ContactManager

In this advanced Rails project, you'll create a contact manager. The tools that you will use include the following:

- Testing with RSpec [http://relishapp.com/rspec/] to drive your development
- Creating view templates with Haml [http://haml-lang.com/] and Sass [http://sass-lang.com/]
- Building reusable view code with helpers and partials
- Refactoring
- Managing authentication and authorization
- Server and client-side validations
- Deployment and monitoring

This project assumes you have already completed the general Ruby setup [http://tutorials.jumpstartlab.com/topics/environment/environment.html] and I'm using *Ruby 2.1.3*. We'll rely on the Bundler system to install other gems for us along the way.

In addition, I recommend you use the Atom IDE available here [https://atom.io/] .

We'll use an iterative approach to develop one feature at a time. Here goes!

I0: Up and Running

Let's lay the groundwork for our project. In your terminal, switch to the directory where you'd like your project to be stored.

Lets install Rails or ensure that we have it installed.

Terminal

```
$ rails -v
  Rails is not currently installed on this system.
$ gem install rails
  ...
$ rails -v
  Rails 4.1.6
```

NOTE

It is not necessary to have the same exact version of Rails specified here in the tutorial. The tutorial will definitely work with the specified version but will also likely work with a similar version.

Let's create a new Rails project:

Terminal

```
$ rails new contact_manager --skip-test-unit
$ cd contact_manager
```

NOTE

The `–skip-test-unit` option here appended to the **rails** command tells Rails not to generate a `test` directory associated with the default **Test::Unit** framework.

Open the project in your editor of choice.

Using RSpec instead of TestUnit

Rails by default uses the TestUnit [https://github.com/test-unit/test-unit] testing library. For this tutorial we instead want to use RSpec [https://github.com/rspec/rspec-rails] .

First, we need to add RSpec to the list of dependencies.

Open the `Gemfile` and add:

```
group :development, :test do
  gem 'rspec-rails'
end
```

Second, we need to install this new dependency:

Terminal

```
$ bundle install
```

Now rspec-rails is available, but we still need to do some setup to make it all work. Running this generator will perform the setup for you:

Terminal

```
$ bundle exec rails generate rspec:install
  create  .rspec
  create  spec
  create  spec/spec_helper.rb
  create  spec/rails_helper.rb
```

Now your project is set to use RSpec and the generators will use RSpec by default.

Using Unicorn instead of Webrick

Open a second terminal window, move into your project directory, then start your server with:

Terminal

```
$ bundle exec rails server
```

This will, by default, use the Webrick server which is slow as molasses. Hit `Ctrl-C` to stop it. Some of the alternative servers are mongrel [https://github.com/mongrel/mongrel] , unicorn [http://unicorn.bogomips.org/] , thin [http://code.macournoyer.com/thin/] , and puma [http://puma.io/] .

Here's how to setup unicorn.

> NOTE
>
> Unicorn will not work on Windows. You can follow the same steps below though by substituting in 'thin' for 'unicorn'.

First, add this the dependency to your `Gemfile`:

```
1  gem 'unicorn'
```

Second, we need to install this new dependency:

Terminal

```
$ bundle install
```

Now, start the server:

Terminal

```
$ bundle exec unicorn
```

Load [http://0.0.0.0:8080] http://0.0.0.0:8080 [http://0.0.0.0:8080] in your browser and you should see the Rails splash screen. Click the "About your application's environment" link and you should see all your library versions. If your database were not installed properly or inaccessible, you'd see an error here.

Git Setup

I'll assume that you've already setup Git [http://git-scm.com/] on your system.

First, we need to tell git that we want to start tracking changes for our current project.

Within your project directory initialize your project as a git repository.

Terminal

```
$ git init
```

Second, we need to save our work. We do that by adding all the files we want to save to a list. Then committing that list of files.

We add all the files in our project (e.g. `.`) to that list and then commit them:

Terminal

```
$ git add .
$ git commit -m "Generate initial project"
```

At this point if you're using GitHub [https://github.com/] , you could create a repository [https://github.com/new] , add that remote and push to it. For purposes of this tutorial, we'll just manage the code locally.

Shipping to Heroku

We want to host our Rails application on the internet using the popular Heroku [http://www.heroku.com/] service.

If you don't already have one, you'll need to create a Heroku account [https://id.heroku.com/signup/www-header] . After creating your account download and install the Heroku Toolbelt [https://toolbelt.heroku.com/] .

Heroku requires applications to use a PostgresSQL database and not a Sqlite database. So we need to update our application to use the PostgresSQL gem (named **pg**). Along with installing the **pg** gem we will also need to install and configure a PostgreSQL database. This could be trivial amount of work or may consume an entire afternoon.

An ideal situation is if we could continue to use Sqlite locally and PostgresSQL only on Heroku. This configuration is indeed possible.

Our application, by default is run in **development** mode.

When we run our tests the application runs in **test** mode.

When we deploy to Heroku, our application is run in **production** mode.

We want to continue to use Sqlite while in **development** and **test**. We will use PostgresSQL in **production**.

Find the line in your gem file that says `gem 'sqlite3'` and move this into the `:development, :test` group:

```
1  group :development, :test do
2    gem 'rspec-rails'
3    gem 'sqlite3'
4  end
```

Create a new group called :production, and add the PostgreSQL gem to it.

```
group :production do
  gem 'pg'
end
```

Run the `bundle install` command again to update your database dependencies.

Commit your code again.

Terminal

```
  git add .
$ git commit -m "Update database dependencies"
```

Now, let's update `production.rb` to serve static assets.

```
config.serve_static_assets = true
```

Without the line above, you may have difficulty deleting once deployed to Heroku.

Next let's integrate Heroku [http://www.heroku.com/] .

Terminal

```
$ heroku create
```

The toolbelt will ask for your username and password the first time you run the create, but after that you'll be using an SSH key for authentication.

After running the create command, you'll get back the URL where the app is accessible. Try loading the URL in your browser and you'll see the generic Heroku splash screen. It's not running your code yet so push your project to Heroku.

Terminal

```
$ git push heroku master
-----> Launching... done
http://ADJECTIVE-WORD-NUMBER.heroku.com deployed to Heroku
```

If you see a `Permission denied (publickey).` error, your SSH key needs to be uploaded to Heroku.

Terminal

```
heroku keys:add ~/.ssh/id_rsa.pub
```

```
Uploading SSH public key ...
```

You should now be able to deploy.

Terminal

```
$ git push heroku master
-----> Launching... done
http://ADJECTIVE-WORD-NUMBER.heroku.com deployed to Heroku
```

Refresh your browser and you should see an error that says `The page you were looking for doesn't exist.`. Don't worry, everything should be running fine.

Now we're ready to actually build our app!

I1: Building People

We're building a contact manager, so let's start with modeling people.

> NOTE
>
> Since this is an advanced tutorial we won't slog through the details of implementing a Person model, controller, and views. Instead we'll take advantage of scaffolding tools.

A Feature Branch

But first, let's make a feature branch in git:

Terminal

```
$ git checkout -b implement-people
```

Now all our changes will be made on the **implement-people** branch. As we finish the iteration we'll merge the changes back into master and ship it.

Scaffold

Let's use the default rails generators to generate a scaffolded model named `Person` that has a `first_name` and `last_name`:

Terminal

```
$ bundle exec rails generate scaffold Person first_name:strin
$ bundle exec rake db:migrate
```

The generators created test-related files for us. They saw that we're using RSpec and created corresponding controller and model test files. Let's run those tests now:

Terminal

```
$ bundle exec rspec
```

There should be 0 failing tests and 17 pending.

This is a great time to add and commit your changes.

Terminal

```
$ git add .
$ git commit -m "Generated Person model"
```

Open your browser to http://localhost:8080/people [http://localhost:8080/people] and try creating a few sample people.

Starting with Testing

When you ran your tests with `bundle exec rspec` you probably got a several pending tests:

Terminal

```
$ bundle exec rspec
  **.***************.............
  Pending:
  _Results Omitted_
  Finished in 0.25264 seconds (files took 2.51 seconds to loa
  30 examples, 0 failures, 17 pending
```

We're not going to be using the `PeopleHelper` so let's just get rid of the test file:

Terminal

```
$ git rm spec/helpers/people_helper_spec.rb
```

Commit your changes:

Terminal

```
$ git commit -m "Delete extraneous spec file"
```

Let's move on to the controller and get those tests to pass. The first part of my output currently looks like this:

Terminal

```
$ bundle exec rspec
**.**************.............
Pending:
PeopleController GET index assigns all people as @people
# Add a hash of attributes valid for your model
# ./spec/controllers/people_controller_spec.rb:40
PeopleController GET show assigns the requested person as @
# Add a hash of attributes valid for your model
# ./spec/controllers/people_controller_spec.rb:48
PeopleController GET edit assigns the requested person as @
# Add a hash of attributes valid for your model
# ./spec/controllers/people_controller_spec.rb:63
_REMAINING OUTPUT OMITTED_
```

You'll notice that each of these pending examples repeats the message `Add a hash of attributes valid for your model`. Let's open `spec/controllers/people_controller_spec.rb` and search for that line of text. You should see code that looks like this:

```
1    let(:valid_attributes) {
2      skip("Add a hash of attributes valid for your model")
3    }
```

The `skip` makes it so the tests that call `valid_attributes` are marked as pending. The parameter that is passed is the message that should be displayed.

Let's update the code to provide valid attributes. These are going to be the minimum fields required to create a new Person record. In our case, we want a person to have a valid `first_name` and `last_name`. Let's update the code to look like this:

```
1    let(:valid_attributes) {
2      { first_name: 'Jane', last_name: 'Doe' }
3    }
```

Go back to your terminal and run your tests.

Terminal

```
$ bundle exec rspec
.......***..**..*.............
Pending:
PeopleController POST create with invalid params assigns a
# Add a hash of attributes invalid for your model
```

```
# ./spec/controllers/people_controller_spec.rb:91
PeopleController POST create with invalid params re-renders
# Add a hash of attributes invalid for your model
# ./spec/controllers/people_controller_spec.rb:96
PeopleController PUT update with valid params updates the r
# Add a hash of attributes valid for your model
# ./spec/controllers/people_controller_spec.rb:109
PeopleController PUT update with invalid params assigns the
# Add a hash of attributes invalid for your model
# ./spec/controllers/people_controller_spec.rb:130
PeopleController PUT update with invalid params re-renders
# Add a hash of attributes invalid for your model
# ./spec/controllers/people_controller_spec.rb:136
Person add some examples to (or delete) /Users/jmejia/code/
# Not yet implemented
# ./spec/models/person_spec.rb:4
Finished in 0.34279 seconds (files took 2.65 seconds to loa
29 examples, 0 failures, 6 pending
```

Prior to that change we had 16 pending tests. Now we have 6. That simple change confirms that we are able to create new `People` with both a `first_name` and a `last_name`.

Let's commit this change.

Terminal

```
$ git add .
$ git commit -m "Added valid attributes to people controller
```

Looking back at your test output (run `bundle exec rspec` if you need to), you can see the repeated message `Add a hash of attributes invalid for your model`. Open `spec/controllers/people_controller_spec.rb` and search for that line. You should see the following code:

```
1    let(:invalid_attributes) {
2      skip("Add a hash of attributes invalid for your model")
3    }
```

Here is where we need to specify what invalid attributes should look like. In our case, we don't want to be able to create a user that is missing a `first_name` or `last_name`. Let's update this code to look like this:

```
1    let(:invalid_attributes) {
2      { first_name: nil, last_name: nil }
3    }
```

Now when we run `bundle exec rspec` the end of your output should look like this:

Terminal

```
Finished in 0.34863 seconds (files took 2.2 seconds to load
29 examples, 3 failures, 2 pending
Failed examples:
rspec ./spec/controllers/people_controller_spec.rb:91 # Peo
rspec ./spec/controllers/people_controller_spec.rb:96 # Peo
rspec ./spec/controllers/people_controller_spec.rb:136 # Pe
```

We have 3 failures because we said `nil` values for first and last names should not be valid. As it stands, we haven't written any code to make empty strings invalid.

This is a good point to discuss models. The model is the application's representation of the data layer, the foundation of any functionality. In the same way we'll build low-level tests on the models which will be the foundation of our test suite. By focusing on the model level we will actually get the failing controller specs to pass.

To start, let's run *only* `person_spec.rb` which tests the `Person` model. Up until now we have been running the entire test suite.

Terminal

```
$ bundle exec rspec spec/models/person_spec.rb
*
Pending:
Person add some examples to (or delete) /Users/jmejia/code/
# Not yet implemented
# ./spec/models/person_spec.rb:4
Finished in 0.00067 seconds (files took 2.78 seconds to loa
1 example, 0 failures, 1 pending
```

We have 1 pending test and a message that says `Not yet implemented`. Let's write a useful test.

Open `spec/models/person_spec.rb` and you'll see this:

```
1  require 'rails_helper'
2
3  RSpec.describe Person, :type => :model do
```

```
4     pending "add some examples to (or delete) #{__FILE__}"
5   end
```

The `describe` block will wrap all of our tests (also called examples) in RSpec parlance.

Testing for Data Presence

Let's create an example using the `it` method to test that a `Person` without a `first_name` is invalid:

```
1   require 'rails_helper'
2
3   RSpec.describe Person, :type => :model do
4     it 'is invalid without a first name' do
5       person = Person.new(first_name: nil)
6       expect(person).not_to be_valid
7     end
8   end
```

Run your test again:

Terminal

```
$ bundle exec rspec spec/models/person_spec.rb
F
Failures:
1) Person is invalid without a first name
Failure/Error: expect(person).not_to be_valid
expected # not to be valid
# ./spec/models/person_spec.rb:6:in `block (2 levels) in '
Finished in 0.01008 seconds (files took 2.24 seconds to loa
1 example, 1 failure
Failed examples:
rspec ./spec/models/person_spec.rb:4 # Person is invalid wi
```

The test failed because it expected a person with no first name to be invalid, but instead it *was* valid. We can fix that by adding a validation for first name inside the model:

```
1   class Person < ActiveRecord::Base
2     validates :first_name, presence: true
3   end
```

If you run the test again you'll see that we now have a passing test.

Terminal

```
$ bundle exec rspec spec/models/person_spec.rb
.
Finished in 0.01684 seconds (files took 2.45 seconds to loa
1 example, 0 failures
```

Looking good! Let's also require people to have last names.

Create a test that asserts that a `Person` without a last name is invalid:

```
1  it 'is invalid without a last name' do
2    person = Person.new(first_name: 'Bob', last_name: nil)
3    expect(person).not_to be_valid
4  end
```

Run `bundle exec rspec spec/models/person_spec.rb`. The test you just wrote should be failing.

Fix the test by adding a validation to the model:

```
1  class Person < ActiveRecord::Base
2    validates :first_name, :last_name, presence: true
3  end
```

Run `bundle exec rspec spec/models/person_spec.rb` and you should see 2 passing tests.

And now if you run the entire suite, the controller tests that were failing should be passing.

Terminal

```
$ bundle exec rspec
.........*.....................
Pending:
PeopleController PUT update with valid params updates the r
# Add a hash of attributes valid for your model
# ./spec/controllers/people_controller_spec.rb:109
Finished in 0.35373 seconds (files took 2.5 seconds to load
30 examples, 0 failures, 1 pending
```

Very nice! We have one last pending test. Open `spec/controllers/people_controller_spec.rb` and find the line containing `Add a hash of attributes valid for your model`. The code should look like this:

```
describe "PUT update" do
  describe "with valid params" do
    let(:new_attributes) {
      skip("Add a hash of attributes valid for your model")
    }

    it "updates the requested person" do
      person = Person.create! valid_attributes
      put :update, {:id => person.to_param, :person => new_attribute
      person.reload
      skip("Add assertions for updated state")
    end

    # remaining code omitted
  end
end
```

This test is checking to make sure we can update the attributes of a person through the controller. To get this to pass we first need to provide valid attributes to update the person. Update the `new_attributes` to look like this:

```
let(:new_attributes) {
  {first_name: 'NewFirstName', last_name: 'NewLastName'}
}
```

Next, we need to update the test to check that the new values have persisted.

```
it "updates the requested person" do
  person = Person.create! valid_attributes
  put :update, {:id => person.to_param, :person => new_attributes}, 
  person.reload
  expect(person.first_name).to eq('NewFirstName')
  expect(person.last_name).to eq('NewLastName')
end
```

Run your test suite and if everything is passing, commit your changes:

Terminal

```
$ git add .
$ git commit -m "Implement validations on person"
```

Experimenting with Our Tests

Go into the `person.rb` model and temporarily remove `:first_name` from the `validates` line. Run your tests. What happened?

This is what's called a false positive. The `is invalid without a first_name` test is passing, but not for the right reason. Even though we're not validating `first_name`, the test is passing because the model it's building doesn't have a valid `last_name` either. That causes the validation to fail and our test to pass. We need to improve the Person object created in the tests so that only the attribute being tested is invalid. Let's refactor.

First, just below the `describe` line of our `person_spec`, let's add this code which will be available to all of our examples:

```
let(:person) do
  Person.new(first_name: 'Alice', last_name: 'Smith')
end
```

Update your first name and last name tests to use the person object defined in the `let`.

```
it 'is invalid without a first name' do
  person.first_name = nil
  expect(person).to_not be_valid
end

it 'is invalid without a last name' do
  person.last_name = nil
  expect(person).to_not be_valid
end
```

Run your tests and now the `is invalid without a first name` test should fail. Read the output that RSpec gives you to help find the problem. In this case, it's easy – just add `:first_name` back where you removed it from the validation in `person.rb`.

Now that everything is passing, commit your changes.

Terminal

```
$ git add .
$ git commit -m "Fix false positive in person specs"
```

Checking the Checkers

The `let` clause we wrote will make writing test examples a lot easier, but we had better have a test to ensure that what we're calling `person` is actually valid! You can do this by adding a test:

```
it 'is valid' do
  expect(person).to be_valid
end
```

Run the tests. If they're passing (and they should be) commit your changes.

Terminal

```
$ git add .
$ git commit -m "Confirm person is valid in person spec"
```

Ship It

This branch is done. Let's go back to the master branch and merge it in:

Terminal

```
$ git checkout master
$ git merge implement-people
```

Now it's ready to send to Heroku and run our migrations:

Terminal

```
$ git push heroku master
$ heroku run rake db:migrate
```

Open up your production app in your browser and visit `/people`. You should be able to create sample people like you did on your development server.

I2: Phone Numbers

A Feature Branch

Let's again make a feature branch in git:

Terminal

```
$ git checkout -b implement-phone-numbers
```

Now all our changes will be made on the **implement-phone-numbers** branch. As we finish the iteration we'll merge the changes back into master and ship it.

Modeling The Objects

First, let's think about the data relationship. A person is going to have multiple phone numbers, and a phone number is going to belong to one person. In the database world, this is a one-to-many relationship, one person has many phone numbers.

One-to-Many Relationship

The way this is traditionally implemented in a relational database is that the "many" table (the phone number table, in this case) holds a unique identifier, called a foreign key, pointing back to the row from the "one" (person) table that it belongs to. For example, we might have a person with ID 6. That person would have phone numbers that would each have a foreign key `person_id` with the value 6.

A Person Has Phone Numbers

With that understanding, let's write a test. We just want to check that a person is capable of having phone numbers. In your `person_spec.rb` let's add this test:

```
1  it 'has an array of phone numbers' do
2    expect(person.phone_numbers).to eq([])
3  end
```

Run the tests and make sure the test fails with `undefined method 'phone_numbers'`. Now we're ready to create a `PhoneNumber` model and corresponding association in the `Person` model.

Scaffolding the Phone Number Model

We'll use the scaffold generator again to save us a little time. For now we'll keep the phone number simple that contains the phone number, represented as a String, and a reference to the Person that owns the number. Generate it with this command at the terminal:

Terminal

```
$ bundle exec rails generate scaffold PhoneNumber number:stri
```

Run the migrations:

Terminal

```
$ bundle exec rake db:migrate
```

Run the tests again. The failing test is still failing and there are now 17 pending tests. Let's stay focused on the failure.

Setting Relationships

Next open the `person.rb` model and add the following association:

```
1  has_many :phone_numbers
```

Run the tests and you should have no failing tests.

We have pending tests in the `phone_number_spec.rb`, `phone_numbers_controller_spec.rb` and `phone_numbers_helper_spec.rb`.

If your tests are all passing or pending, commit all your changes:

Terminal

```
$ git add .
$ git commit -m "Generate phone number and associate with per
```

You can try out the new association by going into the console via `bundle exec rails console` and adding a phone number manually:

IRB

```
2.1.1 :001>  p = Person.first
2.1.1 :002>  p.phone_numbers.create(number: '2024605555')
```

Validating Phone Numbers

Right now the phone number is just stored as a string, so maybe the user enters a good-looking one like "2024600772" or maybe they enter "please-don't-call-me". Let's add some validations to make sure the phone number can't be blank.

Go into `phone_number_spec.rb` and mimic some of the same things we did in `person_spec.rb`. We can start off by writing a `let` block to setup our `PhoneNumber` object. Enter this just below the `describe` line:

```
1   let(:phone_number) { PhoneNumber.new }
```

Delete the `pending` test, and add a test for a valid number:

```
1   it 'is valid' do
2     expect(phone_number).to be_valid
3   end
```

Run your tests. They should be passing.

We want to start working on valid formats for a phone number, so let's write a test that states that a phone number cannot be blank:

```
1   it 'is invalid without a number' do
2     phone_number.number = nil
3     expect(phone_number).to_not be_valid
4   end
```

If you run your tests, this test should be the only failing test.

Go into the `phone_number.rb` model and add a validation checking the existence of the `number`, run your tests again. This test passes, but now the first one is failing.

Update the `let` block:

```
1   let(:phone_number) { PhoneNumber.new(number: "1112223333") }
```

Make sure your tests are green, and then commit your changes.

A `PhoneNumber` shouldn't be allowed to float out in space, so let's require that it be attached to a `Person`:

```
it 'must have a reference to a person' do
  phone_number.person_id = nil
  expect(phone_number).not_to be_valid
end
```

Run the tests again and your new test should fail. Add a validation that checks the presence of `person_id` in your phone number, then run your tests again.

That got the new test to pass but broke another test. This is the spec for the valid phone number:

```
Failures:

  1) PhoneNumber is valid
     Failure/Error: expect(phone_number).to be_valid
       expected #<PhoneNumber id: nil, number: "1112223333", person_
     # ./spec/models/phone_number_spec.rb:7:in `block (2 levels) in
```

Update your `let` block in the `spec/models/phone_number_spec.rb` again, giving it a `person_id`.

```
let(:phone_number) { PhoneNumber.new(number: "1112223333", person_id
```

Now let's get the pending controller tests to pass.

Open up the file `spec/controllers/phone_numbers_controller_spec.rb` and find the method definition for `valid_attributes`. In order to pass our validations we need to add `number` and `person_id` attributes.

```
let(:valid_attributes) {
  { number: "MyString", person_id: 1 }
}
```

Re-run your tests and you should be down to 6 pending tests.

Now add invalid attributes.

```
  let(:invalid_attributes) {
    { number: nil, person_id: nil }
  }
```

Re-run your tests and there should be 2 pending tests. Let's write a passing test for the last pending controller spec. Within `spec/controllers/phone_numbers_controller_spec.rb` find the line containing `skip("Add a hash of attributes valid for your model")`. Update the `:new_attributes` let block and the test just below it:

```
let(:new_attributes) {
  {number: 'MyNewString', person_id: 2}
```

```
  }

  it "updates the requested phone_number" do
    phone_number = PhoneNumber.create! valid_attributes
    put :update, {:id => phone_number.to_param, :phone_number => new_a
    phone_number.reload
    expect(phone_number.number).to eq('MyNewString')
    expect(phone_number.person_id).to eq(2)
  end
```

Run your tests and there should be 1 pending and 0 failing.

That's a lot of work for two validations, but these are an important part of our testing base. If somehow one of the validations got deleted accidentally, we'd know it right away.

If your tests are all passing, go ahead and commit the changes so you don't lose all that hard work!

We're not going to be using the `PhoneNumberHelper` so let's get rid of the test file:

Terminal

```
$ git rm spec/helpers/phone_numbers_helper_spec.rb
```

Commit your changes:

Terminal

```
$ git commit -m "Delete extraneous spec file"
```

All tests should be passing.

Finally, let's connect the phone number to a person.

Within `phone_number_spec.rb`, write a test ensuring that a `PhoneNumber` has a method to give you back the associated `Person` object.

```
  it 'is associated with a person' do
    expect(phone_number).to respond_to(:person)
  end
```

Run your test. Notice the failure. That is because the relationship from Phone Numbers and a Person has not been established.

Open `phone_number.rb` and add the following association:

```
  belongs_to :person
```

Commit your changes.

Creating some seed data

Go into your console (`bundle exec rails console`) and create a person and a couple of phone numbers:

IRB

```
2.1.1 :001> person = Person.create(first_name: 'Alice', last_na
2.1.1 :002> person.phone_numbers.create(number: '555-1234')
2.1.1 :003> person.phone_numbers.create(number: '555-9876')
```

Then get the person ID for that person:

IRB

```
2.1.1 :001> person.id
```

Building a Web-based Workflow

Up to this point we've created phone numbers through the console. Let's build up a web-based workflow.

What You Got From Scaffolding

When the scaffold generator ran it gave us a controller and some view templates. Check out the *new* form by loading up /phone_numbers/new [http://localhost:8080/phone_numbers/new] in your browser.

It's not that bad, but it's not good enough.

Person-centric Workflow

Here's what the customer wants:

"When I am looking at a single person's page, I click an add link that takes me to the page where I enter the phone number. I click save, then I see the person and their updated information"

Go to the show page for the person you just created in the console. If that person's ID is `1`, then the url is `/people/1`.

The show page doesn't even list the existing phone numbers.

Let's add that.

How do you test views?

We don't want to lose the value of testing, so we need a way to test the person's show view. We want to see that the rendered HTML lists the phone numbers.

When we want to test the output HTML we're talking about an *integration test*. My favorite way to build those is using the Capybara gem with RSpec. Let's get Capybara setup by opening your `Gemfile` and adding this line inside the `:development` group:

```
gem 'capybara'
```

Then run `bundle` from your command line and it'll install the gem.

Create a new folder named `spec/features`. In that folder let's make a file named `person_view_spec.rb`. Then here's how I'd write the examples:

```
require 'rails_helper'

describe 'the person view', type: :feature do

  let(:person) { Person.create(first_name: 'John', last_name: 'Doe')

  before(:each) do
    person.phone_numbers.create(number: "555-1234")
    person.phone_numbers.create(number: "555-5678")
    visit person_path(person)
  end

  it 'shows the phone numbers' do
    person.phone_numbers.each do |phone|
      expect(page).to have_content(phone.number)
    end
  end

end
```

Run your tests. They should fail.

Open up `app/views/people/show.html.erb` and add this code above the edit link:

```
<ul>
  <% @person.phone_numbers.each do |phone| %>
    <li><%= phone.number %></li>
  <% end %>
</ul>
```

Re-run the tests, and they should pass.

The customer wants to be able to add new phone numbers, so let's write a test for that.

```
it 'has a link to add a new phone number' do
  expect(page).to have_link('Add phone number', href: new_phone_numb
end
```

The test will fail. Go back to the `show` view, and add a link to add a new phone number:

```
1  <%= link_to 'Add phone number', new_phone_number_path %> |
```

Now the test should pass.

Ok, so now we need to be able to actually add a phone number. Let's write a test that checks that when we follow the 'Add phone number' link and add a new phone number we end up back on the show page for the person, and that number is in the page.

```
1  it 'adds a new phone number' do
2    page.click_link('Add phone number')
3    page.fill_in('Number', with: '555-8888')
4    page.click_button('Create Phone number')
5    expect(current_path).to eq(person_path(person))
6    expect(page).to have_content('555-8888')
7  end
```

This fails because we've been redirected to the wrong location. We need to step down one level. Let's mark this test pending for now by adding an `x` in front of `it`:

```
1    xit 'adds a new phone number' do
2      # ...
3    end
```

Open up the `spec/controllers/phone_numbers_controller_spec.rb` and find the test called `it "redirects to the created phone_number"`.

This is not the behavior we are looking for. Let's change the expectation:

```
1  it "redirects to the phone number's person" do
2    alice = Person.create(first_name: 'Alice', last_name: 'Smith')
3    valid_attributes = {number: '555-8888', person_id: alice.id}
4    post :create, {:phone_number => valid_attributes}, valid_session
5    expect(response).to redirect_to(alice)
6  end
```

Run `bundle exec rspec spec/controllers/phone_numbers_controller_spec.rb`, and this test now fails.

Open up `app/controllers/phone_numbers_controller.rb` and look at the `create` action. When the phone number successfully saves, redirect to the phone number's attached person `redirect_to @phone_number.person`.

Re-run your tests, and this test passes, but now two other tests are failing!

Both are failing for the same reason:

```
1    ActionController::ActionControllerError:
2      Cannot redirect to nil!
```

```
3     # ./app/controllers/phone_numbers_controller.rb:47:in `block (2 le
```

In the create method of the phone numbers controller, we're trying to redirect to nil. That makes sense, since the phone number we created is using the default valid attributes, which lists the `person_id` as `1`. When the code asks the database for the person with id `1`, the database can't find a match, and we end up with a `nil`. That's not going to work, so let's give those two tests a real person to work with.

We can use the person object we just created for our previous test. Promote `alice` to a `let`. We want it to be valid for all of the `create` specs, so we need to place that line of code inside the describe for the `create` `with valid params`:

```
1  describe "POST create" do
2    describe "with valid params" do
3
4      let(:alice) { Person.create(first_name: 'Alice', last_name: 'Smi
```

We need to do the same with the `valid_attributes` local variable. Put it right below the `let(:alice)`.

```
1  let(:valid_attributes) { {number: '555-1234', person_id: alice.id} }
```

Run the controller tests again, and this time, they should pass.

Delete the `pending` declaration in your integration test and run all the tests. It is still failing. Let's take a look at what's going on here.

Open up your application and go to /people/1 [http://localhost:8080/people/1] . Click to add a phone number, and now click the `Create Phone number` button.

You get an error message. The form includes a field for providing the `person_id`, but our test is not filling it in. Without a `person_id`, the number can't be saved (remember, we made `person_id` a required attribute for phone numbers).

We could require our users to manually fill in the ID for the person they are creating a phone number for, but this seems tedious. Since the user has just come from the person page, we should be able to fill in the `person_id` for them automatically. One way to do this is by encoding it into the URL as a query parameter.

Make the test pending again, and go back to the one above it. Change it so that the phone number path takes a person's id.

```
1  it 'has a link to add a new phone number' do
2    expect(page).to have_link('Add phone number', href: new_phone_numb
3  end
```

Run the test and see it fail. Then open the `show` view and change the link:

```
1  <%= link_to 'Add phone number', new_phone_number_path(person_id: @pe
```

The test now passes.

Go back to the feature and remove the `pending` declaration again.

Re-run the tests.

Still failing! We are passing the `person_id` to the `new_phone_number` route, but the form doesn't take it into account when creating the number.

Go into the phone numbers controller, into the `new` action, and instead of `@phone_number = PhoneNumber.new` let's say `@phone_number = PhoneNumber.new(person_id: params[:person_id])`

Run the tests again, and finally they pass!

If you go to the /people/1 [http://localhost:3000/people/1] page and click to create a new phone number, you'll see that we are exposing the field for the person id to the user. That's unnecessary.

Open up the `app/views/phone_numbers/_form.html.erb` file and change the `number_field` to be a `hidden_field`. Go ahead and delete the label for the person id as well.

Now all your tests should be passing, and you have a svelte phone number form to boot, so this is a good time to commit your changes.

Workflow for Editing Phone Numbers

OK, we have the functionality to add a number, let's make it possible to edit phone numbers.

In `spec/features/person_view_spec.rb` add a test:

```
1  it 'has links to edit phone numbers' do
2    person.phone_numbers.each do |phone|
3      expect(page).to have_link('edit', href: edit_phone_number_path(p
4    end
5  end
```

Run the tests, and they'll fail. Open up the `people/show` template, and change the phone number list item:

```
1  <li><%= phone.number %> <%= link_to('edit', edit_phone_number_path(p
```

The tests pass.

Let's make sure that the edit workflow works the way the customer expects it to.

```
 it 'edits a phone number' do
   phone = person.phone_numbers.first
   old_number = phone.number

   first(:link, 'edit').click
   page.fill_in('Number', with: '555-9191')
   page.click_button('Update Phone number')
   expect(current_path).to eq(person_path(person))
   expect(page).to have_content('555-9191')
   expect(page).to_not have_content(old_number)
 end
```

The test is failing because it's redirecting to the wrong place. Make the test pending while we go update the controller test.

In `spec/controllers/phone_numbers_controller_spec.rb` find the test `it 'redirects to the phone number'`. Create a person `bob` who has a phone number, and make sure the test expects to redirect to `bob` rather than the phone number.

```
it "redirects to the phone_number" do
  bob = Person.create(first_name: 'Bob', last_name: 'Jones')
  valid_attributes = {number: '555-5678', person_id: bob.id}
  phone_number = PhoneNumber.create! valid_attributes
  put :update, {:id => phone_number.to_param, :phone_number => valid
  expect(response).to redirect_to(bob)
end
```

Run the tests, and this test will fail.

Find the `def update` action in the corresponding controller, and change it to redirect to `@phone_number.person`.

Re-run the tests, and now the test we just wrote should pass, but a couple of other tests are failing with the familiar `Cannot redirect to nil!` error.

Promote `bob` and his `valid_attributes` to a `let` in the describe block for `with valid params` within the `PUT update` describe block. We'll also update `:new_attributes` to use `bob.id` so it has a valid person to redirect to.

```
describe "PUT update" do
  describe "with valid params" do

  let(:bob) { Person.create(first_name: 'Bob', last_name: 'Jones') }
  let(:valid_attributes) { {number: '555-5678', person_id: bob.id} }
  let(:new_attributes) { {number: 'MyNewString', person_id: bob.id}
```

Run the tests, and they should now be passing.

Go back to the `person_view_spec.rb` feature, remove the `pending` declaration, and rerun the tests.

Everything should be passing. Go ahead and commit your changes.

Destroying Phone Numbers

Lastly the customer wants a delete link for each phone number. Follow a similar process to...

- Write a test that looks for a delete link for each phone number
- Modify the partial to have that link
- Try it in your browser and destroy a phone number
- Update the expectation in the controller spec to specify that you get redirected to the phone number's person after destroy
- Fix the controller to redirect to the phone number's person after destroy
- Fix the resulting spec failure in the controller spec

One note: the "destroy" action of a rails controller is triggered by sending an HTTP DELETE request to the appropriate path. You will need to use the `method` option on the `link_to` helper to include this in the delete links. It will look something like:

```
1  <%= link_to('delete', phone_number_path(phone_number), :method => "d
```

Once your tests are passing, let's commit!

Terminal

```
$ git add .
$ git commit -m "Finish implementing phone number functionali
```

If that was all too easy try this *challenge*:

Write an integration test that destroys one of the phone numbers then ensure's that it's really gone from the database. You'll need to use Capybara features like...

- `page.click_link` to activate the destroy link
- `expect(current_path).to ==` to ensure you arrive at the show page
- then check that the object is gone (one idea: verify that there is *no* delete link

Phone Numbers are Done...For Now!

Wow, that was a lot of work, right? Just to list some phone numbers? Test-Driven Development (TDD) is really slow when you first get started, but after a few years you'll have the hang of it! I wish I were joking.

Let's Ship

Hop over to your command prompt and let's work with git. First, ensure that everything is committed on our branch:

Terminal

```
$ git status
```

This branch is done. Let's go back to the master branch and merge it in:

Terminal

```
$ git checkout master
$ git merge implement-phone-numbers
```

Now it's ready to send to Heroku and run our migrations:

Terminal

```
$ git push heroku master
$ heroku run rake db:migrate
```

Open up your production app in your browser and it should be rockin'!

I3: Email Addresses

What good is a contact manager that doesn't track email addresses? We can take most of the ideas from `PhoneNumber` and apply them to `EmailAddress`. This iteration is going to be largely independent because you've seen it all before.

Start a Feature Branch

Let's practice good source control and start a feature branch:

Terminal

```
$ git checkout -b implement-email-addresses
```

Now we're ready to work!

Writing a Test: A Contact Has Many Email Addresses

In your `person_spec.rb` refer to the existing example "has an array of phone numbers" and create a similar example for email addresses. Verify that the test fails when you run `bundle exec rspec`.

Creating the Model

Use the `scaffold` generator to scaffold a model named `EmailAddress` which has a string field named `address` and an integer field named `person_id`.

If you got your `rails generate` command messed up, go to your terminal window and hit the arrow-up key to get the command that was wrong, and then change `rails generate` to `rails destroy`. The files previously generated will be removed.

Run `bundle exec rake db:migrate db:test:prepare` then ensure that your test still isn't passing with `bundle exec rspec`.

Setting Relationships

Open the `Person` model and declare a `has_many` relationship for `email_addresses`.
Open the `EmailAddress` model and declare a `belongs_to` relationship with `Person`.

Now run your tests. If you have 0 failures, and several pending specs *commit your changes*.

Adding Model Tests and Validations for Email Addresses

Let's add some quality controls to our `EmailAddress` model.

- Open the `email_address_spec.rb`
- Delete the pending example
- Add a `let` block named `email_address` that returns `EmailAddress.new`
- Add an example `it 'is valid'` to check that the `email_address` in the `let` block is valid
- Look at the example "is invalid without a first_name" in `person_spec.rb` and create a similar example in `email_address_spec` which makes sure an `EmailAddress` is not valid without an address
- Run your tests and make sure it *fails*
- Add a validation to ensure that the `address` attribute `EmailAddress` is present
- Run your tests and see that your `it 'is valid'` test fails.
- Update the `let` block giving your `EmailAddress` and `address`.
- Run the tests and see that they pass.
- Update the controller spec and get the pending tests to pass.
- Commit.

- Write a test to check that an `EmailAddress` isn't valid unless it has a `person_id`

- See it fail.

- Update the model to validate the presence of that field.

- Run the tests, and see the `it 'is valid'` test fail as well as several controller specs.

- Update the `valid_attributes` method in the controller specs.

- Update the `let` block in the email address spec.

- Run your tests and make sure it *passes*

If you're green, go ahead and check in those changes.

Completing Email Addresses

Now let's shift over to the integration tests.

Displaying Email Addresses

Before you play with displaying email addresses, create a few of them manually in the console.

- Open the `person_view_spec.rb`
- Wrap all the existing tests, including the `before` block in a `describe` block for phone numbers.
- Create a new `describe` for the email addresses.
- Create a `before` block within the new `describe` block that adds a couple of email addresses to the person and visits the person page.
- Write a test that looks for LIs for each address. Try using this:

```
1  expect(page).to have_selector('li', text: 'SOME_EMAIL_ADDRESS')
```

- Make sure the test *fails*.
- Add a UL to the person's `show` template that renders a list of `email_addresses`.

Tests should be green here, so check in your changes. Then continue...

Create Email Address Link

- Write a test named `"has an add email address link"` that looks for a link with ID `new_email_address`, clicks it, and verifies that it goes to the `new_email_address_path`
- Verify that it *fails*
- Add the link to the `show` page
- Verify that it *passes*

If everything passes, check in your changes.

Email Address Creation Workflow

- Write a test that clicks the link with the id `new_email_address`, fills in the form with an email address, clicks the submit button, and expects to be back on the person page with the new email address listed.
- Make sure the test is failing.
- Make the spec pending while we drop into a lower level and fix this
- Open up the email addresses controller specs
- change the `it "redirects to the created email_address"` spec so that it redirects to the email address's person. You'll need to create the related person.
- see it fail
- fix the controller
- see the other specs fail
- fix the specs
- go back to the `person_view_spec` and remove the pending declaration
- run the tests and see that it is still failing. When trying to create an email address, the form is failing to submit, because we haven't connected it to a person.
- Make the failing spec pending
- Update the test in `person_view_spec` that has the link to add new email addresses, and make sure that it expects the `current_url` to be the url containing the

`person_id`.
- Remove the pending declaration on the test that was failing, and see that it is *still* failing.
- Go to the controller and pass the `person_id` to the new `EmailAddress`
- Finally, the test passes.
- Go ahead and hide the `person_id` in the form.

Commit your changes.

- now do the same for the update and destroy actions as well

When you're green, check in your changes.

Ship it!

Let's ship this feature:

- Switch back to your master branch with `git checkout master`
- Merge in the feature branch with `git merge implement-email-addresses`
- Throw it on Heroku with `git push heroku master`
- Run your migrations with `heroku run rake db:migrate`

I4: Tracking Companies

Our app can track people just fine, but what about companies? What's the difference between a company and person? The main one, for now, is that a person has a first name and last name, while a company will just have one name.

Thinking about the Model

As you start to think about the model, it might trigger your instinct for inheritance. The most common inheritance style is Single Table Inheritance (STI) where you would store both people and companies into a table named *contacts*, then have a model for each that stores data in that table.

> NOTE
>
> STI has always been controversal, and every time I've used it, I've regretted it. For that reason, I ban STI!

Instead we'll build up companies in the most simplistic way: duplicating a lot of code. Once we see where things are duplicated, we'll extract them out and get the code DRY. A robust test suite will permit us be aggressive in our refactoring.

In the end, we'll have clean, simple code that follows *The Ruby Way*.

Start a Feature Branch

It's always a good practice to develop on a branch:

- `git checkout -b implement-companies`

Starting up the Company Model

Use the `scaffold` generator to create a `Company` that just has the attribute `name`.

Run `rake db:migrate` to update your database.

Run your tests, make sure there are no failures (pending are OK), then check your code into git.

Company Phone Numbers

Then we want to add phone numbers to the companies. We already have a `PhoneNumber` model, a form, and some views. We can reuse much of this…with one problem.

Let's think about the implementation later, though. Write your tests first.

Starting with Model

Open up the `company_spec.rb` and delete the pending example.

- Create a `let` block that creates a Company
- Write an `is valid` example that tests that the company is valid
- Make sure the tests are passing
- Write a test to ensure that Company is not valid without a name
- See that it fails
- Implement the validation
- See that the 'is valid' test fails
- Fix the `let` block
- See that all your tests pass

Commit your changes.

Moving Towards Phone Numbers

Now we're rolling with some tests, so we should just bring *everything* over from *person_spec* to *company_spec*, right? I wouldn't.

Just bring over and adapt the `"has an array of phone numbers"` example. Run it and it'll fail.

Now we get to think about implementation. To solve this for `Person`, we said that the `PhoneNumber` would `belongs_to` a `Person` and the `Person` would `has_many :phone_numbers`. Try it again here:

- Express the `has_many :phone_numbers` in the `Company` model
- Add a `belongs_to :company` in the `PhoneNumber` model
- Run your examples and you should see `no such column: phone_numbers.company_id`
  .

When we say that a `PhoneNumber` `belongs_to :company` we imply that the `phone_numbers` table has a column named `company_id`. This is not the case, it has a `person_id` but no `company_id`.

We could add another column for `company_id`, but that would imply that a single number could be attached to one `Person` *and* one `Company`. That doesn't make sense for our contact manager.

What we want to do is to abstract the relationship. We'll say that a `PhoneNumber` `belongs_to` a *contact*, and that *contact* could be a `Person` or a `Company`. This is called a polymorphic relationship.

Setup for Polymorphism

Our tests are still red so we're allowed to write code. To implement a polymorphic join, the `phone_numbers` table needs to have the column `person_id` replaced with `contact_id`. Then we need a second column named `contact_type` where Rails will store the class name of the associated contact.

We need a database migration:

Terminal

```
$ rails generate migration ChangePhoneNumbersToContacts
```

In the migration we need to:

- destroy all the existing `PhoneNumbers` with `PhoneNumber.destroy_all`
- remove the column `person_id`
- add a column named `contact_id` that is an `:integer`
- add a column named `contact_type` that is a `:string`
- in the `down` method, `raise ActiveRecord::IrreversibleMigration`

```
1  class ChangePhoneNumbersToContacts < ActiveRecord::Migration
2    def up
3      PhoneNumber.destroy_all
4      remove_column :phone_numbers, :person_id
5      add_column :phone_numbers, :contact_id, :integer
6      add_column :phone_numbers, :contact_type, :string
7    end
8
9    def down
10     raise ActiveRecord::IrreversibleMigration
11   end
12 end
```

Then run the migration. Bye-bye, sample phone number data!

Build the Polymorphic Relationship

Run your tests and feel comforted by them *BLOWING UP*! If you significantly change your database structure like this and you don't cause a bunch of tests to fail, be

concerned about your test coverage.

There are far too many failing tests to tackle all of them at once. Let's just work on the phone number model specs for now.

Run the tests with the following command:

Terminal

```
$ bundle exec rspec spec/models/phone_number_spec.rb
```

I have four failing tests, and all of them are failing for the same reason: `unknown attribute: person_id`.

In the failing spec, change both references to `person_id` to `contact_id`. In the `let`, let's add `contact_type: 'Person'`.

Rerun the phone number model specs.

Now they're complaining that about an `undefined method` person*id'*
`. OK, no problem. Open up the phone number model. Update any validations that include`
*person*id `to` contact_id`.

Run the tests again, and they should be passing, but for the wrong reasons. Let's improve our tests. We still have references to `person` that need to be changed to `contact`. When updated they should look something like this:

```
1  require 'rails_helper'
2
3  RSpec.describe PhoneNumber, :type => :model do
4    let(:person) { Person.create(:first_name => "Jimbob", :last_name =>
5    let(:phone_number) { PhoneNumber.new(number: "111-222-3333", conta
6
7    it 'is valid' do
8      expect(phone_number).to be_valid
9    end
10
11   it 'is invalid without a number' do
12     phone_number.number = nil
13     expect(phone_number).to_not be_valid
14   end
15
16   it 'must have a reference to a contact' do
17     phone_number.contact_id = nil
18     expect(phone_number).not_to be_valid
19   end
20
21   it 'is associated with a contact' do
22     expect(phone_number).to respond_to(:contact)
23   end
24 end
```

Now we should have one failure saying
`expected #<PhoneNumber id: nil, number: "111-222-3333", created_at: nil, updated_at: nil, contact_id: 1, contact_type: "Person"> to respond to :contact`
.

In the model change `belongs_to :person` to `belongs_to :contact, polymorphic: true`. Let's also remove `belongs_to :company`. Your model should look like this:

```
1  class PhoneNumber < ActiveRecord::Base
2    validates :number, :contact_id, presence: true
3    belongs_to :contact, polymorphic: true
4  end
```

Now when you run them, all the tests in the phone number model spec are passing. Let's move on to the phone number controller specs.

Run just the phone number controller tests with the following command:

Terminal

```
$ bundle exec rspec spec/controllers/phone_numbers_controller
```

We have a bunch of failures complaining about `unknown attribute: person_id`.

Update `let(:valid_attributes)` by replacing this attribute with our new attributes.

```
1    let(:valid_attributes) {
2      { number: "MyString", contact_id: 1, contact_type: "Person" }
3    }
```

Go ahead and update all of our attribute hashes in a similar way - replace `person_id` with `contact_id` and add `contact_type`. The `contact_id` values should keep the values of the `person_id` it is replacing. With the exception of `invalid_attribues`, the `contact_type` should be `"Person"`. `invalid_attributes` should set `contact_type` value to `nil`. Replace any other reference to `person_id` to `contact_id` and `person` to `contact`.

The tests are still complaining. Open up the phone numbers controller and replace references to `person_id` to `contact_id` and `person` to `contact`.

I'm now seeing a new error for *all* of my tests when running them - `Cannot redirect to nil!`. If you are seeing different errors, retrace your steps and read through your code thoroughly. These errors should be happening for tests that are trying to create phone numbers.

The reason we are seeing this failure is due to Rails use of strong parameters in the controller. There you should see a method named `phone_number_params`. Notice any

attributes missing from `permit? We don't have` :contact_type`. Go ahead and add it. These are the attributes our controller will allow to be changed.

```
def phone_number_params
  params.require(:phone_number).permit(:number, :contact_id, :contac
end
```

Run the controller tests and they should be passing now. Whew! That was tough.

Let's move on to the `person_view_spec`:

Terminal

```
$ bundle exec rspec spec/features/person_view_spec.rb
```

We are getting `undefined method` phone*numbers'* `. We need to update` *has*many :phone_numbers` in our person model.

```
has_many :phone_numbers, as: :contact
```

Now when we run our tests we should see different failures (confirm you don't see `undefined method` phone_numbers' `). The new error is` ActionView::Template::Error: undefined method `person_id'`. As the error indicates, the failure is occuring in our view. In the person `show` template in the link to 'Add new phone number' change `person_id` to `contact_id`. Also, add in the `contact_type: 'Person'` here.

This helps, but we're still seeing the same error – what gives!? If you look closely you'll see that the "undefined method person_id" error has now moved to a different place: `app/views/phone_numbers/_form.html.erb`. Recall that our feature test goes through the steps of creating and editing new phone numbers, which uses the phone numbers form. So to fix these errors we also need to update the phone number form to remove the outdated `person_id` field.

Open up the form template and change the `person_id` to be a `contact_id`. We also need to tell the phone number what `contact_type` to use, so let's add a new form field for contact_type:

```
  <div class="field">
    <%= f.hidden_field :contact_type %>
  </div>
```

Finally we need to update `app/controllers/phone_numbers_controller.rb` to use this attribute:

```
def new
  @phone_number = PhoneNumber.new(contact_id: params[:contact_id], c
end
```

We should be down to one failure in our feature spec now. If you're still seeing other issues, check through your `PhoneNumbersController` and phone number form to make sure you've swapped all the relevant instances of `person_id` to `contact_type`. Now let's address the remaining failure:

```
1  Failure/Error: expect(page).to have_link('Add phone number', href: n
2    Capybara::ExpectationNotMet:
3      expected to find link "Add phone number" but there were no match
```

This is because we've updated our links to include the new contact information (`contact_type`) in our "Add phone number" link, but have not updated the spec to match. Let's do that now:

```
1  it 'has a link to add another' do
2    expect(page).to have_link('Add phone number', href: new_phone_numb
3  end
```

The `person_view_spec` should now be passing.

Whew

Run all the tests again. What are we still missing?

Well, the `spec/views/phone_numbers/new.html.erb_spec.rb` has some failing tests. Let's get those straightened out.

Change any reference to `person` to be `contact` and run the tests again.

That fixes the `phone_numbers/new` view spec. Next up is `spec/views/phone_numbers/edit.html.erb_spec.rb`. Do the same thing there. Use the same technique for the failing `index.html.erb_spec.rb` and `show.html.erb_spec.erb`. You will also need to update the views that those specs are testing in a similar way.

And now... finally! The only failing test is the company test that we started out with.

Open up the `app/models/company.rb` file and the following relationship between `phone_numbers` should look like this:

```
1  has_many :phone_numbers, as: :contact
```

Re-run all the tests. See *green* and breathe a sigh of relief. Check in your code and rejoice!

Now go after the pending controller tests for the companies controller spec and *check-in* your code.

We're almost done here. We have some weak tests that need to be improved.

Let's start with the person model.

There are two tests: 'has an array of phone numbers' and 'has an array of email addresses'. Right now they are only checking for empty arrays, but they don't really test the relationships that they should.

Update these to:

```
1  it 'responds with its created phone numbers' do
2    person.phone_numbers.build(number: '555-8888')
3    expect(person.phone_numbers.map(&:number)).to eq(['555-8888'])
4  end
5
6  it 'responds with its created email addresses' do
7    person.email_addresses.build(address: 'me@example.com')
8    expect(person.email_addresses.map(&:address)).to eq(['me@example.c
9  end
```

Run the tests and they should all pass.

Now, let's update the company model spec in a similar way. Replace `has an array of phone numbers` with:

```
1  it "responds with its phone numbers after they're created" do
2    phone_number = company.phone_numbers.build(number: "333-4444")
3    expect(phone_number.number).to eq('333-4444')
4  end
```

If your tests are green, commit your changes.

Integration tests for Company

Take a look at the `person_view_spec.rb`. There are several examples that would apply to companies, too.

Create a `company_view_spec.rb` and bring over anything related to phone numbers. Refactor the `before` block and the copied tests to reflect companies.

Make all of the tests except the first one pending so we can deal with this one test at a time.

Implementing Lists, Links, and Partials

Run your tests with

Terminal

```
$ bundle exec rspec spec/features/company_view_spec.rb
```

The first test I have is about displaying phone numbers, and it is failing.

```
1  1) the company view phone numbers shows the phone numbers
2       Failure/Error: expect(page).to have_content(phone.number)
```

```
3         expected to find text "555-1234" in "Name: ACME Edit | Back"
```

That's fair, since we haven't written any code for that behavior yet.

Open up the `app/views/companies/show.html.erb`. Copy and paste the phone number section from the `app/views/people/show.html.erb` template, edit it to taste, and re-run the tests.

Open up the `company_view_spec.rb` file and remove the `pending` declaration in the next test.

Run the specs with `bundle exec rspec spec/features/company_view_spec.rb`. Go ahead and copy/paste from the `people/show` file to get the test to pass.

Remove each `pending` declaration from the specs and copy and copy, paste, and adapt any code from the person's phone number implementation to make it pass.

We'll deal with the duplication a bit later.

Check it in!

Run all the tests and if everything is green, commit your changes.

Companies and Email Addresses

You've done it for phone numbers, now go through the whole process to make email addresses work for companies.

Start with the model spec for Company that says that the company responds with its created email addresses.

Then implement the polymorphism for `EmailAddress` to make it work.

Don't freak out if a bunch of tests are failing, just pick a single spec file and run the specs for that file, ignoring all the others. Get one test passing at a time.

If it helps, mark the failing specs as pending by using `xit` for the `it` blocks. Remove one `x` at a time and get the test to pass.

Once you're green, add integration tests for email addresses to the `company_view_spec.rb`. Deal with one spec at a time until everything works.

That's TDD for you. Now before you take a nap, poke around in the web interface. I think we've got everything working.

Ship It

- You're green, so check in your code to the feature branch.
- Switch back to your master branch and merge in your feature branch. If your forget the branch's name, try `git branch` to list them.
- Push it to Heroku

- Run your migrations
- ...
- Profit!

I5: Removing Code

Russ Olsen, in his book "Eloquent Ruby," has a great line: *"The code you don't write never stops working."*

Our app has entirely too much code. It works, so we can't call it "bad," but we can up the code quality and drop the maintenance load by refactoring.

Refactoring is the process of taking working code and improving it. That might mean reducing complexity, isolating dependencies, removing duplication – anything that leaves the external functionality the same while cleaning up the internals. In my opinion, the art of refactoring is the difference between people who write computer programs and people who are programmers. But there's a catch to refactoring...

"Don't change anything that doesn't have [good test] coverage, otherwise you aren't refactoring – you're just changing [stuff]."

Our app has solid test coverage since it was written through TDD. That coverage gives us permission to refactor.

Start a feature branch

You're on your `master` branch. Create and check out a branch named `refactoring` where we'll make all our changes.

Helper Hitlist

Running the generators is great, but they create a lot of code. There are several files we can delete right away.

One thing you'll see in many Rails projects is a `helpers` folder full of blank files. One of the things I hate about helpers is the naming convention from the generators is to create one helper for each model. Instead, you most often want your helper files grouping related functions, like a `TimeAndDateHelper` or `CurrencyHelper`.

Many developers get tricked into thinking helper classes should be tied to models and every model should have a helper class. That's simply not true.

Run the following command.

Terminal

```
$ wc app/helpers/*
```

All the helpers have 2 lines of code in them. If you open up one of those you'll see that those are just an empty class definition.

You can delete them all. Be sure to also delete the generated specs for those files.

Terminal

```
$ git rm app/helpers/*
$ git rm spec/helpers/*
```

Run your tests and, if you're green, check in the changes.

Revising Controllers

Open up the `phone_numbers_controller.rb`. There are all the default actions here. Do we ever `show` a single phone number? Do we use the index to view all the phone numbers separate from their contacts? No. So let's delete those actions.

Remember to delete the corresponding views, view specs, request specs, and controller specs for the `index` and `show` actions.

We don't want to expose these endpoints to the world, so let's make sure that they're no longer available as routes, either.

change the `resources :phone_numbers`, in `routes.rb`, to read as follows:

```
1  resources :phone_numbers, :except => [:index, :show]
```

You're going to have to delete the corresponding routing specs as well.

If your tests are green, commit your changes.

Now do the same thing for the `EmailAddressesController`

About Before Actions

Take a look at the email addresses controller. You should have this line near the top of the class:

```
1  before_action :set_email_address, only: [:edit, :update, :destroy]
```

This will run the `set_email_address` method *before* the `edit`, `update` or `destroy` actions are run. This allows us to share code without duplication. Take a look at the `set_email_address` method to see what it is doing. Now look at the actions that are dependant on this method.

Experiment a little. What happens if you comment out the before action? How would you get the edit page to load without it? (To be clear, we want the before action so

make sure it's there after experimenting).

Also, note the `private` doesn't have an `end` line. Any method defined after the `private` keyword in a Ruby class will be private to instances of that object. It's common practice to make methods that get used in before actions private.

Removing Blank Controller Actions?

Take a look at the `edit` action. It's totally blank. You can, then, remove the method from the controller. The `before_filter` will still be activated and the view template renders so things will work great.

But I don't think it's worth the developer cost. Not having the method in the file then having it work in the app is *confusing*. I'd recommend you just leave the stub.

Playing with `ApplicationController`

Did you notice that all those controllers inherit from `ApplicationController`? We've now got a private method in each controller that's almost exactly the same. Could we condense them all into one method in the parent class?

Here's what the method would have to do:

- Figure out which controller called the method
- Figure out the model name for that controller
- Run the find action of that model
- Store it into an instance variable with the right name (like `@person` or `@company`)

It's a bit of metaprogramming that took me some experimenting, and here's what I ended up with in my `ApplicationController`:

```
1  def find_resource
2    class_name = params[:controller].singularize
3    klass = class_name.camelize.constantize
4    self.instance_variable_set "@" + class_name, klass.find(params[:id
5  end
```

Then I call that method from `before_action` lines in each controller. The `before_action` call could be pulled into the application controller, but I think that makes the individual controllers too opaque.

Removing Model Duplication

If you look at the `Person` and `Company` models you'll see there are two lines exactly duplicated in each:

```
1  has_many :phone_numbers, as: :contact
2  has_many :email_addresses, as: :contact
```

Essentially each of these models shares the concept of a "contact," but we decided not to go down the dark road of STI. Instead, we should abstract their common code into a

module. Right now it's only two lines, but over time the common code will likely increase. The module will be the place for common code to live.

There are many opinions about where modules should live in your application tree. In this case we're going to create a `Contact` module and it's almost like a model, so let's drop the file right into the models folder.

- Create a file `app/models/contact.rb`
- In it, define a module like this:

```
1  module Contact
2  end
```

- Move the two `has_many` lines from the `Person` model into that `module`
- In their place within the `Person`, add this line:

```
1  include Contact
```

Run your tests and everything will be broken. When we write a module we need to distinguish between three types of code:

- code that should be run in the containing class when the module is included
- methods that should be defined on the including class (like Person.first)
- methods that should be defined for instances of the including class (like Person.first.name)

The normal Ruby syntax to accomplish these jobs is a little ugly. Starting in Rails 3, there's a feature library that cleans up the implementation of modules. We use it like this:

```
1  module Contact
2    extend ActiveSupport::Concern
3
4    included do
5    end
6
7    module ClassMethods
8    end
9
10 end
```

Any code defined inside the `included` block will be run on the class when the module is included. Any methods defined in the `ClassMethods` submodule will be defined on the including class. And methods defined directly in the module will be attached to instances of the class.

Where should your two `has_many` lines go? Figure it out on your own and use your tests to prove that it works. When you're *green*, check it in. (hint: check out the docs if you get stuck: http://api.rubyonrails.org/classes/ActiveSupport/Concern.html [http://api.rubyonrails.org/classes/ActiveSupport/Concern.html])

Cutting Down View Redundancy

Do you remember copying and pasting some view code? I told you to do it, so don't feel guilty.

Extracting view partials.

Partials are view templates which represent reuseable chunks of markup and display logic. They're especially useful for representing similar data in multiple places in your application, as we are doing now with phone numbers and email addresses. Let's extract some of this duplicated markup into partials.

Create a partial `app/views/phone_numbers/_phone_numbers.html.erb`. Copy the phone number list from the `companies/show.html.erb` template into the partial, and then replace the list in the list in the companies template with a call to render that partial:

```
1  <%= render 'phone_numbers/phone_numbers' %>
```

Run your company view integration tests again:

Terminal

```
$ bundle exec rspec spec/features/company_view_spec.rb
```

The test passes.

We've only fixed half the duplication. Run the person view specs to make sure that they pass:

Terminal

```
$ bundle exec rspec spec/features/person_view_spec.rb
```

Now open up the `people/show.html.erb` file and replace the phone number list with a call to render the new partial:

```
1  <%= render 'phone_numbers/phone_number' %>
```

Run the person view tests again, and all of them are failing with the following error:

```
1  Failure/Error: visit person_path(person)
2  ActionView::Template::Error:
3   undefined method `phone_numbers' for nil:NilClass
```

Open up the `phone_numbers/_phone_numbers.html.erb` partial. We have an explicit reference to the `@company` in there, but now we're rendering the partial from the person context, `@company` is not defined, but `@person` is.

When we write ruby code, we always aim to keep our classes as independent and decoupled as possible. The same principle applies to writing good, reusable view templates. Here we need to decouple the phone numbers partial from the surrounding context, and one easy way to do this is by having the list of phone numbers passed in explicitly as a template local variable.

Luckily rails facilitates this by allowing us to pass an arbitrary hash of local variables to the render method.

First, in `phone_numbers/_phone_numbers.html.erb` delete `@company`.

Then, in `people/show.html.erb` change the call to render to be as follows:

```
1  <%= render 'phone_numbers/phone_numbers', :phone_numbers => @person.
```

And finally, in `companies/show.html.erb`, update the call to render to send in the `@company.phone_numbers`.

Run all the tests. Everything should be passing.

Extracting an Email Addresses Partial

Then repeat the exact same process for the email addresses.

Check It In

If your tests are green, check in the code.

Simplifying Views

Our views have a ton of markup in them and the output is *ugly*! Let's cut it down.

Companies & People Index

Open the `app/views/companies/index.html.erb` and...

- Turn the three `td` elements with the "Show", "Edit", and "Destroy" links into a single `td` where each link is a list item inside a `ul` with the class name `actions`
- Add the id `"new_company"` to the link for the new company page

Run your tests and everything should be green.

Now go through the *same process* for `app/views/people/index.html.erb` using the word `person` instead of `company` where appropriate. Also combine the first name and last name into a single `td` for `name`.

This last thing will break a test, so make sure you fix that before checking in.

Company & Person Show

Let's make a similar set of changes to `app/views/companies/show.html.erb`...

- Create a heading line to use the name of the company
- Remove the paragraph with the company name
- Make sure the phone numbers just renders the partial (not wrapped in a paragraph etc.)
- Do the same for the email addresses
- Change the actions so they're inside `li` tags inside a `ul` with the class name `"actions"`
- Wrap the whole view in a div with class name `"company"`

Run your tests and they should be green. Then, *repeat the process* for `app/views/people/show.html.erb`

When you're green, check it in.

Company & Person Edit

Just a few small changes to the `edit` template:

- Change the heading so it uses the name of the company/person
- Change the links and the bottom to be wrapped in `li` tags inside a `ul` with classname `"actions"`

PhoneNumber & Email Address New/Edit

Open the `email_addresses/new.html.erb` and change the heading from

```
1  <h1>Editing email_address</h1>
```

To this:

```
1  <h1><%= "New Email Address for #{@email_address.contact}" %></h1>
```

View it in your browser and...what is that? You probably see something like this:

```
1  New Email Address for #<Person:0x007f902f76f248>
```

That person-like thing is what you get when you call the `to_s` method on an object that doesn't have a `to_s`. This is the version all objects inherit from the `Object` class.

Testing a `to_s` Method

We want to write some code in our models, but we don't have permission to do that without a failing test. Pop open the `person_spec.rb` file. Then add an example like this:

```
1  it "convert to a string with last name, first name" do
2    expect(person.to_s).to eq "Smith, Alice"
3  end
```

The value on the right side will obviously depend on what value you setup in the `let` block. Run the test and it'll go red.

Implementing a `to_s`

Now you get to open `models/person.rb` and define a `to_s` method like this:

```
1  def to_s
2    "#{last_name}, #{first_name}"
3  end
```

That should make your test pass. Go through the same process writing a test for the `to_s` of `Company` then implementing the `to_s` method.

Did It Work?

Flip over to your browser and you'll see that the `title` on the new email address page should look much better. It isn't making a test go green, though, and that makes me feel guilty. We've knowingly spent time implementing untested code.

Let's write a quick view test. In the view spec `email_addresses_new.html.erb_spec.rb`, add this example:

```
1  it "shows the contact's name in the title" do
2    render
3    assert_select("h1", text: "New Email Address for #{email_address.c
4  end
```

You will probably get some errors about `email_address` being undefined. Lets fill in this as well as a person with some `let`s at the top of the file:

```
1    let(:person) { Person.create(:first_name => "Bob", :last_name => "
2    let(:email_address) { EmailAddress.new(:contact_id => person.id, :
3    before(:each) do
4      assign(:email_address, email_address)
5    end
```

Now our example should pass since we already implemented the `to_s` in `person.rb`. Try a little *"Comment Driven Development"*:

- Comment out the `to_s` method in `person.rb`
- Run the test and see it *fail*
- Un-comment the `to_s`
- See it *pass*!

Now in the "edit" view spec (`spec/views/email_addresses/edit.html.erb_spec.rb`), implement a similar example for the `edit` form, then change the view template to make it work.

More Form Tests & Tweaks

Implement the same technique on...

- the `new` template for phone numbers
- the `edit` template for phone numbers

Making Use of the `to_s` Method

Lastly, consider searching your other views and simplifying calls to `@company.name` or `@person.first_name` with `@person.last_name` to just use the implicit `to_s`.

Ship It

We're done with this iteration and your tests are green – it's time to ship it.

Make sure everything's checked in on your feature branch, `checkout` master, `merge` in the branch, then `push` it to Heroku. Check out the results in your browser. Look at the beauty of those minus signs and many deleted files.

EI6: Supporting Users

Note: the iterations six and seven need revision to conform with Iterations 1-5. The step-by-step instructions below will need debugging, but the concepts are the same.

What's the point of a web application if only one person can use it? Let's make our system support multiple users. There are three pieces to making this happen:

- *Authentication* - Establish identity
- *Ownership* - Attach data records to user records
- *Authorization* - Control who is allowed to do what

Background on Authentication

There have been about a dozen popular methods for authenticating Rails applications over the past five years.

The most popular right now is Devise [https://github.com/plataformatec/devise] because it makes it very easy to get up and running quickly. The downside is that the implementation uses very aggressive Ruby and metaprogramming techniques which make it very challenging to customize.

In the past I've been a fan of AuthLogic [https://github.com/binarylogic/authlogic] because it takes a very straightforward model/view/controller approach, but it means you have to write a lot of code to get it up and running.

As we learn more about constructing web applications there is a greater emphasis on decoupling components. It makes a lot of sense to depend on an external service for our authentication, then that service can serve this application along with many others.

Why OmniAuth?

The best application of this concept is the OmniAuth [https://github.com/intridea/omniauth] . It's popular because it allows you to use multiple third-party services to authenticate, but it is really a pattern for component-based authentication. You could let your users login with their Twitter account, but we can also build our own OmniAuth provider that authenticates all your companies' apps. Maybe you can use the existing LDAP provider to hook into ActiveDirectory or OpenLDAP, or make use of the Google Apps interface?

Better yet, OmniAuth can handle multiple concurrent strategies, so you can offer users multiple ways to authenticate. Your app is just built against the OmniAuth interface, those external components can come and go.

Starting a Feature Branch

Before we start writing code, let's create a branch in our repository.

Terminal

```
$ git checkout -b add-authentication
```

Now you're ready to write code.

Getting Started with OmniAuth

The first step is to add the dependency to your `Gemfile`:

```
1  gem 'omniauth'
2  gem 'omniauth-twitter'
```

Then run `bundle` from your terminal.

OmniAuth runs as "Rack Middleware" which means it's not really a part of our app, it's a thin layer between our app and the client. To instantiate and control the middleware, we need an initializer. Create a file `/config/initializers/omniauth.rb` and add the following:

```
1  Rails.application.config.middleware.use OmniAuth::Builder do
2    provider :twitter, "EZYxQSqP0j35QWqoV0kUg", "IToKT8jdWZEhEH60wFL94
3  end
```

What is all that garbage? Twitter, like many API-providing services, wants to track who's using it. They accomplish this by distributing API accounts. Specifically, they use the OAuth protocol which requires a "comsumer key" and a "consumer secret." If you want to build an application using the Twitter API you'll need to register and get your own credentials [https://dev.twitter.com/apps] . For this tutorial, I've registered a sample application and given you my key/secret above.

Trying It Out

You need to *restart your server* so the new library and initializer are picked up. In your browser go to [http://127.0.0.1:8080/auth/twitter] http://127.0.0.1:8080/auth/twitter [http://127.0.0.1:8080/auth/twitter] and, after a moment or two, you should see a Twitter login page. Login to Twitter using any account, then you should see a *Routing Error* from your application. If you've got that, then things are on the right track.

If you get to this point and encounter a *401 Unauthorized* message there is more work to do. You're probably using your own API key and secret. You need to go into the settings on Twitter for your application [https://dev.twitter.com/apps/] , and add http://127.0.0.1 [http://127.0.0.1] as a registered callback domain. I also add http://0.0.0.0 [http://0.0.0.0] and http://localhost [http://localhost] while I'm in there. Now give it a try and you should get the *Routing Error*

Handling the Callback

The way this authentication works is that your app redirects to the third party authenticator, the third party processes the authentication, then it sends the user back to your application at a "callback URL". Twitter is attempting to send the data back to your application, but your app isn't listening at the default OmniAuth callback address, **/auth/twitter/callback**. Let's add a route to listen for those requests.

Open `app/config/routes.rb` and add this line:

```
1  get '/auth/:provider/callback' => 'sessions#create'
```

Re-visit http://localhost:8080/auth/twitter [http://localhost:8080/auth/twitter] , it will process your already-existing Twitter login, then redirect back to your application and give you *Uninitialized Constant SessionsController*. Our router is attempting to call the `create` action of the `SessionsController`, but that controller doesn't exist yet.

Creating a Sessions Controller

Create a controller at `app/controllers/sessions_controller.rb` that looks like this:

```
1  class SessionsController < ApplicationController
2    def create
3      render text: request.env["omniauth.auth"].inspect
4    end
5  end
```

Revisit /auth/twitter [http://localhost:8080/auth/twitter] and, once it redirects to your application, you should see a bunch of information provided by Twitter about the authenticated user! Now we just need to figure out what to *do* with all that.

Creating a User Model

Even though we're using an external service for authentication, we'll still need to keep track of user objects within our system. Let's create a model that will be responsible for

that data.

As you saw, Twitter gives us a ton of data about the user. What should we store in our database? The minimum expectations for an OmniAuth provider are three things:

- *provider* - A string name uniquely identifying the provider service
- *uid* - An identifying string uniquely identifying the user within that provider
- *name* - Some kind of human-meaningful name for the user

Let's start with just those three in our model. From your terminal:

Terminal

```
$ rails generate model User provider:string uid:string name:s
```

Then update the databases with `rake db:migrate`.

Creating Actual Users

How you create users might vary depending on the application. For the purposes of our contact manager, we'll allow anyone to create an account automatically just by logging in with the third party service.

Let's write a test for our `SessionsController`. Make a new file `spec/controllers/sessions_controller_spec.rb`. We don't need all of the data that came back from `Twitter`, just the data that we're interested in.

```
1  require 'spec_helper'
2
3  describe SessionsController do
4
5    describe "#create" do
6
7      it "creates a user from twitter data" do
8        @request.env["omniauth.auth"] = {
9          'provider' => 'twitter',
10         'info' => {'name' => 'Alice Smith'},
11         'uid' => 'abc123'
12       }
13
14       post :create
15       user = User.find_by_uid_and_provider('abc123', 'twitter')
16       expect(user.name).to eq("Alice Smith")
17     end
18
19   end
20
21 end
```

Run this test, and it will fail because we don't have a route for the `sessions#create` action.

We do have a route that goes there, but we can't call it from this controller test. We could add this line to the `config/routes.rb`:

```
1  resource :sessions, :only => [:create]
```

Now the test should fail because we don't actually do anything useful in the controller action yet.

Go ahead and make this pass by just creating a user right there in the controller.

My controller looks like this:

```
1  class SessionsController < ApplicationController
2    def create
3      data = request.env['omniauth.auth']
4      User.create(:provider => data['provider'], :uid => data['uid'],
5      render :nothing => true
6    end
7  end
```

It's ugly, but it will do for now.

We don't always want to create a new user when someone logs in. They might already be in the system.

Let's add a test for that case:

```
1  it "doesn't create duplicate users" do
2    @request.env["omniauth.auth"] = {
3      'provider' => 'twitter',
4      'info' => {'name' => 'Bob Jones'},
5      'uid' => 'xyz456'
6    }
7    User.create(provider: 'twitter', uid: 'xyz456', name: 'Bob Jones')
8
9    post :create
10   expect(User.count).to eq(1)
11 end
```

To get this to pass I changed my create method to this:

```
1  def create
2    data = request.env['omniauth.auth']
3    user = User.where(:provider => data['provider'], :uid => data['uid
4    render :nothing => true
5  end
```

We're still not logging the user in, though. Let's update the tests to expect the current session to have the user ID in them.

I'm renaming my tests to be

```
it 'logs in a new user'
it 'logs in an existing user'
```

```
it "logs in a new user" do
  @request.env["omniauth.auth"] = {
    'provider' => 'twitter',
    'info' => {'name' => 'Alice Smith'},
    'uid' => 'abc123'
  }

  post :create
  user = User.find_by_uid_and_provider('abc123', 'twitter')
  expect(controller.current_user.id).to eq(user.id)
end

it "logs in an existing user" do
  @request.env["omniauth.auth"] = {
    'provider' => 'twitter',
    'info' => {'name' => 'Bob Jones'},
    'uid' => 'xyz456'
  }
  user = User.create(provider: 'twitter', uid: 'xyz456', name: 'Bob .

  post :create
  expect(User.count).to eq(1)
  expect(controller.current_user.id).to eq(user.id)
end
```

Now the tests fail because we don't have a `current_user` in the controller. Let's add a helper method for that.

Open up `app/controllers/application_controller.rb` and add the following to it:

```
helper_method :current_user

def current_user
  @current_user ||= User.find(session[:user_id])
end
```

The test fails because when we say `User.find(session[:user_id])` this comes back as nil.

Let's put the user id in the session. Go back to the sessions controller and update the `create` method:

```
def create
  data = request.env['omniauth.auth']
  user = User.where(provider: data['provider'], uid: data['uid'], na
  session[:user_id] = user.id
  render :nothing => true
end
```

Our tests pass, but there's a lot of logic in this controller. Let's refactor to let the model handle most of this.

Open up `app/models/user.rb` and add the following to it:

```
def self.find_or_create_by_auth(auth_data)
  user = User.where(provider: auth_data['provider'], uid: auth_data[
  if user.name != auth_data["info"]["name"]
    user.name = auth_data["info"]["name"]
    user.save
  end
  user
end
```

To walk through that step by step... * Look in the users table for a record with this provider and uid combination. If it's found, you'll get it back. If it's not found, a new record will be created and returned * Compare the user's name and the name in the auth data. If they're different, either this is a new user and we want to store the name or they've changed their name on the external service and it should be updated here. Then save it. * Either way, return the user

Now, back to `SessionsController`. Update the action to use the new method on the User class:

```
def create
  user = User.find_or_create_by_auth(request.env['omniauth.auth'])
  session[:user_id] = user.id
  render :nothing => true
end
```

Finally, we're going to want to redirect to send them to the `root_path` after login:

Add a test for this behavior:

```
it 'redirects to the companies page' do
  request.env["omniauth.auth"] = {
    'provider' => 'twitter',
    'info' => {'name' => 'Charlie Allen'},
    'uid' => 'prq987'
  }
  user = User.create(provider: 'twitter', uid: 'prq987', name: 'Char
  post :create
  expect(response).to redirect_to(root_path)
end
```

This is horrible! All that setup, just because we want to test the redirect? Right now there's no way around it. Maybe we can figure out a better way later.

Ok, the test fails, because it doesn't know about the `root_path`. If haven't already, add a root path to `routes.rb` - `root to: 'companies#index'` .

Finally, we're failing for the right reason: we're expecting a redirect, but we're getting a render.

Update the controller action:

```
def create
  user = User.find_or_create_by_auth(request.env['omniauth.auth'])
  session[:user_id] = user.id
  redirect_to root_path, notice: "Logged in as #{user.name}"
end
```

To get the notice to display, add the following code just above the `yield` tag in `application.html.erb`

```
<% flash.each do |name, msg| %>
  <%= content_tag :div, msg, class: "alert alert-info" %>
<% end %>
```

This gets the test to pass. We'll leave it at this for now.

Now visit /auth/twitter [http://localhost:8080/auth/twitter] and you should eventually be redirected to your Companies listing and the flash message at the top will show a message saying that you're logged in.

UI for Login/Logout

That's exciting, but now we need links for login/logout that don't require manually manipulating URLs.

We need a test that will make us put a login link in the page.

Create a new file `spec/features/authentication_spec.rb` and put this code in it:

```
require 'spec_helper'
require 'capybara/rails'
require 'capybara/rspec'

describe 'the application', type: :feature do

  context 'when logged out' do
    before(:each) do
      visit root_path
    end

    it 'has a login link' do
      expect(page).to have_link('Login', href: login_path)
    end
  end

end
```

The test fails because we don't have a `root_path`. Add this to your `config/routes.rb` file:

```
root to: 'site#index'
```

Now the test is failing because we don't have a method `login_path`.

Just because we're following the REST convention doesn't mean we can't also create our own named routes. The view snipped we wrote is attempting to link to `login_path`, but our application doesn't yet know about that route.

Open `/config/routes.rb` and add a custom route:

```
1  match "/login" => redirect("/auth/twitter"), as: :login
```

Finally, the test is failing because we don't have a login link in the page.

Anything like login/logout that you want visible on every page goes in the layout.

Open `app/views/layouts/application.html.erb` and you'll see the framing for all our view templates. Let's add in the following right above the `yield` statement:

```
1  <div id="account">
2    <%= link_to "Login", login_path, id: "login" %>
3  </div>
```

It's still failing. What the heck?

It turns out, we still have the `public/index.html` file hanging around so the root path will redirect to '/', but the application won't even bother looking up routes, it will simply show the `public/index.html` page.

Go ahead and `git rm public/index.html`.

Now we get a new error `uninitialized constant SiteController`. Let's create a new file `app/controllers/site_controller.rb` and put the following code in it:

```
1  class SiteController < ApplicationController
2  end
```

Now the tests complain that there's no index action. Add one.

Next we get a complaint about a missing template. Make a new directory `app/views/site` and add an empty `index.html.erb` file to it.

Finally, that gets the test to pass. Now we need to show the logout link if we're logged in.

Add a test:

```
1  context 'when logged in' do
2    before(:each) do
3      visit root_path
4    end
5
6    it 'has a logout link' do
```

```
7       expect(page).to have_link('Logout', href: logout_path)
8     end
9   end
```

The tests give us an error:

```
1   undefined local variable or method `logout_path'
```

Fix that by adding a route:

```
1   delete "/logout" => "sessions#destroy", as: :logout
```

To get the test to pass, expand the 'account' section in the application layout:

```
1   <div id="account">
2     <%= link_to "Login", login_path, id: "login" %>
3     <%= link_to "Logout", logout_path, id: "logout", method: 'delete' 
4   </div>
```

This works, but we're always showing both the login and the logout link. Let's make sure that we only show the link that we need.

First, let's add a test to the `when logged out` context:

```
1   it 'does not link to logout' do
2     expect(page).not_to have_link('Logout', href: logout_path)
3   end
```

The test fails, naturally.

Let's only show the logout link if we're logged in:

```
1   <div id="account">
2     <%= link_to "Login", login_path, id: "login" %>
3     <% if current_user %>
4       <%= link_to "Logout", logout_path, id: "logout" %>
5     <% end %>
6   </div>
```

Now our 'logged in' context is failing, because we aren't actually logged in.

We can't just access the session, because we're using Capybara here, and that isn't meant to give you access to the internals of your app.

Going through twitter to log in is going to be too painful, so let's create a fake login action that just these tests have access to.

At the top of this page, create a controller:

```
require 'spec_helper'
require 'capybara/rails'
require 'capybara/rspec'

class FakeSessionsController < ApplicationController
  def create
    session[:user_id] = params[:user_id]
    redirect_to root_path
  end
end

describe 'the application', type: :feature do
  # ...
end
```

Then change the ‘logged in’ context to set up the routes we need:

```
context 'when logged in' do
  before(:each) do
    Rails.application.routes.draw do
      root to: 'site#index'
      get '/fake_login' => 'fake_sessions#create', as: :fake_login
      match '/login' => redirect('/auth/twitter'), as: :login
      delete "/logout" => "sessions#destroy", as: :logout
    end
    user = User.create(name: 'Jane Doe')
    visit fake_login_path(:user_id => user.id)
  end

  after(:each) do
    Rails.application.reload_routes!
  end

  it 'has a logout link' do
    expect(page).to have_link('Logout', href: logout_path)
  end
end
```

This gets our tests passing again.

Add another test to the ‘logged in’ context:

```
it 'does not have a login link' do
  expect(page).not_to have_link('Login', href: login_path)
end
```

This fails. Make it pass by updating the account section in the layout.

```
<div id="account">
  <% if current_user %>
    <%= link_to "Logout", logout_path, id: "logout", method: 'delete
  <% else %>
    <%= link_to "Login", login_path, id: "login" %>
  <% end %>
</div>
```

There, it works. Run all your tests, and if they're passing check it all in.

Implementing Logout

Our login works great, but we can't logout! When you click the logout link it's attempting to call the `destroy` action of `SessionsController`. Let's implement that.

- Open `sessions_controller_spec`
- Write a test that has a user id in the session, hits the destroy action of the sessions controller, and asserts that you no longer have a user id in the session.
- Make the test pass.

I had to update the before filter in the sessions controller spec to include the `:destroy` route:

```
1  before(:each) do
2    Rails.application.routes.draw do
3      resource :sessions, :only => [:create, :destroy]
4    end
5  end
```

And then my test calls that route like this:

```
1  delete :destroy
```

- Write another test that asserts that the user gets redirected to the root path.

Ship It

If all your tests are passing, hop over to a terminal and `add` your files, `commit` your changes, `merge` the branch, and `push` it to Heroku.

EI7: Adding Ownership

We've got users, but they all share the same contacts. That obviously won't work. We need to rethink our data model to attach contacts to a `User`.

Let's start with some tests. Open up `user_spec.rb`, delete the pending spec, and add this example:

```
1  let(:user) { User.new }
2
3  it 'has associated people' do
4    expect(user.people).to be_instance_of(Array)
5  end
```

If you run your tests that test will fail because `people` is undefined for a `User`.

Open your `User` model and express a `has_many` relationship to `people`.

Re-run your tests. They should be passing.

Setting up a Factory

So far each of our test files has been making the objects it'll need for the tests. If now decide that a `Person` had a required attribute of `title`, we'd have to update several spec files to create the objects properly.

This duplication makes our tests more fragile than they should be. We need to introduce a factory.

The most common libraries for test factories are FactoryGirl [https://github.com/thoughtbot/factory_girl] and Machinist [https://github.com/notahat/machinist] . Each of them has hit a rough patch of maintenance, though, which guided me towards a third option.

Let's use Fabrication [https://github.com/paulelliott/fabrication] which is more actively maintained. Open up your **Gemfile** and add a dependency on `"fabrication"` in the test/development environment. Run `bundle` to install the gem.

We can also change the behavior of Rails generators to create fabrication patterns instead of normal fixtures. Open up `config/application.rb`, scroll to the bottom, and just before the config section closes, add this:

```
1  config.generators do |g|
2    g.test_framework      :rspec, fixture: true
3    g.fixture_replacement :fabrication
4  end
```

Using Fabrication

Now we need to make our fabricator. Create a folder `spec/fabricators/` and in it create a file named `user_fabricator.rb`. In that file add this definition:

```
1  Fabricator(:user) do
2    name "Sample User"
3    provider "twitter"
4    uid {Fabricate.sequence(:uid)}
5  end
```

Then go back to `user_spec.rb` and replace the implementation of the `let` block:

```
1  let(:user) { Fabricate(:user) }
```

Now your tests fail because you don't have a user_id column on the people table.

Now your tests will fail because we're missing the relationship in the database. Generate a migration to add the integer column named "user_id" to the people table. Run the migration, run your examples again, and they should pass.

Working on the Person

Let's take a look at the `Person` side of this relationship. Open the `person_spec.rb`. First, let's refactor the `let` block to use a Fabricator.

Create the `spec/fabricators/person_fabricator.rb` file and add this definition:

```
Fabricator(:person) do
  first_name "Alice"
  last_name "Smith"
end
```

Then in the `let` block of `person_spec.rb`, use the fabricator like this:

```
let(:person) { Fabricate(:person) }
```

Run your tests and make sure the examples are still passing.

Testing that a Person Belongs to a User

Add an example checking that the `person` is the child of a `User`.

```
it 'is a child of the user' do
  expect(person.user).to be_instance_of(User)
end
```

The test fails because the person doesn't have a method name user.

Add the `belongs_to` association in `Person`.

Run the tests. Now they fail with this message:

```
expected nil to be an instance of User
```

This is really testing two things: that the person responds to the method call `user` and that the response is a `User`.

Revising the Person Fabricator

We need to work more on the fabricator. When we create a `Person`, we need to attach it to a `User`. It's super easy because we've already got a fabricator for `User`. Open the `person_fabricator.rb` and add the line `user` so you have this:

```
Fabricator(:person) do
  first_name "Alice"
  last_name "Smith"
  user
end
```

Now when a `Person` is fabricated it will automatically associate with a user. Run your tests and they should pass.

More From the User Side

Let's check that when a `User` creates `People` they actually get associated. Try this example in `user_spec`:

```
it 'builds associated people' do
  person_1 = Fabricate(:person)
  person_2 = Fabricate(:person)
  [person_1, person_2].each do |person|
    expect(user.people).not_to include(person)
    user.people << person
    expect(user.people).to include(person)
  end
end
```

Run your tests and it'll pass because we're correctly setup the association on both sides.

Now for Companies

Write a similar test for companies:

```
it 'builds associated companies' do
  company_1 = Fabricate(:company)
  company_2 = Fabricate(:company)
  [company_1, company_2].each do |company|
    expect(user.companies).not_to include(company)
    user.companies << company
    expect(user.companies).to include(company)
  end
end
```

Run your tests and it'll fail for several reasons. Work through them one-by-one until it's passing. Here's how I did it:

- Create a `Fabricator` for `Company` similar to the one for `Person`
- Add the `belongs_to :user` association for `Company`
- Add the `has_many :companies` association for `User`
- Create a migration to add `user_id` to the companies table

With that, the tests should pass.

Refactoring the Interface

The most important part of adding the `User` and associations is that when a `User` is logged in they should only see their own contacts. How can we ensure that?

Writing an Integration Test

Let's write integration tests to challenge this behavior. Create a new file `spec/features/people_view_spec.rb`:

```
require 'spec_helper'
require 'capybara/rails'
require 'capybara/rspec'

describe 'the people view', type: :feature do

  context 'when logged in' do
    let(:user) { Fabricate(:user) }

    it 'displays people associated with the user' do
      person_1 = Fabricate(:person)
      person_1.user = user
      person_1.save
      visit(people_path)
      expect(page).to have_text(person_1.to_s)
    end

  end

end
```

Run that and it should pass. #### The Negative Case Now let's make sure they don't see other user's contacts. Here's an example.

```
it "does not display people associated with another user" do
  user_2 = Fabricate(:user)
  person_2 = Fabricate(:person)
  person_2.user = user_2
  person_2.save
  visit(people_path)
  expect(page).not_to have_text(person_2.to_s)
end
```

We create a second user, attach them to a second person, then visit the listing. Run your tests and this test will fail because the index is still showing *all* the people in the database.

Scoping to the Current User

Open up the `PeopleController` and look at the `index` action. It's querying for `Person.all`, but we want it to only display the people for `current_user`. Change the action so it looks like this:

```
def index
  @people = current_user.people
end
```

Then run your tests and you'll find your test is crashing because `current_user` is `nil`. Our tests aren't logging in, so there is no `current_user`.

Faking a Login

We need to have our tests "login" to the system. We don't want to actually connect to the login provider, we want to mock a request/response cycle.

Here's one way to do it. Create a folder `spec/support` if you don't have one already. In there create file named `omniauth.rb`. In this file we can define methods that will be available to all specs in the test suite. Here's how we can fake the login:

```
def login_as(user)
  OmniAuth.config.test_mode = true
  OmniAuth.config.mock_auth[:twitter] = {
      "provider" => user.provider,
      "uid" => user.uid,
      "info" => {"name"=>user.name}
  }
  visit(login_path)
end
```

Now we can call the `login_as` method from any spec, passing in the desired `User` object, then the system will believe they have logged in.

Update the tests so that the user logs in right before visiting the people path.

This should get the people view feature specs passing.

Run all your tests. We have two failures, both are because we changed the behavior of the index action in the `PeopleController`.

Let's start with the people controller specs. The test is failing here because we assert that the people are assigned to the page, but this only happens if you're logged in now.

Let's update the test:

```
describe "GET index" do
  it "assigns the current user's people" do
    user = User.create
    person = Person.create! valid_attributes.merge(user_id: user.id)
    get :index, {}, {:user_id => user.id}
    assigns(:people).should eq([person])
  end
end
```

The tests fail:

```
Can't mass-assign protected attributes: user_id
```

Add the user to the list of attributes that are `attr_accessible` inside the `Person` model.

Note to self: exposing the user_id might not be the best idea ever. Security hole. Should we rework this section?

The people controller specs should now be passing.

Run all your specs. The last failure is the `spec/request/people_spec.rb`. Since this spec is duplicating tests that we already have in the features, go ahead and delete the test.

Refactoring `person_view_spec`

Now that we want to scope down to just people attached to the current user we'll need to make some changes to `person_view_spec.rb`.

First, let's use the person fabricator in the `let` block for `:person`.

The tests should still pass.

Add a `user` that the specs can use to log in:

```
1  let(:user) { person.user }
```

Now inside of the `describe 'phone numbers'` example group, log the user in before visiting the person path:

```
1  before(:each) do
2    person.phone_numbers.create(number: "555-1234")
3    person.phone_numbers.create(number: "555-5678")
4    login_as(user)
5    visit person_path(person)
6  end
```

Do the same thing with the `describe 'email addresses'`:

```
1  before(:each) do
2    person.email_addresses.create(address: 'one@example.com')
3    person.email_addresses.create(address: 'two@example.com')
4    login_as(user)
5    visit person_path(person)
6  end
```

Before we actually make the change to scope the data down to the current user, let's also update the controller specs:

Add the following right inside the first `describe`:

```
1  let(:user) { Fabricate(:user) }
```

Update the `valid_attributes` and `valid_session` methods to include the `user_id`:

```
1  def valid_attributes
2    {"first_name" => "Alice", "last_name" => "Smith", "user_id" => use
3  end
```

```
4
5  def valid_session
6    {user_id: user.id}
7  end
```

The tests still pass, but we haven't actually scoped the page down to show only the current user's data.

In the `PeopleController` change the `lookup_user` method to only search in the current user's people:

```
1  def lookup_person
2    @person = current_user.people.find(params[:id])
3  end
```

And Now, Companies

You've done good work on people, but now we need to scope companies down to just the logged in user and refactor the tests as necessary.

Follow the same processes and go to it!

Breathe, Ship

That was a tough iteration but thinking about the tests helped up find the trouble spots. If your tests are *green*, check in your code, `checkout` the master branch, `merge` your feature branch, and ship it to Heroku!