Extracted from:

Rails for .NET Developers

```
<%= f.collection_select :origin, airports, :code, :city, {},
                        { :size => 10 } %>
```

Other Data-Bound Controls

The form_for block object can help build other controls as well:

- check_box binds a checkbox to a boolean attribute.
- password_field binds a textbox to a string attribute but masks the input.
- hidden_field creates a hidden value that will be sent to the application when the form is submitted.
- radio_button binds a radio button to a model attribute. Multiple radio buttons for the same attribute are considered to be a radio button group.
- text_area is similar to text_field but allows multiple lines of input.

Each of these helper methods "bind" HTML controls to data columns simply because they'll use generate the proper name values in the resulting HTML code. Your create or update actions in your controller perform the heavy lifting by calling the create or update_attributes method on your ActiveRecord model.

We've toured the basics of creating forms in Rails. The following sections will take us into two other aspects of view management in Rails: layouts that help us provide a consistent look and feel among all of our pages; and partials, which will help us avoid repeating the same code from one view to the next.

8.2 Using Layouts Instead of Master Pages

ASP.NET 2.0 introduced the concept of *master pages* so that site-wide logic and HTML can be applied across all pages that share the same structure and layout.

In Rails, we use *layouts* to apply a common HTML structure to our entire application or for specific controllers. Since layouts in Rails are normal view templates that can contain embedded Ruby code, layouts provide one way for us to dynamically "theme" our site. We can determine styles and page structure dynamically based on environment settings, user profiles, configuration files, time of day, data in our database, or anything else we can dream of.

Layouts also serve a second purpose. Ruby developers like to keep their code DRY. Refactoring reduces the number of lines of application code,

isolates faulty code to a single spot so that a fix can be performed in just one place, and helps maintain readable code as the size of the application grows larger. We want to treat our view templates as first-class code citizens in our application and strive to keep our view code as DRY as possible as well. Layouts help us refactor common code out of our views so that we don't need to repeat the same code in each view template.

Master Pages in ASP.NET

Master pages help enforce a standard structure or layout for every page on the site. Master pages have the file extension .master and use a @ Master directive instead of the usual @ Page. This placeholder would be replaced at runtime with the content of the current page.

Master pages are similar to regular .aspx files, but at some point in the code there must be a ContentPlaceHolder control. As its name suggests, the placeholder is replaced at runtime by an actual .aspx page, wrapped by the outer master layout. The page being wrapped is often referred to as the *content* page.

Connecting a content page with its master page involves setting the MasterPageFile property in the content page's @ Page directive. We usually do this from the Add Item Wizard in Visual Studio, but it can also be done by hand later. And although they are similar to regular .aspx pages, content pages do not contain HTML tags such as <html>, <body>, or even <form>.

The motivation behind master pages is to help ease the development of sites where pages share a common layout so that the common elements can be specified once in the master page. Changes to the common structure can be done in the master page, and all pages that use that master page will automatically use the new structure. At runtime, ASP.NET will merge the content page into the surrounding master page.

Rails also provides a mechanism for sharing common structure among pages by introducing the notion of a *layout template*. If you've been using master pages in your ASP.NET projects, using layouts will feel natural and easier to use than master pages.

Using Layouts in Rails

Layouts differ from master pages in a few ways:

- Layouts are either controller-wide or site-wide.
- Instead of a "placeholder" tag in the template, layouts use the Ruby keyword yield.
- Views are automatically merged into the controller or site-wide layout. No separate step is needed to connect a view to its layout.

Layouts are normal embedded Ruby templates to generate HTML, with a twist. Layouts will be "wrapped around" your more specific, action-based view templates. The layout combines with an action-specific template to generate the complete HTML for a web page. Using layout files is entirely optional, but most projects benefit from using even simple layouts.

You can incorporate layouts into your views in three ways:

- A controller-specific layout will automatically be used if it exists. Each controller can contribute a file into the app/views/layouts/ directory if it follows the specific naming convention of controller.html.erb. For example, app/views/layouts/flights.html.erb is the layout template that will wrap around all templates served by the FlightsController.
- A site-wide default layout will be used if a controller-specific layout does not exist. The site-wide layout file must be named application.html.erb. If the site-wide layout and a controller layout both exist, the controller layout takes priority. This means you can define a site-wide layout but override it for specific controllers as needed.
- Specify the layout file in your controller's Ruby code. You can explicitly call the layout class method in your controller to define the layout file you'd like to use for your controller:

```
class FlightsController < ApplicationController

  # Use app/views/layout/my_special_layout
  # for every action rendered by this controller

  layout "my_special_layout"
end
```

If needed, you can override the layout setting for a specific action:

```
class FlightsController < ApplicationController

  # Use app/views/layouts/my_special_layout.html.erb
  # only for the index action.
  # All other actions will use flights.html.erb
  # or application.html.erb if they exist

  layout "my_special_layout", :only => :index
end
```

or if you choose, specify the layout during an explicit render call:

```
class FlightsController < ApplicationController

  def index
    # render index.html.erb wrapped by
    # app/views/layouts/my_special_layout.html.erb

    render :action => :index, :layout => 'my_special_layout'
  end
end
```

Let's see how we can put layouts to practical use.

Using Layouts for Controller-wide Themes

Layout templates are just like normal action view templates, but they wrap their contents around action view templates. When no specific layout directive is found in the controller code, the controller-specific layout found in the app/views/layouts directory will automatically wrap around every action rendered by that controller.

Inside the layout, you decide exactly where the action template's code is to be inserted by calling yield, as shown in the flights.html.erb layout.

Ruby Download view/layouts/flights.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
  <title>Flights: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>

<p style="color: green"><%= flash[:notice] %></p>
```

```
    <%= yield  %>

  </body>
  </html>
```

We can modify the output rendering of every action from the FlightsController by enhancing this template. Let's add a nice header and footer to our views:

Ruby

Download view/layouts/flights_enhanced.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
  <title>Flights: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>

    <h1 id="flights">Flight Administration</h1>

    <p style="color: green"><%= flash[:notice] %></p>

    <div id="action">
      <%= yield  %>
    </div>

</body>
</html>
```

We will now tweak the scaffold.css code a bit:

Download view/layouts/scaffold.css

```
#header { border-bottom: solid 2px green; padding: 20px;}
#action { margin-left: 100px;}
```

To see the layout wrap around every action of the FlightsController, go to http://localhost:3000/flights, and use the scaffold-generated pages. You'll see that all the pages automatically inherit the same heading and margin styles because they are now rendered from within the layout.

Creating a Site-wide Layout

Often we want to enforce a site-wide structural layout. A site-wide layout is implemented by simply naming the template application.html.erb. Rails refers to this file as the *application layout*, and Rails will use it for every controller that does not specify its own layout.

To demonstrate, let's convert our newly made flights layout into a site-wide, or application, layout:

1. Rename flights.html.erb to application.html.erb in the app/views/layouts directory.
2. Edit the title tag as desired, and let's change the `h1` tag as well:

Ruby Download view/layouts/application.html.erb

```
<h1 id="header">Airline Management App</h1>
```

3. Delete the existing controller-specific layouts.

Now, every page will be rendered with our site-wide layout.

Controllers can override the site-wide layout, but you can't have it both ways: actions can be wrapped by their controller's layout or the application layout, but not both.

8.3 Creating Partials Instead of User Controls

Websites are often composed of common elements. We have already seen how layouts help promote a common structure without needlessly repeating code inside each view template and how layouts can be seen as a kind of equivalent to .NET master pages. Although layouts are useful for providing overall structure and "wrapper" content, we often want to reuse small components or self-contained sections across many pages. .NET provides this facility with user controls, which help encapsulate appearance and behavior together as a unit. In this section, we will explore *partials*, which is another facility provided by Rails to help us reuse small sections or "components" of a web page in other pages as well.

User Controls in ASP.NET

User controls and server controls are powerful features of ASP.NET. They provide several important benefits to ASP.NET projects that use them:

- They help encapsulate often-used UI and behavior in one place.
- They help factor common functionality from individual pages.
- They can be derived from DataBoundControl to bind the control's UI to the database.
- They can inherit from other user controls, leveraging existing code and again helping reduce code redundancy.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Rails for .NET Developers' Home Page
http://pragprog.com/titles/cerailn
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/cerailn.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com