *modules/networking/alb/main.tf*:

```
resource "aws_lb" "example" {
  name                = var.alb_name
  load_balancer_type = "application"
  subnets            = var.subnet_ids
  security_groups     = [aws_security_group.alb.id]
}

resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = local.http_port
  protocol          = "HTTP"

  # By default, return a simple 404 page
  default_action {
    type = "fixed-response"

    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}

resource "aws_security_group" "alb" {
  name = var.alb_name
}

# (...)
```

If you compare this code to the Ruby code, one difference should be fairly obvious: you can't deploy AWS ALBs, target groups, listeners, security groups, and all the other infrastructure on your own computer.

This brings us to *key testing takeaway #1*: when testing Terraform code, you can't use localhost. This applies to most IaC tools, not just Terraform. The only practical way to do manual testing with Terraform is to deploy to a real environment (i.e., deploy to AWS). In other words, the way you've been manually running `terraform apply` and `terraform destroy` throughout the book is how you do manual testing with Terraform.

This is one of the reasons why it's essential to have easy-to-deploy examples in the *examples* folder for each module, as described in [Chapter 8](#). The easiest way to manually test the `alb` module is to use the example code you created for it in *examples/alb*:

```
provider "aws" {  
  region = "us-east-2"  
}  
  
module "alb" {  
  source = "../../modules/networking/alb"  
  
  alb_name      = "terraform-up-and-running"  
  subnet_ids    = data.aws_subnets.default.ids  
}
```

As you've done many times throughout the book, you deploy this example code using `terraform apply`:

```
$ terraform apply
```

```
(...)
```

```
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
```

```
Outputs:
```

```
alb_dns_name = "hello-world-stage-477699288.us-east-  
2.elb.amazonaws.com"
```

When the deployment is done, you can use a tool such as `curl` to test, for example, that the default action of the ALB is to return a 404:

```
$ curl \  
  -s \  
  -o /dev/null \  
  -w "%{http_code}" \  
  hello-world-stage-477699288.us-east-2.elb.amazonaws.com
```

```
404
```