

5. Using the database in our tests

Getting ready for ActiveRecord

In this chapter we will explore how to incorporate database access in our cucumber tests. We will see how to create test data for our application and we will see how to read the database to verify results at the end of a *Scenario*. Along the way we will learn about several new ruby gems.

Let's start by taking a look at a file in the `features/support` directory. Open the file named `database.rb`. Here are some of the contents.

```
require 'active_record'
require 'logger'
require 'database_cleaner'
require 'factory_girl'
require 'pickle/world'

...

ActiveRecord::Base.establish_connection(
  :adapter => 'sqlite3',
  :database => '../depot/db/development.sqlite3')

#ActiveRecord::Base.logger = Logger.new(STDERR)
```

We are creating a connection to the depot database using *ActiveRecord*. We will use this connection to perform queries and validate data. Before we start writing tests let's spend a little time with *ActiveRecord*.

Conventions

ActiveRecord allows us to map database tables and rows to simple classes. This makes it very easy and natural to use database data in our tests.

ActiveRecord uses an approach often called *Convention Over Configuration*. What this means is that by default it assumes everything is a certain way (the convention). If things are that way then you have to do very little in order to get *ActiveRecord* to work. If things are not that way then you have a little extra work to perform (the configuration). Let's take a look at a few examples of how this works.

Let's say we have a table in the database that represents users of our system. We want to create a class named `User` that maps to this table. This is how it would look.

```
class User < ActiveRecord::Base
end
```

We are using something new here for the first time. The `<` symbol means that our `User` class *inherits* all of the functionality from `ActiveRecord::Base`. As a result our `User` class has a lot of capability and assumes a certain *convention*.

First of all *ActiveRecord* assumes that the table associated with our class is named *Users*. Can you think of a better name for a table to store a bunch of user objects? If that is the name of the table then you have to do nothing in your class. Imagine a situation where the table already existed and was not named *Users*. Let's say it was named *USR*. How can we tell *ActiveRecord* to map our `User` class to that table? Let's take a look.

```
class User < ActiveRecord::Base
  set_table_name :usr
end
```

That's all there is to it. Let's continue in our imaginary world. *ActiveRecord* assumes that there is a primary key column on the table named *id*. Again, if that column exists in the table and it is the primary key you have to do nothing. For our example let's pretend the primary key column is named *usr_id*. What do we have to do to make our class aware of this breach of convention?

```
class User < ActiveRecord::Base
  set_table_name :usr
  set_primary_key :usr_id
end
```

How simple was that? We have told our class what table to use and which column contains the primary key. What else is left to do?

ActiveRecord automatically maps columns to variables so we can access the data in rows. For example, if there is a column in our *USR* table named *fname* you can access it directly using `user.fname`. You can also perform queries on any of the columns using the column name like `User.find_by_fname`. In many cases this works fine but in some cases we have odd column names and we may want our code to read better. Let's pretend that our *USR* table has three columns named *fname*, *lname*, and *uname*. What does *ActiveRecord* provide to help us clean this up?

```
class User < ActiveRecord::Base
  set_table_name :usr
  set_primary_key :usr_id
  attribute_alias :first_name, :fname
```

```

    attribute_alias :last_name, :lname
    attribute_alias :username, :uname
end

```

We have just mapped the unclear column names to new names that help us better state their intent. Now we can also perform a query to find a `User` using the new names like `User.find_by_first_name`.

The final item to discuss in this section is how to represent relationships between tables. *ActiveRecord* has three methods to assist in this; `has_one`, `has_many`, and `belongs_to`. For the sake of our example let's say that our *USR* table has a *one-to-many* relationship with a table that holds addresses. We would represent that in our classes like this.

```

class User < ActiveRecord::Base
  has_many :addresses
  ...
end

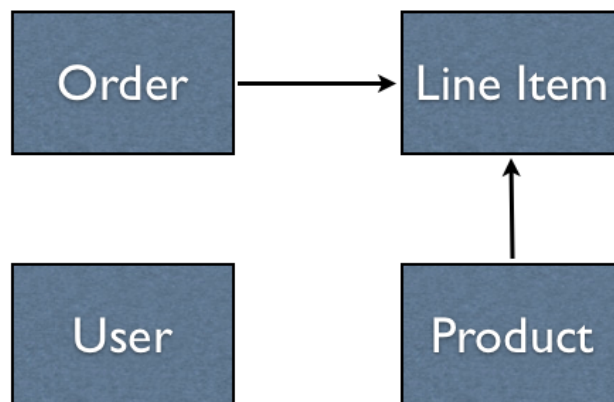
class Address < ActiveRecord::Base
  belongs_to :user
  ...
end

```

After adding the relationships we can now access the addresses belonging to a user by calling `user.addresses`. This method will return an array of `Address` objects that belong to the `User`.

The depot table structure

Let's spend a minute and take a look at the table structure for the depot application. There are four tables and they form a relationship like this:



The list of available books reside in the *Product* table. When a visitor of the site places an order the application creates an entry in the *Order* table with a entry in the *LineItem* table for each book in the shopping cart.

You are in luck with this table structure because it is in full compliance with the *ActiveRecord* conventions. With that in mind our first activity of this chapter is to create a set of classes that map to the tables above. Start by creating a new directory in the `features` directory called `database`. Create a separate file for each class and build the relationships as described in the previous section.

```
# In the file user.rb
class User < ActiveRecord::Base
end

# In the file product.rb
class Product < ActiveRecord::Base
  has_many :line_items
end

# In the file order.rb
class Order < ActiveRecord::Base
  has_many :line_items
end

# In the file line_item.rb
class LineItem < ActiveRecord::Base
  belongs_to :order
  belongs_to :product
end
```

Our first database cuke

It's time for us to take our newfound understanding of how *ActiveRecord* works and apply it to our tests. Let's create a new *Feature* file and add the following scenario.

```
Given I know how many orders I have
When I create a new order
Then I should have one additional order
```

Since there are several new concepts here I'll go ahead and jump straight into the step definitions.

```
Given /^I know how many orders I have$/ do
  @number_orders = Order.count
end
```

```

When /^I create a new order$/ do
  order = Order.new
  order.name = "cheezy"
  order.address = "123 main"
  order.email = "cheezy@example.com"
  order.pay_type = "cc"
  order.save
end

Then /^I should have one additional order$/ do
  Order.count.should == @number_orders + 1
end

```

In the *Given* and *Then* steps I use the `count` method to determine how many items are in the database. This is one of many methods that was added to our class when we inherited from `ActiveRecord::Base`. The *When* step creates a new `Order` object, populates a few of its' values, and saves it in the database.

With that behind us it's time to write a second scenario to practice using *ActiveRecord*. In this next scenario you will need to use the `delete_all` method.

```

Given I have no orders
When I create a new order
Then I should have 1 order

```

This *Scenario* reuses one of the steps from the first one. Here are the other two steps.

```

Given /^I have no orders$/ do
  Order.delete_all
end

Then /^I should have (\d+) order$/ do |num_orders|
  Order.count.should == num_orders.to_i
end

```

ActiveRecord made it very easy for us to access the database. The only step definition that has any complexity to it is the step where we are creating the `Order` object. I can imagine that this could get very complex and require a lot of code if we have a significant amount of data to setup for our tests. In the next section we'll see what can be done about this.

Default data

In chapter 4 we learned about using default data with our page objects. We learned

that default data allowed us to specify as much or little data as was necessary for our specific *Scenario*. This section is about applying the same pattern to our database access.

First of all we need to introduce a new gem - *factory_girl*. *factory_girl* allows us to create “*Factories*” for database tables. A *Factory* is nothing more than a set of default data. Let’s see what a *Factory* would look like for our `Order` class. Open up the class file and update it to contain the following.

```
class Order < ActiveRecord::Base
  has_many :line_items
end

Factory.define :order do |record|
  record.name "cheezy"
  record.address "123 main"
  record.email "cheezy@example.com"
  record.pay_type "cc"
end
```

We have just added a set of default data that can be used for our `Order` class. I would like you to add factories for each of the other *ActiveRecord* classes we created. In order to do that you will need to understand some of the table structure so the following lines has the name of the class and a list of some of it’s attributes.

Product: title, description, image_url, price
LineItem: quantity, total_price
User: name, password

Now it’s up to you to create the factories for these classes.

Cukes that use the factories

Now that we have created our *Factories* we need to know how to use them. *factory_girl* has a very simple usage model. In order to use the factory we created for `Order` we could do one of the following.

```
Factory.create(:order)
# Or
Factory.create(:order, :name => "Katie")
```

In the first example we are using all default data and writing the data to the database. In the second we are using the name passed in instead of the default name. I think we have achieved our goal of providing a simple way to emulate our page object default data with database access.

If we rewrite our step that creates the order object we can change it from

```
When /^I create a new order$/ do
  order = Order.new
  order.name = "cheezy"
  order.address = "123 main"
  order.email = "cheezy@example.com"
  order.pay_type = "cc"
  order.save
end
```

to

```
When /^I create a new order$/ do
  Factory.create(:order)
end
```

Much simpler. I think we'll stay with this.

We should know when an order shipped

The order table has a column named `shipped_at` that gets updated whenever an order ships. In this section we are going to write a *Feature* that verifies that value gets set with the proper value. Let's start with the *Scenario*.

```
Given I have 1 order
When I ship the order
Then the order should know it was shipped today
```

Note that I said nothing about the column in the database. You see, those sort of things tend to change from time to time and I do not want to couple my test to any implementation specific knowledge if I can help it.

The first step is very easy to implement using our factory. Here it is.

```
Given /^I have (\d+) order$/ do |num_orders|
  Order.delete_all
  num_orders.to_i.times do
    Factory.create(:order)
  end
end
```

The next step is a bit more complicated. We will need to open a browser, go to the correct page and ship the order we just created. We will need two page objects to

represent the two pages we visit along the way. Here's the login page object.

```
class LoginPage
  include WatirHelper

  text_field(:username, :id => 'name')
  text_field(:password, :id => 'password')
  button(:login, :value => 'Login')

  def initialize(browser)
    @browser = browser
  end

  def visit
    @browser.goto "http://localhost:3000/admin/login"
  end

  def login_with(username, password)
    self.username = username
    self.password = password
    login
  end
end
```

Here's the page object for our shipping page.

```
class ShippingPage
  include WatirHelper

  button(:ship_orders, :value => " SHIP CHECKED ITEMS ")

  def initialize(browser)
    @browser = browser
  end

  def visit
    @browser.goto "http://localhost:3000/orders"
  end

  def ship_this_many_orders(num_orders)
    num_orders.to_i.times do |index|
      @browser.checkbox(:index => index + 1).set
    end
    ship_orders
  end
end
```

And finally the step definition.

```
When /^I ship the order$/ do
  login = LoginPage.new(@browser)
  login.visit
  login.login_with('steve', 'secret')
  shipping = ShippingPage.new(@browser)
  Shipping.visit
  shipping.ship_this_many_orders(1)
end
```

There really is a lot going on in this step. I might be tempted to pull this out into an `OrderShipper` class but I'll leave that as an exercise for you to do.

The final step is to ensure that the date gets set in the table. We can do this by reading the object from the database and validating its contents.

```
Then /^the order should know it was shipped today$/ do
  order = Order.first
  order.shipped_at.should be > Date.today
  order.shipped_at.should be < Date.tomorrow
end
```

Since the `shipped_at` field holds a timestamp we have to check to see if it is greater than 12:00 A.M. today and less than 12:00 A.M. tomorrow.

Should we test the design?

There are many in the testing community that would not accept the test we created in the previous section. The fact that we created the order without going through the application would make them nervous. Their view is that you should only test “through the application”. There is a lot of validity in their view and for me it is all about understanding the tradeoffs.

If we were to only test through the application the last test would have run significantly slower. We were able to create an order in the database in less than a second. If it had to run through the user interface it would have taken many seconds. The tradeoff here was speed.

Another issue we would have experienced in the last test was that the shipped date is not visible anywhere on the page. We would have to either expose it someplace in the application so we could verify it or read the database directly. We went the simple route and read the database.

It's just a pickle

There is another gem I would like to introduce you to. The gem is called *pickle* and it adds step definitions to your application that assist in creating data. The nice thing is that it works out-of-the-box with *ActiveRecord* and *factory_girl*. In this section we'll take a look at a few ways that you might be able to leverage this gem.

Here are a few examples of *Given* steps that *pickle* provides and that will work with the *Factories* we created earlier.

```
Given a user exists
Given a user exists with name: "cheezy"
Given 10 users exist
Given the following users exist:
| name      | password |
| cheezy    | 123abc   |
| jared     | xyz555   |
```

Here are a few examples of *Then* steps that will work with *ActiveRecord* objects to validate the existence of data.

```
Then a user should exist
Then a user should exist with name: "cheezy"
Then 10 users should exist
Then the following users should exist:
| name      | password |
| cheezy    | 123abc   |
| jared     | xyz555   |
```

The beauty here is that you do not have to write the step definitions at all. This is just a small example of the steps that are provided by this gem. I suggest you spend some time learning the full capabilities of *pickle* as it will save you a lot of time when accessing databases.

Keeping the database clean

The final topic we want to discuss in this chapter is keeping our database clean. If we continue to put data into the database and never clean it up we will have bad unwanted results. We will have times when tests fail due to data that was placed in the database by previous tests. These are very hard to troubleshoot and can waste a lot of time.

The solution is to remove the data put in the database by our scenarios after each scenario runs. Fortunately there is a gem that can help us with this. It is called *databae_cleaner*. The `database.rb` file we look at earlier in this chapter has the configuration for this gem. Here it is.

```
DatabaseCleaner.strategy = :truncation, {  
  :except => %w[users product]}
```

This configuration will truncate all tables except for the *users* and *product* tables. We just have to place hooks in our configuration and it will be activated. In order for you to use it you will have to place the following in your `Before` and `After` hooks.

```
Before do  
  DatabaseCleaner.start  
end
```

```
After do  
  DatabaseCleaner.clean  
end
```