Chapter 5

The Depot Application

We could mess around all day hacking together simple test applications, but that won’t help us pay the bills. So let’s get our teeth into something meatier. Let’s create a web-based shopping cart application called Depot.

Does the world need another shopping cart application? Nope, but that hasn’t stopped hundreds of developers from writing one. Why should we be different?

More seriously, it turns out that our shopping cart will illustrate many of the features of Rails development. We’ll see how to create simple maintenance pages, link database tables, handle sessions, and create forms. Over the next eight chapters, we’ll also touch on peripheral topics such as unit testing, security, and page layout.

5.1 Incremental Development

We’ll be developing this application incrementally. We won’t attempt to specify everything before we start coding. Instead, we’ll work out enough of a specification to let us start and then immediately create some functionality. We’ll try ideas out, gather feedback, and continue on with another cycle of minidesign and development.

This style of coding isn’t always applicable. It requires close cooperation with the application’s users, because we want to gather feedback as we go along. We might make mistakes, or the client might discover they asked for one thing but really wanted something different. It doesn’t matter what the reason—the earlier we discover we’ve made a mistake, the less expensive it will be to fix that mistake. All in all, with this style of development there’s a lot of change as we go along.

Because of this, we need to use a toolset that doesn’t penalize us for changing our mind. If we decide we need to add a new column to a database table or change the navigation between pages, we need to be able to get in there
5.2 What Depot Does

Let’s start by jotting down an outline specification for the Depot application. We’ll look at the high-level use cases and sketch out the flow through the web pages. We’ll also try working out what data the application needs (acknowledging that our initial guesses will likely be wrong).

Use Cases

A use case is simply a statement about how some entity uses a system. Consultants invent these kinds of phrases to label things we’ve all known all along—it’s a perversion of business life that fancy words always cost more than plain ones, even though the plain ones are more valuable.

Depot’s use cases are simple (some would say tragically so). We start off by identifying two different roles or actors: the buyer and the seller.

The buyer uses Depot to browse the products we have to sell, select some to purchase, and supply the information needed to create an order.

The seller uses Depot to maintain a list of products to sell, to determine the orders that are awaiting shipping, and to mark orders as shipped. (The seller also uses Depot to make scads of money and retire to a tropical island, but that’s the subject of another book.)

For now, that’s all the detail we need. We could go into excruciating detail about “what it means to maintain products” and “what constitutes an order ready to ship,” but why bother? If there are details that aren’t obvious, we’ll discover them soon enough as we reveal successive iterations of our work to the customer.

Talking of getting feedback, let’s not forget to get some right now—let’s make sure our initial (admittedly sketchy) use cases are on the mark by asking our user. Assuming the use cases pass muster, let’s work out how the application will work from the perspectives of its various users.

Page Flow

I always like to have an idea of the main pages in my applications, and to understand roughly how users navigate between them. This early in the development, these page flows are likely to be incomplete, but they still help me focus on what needs doing and know how actions are sequenced.
Some folks like to mock up web application page flows using Photoshop, Word, or (shudder) HTML. I like using a pencil and paper. It’s quicker, and the customer gets to play too, grabbing the pencil and scribbling alterations right on the paper.

![Diagram of Buyer Pages Flow](image)

**Figure 5.1: Flow of Buyer Pages**

Figure 5.1 shows my first sketch of the buyer flow. It’s pretty traditional. The buyer sees a catalog page, from which he or she selects one product at a time. Each product selected gets added to the cart, and the cart is displayed after each selection. The buyer can continue shopping using the catalog pages or check out and buy the contents of the cart. During checkout we capture contact and payment details and then display a receipt page. We don’t yet know how we’re going to handle payment, so those details are fairly vague in the flow.

The seller flow, shown in Figure 5.2, on the next page, is also fairly simple. After logging in, the seller sees a menu letting her create or view a product or ship existing orders. Once viewing a product, the seller may optionally edit the product information or delete the product entirely.
The shipping option is very simplistic. It displays each order that has not yet been shipped, one order per page. The seller may choose to skip to the next, or may ship the order, using the information from the page as appropriate.

The shipping function is clearly not going to survive long in the real world, but shipping is also one of those areas where reality is often stranger than you might think. Overspecify it up front, and we’re likely to get it wrong. For now let’s leave it as it is, confident that we can change it as the user gains experience using our application.

**Data**

Finally, we need to think about the data we’re going to be working with.

Notice that we’re not using words such as schema or classes here. We’re also not talking about databases, tables, keys, and the like. We’re simply talking about data. At this stage in the development, we don’t know whether we’ll even be using a database—sometimes a flat file beats a database table hands down.

Based on the use cases and the flows, it seems likely that we’ll be working with the data shown in Figure 5.3, on the following page. Again, pencil and
paper seems a whole lot easier than some fancy tool, but use whatever works for you.

Working on the data diagram raised a couple of questions. As the user buys items, we'll need somewhere to keep the list of products they bought, so I added a cart. But apart from its use as a transient place to keep this product list, the cart seems to be something of a ghost—I couldn’t find anything meaningful to store in it. To reflect this uncertainty, I put a question mark inside the cart’s box in the diagram. I’m assuming this uncertainty will get resolved as we implement Depot.

Coming up with the high-level data also raised the question of what information should go into an order. Again, I chose to leave this fairly open for now—we’ll refine this further as we start showing the customer our early iterations.

Finally, you might have noticed that I've duplicated the product’s price in the line item data. Here I’m breaking the “initially, keep it simple” rule slightly, but it’s a transgression based on experience. If the price of a product changes, that price change should not be reflected in the line item price of currently open orders, so each line item needs to reflect the price of the product at the time the order was made.

Again, at this point I'll double-check with my customer that we’re still on the right track. (My customer was most likely sitting in the room with me while I
5.3 Let's Code

So, after sitting down with the customer and doing some preliminary analysis, we're ready to start using a computer for development! We'll be working from our original three diagrams, but the chances are pretty good that we'll be throwing them away fairly quickly—they'll become outdated as we gather feedback. Interestingly, that's why we didn't spend too long on them—it's easier to throw something away if you didn't spend a long time creating it.

In the chapters that follow, we'll start developing the application based on our current understanding. However, before we turn that page, we have to answer just one more question. What should we do first?

I like to work with the customer so we can jointly agree on priorities. In this case, I'd point out to her that it's hard to develop anything else until we have some basic products defined in the system, so I'd suggest spending a couple of hours getting the initial version of the product maintenance functionality up and running. And, of course, she'd agree.
In this chapter, we’ll see

- creating a new application
- configuring the database
- creating models and controllers
- running database migrations
- performing validation and error reporting
- working with views and helpers

Chapter 6

Task A: Product Maintenance

Our first development task is to create the web interface that lets us maintain our product information—create new products, edit existing products, delete unwanted ones, and so on. We’ll develop this application in small iterations, where small means “measured in minutes.” Let’s get started.

6.1 Iteration A1: Get Something Running

Perhaps surprisingly, we should get the first iteration of this working in almost no time. We’ll start off by creating a new Rails application. This is where we’ll be doing all our work. Next, we’ll create a database to hold our information (in fact we’ll create three databases). Once that groundwork is in place, we’ll

- configure our Rails application to point to our database(s),
- create the table to hold the product information, and
- have Rails generate the initial version of our product maintenance application for us.

Create a Rails Application

Back on page 45 we saw how to create a new Rails application. Go to a command prompt, and type rails followed by the name of our project. In this case, our project is called depot, so type

```
work> rails depot
```

We see a bunch of output scroll by. When it has finished, we find that a new directory, depot, has been created. That’s where we’ll be doing our work.

```
work> cd depot
depot> ls -p
README config/ lib/ script/ vendor/
Rakefile db/ log/ test/
app/ doc/ public/ tmp/
```
Create the Database

For this application, we’ll use the open source SQLite database (which you’ll need too if you’re following along with the code). I’m using SQLite version 3 here. If you’re using a different database, the commands you’ll need to create the database and grant permissions will be different.

SQLite3 is the new default database for Rails, starting with version 2.0.2. With SQLite3 there are no steps required to create a database, nor are there special user accounts or passwords to deal with. So now you get to experience one of the benefits of going with the flow. If you’re using SQLite3, you can now skip forward to Section 6.2, Create the Products Model and Maintenance Application, on page 76.

If you are still reading this section, it is because you are insisting on using different database server software. There really is no need to do so, as we are only talking about a development database at the moment. Rails will let you use an entirely different database for testing and production. But if you insist, the following examples based on another popular open source database, namely MySQL, may help. You may, of course, choose a different database, but the same basic steps apply to all of the databases that Rails supports.

Getting the various charset and collation options configured correctly often poses problems. For this reason, Rails provides a rake task that can take care of that for you.

```ruby
depot> rake db:create RAILS_ENV='development'
```

For MySQL, you can instead use the `mysqladmin` command-line client to create your database, or if you’re more comfortable with tools such as phpmyadmin or CocoaMySQL, go for it.

```bash
depot> mysqladmin -u root create depot_development
```

If you picked a different database name, remember it, as you will need to adjust the configuration file later to match the name you picked.

Configure the Application

In many simple scripting-language web applications, the information on how to connect to the database is embedded directly into the code—you might find a call to some `connect` method, passing in host and database names, along with a user name and password. This is dangerous, because password information sits in a file in a web-accessible directory. A small server configuration error could expose your password to the world.

The approach of embedding connection information into code is also inflexible. One minute you might be using the development database as you hack away.

---

1. Or, *convention over configuration*, as Rails folks say (ad nauseam)
Next you might need to run the same code against the test database. Eventually, you’ll want to deploy it into production. Every time you switch target databases, you have to edit the connection call. There’s a rule of programming that says you’ll mistype the password only when switching the application into production.

Smart developers keep the connection information out of the code. Sometimes you might want to use some kind of repository to store it all (Java developers often use JNDI to look up connection parameters). That’s a bit heavy for the average web application that we’ll write, so Rails simply uses a flat file. You’ll find it in `config/database.yml`.²

```yaml
# ... some comments ...

development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000
```

database.yml contains information on database connections. It contains three sections, one each for the development, test, and production databases. Since you’ve decided to rebel and use a different configuration, you will need something different in this file. You may even need to edit this file directly.

But before you do that, let’s start over. Delete all the files that Rails generated for you and generate a new set. After all, you aren’t heavily invested in these files just yet. Simply type³

```
depot> cd ..
work> rm -rf depot
work> rails --database=mysql depot
work> cd depot
```

The new `database.yml` file will look something like the following:

```
development:
  adapter: mysql
  encoding: utf8
  database: depot_development
  username: root
  password: 
  host: localhost
```

Remember that you’ll need the appropriate Ruby libraries for the database you select. The comments in the `database.yml` file that appear before the above

² The `.yml` part of the name stands for YAML, or YAML Ain’t a Markup Language. It’s a simple way of storing structured information in flat files (and it isn’t XML). Ruby releases since 2003 include built-in YAML support.

³ The command `rails --help` gives a list of available database options. Windows users will want to use the `rd /S /Q depot` command instead of `rm -rf depot`.
lines may contain helpful hints on this subject.

If you’re going with the flow (why start now?) and use MySQL with root as the user name, and did not change the database name when you created the database, then this database.yml file will probably get you started. However if you are continuing to rebel and use a different configuration, you might need to edit this file. Just open it in your favorite text editor, and edit any fields that need changing. The numbers in the list that follows correspond to the numbers next to the source listing above.

1. The adapter section tells Rails what kind of database you’re using (it’s called adapter because Rails uses this information to adapt to the peculiarities of the database). If you are using MySQL, the adapter name is mysql. A full list of different database adapter types is given in Section 18.4, Connecting to the Database, on page 327. If you’re using a database other than MySQL, you’ll need to consult this table, because each database adapter has different sets of parameters in database.yml.

2. The database parameter gives the name of the database. (Remember, we created our depot_development database using mysqladmin back on page 71.) If you picked a different database name, this is the line you need to change to match the one you picked.

3. The username and password parameters let your application log in to the database. Note that the default file that Rails provides you with assumes that you will be using the user root with no password. You’ll need to change these fields if you’ve set up your database differently. In particular, you know that you really should set a password, right?

   If you leave the user name blank, MySQL might connect to the database using your login name. This is convenient, because it means that different developers will each use their own user names when connecting. However, we’ve heard that with some combinations of MySQL, database drivers, and operating systems, leaving these fields blank makes Rails try to connect to the database as the root user. Should you get an error such as “Access denied for user ‘root’@’localhost.localdomain’,” put an explicit user name and password in these two fields.

4. Finally, there’s the host parameter. This parameter tells Rails the name of the computer that is running your database server. Most developers run a local copy of MySQL on their own machine, so the default value of localhost is fine.

There’s are a few additional parameters that may or may not appear in your default database.yml file.
socket: tells the MySQL database adapter where to find the socket that’s used to talk with the MySQL server. If this value is incorrect, Rails may not be able to find the MySQL socket. If you’re having problems connecting to your database, you can try commenting this line out (by putting a # in front of it). Alternatively, you can find the correct path to the socket by running the command `mysql_config --socket`.

timeout: tells the SQLite3 database adapter how long you are willing to wait, in milliseconds, when you need to acquire an exclusive lock.

pool: tells Rails how many concurrent connections your application can have to the database server.

Remember—if you’re just getting started and you’re happy to use the Rails defaults, you shouldn’t have to worry about all these configuration details.

Testing Your Configuration

Before you go too much further, we should probably test your configuration so far—we can check that Rails can connect to your database and that it has the access rights it needs to be able to create tables. From your application’s top-level directory, type the following magic incantation at a command prompt. (It’s magic, because we don’t really need to know what it’s doing quite yet. We’ll find out later.)

```
depot> rake db:migrate
```

One of two things will happen. Either you’ll get a single line echoed back (saying something like “in (/Users/dave/work/depot)”), or you’ll get an error of some sort. The error means that Rails can’t currently work with your database. If you do see an error, don’t panic! It’s probably a simple configuration issue. Here are some things to try.

- Check the name you gave for the database in the development section of `database.yml`. It should be the same as the name of the database you created (using `mysqladmin` or some other database administration tool).
- Check that the user name and password in `database.yml` match what you created on page 71.
- Check that your database server is running.
- Check that you can connect to it from the command line. If using MySQL, run the following command.

```
depot> mysql -u root depot_development
mysql>
```
• If you can connect from the command line, can you create a dummy table? (This tests that the database user has sufficient access rights to the database.)

```sql
mysql> create table dummy(i int);
mysql> drop table dummy;
```

• If you can create tables from the command line but `rake db:migrate` fails, double-check the `database.yml` file. If there are `socket:` directives in the file, try commenting them out by putting a hash character (#) in front of each.

• If you see an error saying “No such file or directory...” and the filename in the error is `mysql.sock`, your Ruby MySQL libraries can’t find your MySQL database. This might happen if you installed the libraries before you installed the database, or if you installed the libraries using a binary distribution, and that distribution made the wrong assumption about the location of the MySQL socket file. To fix this, the best idea is to reinstall your Ruby MySQL libraries. If this isn’t an option, double check that the `socket:` line in your `database.yml` file contains the correct path to the MySQL socket on your system.

• If you get the error “Mysql not loaded,” it means you’re running an old version of the Ruby MySQL library. Rails needs at least version 2.5.

• Some readers also report getting the error message “Client does not support authentication protocol requested by server; consider upgrading MySQL client.” This incompatibility between the installed version of MySQL and the libraries used to access it can be resolved by following the instructions at [http://dev.mysql.com/doc/mysql/en/old-client.html](http://dev.mysql.com/doc/mysql/en/old-client.html) and issuing a MySQL command such as `set password for 'some_user'@'some_host' = OLD_PASSWORD('newpwd');`.

• If you’re using MySQL under Cygwin on Windows, you may have problems if you specify a host of `localhost`. Try using `127.0.0.1` instead.

• You may have problems if you’re using the pure-Ruby MySQL library (as opposed to the more performant C library). Solutions for various operating systems are available on the Rails wiki.

• Finally, you might have problems in the format of the `database.yml` file. The YAML library that reads this file is strangely sensitive to tab characters. If your file contains tab characters, you’ll have problems. (And you thought you’d chosen Ruby over Python because you didn’t like Python’s significant whitespace, eh?)

---

If all this sounds scary, don’t worry. In reality, database connections work like a charm most of the time. And once you’ve got Rails talking to the database, you don’t have to worry about it again.

6.2 Create the Products Model and Maintenance Application

Back in Figure 5.3, on page 68, we sketched out the basic content of the products table. Now let’s turn that into reality. We need to create a database table and a Rails model that lets our application use that table, a number of views to make up the user interface, and a controller to orchestrate the application.

At this point, we have a decision to make. How do we specify the structure of our database table? Should we use low-level Data Definition Language (DDL) statements (create table and friends)? Or is there a higher-level way, one that makes it easier to change the schema over time? Of course there is! In fact, there are a number of alternatives.

Many people like using interactive tools to create and maintain schemas. The SQLite Manager plugin, for example, lets you maintain a SQLite3 database within your Firefox browser. At first sight this approach to database maintenance is attractive—after all, what’s better than just typing some stuff into a form and having the tool do all of the work? However, this convenience comes at a price: the history of the changes you’ve made is lost, and all your changes are effectively irreversible. It also makes it hard for you to deploy your application: you have to remember to make the same changes to both your development and production databases (and we all know that if you’re going to fat finger something, it’ll be when you’re editing the production schema).

Fortunately, Rails offers a middle ground. With Rails, you can define database migrations. Each migration represents a change you want to make to the database, expressed in a source file in database-independent terms. These changes can update both the database schema and the data in the database tables. You apply these migrations to update your database, and you can unapply them to roll your database back. We have a whole chapter on migrations starting on page 297, so for now, we’ll just use them without too much more comment.

Just how do we create these migrations? Well, when you think about it, we normally want to create a database table at the same time as we create a Rails model that wraps it. So Rails has a neat shortcut. When you use the generator to create a new model, Rails automatically creates a migration that you can use to create the corresponding table. (As we’ll see later, Rails also makes it easy to create just the migrations.)
So, let’s go ahead and create the model, views, controller and migration for our products table. Note that on the command line⁵ that follows, we use the singular form, product. In Rails, a model is automatically mapped to a database table whose name is the plural form of the model’s class. In our case, we asked for a model called Product, so Rails associated it with the table called products. (And how will it find that table? The development entry in config/database.yml tells Rails where to look for it. For SQLite3 users, this will be a file in the db directory.)

```
depot> ruby script/generate scaffold product \
   title:string description:text image_url:string
exists app/models/
exists app/controllers/
exists app/helpers/
create app/views/products
exists app/views/layouts/
exists test/functional/
exists test/unit/
exists public/stylesheets/
create app/views/products/index.html.erb
create app/views/products/show.html.erb
create app/views/products/new.html.erb
create app/views/products/edit.html.erb
create public/stylesheets/scaffold.css
create app/controllers/products_controller.rb
create test/functional/products_controller_test.rb
create app/helpers/products_helper.rb
route map.resources :products
dependency model
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/product.rb
create test/unit/product_test.rb
create test/fixtures/products.yml
create db/migrate
create db/migrate/20080601000001_create_products.rb
```

The generator creates a bunch of files. The one we’re interested in first is the migration 20080601000001_create_products.rb.⁶ The migration has a UTC-based timestamp prefix (20080601000001), a name (create_products), and a file extension (.rb, because it’s a Ruby program).

Since we already specified the columns we wanted to add on the command line, we don’t need to modify this file. All we need to do is to get Rails to

---

⁵ This command is too wide to fit comfortably on the page. To enter a command on multiple lines, simply put a backslash as the last character on all but the last line, and you will be prompted for more input. Windows users will need to put the entire command on one line, without the backslash.

⁶ The timestamps used in this book are clearly fictitious. Typically your timestamps will not be consecutive but will reflect the time the migration was created.
apply this migration to our development database. We do this using the rake command. Rake is like having a reliable assistant on hand all the time: you tell it to do some task, and that task gets done. In this case, we’ll tell Rake to apply any unapplied migrations to our database.

depot> rake db:migrate
(in /Users/rubys/work/depot)
== 20080601000001 CreateProducts: migrating ==================================================
-- create_table(:products)
    -> 0.0027s
== 20080601000001 CreateProducts: migrated (0.0028s) =========================

And that’s it. Rake looks for all the migrations not yet applied to the database and applies them. In our case, the products table is added to the database defined by the development: section of the database.yml file.7

How does Rake know which migrations have and have not been applied to your database? Have a look at your database after running a migration. You’ll find a table called schema_migrations that it uses to keep track of the version number.8

depot> sqlite3 db/development.sqlite3 "select version from schema_migrations"
20080601000001

OK. All the groundwork has been done. We set up our Depot application as a Rails project. We created the development database and configured our application to be able to connect to it. We created a product controller and a product model and used a migration to create the corresponding products table. And a number of views have been created for us. Time to see all this in action.

Run the Maintenance Application

With three commands we have created an application and a database. Or a table inside an existing database, if you chose something other than SQLite3. Before we worry too much about just what happened behind the scenes here, let’s try our shiny new application. First, we’ll start a local WEBrick-based web server, supplied with Rails.

depot> ruby script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000

7. If you’re feeling frisky, you can experiment with rolling back the migration. Just type depot> rake db:migrate VERSION=0
Your schema will be transported back in time, and the products table will be gone. Calling rake db:migrate again will re-create it.

8. Sometimes this schema_migrations table can cause you problems. For example, if you create the migration source file and run db:migrate before you add any schema-defining statements to the file, the database will think it has been updated, and the schema info table will contain the new version number. If you then edit that existing migration file and run db:migrate again, Rails won’t know to apply your new changes. In these circumstances, it’s often easiest to drop the database, re-create it, and rerun your migration(s).
Just as it did with our demo application in Chapter 4, *Instant Gratification*, this command starts a web server on our local host, port 3000. Let's connect to it. Remember, the URL we give to our browser contains both the port number (3000) and the name of the controller in lowercase (products).

![Listing products](image)

That's pretty boring. It's showing us an empty list of products. Let's add some. Click the *New product* link, and a form should appear. Go ahead and fill it in.

![New product](image)

Click the Create button, and you should see the new product was successfully created. If you now click on the back link, you should see the new product in the list.

---

9. You might get an error saying “Address already in use” when you try to run WEBrick. That simply means that you already have a Rails WEBrick server running on your machine. If you've been following along with the examples in the book, that might well be the *Hello World!* application from Chapter 4. Find its console, and kill the server using control-C.
Perhaps it isn’t the prettiest interface, but it works, and we can show it to our client for approval. She can play with the other links (showing details, editing existing products, and so on...). We explain to her that this is only a first step—we know it’s rough, but we wanted to get her feedback early. (And three commands probably counts as early in anyone’s book.)

6.3 Iteration A2: Add a Missing Column

So, we show our scaffold-generated code to our customer, explaining that it’s still pretty rough-and-ready. She’s delighted to see something working so quickly. Once she plays with it for a while, she notices that something was missed—our products have no prices.

This means we’ll need to add a column to the database table. Some developers (and DBAs) would add the column by firing up a utility program and issuing the equivalent of the command

```
alter table products add column price decimal(8,2);
```

But we know all about migrations. Using a migration to add the new column will give us a version-controlled history of the schema and a simple way to re-create it.

We’ll start by creating the migration. Previously we used a migration generated automatically when we created the product model. This time, we have to create one explicitly. We’ll give it a descriptive name—this will help us remember what each migration does when we come back to our application a year from now. Our convention is to use the verb create when a migration creates tables and add when it adds columns to an existing table.

```
depot> ruby script/generate migration add_price_to_product price:decimal
exists db/migrate
create db/migrate/20080601000002_add_price_to_product.rb
```

Notice how the generated file has a UTC based timestamp prefix (in this case 20080601000002) UTC stands for "coordinated universal time" and was formerly known as Greenwich mean time (GMT). The format of the timestamp is YYYYMMDDhhmmss. Rails uses this timestamp to keep track of what migrations have
been and have not been added to the schema (and also to tell it the order in which migrations should be applied).

Open the migration source file, and edit the \texttt{up} method, inserting the code to add the \texttt{:precision}, \texttt{:scale}, and \texttt{:default} arguments\(^{10}\) to the definition of the price column in the \texttt{products} table, as shown in the code that follows. The \texttt{down} method uses \texttt{remove_column} to drop the column.

\begin{lstlisting}[language=ruby] class AddPriceToProduct < ActiveRecord::Migration  
def self.up  
  add_column :products, :price, :decimal,  
             :precision => 8, :scale => 2, :default => 0  
end

def self.down  
  remove_column :products, :price  
end
end
\end{lstlisting}

The \texttt{:precision} argument tells the database to store eight significant digits for the price column, and the \texttt{:scale} option says that two of these digits will fall after the decimal point. We can store prices from -999,999.99 to +999,999.99.\(^{11}\)

This code also shows another nice feature of migrations—we can access features of the underlying database to perform tasks such as setting the default values for columns. Don't worry too much about the syntax used here: we'll talk about it in depth later.

Now we can run the migrations again.

\begin{verbatim}
depot> rake db:migrate (in /Users/rubys/work/depot)  
== 20080601000002 AddPriceToProduct: migrating --------------------------  
-- add_column(:products, :price, :decimal, (:precision=>8))  
  -> 0.0061s  
== 20080601000002 AddPriceToProduct: migrated (0.0062s)  
\end{verbatim}

Rails knows that the database is currently at version 20080601000001, so applies only our newly created 20080601000002 migration.

But we are not done yet. That only takes care of the model. The flow of control doesn’t change, so no changes are needed to the controller. All that is left is the view. While this does mean that you have to edit four files, the changes are very straightforward. Note that the change is subtly different in the \texttt{show.html.erb} file.

\(^{10}\) Be sure to remember to add a comma to the preceding line
\(^{11}\) At the time of this writing, the latest version of SQLite3 (namely version 3.5.9) will parse and store this schema information, but will otherwise ignore it, giving you instead 16 digits of precision and a variable scale. If this is important to you, you might want to select a different database to use for deployment.
Prices, Dollars, and Cents

When we defined our schema, we decided to store the product price in a decimal column, rather than a float. There was a reason for this. Floating-point numbers are subject to round-off errors: put enough products into your cart, and you might see a total price of 234.99 rather than 235.00. Decimal numbers are stored both in the database and in Ruby as scaled integers, and hence they have exact representations.

David Says...

Where Did Dynamic Scaffolding Go?

Rails used to have a way of declaring that a controller was acting as a scaffold interface for a given model. From a single line in the controller, you could animate to life the complete scaffold interface. It looked great in demos! Just add one line and voila, you had a mini web application there for you already.

The magic wonder of that voila turned out to be more curse than blessing, though. The whole point of scaffolding is to teach people how to use Rails for the simple CRUD scenario. To give you a simple, no-frills inline tutorial that you can tweak, change, extend and learn from. None of that is possible if the scaffold is hidden behind smoke and mirrors.

There is one drawback from being explicit, though, which is that you can’t just update your model with another field and have the scaffold automatically add the new field. You’ll need to either rerun the script generate/scaffold command (which will overwrite any changes you made) or update it by yourself. The latter is a great way of learning, of course, but it can seem a little cumbersome.

Don’t worry too much about the details at the moment; we’ll revisit the user interface shortly.

<!DOCTYPE html>
<html>
<head>
<title>Listing products</title>
</head>
<body>
<h1>Listing products</h1>
<table>
<tr>
<th>Title</th>
<th>Description</th>
<th>Image url</th>
<th>Price</th>
</tr>
</table>
</body>
</html>
<% for product in @products %>
<tr>
  <td><%=h product.title %></td>
  <td><%=h product.description %></td>
  <td><%=h product.image_url %></td>
  <td><%=h product.price %></td>
  <td><%= link_to 'Show', product %></td>
  <td><%= link_to 'Edit', edit_product_path(product) %></td>
  <td><%= link_to 'Destroy', product, :confirm => 'Are you sure?', :method => :delete %></td>
</tr>
<% end %>
</table>
<br />
<%= link_to 'New product', new_product_path %>

<h1>New product</h1>
<% form_for(@product) do |f| %>
  <%= f.error_messages %>
  <p>
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :description %><br />
    <%= f.text_area :description, :rows => 6 %>
  </p>
  <p>
    <%= f.label :image_url %><br />
    <%= f.text_field :image_url %>
  </p>
  <p>
    <%= f.label :price %><br />
    <%= f.text_field :price %>
  </p>
  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>
<%= link_to 'Back', products_path %>

<h1>Editing product</h1>
<% form_for(@product) do |f| %>
  <%= f.error_messages %>
  <p>
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :description %><br />
    <%= f.text_area :description, :rows => 6 %>
  </p>
  <p>
    <%= f.label :image_url %><br />
    <%= f.text_field :image_url %>
  </p>
  <p>
    <%= f.label :price %><br />
    <%= f.text_field :price %>
  </p>
  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>
<%= link_to 'Back', products_path %>

Download depot_a/app/views/products/new.html.erb
Download depot_a/app/views/products/edit.html.erb
Here's the cool part. Go to your browser, which is already talking to our application. Hit Refresh, and you should now see the price column included in these four pages.
We said that the Product model went to the products table to find out what attributes it should have. In development mode, Rails reloads models each time a browser sends in a request, so the model will always reflect the current database schema. And you have updated the views so that they can use this model information to update the screens that are displayed.

There’s no real magic here at the technical level. However, this capability has a big impact on the development process. How often have you implemented exactly what a client asked for, only to be told “Oh, that’s not what I meant” when you finally showed them the working application? Most people find it far easier to understand ideas when they can play with them. The speed with which you can turn words into a working application with Rails means that you’re never far from being able to let the client play with the results. These short feedback cycles mean that both you and the client get to understand the real requirements sooner, and you waste far less time reworking your application.

As a quick example, the markup that you entered in the description appears when showing the product. You can fix this by deleting the \( h \) that appears on the \@product.description line in app/views/products/show.html.erb.

```erb
<%= h @product.title %>

<%= @product.description %>

<%= h @product.image_url %>

<%= h @product.price %>

<%= link_to 'Edit', edit_product_path(@product) %> |
<%= link_to 'Back', products_path %>
```

6.4 Iteration A3: Validate!

While playing with the results of iteration 2, our client noticed something. If she entered an invalid price or forgot to set up a product description, the appli-
ication happily accepted the form and added a line to the database. Although a missing description is embarrassing, a price of $0.00 actually costs her money, so she asked that we add validation to the application. No product should be allowed in the database if it has an empty title or description field, an invalid URL for the image, or an invalid price.

So, where do we put the validation? The model layer is the gatekeeper between the world of code and the database. Nothing to do with our application comes out of the database or gets stored into the database that doesn’t first go through the model. This makes models an ideal place to put validations; it doesn’t matter whether the data comes from a form or from some programmatic manipulation in our application. If a model checks it before writing to the database, then the database will be protected from bad data.

Let’s look at the source code of the model class (in app/models/product.rb).

```ruby
class Product < ActiveRecord::Base
  end
```

Not much to it, is there? All of the heavy lifting (database mapping, creating, updating, searching, and so on) is done in the parent class (ActiveRecord::Base, a part of Rails). Because of the joys of inheritance, our Product class gets all of that functionality automatically.

Adding our validation should be fairly clean. Let’s start by validating that the text fields all contain something before a row is written to the database. We do this by adding some code to the existing model.

```ruby
validates_presence_of :title, :description, :image_url
```

The validates_presence_of method is a standard Rails validator. It checks that the named fields are present and their contents are not empty. Figure 6.1, on the following page shows what happens if we try to submit a new product with none of the fields filled in. It’s pretty impressive: the fields with errors are highlighted, and the errors are summarized in a nice list at the top of the form. Not bad for one line of code. You might also have noticed that after editing and saving the product.rb file you didn’t have to restart the application to test your changes—the same reloading that caused Rails to notice the earlier change to our schema also means it will always use the latest version of our code.

Now we’d like to validate that the price is a valid, positive number. We’ll attack this problem in two stages. First, we’ll use the delightfully named validates_numericality_of method to verify that the price is a valid number.

```ruby
validates_numericality_of :price
```

Now, if we add a product with an invalid price, the appropriate message will appear, as shown in Figure 6.2, on page 88.
Figure 6.1: Validating That Fields Are Present
Next, we need to check that the price is greater than zero. We do that by writing a method named `price_must_be_at_least_a_cent` in our `Product` model class. We also pass the name of the method to the `ActiveRecord::Base.validate` method so that Rails will know to call this method before saving away instances of our product. We make it a protected method, because it shouldn’t be called from outside the context of the model. Be careful as you add methods to this model as we work further on the product—if you add them after the protected declaration, they’ll be invisible outside the class. New actions must go before the protected line.

12. SQLite3 gives Rails enough metadata to know that price contains a number, so Rails stores it internally as a BigDecimal. With other databases, the value might come back as a string, so you’d need to convert it using `BigDecimal(price)` (or perhaps `Float(price)`) if you like to live dangerously) before using it in a comparison.
validate :price_must_be_at_least_a_cent

protected
def price_must_be_at_least_a_cent
  errors.add(:price, 'should be at least 0.01') if price.nil? || price < 0.01
end

If the price is less than one cent, the validate method uses errors.add(...) to record the error. Doing this stops Rails writing the row to the database. It also gives our forms a nice message to display to the user. The first parameter to errors.add is the name of the field, and the second is the text of the message.

Note that before we compare the price to 0.01, we first check to see whether it's nil. This is important: if the user leaves the price field blank, no price will be passed from the browser to our application, and the price variable won't be set. If we tried to compare this nil value with a number, we'd get an error.

Two more items to validate. First, we want to make sure that each product has a unique title. One more line in the Product model will do this. The uniqueness validation will perform a simple check to ensure that no other row in the products table has the same title as the row we're about to save.

validates_uniqueness_of :title

Lastly, we need to validate that the URL entered for the image is valid. We'll do this using the validates_format_of method, which matches a field against a regular expression. For now we'll just check that the URL ends with one of .gif, .jpg, or .png.

validates_format_of :image_url,
  :with => %r{\.(gif|jpg|png)$i},
  :message => 'must be a URL for GIF, JPG or PNG image.'

So, in a couple of minutes we've added validations that check

• The field's title, description, and image URL are not empty.
• The price is a valid number not less than $0.01.
• The title is unique among all products.
• The image URL looks reasonable.

This is the full listing of the updated Product model.

---

13. Why test against one cent, rather than zero? Well, it's possible to enter a number such as 0.001 into this field. Because the database stores just two digits after the decimal point, this would end up being zero in the database, even though it would pass the validation if we compared against zero. Checking that the number is at least one cent ensures only correct values end up being stored.

14. Later on, we'd probably want to change this form to let the user select from a list of available images, but we'd still want to keep the validation to prevent malicious folks from submitting bad data directly.
### Iteration A4: Prettier Listings

Our customer has one last request (customers always seem to have one last request). The listing of all the products is ugly. Can we “pretty it up” a bit? And, while we’re in there, can we also display the product image along with the image URL?

We’re faced with a dilemma here. As developers, we’re trained to respond to these kinds of requests with a sharp intake of breath, a knowing shake of the head, and a murmured “you want what?” At the same time, we also like to show off a bit. In the end, the fact that it’s fun to make these kinds of changes using Rails wins out, and we fire up our trusty editor.

Before we get too far, though, it would be nice if we had a consistent set of test data to work with. We could use our scaffold-generated interface and type data in from the browser. However, if we did this, future developers working on our codebase would have to do the same. And, if we were working as part of a team on this project, each member of the team would have to enter their own data. It would be nice if we could load the data into our table in a more controlled way. It turns out that we can. Migrations to the rescue!

Let’s create a data-only migration. The up method clears out the products table, then adds three rows containing typical data to our products table. The down method empties the table. The migration is created just like any other.

```bash
depot> ruby script/generate migration add_test_data
```

Nearing the end of this cycle, we ask our customer to play with the application, and she’s a lot happier. It took only a few minutes, but the simple act of adding validation has made the product maintenance pages seem a lot more solid.
exists db/migrate
create db/migrate/20080601000003_add_test_data.rb

We then add the code to populate the products table. This uses the create method of the Product model. The following is an extract from that file. (Rather than type the migration in by hand, you might want to copy the file from the sample code available online.15 Copy it to the db/migrate directory in your application, and delete the one you just generated. Don’t be concerned if the timestamp of the file you downloaded is before others that you have already migrated as Rails knows which migrations have been completed and which ones have yet to be done.

While you’re there, copy the images16 and the file depot.css17 into corresponding places (public/images and public/stylesheets in your application). Be warned: this migration removes existing data from the products table before loading in the new data. You might not want to run it if you’ve just spent several hours typing your own data into your application!

```ruby
class AddTestData < ActiveRecord::Migration
  def self.up
    Product.delete_all
    Product.create(:title => 'Pragmatic Version Control',
                   :description =>%
                   This book is a recipe-based approach to using Subversion that will get you up and running quickly---and correctly. All projects need version control: it’s a foundational piece of any project’s infrastructure. Yet half of all project teams in the U.S. don’t use any version control at all. Many others don’t use it well, and end up experiencing time-consuming problems.
                 </p>,
                   :image_url => '/images/svn.jpg',
                   :price => 28.50)
    #...
  end

  def self.down
    Product.delete_all
  end
end
```

(Note that this code uses %(...). This is an alternative syntax for double-quoted string literals, convenient for use with long strings. Note also that, because it uses Rails’ create method, it will fail silently if records cannot be inserted due to validation errors.)

Running the migration will populate your `products` table with test data.

```
depot> rake db:migrate
```

Now let’s get the product listing tidied up. There are two pieces to this. Eventually we’ll be writing some HTML that uses CSS to style the presentation. But for this to work, we’ll need to tell the browser to fetch the stylesheet.

We need somewhere to put our CSS style definitions. All scaffold-generated applications use the stylesheet `scaffold.css` in the directory `public/stylesheets`. Rather than alter this file, we created a new application stylesheet, `depot.css`, and put it in the same directory. A full listing of this stylesheet starts on page 718.

Finally, we need to link these stylesheets into our HTML page. If you look at the `.html.erb` files we’ve created so far, you won’t find any reference to stylesheets. You won’t even find the HTML `<head>` section where such references would normally live. Instead, Rails keeps a separate file that is used to create a standard page environment for all products pages. This file, called `products.html.erb`, is a Rails layout and lives in the `layouts` directory.

```html
<!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>Products: <%= controller.action_name %></title>
<%= stylesheet_link_tag 'scaffold' %>
</head>
<body>
<p style="color: green"><%= flash[:notice] %></p>
<%= yield %>
</body>
</html>
```

The eighth line loads the stylesheet. It uses `stylesheet_link_tag` to create an HTML `<link>` tag, which loads the standard scaffold stylesheet. We’ll simply add our `depot.css` file here (dropping the `.css` extension). Don’t worry about the rest of the file: we’ll look at that later.

```html
<!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>
```

Prepared exclusively for Beat Seeliger
Now that we have the stylesheet all in place, we'll use a simple table-based template, editing the file index.html.erb in app/views/products, replacing the scaffold generated view.

```html
<div id="product-list">
  <h1>Listing products</h1>
  <table>
    <% for product in @products %>
    <tr class="<%= cycle('list-line-odd', 'list-line-even') %>">
      <td>
        <%= image_tag product.image_url, :class => 'list-image' %>
      </td>
      <td class="list-description">
        <dl>
          <dt><%=h product.title %></dt>
          <dd><%=h truncate(product.description.gsub(/<.*?>/,''), :length => 80) %></dd>
        </dl>
      </td>
      <td class="list-actions">
        <%= link_to 'Show', product %><br/>
        <%= link_to 'Edit', edit_product_path(product) %><br/>
        <%= link_to 'Destroy', product, :confirm => 'Are you sure?', :method => :delete %>
      </td>
    </tr>
    <% end %>
  </table>
  <br />
  <%= link_to 'New product', new_product_path %>
</div>
```

Even this simple template uses a number of built-in Rails features.

- The rows in the listing have alternating background colors. This is done by setting the CSS class of each row to either list-line-even or list-line-odd. The Rails helper method called cycle does this, automatically toggling between the two style names on successive lines.
- The truncate helper is used to display just the first 80 characters of the
What’s with :method => :delete?

You may have noticed that the scaffold-generated “Destroy” link includes the parameter :method => :delete. This parameter was added to Rails 1.2. This determines which method is called in the ProductsController and also affects which HTTP method is used.

Browsers use HTTP to talk with servers. HTTP defines a set of verbs that browsers can employ and defines when each can be used. A regular hyperlink, for example, uses an HTTP GET request. A GET request is defined by HTTP to be used to retrieve data: it isn’t supposed to have any side effects. So, the Rails team changed the scaffold code generator to force the link to issue an HTTP DELETE*. These DELETE requests are permitted to have side effects and so are more suitable for deleting resources.

* In some cases, Rails will substitute the POST HTTP method for DELETE, based on whether or not the browser is capable of issuing a DELETE method. Either way, the request will not be cached or triggered by web crawlers.

description. But before we call truncate, we called gsub in order to remove the HTML tags from the description.18

• We also used the h method to ensure that any remaining HTML in the product title and description is escaped.

• Look at the link_to ‘Destroy’ line. See how it has the parameter :confirm => ‘Are you sure?’. If you click this link, Rails arranges for your browser to pop up a dialog box asking for confirmation before following the link and deleting the product. (Also, see the sidebar on this page for some scoop on this action.)

So, we’ve loaded some test data into the database, we rewrote the index.html.erb file that displays the listing of products, we added a depot.css stylesheet, and we linked that stylesheet into our page by editing the layout products.html.erb. Bring up a browser, point to localhost:3000/products, and the resulting product listing might look something like the following.

18. If you get a message such as undefined method ‘-’ for {:length=>80}:Hash, then you probably aren’t running Rails 2.2.2 or later. See Chapter 3, Installing Rails, on page 32 for upgrade information, or simply remove :length => from this call (leaving the 80).
A Rails scaffold provides real source code, files that we can modify and immediately see results. We can customize a particular source file and leave the rest alone—changes are both possible and localized.

So, we proudly show our customer her new product listing, and she’s pleased. End of task. Time for lunch.

What We Just Did
In this chapter we laid the groundwork for our store application.

• We created a development database and configured our Rails application to access it.

• We used migrations to create and modify the schema in our development database and to load test data.

• We created the products table and used the scaffold generator to write an application to maintain it.

• We augmented that generated code with validation.

• We rewrote the generic view code with something prettier.

Playtime
Here’s some stuff to try on your own.

• The method validates_length_of (described on page 408) checks the length of a model attribute. Add validation to the product model to check that the title is at least 10 characters long.
• Change the error message associated with one of your validations.
• Add the product price to the output of the index action.

(You’ll find hints at http://pragprog.wikidot.com/rails-play-time)