

22

Case Study: Payment Calculator — Ruby on Rails

Throughout this book, you have learned how XML can be used to construct and validate documents and how it is used for communications between systems. You have also learned how to use several important XML display formats. Sometimes it can be difficult seeing how all of these technologies fit together without a real-world business case. This case study demonstrates how you can build an online home loan calculator using a public web service, a Ruby on Rails web application, JavaScript, and several of the XML technologies you have learned.

In this chapter, you will:

- ❑ Create a Ruby on Rails application.
- ❑ Create a web page to enter loan information.
- ❑ Call a web service to calculate the payments using SOAP.
- ❑ Display the results using Ajax (Asynchronous JavaScript and XML) and SVG.

Mortgage Calculations

Mortgages are commonly used throughout the world when purchasing a home or land. The word “mortgage” comes from French and literally means “death pledge.” Before making such a pledge, the consumer often wants to see what the proposed payments for the loan would be. The payments may generally include interest and a principal reduction component, depending on various laws or religious guidelines.

In the most common mortgages, the schedule of payments (based on a repayment agreement or Note) must be determined before the annual percentage rate (APR) can be calculated. The amount of interest and principal paid in each payment is based on the loan terms selected and generally changes over the course of the loan. Depending on the kind of loan and the country where the loan is being originated, these calculations can become extremely complex. In the United States,

the complexity of the calculations is only the first hurdle. The calculations must adhere to a strict and evolving set of laws that can vary by the combination of jurisdictions in effect.

In this case study, you will use a mortgage calculation web service from Compliance Studio (<http://compliancestudio.com>). The Compliance Studio engine has been used by mortgage industry companies in the U.S. to handle complex lending compliance checks and simple mortgage estimates. Luckily, Compliance Studio offers a number of its programs for free, which will allow you to build a professional grade application quickly. Because Compliance Studio is a United States-based company, the calculations may not be applicable for loans in other countries. Regardless, the examples in this case study can be applied to any number of alternate calculation engines.

What You Need for the Example

In this case study, you will use Ruby on Rails to create your web application. For this sample, you will just use the built-in web server that comes with Rails. (If you prefer another web server, you are welcome to use it.) An alternate version of this case study is available within the printed version of *Beginning XML* which uses .NET instead of Ruby on Rails and the Internet Information Server (IIS) instead of the built-in Rails web server.

For the examples in this chapter, you need:

- Ruby and Rails
- A text editor
- An SVG-enabled browser or an SVG plug-in for Internet Explorer

Ruby on Rails is a framework for web applications that is based on the Ruby programming language. Ruby on Rails can be downloaded and installed on most platforms and is available at <http://www.rubyonrails.com>. The examples in this chapter assume that you are working in Linux. If so, you will need to install Ruby and RubyGems and then run the following command to install Rails:

```
gem install rails --include-dependencies
```

The sample code included with this chapter assumes you are using Rails 1.2.1. If you are using another version of Rails it should be compatible, however, you may need to modify the version in the `environment.rb` file in the `config` folder. If you are working in Windows, follow the instructions on the website for installing Instant Rails.

A list of SVG-enabled browsers and viewers can be found in Chapter 19. If you plan on using Internet Explorer to test and debug your website, at the time of this writing you will need an SVG plug-in. Currently, the most popular plug-in is Adobe's SVG Viewer, available for download at <http://www.adobe.com/svg>. Recent versions of Firefox, Opera, and Konqueror have built-in SVG support.

Some of the samples in this chapter are long. You can download all of the code for these examples from this book's website instead of typing them in yourself.

Creating the Project

Before you can begin building the various parts of the sample you need to create a new Ruby on Rails application. Once you have installed Ruby on Rails this is very easy.

1. Open a terminal window and change to the directory that should contain your project. This can be in your home directory or at another path such as `/var/www`. Run the command:

```
rails loancalculator
```

This will create a new application path called `loancalculator` in the current folder. It also creates many of the files your application will need.

2. After the command completes, change to the new directory:

```
cd loancalculator
```

3. At this point the application should be ready to run. It won't do much yet, but it is worth testing that everything runs correctly. Start the built-in web server by running:

```
script/server
```

In other environments you may need to specify that the server should be run using Ruby:

```
ruby ./script/server
```

If there are no errors (warnings are okay), you should be able to view the default Ruby on Rails page by opening your web browser and going to `http://localhost:3000`, as shown in Figure 22-1.

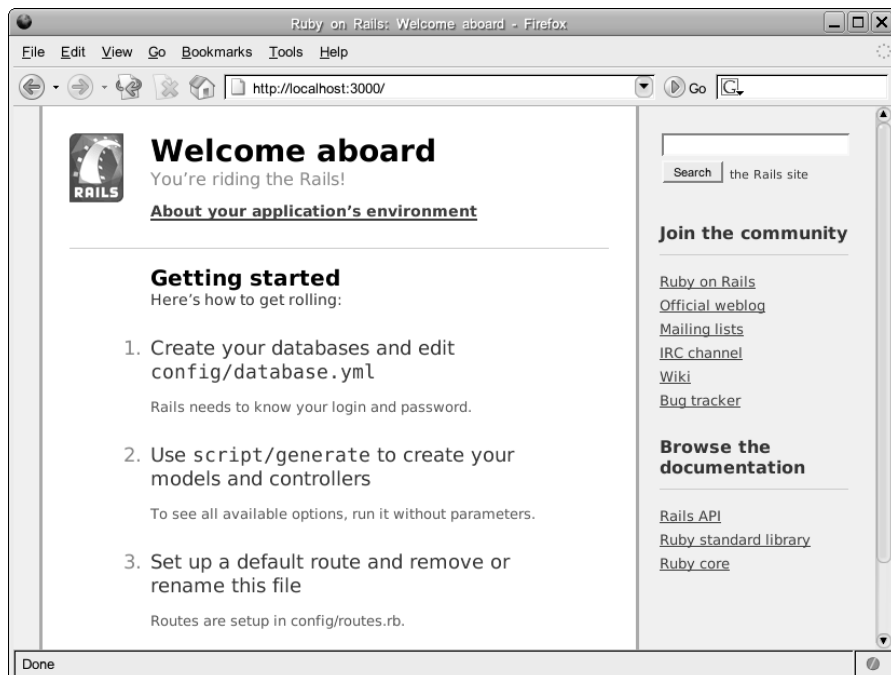


Figure 22-1

This page is really just a placeholder and doesn't mean much. In fact, it suggests that you set up the database you will be using in your application. This example will not use a database so you can skip that step. What it does prove is that your Rails application is set up correctly and you can move on to creating the rest of your calculator application.

You can leave this server running as you work through the chapter. In general, as you make changes to your application the web server will use the new versions automatically.

Building the Online Loan Calculator

There are many different ways to build an online mortgage calculation tool. Throughout the rest of this chapter you will build a simple calculation tool that accepts loan request information from the user, executes a number of calculations and returns the proposed payments and balance. To develop the application you'll need to:

1. Develop the main web page to collect information
2. Integrate the calculation web service
3. Add Ajax support
4. Enhance the display with SVG

Developing the Main Web Page

As in the development of any web application, the easiest place to begin is the main web page. This page will serve as the entry point into your complex server interactions. It will allow the user to input various mortgage details, request a calculation, and see the resulting payment schedule. For starters, this page will be simple; it will contain some basic text, an entry form, and the payment schedule. You shouldn't spend a lot of time on the design of the page at first, but you can make basic styling decisions using CSS.

The Rails Framework follows the Model, View, Controller programming methodology. This means that data (the model) is kept separate from the application logic (the controller) and that the display (the view) of data is handled in another layer. When you created your application, the `rails` command generated folders for `models`, `controllers`, and `views` inside of the `app` folder.

Building the calculator page requires two steps:

1. Creating a calculator controller to display the page.
2. Creating a view for the calculator.

The Try It Outs in this section lead you through these two steps.

Because this example doesn't store any data, you don't need to worry about creating a model. If you wanted to store the results of the calculation in a database you would also need to generate classes for your data models.

Try It Out Building the Loan Calculator

In this Try It Out you generate the basic controller for your loan calculator. Throughout this chapter you will continue to improve the controller. You will also construct the HTML page that will be displayed in the browser by defining a view for the action and a layout for the controller.

1. Open a terminal window and change to the `loancalculator` folder you created using the `rails` command. Run the following command to generate the controller:

```
script/generate controller Calculator
```

You should see the following output:

```
exists app/controllers/  
exists app/helpers/  
create app/views/calculator  
exists test/functional/  
create app/controllers/calculator_controller.rb  
create test/functional/calculator_controller_test.rb  
create app/helpers/calculator_helper.rb
```

The generator has created a number of default files and folders. Notice that it has automatically created test files for you. Because this chapter focuses on how to use XML in Ruby on Rails, it does not go into detail about its support for test-driven development.

2. Even though `rails` generated the skeleton of the application there is still some work to do. Open the file `app/controllers/calculator_controller.rb` in a text editor (the `app` directory will be located inside your main the `loancalculator` directory). By default the `generate` command only created the class:

```
class CalculatorController < ApplicationController  
end
```

You need to add “actions” to the controller so that it can respond to requests. Do this by adding public functions to the class. The default action for a controller is the `index` action:

```
class CalculatorController < ApplicationController  
  def index  
    # render index.rhtml  
  end  
end
```

The `index` action for the loan calculator doesn’t need to do very much. It simply needs to render a view template that can be returned to the browser. It might be helpful to leave a comment in the function to remind others reading your code that this is expected. Once you have added the `index` action you should save the file

3. By default the controller will look for a template with the same name as the action, in this case `index`. In addition to the view template, the controller will also look for a layout that has the same name as the controller, in this case `calculator`. Having a layout for a controller is optional, but it is considered good practice. It allows you to design a consistent look that can be reused for each action. Create a new layout file called `calculator.rhtml` in the `app/views/layouts` folder. Rails templates, or `rhtml` files, are nothing more than XHTML files that contain embedded commands. Begin your layout like this:

```
<!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>
    <title>Loan Calculator</title>
    <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
  </head>
  <body>
```

The template page begins with a `DOCTYPE` declaration pointing to the transitional version of XHTML 1.0. Remember, this will tell the web browser what specific version and flavor of XHTML you're using. Also notice that you haven't included an `<?xml version="1.0"?>` declaration at the start of the page. Though it is allowable in the template, some web browsers will render the page in "quirks mode" if this included. *Quirks mode* is a rendering mode that doesn't always adhere to publish web standards. This means that you can't be sure that your page will be rendered correctly in the user's browser. To avoid that, it is best to leave off the XML declaration.

The namespace declaration `http://www.w3.org/1999/xhtml` is also included in the root element. It is a default namespace declaration that indicates that all of the elements in this document belong to the XHTML namespace by default.

In the `<head>` element, there is a `<title>` element. The `<title>` element is required for the web page to be valid. Also included is a `<meta>` element. This element isn't required; in fact, it uses a non-standard media type as well. Regardless, this helps avoid other problems you might face when using different browsers.

4. The main content of the `calculator.rhtml` layout is fairly simple. To start with, it will contain some basic header text:

```
<div id="container">
  <div id="header">
    <div id="title">
      <h1><span>Loan Calculator</span></h1>
    </div>
  </div>
```

Notice that there are quite a few extra `<div>` elements, and inside the `<h1>` there is an extra `` element. All of this is unnecessary for the page rendering and validation; but it will provide many more options when applying a Cascading Style Sheet to the page. If you are working with a professional designer for your website, he or she will appreciate the increased flexibility.

5. Next you will need a little bit of Rails magic. The view for each action needs to be rendered inside of the layout:

```
<%= yield %>
```

In Rails templates you can include commands with the special symbols `<%` and `%>`. Of course, this isn't legal XML markup, but the Rails framework will process and remove all of these template pieces before the page is sent to the browser. This command begins with `<%=` which indicates that result of any processing should be included in the template. Here the `yield` function is called, which tells the controller to take over and render the view for the current action.

6. After rendering the view, the controller will return to complete the current template. All that's left is add the closing tags for the container `<div>`, `<body>` and `<html>` elements:

```
</div>
</body>
</html>
```

Save the file `calculator.rhtml`.

7. Before you can view the page you will need to make the `index.rhtml` template. Because the layout already contains all of the header information, the `index.rhtml` template doesn't need to repeat the `<html>` and `<head>` elements. In fact, you should only include a sub-header and the form components themselves. This follows the Ruby on Rails principle of not repeating yourself. Create a new file called `index.rhtml` in the `app/views/calculator` folder and copy the following:

```
<h2><span>Tell us about the loan you would like</span></h2>
```

Having a sub-heading is useful for introducing the form. The form itself will use the Rails `form_tag` helper:

```
<% form_tag :action => :calculate do %>
```

The `form_tag` helper is useful when creating a form that isn't connected to a particular model object in Rails. The form points to the `calculate` action, which means that when the user submits the form data it will be passed to the `calculate` method defined on the server. You'll create the `calculate` action a little later in this chapter.

Inside of the form tag you'll need to include the `<input>` and `<select>` elements for the loan details. Again, you should use a mix of Rails helper functions and XHTML elements:

```
<div id="program_group">
  <label for="program_name">Choose the loan program</label>
  <%= select_tag :program_name, options_for_select(
    {"Fixed" => "FIXED 360/360",
     "Fixed Hybrid" => "FIXED 50 30 10",
     "Interest Only Option" => "GMAC IO Option Neg ARM"}) %>
</div>
<div id="amounts_group">
  <label for="original_loan_amount">Loan Amount</label>
  <%= text_field_tag :original_loan_amount %>
  <label for="disclosed_total_sales_price_amount">Sales Price</label>
  <%= text_field_tag :disclosed_total_sales_price_amount %>
  <label for="property_appraised_value_amount">Appraised Value</label>
  <%= text_field_tag :property_appraised_value_amount %>
</div>
<div id="terms_fees_group">
  <label for="loan_original_maturity_term_months">Term</label>
  <%= text_field_tag :loan_original_maturity_term_months %>
  <label for="total_apr_fees_amount">Total Fees</label>
  <%= text_field_tag :total_apr_fees_amount %>
</div>
<div id="rate_group">
  <label for="note_rate_percent">Note Rate</label>
  <%= text_field_tag :note_rate_percent %>
```

```
<label for="index_value">Index Value</label>
<%= text_field_tag :index_value %>
<label for="margin_value">Margin Value</label>
<%= text_field_tag :margin_value %>
</div>
<div id="buttons">
  <%= submit_tag "Calculate" %>
</div>
```

Here the elements are broken into logical groups based on the kind of data. Again, this will be useful when applying a stylesheet to the page. Also included are `<label>` elements to describe each of the inputs. Technically, you could use `<p>` elements for the descriptions, but the `<label>` element allows you to link the `<label>` and the form control using a `for` attribute. Most browsers have special support for labels that utilize the `for` attribute, such as focusing the control when the `<label>` is clicked.

Additional information on each of the mortgage concepts is available on the Compliance Studio website or at <http://en.wikipedia.org/wiki/Mortgage>.

Finally, you need to include a matching `end` command for the `form_tag`:

```
<% end %>
```

This completes the `index.rhtml` template. Save the file.

- At this point you should be able to navigate to the newly created page. If your web server is still running, open `http://localhost:3000/calculator` in your web browser (see Figure 22-2).

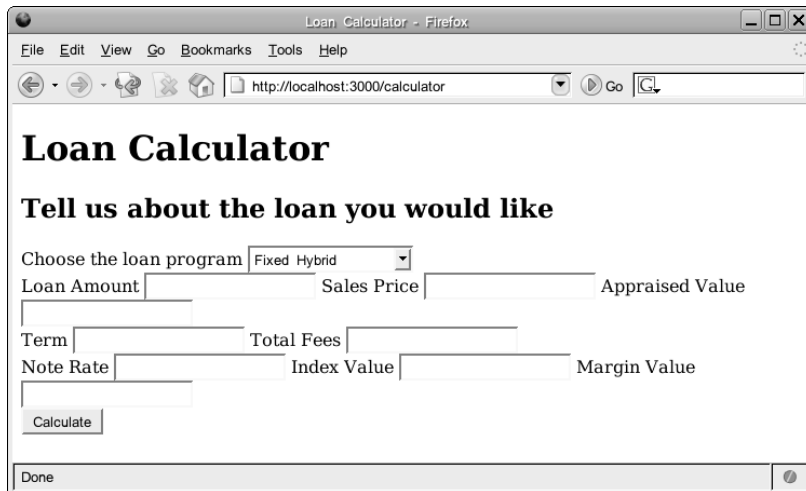


Figure 22-2

How It Works

In this Try It Out, you built the main web page for the loan calculator. Though the page was divided across several files, the Rails engine was able to process all of the templates and create a valid XHTML

page. In fact, if you view the source of the page in the web browser you won't see any of the `<%` and `%>` symbols. One of the strengths of the Rails is that the creators focused on making sure that the engine would output standards-compliant markup. It means that for the most part you can trust that your page will work in most browsers.

The page is fairly simple and, apart from some extra `<div>` elements, is entirely content driven. Notice that you haven't included any extra information about the layout of the page or any visuals. Separating the content of a web page from its presentation layer simplifies maintenance of the page in the future. Unfortunately, though, because you haven't yet created a stylesheet for the page, it isn't very pretty.

Before you add more functionality, you can add a basic stylesheet to the page.

Try It Out Improving the Look of the Loan Calculator

In many professional sites, a design team is hired to create stunning graphics and page layouts. In this case study, you will focus on the basics and try to build a CSS document that makes testing a little more enjoyable.

1. Open your text editor, and create a new stylesheet document named `public/stylesheet/calculator.css`. The `public/stylesheet` folder should have been generated by the `rails` command and can be found inside of the `loancalculator` folder.

The stylesheet should begin with a default rule:

```
* { margin: 0; padding: 0; }
```

Even though you are building a basic stylesheet, it is good to follow best practice guidelines. Beginning a stylesheet by setting the margin and padding for all elements to 0 ensures that different browsers will treat these properties the same. Tips like this are shared freely in online CSS communities like <irc://irc.freenode.net/css>.

2. Next, define a template for the `<body>` tag:

```
body {  
  background-color:white;  
  color:black;  
  font-family:arial, sans-serif;  
  margin-left:10px;  
}
```

Again, it isn't required to set default background and text colors but it is good practice. Also, choosing a font you like and providing a fallback font such as `sans-serif` will guarantee that the page will remain fairly consistent across various platforms.

3. The biggest problem with the loan calculator is the layout of the `<input>` elements. Because you have grouped them in uniquely named `<div>` elements, you can be very precise with their position. Use absolute positioning to define the layout of the various groups based on the `id` attribute of each `<div>` element:

```
#program_group {  
  position:absolute;  
  left:10px;  
  top:70px;
```

```
}
#amounts_group {
  position:absolute;
  left:10px;
  top:120px;
}
#terms_fees_group {
  position:absolute;
  left:170px;
  top:120px;
}
#rate_group {
  position:absolute;
  left:330px;
  top:120px;
}
```

4. The `<label>` elements also need a template. Right now the labels appear next to the `<input>` controls. Instead, treat them as block level elements with a line break after each one:

```
label {
  display:block;
}
```

5. Create a template for the `<form>` element which is generated by the `form_tag` helper. Because you used absolute positioning for the `<input>` groups, the `<form>` has no actual height, but you need to leave a space where the form contents should go. The `height` and `min-height` CSS properties are not implemented consistently in all browsers, so you can cheat and add padding to the top and bottom of the form instead:

```
form {
  padding-top:190px;
  padding-bottom:10px;
}
```

Save the stylesheet.

6. Finally, modify the layout to refer to the stylesheet. In the `calculator.rhtml` layout file in `app/views/layouts` add a `<link>` element into the `<head>` section of the document. Add the highlighted line and save the file:

```
<head>
  <title>Loan Calculator</title>
  <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
  <%= stylesheet_link_tag 'calculator' %>
</head>
```

Once you have completed the changes, save the file.

How It Works

In this Try It Out, you added a CSS stylesheet to the main web page for the loan calculator. The stylesheet may not win any design awards, but working with the input fields and visualizing the results is now much more pleasant, as you can see in Figure 22-3. In the stylesheet you followed best-practice

guidelines and used a mix of CSS features. Some elements were styled using ID selectors, while others used element names. Using this stylesheet as a basis, you could alter the look of your page very quickly.

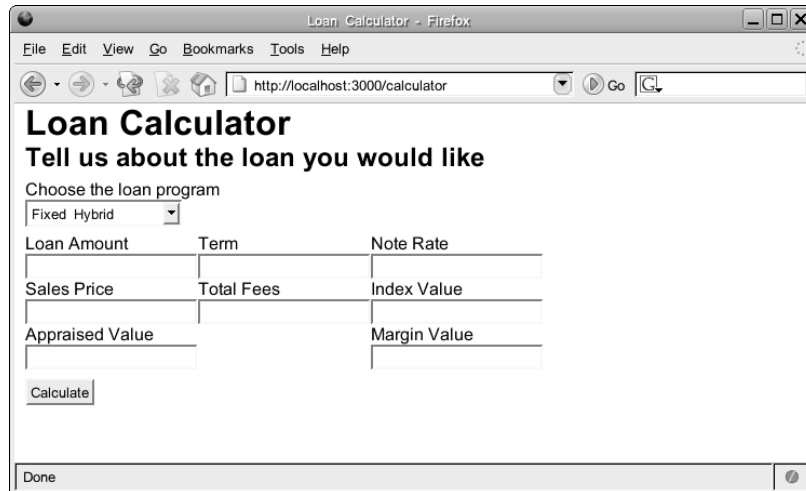
A screenshot of a web browser window titled "Loan Calculator - Firefox". The address bar shows "http://localhost:3000/calculator". The page content includes a heading "Loan Calculator" and a sub-heading "Tell us about the loan you would like". Below this is a form with a dropdown menu for "Choose the loan program" set to "Fixed Hybrid". The form contains several input fields: "Loan Amount", "Term", "Note Rate", "Sales Price", "Total Fees", "Index Value", "Appraised Value", and "Margin Value". A "Calculate" button is located below the input fields. The browser's status bar at the bottom shows "Done".

Figure 22-3

Integrating the Calculation Web Service

Now that you have the main loan calculator web page, all you have to do is connect it to the Compliance Studio web service and display the results. Unfortunately, the web page cannot communicate directly with the Compliance Studio web service because the web page and the service are hosted in different domains. Currently, browsers do not permit web pages to make HTTP requests to URLs that have a different domain. These requests are commonly called *Cross-Domain XML HTTP Requests* or *Cross Site Scripting (XSS)*. Cross Site Scripting is the source of many common security problems. Instead, you need to have the browser call an action on your own web server and let the web server talk with the Compliance Studio calculation service. This middle-man approach allows you to use the server as a proxy for the interaction.

The action for the loan calculator form was `calculate`. The `calculate` action is just another method within the calculator controller. It will need to do three things:

- ❑ Convert the incoming form data to XML.
- ❑ Call the Compliance Studio web service using SOAP.
- ❑ Display the payments for the loan.

Try It Out The calculate Action

In this Try It Out you create the various parts of the `calculate` action to communicate with the Compliance Studio web service.

1. Begin by modifying the `calculator_controller.rb` inside the `app/controllers` folder to define a new method called `calculate`. The `calculate` method definition can go immediately after the `index` method. Unlike the `index` method, though, you need to do more than let `calculate` render the default template:

```
def calculate
  begin
    @xml = Builder::XmlMarkup.new
    @soap_request = render_to_string :partial => "soap"
    @calculation_result = call_calculation_service
    @payments = @calculation_result["TRANSACTION"]["RESPONSE"]["PAYMENTSTREAM"]
  rescue Exception => e
    flash.now[:error] = "There was an error communicating with the service"
  end
  render :partial => "calculate", :layout => true
end
```

This method starts with the `begin` keyword. In Ruby you can handle exceptions by creating a `begin/rescue/end` block. If an exception is raised within the `begin` block the code within the `rescue` will be executed and the function will continue. When communicating with remote services it is a good idea make sure your code can handle exceptions.

You will use the `RequestCalculation` function within the `The Compliance Studio` web service. You can find information about the `RequestCalculation` function at

<http://compliancestudio.com/apr/1.0/service.asmx?op=RequestCalculation>.

The `RequestCalculation` function expects only one parameter, `TransactionEnvelope`, which should contain an XML document fragment with all of the loan details. Compliance Studio provides an XML Schema for the document fragment at

<http://compliancestudio.com/apr/APRData-1-0.xsd>.

Ruby on Rails has several different tools to create XML in your application. Using an `XmlMarkup` builder with an RXML template is the most common. Though you could create an XML document using string concatenation, it is considered bad practice. Outputting XML directly can often lead to well-formedness errors.

```
@soap_request = render_to_string :partial => "soap"
```

This example uses the `render_to_string` helper to set the variable `@soap_request`. The `@` symbol indicates that the variable is an *instance variable*. Instance variables can be used by other Rails functions and classes. A partial template called `soap` is rendered. Partial templates are rendered without the controller layout you created earlier in the chapter. You haven't created the `soap` template yet; you'll do that in the next step.

```
@calculation_result = call_calculation_service
@payments = @calculation_result["TRANSACTION"]["RESPONSE"]["PAYMENTSTREAM"]
```

The actual SOAP interaction will be done in the `call_calculation_service` function. The result of that function will contain a set of SOAP mappings which correspond to the returned XML response. The `PAYMENTSTREAM` mappings can be used to display each payment back to the

user. Again, these are saved as an instance variable so that they can be used in other functions and templates.

```
rescue Exception => e
  flash.now[:error] = "There was an error communicating with the service"
end
```

Whenever there is an error communicating with the web service, it is caught and added to the `flash` hash. Using `flash.now` ensures that the message will be removed from the hash after the next render. The exception message is not included in the string that is added to the flash message for security reasons. (In production sites you would probably want to provide a more descriptive message.)

```
render :partial => "calculate", :layout => true
```

Finally, the method renders a partial template called `calculate`. The only reason to use a partial template instead of letting the controller render the default template for the action is for later reuse. Because partial templates do not include the layout by default, you need to explicitly specify that you want the layout included. You haven't created the `calculate` template yet either; you'll do that in a couple more steps. Save the file.

2. The `calculate` method tries to build the XML for the SOAP request using a template. Unlike the other templates used in this chapter, it will need to be a Rails XML (RXML) template. Create a new file called `_soap.rxml` inside of the `app/views/calculator` folder. The file needs the leading underscore because it is a partial template. You can construct the XML using the `XmlMarkup` builder called `xml`, which is built into every RXML template. The builder interprets unknown method names as element creation calls. This is a form of meta-programming. It means that you can call `xml.TRANSACTION` to create a `<TRANSACTION>` element. Some of the elements need to contain other elements. To support this, the builder allows you to define the element using the meta-programming method and a `do..end` block. You can also pass a list of attributes and values as options to the meta-functions. This makes creating the XML very simple:

```
xml.TRANSACTION do
  xml.REQUEST do
    xml.REQUESTOPTIONS(
      "PaymentStreamAndApr" => "true",
      "ReverseApr" => "false",
      "ManualPaymentStream" => "false",
      "AdditionalPrincipal" => "false") do
      xml.DATA(
        "PaymentStreamRequestType" => "Long",
        "AllowOddLastPayment" => "true",
        "DaysPerYear" => "360",
        "ConstructionTILType" => "Seperate",
        "AprIterations" => "1",
        "FeeIterations" => "0")
      end
    end
    xml.APRREQUEST "TotalAPRFeesAmount" => "#{params[:total_apr_fees_amount]}"

    if (params[:index_value] && params[:index_value].to_f > 0)
      xml.INDEXVALUES(
        "IndexValue" => "#{params[:index_value]}",
        "IndexMonths" => "#{params[:loan_original_maturity_term_months]}")
    end
  end
end
```

```
if (params[:margin_value] && params[:margin_value].to_f > 0)
  xml.MARGINVALUES(
    "MarginValue" => "#{params[:margin_value]}",
    "MarginMonths" => "#{params[:loan_original_maturity_term_months]}")
end

xml.LOANDATA do
  xml.TERMS(
    "ProgramName" => "#{params[:program_name]}",
    "LoanOriginationSystemLoanIdentifier" => "Beginning XML Calculator",
    "OriginalLoanAmount" => "#{params[:original_loan_amount]}",
    "DisclosedTotalSalesPriceAmount" =>
      "#{params[:disclosed_total_sales_price_amount]}",
    "PropertyAppraisedValueAmount" =>
      "#{params[:property_appraised_value_amount]}",
    "NoteRatePercent" => "#{params[:note_rate_percent]}",
    "InitialPaymentRatePercent" => "#{params[:note_rate_percent]}",
    "LoanOriginalMaturityTermMonths" =>
      "#{params[:loan_original_maturity_term_months]}",
    "ApplicationSignedDate" => "2007-01-15",
    "LoanEstimatedClosingDate" => "2007-01-15",
    "ScheduledFirstPaymentDate" => "2007-01-15",
    "EstimatedPrepaidDays" => "15")
  end
end
end
```

Notice that most of the attribute values are strings that contain the interpolation evaluator `#{}` . Ruby will evaluate everything inside of the `#{ }` braces and output a string in its place. This allows you to insert values from the `params` hash that were submitted by the loan calculator form. (Again, this case study does not define these mortgage concepts in detail; more information about each of these terms can be found on the Compliance Studio website or at <http://en.wikipedia.org/wiki/Mortgage>.) Save the file.

3. Before you define the `call_calculation_service` method, we should take a small detour. Ruby on Rails comes with a default SOAP library called `soap4r`. The library makes SOAP communications very simple. Accessing the elements in the SOAP response is also very easy. Unfortunately though, it doesn't provide friendly methods to retrieve attribute values from the SOAP response. Luckily, Ruby allows you to extend built-in classes to add new functionality. Create a new file called `soap_extensions.rb` in the `lib` folder which was generated inside your main `loancalculator` folder. Start by including the existing SOAP and XML Parser libraries:

```
require 'soap/wsdlDriver'
require 'xsd/xmlparser/rexmlparser'
```

Next, declare the new module. In this case you can call it `SOAP::Mapping::Extensions`. You could actually call it anything you like, but having a more expressive name is considered good practice:

```
module SOAP::Mapping::Extensions
```

Inside the module, create a function to retrieve an attribute value by namespace URI and attribute name:

```
def attribute_value(uri, name)
  __xmlattr[XSD::QName.new(uri, name)]
end
```

This function creates a `QName` object for the `uri` and `name` parameters and passes it to the fundamental method `__xmlattr` which already exists inside of the `SOAP::Mapping::Object` class. The result of the `__xmlattr` method is returned as the result of the function. If the attribute is not found, the function will return `nil`.

To make the extension module even more useful, you could also add a meta-programming method by overriding the `method_missing` function:

```
def method_missing(method_id, *arguments, &block)
  attr = attribute_value(nil, method_id.to_s)
  attr || super
end
```

If you try to call a method that doesn't exist, the `method_missing` function will be executed. The missing method name is passed to the new `attribute_value` function as an attribute name and assigned to the `attr` variable. The last line in the function returns the `attr` variable if it is not `nil`, otherwise it calls the inherited `method_missing` function by using the `super` keyword. You could continue to add extension functions but that's all that is needed for this example. Add the module `end` statement:

```
end
```

Of course, this module isn't actually part of the `SOAP` module yet. You need to tell Ruby to include the methods in the new module inside of the `SOAP::Mapping::Object` class. At the end of the file (after the module `end` statement) add the following:

```
SOAP::Mapping::Object.send( :include, SOAP::Mapping::Extensions )
```

This tells Ruby to include the methods in your new extension module inside of `SOAP::Mapping::Object`.

4. With the extensions module completed you can now implement the `call_calculation_service` and communicate with the Compliance Studio web service. Add this as a private method of the `CalculatorController` class inside of the `calculator_controller.rb` file. Just before the end of the class add the `private` keyword, and then below that define the method:

```
private
def call_calculation_service
  require "soap_extensions"
  endpoint_url = "http://compliancestudio.com/apr/1.0/service.asmx?WSDL"
  factory = SOAP::WSDLDriverFactory.new(endpoint_url)
  compliance_studio_service = factory.create_rpc_driver
  defined_elements =
    compliance_studio_service.proxy.literal_mapping_registry.definedelements
  defined_elements.delete(defined_elements.find_name("RequestCalculation"))
  param_name = XSD::QName.new("http://compliancestudio.com/apr/1.0",
    "TransactionEnvelope")
  result = compliance_studio_service.RequestCalculation(param_name =>
```

```
@soap_request)
  result["RequestCalculationResult"]
end
```

Let's break down each part of the method. At the start of the function you included the new `soap_extensions` module.

```
require "soap_extensions"
```

The SOAP classes are created dynamically based on the Web Service Description at Compliance Studio.

```
endpoint_url = "http://compliancestudio.com/apr/1.0/service.asmx?WSDL"
factory = SOAP::WSDLDriverFactory.new(endpoint_url)
compliance_studio_service = factory.create_rpc_driver
```

Creating the service wrapper dynamically allows your application to deal with changes more gracefully. Unfortunately, there is a significant performance penalty in dynamically creating the service. In a production application you would want to generate the service class once and include it as a library file. This can be done using the `wsdl2ruby` library.

The built-in SOAP classes cannot send arbitrary XML parameters by default. Many .NET based web services use this style of parameter passing. To get around this you need to remove the default definition for the `RequestCalculation` element:

```
defined_elements =
  compliance_studio_service.proxy.literal_mapping_registry.definedelements
defined_elements.delete(defined_elements.find_name("RequestCalculation"))
```

Finally, the function calls the remote method `RequestCalculation`, with the `TransactionEnvelope` parameter set to the `@soap_request` instance variable that was created earlier.

```
param_name = XSD::QName.new("http://compliancestudio.com/apr/1.0",
  "TransactionEnvelope")
result = compliance_studio_service.RequestCalculation(param_name =>
  @soap_request)
result["RequestCalculationResult"]
```

The mapped object `RequestCalculationResult` is returned as the result of the method. Save the controller.

5. With all of the main functionality completed, all that's left is to display the payments. To do this you need to create a partial template for the `calculate` action. Create a new file called `_calculate.rhtml` inside of the `app/views/calculator` folder (remember, the leading underscore indicates it is a partial template). The template should begin by displaying any error messages that were appended to the `flash` hash.

```
<p class="flash"><%= flash[:error] %></p>
```

As with the `index.rhtml` template, you want to include a sub-heading:

```
<h2><span>Payments</span></h2>
```

Next, create a `<div>` to hold the payments. If there are no payments listed, a friendly message should be displayed asking the user to submit a new request, otherwise the payments, payment dates, and ending balances should be displayed:

```
<div id="payments_table">
  <% if @payments && @payments.length > 0 %>
    <table border="0" cellpadding="0" cellspacing="0">
      <tr>
        <th>Payment Date</th>
        <th>Payment Amount</th>
        <th>Remaining Balance</th>
      </tr>
      <% for payment in @payments do %>
        <tr class="<%= cycle("even_row", "odd_row") -%>">
          <td>
            <%= payment.PmtDate %>
          </td>
          <td class="numeric_cell">
            <%= number_to_currency(payment.PmtTotal) %>
          </td>
          <td class="numeric_cell">
            <%= number_to_currency(payment.PmtEndingBalance) %>
          </td>
        </tr>
      <% end %>
    </table>
  <% else %>
    Please submit a request and the payments will be displayed.
  <% end %>
</div>
```

Here the rows for the payment table are constructed using a `for` loop. The `class` attribute of the `<tr>` element is set using the helper function `cycle`. The `cycle` function alternates between two options. The first time it is called it will return `even_row`, the second time `odd_row`. If you define CSS styles for the `even_row` and `odd_row` classes the payment rows can be displayed using alternating colors.

The `number_to_currency` helper function allows you to easily format floating point numbers as strings. The default uses dollars and cents, but the units and precision can be modified to make it compatible with other kinds of currency. Each `payment` object is a `SOAP::Mapping::Object` class. The attributes `PmtDate`, `PmtTotal`, and `PmtEndingBalance` are accessed using the meta-programming extension you created in your `SOAP::Mapping::Extensions` module.

Save the template.

6. At this point, you should be able to use the loan calculator form to find out the payments for a specific loan. Because you have added a new library, you need to restart the built-in web server. Once it is running, open your browser to `http://localhost:3000/calculator`. Fill in the form with the values shown in Figure 22-4 or make up your own.

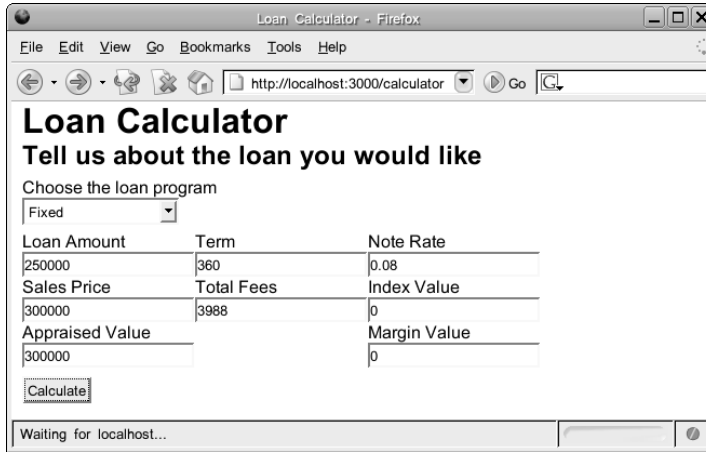


Figure 22-4

After inputting the values, click the Calculate button. You should see the output shown in Figure 22-5.

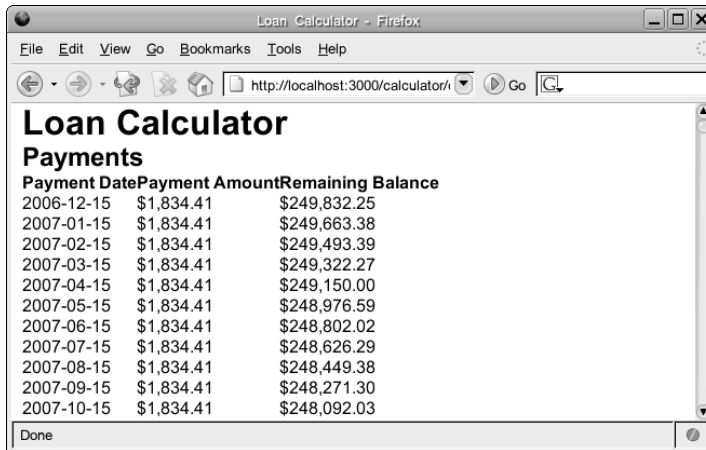


Figure 22-5

If you don't see the expected output you have probably left an empty value. The error messages that come back from the application are very cryptic and difficult to solve. Check the Rails logs for more detailed information if you see an error, and try to correct the values.

How It Works

In this Try It Out, you completed the basic functionality of the loan calculator. The form now calls the `calculate` action, which builds an XML request sends it to the Compliance Studio service and displays the result. In addition to the view templates and actual controller code, you added a custom extension

module to simplify access to attribute values in the response. Ruby's extensibility is one of the things that makes Rails so powerful.

Before you add more functionality, you can add styles for the payments in the `loancalculator.css` stylesheet.

Try It Out Improving the Look of the Payments

Like the other styles in this chapter, the payment style declarations will be fairly simple. Again, in professional sites you may have a team of designers to modify the style definitions.

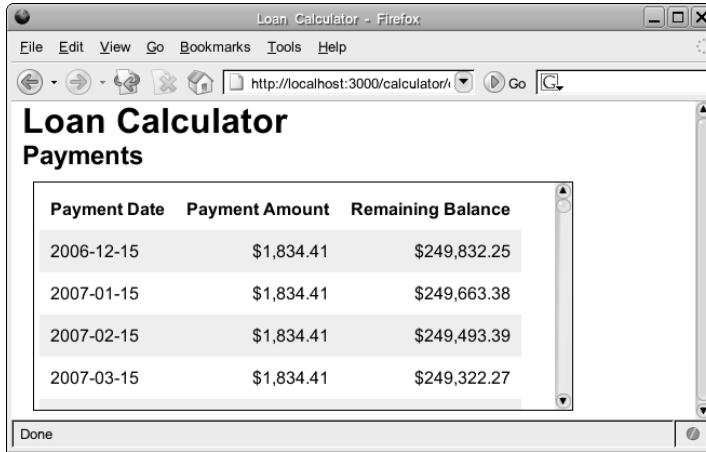
1. Open your text editor, and open the stylesheet document `public/stylesheet/calculator.css`. The `public/stylesheet` folder should have been generated by the `rails` command and can be found inside of the `loancalculator` folder. Add the following declarations to the end of the stylesheet:

```
#payments_table {
  overflow:auto;
  margin:10px;
  padding:5px;
  width:488px;
  height:200px;
  border:1px solid black;
}
#payments_table td, #payments_table th{
  margin:0px;
  padding:10px;
}
.numeric_cell {
  text-align:right;
}
.even_row {
  color:black;
  background-color:#eee;
}
```

Notice that the `overflow` property of the `payments_table` template is set to `auto`. Each loan can have a lot of payments, so instead of causing the page to become very long; the value `auto` means that the contents of the `payments_table <div>` element will scroll. Also, there is a style for the `even_row` class. This class is inserted using the `cycle` helper and is used by alternating table rows. Setting the background to light gray (`#eee`) gives the table a spreadsheet look. Once you have added the new declarations, save the stylesheet.

How It Works

In this Try It Out, you added declarations to the CSS stylesheet for the loan calculator. The enhanced stylesheet improved the look of the payments table. Again, the declarations that were added are fairly simple and can easily be improved. You can submit the calculate form again and you should see resulting payments with improved style, as in Figure 22-6.



Payment Date	Payment Amount	Remaining Balance
2006-12-15	\$1,834.41	\$249,832.25
2007-01-15	\$1,834.41	\$249,663.38
2007-02-15	\$1,834.41	\$249,493.39
2007-03-15	\$1,834.41	\$249,322.27

Figure 22-6

Adding Ajax Support

Asynchronous JavaScript and XML, or *Ajax*, has been growing in popularity over the past few years. In fact, Ajax has been possible in browsers for much longer than that. In the loan calculator you can use Ajax to call the `calculate` action you built in the previous examples. When the user clicks Calculate, instead of changing pages, the payment table can be inserted into the current page. This kind of behavior allows you to create very advanced web applications that improve the user experience.

Unfortunately, at the time of this writing, each of the major browsers provides different mechanisms for using XML over HTTP. To get around this, you will need to use a custom library that hides the differences. Though there are many JavaScript libraries that do this, Ruby on Rails has built-in support for the Prototype library. Because of this you should use it within the loan calculator. You will also want to use the Script.aculo.us JavaScript library to include advanced visual effects. The library is included with Rails but more information about it can be found at <http://script.aculo.us/>.

You may be wondering why we spent time building a traditional web application prior to building the Ajax version. Some browsers do not support JavaScript. Sometimes users have turned off their JavaScript support for security reasons. Because of this it is important to have a non-JavaScript fallback for your application. Even though you will be adding JavaScript functionality to the loan calculator, the existing functionality will still be in place for users that do not have JavaScript support.

Try It Out Adding Ajax to the Loan Calculator

In this Try It Out, you make a number of changes to enable Ajax in the loan calculator. These include making sure your layout includes the Prototype and Script.aculo.us libraries, modifying the loan calculator form, and displaying a status indicator to the user.

1. From your text editor, open the `calculator.rhtml` layout document in the folder `app/views/layouts`. You need to include the Prototype and Script.aculo.us JavaScript libraries in the `<head>` element. Again, Ruby on Rails has built-in support for these libraries, so you can simply instruct the layout to include the JavaScript defaults. Add the highlighted line:

```
<head>
  <title>Loan Calculator</title>
  <meta http-equiv="content-type" content="text/html; charset=iso-8859-1" />
  <%= stylesheet_link_tag 'calculator' %>
  <%= javascript_include_tag :defaults %>
</head>
```

2. Next, update the `index.rhtml` template in the folder `app/views/calculator`. Instead of using the `form_tag` helper, change it to use the `form_remote_tag` helper. The Rails framework considers Ajax forms “remote” forms because they will handle the `POST` remotely. Replace the existing `form_tag` with the following:

```
<% form_remote_tag :url => { :action => :calculate },
  :before => visual_effect(:appear, :working, :duration => 1.2 ),
  :complete => visual_effect(:fade, :working, :duration => 1.2 ),
  :update => :payments do %>
```

The `form_remote_tag` helper is quite a bit different from the original `form_tag` helper. Instead of including the `action` directly, it has been included as part of the `url` parameter. By default, when the form is submitted it will attempt to send a `POST` to the `url`. If JavaScript is not enabled it will use a traditional `POST` instead. This example also includes parameters to handle event callbacks. The `form_remote_tag` actually provides quite a few callback events; however, the `before`, `complete` and `update` are the most common.

```
:before => visual_effect(:appear, :working, :duration => 1.2 )
```

The `before` callback allows you to include JavaScript to be executed before the remote action is called. Instead of including JavaScript directly, this example uses the `visual_effect` helper that is part of the Script.aculo.us library. The `visual_effect` function will generate the JavaScript code to make the specified element appear over a duration of 1.2 seconds. In this case the specified element is called `working`. The Script.aculo.us library will look for an element in the current document with the `id` attribute set to `working`. Use this to provide an indication that the remote call is being performed. When the remote call completes it will perform another `visual_effect` to hide the `working` element:

```
:complete => visual_effect(:fade, :working, :duration => 1.2 )
```

Finally, you need to specify what the `form_remote_tag` should do with the response it receives from the `calculate` action:

```
:update => :payments
```

In this case, the `form_remote_tag` will update the specified element `payments` using the response. The helper will generate JavaScript to find an element in the current document with the `id` attribute set to `payments` and will update its inner HTML with the HTML response from the `calculate` action.

2. Add a new element that has an `id` attribute set to `working`. You can put it just after the submit button:

```
<div id="buttons">
  <%= submit_tag "Calculate" %>
  <span id="working" style="display:none">
    <%= image_tag "indicator.gif", :alt => "Working..." %>
  </span>
</div>
```

Here the `` element has its `id` attribute set to `working`. This means that the code generated by the `visual_effect` helper will be able to modify its `display`. When the page is loaded the form isn't doing any communications, so you should default the `display` style to `none`. Though you could do this in your external stylesheet, the `Script.aculo.us` library cannot override an external style definition.

Inside the `` element is an `image_tag` helper for the image `indicator.gif`. The image, which can be downloaded with the rest of the code at <http://www.wrox.com/>, was designed by Jakob Skjerning and is available at <http://mentalized.net/activity-indicators/> along with several other public domain indicator graphics. Download the image and save it in the `public/images` folder inside of your `loancalculator` folder. If the image can't be found, or if the user does not have support for images, the `alt` text will be displayed.

3. You also need a placeholder for the payments in the calculation response. Include an empty `<div>` element just after the end of the template:

```
<div id="payments"></div>
```

Again, the element's `id` attribute should be set to `payments` so that the `update` callback in the `form_remote_tag` can find it.

4. You should be able to see Ajax in action. Make sure that the built-in web server is running, fill in the form, and click Calculate (see Figure 22-7).

Loan Calculator
Tell us about the loan you would like

Choose the loan program
Fixed

Loan Amount	Term	Note Rate
250000	360	0.08
Sales Price	Total Fees	Index Value
300000	3988	0
Appraised Value		Margin Value
300000		0

Calculate

Loan Calculator
Payments

Payment Date	Payment Amount	Remaining Balance
2006-12-15	\$1,834.41	\$249,832.25
2007-01-15	\$1,834.41	\$249,663.38
2007-02-15	\$1,834.41	\$249,493.39
2007-03-15	\$1,834.41	\$249,322.27

Done

Figure 22-7

How It Works

In this Try It Out, you added Ajax support to the loan calculator. Instead of using a traditional form, you used a remote form. When the user clicks Calculate, an Ajax request is created and the contents of the form are passed to the server without the page changing. When the response is returned it is dynamically inserted into the page. In addition to the basic communications, the `form_remote_tag` helper also let you perform specialized callbacks to make an indicator fade in and fade out. Note that the loan calculator will still work even when the user has no JavaScript support. You could test this by temporarily disabling your JavaScript support in your browser. The loan calculator should revert to the traditional form behavior.

Chapter 22: Case Study: Payment Calculator — Ruby on Rails

You may have noticed that the “Loan Calculator” heading appeared twice on the page. This is because the `calculate` action rendered its response using the default controller layout. This is important when the response is displayed by itself, but when it is being accessed by a remote form through Ajax it should be left out. You can do this by making a small change to the `CalculatorController` class.

Try It Out Removing the Layout in the Ajax Response

You’ll need to modify the `calculate` method inside of the `CalculatorController`. If the incoming request is an Ajax request, you need to render the `_calculate.rhtml` template without the default layout, otherwise the method should render the template normally.

1. Open your text editor, and open the `calculator_controller.rb` document in the folder `app/controllers`. Add the highlighted code to the end of the `calculate` method:

```
def calculate
  begin
    @soap_request = render_to_string :action => 'soap', :layout => false
    @calculation_result = call_calculation_service
    @payments = @calculation_result["TRANSACTION"]["RESPONSE"]["PAYMENTSTREAM"]
  rescue Exception => e
    flash.now[:error] = "There was an error communicating with the service"
  end
  if request.xhr?
    render :partial => "calculate"
  else
    render :partial => "calculate", :layout => true
  end
end
```

Each method within the controller has access to the `request` object. The `request` object contains important information about the HTTP request and also a number of helper methods. The `xhr?` method will return `true` if the request is an Ajax request and `false` otherwise. The “`xhr`” refers to the `XMLHttpRequest` object, which is used to make Ajax requests from web browsers. Though you haven’t explicitly used this object in your code, the Prototype library which is used by the `form_remote_tag` uses it to send Ajax requests to the web server.

If the `xhr?` method returns `true` then the `calculate` method will render without the layout (partials do not include the layout by default). If the `xhr?` method returns `false`, you need to render the partial template with the layout included.

Save the file.

How It Works

In this Try It Out, you made sure that the `calculate` action didn’t render the response using a layout if the request was made using Ajax. This fixed the problem of the duplicated “Loan Calculator” heading. If you input the loan details and click Calculate again, the duplicate heading is removed (see Figure 22-8).

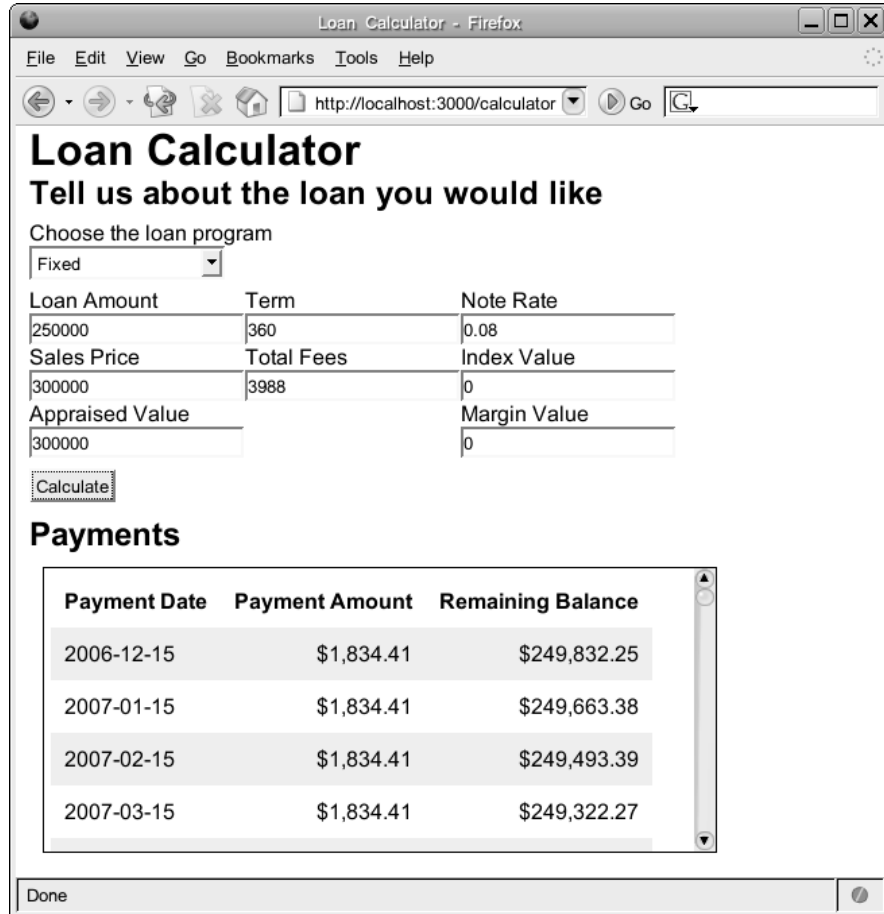


Figure 22-8

Enhancing the Display with SVG

Displaying the payments and ending balance in a table is very useful, but by using a chart of the balance over time, you can quickly compare the various loan programs visually. You can use Scalable Vector Graphics to construct the chart and display it in the loan calculator web page. Again, for your users to see the SVG graphic, they will need to have a browser that supports SVG or an SVG plug-in.

To build the chart, you need to:

1. Create a base SVG document for the chart.
2. Display the SVG in an `<iframe>` element.
3. Use JavaScript to replace the SVG and assign the chart data.

Notice that an `<iframe>` is used to display the SVG content. Ideally, you could include the `<svg>` elements directly in an XHTML page and, using namespaces, the browser would know how to render the document. At the time of this writing, however, that isn't possible in some of the major browsers. In fact, because Internet Explorer relies on a plugin to render SVG content, `<iframe>` is the only standard cross-browser solution. Though, the `<object>` tag is another solution, support for it is lacking in older browsers.

Because of this, you need to create an SVG document that can be loaded into the `<iframe>` element when the page is loaded. Manipulating the contents of an `<iframe>` through JavaScript (especially when that frame has an ActiveX object in its content) can be cumbersome. The .NET version of this chapter shows how the parent XHTML document and SVG can communicate with one another. In this chapter, however, you will instead use JavaScript and Ajax to replace the entire SVG document. This greatly simplifies the code and more easily integrates with Rails using Remote JavaScript (RJS) templates.

Try It Out Creating a Base SVG Chart

In this Try It Out, you build the base for the loan balance chart that will be displayed in the loan calculator web page. You will add an `<iframe>` to the `index.rhtml` template, define a new action for the chart inside of the controller, and create an SVG template that can display the chart.

1. Open the `index.rhtml` file in the folder `app/views/calculator` to add an `<iframe>` element that refers to the chart to the bottom of the template. Add the highlighted code and then save the file:

```
<div id="chart">
  <h3><span>Lending Balance</span></h3>
  <iframe src="<%= url_for :action => :chart -%>" width="650" height="260"
    frameborder="0" id="chart_frame" name="chart_frame">
    <p>It looks like your browser doesn't support frames</p>
  </iframe>
</div>
```

The `<iframe>` uses the `url_for` helper to generate the `src` URL for the SVG. In this case, all that is needed is to point to a new action inside of the current controller called `chart`. In addition to the URL, you should specify `width` and `height` parameters and clear the frame border. You'll need to refer to the frame from JavaScript later, so it is a good idea to provide `name` and `id` attributes. Just as you saw with JavaScript and images, you need to provide a fallback in case the user's browser doesn't support frames. In this case the fallback isn't very useful, just an indication that certain functionality has been turned off because the browser doesn't support frames.

2. Create the new `chart` action that the `<iframe>` refers to. Open the `calculator_controller.rb` file in the folder `app/controllers`. You will need to add a new method called `chart`. The method cannot be private so make sure that you put it above the `private` keyword in the class. Add the following:

```
def chart
  render :partial => "chart", :content_type => "image/svg+xml"
end
```

Again, because you are rendering a partial template, you don't need to specify that you don't want the template to include the layout. Also, in order for the SVG to be viewable in your browser (if you have the correct plug-in installed or if you are using a browser that supports SVG natively), you must specify the content as the SVG MIME ("image/svg+xml"). Some web servers do not have this configured by default so it is useful to send within your Rails application. For more information, see http://wiki.svg.org/MIME_Type.

Save the controller.

3. The `chart` action will attempt to render a view template called `_chart.rhtml`. Because SVG is an XML format, you can simply include the SVG information in the template. You can also use the built in Rails helpers as you would in any other template. The file extension itself is not important because you are explicitly setting the content-type to `image/svg+xml`. Create a new file called `_chart.rhtml` in the `app/views/calculator` folder.

Begin creating this chart as you would begin any SVG document. Include a `<title>` element even though it won't be displayed:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <g id="contents">
    <title>Loan Balance over Time</title>
    <g id="grid">
      <rect x="10" y="10" width="500" height="200" fill="#f0f0f0"
        stroke="#000" stroke-width="2"/>
      <path stroke="#ddd" d="M11,60 1 498,0
                            M11,110 1 498,0
                            M11,160 1 498,0
                            M11,210 1 498,0"/>
    </g>
  </g>
</svg>
```

The SVG is very simple; it has one main group element (`<g>`) with its `id` set to `contents`. It also includes a grid for the data. The grid is drawn using a simple `<rect>` and a `<path>` which draws four lines. In a more complete application, you might enhance this with a decorative gradient or other design features. Save the chart.

How It Works

In this Try It Out, you built a basic chart using SVG. The SVG is included as part of the `index.rhtml` template using an `<iframe>` element. Like the other pieces of loan calculator, you defined the SVG using a partial template and within the `chart` method, rendered it with a specific content-type. If you open your browser to `http://localhost:3000/calculator` (or refresh if necessary), you should see the rendered chart (see Figure 22-9).

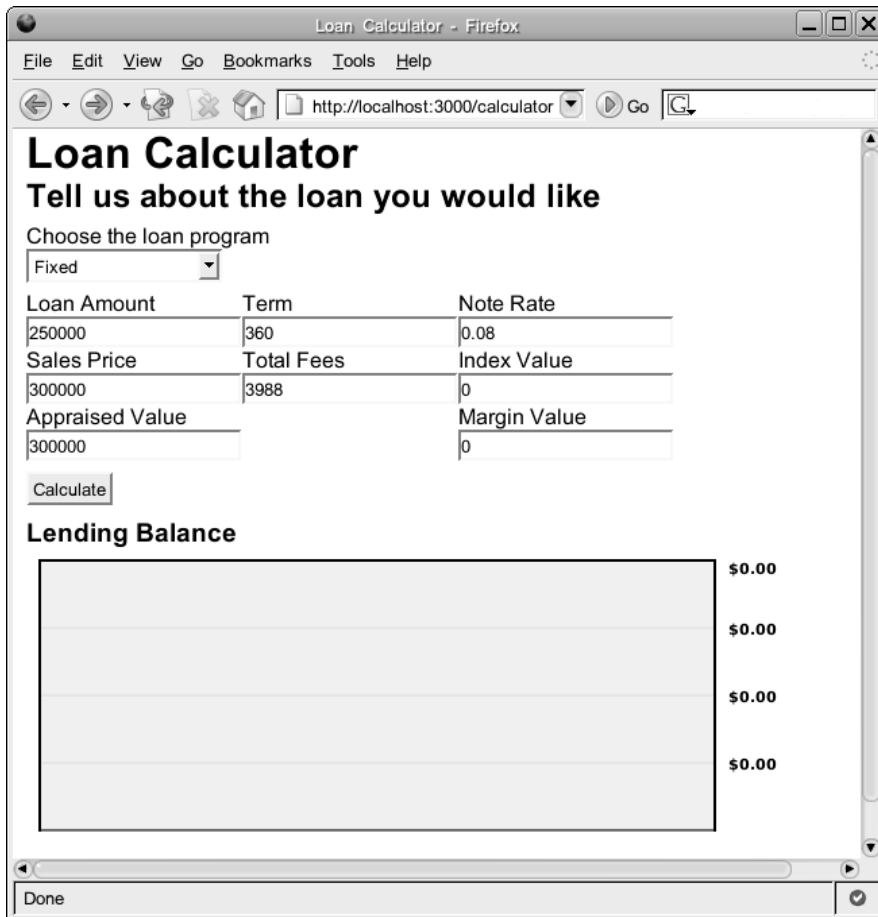


Figure 22-9

If you click Calculate, you will see the payments appear but the chart itself won't change. You'll need to add some additional code for this. With the basic functionality complete, you can integrate the chart with the rest of the Ajax callbacks.

Try It Out Bringing the Chart to Life

In this Try It Out, you add data to the chart and have it update whenever the payments are calculated. To do this you will need to utilize an RJS template, add a JavaScript function to replace the contents of the SVG document, and modify your `_chart.rhtml` template.

1. Currently, the `form_remote_tag` inside of the `index.rhtml` template calls the `calculate` action and updates the `payments` element using the response:

```
<% form_remote_tag :url => { :action => :calculate },
  :before => visual_effect(:appear, :working, :duration => 1.2 ),
  :complete => visual_effect(:fade, :working, :duration => 1.2 ),
  :update => :payments do %>
```

Unfortunately, you need the `form_remote_tag` to update the `payments` element *and* the SVG chart. You can't do this with a single HTML response. Luckily, the Rails framework allows you to define Remote JavaScript that can be generated on the server and passed back as a single response to be executed in the user's browser. When using RJS, you don't need specify an update action in your `form_remote_tag`; instead you handle the replacement in the `calculate` action in your controller. Modify the `form_remote_tag` as follows:

```
<% form_remote_tag :url => { :action => :calculate },
  :before => visual_effect(:appear, :working, :duration => 1.2 ),
  :complete => visual_effect(:fade, :working, :duration => 1.2 ) do %>
```

The Prototype JavaScript library doesn't have support for working with SVG content, so you need to create a special JavaScript function that can be used to replace the SVG. You can add another `<script>` element to the end of the `index.rhtml` template as follows:

```
<script type="text/javascript">
function replace_svg(content) {
  // try to retrieve the frame
  var frame = frames['chart_frame'];
  if (!frame) return;
  var frame_doc = frame.document;
  // parse the content
  try {
    var doc = null;
    if (DOMParser) {
      doc = new DOMParser().parseFromString(content, "text/xml");
    } else if (frames['chart_frame'].parseXML) {
      doc = frames['chart_frame'].parseXML(content, null);
    }
    // replace the content
    var old_contents = frame_doc.getElementById("contents");
    var new_contents = doc.getElementById("contents");
    new_contents = frame_doc.importNode(new_contents, true);
    frame_doc.documentElement.replaceChild(new_contents, old_contents);
  } catch(e) {
    alert("There was an error attempting to update the chart");
  }
}
</script>
```

The `replace_svg` function expects a single parameter called `content`. The first part of the function attempts to find the frame for the chart in the current document:

```
var frame = frames['chart_frame'];
if (!frame) return;
var frame_doc = frame.document;
```

Once the frame is retrieved the function parses the content:

```
if (DOMParser) {
  doc = new DOMParser().parseFromString(content, "text/xml");
} else if (frames['chart_frame'].parseXML) {
  doc = frames['chart_frame'].parseXML(content, null);
}
```

Chapter 22: Case Study: Payment Calculator — Ruby on Rails

Unfortunately, the methods for parsing a string as XML aren't standardized across all browsers. In this function there are two ways to parse the incoming content: either using a `DOMParser` object if it exists or by using the `parseXML` function if it exists. The `DOMParser` will be used in Firefox and Opera browsers, while the `parseXML` method will be used by the Adobe SVG plugin within Internet Explorer.

Replacing the entire document can have side-effects in some browsers. To get around this you can instead replace the `contents` group within the SVG document. To do this, import the newly parsed XML into the old document's object hierarchy using the `importNode` DOM method and then call `replaceChild`:

```
var old_contents = frame_doc.getElementById("contents");
var new_contents = doc.getElementById("contents");
new_contents = frame_doc.importNode(new_contents, true);
frame_doc.documentElement.replaceChild(new_contents, old_contents);
```

All of the code is safely wrapped inside of a `try..catch` block to handle any errors.

2. You also need to modify the `calculator_controller.rb` file inside of the `app/controllers` folder. The `calculate` method will need to respond to Ajax requests using RJS instead of an RHTML template. Additionally, you can calculate the maximum ending balance for the payment stream. This will be useful when updating the chart template in the next step. Add the highlighted code:

```
def calculate
  begin
    @soap_request = render_to_string :partial => "soap"
    @calculation_result = call_calculation_service
    @payments = @calculation_result["TRANSACTION"]["RESPONSE"]["PAYMENTSTREAM"]
    @max_balance = @payments.max { |a,b|
      a.PmtEndingBalance.to_f <=> b.PmtEndingBalance.to_f
    }.PmtEndingBalance.to_f
  rescue Exception => e
    flash.now[:error] = "There was an error communicating with the service"
  end
  if request.xhr?
    render :update do |page|
      page.replace_html "payments", :partial => "calculate"
      page.call "replace_svg", page.send(:render, {:partial => "chart"})
    end
  else
    render :partial => "calculate", :layout => true
  end
end
```

The maximum ending balance is assigned to the instance variable `@max_balance`:

```
@max_balance = @payments.max { |a,b|
  a.PmtEndingBalance.to_f <=> b.PmtEndingBalance.to_f
}.PmtEndingBalance.to_f
```

The maximum was determined using a bit of Ruby magic. In Ruby the `array` class has a `max` function that requires a comparison block. Inside the block you can write a comparison that will be used for all of the items in the array to determine the one with the highest value. The `max` function returns the actual item from the array, so at the end of the function you need to again

convert the `PmtEndingBalance` attribute to a floating point number. Though this code could have been included in the chart template itself, it is best to include as little code in your views as possible.

The function no longer renders a single template in response to Ajax requests. Instead an inline RJS block is rendered:

```
render :update do |page|
  page.replace_html "payments", :partial => "calculate"
  page.call "replace_svg", page.send(:render, {:partial => "chart"})
end
```

Inside the block, two RJS commands are executed on the `page` variable. The first is very similar to the `:update` parameter that was part of the `form_remote_for` helper. An element with the `id` attribute `payments` has its HTML content replaced with the output of the rendered partial template `calculate`. The second RJS line is similar. The custom JavaScript function `replace_svg` that you added to the `index.rhtml` template is called. Remember that the `replace_svg` function expected a single string argument which contained the XML for the SVG. Unfortunately you cannot generate the string using the `render_to_string` function you used earlier in the chapter. In the current version of Rails the `render_to_string` function conflicts with the RJS response. Instead you need to call the `page` object's internal `render` method which will return a string.

Save the changes to the controller.

3. Finally, modify the `_chart.rhtml` template inside of the `app/views/calculator` folder. The beginning of the SVG will remain the same:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <g id="contents">
    <title>Loan Balance over Time</title>
    <g id="grid">
      <rect x="10" y="10" width="500" height="200" fill="#f0f0f0"
        stroke="#000" stroke-width="2"/>
      <path stroke="#ddd" d="M11,60 l 498,0
                          M11,110 l 498,0
                          M11,160 l 498,0
                          M11,210 l 498,0"/>
    </g>
  </g>
```

Following the initial grid group, you begin outputting the data. Of course, you only want to do this if the `@max_balance` is greater than 0 and if the `@payments` array is not blank:

```
<% if @max_balance && !@payments.blank? %>
```

Inside the block, start with a couple of calculations:

```
<% payment_width = 500.to_f / @payments.length %>
<% balance_height = 200.to_f / @max_balance %>
```

In order to accurately display the chart data you will need to scale the information to the existing chart. To do this you can divide the width of the chart (500) by the number of payments (`@payments.length`). This way the information for each payment can be spaced to fill the entire width of the chart. A similar calculation is made to determine the amount of height that should be used when plotting the ending balance for each payment.

Chapter 22: Case Study: Payment Calculator — Ruby on Rails

Why include calculations like this in the view? In most cases it is better to include calculations like this in the controller instead of the view template. However, because the calculation relies on the actual height and width of the chart (500 and 200 respectively), it should be kept inside the view. This way if the chart dimensions are modified later, the calculation can quickly be updated as well.

With the `payment_width` and `balance_height` determined you can plot the information on the chart:

```
<g id="data" transform="translate(11, 10)">
  <path id="balance" fill="#77c" stroke="#aaf" fill-opacity="0.3"
    transform="scale(<%=payment_width%>, <%=balance_height%>)"
    d="M 0,<%=@max_balance%>
      <% @payments.each_with_index do |payment, i| %>
        L <%=i%>,<%=@max_balance-payment.PmtEndingBalance.to_f%>
      <% end %>
      L <%=@payments.length%>,<%=@max_balance%> z" />
</g>
```

The data is plotted using a `<path>` element, which is placed inside of a `<g>` element for convenience. The `<path>` itself is transformed using a `scale` command. The data is scaled using the `payment_width` and `balance_height` variables. The data for the `<path>` element begins with the move command (`M`) and moves the cursor to the point `0, @max_balance`. In this example the `@max_balance` variable will have the value 249832.25. But the chart is only 200 pixels high! The `scale` command included in the `transform` attribute actually modifies the coordinate system of the `<path>` such that, when scaled the coordinate `(0, 249832.25)` will place the cursor in the lower left corner of the chart.

The rest of the data is drawn by looping through the `@payments` array and drawing a line (`L`) from the current point to a point with an `X` position equal to the index in the array and a `Y` position which is determined by the payment ending balance. Remember, because of the scale, the payments will be equally spaced across the chart.

A final line (`L`) is added to the end of the chart which places the cursor in the lower right corner of the chart. The close-path command (`z`) completes the data.

In order for users to understand the data in the chart, you need to provide informative labels. You can do this by adding `<text>` labels in the SVG:

```
<g id="balance_labels" font-size="8pt" font-weight="bold">
  <text id="balance100Percent" x="520" y="20">
    <%= number_to_currency(@max_balance) %>
  </text>
  <text id="balance75Percent" x="520" y="65">
    <%= number_to_currency(0.75 * @max_balance) %>
  </text>
  <text id="balance50Percent" x="520" y="115">
    <%= number_to_currency(0.5 * @max_balance) %>
  </text>
  <text id="balance25Percent" x="520" y="165">
    <%= number_to_currency(0.25 * @max_balance) %>
  </text>
</g>
```

The labels are placed to the right of the chart and are spaced at 25 percent intervals. The text for the label is constructed using the `number_to_currency` helper. The value for each label is determined by multiplying the `@max_balance` by the appropriate factor.

In addition to the balance labels, you should add labels for the payment dates along the bottom of the chart. Instead of adding a label for every payment, you can add one label per year:

```
<g id="date_labels" font-size="6pt" font-weight="normal"
  transform="translate(20, 220)">
  <% (0..@payments.length).step(12) do |i| %>
    <text text-anchor="end"
      transform="translate(<%= (i*payment_width)-%>,0) rotate(-45)">
      <%= @payments[i].PmtDate -%>
    </text>
  <% end %>
</g>
```

The loop for the payments uses another special Ruby construction. An inclusive range from 0 to the number of payments is created (using the `..` operator). The `step` function is used to step through every 12th item in the range. For each iteration, a new `<text>` element is added with the payment date as the value. Instead of scaling the entire container, the offset of each payment is set using the `scale`. Each payment is also rotated at a 45 degree angle so that they don't overlap.

Include the `end` command for the block:

```
<% end %>
```

All of the information for the chart is completed, so the final step is to finish the SVG:

```
</g>
</svg>
```

Save the changes to the chart. You should be able to see Ajax and chart in action. Make sure that the built-in web server is running, fill in the form, and click Calculate. Open your browser to `http://localhost:3000/calculator` (or refresh if necessary), you should see the rendered chart with data, as in Figure 22-10.

How It Works

In this Try It Out, you added the data for the SVG chart. You also modified the `calculate` action to use Remote JavaScript. Using the RJS you could simultaneously update the `payments` table and chart SVG. Having a visual representation of the loan payoff makes comparing different programs very easy. In the fixed loan program, the payment amount doesn't change. However, at the start of the loan much of the payment is applied to the interest on the loan rather than the balance. Because of this the chart shows a curve for the remaining balance.

The ComplianceStudio calculation can handle any number of programs and loan details. For example, if you change the term to 180 months and recalculate, you will notice that the payments do not affect the balance much at all. At the end of the loan one final payment (called a balloon payment) is made to pay off the loan.

To create the chart you needed to use a number of Ruby tricks and SVG tricks. The power of XML is realized when combining many different technologies. In this case you have created a very useful display by combining XHTML, SOAP, and SVG.

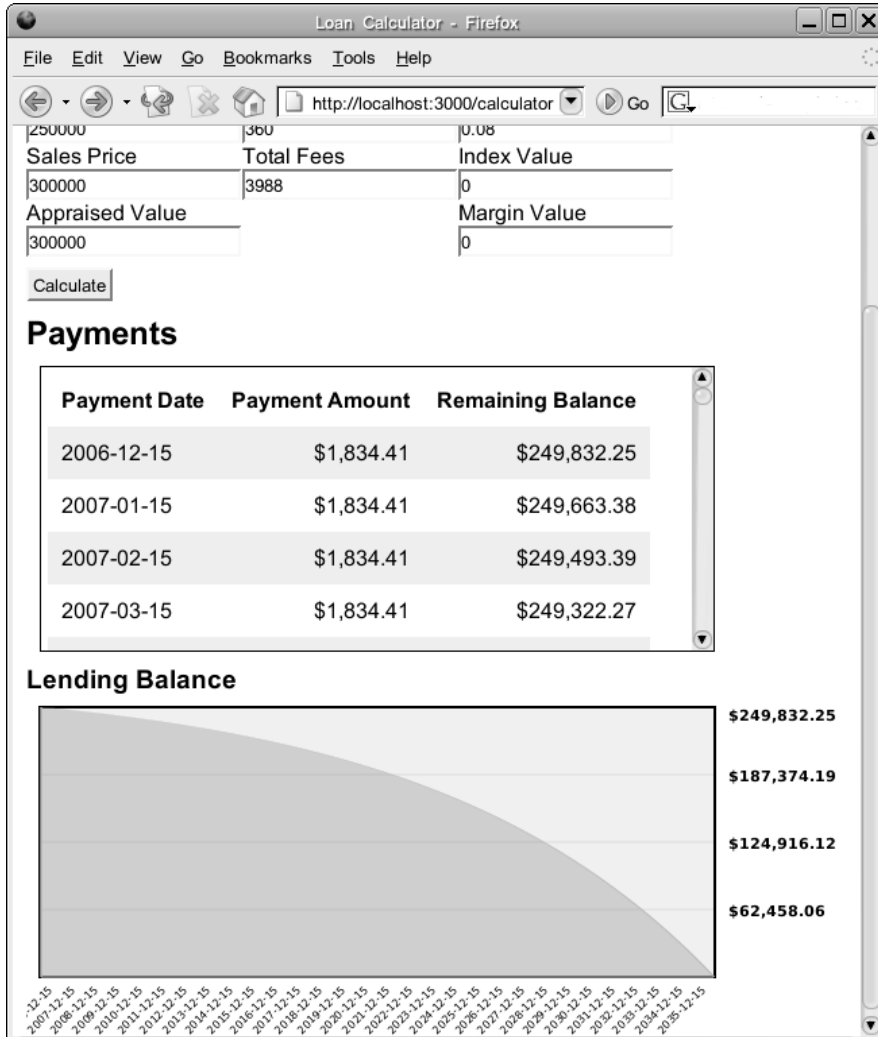


Figure 22-10

Summary

In this case study, you used a variety of XML technologies to build an advanced online loan calculator. By connecting to a freely available web service using a local proxy service, you were able to quickly execute advanced mortgage calculations and display the results to users. Though you may not be working in the mortgage industry, this pattern of connecting to a web service and displaying the results in your own page is used throughout the web.

Of course, there is no reason to limit the loan calculator to a single web service. You could easily connect to other web services and combine the results on your page. Often online applications combine calculation engines, search engines, and mapping engines and even other mashups to build successful sites.