# 8 XML and multiple representations

## It all looks different now...

**You can't please everyone all of the time. Or can you?**

So far we've looked at how you can use Rails to quickly and easily develop web apps that **perfectly fit one set of requirements**. But what do you do when **other requirements come along**? What should you do if some people want **basic web pages**, others want a **Google mashup**, and yet more want your app available as an **RSS feed**? In this chapter you'll create **multiple representations** of the same basic data, giving you the **maximum flexibility** with **minimum effort**.

## Climbing all over the world

Head First Climbers is a web site for mountaineers all over the world. Climbers report back from expeditions to record the locations and times of mountains they have climbed, and also to report dangerous features they've discovered, like rock slides and avalanches.

The information is obviously very important for the safety of other climbers, and many climbers use mobile phones and GPS receivers to read and record information straight from the rock face. Used in the right way, the system will save lives and yet—somehow—the web site's not getting a lot of traffic.

### So why isn't it popular?

The application is very basic. It's simply a scaffolded version of this data structure:

| Incident | |
|---|---|
| **mountain** | *string* |
| **latitude** | *decimal* |
| **longitude** | *decimal* |
| **when** | *datetime* |
| **title** | *string* |
| **description** | *text* |

| id | mountain | latitude | longitude | when | title | description |
|---|---|---|---|---|---|---|
| 1 | Mount Rushless | 63.04348055... | -150.993963... | 2009-11-21 11:... | Rock slide | Rubble on the ... |
| 2 | Mount Rushless | 63.07805277... | -150.977869... | 2009-11-21 17:... | Hidden crev... | Ice layer cove... |
| 3 | Mount Lotopaxo | -0.683975 | -78.4365055... | 2009-06-07 12:... | Ascent | Living only on... |
| 4 | High Kanuklima | 11.123925 | 72.72135833... | 2009-05-12 18:... | Altitude si... | Overcome by th... |

As you've noticed by now, scaffolding is a great way to *start* an application, but you'll almost always need to modify the code to change the generic scaffolding code into something that's more appropriate for the problems your users are trying to solve.

**So what needs to change about this application?**

Do this!

Create a scaffolded application that matches this data structure.

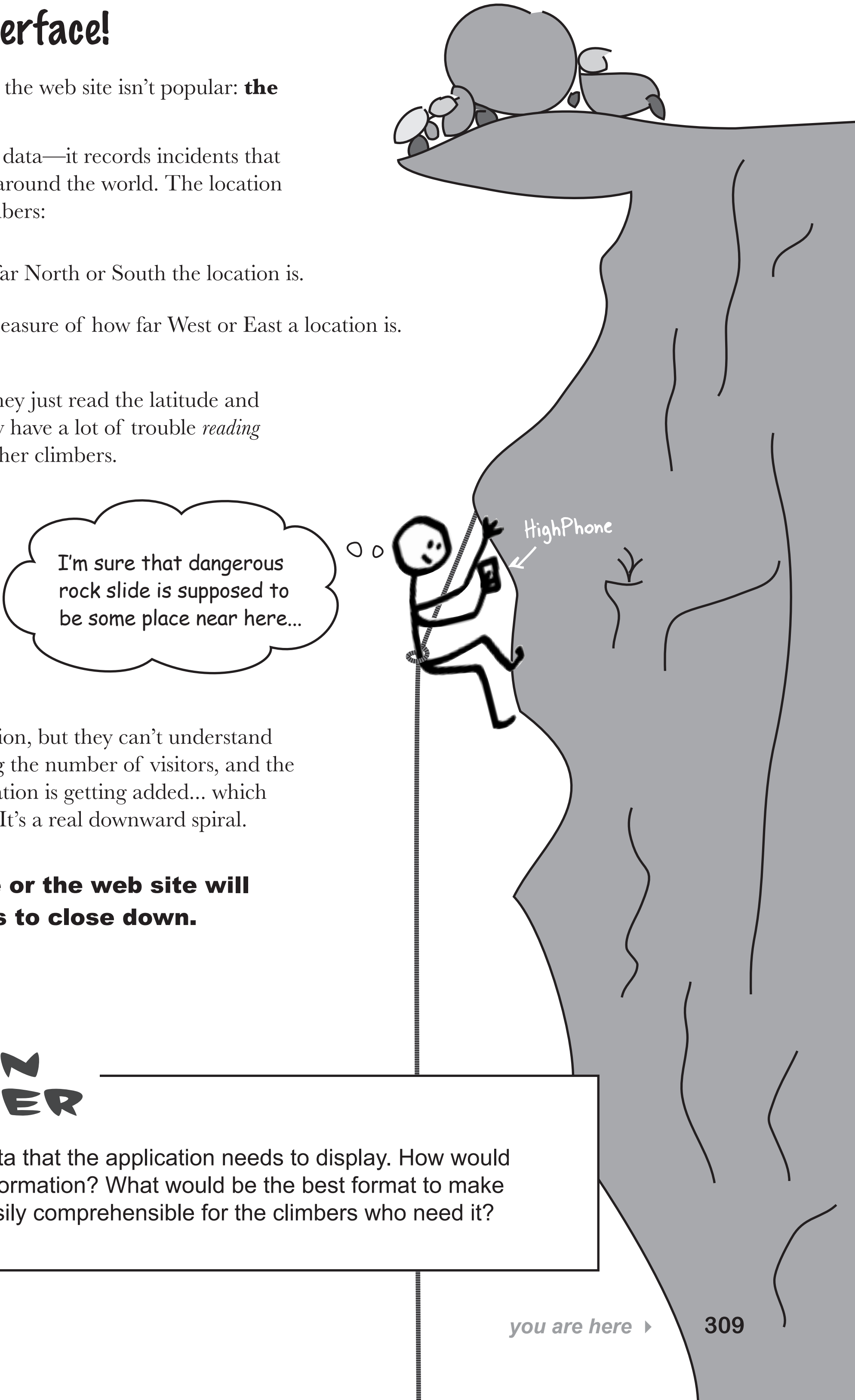
## The users hate the interface!

It doesn't take too long to find out why the web site isn't popular: **the user interface**.

The system is used to manage ***spatial*** data—it records incidents that happen at particular places and times around the world. The location information is recorded using two numbers:

- The ***latitude***. This is how far North or South the location is.
- The ***longitude***. This is a measure of how far West or East a location is.

The users can record their data OK: they just read the latitude and longitude from GPS receivers. But they have a lot of trouble *reading* and *interpreting* the information from other climbers.


I'm sure that dangerous rock slide is supposed to be some place near here...
HighPhone

So people can add data to the application, but they can't understand the data they get from it. That's cutting the number of visitors, and the fewer visitors there are the less information is getting added... which causes even less people to use the app. It's a real downward spiral.

**Something needs to be done or the web site will lose so much business it has to close down.**

**BRAIN POWER**

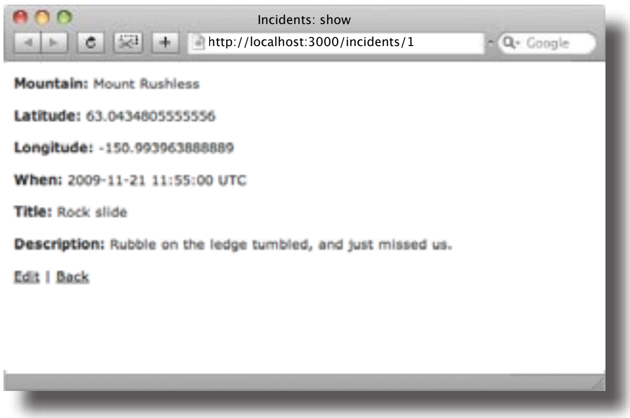
Think about the data that the application needs to display. How would **you** display the information? What would be the best format to make the information easily comprehensible for the climbers who need it?

## The data needs to be on a map

The system records geographic data and it should be displayed on a map.

The correct data is being stored, and the basic functions (create, read, update, and delete) are all available. The problem is **presentation**. The location is *stored* as two numbers—the latitude and longitude—but that doesn't mean it has to be *displayed* that way.

Instead of seeing this...


Incidents: show

http://localhost:3000/incidents/1

**Mountain:** Mount Rushless

**Latitude:** 63.0434805555556

**Longitude:** -150.993963888889

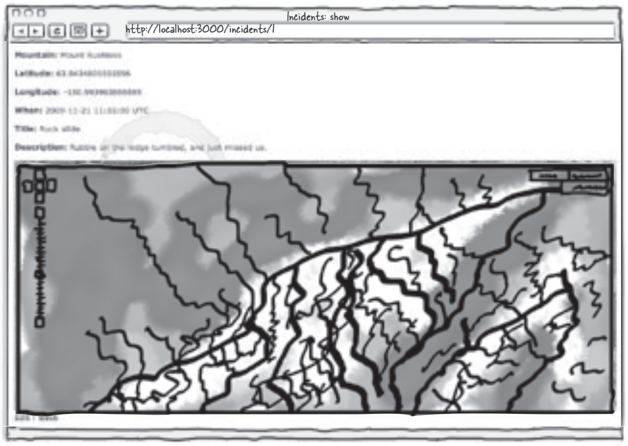
**When:** 2009-11-21 11:55:00 UTC

**Title:** Rock slide

**Description:** Rubble on the ledge tumbled, and just missed us.

Edit | Back

...climbers need to see something like this:



Now this is obviously going to be a pretty big change to the interface, so the web site guys have decided that rather than change the whole application, they are going to run a small pilot project to create a version of the page that displays an incident and get it to display a map. But they have no idea what to do, and need your help.

**What's the first thing YOU would do?**

## We need to create a new action

We don't want to *change* the existing code—we only want to *add* to it. Until we are sure that the new interface works, we don't want to upset any of the existing users. After all, there aren't that many left...

So we'll add a new action called `show_with_map`. At the moment, someone can see one of the incidents using a URL like this:

```
http://localhost:3000/incidents/1
```

We'll create a new version of the page at:

```
http://localhost:3000/incidents/map/1
```

This way, the pilot users only need to add `/map` to get the new version of the page. We'll use this for the route:

Remember to add this as the first route in your config/routes.rb file.

```
map.connect 'incidents/map/:id', :action=>'show_with_map', :controller=>'incidents'
```

**Sharpen your pencil**

We can create the page template by copying the `app/views/incidents/show.html.erb` file. What will the new file be called?

..............................................................................................................................................

The incidents controller will need a new method to read the appropriate `Incident` model object and store it in an instance variable called `@incident`. Write the new method below:

..............................................................................................................................................

..............................................................................................................................................

..............................................................................................................................................

..............................................................................................................................................

..............................................................................................................................................

**Sharpen your pencil Solution**

We can create the page template by copying the `app/views/incidents/show.html.erb` file. What will the new file be called?

app/views/incidents/show_with_map.html.erb

The incidents controller will need a new method to read the appropriate `Incident` model object and store it in an instance variable called `@incident`. Write the new method below:

```
def show_with_map
  @incident = Incident.find(params[:id])
end
```

show_with_map is the name of the action.

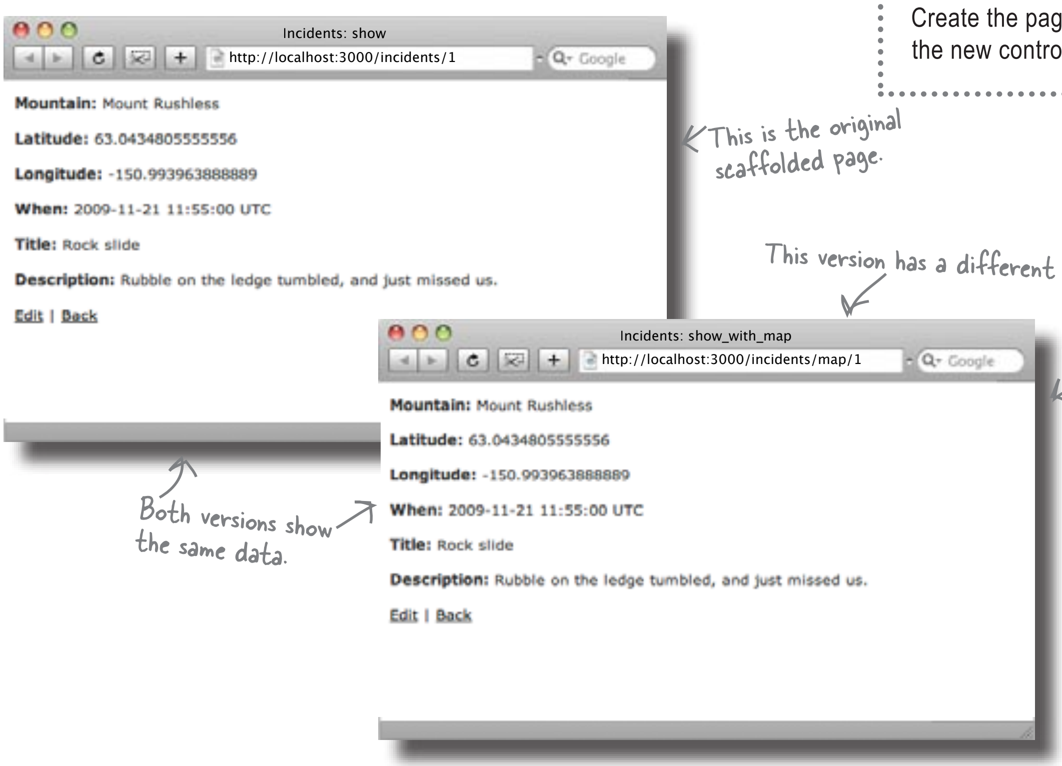
This will be the id number from the URL.

## The new action seems to work...

Do this!

Create the page template and the new controller method now.

If you now look at the two versions of the incidents page, we see that they both display the correct data. What do you notice?


Incidents: show
http://localhost:3000/incidents/1

Mountain: Mount Rushless

Latitude: 63.0434805555556

Longitude: -150.993963888889

When: 2009-11-21 11:55:00 UTC

Title: Rock slide

Description: Rubble on the ledge tumbled, and just missed us.

Edit | Back

Incidents: show_with_map
http://localhost:3000/incidents/map/1

Mountain: Mount Rushless

Latitude: 63.0434805555556

Longitude: -150.993963888889

When: 2009-11-21 11:55:00 UTC

Title: Rock slide

Description: Rubble on the ledge tumbled, and just missed us.

Edit | Back

This is the original scaffolded page.

This version has a different URL.

This is the version that calls the new show_with_map action.

Both versions show the same data.

Both versions of the incidents page look identical—and that's a problem.

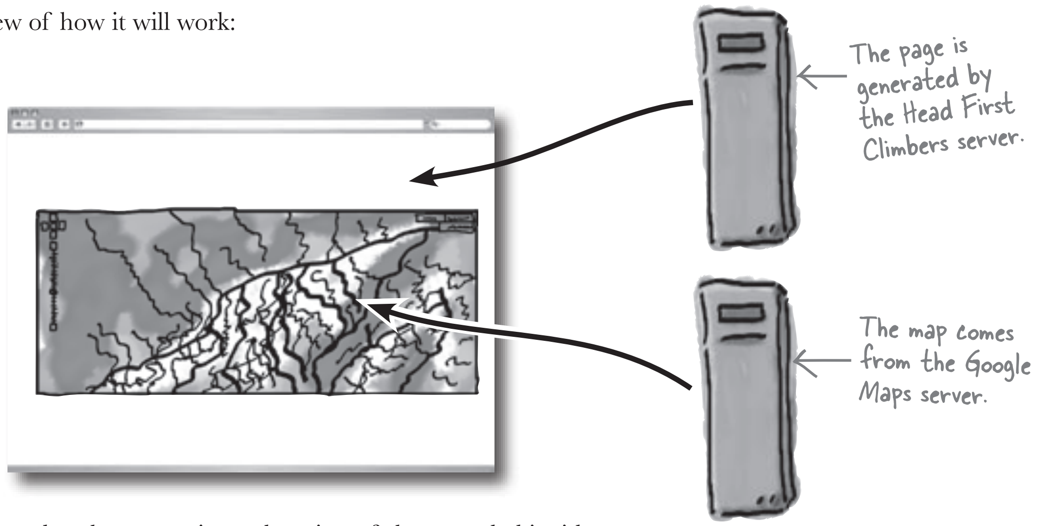
## The new page needs a map... that's the point!

But of course we *don't* want the new version of the page to look the same. We want to add a map.

So how will we do that? There's no way we're going to build our own mapping system. Instead we'll create a **mashup**. A mashup is a web application that integrates data and services from other places on the web.

Most of the mapping services allow you to embed maps inside your own web application, but we'll use the one provided by Google. Google Maps give you a lot of flexibility. Not only can you embed a map in a web page, but you can also, without too much work, add your own data onto the map and program how the user interacts with the map and data.

Here's a high-level view of how it will work:


The page is generated by the Head First Climbers server.
The map comes from the Google Maps server.

The map will be displayed at the approximate location of the recorded incident, and a symbol mark the exact point.

The Head First Climbers application will generate the code to call the map, and the data to display on it, but the map itself, and the bulk of the code that allows the user to do things like drag the map or zoom in and out, will come from the Google Maps server. Even though Google will provide the bulk of the code, we still need to provide two things:

- The HTML and JavaScript to call the map. This will be a little complex, so we will put the HTML and JavaScript we need in a separate partial that we can call from our page template.

- The data we need to display on the map. To begin with we will use an example data file to make sure the map's working.

So what will the map code look like?

## So what code do we need?

We need to have the following code in a partial called `_map.html.erb`:

Download It!

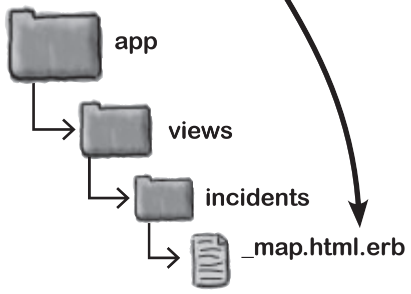
```
<%
  google_key='ABQIAAAAnfs7bKE82qgb3Zc2YyS-oBT2yXp_' +
    'ZAY8_ufC3CFXhHIE1NvwkxSySz_REpPq-4WZA27OwgbtyR3VcA'
  full_page ||= false
  show_action ||= nil
  new_action ||= nil
  data ||= nil
%>
<div id="map"
  align="right"
  style="border: 1px solid #979797;
                min-width: 400px;
<%              if full_page -%>
                min-height: 800px;
                height: 800px;
<%              else -%>
                min-height: 400px;
                height: 400px;
<%              end -%>
                background-color: #FFFFFF;
                border: 1px solid #999999;
                padding: 10px;"></div>
...
```

This key makes the map work for 'localhost'

There's not enough space to display all of the partial code here, but you can download the file at http://tinyurl.com/hfrailsmap

So what does this code do? First of all it calls some JavaScript on the Google Maps server that will generate a map on the web page. The map will have all of the basic drag and zoom functions built in.

But the basic Google code doesn't do *everything* we need. It doesn't load and display any of our local data. So the code in the `_map.html.erb` partial also loads location data from a file, which it uses to move the map to the correct place and display an icon at a given point.



**But there's a little complication with the code...**

## The code will only work for localhost

Google places a restriction on the use of the code. They insist that you say which host you're going to use it on. That means before you can use it on www.yourowndomain.com, you need to tell Google about it. In order to make sure that people comply with this condition, the code will only run if you provide it with a **Google Maps key**. The key is generated for a particular host name, and if you try to embed a Google map into a page coming from anywhere else, the map will refuse to run.

But for now, there's not a problem. The **`_map.html.erb`** partial we're going to use has the Google Maps key for localhost—so as long as you run the code on your own machine it will be fine. But remember, you'll need to apply for your own key before running the code anywhere else.

### Geek Bits

If you want to embed Google Maps in your own web apps, you need to sign up with Google. To do this, visit the following URL: http://tinyurl.com/mapreg

### Sharpen your pencil

You need to include the map partial in the `show_with_map.html.erb` template. We need to pass a local variable called `data` containing the path to the map data. We'll use a test file for this at `/test.xml`.

Write the code to call the partial.

..........

..........

..........

**Sharpen your pencil Solution**

You need to include the map partial in the `show_with_map.html.erb` template. We need to pass a local variable called `data` containing the path to the map data. We'll use a test file for this at `/test.xml`.

Write the code to call the partial.

You need to insert this line of code in the show_with_map.html.erb file

```
<%= render (:partial=>'map', :locals=>{:data=>'/test.xml'}) %>
```

## Now we need the map data

Before we can try out the embedded map, we need to provide it with map data. To begin with we will just use the test.xml test file. This is what it looks like:

public
test.xml

```
<data>
  <description>This is an example description</description>
  <latitude>63.0434805555556 </latitude>
  <longitude>-150.993963888889</longitude>
  <title>Test Data</title>
</data>
```

**Download It!**

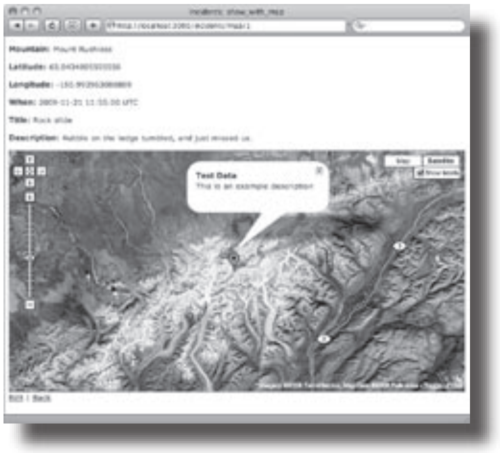
To save you typing in the long numbers, you can download the test.xml file from http://tinyurl.com/maptest

The mapping data provides the latitude and longitude of the test incident. When the Google map loads, our map partial will pass it the contents of this file and the incident should be displayed and centered.

## Test Drive

So what happens if we go to a URL like:

`http://localhost:3000/incidents/map/1`



**Mountain:** Mount Rushless

**Latitude:** 63.0434801555556

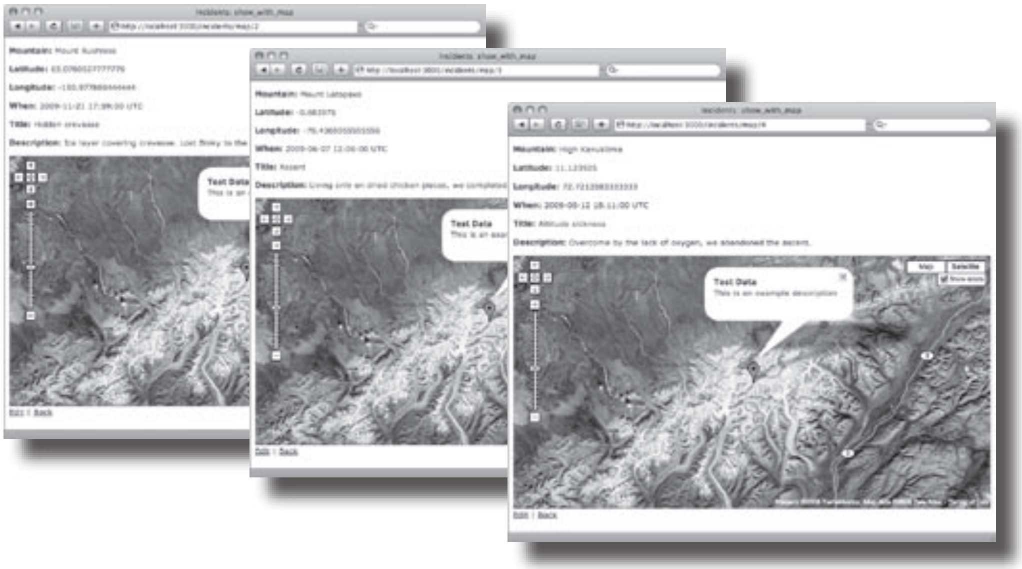
**Longitude:** -151.9929636666669

**When:** 2009-11-21 11:55:00 UTC

**Title:** Rock slide

**Description:** Rubble on the ledge tumbled, and just missed us.

The map works! But what if we go to a different URL?



**Mountain:** Mount Rushless

**Latitude:** 63.0760127777776

**Longitude:** -151.977866444444

**When:** 2009-11-21 17:39:00 UTC

**Title:** Hidden crevasse

**Description:** Ice layer covering crevasse. Lost Binky to the

**Mountain:** Mount Latopwo

**Latitude:** -0.683375

**Longitude:** -78.4366555551556

**When:** 2009-06-07 12:06:00 UTC

**Title:** Ascent

**Description:** Living only on dried chicken pieces, we completed

**Mountain:** High Kanuslima

**Latitude:** 11.123025

**Longitude:** 72.7212583333333

**When:** 2009-05-12 18:11:00 UTC

**Title:** Altitude sickness

**Description:** Overcome by the lack of oxygen, we abandoned the ascent.

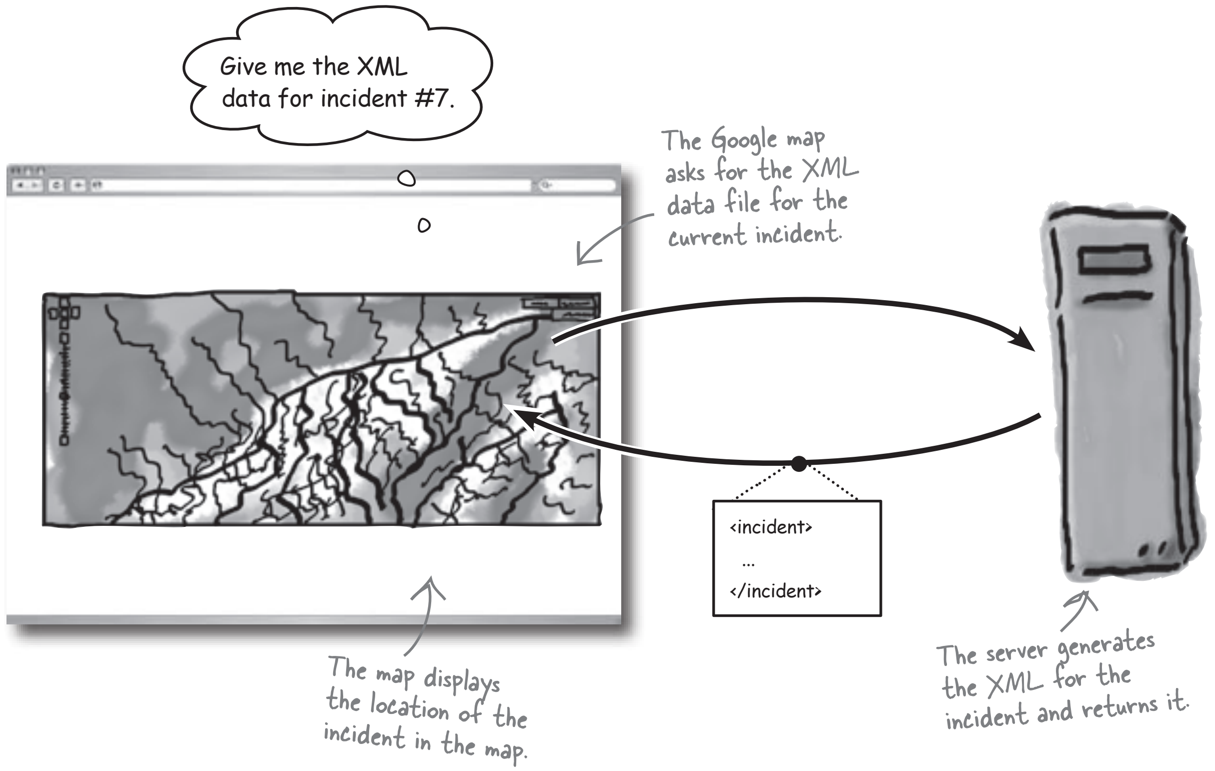
Every map looks exactly the same, regardless of the data. That's because each map is using the same data: the contents of the test.xml file.

**In order to make the map display the location of a given incident, we need to generate a data file for each page.**

## What do we need to generate?

We're passing XML data to the map, and the XML data describes the location of a single incident. The location is given by the latitude, the longitude, the title, and the description. We need to generate XML like this for *each* incident.

So the system will work something like this:

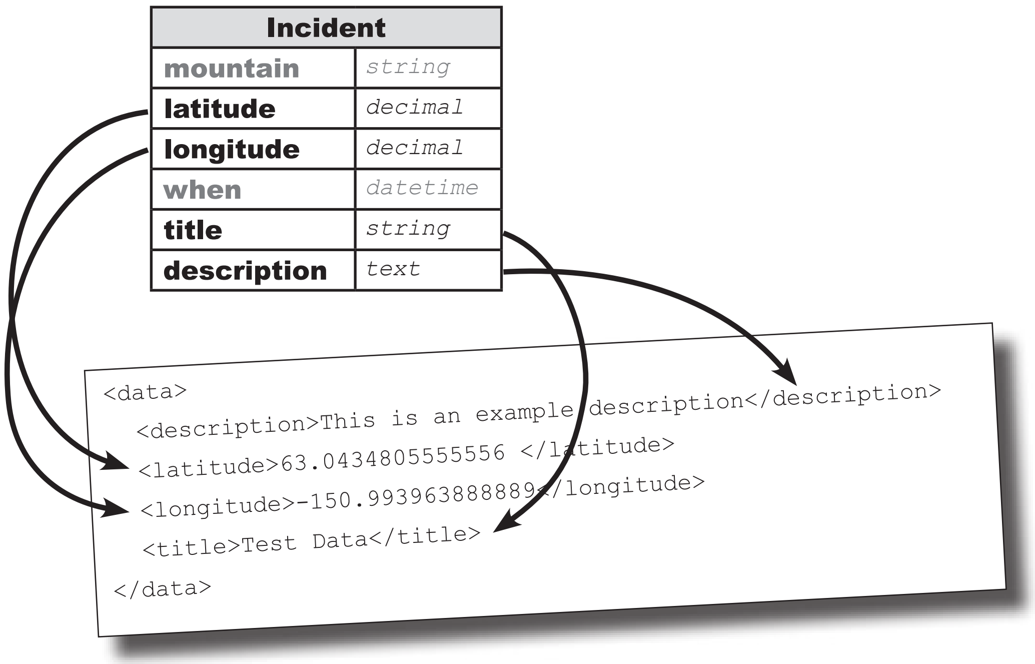

Give me the XML data for incident #7.
The Google map asks for the XML data file for the current incident.
<incident>
...
</incident>
The map displays the location of the incident in the map.
The server generates the XML for the incident and returns it.

If this is starting to feel familiar, good! The Google Map is actually using Ajax to work. Remember how we used Ajax to download new version of the seat list in the previous chapter? In the same way, the Google Map will request XML data for the location of an incident.

**So the next thing is to generate the data. Where will we get the data from?**

## We'll generate XML from the model

The data for the generated XML will come from the Incident model. We'll be using just four of the attributes, the latitude, longitude, title, and description.

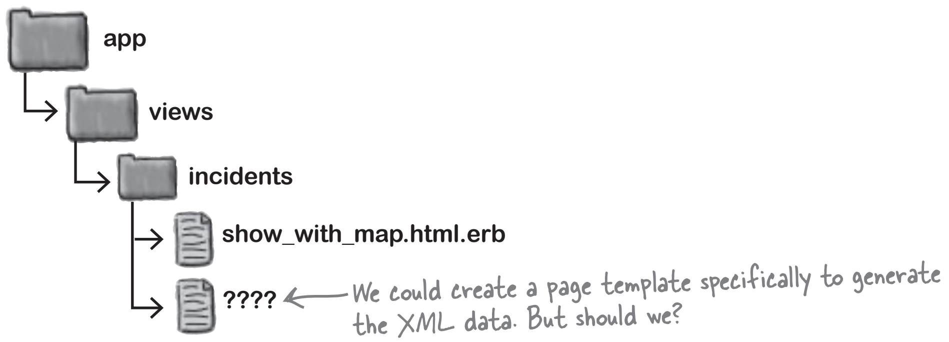


| Incident | |
|---|---|
| mountain | string |
| latitude | decimal |
| longitude | decimal |
| when | datetime |
| title | string |
| description | text |

```
<data>
  <description>This is an example description</description>
  <latitude>63.0434805555556 </latitude>
  <longitude>-150.993963888889</longitude>
  <title>Test Data</title>
</data>
```

But how do we generate the XML? In a way, this is a little like generating a web page. After all, XML and HTML are very similar. And just as web pages contain data from the model, our XML files will also contain data from the model.

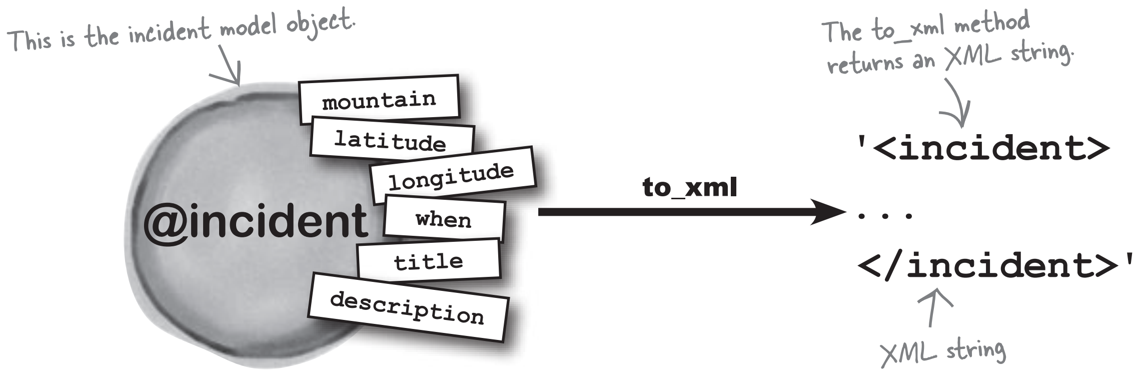
So one option would be to create a page template containing XML tags instead of HTML tags:

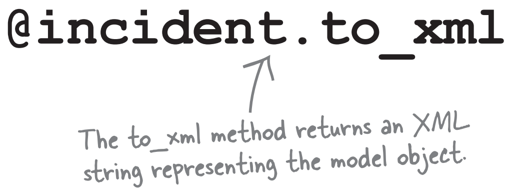

app
views
incidents
show_with_map.html.erb
????
We could create a page template specifically to generate the XML data. But should we?

**That way *would* work, but there's a better way...**

## A model object can generate XML

Model objects contain data. XML files contain data. So it kind of makes sense that model objects can generate XML versions of themselves. Each model object has a method `to_xml` that returns an XML string:




@incident.to_xml
The to_xml method returns an XML string representing the model object.

But creating the XML is only half the story. The other half is returning that XML to the browser. We're not using a page template, so the whole job will have to be handled by the controller rendering the XML...

## What will the controller code look like

We can amend the `show_with_map` method to output the XML:

```
def show_with_map
  @incident = Incident.find(params[:id])
  render :text=>@incident.to_xml
end
```

This is the incident object we were already reading.

The render method returns the XML.

This will create an XML string that describes the incident object.

The text parameter says what we'll be returning to the browser.

The `render` method returns the XML to the browser. We've seen the `render` method before, but this is a slightly different version. Most of the time you use `render` to generate a web page from a template or partial. But you can also just pass it a string object—and that's what we're doing here.

### Geek Bits

To make your life simpler, the Rails folks also allow you to pass a parameter to the `render` method called :xml

```
render :xml=>@incident
```

If the `render` method is passed an object using the :xml parameter, it will call the `to_xml` method on the object and send that back to the browser. The :xml version of the `render` command will generate the same content as the `render` command in our controller, but it will also set the mime-type of the response to text/xml. But for now, we will use the :text version above.

### there are no Dumb Questions

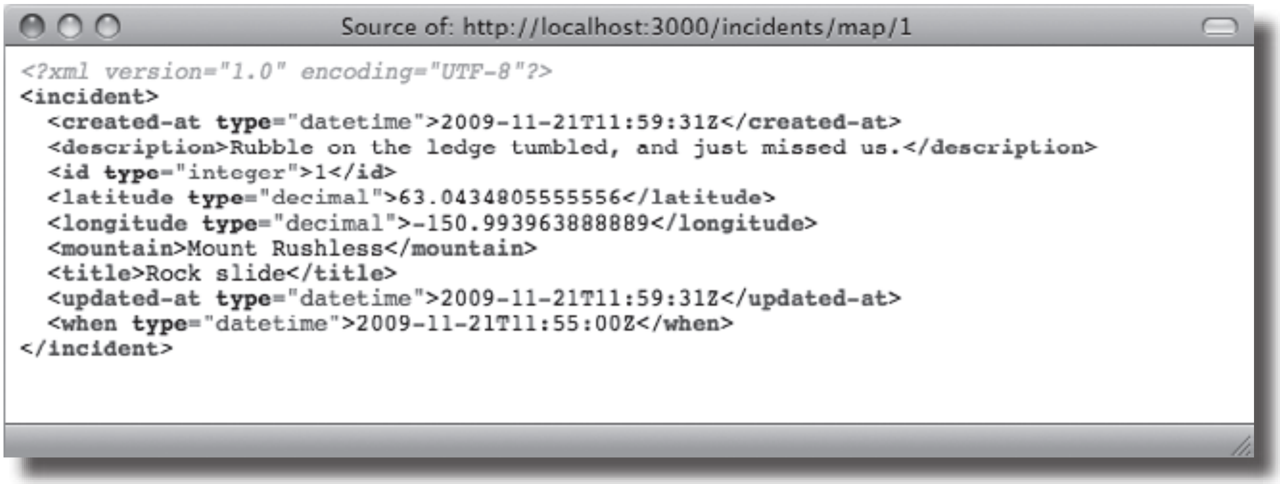
**Q: Remind me, what does the render method do again?**

A: `render` generates a response for the browser. When your browser asks for a page, that's a request. `render` generates what gets sent back.

## Test Drive

So what do we get now if we go to:

`http://localhost:3000/incidents/map/1`



Source of: http://localhost:3000/incidents/map/1

```
<?xml version="1.0" encoding="UTF-8"?>
<incident>
  <created-at type="datetime">2009-11-21T11:59:31Z</created-at>
  <description>Rubble on the ledge tumbled, and just missed us.</description>
  <id type="integer">1</id>
  <latitude type="decimal">63.0434805555556</latitude>
  <longitude type="decimal">-150.993963888889</longitude>
  <mountain>Mount Rushless</mountain>
  <title>Rock slide</title>
  <updated-at type="datetime">2009-11-21T11:59:31Z</updated-at>
  <when type="datetime">2009-11-21T11:55:00Z</when>
</incident>
```

The controller is now returning XML containing the data from the incident object with `id = 1`.

But is there a problem? The XML we're generating looks *sort* of the same as the example XML, but there are a few differences:

- We're generating too many attributes. The example data file only contained information about the latitude, longitude, title, and description. But this piece of XML contains **everything** about an incident, even the date and time that the incident record was created.

- The root of the XML file has the wrong name. The generated XML takes its root name from the variable we were using, `<incident>`. But we need the XML to have a root named `<data>`.

```
<data>
  <description>This is an example
    description</description>
  <latitude>63.0434805555556 </latitude>
  <longitude>-150.993963888889</longitude>
  <title>Test Data</title>
</data>
```

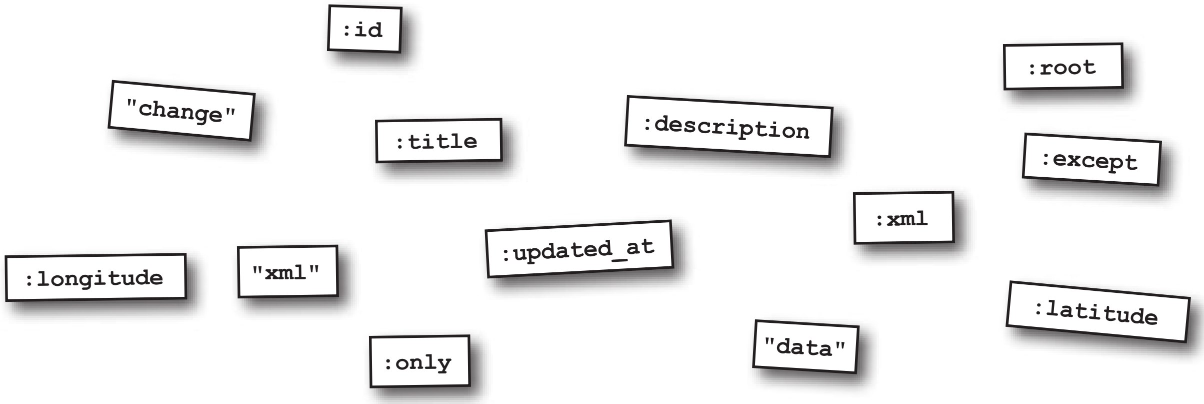
The XML is *almost* in the right format, but *not quite*.

**We need to modify the XML that `to_xml` produces.**

## Code Magnets

The `to_xml` method has some optional parameters that let us modify the XML that it returns. See if you can work out what the values of the parameters should be:

```
def show_with_map
  @incident = Incident.find(params[:id])
  render :text=>@incident.to_xml(
  ..................=>[........................,..........................,....................,...............................],
  ..................=> ...................)
end
```



## Code Magnets Solution

The `to_xml` method has some optional parameters that let us modify the XML that it returns. See if you can work out what the values of the parameters should be:

```
def show_with_map
  @incident = Incident.find(params[:id])
  render :text=>@incident.to_xml(
    :only =>[ :latitude , :longitude , :title , :description ],
    :root => "data" )
end
```

Because we're using the render :text=>... version of the render command we can use the options in to_xml and modify the output.

Unused magnets: `:id`, `"change"`, `:except`, `:xml`, `:updated_at`, `"xml"`

### there are no Dumb Questions

**Q: Shouldn't we generate the XML in the model?**

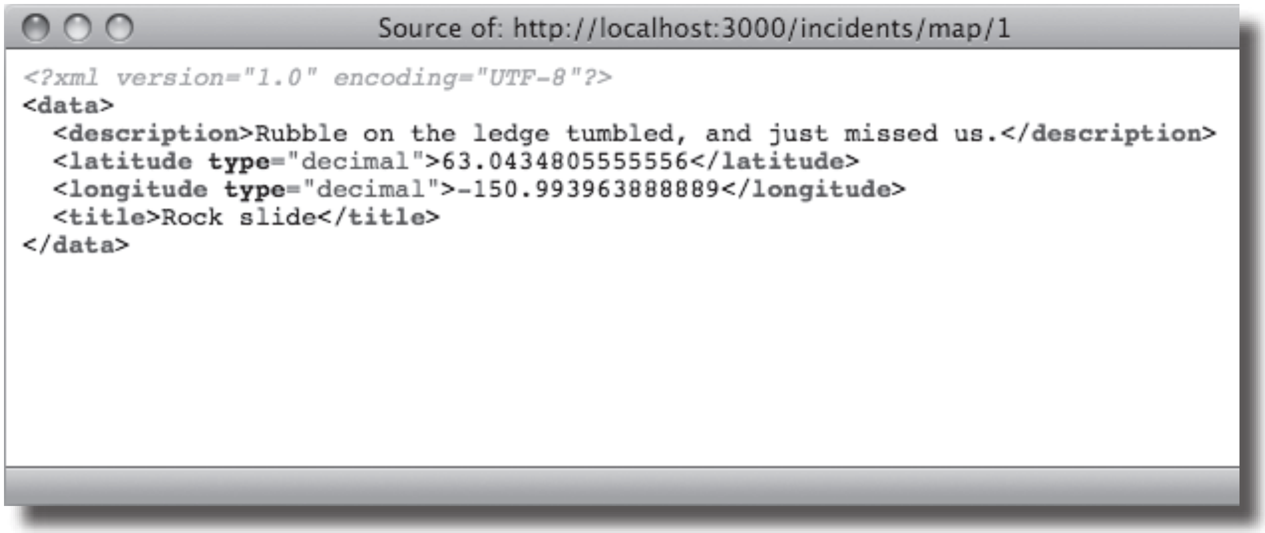
A: You could, but it's not a good idea. You may need to generate different XML in different situations. If you added code to the model for each of those XML formats, the model would quickly become overloaded.

## Test Drive

Now when we go to:

`http://localhost:3000/incidents/map/1`

we get XML that looks a little different.



```
<?xml version="1.0" encoding="UTF-8"?>
<data>
  <description>Rubble on the ledge tumbled, and just missed us.</description>
  <latitude type="decimal">63.0434805555556</latitude>
  <longitude type="decimal">-150.993963888889</longitude>
  <title>Rock slide</title>
</data>
```

You've managed to modify the XML so that it only displays the data we need and has a properly named root element. It looks a lot closer to the example XML file.

The `to_xml` method doesn't allow you to make a lot of changes to the XML it produces, but it's good enough for most purposes... including sending the XML to Google for some custom mapping.

**With very little work, `to_xml` gave us exactly the XML we wanted.**

## Meanwhile, at 20,000 feet...

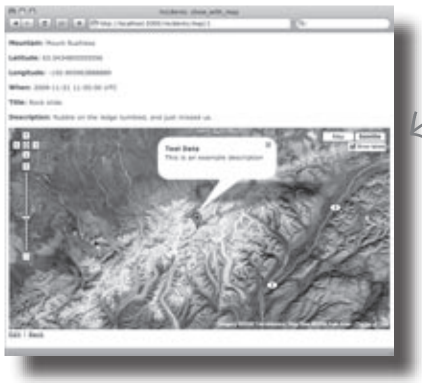
**Some people on the pilot program have a problem.**

The web pages have disappeared! Before the last amendment a URL like:

```
http://localhost:3000/incidents/map/1
```

generated a web page. The trouble is, now that URL just returns XML, instead of a nice Google map.

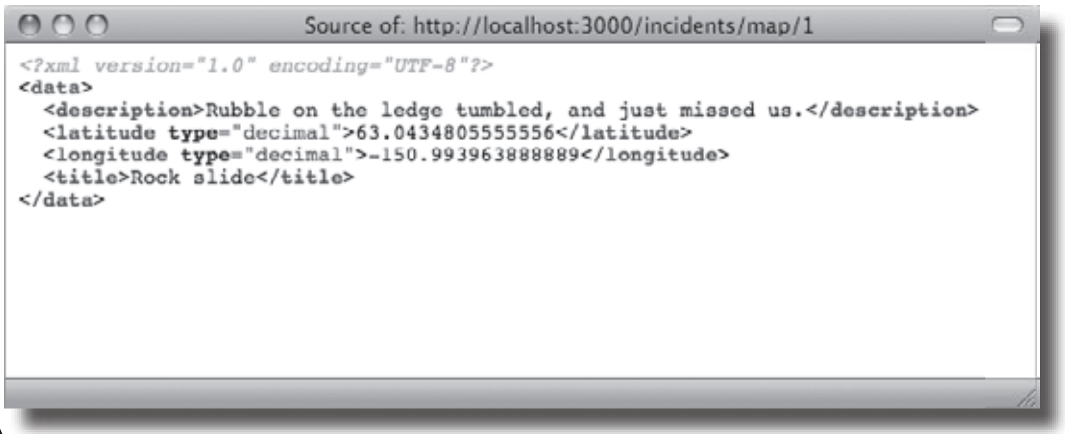
### Before your latest changes:



Before the amendment, we had a web page showing our data on a Google map.

After the amendment, all we got back was this XML.

### After your latest changes:

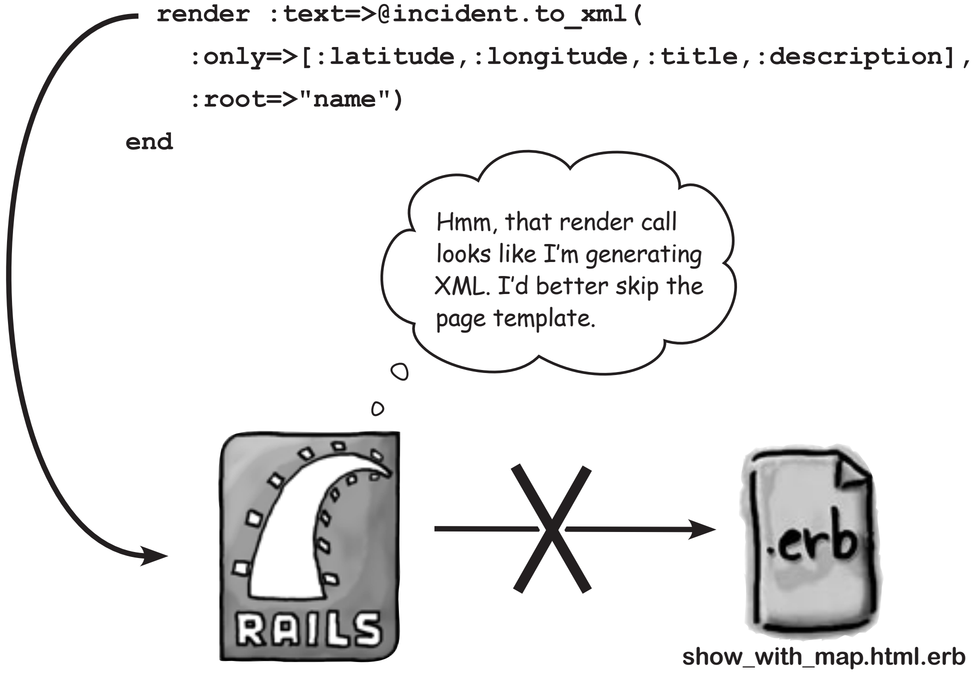


```
<?xml version="1.0" encoding="UTF-8"?>
<data>
  <description>Rubble on the ledge tumbled, and just missed us.</description>
  <latitude type="decimal">63.0434805555556</latitude>
  <longitude type="decimal">-150.993963888889</longitude>
  <title>Rock slide</title>
</data>
```

## We need to generate XML and HTML

The `show_with_map` action originally generated a web page with the `show_with_map.html.erb` page template. But once we added a `render` call to the controller method, Rails ignored the template and just generated the XML:

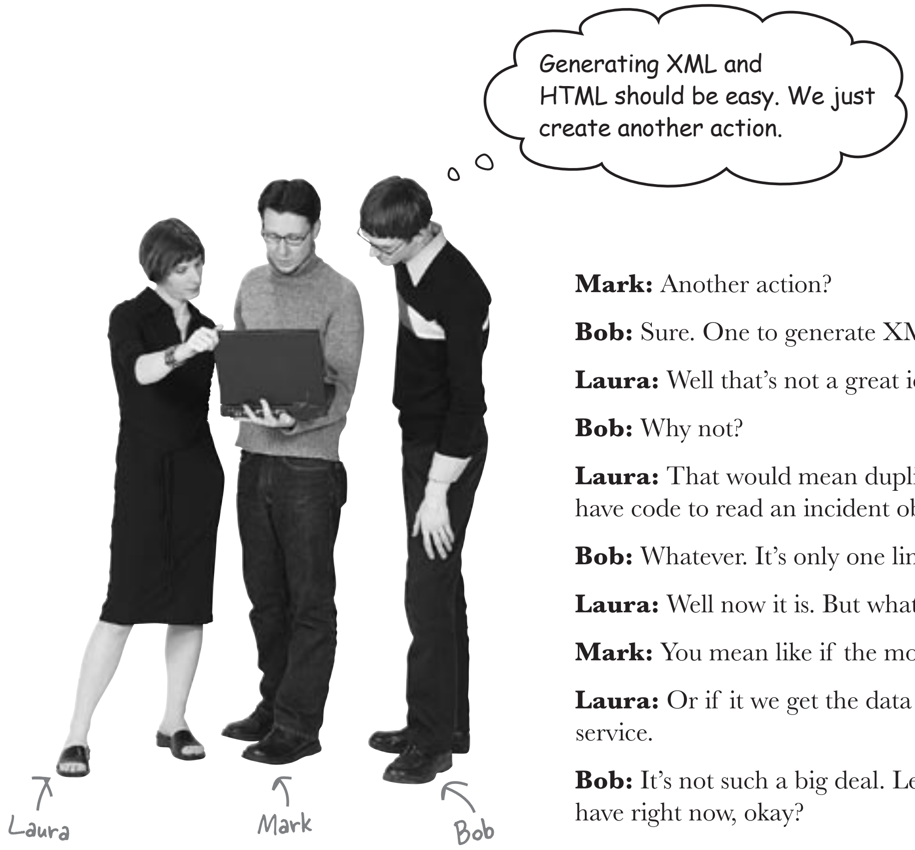
```
def show_with_map
  @incident = Incident.find(params[:id])
  render :text=>@incident.to_xml(
    :only=>[:latitude,:longitude,:title,:description],
    :root=>"name")
end
```


Hmm, that render call looks like I'm generating XML. I'd better skip the page template.
RAILS
.erb
show_with_map.html.erb

Of course, that makes sense, because there's no way an action can generate XML and HTML ***at the same time***.

But we still need a web page to display the map, and the map still needs XML map data. So what do we do?

**We need some way of calling the controller in one way to generate HTML, and calling the controller in another way to generate XML.**


Generating XML and HTML should be easy. We just create another action.
Laura
Mark
Bob

**Mark:** Another action?

**Bob:** Sure. One to generate XML and another to generate HTML.

**Laura:** Well that's not a great idea.

**Bob:** Why not?

**Laura:** That would mean duplicating code. Both methods would have code to read an incident object.

**Bob:** Whatever. It's only one line.

**Laura:** Well now it is. But what if we change things in the future?

**Mark:** You mean like if the model changes?

**Laura:** Or if it we get the data from somewhere else, like a web service.

**Bob:** It's not such a big deal. Let's worry about the problems we have right now, okay?

**Mark:** I don't know. Laura, what would you do?

**Laura:** Simple. I'd pass a parameter to the action. Tell it what format we want.

**Mark:** That might work.

**Bob:** Come on, too much work.

**Laura:** Less work than creating another action.

**Mark:** But one thing...

**Laura:** Yes?

**Mark:** Doesn't the URL identify the information we want?

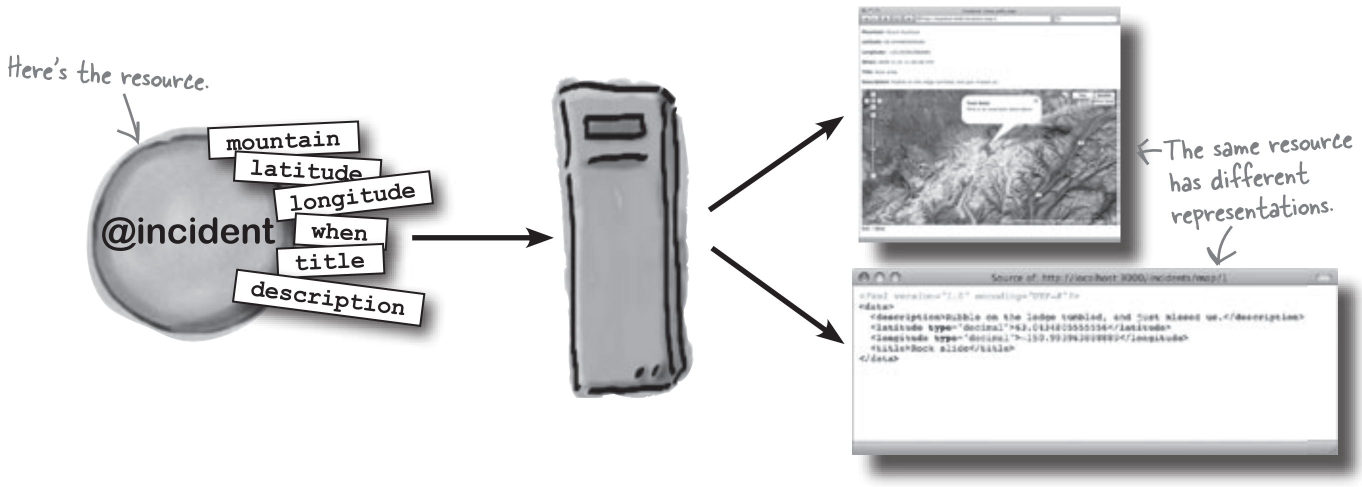
**Laura:** So?

**Mark:** Shouldn't we use the same URL for both formats?

## XML and HTML are just representations

Although the HTML and XML look very different, they are really visual representations of the *same thing*. Both the HTML web page and the XML map data are both describing the same Incident object data. That incident is the core data, and it's sometimes called the **resource**.

A **resource** is the data being presented by the web page. And the web page is called a **representation** of the resource. Take an Incident object as an example. The Incident object is the resource. The incident web page and the map data XML file are both representations of the resource.


Here's the resource.
@incident
mountain
latitude
longitude
when
title
description
The same resource has different representations.
Source of: http://localhost:3000/incidents/map/1

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
  <description>Rubble on the ledge tumbled, and just missed us.</description>
  <latitude type="decimal">[illegible]</latitude>
  <longitude type="decimal">[illegible]</longitude>
  <title>Rock slide</title>
</data>
```

Thinking about the web as a set of resources and representations is part of a design architecture called **REST**. REST is the **architecture of Rails**. And the more RESTful your application is, the better it will run on Rails.

But how does this help us? Well, to be strictly RESTful, both the XML data and the web page should have the same URL (Uniform *Resource* Locator) because they both represent the same resource. Something like this:

**`http://localhost:3000/incidents/maps/1`**

But to simplify things, we can compromise the REST design (a little bit) and use these URLs for the two representations:

**`http://localhost:3000/incidents/maps/1.xml`**
**`http://localhost:3000/incidents/maps/1.html`**

One URL returns the XML data; the other returns the HTML.

## How should we decide which format to use?

If we add an extra route that includes the format in the path:

```
map.connect 'incidents/map/:id.:format', :action=>'show_with_map',
   :controller=>'incidents'
```

This will record the format from the extension.

we will be able to read the requested format from the XML and then make decisions in the code like this:

```
if params[:format] == 'html'
  # Generate the HTML representation
else
  # Generate the XML representation
end
```

http://localhost:3000/incidents/map/1.html

This extension will be stored in the :format field.

http://localhost:3000/incidents/map/1.xml

But that's not how most Rails applications choose the format to generate. Instead they call a method called `respond_to do` and an object called a **`responder`**:

```
respond_to do |format|
  format.html {
    ____________________
  }
  format.xml {
    ____________________
  }
end
```

format is a 'responder' object.

The code to generate a web page goes here.

The code to generate the XML goes here.

This code does more or less the same thing. The `format` object is a `responder`. A responder can decide whether or not to run code, dependent upon the format required by the request. So if the user asks for HTML, the code above will run the code passed to `format.html`. If the user asks for XML, the responder will run the code passed to `format.xml`.

So why don't Rails programmers just use an `if` statement? After all, wouldn't that be simpler code? Well, the `responder` has **hidden powers**. For example, it sets the mime type of the response. The mime type tells the browser what data-type the response is. In general, it is much better practice to use `respond_to do` to decide what representation format to generate.

**Exercise**

The `show_with_map` method in the controller needs to choose whether it should generate XML or HTML. Write a new version of the method that uses a responder to generate the correct representation.

Hint: If you need to generate HTML, other than reading a model object, what else does the controller need to do?

The `show_with_map.html.erb` page template currently calls the map partial and passes it the `/test.xml` file. What will the partial call look like if it is going to call the generated XML file?

**Exercise Solution**

The `show_with_map` method in the controller needs to choose whether it should generate XML or HTML. Write a new version of the method that uses a responder to generate the the correct representation.

Hint: If you need to generate HTML, other than reading a model object, what else does the controller need to do?

Nothing! When generating HTML we can leave Rails to call the show_with_map.html.erb template

```
def show_with_map
  @incident = Incident.find(params[:id])
  respond_to do |format|
    format.html {
    }
    format.xml {
      render :text=>@incident.to_xml(
        :only=>[:latitude,:longitude,:title,:description],
        :root=>"name")
    }
  end
end
```

We can leave this empty – Rails will call the template for us

The `show_with_map.html.erb` page template currently calls the map partial and passes it the `/test.xml` file. What will the partial call look like if it is going to call the generated XML file?

```
<%= render(:partial=>'map', :locals=>{:data=>"#{@incident.id}.xml"}) %>
```

## there are no Dumb Questions

**Q: If the format.html section doesn't need any code, can we just skip it?**

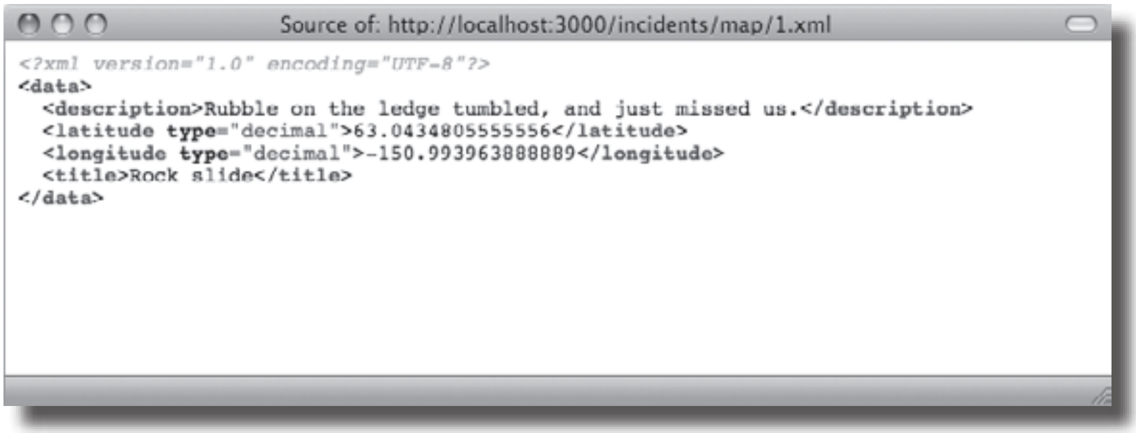
A: No. You still need to include format.html, or Rails won't realize that it needs to respond to requests for HTML output.

## Test Drive

If we look at the XML version of the page at:

`http://localhost:3000/incidents/map/1.xml`

we get an XML version of the incident:

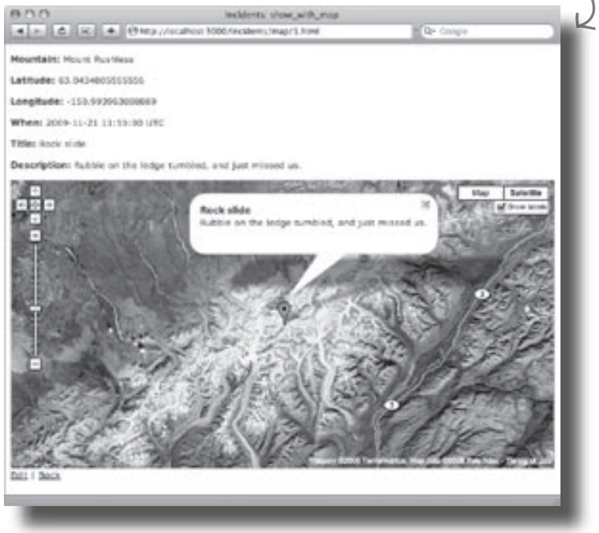


Source of: http://localhost:3000/incidents/map/1.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
  <description>Rubble on the ledge tumbled, and just missed us.</description>
  <latitude type="decimal">63.0434805555556</latitude>
  <longitude type="decimal">-150.993963888889</longitude>
  <title>Rock slide</title>
</data>
```

So what about the HTML version:

http://localhost:3000/incidents/map/1.html



Incidents: show_with_map

**Mountain:** Mount Rushless

**Latitude:** 63.0434805555556

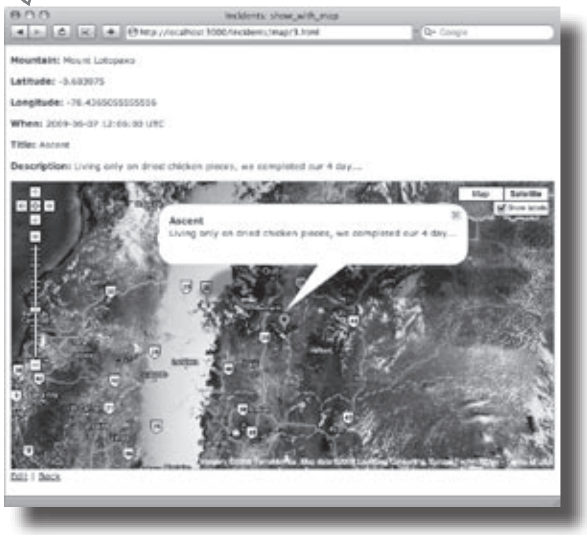
**Longitude:** -150.993963888889

**When:** 2009-11-21 11:55:00 UTC

**Title:** Rock slide

**Description:** Rubble on the ledge tumbled, and just missed us.

http://localhost:3000/incidents/map/3.html



Incidents: show_with_map

**Mountain:** Mount Lotopaxo

**Latitude:** -3.683975

**Longitude:** -76.4365055555556

**When:** 2009-06-07 12:06:00 UTC

**Title:** Ascent

**Description:** Living only on dried chicken pieces, we completed our 4 day...

Map
Satellite
Show labels
Ascent
Living only on dried chicken pieces, we completed our 4 day...
Edit | Back

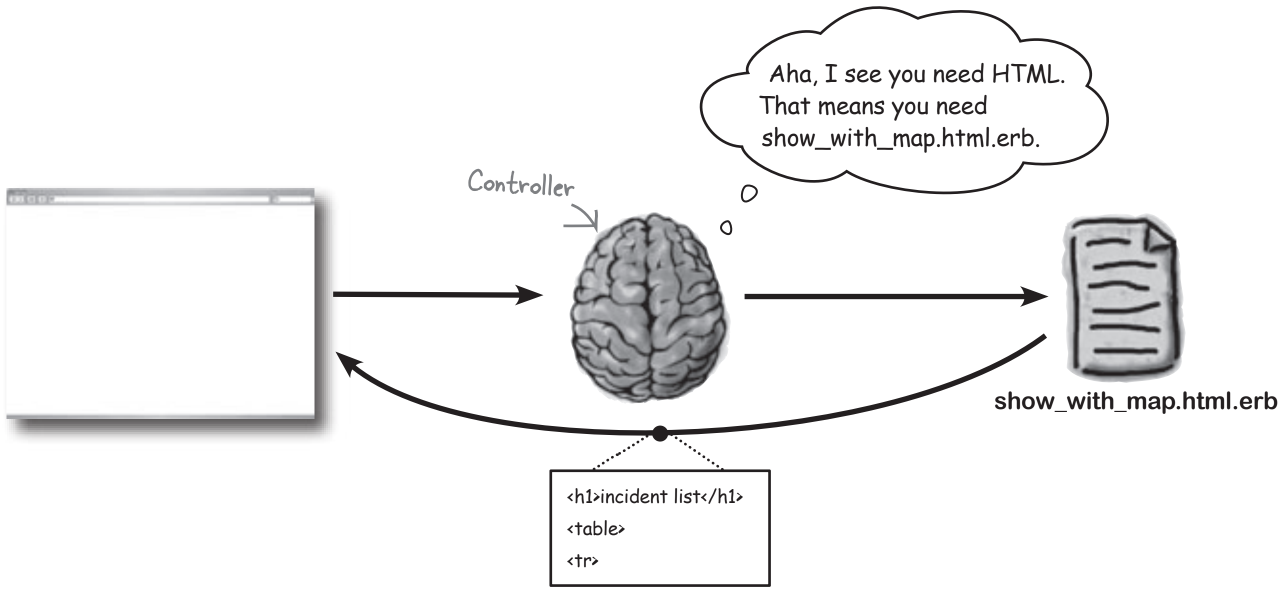
It works. Now different incidents show different maps. But before we replace the live version of the code, we better make sure we understand exactly how the code works.

**So what really went on here?**

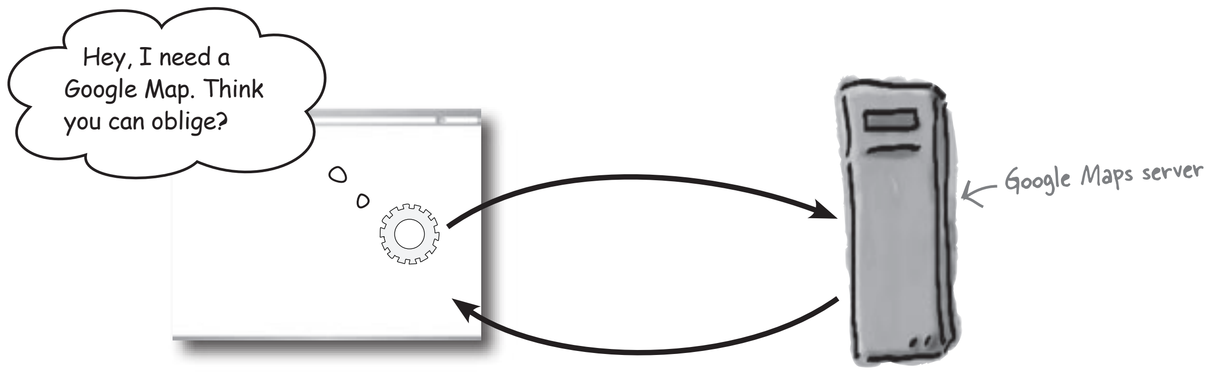
## How does the map page work?

Let's take a deeper look at what just happened and how the HTML page is rendered.

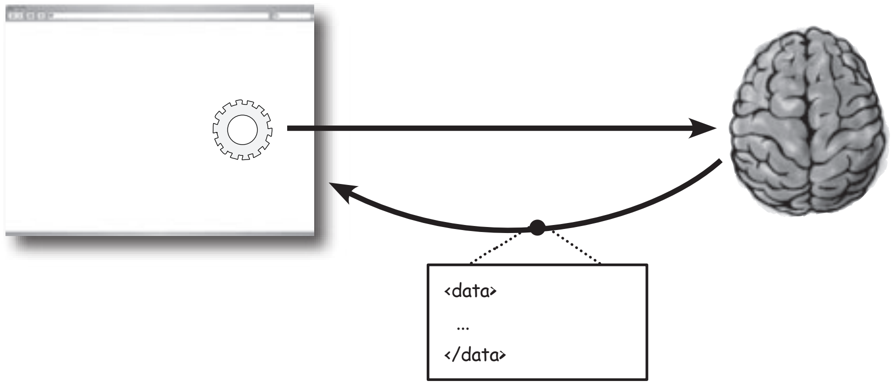
1. **The controller spots that an HTML page is needed.**
The browser points to the HTML version of the page. The controller realizes that HTML rather than XML is required, and so calls `show_with_map.html.erb`. HTML is sent back to the client browser.


Aha, I see you need HTML. That means you need show_with_map.html.erb.
Controller
show_with_map.html.erb
<h1>incident list</h1>
<table>
<tr>

2. **JavaScript requests the Google Map.**
JavaScript within the web page requests map data from the Google Maps server. The Google Maps server returns it.



**3** **JavaScript requests the incident XML.**
JavaScript within the page requests XML for the incident from the controller. It then displays it on the map.



## there are no Dumb Questions

**Q: You say that a resource should always have the same URL. Why is that?**

A: It doesn't *have to*, but REST—Rails' main design principle—says it should.

**Q: But if the format is in the URL, doesn't that mean that different URLs are used for the same resource?**

A: Yes, sure does. Adding the format to the URL compromises the RESTfulness of the design... a little bit. But it's a common trick. It's simple, and works well.

**Q: So there's no way to use the same URL for different formats?**

A: There is a way to do it. If the request contains an "Accepts:" header say—for example—that the request is for "text/xml", the responder will run the code for the XML format.

**Q: Is there a way of listing the attributes you *don't* want to include in to_xml output?**

A: Yes. Instead of using the `:only` parameter, you can use the `:except` parameter. Rails is remarkably consistent and you will found several places where calls in Rails have optional `:only` parameters. In all cases you can switch them for `:except` parameters to say which things you ***don't*** want.

**Q: Is there some way that the controller can tell the difference between an Ajax request from JavaScript and a browser request?**

A: Sort of. The expression `request.xhr?` usually returns 'true' for Ajax requests and 'false' for simple browser requests. The problem is that while it works for the requests generated by the Prototype library, it doesn't work with *all* Ajax libraries.

**Q: Why do I have to call render sometimes and not others?**

A: If you are happy to run the default template (the one whose name matches the action), you can omit the `render` call.

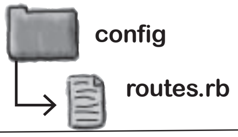
**Q: You say that the generated XML and the HTML are different representations, but they don't contain the same information, do they?**

A: That's true—they don't. The XML generated for a single incident contains a smaller amount of data than the HTML representation. But they both present information about the same resource, so they are both representations of the same thing.

## The code is ready to go live

Our new version of the location page works well, so let's replace the scaffolded show action with the `show_with_map` code.

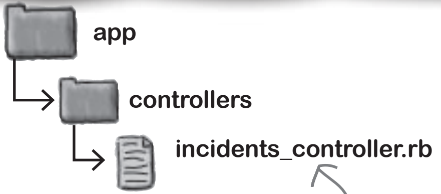
1 **Remove the routes.**
We created custom routes for the test code, so we need to remove them from the routes.rb file:



Get rid of these lines.

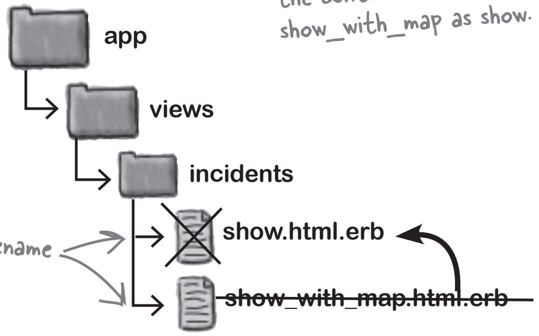
```
ActionController::Routing::Routes.draw do |map|
  ~~map.connect 'incidents/map/:id', :action=>'show_with_map', :controller=>'incidents'~~
  ~~map.connect 'incidents/map/:id.:format', :action=>'show_with_map', :controller=>'incidents'~~
  map.resources :incidents
```

2 **Rename the show_with_map method in the controller.**
`show_with_map` is going to become our new `show` method. So delete the existing `show` method and rename `show_with_map` to `show`.



Delete the show method in the controller, then rename show_with_map as show.

3 **Then rename the show_with_map.html.erb template.**
That means we need to delete the existing `show.html.erb` and replace it with the `show_with_map.html.erb` template.



Delete show.html.erb, and rename show_with_map.html.erb as show.html.erb.

### there are no Dumb Questions

**Q: If the route disappeared, how did the right format get chosen?**

A: The map.resource route sets up a whole set of routes. These routes all include the format.

**Q: How come the index page went to "/incidents/1" instead of "/incidents/1.html"? How did Rails know it was going to be HTML?**

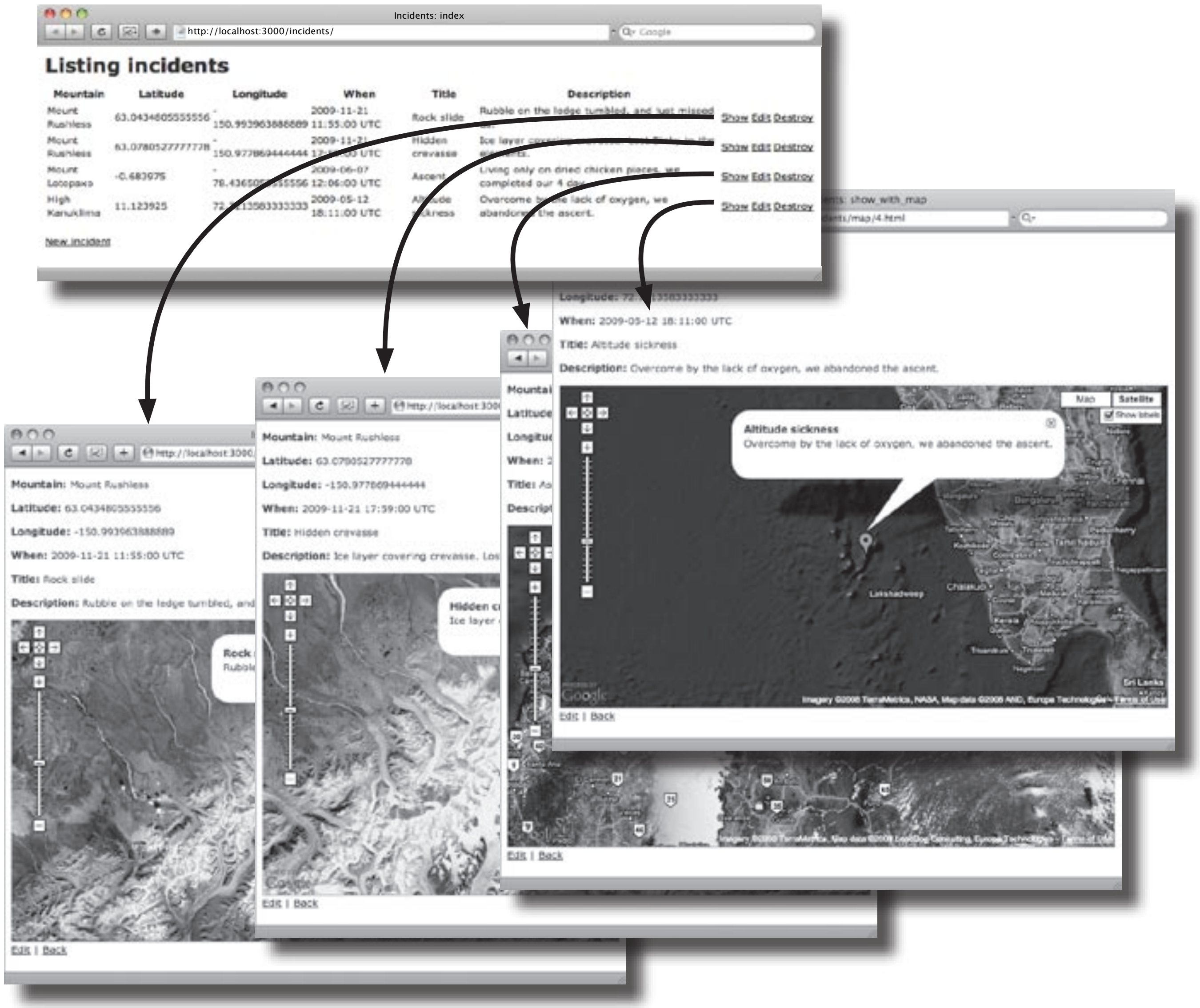
A: If the format isn't given, Rails assumes HTML... which we used to our advantage.

**Q: What does map.resources mean?**

A: That generates the standard set of routes used by scaffolding.

## Test Drive

Now the the mapped pages have replaced the default "show" action. So now the main index page links to the mapping pages, not the text versions.



Incidents: index

http://localhost:3000/incidents/

**Listing incidents**

| Mountain | Latitude | Longitude | When | Title | Description | | | |
|---|---|---|---|---|---|---|---|---|
| Mount Rushless | 63.0434805555556 | -150.993963888889 | 2009-11-21 11:55:00 UTC | Rock slide | Rubble on the ledge tumbled, and just missed us. | Show | Edit | Destroy |
| Mount Rushless | 63.0780527777778 | -150.977869444444 | 2009-11-21 [illegible] UTC | Hidden crevasse | Ice layer [illegible] elements. | Show | Edit | Destroy |
| Mount Lotopaxo | -0.683975 | -78.4365[illegible]5556 | 2009-06-07 12:06:00 UTC | Ascent | Living only on dried chicken pieces, we completed our 4 day[illegible] | Show | Edit | Destroy |
| High Kanuklima | 11.123925 | 72.[illegible]13583333333 | 2009-05-12 18:11:00 UTC | Altitude sickness | Overcome by the lack of oxygen, we abandoned the ascent. | Show | Edit | Destroy |

New incident

**Mountain:** Mount Rushless

**Latitude:** 63.0434805555556

**Longitude:** -150.993963888889

**When:** 2009-11-21 11:55:00 UTC

**Title:** Rock slide

**Description:** Rubble on the ledge tumbled, and

Edit | Back

**Mountain:** Mount Rushless

**Latitude:** 63.0780527777778

**Longitude:** -150.977869444444

**When:** 2009-11-21 17:59:00 UTC

**Title:** Hidden crevasse

**Description:** Ice layer covering crevasse. Los

Edit | Back

**Longitude:** 72.[illegible]13583333333

**When:** 2009-05-12 18:11:00 UTC

**Title:** Altitude sickness

**Description:** Overcome by the lack of oxygen, we abandoned the ascent.

Edit | Back

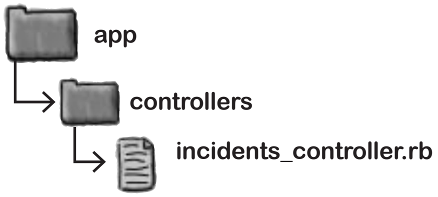
**One thing though - isn't that index page kind of... boring? Especially compared to all those nice visual map pages!**

## Long Exercise

The users have asked if the index page can display a whole set of all the incidents that have been recorded, and fortunately the `_map.html.erb` partial can generate multiple points if it is given the correct XML data.

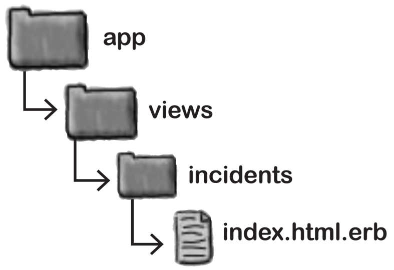
This is the existing index method in the incidents controller. Rewrite the method to generate XML from the array of all incidents. You only need to change the root element to "data".

```
def index
  @incidents = Incident.find(:all)

  respond_to do |format|
    format.html # index.html.erb
    format.xml  { render :xml => @incidents }
  end
end
```



The index page will need to include a map. Write the code to insert the map at the given point. You will need to pass the path of the XML version of the index page as data for the map.

```
<h1>Listing incidents</h1>

<table>
  <tr>
    <th>Mountain</th>
    <th>Latitude</th>
    <th>Longitude</th>
    <th>When</th>
    <th>Title</th>
    <th>Description</th>
  </tr>

<% for incident in @incidents %>
  <tr>
    <td><%=h incident.mountain %></td>
    <td><%=h incident.latitude %></td>
    <td><%=h incident.longitude %></td>
    <td><%=h incident.when %></td>
    <td><%=h incident.title %></td>
    <td><%=h incident.description %></td>
    <td><%= link_to 'Show', incident %></td>
    <td><%= link_to 'Edit', edit_incident_path(incident) %></td>
    <td><%= link_to 'Destroy', incident, :confirm => 'Are you sure?',
            :method => :delete %></td>
  </tr>
<% end %>
</table>


.......................................................................

<br />

<%= link_to 'New incident', new_incident_path %>
```

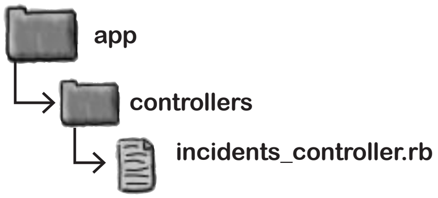


The users have asked if the index page can display a whole set of all the incidents that have been recorded and fortunately the `_map.html.erb` partial can generate multiple points if it is given the correct XML data.

This is the existing index method in the incidents controller. Rewrite the method to generate XML from the array of all incidents. You only need to change the root element to "data".

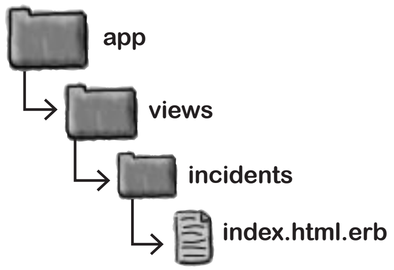
```
def index
  @incidents = Incident.find(:all)

  respond_to do |format|
    format.html # index.html.erb
    format.xml  { render :xml => @incidents }
  end
end
```

```
def index
  @incidents = Incident.find(:all)

  respond_to do |format|
    format.html # index.html.erb
    format.xml  {
      render :text=>@incidents.to_xml(:root=>"data")
    }
  end
end
```



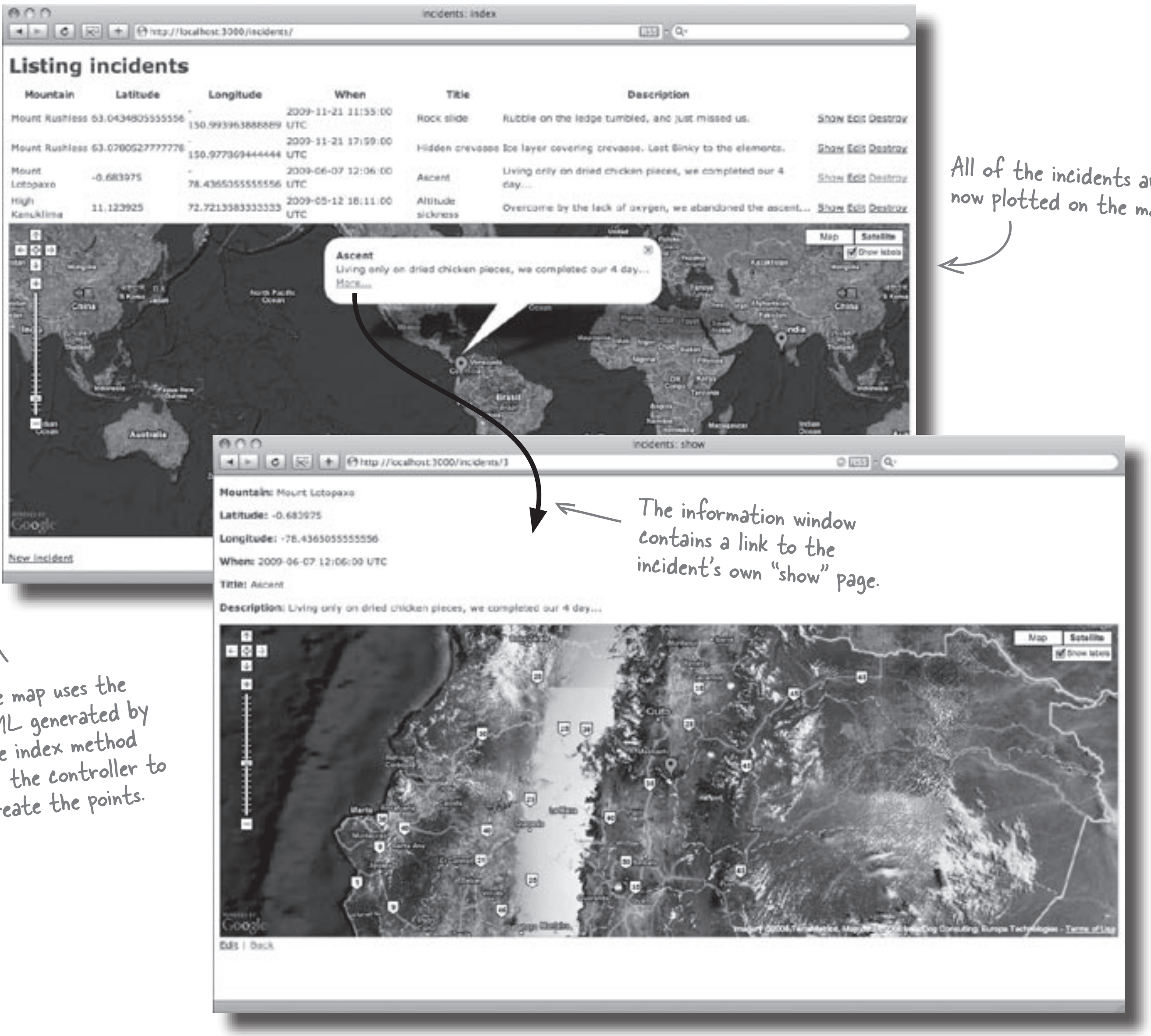
The index page will need to include a map. Write the code to insert the map at the given point. You will need to pass the path of the XML version of the index page as data for the map.

```
<h1>Listing incidents</h1>

<table>
  <tr>
    <th>Mountain</th>
    <th>Latitude</th>
    <th>Longitude</th>
    <th>When</th>
    <th>Title</th>
    <th>Description</th>
  </tr>

<% for incident in @incidents %>
  <tr>
    <td><%=h incident.mountain %></td>
    <td><%=h incident.latitude %></td>
    <td><%=h incident.longitude %></td>
    <td><%=h incident.when %></td>
    <td><%=h incident.title %></td>
    <td><%=h incident.description %></td>
    <td><%= link_to 'Show', incident %></td>
    <td><%= link_to 'Edit', edit_incident_path(incident) %></td>
    <td><%= link_to 'Destroy', incident, :confirm => 'Are you sure?',
            :method => :delete %></td>
  </tr>
<% end %>
</table>

  <%= render (:partial=>'map', :locals=>{:data=>"/incidents.xml"}) %>

<br />

<%= link_to 'New incident', new_incident_path %>
```



## Test Drive

Now when users go to the front page, they see the incidents in a list and on the map. When an incident is clicked, the details are displayed, as well as a link to the incident's own page.


incidents: index
http://localhost:3000/incidents/
Map
Satellite
Show labels
incidents: show
http://localhost:3000/incidents/3
Edit | Back

Listing incidents

| Mountain | Latitude | Longitude | When | Title | Description | |
|---|---|---|---|---|---|---|
| Mount Rushless | 63.0434805555556 | -150.993963888889 | 2009-11-21 11:55:00 UTC | Rock slide | Rubble on the ledge tumbled, and just missed us. | Show Edit Destroy |
| Mount Rushless | 63.0780527777778 | -150.977369444444 | 2009-11-21 17:59:00 UTC | Hidden crevasse | Ice layer covering crevasse. Lost Binky to the elements. | Show Edit Destroy |
| Mount Lotopaxo | -0.683975 | -78.4365055555556 | 2009-06-07 12:06:00 UTC | Ascent | Living only on dried chicken pieces, we completed our 4 day... | Show Edit Destroy |
| High Kanuklima | 11.123925 | 72.7213583333333 | 2009-05-12 18:11:00 UTC | Altitude sickness | Overcome by the lack of oxygen, we abandoned the ascent... | Show Edit Destroy |

Ascent
Living only on dried chicken pieces, we completed our 4 day...
More...

New incident

**Mountain:** Mount Lotopaxo

**Latitude:** -0.683975

**Longitude:** -78.4365055555556

**When:** 2009-06-07 12:06:00 UTC

**Title:** Ascent

**Description:** Living only on dried chicken pieces, we completed our 4 day...

All of the incidents are now plotted on the map.

The information window contains a link to the incident's own "show" page.

The map uses the XML generated by the index method of the controller to create the points.

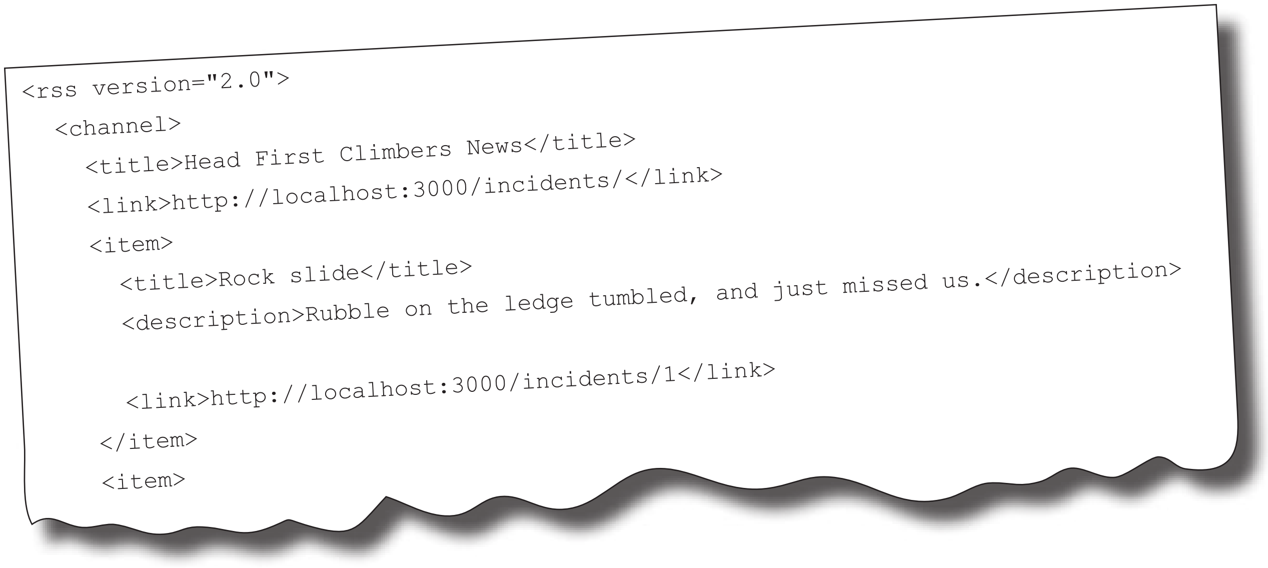
Hey, there's so much data now! I'd really like to know about the incidents that have been posted in the last 24 hours. How about a news feed?

Most web sites now provide ***RSS news feeds*** to provide easy links to the main resources on a site.

**But what does an RSS news feed look like?**

## RSS feeds are just XML

This is what an RSS feed file would look like for the climbing site:



```
<rss version="2.0">
  <channel>
    <title>Head First Climbers News</title>
    <link>http://localhost:3000/incidents/</link>
    <item>
      <title>Rock slide</title>
      <description>Rubble on the ledge tumbled, and just missed us.</description>

      <link>http://localhost:3000/incidents/1</link>
    </item>
    <item>
```

This is just an XML file. If you use an RSS news reader, or if your browser can subscribe to RSS news feeds, they will download a file just like this, which contains a list of links and descriptions to news stories.

**So how can WE generate an RSS feed like this?**

**BRAIN POWER**

Do any of the tags in the RSS look particularly surprising or unclear? What do you think channel does? What about link?

## We'll create an action called news

Let's create a new route as follows:

```
map.connect '/incidents/news', :action=>'news', :controller=>'incidents', :format=>'xml'
```

### Sharpen your pencil

Write the controller method for the new action. It needs to find all incidents with `updated_at` in the last 24 hours. It should then render the default XML by calling `to_xml` on the array of matching incidents.

Hint: The Ruby expression `Time.now.yesterday` returns a date-time value from exactly 24 hours ago.

## Sharpen your pencil Solution

Write the controller method for the new action. It needs to find all incidents with `updated_at` in the last 24 hours. It should then render the default XML by calling `to_xml` on the array of matching incidents.

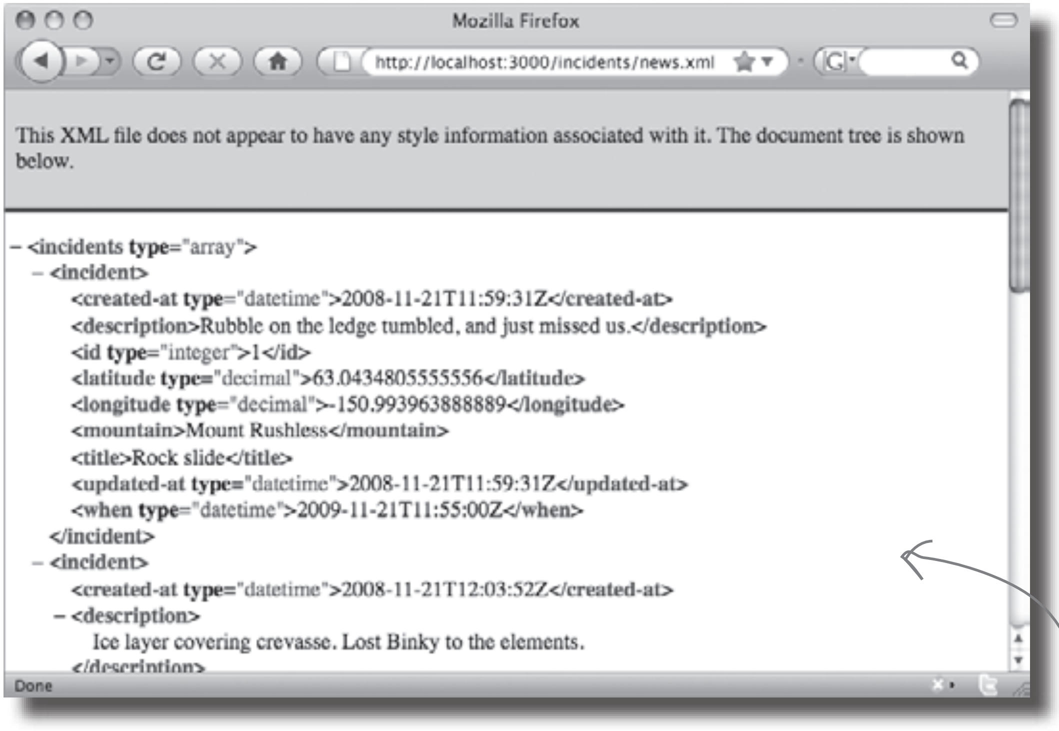
Hint: The Ruby expression `Time.now.yesterday` returns a date-time value from exactly 24 hours ago.

```
def news
  @incidents = Incident.find(:all, :conditions=>['updated_at > ?', Time.now.yesterday])
  render :xml=>@incidents
end
```

You could have also used :text=>@incidents.to_xml.

## Test Drive

This is the XML that is generated by the news action:



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
– <incidents type="array">
  – <incident>
      <created-at type="datetime">2008-11-21T11:59:31Z</created-at>
      <description>Rubble on the ledge tumbled, and just missed us.</description>
      <id type="integer">1</id>
      <latitude type="decimal">63.0434805555556</latitude>
      <longitude type="decimal">-150.993963888889</longitude>
      <mountain>Mount Rushless</mountain>
      <title>Rock slide</title>
      <updated-at type="datetime">2008-11-21T11:59:31Z</updated-at>
      <when type="datetime">2009-11-21T11:55:00Z</when>
    </incident>
  – <incident>
      <created-at type="datetime">2008-11-21T12:03:52Z</created-at>
    – <description>
        Ice layer covering crevasse. Lost Binky to the elements.
      </description>
```

Remember – this is time dependent so incidents will only appear if they've been modified in the last 24 hours

We've generated XML for the correct data, but it's not the sort of XML we need for an RSS news feed. That's OK though, we had that problem before. When we were generating XML data for the location data it was in the wrong format, and we were able to adjust it then.

**We just need to modify this XML in the same way... don't we?**

### Brain Power

Is there a problem converting the XML to match the structure of the RSS news feed?

## We have to change the structure of the XML

The `to_xml` method allows us to make a few simple changes to the XML it produces. We can swap names and choose which data items to include. But will it give us enough power to turn the XML we *have* into the XML we *want*?

This is what we have...

```
<?xml version="1.0" encoding="UTF-8"?>
<incidents type="array">
  <incident>
    <created-at type="datetime">2008-11-21T11:59:31Z</created-at>
    <description>Rubble on the ledge tumbled, and just missed us.</description>
    <id type="integer">1</id>
    <latitude type="decimal">63.0434805555556</latitude>
    <longitude type="decimal">-150.993963888889</longitude>
    <mountain>Mount Rushless</mountain>
    <title>Rock slide</title>
```

```
<rss version="2.0">
  <channel>
    <title>Head First Climbers News</title>
    <link>http://localhost:3000/incidents/</link>
    <item>
      <title>Rock slide</title>
      <description>Rubble on the ledge tumbled, and just missed us.</description>
      <link>http://localhost:3000/incidents/1</link>
    </item>
    <item>
```

... but this is what we want.

### We need more XML POWER

The news feed XML can't be generated by the `to_xml` method. While `to_xml` can modify XML output slightly, it can't radically change XML structure. For instance, `to_xml` can't move elements between levels. It can't group elements within other elements. `to_xml` is designed to be quick and easy to use, but that also makes it a bit inflexible.

**For true XML power, we need something more...**

## So we'll use a new kind of template: an XML builder

If we created another HTML page template, we could generate whatever XML output we like. After all, HTML is similar to XML:

This actually looks a whole lot like HTML...

```
<rss version="2.0">
  <channel>
    <title>Head First Climbers News</title>
    <link>http://localhost:3000/incidents/</link>
    <% for incident in @incidents %>
    <item>
      <title><%= h incident.title %></title>
      <description><%= h incident.description %></description>
```

But Rails provides a special type of template that is specifically designed to generate XML; it's called an **XML Builder Template**.

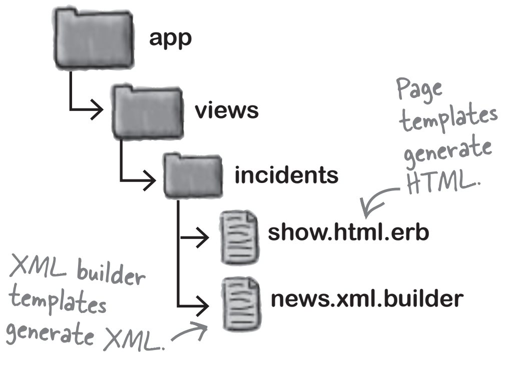
XML Builders live in the same directory as page templates, and they are used in a similar way. If someone has requested an XML response (by adding .xml to the end of the URL), the controller only needs to read the data from the model, and Rails will automatically call the XML builder template. That means we can lose a line from the **news** action:

This is the "new" method from the incidents controller.

```
def news
  @incidents = Incident.find(:all, :conditions=>['updated_at > ?', Time.now.yesterday])
  ~~render :xml=>@incidents~~
end
```

This code will now just read the data from the model and the XML bulder template will do the rest.

**So what does an XML builder look like?**


app
views
incidents
show.html.erb
news.xml.builder
Page templates generate HTML.
XML builder templates generate XML.

## XML Builders Up Close

Page templates are designed to look like HTML files with a little Ruby sprinkled in. XML builders are different. They are pure Ruby but are designed to have a structure similar to XML. For example, this:

```
xml.sentence(:language=>'English') {
  for word in @words do
    xml.word(word)
  end
}
```

might generate something that looks like this:

```
<sentence language="English">  <-- Attribute
  <word>XML</word>
  <word>Builders</word>
  <word>Kick</word>  <-- Elements
  <word>Ass!</word>
</sentence>
```

So why did the Rails folks make a different kind of template? Why doesn't XML Builder work just like a Page Template? Why doesn't it use Embedded Ruby?

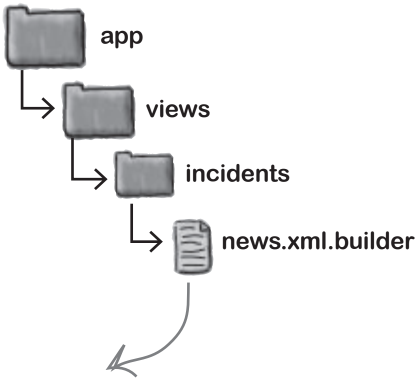
Even though XML and HTML are very similar—and in the case of XHTML, they are technically equal—the ways in which people use HTML and XML are subtly different.

- Web pages usually contain a lot of **HTML** markup to make the page look nice, and just a *little* data from the database.
- Most of the content of the **XML**, on the other hand, is likely to come from the data and conditional logic and far less from the XML markup.

Using Ruby—instead of XML—as the main language, makes XML Builders more concise and easier to maintain.

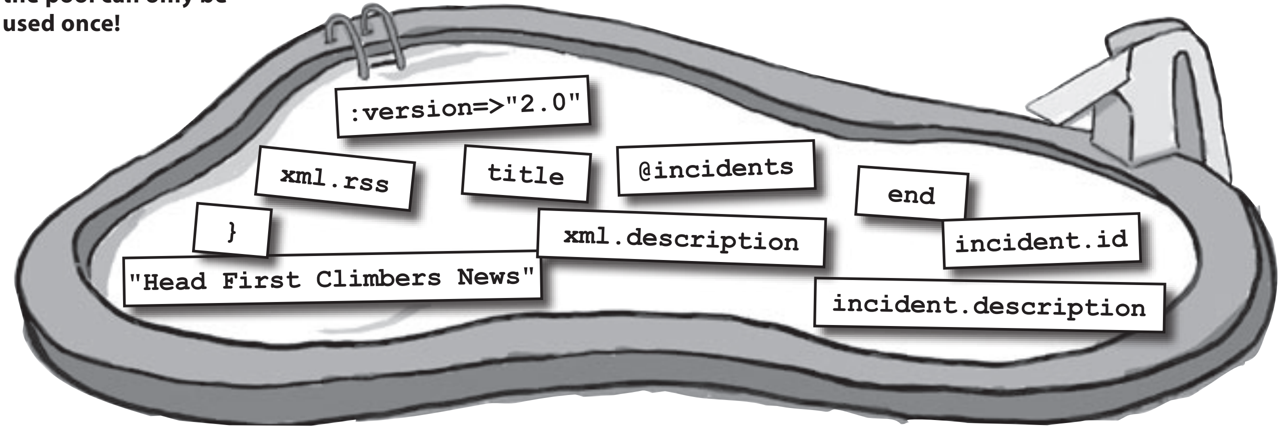
## Pool Puzzle

Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to complete the XML builder template that will generate RSS.



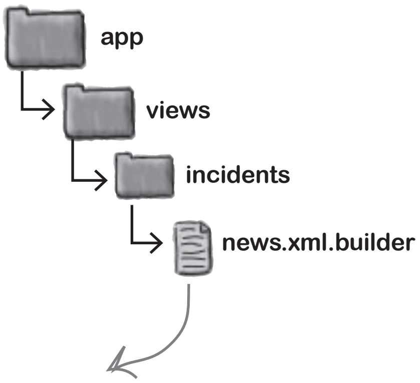
```
.................(......................................) {
   xml.channel {
     xml.title(......................................................)
     xml.link("http://localhost:3000/incidents/")
     for incident in ..........................
       xml.item {
         xml.................(incident.title)
        ........................................( ...........................................)
         xml.link("http://localhost:3000/incidents/#{.........................}")
       }

     ....................

    ....................
 }
```

**Note: each thing from the pool can only be used once!**

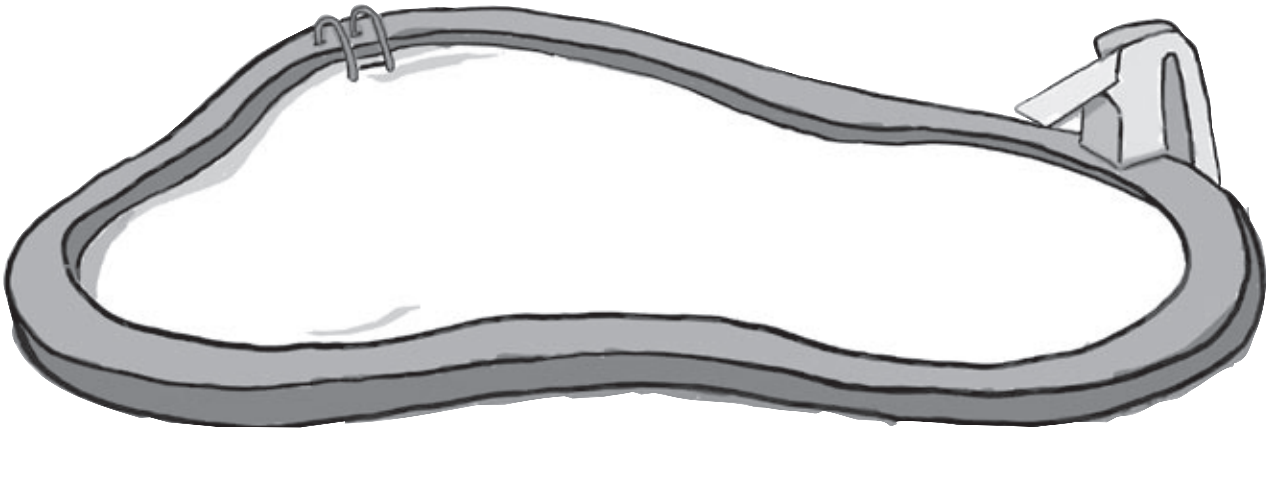


## Pool Puzzle Solution

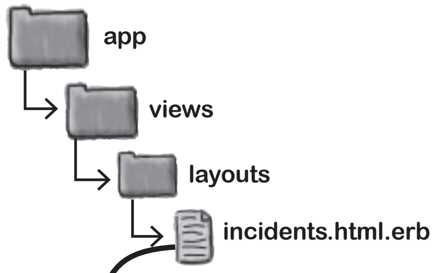
Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to complete the XML builder template that will generate RSS.



```
xml.rss ( :version=>"2.0" ) {
  xml.channel {
    xml.title( "Head First Climbers News" )
    xml.link("http://localhost:3000/incidents/")
    for incident in @incidents
      xml.item {
        xml.title (incident.title)
        xml.description ( incident.description )
        xml.link("http://localhost:3000/incidents/#{ incident.id }")
      }
    end
  }
}
```



## Now let's add the feed to the pages



But how will users find the feed? Browsers sense the presence of a news feed by looking for a `<link... />` reference within a page.

The folks at Head First Climbers want the news feed to appear on every page, so we will add a reference to the RSS feed in the `incidents` layout file, using the `auto_discovery_link` helper:

```
<!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>Incidents: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
  <%= auto_discovery_link_tag(:rss, {:action=>'news'}) %>
</head>
<body>

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

<%= yield  %>

</body>
</html>
```

This should create a link like this:

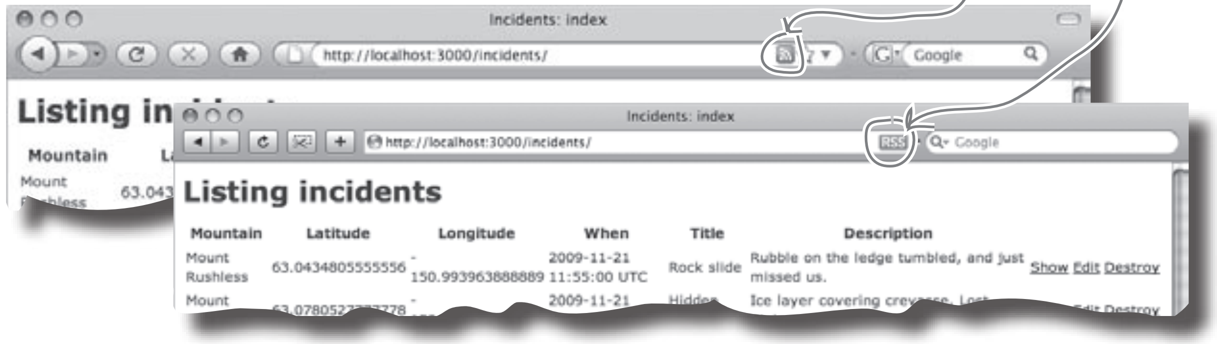
```
<link href="http://localhost:3000/incidents/news.xml"
  rel="alternate" title="RSS" type="application/rss+xml" />
```

**But to see if it works, we need to fire up our web browser again.**

## Test Drive

Now, when a user goes to the web site, an RSS feed icon appears in their browser:

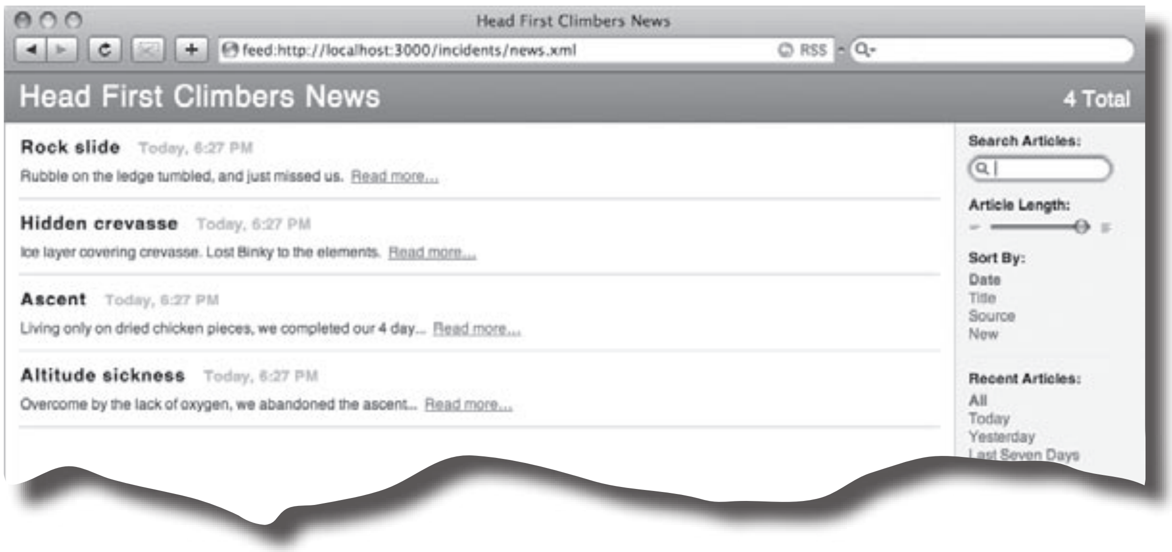
Different browsers have different ways of showing they have found a news feed.



Listing incidents

| Mountain | Latitude | Longitude | When | Title | Description | | | |
|---|---|---|---|---|---|---|---|---|
| Mount Rushless | 63.0434805555556 | -150.993963888889 | 2009-11-21 11:55:00 UTC | Rock slide | Rubble on the ledge tumbled, and just missed us. | Show | Edit | Destroy |
| Mount | 63.0780527777778 | - | 2009-11-21 | Hidden | Ice layer covering crevasse. Lost | | Edit | Destroy |

And if they subscribe to the feed, or simply read it, they will see links to incidents that have been posted in the previous 24 hours.



Head First Climbers News 4 Total

**Rock slide** Today, 6:27 PM
Rubble on the ledge tumbled, and just missed us. Read more...

**Hidden crevasse** Today, 6:27 PM
Ice layer covering crevasse. Lost Binky to the elements. Read more...

**Ascent** Today, 6:27 PM
Living only on dried chicken pieces, we completed our 4 day... Read more...

**Altitude sickness** Today, 6:27 PM
Overcome by the lack of oxygen, we abandoned the ascent... Read more...

## On top of the world!

One of the first news items on the web site is posted by our intrepid climber, and thousands of climbers hear of the good news.

## Tools for your Rails Toolbox

**You've got Chapter 8 under your belt, and now you've added the ability to use XML to represent your pages in multiple ways.**

### Rails Tools

to_xml generates an XML for any model object

:only and :root parameters allow you to modify the to_xml format

respond_to creates a _responder_ object that will help you generate multiple representations for a resource

XML builder templates are like page templates for creating XML

XML builder templates give you more flexibility than by simply using to_xml

responders set the response mime-type and also decide whether to call page templates or XML builder templates