01 Getting Started

Setting up Rails

Step 1: Install the latest version of rails (3.1rc4)

```
$ gem install rails --pre
```

Step 2: Generate a new rails application

```
$ rails new todolist
```

Check out all of the code generated for us. It follows the conventions discussed earlier and even runs bundler for us to ensure we are ready to go.

Welcome Aboard!

The rails application runs out of the box.

```
$ cd todolist
$ rails server
```

Visit locahost:3000 in your browser of choice.

Scaffolding.

The quickest (and dirtiest) way to get a rails app doing something useful is to use scaffolding. While you're learning the framework this is a great tool. Once you get your head around rails you will find yourself using this less and less.

```
$ rails generate scaffold todo title:string note:text due_by:datetime complete:boolean
```

The console output will let you know which files it has created. There will be a migration file, a model file, tests, a restful controller, views, a helper file, css and javascript files.

Have a look at each generated file to get an understanding of what it does. Scaffolding generates a lot of template files, hence their dwindling use as you understand rails.

Migrations

Before you can do anything with the generated code you need to run the migration that was generated.

```
$ rake db:migrate
```

This creates a table in your database with the specified columns from the scaffold command plus a few extras.

```
t.timestamps
```

Timestamps is a shorthand for creating two datetime fields: created*at*, *updated*at.

ActiveRecord (the default ORM in rails) will fill these in for you as you create and update records.

REST in Action

You've run the scaffold, generated a bunch of code, migrated your database. So what have you actually achieved?

```
$ rails server
```

Visit http://localhost:3000/todos

Fully working CRUD.

You can create, read, update and delete todo items.

Testing

So far rails has generated a lot of code for you, you can be pretty sure it works. Ruby developers like writing tests, rails makes this easy.

```
$ rake test
```

You just ran the generated tests, rails is awesome like that.

TDD says we should write failing tests before writing code, so lets write a simple validation test:

```
test "title should be present" do
    todo = Todo.new
    assert !todo.valid?

    todo.title = "My Title"
    assert todo.valid?
end
```

Run the tests again, and this should fail. To make it pass we can use one of the built in ActveRecord validation helpers.

```
validates_presence_of :title
```

Try it out in the browser.

More Testing

One big todo list isn't very appealing. Most people want to create some kind of bucket to separate out tasks. Let's build a list to hold our todos.

```
$ rails generate scaffold list title:string
```

We should probably validate that our lists have a title and that it is unique. ActiveRecord has another helpful validation to ensure that entries are unique.

```
validates_uniqueness_of :title
```

You should definitely add a database level constraint for that in a production system, but we'll skip that for now.

It's up to you to write failing tests for the list model.

You'll no doubt run into some issues here. Call out if you can't figure out what's going on.

Re-inventing the Wheel is a Bad Idea

Writing tests is sometimes tedious. Thankfuly many people have written them before you.

```
https://github.com/thoughtbot/shoulda
```

Shoulda is designed to help make testing rails code easier. It includes a lot of helpful methods, that allow you to concentrate on getting things done, instead of figuring out how to test them.

Add shoulda to the :test section of your Gemfile.

```
group :test do
    # Pretty printed test output
```

```
    gem 'turn', :require => false
    gem 'shoulda'
    gem 'shoulda-matchers'
end
```

Then use bundler to update your stack.

```
bundle
```

Using the documentation on github, re-write your model tests to use shoulda.

Associations

Now lets link our todo and list models with an association.

We're going to create a many to one relationship. One list has many todo items. In order to associate a todo item with a list, we need to add a column to the todo table that stores the id of the list.

```
$ rails generate migration add_list_id_to_todos
```

The magic ends here, now we have to do some work ourselves.

In the migration file add the following:

```
class AddListIdToTodos < ActiveRecord::Migration
    def change
        add_column :todos, :list_id, :integer
    end
end
```

In a production system, you would add a database index on the list_id column as well.

Run the migrations again to add the column.

```
$ rake db:migrate
```

Our database now knows about the relationship between a list and a todo, however our models do not.

Write failing tests and then make them pass.

Todo items should definitely belong to a list. Write a test for that, and make it pass.

This will expose a problem in the functional tests. We can no longer create and update todos, as the fixtures that power them have no lists.

We can fix this easily by updating the fixture file todos.yml:

```
one:
    list: one
    title: MyString
    note: MyText
    due_by: 2011-07-03 15:31:47
    complete: false

two:
    list: one
    title: MyString
    note: MyText
    due_by: 2011-07-03 15:31:47
    complete: false
```

Now our functional tests pass, but our user interface is somewhat broken. Unforetunately, rails doesn't generate

any integration tests for us, otherwise these would now be failing.

It's up to you to figure out the best way to write integration tests yourself. There are plenty of tools out there, Cucumber, Steak and many more.

We're going to jump straight ahead and fix the user interface.

Routing

The routes file is where we configure the URL structure that our application. The generators have filled in some routes for us, but that was before we created our associations.

Right now we have unrelated two URI's.

```
/todos
/lists
```

But what we really want is a nested URI.

```
/lists/:list_id/todos
```

Open config/routes.rb and read the comments. Try and created a nested route structure. You can inspect your routes by running the routes rake task:

```
$ rake routes
```

When you're done, run your tests. Guess what? They're broken again. Our todos route is now a list_todos route.

Our todos controller expects to know about a list, so that it can scope access to the list and only show the todos that belong to it.

Add a method to the todos_controller to find the correct list.

```
def find_list
@list = List.find(params[:list_id])
end
```

Because we want this method to run before every action, as we can't do anything without a list, we should add it to a before_filter:

```
before_filter :find_list
```

Now that we have a list, we can update the routes in the controller to point to use the new nested route. We can also scope every access to a todo through the @list object:

```
@list.todos
```

I think you're armed with enough information to get started fixing the application.

Remember, you can run your tests and your application in your browser.