Lesson 1: Introduction to Ruby on Rails

Introduction to Rails

Your First Program

Restart the Web Server

Viewing Your Rails Application

What Just Happened?

Editing Your First Page

The Ruby Language

Adding Ruby Content to a Web Page

Congratulations!

Quiz 1 Project 1

Lesson 2: Your First Database Application

Creating Applications that Store Data

What Will the Code Need to Do?

Generated CRUD Code is Called "Scaffolding"

Step 1: Generate the Scaffolded Code

What Code Gets Created?

Step 2: Create Storage for the Task Data

Now Try Out the Scaffolded Pages

Project 1 Quiz 1

Lesson 3: Modifying a Database Application

Modifying the Scaffolding

Edit show.html.erb

Preserving Line Breaks

Helper Functions

What Just Happened?

Functions That Take Multiple Arguments

Alternative Helper: time_ago_in_words

What Just Happened?

Functions with Options

What Just Happened?

Project 1

Lesson 4: Adding Field Validation

Sometimes You Need Rules

Making Fields Mandatory

Let's Review

Making More Than One Field Mandatory

Unique Task Names

Checking Numbers

What Just Happened?

Quiz 1 Project 1
Lesson 5: **Adding Custom Ruby Code**

Making Your App Smarter with Ruby

Writing Custom Validation

  Validations Fail if They Create a Message

A Little More Ruby

Validate Every Time?

Hide Time Remaining?

  `<%= ... %>` Contains a VALUE
  `<% ... %>` Contains LOGIC

What Just Happened?

Quiz 1

Lesson 6: **Custom Database Queries**

Asking New Questions

Behind the Scenes: How Rails Displays a Task

The ActiveRecord Library Talks to the Database

How to Read All the Tasks

Reading Single Tasks from the Array

Searching for Data

Finding All Matches

  Which Tasks Are Due on November 6th, 2014?
  How Many Tasks Are an Hour Long?

Sorting Results

What Just Happened?

Quiz 1

Lesson 7: **Adding Custom Data Pages**

Putting Data on the Page

Introducing the Controller

How to List Only Incomplete Tasks

How Do You See the Complete Tasks?

Routes Define a Path through the App

Creating a Route

Creating the Template File

What Just Happened?

  Project 1 Quiz 1

Lesson 8: **Layouts**

The Look and the Links

Adding the Link to the Other Pages

  What is a Rails Layout?

Change the Page Layouts with application.html.erb

Creating Separates for Different Page Groups

  How Do You Make a Link Not a Link?

What Just Happened?

  Quiz 1 Project 1
Lesson 9: **Partials**
- Sharing Page Fragments with Partials
- Creating Your First Partial
- Removing More Duplication
- Sharing Code is Good
  - Adding a Second Table to a Page
- How Do We Display the Overdue Tasks?
- Why Do Local Variables Help?
- What Just Happened?
  - Project 1 Quiz 1

Lesson 10: **Logging In**
- Adding Security to the Incomplete Tasks Page
- How Basic Authentication Works
- Securing the Other Task Pages
  - The before-filter Step by Step
- What Just Happened?
  - Quiz 1 Project 1

Lesson 11: **Managing Users**
- Managing Multiple Users
- Authenticating Against the Database
- Displaying the Current User in the Application
- What Just Happened?
  - Project 1 Quiz 1

Lesson 12: **Associating Data with Users**
- Giving Users Their Own Tasks
- Database Relationships
  - Keys and Foreign Keys
- Add the user_id with a Migration
- Joining Tables Together
- Setting the user_id of a Task
- Finding a Task's User
- What Just Happened?
  - Quiz 1 Project 1

Lesson 13: **Creating Tasks for a User**
- Recording Who Created a Task
  - Determining Which Code Creates a Task
- Finding All of a User's Tasks
- has_many is the Reverse of belongs_to
- The Collection of Tasks Has Its Own Finders
- What Just Happened?
  - Project 1 Quiz 1

Lesson 14: **Creating a Search Feature**
- Forms: Adding Search
  - Adding a Custom Search Route
Lesson 15: **Adding Fields**

Adding Fields to Your Application

Add the Field to the Database

Add the Field to the View

Adding the Priority to the Show Task Page

Validating the Priority Attribute

What Just Happened?

Quiz 1 Quiz 2 Project 1

---

*Copyright © 1998-2014 O'Reilly Media, Inc.*

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See [http://creativecommons.org/licenses/by-sa/3.0/legalcode](http://creativecommons.org/licenses/by-sa/3.0/legalcode) for more information.
Introduction

Welcome to the O'Reilly School of Technology's Introduction to Rails course.

Course Objectives

When you complete this course, you will be able to:

- use code generation to get an application up and running.
- model complex business data with database relationships.
- add login security to your application.
- manage your web interface efficiently with partial templates and layouts.
- add search to your application.

Lessons in this course begin with a list of the specific goals for each lesson, like this:

Lesson Objectives

When you complete this lesson, you will be able to:

- navigate the O'Reilly School of Technology UserActive approach to learning.
- navigate in the OST learning environment called CodeRunner.
- write and run a basic program using Rails.

Before we begin, let's talk a bit about the programming environment you'll be using for the course. We've installed all of the software you'll need and configured it for you so you can get up and running quickly. Everything you'll need is available through a web browser. You don't need to download or install anything, it's all here, ready for you to use!

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a user-active approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you learn to learn a new skill. We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you learn to learn. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, and you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you do break something, that's an indication to us that we need to improve our system!

- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.

- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.

- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from
Accept guidance, but don't depend on it. Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.

- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

## Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll type the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

```
CODE TO TYPE:

White boxes like this contain code for you to try out (type into a file to run).
If you have already written some of the code, new code for you to add looks like this.
If we want you to remove existing code, the code to remove will look like this.
We may also include instructive comments that you don't need to type.
```

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

```
INTERACTIVE SESSION:

The plain black text that we present in these INTERACTIVE boxes is provided by the system (not for you to type). The commands we want you to type look like this.
```

Code and information presented in a gray OBSERVE box is for you to inspect and absorb. This information is often color-coded, and followed by text explaining the code in detail:

```
OBSERVE:

Gray "Observe" boxes like this contain information (usually code specifics) for you to observe.
```

The paragraph(s) that follow may provide addition details on information that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

<table>
<thead>
<tr>
<th>Note</th>
<th>Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tip</td>
<td>Tips provide information that might help make the tools easier for you to use, such as shortcut keys.</td>
</tr>
<tr>
<td>WARNING</td>
<td>Warnings provide information that can help prevent program crashes and data loss.</td>
</tr>
</tbody>
</table>

## The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's
a quick overview of the various areas of the screen:

These videos explain how to use CodeRunner:

File Management Demo
Code Editor Demo
Coursework Demo

Introduction to Rails

What is Ruby on Rails? Ruby on Rails, often shortened to Rails, is an open source, full-stack web application framework for the Ruby programming language. Ruby on Rails runs on the general-purpose programming language Ruby, which predates it by more than a decade.

You may have used Rails before without ever realizing it. Rails powers websites like Hulu, Yellow Pages, and Twitter. It's also used frequently by high-tech startups, because Rails development is fast. Most websites perform many of the same kinds of tasks, for instance, store information that a user has entered into a web page. Rails uses code generation to do lots of the most common types of programming for you.

Rails development usually looks like this:

You generate some code, modify it, and then repeat; this means Rails applications can be created quickly. In fact, by the end of this lesson, you will have created your very first Rails application!

Your First Program

Rails is used to create complete websites; a website such as Twitter is a single web application. That's a little different
from most other types of web programming. If you've used Perl or PHP, you've probably created websites one page at a time. Each page on the site is like a separate program. Rails applications are different. A Rails application needs lots of files and pages before it runs, but the good news is that Rails can generate most of the files itself.

Let's get started.

Open a Unix shell by clicking the **New Terminal** icon at the top of the Code Editor window:

You're prompted for your login and password. The login name will appear automatically. If for some weird reason it doesn't, you can find it on the Student Start Page listed as "Your Sandbox login." Remember this login name, because you'll need it again shortly.

```
INTERACTIVE SESSION:

cold1 login: login name
Password: 
cold1:~$
```

You're prompted for your login and password. The login name will appear automatically. If for some weird reason it doesn't, you can find it on the Student Start Page listed as "Your Sandbox login." Remember this login name, because you'll need it again shortly.

Once you log in, you see the **cold1:~$** prompt. This means the computer is ready to accept your instructions. Create a directory that will contain your Rails applications. Type these commands *exactly as you see them here*:

```
INTERACTIVE SESSION:

cold1:~$ mkdir railsapps (if the directory already exists, just continue to the next command)
cold1:~$ cd railsapps
cold1:~/railsapps$
```

**WARNING** The commands you need to type are `mkdir railsapps` and `cd railsapps` as shown in blue. Don't type the `cold1:~$` parts—they're just *prompts* from the computer that tell you it's ready to go.

```
Note Did you notice that the prompt changed to "cold1:~/railsapps$" after you entered the `cd` command? The prompt always shows you which directory you are in currently. Whenever you run commands in the Unix shell, make sure you are in the correct directory.
```

The first command creates a directory named `railsapps`; the second command moves you into that directory. It's extremely important that you name this directory `railsapps`, because it's the name our servers expect. If you name it anything else, your Rails programs won't work.

Now that you're in the `railsapps` directory, you're ready to create your first Rails application. We'll name the application `ostapp`. Type the command exactly as you see it. The server is configured to expect an application named `ostapp`:
Rails generates a large number of files and directories—far more than we show here. Don’t worry about the number of files Rails generates. Most of them are just default code that you won’t need to change.

All of those files are created in the directory named `ostapp` that you can see in the File Browser panel on the left. The directory name matches the name you gave the application.

**Note**  
If you don't see the `ostapp` directory in the File Browser panel, right-click the `Home` directory and select `Reload` from the menu that appears. This updates the browser panel to show the latest files and directories.

## Restart the Web Server

Once you’ve created your application, you need to connect it to a web server. We've already configured your web server so it knows where to find it, all you need to do is restart the web server:

```
cold1:~/railsapps$ ~/httpd/bin/apachectl restart
cold1:~/railsapps$
```

When your web server is restarted, your application will begin running and it's ready to use.

## Viewing Your Rails Application

Open a new web browser window or tab and go to `http://login-id.oreillystudent.com/ostapp/`. Replace the `login-id` part of the address with the name you used to log in to the terminal.

When you open the browser, it will look something like this:
What Just Happened?

When you type `rails new ostapp`, you're telling Rails that you want to create a new application named `ostapp`. Rails then creates an `ostapp` folder and fills it with a whole heap of files and folders that make up the new application.

When you point your browser at `http://login-id.oreillystudent.com/ostapp/`, your browser contacts the Rails server. The Rails server has already been configured to run the application you created in the `ostapp` directory. So why did the server return this particular welcome page?

This is the default home page for your application. Every Rails application has to have a home page, though you'd never want an application to go live with the default version of that page, so you'll need to edit it.

Editing Your First Page

The home page of your application is static—it always looks the same. All of the static content of your application—such as the images, stylesheets, and the home page—is in the `public` subdirectory of `ostapp`. Your home page is stored in the `index.html` file.
To edit `index.html`, open the `/railsapps/ostapp/public` directory in the File Browser panel and then double-click the `index.html` file. The `index.html` file opens in the Code Editor panel at the bottom of the screen.

The `index.html` file is just an ordinary HTML web page. When a user makes a request for a URL like `http://login-id.oreillystudent.com/ostapp/`, the Rails server sends back the contents of the `railsapps/ostapp/public/index.html` file.

**OBSERVE:**

```html
<!DOCTYPE html>
<html>
<head>
  <title>Ruby on Rails: Welcome aboard</title>
  <style type="text/css" media="screen"> 
    body {
      margin: 0;
      margin-bottom: 25px;
      padding: 0;
      background-color: #f0f0f0;
    }
  </style>
</head>
```

Scroll down to the `<body>` section (around line 186), and then add some content, like this:
When you finish editing the file, save it by clicking the **Save** button on the toolbar:

When you finish editing, click **Save**.

**Tip**

Although the editor toolbar has a **Preview** button that will open the current page in a web browser, it won't work for most of the pages you create in Rails. Instead, keep a separate browser window open to view your application.

Switch back to the [http://login-id.oreillystudent.com/ostapp/](http://login-id.oreillystudent.com/ostapp/) window and click on the browser's **Refresh** button. Your page is updated:
All of the files in the /public directory are static, but most web applications display dynamic content. When you go to http://twitter.com, you expect to see a list of the latest tweets. When you do a search on Google, you see a page containing a list of search results. All of the dynamic content—whether it's a list of tweets, or a set of search results—has been generated somewhere by a piece of code. In the case of Ruby on Rails, the code that generates dynamic content is written in the Ruby language.

We've already used HTML, but what does Ruby code look like?

The Ruby Language

Before we add Ruby code to our application, let's try out some Ruby code in the Unix shell. On the O'Reilly School servers, we've installed a program called Interactive Ruby, or irb. Go back to the Terminal tab or open a new one, and type irb at the cold: prompt as shown:
Interactive Ruby is a tool that runs small pieces of Ruby code. `irb` will run a piece of Ruby code as soon as you enter it. For example, you can add two numbers together like this:

```
INTERACTIVE SESSION:
irb(main):001:0> 1 + 1
=> 2
irb(main):002:0>
```

1 + 1 is a Ruby expression. An expression is a piece of code that has a value. In this case, Ruby returns the value 2—that's what \( \Rightarrow 2 \) means.

Ruby can work with text as well as numbers. For example:

```
INTERACTIVE SESSION:
irb(main):002:0> "Hello " + "World!"
=> "Hello World!"
irb(main):003:0>
```

If Ruby sees a sequence of characters surrounded in quotes, it treats the characters as a single piece of text called a string. If you add two strings together using the + symbol, Ruby creates a new string by connecting them together. Ruby even lets you multiply a string by a number:

```
INTERACTIVE SESSION:
irb(main):003:0> "Hello " * 10
=> "Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello "
irb(main):004:0>
```

Multiplying a string by 10 creates a new string containing 10 copies of the original. Ruby can process numbers and strings, but that's not all. For example, Ruby can tell you the current date and time:

```
INTERACTIVE SESSION:
irb(main):004:0> Time.now
=> Tue Apr 11 07:30:35 -0500 2012
irb(main):005:0>
```

The `Time.now` expression is a little more interesting than the others, because it will produce a different result each time you run it:
Tip To repeat a command you’ve typed before in the Unix shell, use the up arrow key. You can scroll through previous commands using the up and down arrow keys.

To leave the irb session, use the quit command:

```
=> Tue Apr 11 07:32:29 -0500 2012
irb(main):008:0> quit
```

## Adding Ruby Content to a Web Page

Now, what if you want to include the value of a piece of Ruby code inside a web page? For example, let’s say you want to build a web page that works like a timer:

```
<%= Time.now %>
```

You could build a web page that displays the current value of the `Time.now` expression.

To insert a dynamic value like `Time.now` into a web page, we’ll combine HTML code and Ruby code in the same page.
We'll begin by generating some code. Remember, in Rails, you generate the code first and then modify it. You'll generate a dynamic web page by entering a command into the Unix shell. You run the Rails command from the application's directory—`~/railsapps/ostapp`, so you need to change into the `ostapp` directory and then run the command to generate the web page:

**INTERACTIVE SESSION:**

```
cold1:~/railsapps$ cd ~/railsapps/ostapp
cold1:~/railsapps/ostapp$ rails generate controller my_timer show_current_time
  create  app/controllers/my_timer_controller.rb
  route  get "my_timer/show_current_time"
  invoke  erb
  create  app/views/my_timer
  create  app/views/my_timer/show_current_time.html.erb
  invoke  test_unit
  create  test/functional/my_timer_controller_test.rb
  invoke  helper
  create  app/helpers/my_timer_helper.rb
  invoke  test_unit
  create  test/unit/helpers/my_timer_helper_test.rb
cold1:~/railsapps/ostapp$
```

We use the `rails generate` command to add code to an existing application. Let's break the command down and see what each part of it means:

**OBSERVE:**

```
rails generate controller my_timer show_current_time
```

- `rails generate` tells Rails that we want it to create some extra code for the application.
- `controller` indicates which kind of code we want to add to the application—in this case, we're creating controller code (we'll explore controller code in more detail later in the course, but for now just be aware that it will include a dynamic web page that we can edit).
- `my_timer` is the name of the subdirectory where the code will be stored. Rails always stores dynamic web pages in the `app/views` subdirectory. The `my_timer` value indicates that we want Rails to store our new web page in the `app/views/my_timer` subdirectory.
- `show_current_time` is the name of our new web page file. Rails will add an `.html.erb` extension automatically, so the dynamic web page file will be named `app/views/my_timer/show_current_time.html.erb`.

The `rails generate` command creates a new web page file named `~/railsapps/ostapp/app/views/my_timer/show_current_time.html.erb`. In fact, the `~/railsapps/ostapp/app` directory contains most of the code you'll use. You can view the page that the `show_current_time.html.erb` file sends out by opening a browser at `http://login-id.oreillystudent.com/ostapp/my_timer/show_current_time.html`.
The file in the Rails application has the extension .html.erb, but when you view it on the web, you use an .html extension in the URL. The web page file is

~/railsapps/ostapp/app/views/my_timer/show_current_time.html.erb, but the matching URL is

http://login-id.oreillystudent.com/ostapp/my_timer/show_current_time.html. Rails will use the show_current_time.html.erb file to create a response every time a browser asks for show_current_time.html.

This new web page is a little different from the static home page you edited before; the new web page can be modified to include some Ruby code.

Edit the new web page file by double-clicking

~/railsapps/ostapp/app/views/my_timer/show_current_time.html.erb in the File Browser panel:
This file is called a template. A template is similar to an ordinary static HTML page, except it's a little shorter and can be modified to include Ruby code.

**Note**
Templates are much shorter than static pages because Rails will add things like the `<head>...` section for you automatically. Later in the course you'll see how you can control the additional HTML that Rails will add.

To add Ruby code to a template, insert it between `=<%` and `%>` markers. These markers tell Rails that the text between them is a Ruby expression. Rails will evaluate the code between the markers and send the value of the expression to the web browser. Modify the `show_current_time.html.erb` file to include the `Time.now` Ruby expression to see how this works:
MyTimer#show_current_time

The current time is: 
<%= Time.now() %>

Instead of displaying the Time.now text, Rails replaces it with the current date and time. If you refresh the web page, the date and time changes and shows the current time.

Congratulations!

You just created your first dynamic Rails application! Let's review:

- You created a brand-new application using the rails new ostapp command.
- You edited the front page of the application by changing the contents of ostapp/public/index.html.
- You created some dynamic controller code with rails generate controller my_timer show_current_time.
- You modified the page to dynamically include the date and time every time a user looks at it.

When you finish each lesson, go back to your syllabus page and complete the quizzes and projects to make sure you’re ready to move on to the next lesson. See you there!
Your First Database Application

Lesson Objectives

When you complete this lesson, you will be able to:

- create a Database Application using Rails.
- use Rails scaffolding to generate a set of web pages that will allow a user to create, read, update, and delete tasks.

Creating Applications that Store Data

Most applications need to store and retrieve information. Facebook stores your posts and lets other people read them. Google stores data about web pages that you can search through. Whatever you design your application to do, it will probably need to store and retrieve information.

Let's look at an example. Suppose you want to help people manage tasks online. A task might have these elements:

<table>
<thead>
<tr>
<th>Element</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Buy eggs</td>
</tr>
<tr>
<td>Description</td>
<td>Go to the market and buy six eggs. Try to get organic, if possible.</td>
</tr>
<tr>
<td>Duration</td>
<td>60</td>
</tr>
<tr>
<td>Due date</td>
<td>12/12/2014</td>
</tr>
<tr>
<td>Complete?</td>
<td>X</td>
</tr>
</tbody>
</table>

Although the details of each task will be different, the kinds of data stored for each task will be the same. For example, a task will always be given a name. Here is a list of elements that a user might want to record for a task:

- **Name** is a short piece of text that identifies the task.
- **Description** is a longer piece of text that provides more details about exactly what the task involves.
- **Duration** is the number of minutes the task is likely to take.
- **Due date** is the date when the task should be completed.
- **Complete** is a true/false value that indicates whether or not the task is complete.

These elements are called the *attributes* of the task. Each attribute has a *data type* that specifies which kind of data the attribute will store. Rails supports just a few data types:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>a short piece of text</td>
</tr>
<tr>
<td>text</td>
<td>a longer piece of text of up to 64,000 characters</td>
</tr>
<tr>
<td>date</td>
<td>a date</td>
</tr>
<tr>
<td>boolean</td>
<td>a true/false value</td>
</tr>
<tr>
<td>integer</td>
<td>a whole number with no fractions</td>
</tr>
<tr>
<td>float</td>
<td>a number that includes fractions</td>
</tr>
<tr>
<td>decimal</td>
<td>a base-10 fractional number</td>
</tr>
<tr>
<td>binary</td>
<td>a series of 1s and 0s—most of the time you will use a number instead</td>
</tr>
<tr>
<td>datetime</td>
<td>a date that also includes the time of day</td>
</tr>
<tr>
<td>time</td>
<td>just the time of day, without a date</td>
</tr>
<tr>
<td>timestamp</td>
<td>an unformatted number that refers to a particular date and time</td>
</tr>
<tr>
<td>references</td>
<td>a reference to some other piece of data</td>
</tr>
</tbody>
</table>

By telling Rails the data type of each attribute, you can prevent a user from entering inappropriate information, such as storing a piece of text in the duration attribute, or setting the due date to a number.
Rails will see each task as a collection of attributes, like this:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>string</td>
</tr>
<tr>
<td>Description</td>
<td>text</td>
</tr>
<tr>
<td>Duration</td>
<td>integer</td>
</tr>
<tr>
<td>Due date</td>
<td>date</td>
</tr>
<tr>
<td>Complete?</td>
<td>boolean</td>
</tr>
</tbody>
</table>

Once you've got a list of attributes with data types, you're ready to write some Rails code to manage the data.

**What Will the Code Need to Do?**

If users are going to manage tasks, your application needs to be able to:

- **Create tasks**: Users will need to create tasks; those tasks need to be stored safely.
- **Read tasks**: Users will need to be able to read the tasks they've created.
- **Update tasks**: After users create tasks, they may want to edit them; they may want to correct spelling, change the date, or mark the task as complete.
- **Delete tasks**: Users might want to delete tasks.

These four operations are known as the CRUD operations. CRUD is a charming acronym devised from: Create, Read, Update, and Delete. CRUD operations are universal. Any change you'll ever make to any set of data will involve creating, reading, updating, or deleting.

Most applications you write in Rails will need to do these four operations. Rather than writing CRUD code yourself, you can get Rails to generate the code for you.

**Generated CRUD Code is Called "Scaffolding"**

In Rails, we create code by generating it, then modifying it. You've seen that Rails can generate a single dynamic page for you, but you can also generate an entire set of web pages that will allow you to create, read, update, and delete a piece of information. The generated code that does this has a special name—it's called scaffolding.

Scaffolded code forms the basis of most Rails applications. In the same way that you'd construct scaffolding if you were building a house, you create scaffolding when you're building a Rails application. This is how we will scaffold the code for our task data:

- **Generate the scaffold**: This will create a set of web pages to create and edit the task data.
- **Prepare the storage**: Once the web pages exist to create tasks, we'll need somewhere to store the data.
- **Customize the code**: Rails will do the hard work of creating most of the code for you, but you'll often make changes to the code Rails creates.

**Step 1: Generate the Scaffolding Code**
We'll use the `rails generate` command to generate the scaffolding code. Remember—`rails generate` is the command you use whenever you want to create extra code for an application that already exists. Make sure you are in the `~/railsapps/ostapp` directory, and then run this command (make sure you type all of it!):

**INTERACTIVE SESSION:**

```
cold1:~/railsapps/ostapp$ rails generate scaffold Task name:string description:text duration:integer due_date:date complete:boolean
invoke  active_record
create   db/migrate/20120401085758_create_tasks.rb
create   app/models/task.rb
invoke   test_unit
create   test/unit/task_test.rb
create   test/fixtures/tasks.yml
route    resources :tasks
invoke   scaffold_controller
create   app/controllers/tasks_controller.rb
invoke   erb
create   app/views/tasks
create   app/views/tasks/index.html.erb
create   app/views/tasks/edit.html.erb
create   app/views/tasks/show.html.erb
create   app/views/tasks/new.html.erb
create   app/views/tasks/_form.html.erb
invoke   test_unit
create   test/functional/tasks_controller_test.rb
invoke   helper
create   app/helpers/tasks_helper.rb
invoke   test_unit
create   test/unit/helpers/tasks_helper_test.rb
invoke   stylesheets
create   public/stylesheets/scaffold.css
cold1:~/railsapps/ostapp$
```

Let's look at the command in detail:

**OBSERVE:**

```
rails generate scaffold Task name:string description:text duration:integer due_date:date complete:boolean
```

- **`rails generate`**: Always use this command when you want to create extra code for an application.
- **scaffold**: This is the kind of code you want to create. Scaffold code will create all of the pages you need to create, read, update, and delete data.
- **Task**: This is the name we'll use for the data. The name must begin with an uppercase letter, it must be a singular word—so here we're using `Task` rather than `Tasks`, and the name must not contain any spaces.

The rest of the command tells Rails exactly what the task data will look like. These are the data attributes we looked at earlier:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
</tr>
</tbody>
</table>
We tell Rails about each of these attributes using the format `attribute: data-type`.

**What Code Gets Created?**

Earlier you used the `rails generate` command to create a dynamic web page:

```
$ rails generate controller my_timer show_current_time
```

The `controller` code included a single dynamic web page at

```
~/.railsapps/ostapp/app/views/my_timer/show_current_time.html.erb
```

but when you generate scaffolding code, Rails creates several web pages. Each of the web pages performs a specific task—for example, one page allows the user to enter a new task and another displays a list of all tasks. Rails stores these web pages in a single directory in `~/.railsapps/ostapp/app/views`. Because you told Rails to generate scaffolding code for `Task` data, Rails named the directory `~/.railsapps/ostapp/app/views/tasks` (that's why we used the singular `Task` in the `generate scaffold` command):

Each of these four pages corresponds to an action that you want to perform on the task data. Each action has a URL:

<table>
<thead>
<tr>
<th>Scaffolded action</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>List/index all tasks</td>
<td><a href="http://login-id.oreillystudent.com/ostapp/tasks/">http://login-id.oreillystudent.com/ostapp/tasks/</a></td>
</tr>
<tr>
<td>Display a form to create a task</td>
<td><a href="http://login-id.oreillystudent.com/ostapp/tasks/new.html">http://login-id.oreillystudent.com/ostapp/tasks/new.html</a></td>
</tr>
<tr>
<td>Display a form to edit a task</td>
<td><a href="http://login-id.oreillystudent.com/ostapp/tasks/edit.html">http://login-id.oreillystudent.com/ostapp/tasks/edit.html</a></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

When you ran the `generate scaffold Task` command, Rails created the `/tasks` directory under the

```
~/.railsapps/ostapp/app/views
```

folder and then put the `.html.erb` template files into it. You specified the attributes each task would contain, so Rails was able to create template files that contained places for the...
name, description, and such for the task data.

Yet even though Rails has generated all of the actions to manage the Task data, the scaffolding is not ready to be used yet. You still need to create a place for Rails to store the task data.

**Step 2: Create Storage for the Task Data**

How do you prepare the storage? The good news is that the scaffolding code already includes everything you need to prepare a place to store the task data. All you need to do is run it. Rails has created a script file in the ~/railsapps/ostapp/db/migrate directory with a name that includes these elements: `timestamp_create_tasks.rb`.

Take a look at the script; you'll see it contains some Ruby code:

```ruby
class CreateTasks < ActiveRecord::Migration
  def self.up
    create_table :tasks do |t|
      t.string :name
      t.text :description
      t.integer :duration
      t.date :due_date
      t.boolean :complete
      t.timestamps
    end
  end
  def self.down
    drop_table :tasks
  end
end
```

This Ruby code was generated at the same time as the scaffolded template files, but even though Rails created this file, the code inside it hasn't been run yet.

Rails created the script in `db/migrate`. The script name begins with the date and time.

Note

Rails does not run this script automatically, as soon as it creates it. That's because database changes can sometimes take a long time to run, and if you're sharing a database with several other developers, you might want to decide exactly when database changes are made. Fortunately, you have exclusive use of the database you're working on here, so you can make changes to it whenever you like.
So far you’ve only used small Ruby expressions, but this script is much longer. You don’t need to understand the details of this Ruby code just yet—you just need to know that it will create a table to store the task data. Rails stores data in a database using tables. If you’ve never worked with a database before, think of a table as something similar to a spreadsheet. The Task table will look like this:

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
<th>duration</th>
<th>due_date</th>
<th>complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buy eggs</td>
<td>Go to the store and buy eggs</td>
<td>60</td>
<td>12/11/2014</td>
<td>false</td>
</tr>
<tr>
<td>Visit gym</td>
<td>Run 8m</td>
<td>30</td>
<td>11/12/2014</td>
<td>false</td>
</tr>
</tbody>
</table>

A table is a storage area in a database. Each task is stored in a separate row in the table. The row contains several columns—one for each of the task attributes. The `create_tasks.rb` script will create this table for you. You can run the script using the `rake` command:

```
INTERACTIVE SESSION:

cold1:~/railsapps/ostapp$ rake db:migrate
(in ~/railsapps/ostapp)
== CreateTasks: migrating ===================================
-- create_table(:tasks)
   -> 0.0046s
== CreateTasks: migrated (0.0048s) ===========================
cold1:~/railsapps/ostapp$
```

**Note** The `rake db:migrate` command runs the migration script in `/db/migrate`. `rake` is a tool that helps you manage your Rails application. It’s like a personal assistant. It doesn’t generate any code itself, but it helps you manage the application’s environment. In this case, `rake` is told to go and look for any new scripts in the `db/migrate` directory and run them. The `db/migrate` directory is used to store Ruby scripts that change the database storage. For that reason, the scripts in the `db/migrate` directory are called migrations and the `rake db:migrate` command means, "Run any new migration scripts."

Once you’ve created the table, refresh the browser pointed at `http://login-id.oreillystudent.com/ostapp/tasks/` and you’ll see this:

Once you create the table, an empty list of tasks appears on the page.

### Now Try Out the Scaffolded Pages

If you can see an empty list of tasks, your scaffolding code is up and running. Take some time to use the
pages to create and edit some tasks. Start by clicking the **New Task** link:

To add a task, click **New Task**.

Then, enter the details of a new task:
Rails creates the form to match the attributes of the data. The `name` attribute of a task is a `string`, so Rails generates a label on the page containing the text "Name" and then places it above a simple text entry field:

```
Name
Buy eggs
```

The `description` attribute is a longer `text` value, so Rails generates a large `textarea` for the user to enter the description:

```
Description
Go to the market and buy six eggs.
Get organic, if possible.
```

Rails creates the form to match the attributes of the data. The `name` attribute of a task is a `string`, so Rails generates a label on the page containing the text "Name" and then places it above a simple text entry field:

```
Name
Buy eggs
```

The `description` attribute is a longer `text` value, so Rails generates a large `textarea` for the user to enter the description:

```
Description
Go to the market and buy six eggs.
Get organic, if possible.
```

Rails creates the form to match the attributes of the data. The `name` attribute of a task is a `string`, so Rails generates a label on the page containing the text "Name" and then places it above a simple text entry field:

```
Name
Buy eggs
```

The `description` attribute is a longer `text` value, so Rails generates a large `textarea` for the user to enter the description:

```
Description
Go to the market and buy six eggs.
Get organic, if possible.
```
Rails creates another text field for the user to enter a duration value:

```
Duration
60
```

The due_date attribute is a date, so Rails has to work a little harder. First, it converts the due_date attribute name into a more readable "Due date" label, and then it creates three separate drop-down select boxes for the user to enter the date value:

```
Due date
2013  February  9
```

Rails generates a simple checkbox for the boolean complete attribute:

```
Complete
```

Because you already told Rails the data types of the task attributes, Rails uses that information to derive the kinds of fields it needs in the application.

When you finish entering the data in the form, click Create Task:
The details of the task are sent to the Rails application, where they are stored in the database. Then, the application displays a page containing the details of the new task:

To save the task, click Create Task.

When you click Create Task, the application saves the task to the database, and then displays a page with the task details.

To return to the list, click Back.

That means you used the application to create and read task data. To return to the main tasks page, click the Back link. You will see your new task listed. If you like, you can click New Task and add another task. Each time you add a task, it gets added to the list of tasks on the front page:
If you want to change any of the tasks, click the **Edit** link and you'll see the task in a form:

### Listing tasks

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Duration</th>
<th>Due date</th>
<th>Complete</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buy eggs</td>
<td>Go to the market and buy six eggs. Get organic, if possible.</td>
<td>60</td>
<td>2013-02-09</td>
<td>false</td>
<td>Show, Edit, Destroy</td>
</tr>
<tr>
<td>Morning run</td>
<td>Run 8m. Aim for 7:20 splits.</td>
<td>60</td>
<td>2013-06-09</td>
<td>false</td>
<td>Show, Edit, Destroy</td>
</tr>
</tbody>
</table>

New Task

Create TinyURL
Note

This form might look familiar. Rails uses the same form to create and edit tasks. The code that displays the form is in app/views/tasks/_form.html.erb and the new.html.erb and create.html.erb templates use it. You'll see how one template can include the contents of another template later in the course.

If you change the details of the task and then click Update Task, you'll be shown the details of the amended
task.

So now you can create, read, and update tasks, but what about deleting tasks? Next to each task on the main list, there’s a link labeled **Destroy**. Click on this link and the application will ask you if you’re sure you want to delete the task:

Click **OK** to remove the task from the list:

The scaffolded pages allow you to create, read, update, and delete task data. This is how all of those scaffolded pages link together:
Note  The Delete function doesn’t have a page of its own. When you click the Delete link on the list/index page, it removes the task and then returns you to the list.

You can enter and maintain any number of tasks. All of the pages and all of the code you need have been generated for you. Because the data is stored inside a database, it doesn’t matter if your application crashes or if the server machine is restarted; your tasks remain on the system until someone deletes them. Your data is persistent.

Does this mean that scaffolding creates the entire application for you? Not quite. Scaffolding creates general code that allows anyone to change data in a general way. It doesn’t have any security. It always displays all of the data and, of course, it has a standardized look and feel.

Scaffolding gives you a head start on Rails development. It creates the bulk of the code that you'll need and it will get your application up and running really quickly. However, scaffolded code is really just a starting point in Rails development. To go further, you need to understand more about how scaffolding works, so that you can customize scaffolded code and make the application do exactly what you want.

In the next lesson we'll begin to modify scaffolded code. See you there!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.
Lesson Objectives
When you complete this lesson, you will be able to:

- use options in Rails to modify your applications.

Modifying the Scaffolding
We've used Rails scaffolding to generate a set of web pages that will allow a user to create, read, update, and delete task data using our application. Scaffolded code is a great way to get started on an application. It provides a lot of the basic code to do the standard operations a user will need, but scaffolded code is only a start. You'll almost never send an application live that just uses the default scaffolded code. You might want to change the links within the generated pages, or format the data differently.

For example, if a user enters a really long description, they might want to break it up into separate paragraphs to make it more readable:
Editing task

Name
Morning run

Description
Run 8 miles.

Aim for 7:20 min splits. No more than 8 min.

OK, maybe just 1 mile again--but no calling Mom to pick me up at the side of the road this time.

Duration
70

Due date
2018 June 9

Complete

Update Task

The scaffolded pages hide the line breaks when you view the task:
You need to modify the scaffolded code to preserve the line breaks. So, if you want to modify how the task description is displayed, what code will you need to change? Think back to when you created a dynamic web page that displayed the current date and time:

```
<%= Time.now %>
```

This page was generated dynamically using a template file called `show_current_time.html.erb`. Every time a user requests the web page, the template is used to generate a response. Scaffolding works in exactly the same way, but instead of using just one template file, it uses several. We saw earlier how each of those pages linked together.
Each of the pages has a matching template file in the `app/views/tasks` directory. To change the appearance of a page, you need to edit its template file.

**Edit show.html.erb**

To modify the page that displays an individual task, you'll need to edit the `show.html.erb` file. Open that file in the editor now by double-clicking on the file in the File Browser panel:

Rails created the pages in `app/views/tasks`.

- **Edit a task.**
- **List (index) the tasks.**
- **Create a task.**
- **Show a single task.**

The file appears in the Code Editor at the bottom of this screen:
The `show.html.erb` file includes Ruby expressions for each of the attributes of the task. Each expression uses the period operator to read the value of each of the attributes.

We want to modify the way the description appears to make sure we can see line breaks, so we'll modify the `@task.description` expression.

**Preserving Line Breaks**

So you've found the template file you need to change, and you've identified the line of code that's inserting the description. Now you need to find out why the line breaks are disappearing.

It has to do with the way HTML works. View the task in a browser and then look at the HTML source (to do that, you can usually right-click the web page and choose something like, "View Source"). The line breaks in the original description text are included in the source of the web page:
However, when a browser reads that HTML source, all of the line breaks are skipped and the entire description displays as a single line of text.

To preserve the line breaks, you use a helper function.

## Helper Functions

Rails helper functions format the data in web pages. A function is a piece of code that reads one or more pieces of data and creates some new value. For example, you might have a function that reads two numbers and calculates their sum. The value that the function creates is called a return value. Rails helper functions take the data they are given and convert it into formatted text that displays well on a web page.

For example, to preserve the line breaks in a piece of text, you can edit the `show.html.erb` file and use the `simple_format` helper function:

```
<%= simple_format(@task.description) %>
```

However, when a browser reads that HTML source, all of the line breaks are skipped and the entire description displays as a single line of text.

To preserve the line breaks, you use a helper function.

## Helper Functions

Rails helper functions format the data in web pages. A function is a piece of code that reads one or more pieces of data and creates some new value. For example, you might have a function that reads two numbers and calculates their sum. The value that the function creates is called a return value. Rails helper functions take the data they are given and convert it into formatted text that displays well on a web page.

For example, to preserve the line breaks in a piece of text, you can edit the `show.html.erb` file and use the `simple_format` helper function:

```
<%= simple_format(@task.description) %>
```
When Rails creates a web page, it doesn't simply insert the value of the task's description. Instead, it passes the description to the `simple_format` helper function, which creates a formatted version of the description to insert into the web page.

To see what the code change does, open (or switch back to) a browser window at `http://login-id.oreillystudent.com/ostapp/tasks/` and then click the **Show** link next to a task that contains line breaks in the descriptions:

The line breaks are now preserved and appear when you view the task. To see what happened in the HTML, view the source of the web page:

The `simple_format` helper was given the `@task.description` containing line breaks, and it created a new piece of text that wrapped each of the lines in the description with `<p>`...`</p>` tags. This preserves the line breaks when the description is viewed in a browser.
The `simple_format` helper demonstrates the basic behavior of all Ruby functions. When you *call* (execute) the function, you pass an *argument* to it in the form of the task description. The helper function processes this argument and creates a *return value*, which in this case is a newly formatted version of the description. You'll see throughout this course that most Ruby code is written as functions.

While we're talking about functions, let's talk about formatting too. In the code above, the argument to the `simple_format` function is wrapped in parentheses. That's a good way to separate a function name from its argument. Still, some Ruby programmers opt to call functions in *poetry style*—which means they leave out the parentheses:

```ruby
<%= simple_format @task.description %>
```

Some coders prefer this format because it makes simple function calls easier to read. Other programmers prefer parentheses because it makes the code more explicit.

Which do you prefer? Can you think of any problems with poetry-style coding? Try out poetry-style coding as we continue through the lab and you can decide for yourself whether you like it.

In either case, you should be *consistent* in your coding and stick with one or the other.

**What Just Happened?**

- We identified which template file to change.
- We used the `simple_format` helper function to preserve line breaks.
- We looked at poetry-style code formatting.
Functions That Take Multiple Arguments

What if you want to display how much time is left until a task is due? The show.html.erb template file currently displays the @task.due_date; the amount of time left for a task is just the difference between the due date and the current date. You know how to get the value of the current time—use the Ruby expression `Time.now`. How about the current date? You can do that in Ruby with `Date.today`. To display the difference between the due date for the task and today, edit the show.html.erb file and subtract one date from the other:

```ruby
<%= @task.due_date - Date.today %>
```

Save the show.html.erb file and open or refresh the web browser at `http://login-id.oreillystudent.com/ostapp/tasks`. The Ruby expression `@task.due_date - Date.today` calculates the number of days between the due date and today.
If there is a large number of days remaining, wouldn't it be better to display some more descriptive amount of time, for example, measuring the time in months or years?

It would take some pretty complex programming to switch between days, months, and years. Fortunately, Ruby has a `date helper function` that is designed specifically to translate time differences into English phrases:
The `distance_of_time_in_words` helper converts the difference between dates/times into a text description. That means the `distance_of_time_in_words` must be given two arguments—in this case, the current date and the due date. To pass multiple arguments to a function in Ruby, you need to separate them with commas:

```
distance_of_time_in_words(Date.today, @task.due_date)
```

The number of arguments that you provide and the order they come in, depends on the function you’re calling. Add the date helper function to the `show.html.erb` file like this:

```html
<p id="notice"><%= notice %></p>
<p>
<b>Name:</b> <%= @task.name %></p>
<p>
<b>Description:</b> <%= simple_format(@task.description) %></p>
<p>
<b>Duration:</b> <%= @task.duration %></p>
<p>
<b>Due date:</b> <%= @task.due_date %></p>
<p>
<b>Time remaining:</b> <%= @task.due_date - Date.today %> days
<%= distance_of_time_in_words(Date.today, @task.due_date) %></p>
<p>
<b>Complete:</b> <%= @task.complete %></p>
<%= link_to 'Edit', edit_task_path(@task) %> |
<%= link_to 'Back', tasks_path %>
```

Save your changes and then refresh the page that’s displaying the details of the task.
Ruby makes a distinction between date values and what it calls date-time values. A date-time value is a combination of a date and a time. When you insert the `Time.now` value into a web page, that's a date-time value:

```ruby
MyTimer#show_current_time
```

The current time is: Mon Feb 11 13:27:07 -0600 2013

This is a date-time value.

The expression `Date.today` is a simple date value—it doesn't include a time of day. Ruby won't let you subtract a date from a date-time, or vice-versa. That's why we calculated the time remaining on a task manually, using the `Date.today` value:

```ruby
@task.due_date - Date.today
```

This is a date value. This is also a date value.

If we'd tried to subtract `Time.now` from the `due_date`, Ruby would have displayed an error:
But the `distance_of_time_in_words` helper can calculate the difference between any two date-time values, so you can measure the distance to `@task.due_date` from either `Time.now` or `Date.today`:

```ruby
@task.due_date - Time.now
```

Both of these functions will work.

You can use dates...

...or date-times.

Alternative Helper: `time_ago_in_words`

There's a function similar to `distance_of_time_in_words` named `time_ago_in_words`—it also describes a period of time in words, but takes only a single date or time value and it measures the distance between the given value and now:

```ruby
distance_of_time_in_words(Time.now, @task.due_date)
distance_of_time_in_words(Date.today, @task.due_date)
```

These two function calls are equivalent.

This helper measures time from now.

What Just Happened?

- We calculated the number of days between dates using subtraction.
- We used a helper function to describe time difference in words.
- We passed multiple arguments to the helper function separated by commas.
- We discussed the `time_ago_in_words` helper to calculate time from now.

Functions with Options

The task duration is measured in minutes, but if you want to display it in hours, we can do that—in the `show.html.erb` file, you just divide the value by 60.0:
Save the `show.html.erb` file and then refresh the browser window that's displaying a task:

**Note**

We divide the duration by **60.0** rather than **60** to make sure Ruby will include fractions in the result. If you divide two whole numbers in Ruby, the result will always be rounded to a whole number.
Still, there’s a problem with simply dividing the duration by 60.0—the result contains way too many decimal places. You’ll need to format the number in some way. Rails has a helper function to deal with this: the `number_with_precision` function. You might want to specify:

- the number of decimal places.
- the overall number of significant figures.
- the separator characters between the whole number and fractional parts.

All of these different options will need to be specified as function arguments, but how do you pass some arguments to a function and not others?

The designer of Ruby got around this problem with function options. Options are similar to normal arguments except that:

- options are passed to the function after all other arguments.
- options are passed with names.

Here’s how you would format a number to include 5 decimal places and use the comma (",") character to separate the whole numbers from the fractions:

```
number_with_precision(@task.duration / 60.0, :precision => 5, :separator => ',')
```

Each of the options is listed with a name and a value separated by the characters `=>`. This is called the hash rocket by Ruby programmers. Hash because a list of named values in this format is called a hash map, and rocket because `=>` looks a little like a sideways rocket.

Option names begin with a colon (:) character. A colon followed by a series of characters is called a symbol in Ruby. A symbol is kind of like a string. Symbols are used to store names in Ruby, so options are listed with symbols.

Here are some more examples of the `number_with_precision` helper:

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>number_with_precision(1.16666667, :precision =&gt; 5, :separator =&gt; ',')</td>
<td>1,16667</td>
</tr>
<tr>
<td>number_with_precision(1.16666667, :precision =&gt; 3)</td>
<td>1.167</td>
</tr>
<tr>
<td>number_with_precision(1.16666667, :significant=&gt;true, :precision =&gt; 3)</td>
<td>1.17</td>
</tr>
<tr>
<td>number_with_precision(4567890.0, :delimiter=&gt;' ', :precision=&gt;6)</td>
<td>4 567 890.000000</td>
</tr>
</tbody>
</table>

So if you want to display the duration in hours to 2 decimal places, modify the `show.html.erb` file as shown:
The call to `number_with_precision` takes the value of the duration / 60.0, as well as a single `:precision` option. Save `show.html.erb` and then refresh the task page, you'll see the duration formatted to two decimal places.

What Just Happened?

In this lesson, we learned:

- Some functions accept options.
- An option is an argument given by name.
- Options appear after all the other arguments
- Option names are given as `:symbols` — which are similar to "strings"
- Option names are separated from values using `=>` — the hash rocket.

Excellent work so far. Practice what you've learned so far as you do your homework. See you in the next lesson!

*Copyright © 1998-2014 O'Reilly Media, Inc.*

*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See [http://creativecommons.org/licenses/by-sa/3.0/legalcode](http://creativecommons.org/licenses/by-sa/3.0/legalcode) for more information.*
Adding Field Validation

Lesson Objectives

When you complete this lesson, you will be able to:

- apply rules in your application.
- make your Rails application check for (and prevent) empty data for a particular field.
- make your app check that entries are of a particular type (for example, numeric).

Sometimes You Need Rules

The existing version of the task application lets users create, update, and delete tasks, but it doesn’t apply any rules to the task data. For example, suppose a user clicks the New Task link on the front page and sees a new form like this (if you're trying this, enter a value for the Duration as shown; otherwise you’ll get an error message):
Nothing prevents them from clicking **Create Task** and saving the data, even though most of the fields are blank:

**The fields are blank by default.**
When you create an application that allows users to create and update data, you usually want to apply rules. You may want to make sure the user provides a description, you probably don't want two different tasks to have the same name, or you might want the duration value to be within the range 1-240, consistently.

In order to accomplish tasks like these, you want the system to validate the data the user enters. So, how do you specify validation rules, and where would you write them in a Rails application?

Making Fields Mandatory

To begin the process of validation, let's make some fields mandatory, so that users can't save a record if any of those fields are empty. All of the code we've modified so far has changed the way the application looks to the user. Code that controls how the application looks is called view code. That's why all of the page templates that generate the Task pages are stored in the app/views folder:
Rails adheres to several key design principles; one of them is Convention over Configuration. Instead of configuring each application to work in a different way, Rails apps all use the same conventions. So view code is always stored in the /app/views folder, static content is in the /public folder, and so on.

**PRINCIPLE:**
Convention over Configuration

Rails applications have strong conventions about where code should be stored, so Rails programmers can find the code they need fast. Rails has configuration information built into it, like what the name of a particular type of file is going to be (e.g., "Timestamp_create_tasks.rb"), so programmers don’t have to tell it that stuff. For example, you don’t have to create a file or otherwise tell Rails that the view files are in the /app/views folder, because Rails already knows!

But what about validation code? Where does that live?

Validation code checks the data before it’s stored in the database. The code that controls all of the data in the application is called model code. The model is the business model of your application—it indicates what data your application has to manage and how it will behave. So if you were writing Twitter, the model would be the collection of tweets. If you were writing an online game, the model would be the coordinates of the buildings and the locations of the players.

Rails stores all of the model code in your application in the /app/models/ folder:

When you generated the scaffolding for the task pages, Rails created a file called /app/models/task.rb. Open that now by double-clicking it in the File Browser:
This piece of Ruby code is called whenever your application creates or reads or writes a piece of task data. As you can see, there’s not much to it just yet. What little code there is just means “Task data is stored in the database”. There’s no other code in the task.rb file yet because Rails hasn’t been told to do anything else with task data. The task.rb file doesn’t even say anything about the attributes stored against each task. Rails can work out what they are by looking into the database.

If you want to validate the task data, you need to do that here. Add this line to the code:

```
class Task < ActiveRecord::Base
  validates(:description, :presence => true)
end
```

Make sure you type the code exactly as you see it here. This code is a function (like the functions we used in the last lab), but this function is run every time the Rails application accesses task data. The first parameter is the name of an attribute of the task, in this case, the symbol :description, because we want to validate the description field. A symbol is much like a string; it always begins with a colon (:) character. Symbols are often used for the names of things, like attributes in Ruby. The second parameter is an option called :presence. If Rails sees the :presence option, it checks for the presence of a value for the attribute. Setting the :presence option to true means that the :description attribute must be present before the task can be saved to the database.

Save the task.rb file, open a browser at http://<login-id>.oreillystudent.com/ostapp/tasks, and click New Task. You’ll see the New Task form just like before:
On the New Task page, leave the Description field blank (enter a value for Duration) and click Create Task. You'll see this:
Let's Review

The app/models/task.rb stores the custom model code for Task data. Previously, there wasn’t much in that file because there were no rules associated with task data, but we added this line of code:
The `validates` code here means "Check that the task description has a value." We called the `validates` function with two arguments to check that the `description` value is present before a task is saved.

**Note** The `validates` function is often called in poetry format, without parentheses.

`validates` is a *data validator*. The validator checks the data before it’s saved to the database. If the validator says there’s a problem, Rails sends the page back to the user:

When the user submits the New Task form, Rails calls the code in the `task.rb` script before the data is saved to the database. If the validation code passes, the data is stored in the database, but if any of the validators in `task.rb` fail, Rails sends the form back to the user with the errors highlighted.

One of the benefits of validating data in the model code is that it will also run when the user updates the data. So if you edit one of your existing tasks and try to blank out the Description, this happens:
Rails separates the code that generates web pages from the code that handles the data. This is one of the key principles in the design of Rails applications. It allows you to find the piece of code you need to amend without much hassle, and you get to reuse your code. If the validation code had been stored inside a page template, you would have had to create separate validation in both the New Task page and the Editing Task page. Instead, because the validation is done in the model code, and the same model code is shared by any piece of code that writes data to the database, you can maintain the code in just one place.
**PRINCIPLE: DRY**

Don’t Repeat Yourself. Rails was designed so that code almost never needs to be repeated. By separating the validation code from the page code, the New task and Editing task pages can both use the same validation without having separate copies of the code.

---

**Making More Than One Field Mandatory**

You can add as many validators to the model code as you like; to make the name and duration mandatory, edit `task.rb` like this:

```ruby
app/models/task.rb

class Task < ActiveRecord::Base
  validates :description, :presence => true
  validates :name, :presence => true
  validates :duration, :presence => true
end
```

Save the task.rb file, open or refresh the browser, and try adding a new task without a name, description, and duration.
Rails allows you to combine several validations together into a single line of code, like this:
class Task < ActiveRecord::Base
  validates :description, :presence => true
  validates :name, :presence => true
  validates :duration, :presence => true
  validates :name, :description, :duration, :presence => true
end

Note
All validations are applied when the data is saved, so each of the messages will appear in the web page. Ordering only affects the order of the error messages that will appear. We changed the order of the attribute names here so the validation messages will appear in the same order that the fields appear on the page.

Tip
Any time you want to apply the same validation to several attributes, just list all of the attribute names in the validates call.

Unique Task Names
You've seen how to make fields mandatory, but there are many other checks you might want to make. For example, you might want each task to have a unique name.

Create a task with the exact same name as one of your existing tasks; for example, "Buy eggs" in the example shown here:
When you return to Listing tasks, you see two tasks with the same name.
It can be complex to check for duplicate values. Every time you save a new task to the database, you'd need to read through the whole database to see if there's already a task with that name. You'd need to do the same thing each time someone edits a task. Fortunately, Rails helps you to check for uniqueness—just use the `uniqueness` rule:

```
CODE TO TYPE: app/models/task.rb

class Task < ActiveRecord::Base
  validates :name, :description, :duration, :presence => true
  validates :name, :uniqueness => true
end
```

When Rails sees this in the model code it will check for uniqueness for you. If it finds the name has already been used, it will send the form back with an error.

- Save the task.rb file, open or refresh the browser, edit the second "Buy eggs" task, and click **Update task**.
Checking Numbers

Rails is remarkably forgiving. Edit the Morning run task and use the number "0" as the letter "O" in the Duration field as shown:
What should Rails do? The default behavior is to ignore the non-numeric characters. That means that if you enter 7O, Rails treats it as 7.

By default, Rails ignores the "O" because it's not a number.
Rails is designed to be forgiving so that the applications are easier to use on the internet. That's great, but what if you want to be more strict? Sometimes a numeric value is really important, and if the user enters the letter O instead of the number 0, you don't want to just ignore it; you want to actively reject it. That's when you'll use `validates_numericality_of`. It checks to see if a field contains a number; if it doesn't it won't let the user save the record. Modify `/app/models/task.rb` as shown:

```ruby
class Task < ActiveRecord::Base
  validates :name, :description, :duration, :presence => true
  validates :name, :uniqueness => true
  validates_numericality_of :duration
end
```

Save the task.rb file, open or refresh the browser. Try changing the Duration for Morning run to 7O (with the letter "O") again.

Rails ignored the "O" and saved the value as 7 minutes (.12 hours).
Fix the Duration by changing it back to **70**, and click **Update Task**. Now the Duration is back to 1.17 hours.

**WARNING** validates_numericality_of checks that an attribute contains a valid number, but it also means that the attribute **cannot** be blank. Keep in mind that every time you say that an attribute is numeric, you are implicitly also saying that it’s required.
Once you know a field contains a number, there may be some other checks you'd like to make as well. For example, the duration of a task is *supposed* to be a number between 1 and 240, but there's nothing in the system that prevents the user from typing something else. In the **Morning run** task, change the Duration to *-120* as shown:

Click **Update Task**. Rails knows that duration is mandatory and numeric, but we haven't told it anything about the kinds of values it can take. In the same way that we could pass a rule to the `validates` method, we can pass a rule to `validates_numericality_of`:
This code change ensures that \texttt{duration} is a number \textit{and} it's $\geq 0$. Save your code and see what happens if you try to enter \texttt{-120} again:
There are seven rules available in the version of Rails you're using, and more will probably be added. For now, you can use these rules:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Means...</th>
</tr>
</thead>
<tbody>
<tr>
<td>:greater_than =&gt; x</td>
<td>Must be &gt; x</td>
</tr>
</tbody>
</table>

Change the Duration back to 70 and save the task.
What if you need to apply more than one rule at a time? Duration should be in the range `1..240`, so you could write it like this:

```ruby
class Task < ActiveRecord::Base
  validates :name, :description, :duration, :presence => true
  validates :name, :uniqueness => true
  validates_numericality_of :duration, :greater_than_or_equal_to => 1
  validates_numericality_of :duration, :less_than_or_equal_to => 240
end
```

But, like with our presence validation, there's a shorter format you can use. Modify `task.rb` as shown:

```ruby
class Task < ActiveRecord::Base
  validates :name, :description, :duration, :presence => true
  validates :name, :uniqueness => true
  validates_numericality_of :duration, { :greater_than_or_equal_to => 1, :less_than_or_equal_to => 240 }
end
```

Save it, open or refresh the browser, and test the new rules.

**What Just Happened?**

- You learned that by default, Rails doesn't complain if you enter a non-number into a numeric field.
- You learned that `validates_numericality_of` checks to see whether an attribute is numeric.
- You can apply additional rules by adding options to the function.

You're on your way. Let's keep going! After you work through the homework, I'll meet you in the next lesson...

*Copyright © 1998-2014 O'Reilly Media, Inc.*

*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See [http://creativecommons.org/licenses/by-sa/3.0/legalcode](http://creativecommons.org/licenses/by-sa/3.0/legalcode) for more information.*
Lesson Objectives
When you complete this lesson, you will be able to:

- create and use built-in and custom validation functions.
- run Ruby code on the Rails console.
- assign values to variables and use boolean expressions.
- use Ruby logic inside a web page.

Making Your App Smarter with Ruby
Rails can cut down on your development time by generating a lot of code for you, but you'll likely want to change the way the generated application looks and behaves. Custom code can be added to every part of a Rails application—from the code that talks to the database, to the code that sends web pages back to the user. We've already looked at ways of using Ruby to call built-in validations. Now let's see how we can write our own validations using Ruby.

Writing Custom Validation
Rails has a number of built-in validators that can check the values that users enter on a web page. We've seen that you can check to see that a value is present, or unique, or a number in a certain range. So how do you perform validation that isn't covered by any of the standard validators? You'll create your own custom validation in Ruby code.

Let's look at an example. Create this task:
New task

Name
Buy candy

Description
Buy extra candy.
See if they have any of those jelly worms.

Duration
45

Due date
2010 October 31

Complete

Create Task
It doesn't make sense to create a brand new task that's due some time in the past, so let's write a validator to prevent that from happening.

All of the existing validation is done in the task.rb file. This is the model code that controls how data is written to and read from the database, so this is where we'll add our custom validation code. Modify app/models/task.rb as shown:

```ruby
class Task < ActiveRecord::Base
  validates :name, :description, :duration, :presence => true
  validates :name, :uniqueness => true
  validates_numericality_of :duration, {:greater_than_or_equal_to => 1, :less_than_or_equal_to => 240}
  validate :due_in_future
  def due_in_future
    if due_date < Date.today then
      errors.add(:due_date, "cannot be in the past")
    end
  end
end
```

Save task.rb, then open or refresh your separate browser window at http://login-id.oreillystudent.com/ostapp/tasks and try to create a task with a due date in the past:
The custom validation code you just wrote prevents the new task from being saved. Now change the due date to November 6, 2014 and create the task to see how that code works.
validate calls custom code.

due_in_future is the name of a custom validation function.

validate :due_in_future

Begins with a colon because this is a symbol.

The first line we entered specified a custom validation function. A function is just a piece of code that you can run by name. In this case, we gave validate the name of a function called due_in_future. Name-values are given as symbols, so :due_in_future begins with a colon (:). Calling validate in the task.rb file like this tells Rails that the due_in_future function needs to be called whenever a Task object is going to be saved or updated in the database.

WARNING

Be careful not to confuse "validate" with "validates". "validates" is used to call built-in validations, but "validate" is used for your own custom validations.

Your custom due_in_future function will be run alongside all of the other validators that you've created. The due_in_future function checks to see that the due date is set in the future; otherwise, it creates an error message.

Validations Fail if They Create a Message

So how does Rails know if a Task failed validation? It looks to see if any error messages were created. There might be errors from built-in validators or from due_in_future; either way, if there's even one error message, the user is sent back to the form to correct the mistake.

Let's look a little closer at the due_in_future function. The function is defined between the def and end keywords. These two words mark the beginning and end of the function. The name of the function —due_in_future—follows the def keyword; after that you have the code of the function itself.
The function performs a test—it compares the task’s due date with today’s date. If the due date is in the past, the function creates a new error message that displays next to the due_date field on the page. By creating this message, the validation function will prevent the Task being stored in the database and Rails will return the page of errors to the user.

A Little More Ruby

The due_in_future function contains a little more Ruby code than we've seen before, and it's worth spending some time looking at how Ruby uses functions and tests.

When we looked at Ruby expressions earlier, we ran a program called Interactive Ruby (irb). Interactive Ruby is a great way to try small pieces of pure Ruby code. But there’s a problem with using the basic version of irb when you want to run Rails code: some Ruby expressions rely on extra libraries that are loaded specifically by Rails. A library is a set of pre-written code that allows you to execute tasks like manage dates or talk to a database. For example, start irb and then try to print out today’s date:

```
INTERACTIVE SESSION:

irb(main):001:0> Date.today
NameError: uninitialized constant Date
from (irb):1
from :0
irb(main):002:0> quit
cold1:~$
```

Your Rails application can use dates, but the irb program doesn't understand dates. You'll need to address that if you ever want to run code from your application. You can't use the irb tool, instead, you'll need an augmented version of irb called the Rails console. The Rails console is a copy of irb that can see all of the code that's available to your application.

To start the console, open a shell, change to the ~/.railsapps/ostapp directory, and type in rails console:

```
INTERACTIVE SESSION:

cold1:~$ cd railsapps/ostapp
cold1:~/railsapps/ostapp$ rails console
Loading development environment (Rails 3.0.3)
```

You need to be in the same directory as your application so that the Rails console can load the code and libraries from your application. Reading all of that code may take a while, so it might take a few seconds before the Rails console is up and running.
Once you’re in the Rails console, try printing today’s date again:

```
INTERACTIVE SESSION:

irb(main):001:0> Date.today
=> Tue, 12 Feb 2013
irb(main):002:0>
```

This time, Ruby understands the expression `Date.today`, because it has access to the relevant date code inside Rails. The `Date` code is in a Rails library and the Rails console loaded that library when you started the console.

Now let’s try to create a new date from a string:

```
INTERACTIVE SESSION:

irb(main):002:0> Date.parse('2015-10-31')
=> Sat, 31 Oct 2015
irb(main):003:0>
```

By calling `Date.parse` and passing it a string in the format `year-month-day`, you can create a new `Date` object. When you fill in a date field in a Rails web page and then submit it to the server, Rails converts the text you entered into a `Date` value.

Like any piece of data, you can store the `Date` in a variable, like this:

```
INTERACTIVE SESSION:

irb(main):003:0> due_date = Date.parse('2015-10-31')
=> Sat, 31 Oct 2015
irb(main):004:0>
```

The `due_date` variable now contains the newly created date. We refer to storing data this way as assigning it to a variable. Once the value is assigned to the variable, you can retrieve it by evaluating the variable:

```
INTERACTIVE SESSION:

irb(main):004:0> due_date
=> Sat, 31 Oct 2015
irb(main):005:0>
```

The value of `due_date` is displayed in the console right after the `=>` symbol (which means "has the value").

Take a look at the current date:

```
INTERACTIVE SESSION:

irb(main):005:0> Date.today
=> Tue, 12 Feb 2013
irb(main):006:0>
```

The value you get here depends on the date you run the code. Today’s date and the new date you created both have the same format, because they are both the same kind of data; they have the same type.

You can compare `Dates` against each other. For example, if you want to see whether the value in the `due_date` variable is older than today’s date, you can do this:
When you’re comparing dates, the < operator means is before. So our expression means due_date is before today. In this case, the expression has the value false.

Expressions that have the values true or false are called boolean expressions. Boolean expressions are useful because they help Rails make decisions. For example, if you want to display a message on the console, you call the.puts function:

```
INTERACTIVE SESSION:

irb(main):007:0> puts("I am a message on the console")
I am a message on the console
=> nil
irb(main):008:0>
```

So, what if you want to display a message on the console only if some condition is true? You could do it like this:

```
INTERACTIVE SESSION:

irb(main):008:0> if due_date > Date.today then
  irb(main):009:1* puts("Due date is in the future")
irb(main):010:1> end
Due date is in the future
=> nil
irb(main):011:0>
```

Unlike other pieces of code you've typed so far, this code takes several lines and Ruby won't run the code until you've typed in all of it. This is an if statement. An if statement checks the value of some expression; if that expression has the value true, Ruby runs the code between the line beginning if ... and the line end.

The if statement tells the computer to run something if a condition is true, but what if you want the program to do something else if the condition is false? The else keyword allows you to specify an alternative, like this:

```
INTERACTIVE SESSION:

irb(main):011:0> if due_date < Date.today then
  irb(main):012:1* puts("Due date is in the past")
irb(main):013:1> else
irb(main):014:1* puts("Due date is not in the past")
irb(main):015:1> puts("That's good")
irb(main):016:1> end
Due date is not in the past
That's good
=> nil
irb(main):017:0>
```

This code will still display a message if the due_date value is in the past, but if the due_date is not in the past, it will...
run the code between the `else` and `end` lines and display a different set of messages.

If you've used other languages like PHP, JavaScript, or Java, `if` statements will be familiar to you. Ruby does have a few variations on the `if` statement though. For example, Ruby has an `unless` statement that does the exact opposite of the `if` statement; `unless` will run code only if the expression it is given has the value `false`:

```
INTERACTIVE SESSION:
irb(main):017:0> unless due_date <= Date.today then
  puts("Date is in the future")
end
Date is in the future
=> nil
```

You may choose to write a particular piece of logic using an `if` or an `unless`—but make sure you understand how each works.

Another feature of Ruby is the `one-line-if`. If you just have a single line of code that you want to run based on a condition, you can write it on a single line, like this:

```
INTERACTIVE SESSION:
irb(main):021:0> puts("Date is in the future") if due_date > Date.today
Date is in the future
=> nil
```

This code can read a little like it was written by Yoda from the movie, `Star Wars`. The `if` condition is at the end of the line. Ruby coders normally write one-line-ifs if the expression is expected to have the value `true` most of the time. You can do the same with the `unless` statement, so you might see Ruby code that looks like this:

```
eat_donut if hungry  # Will only run when hungry is true.
go_to_work unless win_lottery  # Will only run when win_lottery is false.
```

Ruby programmers will switch between `ifs`, `elss` and one-liners, just as you switch between sentence forms in everyday speech. Ruby code is meant to be expressive, which means that not only is it meant to `run`, it's meant to be `read`. A good Ruby programmer knows what her code will do, and how to express it.

Okay, now let's look at functions. Functions are pieces of code with names. Here's how you create a function in Ruby:

```
INTERACTIVE SESSION:
irb(main):022:0> def happy_halloween
  puts("Happy Horrid Halloweeeeen!!!")
end
=> nil
```

This creates a function named `happy_halloween`. The body of the function (the code it contains) is between the `def` and `end` keywords. When you type in a function, the code it contains is not run right away—it's stored in memory. Functions take several lines of code to create, but unlike `if` and `unless` code, the console won't display `*` characters at the start of each line.

The code of the function is `indented` with a couple spaces. We did that with the code inside the `if` and `unless` statements above as well. You indent code in Ruby to make it more readable; it's up to you to decide how many spaces or `Tabs` to use.
Functions are a way of extending the Ruby language. Creating a function is like creating a new Ruby command. If you want to run the code inside the function, you just need to type its name:

<table>
<thead>
<tr>
<th>INTERACTIVE SESSION:</th>
</tr>
</thead>
<tbody>
<tr>
<td>irb(main):025:0&gt; happy_halloween</td>
</tr>
<tr>
<td>Happy Horrid Halloweeeen!!!</td>
</tr>
<tr>
<td>=&gt; nil</td>
</tr>
<tr>
<td>irb(main):026:0&gt;</td>
</tr>
</tbody>
</table>

Once you’ve created a function, you can call it as many times as you want without having to type the function’s code each time:

<table>
<thead>
<tr>
<th>INTERACTIVE SESSION:</th>
</tr>
</thead>
<tbody>
<tr>
<td>irb(main):026:0&gt; happy_halloween</td>
</tr>
<tr>
<td>Happy Horrid Halloweeeen!!!</td>
</tr>
<tr>
<td>=&gt; nil</td>
</tr>
<tr>
<td>irb(main):027:0&gt; happy_halloween</td>
</tr>
<tr>
<td>Happy Horrid Halloweeeen!!!</td>
</tr>
<tr>
<td>=&gt; nil</td>
</tr>
<tr>
<td>irb(main):028:0&gt;</td>
</tr>
</tbody>
</table>

Finally, you can leave the Rails console by typing `quit`:

<table>
<thead>
<tr>
<th>INTERACTIVE SESSION:</th>
</tr>
</thead>
<tbody>
<tr>
<td>irb(main):028:0&gt; quit</td>
</tr>
<tr>
<td>cold1:~/railsapps/ostapp$</td>
</tr>
</tbody>
</table>

The Rails console shows you the world as Rails sees it. Every time Rails generates a web page, or reads a form, or saves something to the database, it’s running pieces of Ruby code.

OK—now that we’ve looked in more depth at how the Ruby validation code works, let’s get back to the application.

**Validate Every Time?**

The custom validation will prevent users from saving a task with a due date in the past.

It also makes perfect sense to prevent people creating tasks that are due in the past.
Now suppose a user wants to edit a task. Is it possible that a user might submit a task with an old due date? For example, what if a user wants to mark an old task as complete:

You don't want users to create tasks with due dates in the past.
In that case you'd want Rails to run your custom validation when the task is being created, but not when it's being updated.

Take a look at the existing task.rb file. Where do you think Rails would allow you the option to run a validation only at creation time?
OBSERVE: app/models/task.rb

```ruby
class Task < ActiveRecord::Base

  validates :name, :description, :duration, :presence => true
  validates :name, :uniqueness => true
  validates_numericality_of :duration, {:greater_than_or_equal_to => 1, :less_than_or_equal_to => 240}
  validate :due_in_future

  def due_in_future
    if due_date < Date.today then
      errors.add(:due_date, "cannot be in the past")
    end
  end
end
```

You want the `validate` line to run only when the task is created, so it would be useful if the `validate` command had an option to specify exactly when it should be run. Sure enough, it does! The `:on` option specifies which times in the lifecycle of an object the validation should be run:

```ruby
class Task < ActiveRecord::Base

  validates :name, :description, :duration, :presence => true
  validates :name, :uniqueness => true
  validates_numericality_of :duration, {:greater_than_or_equal_to => 1, :less_than_or_equal_to => 240}
  validate :due_in_future, :on => :create

  def due_in_future
    if due_date < Date.today then
      errors.add(:due_date, "cannot be in the past")
    end
  end
end
```

In this case, we want to run the validate function `due_in_future` only if we create a new task. We don’t want the validation to run if the user is updating an existing task. So, we set the optional argument `:on` to `:create`—that tells Rails to run this function only when a new task is created. When a user updates a task, the validation won’t run at all.

To create a validation function that only runs when a record is being updated, you’d need to set `:on` to `:update`. You can also use the `:on` option with the `validates` function and built-in validators.

Once you’ve made the amendment to the `task.rb` file, you can edit existing tasks with due dates that are in the past.
Did you notice anything strange when you saved a task with a due-date in the past? The Time remaining field said the task wasn’t due for another two years:

The code that calculates the time remaining uses the `distance_of_time_in_words` helper:
The `distance_of_time_in_words` helper calculates the difference between two dates or times, and then converts the result into something readable. Here, it's giving the result "almost 2 years," but the helper doesn't care about the order of the times it's given. It doesn't tell you if the time difference goes into the past or the future.

The page would make a lot more sense if `time remaining` was hidden for dates in the past. If your Rails application is going to hide the field, it will need code to compare the current date with the due-date of the task. That means you'll need to add some sort of conditional `if` or `unless` code to the `show.html.erb` file. You already have Ruby code inside this file:

```html
<p id="notice"><%= notice =%></p>
<p>Name: <%= @task.name =%></p>
<p>Description: <%= simple_format(@task.description) =%></p>
<p>Duration: <%= number_with_precision(@task.duration / 60.0, :precision => 2) =%> hours</p>
<p>Due date: <%= @task.due_date =%></p>
<p>Time remaining: <%= distance_of_time_in_words(Date.today, @task.due_date) =%></p>
<p>Complete: <%= @task.complete =%></p>
<%= link_to 'Edit', edit_task_path(@task) ==%> | <%= link_to 'Back', tasks_path =%>
```

Each of these Ruby expressions inserts values into the web page that's sent back to the user. If we want to add logic to the web page, it's a little different from simply inserting another Ruby value. Let's look a little closer at what happens when Rails generates a web page from a template file.

```html
<%= ... %> Contains a VALUE
```

When Rails reads a template file, it replaces each of the Ruby expressions surrounded by `<%= ... %>` with the value of the Ruby expression:
Rails does this by converting the template file into an intermediate Ruby script. Each of the `<%= ... %>` expressions and each piece of static HTML is replaced by a command that will generate output to the user's web browser. It's as if Rails converts the template file into a series of `puts()` commands:

```
<% ... %> Contains LOGIC
```

In reality, Rails doesn't use the `puts` command—it generates code to send the output over the network to the user's browser—but the principle is the same.

When the intermediate script is run, the output from the script is sent to the web browser. If you want your web page to show or hide a section based on a Ruby condition, you need the intermediate script to contain Ruby logic, like this:

```
unless @task.due_date < Date.today then
  puts('<p>')</n  puts('Time remaining:')
  puts(distance_of_time_in_words(Date.today, @task.due_date))
  puts('</p>')
end
```

The `<%= ... %>` notation is great if you want to insert a value into a page, but if you want to add some page logic—for example, if you want to switch an entire section of a page on or off—you need a different kind of notation.

The `<%= ... %>` markers specify a value that will be inserted into the web page, but to add logic you need to use markers that look like this:

```
<% unless ... then %>
  Some HTML
<% end %>
```

If Rails sees a marker that begins with `<% without the equals sign (=), it knows that marker contains logic rather than a value. To see how this works, let's use `<% ... %>` to hide the Time remaining information in
Save `show.html.erb` and then look at some tasks with a browser. If the task’s due date is in the past, the time remaining is hidden; otherwise, it’s visible:
What Just Happened?

We covered a lot of ground in this lesson!

We learned:

- Sometimes you need to create your own custom validation.
- You can create a custom validation function inside your model code.
- Built-in validators are specified by calling the `validates` function.
- Custom validation can be specified by calling the `validate` function.
- The `validate` function needs to be given the name of your function as a `:symbol`.
- You can run Ruby code on the Rails console.
- The Rails console is just a copy of Interactive Ruby (irb) that can see the code inside your application.
- Dates can be created with `Date.parse("2010-10-31")`.
- You can assign values to variables with `=`.
- Some expressions have true/false values. These are called boolean expressions.
- Boolean expressions can run code conditionally with the `if` and `unless` statements.
- Ruby has various ways of writing `if` and `unless` code.
- The `puts` function displays code on the console.
• The `:on` option allows you to specify when a (built-in/custom) validator will run.
• To use Ruby logic inside a web page, use the `<% ... %>` markers.

Put your new skills to work in the homework assignment and I'll see you shortly!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.
Lesson Objectives

When you complete this lesson, you will be able to:

- read single records from the database.
- search through the database to find all the records that satisfy a set of conditions.

Asking New Questions

Let’s get started by adding these tasks to your database:

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
<th>duration</th>
<th>due date</th>
<th>complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write paper</td>
<td>Henry Knox and the Siege of Boston</td>
<td>240</td>
<td>2014-Nov-06</td>
<td>true</td>
</tr>
<tr>
<td>Buy turkey</td>
<td>Don’t forget the cranberries.</td>
<td>60</td>
<td>2014-Nov-27</td>
<td>false</td>
</tr>
<tr>
<td>Library study</td>
<td>Check that the wifi’s working</td>
<td>90</td>
<td>2014-Nov-06</td>
<td>true</td>
</tr>
<tr>
<td>Movie night</td>
<td>Chinatown</td>
<td>140</td>
<td>2014-Nov-12</td>
<td>false</td>
</tr>
<tr>
<td>Write paper</td>
<td>Henry Knox and the Siege of Boston</td>
<td>240</td>
<td>2014-Nov-06</td>
<td>true</td>
</tr>
<tr>
<td>Buy turkey</td>
<td>Don’t forget the cranberries.</td>
<td>60</td>
<td>2014-Nov-27</td>
<td>false</td>
</tr>
<tr>
<td>Library study</td>
<td>Check that the wifi’s working</td>
<td>90</td>
<td>2014-Nov-06</td>
<td>true</td>
</tr>
<tr>
<td>Movie night</td>
<td>Chinatown</td>
<td>140</td>
<td>2014-Nov-12</td>
<td>false</td>
</tr>
</tbody>
</table>

Once you have collected a decent set of task data, there are questions you may want to ask:

There are some questions that the scaffolded code will never answer. In most applications that you create, you’ll need to create custom database code. If you know how to talk to the database, you can create features that will let users get more out of your application.

Rails applications are database-centric, which means that pretty much everything they do involves reading or writing data to the database. In the database, everything is stored in tables. For example, the database stores all of the task data in a table named tasks (your data may be slightly different from our examples):
Each row of data represents a single task and each column represents an attribute of that task. In addition to the attributes that you specified when you created the task scaffolding, there are three attributes that Rails created for you automatically. They are known as the magic attributes and they are:

- **id**: a unique identifier that Rails assigns to every new task.
- **created_at**: the date and time that the task was created.
- **updated_at**: the date and time (if ever) the task was last edited.

Each of the tables that Rails creates will include these three extra attributes. They help Rails keep track of the data it's managing. For example, Rails uses the id attribute to associate a task with a URL. Let's see how that works.

**Behind the Scenes: How Rails Displays a Task**

When a user submits a request to view a single task, the URL contains the id number of the task to display. Rails extracts the id number from the URL, reads the task from the database, and converts it into a Ruby value that will then be displayed by a template:

```
http://<login-id>.userworld.com/ostapp/tasks/21
```
All of the task pages in your application work in a similar way. They read or write some value from a database and then generate a web page response. Rails needs to convert the database data into Ruby values for this to work. The Ruby values can be passed to the templates to generate the web page that will be returned to the browser that made the request.

Now, if you want to add pages that display, for example, the tasks due today, or statistics about the number of complete and incomplete tasks, you’ll need to know how to talk to the database in the same way as Rails.
The **ActiveRecord** library converts database data into Ruby data. The scaffolded code uses the ActiveRecord library every time it needs to do anything with the database. To write custom database code, you need to know how to use ActiveRecord.

### The ActiveRecord Library Talks to the Database

Rails uses the **ActiveRecord** library to read and write data to the database. It also allows Rails to convert a single database row in the task table, like this:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>description</th>
<th>duration</th>
<th>due_date</th>
<th>complete</th>
<th>created_at</th>
<th>updated_at</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Stuck up on c...</td>
<td>Costa Rican m...</td>
<td>30</td>
<td>2014-11-06</td>
<td>t</td>
<td>2014-11-04</td>
<td>11:41</td>
</tr>
<tr>
<td>12</td>
<td>Write paper</td>
<td>Henry Knox an...</td>
<td>240</td>
<td>2014-11-06</td>
<td>t</td>
<td>2014-11-06</td>
<td>13:27</td>
</tr>
<tr>
<td>18</td>
<td>Buy turkey</td>
<td>Don't forget</td>
<td>60</td>
<td>2014-11-27</td>
<td>f</td>
<td>2014-11-26</td>
<td>17:54</td>
</tr>
<tr>
<td>21</td>
<td>Library study</td>
<td>Check the wifi...</td>
<td>90</td>
<td>2014-11-06</td>
<td>t</td>
<td>2014-11-02</td>
<td>08:36</td>
</tr>
<tr>
<td>22</td>
<td>Movie night</td>
<td>Chinatown</td>
<td>120</td>
<td>2014-11-11</td>
<td>f</td>
<td>2014-11-09</td>
<td>11:13</td>
</tr>
<tr>
<td>23</td>
<td>Buy candy</td>
<td>Buy extra can...</td>
<td>45</td>
<td>2010-10-31</td>
<td>t</td>
<td>2010-10-28</td>
<td>15:04</td>
</tr>
<tr>
<td>27</td>
<td>Buy eggs</td>
<td>Go to the mar...</td>
<td>50</td>
<td>2014-11-12</td>
<td>f</td>
<td>2014-11-11</td>
<td>12:38</td>
</tr>
</tbody>
</table>

...to a piece of Ruby data like this:

```
irb(main):001:0> Task
=> Task(id: integer, name: string, description: text, duration: integer, due_date: date, complete: boolean, created_at: datetime, updated_at: datetime)
```

To see how this works, open a Unix shell, change into the **ostapp** application directory, and start the Rails console:

```
cold1:~$ cd ~/railsapps/ostapp/
cold1:/railsapps/ostapp$ rails console
Loading development environment (Rails 3.0.3)
irb(main):001:0>
```

On the console, you can access the **Task** data by typing its name:

```
irb(main):001:0> Task
=> Task(id: integer, name: string, description: text, duration: integer, due_date: date, complete: boolean, created_at: datetime, updated_at: datetime)
irb(main):002:0>
```
It returns the names of all of the attributes of a typical Task object. Task found these by connecting to the database and reading the column names and data-types from the tasks table. It found the table name by converting its own name—Task—into the lowercase plural tasks.

How to Read All the Tasks

As well as informing you about the structure of the table, Task can read its contents:

```
INTERACTIVE SESSION:

irb(main):003:0> Task.all
```

That's a lot of data! The text that appears on the screen represents a whole list of Task objects, surrounded by square brackets and separated by commas:

```
[<Task id: 11, name: "Stock up on coffee",...>, <Task id: 12..., ....]
```

In Ruby, a list of objects like this is called an array. An array allows you to manage a list of separate pieces of data as a single collection. Ruby arrays can store any type of data; for example:

```
INTERACTIVE SESSION:

irb(main):004:0> birthdays=['June 24th', 'February 1st', 'January 8th']
=> ['June 24th', 'February 1st', 'January 8th']
irb(main):005:0> scores=[65,25,7,43,2]
=> [65, 25, 7, 43, 2]
irb(main):006:0>
```

Reading Single Tasks from the Array

Once you have an array of tasks, you need some way to extract the individual tasks. You can access individual items in Ruby using an index. An index is a number that refers to the position of an item in an array. For example, to read the first task from the Task.all array, try this:
INTERACTIVE SESSION:

```ruby
irb(main):007:0> Task.all[0]
irb(main):008:0>
```

That's right—the first item in an array has an index of zero. The second task in the list has index 1:

```ruby
irb(main):008:0> Task.all[1]
irb(main):009:0>
```
Don’t confuse the id of a task object with the index of the task array. The id of a task object is a unique number that identifies it. The index is its position in the array.

Indexes begin at zero because an index is actually an offset—it records how far an item is from the start of the array. The first item in an array is 0 places from the start, the second item is 1 place from the start, and so on.

You can find out how many tasks are stored in the database by asking for the size of the array:

```
INTERACTIVE SESSION:

irb(main):009:0> Task.all.size
=> 7
irb(main):010:0>
```

Given that you can find the size of an array, and you know that indexes begin at zero, how might you display the last task in the list? You can do it like this:

```
INTERACTIVE SESSION:

irb(main):010:0> Task.all[Task.all.size - 1]
=> #<Task id: 27, name: "Buy eggs", description: "Go to the market and buy six eggs.\n\nTry to get organ...", duration: 60, due_date: "2014-11-12", complete: false, created_at: "2014-11-11 12:38:00", updated_at: nil>
irb(main):011:0>
```

Because there are 7 tasks stored in the database, they will be returned with indexes ranging from 0 to 6. The last element’s index—6—is one less than the size of the array. In any array, the last element’s index is size - 1.

This returns the last task in the array.

That's a complex expression; there's a much simpler way of finding the last item in an array:

```
Task.all[Task.all.size - 1]
```
That's better and much easier to remember. You can find the first item in the array in the same way:

Whenever you’re working with a database, you need to consider performance. When you have a small amount of data, reading from the database takes almost no time, but if you have a large amount of data, you might find that your application spends most of its time talking to the database. In all of our examples, each time we queried Task.all from the console, it connected to the database and asked for all of the data stored in the tasks table. If that table contained a million records, you might still be waiting for the response! That's the reason most Rails programmers will store the results in a variable before looking at the contents of the array:
Rails will read the data from the database once and store it in the @tasks variable. Then, you can study the results in @tasks without forcing Rails to connect to the database each time.

### Searching for Data

You may want to search for a single task by the value of one of its attributes. For example, you might want to find the task with the name "Buy eggs". You can do that by using a finder:

```ruby
irb(main):017:0> Task.find_by_name("Buy eggs")
=> #<Task id: 27, name: "Buy eggs", description: "Go to the market and buy six eggs.\n\nTry to get organ...", duration: 60, due_date: "2014-11-12", complete: false, created_at: "2014-11-11 12:38:00", updated_at: nil>
```
But what if there might be several tasks that match what you’re looking for?

**Finding All Matches**

In some cases, you’ll want to find *all* the tasks—not just the *first* one—that match particular criteria.
The simple finders will only find the first matching record. To find all matches, you need `find_all_by_...`:

**INTERACTIVE SESSION:**

```ruby
irb(main):021:0> Task.find_all_by_complete(false)
=> [#<Task id: 18, name: "Buy turkey", description: "Don't forget the cranberries.", duration: 60, due_date: "2014-11-27", complete: false, created_at: "2014-11-26 17:54:00", updated_at: nil>,
    #<Task id: 27, name: "Buy eggs", description: "Go to the market and buy six eggs.
    try to get orga...", duration: 60, due_date: "2014-11-12", complete: false, created_at: "2014-11-11 12:38:00", updated_at: nil>]
irb(main):022:0>
```

**Note** We formatted the output here to make it easier to read and understand. Your output in the Rails console will be formatted differently.

The `find_all_by_...` finders return an array of results, like `Find.all`. That means you can count the results and read them using an index in the same way:

**INTERACTIVE SESSION:**

```ruby
irb(main):022:0> Task.find_all_by_complete(false).size
=> 3
irb(main):023:0> Task.find_all_by_complete(false).first
=> #<Task id: 18, name: "Buy turkey", description: "Don't forget the cranberries.", duration: 60, due_date: "2014-11-27", complete: false, created_at: "2014-11-26 17:54:00", updated_at: nil>
irb(main):024:0>
```

**INTERACTIVE SESSION:**

```ruby
irb(main):024:0> Task.find_all_by_complete(false).last
=> #<Task id: 27, name: "Buy eggs", description: "Go to the market and buy six eggs.\n\nTry to get orga...", duration: 60, due_date: "2014-11-12", complete: false, created_at: "2014-11-11 12:38:00", updated_at: nil>
irb(main):025:0>
```
INTERACTIVE SESSION:

```
irb(main):025:0> Task.find_all_by_complete(false)[1]
irb(main):026:0>
```

Here are some examples that use `find_all_by_...`:

**Which Tasks Are Due on November 6th, 2014?**

```
INTERACTIVE SESSION:

irb(main):026:0> Task.find_all_by_due_date('2014-11-06')
    #<Task id: 21, name: "Library study", description: "Check the wifi's working.", duration: 90, due_date: "2014-11-06", complete: true, created_at: "2014-11-02 08:36:00", updated_at: nil>]
irb(main):027:0>
```

**How Many Tasks Are an Hour Long?**

```
INTERACTIVE SESSION:

irb(main):027:0> Task.find_all_by_duration(60).size
=> 2
irb(main):028:0>
```
Sorting Results

The ActiveRecord code we've looked at so far has not told the database the `order` in which it wants the data to appear. Typically this means that the data is returned from the database in the order that it was created, so you see the oldest records first and the newest records last. Still, this is not guaranteed. Different database systems may order their results differently. If you want to specify an order, you need to pass more information to the database, like this:

```
INTERACTIVE SESSION:

irb(main):028:0> Task.find_all_by_complete(false, :order=>'due_date')
    #<Task id: 27, name: "Buy eggs", description: "Go to the market and buy six eggs.\n\nTry to get organ...", duration: 60, due_date: "2014-11-12", complete: false, created_at: "2014-11-11 12:38:00", updated_at: nil>,
    #<Task id: 18, name: "Buy turkey", description: "Don't forget the cranberries.", duration: 60, due_date: "2014-11-27", complete: false, created_at: "2014-11-26 17:54:00", updated_at: nil>]
irb(main):029:0>
```

Task.find_all_by_complete(false, :order=>'due_date')

The `:order` option specifies how the data will be sorted by the database before it's returned to your application. You can use it with all of the finders and with all:

```
Note

The images here show how the `:order` option changes the order in the output array; the order in the database doesn't change.
```

The `:order` option specifies how the data will be sorted by the database before it's returned to your application. You can use it with all of the finders and with all:
INTERACTIVE SESSION:

```
irb(main):029:0> Task.all(:order=>'name')
=> [#<Task id: 23, name: "Buy candy", description: "Buy extra candy. \n\nSee if they have any of those jel...", duration: 45, due_date: "2010-10-31", complete: true, created_at: "2010-10-28 15:04:00", updated_at: nil>,
  #<Task id: 27, name: "Buy eggs", description: "Go to the market and buy six eggs. \n\nTry to get organ...", duration: 60, due_date: "2014-11-12", complete: false, created_at: "2014-11-11 12:38:00", updated_at: nil>,
  #<Task id: 18, name: "Buy turkey", description: "Don't forget the cranberries.", duration: 60, due_date: "2014-11-27", complete: false, created_at: "2014-11-26 17:54:00", updated_at: nil>,
  #<Task id: 21, name: "Library study", description: "Check the wifi's working.", duration: 90, due_date: "2014-11-06", complete: true, created_at: "2014-11-02 08:36:00", updated_at: nil>,
irb(main):030:0>
```

To sort by multiple columns, set the `:order` option to a comma-separated list of attribute names. To sort first by `duration` and then by `due_date`, try this:
INTERACTIVE SESSION:

```ruby
irb(main):030:0> Task.all(:order=>'duration, due_date')
    #<Task id: 23, name: "Buy candy", description: "Buy extra candy.\n\nSee if they have any of those jel...", duration: 45, due_date: "2010-10-31", complete: true, created_at: "2010-10-28 15:04:00", updated_at: nil>,
    #<Task id: 27, name: "Buy eggs", description: "Go to the market and buy six eggs.\n\nTry to get orga...", duration: 60, due_date: "2014-11-12", complete: false, created_at: "2014-11-11 12:38:00", updated_at: nil>,
    #<Task id: 18, name: "Buy turkey", description: "Don't forget the cranberries.", duration: 60, due_date: "2014-11-27", complete: false, created_at: "2014-11-26 17:54:00", updated_at: nil>,
    #<Task id: 21, name: "Library study", description: "Check the wifi's working.", duration: 90, due_date: "2014-11-06", complete: true, created_at: "2014-11-02 08:36:00", updated_at: nil>,
irb(main):031:0>
```

Where multiple tasks have the same duration, they are further sorted by due_date.

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>description</th>
<th>duration</th>
<th>due_date</th>
<th>complete</th>
<th>created_at</th>
<th>updated_at</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Stock up on coffee</td>
<td>Costa Rican medium roast</td>
<td>30</td>
<td>2014-11-06</td>
<td>t</td>
<td>2014-11-04 11:41</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Buy candy</td>
<td>Buy extra candy. See if they have any of those jel...</td>
<td>45</td>
<td>2010-10-31</td>
<td>t</td>
<td>2010-10-28 15:04</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Buy eggs</td>
<td>Go to the market and buy six eggs. Try to get orga...</td>
<td>60</td>
<td>2014-11-12</td>
<td>f</td>
<td>2014-11-11 12:38</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Buy turkey</td>
<td>Don't forget the cranberries.</td>
<td>60</td>
<td>2014-11-27</td>
<td>f</td>
<td>2014-11-26 17:54</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Library study</td>
<td>Check the wifi's working.</td>
<td>90</td>
<td>2014-11-06</td>
<td>t</td>
<td>2014-11-02 08:36</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Movie night</td>
<td>Chinatown</td>
<td>120</td>
<td>2014-11-11</td>
<td>f</td>
<td>2014-11-09 11:13</td>
<td></td>
</tr>
</tbody>
</table>

Sorted by **duration**... ...then **due_date**.

To reverse the order of the sorting, add "DESC" after the attribute name. "DESC" sorts in a DESCending order from highest to lowest:
**What Just Happened?**

In this lesson, we learned:

- Rails talks to the database with a library called **ActiveRecord**.
- All of your Task data is stored in a database table called **tasks**.
- **Task** tells you the structure of the tasks table.
- **Task.all** returns all of the tasks as an array of Task objects.
- You can read single tasks with an index: **Task.all[2]**.
- You can find out the number of tasks with **Task.all.size**.
- You can read the first and last task in the table with **Task.all.first** and **Task.all.last**.
- You can search for a single task with a **finder**.
- **Task.find_by_name('Buy eggs')** will find a task by name.
- There’s a **find_by_...** finder for every attribute.
- **find_all_by_...** finders return an array of all matching tasks.
- The **:order** option will sort the results from a finder or Task.all.

That’s quite an impressive list of newly attained skills. Good job!
Lesson Objectives

When you complete this lesson, you will be able to:

- use the controller.
- use the action method in Rails and use it in the controller.
- display the results of a custom query on a custom web page.

Putting Data on the Page

You've seen how to create custom queries in Rails with ActiveRecord. You can find out, for example, which tasks still need to be completed by typing this into the Rails console:

```
INTERACTIVE SESSION:

irb(main):001:0> Task.find_all_by_complete(false)
=> [#<Task id: 18, name: "Buy turkey", description: "Don't forget the cranberries.", duration: 60, due_date: "2014-11-27", complete: false, created_at: "2014-11-26 17:54:00", updated_at: nil>,
  #<Task id: 27, name: "Buy eggs", description: "Go to the market and buy six eggs.\n\nTry to get orga...", duration: 60, due_date: "2014-11-12", complete: false, created_at: "2014-11-11 12:30:00", updated_at: nil>]
```

It's useful to have this information available on the command line, but what if you want to put it on the web? How can you display the results of a custom query on a custom web page?

Introducing the Controller

When your Rails application receives a request for a web page, typically, it reads the relevant data from the database and then passes it as a variable to a page template. The template uses the data in the variable to generate the web page that's returned:
So when a user asks to display the task with id 21, the Rails application reads the task from the database, stores it in a variable called `@task` and then displays that variable using `show.html.erb`. When a user asks for any dynamic page, the application performs a similar sequence of events. It might read a single task record or several. It might delete or update a record instead of reading it, but in every case, the application will talk to the database and then generate some sort of response.

The code that contains the exact instructions to execute in each case is called the controller. Your controllers are in your `app/controllers` folder. The controller controls how the application responds to requests.

There's a controller for each type of data that the application uses. The tasks controller looks like this:
class TasksController < ApplicationController
  # GET /tasks
  # GET /tasks.xml
  def index
    @tasks = Task.all
    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @tasks }
    end
  end

  # GET /tasks/1
  # GET /tasks/1.xml
  def show
    @task = Task.find(params[:id])
    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @task }
    end
  end

  # GET /tasks/new
  # GET /tasks/new.xml
  def new
    @task = Task.new
    respond_to do |format|
      format.html # new.html.erb
      format.xml  { render :xml => @task }
    end
  end

  # GET /tasks/1/edit
  def edit
    @task = Task.find(params[:id])
  end

  # POST /tasks
  # POST /tasks.xml
  def create
    @task = Task.new(params[:task])
    respond_to do |format|
      if @task.save
        format.html { redirect_to(@task, :notice => 'Task was successfully created.') }
        format.xml  { render :xml => @task, :status => :created, :location => @task }
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @task.errors, :status => :unprocessable_entity }
      end
    end
  end

  # PUT /tasks/1
  # PUT /tasks/1.xml
  def update
    @task = Task.find(params[:id])
    respond_to do |format|
      if @task.update_attributes(params[:task])
        format.html { redirect_to(@task, :notice => 'Task was successfully updated.') }
        format.xml  { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml  { render :xml => @task.errors, :status => :unprocessable_entity }
      end
    end
  end

  # DELETE /tasks/1
  # DELETE /tasks/1.xml
  def destroy
    @task = Task.find(params[:id])
    @task.destroy
    respond_to do |format|
      format.xml  { head :ok }
    end
  end
end
That's a very long Ruby script—and so it should be. The controller is the real brains of the application. It's the code that's in charge of exactly how the application works. View code, like the page templates, controls how the application looks. Model code is in charge of all of the data—how it's stored in the database and determining whether it's valid. The controller, however, is really the boss. It coordinates the model and the view and brings the application to life.

**How to List Only Incomplete Tasks**

Suppose that instead of seeing every task listed on the index page, you only wanted to see the tasks that are incomplete:

How would you change that? When you go to the index page, Rails generates the display using the `index.html.erb`:

```
OBSERVE:
...
  def index
    @tasks = Task.all
    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @tasks } 
    end
  end
...
```

This code runs when a user requests `index.html`. It **reads all the tasks and stores them in the @tasks variable**. The lines between `respond` and `end` generate the response, including generating the web page with the `index.html.erb` template.

To make the index page show just the incomplete tasks, change just one line of `app/controllers/tasks_controller.rb`:
class TasksController < ApplicationController
  # GET /tasks
  # GET /tasks.xml
  def index
    @tasks = Task.all
    @tasks = Task.find_all_by_complete(false)
    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @tasks }
    end
  end
end

By setting the @tasks variable to Task.find_all_by_complete(false), we ensure that the index.html.erb sees only the incomplete tasks. Open a browser window on the index page, and you'll see this:

When the application received the request for ...tasks/, first it ran the index function in tasks_controller.rb. That read the incomplete tasks from the database and stored them in the @tasks variable. Then the controller called the index.html.erb template to generate a web page to provide a response. The index.html.erb creates a web page using the @tasks just as it did before. Though now, instead of holding a list of all of the tasks in the database, the @tasks variable contains only the incomplete tasks.

How Do You See the Complete Tasks?

Of course the downside to modifying the index function in the controller is that you can no longer see the completed tasks. In a perfect world, you'd have the original ...tasks/ URL show the list of all tasks, and some other URL set aside to display the incomplete tasks. That way, the user could decide which set of tasks to view. Could you add a custom page to the set of scaffolded pages you created for task data?

The first step would be to reset the index function in the task controller so that it generates a page for all tasks. Then you'd create another function called incomplete that only reads the incomplete tasks:
Now, the controller knows how to read everything and how to read just the incomplete tasks. Still, there's a problem. If you make the modifications to tasks_controller.rb, save the file, and then refresh the list page, you can certainly see all of the tasks again, but if you modify the URL to run the incomplete function, this happens:
The application doesn’t appear to be running the new `incomplete` function at all. To see why the code doesn’t work, you’ll need to learn a little about Rails **routes**.

**Routes Define a Path through the App**

When Rails receives a request for a URL, it has to decide which code it needs to run. For example, when Rails receives a request like this:

```
http://login-id.oreillystudent.com/ostapp/tasks/2
```

The application runs the `show` function in the tasks controller, and then uses the `show.html.erb` template to display the task with id == 2. So, how does it know that it needs to do that? It uses a Rails **route**.

A route is a rule that tells Rails the code it needs to run for a URL. The code in Rails is grouped into **actions**. An action is a set of different pieces of code that follows a naming convention. For example, the `index` function in `tasks_controller.rb` and the `index.html.erb` template file together make up the `index` action. The `show` function in the `tasks_controller.rb` and the `show.html.erb` template make up the `show` action. The functions in the controller are called **action methods**.

A Rails route associates a particular set of URLs with a particular action, for example:

<table>
<thead>
<tr>
<th>If Rails is asked for URLs like this</th>
<th>it runs this action</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://login-id.oreillystudent.com/ostapp/tasks/">http://login-id.oreillystudent.com/ostapp/tasks/</a></td>
<td>index</td>
</tr>
<tr>
<td><a href="http://login-id.oreillystudent.com/ostapp/tasks/2">http://login-id.oreillystudent.com/ostapp/tasks/2</a></td>
<td>show</td>
</tr>
</tbody>
</table>

By default, scaffolding creates a set of standard routes. To see what these are, open a Terminal session, change into the `~/railsapps/ostapp` folder, and type this command:
These are the routes that are already defined in the application. Yours may look a little different, depending on the changes you’ve made to your own application. Each displayed line represents a single route. This is the route that tells Rails what to do if someone asks for a page that displays a single task:

**OBSERVE:**

<table>
<thead>
<tr>
<th>Route</th>
<th>Path Pattern</th>
<th>Action</th>
<th>Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>/tasks/:id(.:format)</td>
<td>show</td>
<td>tasks</td>
</tr>
</tbody>
</table>

Here, we specify that `task` is the name of the route and `/tasks/` is the path pattern. If a user requests `/tasks/2`, Rails will use the “show” action and the “tasks” controller.

If someone asks for a URL like:

**OBSERVE:**

`http://login-id.oreillystudent.com/ostapp/tasks/2.html`

Rails will match this URL with the `task` because the part of the URL following the root of the application (the "http://login-id.oreillystudent.com/ostapp" part) is `/tasks/2.html`. This matches the `task` path pattern `/tasks/:id(.:format)`. The `task` route tells Rails that it needs to run the “show” action using the “tasks” controller.

**Note**  
This route will match URLs that end `.../tasks/2`, as well as `.../tasks/2.html`. The `.html` format extension is optional, which is why it is wrapped in parentheses in the path pattern: `/tasks/:id(.:format)`. 

Rails didn’t run our `incomplete` code because there was no matching route. In order to make Rails run our code, we need to create a new route for it.

Can you think of how a route for the `incomplete` tasks might look?

**Creating a Route**
Open the `~/.railsapps/ostapp/config/routes.rb` file. This is the Ruby script that defines all of the routes in the application.

```ruby
Ostapp::Application.routes.draw do
  resources :tasks
  get "intro/dynamic"
  get "my_timer/show_current_time"
end
```

Your version of the file may look slightly different. The comments have been omitted here to make the file more readable. When Rails receives a request for a URL, it checks the routes in this file, and uses the first matching route it finds. The commands in `routes.rb` don't contain a lot of detail about each of the routes. Rails has a very strong set of conventions, so a single command like:

```
resources :tasks
```

...tells Rails that it needs to create a set of standard routes for the `task` data. This single `resources` command creates all of these routes:

```
<table>
<thead>
<tr>
<th>Action</th>
<th>Routing</th>
<th>Parameters</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/tasks/</td>
<td>:id(.:format)</td>
<td>{:action=&gt;&quot;index&quot;, :controller=&gt;&quot;tasks&quot;}</td>
</tr>
<tr>
<td>POST</td>
<td>/tasks/</td>
<td>:id(.:format)</td>
<td>{:action=&gt;&quot;create&quot;, :controller=&gt;&quot;tasks&quot;}</td>
</tr>
<tr>
<td>GET</td>
<td>/tasks/new</td>
<td>:id(.:format)</td>
<td>{:action=&gt;&quot;new&quot;, :controller=&gt;&quot;tasks&quot;}</td>
</tr>
<tr>
<td>GET</td>
<td>/tasks/edit</td>
<td>:id(.:format)</td>
<td>{:action=&gt;&quot;edit&quot;, :controller=&gt;&quot;tasks&quot;}</td>
</tr>
<tr>
<td>GET</td>
<td>/tasks/show</td>
<td>:id(.:format)</td>
<td>{:action=&gt;&quot;show&quot;, :controller=&gt;&quot;tasks&quot;}</td>
</tr>
<tr>
<td>PUT</td>
<td>/tasks/update</td>
<td>:id(.:format)</td>
<td>{:action=&gt;&quot;update&quot;, :controller=&gt;&quot;tasks&quot;}</td>
</tr>
<tr>
<td>DELETE</td>
<td>/tasks/destroy</td>
<td>:id(.:format)</td>
<td>{:action=&gt;&quot;destroy&quot;, :controller=&gt;&quot;tasks&quot;}</td>
</tr>
</tbody>
</table>
```

Currently, when Rails receives a request for this page:

```
http://login-id/oreillystudent.com/ostapp/tasks/incomplete
```

...it is matching to the `task` route, because Rails has been told that pages that match the task route are structured like this:

```
/tasks/:id(.:format)
```

They begin with `/tasks/` and are followed by an `:id`. Rails doesn't know that we intend this id value to be a number. The route just says that `something` will follow `/tasks/` and that Rails needs to remember this as the `id`. In our URL there is something following `/tasks/`—it's the string `"incomplete"`, so Rails interprets `"incomplete"` as the id.

Next, the controller looks for a task with database `id` value of `"incomplete"`. It doesn't find any, that's why you got the error message.
You need to create a new route to tell Rails to run the `incomplete` function in the `tasks` controller. Modify `~/railsapps/ostapp/config/routes.rb` like this:

```ruby
Ostapp::Application.routes.draw do
  get "tasks/incomplete"
  resources :tasks
end
```

Note: Make sure you include a `get "tasks/incomplete"` before `resources :tasks`, to ensure that Rails will match against your new route before any of the others.

The single Ruby command `get "tasks/incomplete"` tells Rails all it needs to know about the new route. Because Rails has very strong naming conventions, it will imply that a route with the path pattern `"tasks/incomplete"` will call the `incomplete` action on the tasks controller. You can check this by running `rake routes` again:
Creating the Template File

You now have a new route named tasks_incomplete. While the route is now in place, we've still got work to do. Open or reload the browser window at http://login-id.oreillystudent.com/ostapp/tasks/incomplete, and you'll see this:

That's right—Rails can't find a template file. The route told Rails to use the tasks controller and the incomplete action, but the action needs code to generate the web page response. Because the action is called incomplete, Rails tries to generate a web page from a template file called incomplete.html.erb, but doesn't find it.

The template file must contain the incomplete function in the controller that is storing the tasks in a variable called @tasks, so the incomplete.html.erb template will need to display the contents of @tasks. You already have a template file that does that—index.html.erb:
<h1>Listing tasks</h1>
<table>
  <tr>
    <th>Name</th>
    <th>Description</th>
    <th>Duration</th>
    <th>Due date</th>
    <th>Complete</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>
  <% @tasks.each do |task| %>
  <tr>
    <td><%= task.name %></td>
    <td><%= task.description %></td>
    <td><%= task.duration %></td>
    <td><%= task.due_date %></td>
    <td><%= task.complete %></td>
    <td><%= link_to 'Show', task %></td>
    <td><%= link_to 'Edit', edit_task_path(task) %></td>
    <td><%= link_to 'Destroy', task, :confirm => 'Are you sure?', :method => :delete %></td>
  </tr>
  <% end %>
</table>

<br />
<%= link_to 'New Task', new_task_path %>

Now you can create your new `<%= complete.html.erb %>` template by copying `index.html.erb`. In a Terminal session, type the command shown:

**INTERACTIVE SESSION:**

```
cold1:~$ cp ~/railsapps/ostapp/app/views/tasks/index.html.erb ~/railsapps/ostapp/app/views/tasks/incomplete.html.erb
cold1:~$
```

Once you’ve copied the file, you can edit it and change the heading:
Now, refresh the Incomplete tasks page; it displays the incomplete tasks:

**Incomplete tasks**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Duration</th>
<th>Due date</th>
<th>Complete</th>
<th>Show</th>
<th>Edit</th>
<th>Destroy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buy eggs</td>
<td>Go to the market and buy six eggs. Try to get organic, if possible</td>
<td>50</td>
<td>2015-12-12</td>
<td>false</td>
<td>Show</td>
<td>Edit</td>
<td>Destroy</td>
</tr>
<tr>
<td>Buy turkey</td>
<td>Don't forget the cranberries.</td>
<td>60</td>
<td>2014-11-27</td>
<td>false</td>
<td>Show</td>
<td>Edit</td>
<td>Destroy</td>
</tr>
<tr>
<td>Movie night</td>
<td>Chinatown</td>
<td>140</td>
<td>2014-11-12</td>
<td>false</td>
<td>Show</td>
<td>Edit</td>
<td>Destroy</td>
</tr>
</tbody>
</table>

**What Just Happened?**

In this lesson, we learned:

- The **controller** coordinates the model code and the view code to generate a response.
- An **action** is the set of code used to respond to a request.
- An action is commonly made up of a function in the controller and a template file.
• A function inside the controller is called an action method.
• Actions are defined by naming conventions; for example, an index function and an index.html.erb template.
• When Rails receives a request, it uses a route to decide which action and controller to use.
• Rails will then call the correct action method in the controller.
• The action method will read or write data to the database and will then normally generate a response with a template with a matching name.
• If you read data with the index action method, the response will be generated using the index.html.erb template.

You've got a whole lot of new stuff to try out in the homework. Get to it and when you're done, I'll see you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.
Lesson Objectives

When you complete this lesson, you will be able to:

- change the look and feel of your code.
- use layouts and URL helpers to manage large numbers of pages efficiently.

The Look and the Links

Let's begin by linking our new *Incomplete tasks* page to the rest of the application. People are likely to open your application on the main tasks index page:

To add a link to the top of that page, edit `/app/views/tasks/index.html.erb` as shown:
Save it and open or refresh the task index page:

Even though this is the most straightforward way to add a link to the page, most Rails developers don’t create links this way. Why? Because, if you decide to change the URL of the Incomplete tasks page later, you’ll also have to change all of the manual links you’ve created. There’s a better tool to use to add links to a page: the **link_to** helper function.

Helper functions can add HTML code to a page for you. We can use `link_to` helper to add hyperlinks to pages, and `link_to` will create a link using a *named route*.

Open a Unix shell and look at the routes in your application. Many of them have been given names. The names appear in the first column of the output when you change into your application's directory and run the `rake routes` command:
The routes have names like `new_task` and `edit_task`. When you added the route to the `Incomplete tasks` page, Rails gave the route the name `tasks_incomplete`. Because the path to the `Incomplete tasks` page has a named route, you can create a hyperlink by passing the name of the route to the `link_to` helper. Edit `index.html.erb` again and replace the manual hyperlink with a call to the `link_to` helper:

```erb
<% @tasks.each do |task| %>
<tr>
  <td><%= task.name %></td>
  <td><%= task.description %></td>
  <td><%= task.duration %></td>
  <td><%= task.due_date %></td>
  <td><%= task.complete %></td>
  <td><%= link_to 'Show', task %></td>
  <td><%= link_to 'Edit', edit_task_path(task) %></td>
  <td><%= link_to 'Destroy', task, :confirm => 'Are you sure?', :method => :delete %></td>
</tr>
<% end %>
</table>
<br />
<%= link_to 'New Task', new_task_path %>
```

We give the `link_to` helper two pieces of information: the **text to display in the hyperlink** and the **name of the route to use**:

```erb`
<%= link_to "Incomplete tasks", :tasks_incomplete %>
```
Refresh the index page. It looks exactly the same as before.

Still, we switch the manual hyperlink with a call to link_to because it makes your application more configurable. If you ever need to change the URL of the Incomplete Tasks page, all of your hyperlinks will send the user to the correct address automatically, because they'll locate the page by the name of its route, rather than its address.

Note The link_to helper inserts the full path to its destination page: /ostapp/tasks/incomplete, so the link will work no matter which page contains it.

Adding the Link to the Other Pages

Users will probably want to get to the Incomplete Tasks from pretty much any of the task pages in the application. To provide that feature, you could just copy and paste the link-code from index.html.erb into each of the other task templates, but that seems pretty inefficient. In addition, if you wanted to add more links to the task pages and create a menu, say, of the index page, the incomplete tasks page, and the New Task page, you'd have to go to each of the task pages and edit the list of links. That defies the Rails Don't Repeat Yourself principle in a big way. If you need to create some piece of shared code, use a Rails layout instead.

What is a Rails Layout?

We saw early on that Rails template files don't generate full web pages—they just generate page fragments. Each template contains the main content of a web page, without the <html>, <head> sections that will be common to all pages.

When the page generated from a template is sent back to the browser though, it includes all of those tags and more. View the Listing tasks page, right-click it, and select View Source:

That extra HTML comes from the layout.

A layout is a template file that contains some boilerplate HTML to be shared by a few pages. The output from one of the other template files—like index.html.erb or show.html.erb—is inserted into the layout file, and this more complete web page is sent back to the web browser.
The output of the template file is inserted into the layout at the yield point in the layout. Let's modify an actual layout file and see how it works.

Change the Page Layouts with application.html.erb

Open app/views/layouts/application.html.erb:

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

This is the layout used for all the pages. The output of each template is placed at the yield point. Whatever you change in this file will be visible on all the pages, so add the link at the top of the page like this:

```
<!DOCTYPE html>
<html>
<head>
  <title>Ostapp</title>
  <%= stylesheet_link_tag :all %>
  <%= javascript_include_tag :defaults %>
  <%= csrf_meta_tag %>
</head>
<body>
  <%= link_to "Incomplete tasks", :tasks_incomplete %>
  <%= yield %>
</body>
</html>
```

Next, edit index.html.erb and remove the code you added there:
Save both files, then refresh the browser to display the list of all tasks. The link still appears at the top of the page:

![Browser window showing the list of tasks](image)

The link looks the same, but now it comes from `application.html.erb`.

Now the link comes from the layout rather than the `index.html.erb` template, and the link appears on each of the other pages:
Creating Separates for Different Page Groups

The problem with editing the application.html.erb layout file though, is that it changes the output of all the pages. Not just the pages that are used to manage the tasks, but every other generated page on the site—even non-task-related pages:

Each Rails application contains a single layout by default—application.html.erb—but you can also create a layout that will be applied to a single category. For example, you can create a layout that will be used with only the task pages.

First, remove the link code that you just added to application.html.erb:
Save the `application.html.erb` file.

This new layout file will be used exclusively by the task pages, so we can add a little more than just a single link. We'll add a complete side menu to the layout.

In the CodeRunner Editor window, click New File (Create new file) and add the code as shown:

```html
<!DOCTYPE html>
<html>
<head>
  <title>Ostapp</title>
  <%= stylesheet_link_tag :all %>
  <%= javascript_include_tag :defaults %>
  <%= csrf_meta_tag %>
</head>
<body>
  <%= link_to "Incomplete tasks", :tasks_incomplete %>
  <%= yield %>
</body>
</html>
```

Save the file in `app/views/layouts` as `tasks.html.erb`.

You'll notice we've created links to three of the named routes in the application:

- **tasks** links to main task index page.
- **tasks_incomplete** links to the Incomplete tasks page you created.
- **new_task** links to the page where a user can create a new link.

The styling in the HTML will add these links to the left side of a typical task page. The main content of each page will be inserted into this layout at the `<%= yield %>` code.

Check a non-task page in your application to be sure that the link at the top of the page has disappeared:
Go back to your task pages; they all have the new look and feel:

The **application.html.erb** file is the layout that a page will use if no others are available. If a template finds a layout that matches its folder name though—for example, **tasks.html.erb**—it uses that one instead.

**How Do You Make a Link Not a Link?**

Layouts are often used to create standard menu bars which link to the major parts of your application, but if you're on the Incomplete tasks page, you don't need to have a link to it in the sidebar. Also, in order to highlight the page you're currently on, it's useful to switch off any links to the current page.

The current layout doesn't do that, so when you are on the **Incomplete tasks** page, the side menu still has an **Incomplete tasks** link:
We can fix that! The `current_page` helper function (the question mark ? is required) tells you whether the page of the current page matches a given path. For example, this will be `true` if you are currently on the Incomplete tasks page:

```
current_page?(:tasks_incomplete)
```

This is the name of a route.

How does that help? Remember that you can embed logic within a web page using `<%...%>` markers. That means you can add logic to replace a link with a simple piece of text like this:
This code checks to determine whether you are currently on the index page (which matches the :tasks route) and if so, generates just the text, All tasks. If you are not on the index page, it generates a link to the index page instead. Save app/views/layouts/tasks.html.erb and look at the main index of tasks; you will see this:

When you change to another page, the link appears:
You could write a similar piece of code for each of the links on the menu, but you're not going to do that! There's a better way to switch links on and off in Rails:

```html
<% if current_page?(:tasks) %>
All tasks
<% else %>
<%= link_to 'All tasks', :tasks %>
<% end %>
```

The `link_to_unless_current` helper does exactly what our piece of embedded logic did— it checks to see if the link it's been given goes to the current page address, and if so, it replaces the link with a small piece of text.
Most of the helper functions that are used in Rails work this way. Sometime, somewhere a programmer has found themselves writing the same piece of code several times. It might not be a long piece of code—but if you have write any code even twice, it's better to have a helper function do it for you.

```
Note
As we've done before, we passed parameters to the `link_to_unless_current` without parentheses, so we didn't write `link_to_unless_current("All tasks", :tasks)`. It's entirely your choice whether you include parentheses or use poetry-style code.
```

Okay—let's try out our new menu. Save the `app/views/layouts/tasks.html.erb` file and then refresh a browser that's showing the main list of tasks. You'll see this:

![Browser showing tasks page with All tasks link not displayed](image)

Because all of the tasks are displayed already, the `All tasks` link is just a piece of text. Now click on the `Incomplete tasks` link:

![Browser showing tasks page with Incomplete tasks link not displayed](image)

The same thing happens again, except this time Rails has switched off the `Incomplete tasks` link. This also happens on the `New task` page:
If you go to a page that doesn’t appear on the menu, all the links are enabled:

With just a little code, we’ve added a lot of functionality to the application! You can treat a Rails template file like a simple web page file, but by using the power of layouts and the page helpers, you can quickly manage the structure of your site.

**What Just Happened?**

In this lesson, we learned:

- The `link_to` helper can create hyperlinks to pages using named routes.
- A `layout` file wraps additional content around the basic content generated by each template file.
- The `application.html.erb` layout file is used by all templates by default.
- You can create a separate layout file for each group of pages in your application.
The name of the file needs to match the group of pages. For example, tasks.html.erb is used for all the templates in the /app/views/tasks directory.

- The current_page? can tell you if the current page matches the given route.
- You can use current_page? to switch links to the current page off.
- It's much easier to switch links off automatically with the link_to_unless_current helper.

Moving right along...

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.
Partials

Lesson Objectives
When you complete this lesson, you will be able to:

- include partials in a template file.

Sharing Page Fragments with Partials
Welcome back! Let's get right to work and learn about another tool we have in Rails to write code efficiently: partials. A partial is a fragment of a web page that can be pulled into a template file. Suppose you want to add a standard piece of text to the pages in your application. For example, let's add a copyright statement to the All tasks and Incomplete tasks pages:

You could type the copyright text into one of these pages, and then copy it into the other, but let me reiterate, in Rails, as in most programming situations, we never want to duplicate code: D.R.Y!

Instead of typing the code into both files, we could store the common text in a third file, then we could include it in the All Tasks and Incomplete Tasks pages:

We do that in Rails using Partial Page Templates, which are usually just called Partials.
Creating Your First Partial

In the text editor, create a new file and type the code as shown:

**CODE TO TYPE:**

```
<hr>
<h4>Copyright</h4>
<p>© Copyright 2014 Taskmasters Unltd.</p>
<hr>
```

Save it as `railsapps/ostapp/app/views/tasks/_copyright.html.erb`. The name of the file is important: all partials have names that begin with an underscore (_) character, to distinguish them from normal template files. Now that we've created the partial, we can modify the `app/views/tasks/index.html.erb` template to reference it:

**CODE TO TYPE:**

```
<h1>Listing tasks</h1>
<table>
  <tr>
    <th>Name</th>
    <th>Description</th>
    <th>Duration</th>
    <th>Due date</th>
    <th>Complete</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>
  <tbody>
    <% @tasks.each do |task| %>
      <tr>
        <td><%= task.name %></td>
        <td><%= task.description %></td>
        <td><%= task.duration %></td>
        <td><%= task.due_date %></td>
        <td><%= task.complete %></td>
        <td><%= link_to 'Show', task %></td>
        <td><%= link_to 'Edit', edit_task_path(task) %></td>
        <td><%= link_to 'Destroy', task, :confirm => 'Are you sure?', :method => :delete %></td>
      </tr>
    <% end %>
  </tbody>
</table>
<%= render "copyright" %>
<br />
<%= link_to 'New Task', new_task_path %>
```

We include the contents of the `_copyright.html.erb` partial with the `render` function. The `render` function evaluates the contents of a template or partial file and converts it to a string. By including the `render` function inside `index.html.erb`, and wrapping it in the `<%= ... %>` tags, we're inserting the contents of the partial into the template file.

**Note** We don't need to give the full filename for the partial; we just refer to the partial as "copyright" because all partial files begin with an underscore and end with the `.html.erb` extension.

Load or refresh the All Tasks page; you can see the copyright statement at the bottom:
We can now make the same change to the incomplete.html.erb template:

```html
<% @tasks.each do |task| %>
  <tr>
    <td><%= task.name %></td>
    <td><%= task.description %></td>
    <td><%= task.duration %></td>
    <td><%= task.due_date %></td>
    <td><%= task.complete %></td>
    <td><%= link_to 'Show', task %></td>
    <td><%= link_to 'Edit', edit_task_path(task) %></td>
    <td><%= link_to 'Destroy', task, :confirm => 'Are you sure?', :method => :delete %></td>
  </tr>
<% end %>
</table>
```

Take a look at the Incomplete tasks page and you see the same copyright statement. So now you have two pages reusing the text in the _copyright.html.erb file:
By putting the shared code into a separate file, we reduce the amount of code in the application, and make the application easier to manage—now when you need to change the year in the copyright statement, you’ll only need to change it in one place.

Removing More Duplication

We used a partial to avoid duplication in the index.html.erb and incomplete.html.erb files, but what about the duplicated code that was already in those files? We created incomplete.html.erb by copying the index.html.erb file. Even though we made some changes to the incomplete.html.erb file—for example, we changed the heading—most of the code is still exactly the same in both files:

Both templates generate an HTML table that contains a list of tasks. To avoid this duplication, we can move the table-generation code into a separate partial that index.html.erb and incomplete.html.erb can share.

You already typed this code once, so we won’t make you type it again. In the editor, select the HTML table generation code and copy it to the clipboard:
Create a new file, right-click in line 1, and select **Paste** to paste the copied code:

```
CODE TO TYPE:

<table>
  <tr>
    <th>Name</th>
    <th>Description</th>
    <th>Duration</th>
    <th>Due date</th>
    <th>Complete</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>
  <% @tasks.each do |task| %>
    <tr>
      <td><%= task.name %></td>
      <td><%= task.description %></td>
      <td><%= task.duration %></td>
      <td><%= task.due_date %></td>
      <td><%= task.complete %></td>
      <td><%= link_to 'Show', task %></td>
      <td><%= link_to 'Edit', edit_task_path(task) %></td>
      <td><%= link_to 'Destroy', task, :confirm => 'Are you sure?', :method => :delete %></td>
    </tr>
  <% end %>
</table>
```

Save it as `/railsapps/ostapp/app/views/tasks/_tasktable.html.erb`. Now switch back to `index.html.erb` and replace the HTML table code with a call to the new partial:
Save `index.html.erb` and then look at the list of All Tasks in the application. It’s identical to the earlier version, but now the entire table comes from a different file.
Now we can make the same change to the incomplete.html.erb file:

```html
<% @tasks.each do |task| %>
  <tr>
    <td><%= task.name %></td>
    <td><%= task.description %></td>
    <td><%= task.duration %></td>
    <td><%= task.due_date %></td>
    <td><%= task.complete %></td>
    <td><%= link_to 'Show', task %></td>
    <td><%= link_to 'Edit', edit_task_path(task) %></td>
    <td><%= link_to 'Destroy', task, :confirm => 'Are you sure?', :method => :delete %></td>
  </tr>
<% end %>
<% end %>
<% render "tasktable" %>
<% render "copyright" %>
<br />
<%= link_to 'New Task', new_task_path %>
```

Save this change, and your _tasktable.html.erb partial generates the HTML tables on the Incomplete Tasks page:
Partials may contain the same kinds of rich content that you can put into *normal* template files. So partials are good for sharing content, as well as code.

**Sharing Code is Good**

The two HTML tables look just like they did before. By creating a partial, any future changes to the code only need to be made in one place, rather than two—or three, or four, or ten!

Let's work through an example. Modify the `_tasktable.html.erb` partial. A user will probably want to know which tasks will take a long time to perform, so let's highlight all the tasks that are expected to last longer than an hour. Modify `_tasktable.html.erb` as shown:

```ruby
<% @tasks.each do |task| %>
<tr<br % if task.duration > 60 then%>bgcolor="MistyRose"% end %>
  <td><%= task.name %></td>
  <td><%= task.description %></td>
  <td><%= task.duration %></td>
  <td><%= task.due_date %></td>
  <td><%= task.complete %></td>
  <td><%= link_to 'Show', task %></td>
  <td><%= link_to 'Edit', edit_task_path(task) %></td>
  <td><%= link_to 'Destroy', task, :confirm => 'Are you sure?', :method => :delete %>
</tr>
<% end %>
</table>
```

Save it and reload the *Listing tasks* page:
This additional logic changed the background color of the tasks that have a duration of greater than 60 minutes. Because we changed the code in the _tasktable.html.erb partial, the change also appears on the Incomplete tasks page:
The Incomplete Tasks page displays a single table of the outstanding tasks on the system. What if we want to add a second table that shows the incomplete tasks that are past their due date?

We can find this second set of incomplete and overdue tasks by adding a line to /railsapps/ostapp/app/controllers/tasks_controller.rb:

```ruby
# in tasks_controller.rb
def incomplete
  # code to display incomplete tasks
end
```
class TasksController < ApplicationController
  def index
    @tasks = Task.all
    respond_to do |format|
      format.html # index.html.erb
      format.xml { render :xml => @tasks }
    end
  end

  def incomplete
    @tasks = Task.find_all_by_complete(false)
    @overdue = Task.where("due_date < ? and complete = ?", Date.today, false)
    respond_to do |format|
      format.html
      format.xml { render :xml => @tasks }
    end
  end

  def show
    @task = Task.find(params[:id])
    respond_to do |format|
      format.html # show.html.erb
      format.xml { render :xml => @task }
    end
  end

  def new
    @task = Task.new
    respond_to do |format|
      format.html # new.html.erb
      format.xml { render :xml => @task }
    end
  end

  def edit
    @task = Task.find(params[:id])
  end

  def create
    @task = Task.new(params[:task])
    respond_to do |format|
      if @task.save
        format.html { redirect_to(@task, :notice => 'Task was successfully created.') }
        format.xml { render :xml => @task, :status => :created, :location => @task }
      else
        format.html { render :action => "new" }
        format.xml { render :xml => @task.errors, :status => :unprocessable_entity }
      end
    end
  end

  def update
    @task = Task.find(params[:id])
    respond_to do |format|
      if @task.update_attributes(params[:task])
        format.html { redirect_to(@task, :notice => 'Task was successfully updated.') }
        format.xml { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml { render :xml => @task.errors, :status => :unprocessable_entity }
      end
    end
  end

  def destroy
    @task = Task.find(params[:id])
    @task.destroy
    respond_to do |format|
      format.html { redirect_to(tasks_url, :notice => 'Task was successfully destroyed.') }
      format.xml { head :ok }
    end
  end
end
This additional line of code will find all of the tasks that are both incomplete and past their due date:

**OBSERVE:**

```ruby
@overdue = Task.where(["due_date < ? and complete = ?", Date.today, false])
```

This is the first time we've seen this type of ActiveRecord code. The `Task.where(...)` function allows us to find all the tasks in the database that satisfy a complex set of conditions. In this case, we want all of the tasks for which `complete = false` and `due_date < today` is `true`. The `?` characters are placeholders for the `false` and `Date.today` values we want to find.

**Note**

The conditions are not written in the Ruby language, but in a lower-level database language called Structured Query Language, better known by its initials SQL (pronounced sequel). SQL is beyond the scope of this course, but the syntax is relatively straightforward.

When the `incomplete.html.erb` template is called, it will see two variables:

- `@tasks` is a list of all of the incomplete tasks.
- `@overdue` is list of all of the incomplete tasks that are also past their due dates.

### How Do We Display the Overdue Tasks?

We have variables containing the list of incomplete tasks, and the list of overdue incomplete tasks, but can we use `_tasktable.html.erb` to display both? The partial currently displays the contents of only the `@tasks` variable:

**OBSERVE:**

```erb
<% @tasks.each do |task| %>
  <tr <%= if task.duration > 60 then%>bgcolor="MistyRose"<% end %>>
    <td><%= task.name %></td>
    <td><%= task.description %></td>
    <td><%= task.duration %></td>
    <td><%= task.due_date %></td>
    <td><%= task.complete %></td>
    <td><%= link_to 'Show', task %></td>
    <td><%= link_to 'Edit', edit_task_path(task) %></td>
    <td><%= link_to 'Destroy', task, :confirm => 'Are you sure?', :method => :delete %></td>
  </tr>
<% end %>
```

The partial will display the incomplete tasks stored in the `@tasks` variable, but it won't display the overdue tasks stored in `@overdue`. We can fix this problem by handing the data to the partial explicitly in a local variable.

A local variable is a variable that is visible to only a small amount of code—in this case we'll create a local variable that will be visible only to the `_tasktable.html.erb` partial. Instead of using the `@tasks` variable, we'll modify the `_tasktable.html.erb` partial so that it displays the tasks stored in a local variable called `tasks`:
Local variable names don’t begin with @. The partial will need to be told each time which value to use for the local variable. That means that we’ll need to change the code in the index.html.erb and incomplete.html.erb files:

Now that we pass the name of a partial, as well as a set of locals, we have to name our parameters. So, rather than typing render "tasktable", we need to type render :partial => "tasktable".... Next, we pass the set of locals using :locals=>.... The :locals=>{:tasks=>@tasks} code tells the partial that it has a local variable called tasks that will have the same value as the @tasks variable. As always in Ruby, because we are passing the name of something to a function, we do it with a symbol. Because the local variable is called tasks, we pass the symbol :tasks to the render function.

Once you’ve modified both index.html.erb and incomplete.html.erb, save the files and then check to make sure that the All Tasks and Incomplete Tasks pages still work. They won’t show any past-due tasks yet.

Why Do Local Variables Help?

Why did we introduce a local variable into the _tasktable.html.erb file? After all, the pages now work exactly the same way they did before—the only difference is that we now have a little more code.

We added the local variable to our _tasktable.html.erb partial to make it more reusable. Previously, the partial would only display the tasks stored in the @tasks variable. The @tasks variable was defined just once per page, so we couldn’t display two different tables in the same page.
Now that we specify the data the _tasktable.html.erb partial will display each time, we can use it to generate multiple tables in the same page, with each containing different data.

To see how this works, modify incomplete.html.erb as shown:

```html
<%= render :partial=>'tasktable', :locals=>{:tasks=>@tasks} %>
<%= render :partial=>'tasktable', :locals=>{:tasks=>@overdue} %>
<%= render "copyright" %>
<%= link_to 'New Task', new_task_path %>
```

Now we call the _tasktable.html.erb for a second time on the same page. This time the second table passes the overdue list of tasks to the partials by setting the partial’s local tasks variable to the value of the @overdue variable that was created in the controller. To see the second table, open a browser window and look at the Incomplete Tasks page.

**Note**

With the current data, you have no incomplete tasks with a due date in the past. To make sure that the table is working, edit the Buy candy task and clear the Complete check box for it.

What Just Happened?

We’ve seen how partials can reduce duplicate code and make your application more manageable. When they’re used alongside templates and layouts, partials help you create advanced HTML interfaces without a significant increase the amount of code you need to manage.

In this lesson, we learned:
A partial is a page fragment that can be included in a template file. You include partials by calling the `render` function.

The `render` function evaluates the contents of the partial and then returns a string, which can be inserted into the template output.

When you create a partial, you can call it from several template files. Local variables increase the reusability of partials.

Work with partials some more as you do the homework assignment. Then meet me in the next lesson!
Lesson Objectives
When you complete this lesson, you will be able to:

- secure your application using authentication.
- add authentication to all of your methods using the before-filter.

Adding Security to the Incomplete Tasks Page

We have a lot of features in our application now, but there's one major feature missing: security.

All of the data in our application can be edited by anybody, from anywhere, at any time. Under those conditions, when your application is available on the web, bad things can happen.
One way to secure your application is with **authentication** which forces users to identify themselves. The first type of authentication we'll go over is **HTTP basic authentication**. This kind of security displays a username and password dialog box whenever you try to access a secure web page:
It takes a relatively small amount of code to add basic authentication to your application. Suppose you want users to log in before they can see the page displaying the list of incomplete tasks. You can do that.

Modify /railsapps/ostapp/app/controllers/tasks_controller.rb as shown:

```ruby
class TasksController < ApplicationController
def index
  @tasks = Task.all
  respond_to do |format|
    format.html # index.html.erb
    format.xml  { render :xml => @tasks }
  end
end
def incomplete
  authenticate_or_request_with_http_basic("Incomplete tasks") do |username, password|
    username == 'admin' && password == 'kingFish'
  end
  if response_body == nil then
    @overdue = Task.where("due_date < ? and complete = ?", Date.today, false)
    respond_to do |format|
      format.html
      format.xml  { render :xml => @tasks }
    end
  end
end
def show
  @task = Task.find(params[:id])
  respond_to do |format|
    format.html # show.html.erb
    format.xml  { render :xml => @task }
  end
end...
```

This code requires the user to sign in to the Incomplete Tasks page with the username admin and the password kingFish. Open or refresh the application and go to the main list of tasks. The list of tasks appears just as it did before:
When you're writing code to test HTTP authentication, you may want to use a separate browser from the one you use to access the O'Reilly School, because once you are logged in, you will stay logged in until you close the browser. You won't want to close down this lab, so if possible, access this lab in your usual browser, and use a different browser (Chrome, Safari, Firefox...) to test your application. Also, when you log in, make sure you don't ask the browser to remember the username and password. You may want to sign in with several different usernames and you won't be able to do this if the browser remembers any previous credentials.

When you click on the link to display the incomplete tasks, a username/password dialog box appears. The dialog box that appears includes the realm name. This is the string that we passed to the authenticate_or_request_with_http_basic function:
OBSERVE:

```ruby
authenticate_or_request_with_http_basic("Incomplete tasks") do |username, password|
  username == 'admin' && password == 'kingFish'
```

It identifies the section of the website that is secured, which means that you can split your website into separate sections that are secured independently. So, you might have a section of your application that is accessible to administrators only, a section that is accessible to registered users, and a part of your application that is completely public.

**How Basic Authentication Works**

When you enter the URL of a secured web page, the browser sends a request to the server. If a user asks to see the *Incomplete tasks* page, Rails runs the `incomplete` action-method in the `tasks_controller`. The code we added checks to see if there's a username and password present in the request.

If the browser includes a username and password, this code checks to make sure that they're valid. If the username and password are incorrect or not present, the `authenticate_or_request_with_http_basic` function generates a response that tells the browser that the request was not authorized.

The first time your browser tries to access the *Incomplete Tasks* page, it won't include a username or password. The server will respond immediately with a *you-are-not-authorized* response. That's why the first time you go to the page, the browser displays the username/password dialog. The browser knows now that it needs a username and password, so it asks you for them. Once you sign in, the browser repeats the request, but this time it includes the username and password you supplied.

If the username and password are correct, the `authenticate_or_request_with_http_basic` doesn't generate an error response. If there's no error, the controller runs the original code that generates the *Incomplete Tasks* page:
The controller code checks for the existence of an error response by looking at the `response_body` variable. Whenever the application generates any kind of response, it stores it in this variable. If the `response_body` variable still has the value `nil`, we know there was no authentication error, so we can send back the Incomplete Tasks page.

Once a user enters the correct username and password, the browser remembers them. If the server ever returns an unauthorized response for another page in the same security domain (the place where your application's files are hosted), the browser will make a second request automatically, including the username and password, without asking the user. This means that once you sign in, you will stay logged into an area of the website until you close the browser, provided that that area of the website accepts the same username and password credentials.

Securing the Other Task Pages

Now that you know how to secure a single action in the tasks controller, you'll want to secure all the other task pages. You can do this by repeating the same security code at the start of each of the action methods, but that would involve duplicating code. Fortunately, Rails has a feature that allows you to run a piece of code for a whole set of methods.

Rails filters are functions that can be run each time Rails needs to access one of the action-methods.

A before-filter is a function that Rails will run before it runs an action-method.
A before-filter works like this:

1. A request comes in that needs to run an action-method in the controller.
2. If a before-filter exists, Rails runs it.
3. If the before-filter generates any kind of response, Rails returns the response and the action-method is never called.
4. If the before-filter generates no response, Rails continues to run the action-method in the usual way.

So, if you put your authentication code into a separate function, you can use that function as a before-filter for each of the action-methods in the controller. If the user hasn’t provided a correct username and password, your before-filter will generate an error response, and the action-method will not be called.

In the `app/controllers/tasks_controller.rb` file, move the security code into a separate method named `login`. We'll register `login` as a before-filter:
class TasksController < ApplicationController
  def index
    @tasks = Task.all
    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @tasks }
    end
  end
  def incomplete
    authenticate_or_request_with_http_basic("Incomplete tasks") do |username, password|
      username == 'admin' && password == 'kingFish'
    end
    if response_body == nil then
      @tasks = Task.find_all_by_complete(false)
      @overdue = Task.where("due_date < ? and complete = ?", Date.today, false)
      respond_to do |format|
        format.html
        format.xml  { render :xml => @tasks }
      end
    end
  end
  def show
    @task = Task.find(params[:id])
    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @task }
    end
  end
  def new
    @task = Task.new
    respond_to do |format|
      format.html # new.html.erb
      format.xml  { render :xml => @task }
    end
  end
  def edit
    @task = Task.find(params[:id])
  end
  def create
    @task = Task.new(params[:task])
    respond_to do |format|
      if @task.save
        format.html { redirect_to(@task, :notice => 'Task was successfully created.') }
        format.xml  { render :xml => @task, :status => :created, :location => @task }
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @task.errors, :status => :unprocessable_entity }
      end
    end
  end
  def update
    @task = Task.find(params[:id])
    respond_to do |format|
      if @task.update_attributes(params[:task])
        format.html { redirect_to(@task, :notice => 'Task was successfully updated.') }
        format.xml  { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml  { render :xml => @task.errors, :status => :unprocessable_entity }
      end
    end
  end
  def destroy
    @task = Task.find(params[:id])
    @task.destroy
    respond_to do |format|
    end
end
You've moved the security code from the `incomplete` action-method, and placed it in its own function named `login`. The `login` function is registered as a before-filter for the whole class with the line `before_filter :login`. The `login` function will now run before any of the action-methods are called. If the `authenticate_or_request_with_http_basic` generates an error response, Rails will not run the action-method and return the error back to the browser.

We need to add one more element to the before-filter in the controller. Modify `tasks_controller.rb` as shown:

```ruby
... before_filter :login
private
def login
  authenticate_or_request_with_http_basic("The task pages") do |username, password|
    username == 'admin' && password == 'kingFish'
  end
end
```

The `private` keyword hides the `login` function from the outside world and prevents it from being called directly as an action-method. Always use this security precaution when you're using security filters. It reduces the ability of an attacker to create a request that will break through your security code.

**The before-filter Step by Step**

Let's try out our new before-filter. Close the browser you're using to test the website to make sure that the browser forgets any usernames and passwords that you sent previously. Then reopen the browser and try to access the main task page of the website:
Rails received the request for the index page; because there was a before-filter registered on the test controller, it first ran the **login** function:

- There was no password.
- The login filter generated a response...
- ...so the action was not called.

The **login** function checked for the existence of a username and password in the request and found none, so it returned a response to the browser saying that the user wasn’t logged in to this page:
So, the browser displays the username and password dialog box.

Enter the username `admin` and the password `kingFish`, and submit the request again:

The browser submits the request again, but this time it includes the username and password:
These match the credentials in the login, so Rails runs the index action-method and returns the page correctly.

If you try to access another page, let’s say the Incomplete Tasks, the browser remembers the username and password, and can reuse them for any of the pages in the same security domain.
The Listing tasks and Incomplete tasks pages are located within a realm named The task pages. This is the name provided by the code in the login function.

When you request the Incomplete tasks page, the server returns an error response that indicates that you weren’t logged in. Then the browser, automatically, repeats the request and includes the username and password that you’ve already provided. So once you’re logged in to a website with HTTP basic authentication, you’ll stay logged in until you close the browser.

You’ve now secured every task page in your application using HTTP basic authentication.

What Just Happened?

In this lesson, we learned:
That Rails applications need security.

The simplest form of security is called HTTP basic authentication.

You can add HTTP basic authentication by calling the `authenticate_or_request_with_http_basic` method.

This function checks for the existence of a username and password in the request.

If the username and password are not correct, the server returns an `unauthorized` error to the browser.

You can apply authentication to all of the methods in your controller by using a `before-filter`.

Before-filters are run before each action-method in the controller.

Before-filters need to be marked as `private`. This prevents would-be attackers from accessing them directly.

Now that we’re feeling more secure, I’m confident you’ll do well on the homework.
Managing Users

Lesson Objectives

When you complete this lesson, you will be able to:

- allow multiple users to log in to your applications simultaneously.
- add security by creating an administrator password.
- display user information.

Managing Multiple Users

We've created an application that allows a single user to log in with a username and password using HTTP authentication, but we probably want to allow several different people to log in simultaneously.

We could accomplish that by modifying the login code. For example, to create a second user, you can change the login function to take a second username and password, like this:

```
OBSERVE:

def login
    authenticate_or_request_with_http_basic("The task pages") do |username, password|
        username == 'admin' && password == 'kingFish'
        || username == 'courtney' && password == 'palimpsest'
    end
end
```

There's a problem with that approach though—every time you want to add a user to the system, you'll need to modify this code. That's not practical. How else might you do it?

Usernames and passwords are just data. Whenever we've had a set of data to manage before, we've stored the data in the database and used scaffolded web pages to edit it.

So let's create a users table and some scaffolded web pages. We need to provide a name for the table, and we need to specify the name and data type of each attribute that we want to store for each user. We'll record a username, a password, and the user's full name. Start a Terminal session and enter this command:
INTERACTIVE SESSION:

cold1:~-$ cd railsapps/ostapp/
cold1:~/railsapps/ostapp$ rails generate scaffold User full_name:string username:string password:string
  invoke  active_record
  create  db/migrate/20121114134326_create_users.rb
  create  app/models/user.rb
  invoke  test_unit
  create  test/unit/user_test.rb
  create  test/fixtures/users.yml
  route  resources :users
  invoke  scaffold_controller
  create  app/controllers/users_controller.rb
  invoke  erb
  create  app/views/users
  create  app/views/users/index.html.erb
  create  app/views/users/edit.html.erb
  create  app/views/users/show.html.erb
  create  app/views/users/new.html.erb
  create  app/views/users/_form.html.erb
  invoke  test_unit
  create  test/functional/users_controller_test.rb
  invoke  helper
  create  app/helpers/users_helper.rb
  invoke  test_unit
  create  test/unit/helpers/users_helper_test.rb
  invoke  stylesheets
  identical  public/stylesheets/scaffold.css

cold1:~/railsapps/ostapp$

We're generating code, so we're using the **rails** command. Scaffolding creates all of the web pages you need, as well as the controller code, the model code, and the migration script to create a table in the database. A migration, remember, is a script that's used to modify the structure of the database. Although the scaffolding command will create the migration, it won't run it. In order to do that, we need to use the **rake** command:

INTERACTIVE SESSION:

cold1:~/railsapps/ostapp$ rake db:migrate
(in /users/dgriffit1/railsapps/ostapp)
  => CreateUsers: migrating
  => create_table(:users)
    -> 0.0104s
  => CreateUsers: migrated (0.0110s)

cold1:~/railsapps/ostapp$

Whenever we want to manage the application's environment, we use the **rake** command. Here we're using **rake** to run the new migration scripts we just created. You've created the table, and you can now access the page at http://username.oreillystudent.com/ostapp/users, and create a new user as shown:
Are we forgetting something? We need to make sure that the user pages themselves are secured. If we don’t, anyone could create their own user account and then use it to log in to the main system.

We can secure the user pages exactly the same way we secured the task pages: by modifying the controller. Open /railsapps/ostapp/app/controllers/users_controller.rb and make these changes:
class UsersController < ApplicationController
  def index
    @users = User.all
    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @users }
    end
  end
  def show
    @user = User.find(params[:id])
    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @user }
    end
  end
  def new
    @user = User.new
    respond_to do |format|
      format.html # new.html.erb
      format.xml  { render :xml => @user }
    end
  end
  def edit
    @user = User.find(params[:id])
  end
  def create
    @user = User.new(params[:user])
    respond_to do |format|
      if @user.save
        format.html { redirect_to(@user, :notice => 'User was successfully created.') }
        format.xml  { render :xml => @user, :status => :created, :location => @user }
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @user.errors, :status => :unprocessable_entity }
      end
    end
  end
  def update
    @user = User.find(params[:id])
    respond_to do |format|
      if @user.update_attributes(params[:user])
        format.html { redirect_to(@user, :notice => 'User was successfully updated.') }
        format.xml  { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml  { render :xml => @user.errors, :status => :unprocessable_entity }
      end
    end
  end
  def destroy
    @user = User.find(params[:id])
    @user.destroy
    respond_to do |format|
      format.html { redirect_to(users_url) }
      format.xml  { head :ok }
    end
  end
before_filter :login
private
def login
  authenticate_or_request_with_http_basic("The admin area") do |username, password|
    username == 'admin' && password == 'kingFish'
  end
end
end
We set the realm name to "The admin area," which will be displayed for the user in the login dialog box.

You've created a user in the database and you've secured the administrator pages. Next, you need to modify the `task_controller` so it validates the username and password against the data that you stored in the `users` table.

### Authenticating Against the Database

Currently, the `task_controller` authenticates against a single username and password. This is the existing login code in the `task_controller`:

```ruby
before_filter :login
private
  def login
    authenticate_or_request_with_http_basic("The task pages") do |username, password|
      username == 'admin' && password == 'kingFish'
    end
  end
end
```

You need to modify this code so that instead of checking the username and password variables against static values, it checks them against the data stored in the `users` table. You can find the user record in the database, using the username like this:
OBSERVE:

\[ \text{user} = \text{User.find_by_username}(\text{username}) \]

If the user record exists in the database, this expression stores it in the \text{user} variable. If the user doesn't exist in the database, the \text{user} variable is set to the value \text{nil}. \text{nil} is a special Ruby value that means no value.

You need to check the \text{user} variable to make sure that it's not \text{nil}, and that its password matches the one provided by the user.

Modify \text{app/controllers/tasks_controller.rb} as shown:

```ruby
... before_filter :login
private
def login
  authenticate_or_request_with_http_basic("The task pages") do |username, password|
    username == 'admin' && password == 'kingFish'
    user = User.find_by_username(username)
    user != nil && password == user.password
  end
end
end
```

Once this code is in place, try it out. Open a separate browser on one of the task pages, and then type in an invalid username and password, like this:

If you try to log in with an invalid username and/or password...

...the login fails and you're asked to try again

Now, enter the username (courtney) and the password (palimpsest) that you created in the database.

If you sign in correctly, you will have full access to the Task pages.

**Displaying the Current User in the Application**

When you sign in, it might be useful to see your current user's name appearing somewhere on each of the web pages.
In order to display the current user, you need to make the data available to the page templates. The existing code stores the user in a variable named `user`. To make this data visible to the page templates, you'll need to prefix it with an @ symbol. Edit `/railsapps/ostapp/app/controllers/tasks_controller.rb` to rename the `user` variable:
Now, when a user logs in, their details will be stored in the `@user` variable. Variables with names that begin with the `@` symbol are visible to page template files, so they can use the `@user` variable in our generated web pages. So, which pages should we modify? We could edit each of the separate page template files and add in information about the current user, but we don’t need to do that. If we modify the Task layout file, we can make sure that the user information appears on each of the task pages. Modify `railsapps/ostapp/app/views/layouts/tasks.html.erb` as shown:

```html
<!-- DOCTYPE html -->
<html>
<head>
  <title>Ostapp</title>
  <%= stylesheet_link_tag :all %>
  <%= javascript_include_tag :defaults %>
  <%= csrf_meta_tag %>
</head>
<body>
  <div style="float: right">
    Signed in as: <strong><%= @user.full_name %></strong>
  </div>
  <img src="http://www.oreillyschool.com/images/OST_Logo.gif"/>
  <br/>
  <div style="display:table-cell; width: 200px;">
    ![TASKS](%h1>
    <ul>
      <li><%= link_to_unless_current 'All tasks', :tasks %></li>
      <li><%= link_to_unless_current 'Incomplete tasks', :tasks_incomplete %></li>
      <li><%= link_to_unless_current 'New task', :new_task %></li>
    </ul>
  </div>
  <div style="display:table-cell">
    <%= yield %>
  </div>
</body>
</html>
```

By modifying the layout, we make sure that the same change will apply to all of the task pages. Refresh the browser that’s logged in to one of the task pages; you'll see your current user’s full name at the top right corner of the screen.
The application now supports multiple users. They can be managed by an administrator, who can enter and update all of the user information. Each of those users can log in to modify the tasks that are held on the system.

Note
In a complete, robust application, you would add more validation checking; for example, checking for the presence of a username and password when a user is created or updated, and a check to validate the uniqueness of usernames.

What Just Happened?
In this lesson, we learned:

- If you want to support multiple logins, store the user information in the database.
- You can create the user pages and table with scaffolding.
- You need to make sure that your user pages are secure with an administrator password.
- You need to make sure that your user pages have a different security level from your other pages by providing a unique body to the authenticate_or_request_with_http_basic command.
- When you're checking the login information provided by the user, you need to make sure that their user record exists and their password is correct.
- If you want to display the current user's information on the web pages, store it in a variable that begins with the @ symbol.
- By modifying the tasks layout, you can display the current user's information on all of the task pages.

So now you know. Practice using these new skills and I'll catch up with you in our next lesson...
Lesson Objectives

When you complete this lesson, you will be able to:

- use ActiveRecord to connect users to their tasks.

Giving Users Their Own Tasks

Our application now requires users to log in before they have access to the list of tasks, but it's the same list of tasks for each user. We want users to see only the tasks they created, so we've still got some work to do.

Most applications on the web need to personalize data—if you check your email, you want to see your email, not someone else's. In this lab, we'll look at ways that ActiveRecord can associate one set of data with another so that you can connect a user to her tasks.

Database Relationships

Our Task pages now use two different kinds of data: tasks and users, but we don't have any way yet of associating a user with the list of tasks that they created.

In ActiveRecord, we connect pieces of data together using database relationships. A database relationship is a connection from a record in one table with one or more records in another table; a database relationship could be used to jump from a task to the user who owns the task.
How are relationships formed between records? How does the database indicate that Courtney Jones is the owner of the Get a haircut task? It does it with ids.

Each record that is stored in the database has an id value stored with it. The id is a number that identifies the record in the table. You don’t need to set the id yourself because ActiveRecord does it for you automatically, every time you save a new record. Rails uses an id when it needs to display a single task:
ids are also useful if you want to connect a record in one table to a record in another table. For example, if you want to say that the Stock up on coffee task belongs to the user Courtney Jones, you can do it by recording Courtney's id against the task record:

![Diagram showing a relationship between Task and User tables with an arrow pointing from User to Task, indicating the user_id foreign key.]

Rails knows to display the task with id==22 because it's mentioned in the URL.

We can record the owner of a task by storing the owner's user_id in the task.

By storing a user's id against a task record, you can record which users own which tasks.

**Keys and Foreign Keys**
In database programming, the id that’s recorded against each record in the database is called a key, because it gives you access to the record. If we want to record a user-id on a task, the user-id is called a foreign key. It’s called a foreign key because the user-id is a key in a different, or foreign, table.

In ActiveRecord, the names you give to keys and foreign keys are important. Keys are generally called id, and Rails manages them for you. If you add a foreign key, always name it name of foreign data_id:

<table>
<thead>
<tr>
<th>OBSERVE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_id</td>
</tr>
</tbody>
</table>

In the code above, the foreign key user_id refers to the id in the user table.

That means if you want to store the id of a user in the tasks table, you should call it user_id. There are ways to tell ActiveRecord that you want to use a different name, but Rails developers almost never do this, because of the principle of Convention over Configuration. Strong naming conventions allow Rails to determine what the application needs to do. When Rails sees the name user_id, it knows that it refers to the id key of the User table.

In order to associate users with tasks, you’ll need to add an extra attribute to the tasks called user_id.

**Add the user_id with a Migration**

We created the user and task pages with scaffolding. That generated a bunch of pages and Ruby code for us—including migrations. Migrations are the scripts that created the tables in the database.

```
railsapps
|-- ostapp
|   |-- app
|   |   |-- config
|   |   |-- db
|   |   |   |-- migrate
|   |   |   |   |-- 20130208234243_create_tasks.rb
|   |   |   |   |-- 20130226142912_create_users.rb
```

The migrations are in the db/migrate folder.

We run these scripts with the rake db:migrate command.

Migrations aren't just used for creating tables though. A migration can be used to make any kind of structural change to
the database. If you want to add an attribute to a data-type, create a table, or delete a table, do it with a migration.

If we want to add a user_id to the tasks table, we'll need to do it with a migration.

Whenever we need to generate code, we do it with the rails command. We've used rails several times before. In most cases (with the exception of running the Rails console), the rails command has generated code for us, but the rails command can generate so many different types of code, that it's easy to forget the format of each command. Let's go into the application directory and ask rails for help:

INTERACTIVE SESSION:

cold1:~$ cd railsapps/ostapp/
cold1:~/railsapps/ostapp$ rails -h
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")
dbconsole Start a console for the database specified in config/database.yml
    (short-cut alias: "db")
new Create a new Rails application. "rails new my_app" creates a
    new application called MyApp in "./my_app"

In addition to those, there are:
application Generate the Rails application code
destroy Undo code generated with "generate"
benchmarker See how fast a piece of code runs
profiler Get profile information from a piece of code
plugin Install a plugin
runner Run a piece of code in the application environment

All commands can be run with -h for more information.

We want to generate a migration script to change the database, so let's find out more about the generate options:

INTERACTIVE SESSION:

cold1:~/railsapps/ostapp$ rails generate -h
Usage: rails generate GENERATOR [args] [options]
General options:
  -h, [--help]    # Print generator's options and usage
  -p, [--pretend] # Run but do not make any changes
  -f, [--force]   # Overwrite files that already exist
  -s, [--skip]    # Skip files that already exist
  -q, [--quiet]   # Suppress status output

Please choose a generator below.
Rails:
  controller
generator
  helper
 integration_test
mailer
migration
model
observer
 performance_test
plugin
resource
 scaffold
 scaffold_controller
 session_migration
stylesheets
To repeat a command you've typed before in a terminal session, press the up-arrow key. Then you can edit the command. For example, to repeat the `rails generate -h` command above and change it to the command below, press the up-arrow key once, and then use the left and right arrow keys to move within the command line to insert the word `migration`. You can cycle through all commands entered in the current session with the up and down arrow keys.

The *generators* are built-in tools that generate code. We want to generate a *migration*; find out how by typing this code:

```
INTERACTIVE SESSION:

cold1:~/railsapps/ostapp$ rails generate migration -h
Usage:
  rails generate migration NAME [field:type field:type] [options]
Options:
  -o, --orm=NAME  # Orm to be invoked
      # Default: active_record
Runtime options:
  -s, [--skip]    # Skip files that already exist
  -p, [--pretend] # Run but do not make any changes
  -q, [--quiet]   # Suppress status output
  -f, [--force]   # Overwrite files that already exist
Description:
  Create rails files for migration generator.
cold1:~/railsapps/ostapp$
```

So to generate our migration, we need to give the migration a *NAME* and also say which fields (columns) we want to add. Enter this command to create a migration to add a *user_id* column to the *tasks* table:

```
INTERACTIVE SESSION:

cold1:~/railsapps/ostapp$ rails generate migration AddUserIdColumnToTasks user_id:integer
    invoke  active_record
    create  db/migrate/20141104110018_add_user_id_column_to_tasks.rb

cold1:~/railsapps/ostapp$
```

```
OBSERVE:

AddUserIdColumnToTasks  user_id:integer
```

Rails determines from the name of the migration what the migration script needs to do. In this case, because the migration name begins with the word *Add*, we probably want to add a column to a table. The name ends with *ToTasks*, which tells Rails to add the column to the *tasks* table. After the name, we have the specification for the column that we want to add to the table, which is an *integer* value named *user_id*.

When you generate the Rails migration, you don’t change the table—you create a script. You can view the script in the *db/migrate* folder; look for a name like this:

```
db/migrate/20141104110018_add_user_id_column_to_tasks.rb
```

This part of the filename is a timestamp; your file will have a different name.

The filename always begins with a timestamp, so your filename won’t look exactly like this example. The rest of the filename comes from the name we gave to the migration. The migration file contains this code:
Observation:

class AddUserIdColumnToTasks < ActiveRecord::Migration
  def self.up
    add_column :tasks, :user_id, :integer
  end
  def self.down
    remove_column :tasks, :user_id
  end
end

Migration scripts contain two functions:

- The self.up function which makes the database change. This function contains the command to add a column to the task table named user_id of type integer.
- The self.down function which can reverse the database change. This function will only be run if an administrator decides to undo a migration.

In order to run the migration and create the user_id column on the table, we need to run the command rake db:migrate. In general, the rake command is used to modify the development environment on your machine:

Interactive Session:

cold1:~/railsapps/ostapp$ rake db:migrate
(in /users/dgriffit1/railsapps/ostapp)
== AddUserIdColumnToTasks: migrating ================================
-- add_column(:tasks, :user_id, :integer)
  -> 0.0080s
== AddUserIdColumnToTasks: migrated (0.0082s) ================================
cold1:~/railsapps/ostapp$

This command runs all outstanding migrations. The other migration scripts in db/migrate have already been run, so rake will only run the migration you just created. Once the migration has run, the user_id column should be in the tasks table. To verify this, run the Rails console and view the attributes on the tasks table by typing Task:

Interactive Session:

cold1:~/railsapps/ostapp$ rails console
Loading development environment (Rails 3.0.3)
irb(main):001:0> Task
=> Task(id: integer, name: string, description: text, duration: integer, due_date: date,
  complete: boolean, created_at: datetime,
  updated_at: datetime, user_id: integer)
irb(main):002:0> quit
cold1:~/railsapps/ostapp$

Now that we've created the user_id attribute, we need to associate it with the id attribute on the users table.

Joining Tables Together

To associate the user_id attribute of the tasks table with the id attribute of the user table, we need to create a database relationship. Database relationships are specified in the model code. Modify /railsapps/ostapp/app/models/task.rb as shown:
The `belongs_to` line creates a relationship between the `tasks` table and the `users` table. Specifying a relationship makes it easier to set the `user_id` of a task record to the correct value, and jump from a `task` to the `user` who owns it.

### Setting the user_id of a Task

To see how to use the new relationship, go to a Unix shell and start the Rails console.

**WARNING** If you're already inside the Rails console, exit the console and then restart it to make sure that you pick up the new changes to the task model.

Now, in the Rails console, let's find a user record from the database. It doesn't matter which user we find—for now we'll read the first user on the database:

INTERACTIVE SESSION:

```
cold1:~/railsapps/ostapp$ rails console
Loading development environment (Rails 3.0.3)
irb(main):001:0> a_user = User.first
irb(main):002:0>
```

Don't close the console session! This line of code stores Courtney's user details in the `a_user` variable. Now that we have a user from the database, we can associate it with a task. Let's read the first task from the database and store it in a variable called `a_task`:

INTERACTIVE SESSION:

```
irb(main):002:0> a_task = Task.first
=> #<Task id: 17, name: "Buy eggs", description: "Go to the market and buy six eggs.\n\nTry to get or...", duration: 60, due_date: "2015-12-12", complete: false, created_at: "2013-02-14 16:00:28", updated_at: "2013-02-20 16:45:33", user_id: nil>
irb(main):003:0>
```

Now we have a task and the user object that we’ve read from the database, so we can use the database relationship to associate the user with the task. If we didn’t have that relationship, we could associate the two objects by setting the `user_id` attribute of the task to the id of the user object, like this:
Even though this is a pretty small piece of code, you can still run into problems, especially if you play around much with the ids. Every record you read from the database will have an id attribute and it's easy to mix them up. The `belongs_to` relationship helps to simplify this code. Instead of looking at the ids of two objects, we can set the task's user, like this:

```
a_task.user_id = a_user.id
```

This code does exactly the same thing as the previous code. It sets the `user_id` of the task to the id of the user. However, because the code uses the relationship, we don't need to refer to the ids of the task and user objects. That makes the code more readable. Let's try the code on the console:

```
INTERACTIVE SESSION:

irb(main):003:0> a_task.user = a_user
irb(main):004:0>
```

If you view the attributes of `a_task`, you'll see that the `user_id` has been set correctly:

```
INTERACTIVE SESSION:

irb(main):004:0> a_task
=> #<Task id: 17, name: "Buy eggs", description: "Go to the market and buy six eggs.\nTry to get or...", duration: 60, due_date: "2015-12-12", complete: false, created_at: "2013-02-14 16:00:28", updated_at: "2013-02-20 16:45:33", user_id: 1>
irb(main):005:0>
```

### Finding a Task's User

The `belongs_to` relationship allows us to associate a user with a task more efficiently, but it saves just a small amount of code. If you already have a task, and you want to find out more about its user, then the relationship will save you a lot of code.

Let's say you already have a task stored in the `a_task` variable, and you want to find the full name of the task's user. Without the relationship, you would have to read the user record from the database, using code that looks like this:

```
User.find_by_id(a_task.user_id).full_name
```

We need to read the user from the database, by id...

...so we need the id of the user...

...and once we have the user, we can read the full_name.
With the relationship, we can find the full name of the task's user:

\[ a\_\text{task}.\text{user.full_name} \]

You can leap straight from the `a_task` to the user with the expression `a_task.user`. You don't have to tell ActiveRecord to perform a second read from the database—you can access the task's user as if that user was an attribute of the task:

```
INTERACTIVE SESSION:

irb(main):005:0> a_task.user.full_name
=> "Courtney Jones"
irb(main):006:0> quit
```

Note

Because of the `belongs_to` relationship, you don't have to read the user from the database explicitly because ActiveRecord will do it implicitly for you. ActiveRecord will read the user details the first time you ask for them. It will then cache them in memory, so, if you ask for the task's user a second time, ActiveRecord can avoid reading them again. All of this magic is hidden from you by the relationship.

The `belongs_to` relationship allows us to specify a task's owner without messing with ids, and helps you to find the details of a task's owner without having to make a separate call to the database. Once we've read the task from the database, we have located its owner.

Now we can modify the application so that each user sees only the tasks that they've created.

**What Just Happened?**

In this lesson, we learned:

- All records in the database have an id.
- The id is called a key because it gives you access to the data.
- You can make one piece of data, like a task, refer to another piece of data, like a user, by storing the user's id on the task.
- The user-id stored on the task is called a foreign key.
- ids can be tricky to manage.
- A database relationship provides an easier way to associate pieces of data with each other.
- A `belongs_to` relationship associates a piece of data with another piece of data that owns it.
- Once you have a relationship, it will manage the ids for you.
- The `belongs_to` relationship will read the owner data implicitly from the database when you need it.

Work through some relationship issues in the homework and I'll see you in the next lesson!
Creating Tasks for a User

Lesson Objectives

When you complete this lesson, you will be able to:

- set the owner of a piece of data in the controller.
- connect users to their tasks outside of the Rails console.

Recording Who Created a Task

We've learned how ActiveRecord relationships help us connect data from different tables together. We created a belongs_to relationship between Users and Tasks, and now we can connect tasks and their owners, like this:

```ruby
a_task.user = a_user
```

So far, we've used this code to associate users and tasks on the Rails console. In this lab, we'll modify the application so that each time a user creates a task, that user will be recorded as the task's owner. In order to do that, we'll need to set the user attribute of the task when the task is created.

Determining Which Code Creates a Task

In order to set the user of each new task, modify the code that creates a new task and saves it to the database.

You can find the code that needs to be modified by observing what happens when a user submits a new task. Go to the New Task form and view the source of the web page:
The new task form sends a post request to the /ostapp/tasks path. A post is one way that web browsers send form data to the server, another is by using get requests, which record all of the data in the URL. get and posts are request methods.

When Rails receives the request from the form, it determines which code to run using routes. Let’s take a closer look at routes; open a Unix session, change into the application directory, and view the routes using the rake routes command:
Each route has both a path and a method. Rails will use both the method (POST) and the path to find a matching route.

The browser sends the New Task data to /ostapp/tasks, but the O'Reilly servers are configured so that all the paths begin with /ostapp. So, Rails will look for a POST route that matches just the /tasks portion of the path.

This is the route it will find:

POST /tasks{:.format} { :action=>"create", :controller=>"tasks"}

If a user submits the details of a new task from the web, Rails runs the create action-method in the Tasks controller. Open the app/controllers/tasks_controller.rb file and take a look at that action method:
The `create` action method *converts the form data into a task*. Then it *decides what response is needed* (usually, it's HTML). Then, *it attempts to save the task to the database and, if successful, sends the HTML page displaying the new task to the browser.*

In the `create` action method, each new `Task` object gets created and saved into the database. We would also set the user for each new task in the `create` method. To set the user, we use the same kind of code we tried out in the console. Once we’ve created a brand new task object, we set the `user` attribute of the task to the user who’s logged in. Fortunately, when a user logs in, we store their user details in the `@user` variable. This was the code we added earlier to make sure the user was logged in before they could access the task pages. The login method is located at the end of the tasks controller:

```ruby
railsapps/ostapp/app/controllers/tasks_controller.rb

before_filter :login
private
  def login
    authenticate_or_request_with_http_basic("The task pages") do |username, password|
      @user = User.find_by_username(username)
      @user != nil &amp; password == @user.password
    end
  end
end
```

In this method, the *user information* is stored in the `@user` variable.

---

**Note**

The `@` character in `@user` is important. Variables with names that begin with `@` are called *instance variables*. They can be read by code in a page template file. Also, they can be read by other action-methods in the controller. Because we stored the current user in a variable called `@user`, we can read its value in the `create` method. If we’d stored it in a variable named `user` instead, we wouldn’t be able to read it.

To set the user of a newly created task, add this code to the `create` action-method:
class TasksController < ApplicationController
  def index
    @tasks = Task.all
    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @tasks }
    end
  end
  def incomplete
    @tasks = Task.find_all_by_complete(false)
    @overdue = Task.where(["due_date < ? and complete = ?", Date.today, false])
    respond_to do |format|
      format.html
      format.xml  { render :xml => @tasks }
    end
  end
  def show
    @task = Task.find(params[:id])
    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @task }
    end
  end
  def new
    @task = Task.new
    respond_to do |format|
      format.html # new.html.erb
      format.xml  { render :xml => @task }
    end
  end
  def edit
    @task = Task.find(params[:id])
  end
  def create
    @task = Task.new(params[:task])
    @task.user = @user
    respond_to do |format|
      if @task.save
        format.html { redirect_to(@task, :notice => 'Task was successfully created.') }
        format.xml  { render :xml => @task, :status => :created, :location => @task }
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @task.errors, :status => :unprocessable_entity }
      end
    end
  end
  def update
    @task = Task.find(params[:id])
    respond_to do |format|
      if @task.update_attributes(params[:task])
        format.html { redirect_to(@task, :notice => 'Task was successfully updated.') }
        format.xml  { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml  { render :xml => @task.errors, :status => :unprocessable_entity }
      end
    end
  end
  def destroy
    @task = Task.find(params[:id])
    @task.destroy
  end
end
That single line of code sets the `user_id` value of every new task that is created. Let's try it out.

Open a separate browser window and go to the task pages of your application. You'll need to log in with a username and password that you've created in the `users` table.

**Note**

If you can't remember the username and password of an existing user, log in to the user administration pages at http://login-id.oreillystudent.com/ostapp/users/ with the admin/kingFish username and password, and either find an existing user or create a new one.

Once you've logged in to the task pages, go to the `New Task` page and enter a new task:
Even though we’ve added a `user_id` attribute to all the tasks, we don’t see the user-id on any of the pages. Because we generated the pages earlier and once Rails has generated a page, it won’t go back and change it later. We’ll learn how to modify pages and forms to include any new attributes in a later lesson.

When you submit the new task, Rails runs the `create` action-method in the `tasks` controller, which creates a new task based on task data from the form. Your new code sets the `user` of the task just before the task is
To make sure that the new code worked, you can either read all of the data from the Task table, or use a finder to track it down. Open a Terminal session and enter the commands as shown:

```
INTERACTIVE SESSION:
cold1:~$ cd railsapps/ostapp
cold1:~/railsapps/ostapp$ rails console
Loading development environment (Rails 3.0.3)
irb(main):001:0> a_task = Task.find_by_name('Write presentation')
irb(main):002:0>
```

In this case, we found the task by looking for it by name. You can see that this new task has the user_id set to the value 1. That's presumably the correct user-id, but because we've saved the task into the a_task variable, we can use the relationship to find the task creator's details:

```
INTERACTIVE SESSION:
irb(main):002:0> a_task.user
irb(main):003:0>
```

It worked! We're now stamping each new task with the id of the person who created it—but we're only half way done. Sure, we know who owns each task, but we still need to use that information to make sure that a user only sees their own tasks.

**Finding All of a User's Tasks**

The belongs_to relationship we added to the app/models/task.rb script gave us a convenient way to set and find the owner of a single task:

```
a_task.user = a_user
```

This sets the user.

```
a_task.user.full_name
```

This finds the user.
Now, we need to go back to the other way. If we know the current user, how do we find all of the tasks that belong to that user? We need a relationship that goes from the **User** model back to the **Task** model:

If we had a relationship that worked in the reverse direction of the `belongs_to` relationship, then we could jump from a user to the tasks that they own. Each user might own several tasks, so we need to tell Rails that each user potentially `has many` tasks. As it turns out, the reverse of the `belongs_to` relationship is called the `has_many` relationship.

**has_many is the Reverse of belongs_to**

When we created a `belongs_to` relationship from tasks to users, we created the code in the `task.rb` model script. To create the reverse relationship (from users to tasks), we need to do that in the `user.rb` model script. Modify `app/models/user.rb` as shown:

```ruby
railsapps/ostapp/app/models/user.rb

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

The `has_many` relationship allows us to find all of the **Tasks** a **User** owns. We create the relationship by adding
has_many :tasks to the user.rb script. It's important to use the plural form :tasks here. When you're writing relationship code the expressions are designed to be read like English.

Save the user.rb file, then close and reopen the Rails console. Now we can find out if the new relationship helps us find all of the tasks belonging to a user.

---

**Note**

When you start the Rails console, it loads all of the code in your application. If you change a script after that, like app/models/user.rb, the Rails console won't see the new changes. So, if you're already logged into a console, you have to quit and then restart it. That way you'll be able to use the new has_many relationship. Also, you have to be in the ~/railsapps/ostapp application directory for the console to work correctly.

---

### INTERACTIVE SESSION:

coldi:~/railsapps/ostapp$ rails console
Loading development environment (Rails 3.0.3)
irb(main):001:0>

Once you're in the console, find a user from the database and store it in a variable, like this:

### INTERACTIVE SESSION:

irb(main):002:0> a_user=User.find_by_username('courtney')
irb(main):003:0>

Now the has_many relationship will let you find all of the tasks that were created by this user, like this:

### INTERACTIVE SESSION:

irb(main):003:0> a_user.tasks
irb(main):004:0>

Behind the scenes, ActiveRecord used the has_many relationship to find all of the tasks in the database with a user_id that matches Courtney's id. Now that we know how to find all of Courtney's tasks, we can modify the list page so that instead of displaying every task on the database, it just displays those that belong to Courtney:
The All Tasks page displays data read from the database in the index action-method inside the tasks_controller.rb script:

```ruby
def index
  @tasks = Task.all
  respond_to do |format|
    format.html # index.html.erb
    format.xml  { render :xml => @tasks }
  end
end
```

The index action-method reads all of the tasks in the database using Task.all, and then stores them in the @tasks variable. The index.html.erb page template then displays whatever's stored in the @tasks variable.

If you just store the tasks for the current user in the @tasks variable, then those will be the only tasks displayed on this page.

Let's set @tasks to the current user's tasks—@user.tasks. Modify the app/controllers/tasks_controller.rb file like this:
class TasksController < ApplicationController
  def index
    @tasks = Task.all
    @tasks = @user.tasks
    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @tasks }
    end
  end
end

Now refresh the browser window that lists the tasks. You'll see only the tasks that your logged-in user created:

Test your code further. Go to the New Task page and enter another task as shown:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Duration</th>
<th>Due date</th>
<th>Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>Write filtration presentation for lab. Pad out with plenty of charts</td>
<td>120</td>
<td>2014-11-18</td>
<td>false</td>
</tr>
</tbody>
</table>

New Task
Then go back to the All Tasks page and view your new task in the list:

**New task**

Name

Book flight

Description

Get the time of the latest departure.

Duration

30

Due date

2014 November 19

Complete

☐

Create Task

Back
The application is now much more user-specific, but you can still see other people’s tasks on the Incomplete tasks page:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Duration</th>
<th>Due date</th>
<th>Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buy eggs</td>
<td>Go to the market and buy six eggs. Try to get organic, if possible.</td>
<td>60</td>
<td>2015-12-12</td>
<td>false</td>
</tr>
<tr>
<td>Buy candy</td>
<td>Buy extra candy. See if they have any of those jelly worms.</td>
<td>45</td>
<td>2014-10-31</td>
<td>false</td>
</tr>
<tr>
<td>Buy turkey</td>
<td>Don’t forget the cranberries.</td>
<td>60</td>
<td>2014-11-27</td>
<td>false</td>
</tr>
<tr>
<td>Movie night</td>
<td>China town</td>
<td>140</td>
<td>2014-11-12</td>
<td>false</td>
</tr>
<tr>
<td>Write</td>
<td>Write filtration presentation for lab. Pad out with presentation plenty of charts and don’t mention the chemical spill.</td>
<td>120</td>
<td>2014-11-19</td>
<td>false</td>
</tr>
<tr>
<td>Book flight</td>
<td>Get the time of the latest departure.</td>
<td>30</td>
<td>2014-11-19</td>
<td>false</td>
</tr>
</tbody>
</table>

Copyright
© 2014 TaskMasters Unlimited.
The incomplete tasks are read from the database in the `incomplete` action-method, using a finder:

```ruby
def incomplete
  @tasks = Task.find_all_by_complete(false)
  @overdue = Task.where(["due_date < ? and complete = ?", Date.today, false])
  ...
end
```

We know how to find all of the tasks for a given user, but how can we find just the incomplete tasks for a user?

## The Collection of Tasks Has Its Own Finders

We've already seen that you can search for data on the database using finders:

```
Task.find_all_by_complete(false)
```

Task has a set of finders. Returns all of the incomplete tasks from the database.

Task has a set of finders that can search the database for tasks that match some condition—you can find all of the tasks that are incomplete, you can search for a task with a particular name, and so on. Instead of reading all of the tasks from the database, the finders give you a restricted list of tasks.

Now, instead of searching through all of the tasks on the database to locate those that are incomplete, you just search through the tasks given by `@user.tasks` to locate those that are incomplete.

This is where ActiveRecord is really smart. The expression `@user.tasks` looks like a simple list of tasks, but it comes complete with a full set of Task finders. That means that in the same way that you can find all of the incomplete tasks with `Task.find_all_by_complete(false)`, you can also find the incomplete tasks belonging to `@user`, like this:

```
@user.tasks.find_all_by_complete(false)
```

In all the tasks belonging to @user... find all of the incomplete tasks.

The `@user.tasks` part finds all of the tasks for `@user`, and then the `.find_all_by_complete(false)` piece filters that list so that it contains only those tasks that are incomplete. So, you can personalize the Incomplete tasks pages by modifying `app/controllers/tasks_controller.rb` as shown:
class TasksController < ApplicationController
  def index
    @tasks = @user.tasks
    respond_to do |format|
      format.html # index.html.erb
      format.xml { render :xml => @tasks }
    end
  end
  def incomplete
    @tasks = Task.find_all_by_complete(false)
    @overdue = Task.where("due_date < ? and complete = ?", Date.today, false)
    @tasks = @user.tasks.find_all_by_complete(false)
    @overdue = @user.tasks.where("due_date < ? and complete = ?", Date.today, false)
    respond_to do |format|
      format.html
      format.xml { render :xml => @tasks }
    end
  end
...
Most web applications need to associate data together because information in the real world can rarely be stored as a simple, linear list of items. Database relationships like `belongs_to` and `has_many` allow you to deal with that complexity by connecting data sets together, and allow you to navigate between them. In this lesson, you added the `belongs_to` relationship between the tasks and users database tables, to connect tasks to users, and added the `has_many` relationship between the users and tasks database tables, to connect users to tasks. That's pretty advanced stuff!

**What Just Happened?**

In this lesson, we learned:

- You can set the owner of a piece of data in the `create` action-method of the controller.
- The `belongs_to` connects a piece of data to its owner.
- The `has_many` relationship is the reverse of the `belongs_to` relationship.
- The `has_many` relationship returns a list of the owned data.
- The list returned by `has_many` has its own set of finders.

Your relationships are becoming really complicated. Good luck with your homework and I'll see you after you get all of your various relationships handled!

Copyright © 1998-2014 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.
Lesson Objectives

When you complete this lesson, you will be able to:

- You'll access the parameters sent by a form.
- use absolute paths to allow forms to be used by any page in your application.

Forms: Adding Search

In this lesson, we'll look at forms. We've used forms throughout the course, but so far every one has been generated using scaffolded code. In order to see at how forms work, let's add a new feature: search.

We've already covered most of the features we'll need for search. If a user submits a search request, the application will need to find matching tasks on the database, store them in a variable, and then display them in a web page. Let's go ahead and add those basics to the application.

Adding a Custom Search Route

If we're going to add a brand-new function to the application, we need to add a route first. The route will associate an action (a set of code) with a path. Open the config/routes.rb file in the text editor and add this route:

```ruby
Ostapp::Application.routes.draw do
  resources :users
  get "tasks/incomplete"
  get "tasks/search"
  resources :tasks
  get "my_timer/show_current_time"
end
```

Note

Your routes.rb will look a little different if you've been doing your homework!

The get "tasks/search" line tells Rails to associate any GET requests to the ..tasks/search path with a search action using the tasks controller. Remember—Rails can infer this because of its strong naming conventions. So if you open a browser at http://login-id.oreillystudent.com/ostapp/tasks/search, Rails will immediately look for an action-method in the tasks_controller.rb file called search. Let's add that method next.

Adding an action-method

Open the app/controllers/tasks_controller.rb and add a new search action-method as shown:

```ruby
class TasksController < ApplicationController
  def search
    # Add your search logic here
  end
end
```
class TasksController < ApplicationController
  def search
    terms = 'CHEMICAL'
    @tasks = @user.tasks.where(['upper(name) like ? OR upper(description) like ?', '%#{terms}%', '%#{terms}%'])
    respond_to do |format|
      format.html
      format.xml { render :xml => @tasks }
    end
  end
end

def index
  @tasks = @user.tasks
  respond_to do |format|
    format.html # index.html.erb
    format.xml { render :xml => @tasks }
  end
end
...

Let's look at that new search action-method and break it down a little:

OBSERVE:

def search
  terms = 'CHEMICAL'
  @tasks = @user.tasks.where(['upper(name) like ? OR upper(description) like ?', '%#{terms}%', '%#{terms}%'])
  respond_to do |format|
    format.html
    format.xml { render :xml => @tasks }
  end
end

First, we provide a default string to search for (the user will be able to enter a different term). Then, we search the database for tasks where the name or description matches the search term. We indicate how we want Rails to respond (with XML or HTML), and we tell Rails to render the page as we did with the index and incomplete methods. For all three methods, the code will:

- find a list of matching data.
- store the data in a variable called @tasks.
- tell Rails to generate a response.

The only significant difference is that the search action-method finds its data using the code:

OBSERVE:

@tasks = @user.tasks.where(['upper(name) like ? OR upper(description) like ?', '%#{terms}%', '%#{terms}%'])

This is a complex piece of code. It begins with @user.tasks, which means that we'll only search for tasks that belong to the current user—the @user variable stores the user who's logged in. Then we create a query using the where function. This passes a custom condition to the database and returns the records that match. In this case, the condition means "Give me all the tasks with a Name or Description that contains the string in the terms variable."

The terms variable contains the uppercase string "CHEMICAL". The where function converts each name and description to uppercase before searching for the string. That means that we should find any tasks containing "Chemical," "chemical," "cHeMiCAl," and so on, regardless of the letters’ case.
where(...) can take an array of conditions, all of which will need to be true for a task to be selected. In our code, we have only a single condition ("Does the name of condition contain..."), but we still need to place the condition in square brackets.

The call to @user.tasks.where(...) returns a list of Tasks, which we then store in the @tasks variable.

Adding a Template for the Search Results

Now we need a template file to display the contents of the @tasks search results. This is a straightforward task because we've already created similar templates for the index and incomplete actions. In the text editor, start a new file and type the code as shown:

```html
<h1>Search</h1>
<% render :partial => "tasktable", locals => {:tasks => @tasks} %>
<% render :partial => "copyright" %>
<br />
<%= link_to 'New Task', new_task_path %>
```

Save it as app/views/tasks/search.html.erb. This probably looks familiar—it's almost identical to the index.html.erb template:

```html
<h1>Listing tasks</h1>
<% render :partial => "tasktable", locals => {:tasks => @tasks} %>
<% render :partial => "copyright" %>
<br />
<%= link_to 'New Task', new_task_path %>
```

Both templates display the contents of the @tasks variable by calling the _tasktable.html.erb partial template. The only difference between index.html.erb and search.html.erb is the title in the <h1>...</h1> tag.

Now we can try out the new search code by opening a browser at http://login-id.oreillystudent.com/ostapp/tasks/search. If you're not already logged in, you may be asked to do so. The page looks like this:
When you enter the search URL, Rails finds a matching route in the `routes.rb` file and then runs the search action on the `Tasks` controller. The search action method inside the tasks controller finds all of the records for the current user that contain the word `chemical` in the name or description. These are stored in the `@tasks` variable, which is then displayed with the `search.html.erb` template. Now, we need to modify the search so that instead of searching for the selected word ("CHEMICAL"), it will search for a word or phrase provided by the user.

**Passing Data to the Application**

There are two ways that a browser can send data to a web server—as part of the URL, or in the hidden part of the request. A request is when a browser contacts a web server. Let's take a look at how to send the search terms in the URL. We've encountered this type of request before—it's called a `GET` request. Let's modify the application so that if the user makes the call to `http://login-id.oreillystudent.com/ostapp/tasks/search?terms=charts`, the application will search all of the user's tasks for the word `charts`. The information at the end of the URL after the ? character is called the query. The query can contain a number of parameters. A parameter is a name with an associated value. In this case we have the `terms` parameter, with the value `charts`.

You don't have to use the name `terms` for the search value. You can name parameters in whatever way is meaningful to you, so long as you are consistent throughout your code, and you avoid using characters that have specific URL meanings (such as "?" or "&").

In our controller code, instead of searching for the fixed string `CHEMICAL`, we need to search for the string passed in as the `terms` parameter:

```ruby
def search
  terms = 'CHEMICAL'
  ...
end
```

Rails makes this value available with the `params` object. The `params` object allows you to find a particular parameter value by specifying the parameter's name in brackets, like this:
The name of the parameter, as with all names in Rails, is specified with a symbol. This one begins with a colon—`:terms`. The expression `params[:terms]` evaluates to the string value passed in the URL—in the above example URL, this is "charts". The existing code looks for an uppercase string ("CHEMICAL"), so before we replace that static value with the parameter value, we will need to convert the parameter value to uppercase. In Rails we can find the uppercase value of any string by adding `.upcase` to the end of it. For example, if we want an uppercase version of the `terms` parameter, we can find it with this expression:

`params[:terms].upcase`

Replace the static value in `tasks_controller.rb` with the value of the `terms` variable, as shown:

```ruby
class TasksController < ApplicationController
def search
  terms = 'CHEMICAL'
  terms = params[:terms].upcase
  @tasks = @user.tasks.where(["upper(name) like ? OR upper(description) like ?", "%#{terms}%", "%#{terms}%"])
  respond_to do |format|
    format.html
    format.xml { render :xml => @tasks }
  end
end
...
```

Save it and go to `http://login-id.oreillystudent.com/ostapp/tasks/search?terms=departure`; you’re results will look like this:
If you mistype the name of your "terms" parameter in the code, you'll see an error message when you try to perform a search. If you try to read the wrong parameter from `params[...]`, Ruby won't be able to return a meaningful value and your application will generate an array when it tries to convert the unusable value to uppercase.

If you have a different set of task data, your search results will vary. To make sure that the code is really picking up the search terms from the URL, try some different values:

Adding a Form

Now that we've changed the code to include the search terms in the URL, we can create a form for the user to specify which search terms they want. If we create an HTML form with a `terms` text field that sends the form to the `../tasks/search` URL, our code will search for whatever string the user entered in the form.

It would be useful if the search form was available on each one of the tasks pages. We can add to required HTML code to every task page by adding the code to the tasks `layout` file at `tasks.html.erb`:
Save the `tasks.html.erb` file and then go to any of the task pages (it doesn’t matter which one; they all contain the search form now):

![Screen capture of the task page with the search form displayed.](image)

The form accepts a text value called `terms` and then sends the form to `/ostapp/tasks/search`. Because the form's
method is GET, when the form is sent to the server, the value of the parameter is added to the URL.

Enter some text in the search box, then click Search. The browser sends the form to the server, and generates the search results:

Absolute vs. Relative Paths

Our site now has a fully functional search form. Let's take a closer look at the HTML for the form:

```html
<form action="/ostapp/tasks/search" method="get">
  <label for="terms">Search for:</label>
  <input id="terms" name="terms" type="text">
  <input type="submit" value="Search">
</form>
```

The form uses an absolute path for the action—that means it specifies the full path from the root of the website URL to the location of the search code. This allows the search form to be called from anywhere.

For example, if the form had set the action to the relative path "search" like this:
The form would still work great from the main tasks Index page, http://login-id.oreillystudent.com/ostapp/tasks, because from there the form would send its data to http://login-id.oreillystudent.com/ostapp/tasks/search.

But what if the user calls the search page from a single task page, like http://login-id.oreillystudent.com/ostapp/tasks/22?

From that URL, a relative search path would lead to http://login-id.oreillystudent.com/ostapp/tasks/22/search.

It’s really important that our search form has an absolute path. Without it, the search simply will not work on some pages.

However, the big downside to setting an absolute path is that we lose flexibility. We’re tying ourselves down to a particular URL structure. Your Rails application might be deployed on different web servers in different ways. On the O’Reilly School web servers, everything is published under the /ostapp path; all of your URLs begin with http://login-id.oreillystudent.com/ostapp/. But what if you gave your application to another website? They might deploy it on http://www.megacorp.com/productivity/taskapp/... That means your search feature would live on http://www.megacorp.com/productivity/taskapp/tasks/search, which would break all of your code immediately.

So we have a conflict:

- We need to have an absolute path for the search form so that it will work on all pages.
- We don’t want to store the absolute path in the form because it will break the application if it’s deployed elsewhere.

Does any of this sound familiar? We had a similar problem in an earlier lab when we wanted to create links between pages. Instead of creating a link in the Tasks layout:

```
<a href="/ostapp/tasks/incomplete">Incomplete tasks</a>
```

We used a helper function:

```
<%= link_to "Incomplete tasks", :tasks_incomplete %>
```

Instead of creating a hyperlink with raw HTML, we used a helper function to generate the HTML for us. The helper function generated an absolute path for the link, and the absolute path would work, even if the application was deployed in a different way on a different machine.

This is just one reason that Rails coders prefer to use helper functions rather than raw HTML. In addition, helper functions:
• have extra functionality that can be called on without requiring lots of extra code.
• are usually smaller than the HTML that they generate.
• are easier to read.
• make typos easier to spot. For example, in the link_to helper above, misspelling tasks_incomplete would have generated an error for the page, but mistyping /ostapp/tasks/incomplete wouldn’t.

For all of these reasons, it’s better to generate our form in a helper function than to write it in raw HTML.

Using a Form Helper Function

We need a form-helper that generates HTML code equivalent to this HTML:

```
<form action="/ostapp/tasks/search" method="get">
<label for="terms">Search for: </label>
<input id="terms" name="terms" type="text">
<input type="submit" value="Search">
</form>
```

Here’s how to write the equivalent code with helpers:

```
<%= form_tag({:action=> "search"}, :method=>"GET") do %>
  <%= label_tag(:terms, "Search for:") %>
  <%= text_field_tag(:terms) %>
  <%= submit_tag("Search") %>
<% end %>
```

We’re using not one, but several helpers. Each helper generates a specific type of HTML tag. HTML forms contain fields; in HTML we place the fields between the <form> and </form> tags. Field helpers are included inside form helpers by using do and end markers. In Ruby, the code between do and end markers is called a block. So the code above creates a form with the form_tag helper and then passes it a block of fields created by calling the label_tag, text_field_tag, and submit_tag helpers. Here’s a list of the helpers we just used, along with a description of each:

- **form_tag** is a helper to generate <form>...</form> tags. By passing in values for the :action and :method, the form_tag can create an absolute path for the correct code. The form_tag is passed a block of code between do and end markers.
- **label_tag** creates a <label/> for a form field. It’s given a symbol that represents the field it is labeling (:terms) and a string that specifies the text of the label.
- **text_field_tag** generates a <input type="text".../> field. It is given a symbol that represents its field value (:terms).
- **submit_tag** creates a <input type="submit".../> tag. You can give submit_tag a string to replaces the default Submit text on the button.

Modify tasks.html.erb as shown:
Signed in as: <span style="font-weight: bold;">%= @user.full_name %></span>

<form action="/ostapp/tasks/search" method="get">
  <label for="terms">Search for:</label>
  <input id="terms" name="terms" type="text">
  <input type="submit" value="Search">
</form>

<% form_tag({:action => "search"}, :method => "GET") do %>
  <label for="terms">Search for:</label>
  <input id="terms" name="terms" type="text">
  <input type="submit" value="Search">
<% end %>
</div>

Save it and do a search on one of your tasks pages:
Okay—the search form looks the same and works the same way as before. View the source code for the page and see if it looks like the same kind of HTML is being produced:
The HTML code looks similar to the code you wrote by hand, but it's actually slightly more complicated because it includes information about how the character set of the form is encoded. These kinds of details can be tedious to create by hand, but you get them for free if you use a form helper!

**What Just Happened?**

In this lesson, we learned:

- **params** gives you access to the parameters sent by a form.
  - To access a parameter named `terms`, use `params[:terms]`.
  - Adding `.upcase` to the end of a string gives you an uppercase version of the string.
  - Absolute paths allow forms to be used by any page in your application.
  - Absolute paths may cause problems if you deploy your application to a different server.
  - Form helpers avoid path problems by generating absolute paths at runtime.
  - The `form_tag` helper creates a form tag with smart paths that will match actions in your application.
  - `label_tag`, `text_tag`, and `submit_tag` can generate labels and fields within a form.

Wow. We are heading into the home stretch! Where does the time go? Work through the homework and I'll see you soon in the next and final lesson!
Lesson Objectives
When you complete this lesson, you will be able to:

- add fields to your application.

Adding Fields to Your Application
No application remains the same for long. As requirements evolve, you'll often need new data, which means you'll need to add fields to your applications. In this lesson, we'll add a **Priority** field to the system which will allow users to prioritize their tasks. In doing that, we'll need to revise many of the skills we've learned throughout the course.

The **priority** field will contain a value from 1 to 10, or the user might choose to leave the field blank.

If we want to add this field to the application, what do we need to do?

Add the Field to the Database
Before we do anything else, we need to add the priority field to the **tasks** table in the database. We've added a field to this database before, back when we created a relationship between the **tasks** and **users** tables.

To modify the structure of the database, for instance to add a column to a table, we create a migration. If we generate a migration with an appropriate name, Rails will be able to infer the change we want to make. Open a Terminal session,
change into the application directory, then create the migration, like this:

```
INTERACTIVE SESSION:

cold1:~$ cd railsapps/ostapp/
cold1:~/railsapps/ostapp$ rails generate migration AddPriorityColumnToTasks priority:integer
   invoke  active_record
cold1:~/railsapps/ostapp$
```

The name tells Rails exactly what to do, so you don’t have to write any additional code yourself.

Add...  ...a column named priority...  ...to the tasks table.

AddPriorityColumnToTasks

At the end of the migration command, `priority:integer` tells Rails that we want this new column to be an integer.

The `rails generate migration` command, creates the migration script in the `db/migrate` directory. The name of the migration script will vary, because it begins with a timestamp and ends with `_add_priority_column_to_tasks`. Open the migration in the editor; it will look like this:

```
OBSERVE:

class AddPriorityColumnToTasks < ActiveRecord::Migration
def self.up
  add_column :tasks, :priority, :integer
end

def self.down
  remove_column :tasks, :priority
end
end
```

The `self.up` function inside the script will create the column on the table, but before that can happen, we need to run the script, using the `rake` command that comes with Rails. Remember—if we’re generating code, we use the `rails` command, but if we’re managing our environment, we almost always use the `rake` command.

```
INTERACTIVE SESSION:

cold1:~/railsapps/ostapp$ rake db:migrate
(in /users/dgriffit1/railsapps/ostapp)
  == AddPriorityColumnToTasks: migrating ===============================
  -- add_column(:tasks, :priority, :integer)
    -> 0.0080s
  == AddPriorityColumnToTasks: migrated (0.0090s) ===============================
cold1:~/railsapps/ostapp$
```

To make sure that the column was added to the table, open the Rails console and take a look at the structure of the `task` object. We’ve used the rails console throughout the course whenever we wanted to try out code in our application. The Rails console is an extension of the basic Ruby `Interactive Ruby tool (irb)`, but unlike `irb`, the Rails console also has access to all of the code in our application.
By typing `Task` in the console, we can examine the structure of the task objects. The `priority` attribute is listed at the end of the `Task` structure.

So, we’ve added the priority column to the table, now what?

### Add the Field to the View

Every Rails app splits into three parts: the model, the controller, and the view. The model is the part of the application that manages the data. When we add a `priority` to the task objects, we make a change to the model. The controller handles the basic flow of the application; we’ve spent a lot of time modifying the controller code.

View code manages the application’s appearance. We’ve added the priority column to the database, but if we go to the web pages in the application, there’s still no sign of it.

It's not there because we haven't changed any of the view code to display the new priority attribute.
When we first created the application using scaffolding, the view code was generated for us automatically. You might wonder why Rails didn't also add the new attribute to the view code when we created the migration. It doesn't do that because the view code may well have been modified since it was created by Rails. Once Rails generates code, it never goes back to modify it. Once the code is generated, it's up to you, the programmer, to manage everything from then on.

First, let's add the Priority field to the **New Task** and **Editing Task** forms. Open the `app/views/tasks/edit.html.erb` template file; the form is called in from a partial template named `form`:

```erb
<% if @task.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(@task.errors.count, "error") %> prohibited this task from being saved:</h2>
    <ul>
      <% @task.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
    </ul>
  </div>
<% end %>

<%= form_for(@task) do |f| %>
  <%= f.label :name %><br />
  <%= f.text_field :name %>

  <%= f.label :description %><br />
  <%= f.text_area :description %>

  <%= f.label :duration %><br />
  <%= f.text_field :duration %>

  <%= f.label :due_date %><br />
  <%= f.date_select :due_date %>

  <%= f.label :complete %><br />
  <%= f.check_box :complete %>

  <%= f.submit %>
<% end %>
```

A partial page template is a small template file that can be called by another template, using the `render` function. The `render 'form'` call will include the contents of the `_form.html.erb` file (all partial page template files begin with the underscore character `_` and end with the extension `.html.erb`). Open the `app/views/tasks/_form.html.erb` partial file:

```erb
<% if @task.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(@task.errors.count, "error") %> prohibited this task from being saved:</h2>
    <ul>
      <% @task.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
    </ul>
  </div>
<% end %>

<%= f.label :name %><br />
<%= f.text_field :name %>

<%= f.label :description %><br />
<%= f.text_area :description %>

<%= f.label :duration %><br />
<%= f.text_field :duration %>

<%= f.label :due_date %><br />
<%= f.date_select :due_date %>

<%= f.label :complete %><br />
<%= f.check_box :complete %>

<%= f.submit %>
```

This partial contains a *form*. When we add the search function, we create a form that consists of a text field—the data in the form doesn't correspond to any particular piece of information in the database. *This form is a little different. It's*
used to edit the task object stored in the `@task` variable. Whereas the search form could simply contain a blank text field, the form contained in the `_form.html.erb` file needs to display the existing attributes of the `@task` object.

Because the form needs to include the existing attributes of the `@task` object, it has to be generated with some slightly different form helper functions. When we created the search form, we used a form helper called `form_tag`:

```
<%= form_tag path: '/tasks', method: 'put' do %>
  
  <%= label_tag :name, 'Name' %>
  <%= text_field_tag :name, @task.name %>

  <%= label_tag :description, 'Description' %>
  <%= text_area_tag :description, @task.description %>

  <%= label_tag :duration, 'Duration' %>
  <%= text_field_tag :duration, @task.duration %>

  <%= label_tag :due_date, 'Due date' %>
  <%= date_select :due_date, :year => 2014, :month => 11, :day => 18 %>

  <%= label_tag :complete, 'Complete' %>
  <%= check_box_tag :complete, @task.complete %>

  <%= submit_tag 'Save' %>
<% end %>
```
In order to create a form that's based on a particular task object, we need to use the `form_for(@task)` helper. This helper function examines the `Task` object it's been passed, and uses the task's attributes to set default values for the fields in the form. The `form_for(...)` helper creates a form object named `f` that can generate form fields for each of the attributes of the `@task` object. For example, this is how the Description field is created:

```erb
<%= form_for(@task) do |f| %>
  ...
  <div class="field">
    <%= f.label :description %><br />
    <%= f.text_field :description %>
  </div>
<% end %>
```

The `form_for` is given a `@task` object, creating the `f` form object (the `|` bars indicate that `f` was created by the `form_for` call). The form object creates the label and the text_field for the `@task`'s `description` attribute.

That means that this form works in a way similar to our search form—the main difference is that this form will set default values for each of the fields based on the values of the attributes of the `@task` variable.

We've added a priority attribute to each task object, so add it to `app/views/tasks/_form.html.erb` like this:
That extra code generates a label and a text field for the `priority` attribute. If there’s an existing priority value on the task being edited, the text field is set to that value.

- Save the `_form.html.erb` partial and then refresh the Editing task page. The priority field now appears on the page:
Next, we'll need to add the priority field to the *New task* form. But wait—the *new.html.erb* file calls the same *form* partial as *edit.html.erb*:

- **Priority** field is here now.
That means that we don't have to do anything at all to the New task page—it just works!
Try out the form by creating a task with a priority as shown:

The same partial generates the Editing tasks form and this one...

...so the Priority field is already here!
Adding the Priority to the Show Task Page

The user can now enter a priority against each task, but if they look at a single task, the priority doesn't appear:
We can fix that. Modify the `show.html.erb` template to include the `priority` value as shown:
Save it and view the new task. Now the Priority appears:
Validating the Priority Attribute

We looked at validation in an earlier lesson. Let's see how that might be useful here. We want the priority to be a number in the range 1 - 10. What happens if a user enters something else?
Rails model code manages everything to do with data, so if we want to validate something, we'll add the code to the `app/models/task.rb` model script. We already have quite a few validations in there:

```ruby
class Task < ActiveRecord::Base
  belongs_to :user
  validates :description, :name, :duration, :presence => true
  validates :name, :uniqueness => true
  validates_numericality_of :duration, {:greater_than_or_equal_to => 1, :less_than_or_equal_to => 240}
  validate :due_in_future, :on=>:create
  def due_in_future
    if due_date < Date.today then
      errors.add(:due_date, "cannot be in the past")
    end
  end
end
```

If we want to add validation to the **priority**, we'd put it in this script. These are the rules we need to apply to the priority:

- It must be a number.
- It must be in the range 1-10.
- It can be left blank.

We perform validation on the `duration` attribute; let's create a similar validation for the `priority` attribute. Modify the file...
as shown:

**CODE TO TYPE:**

```ruby
class Task < ActiveRecord::Base
  belongs_to :user
  validates :description, :name, :duration, :presence => true
  validates :name, :uniqueness => true
  validates_numericality_of :duration, {:greater_than_or_equal_to => 1, :less_than_or_equal_to => 240}
  validates_numericality_of :priority, {:greater_than_or_equal_to => 1, :less_than_or_equal_to => 10}
  validate :due_in_future, :on=>:create
  def due_in_future
    if due_date < Date.today then
      errors.add(:due_date, "cannot be in the past")
    end
  end
end
```

- Save it and refresh the page. You can no longer enter a large number or a non-numeric value in the Priority field:
So, what happens if you try to leave the priority field blank?:

- Priority must be less than or equal to 10
The validation rule says that the priority must be numeric—a blank space is not numeric. So, we have to modify our validation slightly and use an option we haven't seen before—:allow_nil. Modify task.rb again, as shown:
CODE TO TYPE:

```ruby
class Task < ActiveRecord::Base
  belongs_to :user
  validates :description, :name, :duration, :presence => true
  validates :name, :uniqueness => true
  validates_numericality_of :duration, {:greater_than_or_equal_to => 1, :less_than_or_equal_to => 240}
  validates_numericality_of :priority, {:greater_than_or_equal_to => 1, :less_than_or_equal_to => 10, :allow_nil => true}
  validate :due_in_future, :on=>:create
  def due_in_future
    if due_date < Date.today then
      errors.add(:due_date, "cannot be in the past")
    end
  end
end
```

Again, `nil` is a special value in Ruby which means "no value". The `:allow_nil` option means the user can leave the value blank. Save `task.rb` again, refresh the page, and then try saving a task with a blank priority:

![Task was successfully updated.](image)

**What Just Happened?**

In this lesson, we tied together a lot of the skills we've acquired throughout the course, and learned a few new ones:
Rails code is split into model, view, and controller code.
- Database code is model code.
- You can add a column to a database table using a migration script.
- Rails can infer what the migration has to do by the name you give it.
- Use `rake db:migrate` to run the migration.
- If you add a column to the database, the view code will not automatically display it.
- The main form for creating/editing is in the `_form.html.erb` partial template file.
- Forms that are based on existing data use the `form_for` helper.
- The controller will read the existing task from the database and store it in the `@task` variable.
- `form_for` creates a form with default values based on attribute values.
- Validation is performed by model code.
- The `validates_numericality_of` validator has an `:allow_nil` option if a value is not mandatory.

We've arrived at the end of the lesson, and the end of the course. Thank you for immersing yourself in our Rails course, it's been a real pleasure! Congratulations on the new skills you've earned so far, and good luck with your final project!