

In this chapter, you'll see:

- Performing validation and error reporting
- Unit testing

CHAPTER 7

Task B: Validation and Unit Testing

At this point, we have an initial model for a product as well as a complete maintenance application for this data provided for us by Rails scaffolding. In this chapter, we're going to focus on making the model more bulletproof—as in, making sure that errors in the data provided never get committed to the database—before we proceed to other aspects of the Depot application in subsequent chapters.

Iteration B1: Validating!

While playing with the results of iteration A1, our client noticed something. If she entered an invalid price or forgot to set up a product description, the application happily accepted the form and added a line to the database. A missing description is embarrassing, and a price of \$0.00 costs her actual money, so she asked that we add validation to the application. No product should be allowed in the database if it has an empty title or description field, an invalid URL for the image, or an invalid price.

So where do we put the validation? The model layer is the gatekeeper between the world of code and the database. Nothing to do with our application comes out of the database or gets stored into the database that doesn't first go through the model. This makes models an ideal place to put validations; it doesn't matter whether the data comes from a form or from some programmatic manipulation in our application. If a model checks it before writing to the database, the database will be protected from bad data.

Let's look at the source code of the model class (in `app/models/product.rb`):

```
class Product < ApplicationRecord
end
```

Adding our validation should be fairly clean. Let's start by validating that the text fields all contain something before a row is written to the database. We do this by adding some code to the existing model:

```
validates :title, :description, :image_url, presence: true
```

The `validates()` method is the standard Rails validator. It checks one or more model fields against one or more conditions.

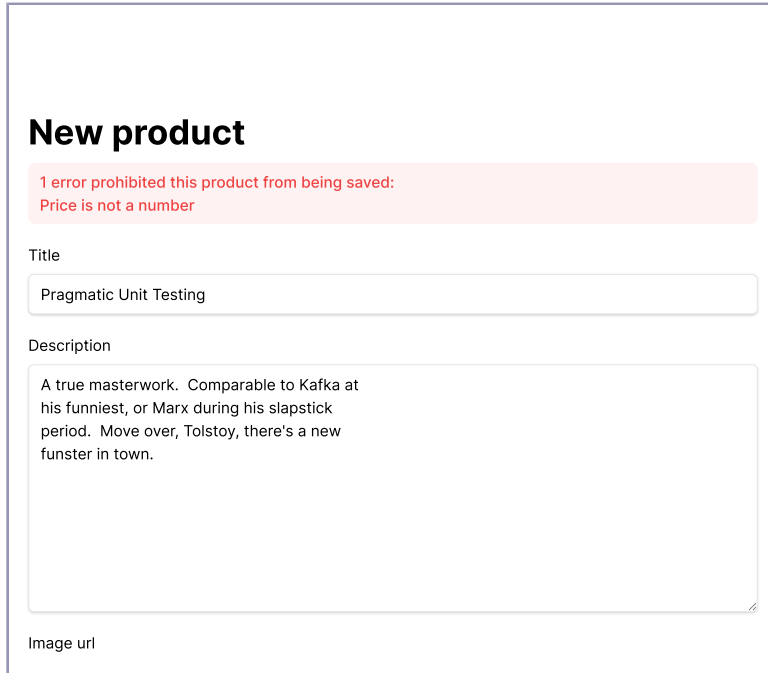
`presence: true` tells the validator to check that each of the named fields is present and that its contents aren't empty. The following screenshot shows what happens if we try to submit a new product with none of the fields filled in. Try it by visiting <http://localhost:3000/products/new> and submitting the form without entering any data. It's pretty impressive: the fields with errors are highlighted, and the errors are summarized in a nice list at the top of the form. That's not bad for one line of code. You might also have noticed that after editing and saving the `product.rb` file, you didn't have to restart the application to test your changes. The same reloading that caused Rails to notice the earlier change to our schema also means it'll always use the latest version of our code.

The screenshot shows a web form titled "New product". At the top, a red error message box states: "3 errors prohibited this product from being saved: Title can't be blank, Description can't be blank, Image url can't be blank". Below this, the form has four input fields: "Title" (a single-line text box), "Description" (a large multi-line text area), "Image url" (a single-line text box), and "Price" (a single-line text box with the value "0.0"). At the bottom of the form are two buttons: "Create Product" (a blue button) and "Back to products" (a grey button).

We'd also like to validate that the price is a valid, positive number. We'll use the delightfully named `numericality()` option to verify that the price is a valid number. We also pass the rather verbosely named `:greater_than_or_equal_to` option a value of 0.01:

```
validates :price, numericality: { greater_than_or_equal_to: 0.01 }
```

Now, if we add a product with an invalid price, the appropriate message will appear, as shown in the following screenshot.



The screenshot shows a web form titled "New product". At the top, there is a red error message box that says "1 error prohibited this product from being saved: Price is not a number". Below this, the form has three sections: "Title" with a text input field containing "Pragmatic Unit Testing"; "Description" with a large text area containing the text "A true masterwork. Comparable to Kafka at his funniest, or Marx during his slapstick period. Move over, Tolstoy, there's a new funster in town."; and "Image url" with an empty text input field.

Why test against one cent rather than zero? Well, it's possible to enter a number such as 0.001 into this field. Because the database stores just two digits after the decimal point, this would end up being zero in the database, even though it would pass the validation if we compared against zero. Checking that the number is at least one cent ensures that only correct values end up being stored.

We have two more items to validate. First, we want to make sure that each product has a unique title. One more line in the `Product` model will do this. The uniqueness validation will perform a check to ensure that no other row in the `products` table has the same title as the row we're about to save:

```
validates :title, uniqueness: true
```

Lastly, we need to validate that the URL entered for the image is valid. We'll do this by using the format option, which matches a field against a regular expression. For now, let's just check that the URL ends with one of .gif, .jpg, or .png:

```
validates :image_url, allow_blank: true, format: {
  with:   %r{\.(gif|jpg|png)\z}i,
  message: 'must be a URL for GIF, JPG or PNG image.'
}
```

The regular expression matches the string against a literal dot, followed by one of three choices, followed by the end of the string. Be sure to use vertical bars to separate options, and backslashes before the dot and the lowercase z. If you need a refresher on regular expression syntax, see [Regular Expressions, on page 50](#).

Note that we use the allow_blank option to avoid getting multiple error messages when the field is blank.

Later, we'd probably want to change this form to let the user select from a list of available images, but we'd still want to keep the validation to prevent malicious folks from submitting bad data directly.

So in a couple of minutes we've added validations that check the following:

- The title, description, and image URL fields aren't empty.
- The price is a valid number not less than \$0.01.
- The title is unique among all products.
- The image URL looks reasonable.

Your updated Product model should look like this:

```
rails7/depot_b/app/models/product.rb
class Product < ApplicationRecord
  validates :title, :description, :image_url, presence: true
  validates :title, uniqueness: true
  validates :image_url, allow_blank: true, format: {
    with:   %r{\.(gif|jpg|png)\z}i,
    message: 'must be a URL for GIF, JPG or PNG image.'
  }
  validates :price, numericality: { greater_than_or_equal_to: 0.01 }
end
```

Nearing the end of this cycle, we ask our customer to play with the application, and she's a lot happier. It took only a few minutes, but the simple act of adding validation has made the product maintenance pages seem a lot more solid.

Iteration B2: Unit Testing of Models

One of the joys of the Rails framework is that it has support for testing baked right in from the start of every project. As you've seen, from the moment you create a new application using the rails command, Rails starts generating a test infrastructure for you. Let's take a peek inside the models subdirectory to see what's already there:

```
depot> ls test/models
product_test.rb
```

product_test.rb is the file that Rails created to hold the unit tests for the model we created earlier with the generate script. This is a good start, but Rails can help us only so much. Let's see what kind of test goodies Rails generated inside test/models/product_test.rb when we generated that model:

```
rails7/depot_a/test/models/product_test.rb
require "test_helper"

class ProductTest < ActiveSupport::TestCase
  # test "the truth" do
  #   assert true
  # end
end
```

The generated ProductTest is a subclass of ActiveSupport::TestCase.¹ The fact that ActiveSupport::TestCase is a subclass of the MiniTest::Test class tells us that Rails generates tests based on the MiniTest² framework that comes preinstalled with Ruby. This is good news, because it means if we've already been testing our Ruby programs with MiniTest tests (and why wouldn't we be?), we can build on that knowledge to test Rails applications. If you're new to MiniTest, don't worry. We'll take it slow.

Inside this test case, Rails generated a single commented-out test called "the truth". The test...do syntax may seem surprising at first, but here ActiveSupport::TestCase is combining a class method, optional parentheses, and a block to make defining a test method the tiniest bit simpler for you. Sometimes it's the little things that make all the difference.

The assert line in this method is a test. It isn't much of one, though—all it does is test that true is true. Clearly, this is a placeholder, one that's intended to be replaced by your actual tests.

1. <http://api.rubyonrails.org/classes/ActiveSupport/TestCase.html>

2. <http://docs.seattlerb.org/minitest/>

A Real Unit Test

Let's get on to the business of testing validation. First, if we create a product with no attributes set, we'll expect it to be invalid and for an error to be associated with each field. We can use the model's `errors()` and `invalid?()` methods to see if it validates, and we can use the `any?()` method of the error list to see if an error is associated with a particular attribute.

Now that we know *what* to test, we need to know *how* to tell the test framework whether our code passes or fails. We do that using *assertions*. An assertion is a method call that tells the framework what we expect to be true. The simplest assertion is the `assert()` method, which expects its argument to be true. If it is, nothing special happens. However, if the argument to `assert()` is false, the assertion fails. The framework will output a message and will stop executing the test method containing the failure. In our case, we expect that an empty Product model won't pass validation, so we can express that expectation by asserting that it isn't valid:

```
assert product.invalid?
```

Replace the test the truth with the following code:

```
rails7/depot_b/test/models/product_test.rb
test "product attributes must not be empty" do
  product = Product.new
  assert product.invalid?
  assert product.errors[:title].any?
  assert product.errors[:description].any?
  assert product.errors[:price].any?
  assert product.errors[:image_url].any?
end
```

We can rerun just the unit tests by issuing the `rails test:models` command. When we do so, we now see the test execute successfully:

```
depot> bin/rails test:models
Run options: --seed 63304

# Running:

.

Finished in 0.021068s, 47.4654 runs/s, 237.3268 assertions/s.
1 runs, 5 assertions, 0 failures, 0 errors, 0 skips
```

Sure enough, the validation kicked in, and all our assertions passed.

Clearly, at this point we can dig deeper and exercise individual validations. Let's look at three of the many possible tests.

First, we'll check that the validation of the price works the way we expect:

```
rails7/depot_c/test/models/product_test.rb
test "product price must be positive" do
  product = Product.new(title: "My Book Title",
                        description: "yyy",
                        image_url: "zzz.jpg")

  product.price = -1
  assert product.invalid?
  assert_equal ["must be greater than or equal to 0.01"],
    product.errors[:price]

  product.price = 0
  assert product.invalid?
  assert_equal ["must be greater than or equal to 0.01"],
    product.errors[:price]

  product.price = 1
  assert product.valid?
end
```

In this code, we create a new product and then try setting its price to -1, 0, and +1, validating the product each time. If our model is working, the first two should be invalid, and we verify that the error message associated with the price attribute is what we expect.

The last price is acceptable, so we assert that the model is now valid. (Some folks would put these three tests into three separate test methods—that's perfectly reasonable.)

Next, we test that we're validating that the image URL ends with one of .gif, .jpg, or .png:

```
rails7/depot_c/test/models/product_test.rb
def new_product(image_url)
  Product.new(title: "My Book Title",
              description: "yyy",
              price: 1,
              image_url: image_url)
end

test "image url" do
  ok = %w{ fred.gif fred.jpg fred.png FRED.JPG FRED.Jpg
    http://a.b.c/x/y/z/fred.gif }
  bad = %w{ fred.doc fred.gif/more fred.gif.more }

  ok.each do |image_url|
    assert new_product(image_url).valid?,
      "#{image_url} must be valid"
  end
end
```

```

bad.each do |image_url|
  assert new_product(image_url).invalid?,
    "#{image_url} must be invalid"
end
end

```

Here we've mixed things up a bit. Rather than write the nine separate tests, we've used a couple of loops—one to check the cases we expect to pass validation and the second to try cases we expect to fail. At the same time, we factored out the common code between the two loops.

You'll notice that we also added an extra parameter to our assert method calls. All of the testing assertions accept an optional trailing parameter containing a string. This will be written along with the error message if the assertion fails and can be useful for diagnosing what went wrong.

Finally, our model contains a validation that checks that all the product titles in the database are unique. To test this one, we need to store product data in the database.

One way to do this would be to have a test create a product, save it, then create another product with the same title and try to save it too. This would clearly work. But a much simpler way is to use Rails fixtures.

Test Fixtures

In the world of testing, a *fixture* is an environment in which you can run a test. If you're testing a circuit board, for example, you might mount it in a test fixture that provides it with the power and inputs needed to drive the function to be tested.

In the world of Rails, a test fixture is a specification of the initial contents of a model (or models) under test. If, for example, we want to ensure that our products table starts off with known data at the start of every unit test, we can specify those contents in a fixture, and Rails takes care of the rest.

You specify fixture data in files in the `test/fixtures` directory. These files contain test data in YAML format. Each fixture file contains the data for a single model. The name of the fixture file is significant: the base name of the file must match the name of a database table. Because we need some data for a Product model, which is stored in the products table, we'll add it to the file called `products.yml`.

Rails already created this fixture file when we first created the model:


```
rails7/depot_a/test/fixtures/products.yml
# Read about fixtures at
# https://api.rubyonrails.org/classes/ActiveRecord/FixtureSet.html

one:
  title: MyString
  description: MyText
  image_url: MyString
  price: 9.99

two:
  title: MyString
  description: MyText
  image_url: MyString
  price: 9.99
```

The fixture file contains an entry for each row that we want to insert into the database. Each row is given a name. In the case of the Rails-generated fixture, the rows are named one and two. This name has no significance as far as the database is concerned—it isn't inserted into the row data. Instead, as you'll see shortly, the name gives us a convenient way to reference test data inside our test code. They also are the names used in the generated integration tests, so for now, we'll leave them alone.



David says:

Picking Good Fixture Names

As with the names of variables in general, you want to keep the names of fixtures as self-explanatory as possible. This increases the readability of the tests when you're asserting that `product(:valid_order_for_fred)` is indeed Fred's valid order. It also makes it a lot easier to remember which fixture you're supposed to test against, without having to look up `p1` or `order4`. The more fixtures you get, the more important it is to pick good fixture names. So starting early keeps you happy later.

But what do we do with fixtures that can't easily get a self-explanatory name like `valid_order_for_fred`? Pick natural names that you have an easier time associating to a role. For example, instead of using `order1`, use `christmas_order`. Instead of `customer1`, use `fred`. Once you get into the habit of natural names, you'll soon be weaving a nice little story about how `fred` is paying for his `christmas_order` with his `invalid_credit_card` first, then paying with his `valid_credit_card`, and finally choosing to ship it all off to `aunt_mary`.

Association-based stories are key to remembering large worlds of fixtures with ease.

Inside each entry you can see an indented list of name-value pairs. As in your `config/database.yml`, you must use spaces, not tabs, at the start of each of the data lines, and all the lines for a row must have the same indentation. Be careful as you make changes, because you need to make sure the names of

the columns are correct in each entry; a mismatch with the database column names can cause a hard-to-track-down exception.

This data is used in tests. In fact, if you rerun `bin/rails test` now you'll see a number of errors, including the following error:

```
Error:
ProductsControllerTest#test_should_get_index:
ActionView::Template::Error: The asset "MyString" is not present in
the asset pipeline.
```

The reason for the failure is that we recently added an `image_tag` to the product index page and Rails can't find an image by the name `MyString` (remember that `image_tag()` is a Rails helper method that produces an HTML `` element). Let's correct that error and, while we're here, add some more data to the fixture file with something we can use to test our Product model:

```
rails7/depot_c/test/fixtures/products.yml
# Read about fixtures at
# https://api.rubyonrails.org/classes/ActiveRecord/FixtureSet.html

one:
  title: MyString
  description: MyText
  image_url: lorem.jpg
  price: 9.99

two:
  title: MyString
  description: MyText
  image_url: lorem.jpg
  price: 9.99

ruby:
  title:      Programming Ruby 1.9
  description:
  Ruby is the fastest growing and most exciting dynamic
  language out there. If you need to get working programs
  delivered fast, you should add Ruby to your toolbox.
  price:      49.50
  image_url:  ruby.jpg
```

Note that the images referenced in `image_url` do need to exist for the tests to succeed. It doesn't matter what they are as long as they're in `app/assets/images` when the tests run. You can either create some yourself or use the ones provided in the downloadable code.

Now that we have a fixture file, we want Rails to load the test data into the products table when we run the unit test. And, in fact, Rails is already doing

this (convention over configuration for the win!), but you can control which fixtures to load by specifying the following line in `test/models/product_test.rb`:

```

class ProductTest < ActiveSupport::TestCase
  fixtures :products
  #...
end

```

The `fixtures()` directive loads the fixture data corresponding to the given model name into the corresponding database table before each test method in the test case is run. The name of the fixture file determines the table that's loaded, so using `:products` will cause the `products.yml` fixture file to be used.

Let's say that again another way. In the case of our `ProductTest` class, adding the `fixtures` directive means that the `products` table will be emptied out and then populated with the three rows defined in the fixture before each test method is run.

Note that most of the scaffolding that Rails generates doesn't contain calls to the `fixtures` method. That's because the default for tests is to load *all* fixtures before running the test. Because that default is generally the one you want, there usually isn't any need to change it. Once again, conventions are used to eliminate the need for unnecessary configuration.

So far, we've been doing all our work in the development database. Now that we're running tests, though, Rails needs to use a test database. If you look in the `database.yml` file in the `config` directory, you'll notice Rails actually created a configuration for three separate databases.

- `db/development.sqlite3` will be our development database. All of our programming work will be done here.
- `db/test.sqlite3` is a test database.
- `db/production.sqlite3` is the production database. Our application will use this when we put it online.

Each test method gets a freshly initialized table in the test database, loaded from the fixtures we provide. This is automatically done by the `bin/rails test` command but can be done separately via `bin/rails db:test:prepare`.

Using Fixture Data

Now that you know how to get fixture data into the database, we need to find ways of using it in our tests.

Clearly, one way would be to use the finder methods in the model to read the data. However, Rails makes it easier than that. For each fixture it loads into a test, Rails defines a method with the same name as the fixture. You can

use this method to access preloaded model objects containing the fixture data: simply pass it the name of the row as defined in the YAML fixture file, and it'll return a model object containing that row's data.

In the case of our product data, calling `products(:ruby)` returns a `Product` model containing the data we defined in the fixture. Let's use that to test the validation of unique product titles:

```
rails7/depot_c/test/models/product_test.rb
test "product is not valid without a unique title" do
  product = Product.new(title:      products(:ruby).title,
                        description: "yyy",
                        price:       1,
                        image_url:   "fred.gif")

  assert product.invalid?
  assert_equal ["has already been taken"], product.errors[:title]
end
```

The test assumes that the database already includes a row for the Ruby book. It gets the title of that existing row using this:

```
products(:ruby).title
```

It then creates a new `Product` model, setting its title to that existing title. It asserts that attempting to save this model fails and that the title attribute has the correct error associated with it.

If you want to avoid using a hardcoded string for the Active Record error, you can compare the response against its built-in error message table:

```
rails7/depot_c/test/models/product_test.rb
test "product is not valid without a unique title - i18n" do
  product = Product.new(title:      products(:ruby).title,
                        description: "yyy",
                        price:       1,
                        image_url:   "fred.gif")

  assert product.invalid?
  assert_equal [I18n.translate('errors.messages.taken')],
               product.errors[:title]
end
```

We'll cover the `I18n` functions in [Chapter 15, Task J: Internationalization, on page 225](#).

Before we move on, we once again try our tests:

```
$ bin/rails test
```

This time we see two remaining failures, both in `test/controllers/products_controller_test.rb`: one in `should create product` and the other in `should update product`. Clearly, something we did caused something to do with the creation and update of products to fail. Since we just added validations on how products are created or updated, it's likely this is the source of the problem, and our test is out-of-date.

The specifics of the problem might not be obvious from the test failure message, but the failure for `should create product` gives us a clue: "Product.count didn't change by 1." Since we just added validations, it seems likely that our attempts to create a product in the test are creating an invalid product, which we can't save to the database.

Let's verify this assumption by adding a call to `puts()` in the controller's `create()` method:

```
def create
  @product = Product.new(product_params)

  respond_to do |format|
    if @product.save
      format.html { redirect_to @product,
        notice: "Product was successfully created." }
      format.json { render :show, status: :created,
        location: @product }
    else
      puts @product.errors.full_messages
      format.html { render :new,
        status: :unprocessable_entity }
      format.json { render json: @product.errors,
        status: :unprocessable_entity }
    end
  end
end
```

If we rerun just the test for creating a new product, we'll see the problem:

```
> bin/rails test test/controllers/products_controller_test.rb:19
# Running:
```

```
Title has already been taken
```

```
F
```

```
Failure:
```

```
ProductsControllerTest#test_should_create_product [«path to test»]
```

```
"Product.count" didn't change by 1.
```

```
Expected: 3
```

```
Actual: 2
```

```
bin/rails test test/controllers/products_controller_test.rb:18
```

```
Finished in 0.427810s, 2.3375 runs/s, 2.3375 assertions/s.
```

```
1 runs, 1 assertions, 1 failures, 0 errors, 0 skips
```

Our `puts()` is printing the validation error, which in this case is “Title has already been taken.” In other words, we’re trying to create a product whose title already exists. Instead, let’s create a random book title and use that instead of the value coming out of the test fixture. First, we’ll create a random title in the `setup()` block:

```
rails7/depot_b/test/controllers/products_controller_test.rb
```

```
require "test_helper"
```

```
class ProductsControllerTest < ActionDispatch::IntegrationTest
```

```
  setup do
```

```
    @product = products(:one)
```

```
➤    @title = "The Great Book #{rand(1000)}"
```

```
  end
```

Next, we’ll use that instead of the default `@product.title` that the Rails generator put into the test. The actual change is highlighted (the use of `@title`), but the code had to be reformatted to fit the space, so this will look a bit different for you:

```
rails7/depot_b/test/controllers/products_controller_test.rb
```

```
test "should create product" do
```

```
  assert_difference("Product.count") do
```

```
    post products_url, params: {
```

```
      product: {
```

```
        description: @product.description,
```

```
        image_url: @product.image_url,
```

```
        price: @product.price,
```

```
➤        title: @title,
```

```
      }
```

```
    }
```

```
  end
```

```
  assert_redirected_to product_url(Product.last)
```

```
end
```

```
rails7/depot_b/test/controllers/products_controller_test.rb
```

```
test "should update product" do
```

```
  patch product_url(@product), params: {
```

```
    product: {
```

```
      description: @product.description,
```

```
      image_url: @product.image_url,
```

```
      price: @product.price,
```

```
➤      title: @title,
```

```
    }
```

```
  }
```

```
  assert_redirected_to product_url(@product)
```

```
end
```

After making these changes, we rerun the tests, and they report that all is well.

Now we can feel confident that our validation code not only works but will continue to work. Our product now has a model, a set of views, a controller, and a set of unit tests. It'll serve as a good foundation on which to build the rest of the application.

What We Just Did

In about a dozen lines of code, we augmented the generated code with validation:

- We ensured that required fields are present.
- We ensured that price fields are numeric and at least one cent.
- We ensured that titles are unique.
- We ensured that images match a given format.
- We updated the unit tests that Rails provided, both to conform to the constraints we've imposed on the model and to verify the new code we added.

We show this to our customer, and although she agrees that this is something an administrator could use, she says that it certainly isn't anything that she would feel comfortable turning loose on her customers. Clearly, in the next iteration we're going to have to focus a bit on the user interface.

Playtime

Here's some stuff to try on your own:

- If you're using Git, now is a good time to commit your work. You can first see which files we changed by using the git status command:

```
depot> git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes
#     in working directory)
#
# modified:   app/models/product.rb
# modified:   test/fixtures/products.yml
# modified:   test/controllers/products_controller_test.rb
# modified:   test/models/product_test.rb
# no changes added to commit (use "git add" and/or "git commit -a")
```

Since we modified only some existing files and didn't add any new ones, you can combine the git add and git commit commands and simply issue a single git commit command with the -a option:

```
depot> git commit -a -m 'Validation!'
```

With this done, you can play with abandon, secure in the knowledge that you can return to this state at any time by using a single `git checkout .` command.

- The `:length` validation option checks the length of a model attribute. Add validation to the `Product` model to check that the title is at least ten characters.
- Change the error message associated with one of your validations.