Ruby on Rails 3.0

a free student manual

Foreword

Computer Science teachers have it easy. Every time a new semester rolls in, they can simply reuse the material they've been using for years.

As a teacher of a quickly evolving web framework, I do not have that luxury.

As I write this, less than 24 hours has passed since the Rails Core team released the new version of Ruby on Rails: version 3.1.0. This means that I now have to update my student manual (i.e. this document) for upcoming classes to use this new version. Having done that before when we moved from Rails 2.3. to Rails 3.0, I know how much of this document will be changed: sections will be gutted, swaths of code rewritten, and at least one new chapter would be added.

And, yet again, I will not be paid a single cent for those updates.

So instead of just letting this nearly obsolete document go to waste, I've decided to give it away for free.

License

Table of Contents

Introduction to Ruby on Rails

What is Ruby on Rails?

Ruby on Rails (often shortened to *Rails* or *RoR*) is a web development framework written in the Ruby programming language. Some popular examples of websites using Rails are Hulu, BaseCamp, and GitHub.

The biggest draw for Rails is the radical improvement in productivity compared to other web application platforms. Rails makes programming web applications easier by making several assumptions about what every developer needs to get started. This allows you to write less code while accomplishing more than many other languages and frameworks. Longtime Rails developers also report that it makes web application development more fun.

Prerequisites of this Course

This course assumes that the student is an experienced web developer and is familiar with:

1. Basic web development: HTML, CSS, and JavaScript
2. Web application development: HTTP 1.1, server-side scripting
3. Object-Oriented Programming
4. SQL and RDBMS concepts

This course will focus on the *Rails* part of *Ruby on Rails*. This means that knowing the Ruby programming language is not a prerequisite of this course, nor will we dwell too much on the language. We will only discuss just enough Ruby as we go along the lessons – don't worry; you can still do a lot in Ruby on Rails even with limited Ruby knowledge.

Learning Ruby beforehand will help, of course. Some suggested sites for learning Ruby:

- TryRuby.org
- Learn Ruby The Hard Way

Required Software

For this training course, we will be using the following software:

- Ruby 1.9.2
- Ruby on Rails 3.0.10
- RSpec Rails 2.5
- SQLite 3 or MySQL 5

Installing Ruby on Rails

The Apple Macintosh is the platform of choice for developing Ruby on Rails apps. However, the reality is that most of us struggling developers don't have the money to spare in buying expensive Apple computers.

This section will cover how to install a Ruby on Rails development environment on Windows and Linux.

Installing on Windows

Download and use the latest RailsInstaller from http://railsinstaller.org/. This installer already includes Ruby 1.8.7, Rails 3.0.7, SQLite 3 as well as Git and essential tools for building native extensions in Windows.

A smaller alternative would be RailsFTW from http://railsftw.bryanbibat.net/. This installer does not contain Git and build tools (you can download them separately from http://code.google.com/p/msysgit/ and http://rubyinstaller.org/downloads/, respectively), but it's smaller, comes in both Ruby 1.8.7 and 1.9.2, and includes MySQL support.

Don't worry about these installers using slightly older versions of Rails; version 3.0.7 will still work with the examples in this book.

Installing on Linux

Installing RVM

There are several ways of installing Ruby in Linux. Here we will be using Ruby Version Manager (RVM) which will handle the installation for us. RVM requires Git, curl and essential build tools, while Ruby and SQLite have their own requirements. We install them all using (note that the following command must be in a *single line*!)

```
$ sudo apt-get install curl build-essential git-core bison openssl libreadline6
  libreadline6-dev zlib1g zlib1g-dev libssl-dev libyaml-dev libxml2-dev libxslt-dev
  autoconf libc6-dev libsqlite3-0 libsqlite3-dev sqlite3
```

(For the sake of simplicity, we'll assume you're using Ubuntu for the entire *Installing on Linux* section.)

After installing the required packages, we install RVM using the following command:

```
$ bash < <(curl -s https://rvm.beginrescueend.com/install/rvm)
```

To complete the installation, we must first tell bash startup RVM whenever we open bash shell by adding the following line at the end the `.bashrc` file in your home folder:

```
[[ -s "$HOME/.rvm/scripts/rvm" ]] && . "$HOME/.rvm/scripts/rvm"
```

Open a new terminal window to startup RVM and run the following command to install the latest version of Ruby 1.9.2:

```
$ rvm install 1.9.2
$ rvm 1.9.2 --default
```

Installing Rails and SQLite

Installing Rails and SQLite is a matter of calling RubyGems, Ruby's package manager. Thanks to RVM, we do not need to have admin rights to install Rails:

```
$ gem install rails -v=3.0.10
$ gem install sqlite3
```

Linux on Windows

Compared to Mac OS X and Linux, certain Rails commands are noticeably slower in Windows. This is more pronounced once you start using RSpec.

If your computer is fairly new (i.e. multi-core processor and 2 or more GB of RAM are cheap nowadays), you might want to consider running Linux in a virtual machine; Ruby can run faster in Linux even though it's just a virtual machine.

There are free virtualization software available on the Internet. The most popular ones are Sun VirtualBox (http://www.virtualbox.org/) and VMware Player (http://www.vmware.com/products/player/).

As for the Linux distribution, we would recommend Ubuntu (http://www.ubuntu.com/) so that you could just follow the installation instructions above for installing Ruby on Rails.

IDE

ASP.NET and Java EE developers might be surprised to know that the most popular "IDE" for Ruby on Rails is TextMate, a relatively simple text editor on the Mac. As we shall see later, there are some reasons why Ruby on Rails does not require full-fledged IDEs for development.

Any text editor will do fine for Ruby on Rails development. For Windows, Redcar (http://redcareditor.com/) and Sublime Text 2 (http://www.sublimetext.com/2) are good choices because they're free (as of this writing), has Ruby syntax highlighting, and can handle Linux and Mac line breaks properly (unlike Notepad). Linux users can use *vim* or *emacs*; both editors have steep learning curves but they can be more productive than IDEs once you get the hang of them.

If you still insist on using an IDE, there's Aptana RadRails (http://aptana.com/), NetBeans 6.9.1 (http://netbeans.org/), and JetBrains RubyMine (http://www.jetbrains.com/ruby/).

API Documentation

This document is meant to be more of a training manual than a reference manual. It will not contain detailed descriptions of each function available in Ruby on Rails, instead, you will have to refer to the official API documentation for those details.

While API docs for Rails and the other software above are freely available online, you can also download searchable API docs from http://railsapi.com/ for local viewing.

Web Application in 5 Minutes (or Less)

Aling Nena's Request

Imagine one month from now you're walking down your street. Passing by your neighborhood *sari-sari* store, its owner, Aling Nena, calls you over.

"You know how to make websites, right?"

Now, you've known Aling Nena since childhood so you know that she's not your typical sari-sari store owner. Instead of a TV, she's got a desktop computer with broadband connection to occupy herself while waiting for customers. It's not uncommon to see her browsing social networking sites or chatting with her children and grandchildren abroad whenever you buy something from her store.

"I'd like to have a website for tracking the debts of my customers. I'm using a spreadsheet now, but it's getting more and more of a hassle to open the spreadsheet file when I could just open a new tab in my browser. Think you could do it? I'll give you a week-long supply of cornik for your trouble..."

Once you heard the last part, you immediately accepted the request. I mean, who could turn down such an irresistible offer like a *week-long supply of cornik*?

And so Aling Nena enumerated what she wants the website to do:

- It must be able to list all of the her customers' debts.
- It must be able to create a new debt record.
- It must be able to retrieve and display the details of a single debt.
- It must be able to update the details of a debt.
- It must be able to delete a debt record.

Also, a debt record should have the following details:

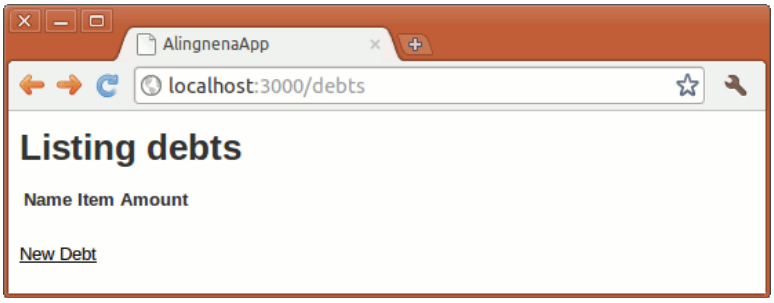
- name – the name of the customer with the debt
- item – contains the items bought under credit; can be much longer than the name field
- amount – the amount the customer owes Aling Nena; should allow cents.

"Sure, no problem," you replied after seeing the requirements.

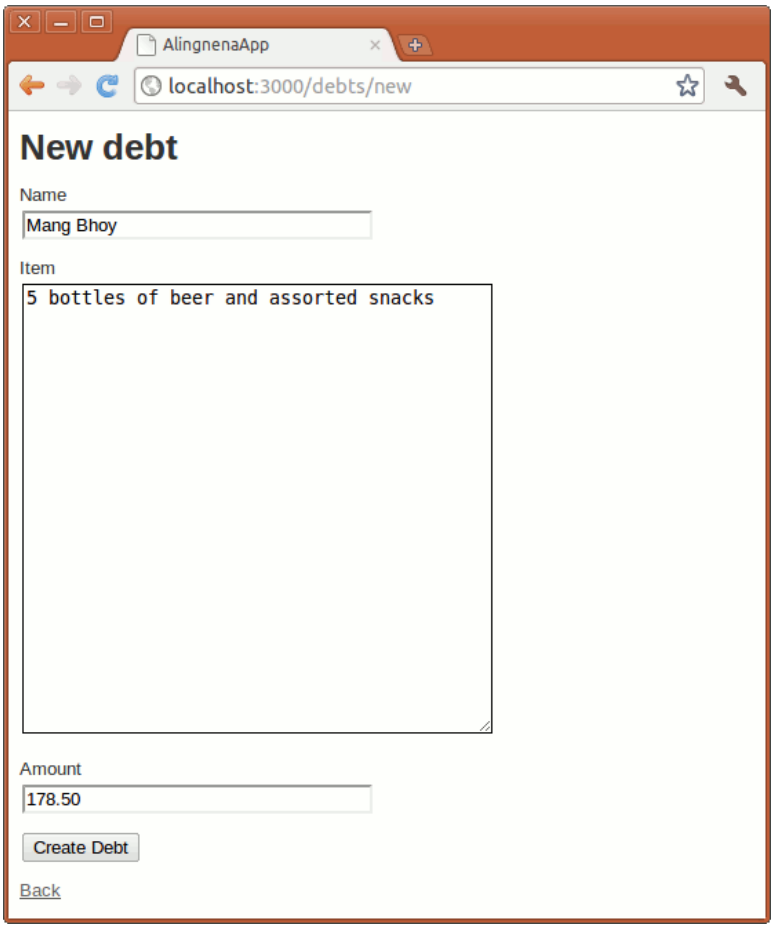
You take out your laptop from your bag, booted it up, and then entered the following commands in the command line:

```
$ rails new alingnena-app
$ cd alingnena-app
$ rails generate scaffold debt name:string item:text amount:decimal
$ rake db:migrate
$ rails server
```

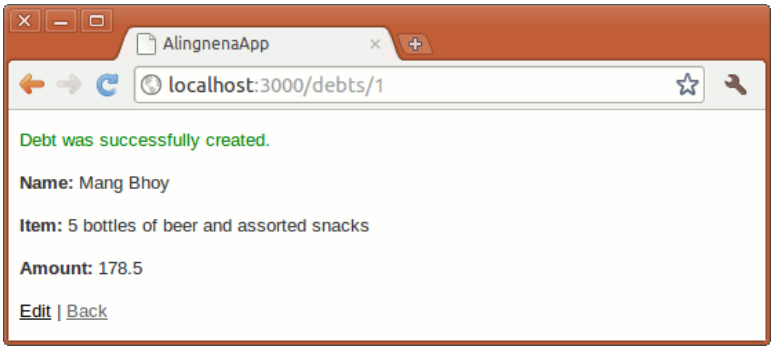
After running these commands, you open your browser to http://localhost:3000/debts and showed the results to Aling Nena.



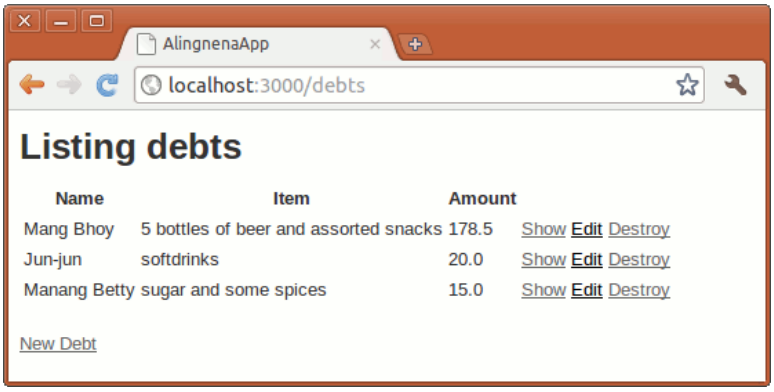
Clicking the New Debt Link will open the New Debt page.



Clicking Create will save the new debt.



Finally, clicking the Back link will return you to the List Debts page. Each existing record has Show, Edit, and Destroy links so that Aling Nena can view, update, and delete the records, respectively.



If you're like Aling Nena, who's staring speechlessly at the screen amazed at how fast you created the web application, you might be wondering, "*That's it?!?*"

Yes, that's all you need to do to create a working Ruby on Rails application. In the next few sections, we shall discuss what just happened here.

MySQL notes

This training course assumes you're using SQLite. If you're using MySQL, you must first install the MySQL gem. In Ubuntu, this can be done via

```
$ sudo apt-get install libmysqlclient-dev
$ gem install mysql2
```

Then you must specify that you're using MySQL at the rails command with the `-d` option. You will also have to modify the database settings at `config/database.yml` to point to the correct server and provide the correct credentials. You may refer to page 167 for more details about database settings.

Also, SQLite does not require you to create a new database schema when you run `rake db:migrate`. For MySQL, you can create the schema via Rails through the `rake db:create` command (as long as the user has been granted rights to create the database):

```
$ rails new alingnena-app -d mysql
$ cd alingnena-app
$ rails generate scaffold debt name:string item:text amount:decimal
$ rake db:create
$ rake db:migrate
$ rails server
```

rails destroy

Made a typo when using the `rails generate` command? Simply re-run the command replacing `generate` with `destroy` to delete all of the new files generated by that command. For example:

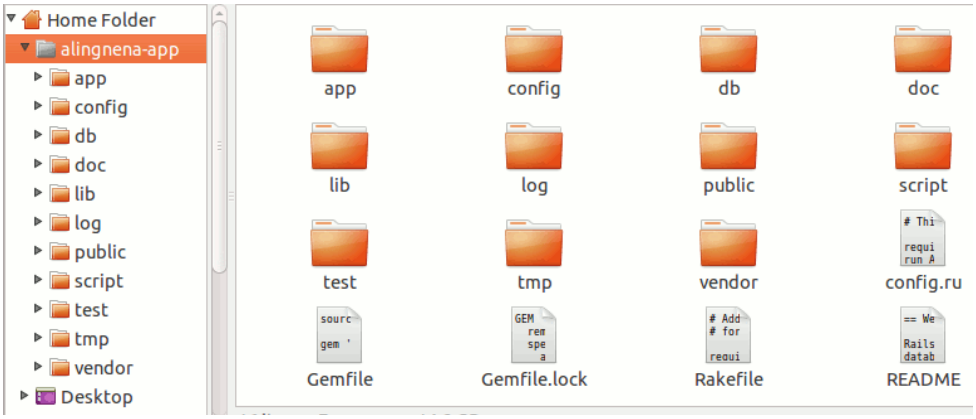
```
$ rails destroy scaffold debt
```

Creating the Application

The first command tells Rails to create the directory structure and the files needed by the application.

```
$ rails new alingnena-app
```

In this case, we named our application "`alingnena-app`" and the following directory structure was created by rails:



Here's a brief explanation for each item in the application folder:

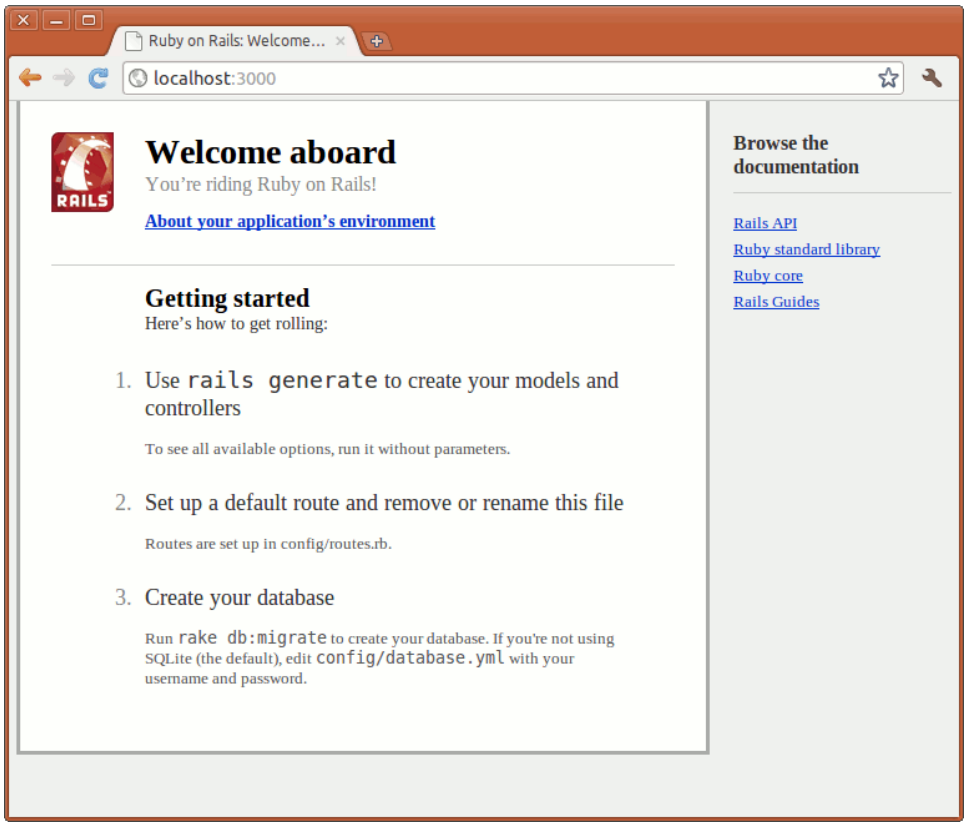
File/Folder	Purpose
`Gemfile`	This file allows you to specify what gem dependencies are needed for your Rails application.
`README`	This is a brief instruction manual for your application. Use it to tell others what your application does, how to set it up, and so on.
`Rakefile`	This file contains batch jobs that can be run from the terminal.
`app/`	Contains the controllers, models, and views for your application. We'll focus on this folder throughout the course.
`config/`	Configure your application's runtime rules, routes, database, and more.
config.ru	Rack configuration for Rack based servers used to start the application.
`db/`	Shows your current database schema, as well as the database migrations. You'll learn about migrations shortly.
`doc/`	In-depth documentation for your application.
`lib/`	Extended modules for your application.
`log/`	Application log files.
`public/`	The only folder seen to the world as-is. This is where your images, JavaScript, stylesheets (CSS), and other static files go.
`script/`	Scripts provided by Rails to do recurring tasks, such as benchmarking, plugin installation, and starting the console or the web server.
`test/`	Unit tests, fixtures, and other test apparatus.

File/Folder	Purpose
`tmp/`	Temporary files
`vendor/`	A place for third-party code. In a typical Rails application, this includes Ruby Gems, the Rails source code (if you install it into your project) and plugins containing additional prepackaged functionality.

At this point, we can already test if the application is setup properly by skipping the 3rd and 4th commands above i.e.:

```
$ cd alingnena-app
$ rails server
```

Opening your browser to http://localhost:3000/ will result in the following screen:



If you've read the table above thoroughly (which I bet you *didn't*), you'll realize that http://localhost:3000/ points to the `public/` folder. Like many web servers, the default HTML file returned by the server is `index.html` when you don't specify the page. You could verify this by manually opening the said file.

Basically that `index.html` file is the "home page" of the application. Later in this course, we shall discuss how to replace that static web page with a dynamic one.

Generating the Scaffolding

The next command generates everything needed for the debt application: the code for setting up the database tables, the code to handle the user's actions, the web pages displayed to the user, etc. The command is a bit complicated so let's dissect the command part-by-part:

```
$ rails generate scaffold debt name:string item:text amount:decimal
```

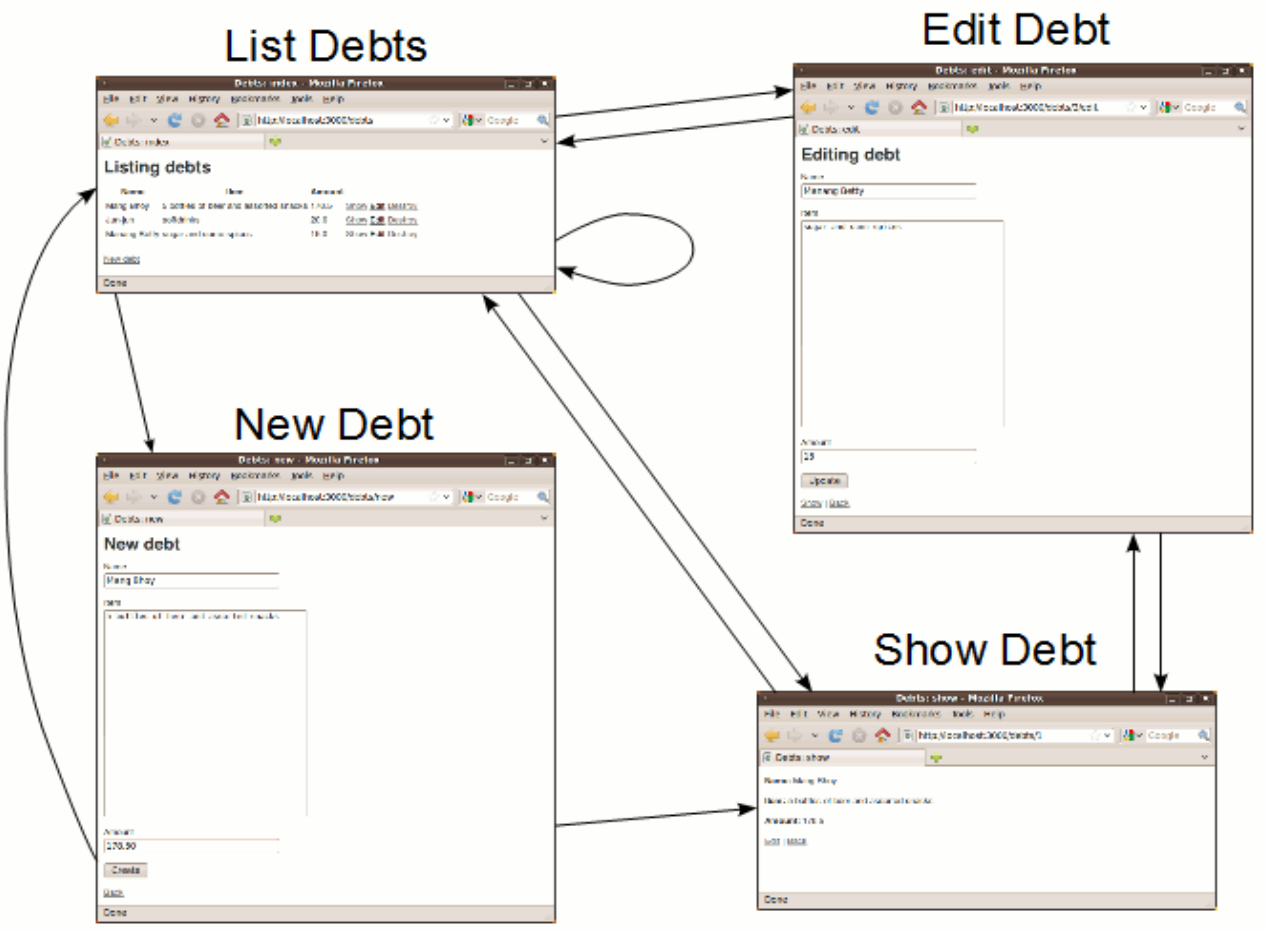
- `rails generate` – tells ruby to run the "generate" script
- `scaffold` – tells the generate script to create a scaffolding. Like its construction counterpart, a scaffold is just a temporary structure where we start building our applications.
- `debt` – the name of this part of the application. In the next chapter, we shall discuss the concept of *Convention Over Configuration* and this is a prime example of that approach. Take note of how Ruby on Rails uses this single word as a *convention* throughout the application:
    - "Debts" will become the name of the resource i.e. the table in the database.
    - The model class that will handle the database operations for that resource will be named "Debt."
    - The debt handling application will be accessed via `http://localhost/debts`.
    - "Debts" will be the name of the debt handling application as we shall later see in the Action Controllers.
- `name:string item:text amount:decimal` – these define the fields for the Debts table. As you may have guessed, this part tells the script that the table will have a "name" field containing strings, an "item" field containing text (longer than a string), and an "amount" field containing a decimal value. *Convention Over Configuration* is also active here, by looking at the type of field, the script knows that "name" should use an `<input type="text">` element in the web page while "item" would use `<textarea>`.

The command generates the following files:

File	Purpose
`app/models/debt.rb`	The Debt model. The Model-View-Controller architecture will be discussed in the next chapter.
`db/migrate/2011xxxxxxxxx_create_debts.rb`	Migration to create the debts table in your database (your name will include a different timestamp)
`app/views/debts/index.html.erb`	A view to display an index of all debts
`app/views/debts/show.html.erb`	A view to display a single post
`app/views/debts/new.html.erb`	A view to create a new post
`app/views/debts/edit.html.erb`	A view to edit an existing post
`app/views/debts/_form.html.erb`	A partial to control the overall look and feel of the form used in edit and new views

File	Purpose
public/stylesheets/scaffold.css	Cascading style sheet to make the scaffolded views look better
app/controllers/debts_controller.rb	The Debts controller
test/functional/debts_controller_test.rb	Functional testing harness for the debts controller
app/helpers/debts_helper.rb	Helper functions to be used from the debts views
config/routes.rb	Edited to include routing information for debts
test/fixtures/debts.yml	Dummy debts for use in testing
app/helpers/debts_helper.rb	Unit testing harness for the debts model
test/unit/helpers/debts_helper_test.rb	Unit testing harness for the debts helper

One thing to note here is that generated scaffolding only has one flow:



While this might be sufficient for simple applications, this flow is rarely used in typical real world applications. Being a temporary structure, it's very likely that you will modify the scaffolding to the point that the finished product is very different from where you started. That said, many experienced Ruby on Rails developers avoid scaffolding entirely, preferring to write all or most of their source code from

scratch.

Setting Up the Database

The final command deals with setting up the database:

```
$ rake db:migrate
```

Database migrations are Ruby classes that are designed to make it simple to create and modify database tables. The rake command above applies all migrations not yet applied to the target database.

In our example, the command above applies the migration that was part of the generated scaffolding, `2011xxxxxxxxx_create_debts.rb`. Let's take a look at the contents of that file:

```
class CreateDebts < ActiveRecord::Migration
  def self.up
    create_table :debts do |t|
      t.string :name
      t.text :item
      t.decimal :amount

      t.timestamps
    end
  end

  def self.down
    drop_table :debts
  end
end
```

This is an example of a real working Ruby code. You might notice how high-level Ruby code is; even if this is the first Ruby code you've seen in your life, it doesn't take much effort to see that this creates a table named "debt" along with the fields we've specified before. This file also defines ("`def`") the "`down`" behavior, namely, to drop the created table (the opposite of the "`up`" behavior). Later in the course, we shall discuss how to do other things with migrations like modifying tables and rolling back changes.

Frequently Asked Questions

Did Ruby on Rails compile the generated code?

No. Ruby is an interpreted language like JavaScript and so it isn't compiled. You could even edit your code while the server is running – you would immediately see the changes the next time you access the updated application.

Can I use a different web server aside from the bundled one?

For most cases, the built in web server (WEBrick) is sufficient for development. You can just switch to more full fledged servers like Apache and nginx once you deploy the live version of your application.

However, if you feel WEBrick is somewhat slow and you're using Linux or OS X, you might want to consider using Unicorn instead. The latter can be installed via:

```
$ gem install unicorn
```

Once installed, you can start the server by running `unicorn` instead of `rails server`. Note that this will

start up the server in port 8080 instead of port 3000.

Some commands start with `rails`*, others with* `rake`*. What's the difference?*

- `rails` – the command used to create an application and run Rails scripts.
- `rake` – similar to C's "`make`" (rake = Ruby + make), it's a tool that allows more complicated scripts for tasks like database migration and running unit tests.

Modifying the Application

Aling Nena's Follow-Up

After being astounded by how fast you build the program, Aling Nena suddenly remembered something.

“Can you add another field where I can put remarks like 'This was partially paid' or 'He promised to pay for it before the end of March' and so on?”

This is a common sight in software development. Software (being *soft*) are often subject to changes. Fortunately for us, Rails provides ways to easily handle changes in the requirements.

Looking at this change request, it is clear that we need to do two things:

1. update the table in the database to add the new field, and
2. update our application to accommodate the new field.

Rails Migrations

As mentioned in the previous chapter, Rails uses migrations to simplify database tasks. Before we move on to how to use migrations to meet our task (i.e. add a new field to the database), let's discuss why there are migrations in Rails.

In other more traditional software development platforms, when you want to add a new field to the database, you'll need to write an SQL statement like

```
ALTER TABLE debts ADD remarks VARCHAR(255);
```

and give it to someone who is responsible for maintaining the databases in order for it to be applied to all those databases. Now, granted SQL isn't that hard to learn, and having a DBA managing the database changes can be a somewhat efficient way of handling the data, but there are some reasons why migrations have the upper hand over this approach.

The first benefit of migrations is that it simplifies the whole database management process. Rails includes scripts to apply changes to different databases allowing any developer (or even a batch script) to deploy changes without the need for a database administrator.

Rails migrations also have a limited set of data types which are internally converted to the appropriate data type for different kinds of databases. In other words, unlike with manual SQL wherein you have to take into account the differences between Oracle, MySQL, IBM DB2, etc., you don't need to worry about these things in migrations.

Another benefit of migrations is that it provides ways to *migrate* to a certain snapshot of the database structure. As there are timestamps in the file name, anyone can use a command to choose which timestamp to roll back to. This is useful when you're testing old builds which require an older version of database schema to work. (We will discuss this command in a later chapter.)

Generating the Migration

As migrations have timestamps in their file names, it may not be practical to create them from scratch,

with the developer defining the current timestamp in the file name. A better approach would be to use the built in migration generator which creates an empty migration file with a proper timestamp.

The format of the command to generate the migration is:

```
$ rails generate migration MigrationName
```

MigrationName can be in camel case or lower case separated by underscores (Rails converts the former to the latter in the actual file name, and the latter to the former in the class name). Here is a sample file produced by the command:

```
class MigrationName < ActiveRecord::Migration
  def self.up
  end

  def self.down
  end
end
```

The generated file, as mentioned above, is empty, and it is up to the developer to define the changes to be applied to the database using migration commands. Migration methods called in `self.up` will be applied when the migration is executed, while methods in `self.down` will be called when the migration is reversed or rolled back.

Rails provides a shortcut for creating migrations that add or remove columns from tables. When you name your migrations as `Add[`*`ColumnName`*`]To[`*`TableName`*`]` or `Remove[`*`ColumnName`*`]From[`*`TableName`*`]` followed by a series of `field_name:data_type` pairs, Rails will automatically create the add column or remove column statements, respectively, in the migration. For example, we can perform the required change to our application without having full knowledge of the migration commands:

```
$ rails generate migration AddRemarksToDebt remarks:text
```

This command will create the following migration file:

```
class AddRemarksToDebt < ActiveRecord::Migration
  def self.up
    add_column :debts, :remarks, :text
  end

  def self.down
    remove_column :debts, :remarks
  end
end
```

Rails supports the following data types:

Data Type	Description
`:string`	A relatively short string field
`:text`	An unbounded string field, used for longer text.
`:integer`	A number with no fractional part.
`:float`	A floating point number. May or may not be precise depending on the database.
`:decimal`	A number with a fractional part. Always precise.
`:datetime`	A date and time field.

`:timestamp`	Same as datetime, but with fractional seconds.
`:time`	A time field.
`:date`	A date field.
`:binary`	Field for storing binary data e.g. image files.
`:boolean`	A field that stores either true or false.

We'll discuss the other possible migration options in a later chapter. For now, let us just stick with `add_column` / `remove_column`.

Ruby Corner

From this point onwards, you will see small sections like this one explaining bits and pieces about Ruby. The alternative to this is that we dedicate an entire chapter of the course just to study Ruby.

We chose the latter approach for the sake of efficiency; not only are we not distracting ourselves too much away from our primary goal (i.e. learn Ruby on Rails), explaining Ruby's features in context of actual working code allows for better understanding and retention.

Ruby Corner – Interactive Ruby

One way of writing and running ruby programs is to put the code inside a `.rb` file then execute it with "`ruby [ruby file]`". The other (simpler) method is to write and run the code under a Ruby shell.

There are two shells that you can use in Rails: `irb`, which is the **i**nteractive **r**u**b**y shell that comes with Ruby, and the Rails script "`rails console`", which creates an environment based on the current Rails app, allowing you to test the classes inside.

Since Ruby is an interpreted scripting language, not only can you run single statements inside the shells, you can also declare functions and classes as well. For example:

```
$ irb
irb(main):001:0> def hello(x)
irb(main):002:1>   "hello #{x}"
irb(main):003:1> end
=> nil
irb(main):004:0> hello("world")
=> "hello world"
```

Shortcut Corner

There are shortcut aliases for certain `rails` commands. You would probably be familiar with them from the message displayed when you enter an incorrect rails command:

```
$ rails server
Error: Command not recognized
Usage: rails COMMAND [ARGS]

The most common rails commands are:
 generate    Generate new code (short-cut alias: "g")
 console     Start the Rails console (short-cut alias: "c")
 server      Start the Rails server (short-cut alias: "s")
...
```

The shortcut aliases can be used in place of their full values. For example "`rails c`" is equivalent to "`rails console`".

Ruby Corner – Dynamic Typing and Variables

Ruby has your typical programming language data types e.g. `String`, `Integer`, `Float`, etc. Being a dynamically typed object oriented programming language, you don't have to worry about creating and assigning the correct data type to the correct variable. Everything is an object and thus can be assigned to any variable.

A variable name in Ruby must begin with an underscore (_) or lower-case letter followed by any combination of alphanumeric characters and underscores. Variable names are case sensitive in Ruby. As with other programming languages, you cannot use reserved words as variable names. The standard for naming variables is to use all lower-case letters and separate words with underscores.

You can assign values to variables using the = operator.

```
x = 10
x = "20"
```

You can create variables anywhere in a ruby program. Location simply dictates the scope of the variable, as all basic variables are local variables (they're only available in their scope).

Constants are variables that start with a capital letter. They can only be assigned a value once.

Other variations of the variable exist, and later we will discuss how instance variables are defined in Ruby.

Ruby Corner – Basic Data Types

Ruby is a fully object-oriented programming language so it doesn't have primitive data types per se. Everything in Ruby is an object, for example, the number literals have methods and you can see stuff like 20.even?() in a typical Ruby code.

Rails does have some classes that you might call basic data types considering the convenience of their use:

- Numbers – integers are Fixnum objects by default (i.e. 100 is parsed as a Fixnum), when they get too large they are converted to Bignum, an infinite precision integer. Decimal values are Float objects, though if you want arbitrary-precision (e.g. you're calculating monetary values where floating point errors are a big no-no), you can convert them to BigDecimal.

- Strings – Like in JavaScript, strings can be enclosed with either single-quotes or double-quotes. When using double-quotes, you can insert values inside the string using string interpolation with `#{expression}`. For example:

```
"1 + 2 = #{1 + 2}"      # this will return "1 + 2 = 3"
```

- Boolean – `true` and `false` are the boolean objects in Ruby

- Null – `nil` represents the null object in Ruby

Ruby Corner – Operators

The precedence of operators in Ruby are described by the following table:

Method?	Operator	Description
Yes	`[ ] [ ]=`	Element reference, element set
Yes	`**`	Exponentiation (raise to the power)
Yes	`! ~ + -`	Not, complement, unary plus and minus (method names for the last two are +@ and -@)
Yes	`* / %`	Multiply, divide, and modulo
Yes	`+ -`	Addition and subtraction
Yes	`>> <<`	Right and left bitwise shift
Yes	`&`	Bitwise `AND'
Yes	`^ |`	Bitwise exclusive `OR' and regular `OR'
Yes	`<= < > >=`	Comparison operators
Yes	`<=> == === != =~ !~`	Equality and pattern match operators (!= and !~ may not be defined as methods)
	`&&`	Logical 'AND'
	`||`	Logical 'OR'
	`.. ...`	Range (inclusive and exclusive)
	`? :`	Ternary if-then-else

	`= %= { /= -= += |= &= >>= <<= *= &&= ||= **=`	Assignment
	`defined?`	Check if specified symbol defined
	`not`	Logical negation
	`or and`	Logical composition
	`if unless while until`	Expression modifiers
	`begin/end`	Block expression

As you can see, these operators should be familiar with non-Ruby save for the few Ruby-specific ones. Studying the use of these operators is left to the reader. Here are some points to note, though:

- The parenthesis takes precedence over all operators.
- The operators with a “Yes” in the “Method?” column are implemented as methods, and may be overridden.
- Logical operators (`and`, `or`, `&&`, `||`) use short-circuit evaluation. Also, every expression evaluates to `true` except for `nil` and `false`.
- Unlike C and C-based languages, there are no `++` and `--` operators in Ruby. Use `+= 1` and `-= 1` instead.

Ruby Corner – Methods

A method in Ruby looks like the following:

```
def add(a, b)
  return a + b
end
```

We won't get too much into detail about the structure of a method declaration so we'll just stick to the basics: a method declaration starts with `def`, followed by a method name, followed by parameters enclosed in a parenthesis. The block is then closed with an `end`. You can use `return` to return a value for the method. If you don't, Ruby uses the value of the last line of the method as the return value. This would allow us to shorten that method into:

```
def add(a, b)
  a + b
end
```

The standard for naming methods is the same as local variables, all lower-case with words separated by underscores.

As with most object oriented programming languages, declaring a method inside a class makes it a publicly accessible instance method. It can also be declared outside a class i.e. used in a script.

Let's take a look again at our new migration:

```
class AddRemarksToDebt < ActiveRecord::Migration
```

```
  def self.up
    add_column :debts, :remarks, :text
  end
...
```

The method `self.up` is declared inside the class `AddRemarksToDebt`. Note that the method is actually just up; the `self.` prefix makes the method a class method, letting the system call `AddRemarksToDebt.up` at the time of migration.

Speaking of method calls, you call Ruby methods pretty much the same way as methods in other programming languages. For example, calling:

```
puts add(3, 3)
```

would print out 6.

Non-Ruby programmers might be surprised to know that puts is also a method (it prints a string to the standard output followed by a new line). In Ruby, enclosing method arguments in parenthesis is optional as long as there is no ambiguity. For example, this statement is legal:

```
puts add 3, 3
```

There is no standard regarding when to use parenthesis in method calls. Programmers usually just use their discretion: when the code reads clearly without parenthesis, they don't use parenthesis. When enclosing a method call makes the distinction between nested methods clear, they use parenthesis. You will see a lot of these types of decisions throughout rails. For example, migrations don't use parenthesis in the `add_column` and `remove_column`:

```
add_column :debts, :remarks, :text
```

Ruby Corner – Symbols

One peculiar data type confuses a lot of new Ruby programmers, though: the Symbol.

Symbols look like variables prefixed by a colon (`:`). A good example would be from our migration above:

```
add_column :debts, :remarks, :text
```

`:debts`, `:remarks`, and `:text` are all symbols.

One easy way of explaining what they are to non-Ruby programmers is that they are just like global String constants. You can't change their value (i.e. `:text` is always `:text`), and you can use them in many places where strings are used (e.g. as a key to a hash).

Ruby Corner – Classes and Modules

Classes in Ruby are declared with the class keyword followed by the class name and ended with an end keyword. The class name must begin with an upper-case letter just like constants.

```
class AddRemarksToDebt < ActiveRecord::Migration
...
end
```

The standard for naming classes is to use CamelCase.

Due to the scripting-like nature of Ruby, classes can be defined anywhere in the program just like local variables and methods.

The initialize method acts as the constructor for Ruby classes. To instantiate a class, you can use the new method. Every methods inside a class is, by default, public. We will discuss instance variables and private methods later on.

You can define a class's superclass by using the < operator in the class definition. In the above case, AddRemarksToDebt is a subclass of Migration.

The ActiveRecord in the above example is a module. Modules are like classes except that they can't be instantiated and they can't be subclassed. The declaration is the same, with module replacing the class keyword.

Modules have two uses. First is for defining namespaces. In the example above, the Migration class is under the ActiveRecord module. You can imagine that the code for the two is like this:

```
module ActiveRecord
  class Migration
    ...
  end
end
```

Another use for modules is to create *mixins*, a sort of multiple inheritance approach in Ruby. Since you can define methods inside a module and the linking process in Ruby consists of bringing together code from various files, mixins are classes which include the methods of other modules. For example:

```
module Module1
  def hello
    puts "hello"
  end
end

module Module2
  def world
    puts "world"
  end
end

class HelloWorld
  include Module1
  include Module2
```

```
        def initialize
          hello
          world
        end
      class
```

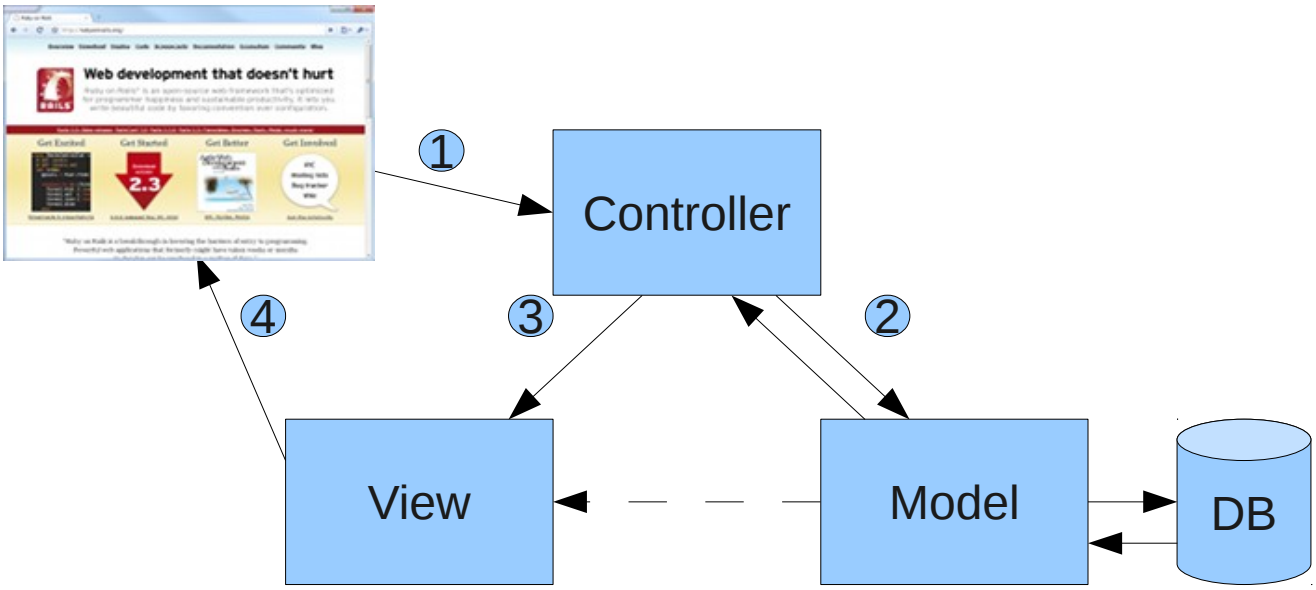
When you call `HelloWorld.new`, the constructor will call the methods taken from the two modules.

An Introduction to the Model-View-Controller Architecture

With the database side of the task out of the way, we can now move on to modifying the application itself. But before that, let's take some time to understand how various components are organized in Rails via the Model-View-Controller (MVC) architecture.

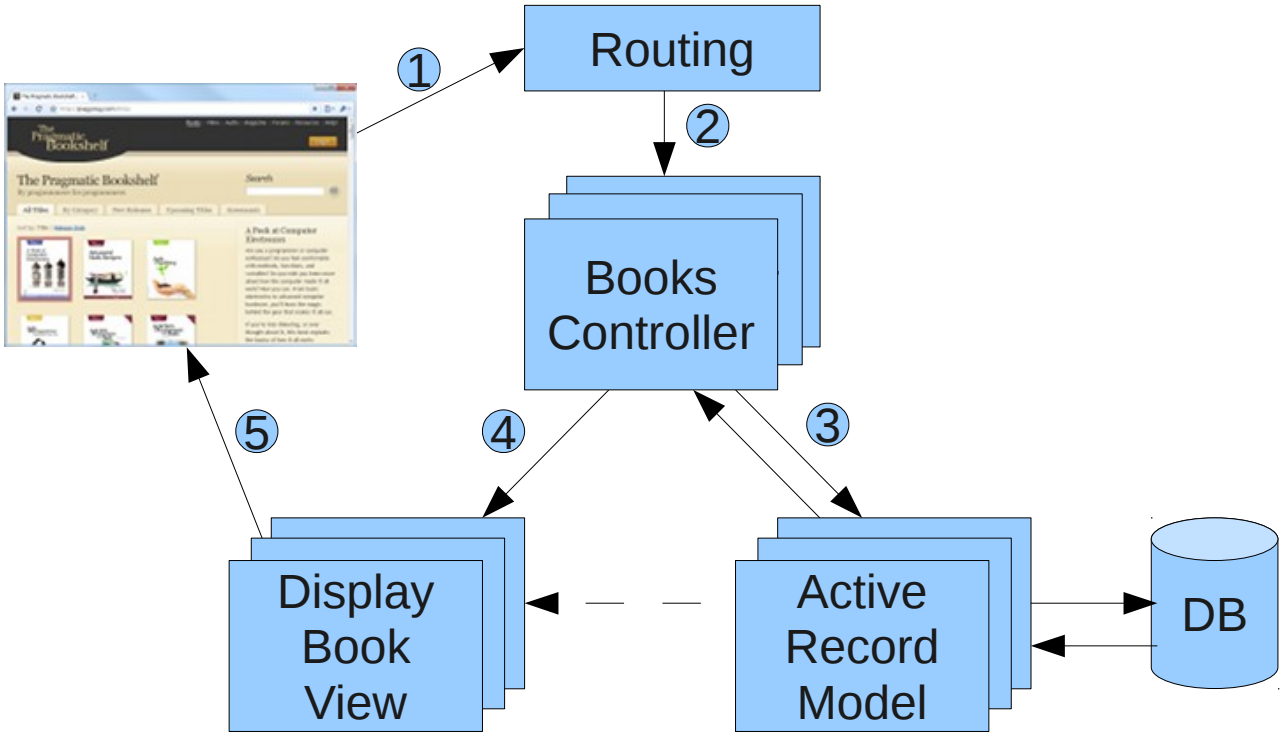
MVC was originally created in the 1980s as a pattern to follow when creating graphical user interfaces in Smalltalk. It found a resurgence in the early 2000s when it became apparent that this model was also applicable for web applications.

In MVC, systems are divided into three parts: the model, the view and the controller. A typical flow through an MVC system would be as follows:



1. User (in the form of a browser here) sends a request to the Controller
2. Controller interacts with the Model
3. Controller invokes the View
4. View renders the information sent to the User.

Putting this in context of Rails applications, when a browser visits a Rails web page, it goes through the following steps:



1. Browser requests a page from the server e.g. http://my.url/books/123
2. Routing finds the Books Controller
3. Controller interacts with the Model
4. Controller invokes the View
5. View renders the page to be sent to the browser

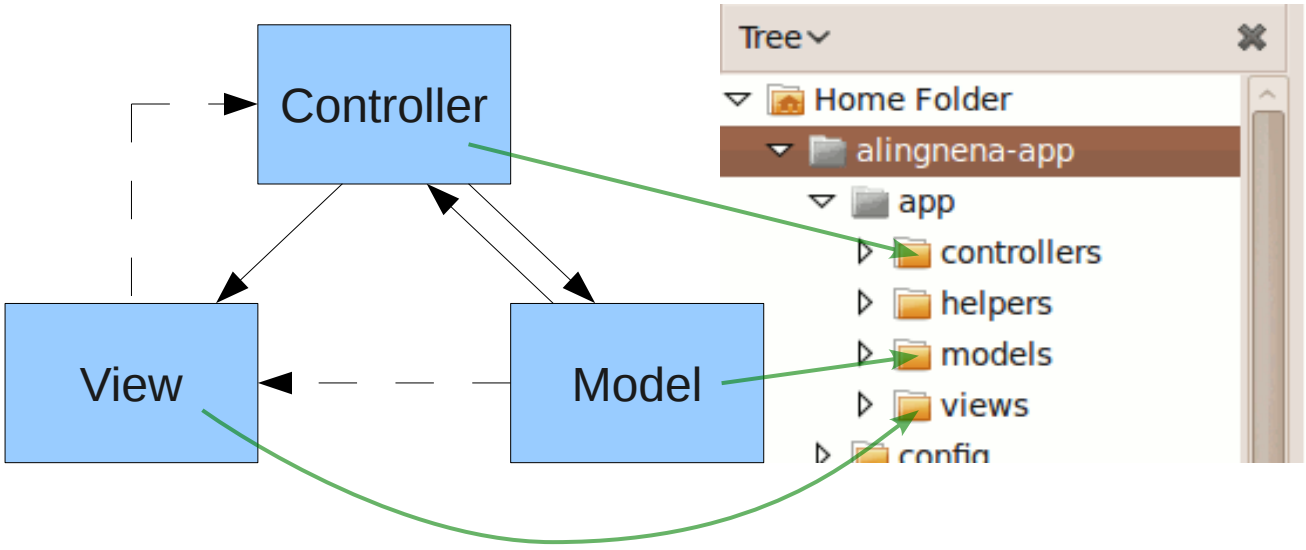
Why use MVC?

In the early days of the web (and even up to now), many web applications were written with all of the processing logic (business processing, routing, rendering) concentrated in a single location. This approach didn't scale well for some reasons:

- Maintenance was hard. Lack of structure meant that the dependencies between modules aren't clear. Changing one part of the system might affect another part of the system without a programmer knowing it.
- Debugging was hard. As all of the processing logic were contained in single files, a programmer hunting for the code that caused a bug would have to scan through a lot of code which has nothing to do with the bug itself before finding it.

MVC addresses these problems. First off, the structure provides a degree of isolation between modules. Sure, changing a Model might still affect a lot of programs in the system, but at least the extent of the changes can be easily predicted.

When you encounter a bug in a Rails application, you also have a better idea where to look for the problem. This diagram shows where the source code for the MVC components are located:



MVC best practices

There are many best practices when it comes to using MVC. We'll list down a few here, but we'll also have to use an analogy to make it clearer to someone who hasn't used MVC before.

Our analogy would be a restaurant analogy.

In a fine dining restaurant, you have your chefs (**Models**) which prepares the food for the customers (**Views**). There are also waiters (**Controllers**) that facilitate communication between the customers and the chefs.

Chefs are expected to do all of the preparing, cooking, and plate presentation on the dishes. Aside from very trivial changes to the dishes (e.g. offering to add pepper to the soup) the waiters do not do anything to the food. Here is the first best practice for MVC: **Models should do most of the processing, controllers should only focus on the routing.** Similarly: **If you find your controllers doing a lot of processing, find a way to move those logic to the model.** This is usually called the "**fat model, skinny controller**" approach.

Some dishes are so complicated that a single chef isn't enough to handle it. In this case, it's alright for multiple chefs to work on a single dish without having to go through the waiter. In other words, **Related models can communicate with each other before passing their result to the controller.**

As for our customers, they should not talk directly to the chef and vice versa i.e. **Views should not directly call models directly, and vice versa.** However, this does not mean that the customers should be mindless. It's up to them how to eat their food. In MVC terms, **Processing logic is allowed in views, as long as it's related to the view or presentation.**

Now that we understand the basic underpinnings of Rails, we now have a basic idea about what to modify in our system to meet the demands of the current task.

Modifying the Model

...or not.

Veterans of other MVC frameworks might think that we need to modify some model related files to reflect the changes to the database. In some frameworks, we might need to add additional fields to certain classes. In other frameworks, we might need to modify certain XML files to include the new fields.

Fortunately for Ruby on Rails, we don't need to do any of those.

At the heart of Rails' model components is **Active Record**, a Ruby package that handles the database related portion of Rails. Not only does Active Record provide functions for saving and retrieving data, it also provides other features that help simplify database management e.g. migration commands.

To make development easier, Active Record *directly checks the database schema to determine the fields of the model*. This is made possible thanks to Convention over Configuration (database fields and model fields should be named the same) and Ruby's dynamic nature (methods can be defined on the fly). In short, there is no need to change any part of the model when the database is changed — the change is automatically applied to the model and we could access the `remarks` field via `@debt.remarks`.

Of course, if we would need to add some field validation (e.g. mandatory checking, field length checking), we will need to modify the `/app/models/debt.rb`. We will leave that topic for a later chapter.

Modifying the Controller?

By looking at the flow of the MVC model, you could see that adding a new field to our screen wouldn't affect our controller: the routing of data is still the same after the change. Therefore, we don't need to modify `debts_controller.rb`.

(By the way, the Ruby package that handles the controller for Rails is **ActionController**.)

Modifying the View

In the end, the only files we have to modify to include the new field is in the View, namely the four view files for Listing, Show, Create, and Edit.

Insert the highlighted lines into `/app/views/debts/index.html.erb`:

```
<h1>Listing debts</h1>

<table>
  <tr>
    <th>Name</th>
    <th>Item</th>
    <th>Amount</th>
    <th>Remarks</th>
  </tr>

<% @debts.each do |debt| %>
  <tr>
    <td><%= debt.name %></td>
    <td><%= debt.item %></td>
    <td><%= debt.amount %></td>
    <td><%= debt.remarks %></td>
    <td><%= link_to 'Show', debt %></td>
```

```
...
```

Insert the highlighted lines into `/app/views/debts/show.html.erb`:

```
...
<p>
  <b>Amount:</b>
  <%= @debt.amount %>
</p>

<p>
  <b>Remarks:</b>
  <%= @debt.remarks %>
</p>
...
```

And finally, insert the highlighted lines into `/app/views/debts/_form.html.erb`:

```
...
  <div class="field">
    <%= f.label :amount %><br />
    <%= f.text_field :amount %>
  </div>
  <div class="field">
    <%= f.label :remarks %><br />
    <%= f.text_area :remarks %>
  </div>
  <div class="actions">
...
```

(Just as in the controller, the Ruby package that handles the view for Rails is **ActionView**.)

Rails Applications Without Scaffold (part 1: Show and List)

Pleased with how you handled the “Debt Records” program, Aling Nena has decided to hire you part time in order to computerize every aspect of her shop.

Part of this job is to create a system to to handle the inventory of her shop's goods. So for your next task, you need to create a program which maintains the list of products in her shop.

This is a good time to discuss how to create a program from scratch i.e. without using the `rails generate scaffold` script.

The Plan

The Product Maintenance program has 6 different parts:

- User must be able to create a product record
- User must be able to view the product record
- User must be able to view a list of product records
- User must be able to modify a product record
- User must be able to delete a product record
- User must be able to search for a product record

In addition to this, we also need to setup the database for a Products table with the appropriate fields. In the end, we have 7 tasks to be done in the following order:

1. Create the migration for Product
2. Program the “Show Product” module
3. Program the “List Products” module
4. Program the “Delete Product” module
5. Program the “Search Products” module
6. Program the “Create Product” module
7. Program the “Edit Product” module

We've chosen this order of tasks so that the tasks are done in increasing coding complexity. So without further ado, let's proceed with the first 2 tasks.

Viewing a Record

First up is the “Show Product” module, a page that displays the details of a certain Product record. We've combined this step with the “generate migration” step because we're going to generate the migration

along the way anyway.

The basic flow for showing a single record from the user is as follows:

1. User provides the id of the record via the URL
2. Program retrieves the record from database
3. Program renders the retrieved data into a web page and sends it to the user

In this lesson, we shall see how these steps are coded in Rails.

Generating a Model and adding test data

Instead of using `rails generate scaffold`, we shall use `rails generate model`, a script that only generates the migration and the model for the specified model and fields. The syntax of the latter is the same as the former:

```
$ rails generate model Product name:string description:text cost:decimal stock:integer
```

We need to modify the generated migration to add some dummy data for our new screen. Add the highlighted lines to the migration file:

```
class CreateProducts < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
      t.string :name
      t.text :description
      t.decimal :cost
      t.integer :stock

      t.timestamps
    end

    Product.create :name => "test product 1", :description => "test description 1",
                   :cost => 1.11, :stock => 10
    Product.create :name => "test product 2", :description => "<b>test description 2</b>",
                   :cost => 2.22, :stock => 20

  end

  def self.down
    drop_table :products
  end
end
```

Generating a Controller and View Page Templates

A controller **action** (not to be confused with *Action Controller*) is a public instance method inside the controller class that processes requests from the user. We can generate controller actions and their corresponding views with the use of the `rails generate controller` script. The syntax for the script is:

```
rails generate controller controller_name action [action2 action3 ...]
```

Rails Conventions

The name of the controller must be the **plural form** of the model.

The name of the action for showing the details of a single model is **show**.

By default, after processing a controller action, Rails will render the view file named `[action_name].html.erb` in the folder `app/views/[controller_name]/`. So for the show action of the products controller, it will render `app/views/products/show.html.erb`.

Given the conventions, here's the script for generating our controller and view:

```
$ rails generate controller products show
```

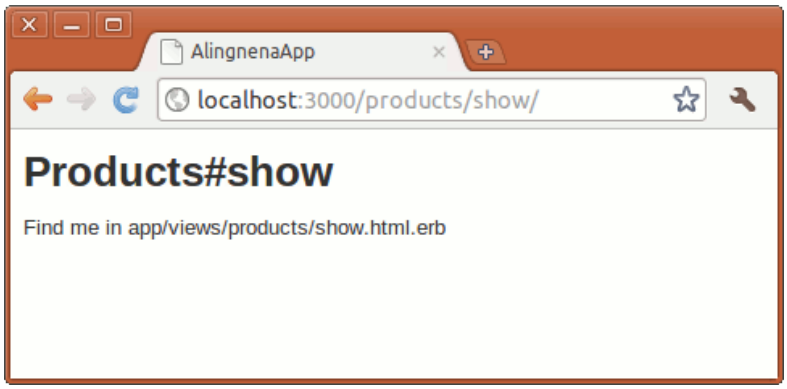
This script will create the controller `app/controllers/products_controller.rb`:

```
class ProductsController < ApplicationController
  def show
  end

end
```

and also the view `app/views/products/show.html.erb`:

```
<h1>Products#show</h1>
<p>Find me in app/views/products/show.html.erb</p>
```

You can now test the new controller and view by running the server (`rails server`) and going to http://localhost:3000/products/show. The following page should be displayed (you may have to restart the server through `rails server`):



The show action in the controller automatically refers to the `show.html.erb` file even without additional code as mentioned in the Rails conventions above.

We still have 2 problems at this point:

1. We still haven't displayed the details of the record.
2. The format of the URL is different from the convention used by the code generated from `rails`

`generate` scaffold i.e. http://localhost:3000/products/1.

The 2nd problem is simpler and deals with routing so let's tackle that first.

Routing

Routing deals with how different pages and controllers are linked to each other in a system. In Rails, there is a central configuration file that determines the available routes in the system. At first glance this may look like a violation of the Convention over Configuration mantra, but as we shall see much later, there are ways to let this routing file use convention to reduce the amount of configuration needed.

For this tutorial, we shall do away with the convention-related shortcuts so that you could understand how rails routes pages together. Take a look at the generated `config/routes.rb` file:

```
AlingnenaApp::Application.routes.draw do
  get "products/show"

  resources :debts

  # The priority is based upon order of creation:
  # first created -> highest priority.

  # Sample of regular route:
  #   match 'products/:id' => 'catalog#view'
  # Keep in mind you can assign values other than :controller and :action

  # Sample of named route:
  #   match 'products/:id/purchase' => 'catalog#purchase', :as => :purchase
  # This route can be invoked with purchase_url(:id => product.id)

  # Sample resource route (maps HTTP verbs to controller actions automatically):
  #   resources :products

  # Sample resource route with options:
  #   resources :products do
  #     member do
  #       get 'short'
  #       post 'toggle'
  #     end
  #
  #     collection do
  #       get 'sold'
  #     end
  #   end

  # Sample resource route with sub-resources:
  #   resources :products do
  #     resources :comments, :sales
  #     resource :seller
  #   end

  # Sample resource route with more complex sub-resources
  #   resources :products do
  #     resources :comments
  #     resources :sales do
  #       get 'recent', :on => :collection
  #     end
  #   end

  # Sample resource route within a namespace:
  #   namespace :admin do
  #     # Directs /admin/products/* to Admin::ProductsController
```

```
  #     # (app/controllers/admin/products_controller.rb)
  #     resources :products
  #   end

  # You can have the root of your site routed with "root"
  # just remember to delete public/index.html.
  # root :to => "welcome#index"

  # See how all your routes lay out with "rake routes"

  # This is a legacy wild controller route that's not recommended for RESTful applications.
  # Note: This route will make all actions in every controller accessible via GET requests.
  # match ':controller(/:action(/:id(.:format)))'
end
```

Ruby Corner – Comments

Single line comments in Ruby begin with a hash sign (#).

Multi-line comments in Ruby are rare, but FYI, they are enclosed between `=begin` and `=end`.

It can be a daunting file at first glance, but removing all of the comments, we can see that it's a pretty simple file:

```
AlingnenaApp::Application.routes.draw do
  get "products/show"

  resources :debts
end
```

Before we proceed with creating our routes, let's discuss first the idea behind the routing conventions in Rails: REST.

Representational State Transfer (REST)

Representational State Transfer (REST) is a scheme for communicating between clients and servers using HTTP. It was promoted as a stateless means of communication in a 2000 paper by Roy Fielding.

We will not go into detail about REST, and instead, we shall only focus on the parts that are relevant to us in Rails. In that regard, there are two main points about REST that we should know:

1. All resources are uniquely identified by a URL. For example, a product record can be identified with the following URL:

   http://my.url/products/1

   In contrast, an item done in a non-REST framework might look like this

   http://my.url/view_product.do

   In this case, all product records share the same URL and are differentiated only based on the user's state, either via session or cookie.

2. Actions done on resources are defined by the HTTP verb of the request. Here are some examples:

- `GET` http://my.url/products – *get* (a list of) all *products*
- `POST` http://my.url/products – *post* a new product to the list of *products*
- `GET` http://my.url/products/1 – *get* the *product* with the id of 1
- `PUT` http://my.url/products/1 – *put* an updated *product* with the id of 1 to the database
- `DELETE` http://my.url/products/1 – *delete* a *product* with the id of 1

The 5 HTTP commands in the 2nd point above are, by convention, how resources should be handled in Rails.

Now that we know the convention for the URL (http://localhost:3000/products/1) and the convention for the controller action (method must be named *show* for viewing records), we can now move on to connecting the two in our `config/routes.rb`.

Configuring Routing in routes.rb

Going back to the diagram in MVC, all browser requests to our Rails server goes to the routing component of the controller first. At this point, Rails checks entries inside the `routes.rb` file from top to bottom to see which controller and action to call.

We want to add a new "route" to the *show* action in the *products* controller and we can do this using the `match` declaration inside the routing block. The basic syntax of `match` is:

```
match(path_string, options_hash)
```

When Rails reaches a `match` line in the `routes.rb` file, it first checks the *path_string* if it matches the request's URL. The Ruby symbols in the *path_string* are placeholders and may match any part of the URL. When these symbols are matched to a part of the URL, they are converted into parameters inside a `params` hash. (See *Ruby Corner – Hash* below for a discussion on hashes)

For example, if you removed the commented from the last line in the default resource

```
match ':controller(/:action(/:id(.:format)))'
```

and then you called http://localhost:3000/products/show/1, Rails would match this request with the route `match ':controller(/:action(/:id(.:format)))'` producing the following parameters:

```
params = { :controller => 'products', :action => 'show', :id => 1 }
```

Based on this `params` hash, Rails can now route the request to the controller and action we created earlier.

We don't need to define the `:controller` and `:action` symbols within a route. We can specify them as defaults using the => form of match:

```
match('products/show' => 'products#show')
```

In this case, opening http://localhost:3000/products/show will make Rails use the controller and action defined in the right side, in this case, the `show` action of the `products` controller.

We can also use the `:via` option to constrain the request to one or more HTTP methods. For example:

```
match('products/show' => 'products#show', :via => :get)
```

Would make the route accessible only via the GET method. Note that the above route has a shorthand version, and it was inserted in our `routes.rb` when we used the generator script:

```
get "products/show"
```

At this point, we can create a route that matches Rails's conventions on “show record” pages:

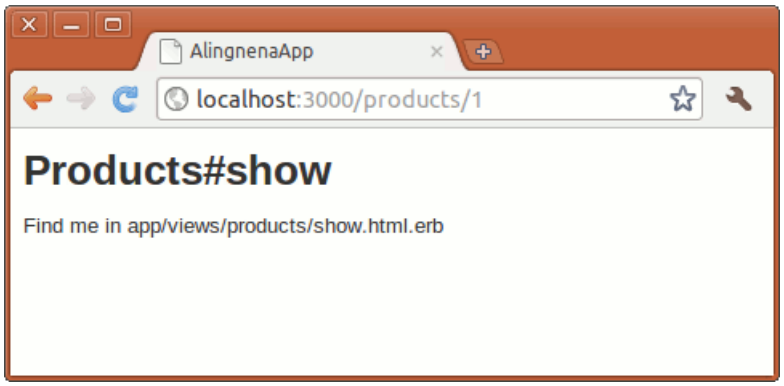
```
AlingnenaApp::Application.routes.draw do
  resources :debts

  match('products/:id' => 'products#show', :via => :get)

end
```

Going to http://localhost:3000/products/1 will create the following parameters:

```
params = { :controller => 'products', :action => 'show', :id => 1 }
```

As you can see, while the whole URL is matched by the path (`'products/:id'`) only the `:id` part is converted to a parameter. The rest of the parameters for controller and action are provided as defaults at the right of the => symbol.

Testing http://localhost:3000/products/1, we now see the following page:



We've completed the routing setup. We can now proceed to building the rest of the “Show Record” program.

Ruby Corner – Hash

Ruby has a built-in collection for storing key-value pairs, the hash. Ruby is a dynamically typed language so you can use any data type for the keys and values.

Hashes can be initialized by a pair of curly braces. Initial key-value pairs can be added using “=>”. For example:

```
my_hash = {}
another_hash = { :key => "value", 1 => "a number" }
```

One way of visualizing hashes is to think of them as lookup tables. Here's one for another_hash:

key	value
:key	"value"
1	"a number"

Values are retrieved and set with the use of square brackets ([]):

```
puts another_hash[:key]
my_hash["test"] = :value
```

Please refer to the API docs for other Hash methods.

Ruby Shortcuts

The curly braces are optional when the hash is the last argument in a method call. Ruby treats the following method calls

```
match(':controller/:action/:id', { :via => :get, :defaults => { :id => 1 } } )

match(':controller/:action/:id', :via => :get, :defaults => { :id => 1 } )
```

as both having only 2 arguments. We can remove the parenthesis to further reduce the characters in the method call:

```
match ':controller/:action/:id', :via => :get, :defaults => { :id => 1 }
```

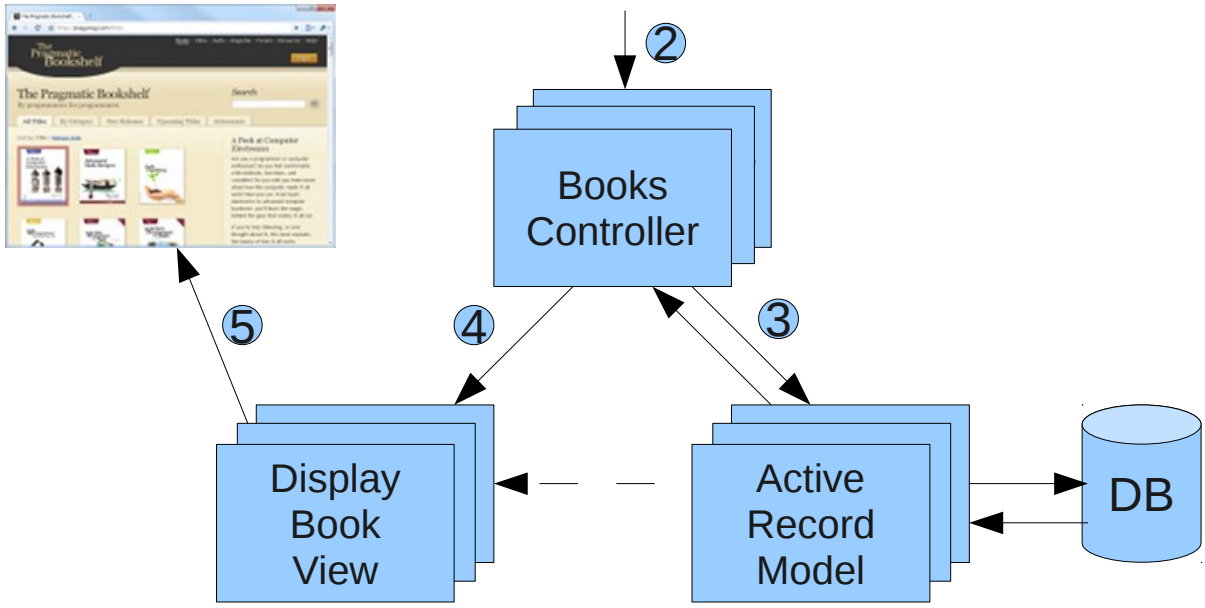
From this point on, we shall use this form for `match`.

Putting it all together: Retrieving a Record

To recall, our basic flow for this program is:

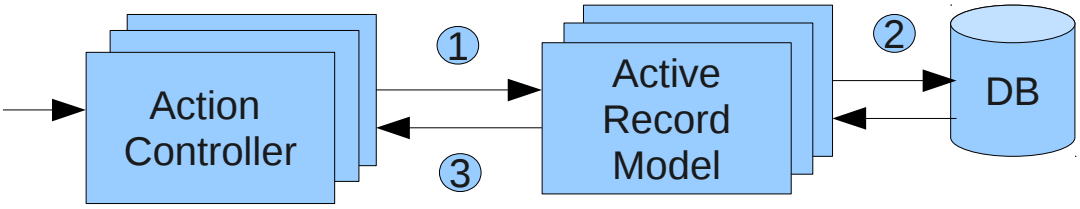
1. ~~User provides the id of the record via the URL~~ – already done
2. Program retrieves the record from database
3. Program renders the retrieved data into a web page and sends it to the user

Steps 2 and 3 can be better illustrated by the MVC diagram we had before for Rails:



Retrieve Record in the Controller using Model

According to the diagram, retrieving the data has 3 steps:



1. After the routing dispatches the request to the controller, the controller calls the model to retrieve the data.
2. The model retrieves the data from the database.
3. The model sends the data back to the controller.

The first half of the first step has already been done. All requests to `'products/:id'` are now dispatched to the *Products* controller's *show* method. All that is left to do is to actually utilize the model to retrieve data from the database. We can do the remaining steps by inserting the following line to the *show* method:

```
class ProductsController < ApplicationController
  def show
    @product = Product.find(params[:id])
  end
end
```

Retrieving the data is done using the `Product.find(id)` method, a class method in Active Record that retrieves a record with an `id` matching the passed `id` argument. In the statement above, this `id` argument is taken from the `params` hash passed to the controller via routing.

To pass the data to the controller, we store the result of the `find` method to an instance variable of the controller. This instance variable will be passed by the controller to the view in the next part of the process.

Ruby Corner – Class Methods

Class methods (aka static methods to Java people) are methods that can be called even without an instance of the class. `Product.find` above is a class method.

Class methods can be declared by adding "`self.`" before the method name. For example, the `up` and `down` methods in our migrations are declared as class methods that are eventually called by Rails.

```
class SomeMigration < ActiveRecord::Migration
  def self.up
  end

  def self.down
  end
end
```

Ruby Corner – Instance Variables

Instance variables are variables with an `@` symbol before the variable name. As expected from instance variables, their scope extends to the entire class.

Instance variables are not publicly accessible. They can only be accessed through accessor and mutator (getter and setter) methods. We can make it *look* like the instance variables are publicly accessible by the use of operator overloading:

```
class Square

  def side_length
    @side_length
  end

  def side_length=(new_length)
    @side_length = new_length
  end
```

```
end

s = Square.new
puts s.side_length         # returns nil
s.side_length = 10
puts s.side_length         # returns 10
```

Rails Conventions

You might have noticed that we didn't define the `id` field anywhere in our scripts or in our migration:

```
...
    create_table :products do |t|
      t.string :name
      t.text :description
      t.decimal :cost
      t.integer :stock

      t.timestamps
    end
...
```

This is because **by default, Rails adds "`id`", an auto-incrementing integer primary key field, to all tables**. By defining this field, developers don't need to worry about defining primary keys in their tables. Also, as we shall see later, this scheme makes table relationships easier to code.

Another thing to note is that **scripts that generate table migrations automatically add two timestamp fields, `created_at` and `updated_at`, through the `t.timestamps` declaration**. As their names imply, `created_at` contains the timestamp when the record was created, while `updated_at` contains the timestamp when the record was last updated.

Display Model data using View

We only have 2 more steps in the Rails MVC diagram:

1. The controller sends the data to the view.
2. The view renders the web page based on the data and sends it to the user.

Rails automatically handles the first step: when you pass control over to the view at the end of the action method, Rails gives the view a bunch of variables to work on:

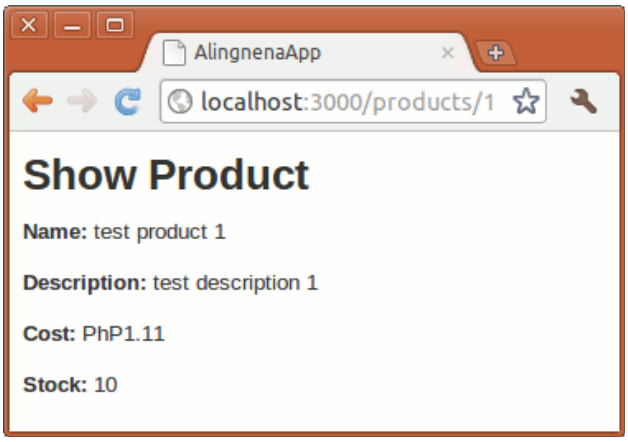
- special variables like the `params` hash are available in the view through accessor methods
- all instance variables in the controller are created instance variable counterparts in the view i.e. when you put values in the `@product` instance variable in the controller, you can access it in the view as `@product`

Now that we know that we can access `@product` in the view, the contents of our view should be easy to code. Replace the contents of `app/views/products/show.html.erb` to:

```
<h1>Show Product</h1>
```

```
<p>
  <b>Name:</b>
  <%= @product.name %>
</p>

<p>
  <b>Description:</b>
  <%= @product.description %>
</p>

<p>
  <b>Cost:</b>
  <%= number_to_currency(@product.cost, {:unit => 'PhP'}) %>
</p>

<p>
  <b>Stock:</b>
  <%= @product.stock %>
</p>
```

Opening http://localhost:3000/products/1 will give us the following result:



And with that, our “Show Product” page is now complete.

Ruby Corner – ERb expressions <%= %>

Our view uses Embedded Ruby (ERb) files to define the contents of the view. As the name suggests, these files are basic text files with some Ruby embedded in them to provide dynamic content.

In our example above, we used <%= %> to insert values into our page. When processing ERb files, Rails scans the files for <%= %> and replaces them with the value of the Ruby expressions inside them. For instance, you could add the following line to the view to print out the current date and time:

```
The time now is <%= Time.now %>
```

J2EE developers might be familiar with this construct: it's the same thing used to insert dynamic values in JSP pages. It's called an “expression” in the JSP world, so we'll use that term here too.

Ruby Corner – Helper Methods

Rails provides some helper methods to use inside a view. We used the h and the number_to_currency methods in our view above.

The `number_to_currency` method is self-explanatory; it simply converts the number provided to currency. It also provides some options for changing the currency symbol, the separator symbol, the format of the currency, etc.

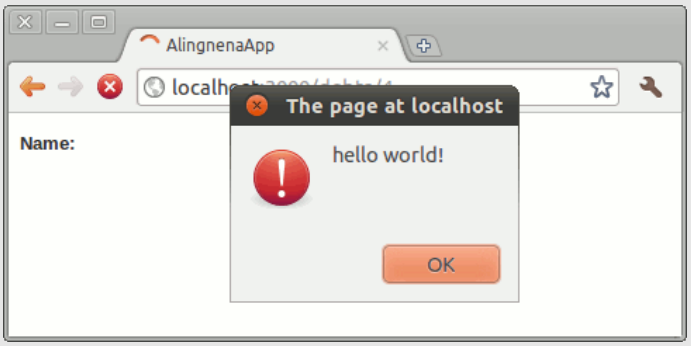
Ruby Corner – Escaping HTML characters

Rails automatically escapes HTML characters in views. Try visiting http://localhost:3000/products/2 and check the contents of the description field.

Rails provides a way of outputting raw data via the `raw` helper method. Try reloading the same page after enclosing the data with a `raw` method call:

```
<%= raw(@product.description) %>
```

Escaping HTML important in web pages: if we do not escape text, malicious users can insert HTML code into our pages and attack our users with Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) attacks. A simple example would be to go back into our Debts program and create a debt with "`<script>alert('hello world!')</script>`" as a customer or item, then add `raw` calls in either the index or the show pages.



Every user who views the item now has a JavaScript code run in their browsers. At this point, hacking the user's personal information should only take the hacker a few more lines of code. Therefore, one must always take care in using displaying raw output via the `raw` helper.

Shortcut Corner

Following our method shortening tips, we can shorten our expressions to:

```
<h1>Show Product</h1>
<p>
  <b>Name:</b>
```

```
    <%= @product.name %>
  </p>

  <p>
    <b>Description:</b>
    <%= @product.description %>
  </p>

  <p>
    <b>Cost:</b>
    <%= number_to_currency @product.cost, :unit => 'PhP' %>
  </p>

  <p>
    <b>Stock:</b>
    <%= @product.stock %>
  </p>
```

Adding a List Page

Let's move on to our List Products page. The approach is similar to the Show Product page, namely, we retrieve the records then render them in the view.

Preparing the New Files

We don't need to create or modify anything for the view so let's proceed with the controller.

Rails Conventions

For screens that list all products, the conventions are as follows:

- the controller action should be named **`index`**
- the URL should be **`/[`*`controller`*`]`** e.g. `/products`. Note the convention for controllers to be in plural form.

There's no script for updating the controller so we have to do the changes by hand. First, let's apply the first convention above to the controller and view. Insert a new action index to the controller (the order of the actions do not matter):

```
class ProductsController < ApplicationController

  def index
  end

  def show
    @product = Product.find(params[:id])
  end
end
```

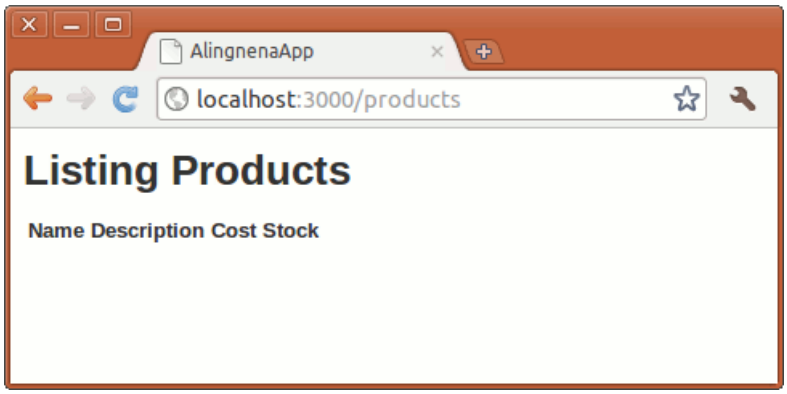
Let's also create a dummy view for the action, `/app/views/products/index.html.erb`:

```
<h1>Listing Products</h1>

<table>
  <tr>
    <th>Name</th>
    <th>Description</th>
    <th>Cost</th>
    <th>Stock</th>
  </tr>

</table>
```

For the 2nd convention, we add a new route in `config/routes.rb` for routing the request to the controller action:

```
AlingnenaApp::Application.routes.draw do
  resources :debts

  match 'products' => 'products#index', :via => :get
  match 'products/:id' => 'products#show', :via => :get

end
```

You can now verify the routing by going to http://localhost:3000/products.



Retrieving All Records

In "show product", we used the Active Record method `find()` to find a single record. For "list products", we shall use `all()`. Add the following line to the `index` action:

```
def index
  @products = Product.all
end
```

Using the `all()` class method instructs Active Record to retrieve all of records of the model. This variation of the method returns an array of objects instead of a single object.

Ruby Corner – Arrays

Arrays in Ruby are similar to arrays in C-based languages. They're zero-indexed (i.e. the first element is indexed at 0) and access and modification is done using square braces ([]):

```
puts array[3]   # outputs the contents of the 4th element of the array
array[1] = 10   # sets the 2nd element of the array to 10
```

Arrays can also be initialized using square braces:

```
my_array = [ "one", "two", "three", "four", "five", "six" ]
```

You can append new items into an array by using the << operator.

```
another_array = [ 10 ]
another_array << 20    # the array now contains [ 10, 20 ]
```

Like in hashes, you can put any combination of data types inside arrays:

```
# array contains a number, string, symbol and an array
mixed_array = [ 10, "twenty", :thirty, [ 40, 50 ] ]
```

Ruby includes some built in functions for arrays. Here are some examples:

```
puts an_array.size     # outputs the size of the array
an_array.delete_at(2)  # deletes the 3rd element of the array
```

Please refer to the API docs for other Array methods.

Displaying the Records

All we need to do now is to display the `@products` array in the view. This part is slightly complicated so it's OK to take a peek at how our Debts program handles it.

```
<% @debts.each do |debt| %>
  <tr>
    <td><%= debt.name %></td>
    <td><%= debt.item %></td>
    <td><%= debt.amount %></td>
    <td><%= debt.remarks %></td>
...
  </tr>
<% end %>
```

There are three new concepts in this code snippet alone. Let's discuss two of them before we proceed with adding the actual code our view.

Ruby Corner – Iteration

Ruby has `for` and `while` looping constructs but Ruby programmers rarely use them.

Why?

Because Ruby provides many ways of iterating through numbers and lists thanks to its functional programming influences. For example, a simple repetition task can be written as:

```
# print "hello world!" 10 times
10.times do
  puts "hello world!"
end
```

Iterating through a list of numbers can be written using:

```
# print numbers 1 to 10
1.upto(10) do |number|
  puts number
end

# print numbers 10 down to 1
10.downto(1) do |number|
  puts number
end
```

When you call `upto()` or `downto()`, the method passes the current number to the variable enclosed in the pipe symbols (`|`) every iteration. The iteration method for arrays is similar:

```
# print the contents of the array
an_array.each do |item|
  puts item
end
```

Please refer to the API docs for other iteration related methods. You can tell if a method is an iteration related method if the final parameter for it is a block. (As we shall see later, blocks can either be enclosed in curly braces or a `do-end` block.) For arrays, you can look at both the API for `Array` as well as the API for its included class `Enumerable`.

```
# print the contents of the array using Enumberable#each_with_index
an_array.each_with_index do |item, index|
  puts "The element at index #{index} is #{item}"
end

# adds 10 to each element in the array using Array#collect!
an_array.collect! do |item|
  item + 10
end
```

Ruby Corner – Scriptlets <% %>

We've already discussed how expressions (<%= %>) insert values into our ERb views. To insert actual code inside the ERb we can use the <% %> construct (note the lack of =). Here's a simple example, rendering the contents of a variable initialized within an ERb:

```
<% temp_variable = 20 * 1024 %>
<%= temp_variable %>
```

Code blocks are possible between these embedded ruby scripts. These blocks are rendered for every iteration. For example, this code renders the contents of an array in an unordered HTML list:

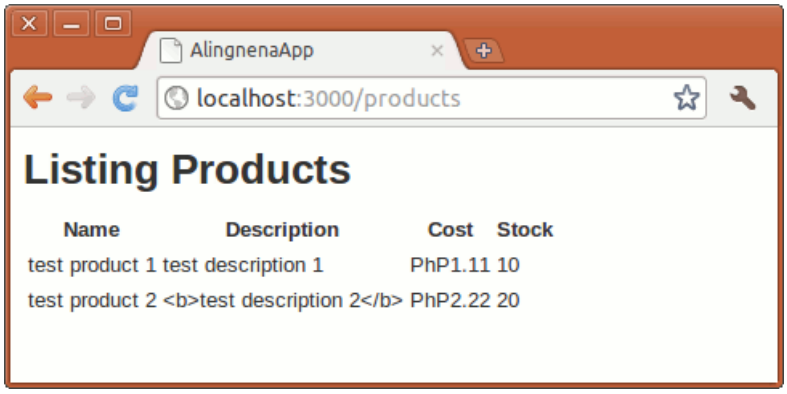
```
<ul>
<% an_array.each do |item| %>
  <li><%= item %></li>
<% end %>
</ul>
```

In J2EE / JSP terms, this construct is called a "scriptlet". Like the JSP expressions, this ERb construct shares the same symbols as the JSP scriptlet. Because of this, we also call them "scriptlets" in this manual.

Now that we are familiar with iteration and scriptlets, we can now code the rest of the `app/views/products/index.html.erb` file:

```
<h1>Listing Products</h1>

<table>
  <tr>
    <th>Name</th>
    <th>Description</th>
    <th>Cost</th>
    <th>Stock</th>
  </tr>

<% @products.each do |product| %>
  <tr>
    <td><%= product.name %></td>
    <td><%= product.description %></td>
    <td><%= number_to_currency product.cost, :unit => 'PhP' %></td>
    <td><%= product.stock %></td>
  </tr>
<% end %>

</table>
```

Once coded, we can now go to http://localhost:3000/products to view the "list products" page:



Since we're done with the “list products” page, we now have time to discuss the third concept introduced in the above code: Ruby Blocks and Procs.

Ruby Corner – Blocks and Procs

Blocks and Procs are the cornerstones of functional programming in Ruby. The way they are used in this programming language is so complex and elegant that explaining them in detail in this manual would be a wasted effort. Instead, we would refer you to the wonderful article over at http://eli.thegreenplace.net/2006/04/18/understanding-ruby-blocks-procs-and-methods/ for your reading pleasure.

Here's a very short explanation on what happens in `@products.each`:

Blocks are like methods, but they still don't have any bound data yet. They are declared using either

```
do |var1, var2, ...|
  ...
end
```

or

```
{ |var1, var2, ...| ... }
```

When a block is passed to a method, the method would bind the values to the block and execute its statements. In the case of the `each` method, that method binds a new value into the block (e.g. the `product` variable) every iteration then runs the block. This is how the looping behavior works.

Linking the List Page with the Show Page

Before we proceed with deleting records, let's add links between the index and show pages together first. It can be annoying to manually type `http://localhost:3000/products` and `http://localhost:3000/products/1` just to go between these two pages.

For this, we can use the `link_to` helper method. This method returns a link based on the arguments you pass it. The syntax is as follows:

```
link_to(name, options = {}, html_options_hash = nil)
```

The `name` argument is the text used in the link. The `options` argument can be three different things:

- a string – this will be the URL used by the link. As this is brittle for internal use (paths can be easily changed in the `routes.rb`), this is only used for external links.
- a hash – the link will be based on both the hash and the `routes.rb`.
- a non-hash, non-string object – Rails will create a URL based on this object, usually an Active Record. This will be discussed after we finish the entire Product program.

Finally, the `html_options_hash` is simply a hash of HTML attributes you wish to assign to the link. For example, this link has the CSS `class` and the `id` set to "external-link" and "google-search", respectively:

```
<%= link_to("Go to Google", "http://www.google.com",
            { :class => "hello world!", :id => "google-search"}) %>
```

The resulting link is:

```
<a href="http://www.google.com" class="hello world!" id="google-search">Go to
Google</a>
```

If the text is too long, you can use the block form of `link_to`:

```
link_to(options = {}, html_options_hash = nil) do
  # name
end
```

To use it, just put the text inside the block:

```
<%= link_to("http://www.google.com") do %>
  <strong>This link goes to Google</strong>
<% end %>
```

Now that we know how to use `link_to`, let's link the show and index pages. Insert the following line to `index.html.erb`:

```
...
<% @products.each do |product| %>
  <tr>
    <td><%= product.name %></td>
    <td><%= product.description %></td>
    <td><%= number_to_currency product.cost, :unit => 'PhP' %></td>
    <td><%= product.stock %></td>
    <td><%= link_to('Show', { :action => 'show', :id => product.id }) %></td>
  </tr>
<% end %>
...
```

And add the following line to the end of `show.html.erb`:

```
<%= link_to('Back to List of Products', { :action => 'index' }) %>
```

Shortcut Corner

link_to is just another method so we could use the usual shortcuts for Ruby methods like removing the parenthesis and the curly braces:

```
<td><%= link_to 'Show', :action => 'show', :id => product.id %></td>
```

The tricky part here is that the method accepts 2 different hashes so you might think the shortcut won't work. However, remember that Ruby parses the last series of hash key-value pairs as a single hash: if we want to pass 2 hashes, we just need to enclose the first hash in curly braces. For example, if we want to set the class of the link to the show product screen, we can use:

```
<td><%= link_to 'Show', { :action => 'show', :id => product.id }, :class => 'link' %></td>
```

Named Routes

There are a couple of problems with using an options hash in link_to. The most obvious problem is that it's too long. Even if the :controller and :action options can be derived from the current page, it's still too long for practical usage.

Another problem is that it's brittle. Suppose we have a link scattered around our system, say an approve account link. Initially our link goes to the approve_account action in the users controller so our links were generated using:

```
<%= link_to 'Approve Account', :controller => 'users', :action => 'approve_account' %>
```

Later, we changed the page that handles the account approval to the approve action of the accounts controller. We would have to manually search all instances of the approve account link and modify it accordingly.

```
<%= link_to 'Approve Account', :controller => 'account', :action => 'approve' %>
```

To remedy these two problems, Rails provides a way for us to name routes in the routes.rb file.

Here's how you modify our routes for index and show to become named routes:

```
AlingnenaApp::Application.routes.draw do
  resources :debts

  match 'products' => 'products#index', :via => :get, :as => 'products'
  match 'products/:id' => 'products#show', :via => :get, :as => 'product'

end
```

By adding an :as option, we tell Rails to provide two new helper functions based on the route:

- xxxx_path – returns the relative path of the matched route. For example, calling products_path will return /products
- xxxx_url – returns the complete URL of the matched route. For example, calling products_url will return http://localhost:3000/products

The path and URL are interchangeable in almost all cases. For this manual, we shall use the xxxx_path methods due to it being used in generator scripts. Here's the named route version of the link in

show.html.erb:

```
<%= link_to 'Back to List of Products', products_path %>
```

When the path in the route contains placeholder symbols, both methods accept arguments to fill up these placeholders. For example, if we want to generate the URL to a certain product, we can use:

```
...
<% @products.each do |product| %>
  <tr>
    <td><%= product.name %></td>
    <td><%= product.description %></td>
    <td><%= number_to_currency product.cost, :unit => 'PhP' %></td>
    <td><%= product.stock %></td>
    <td><%= link_to 'Show', product_path(product.id) %></td>
  </tr>
<% end %>
...
```

Shortcut Corner

You can directly provide Active Record objects when providing arguments for the named route helpers. The helpers will just retrieve the `id` of the record for you. For example:

```
...
    <td><%= product.stock %></td>
    <td><%= link_to 'Show', product_path(product) %></td>
  </tr>
...
```

Later, we shall see how we can directly use Active Record objects as arguments for `link_to`. This approach uses conventions in named routes. At this point, we can actually use our `Product` object because our named route matches the convention for routing.

```
    <td><%= link_to 'Show', product %></td>
```

Rails Applications Without Scaffold (part 2: Delete and Search)

Deleting a Record

Our next task is deleting a record. Deleting is similar to showing a record, the only differences being the HTTP verb involved and that instead of rendering the specified record, we destroy it.

That said, this task has three steps that we need to code:

1. Provide a way to call the controller action for deleting a product
2. Delete the product record in the controller action
3. Display the “list products” page after deleting the product

Rails Conventions

For screens that deletes records, the conventions are as follows:

- the controller action should be named `destroy`
- as mentioned in the REST discussion, the URL for delete is the same as the URL for show. Instead of using the default GET verb, we use the DELETE verb for deletion i.e. `DELETE /products/:id`

Creating a DELETE link

We already know how to create a GET link. In this part, we shall discuss how to create a DELETE link.

First off, let's create a new route for delete based on the conventions:

```
AlingnenaApp::Application.routes.draw do
  resources :debts

  match 'products' => 'products#index', :via => :get, :as => 'products'
  match 'products/:id' => 'products#show', :via => :get, :as => 'product'
  match 'products/:id' => 'products#destroy', :via => :delete

end
```

Here we see how Rails can differentiate between the HTTP methods. The destroy action will only be used when the HTTP method is DELETE.

With the route set, we can now modify our controller to add our new destroy action.

```
  def destroy
    # destroy the specified record
    redirect_to products_path
  end
```

The `redirect_to` method can be used to replace a view. Instead of rendering a view, it tells the user's

browser to redirect to another page, resulting in a fresh new request to the target page. In this case, we redirect the user to the "list products" page after deleting the record.

The parameters for redirect_to is the same as the 2nd parameter for `link_to` (they use the same helper method, `url_for`) so you can use string for direct URLs, hash for route based URLs, or objects for shortcuts.

Now that we've finished setting up the route and the controller, it's time to make the actual link. Add the following to `app/views/products/show.html.erb`:

```
...
</p>
<%= link_to('Delete this record', @product,
            { :confirm => 'Are you sure?', :method => :delete }) %> <br />
<%= link_to 'Back to List of Products', products_path %>
```

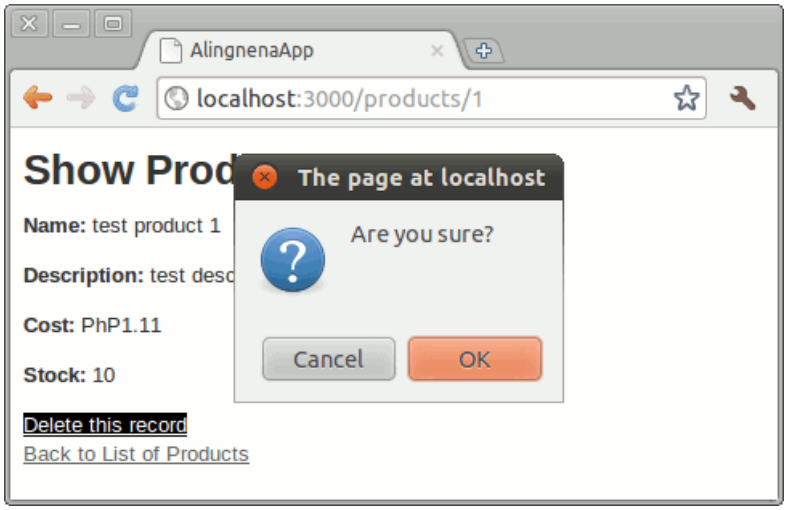
As you can see, this is practically the same as the "Show" link; we only added some new `html_options` to change the behavior of the link.

`link_to` has 3 special `html_options`:

- `:confirm` – you can set a confirmation question which will be prompted to the user on click of the link. The link is processed normally if the user accepts.
- `:method` – this accepts an HTTP verb symbol, either `:post`, `:put`, or `:delete`. With this set, clicking the link will call a JavaScript function which will submit a request to the specified URL with the specified method. If JavaScript is disabled, the link will fall back to GET.
- `:popup` – This will force the link to open in a popup window. By passing `true`, a default browser window will be opened with the URL. You can also specify an array of options that are passed-thru to JavaScript's `window.open` method.

By using `:confirm` and `:method`, we create a link that both prompts the user if he wants to continue and sends a request using the DELETE method.

Trying this out results in:



Clicking OK will redirect us to the “list products” screen, confirming that the link and route works.

Deleting the Product

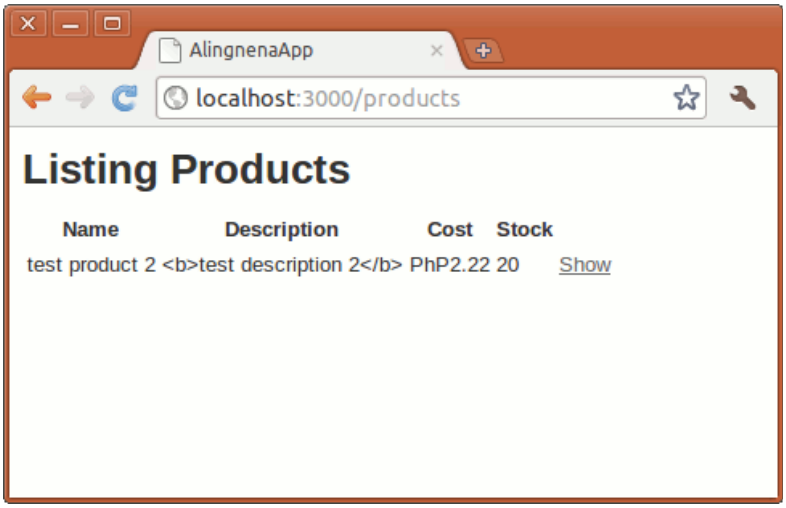
Now it's time to add the missing lines in our destroy method:

```
def destroy
  @product = Product.find(params[:id])
  @product.destroy
  redirect_to products_path
end
```

The first line retrieves the record in from the database. We've already seen this in our show method.

The second line deletes the record using the Active Record destroy method. Simple, huh?

Trying this out, we see:



Re-running a Migration

We're still going to add another feature to our destroy action, but before that, we're going to have to deal with a certain problem first: running out of test data. We only have two test records so we're bound to end up with an empty database when we continuously test our destroy action.

Thankfully, there's a command for re-running a migration:

```
> rake db:migrate:redo
```

This command rolls back the previous migration and runs it again. You can also specify the number of migrations to roll back and re-run by using the `STEP` option:

```
>  rake db:migrate:redo STEP=3
```

Note that re-running a migration will call the `self.down` method of the migration. This may or may not damage your data, but it's something to keep in mind.

Displaying a Message using Flash

The missing feature we're talking about earlier is the lack of confirmation messages. Right now, our program simply deletes the record and redirects the user without telling her if the operation was successful or not.

We use instance variables to pass data from the controller to the view. However, data in the instance variables are only present in the current request. A new request is generated when we use `redirect_to` so all of our instance variables are cleared. So for messages to be displayed after a redirect, we have to use another controller specific container: the `flash` hash.

Like the `params` hash, the `flash` hash is available for use in both the controller and the view. What's different is that items put inside the `flash` hash last until the next request.

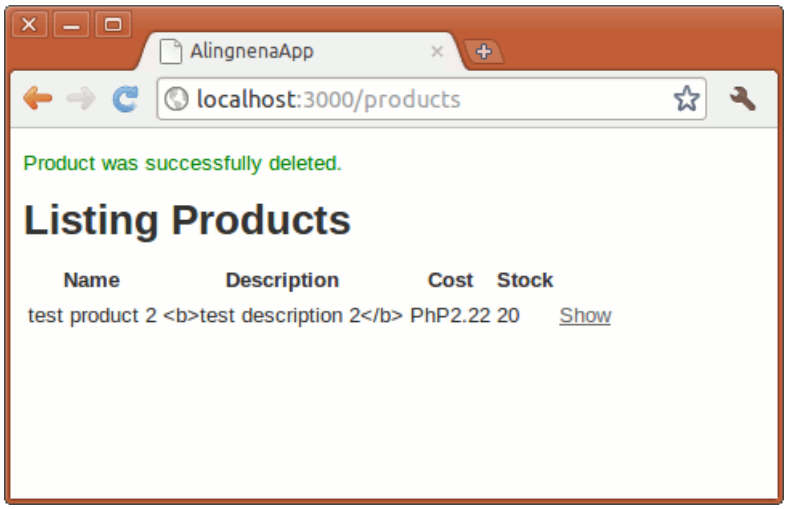
Here's how to modify our destroy action to include a `flash` message:

```
def destroy
  @product = Product.find(params[:id])
  @product.destroy
  flash[:notice] = "Product was successfully deleted."
  redirect_to products_path
end
```

Then we add an additional line at the top of `app/views/products/index.html.erb` to display the message (the scaffold stylesheet already includes `#notice`):

```
<p id="notice"><%= flash[:notice] %></p>

<h1>Listing Products</h1>
...
```

Trying the destroy link now results in:



Note that items in the `flash` hash last until the next request. If you don't redirect, a message will be present in both the current request *and* the next request. You can use `flash.now` instead of flash to set messages that are only available in the current request.

By the way, the most common types of flash messages are `:notice` (information messages, often in green) and `:alert` (error messages, often in red). We shall see these two again later.

Adding a Search Screen

Just as deleting records is similar to showing records, filtering records is also similar to listing records.

Here's the process for searching / filtering product records:

1. Submit a query to a search action
2. The search action retrieves records based on the query
3. Render the retrieved records using the view

In this part, we introduce two new concepts: generating HTML forms in the view, and using an Active Record dynamic finder for searching records. These two concepts deal with step 1 and 2, respectively, while we already know how to do the third step.

Rails Conventions

There is no built-in convention for search screens in Rails. However, since we are performing an operation to the entire table (searching), the request should be done in the `/products` path e.g. "`GET http://localhost:3000/products/search`". The controller action as well as the view file name should follow the URL.

As for the parameters, since this is just a GET operation, query strings can be used as they are normally used e.g. `http://localhost:3000/search?name=test`.

Preparation

The choice for the action name is arbitrary so let's just stick to "search". With that in mind, let's modify our controller and view accordingly. Let's start off with a new route:

```
AlingnenaApp::Application.routes.draw do
  resources :debts

  match 'products' => 'products#index', :via => :get, :as => 'products'
  match 'products/search' => 'products#search', :via => :get, :as =>
'search_products'
  match 'products/:id' => 'products#show', :via => :get, :as => 'product'
  match 'products/:id' => 'products#destroy', :via => :delete

end
```

We added a named route "search_products" pointing to our new action so that we could simply use `search_products_path` in our form later.

Let's add the search action to `app/controllers/products_controller.rb`:

```
  def search
    @products = Product.all
  end
```

We're not doing any filtering yet; we're just copying the approach in the index action. We'll continue with the index-cloning with `app/views/products/search.html.erb`:

```
<h1>Listing Products</h1>
```

```
<table>
  <tr>
    <th>Name</th>
    <th>Description</th>
    <th>Cost</th>
    <th>Stock</th>
  </tr>

<% @products.each do |product| %>
  <tr>
    <td><%= product.name %></td>
    <td><%= product.description %></td>
    <td><%= number_to_currency product.cost, :unit => 'PhP' %></td>
    <td><%= product.stock %></td>
    <td><%= link_to 'Show', product_path(product.id) %></td>
  </tr>
<% end %>
</table>
```

Yes, it's basically a copy of `app/views/products/index.html.erb`.

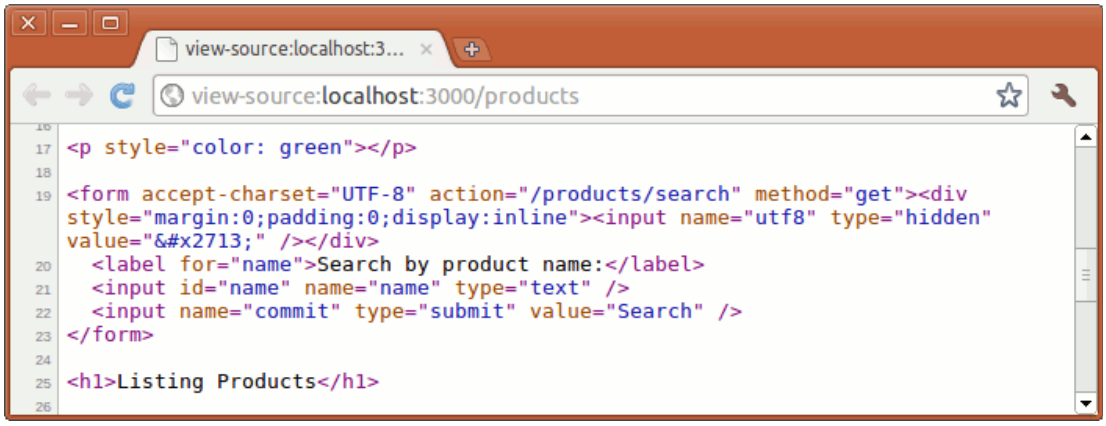
Adding the Search Form to List Products

As web developers, we already know how to tinker with the URL to pass query strings to the server. Our users obviously don't know how to do that manually so we have to provide a search form for them to use.

Insert the following lines to `app/views/products/index.html.erb`:

```
<%= form_tag(search_products_path, {:method => :get} ) do %>
  <%= label_tag("name", "Search by product name:") %>
  <%= text_field_tag("name") %>
  <%= submit_tag("Search") %>
<% end %>
```

If you view the source of the updated form, you will realize that every helper we used corresponds to an HTML form element.



The `form_tag` helper generates a form element, enclosing everything inside the block inside the form. Its syntax is (note that the tag must be in an expression, not a scriptlet):

```
<%= form_tag(url_for_options = {}, options = {}, *parameters_for_url) do %>
  ...
```

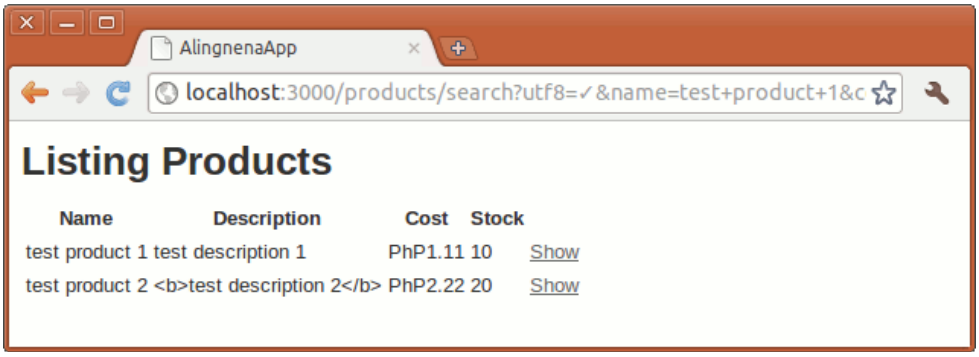
```
<% end %>
```

The `url_for_options` are the same as the options used in `link_to` and `redirect_to` (the string, hash, or object parameter in case you forgot). Like `link_to`, the second hash allows you to set HTML attributes for the form along with two additional options: you can set `:multipart` to true if you want to set the `enctype` to "`multipart/form-data`" (useful for submitting files), and you can set the `:method` to either `:get`, `:put`, `:post`, or `:delete` (`:post` is used by default). The `parameters_for_url` is simply a hash

Here's a quick rundown on the other form helpers we used in the search form:

- `label_tag` – creates a label. It has 3 parameters:
  - first parameter sets the for attribute
  - second parameter sets the text inside the label. Uses the value of the first argument if not specified.
  - third parameter is a hash for setting HTML attributes
- `text_field_tag` – creates a standard text field. Also has 3 parameters
  - first parameter sets the name attribute
  - second parameter sets the value of the field
  - third parameter is a hash for setting HTML attributes. You can also set additional options: you can set :disabled to true to disable the field, you can also set the :size to set the number of visible characters that will fit in the input, and you can also set the :maxlength which is the maximum number of characters that the browser will allow the user to enter.
- `submit_tag` – creates an HTML submit button. It only has two parameters: the first sets the text inside the button (default is "Save changes") and the second is an options hash for setting HTML attributes. Like with `link_to`, `form_tag` and `text_field tag`, this parameter has some special options:
  - `:confirm` – behaves the same way as the `:confirm` in link_to
  - `:disabled` – disables the button if set to true
  - `:disabled_with` – if this is set and the user clicks the button, the button will be disabled and the provided text (e.g. "Please wait...") will replace the button's original text while the form is being submitted. Useful for preventing the user from clicking the button twice on submit.

We only used one field in our form, the name field. Our plan is for this search module to return all records with a name that matches our submitted name. But enough talk, let's see our newly coded form in action:



As you can see, the get method puts the query string in the URL. We could verify that the parameters were passed by looking at the web server logs.

```
Started GET "/products/search?utf8=%E2%9C%93&name=test+product+1&commit=Search"
for 127.0.0.1 at [redacted] +0800
  Processing by ProductsController#search as HTML
  Parameters: {"utf8"=>"✓", "name"=>"test product 1", "commit"=>"Search"}
  Product Load (0.6ms)  SELECT "products".* FROM "products"
Rendered products/search.html.erb within layouts/application (19.8ms)
Completed 200 OK in 37ms (Views: 26.1ms | ActiveRecord: 0.6ms)
```

With this we know that we could access the search parameter "`name`" through the `params` hash entry `params[:name]`.

Shortcut Corner

We can shorten the entire form the same way we shortened our other model forms:

```
<%= form_tag search_products_path, :method => :get do %>
  <%= label_tag "name", "Search by product name:" %>
  <%= text_field_tag "name" %>
  <%= submit_tag"Search" %>
<% end %>
```

Filtering Results

So now we have the search parameter available in our controller. Let's now add the code which will filter the results accordingly. Replace the line in the search action with:

```
def search
  @products = Product.find_all_by_name(params[:name])
end
```

Here we used a *dynamic finder*. In addition to providing accessors for each field in the database, Active Record also provides class methods for retrieving filtered for each field. These methods are in the form of:

- `.find_by_xxxx()` – returns a record whose field `xxxx` matches the provided argument. Like `.find()`, it throws an error if no record is found. (When you think about it, find equivalent to `find_by_id`)

- `.find_all_by_xxxx()` - returns an array of records whose field `xxxx` matches the provided argument.

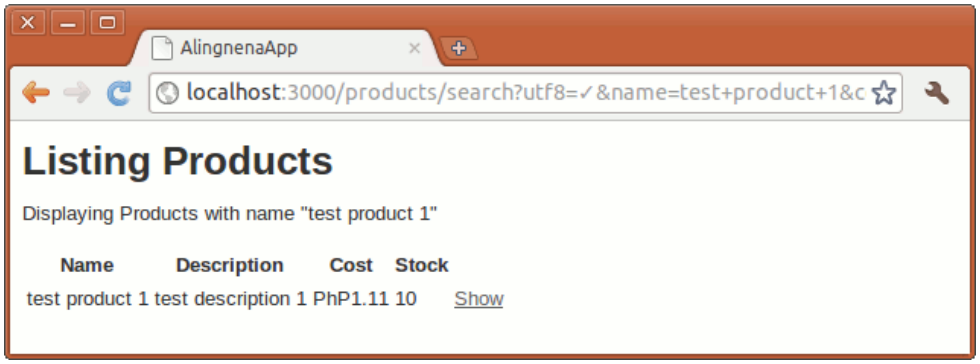
The search function should work properly at this point. Try using the search form at http://localhost:3000/products to see if it works.

Finishing Touches to the View

Let's make some minor changes to our search results screen to make it more functional.

```
<h1>Listing Products</h1>

<p>Displaying Products with name "<%= params[:name] %>"</p>

<table>
  <tr>
    <th>Name</th>
    <th>Description</th>
    <th>Cost</th>
    <th>Stock</th>
  </tr>

<% @products.each do |product| %>
  <tr>
    <td><%= product.name %></td>
    <td><%= product.description %></td>
    <td><%= number_to_currency product.cost, :unit => 'PhP' %></td>
    <td><%= product.stock %></td>
    <td><%= link_to 'Show', product %></td>
  </tr>
<% end %>
</table>

<p><%= link_to "Back to original list", products_path %></p>
```

The first change displays the current search criteria, while the next change links back to the original list, basically resetting the filter.



Rails Applications Without Scaffold (part 3: Insert and Update)

Creating a Record

So far we've encountered tasks that require one controller action and one view as well as a task that required only one action with no views (delete). For next two tasks, creating and updating a record, we will need two actions and one view:

- one action to prepare the form
- one view to render the form to the user
- one action to process the submitted form. This action will simply redirect in case of success or re-render the view in case of an error.

The flow for creating records is slightly more complicated than the previous functions so let's just discuss the flow as we go along.

Rails Conventions

Here are the conventions for screens that create records:

- the controller action that prepares the empty form should be named `new`. The view displaying the empty form also follows (**new.html.erb**)
- the URL to the empty form should be in the form `/[plural model]/new`, so in the `Product` case, this should be `/products/new`. This still follows REST; we are performing an action that affects the entire table.
- the controller action that processes the submitted form should be named `create`.
- the URL to the form processing action should not be `/[plural model]/create`, but `POST /[plural model]` i.e. `POST /products`. The base path is fine because the method tells the server that we are posting a new item to products.

Preparing the New Files

First step is to insert new actions into our controller. The pseudocode below explains the basic flow for this function:

```
def new
  # create a dummy Product
end

def create
  # create a Product based on submitted form
  # save Product
  # if save is successful
  #   show new Product
  # else
```

```
    #   go back to current page and display error
  end
```

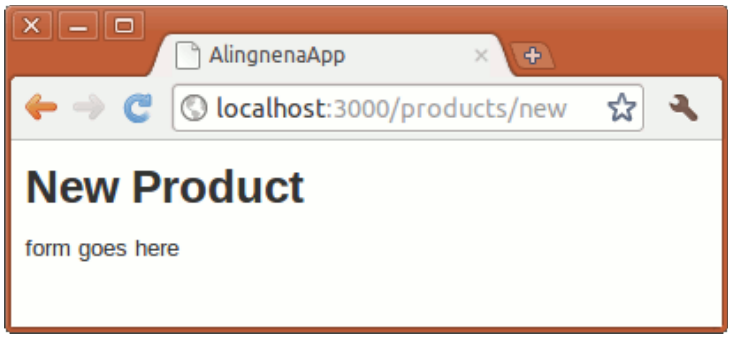
Let's also create a dummy `app/views/products/new.html.erb` so we could test the routing before we add the processing logic.

```
<h1>New Product</h1>

<p>form goes here</p>
```

Here are the new entries for our `routes.rb`:

```
AlingnenaApp::Application.routes.draw do
  resources :debts

  match 'products' => 'products#index', :via => :get, :as => 'products'
  match 'products' => 'products#create', :via => :post
  match 'products/new' => 'products#new', :via => :get, :as => 'new_product'
  match 'products/search' => 'products#search', :via => :get, :as =>
'search_products'
  match 'products/:id' => 'products#show', :via => :get, :as => 'product'
  match 'products/:id' => 'products#destroy', :via => :delete

end
```

Try out the new screen by going to `http://localhost:3000/products/new`:



Making the Create Form

Here's the rest of `new.html.erb`:

```
<h1>New Product</h1>

<%= form_for @product do |f| %>
  <div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :description %><br />
    <%= f.text_area :description %>
  </div>
  <div class="field">
    <%= f.label :cost %><br />
    <%= f.text_field :cost %>
  </div>
  <div class="field">
    <%= f.label :stock %><br />
    <%= f.text_field :stock %>
  </div>
```

```
  <div class="actions">
    <%= f.submit 'Create' %>
  </div>
<% end %>

<%= link_to 'Back', products_path %>
```

We're using a different form helper here; before, we used `form_tag` in our search screen, while here we used `form_for`. The former is called a non-model form because it doesn't use a model. On the other hand, `form_for` is a model form – it takes a model and uses it to populate the fields inside the form.

The syntax for form_for is as follows:

```
form_for(record_or_name_or_array, options = {}) do |f|
  ...
end
```

The `record_or_name_or_array` parameter determines the model/s used in the form. As the name implies, it accepts a single or an array of Active Record objects and also accepts a name, a symbol or a string that refers to the instance variable where the Active Record is stored. The options argument can accept two options:

- `:url` – the URL where the form is supposed to be submitted to. Normally, the URL is derived from the model in the first argument but we can override that by using this option. The expected value for this option is the same expected values for `link_to` and `redirect_to`, namely, either a string, a hash, or an object (i.e. this option also uses `url_for`).

- `:html` – you can pass a hash of HTML attributes to this option

Inside the block are model form helper counterparts to the non-model form helpers we used in search. There are some noticeable differences, though:

- They use symbols for their first parameter. This signifies that they refer to a specific field in the model provided in the `form_for` call.

- They are prefixed by an "`f.`". When the block is called by form_for, form_for passes an object that refers to the model of the form. When we prefix a form helper with the "`f.`", we say that the field is a member of the form.

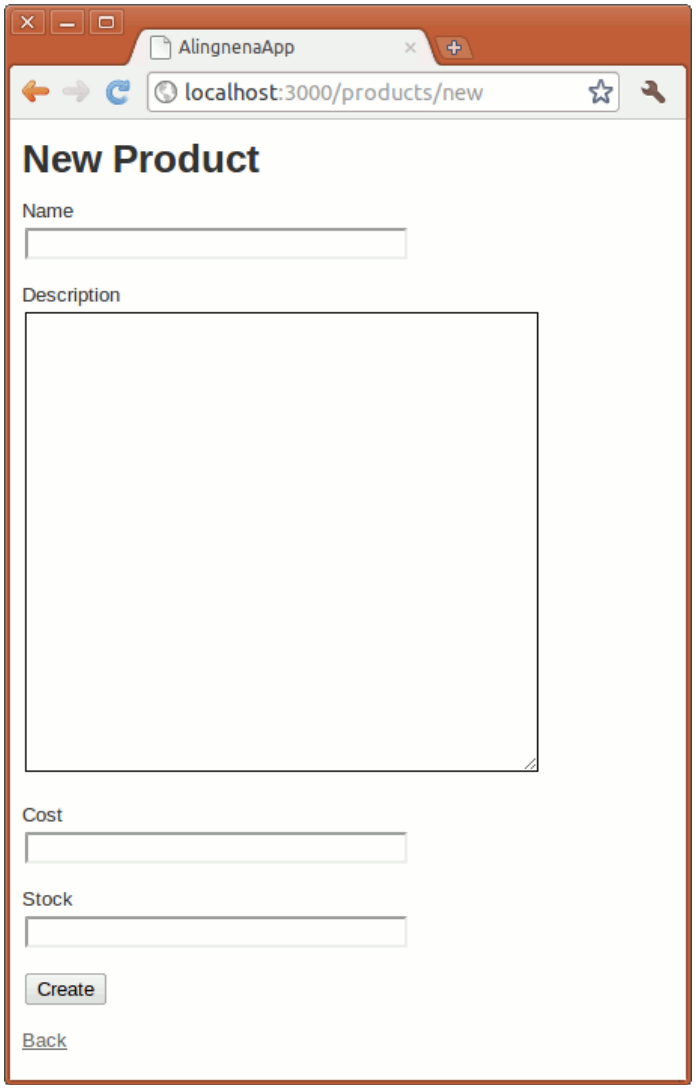
  We can do away with the `f` prefix, but doing that would require us to tell the form helpers what the model should they refer to. e.g.:

  ```
  <%= text_field(:products, :name) %>
  ```

You may notice that we haven't initialized the `@product` instance variable yet. Let's do just that in the new action:

```
def new
  @product = Product.new
end
```

Opening http://localhost:3000/products/new will give us a blank form.



`Product.new` is the constructor of the `Product` class. We can even define the fields at this point passing a hash of "field name – value" pairs. Try replacing `Product.new` with `Product.new(:name => "dummy name")`. You should see the name field in the form populated by "dummy name".

Saving the New Record

When we click on submit, the contents of the form are submitted and Rails converts the form into a single hash parameter.

```
Started POST "/products" for 127.0.0.1 at 2011-02-22 02:37:45 +0800
  Processing by ProductsController#create as HTML
  Parameters: {"utf8"=>"✓", "authenticity_token"=>"RDf30n5ghYVgvJI8/xDYUZA6sltQi
HUKLtOEnjtdOEY=", "product"=>{"name"=>"Cola", "description"=>"fizzy drink", "cos
t"=>"25", "stock"=>"100"}, "commit"=>"Create"}
Completed   in 29ms
```

This hash is now part of the `params` hash and can be accessed directly like any hash entry. Here's a visual representation of `params`:

key	value
:commit	Create
:authenticity_token	...
:product	<table><tr><th>key</th><th>value</th></tr><tr><td>:name</td><td>Cola</td></tr><tr><td>:description</td><td>fizzy drink</td></tr><tr><td>:cost</td><td>25</td></tr><tr><td>:stock</td><td>100</td></tr></table>

Here we can see that we could access the value “Cola” via `params[:product][:name]` or the whole product hash via `params[:product]`.

Now the only thing left to do is code the logic that we wrote in pseudocode, namely, to create a record based on the submitted form, save that record, then redirect the user based on the results of the save.

The first part can be done by using Product's constructor

```
def create
  @product = Product.new(params[:product])

  # save Product
  # if save is successful
  #   show new Product
  # else
  #   go back to current page and display error
end
```

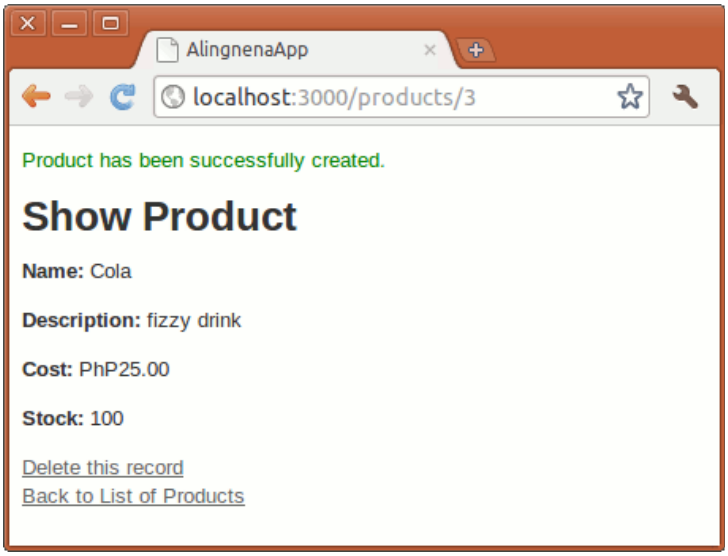
Recall that the constructor accepts a hash and fills out the fields based on that hash. Passing params[:product] (which is a hash) creates a new Product object based on the submitted form.

The saving and redirection can be done via the following code:

```
def create
  @product = Product.new(params[:product])

  if @product.save
    flash[:notice] = "Product has been successfully created."
    redirect_to @product
  else
    render :action => "new"
   end
end
```

Here we called save, an instance method that saves the record to the database. This method returns true if the record passes validation and false otherwise.

On a successful save in our code, the user is redirected to the “show product” page of the newly created Product record. Try using the new product form now.



(To display the success notice in the show product page, add the following line from `app/views/products/index.html.erb` to `show.html.erb`:)

```
<p id="notice"><%= flash[:notice] %></p>
```

If the save fails, it calls the `render` method to render the view of the `new` action. By rendering the view of the new action, the user is presented a form that contains the data that he submitted along with the error messages that comes along with it. This is the reason why we initialize the `@product` variable in the new action instead of initializing it directly at `new.html.erb`.

Unfortunately, we can't see this in action because we haven't set validation rules for Product yet.

Ruby Corner – Decision Making

The `if` statement in Ruby looks like the following:

```
if expression then
  ...
end
```

The `then` keyword is optional unless the whole statement is in a single line

```
if expression then ... end
```

Adding else and else-if in the statement gives us the following:

```
if expression [then]
  ...
elsif expression [then]
  ...
else
  ...
end
```

You can also use `unless` which evaluates the statement if the expression is false:

```
unless expression [then]
  ...
else
  ...
end
```

Both `if` and `unless` can also be used as statement modifiers where the statement would be run if their conditions are met. For example:

```
return true if x > max_value

error_count += 1 unless form.is_valid?
```

Note that as we mentioned before, everything evaluates to `true` except for `nil` and `false` in Ruby.

Shortcut Corner

As mentioned before, `:alert` and `:notice` are so common that the Rails team decided to add convenient ways to set and retrieve these flash messages.

In `redirect_to`, there's the special options `:alert` and `:notice` which allows you to set `flash[:alert]` and `flash[:notice]`, respectively:

```
redirect_to @product, :notice => "Product has been successfully created."
```

There are also `notice` and `alert` helpers that serve a similar purpose:

```
<p id="notice"><%= notice %></p>
```

Also in `redirect_to`, you could use the `:flash` option to set values to any key in `flash`:

```
redirect_to @product, :flash => { :notice => "Product has been successfully created." }
```

Basic Validation

Let's add some basic validation to our Product model. Add the following line to `app/models/product.rb`:

```
class Product < ActiveRecord::Base
  validates_presence_of(:name, :description)
end
```

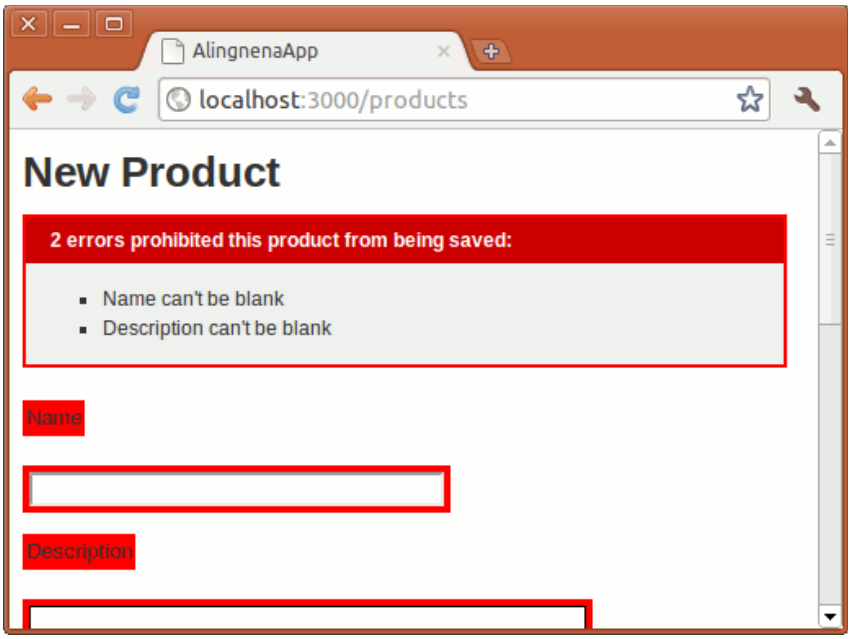
This method call to `validates_presence_of` tells Rails to check whether the specified fields are present when the object undergoes validation. In other words, it defines the mandatory fields for the model.

We also add the following lines to `new.html.erb` to display the error messages:

```
<h1>New Product</h1>
<%= form_for @product do |f| %>
  <% if @product.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@product.errors.count, "error") %> prohibited this product from being
saved:</h2>
```

```
      <ul>
      <% @product.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
      </ul>
    </div>
  <% end %>
  <div class="field">
    <%= f.label :name %><br />
...
```

Try submitting a blank new product form now.



You are redirected to `new.html.erb` along with messages detailing the errors in your form. As you may have guessed, the error messages at the top are the work of the `f.error_messages` call.

Before we end this part, add a link to the "new product" page in the "list products" page (`index.html.erb`):

```
...
</table>

<p><%= link_to "Create a new Product", new_product_path %></p>
```

We're done with creating new product records and we only have one more function to code.

Shortcut Corner

Since the validation is just a method call, we could also write it as:

```
class Product < ActiveRecord::Base
  validates_presence_of :name, :description
end
```

Editing a Record

Editing a record is not that different from creating a record. We still have two controller actions sharing one single view which contains the form; we just don't create the record from scratch this time.

Rails Conventions

Here are the conventions for editing records:

- the controller action that prepares the editing form should be named **`edit`**. The view displaying the empty form also follows (**edit.html.erb**)
- the URL to the form should be in the form `/[plural model]/[id]/edit`, so in the `Product` case, this should be **`/products/:id/edit`**. This RESTful URL tells us that we are doing an action to a particular product resource.
- the controller action that processes the submitted form should be named **`update`**.
- the URL to the form processing action is similar to the one we used in showing and destroying a record: **`PUT /products/:id`**.

Routes and View

Insert the following new entries in `routes.rb`:

```
AlingnenaApp::Application.routes.draw do
  resources :debts

  match 'products' => 'products#index', :via => :get, :as => 'products'
  match 'products' => 'products#create', :via => :post
  match 'products/new' => 'products#new', :via => :get, :as => 'new_product'
  match 'products/search' => 'products#search', :via => :get, :as => 'search_products'
  match 'products/:id' => 'products#show', :via => :get, :as => 'product'
  match 'products/:id' => 'products#destroy', :via => :delete
  match 'products/:id' => 'products#update', :via => :put
  match 'products/:id/edit' => 'products#edit', :via => :get, :as => 'edit_product'
end
```

Create the view file for edit, `app/views/products/edit.html.erb`:

```
<h1>Edit Product</h1>

<%= form_for @product do |f| %>
  <% if @product.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize @product.errors.count, "error" %> prohibited this product from being
saved:</h2>

      <ul>
      <% @product.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= f.label :name %><br />
```

```
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :description %><br />
    <%= f.text_area :description %>
  </div>
  <div class="field">
    <%= f.label :cost %><br />
    <%= f.text_field :cost %>
  </p>
  <div class="field">
    <%= f.label :stock %><br />
    <%= f.text_field :stock %>
  </div>
  <div class="actions">
    <%= f.submit 'Update' %>
  </div>
<% end %>

<%= link_to 'Show', @product %> |
<%= link_to 'Back', products_path %>
```

We did not need to change the declaration for `form_for`, it already checks the model and determines which route to use based on conventions. If the model is new (i.e. `create`), it will use the `products_path`. Otherwise (i.e. `update`), it will use `product_path`.

For convenience, add the following links to `show.html.erb`

```
...
</p>

<%= link_to 'Edit this record', edit_product_path(@product) %> <br />
<%= link_to 'Delete this record', @product,
  :confirm => 'Are you sure?', :method => :delete %> <br />
<%= link_to 'Back to List of Products', products_path %>
```

and `index.html.erb`

```
...
  <% @products.each do |product| %>
    <tr>
      <td><%= product.name %></td>
      <td><%= product.description %></td>
      <td><%= number_to_currency product.cost, :unit => 'PhP' %></td>
      <td><%= product.stock %></td>
      <td><%= link_to 'Show', product_path(product.id) %></td>
      <td><%= link_to 'Edit', edit_product_path(product) %></td>
    </tr>
  <% end %>
...
```

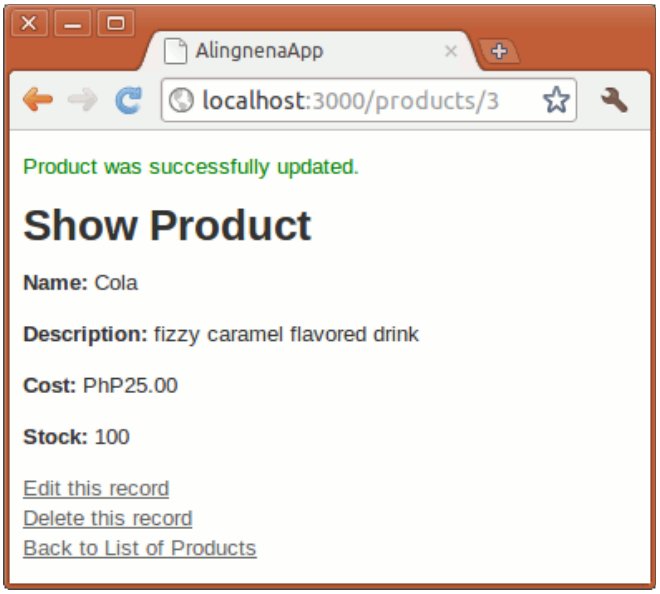
New Controller Actions

Add the following actions to `app/controllers/products_controller.rb`:

```
  def edit
    @product = Product.find(params[:id])
  end

  def update
    @product = Product.find(params[:id])
```

```
    if @product.update_attributes(params[:product])
      redirect_to @product, :notice => 'Product was successfully updated.'
    else
      render :action => "edit"
    end
  end
```

Since we're already familiar with how records are created, this code should be easy to understand. Instead of creating records with `Product.new`, we use `Product.find` to retrieve the record from the database. The `@product.update_attributes()` method is similar to `save`; it first updates the fields based on the hash then validates and saves the object.



And that's it; we finally created a full table maintenance program without using `rails generate scaffold`. We also learned almost everything that goes on behind the scaffold generated code.

There are still a bunch of things we add to our maintenance program. We'll spend the next chapter on those things in order to get them out of the way before the next lesson.

Rails Applications Without Scaffold (part 4: RESTFul Routes, Callbacks, Filters, and Layout)

In this chapter, we will be discussing new Rails features from the Model, Controller, and the View that can make our Product maintenance program better.

Reduce Route Entries with Resource-Based Routing

Creating a new route entry for typical routes sort of defeats the purpose of following *Convention over Configuration*. Fortunately, most of the routing in this program was only done for educational purposes; there is a shortcut for telling Rails that a certain resource follows the REST conventions.

```
AlingnenaApp::Application.routes.draw do
  resources :debts

  match 'products/search' => 'products#search', :via => :get, :as => 'search_products'
  resources :products

end
```

Using `resources :products` adds the following routes and helpers to the system:

helper	HTTP verb	URL	controller	action	use
products_url products_path	GET	/products	Products	index	Display a list of all products
new_product_url new_product_path	GET	/products/new	Products	new	Return a form for creating a new product
	POST	/products	Products	create	Create a new product
product_url product_path	GET	/products/:id	Products	show	Display a product
edit_product_url edit_product_path	GET	/products/:id/edit	Products	edit	Return a form for editing a product
	PUT	/products/:id	Products	update	Update a product
	DELETE	/products/:id	Products	destroy	Delete a product

You can define other routes under the resource by using the `collection` and `member` blocks. Use `collection` for routes that affect the entire collection. In our case, we can set the products search as a collection route:

```
AlingnenaApp::Application.routes.draw do
  resources :debts

  resources :products do
    collection do
      get 'search'
    end
  end
```

```
end
```

Doing this will also create `search_products_xxxx` route helpers (we manually set that named route before).

For actions that only affect one resource, use the `:member` option. For example, if we want to add a "suspend" action for suspending a specific product (i.e. `PUT /products/:id/suspend`), we can use:

```
AlingnenaApp::Application.routes.draw do
  resources :debts

  resources :products do
    collection do
      get 'search'
    end
    member do
      put 'suspend'
    end
  end

end
```

The other options for resource-based routing will be discussed in a later chapter.

Perform Actions While A Model is Saved using Callbacks

Aling Nena has dozens, if not hundreds, of different products in stock. If she would encode all of them, she might not have time to enter detailed descriptions for each item. To help her in this task, we could just set the description to be the same as the product's name every time she doesn't enter a description.

At first glance, the solution to this problem would be in the controller, say, something like this:

```
 def create
    @product = Product.new(params[:product])

    if @product.description.blank?
      @product.description = @product.name
    end
...
```

However, if you would recall our discussion in MVC, most of the business logic must be placed inside the model instead of the controller. So how do we set the description without changing the controller?

For this, we need to understand how to use callbacks.

Callbacks

Callbacks are methods called by an Active Record object before and after the object is inserted, updated, or deleted from the database. Here is the list of callbacks performed in each of these operations:

Creating a new record	Updating a record	Deleting a record
before_validation	before_validation	
before_validation_on_create	before_validation_on_update	
VALIDATION OPERATION		
after_validation	after_validation	
before_save	before_save	
before_create	before_update	before_destroy
INSERT OPERATION	UPDATE OPERATION	DELETE OPERATION
after_create	after_update	after_destroy
after_save	after_save	

Based on this table, the `before_validation callback` is called before the validation of both record creation and update. After that, the `before_validation_on_create` and `before_validation_on_update` are called in creation and update, respectively.

Registering Callbacks

Registering callbacks is similar to setting validations. Instead of providing fields, however, we provide symbols referring to method names.

```
class Product < ActiveRecord::Base
 validates_presence_of :name, :description

 before_validation :assign_default_description

 private

   def assign_default_description
     if description.blank?
       self.description = name
     end
   end
end
```

Here we set the description field of the record before the record is validated (thus preventing an error in description validation).

Note that we set the callback method to be private. This prevents the method from being called outside the object. Another thing to note is the use of `self` to denote that we are modifying the instance attribute; if we didn't use `self.description`, Ruby will think we are simply setting a local variable description.

Another way of registering callbacks is to override the callback itself. The following variation of the callback usage above works fine:

```
class Product < ActiveRecord::Base
  validates_presence_of :name, :description

  def before_validation
    if description.blank?
      self.description = name
```

```
    end
  end
end
```

Ruby Corner – Private Methods

Methods inside a Ruby class are public by default. You can define method visibility by using `public`, `private`, and `protected` keywords: all methods defined below them will use the said keyword for visibility.

In our case above, the method `assign_default_description` has become `private` because it is declared below the `private` keyword.

Rails Corner – Active Support Ruby Extensions

In Ruby, there are two ways to check if an object is `nil`:

```
object == nil

object.nil?
```

The nil? method works because `nil` is an object in Ruby (and not a null pointer) so it has methods of its own. Only the `nil` object returns true when `nil?` is called.

In our code above we used "`blank?`" to check if the string is empty. This method is not part of the methods of a `String` object in the core Ruby library, but an extension done by Rails to the `String` object via the Active Support libraries.

Thanks to the dynamic nature of Ruby, other libraries can choose to extend the functionality of core classes. It might look dangerous, but when done right, the new methods can make coding more convenient to developers.

We'll discuss in a later chapter some other extensions Rails made to Ruby.

Limiting User Access with Filters and Authentication

Aling Nena wants her customers to see the list of available products in her store. However, she doesn't want them to create and edit the products, or worse, delete the products. In short, she wants to add an authentication mechanism in her site.

Controller Filters

Authentication is like validation; we want to validate whether the user has permission to do something in our application. Unlike validation, it is not done in the Model level but in the Controller level, with the help of the controller filters.

Controller filters are more like callbacks than validations. They are methods that we define to be executed before actions (using `before_filter`) or after actions (using `after_filter`). The syntax is also similar:

```
class ProductsController < ApplicationController
  before_filter :check_if_aling_nena

  ...

  private

   def check_if_aling_nena
     #checking here
   end
end
```

In this case, the method `check_if_aling_nena` is called before every action in the Products controller. If we want to limit the actions where the filter is applied, we can use the `:only` option.

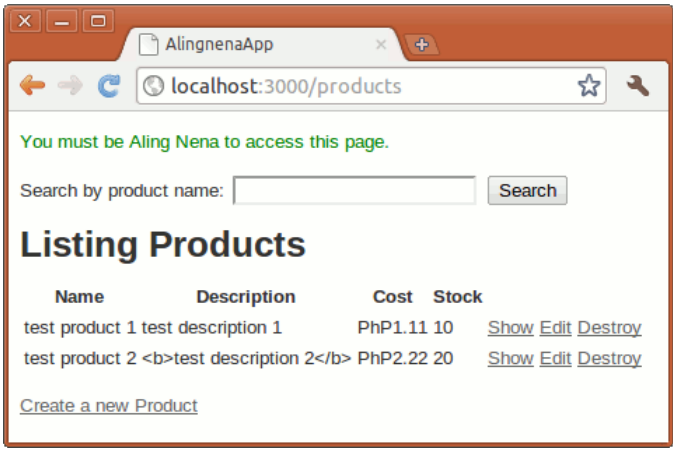
```
before_filter :check_if_aling_nena, :only => [:new, :create, :edit, :update, :destroy]
```

It might make sense to use the :except option here, though.

```
before_filter :check_if_aling_nena, :except => [:index, :show, :search]
```

One important thing about `before_filter` methods is that the processing will not continue to the action if the method renders or redirects. Here's one way to test our new filter, by kicking everyone back to the list products page regardless of who they are:

```
  def check_if_aling_nena
    redirect_to products_path, :notice => "You must be Aling Nena to access this page."
  end
```



(You may want to change that flash message to :alert, just don't forget to add the code to index that would display the message.)

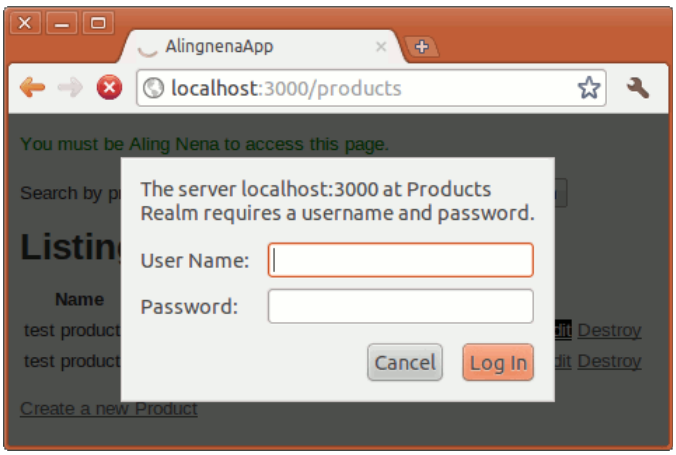
HTTP Authentication

Having verified that the filter works, let's now add the authentication.

Rails has built in methods that make applying basic HTTP authentication easy. Here's the code to apply that authentication to our Product maintenance program (you can change the username and password as you like):

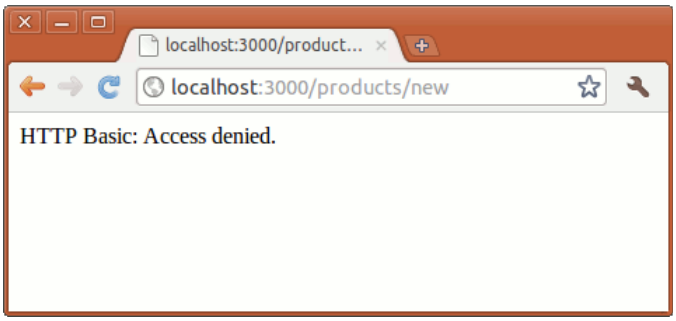
```
def check_if_aling_nena
  authenticate_or_request_with_http_basic("Products Realm") do |username, password|
    username == "admin" and password == "sTr0NG_p4$swOrD"
  end
end
```

You should see browser-specific pop-up prompting you for credentials:



Entering the credentials will let you proceed to the pages. Since HTTP authentication is supported by all browsers, the server knows if you already provided your credentials so you don't have to re-enter it again when you enter the other actions in the program.

Pressing Cancel will redirect you to an error page:



HTTP Authentication is the most basic of all authentications available to Rails. For larger applications with many users, you're better off with full-fledged authentication solutions like the Authlogic or Devise plugins.

Layouts and Rendering in the View

Asset Tags

Static files in our web application are stored in the `public` folder. Since this folder is mapped directly to the application root, going to `http://locahost:3000/` will open `public/index.html`.

By convention, stylesheets, images, and JavaScript files are stored in the `public/stylesheets`, `public/images`, and `public/javascripts` folder, respectively. Rails provides view helpers called asset tags for linking to these files.

We can use **`stylesheet_link_tag`** to link to CSS files under the `public/stylesheets` folder. The syntax is as follows:

```
stylesheet_link_tag( *sources, options = {} )
```

You can pass one or more stylesheets to the method:

```
<%= stylesheet_link_tag("main") %>

<%= stylesheet_link_tag("sidebar", "admin") %>
```

The code above creates the following links:

```
<link href="/stylesheets/main.css" media="screen" rel="stylesheet" type="text/css" />

<link href="/stylesheets/sidebar.css" media="screen" rel="stylesheet" type="text/css" />
<link href="/stylesheets/admin.css" media="screen" rel="stylesheet" type="text/css" />
```

You can also use relative paths:

```
<%= stylesheet_link_tag("red/main") %>

=> <link href="/stylesheets/red/main.css" media="screen" rel="stylesheet"
type="text/css" />
```

You can also use absolute or external paths:

```
<%= stylesheet_link_tag("http://example.com/main") %>

=> <link href="http://example.com/main.css" media="screen" rel="stylesheet"
type="text/css" />
```

The :all option creates links to all of the CSS files under `public/stylesheets`. Adding a `:recursive => true` option creates links for all CSS files in all subfolders of `public/stylesheets`.

```
<%= stylesheet_link_tag(:all) %>

<%= stylesheet_link_tag(:all, {:recursive => true}) %>
```

You can also override the link attributes by adding them as options. For example, you can change the media from screen to print using the following:

```
<%= stylesheet_link_tag("main", { :media => "print"}) %>
```

The asset tag for JavaScript is **`javascript_include_tag`** and it behaves the same way as the `stylesheet_link_tag`:

```
<%= javascript_include_tag("main") %>

=> <script type="text/javascript" src="/public/javascripts/main.js"></script>
```

The `:all` and `:recursive` option also work with `javascript_include_tag`.

For images, the asset tag is `image_tag`. The basic use is similar to the two previous asset tags. By default it appends `.png` to the file name if the extension isn't specified.

```
<%= image_tag "banner" %>

=> <img src="/public/images/banner.png" />

<%= image_tag "profile.jpg" %>

=> <img src="/public/images/profile.jpg" />
```

In addition to the HTML attributes that you can set in the options hash parameter, image_tag has the following special options:

- `:alt` – If no alt text is given, the file name part of the source is used (capitalized and without the extension)
- `:size` – Supplied as “{Width}x{Height}”, so “30x45” becomes width=“30” and height=“45”. :size will be ignored if the value is not in the correct format.
- `:mouseover` – Set an alternate image to be used when the `onmouseover` event is fired, and sets the original image to be replaced `onmouseout`. This can be used to implement an easy image toggle that fires on `onmouseover`.

Layouts

We now know how to link to CSS files in our pages but having to add `<%= stylesheet_link_tag "scaffold" %>` (the CSS used in Debts) to all of our pages can be a hassle, not to mention that it violates the *DRY* (*Don't Repeat Yourself*) principle that we've been following as we applied *Convention over Configuration* to our program. There must be a way for us to apply HTML code to all of our pages.

Thankfully, Rails already provides a way to do that through *layouts*.

Layouts, as you may have guessed, provide the general layout for our views. Rails combines the view and the layout when rendering the page to the user. An example of a layout is the default layout generated when we created our application, `app/views/layouts/application.html.erb`:

```
<!DOCTYPE html>
<html>
<head>
  <title>AlingnenaApp</title>
  <%= stylesheet_link_tag :all %>
  <%= javascript_include_tag :defaults %>
  <%= csrf_meta_tag %>
</head>
<body>

<%= yield %>

</body>
</html>
```

The `yield` section defines where the view will be inserted upon rendering. You can also use multiple yielding regions:

```
<html>
  <head>
  <%= yield :head %>
  </head>
  <body>
  <%= yield %>
  </body>
</html>
```

To render content in a named `yield`, you use the `content_for` method:

```
<% content_for :head do %>
  <title>A simple page</title>
<% end %>

<p>Hello, Rails!</p>
```

This would result in:

```
<html>
  <head>
  <title>A simple page</title>
  </head>
  <body>
  <p>Hello, Rails!</p>
  </body>
</html>
```

By default, Rails checks the `app/views/layouts` directory for a layout that matches the name of the controller. If the layout isn't found, Rails will try `app/views/layouts/application.html.erb`.

There are two other ways to define layouts. First is through the controller:

```
class ProductsController < ApplicationController
  before_filter :check_if_aling_nena, :except => [:index, :show, :search]
  layout "main"
```

```
...
```

This will apply `app/views/layouts/main.html.erb` to all actions in Products.

Layouts are shared downwards in the hierarchy, so applying

```
class ApplicationController < ActionController::Base
  protect_from_forgery

  layout "main"
end
```

will set the layout of every controller to `app/views/layouts/main.html.erb` because all controllers (at least the ones generated by `rails generate controller` and `scaffold`) are subclasses of `ApplicationController`.

Another way of setting the layout is through the render method. Before, the only option we discussed for render is `:action`, so now we introduce a new option, `:layout`, which lets you define the layout of a rendered view:

```
render :layout => "main", :action => "new"
```

You can also remove a layout by setting it to `false`. For example:

```
render :layout => false, :action => "new"
```

When multiple layouts are applied to a view, more specific layouts always override more general ones. For example:

```
class ProductsController < ApplicationController
  layout "main"

  def show
    @product = Product.find(params[:id])
    render :layout => false
  end
...
```

No layout is rendered for the show action above. (Recall that the `:action` option is automatically set by Rails according to the action name if it's not defined.)

Partial Templates

When you look at `app/views/products/new.html.erb` and `app/views/products/edit.html.erb`, you will notice that both share almost the same code for the form. We can eliminate this redundancy by using partial templates, better known as partials.

Partials are fragments of pages that can be called by other pages. We use the `render` method to tell rails to render a partial inside a view. For example, if we call the following inside `app/views/products/new.html.erb`:

```
<%= render({:partial => 'form'})%>
```

The call will insert the contents of `/app/views/products/_form.html.erb` at that line inside the view.

We can use the `:locals` option to pass local variables to the partial. For example, we can eliminate the

redundancy between new and edit by modifying the views:

```
# new.html.erb

<h1>New Product</h1>

<%= render({:partial => "form",
            :locals => { :product => @product, :button_label => "Create" } }) %>

<%= link_to 'Back', products_path %>

# edit.html.erb

<h1>Edit Product</h1>

<%= render({:partial => "form",
            :locals => { :product => @product, :button_label => "Update" } }) %>

<%= link_to 'Show Details', @product %> |
<%= link_to 'Back to List', products_path %>
```

and creating a new partial `/app/views/products/_form.html.erb`:

```
<%= form_for product do |f| %>
  <% if product.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize product.errors.count, "error" %> prohibited this product from being
saved:</h2>

      <ul>
      <% product.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
      </ul>
    </div>

  <% end %>
  <div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :description %><br />
    <%= f.text_area :description %>
  </div>
  <div class="field">
    <%= f.label :cost %><br />
    <%= f.text_field :cost %>
  </div>
  <div class="field">
    <%= f.label :stock %><br />
    <%= f.text_field :stock %>
  </div>
  <div class="actions">
    <%= f.submit button_label %>
  </div>
<% end %>
```

In this case, the `product` local variable was derived from the `@product` instance variables, while the `button_label` was supplied in the hash.

All partials have a local variable with the same name as the partial and you can pass an object to this by

using the `:object` option. In other words, suppose we make a copy of the `_form` partial at `_product_form.html.erb`, and we replace the first line with:

```
<%= form_for product_form do |f| %>
```

We would be able to do this in `edit.html.erb`:

```
<%= render({:partial => "product_form",
            :object => @product,
            :locals => { :button_label => "Update" } }) %>
```

The instance variable is placed inside the `product_form` variable.

We can also use partials to render a collection of objects using the `:collection` option. For instance, we can replace the iteration inside `app/views/products/index.html.erb` with the following:

```
...
  <tr>
    <th>Name</th>
    <th>Description</th>
    <th>Cost</th>
    <th>Stock</th>
  </tr>
  <%= render({ :partial => "item", :collection => @products }) %>
</table>
...
```

And create a new partial, `_item.html.erb`:

```
    <tr>
      <td><%= item.name %></td>
      <td><%= item.description %></td>
      <td><%= number_to_currency item.cost, :unit => 'PhP' %></td>
      <td><%= item.stock %></td>
      <td><%= link_to 'Show', product_path(item.id) %></td>
      <td><%= link_to 'Edit', edit_product_path(item) %></td>
    </tr>
```

The page would still display as it was originally displayed. Each item in the array is placed in a local variable that shares the same name as the partial. In this case, the partial and variable name is `item`.

We can even define a spacer template for the collection:

```
<%= render({ :partial => "item", :collection => @products, :spacer_template => "hr" }) %>
```

Here's a sample `_hr.html.erb`:

```
<tr><td colspan="7"><hr /></td></tr>
```

There is also a shorthand for partials. Assuming `@product` a single `Product` object the following code:

```
<%= render({ :partial => @product }) %>
```

will render `_product.html.erb` and pass `@product` as the value of the local variable product. Similarly, assuming `@products` is a collection of `Product` objects, the following code:

```
<%= render({ :partial => @products }) %>
```

will render the partial `_product.html.erb` for each `Product` object inside the collection, passing the

object to the local variable product.

Using a collection with different object types can produce polymorphic results. For example, if `employee1` and `employee2` are `Employee` objects, while `customer1` and `customer2` are `Customer` objects, the following call:

```
<%= render({ :partial => [customer1, employee1, customer2, employee2] }) %>
```

will call the following partials, either `_customer.html.erb`

```
<p>Name: <%= customer.name %></p>
```

or `_employee.html.erb`:

```
<p>Name: <%= employee.name %></p>
```

Shortcut Corner

Don't forget you can always remove the parenthesis from a method call, or the curly braces from a hash if it's the last hash argument for a method. There are also short-hand default versions of rendering partials. Here are some examples:

```
# <%= render :partial => "form",
#            :locals => { :product => @product, :button_label => "Update" } %>
<%= render :partial => "form", :product => @product, :button_label => "Update" %>

# <%= render({ :partial => @products }) %>
<%= render @products %>
```

Associations

When you're working with RDBMSs, you're bound to encounter cases where you have to handle relationships between tables. One-to-many relationships are common, though you might see many-to-many and one-to-one relationships from time to time.

Does Rails support these relationships?

No and yes.

Rails does not support the various relationship implementations between database vendors. You will have to tweak your migrations to call specific DDL commands in order to set them up (this is outside the scope of this training course).

Rails, however, has its own database-agnostic way of handling entity relationships. In this chapter, we will discuss how to these various relationships are coded in Rails.

Migrations for Associations

If you would recall in our previous lesson, by default all tables in a Rails application use "`id`", an auto-incrementing integer field, as their primary key. In the context of associations, Rails uses this convention to link tables in a system.

> **Rails Conventions**
>
> If a table refers to another table, say an Employee belonging to a Department, the former must have a field that refers to the latter. **This foreign key reference field should be an integer field in the format `[model_name]_id`.**
>
> In the Employee-Department case, our `Employees` table should have a integer field named `department_id`. This field would contain the `id` of the Department where the Employee belongs to.

When generating the migration for a foreign key reference field, you can specify the field normally e.g.

```
$ rails generate migration Employee name:string department_id:integer
```

Or you can make the abstraction clearer by using the `:references` data type:

```
$ rails generate migration Employee name:string department:references
```

This migration will still generate the same field as the previous migration. Note the lack of `_id` in the `:references` argument.

Defining Associations in Models

Had you used `rails generate scaffold` or `rails generate model` instead of `migration`, you will notice that the model generated has a `belongs_to` call:

```
class Employee < ActiveRecord::Base
  belongs_to :department
end
```

Again, another human-readable declaration in Rails: it simply tells Active Record that each Employee record belongs to a Department record.

belongs_to

For every `belongs_to` call inside the model, Active Record generates some instance methods for the class. In the Employee-Department case, these methods are

- `employee.department` – returns the Department object associated to the Employee. This is cached so if you want to reload the list, just pass `true` to the method. Can also be used to assign a Department to the Employee e.g. `employee.department = some_department`

- `employee.build_department(attribute_hash)` – basically a shorthand for `employee.department = Department.new(attribute_hash)`. Also returns the newly created Department.

- `employee.create_department(attribute_hash)` – same as `build_department`, the only difference is the new object is saved (assuming it passes validations, of course).

Even with these methods, using the `department_id` for defining associations is still valid. The following will work:

```
department = Department.find_by_name("Accounting")
employee.department_id = department.id
```

but it's still better to use an approach with a clearer abstraction:

```
employee.department = Department.find_by_name("Accounting")
```

not to mention coding the `build_xxxx` and `create_xxxx` methods manually will take at least 2 more lines of code than just calling those methods.

One-to-One Relationships

With the `belong_to` declaration in one model, we can define the nature of the association in the target table. Let's start with the one-to-one association.

For this tutorial, we're going to expand Aling Nena's system by adding a Purchase entity whose records refer to a single purchase of items from a supplier. We're going to have an Invoice record for each Purchase, and this one-to-one relationship will be the basis of this tutorial.

First, let's setup the Purchase and Invoice entities with scaffold scripts:

```
$ rails generate scaffold purchase description:text delivered_at:date
$ rails generate scaffold invoice purchase:references reference_number:string
$ rake db:migrate
```

This should make working maintenance programs for each entity. When you look at the `new` and `edit` views of the Invoice program, however, you will see that it uses a `text_field` for the purchase. This is inappropriate: not only does it force the user to know the Purchase while editing the field, it also won't work because the data types don't match (`text_field` is String, while the purchase field requires a Purchase).

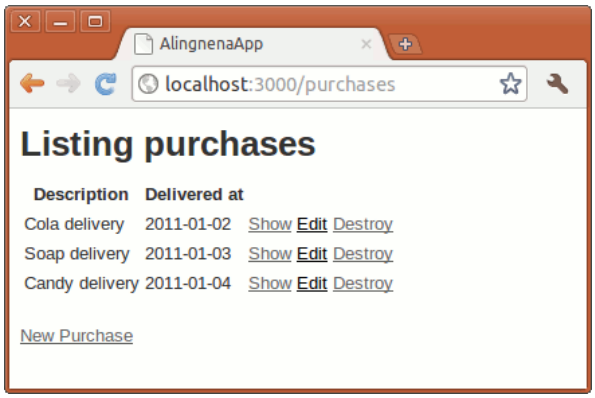
Drop Down List

A better approach would be to provide a drop-down field containing a list of available Purchases. The form helper for drop-down lists is `collection_select`

```
collection_select(object, field_name, collection, value_method, text_method, options = {},
                  html_options = {})

# or the form_for shortcut

f.collection_select(field_name, collection, value_method, text_method, options = {},
                    html_options = {})
```

The `object`, `field_name`, and `options` field are just the same as in our basic input field `text_field`, with the options separated from the `html_options`.

The collection is a list of items where the value and the text are derived from, while the `value_method` and `text_method` determines the names of the methods to call for the value and text. For example, we have the following list of Purchases:



Replacing the purchase field in app/views/invoices/_form.html.erb to

```
...
    </div>
  <% end %>

  <div class="field">
    <%= f.label :purchase_id, "Purchase" %><br />
    <%= f.collection_select :purchase_id, Purchase.all, :id, :description %>
  </div>
  <div class="field">
    <%= f.label :reference_number %><br />
    <%= f.text_field :reference_number %>
  </div>
...
```

will result in the following HTML code:

```
...
  <div class="field">
    <label for="invoice_purchase_id">Purchase</label><br />
    <select id="invoice_purchase_id" name="invoice[purchase_id]">
      <option value="1">Cola delivery</option>
      <option value="2">Soap delivery</option>
      <option value="3">Candy delivery</option>
    </select>
  </div>
...
```

There are three available options in collection_select:

- :include_blank – adds a blank entry at the top of the list. Can be the text of the entry (e.g. "none") or true (which will just put a blank entry).

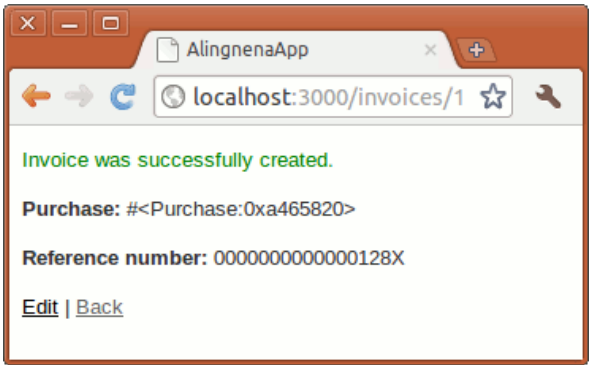
- :prompt – just like include_blank, but has a different purpose: to prompt the user to select one item in the list. When set to true, the text used is "Please select". Setting this option will *not* add validation for checking if the user chose a value from the list. Setting this option along with :include_blank will add 2 blank list entries in the list, one for each call with the prompt on top.

- :disabled – contains a list of items from collection that should be disabled from the list. In the case of collection_select, we can use a Proc to determine which should be disabled. For example, to disable all Purchases that would be delivered in the future, we can set:

```
 <%= f.collection_select :purchase_id, Purchase.all, :id, :description, :prompt => true, :disabled =>
lambda { |purchase| purchase.delivered_at > Date.today } %>
```

Custom Helpers

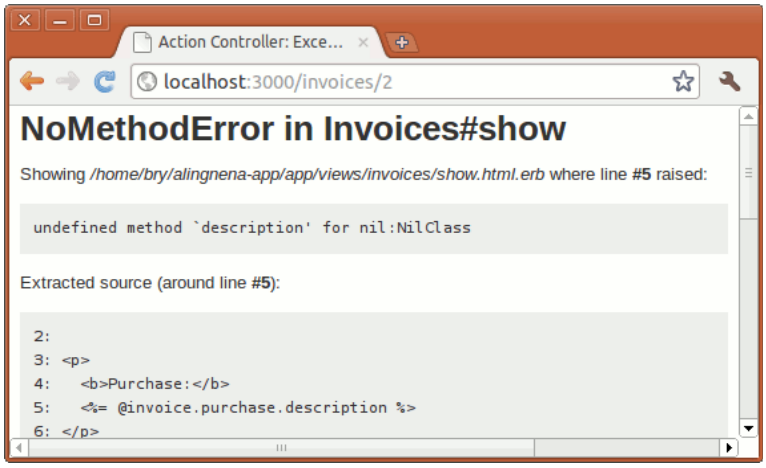
After creating the Invoice for the "Cola delivery", we see a problem in the Show Invoice page:



What is displayed is the internal representation of the Purchase object. Sure, we can remedy this by replacing the code with:

```
<p>
  <b>Purchase:</b>
  <%= @invoice.purchase.description %>
</p>
```

But what if we had created an Invoice without a Purchase?



We get Ruby's equivalent to a Null Pointer Exception.

There are some ways to remedy this problem. The most obvious would be to make the field mandatory through validations. But for entities that allow that foreign key field to be empty, we need to directly address the problem.

One way would be to add a special method inside the model. For example:

```
class Invoice < ActiveRecord::Base
  belongs_to :purchase

  def display_purchase
    return purchase.description unless purchase.nil?
    "(no Purchase set)"
  end
end
```

Then just call `@invoice.display_purchase` from the view. This approach is legal and is actually the preferred approach For educational purposes, however, let's move the logic to the view instead:

```
<p>
  <b>Purchase:</b>
  <% unless @invoice.purchase.nil? %>
    <%= @invoice.purchase.description %>
  <% else %>
    (no Purchase set)
  <% end %>
</p>
```

Our problem this time is that we're adding 5 lines of processing code to our view for a simple task. If we do this a lot, our view would look cluttered and may bring back bad memories of old JSP, ASP, PHP code from older developers. We can take out this code from our views through the **helper files**.

The helper files are located at the app/helpers folder. These are modules where you can add methods that you would use in your views. One helper is created per controller if you generated them through `rails generate` controller or scaffold. We can move our logic from the view into the helper by adding the following lines to `app/helpers/invoices_helper.rb`:

```
module InvoicesHelper
  def display_purchase(invoice)
    unless invoice.purchase.nil?
      invoice.purchase.description
    else
      "(no Purchase set)"
    end
  end
end
```

and just call the helper in our view:

```
<p>
  <b>Purchase:</b>
  <%= display_purchase(@invoice) %>
</p>
```

Due to the "`helper :all`" call in `ApplicationController`, all helper methods from all helpers are available to all views. The file scheme used is just for developers to classify the helpers according to the program they are used.

Helpers are usually used for generating markups. For instance, we want to have the Purchase field to have a link to the original purchase, we can use:

```
module InvoicesHelper
  def display_purchase(invoice)
    unless invoice.purchase.nil?
      link_to invoice.purchase.description, invoice.purchase
    else
      "(no Purchase set)"
    end
  end
end
```

You can't do this inside the model, and this gives helpers an advantage over the model approach.

has_one

We've already associated Invoice with Purchase but not the other way around. In order for us to be able to call methods like `@purchase.invoice`, we must first declare the association in the Purchase model. Given that we're using a one-to-one relationship, the association declaration will be:

```
class Purchase < ActiveRecord::Base
  has_one :invoice
end
```

With the has_one method declared, we can now call the following methods in Purchase:

- `purchase.invoice`, `purchase.invoice=`
- `purchase.build_invoice`, `purchase.create_invoice`

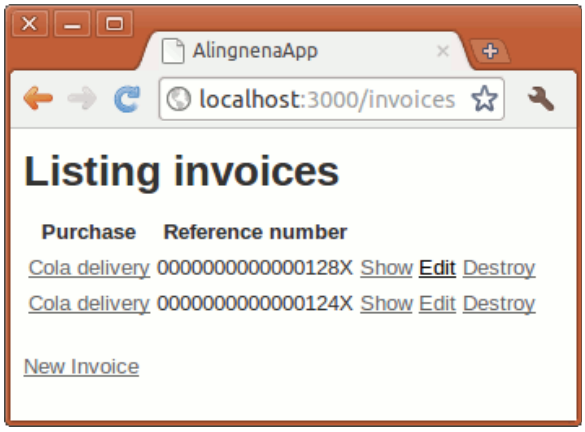
Yes, these are the same methods as the ones in the `belong_to` model.

You can provide options in the `:has_one` declaration. Here are a few:

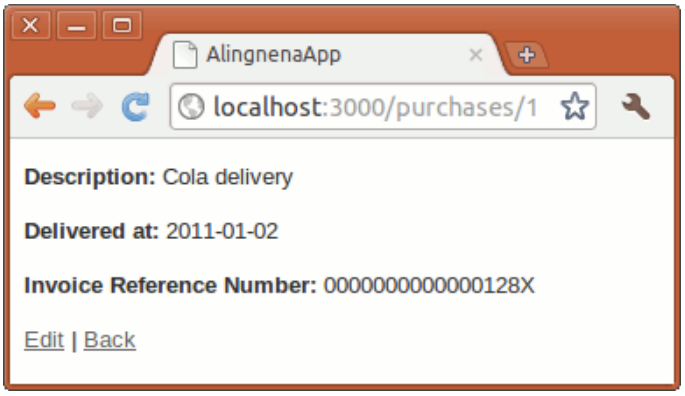
- `:dependent` – provides the behavior when this model is deleted i.e. cascading delete options. Setting it to :nullify simply sets the foreign key reference in the other entity to `NULL`. Using `:destroy` will call the :destroy of the other entity, while using `:delete` will simply delete the related record via SQL.
- `:validate` – when set to `true`, validates the associated object upon save of this object. By default this is false.

Nested Routes and Singular Resources

One problem with our current Purchase-Invoice scheme is that you could assign one Purchase to two Invoices.



Calling `@purchase.invoice` on this will return only the first related Invoice.



The root cause of this problem is that we're creating Invoices outside the context of the Purchase. Wouldn't it make more sense to be able to go to http://localhost:3000/purchases/1/invoice/new to create the invoice for the first Purchase that what we're currently doing right now?

We can do this with the help of 2 new routing concepts: the singular resource and nested resources.

The singular resource just like our typical `resources` call, but instead of dealing with multiple records, it deals with only one object. For example, adding the singular resource route:

```
resource :platform
```

assumes that we only have one Platform entity for the entire system (note we used `resource` instead of `resource`**s**). The generated routes in this instance is:

helper	HTTP verb	URL	controller	action	use
platform_url platform_path	GET	/platform	Platforms	show	Display the platform
new_platform_url new_platform_path	GET	/platform/new	Platforms	new	Return a form for creating the new platform
	POST	/platform	Platforms	create	Create the new platform
edit_platform_url edit_platform_path	GET	/platform/edit	Platforms	edit	Return a form for editing the platform
	PUT	/platform	Platforms	update	Update the platform
	DELETE	/platform	Platforms	destroy	Delete the platform

To use this in the Purchase-Invoice relationship, we'll have to set the Invoice as a singleton child of each Purchase entity by nesting a `resource` call under the `resources` call:

```
AlingnenaApp::Application.routes.draw do
  resources :purchases do
    resource :invoice
  end

  resources :debts
  resources :products do
```

```
    collection do
      get 'search'
    end
  end
end
```

This will create the following additional routes:

helper	HTTP verb	URL	controller	action
purchase_invoice_url purchase_invoice_path	GET	/purchases/:purchase_id/invoice	Invoices	show
new_purchase_invoice_url new_purchase_invoice_path	GET	/purchases/:purchase_id/invoice/new	Invoices	new
	POST	/purchases/:purchase_id/invoice	Invoices	create
edit_purchase_invoice_url edit_purchase_invoice_path	GET	/purchases/:purchase_id/invoice/edit	Invoices	edit
	PUT	/purchases/:purchase_id/invoice	Invoices	update
	DELETE	/purchases/:purchase_id/invoice	Invoices	destroy

We have to change the logic of our Invoice program to handle these changes. First, the controller:

```
class InvoicesController < ApplicationController
  def show
    # we'll integrate the Invoice details in the Show Purchase screen
    redirect_to purchase_path(params[:purchase_id])
  end

  def new
    @purchase = Purchase.find(params[:purchase_id])
    @invoice = @purchase.build_invoice
  end

  def edit
    @purchase = Purchase.find(params[:purchase_id])
    @invoice = @purchase.invoice
  end

  def create
    @purchase = Purchase.find(params[:purchase_id])
    @invoice = @purchase.build_invoice(params[:invoice])

    if @invoice.save
      redirect_to @purchase, :notice => 'Invoice was successfully created.'
    else
      render :action => "new"
    end
  end

  def update
    @purchase = Purchase.find(params[:purchase_id])
    @invoice = @purchase.invoice

    if @invoice.update_attributes(params[:invoice])
      redirect_to @purchase, :notice => 'Invoice was successfully updated.'
    else
      render :action => "edit"
    end
  end
```

```
  def destroy
    @purchase = Purchase.find(params[:purchase_id])
    @invoice = @purchase.invoice
    @invoice.destroy

    redirect_to @purchase
  end
end
```

In case you've forgotten how routes work, the :purchase_id parameter is derived from the URL.

Here's the changes we'll need for the views. Let's modify Invoice's views:

```
# app/views/invoices/new.html.erb

<h1>New Invoice</h1>

<%= render 'form', :purchase => @purchase, :invoice => @invoice  %>

<%= link_to 'Back', invoices_path %>

# app/views/invoices/show.html.erb

<h1>Editing Invoice</h1>

<%= render 'form', :purchase => @purchase, :invoice => @invoice %>

<%= link_to 'Show', @invoice %> |
<%= link_to 'Back', invoices_path %>

# app/views/invoices/_form.html.erb

<%= form_for(invoice, :url => purchase_invoice_path(purchase)) do |f| %>
  <% if invoice.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(invoice.errors.count, "error") %> prohibited this invoice from being
saved:</h2>

      <ul>
      <% invoice.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  <p>
    <label>Purchase</label><br />
    <%= purchase.description %>
  </p>

  <p>
    <%= f.label :reference_number %><br />
    <%= f.text_field :reference_number %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>

<%= link_to 'Back', purchase %>
```

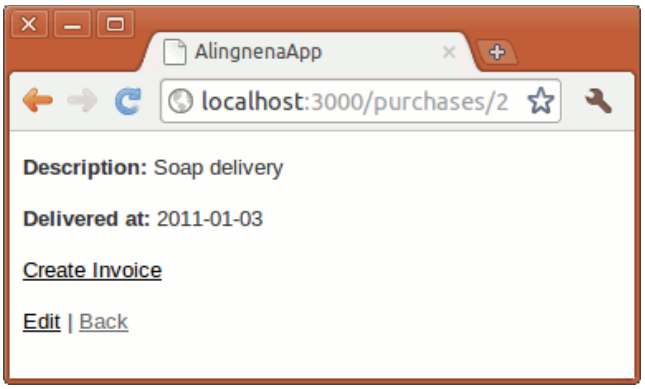
Here's Purchase's app/views/purchases/show.html.erb:

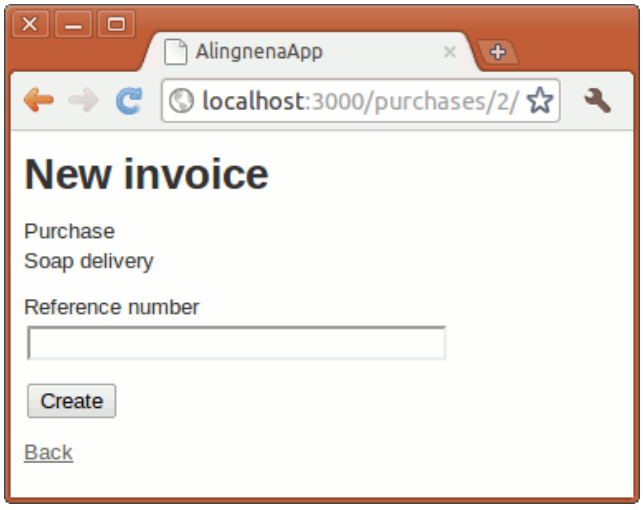
```
<h1>Purchase Details</h1>
<p>
  <b>Description:</b>
  <%= @purchase.description %>
</p>
<p>
  <b>Delivered at:</b>
  <%= @purchase.delivered_at %>
</p>

<% display_invoice @purchase %>

<%= link_to 'Edit', edit_purchase_path(@purchase) %> |
<%= link_to 'Back', purchases_path %>
```

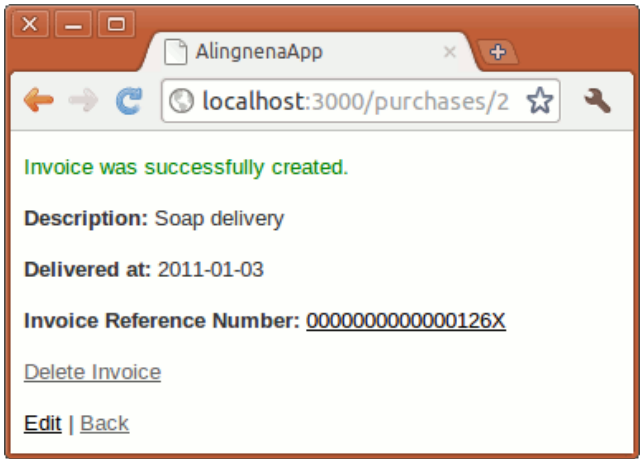
We used a `scriptlet` instead of an expression so that we could use the `concat` method (similar to PHP's `print` or Servlets' `out.println`) in our helper to add multiple elements in the view. Here's the helper method:

```
module PurchasesHelper
  def display_invoice(purchase)
    unless purchase.invoice.nil?
      concat(raw "<p><b>Invoice Reference Number:</b>\n")
      concat(link_to @purchase.invoice.reference_number,
               edit_purchase_invoice_path(@purchase))
      concat(raw "</p>")
      concat(raw "<p>")
      concat(link_to "Delete Invoice", purchase_invoice_path(@purchase),
              :confirm => "Are you sure?", :method => :delete)
      concat(raw "</p>")
    else
      concat(raw "<p>")
      concat(link_to "Create Invoice", new_purchase_invoice_path(@purchase))
      concat(raw "</p>")
    end
  end
end
```

Our new Purchase-Invoice program should work properly now:




AlingnenaApp
localhost:3000/purchases/2/
New invoice
Purchase
Soap delivery
Reference number
Create
Back


AlingnenaApp
localhost:3000/purchases/2
Invoice was successfully created.
Description: Soap delivery
Delivered at: 2011-01-03
Invoice Reference Number: 0000000000000126X
Delete Invoice
Edit | Back

On second thought, using `concat` was an ugly approach. Change the `display_invoice` call to an expression (i.e. change `<% display_invoice...` at `show.html.erb` to `<%= display_invoice...`) and change the helper to the following:

```
module PurchasesHelper
  def display_invoice(purchase)
    unless purchase.invoice.nil?
      render 'invoice', :purchase => @purchase
    else
      render 'no_invoice', :purchase => @purchase
    end
  end
end
```

Then create the following partials:

```
# app/views/purchases/_no_invoice.html.erb

<p>
  <%= link_to "Create Invoice", new_purchase_invoice_path(@purchase) %>
</p>

# app/views/purchases/_invoice.html.erb

<p>
  <b>Invoice Reference Number:</b>
  <%= link_to @purchase.invoice.reference_number,
        edit_purchase_invoice_path(@purchase) %>
</p>
<p>
  <%= link_to "Delete Invoice", purchase_invoice_path(@purchase),
        :confirm => "Are you sure?", :method => :delete %>
</p>
```

One-to-Many Relationships

Once you know how to create a one-to-one relationship, one-to-many associations should be simple. Let's create a Supplier-Purchase one-to-many association, that is, Aling Nena could make multiple purchases from a single supplier.

Here's the script for the basic setup:

```
$ rails generate scaffold supplier name:string contact_number:string
$ rails generate migration AddSupplierToPurchase supplier_id:integer
$ rake db:migrate
```

Note that we didn't use `:references` in the migration because it doesn't work with `add_column`.

Modify the models like so:

```
# app/models/purchase.rb

class Purchase < ActiveRecord::Base
  has_one :invoice
  belongs_to :supplier

  def display_supplier
    return supplier.name unless supplier.nil?
  end

end

# app/models/supplier.rb

class Supplier < ActiveRecord::Base
  has_many :purchases
end
```

The argument `:purchases` should be plural according to convention.

As for the purchase screens, `collection_select` is appropriate for the Supplier-Purchase association:

```
...
    <%= f.date_select :delivered_at %>
  </div>
  <div class="field">
    <%= f.label :supplier_id, "Supplier" %><br />
    <%= f.collection_select :supplier_id, Supplier.all, :id, :name, :prompt => true %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
...
```

If you want to modify the Show Purchase screen to show the Supplier, just use the `display_supplier` method from the model.

Before we can proceed with displaying the list of Purchases under the Supplier, we must first understand what the `has_many` declaration does to a model.

has_many

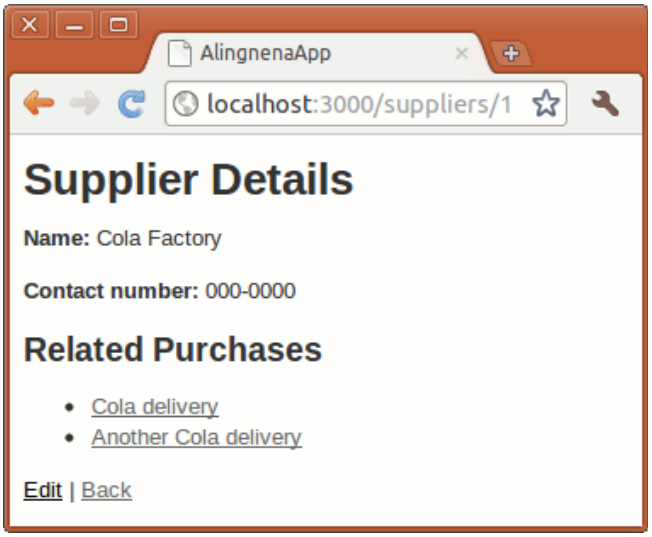
The following instance methods are added to Supplier after we added the `has_many :purchases` in the model class:

- `supplier.purchases` – returns a list of Purchase objects associated with the Supplier. This is also cached so if you want to reload the list, just pass `true` to the method.
- `supplier.purchases <<` *`new_purchase`* – adds a Purchase to the list of Purchases associated with the Supplier.
- `supplier.delete(*`*`purchase`*`)` – removes the association to the object or objects specified. When an association is removed, the action done to the object depends on the `:dependent` option i.e. it will either nullify the field or delete the record.
- `supplier.purchases =` *`list_of_purchases`* – assigns a list of Purchase objects to the Supplier, removing associations as appropriate.
- `supplier.purchase_ids, supplier.purchase_ids =` *`list_of_purchase_ids`* – just like `.purchases`, but instead of dealing with a list of Purchase objects, this just uses a list of `ids` of those objects.
- `supplier.purchases.clear` – removes all associations to Purchase objects.
- `supplier.purchases.empty?` – returns true if there are no associated Purchases
- `supplier.purchases.size` – returns the number of associated Purchases
- `supplier.purchases.find()`, `supplier.purchases.exist?()` – basically just `ActiveRecord.find` and `ActiveRecord.exist?` but only uses the Purchase records associated with the Supplier
- `supplier.purchases.build()`, `supplier.purchases.create()` - just like the `build_xxxx` and `create_xxxx` from `has_one`

With this knowledge, we can now add the list of Purchases in the Show Supplier screen:

```
<h1>Supplier Details</h1>

<p>
  <b>Name:</b>
  <%= @supplier.name %>
</p>
<p>
  <b>Contact number:</b>
  <%= @supplier.contact_number %>
</p>

<h2>Related Purchases</h2>
<ul>
<% @supplier.purchases.each do |purchase| %>
  <li><%= link_to purchase.description, purchase %></li>
<% end %>
</ul>
...
```

The resulting page is:


AlingnenaApp
localhost:3000/suppliers/1
Supplier Details
Name: Cola Factory
Contact number: 000-0000
Related Purchases
Cola delivery
Another Cola delivery
Edit | Back

Many-to-many Relationships

We can use Purchase and Product to demonstrate how many-to-many associations are done in Rails. A Purchase can consist of many Products, while a Product can be part of many Purchases. There are two ways declaring this many-to-many relationship in Rails, and both require a join table.

has_and_belongs_to_many

We can declare `has_and_belongs_to_many` in our models to specify that the models have a many-to-many relationship with each other. For this to work, there must first be an existing join table that satisfies the following conditions:

- It must be named according to the tables involved in the relationship, separated by underscore and arranged alphabetically. In the Product-Purchase case, the table should be `products_purchases`.
- It must only contain references to each table. It should not have any other fields e.g. primary key.

Let's setup the product_purchase table with a migration:

```
$ rails generate migration CreateProductsPurchases
```

Edit the generated migration:

```
class CreateProductsPurchases < ActiveRecord::Migration
  def self.up
    create_table :products_purchases, :id => false do |t|
      t.integer :product_id
      t.integer :purchase_id
    end
  end

  def self.down
    drop_table :products_purchases
  end
end
```

The `:id => false` option removes the default primary key field from the table.

Then add the `has_and_belongs_to_many` declaration in the models:

```
# app/models/product.rb

class Product < ActiveRecord::Base
  validates_presence_of :name, :description
  has_and_belongs_to_many :purchases
...

# app/models/purchase.rb

class Purchase < ActiveRecord::Base
  has_one :invoice
  belongs_to :supplier
  has_and_belongs_to_many :products
...
```

At this point, you can already perform `has_many` methods on both sides. For example:

```
some_product.purchases << new_purchase

some_product.purchases   # this will return a list containing new_purchase
new_purchase.products    # this will return a list containing some_product
```

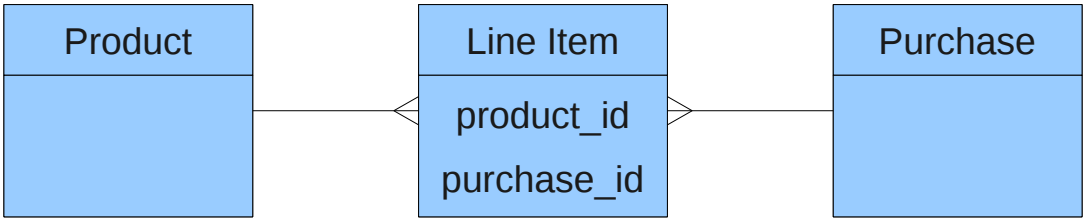
The `has_and_belongs_to_many` declaration is suited for simple many-to-many relationships. One possible scenario is a Book-liking module where Users can "like" Books. In turn, a Book can have many Users that "like" it.

For more complex relationships that would have other details, we would need another approach. For example, the relationship between Purchase and Product may also require us to specify the quantity of the Product in that Purchase. We can't store that in our join table because of the limitations of that table.

The `has_many :through` declaration provides the other approach to many-to-many relationships.

More on One-to-Many

The has_many :through declaration works because it uses two one-to-many associations linked together by one join table:



Once declared, this association works just like has_and_belongs_to_many wherein you could call methods like `some_product.purchases`. It's not that complicated, so let's take this time to explore one-to-many again instead.

Our join table for Product-Purchase would be Line Items, child records of Purchase that would represent lines in the Invoice. This is where Aling Nena would input the quantity, cost, and (most importantly) the Product involved.

Setting up the Line Items program should be easy with `rails generate scaffold`:

```
$ rails generate scaffold line_item purchase:references product:references quantity:integer
cost:decimal
```

Taking a cue from Invoice, we'll make Line Item a child resource for Purchase to make maintenance easier.

Nested Resources

We've previously discussed what routes are generated when a singular resource is assigned as a child to a resource route. Now we'll assign a normal resource under a resource route:

```
AlingnenaApp::Application.routes.draw do
  resources :suppliers

  resources :purchases do
    resource :invoice
    resources :line_items
```

```
  end

  resources :debts
  resources :products do
    collection do
      get 'search'
    end
  end
end
```

This generates the following routes:

helper	HTTP verb	URL	controller	action
purchase_line_item_url purchase_line_item_path	GET	/purchases/:purchase_id/line_items	LineItems	index
new_purchase_line_item_url new_purchase_line_item_path	GET	/purchases/:purchase_id/line_items/new	LineItems	new
	POST	/purchases/:purchase_id/line_items	LineItems	create
	GET	/purchases/:purchase_id/line_items/:id	LineItems	show
edit_purchase_line_item_url edit_purchase_line_item_path	GET	/purchases/:purchase_id/line_items/:id/edit	LineItems	edit
	PUT	/purchases/:purchase_id/line_items/:id	LineItems	update
	DELETE	/purchases/:purchase_id/line_items/:id	LineItems	destroy

With routes set, we proceed with some model changes:

```
# app/models/purchase.rb

class Purchase < ActiveRecord::Base
  has_one :invoice
  belongs_to :supplier
  has_many :line_items
...

# app/models/product.rb

class Product < ActiveRecord::Base
  validates_presence_of :name, :description
  has_many :line_items
...
```

After the model comes the controller rewrite of `app/controller/line_items_controller.rb`:

```
class LineItemsController < ApplicationController
  def index
    # integrate with show purchase
    redirect_to Purchase.find(params[:purchase_id])
  end

  def show
    # integrate with show purchase
    redirect_to Purchase.find(params[:purchase_id])
  end

  def new
    @purchase = Purchase.find(params[:purchase_id])
    @line_item = @purchase.line_items.build
  end
```

```
  def edit
    @purchase = Purchase.find(params[:purchase_id])
    @line_item = @purchase.line_items.find(params[:id])
  end

  def create
    @purchase = Purchase.find(params[:purchase_id])
    @line_item = @purchase.line_items.build(params[:line_item])

    if @line_item.save
      redirect_to @purchase, :notice => 'Line Item was successfully created.'
    else
      render :action => "new"
    end
  end

  def update
    @purchase = Purchase.find(params[:purchase_id])
    @line_item = @purchase.line_items.find(params[:id])

    if @line_item.update_attributes(params[:line_item])
      redirect_to @purchase, :notice => 'Line Item was successfully updated.'
    else
      render :action => "edit"
    end
  end

  def destroy
    @purchase = Purchase.find(params[:purchase_id])
    @line_item = @purchase.line_items.find(params[:id])
    @line_item.destroy

    redirect_to @purchase, :notice => 'Line Item was successfully deleted.'
  end
end
```

And the views (null handling not implemented):

```
# app/views/purchases/show.html.erb

...
<%= display_invoice @purchase %>

<h2>Line Items</h2>

<table>
  <tr>
    <th>Product</th>
    <th>Quantity</th>
    <th>Cost</th>
  </tr>
<% @purchase.line_items.each do |item| %>
  <tr>
    <td><%= item.product.name %></td>
    <td><%= item.quantity %></td>
    <td><%= number_to_currency item.cost, :unit => "PhP" %></td>
    <td><%= link_to "Edit", edit_purchase_line_item_path(@purchase, item) %></td>
    <td>
      <%= link_to "Destroy", purchase_line_item_path(@purchase, item),
            :confirm => "Are you sure?", :method => :delete %>
    </td>
  </tr>
<% end %>
</table>
```

```
<p><%= link_to "New Line Item", new_purchase_line_item_path(@purchase) %></p>

<%= link_to 'Edit', edit_purchase_path(@purchase) %> |
...
```

 # app/views/line_items/new.html.erb

```
<h1>New Line Item</h1>

<%= render 'form', :purchase => @purchase, :line_item => @line_item %>
```

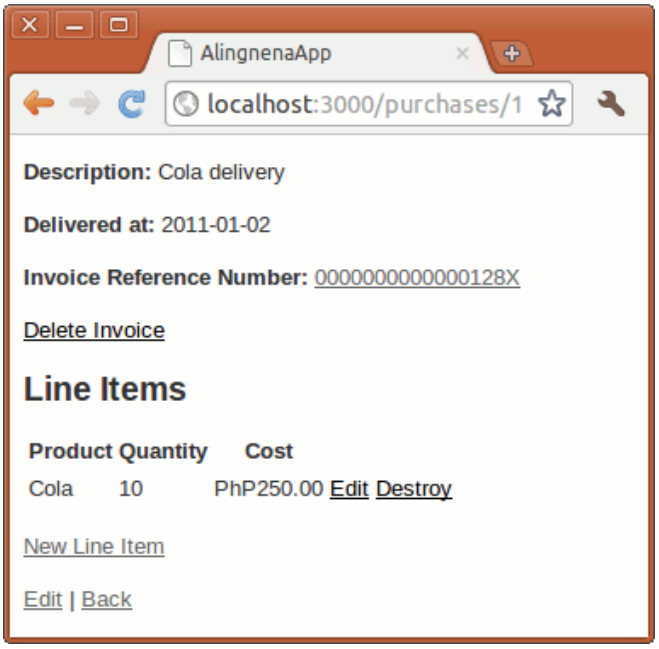
 # app/views/line_items/edit.html.erb

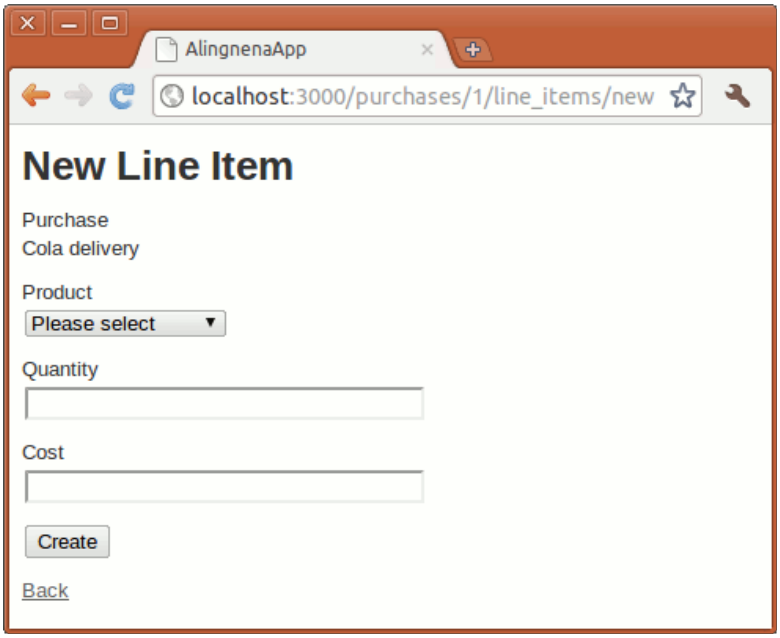
```
<h1>Editing Line Item</h1>

<%= render 'form', :purchase => @purchase, :line_item => @line_item %>
```

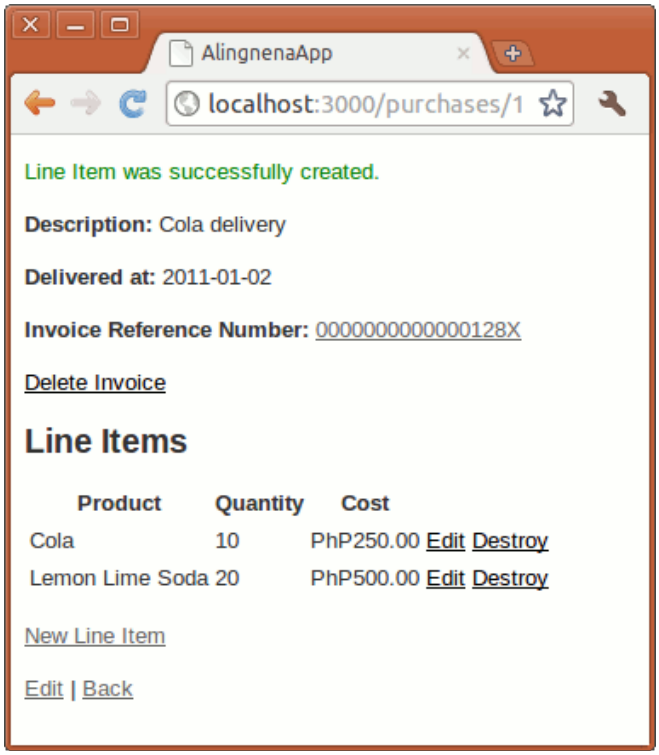
 # app/views/line_items/_form.html.erb

```
<%= form_for([purchase, line_item]) do |f| %>
  <% if line_item.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(line_item.errors.count, "error") %> prohibited this line_item from being
saved:</h2>

      <ul>
      <% line_item.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <label>Purchase</label><br />
    <%= purchase.description %>
  </div>
  <div class="field">
    <%= f.label :product_id, "Product" %><br />
    <%= f.collection_select :product_id, Product.all, :id, :name, :prompt => true %>
  </div>
  <div class="field">
    <%= f.label :quantity %><br />
    <%= f.text_field :quantity %>
  </div>
  <div class="field">
    <%= f.label :cost %><br />
    <%= f.text_field :cost %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>

<%= link_to 'Back', purchase %>
```

Here's the program in action:


AlingnenaApp
localhost:3000/purchases/1
Description: Cola delivery
Delivered at: 2011-01-02
Invoice Reference Number: 0000000000000128X
Delete Invoice
Line Items
Product Quantity Cost
Cola 10 PhP250.00 Edit Destroy
New Line Item
Edit | Back


AlingnenaApp
localhost:3000/purchases/1/line_items/new
New Line Item
Purchase
Cola delivery
Product
Please select
Quantity
Cost
Create
Back



Nothing really new here. Sharp eyed readers might notice the strange usage of `form_for` in Invoice and Line Items. In Invoice, the `form_for` is written as:

```
<% form_for(invoice, :url => purchase_invoice_path(purchase)) do |f| %>
```

The `:url` option overrides the expected URL helper, `invoice_path(purchase)`, with `purchase_invoice_path(purchase)`. Since this is a singleton resource, both create and update URLs are the same so the partial works for both cases.

In Line Items, the form_for is written as:

```
<% form_for([purchase, line_item]) do |f| %>
```

Instead of a single object, we pass an array of objects. Based on this array of objects, Rails knows what URL to use according to the convention. If the `line_item` object was new, the path it will use will be `purchase_line_items_path(purchase)`. If it's already existing, the path will be `purchase_line_item_path(purchase, line_item)`.

has_many :through

Now that we could create Line Items, it's time to show how we can use the many-to-many relationship in the Products side. For this, we need to add the `has_many :through` declaration:

```
class Product < ActiveRecord::Base
  validates_presence_of :name, :description
  has_many :line_items
  has_many :purchases, :through => :line_items

  def before_validation
    if description.blank?
```

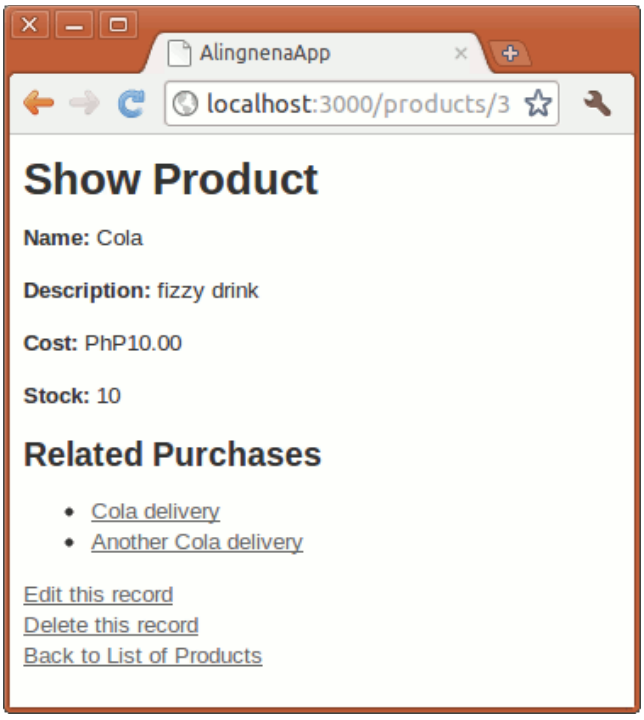
```
      self.description = name
    end
  end
end
```

The `:through` specifies the join table used for the relationship. This join table must also be declared as the target table in a `has_many` declaration.

We can now access the list of Purchases involving the Product through `product.purchases`. Here's one way of doing it at `app/views/products/show.html.erb`:

```
...
<p>
  <b>Stock:</b>
  <%= @product.stock %>
</p>

<h2>Related Purchases</h2>
<ul>
<% @product.purchases.each do |purchase| %>
  <li><%= link_to purchase.description, purchase %></li>
<% end %>
</ul>

<%= link_to 'Edit this record', edit_product_path(@product) %> <br />
...
```

And the result:

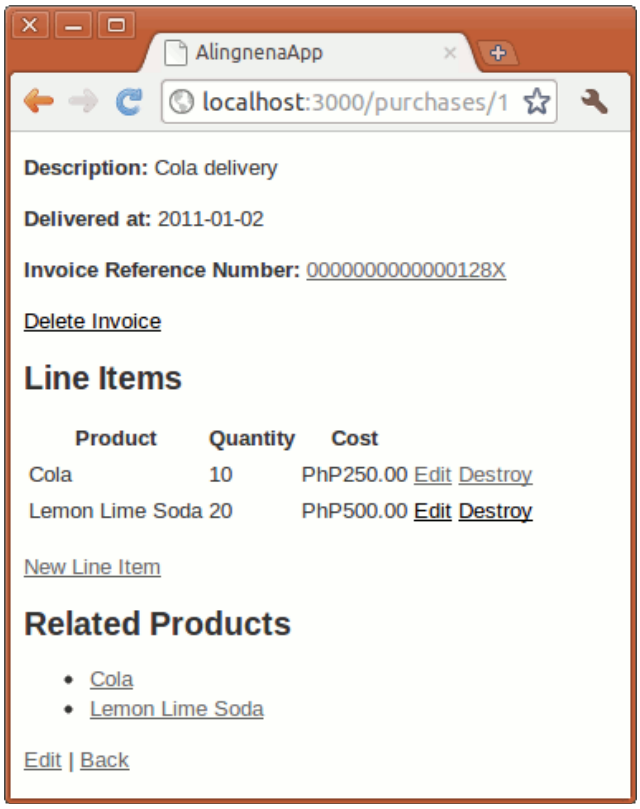


Just to show that the many-to-many goes both ways, let's do the same for Purchase (even though it will look redundant):

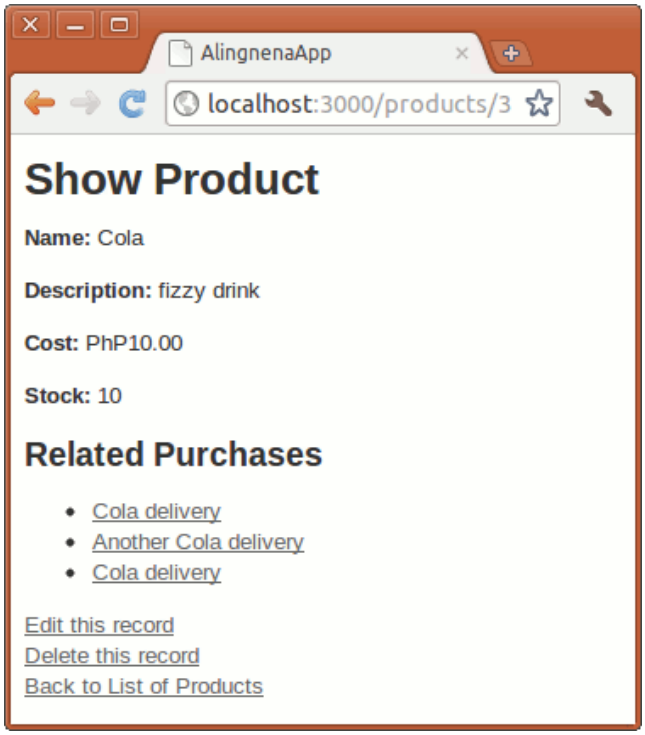
```
# app/models/purchase.rb
```

```
class Purchase < ActiveRecord::Base
  has_one :invoice
  belongs_to :supplier
  has_many :line_items
  has_many :products, :through => :line_items

  def display_supplier
    return supplier.name unless supplier.nil?
  end
end

# app/views/purchases/show.html.erb

...
<p><%= link_to "New Line Item", new_purchase_line_item_path(@purchase) %></p>

<h2>Related Products</h2>
<ul>
  <% @purchase.products.each do |product| %>
    <li><%= link_to product.name, product %></li>
  <% end %>
</ul>

<%= link_to 'Edit', edit_purchase_path(@purchase) %> |
...
```

Here's the updated page:



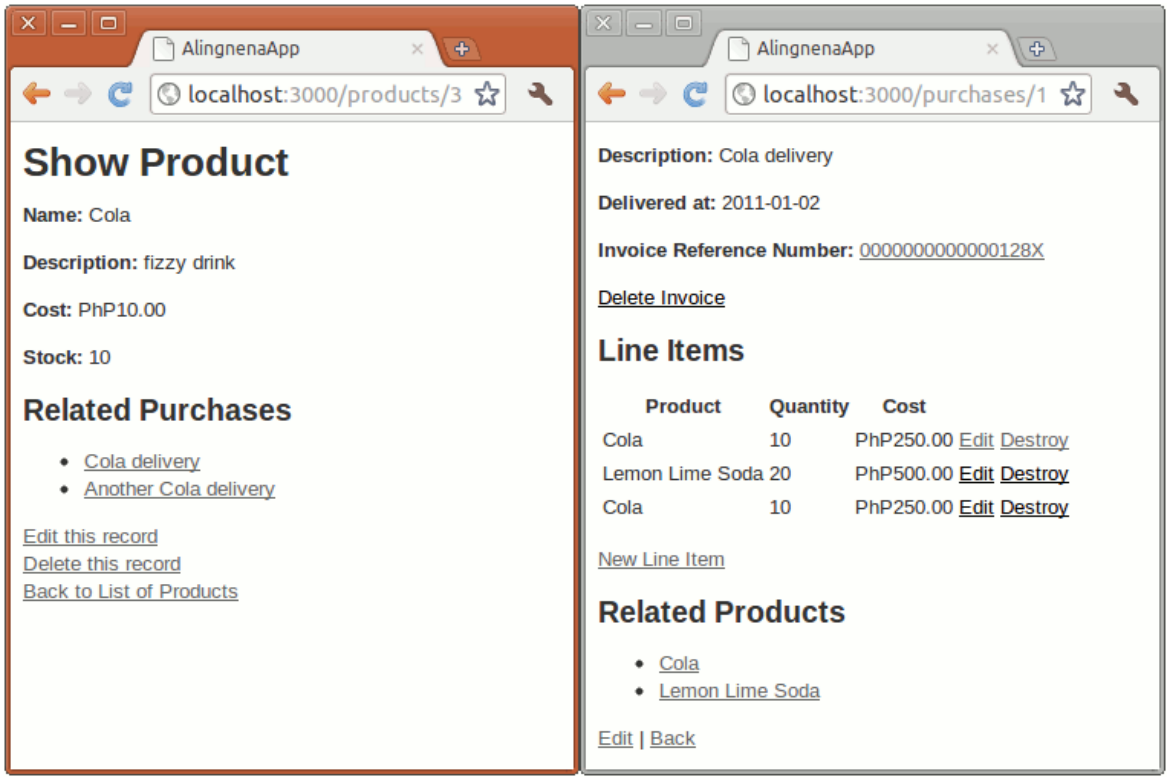
The whole thing looks ok, but what if you add another “Cola” Line Item to “Cola delivery”?



Cola delivery is listed twice since there are two references to Cola in Line Items. To eliminate the redundant results, we can add the `:uniq` option:

```
# app/models/purchase.rb

class Purchase < ActiveRecord::Base
  has_one :invoice
  belongs_to :supplier
  has_many :line_items
  has_many :products, :through => :line_items, :uniq => true

  def display_supplier
    return supplier.name unless supplier.nil?
  end
end

# app/models/product.rb

class Product < ActiveRecord::Base
  validates_presence_of :name, :description
  has_many :line_items
  has_many :purchases, :through => :line_items, :uniq => true

  def before_validation
    if description.blank?
      self.description = name
    end
  end
end
```

Trying the two pages produces:



Another problem is the amount of queries are performed in our Purchase page:

```
  Processing by PurchasesController#show as HTML
  Parameters: {"id"=>"1"}
  Purchase Load (0.2ms)  SELECT "purchases".* FROM "purchases" WHERE "purchases"."id" = 1 LIMIT 1
  Invoice Load (0.3ms)  SELECT "invoices".* FROM "invoices" WHERE ("invoices".purchase_id = 1) LIMIT 1
  LineItem Load (0.3ms)  SELECT "line_items".* FROM "line_items" WHERE ("line_items".purchase_id = 1)
  Product Load (0.2ms)  SELECT "products".* FROM "products" WHERE "products"."id" = 3 LIMIT 1
  Product Load (0.8ms)  SELECT "products".* FROM "products" WHERE "products"."id" = 4 LIMIT 1
  CACHE (0.0ms)  SELECT "products".* FROM "products" WHERE "products"."id" = 3 LIMIT 1
  Product Load (1.3ms)  SELECT DISTINCT "products".* FROM "products" INNER JOIN "line_items" ON "produc
ts".id = "line_items".product_id WHERE (("line_items".purchase_id = 1))
Rendered purchases/show.html.erb within layouts/application (147.3ms)
```

A query is made for every Product record that needs to be displayed in the page. To reduce this to fewer queries, we can chain the `include()` method after the find method for `show`:

```
        def show
           @purchase = Purchase.includes(:line_items => :product).find(params[:id])
        ...
```

This will tell Rails to "eager-load" the line items association (and its child association, product) as opposed to the default "lazy-load" approach where the records are retrieved only when needed (e.g. a `product.name` call).

```
  Processing by PurchasesController#show as HTML
  Parameters: {"id"=>"1"}
  Purchase Load (0.3ms)  SELECT "purchases".* FROM "purchases" WHERE "purchases"."id" = 1 LIMIT 1
  LineItem Load (0.3ms)  SELECT "line_items".* FROM "line_items" WHERE ("line_items".purchase_id = 1)
  Product Load (0.6ms)  SELECT "products".* FROM "products" WHERE ("products"."id" IN (3,4))
  Invoice Load (0.2ms)  SELECT "invoices".* FROM "invoices" WHERE ("invoices".purchase_id = 1) LIMIT 1
  Product Load (0.9ms)  SELECT DISTINCT "products".* FROM "products" INNER JOIN "line_items" ON "produc
ts".id = "line_items".product_id WHERE (("line_items".purchase_id = 1))
```

The action is now reduced to 5 queries from its original 7.

Skimming through Rails

Rails is a simple but fairly large framework. We can't cover everything in this course, so in this chapter, we will go through the features that you might have to use on a regular basis.

Rails Migrations

We've discussed the basics of migrations, namely, how to create or drop a table and how to add or remove columns from them. We also used generator scripts for the migrations instead of coding them manually.

In this part, we'll discuss the various methods available to us in the migrations. We'll also go deeper in the mechanisms behind migrations.

Creating Tables

The `create_table` method accepts a table name and a block where we define the fields in the table. Going back to our first migration:

```
def self.up
  create_table :debts do |t|
    t.string :name
    t.text :item
    t.decimal :amount

    t.timestamps
  end
end
```

This `create_table` call creates the table Debts in the database with an integer primary key "`id`",a string field "`name`", a text field "`item`", a decimal field "`amount`", and two timestamp fields "`created_at`" and "`updated_at`".

We've discussed before the possible data types for use in defining columns. Here's the fields that would be created per database for those data types:

	db2	mysql	openbase	oracle	postgresql	sqlite	sqlserver	sybase
`:binary`	blob(32768)	blob	object	blob	bytea	blob	image	image
`:boolean`	decimal(1)	tinyint(1)	boolean	number(1)	boolean	boolean	bit	bit
`:date`	date	date	date	date	date	date	datetime	datetime
`:datetime`	timestamp	datetime	datetime	date	timestamp	datetime	datetime	datetime
`:decimal`	decimal	decimal	decimal	decimal	decimal	decimal	decimal	decimal
`:float`	float	float	float		float	float	float(8)	float(8)
`:integer`	int	int(11)	integer	number(38)	integer	integer	int	int
`:string`	varchar(255)	varchar(255)	varchar(4096)	varchar2(255)	character varying(256)	varchar(255)	varchar(255)	varchar(255)

:text	clob(32768)	text	text	clob	text	text	text	text
:time	time	time	time	date	time	datetime	datetime	time
:timestamp	timestamp	datetime	timestamp	date	timestamp	datetime	datetime	timestamp

There are also shorthand data types e.g. the `t.references` from the previous lesson.

There are options you could specify to each column declaration to apply database-level constraints:

- `:null` – set this option to true if nulls are allowed for this field, false otherwise

```
t.string :name, :null => false  # sets name field to mandatory
```

- `:default` – set this option to make the database set a default value for the field if the field is left empty. If you want the field default to be NULL, use `nil`.

- `:limit` – set this option to the maximum size of the field This refers to the number of characters for `:string` and `:text`, number of bytes for the `:binary` and `:integer`.

The following options are for `:decimal`:

- `:precision` – the precision of the decimal column, the number of significant digits in the number

- `:scale` – the scale of the decimal column, the number of digits after the decimal point. The number 123.45 has a precision of 5 and a scale of 2.

Note that these column options can also be used in the add_column method. For example:

```
add_column :debts, :remarks, :text, :limit => 200
```

You can also set options for the table by passing a hash of options after the table name:

- `:id => false` – this will prevent the automatic creation of the primary key

- `:primary_key` – if :id is is not set to false, you can set this option to change the name of the primary key from “id” e.g.

```
create_table :debts, :primary_key => "debt_no" do |t|
```

- `:force => true` – this will make the migration drop the table of the same name (if it exists) before creating the table

- `:temporary => true` – the table created is only temporary and will be dropped when the connection is closed.

- `:options` – the value specified in this option will be appended to the table definition SQL. For example:

```
create_table :debts, :options => "ENGINE=InnoDB DEFAULT CHARSET=utf8" do |t|
```

will create:

```
CREATE TABLE debts (
...
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

Other Migration Methods

Here are some other methods you can use in the migrations:

- `change_column(table_name, column_name, type, options = {})` – changes a column to the specified type and options. The column options are the same from `create_table` and `add_column`. For example, you want to change the maximum length of remarks, you could use:

  ```
  change_column :debts, :remarks, :text, :limit => 300
  ```

- `rename_column(table_name, column_name, new_column_name)` – renames a column to `new_column_name`

- `add_index(table_name, column_name, options = {})` – adds an index to the table. You can set the name of the index as well as the unique setting in the options.

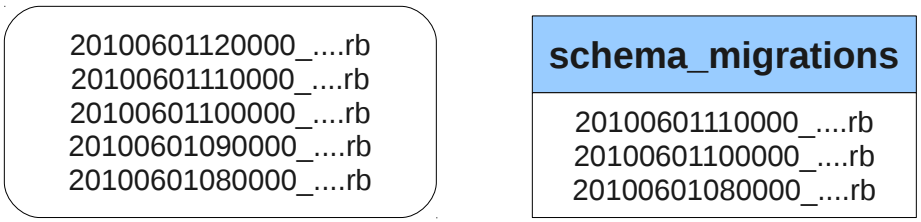
- `remove_index(table_name, options = {})` – removes an index from the table

- `rename_table(table_name, new_name)` – renames table `table_name` to `new_name`

- `change_table(table_name)` – changes columns in the `create_table` syntax.

Migration Mechanism

When you run `rake db:migrate`, Rails first checks the `schema_migrations` table for the list of migrations already applied to the database then it applies all migrations not already inside the table. For example, we have the following situation:



In this case, our `schema_migrations` table is missing the 12 noon and 9AM migrations. The missing 9AM migration is possible, you might have just updated your local version of the application and got that migration from another developer. Running `rake db:migrate` will apply the two migrations into your database and insert two new records to the `schema_migrations` table.

To rollback your database to an older version, you can use the VERSION environment variable to pass a timestamp:

```
> rake db:migrate VERSION=2010060113000
```

This will rollback all migrations after the specified timestamp (if any; it's possible to use this call for a "roll-forward") and apply all migrations on or before the said timestamp.

Active Record

There are still a lot of important concepts in Active Record that we haven't discussed in detail yet.

Querying

Aside from `find()` and `all()`, there are two other general methods for retrieving records:

- `first()` – returns the first record retrieved from the table.
- `last()` – returns the last record retrieved from the table..

As we've seen in the `includes()` method in the previous lesson, these methods can be chained with other methods to modify their behavior. Note that `all()` is used by default so you don't need to specify it in a chain with other methods.

Below are the methods that you can use for querying.

where

You can define the WHERE part of the SQL statement with the `where` method. It accepts either a string, an array that starts with a string, or a hash. Passing a string would simply make it use the string as the SQL `WHERE` clause of the query. For example:

```
User.where("user_name = '#{user_name}' AND password = '#{password}'").first
```

would find the record with a user name and password that matches the supplied credentials. Obviously this approach is prone to SQL injection attacks, so a better approach would be to use an array argument which creates a parameterized `WHERE` clause.

```
User.where("user_name = ? AND password = ?", user_name, password).first
```

The 2nd element onwards are assigned to the question marks after sanitizing the elements. The problem with this form of parameterized queries is that when the query becomes larger, it becomes difficult to determine which parameter is assigned to which question mark. Here we can use named parameters (symbols inside the string) and a hash for the second element:

```
User.where("user_name = :user_name AND password = :password", { :user_name => user_name, :password =>
password }).first
```

For simple queries like the one above, you can use a hash of values. Rails would automatically convert the key-value pairs into a series of 'key = value' entries in the `WHERE` clause, also sanitizing the input in the process.

```
User.where( :user_name => user_name, :password => password ).first
```

order, select, from

The `order()` method allows you to provide the contents of the `ORDER BY` clause in the query. For example:

```
Products.order("name")
LineItem.order("purchase_id DESC, id")
```

The select() method allows you to define the fields for SELECT clause. By default, ActiveRecord uses "*" to select all fields in the table but if your database incurs a performance hit when you query certain fields (e.g. BLOB fields), you might want to remove those fields from the SELECT clause. As the object's attributes are dynamically assigned, the removed fields will not be available in the return objects.

```
>> d = Debt.select("id").first
=> #<Debt id: 1>
>> d.name
ActiveRecord::MissingAttributeError: missing attribute: name
```

Similar to select() and order(), from() allows you to specify the fragment for the FROM clause. You might want to do this if you're using a different table or a view for the query.

:group, :having, Calculations

The group() and having() methods are also like order(), they allow you to provide the SQL fragments for GROUP BY and HAVING clauses in the query, respectively.

```
LineItem.group("product_id")
LineItem.group("purchase_id").having("purchase_id = 1")
```

Since HAVING only applies to GROUP BY, :having will only be applied to the query if there is a :group option specified.

Queries involving group() will only return one record per "group". Because of this, group() and having() are better used with the aggregate functions under ActiveRecord::Calculations. The available functions are:

- count – returns the number of records per group

  ```
  LineItem.group("purchase_id").count
  ```

- average – returns the average value of a specified field per group

  ```
  LineItem.group("purchase_id").average("cost")
  ```

- sum – returns the sum of the values of a specified field per group

  ```
  LineItem.group("purchase_id").sum("cost")
  ```

- maximum, minimum – returns the maximum and minimum value of a specified field per group

  ```
  LineItem.group("purchase_id").maximum("cost")
  LineItem.group("purchase_id").minimum("cost")
  ```

The return value for these methods will be a hash of the "field name used in group" => "calculated value". Note that these aggregate functions only work for one grouping field name, using more than one (e.g. group("purchase_id, product_id")) would return unpredictable results.

Also note that the aggregate functions can be used without the group() method to calculate the value for the entire table (the result will be a single value, not a hash).

:limit, :offset

The `:limit` option allow you to specify how many records are retrieved by the query.

```
Debt.limit(10)
```

When the `:limit` option is used for paginating records, we often employ another option, `:offset`, to determine the "starting point" for the query. The number passed to `:offset` determines the number of records to skip in the query. For example:

```
Debt.offset(5)
```

This query skips the first 5 records retrieved and starts with the 6th record.

:readonly, :lock

Set `readonly` to `true` if you want to prevent updates to the retrieved records.

```
>> d = Debt.readonly(true).first
=> #<Debt id: 1, name: ...>
>> d.save
ActiveRecord::ReadOnlyRecord: ActiveRecord::ReadOnlyRecord
```

You can use `lock()` to add a fragment for locking in the query e.g. "`FOR UPDATE`" or "`LOCK IN SHARE MODE`". Using `lock(true)` uses the default exclusive lock for the connection, usually "`FOR UPDATE`".

find_by_sql

When all else fails, you can always specify an SQL query via the `find_by_sql` method. The attributes of the objects are dynamically assigned from the results of the query.

```
>> line_items = LineItem.find_by_sql("SELECT pu.description purchase_description, pr.name product_name
FROM line_items li, purchases pu, products pr WHERE li.purchase_id = pu.id AND li.product_id = pr.id")
=> [#<LineItem >, #<LineItem >, …]
>> line_items[0].purchase_description
=> "Cola delivery"
>> line_items[0].product_name
=> "Cola"
```

Validation

There are other validations aside from `validates_presence_of` listed in the API docs under `ActiveRecord::Validations::ClassMethods`. Here's a list of the methods listed there:

- `validates_acceptance_of` – validates if the specified field is accepted by the user, usually used in terms of service check boxes.
- `validates_associated` – validates if the objects in the specified associations are valid.
- `validates_confirmation_of` – validates if a copy of the specified field (appended with a `_confirmation`, e.g. `email_confirmation` for `email`) is part of the submitted form and matches the specified field. This confirmation field is not saved along with the record though it can be accessed like any other field i.e. it's a virtual field.

- `validates_each` – validates each attribute against a block
- `validates_inclusion_of`, `validates_exclusion_of` – validates if the value of the field is part of (inclusion) or not part of (exclusion) the specified collection
- `validates_format_of` – validates the field against a regular expression
- `validates_length_of` – validates the length of the field
- `validates_numericality_of` – validates the field if it's a number. Can also check if it's a float or an integer.
- `validates_uniqueness_of` – validates whether the field is unique in the table

These validation methods typically have the following options:

- `:message` – allows you to replace the default message. For example you want to change the "can't be blank" message for `validates_presence_of`, you can use:

```
validates_presence_of :name, :message => "is required"
```

- `:on` – specifies when this validation is active (default is `:save`, other options are `:create` and `:update`).
- `:allow_nil` – skips the validation if the attribute is nil
- `:if`, `:unless` – specifies a method, Proc or string to call to determine if the validation should (`:if`) or should not (`:unless`) occur. The method or Proc would be evaluated, while the string would be converted to ruby code then evaluated.

Aside from those validations, you can also use `validate`, `validate_on_create`, and `validate_on_update` to define your own validations. These methods are used just like `before_filter` is used in controllers.

```
class Employee < ActiveRecord::Base

  validate_on_create :check_status

  private
    def check_status
      if status == "Suspended"
        errors.add(:status, "can't be Suspended when creating a new Employee")
      end
    end
end
```

Each Active Record object has an `errors` collection containing the errors collected during validation. In the case above, we added a new error to the name field. Not only will this determine the complete error message to be displayed (the field name is added at the beginning of the message), this will also mark the field as an error on the form as we shall see later.

For error messages that aren't applicable to a single field, you can use `errors[base]`:

```
errors[base] << "You must specify a contact number or an e-mail address"
```

If you want to populate the `errors` collection without saving the object, you can call `validate` on the object. The methods for checking whether the object is valid or not, `valid?` and `invalid?`, also call

validation.

Additional methods for `errors` can be found in the API docs under `ActiveRecord::Errors`. Here are some of the more commonly used `errors` methods:

- `[]` – returns an array of errors for a specified field

```
>> p = Product.new
=> #<Product id: nil, name: nil, ...>
>> p.valid?
=> false
>> p.errors[:name]
=> "can't be blank"
```

- `size` – returns the number of errors in the collection

```
>> p = Product.new
=> #<Product id: nil, name: nil, ...>
>> p.errors.size
=> 0
>> p.valid?
=> false
>> p.errors.size
=> 2
```

- `clear` – removes all errors in the collection

Transactions

You can group database actions into transactions by putting them inside a transaction block:

```
begin
  ActiveRecord::Base.transaction do
    source_account.withdraw(10000)
    dest_account.deposit(10000)
  end
rescue
  # some error handling
end
```

In this transaction method call, a rollback would be issued if an exception is thrown inside the block. Since this is a class method of `ActiveRecord::Base`, you can switch out the `ActiveRecord::Base` with any model class, for example:

```
Account.transaction do
  ...
end
```

You can also use a model object:

```
source_account.transaction do
  ...
end
```

The only problem with using this shorter form is that inexperienced developers might think that the transaction is in the Accounts table level. In reality, the transaction is in the database level; you can use different tables in the transaction:

```
Student.transaction do
  student.save!
  course.save!
end
```

You can also nest transactions. By default all database actions inside the nested transactions are part of the parent transaction. You can pass the option `:requires_new => true` to create a sub-transaction which would roll back only up to the start of the sub-transaction. Either way, the behavior still depends on the database used. For example, if we are using MySQL in the following nested transactions without `:requires_new`:

```
Product.transaction do
  Product.create(:name => 'Test')
  begin
    Product.transaction do
      Product.create('Test2')
      Product.create!()
    end
  rescue
  end
end
```

would not rollback any transaction i.e. "Test" and "Test2" are created. Using `:requires_new => true` would create a `SAVEPOINT` between "Test" and "Test2" so only "Test" is successfully committed after the exception at Product.create!:

```
Product.transaction do
  Product.create(:name => 'Test')
  begin
    Product.transaction(:requires_new => true) do
      Product.create('Test2')
      Product.create!()
    end
  rescue
  end
end
```

We enclosed the sub-transaction with a `begin-rescue` because all exceptions are still propagated upward after the rollback is processed. The only exception that is not propagated by the transactions is `ActiveRecord::Rollback`. For example, the following will behave the same as the previous example even without the `rescue` block:

```
Product.transaction do
  Product.create(:name => 'Test')
  Product.transaction(:requires_new => true) do
    Product.create('Test2')
    raise ActiveRecord::Rollback
  end
end
```

Ruby Corner – Exception Handling

The exception handling structure for Ruby, `begin-rescue-else-ensure-end`, is similar to the `try-catch-finally` structure in Java and C#:

```
begin
  # attempt doing something
rescue
  # handle the exception
```

```
else
  # do this if no exception was raised
ensure
  # do this regardless if there was an exception or not
end
```

You can have named `rescue` blocks to perform different handling depending on the class of the error raised:

```
begin
  # attempt doing something
rescue RuntimeError
  # handle the RuntimeError
rescue TypeError
  # handle the TypeError
else
```

When inside the `rescue` block, the object representing the current exception is placed in a global variable named `$!`. You can change this name by defining another local variable in the `rescue` block:

```
begin
  # attempt doing something
rescue RuntimeError
  puts "A runtime error occurred: #{$!}"
rescue TypeError => t_error
  puts "A type error occurred: #{t_error}"
else
```

You can throw error objects using the `raise` keyword. In its basic form, `raise` accepts an exception class followed by an optional message. You can also skip the class to use `RuntimeError`.

```
irb(main):001:0> raise "This is a runtime error"
RuntimeError: This is a runtime error
        from (irb):1
        from :0
irb(main):002:0> raise ZeroDivisionError, "Hello I am a random zero division error"
ZeroDivisionError: Hello I am a random zero division error
        from (irb):2
        from :0
irb(main):003:0> raise TypeError
TypeError: TypeError
        from (irb):3
        from :0
```

Action Controller

We've had extensive discussions on Action Controller so the topics below aren't that complicated.

Root Routing

We've discussed almost everything about routing, from `match` to nested resources. The other options available for `match` and `resources` aren't that important so just refer to the Routing API docs if you need to use them.

However, you might miss one important routing method while reading through references: the `root`

method. For example:

```
root :to => 'pages#main'
```

This method is basically just another named route so all of the options available for `match` would work in it. What's special about it is that it maps the root of your application to a controller action.

Going to http://localhost:3000/ won't immediately direct you to the `main` action of `pages`, though. As you may recall in the introduction and our asset tags lesson, the `/public` folder is mapped to the root, and like many web servers, WEBrick and Unicorn will serve you `/public/index.html` when you access the root.

To make your `root` route work, you must first delete `/public/index.html`. As a named route, you can also use the generated helpers `root_url` and `root_path` in your controllers and views.

Multiple Representations

You might be wondering about the purpose of this code fragment in our Debts program:

```
def show
  @debt = Debt.find(params[:id])

  respond_to do |format|
    format.html # show.html.erb
    format.xml  { render :xml => @debt }
  end
end
```

The `respond_to` block is processed depending on the MIME type requested by the user, that is, the blocks inside the `format.xxxx` calls are processed when they mach the format requested by the user. In `format.xml`, `render :xml` was called: the `render` option `:xml` renders an XML representation of the object argument.

In the absence of a block, the default rendering is performed. Thus, `show.html.erb` was used to render the Show Debt page. Had we not provided a block to `format.xml`, `show.xml.erb` would have been rendered as the response for a request for an XML.

Typically, the MIME type request is determined through the Accept request header. A browser would typically send an Accept header like this:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

and Rails would determine the format as `html`. In the next chapter, we shall see a request with an "`Accept: text/javascript, …`" that would be handled in a `format.js` block.

Most browsers don't allow the user to define the `Accept` request header. Thus, Rails provides a way to define the format through the `:format` parameter in routes. Recall the `match` default that we removed from `routes.rb`:

```
match ':controller(/:action(/:id(.:format)))'
```

The parsed `:format` parameter overrides the `Accept` header and tells `respond_to` what format to render and return to the user. So going to http://localhost:3000/promos/show/1.xml would process the `format.xml` block inside the `show` action of the `promos` controller, hopefully returning an XML version of the 1st promo in the database.

This format parameter is already added to your routes if you used `resource` or `resources`.

Cookies and Sessions

HTTP is a stateless protocol. Because of this, all web application platforms and web frameworks have their own means of storing the user's state without having to go to the database. In this part, we will be discussing Rail's implementation of the two most common state persistence schemes: cookies and sessions.

Cookies

You can set and retrieve cookies in the user's browser using the cookies hash available in the controller. Accessing the hash would read the request while manipulating the hash would modify the response.

```
# returns the value of the "username" cookie
cookies[:username]

# sets the "theme" key to "blue"
cookies[:theme] = "blue"
```

Cookies can only store string keys and values. You can configure a cookie's settings by passing a hash instead of a string for the value. (The available settings are listed at the API docs under `ActionController::Cookies`)

```
# Sets a cookie that expires in 1 hour.
cookies[:login] = { :value => "XJ-122", :expires => 1.hour.from_now }
```

The delete method can be used to manually delete cookie entries.

```
cookies.delete :theme
```

Session

Like other frameworks, Rails sessions differ from cookies in that while you can only store strings in cookies, you can store (serializable) objects in the session. Unlike other frameworks, however, Rails does not (by default) store the data in the server. Instead, it stores the data as cookies in the user's browser.

While the idea of cookie based sessions might sound weird, there are important reasons behind it. First, the 4KB limit of cookies and the transparency of cookies promotes good practice: large session objects that contain sensitive data are avoided. Also, since the browser handles the cookie expiry, the server doesn't need to waste processing cycles on session cleanup.

One valid concern about cookie based sessions is that cookies can easily be tampered with. To get around this, Rails cryptographically signs the session (you can find the Rails generated key at `config/initializers/session_store.rb`). The users can read the session values but they cannot tamper with them.

With the background out of the way, let's proceed with using the session.

Using a session is easy; it's simply a hash, session, available in the controller and the view (just like params and flash).

```
session[:current_user] = user
```

You can remove a session entry by setting it to `nil`. You can reset the entire session using the `reset_session` method.

Action View

We've already discussed almost all of the topics regarding Action View. Here, we'll just wrap up the topic of form helpers and displaying form errors.

Other Form Helpers

We've already covered `label`, `text_field`, `text_area`, `collection_select`, and `submit` form helper tags. Here are the other form helper tags:

- `check_box` – creates a check box for the specified field. It uses the default values for boolean form fields in Rails ("1" for true, "0" for false) so if you're using the check box for non-boolean values, you can change the "checked" and "unchecked" values through the arguments.
- `file_field` – creates a file upload input tag. Uploading files is not part of the scope of this training course.
- `radio_button` – creates a radio button input tag. The radio buttons are grouped according to the specified field.
- `hidden_field` – creates a hidden input tag.
- `password_field` – creates a password-type input tag.
- `date_select`, `datetime_select`, `time_select` – produces a set of select tags (e.g. hour, month, etc) for the date/datetime/time field.

Full details can be found at the API docs under `ActionView::Helpers::FormHelper` and `ActionView::Helpers::DateHelper`.

Active Support Time Extensions

We encountered an Active Support extension to Ruby in the form of "blank?". The rest of the Active Support extensions can be found in the API docs under `ActiveSupport::CoreExtensions`. In this section, we'll focus on the changes made by Active Support related to numbers and time.

In Rails, all numbers have methods that make it look like they are being converted to time objects. Here are a few examples:

```
Time.now + 1.minute       # returns the time 1 minute from now
1.minute.since(Time.now)  # also returns the time 1 minute from now
1.minute.from_now         # still returns time 1 minute from now
Time.now – 2.days         # returns the time 2 days ago
2.days.ago                # also returns time 2 days ago
```

Rails also has support for time zones. The default time zone is set in `config/application.rb` in the line:

```
  config.time_zone = 'UTC'
```

This assumes that the server, as well as the user, is using UTC. Typically, you set the time zone value per

user programatically inside a filter:

```
class ApplicationController < ActionController::Base
...
  before_filter :set_time_zone
...
    def set_time_zone
      Time.zone = session[:time_zone]

    end
...
```

The time zone setting not only changes how the time zone is displayed (e.g. if you use the `%Z` format in `Time.strftime`), it also affects how records are stored in the database for the user. For example:

Datetime entered by the user (Time Zone = UTC +08:00)	Datetime saved in the database (Time Zone = UTC +00:00)	Datetime displayed the next time the user views the record
06/12/10 01:00 PM	06/12/10 05:00 AM	06/12/10 01:00 PM

This handling saves us the hassle of manually converting time input from foreign users to UTC and back. Rails also handles daylight saving time, yet another possibly problematic issue for time zone handling.

Running `rake time:zones:all` returns all the possible values for the time zone. You can also use `rake time:zones:local` to view the possible values based on your computer's region settings:

```
> rake time:zones:local
(in /home/user/alingnena-app)

* UTC +08:00 *
Beijing
Chongqing
Hong Kong
Irkutsk
Kuala Lumpur
Perth
Singapore
Taipei
Ulaan Bataar
Urumqi
```

Most of the time, you'll probably set the default setting at the `config/application.rb` to:

```
  config.time_zone = 'Taipei'
```

Ajax

Asynchronous JavaScript and XML, better known as Ajax, is a set of technologies used to improve the user's experience when browsing websites. It does this by allowing the users to perform actions and let the browser and server handle it in the background instead having them go from page to page. This makes the user interfaces in websites more dynamic and responsive to the user.

At the core of Ajax is JavaScript which can send requests to the server in the background thanks to the XMLHttpRequest (XHR) object, and can modify the page based on the results thanks to the Document Object Model (DOM). There are 4 main approaches to Ajax depending on how XHR and DOM are used:

1. status codes – XHR sends the request and DOM modifies the page based on the HTTP status code returned by the server.

2. page fragments – XHR sends the request and the server returns a fragment of an HTML page. DOM inserts this fragment somewhere in the current page.

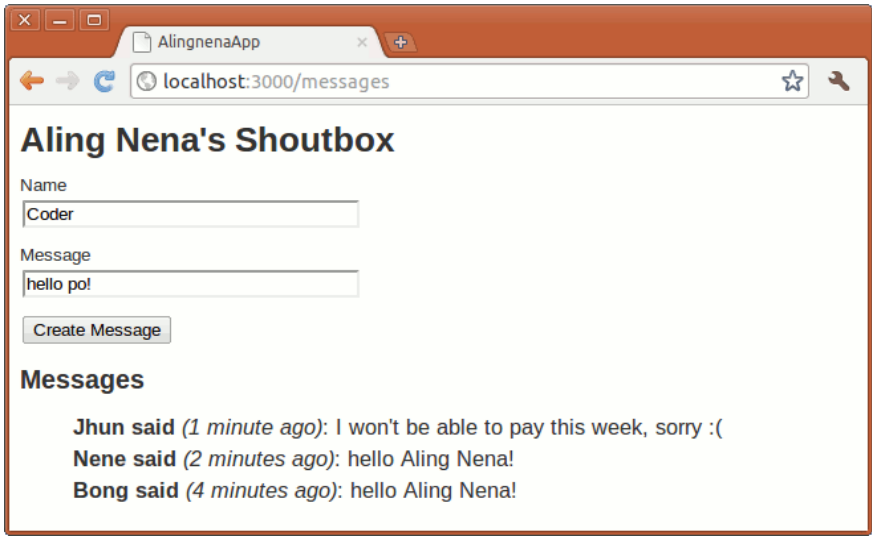
3. code fragments – XHR sends the request and also requests for a JavaScript response. The returned data is in the form of JavaScript code that will modify the page through DOM which is then executed by the browser.

4. data – XHR sends the request and also requests for a specific format (e.g. XML, JSON). The server returns data in the requested format, which will then be processed by JavaScript in the page. This will eventually lead to DOM calls to change the page's contents.

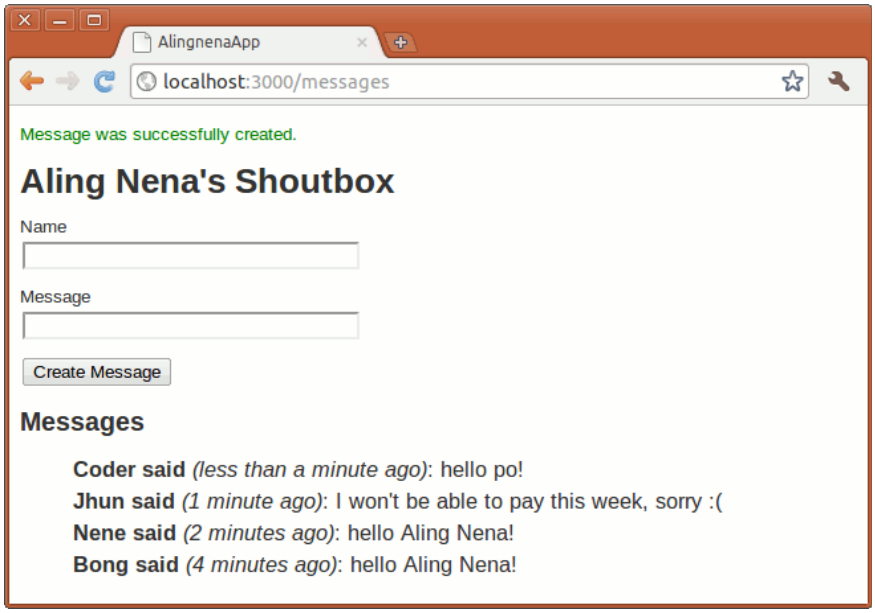
Manually coding XHR and DOM calls can be hard. Fortunately, there are JavaScript libraries available to make these tasks easier. We will discuss two such libraries in this tutorial, Prototype, which comes with Rails, and jQuery, which is used by over 30% of the top 10,000 websites in the Internet.

Prototype

To demonstrate the basic uses of Prototype's Ajax libraries, we'll use a simple shoutbox program that allows visitors of Aling Nena's website to leave a message for all to see:



The program is straightforward: a user can submit his/her name and a message which will be added to the list on the page.



By now you should be able to code this on your own, but if you still can't, just study the code snippets below to have an idea on what you need to do.

Updating a Part of the Page via remote link_to

In this lesson, we want to modify the page so that we could refresh the table to show the new messages without having to refreshing the entire page. The easiest way to do this in Rails 3 would be to use the 3rd Ajax approach, that is, to create JavaScript code which will insert the updated list inside the page, then use Prototype to retrieve run that code.

The first part is easy. We just have to add a new action:

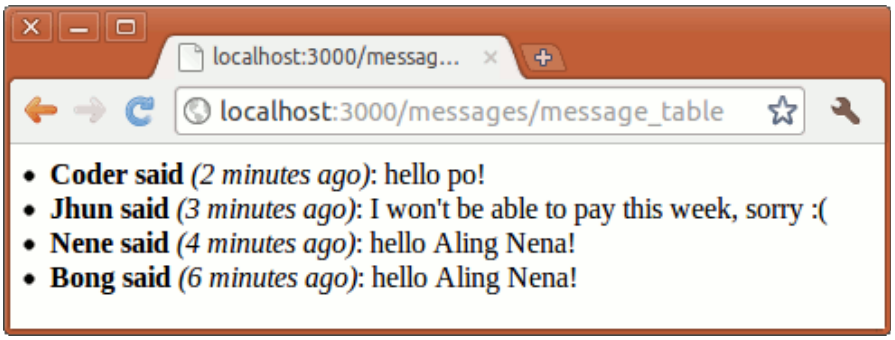
```
# config/routes.rb

  resources :messages do
    collection do
      get "message_table"
    end
  end

# app/controller/messages_controller.rb

  def message_table
    render :partial => Message.order("created_at DESC")
  end
```

and partial for the list:

```
# app/views/messages/index.html.erb

...
<h2>Messages</h2>
<ul style="list-style-type:none;font-size:125%;line-height:150%" id="message_list">
```

```
  <%= render :partial => @messages  %>
</ul>

# app/views/messages/_message.html.erb

<li>
  <strong><%= message.author %> said</strong>
  <em>(<%= time_ago_in_words(message.created_at) %> ago)</em>:
  <%= message.message %>
</li>
```

The partial can now be accessed at http://localhost:3000/messages/message_table



The second part is also easy but will require a bit more effort.

Before we move on, just note that the default Rails JavaScript libraries (which includes Prototype) are already included the `:defaults` argument in the layout so we don't need to change anything or download any plugins to make it work.

```
...
  <title>AlingnenaApp</title>
  <%= stylesheet_link_tag :all %>
  <%= javascript_include_tag :defaults %>
</head>
...
```

RJS and render :update

Since generating this JavaScript DOM code by hand can be hard, Rails provides a way to define these changes with simpler Ruby declarations via RJS templates. These templates are translated to JavaScript and then sent back to the user for the browser to execute.

Just as we can render text directly from the controller, we can also render the JavaScript to be returned to the page directly from the controller without having to create RJS files. For this we need to use the :update option of render. The following code will generate the JavaScript to update the list.

```
# app/controller/messages_controller.rb

  def message_table
    respond_to do |format|
      format.js do
        render :update do |page|
          page.replace_html :message_list, :partial => Message.order("created_at DESC")
        end
      end
```

```
      format.html { render :partial => Message.order("created_at DESC") }
    end
  end
```

Then, all we need to do is to add this link to our page:

```
# app/views/messages/index.html.erb

<%= link_to "Refresh List", message_table_messages_path, :remote => true %>
```

The `:remote => true` option tells Rails that the link should be processed as an AJAX call and not as a normal link. When you click the "Refresh List" link, it will process the RJS that we placed in the controller.

```
page.replace_html :message_list, :partial => Message.order("created_at desc")
```

Translating this code, it will replace (`page.replace_html`) the whole list (id `message_list`) in the page with the partial. You can verify it in the logs that only the `message_table` action was called on click and not the `index` action.

```
Started GET "/messages/message_table" for 127.0.0.1 at [redacted] +0800
  Processing by MessagesController#message_table as JS
  Message Load (0.8ms)  SELECT "messages".* FROM "messages" ORDER BY created_at desc
Rendered messages/_message.html.erb (14.1ms)
Completed 200 OK in 36ms (Views: 34.2ms | ActiveRecord: 0.8ms)
```

Form submission via Ajax

Let's move on from the simple GET example to a slightly more complicated POST example. Here we'll convert our form to use Ajax.

First, modify the `create` action to add the RJS code:

```
def create
  @message = Message.new(params[:message])

  respond_to do |format|
    if @message.save
      format.html { redirect_to(messages_path, :notice => 'Message was successfully created.') }
      format.xml  { render :xml => @message, :status => :created, :location => @message }
      format.js do
        render :update do |page|
          page.replace_html :message_list,
            :partial => Message.all(:order => "created_at DESC")
          page.replace_html :notice, 'Message was successfully created.'
          page[:new_message].reset
        end
      end
    else
      @messages = Message.all(:order => "created_at DESC")
      format.html { render :action => "index" }
      format.xml  { render :xml => @message.errors, :status => :unprocessable_entity }
      format.js do
        render :update do |page|
          page.replace_html :notice, 'There was an error in creating the message.'
        end
      end
    end
  end
```

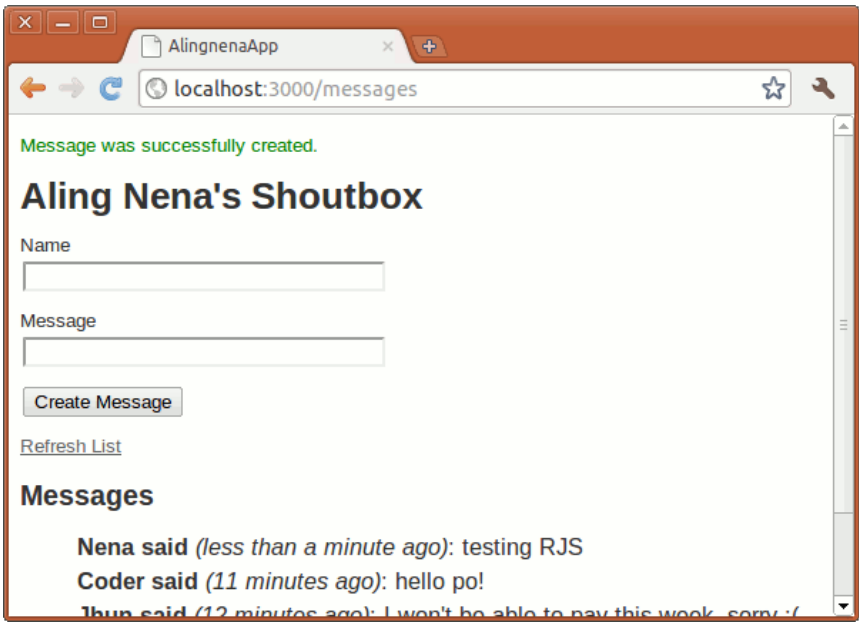
```
  end
```

We also need to update our view in order for the Ajax call to work.

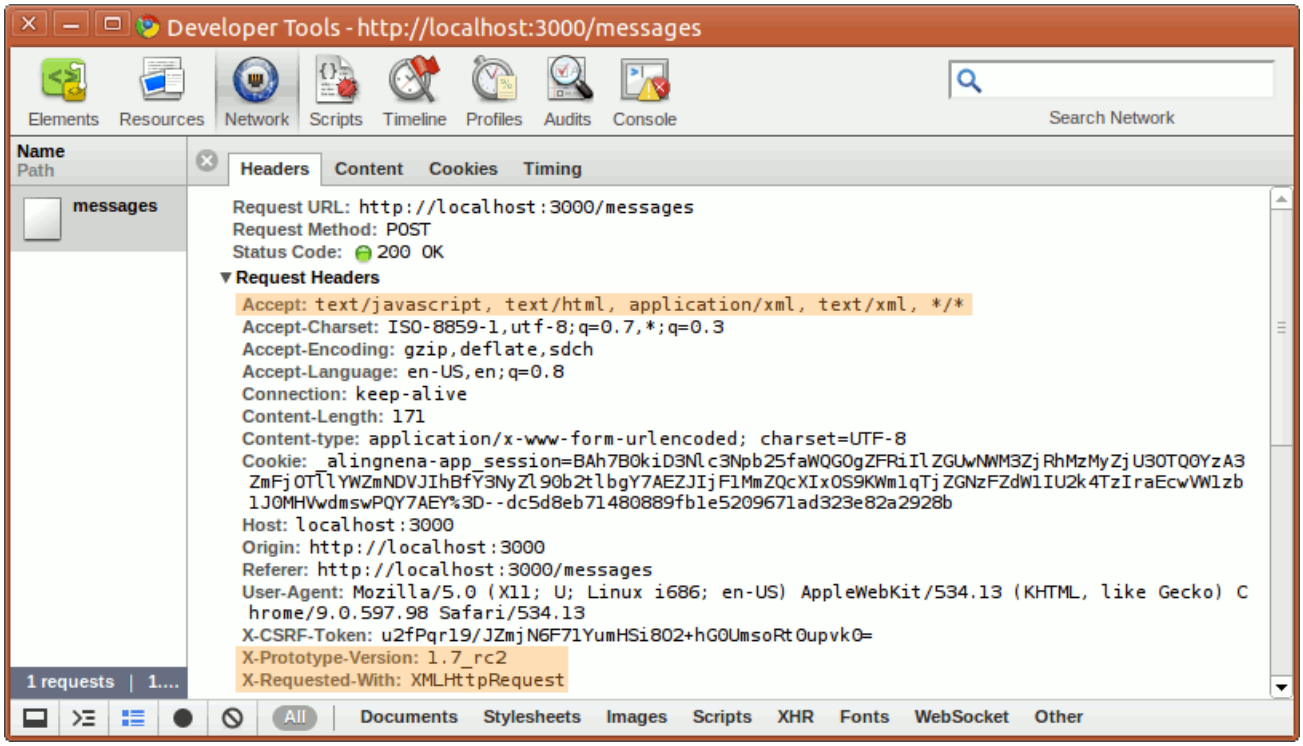
```
# app/views/messages/index.html.erb

<h1>Aling Nena's Shoutbox</h1>

<%= form_for(@message, :remote => true) do |f| %>
  <p>
    <%= f.label :author, "Name" %><br />
...
```

Just like in `link_to`, we only needed to add a `:remote => true` option to tell Rails to process the form submission as an Ajax call.

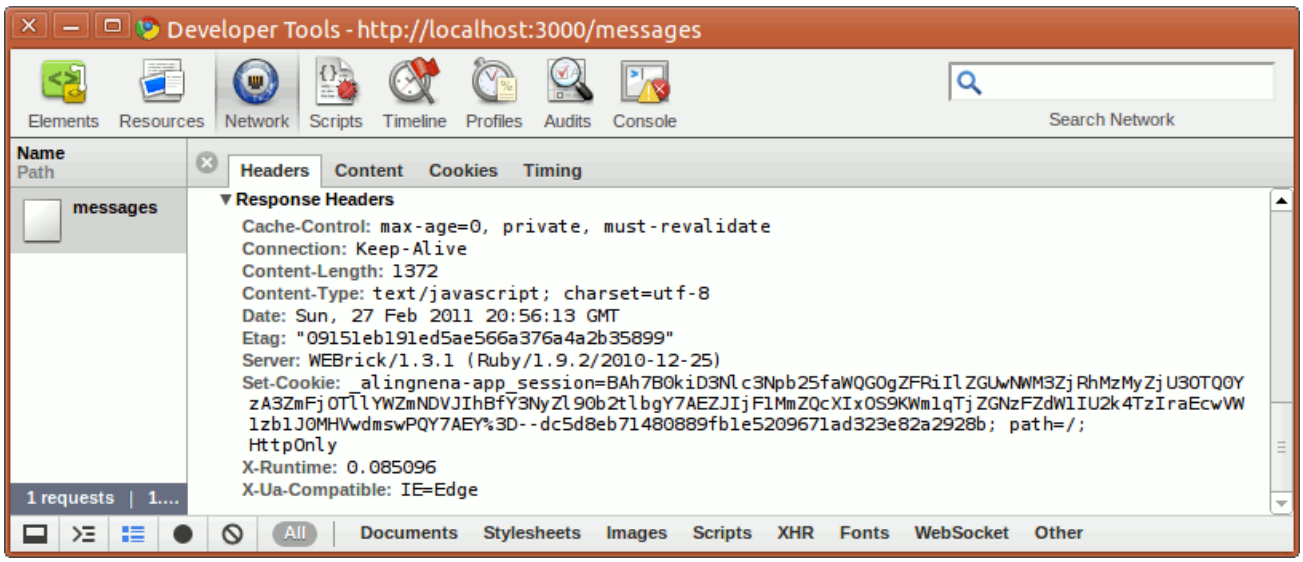
Trying out the code will result in the following:



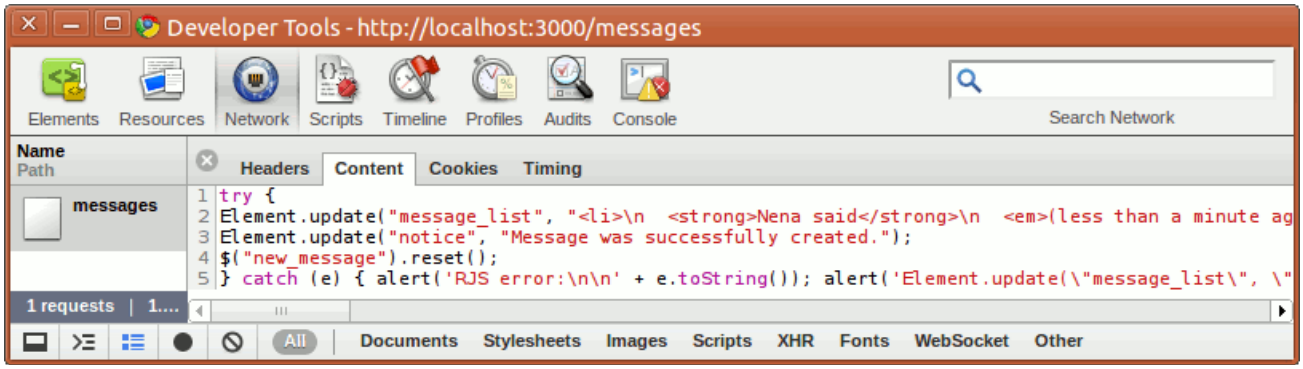
It looks like it behaved normally, but if you use the Mozilla Firefox extension Firebug or Google Chrome's Developer Tools (`Tools -> Developer tools`), you'll notice something different about the request and the response.



From the request headers, we could see that it's using XHR and Prototype. However, the important thing here to note is the `text/javascript` in the Accept header which tells Rails the format the request requires.



As for the response, there's nothing unusual about the headers aside from the explicit disabling of caching. What's interesting here is the body of the response.



As we mentioned before, Ruby code inside the `render :update` is translated to JavaScript. Here we see how the two `page.replace` calls were replaced by `Element.update` calls.

The `page[:new_message]` call, on the other hand, used Prototype's `$()` function which finds the element with the matching `id`. It also applied to the element the method call `reset()` (that we supplied in the controller). Since `reset()` is a valid function for the matched element, it is called normally, thus resetting the form.

Aside from replace_html and the element selector, other methods can be found in the API docs under the `ActionView::Helpers::PrototypeHelper::JavaScriptGenerator::GeneratorMethods` module. For example, you could actually directly copy-paste the API docs sample code for `delay()` to make the notice disappear after 20 seconds:

```
format.js do
  render :update do |page|
    page.show 'notice'
    page.replace_html :message_list,
      :partial => Message.all(:order => "created_at DESC")
    page.replace_html :notice, 'Message was successfully created.'
    page.delay(20) do
      page.visual_effect :fade, 'notice'
    end
    page[:new_message].reset
  end
end
```

RJS templates

By using render :update, we essentially put view processing inside our controller. Not only does it break the MVC paradigm, it also becomes unwieldy once the page manipulation becomes more complex. To solve this problem, we can move the JavaScript generator code into the RJS templates.

RJS templates are simply view files with the `.js.rjs` extension. Just as `format.html` would go to the `.html.erb` file with the same name as the action, by convention `format.js` will process and render the `.js.rjs` file of the same name as the action if the action is called by Prototype.

In our program, we can modify the create action to go to the RJS template for the generator code:

```
def create
  @message = Message.new(params[:message])

  respond_to do |format|
    if @message.save
```

```
        format.html { redirect_to(messages_path, :notice => 'Message was successfully created.') }
        format.xml  { render :xml => @message, :status => :created, :location => @message }
        format.js
      else
        @messages = Message.all(:order => "created_at desc")
        format.html { render :action => "index" }
        format.xml  { render :xml => @message.errors, :status => :unprocessable_entity }
        format.js do
          render :update do |page|
            page.replace_html :notice, 'There was an error in creating the message.'
          end
        end
      end
    end
  end
```

The generator code is then moved to `app/views/messages/create.js.rjs`:

```
page.show 'notice'
page.replace_html :message_list,
  :partial => Message.order("created_at DESC")
page.replace_html :notice, 'Message was successfully created.'
page.delay(20) do
  page.visual_effect :fade, 'notice'
end
page[:new_message].reset
```

The behavior will be the same, though if you would look at the logs, there would be an extra "`Rendering messages/create`" line confirming that `create.js.rjs` was processed.

Unobtrusive JavaScript and graceful degradation

Before we move on to using jQuery, you may want to take note that the JavaScript used in the two examples above are not inline scripts. In other words, the `a` and `form` elements are still basic HTML without any JavaScript within them:

```
<form accept-charset="UTF-8" action="/messages" class="new_message" data-remote="true"
id="new_message" method="post"><div style="margin:0;padding:0;display:inline"><input name="utf8"
type="hidden" value="&#x2713;" /><input name="authenticity_token" type="hidden"
value="Bqyf3d0T+Spa0RpLJqK7YhhSdWAp9tyW080LQcfnBq0=" /></div>

...

</form><a href="/messages/message_table" data-remote="true">Refresh List</a>
```

What Prototype does is look at the elements after the page has loaded and check if the data-remote attribute is set. When set, Prototype adds the necessary event handlers for these elements. Some people call this approach "unobtrusive JavaScript" or UJS, in contrast to old "obtrusive" ways of adding JavaScript into elements via inline scripts.

Aside from being cleaner, UJS has better support for disabled JavaScript. Remember, Ajax's primary goal is to improve user experience. Therefore, it does not make sense for a program to stop working if the user disables JavaScript in his/her browser.

With UJS, one could easily see what would be the behavior in case the JavaScript is disabled. In our example above, `link_to` will just open a list of messages (we could easily redirect this back to `index`) while `form_for` will just behave like it used to.

jQuery

Due to its popularity, we've decided to include the popular JavaScript library, jQuery, into this training course. This library isn't bundled with Rails, though it can be used to either replace or complement Prototype.

When it comes to Ajax, jQuery is suited to handle the 2rd (page fragment), the 3rd (code fragment), and even the 4th (manually parse data) type of approach to Ajax. To keep things simple, we'll just discuss the jQuery equivalent of the RJS example above.

Let's go over the basics of jQuery first before going to Ajax.

Installing the jquery-rails gem

We will be using the `jquery-rails` gem in order to replace the Prototype libraries in our application. To install the gem, modify the `Gemfile` file and add the following line at the end:

```
...
# and rake tasks are available in development mode:
# group :development, :test do
#   gem 'webrat'
# end
gem 'jquery-rails'
```

The `Gemfile` defines the list of RubyGems used by your Rails application. We will discuss more of this later; at the moment let's just use the Bundler tool to install the missing gems from our application using the following command:

```
$ bundle install
Fetching source index for http://rubygems.org/
...
Using rails (3.0.10)
Installing jquery-rails (1.0.13)
Using sqlite3 (1.3.4)
Your bundle is complete! Use `bundle show [gemname]` to see where a bundled gem is installed.
```

Once installed, Rails will have a new generator script which will remove Prototype and replace it with jQuery.

```
$ rails generate jquery:install --force
      remove  public/javascripts/controls.js
      remove  public/javascripts/dragdrop.js
      remove  public/javascripts/effects.js
      remove  public/javascripts/prototype.js
    fetching  jQuery (1.6.2)
      create  public/javascripts/jquery.js
      create  public/javascripts/jquery.min.js
    fetching  jQuery UJS adapter (github HEAD)
       force  public/javascripts/rails.js
```

If you encounter an `certificate verify failed` error, you may temporarily override the default setting for SSL verification by adding the following lines to `config/application.rb`:

```
require File.expand_path('../boot', __FILE__)
require 'openssl'
OpenSSL::SSL::VERIFY_PEER = OpenSSL::SSL::VERIFY_NONE
require 'rails/all'
```

Don't forget to remove the lines once jQuery is installed.

Your application is now using jQuery instead of Prototype. You don't need to change anything from the layout; `javascript_include_tag :defaults` already uses `jquery.js`.

jQuery basics

We can now add code to the latter to run through the basics of jQuery. Add the following code to `/public/javascripts/application.js` (this file is also part of `:defaults`):

```
$(document).ready(function() {
  // do stuff
});
```

Here we can see the first major feature of jQuery: **you can register functions that would handle element events**. In this case, we assigned a function to the ready event of the `document` element of the page. This function would be called when the `document` is ready – something like `window.onload` but works even before all images are loaded.

We could add an alert call to test if it's working:

```
$(document).ready(function() {
  alert("hello world!");
});
```

Open Aling Nena's Shoutbox to see this code in action.

Two things to note about this basic code: first, the `$` is an alias for `jQuery`, thus the call `$(document)` calls the constructor for `jQuery` and in turn creates a `jQuery` object based on `document`. This basically extends the `document` object, allowing us to use jQuery functions like the `ready` event handler we used.

Another thing is the use of an inner function. You could extract the inner function to another function when it becomes more complicated, but since we've been using Procs and Blocks already in Ruby, you should be comfortable with using them for simple tasks.

Using alert for the “hello world!” program doesn't do justice to jQuery's capabilities. Let's modify the program to make it more jQuery-friendly:

```
$(document).ready(function() {
  $("#notice").text("hello world!");
});
```

Here we see the second major feature of jQuery: **you can select elements from the page with CSS selectors**. Here we selected the `p` element with `id="notice"` and called the jQuery manipulation function `text()` to change the text of the element to “hello world!”.

Just like in CSS, selectors can match multiple elements. Calling jQuery functions or registering handlers to a jQuery object will apply those functions and handlers to all elements that the object refers to. For example, using

```
$(document).ready(function() {
  $("li").css("color", "blue");
  $("li").mouseover(function () {
    $(this).css("color", "red");
  }).mouseout(function() {
    $(this).css("color", "black");
```

```
  });
});
```

Will first set the `color` CSS attribute of all `li` elements to blue. We then registered the `mouseover` event handler that would set the `color` to red when the mouse is over the element. Finally, since the `mouseover` function returns the same jQuery element, we can chain another function, this time setting the `mouseout` handler to turn the element to black when the mouse leaves the element.

Now that we're familiar with jQuery, let's use it to replace our RJS Ajax implementation.

Ajax with jQuery and Rails

When we installed the jquery-rails gem, it replaces the behavior of our AJAX-enabled elements (`link_to`, `form_for`) to use jQuery instead of Prototype. If you try using either functions, they will throw an error. We need to replace our RJS with jQuery in order for them to work.

We won't be putting our jQuery code inside `create.js.rjs` simply because it's not RJS code. Instead, we'll put it in an ERb file, `create.js.erb`:

```
$("#message_list").html("<%= escape_javascript(
    render :partial => Message.all(:order => "created_at DESC") ) %>");
$("#notice").show().html("Message was successfully created.").delay(20000).fadeOut("slow");
$("#new_message")[0].reset();
```

This code is just the jQuery equivalent of the RJS code we used in the previous lesson without the visual effects. When the controller renders the JavaScript for `format.js`, it will first look for a `.js.erb` file before looking for a `.js.rjs` file. Thus, the former is rendered and sent back to the browser.

Try out the code to verify if the form submission behaves the same as the RJS implementation.

In case you want to handle the error case, you can simply render the equivalent jQuery code as text in the error handling portion:

```
else
  format.html { render :action => "index" }
  format.xml  { render :xml => @message.errors, :status => :unprocessable_entity }
  format.js { render :text => "$('#notice').show().html('There was an error in creating the
message.')" }
end
```

Simple polling implementation

To wrap up this lesson, let's modify the "Refresh List" code to use jQuery. Also, instead of making the user click the link, let's modify the page such that it automatically polls the server at certain intervals.

To use jQuery, we first remove the RJS code from the controller:

```
  def message_table
    respond_to do |format|
      format.js
      format.html { render :partial => Message.order("created_at desc") }
    end
  end
```

Then we create a new file, app/views/messages/message_table.js.erb, containing our jQuery code:

```
$("#message_list").html("<%= escape_javascript(
     render :partial => Message.all(:order => "created_at desc") ) %>");
```

To move from manual calling to polling, we first remove the link from the page:

```
<% end %>
<%= link_to "Refresh List", message_table_messages_path, :remote => true %>
<h2>Messages</h2>
```

Then we add the polling code to `application.js`:

```
$(document).ready(function() {
  setInterval( function() {
    $.getScript("/messages/message_table.js");
  }, 60000 );
});
```

Here we just used the JavaScript function `setInterval()` to make an Ajax call every minute (60,000 milliseconds). The actual call is done by jQuery's `getScript()` function which makes an asynchronous GET call and executes the returned data as JavaScript.

Automated Testing with RSpec

In this chapter, we will cover the basics of automated testing with RSpec.

Overview to Automated Testing

In traditional software companies, employees are separated into silos according to their purpose: analysts design the system which they pass to developers. The developers code the programs and pass them to QA. The testers test the programs then pass the final working system to Operations.

When a developer from this “assembly-line” encounters the terms “Test Driven Development” or “Automated Testing”, they often ask the same question:

“We already have testers, and we even do developer testing before passing our programs to QA. Why should we spend time programming tests for our programs?”

The answer here always boils down to “immediate feedback”.

Say for example you have a simple table maintenance program. A competent developer can program that in an hour and spend maybe 5 minutes to fully developer test the program before passing it to QA. In this case, coding test for the program feels like a waste of time.

But what if this maintenance program has is slightly more complicated than usual? Maybe there's special handling when the record is saved, or maybe some the flow changes depending on some factors, or maybe the page displays different data on some other factors. Here the combined testing time for the developer and tester might take 30 minutes.

Now what if the modules of that program are used and modified by a new program? In addition to testing the new program, you will have to do some regression testing on the old program, maybe pulling the total testing time up to an hour.

As the system becomes larger and larger, and the modules become more and more interconnected, full regression tests are no longer done except on major releases: no project manager on their right mind would ask the QA team to do a full regression test consisting of a total of over 1,000 test cases for every minor change in the system.

The end result here is always the same: bugs creep into the system and they're reported weeks or months after the change that caused them was applied to the main build.

Automated tests do not guarantee that bugs won't creep into the system. However, they allow the team to detect more bugs and detect them much earlier than manual testing would have.

What would happen if those over 1,000 test cases were automated?

In the developer side, maintenance is no longer like struggling out of quicksand or a tar pit. When maintaining fairly complicated programs, fixing 1 bug produces 2 more bugs somewhere else. Without automated tests to tell you what program you broke because of your fix, those bugs might go undetected for weeks, decreasing the overall stability of the system.

With automated tests, you could immediately know the side effects of your change and act accordingly: either you fix your change or fix the tests that failed because of the new behavior.

In the QA side, the testers are now free from doing trivial and mechanical testing tasks. Instead of

testing if the name field is mandatory for the 100th time, they could spend their time doing exploratory testing, looking for obscure test cases which aren't handled by the current test suite.

In the end, it's all about the return on investment. Your developers are going to spend more time programming because of the tests, but as every software engineer knows, most of the effort is in the bug fixing and maintenance. In large and complicated systems, the return on investment in terms of time and effort is so high that teams who have experienced using automated testing for their projects would never think about going back to full manual testing.

RSpec

Rails has a built in testing framework based on Test::Unit. For this course, we're going to be using a different framework, RSpec.

RSpec is a Behavior Driven Development (BDD) framework. For this lesson, however, we will be using it as a Test Driven Development (TDD) tool. We will discuss the difference between BDD and TDD at the end of this chapter, but we can point out some of the differences between the Test::Unit and RSpec right now:

- "Assertions" are now "expectations", "test methods" are "code examples", and "test cases" become "example groups". This change in semantics reduce the confusion between developer and QA ideas on testing, as "expectations" and "examples" imply that we are also doing design and documentation work, while tests simply imply that there is a function in the system that needs to be tested.

- Specs (files containing one or more example groups) in RSpec are much more human-readable than Test::Unit tests. Well written specs can even be forwarded to your analysts (or even users) for them to verify if the specifications are correct.

Note that we will be using RSpec-2 for this course so most online tutorials/documentation may not be applicable. Please refer to the official docs at http://relishapp.com/rspec for the up to date information.

Installing RSpec

To install RSpec, just add `rspec-rails` under the `:development` and `:test` group in `Gemfile` and let Bundler find and install the necessary dependencies. Here's the updated `Gemfile` without the guide comments:

```
source 'http://rubygems.org'

gem 'rails', '3.0.7'
gem 'sqlite3'
gem 'jquery-rails'

group :development, :test do
  gem 'rspec-rails', '2.5.0'
  gem 'webrat'
end
```

Note that we also added Webrat, an integration testing tool which we will use over Test::Unit for testing views.

Run `bundle install` to install `rspec-rails`, `webrat` and their dependencies:

```
$ bundle install
```

```
Fetching source index for http://rubygems.org/
...
Using rails (3.0.7)
Using jquery-rails (1.0.13)
Installing rspec-core (2.5.2)
Installing rspec-expectations (2.5.0)
Installing rspec-mocks (2.5.0)
Installing rspec (2.5.0)
Installing rspec-rails (2.5.0)
Using sqlite3 (1.3.4)
Installing webrat (0.7.3)
Your bundle is complete! Use `bundle show [gemname]` to see where a bundled gem is installed.
```

As with `jquery-rails`, we need to run a script to install RSpec to your application:

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

As with all Rails test frameworks, tests are executed in the "test" environment as opposed to the default "development" environment. This makes sense because you might want to create specs like "delete all records from table xxxx" and you don't want to let your development test data to be deleted every time you run your tests.

To modify your test environment database settings, go to `config/database.yml` and edit the settings under `test`.

You might have noticed that in the `Gemfile` we placed `rspec-rails` inside a group declaration. Unlike `jquery-rails`, we only use RSpec in development (when we run the generator scripts) and test (when we perform the actual tests). In all other cases, like production and staging environments, RSpec is not necessary. By specifying the correct environment for the gems, we prevent those gems from being installed by Bundler or used by Rails in other environments.

RSpec Generator Scripts

Once `rspec-rails` is added to your application, the generator scripts for `scaffold`, `model` and `controller` are already modified to generate RSpec files instead of the default Test::Unit files.

Let's try creating a new program now with the updated `scaffold` script (you can create a new Rails application for this, we will not be referencing the other programs in this chapter, just don't forget to do the steps in the previous section):

```
$ rails generate scaffold customer name:string active:boolean --webrat
      invoke  active_record
      create    db/migrate/2011xxxxxxxxx_create_customers.rb
      create    app/models/customer.rb
      invoke    rspec
      create      spec/models/customer_spec.rb
       route  resources :customers
      invoke  scaffold_controller
      create    app/controllers/customers_controller.rb
      invoke    erb
      create      app/views/customers
      create      app/views/customers/index.html.erb
      create      app/views/customers/edit.html.erb
      create      app/views/customers/show.html.erb
      create      app/views/customers/new.html.erb
      create      app/views/customers/_form.html.erb
```

```
      invoke    rspec
      create      spec/controllers/customers_controller_spec.rb
      create      spec/views/customers/edit.html.erb_spec.rb
      create      spec/views/customers/index.html.erb_spec.rb
      create      spec/views/customers/new.html.erb_spec.rb
      create      spec/views/customers/show.html.erb_spec.rb
      invoke      helper
      create        spec/helpers/customers_helper_spec.rb
      create      spec/routing/customers_routing_spec.rb
      invoke      rspec
      create        spec/requests/customers_spec.rb
      invoke    helper
      create      app/helpers/customers_helper.rb
      invoke      rspec
      invoke  stylesheets
   identical    public/stylesheets/scaffold.css
```

Note that we added the `--webrat` option in order to generate Webrat expectations inside the generated views.

Now to update the database schema:

```
$ rake db:migrate
$ rake db:test:prepare
```

The `rake db:test:prepare` task copies over the schema from the development database to the test database. If you're using databases like MySQL, you might have to create the databases first:

```
$ rake db:create
$ rake db:create RAILS_ENV=test
$ rake db:migrate
$ rake db:test:prepare
```

Now you can run the server with `rails server`, or you can just run the specs that were generated along with the entire program with:

```
$ rake spec
```

You should see something like:

```
*....*.......................

Pending:
  CustomersHelper add some examples to (or delete) /home/user/alingnena-
app/spec/helpers/customers_helper_spec.rb
    # Not Yet Implemented
    # ./spec/helpers/customers_helper_spec.rb:14
  Customer add some examples to (or delete) /home/user/alingnena-app/spec/models/customer_spec.rb
    # Not Yet Implemented
    # ./spec/models/customer_spec.rb:4

Finished in 0.63858 seconds
29 examples, 0 failures, 2 pending
```

Analysis of the Generated Specs

Let's look at the generated code per Rails module and introduce various RSpec and testing concepts along the way.

Model Specs

Model specs are located at spec/models folder and are named [model_name]_spec.rb. Here's the contents of `spec/models/customer_spec.rb`:

```
require 'spec_helper'

describe Customer do
  pending "add some examples to (or delete) #{__FILE__}"
end
```

The `require 'spec_helper'` call refers to `spec/spec_helper.rb` which provides the libraries for the spec to work and configuration options. Here's the default generated `spec/spec_helper.rb` file:

```
# This file is copied to spec/ when you run 'rails generate rspec:install'
ENV["RAILS_ENV"] ||= 'test'
require File.expand_path("../../config/environment", __FILE__)
require 'rspec/rails'

# Requires supporting ruby files with custom matchers and macros, etc,
# in spec/support/ and its subdirectories.
Dir[Rails.root.join("spec/support/**/*.rb")].each {|f| require f}

RSpec.configure do |config|
  # == Mock Framework
  #
  # If you prefer to use mocha, flexmock or RR, uncomment the appropriate line:
  #
  # config.mock_with :mocha
  # config.mock_with :flexmock
  # config.mock_with :rr
  config.mock_with :rspec

  # Remove this line if you're not using ActiveRecord or ActiveRecord fixtures
  config.fixture_path = "#{::Rails.root}/spec/fixtures"

  # If you're not using ActiveRecord, or you'd prefer not to run each of your
  # examples within a transaction, remove the following line or assign false
  # instead of true.
  config.use_transactional_fixtures = true
end
```

Example Groups

The next part is the example group which is declared by the `describe` method. Following the `describe` are the examples declared by the `it` method. For example, let's remove the pending declaration and add a valid case:

```
require 'spec_helper'

describe Customer do
  it "should create a new instance given valid attributes" do
    Customer.create!(:name => 'John', :active => false)   # will throw error on failure
  end
end
```

One good way of looking at the example group would be to imagine a conversation between you and the user. Therefore:

```
describe Customer do
...
```

```
  it "should create a new instance given valid attributes" do
...
  end
end
```

becomes:

```
You: Describe [the] Customer [model].

User: It should create a new instance given valid attributes.
```

For this spec, our example does not have an explicit expectation. All it does is try to save a record with valid attributes; the example would fail if the `create!` method throws an error. We will look more into expectations when we go into the controller spec.

Set Up and Tear Down

The `before` method contains statements that would be executed before the examples in the group. In testing terms, this would be the "set up" method. The option used here is `:each` which means that the block would be executed for every example. Another possible option is `:all` which runs the block only once before all the examples are executed.

If you need to do things after every example (e.g. delete records created by the example), you could use the `after` method. It has the same options as `before`, but it executes the block after the examples. In testing terms, this would be the "tear down" method.

Controller Specs

Controller specs are located at `spec/controllers/` and are named `[controller_class]_spec.rb`. Here's `spec/controllers/customers_controller_spec.rb`, a far more complicated spec than our model spec (removed some examples to shorten the code snippet):

```
require 'spec_helper'

describe CustomersController do

  def mock_customer(stubs={})
    @mock_customer ||= mock_model(Customer, stubs).as_null_object
  end

  describe "GET index" do
    it "assigns all customers as @customers" do
      Customer.stub(:find).with(:all) { [mock_customer] }
      get :index
      assigns[:customers].should eq([mock_customer])
    end
  end

  describe "GET show" do
    it "assigns the requested customer as @customer" do
      Customer.stub(:find).with("37") { mock_customer }
      get :show, :id => "37"
      assigns[:customer].should be(mock_customer)
    end
  end

...

  describe "POST create" do
```

```
    describe "with valid params" do
      it "assigns a newly created customer as @customer" do
        Customer.stub(:new).with({'these' => 'params'}) { mock_customer(:save => true) }
        post :create, :customer => {'these' => 'params'}
        assigns(:customer).should be(mock_customer)
      end

      it "redirects to the created customer" do
        Customer.stub(:new) { mock_customer(:save => true) }
        post :create, :customer => {}
        response.should redirect_to(customer_url(mock_customer))
      end
    end

    describe "with invalid params" do
      it "assigns a newly created but unsaved customer as @customer" do
        Customer.stub(:new).with({'these' => 'params'}) { mock_customer(:save => false) }
        post :create, :customer => {'these' => 'params'}
        assigns(:customer).should be(mock_customer)
      end

      it "re-renders the 'new' template" do
        Customer.stub(:new) { mock_customer(:save => false) }
        post :create, :customer => {}
        response.should render_template("new")
      end
    end
  end
...
end
```

Unlike in the model where the first `describe` was just there to say that we are testing the Customer class, the first `describe` method here defines the Action Controller class that we test on in this spec. If we put another controller class there instead of `CustomersController`, this spec would test the examples on that controller instead.

Also, since we do not test the class directly like in the model, we must simulate requests from the user. To do that, we can use the `get`, `post`, `put`, and `delete` methods inherited from Action Controller's testing methods. These methods accept an action to test and optional hashes for parameters, session, and flash. For example:

```
  describe "GET show" do
    it "assigns the requested customer as @customer" do
      Customer.stub(:find).with("37") { mock_customer }
      get :show, :id => "37"
      assigns[:customer].should be(mock_customer)
    end
  end
```

This spec introduces the concept of nested example groups. You can nest example groups to group similar example groups for maintainability and readability (like in this spec) or if you want to group examples so that the `before` and `after` method calls would only apply to them.

The "conversation" technique still works with nested examples:

```
You: Describe CustomersController [when you send] POST create with valid params.

User: It assigns a newly created customer as @customer [and]
      it redirects to the created customer.
```

Mocks and Stubs

Database transactions are often the main bottlenecks in web applications. If all of our examples access the database, running specs may take too much time to be practical.

To avoid this penalty, we test our modules in isolation. In other words, the model specs only test the models, the controller specs, the controllers, and the view specs, the views. However, there still lies a problem: our controller and view code uses models, so how to we remove the models from them in our tests?

The answer lies in mocks and stubs.

Mocks are fake objects with the same method signature as the original objects. Because of this, they can be used in place of the object anywhere in our tests. Here's the helper method inside our example group that creates a mock customer object:

```
def mock_customer(stubs={})
  @mock_customer ||= mock_model(Customer, stubs).as_null_object
end
```

Stubs, on the other hand, are fake methods that return a predefined result. For instance, calling `mock_customer(:save => true)` will create a stub in the mock object for `save()` which will return true always.

When you have stubs, there's also the concept of "stubbing" wherein you replace the functionality of a method with a stub. We see this in the GET index example:

```
describe "GET index" do
  it "assigns all customers as @customers" do
    Customer.stub(:all) { [mock_customer] }
    get :index
    assigns[:customers].should eq([mock_customer])
  end
end
```

Here the `find(:all)` method of Customer class is stubbed out with a stub that returns an array of `mock_customer` (i.e. the passed block). By faking the behavior of the model, we can test if the controller is behaving properly without having to access the database.

Accessing Controller Data

You can use the following hashes in the examples:

- `assigns[:key]` – a hash containing the instance variables of the controller.
- `flash[:key]`, `session[:key]` – the flash and session hashes of the controller

Expectations

Expectations replace assertions in RSpec; instead of using `assert(some expression)` we add the call the `should` method. For example:

```
describe "GET index" do
  it "assigns all customers as @customers" do
    Customer.stub(:all) { [mock_customer] }
    get :index
```

```
      assigns(:customers).should eq([mock_customer])
    end
  end
```

Here RSpec verifies if the `@customers` instance variable contains the array of `mock_customer` after the `index` action is processed. If it does, it passes, otherwise the example fails. This would be the output if we replace `[mock_customer]` with `[]`:

```
*.....F.....................

Pending:
  CustomersHelper add some examples to (or delete) /home/user/alingnena-
app/spec/helpers/customers_helper_spec.rb
    # Not Yet Implemented
    # ./spec/helpers/customers_helper_spec.rb:14

Failures:

  1) CustomersController GET index assigns all customers as @customers
     Failure/Error: assigns(:customers).should eq([])

       expected []
            got [#<Customer:0x5027024 @name="Customer_1006">]

       (compared using ==)

       Diff:
       @@ -1,2 +1,2 @@
       -[]
       +[#<Customer:0x5027024 @name="Customer_1006">]
     # ./spec/controllers/customers_controller_spec.rb:17:in `block (3 levels) in <top (required)>'

Finished in 1.41 seconds
29 examples, 1 failure, 1 pending
```

RSpec provides methods for other cases. For example, we could use the `be` method to check for an object's identity (e.g. internal representation) if they are the same object:

```
  describe "GET show" do
    it "assigns the requested customer as @customer" do
      Customer.stub(:find).with("37") { mock_customer }
      get :show, :id => "37"
      assigns[:customer].should be mock_customer
    end
  end
```

One of the key features of should is that it allows us to make our expectations more human-readable than traditional assert calls. For instance, we can further make the expectation above human-readable by removing the parenthesis:

```
      assigns[:customer].should be mock_customer
```

Here's a list of some other expectations:

```
target.should satisfy {|arg| ...}
target.should_not satisfy {|arg| ...}
target.should not_equal <value>
target.should be_close <value>, <tolerance>
target.should_not be_close <value>, <tolerance>
target.should be <value>
target.should_not be <value>
target.should be < 6
```

```
target.should_not eq 'Samantha'
target.should match <regex>
target.should_not match <regex>
target.should be_an_instance_of <class>
target.should_not be_an_instance_of <class>
target.should be_a_kind_of <class>
target.should_not be_a_kind_of <class>
target.should respond_to <symbol>
target.should_not respond_to <symbol>
```

Expectations can also be applied to mock stubs. Here `mock_object` expects to receive a call to `update_attributes`:

```
it "updates the requested customer" do
  Customer.stub(:find).with("37") { mock_customer }
  mock_customer.should_receive(:update_attributes).with({'these' => 'params'})
  put :update, :id => "37", :customer => {:these => 'params'}
end
```

A best practice for writing expectations is to *write only one expectation per example*. This allows you to pinpoint exactly where the problem is when your specs fail.

Isolation and Integration with Views

By default, RSpec will run your controller actions in isolation from their related views. This allows you to spec your controllers before the views even exist, and will keep the specs from failing when there are errors in your views.

If you prefer to integrate views, you can do so by calling `integrate_views`.

```
describe CustomersController do
  integrate_views
...
end
```

When you integrate views in the controller specs, you can use any of the expectations that are specific to views as well.

Response Expectations

There are expectations you can use with the response:

```
response.should be_success
```

Passes if a status of 200 was returned. Note that in isolation mode, this will always return true, so it's not that useful – but at least your specs won't break.

```
response.should be_redirect
```

Passes if a status of 300-399 was returned.

```
get 'some_action'
response.should render_template("path/to/template/for/action")
```

Passes if the expected template is rendered.

```
get 'some_action'
response.should have_text("expected text")
```

Passes if the response contains the expected text

```
get 'some_action'
response.should redirect_to(:action => 'other_action')
```

Passes if the response redirects to the expected action or path. The following forms are supported:

```
response.should redirect_to(:action => 'other_action')
response.should redirect_to('path/to/local/redirect')
response.should redirect_to('http://test.host/some_controller/some_action')
response.should redirect_to('http://some.other.domain.com'
```

View Specs

View specs are located at `spec/views/[controller_name]/` and are named like the views with an additional `_spec.rb` suffix. Here's the contents of `spec/views/customers/ index.html.erb_spec.rb`:

```
require 'spec_helper'

describe "/customers/index.html.erb" do
  include CustomersHelper

  before(:each) do
    assigns[:customers] = [
      stub_model(Customer,
        :name => "Name",
        :active => false
      ),
      stub_model(Customer,
        :name => "Name",
        :active => false
      )
    ]
  end

  it "renders a list of customers" do
    render
    rendered.should have_selector("tr>td", :content => "Name".to_s, :count => 2)
    rendered.should have_selector("tr>td", :content => false.to_s, :count => 2)
  end
end
```

Like the controller specs, you define the view to be rendered in the describe of the first example group.

In this example, we used `assigns[]` to create fake instance variable containing an array of mock objects. By using mocks, we isolate the view from the model.

Along with the `assigns` hash, you can also use the `session` and `flash` hash that we use in the controller spec. Unlike in the controller spec where we could assign the `params` through the get, post, put or delete calls, the `params` is available in the view specs for modification.

rendered.should have_selector

You can test the contents of the response by using Webrat's `have_selector` method. It checks the existence of elements matching the specified CSS selector (http://www.w3.org/TR/css3-selectors/) . It also accepts `:content` and `:count` options to add additional constraints to the pattern search. In our example:

```
  it "renders a list of customers" do
    render
    rendered.should have_selector("tr>td", :content => "Name".to_s, :count => 2)
    rendered.should have_selector("tr>td", :content => false.to_s, :count => 2)
  end
```

The first `have_selector` verifies if there are 2 "Name" inside a `<tr><td>` element in the response, while the second tests if there are 2 "false".

Note that as mentioned in the Controller Specs above, `have_selector` is available to the controller spec if you use the spec in integration mode.

Mocking and stubbing helpers

You need to manually include all the helpers used by the view because it doesn't use the `ApplicationController` where the `helper :all` call resides.

If you wish to isolate the view from the helpers by creating mock/stubbed helper methods, you should use the `view` object. For example, to stub the `display_purchase(invoice)` helper we made back in Associations:

```
view.should_receive(:display_purchase) { ("(no Purchase set)") }
```

Helper Specs

Helper specs are much simpler than the other three spec types so let's just include them under view.

You place helper specs at the `spec/helpers` folder. The naming convention and defining the helper to be tested is the same. For example, the spec for the `InvoicesHelper` we created in the Associations lesson would be `spec/helpers/invoices_helper_spec.rb` and look like:

```
require 'spec_helper'

describe InvoicesHelper do
  describe "display_purchase" do
    it "should display description of the purchase of the invoice" do
      mock_purchase = stub_model(Purchase, :description => "a description")
      mock_invoice = stub_model(Invoice, :purchase => mock_purchase)
      helper.display_purchase(mock_invoice).should include "a description"
    end

    it "should display a default message if there is no purchase" do
      mock_invoice = stub_model(Invoice, :purchase => nil)
      helper.display_purchase(mock_invoice).should == "(no Purchase set)"
    end
  end
end
```

The `helper` object was created by RSpec to include the contents of the class (`InvoicesHelper`) we specified.

(You may want to remove the pending declaration in `CustomerHelper` in preparation for the next section.)

Test Driven Development with RSpec

Let's demonstrate TDD by adding a few simple features to our Customers program:

- When you create a Customer, it should be set to Active (active = true)
- When you delete a Customer, it should not be deleted from the database. Instead it should be set to Inactive (active = false)
- When you view the list of all Customers, only Active Customers shall be displayed
- When you try to view the details of an inactive record, you should be redirected back to the list of Customers
- The Active field should not be displayed at the list of Customers

The first two features are all model based, the next two are for controller, and the last is for the view.

First thing to do is to create pending examples so that we don't have to rely on an external task list. You can do this by defining an example without a block:

```
# spec/models/customer_spec.rb

require 'spec_helper'

describe Customer do
  it "should create a new instance given valid attributes" do
    Customer.create!(:name => 'John', :active => false)
  end

  pending "should set the record as active on create"

  describe "when destroyed" do
    pending "should not delete the record from the database"
    pending "should set the customer as inactive"
  end

end

# spec/controllers/customers_controller_spec.rb

...
  describe "GET index" do
    it "assigns all customers as @customers" do
      Customer.stub(:all) { [mock_customer] }
      get :index
      assigns(:customers).should eq([mock_customer])
    end

    pending "should only retrieve active customers"
  end

  describe "GET show" do
    describe "when the record is active" do
      it "assigns the requested customer as @customer" do
        Customer.stub(:find).with("37") { mock_customer(:active? => true) }
        get :show, :id => "37"
        assigns[:customer].should equal mock_customer
      end
    end
```

```
    describe "when the record is inactive" do
      pending "redirects to the list"
      pending "displays a message"
    end
  end
...

# spec/views/customers/index.html.erb_spec.rb

...
  it "renders a list of customers" do
    render
    response.should have_selector("tr>td", :content => "Name".to_s, :count => 2)
    response.should have_selector("tr>td", :content => false.to_s, :count => 2)
  end

  pending "does not display the Active field"
end
```

Running rake spec would produce:

```
......*.***.*.**.....................

Pending:
  customers/index.html.erb does not display the Active field
    # Not Yet Implemented
    # ./spec/views/customers/index.html.erb_spec.rb:23
  Customer should set the record as active on create
    # Not Yet Implemented
    # ./spec/models/customer_spec.rb:8
  Customer when destroyed should not delete the record from the database
    # Not Yet Implemented
    # ./spec/models/customer_spec.rb:11
  Customer when destroyed should set the customer as inactive
    # Not Yet Implemented
    # ./spec/models/customer_spec.rb:12
  CustomersController GET index should only retrieve active customers
    # Not Yet Implemented
    # ./spec/controllers/customers_controller_spec.rb:20
  CustomersController GET show when the record is inactive redirects to the list
    # Not Yet Implemented
    # ./spec/controllers/customers_controller_spec.rb:33
  CustomersController GET show when the record is inactive displays a message
    # Not Yet Implemented
    # ./spec/controllers/customers_controller_spec.rb:34

Finished in 1.18 seconds
37 examples, 0 failures, 7 pending
```

Red – Green – Refactor

The most basic of ideas in TDD is the concept of "**Red – Green – Refactor**". When performing TDD, you will have go through continuous cycles of these three stages.

Red is the first stage, wherein you write a failing test. This is important: **your test must first fail** before your could proceed to the next stage. If it doesn't fail, it means it isn't testing anything and, as such, a useless test.

Green is the second stage, wherein you **write the minimum amount of code that would make the test pass**. This lets you focus on the feature you are writing and preventing you from "gold-plating" your code with unnecessary features. Whenever you're tempted to add code because you think that you are

going to need it in the future, just remember **YAGNI: you ain't gonna need it**.

Third stage is refactor, wherein you clean up your code if needed. Refactoring is a lot safer here thanks to the tests in place. After refactoring, you proceed to the next feature and do Red – Green – Refactor all over again.

Let's do this in our first feature, setting the default value of active to true. First let's add a failing test:

```
it "should set the record as active on create" do
  customer = Customer.create(:name => "name", :active => false)
  customer.reload
  customer.active.should eq true
end
```

After running rake spec, we now get our "red":

```
......*.F**.*.**.....................

…

Failures:

  1) Customer should set the record as active on create
     Failure/Error: customer.active.should eq true

       expected true
            got false

       (compared using ==)
     # ./spec/models/customer_spec.rb:11:in `block (2 levels) in <top (required)>'

Finished in 0.88928 seconds
37 examples, 1 failure, 6 pending
```

Predicates

Before we proceed with coding the correct code, note that RSpec provides a special method when testing boolean methods: the predicate matchers. For example:

```
customer.should be_active
```

would perform the same checking, but with a much human-readable syntax and error message:

```
1) Customer should set the record as active on create
     Failure/Error: customer.should be_active
       expected active? to return true, got false
     # ./spec/models/customer_spec.rb:11:in `block (2 levels) in <top (required)>'
```

With predicate matchers, all boolean methods can be tested using `be_[method]` with *method* being a boolean method that ends with "?". More information on predicates can be found at the API docs under Matchers.

Going to Green

Back to the TDD walkthrough, turning the red into green should be easy with callbacks:

```
class Customer < ActiveRecord::Base
  before_create :activate
```

```
  private
    def activate
      self.active = true
    end
end
```

Running `rake spec` should now give us green (and yellow):

```
......*..**.*.**.....................
…
Finished in 0.329416 seconds

37 examples, 0 failures, 6 pending
```

Now to the next feature:

```
 describe "when destroyed" do
    fixtures :customers

    it "should not delete the record from the database" do
      customer = customers(:active)
      customer.destroy
      customer.should_not be_destroyed
    end

    pending "should set the customer as inactive"
  end
```

Fixtures

Before we can get the spec to work, we must first understand what fixtures are.

Fixtures are basically test data stored in a file. In the case of RSpec, these are YAML files stored in the `spec/fixtures/` folder. By calling fixtures :customers, we tell RSpec to load the fixtures in the `spec/fixtures/customers.yml` file into the test database. Then we can access these records via the `customers(:key)` call, where :key is the identifier for the record in the YAML file.

Create the file `spec/fixtures/customers.yml` with:

```
active:
  name: Active Customer
  active: true

inactive:
  name: Inactive Customer
  active: false
```

This loads two records in the database, one can be referred as `customers(:active)`, the other `customers(:inactive)`.

So in our example, we retrieved the customer record, called destroy, and expected it not to be destroyed. As expected, this would make the specs fail:

```
  1) Customer when destroyed should not delete the record from the database
     Failure/Error: customer.should_not be_destroyed
       expected destroyed? to return false, got true
     # ./spec/models/customer_spec.rb:20:in `block (3 levels) in <top (required)>'

Finished in 0.91271 seconds
```

```
37 examples, 1 failure, 5 pending
```

Minimum Code

Here's what we need to add to the model to make the spec pass:

```
class Customer < ActiveRecord::Base
  before_create :activate

  def destroy
  end

  private
    def activate
      self.active = true
    end
end
```

Yes, simply overriding the destroy method without doing anything else looks weird. But recall what we said before: *write the minimum amount of code that would make the test pass.* This is the correct approach for the failing example, leaving the "proper" code to the other pending example.

Running rake spec will give us:

```
37 examples, 0 failures, 5 pending
```

Now for the rest of the model specs and code:

```
    it "should set the customer as inactive" do
      customer = customers(:active)
      customer.destroy
      customer.should_not be_active
    end
```

`rake spec` (red):

```
  1) Customer when destroyed should set the customer as inactive
     Failure/Error: customer.should_not be_active
       expected active? to return false, got true
     # ./spec/models/customer_spec.rb:26:in `block (3 levels) in <top (required)>'

Finished in 0.92193 seconds
37 examples, 1 failure, 4 pending
```

Actual code:

```
class Customer < ActiveRecord::Base
  before_create :activate

  def destroy
    self.active = false
  end

  private
    def activate
      self.active = true
    end
end
```

`rake spec` (green):

```
......*.....*.**.....................

Finished in 0.88626 seconds

37 examples, 0 failures, 4 pending
```

At this point, we can do some refactoring. Not on our code, though, but on our examples:

```
  describe "when destroyed" do
    fixtures :customers
    before(:each) do
      @customer = customers(:active)
      @customer.destroy
    end

    it "should not delete the record from the database" do
      @customer.should_not be_destroyed
    end

    it "should set the customer as inactive" do
      @customer.should_not be_active
    end
  end
```

We refactored the retrieve and destroy to the setup method. Running rake spec here would still produce green.

Quick Walkthrough of the Rest of the Features

We won't be introducing anything new so the next pages will just be a walkthrough of the remaining features to be coded.

Spec:

```
    it "should only retrieve active customers" do
      Customer.should_receive(:find_all_by_active).with(true) { [mock_customer] }
      get :index
    end
```

rake spec (red):

```
  1) CustomersController GET index should only retrieve active customers
     Failure/Error: Customer.should_receive(:find_all_by_active).with(true) { [mock_customer] }
       (<Customer(id: integer, name: string, active: boolean, created_at: datetime, updated_at:
datetime) (class)>).find_all_by_active(true)
           expected: 1 time
           received: 0 times
     # ./spec/controllers/customers_controller_spec.rb:21:in `block (3 levels) in <top (required)>'

Finished in 0.91741 seconds
37 examples, 1 failure, 3 pending
```

Actual code:

```
  def index
    @customers = Customer.find_all_by_active(true)
...
```

rake spec (still red):

```
  1) CustomersController GET index assigns all customers as @customers
     Failure/Error: assigns(:customers).should eq([mock_customer])

…
     # ./spec/controllers/customers_controller_spec.rb:17:in `block (3 levels) in <top (required)>'

Finished in 0.9914 seconds
37 examples, 1 failure, 3 pending
```

This time we broke an existing test because of our change. As mentioned before, we have two choices: correct our fix, or correct the failing test. We will go with the latter:

```
    it "assigns all customers as @customers" do
      Customer.stub(:find_all_by_active).with(true) { [mock_customer] }
      get :index
      assigns[:customers].should eq([mock_customer])
    end
```

rake spec (green):

```
......*.......**.....................

Finished in 0.91866 seconds

37 examples, 0 failures, 3 pending
```

New Spec:

```
    describe "when the record is inactive" do
      it "redirects to the list" do
        Customer.stub(:find).with("37") { mock_customer(:active? => false) }
        get :show, :id => "37"
        response.should redirect_to(customers_url)
      end
      pending "displays a message"
    end
```

rake spec (red):

```
  1) CustomersController GET show when the record is inactive redirects to the list
     Failure/Error: response.should redirect_to(customers_url)
       Expected response to be a <:redirect>, but was <200>.
       Expected block to return true value.
     # ./spec/controllers/customers_controller_spec.rb:39:in `block (4 levels) in <top (required)>'

Finished in 0.96183 seconds
37 examples, 1 failure, 2 pending
```

Actual code:

```
  def show
    @customer = Customer.find(params[:id])

    if @customer.active?
      respond_to do |format|
        format.html # show.html.erb
        format.xml  { render :xml => @customer }
      end
    else
      respond_to do |format|
        format.html { redirect_to customers_url }
      end
    end
```

```
  end
```

rake spec (green):

```
......*........*.....................
Finished in 0.348318 seconds

37 examples, 0 failures, 2 pending
```

Spec:

```
    it "displays a message" do
      Customer.stub(:find).with("37") { mock_customer(:active? => false) }
      get :show, :id => "37"
      flash[:notice].should eq "Record does not exist"
    end
```

rake spec (red):

```
  1) CustomersController GET show when the record is inactive displays a message
     Failure/Error: flash[:notice].should eq "Record does not exist"

       expected "Record does not exist"
            got nil

       (compared using ==)
     # ./spec/controllers/customers_controller_spec.rb:44:in `block (4 levels) in <top (required)>'

Finished in 0.95122 seconds
37 examples, 1 failure, 1 pending
```

Actual code:

```
  def show
...
    else
      respond_to do |format|
        format.html { redirect_to customers_url, :notice => "Record does not exist" }
      end
    end
  end
```

rake spec (green):

```
......*.............................

Pending:
  customers/index.html.erb does not display the Active field
    # Not Yet Implemented
    # ./spec/views/customers/index.html.erb_spec.rb:23

Finished in 0.96966 seconds
37 examples, 0 failures, 1 pending
```

Last spec:

```
  it "does not display the Active field" do
    render
    rendered.should_not have_selector("tr>th", :content => "Active")
    rendered.should_not have_selector("tr>td", :content => false.to_s, :count => 2)
  end
```

rake spec (red):

```
  1) customers/index.html.erb does not display the Active field
     Failure/Error: rendered.should_not have_selector("tr>th", :content => "Active")
       expected following output to omit a <tr>th>Active</tr>th>:
...
     # ./spec/views/customers/index.html.erb_spec.rb:25:in `block (2 levels) in <top (required)>'

Finished in 0.969 seconds
37 examples, 1 failure
```

Actual code (remove the active related lines):

```
<table>
  <tr>
    <th>Name</th>
    <th>Active</th>
  </tr>

<% @customers.each do |customer| %>
  <tr>
    <td><%= customer.name %></td>
    <td><%= customer.active %></td>
    <td><%= link_to 'Show', customer %></td>
```

rake spec (still red):

```
  1) customers/index.html.erb renders a list of customers
     Failure/Error: rendered.should have_selector("tr>td", :content => false.to_s, :count => 2)
       expected following output to contain a <tr>td>false</tr>td> tag:
…
     # ./spec/views/customers/index.html.erb_spec.rb:20:in `block (2 levels) in <top (required)>'

Finished in 0.98027 seconds
37 examples, 1 failure
```

Fixing the broken spec:

```
  it "renders a list of customers" do
    render
    rendered.should have_tag("tr>td", "value for name".to_s, 2)
    rendered.should have_tag("tr>td", false.to_s, 2)
  end
```

rake spec (green):

```
.....................................

Finished in 1.18 seconds
37 examples, 0 failures
```

The Last Step

One of the biggest advantages of doing TDD is that you don't need to use the browser when testing your program. One of the biggest disadvantages of doing TDD is that you don't use the browser when testing your program.

As has been stated at the beginning of this chapter, automated testing is not meant to replace manual testing. You must still somehow test your program after you code it; maybe you'll discover incorrect test cases, or maybe you'll find out that you missed some test cases. Putting too much trust on your unit tests

can be a recipe for disaster.

More about Automated Testing

We've only scratched the surface of the large and complicated world of automated testing. Here are some topics that could answer the question “where should I go next?”

Integration Testing

Yet another term that puts developers at odds with the QA people, “integration testing”, in the context of automated testing, simply means a full end-to-end testing of a feature, with the model, view, and controller working together as they should in the real world.

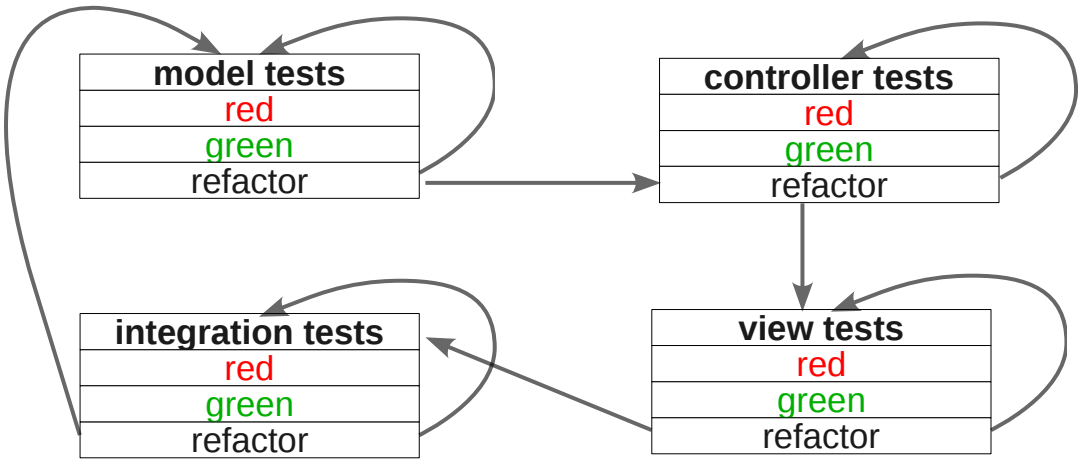
There are different ways of performing integration testing. The first would be through simulating the browser. The *de facto* BDD integration testing tool for Rails, Cucumber, uses this approach with the help of the Webrat library.

Another approach would be to have a tool that records actions from a browser, and then replays the actions again later to test if it produces the expected behavior. Selenium, another testing framework, uses this approach.

Integration tests allow you to automate the testing of features at the User Story level. The drawback here is that integration tests are far slower than their “unit” testing counterparts. Because of this, integration tests are often paired with unit tests; developers run the unit tests while going through the red – green – refactor phase, then they run the integration tests only when they push their changes to the build.

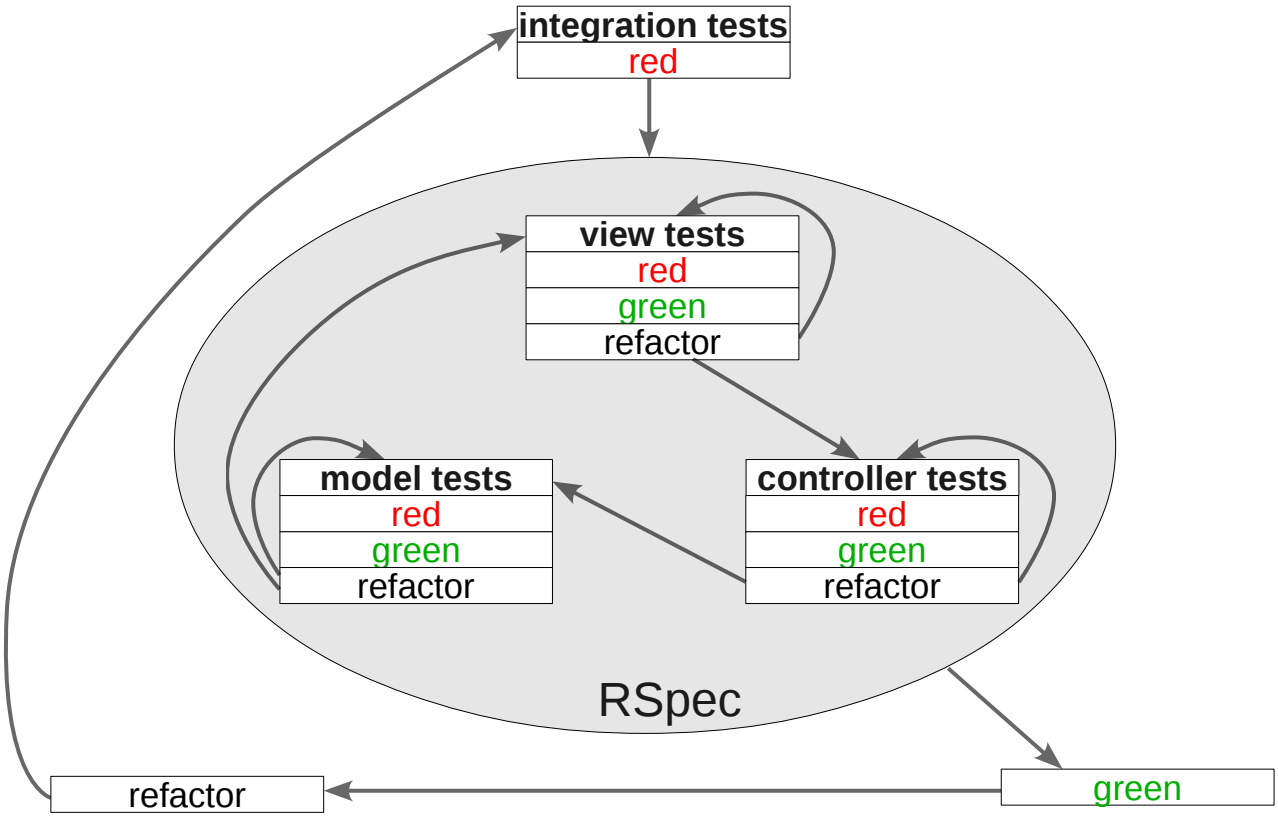
TDD vs BDD

Test Driven Development is a bottom-up approach to development. Before we code a method, we write a test for it. From there we build up our program, writing tests along the way. The flow goes something like this:



Behavior Driven Development, on the other hand, is a top-down approach. We start by translating the user requirements as an integration test, then we go down to the view to translate the UI requirements

to view specs, and so on. This way, we have the user requirements at the forefront of the specs instead of the program's low-level implementation details.



As for which is better, BDD is certainly more pragmatic than TDD. However, being the predecessor, TDD is more widely used than BDD.

Then there's the question of whether to do as the purists do and test everything, or be even more pragmatic and just code what's needed. People in the latter camp mostly use integration tests and only code the most complicated controller and model specs.

Regardless of what camp you fall in, TDD or BDD, full test coverage or just testing features, one thing is clear: any (proper) automated testing is better than no testing at all.

Deployment

This final chapter deals with the various issues and concerns behind deploying your applications.

Configuration

Convention over Configuration means that, in Rails, we don't have to configure most of the settings that need to be configured in a typical web application. However, we still need to configure some settings, especially those settings that change depending on the environment where the application is deployed to or executed.

Rails Settings

You can find the global configuration settings for your Rails application at `config/application.rb`. Here's the default configuration file when you create an application:

```
require File.expand_path('../boot', __FILE__)

require 'rails/all'

# If you have a Gemfile, require the gems listed there, including any gems
# you've limited to :test, :development, or :production.
Bundler.require(:default, Rails.env) if defined?(Bundler)

module AlingnenaApp
  class Application < Rails::Application
    # Settings in config/environments/* take precedence over those specified here.
    # Application configuration should go into files in config/initializers
    # -- all .rb files in that directory are automatically loaded.

    # Custom directories with classes and modules you want to be autoloadable.
    # config.autoload_paths += %W(#{config.root}/extras)

    # Only load the plugins named here, in the order given (default is alphabetical).
    # :all can be used as a placeholder for all plugins not explicitly named.
    # config.plugins = [ :exception_notification, :ssl_requirement, :all ]

    # Activate observers that should always be running.
    # config.active_record.observers = :cacher, :garbage_collector, :forum_observer

    # Set Time.zone default to the specified zone and make Active Record auto-convert to this zone.
    # Run "rake -D time" for a list of tasks for finding time zone names. Default is UTC.
    # config.time_zone = 'Central Time (US & Canada)'

    # The default locale is :en and all translations from config/locales/*.rb,yml are auto loaded.
    # config.i18n.load_path += Dir[Rails.root.join('my', 'locales', '*.{rb,yml}').to_s]
    # config.i18n.default_locale = :de

    # JavaScript files you want as :defaults (application.js is always included).
    # config.action_view.javascript_expansions[:defaults] = %w(jquery rails)

    # Configure the default encoding used in templates for Ruby 1.9.
    config.encoding = "utf-8"

    # Configure sensitive parameters which will be filtered from the log file.
    config.filter_parameters += [:password]
  end
end
```

The first few lines simply loads all of the available packages before running Rails. Here we see how Rails loads the gems defined in Gemfile via Bundler.

The bulk of the file contains global configuration settings. You might remember `config.time_zone` that we already discussed back in the Active Support lesson. The only other default config settings are the encoding settings (set to UTF-8) and filter parameters which replaces all input instances of that field in the server logs to `[FILTERED]` (obviously helpful for password fields).

Environment specific settings exist in our applications. For example, verbose debugging errors may not be as useful in production as it is in development so we set it in the latter and disable it in the former. We can set environment specific settings inside the `config/environments` folder. By default, a Rails application has 3 environments:

- `development` – the default environment, built for development. This is the environment where we've been working on before we went to testing
- `test` – this is the environment where our tests are executed against
- `production` – the environment that the users will eventually use

Each of these environments have a `.rb` file in the `config/environments` folder containing their settings i.e. `development.rb`, `test.rb`, and `production.rb`. Back to our `config.gem` problem, if we add the `config.gem` entries inside `test.rb`, Rails will throw an missing dependency error when we're running test, but will not throw it when we're running either development or production.

Looking at development and production, we could see some significant differences. Here's the default `development.rb` contents:

```
AlingnenaApp::Application.configure do
  # Settings specified here will take precedence over those in config/application.rb

  # In the development environment your application's code is reloaded on
  # every request.  This slows down response time but is perfect for development
  # since you don't have to restart the webserver when you make code changes.
  config.cache_classes = false

  # Log error messages when you accidentally call methods on nil.
  config.whiny_nils = true

  # Show full error reports and disable caching
  config.consider_all_requests_local       = true
  config.action_view.debug_rjs             = true
  config.action_controller.perform_caching = false

  # Don't care if the mailer can't send
  config.action_mailer.raise_delivery_errors = false

  # Print deprecation notices to the Rails logger
  config.active_support.deprecation = :log

  # Only use best-standards-support built into browsers
  config.action_dispatch.best_standards_support = :builtin
end
```

while here is `production.rb`:

```
AlingnenaApp::Application.configure do
  # Settings specified here will take precedence over those in config/application.rb
```

```
  # The production environment is meant for finished, "live" apps.
  # Code is not reloaded between requests
  config.cache_classes = true

  # Full error reports are disabled and caching is turned on
  config.consider_all_requests_local       = false
  config.action_controller.perform_caching = true

  # Specifies the header that your server uses for sending files
  config.action_dispatch.x_sendfile_header = "X-Sendfile"

  # For nginx:
  # config.action_dispatch.x_sendfile_header = 'X-Accel-Redirect'

  # If you have no front-end server that supports something like X-Sendfile,
  # just comment this out and Rails will serve the files

  # See everything in the log (default is :info)
  # config.log_level = :debug

  # Use a different logger for distributed setups
  # config.logger = SyslogLogger.new

  # Use a different cache store in production
  # config.cache_store = :mem_cache_store

  # Disable Rails's static asset server
  # In production, Apache or nginx will already do this
  config.serve_static_assets = false

  # Enable serving of images, stylesheets, and javascripts from an asset server
  # config.action_controller.asset_host = "http://assets.example.com"

  # Disable delivery errors, bad email addresses will be ignored
  # config.action_mailer.raise_delivery_errors = false

  # Enable threaded mode
  # config.threadsafe!

  # Enable locale fallbacks for I18n (makes lookups for any locale fall back to
  # the I18n.default_locale when a translation can not be found)
  config.i18n.fallbacks = true

  # Send deprecation notices to registered listeners
  config.active_support.deprecation = :notify
end
```

We can see here why we don't need to restart the server every time we make a change to our application; in development, the classes and templates are not cached. On the other hand, in production, all classes and templates are cached to improve performance.

Configuration settings for internationalization (i18n) will be discussed in the Internationalization section below. Other configuration settings can be found at the API docs under `Rails::Application`.

Database Settings

We already discussed in the previous chapter where the database settings are located: `config/database.yml`. Here's a sample from an SQLite based application:

```
# SQLite version 3.x
#   gem install sqlite3
development:
```

```
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000

# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
Test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000

production:
  adapter: sqlite3
  database: db/production.sqlite3
  pool: 5
  timeout: 5000
```

And here's one from a MySQL based application (`rails app_name -d mysql`):

```
# MySQL.  Versions 4.1 and 5.0 are recommended.
#
# Install the MySQL driver:
#   gem install mysql2
#
# And be sure to use new-style password hashing:
#   http://dev.mysql.com/doc/refman/5.0/en/old-client.html
development:
  adapter: mysql2
  encoding: utf8
  reconnect: false
  database: mysql_development
  pool: 5
  username: root
  password:
  host: localhost

# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test:
  adapter: mysql2
  encoding: utf8
  reconnect: false
  database: mysql_test
  pool: 5
  username: root
  password:
  host: localhost

production:
  adapter: mysql2
  encoding: utf8
  reconnect: false
  database: mysql_production
  pool: 5
  username: root
  password:
  host: localhost
```

Like in our configuration files, the three default environments are also defined in `database.yml`, each as a YAML entry.

Each environment entry in `database.yml` must at least have an adapter and database value. All the other values are database specific. You can refer to http://wiki.rubyonrails.org/start#database_support for the latest information on installing adapters for different databases as well as the proper `database.yml` settings for those adapters. For example, here's a possible entry for MS SQL Server:

```
development:
  adapter: sqlserver
  mode: ODBC
  dsn: YOUR_DB_DEFINITION_NAME
  username:YOUR_DB_USERNAME
  password:YOUR_DB_PASSWORD
```

You can actually define different databases for each environment, say, SQLite for development, MySQL for testing, and Oracle for production. This is discouraged, however, as the inconsistencies between database implementations might create bugs that won't be detected in development or testing.

Internationalization

Rails provides support for translating text and localizing time through its internationalization (i18n) library. We've already seen the basic settings for this module in config/environment.rb:

```
# The default locale is :en and all translations from config/locales/*.rb,yml are auto loaded.
# config.i18n.load_path += Dir[Rails.root.join('my', 'locales', '*.{rb,yml}').to_s]
# config.i18n.default_locale = :de
```

The first commented setting provides an example of how to add an additional translation path for the translation files. In this case, all `*.rb` and `*.yml` files from the `my/locales` folder are loaded along with the files from `config/locales/` folder.

The second commented setting is an example of how to set the default locale for the application, in this case, German.

There are many ways for setting the user's locale. The simplest would be to have a filter in `ApplicationController` that checks for a locale parameter and assigns it to I18n.locale:

```
before_filter :set_locale

private
  def set_locale
    I18n.locale = params[:locale]
  end
```

Once this is set, going to any page with the locale parameter, say `http://localhost:3000/products?locale=de`, would render a page using the provided locale.

Now that we've set up locale handling, let's move on with the actual internationalization of our application. There are two basic helpers for i18n:

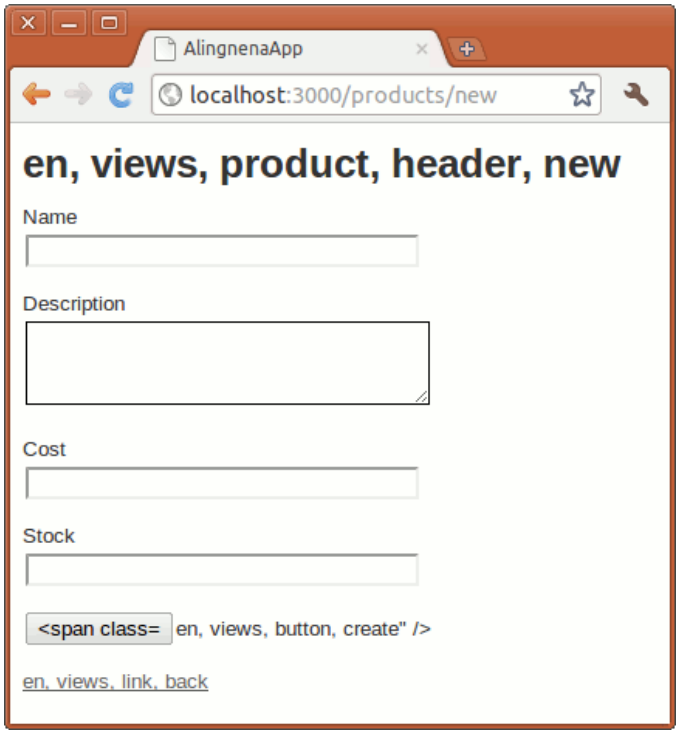
- `t` – alias of translate. Looks up text from the translation files
- `l` – alias of localize. Localizes date and time to local formats

We use these helpers to translate text and localize time in cases where Rails doesn't do the translation and localization for us. For example, field labels and form error messages are automatically translated, but flash messages are not.

Let's start with a simple example, applying i18n to our New Product page.

```
<h1><%=t 'views.product.header.new' %></h1>

<%= render :partial => "form", :locals => { :product => @product, :button_label =>
t('views.button.create') } %>

<%= link_to t('views.link.back'), products_path %>
```

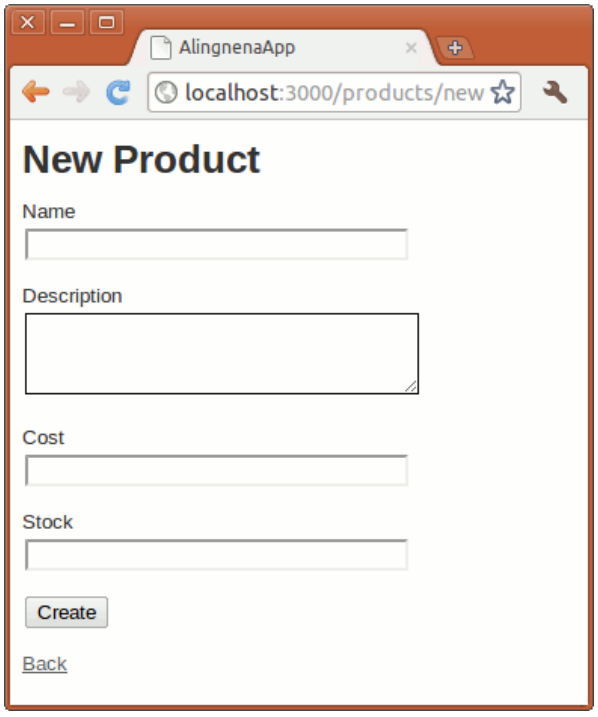
Opening http://localhost:3000/products/new would result in:



We're missing translations for `views.product.header.new`, `views.button.create`, and `views.link.back`. Let's add these to English translation file at `config/locales/en.yml`:

```
en:
  hello: "Hello world"
  views:
    button:
      create: "Create"
    link:
      back: "Back"
    product:
      header:
        new: "New Product"
```

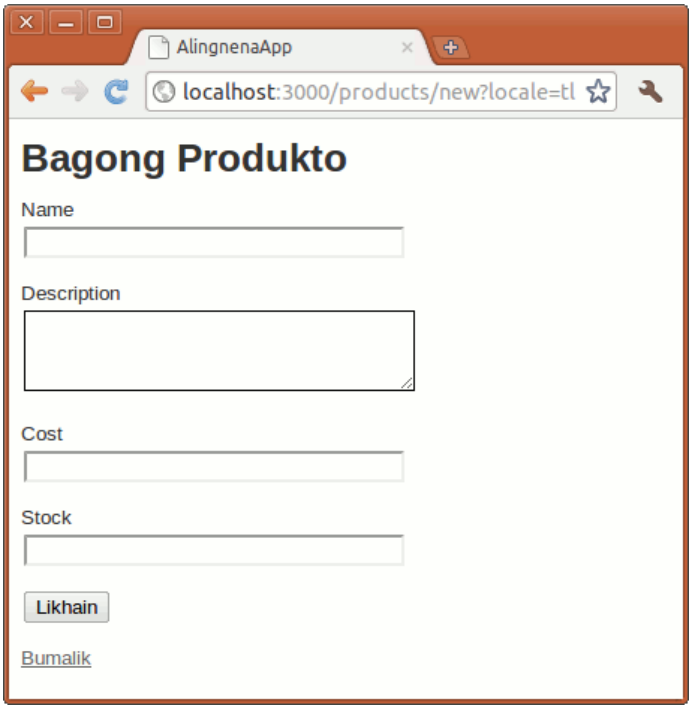
The translation should now work:



Let's try translating the page to Tagalog. Create a new translation file `config/locales/tl.yml`:

```
tl:
  views:
    button:
      create: "Likhain"
    link:
      back: "Bumalik"
    product:
      header:
        new: "Bagong Produkto"
```

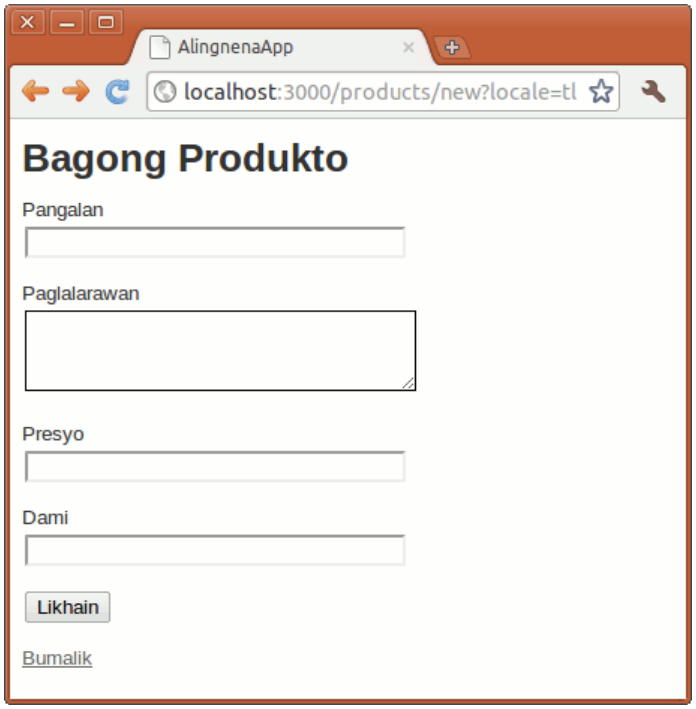
Restart the server then go to http://localhost:3000/products/new?locale=tl:



We're still missing the translations for the fields (let's ignore the title bar). To translate them, we're going to need to add entries for Active Record:

```
tl:
  views:
    button:
      create: "Likhain"
    link:
      back: "Bumalik"
    product:
      header:
        new: "Bagong Produkto"
  activerecord:
    models:
      product: "Produkto"
    attributes:
      product:
        name: "Pangalan"
        description: "Paglalarawan"
        cost: "Presyo"
        stock: "Dami"
```

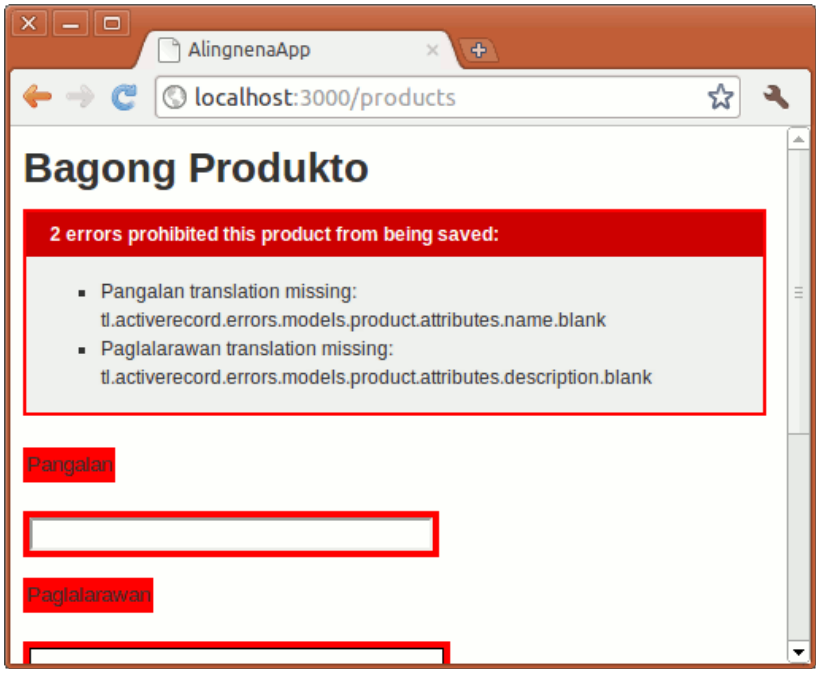
The `activerecord` entry allows us to set the name of the model as well as its attributes/fields.



It looks ok, but if you store the locale into the session in our filter like so:

```
def set_locale
  I18n.locale = params[:locale] || session[:locale]
  session[:locale] = params[:locale] if params.has_key? :locale
end
```

and save a blank product, you'll see that we're missing translations for our errors.



First off, we need to internationalize our error header. This can be done by modifying the line in app/views/products/_form.html.erb to:

```
  <% if product.errors.any? %>
    <div id="error_explanation">
      <h2><%= t "activerecord.errors.template.header", :count => product.errors.count, :model =>
product.class.model_name.human.downcase %>:</h2>

      <ul>
```

And we need to add the following line to config/locales/en.yml:

```
en:
...
  activerecord:
    errors:
      template:
        header:
          one:    "1 error prohibited this %{model} from being saved"
          other:  "%{count} errors prohibited this %{model} from being saved"
```

Here we see two i18n features: pluralization and interpolation.

The pluralization part only requires you to define two translations: one to format the singular message, another to format the plural message. You then pass the :count option to determine if the message is plural or not.

The interpolation feature allows you to interpolate variables inside the messages. You pass these variables as options to `t()` (`:count` included) and the method inserts them to sections defined by "%{}".

Now we are done discussing the header, lets move on to the field errors. Fortunately, they have namespaces to allow you to define the error messages in generic or specific ways.

For example, `validates_presence_of` uses the `:blank` error message. You can define this error message under:

```
activerecord.errors.models.[model_name].attributes.[attribute_name].blank
activerecord.errors.models.[model_name].blank
activerecord.errors.messages.blank
```

That is, Rails will first check if the error message is defined for the specific attribute, then it would check the model, then it would check if a generic message was defined.

Rails i18n provides three values which you can interpolate with your messages, `model` (the model name), `attribute` (the field name), and `count`. For example, our blank message in Tagalog would be:

```
blank: "Dapat ipuno ang %{attribute}"
```

The `count` value is only available for some messages. Here is a list of the default error messages in Rails:

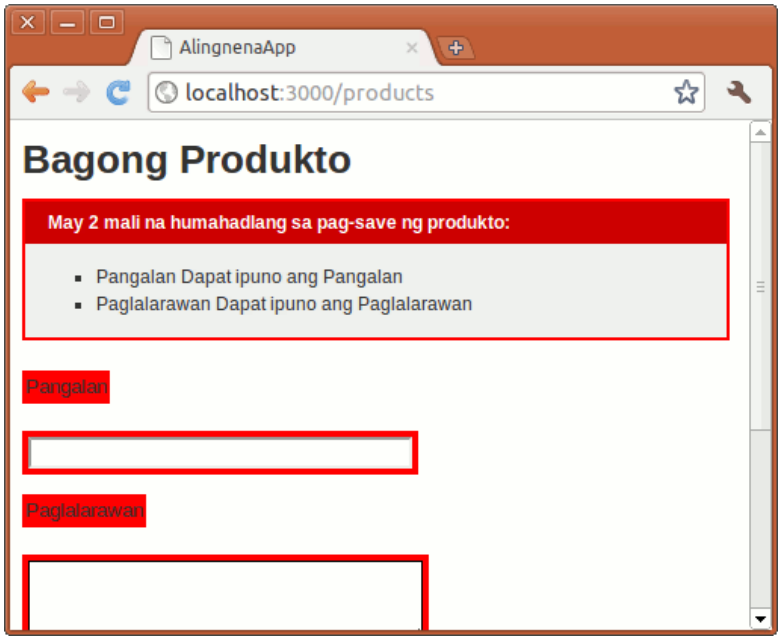
validation	with option	message	default message
validates_confirmation_of		:confirmation	doesn't match confirmation
validates_acceptance_of		:accepted	must be accepted
validates_presence_of		:blank	can't be blank
validates_length_of	:within, :in	:too_short	is too short (minimum is %{count} characters)

validation	with option	message	default message
validates_length_of	:within, :in	:too_long	is too long (maximum is %{count} characters)
validates_length_of	:is	:wrong_length	is the wrong length (should be %{count} characters)
validates_length_of	:minimum	:too_short	is too short (minimum is %{count} characters)
validates_length_of	:maximum	:too_long	is too long (maximum is %{count} characters)
validates_uniqueness_of		:taken	has already been taken
validates_format_of		:invalid	is invalid
validates_inclusion_of		:inclusion	is not included in the list
validates_exclusion_of		:exclusion	is reserved
validates_associated		:invalid	is invalid
validates_numericality_of	:not_a_number	:not_a_number	is not a number
validates_numericality_of	:greater_than	:greater_than	must be greater than %{count}
validates_numericality_of	:greater_than_or_equal_to	:greater_than_or_equal_to	must be greater than or equal to %{count}
validates_numericality_of	:equal_to	:equal_to	must be equal to %{count}
validates_numericality_of	:less_than	:less_than	must be less than %{count}
validates_numericality_of	:less_than_or_equal_to	:less_than_or_equal_to	must be less than or equal to %{count}
validates_numericality_of	:odd	:odd	must be odd
validates_numericality_of	:even	:even	must be even

Our updated `tl.yml` file would be:

```
...
        stock: "Dami"
    errors:
      template:
        header:
          one:   "May isang mali na humahadlang sa pag-save ng %{model}"
          other: "May %{count} mali na humahadlang sa pag-save ng %{model}"
      messages:
        blank: "Dapat ipuno ang %{attribute}"
```

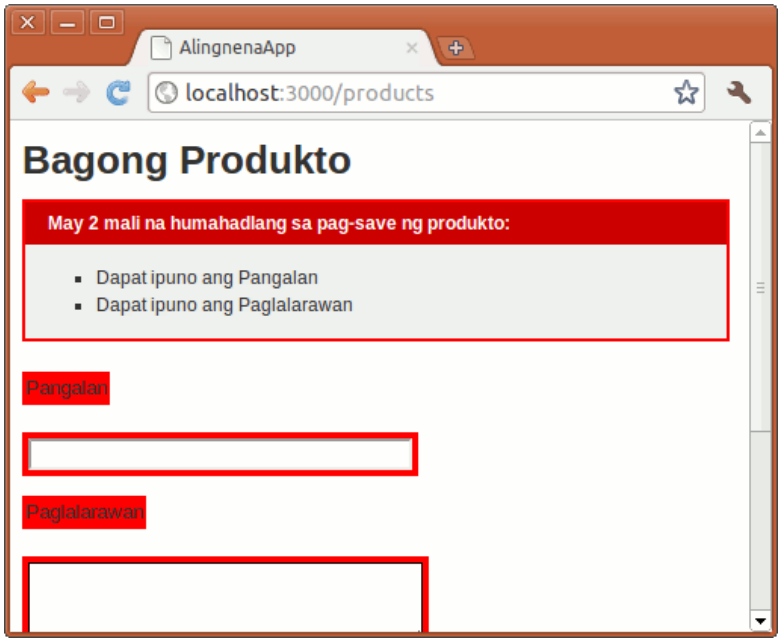
There's still one problem with our error message:



By default, the full error message is in the form "`%{attribute} %{message}`". We can change that format by through the `en.errors.format` entry:

```
tl:
...
  errors:
    format: "%{message}"
```

Now our final message would be:



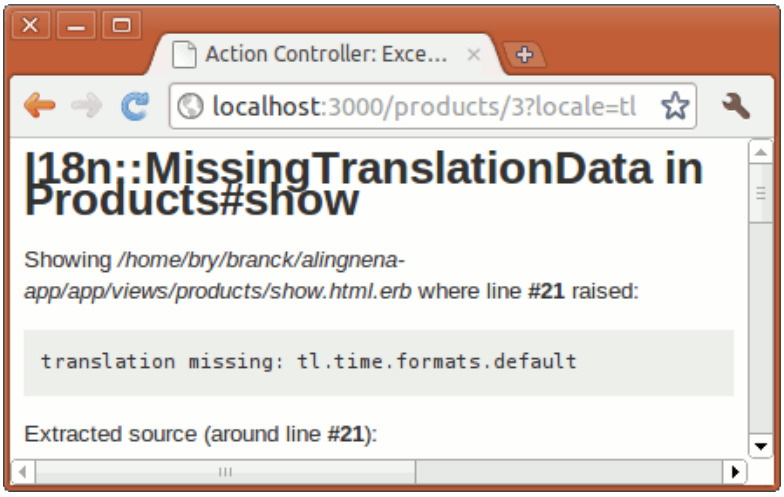
To demonstrate the time localization, let's modify Show Product to add the `created_at` field along with

the other i18n changes:

```
<h1><%=t "views.product.header.show" %></h1>
<p>
  <b><%= label(:product, :name) %>:</b>
  <%= @product.name %>
</p>
<p>
  <b><%= label(:product, :description) %>:</b>
  <%= @product.description %>
</p>
<p>
  <b><%= label(:product, :cost) %>:</b>
  <%= number_to_currency @product.cost, :unit => "PhP" %>
</p>
<p>
  <b><%= label(:product, :stock) %>:</b>
  <%= @product.stock %>
</p>
<p>
  <b><%= label(:product, :created_at) %>:</b>
  <%=l @product.created_at %>
</p>
```

Going to http://localhost:3000/products/3?locale=en will give us (assuming 3 is the id of the Cola entry):

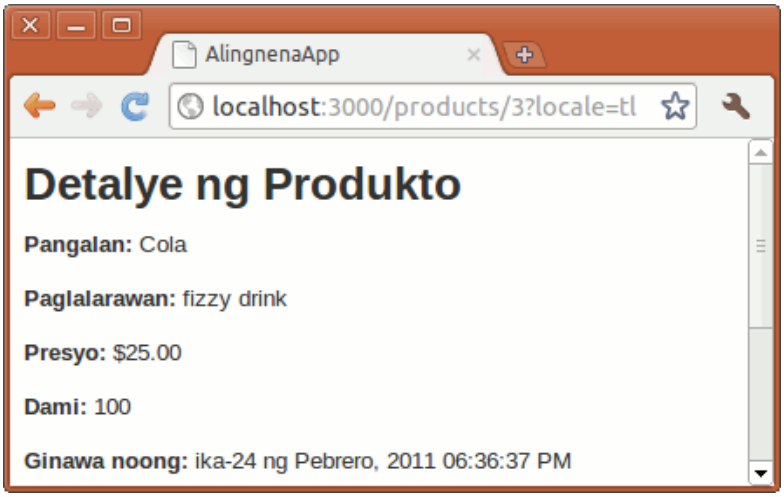
But trying http://localhost:3000/products/3?locale=tl will give us an error:



This is because `l` looks for `time.formats.default` for the default time and datetime format (it looks for `date.formats.default` for date). Let's add the necessary changes to our translation file as well (as some other entries we might need in the future):

```
tl:
  date:
    formats:
      default: "%Y-%m-%d"
      short: "%b %d"
      long: "ika-%d ng %B, %Y"
    day_names: [Linggo, Lunes, Martes, Miyerkules, Huwebes, Biyernes, Sabado]
    abbr_day_names: [Lin, Lun, Mar, Mye, Huw, Bye, Sab]

    # Don't forget the nil at the beginning; there's no such thing as a 0th month
    month_names: [~, Enero, Pebrero, Marso, Abril, Mayo, Hunyo, Hulyo, Agosto, Seteyembre, Oktubre,
Nobyembre, Disyembre]
    abbr_month_names: [~, Ene, Peb, Mar, Abr, May, Hun, Hul, Ago, Set, Okt, Nob, Dis]
  time:
    formats:
      default: "ika-%d ng %B, %Y %I:%M:%S %p"
      short: "%d %b %H:%M"
      long: "ika-%d ng %B, %Y %I:%M %p"
    am: "AM"
    pm: "PM"
  views:
    button:
      create: "Likhain"
    link:
      back: "Bumalik"
    product:
      header:
        new: "Bagong Produkto"
        show: "Detalye ng Produkto"
  activerecord:
    models:
      product: "Product"
    attributes:
      product:
        name: "Pangalan"
        description: "Paglalarawan"
        cost: "Presyo"
        stock: "Dami"
        created_at: "Ginawa noong"
...
```

The date and time formatting options can be found in the API docs under `Time.strftime()`. Applying the changes above produces:

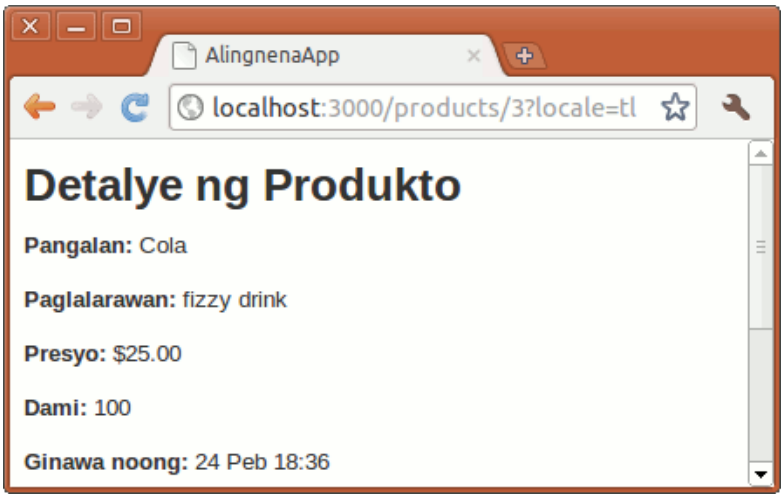


(We removed the `:unit => "PhP"` in preparation for the next section)

The other date and time formats can be specified in the `l` call. For example, if we change the `created_at` `l` call to:

```
<%=l @product.created_at, :format => :short %>
```

We get:

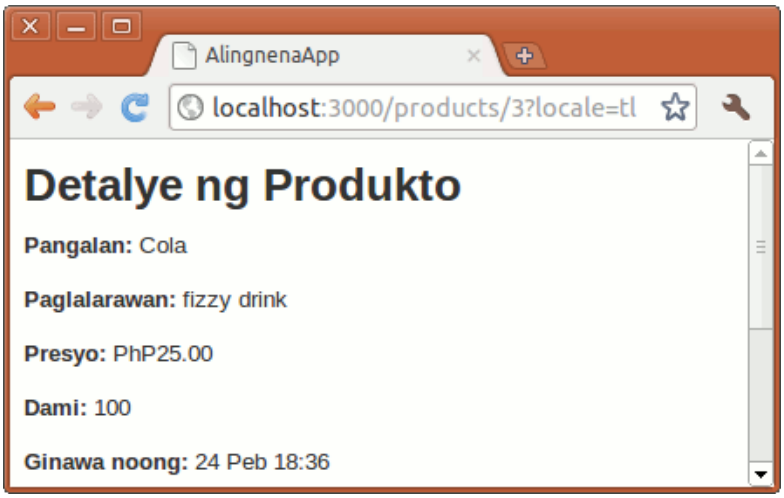


Localization is not limited to time. Currency and numbers are also affected when you use the helper functions `number_to_currency`, `number_with_delimiter`, `number_to_percentage`, `number_to_precision`, and `number_to_human_size`. Note that the text in Cost no longer has a currency symbol and the scale is reduced to 1. Let's fix that by adding the following entries:

```
tl:
  number:
    format:
```

```
      # Sets the separator between the units, for more precision (e.g. 1.0 / 2.0 == 0.5)
      separator: "."
      # Delimits thousands (e.g. 1,000,000 is a million) (always in groups of three)
      delimiter: ","
      # Number of decimals, behind the separator (1 with a precision of 2 gives: 1.00)
      precision: 3
    currency:
      format:
        # Where is the currency sign? %u is the currency unit, %n the number (default: $5.00)
        format: "%u%n"
        unit: "PhP"
        # These three are to override number.format above and are optional
        separator: "."
        delimiter: ","
        precision: 2
  date:
...
```

Opening the page again gives us:



Ruby Corner – Idioms

New Ruby programmers might find the following line weird:

```
I18n.locale = params[:locale] || session[:locale]
```

This statement is equivalent to:

```
if params[:locale].nil?
  I18n.locale = session[:locale]
else
  I18n.locale = params[:locale]
end
```

This is due to how Ruby evaluates the "or" operation; if the `params[:locale]` is not `nil`, the evaluation stops and returns the value of the `params[:locale]`. Otherwise, it will continue the `||` evaluation and check `session[:locale]`. But regardless if `session[:locale]` is `nil` or not, its value will be returned.

Another Ruby idiom was used back in Automated Testing:

```
          @mock_customer ||= mock_model(Customer, stubs).as_null_object
```

This is *roughly* equivalent to:

```
          if @mock_customer.nil?
            @mock_customer = mock_model(Customer, stubs).as_null_object
          end
```

Again, this is due to the `||` handling, though a slightly different one: `a op= b` is equivalent to `a = a op b`. In other words, the line above is also roughly equivalent to:

```
          @mock_customer = @mock_customer || mock_model(Customer, stubs).as_null_object
```

Deployment Options

We've already discussed the different environments available in Rails so running the server in the production environment should be easy. All we need is to change the port number to the default HTTP port (80) and we should be all set:

```
> rake db:migrate RAILS_ENV=production
> rails server -e production -p 80
```

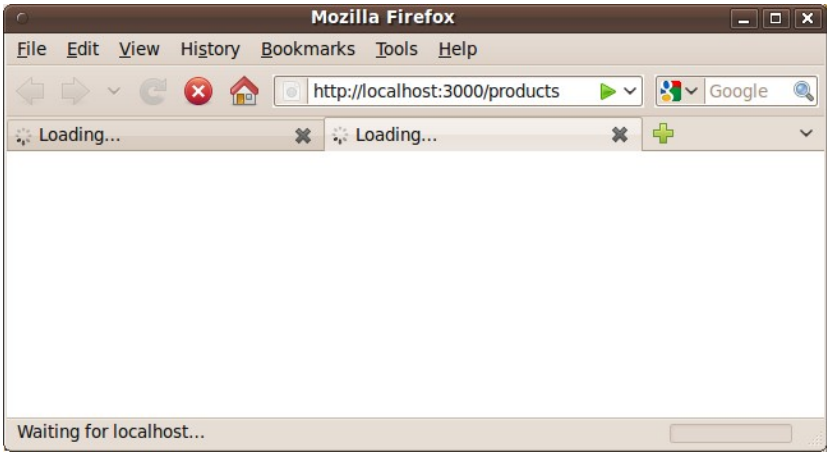
However, there's one big problem to this approach: **Ruby, and by extension Rails and WEBrick, is practically single-theaded**.

Try this experiment: modify the index action of Debt to make it sleep for 30 seconds, simulating an I/O or processing intensive task.

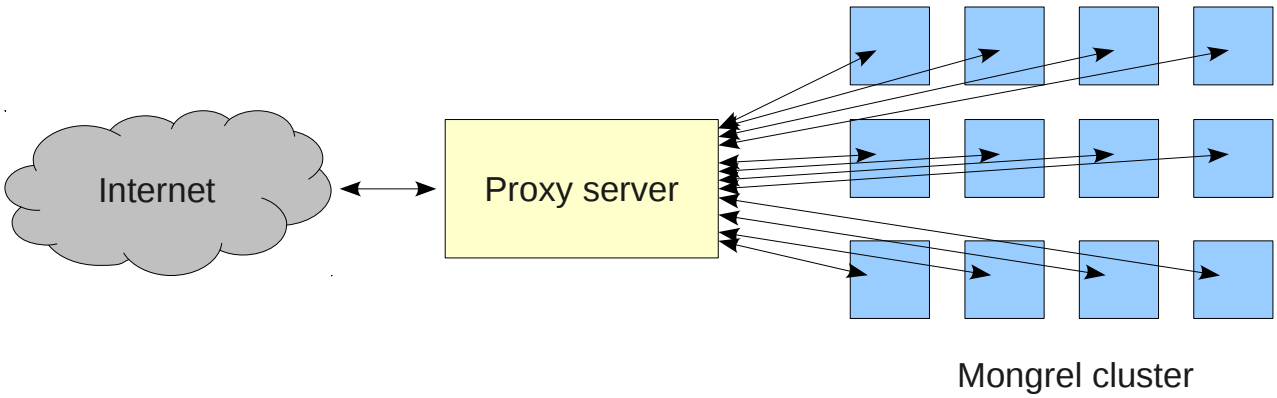
```
def index
  @debts = Debt.all
  sleep 30

  respond_to do |format|
    format.html # index.html.erb
    format.xml  { render :xml => @debts }
  end
end
```

Then open http://localhost:3000/debts. As expected, the page takes 30 seconds to open.

But try opening http://localhost:3000/products in another window while waiting for that page to load. You'll see that the process is blocked.



For years, the “industry-level” solution was to maintain a cluster of Mongrel servers (a faster alternative to WEBrick) and proxy the requests via a proxy server like Apache or lighttpd.



Scaling is easy with this setup; just add additional servers and you're done.

Setup and maintenance, however, was not so easy. You have to configure each of the servers along with the proxy server. You also have to setup mechanisms to handle the rare event that some of the servers crash. It was as if all the ease of development in Rails was offset by the difficulty of setting up the production server.

Then came **Phusion Passenger**.

Phusion Passenger (http://www.modrails.com/) is a module for the industry standard Apache (http://httpd.apache.org/) web server and the fast and lightweight Nginx (http://nginx.org/) web server. It handles the Rails process spawning as well as the proxying to these Rails processes.

Basically what that means is that if you've got Passenger installed and configured inside your Apache or Nginx server, deploying your application is as simple as deploying PHP applications i.e. just upload the files to the server and you're done. No need to worry about clustering or the problems that come with it.

Installing and setting up Phusion Passenger is not included in this course. You can, however, refer to the documentation on the website (http://www.modrails.com/documentation.html) for more details.

Rake

Out last lesson will be a short overview on Rake, the build tool written in Ruby and used extensively by Rails and its plugins.

At the heart of Rake are the rakefiles, files where you specify the processing tasks in Ruby syntax. The simplest way of declaring a task is using the task method. Create a rakefile named rakefile in an empty directory (so as not to overwrite your Rails apps) and put the following code there:

```
task :first_task do
  puts "this is the first task"
end
```

Running the command "`rake first_task`" on the same folder will print out "this is the first task".

Like other build tools like ant, these tasks can be declared as dependent on each other so that executing one task will call the other required tasks. You can declare dependencies by converting the string parameter to the task method to a hash:

```
task :first_task do
  puts "this is the first task"
end

task :second_task => :first_task do
  puts "this is the second task"
end
```

Running rake second_task here will produce:

```
> rake second_task
(in /home/user)
this is the first task
this is the second task
```

You can also use an array as the value of the hash to declare multiple dependencies:

```
task :first_task do
  puts "this is the first task"
end

task :second_task do
  puts "this is the second task"
end

task :third_task => [:first_task, :second_task] do
  puts "this is the third task"
end
```

Running rake third_task will now print the three tasks. Note that when a dependency is already satisfied, it is no longer processed the next time another task requires it. For example:

```
task :first_task do
  puts "this is the first task"
end

task :second_task => :first_task do
  puts "this is the second task"
end

task :third_task => [:first_task, :second_task] do
  puts "this is the third task"
```

```
end
```

Running `rake third_task` here would produce the same results instead of printing "this is the first task" twice.

By the way, "`task :third_task => [:first_task, :second_task] do`" is essentially shorthand for:

```
task :third_task
task :third_task => [:first_task]
task :third_task => [:second_task] do
task :third_task do
end
```

That is, you can add dependencies (and even processing logic) to a task even after the task is initially declared through another `task` call.

Another type of task is the `file` task. Instead of simply checking the dependencies, these tasks also check the existence and the timestamps of the specified files. For example, here's the sample code from the Rake documentation for compiling 3 C files:

```
file 'main.o' => ["main.c", "greet.h"] do
  sh "cc -c -o main.o main.c"
end

file 'greet.o' => ['greet.c'] do
  sh "cc -c -o greet.o greet.c"
end

file "hello" => ["main.o", "greet.o"] do
  sh "cc -o hello main.o greet.o"
end
```

On the first run of rake hello, it will first compile `main.o`, then `greet.o`, then finally the `hello` executable. All three files were created because they did not yet exist; going through the dependencies eventually created them.

Now if you modify the `main.c` and run `rake hello` again, rake will go through the dependencies again, checking each for changes in the timestamp. Since `main.c` was changed, the file task `main.o` was re-run and the file recompiled. When Rake goes back to the `hello` file task, the `main.o` file changed its timestamp so the task is similarly re-executed. Note that since neither `greet.c` nor `greet.o` was changed since the last rake execution, the `greet.o` task was not executed again.

You can describe tasks using the `desc` method on top of the task:

```
desc "prints out the first line"
task :first_task do
  puts "this is the first task"
end

desc "prints out the first two lines"
task :second_task do
  puts "this is the second task"
end

desc "prints out the three lines"
task :third_task => [:first_task, :second_task] do
  puts "this is the third task"
end
```

Running "rake -T" will list out the tasks with a proper `desc` description.

```
> rake -T
(in /home/user)
rake first_task   # prints out the first line
rake second_task  # prints out the first two lines
rake third_task   # prints out the three lines
```

Since `desc` is the standard way of documenting tasks, you can use this command to list the rake tasks for Rails:

```
> rake -T
(in /home/user/alingnena-app)
rake db:abort_if_pending_migrations       # Raises an error if there are pending migra...
rake db:charset                           # Retrieves the charset for the current envi...
rake db:collation                         # Retrieves the collation for the current en...
rake db:create                            # Create the database defined in config/data...
rake db:create:all                        # Create all the local databases defined in …
rake db:drop                              # Drops the database for the current RAILS_ENV
rake db:drop:all                          # Drops all the local databases defined in c...
rake db:fixtures:identify                 # Search for a fixture given a LABEL or ID.
rake db:fixtures:load                     # Load fixtures into the current environment...
rake db:migrate                           # Migrate the database through scripts in db...
...
```

The default task for Rake is the "`default`" task. Usually you just point it to another task or tasks to define which tasks should be run if `rake` is executed without arguments.

```
task :default => :third_task
```

Other tasks and tips on using Rake are found at the online documentation (http://rake.rubyforge.org/).