

Extracted from:

# Rails for .NET Developers

This PDF file contains pages extracted from Rails for .NET Developers, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

**Note:** This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2008 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

# CRUD with ActiveRecord

---

ActiveRecord is the part of Rails that's responsible for talking with the database of our application. Whether that database is MySQL, SQL Server, SQLite, Postgres, or one of the myriad of other RDBMSs that Rails supports, ActiveRecord allows us to tell it what to do in a single language, Ruby.

In this chapter, we'll explore the sweet spot of Rails—the creating, reading, updating, and deleting (CRUD) of data—using ActiveRecord. We've already talked about some of the conventions that we'll need to follow to take full advantage of ActiveRecord; now, we'll put this knowledge to the test by using Rails to do a lot of the same things we're used to doing in .NET. In addition, we'll be concentrating much more on the capabilities of ActiveRecord and the model side of things and only minimally on the controller and view parts of the Rails world. We'll take a much closer look at the controller and view in the next chapter.

## 6.1 Displaying a Grid of Data in a Table

One of the features that you'll see in almost any web application is a collection of records in a database displayed in a human-readable grid/table format. Let's say we'd like display a simple table that shows all passengers in our flight system from our `passengers` table.

### How You Might Approach It in .NET

There's no built-in ActiveRecord-like ORM mapper in .NET, although several open source and commercial packages are similar in style and function. For the most part, though, in out-of-the-box ASP.NET WebForms, you're looking at SQL (whether it's in a stored procedure or dropping a SQL statement into code) or LINQ in order to get an object

that contains your data and then displaying it with a control like the GridView. The following illustrates such an approach:

.NET

Download crud/Passengers.aspx

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs"
    Inherits="_Default" %>

<!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">
<head runat="server">
    <title>Passengers</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:SqlDataSource ID="SqlDataSource1" runat="server"
            ConnectionString="<%$ ConnectionStrings:ConnectionString %>"
            SelectCommand="SELECT * FROM passengers"></asp:SqlDataSource>
        <asp:GridView ID="GridView1" runat="server"
            AutoGenerateColumns="False" DataKeyNames="id"
            DataSourceID="SqlDataSource1">
            <Columns>
                <asp:BoundField DataField="name" HeaderText="name"
                    SortExpression="name" />
                <asp:BoundField DataField="address" HeaderText="address"
                    SortExpression="address" />
                <asp:BoundField DataField="seating_preference"
                    HeaderText="seating_preference"
                    SortExpression="seating_preference" />
            </Columns>
        </asp:GridView>
    </form>
</body>
</html>
```

Here, we create a simple GridView that contains our data by binding the control to a SqlDataSource object that queries for all passengers. Within the GridView definition, we declaratively let the control know which columns we want to display. When we start our application up, this code will translate into HTML for display in the web browser.

## The Rails Way

It's pretty trivial to create the same interface using Rails' scaffolding, as we did in the previous chapter. But we're going to do everything by hand this time so we can dig a little deeper into how Rails works behind the scenes.

As we've already learned, three parts are involved in building the same read-only view of a data collection as the ASP.NET GridView approach: the model, the view, and the controller. Let's quickly use a generator to get all the files we need before exploring our application further.

```
c:\dev\flight> ruby script\generate resource passenger
exists app/models/
exists app/controllers/
exists app/helpers/
create app/views/passengers
exists test/functional/
exists test/unit/
dependency model
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/passenger.rb
create test/unit/passenger_test.rb
create test/fixtures/passengers.yml
exists db/migrate
create db/migrate/20080722201710_create_passengers.rb
create app/controllers/passengers_controller.rb
create test/functional/passengers_controller_test.rb
create app/helpers/passengers_helper.rb
route map.resources :passengers
```

Now that we have the files we need, let's make a few edits to the generated code to get our application working the way we want it.

## The Passenger Model

We'll concentrate on the model layer first. Notice that the generator created a migration file, `db/migrate/20080722201710_create_passengers.rb`. Let's modify this migration to reflect the schema we'd like to create for the passengers table, where each passenger will have a name, address, and seat preference.

Ruby

[Download](#) crud/20080722201710\_create\_passengers.rb

```
class CreatePassengers < ActiveRecord::Migration
  def self.up
    create_table :passengers do |t|
      t.string :name, :address, :seat_preference
      t.timestamps
    end
  end

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

Now we'll run the migration:

```
c:\dev\flight> rake db:migrate
(in c:\dev\flight)
== 20080722201710 CreatePassengers: migrating =====
-- create_table(:passengers)
   -> 0.0026s
== 20080722201710 CreatePassengers: migrated (0.0029s) =====
```

Great. The passengers table is now created based on the information we've provided in the migration file. With no SQL. Just Ruby.

### The Rails Console

Now that we've created our table, it's a good time to mention that Rails ships with a handy utility called console that is a lot like irb, except that it runs within the context of our web application. This means that, in addition to evaluating Ruby interactively, you can also do things like manipulate your application's database, make simulated requests, and execute other Rails-specific commands that wouldn't otherwise be available with vanilla irb. Let's fire it up now:

```
c:\dev\flight> ruby script\console
Loading development environment (Rails 2.1.0)
>>
```

From the console, we're able to learn a lot about the capabilities of the ActiveRecord library. Let's use it now to create some sample records in our passengers table:

```
>> Passenger.create(:name => 'John Doe', :address => '123 Main St',
:seat_preference => 'Aisle')
=> #<Passenger id: 1, name: "John Doe", address: "123 Main St",
seat_preference: "Aisle", created_at: "2008-01-15 15:49:24",
updated_at: "2008-01-15 15:49:24">
```

The create method is a *class* method of the class Passenger that accepts a single parameter—a Hash where the keys are column names (as symbols) and the corresponding values. We could just easily do it the long way and instantiate a new Passenger object, assign the values we'd like, and call the save method:

```
>> passenger = Passenger.new
=> #<Passenger id: nil, name: nil, address: nil, seat_preference:
nil, created_at: nil, updated_at: nil>
>> passenger.name = 'Jane Doe'
=> "Jane Doe"
>> passenger.address = '123 Main St'
=> "123 Main St"
```

```
>> passenger.seat_preference = 'Window'
=> "Window"
>> passenger.save
=> true
```

Both approaches yield the same result: a new record gets created in the `passengers` table with the values you've specified for each column. Some developers like the one-line brevity of the `create` method, and others enjoy the clear intention expressed by the multiline approach. It's up to you to decide which of the styles you like better.

### It's Just SQL

By now, you have probably realized that ActiveRecord, as fantastic a library as it is, is not magic. All it really does is create SQL statements behind the scenes, with the added bonus of being completely database-agnostic. That is, it understands the various idiosyncrasies of various database engines and adjusts the generated SQL accordingly.

The closer your relationship with ActiveRecord, the more productive Rails developer you'll become. And the key to a deeper and more meaningful relationship with ActiveRecord is knowing exactly what SQL is being generated when you call methods like `create`. Fortunately, the raw SQL is exposed through the log file of a running Rails application, and furthermore, you can also examine it in the console. By default, the log output of any commands you execute in the console go straight to your development log (located at `log\development.log`), but you can issue a one-line command to override this and direct the log output to *standard output* instead. We'll do this now so that we can inspect the SQL that ActiveRecord creates quickly:

```
>> ActiveRecord::Base.logger = Logger.new(STDOUT)
=> #<Logger:0x105a608 @default_formatter=#<Logger::Formatter:0x105a5e0
  @datetime_format=nil>, @progname=nil,
  @logdev=#<Logger::LogDevice:0x105a590, @filename=nil,
  mutex=#<Logger::LogDevice::LogDeviceMutex:0x105a52c
  @mon_entering_queue=[], @mon_count=0, @mon_owner=nil,
  @mon_waiting_queue=[]>, @dev=#<IO:0x2e7d4>, @shift_size=nil,
  @shift_age=nil>, @level=0, @formatter=nil>
```

So when we create a new record, we see the underlying SQL right away:

```
>> passenger = Passenger.create(:name => 'Brian Eng', :address =>
'1060 West Addison', :seat_preference => 'Aisle')
Passenger Create (0.000887)  INSERT INTO passengers ("name",
"updated_at", "seat_preference", "address", "created_at")
VALUES('Brian Eng', '2008-01-15 16:13:18', 'Aisle', '1060 West
Addison', '2008-01-15 16:13:18')
```

```
=> #<Passenger id: 3, name: "Brian Eng", address: "1060 West Addison", seat_preference: "Aisle", created_at: "2008-01-15 16:13:18", updated_at: "2008-01-15 16:13:18">
```

One thing to note is that when a record gets created, the `created_at` and `updated_at` columns of the table automatically get filled in. Don't remember creating those columns? That's because you didn't. When we used the generator to create the migration for the `passengers` table, it inserts the `t.timestamps` line in there by default. That's a special method that creates the `created_at` and `updated_at` columns for us. These are special column names that ActiveRecord recognizes and automatically fills in for us when a row is created or updated, respectively.

Also note that the value returned from the `create` method is an instance of the `Passenger` class, which contains the methods `id`, `name`, `address`, `seat_preference`, `created_at`, and `updated_at`.

```
>> passenger.address
=> "1060 West Addison"
```

You might think that these methods were added in by the generator code as well, but if you take a peek at the `Passenger` class, you'll see that's not the case:

Ruby

[Download](#) crud/passenger.rb

```
class Passenger < ActiveRecord::Base
end
```

That's right—a completely empty class definition. ActiveRecord knows what columns you have in your database and automatically generates a method for each column under the covers. All you do is call it.

Now, let's explore a few more ActiveRecord methods from the console and see where it takes us. First we'll find and update a single record in the table using the `find` and `save` methods:

```
>> passenger = Passenger.find(1)
   Passenger Load (0.000601)  SELECT * FROM passengers WHERE
   (passengers."id" = 1)
=> #<Passenger id: 1, name: "John Doe", address: "123 Main St",
seat_preference: "Aisle", created_at: "2008-01-15 15:49:24",
updated_at: "2008-01-15 15:49:24">
>> passenger.address = '321 Main St'
=> "321 Main St"
>> passenger.save
   Passenger Update (0.000567)  UPDATE passengers SET "created_at"
   = '2008-01-15 15:49:24', "name" = 'John Doe', "seat_preference"
   = 'Aisle', "address" = '321 Main St', "updated_at" = '2008-01-15
   16:18:31' WHERE "id" = 1
=> true
```

We can also delete (destroy) a record with the destroy method:

```
>> Passenger.destroy(3)
  Passenger Load (0.000535)  SELECT * FROM passengers WHERE
    (passengers."id" = 3)
  Passenger Destroy (0.000573)  DELETE FROM passengers
  WHERE "id" = 3
=> #<Passenger id: 3, name: "Brian Eng", address: "1060 West
Addison", seat_preference: "Aisle", created_at: "2008-01-15
16:10:34", updated_at: "2008-01-15 16:10:34">
```

And finally, to get all the records in the table, we'll use the find method, passing in the :all option:

```
>> Passenger.find(:all)
  Passenger Load (0.000760)  SELECT * FROM passengers
=> [#<Passenger id: 1, name: "John Doe", address: "123 Main St",
seat_preference: "Aisle", created_at: "2008-01-15 15:49:24",
updated_at: "2008-01-15 15:49:24">, #<Passenger id: 2, name: "Jane
Doe", address: "123 Main St", seat_preference: "Window",
created_at: "2008-01-15 15:55:35", updated_at: "2008-01-15
16:06:10">, #<Passenger id: 3, name: "Brian Eng", address: "1060
West Addison", seat_preference: "Aisle", created_at: "2008-01-15
16:10:34", updated_at: "2008-01-15 16:10:34">]
```

Finding a data collection like this returns an Array of Passenger objects, which we will ultimately loop through to create our grid of data.

## Creating the Controller and View for Our Grid of Data

Now that we've added a couple of rows to our passengers table and we know how to get the data we want in order to display our table of passengers, let's hook up the controller and view code.

First, in the controller, we're going to add a method for the index action:

Ruby

Download [crud/passengers\\_controller.rb](#)

```
class PassengersController < ApplicationController
```

```
  ▶ def index
  ▶   @passengers = Passenger.find(:all)
  ▶ end
```

```
end
```

In the index action, we've defined an *instance variable* called @passengers that holds an Array of Passenger objects corresponding to all the records of the passengers table. Any instance variables we define in the controller are available for use in the rendered view, which by convention is the view file with the same name as the action located in the subdirectory named after the controller.



In this case, it's `app/views/passengers/index.html.erb`.

Ruby

Download `crud/index.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>Showing All Passengers</title>
</head>
<body>
  <table border="1" cellspacing="5" cellpadding="5">
    <tr>
      <th>Passenger Name</th>
      <th>Address</th>
      <th>Seat Preference</th>
    </tr>

    <% @passengers.each do |passenger| %>
    <tr>
      <td><%= passenger.name %></td>
      <td><%= passenger.address %></td>
      <td><%= passenger.seat_preference %></td>
    </tr>
    <% end %>

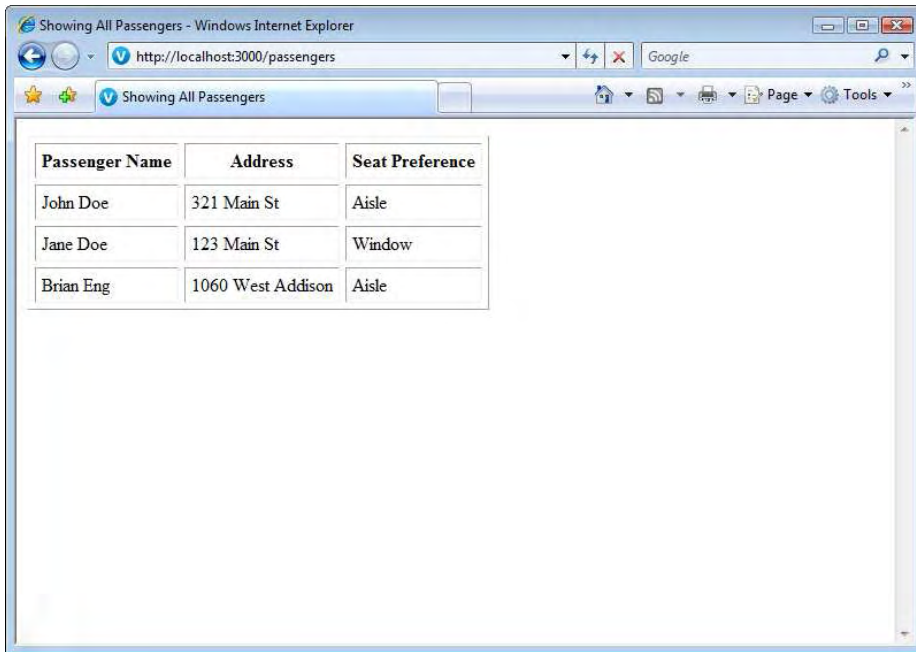
  </table>
</body>
</html>
```

Fire up your server using the `script/server` command, hop over to your browser, and you should see something like what's in Figure 6.1, on the next page.

Rails is definitely a bit “closer to metal” than ASP.NET WebForms. Instead of writing a SQL statement, dropping a control on a page, and letting the framework write the HTML for us, we're writing the HTML ourselves instead, delegating only the most granular data-driven details to the framework. This gives us the ultimate fine-grained control over the final output from the very beginning.

As shown by this example, Rails—unlike ASP.NET—doesn't really have the concept of GUI *controls* that you'd use to build a web form. Your only GUI “toolbox” in Rails is that of the native languages of the Web—HTML, JavaScript, and CSS.

A view, like this one, is simply HTML with some Ruby intermingled in there—otherwise known as Embedded Ruby (ERb). Here, we have



The screenshot shows a Windows Internet Explorer browser window titled "Showing All Passengers". The address bar contains "http://localhost:3000/passengers". The page content is a table with three columns: "Passenger Name", "Address", and "Seat Preference". The table contains three rows of data:

Passenger Name	Address	Seat Preference
John Doe	321 Main St	Aisle
Jane Doe	123 Main St	Window
Brian Eng	1060 West Addison	Aisle

---

Figure 6.1: Showing a table of all passengers

---

a basic HTML page with markup for the table of passengers we want to display. Any Ruby code between the `<% %>` symbols is going to be interpreted at runtime. So, we're dynamically looping through the contents of the `@passengersArray` and creating a table row for each record. Within each `tr` tag, we have three columns represented by the `td` tags, and within each `td` tag, we're again calling on Ruby to give back a value. When Ruby code lives within `<%= %>` tags, we're asking Ruby to actually write the result of the code within the tags out to the resulting HTML, instead of simply running the code. From an HTML generation perspective, Rails is similar in style and spirit to traditional ASP.

## 6.2 Sorting, Filtering, and Paging Data

Now that we can view our data using a simple HTML table, it's time to move on creating more interesting views of that data. For a very small data set, the previous example would work just fine. Most likely, however, we will be working with larger and more complex sets of data

# 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/cerain>

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/cerain](http://pragprog.com/titles/cerain).

## Contact Us

---

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