Application Express Listener

Developer Guide

Table of Contents

Introduction

Oracle Application Express Listener enables data stored in an Oracle Database to be exposed via RESTful Application Programming Interfaces (APIs).

What is a RESTful API?

Representational State Transfer (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web. An API is said to be 'RESTful' when it conforms to the tenets of REST. A full discussion of REST is outside the scope of this document but in summary a RESTful API has the following characteristics:

- Data is modelled as a set of 'Resources', Resources are identified by URLs
- A small set of operations (e.g. `PUT`, `POST`, `GET`, `DELETE`) are used to manipulate Resources.
- A Resource can have multiple representations (for example a blog might have a HTML representation and a RSS representation)
- Services are stateless and since it is likely that the client will want to access related Resources, these should be identified in the representation returned, typically by providing hypertext links

Resource Templates

RESTful APIs are created by configuring 'Resource Templates'. A Resource Template is a configuration file that binds a set of URIs to a SQL query or anonymous PL/SQL block. The set of URIs is identified by a 'URI Template', a simple syntax for describing URIs e.g. `people/{userid}`. The URI Template may contain zero or more parameters (e.g. `{userid}`) which along with the HTTP Headers sent with a HTTP request can be bound to parameters in the SQL query or anonymous PL/SQL block.

Resource Templates Administration

The Resource Template Administration GUI can be accessed at the following URL:

`http://hostname:port/apex/resourceTemplates/`

To access the GUI please input the credentials for the APEX Listener administrator.

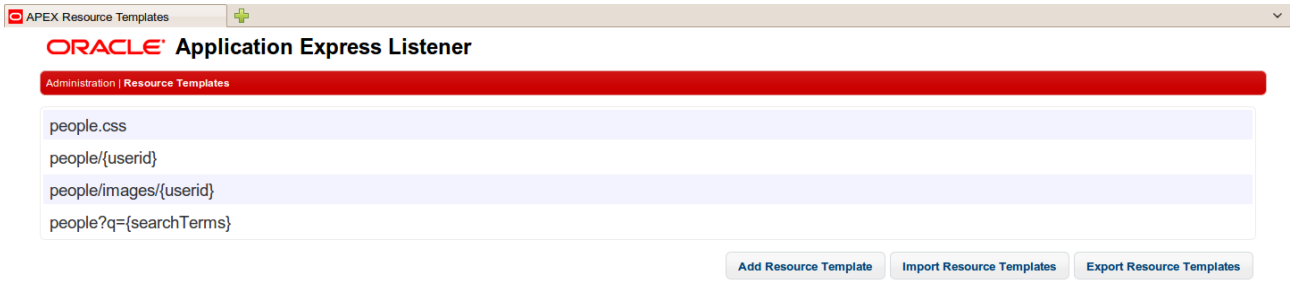
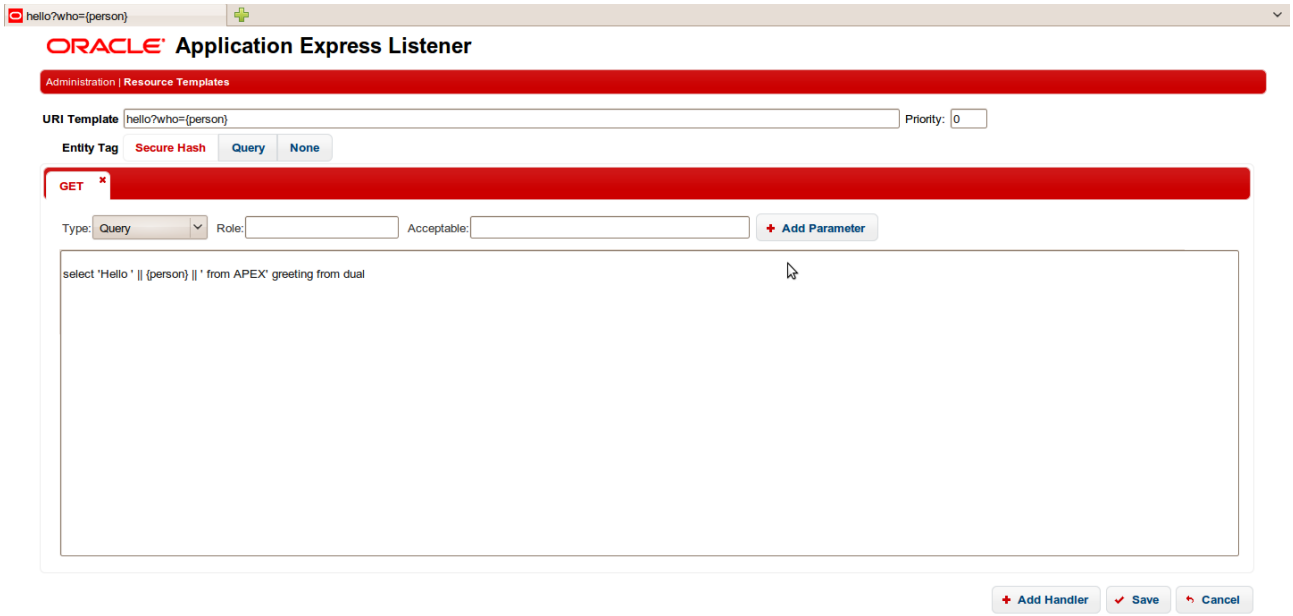


Illustration 1: Resource Templates Administration GUI

This page displays a list of the currently defined Resource Templates, and provides buttons to create a new Resource Template, import Resource Template Definitions, and to export the existing defintiions to a zip file.

Create a Resource Template

Press the 'Add Resource Template' button to create a new Resource Template. The following is displayed:



The GUI is pre-popluated with a sample Resource Template definition. You can hover the mouse cursor over each field in the GUI to see a tooltip that will describe the purpose of the field.

URI Template

A URI Template is a compact syntax for describing a range of URIs. For example assume the APEX Listener is running on a machine named example.com, then the URI Template:

`hello?who={person}`

would match a URI like:

`http://example.com/apex/hello?who=World`.

The literal value '`World`' is bound to the URI Template parameter named: `{person}`

URI Template Grammar

At present there is no standardised URI Template syntax (although work is progressing on one). APEX Listener supports a limited subset of the current draft specification. The following grammar rules apply:

- URI Templates must not start with a slash ('`/`') character
- URI Templates can contain any Unicode character
- Parameters are declared with the use of curly braces ('`{`','`}`')
- Parameter names must start with an alphabetic character followed by zero or more alphanumeric characters and the characters: '`_`' or '`-`'
- There is no support for optional parameters

Entity Tag

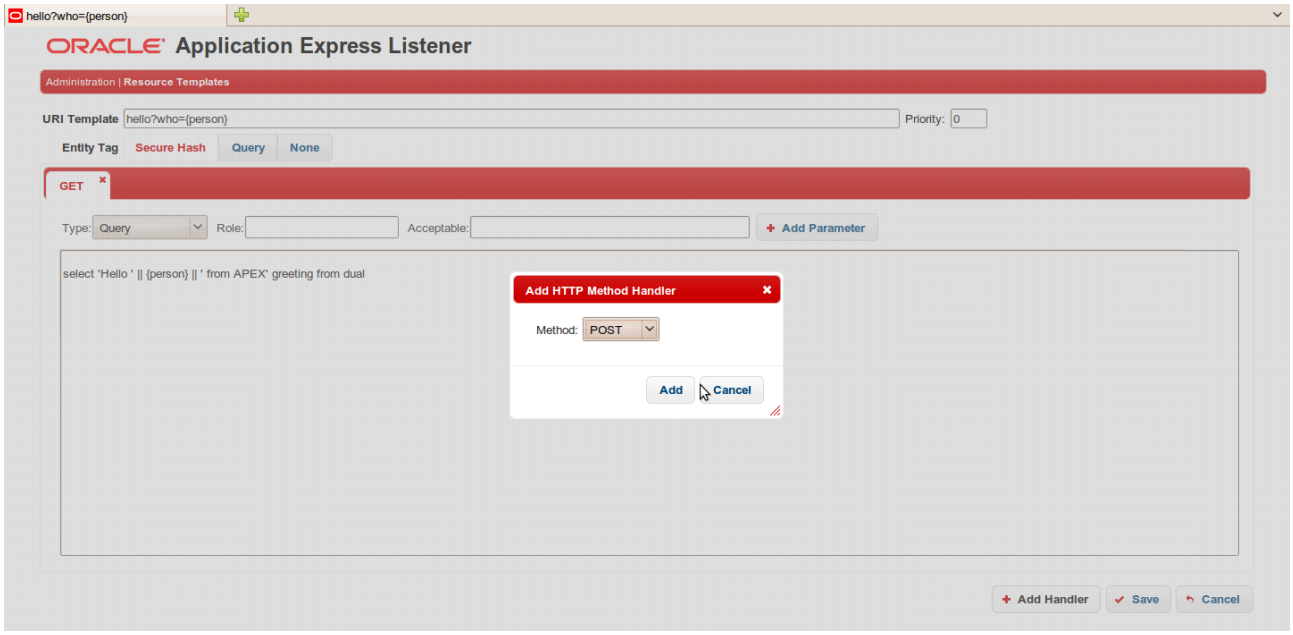
An Entity Tag (ETag) is a HTTP Header that acts as a version identifier for a resource. ETag Headers are used to avoid retrieving previously retrieved resources and to perform optimistic locking when updating resources.

APEX Listener provides three strategies for generating Entity Tags:

- Secure Hash: The contents of the returned resource representation are hashed using a secure digest function to provide a unique fingerprint for a given resource version.
- Query: Manual define a query that uniquely identifies a resource version. Instead of hashin the entire resource representation only the result of the query is hashed. A manually defined query can often generate an Entity Tag more efficently that hasing the entire resource representation.
- None: Do not generate an Entity Tag

Resource Handlers

A Resource Handler is a query or an anonymous PL/SQL block responsible for handling a particular HTTP method. Mutiple Resource Handlers can be defined for a Resource Template, but only one handler per HTTP method is permitted. Additional Handlers are added to a Resource Template using the Add Handler button. Pressing this button displays the following dialog:



Choose the HTTP method you wish to implement from the selection box and press the 'Add' button.

Each Resource Handler is displayed as a separate tab pane identified by the HTTP method that it implements.

HTTP Methods

The HTTP specification defines a number of standard 'methods' which are used to operate on a resource, the four most commonly used operations are:

- `GET`: Retrieve a representation of a resource
- `POST`: Create a new resource or add a resource to a collection
- `PUT`: Update an existing resource
- `DELETE`: Delete an existing resource

Resource Handler Type

Three different strategies are supported for generating representations

- Query: Executes an SQL Query and transforms the result set into a JSON representation
- PL/SQL Block: Executes an anoymous PL/SQL block and transforms any OUT or INOUT parameters into a JSON representation.
- Media Resource: Executes a SQL Query conforming to a specific format and turns the result set into a binary representation with an accompanying HTTP `Content-Type` header identifying the internet media type of the representation.

See below for more detail on each of these strategies.

Resource Handler Parameters

Parameters declared in the URI Template are implicitly passed to the Resource Handler. For example if the URI Template is `people/{userid}` then a parameter named: `userid` is automatically made available to the Resource Handler.

Parameters to a Resource Handler can also be manually defined to bind HTTP headers to the Resource Handler, or to cast a URI Template parameter to a specific data type.

For example a Resource Handler might need to know the value of the HTTP `Accept-Language` header in order to localize the generated representation.

To define a parameter press the 'Add Parameter' button.

The following values can be specified:

- Name: The name of the parameter as passed to the Resource Handler, required.
- Aliasing: The original name of the parameter, if you wish to rename a parameter, optional
- Source: Either Header meaning a HTTP header or URI meaning a parameter for the URI template
- Access: One of `IN`, `INOUT`, or `OUT`. URI Template parameters can only be `IN`. A value of `IN` for a header parameter implies the Header will be present in the HTTP request. A value of `INOUT` indicates the value will be present in both the HTTP request and response and a value of `OUT` indicates the value will only be present in the HTTP response.
- Type: The primitive data type of the parameter, one of: String, Integer, Double, Boolean, or Long

A parameter can be deleted by pressing the 'x' icon to its immediate right.

Referencing Parameters

You can reference parameters in the Resource Handler definition by enclosing the parameter name in curly braces, for example:

```
select 'Hello ' || {person} || ' from APEX' greeting from dual
```

will cause the `{person}` token to be replaced with the actual value of the `person` parameter when the Resource Template is being evaluated. Parameter subsitution uses the standard Prepared Statement facilities of Oracle, so the normal rules apply to when and where parameters can be placed. For example the following example is not valid, because the table name is not allowed to be specified via a parameter:

```
select * from {table} where some_id = {id}
```

Acceptable Media Types

For Resource Handlers that accept a content body (`POST` and `PUT`) you can define what types of content the handler is able to accept by specifiying acceptable MIME types in the 'Acceptable' field. You can specify more than one content type by separating types with a comma, and you can also specify a wildcard character to accept a range of types (e.g. `image/*`).

Resource Handler Types

Query

This Resource Handler type will evaluate the supplied query and turn the resulting rows into a JSON representation. For example the following query:

```
select 'Hello ' || {person} || ' from APEX' greeting from dual
```

will generate a response similar to:

```
[
 {
  "greeting": "Hello World from APEX"
 }
]
```

- The `Content-Type` of the response will be: `application/json`
- The root of the JSON document will be a JSON array
- Each row in the result set will map to a JSON object
- Each column in a row will map to a JSON property
- The column name will always be in lowercase, the case of the value will be preserved

Since a query can only retrieve data and can never change the state of the database, the Query Resource Handler can only be used the the HTTP `GET` Method.

Media Resource

This Resource Handler type will evaluate a specially structured SQL query and return the resulting row as a binary representation. The query must follow this format:

`select` *`content_type`*`,` *`content_body`* `from ... where ...`

- The query must return one row and one row only
- The result set must consist of two columns:
  - the first (*`content_type`*) must be a string indicating the internet media type of the representation
  - the second (*`content_body`*) must contain the content of the representation. This column must be one of the following types: `VARCHAR`, `CLOB`, `BLOB`, `RAW`, `LONG RAW` or `XMLType`

Similar to the Query Resource Handler, the Media Resource Handler can only be used for the HTTP `GET` method, since it can only evaluate an SQL Query.

This Resource Handler type is very useful for generating XML and HTML representations, the following example generates a KML representation of customer locations by state, making it

possible to easily share customer location data with GIS applications:

```
select xmlquery('
<k:kml xmlns:k="http://www.opengis.net/kml/2.2">
<k:Folder>
 <k:name>{$state}</k:name>
{ for $c in ora:view("oe","customers")/ROW
  let $loc := $c/CUST_GEO_LOCATION/SDO_POINT
  where $c/CUST_ADDRESS/STATE_PROVINCE = $state
  return
  <k:Placemark>
   <k:name>{concat($c/CUST_FIRST_NAME," ",
$c/CUST_LAST_NAME)}</k:name>
   <k:Point>
     <k:coordinates>{$loc/X/text()},
{$loc/Y/text()},0</k:coordinates>
   </k:Point>
  </k:Placemark>
}
</k:Folder>
</k:kml>' passing {state} as "state" returning content) from dual
```

PL/SQL Block

This Resource Handler type will evaluate the supplied PL/SQL block and turn any outbound parameters into a JSON representation. This Resource Handler type is typically used to service HTTP `POST`,`PUT` and `DELETE` operations. This handler has a number of features aimed at increasing ease of use:

- When a HTTP request includes a content body two parameters named `{contentType}` and `{body}` are automatically passed to the handler. The former provides the `Content-Type` of the request and the latter provides the content body of the request as a BLOB.
- When the HTTP request includes a content body of with a `Content-Type` of: `application/x-www-form-urlencoded` the body is automatically parsed and each field in the form is converted to a parameter passed to the handler.
- The PL/SQL block can indicate the HTTP Status code via the `X-APEX-STATUS-CODE` response header
- The PL/SQL block can indicate the location of the updated resource via the `X-APEX-FORWARD` response header

Assume we have a Resource Template with the following definition:

URI Template	`gallery/`

HTTP Method	POST
Handler Type	PL/SQL Block

With the following manually defined parameters:

Name	Aliasing	Source	Access	Type
name	Slug	Header	IN	String
status	X-APEX-STATUS-CODE	Header	OUT	Integer
location	X-APEX-FORWARD	Header	OUT	String

and the following PL/SQL block:

```
declare
begin
 insert into scott.gallery values({name},{contentType},{body});
 {status} := 201;
 {location} := 'gallery/' || {name};
end;
```

The above handler inserts the supplied image into the scott.gallery table, returns the location of the stored image and indicates the operation created a new resource.

- The `{contentType}` and `{body}` parameters are automatically passed to the handler, no need to manually define them
- The `{name}` parameter must be manually defined to map it from the `Slug` Header
- The `{status}` parameter is mapped to the `X-APEX-STATUS-CODE` Header as an `Integer`. The value of `201` is specified by HTTP to indicate a new resource was created.
- The `{location}` parameter is mapped to the `X-APEX-FORWARD` Header. When the Listener sees this header in the response, it will abandon generating a JSON representation of the outbound parameters and instead attempt to return a representation of the indicated location (We are assuming in this example that there is a separately defined Resource Template that retrieves resources of the form: `gallery/{name}`)

Tutorial: Creating an Image Gallery

Introduction

Using Resource Templates we will build an Image Gallery with the following features:

- Display thumbnails of all images
- Upload new images

We will build both a web application suitable for use in modern web browsers and a RESTful API for non browser based clients.

Database Schema

We will need to create two tables and two sequences to store the Image Gallery data. Note it is assumed that the Listener is configured to connect to the database via the `apex_public_user` database user. Please adjust the scripts below if you are using a different database user.

Please execute the following scripts as database user `scott` before proceeding with this tutorial

```
create sequence gallery_seq;
/
grant select on gallery_seq to apex_public_user;
/
create table gallery (image_id number, title varchar2(1000),
content_type varchar2(1000), image blob);
/
grant all on gallery to apex_public_user;
```

A Note On URIs

Throughout this tutorial we will refer to URIs and URI Templates by an abbrieviated form ommitting the host name and context path. For example we will refer to a URI:

`gallery`

To access this URI in your browser use a URI of the form:

`http://host:port/apex/gallery`

where:

- `host` is the host the APEX Listener is running on
- `port` is the port the APEX Listener is running on
- `/apex/` is the Web Application context where the Listener is deployed

The API Entry Point

In RESTful designs it is good practice to have only one well known entry point. This minimizes the hard-coded footprint of the API, making it much easier to evolve the APIover time.

The entry point of our application will be a URI named:

`gallery`

This URI will provide a HTML representation that:

- enumerates all the images in the gallery. For each image we need to know:
  - the URI of the image
  - the title of the image
  - The URI of the comments on the image
- provides a link to the URI to post new images to
- Parsing HTML is hard so provide a link to an alternate JSON representation of the gallery that includes all the information above.

Uploading Images

This URI will also act as the endpoint to which clients can submit new images. The upload facility should be able to accept requests containing the actual image data and also requests containing multipart/form-data (as generated by HTML forms). Since the gallery will not be much use without some images, lets build this upload feature first.

Create a new Resource Template with the following data:

URI Template	`gallery`
HTTP Method	`POST`
Handler Type	`PL/SQL Block`

With the following manually defined parameters:

Name	Aliasing	Source	Access	Type
`title`	`Slug`	`Header`	`IN`	`String`
`status`	`X-APEX-STATUS-CODE`	`Header`	`OUT`	`Integer`
`location`	`X-APEX-FORWARD`	`Header`	`OUT`	`String`

and the following PL/SQL block:

```
declare
 image_id integer;
begin
 select scott.gallery_seq.nextval into image_id from dual;
 insert into scott.gallery values(image_id,{title},{contentType},
{body});
 {status} := 201;
```

```
 {location} := 'gallery/' || image_id;
end;
```

- The PL/SQL block will assign a unique id to the image (image_id) and insert the image data, image title and image content type into the scott.gallery table
- The block returns a 201 status which indicates a new resource has been created
- The block indicates the location of the created resource is of the form gallery/{imageid}

When you have created the Resource Template save it and proceed to the next step.

Upload Form

To invoke the above image upload handler we will need a HTML form to allow users to specifiy the image to upload, we'll give it the URI

```
upload
```

Create and save a new Resource Template with the following data:

URI Template	`upload`
HTTP Method	`GET`
Handler Type	`Media Resource`

and the following SQL Query:

```
select 'text/html',
'<html>
 <head>
  <title>Choose Image to Upload</title>
 </head>
 <body>
 <div>
  <h1>Choose Image to Upload</h1>
  <form action="gallery"
       enctype="multipart/form-data"
       method="post">
   <label for="title">Title</label>
   <input type="text" id="title" name="title"><br/>
   <label for="file">File</label>
   <input type="file" id="file" name="file"><br/>
   <div class="buttons">
    <button type="submit" class="button">Upload</button>
    <a href="gallery" class="button">Cancel</a>
   </div>
```

```
 </form>
</div>
</body>
</html>' from dual
```

- This illustrates a simple way to generate static text resources using the APEX Listener, the first column in the query is mapped to the content type of the resource and the second column becomes the body of the resource.
- Note this approach is not generally recommended, as SQL literal strings cannot be very big. It also involves a database round trip for data that is not actually retrieved from the database.
- Instead it is recommended that static resources are best served by your web/application server directly
- This form contains two fields the first indicates the image title, the second the image content

When you have created the Resource Template go ahead and try it out in your browser, enter a URL of the form:

```
http://host:port/apex/upload
```

Try choosing an image and giving it a title and uploading it. The browser should give a 404 Not Found error after you hit the Upload button, this is because the POST handler we configured above tells the browser to redirect to a gallery/{imageid} URI which we have not yet defined, however the image data will have been stored in the scott.gallery table.

Display the Image

Create a new Resource Template with the following data:

URI Template	`gallery/image/{imageid}`
HTTP Method	`GET`
Handler Type	`Media Resource`

With the following manually defined parameters:

Name	Aliasing	Source	Access	Type
`imageid`		`URI`	`IN`	`Integer`

and the following SQL Query:

```
select content_type, image from scott.gallery where image_id =
{imageid}
```

- This query retrieves the content type and image data for the specified image

Display the Gallery

Open the `gallery` Resource Template and add the data below for a GET Resource Handler so that we can display the data we have stored in the gallery.

HTTP Method	`GET`
Handler Type	`Media Resource`

and the following SQL Query:

```
select 'text/x-apex-html', xmlquery('
<html>
 <head>
 <link rel="alternate" type="application/json" href="gallery?
alt=json"/>
 </head>
 <body>
  <h1>Image Gallery</h1>
  <div class="buttons">
   <a href="upload" >Upload Image</a>
  </div>
  <div>
    <span itemscope="itemscope"><a itemprop="gallery"
href="./gallery"/></span>
   <ol id="images">
    {
    for $i in ora:view("scott","gallery")/ROW
    order by $i/IMAGE_ID descending
    return
     <li itemscope="itemscope">
      <a href="./gallery/{$i/IMAGE_ID/text()}">
       <img itemprop="image" src="./gallery/images/
{$i/IMAGE_ID/text()}"/>
       <div itemprop="title">{$i/TITLE/text()}</div>
      </a>
     </li>
    }
   </ol>
  </div>
  <br/>
 </body>
</html>
' returning content) from dual
```

- We use the Media Resource Handler to define a HTML resource, the HTML is generated dynamically using the `xmlquery()` SQL function.
- We specify the content type of the resource as `text/x-apex-html`. This is an APEX specific content type. When the Listener encounters this content type it expects the content

of the resource to be well formed XML It transforms this XML into HTML.

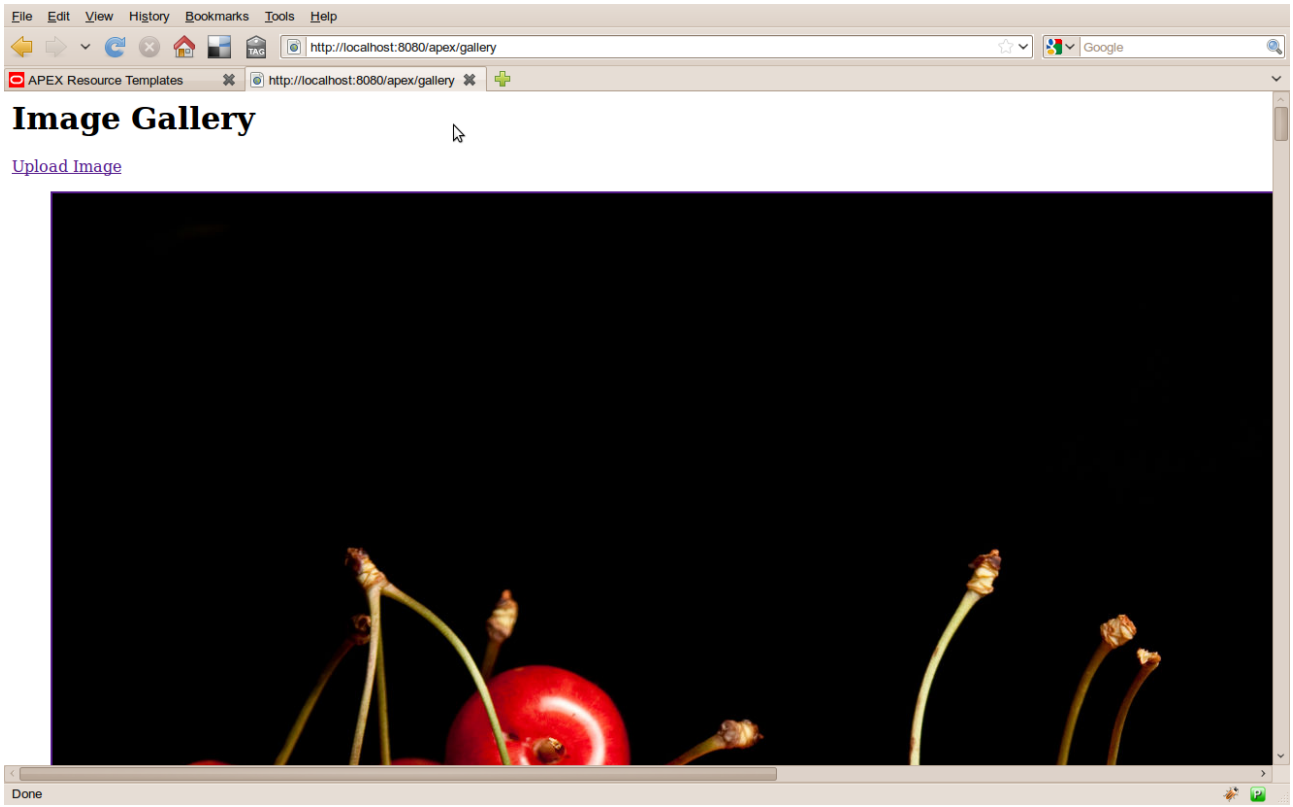
  - The `xmlquery()` function can only generate XML so we need a means to transform this XML into HTML automatically

- The generated HTML includes a link to an alternate JSON represntation of the gallery data
  - `<link rel="alternate" type="application/json" href="gallery?alt=json"/>`
  - This representation is automatically generated by the Listener using its capability to transform HTML marked up with Microdata into JSON
  - For each image in the gallery a `<li>` element is generated. The `<li>` element is marked up with Microdata to assist clients in extracting the gallery data from the HTML
    - Microdata is a feature of HTML5 enabling structured semantic data to be interwoven into HTML documents. This semantic data can be easily and reliably parsed by clients.

When you have updated the Resource Template go ahead and try it out in your browser, enter a URL of the form:

`http://host:port/apex/gallery`

You should see the image you previously upload, the browser will show something like:



Smarten up the Gallery's appearance

As it stands the gallery doesn't look very well, let's create some JavaScript and CSS resources to help improve its appearance and usability.

gallery.css

Create and save a new Resource Template with the following data:

URI Template	gallery.css
HTTP Method	GET
Handler Type	Media Resource

and the following SQL Query:

```
select 'text/css', '

html { font-size: 100%; }

body { font-size: 1em; font-family: Arial, Helvetica, sans-serif;
background : #e2e1e1; margin-left: 3em; margin-right: 3em;}

h1, h2, h3, h4, label { text-shadow: 0px 1px 0px #e5e5ee;}

h1 { text-align: center;}

a img { border:none; }


#images { margin: 1em auto; width: 100%; }

#images li { display: inline; }

#images a { background: #fff; display: inline; float: left;
margin: 0 0 27px 30px; width: auto; padding: 10px 10px 15px; text-
align: center; text-decoration: none; color: #333; font-size:
18px; -webkit-box-shadow: 0 3px 6px rgba(0,0,0,.25); -moz-box-
shadow: 0 3px 6px rgba(0,0,0,.25); -webkit-transform: rotate(-
2deg); -webkit-transition: -webkit-transform .15s linear; -moz-
transform: rotate(-2deg); }

#images img { display: block; width: 190px; margin-bottom: 12px; }

#images li:nth-child(even) a { -webkit-transform: rotate(2deg);
-moz-transform: rotate(2deg); }

#images li:nth-child(3n) a { -webkit-transform: none; position:
relative; top: -5px;  -moz-transform: none; }

#images li:nth-child(5n) a { -webkit-transform: rotate(5deg);
position: relative; right: 5px;  -moz-transform: rotate(5deg); }

#images li:nth-child(8n) a { position: relative; right: 5px; top:
8px; }

#images li:nth-child(11n) a { position: relative; left: -5px; top:
3px; }

#images li.messy a { margin-top: -375px; margin-left: 160px;
-webkit-transform: rotate(-5deg); -moz-transform: rotate(-5deg); }

#images li a:hover { -webkit-transform: scale(1.25); -moz-
transform: scale(1.25); -webkit-box-shadow: 0 3px 6px
rgba(0,0,0,.5); -moz-box-shadow: 0 3px 6px rgba(0,0,0,.5);
position: relative; z-index: 5; }
```

```
.buttons { float: right; margin-top: 1em; margin-bottom: 1em; }
label {font-weight: bold; text-align: right;float: left; width: 120px; margin-right: 0.625em; }
label :after {content(":")}
input, textarea { width: 250px; margin-bottom: 5px;text-align:left}
textarea {height: 150px;}
br { clear: left; }
' from dual
```

- This style sheets transforms our numbered list of images into a much easier to view series of thumbnails. It uses some modern CSS features such as transforms and shadows which will not work on older browsers.

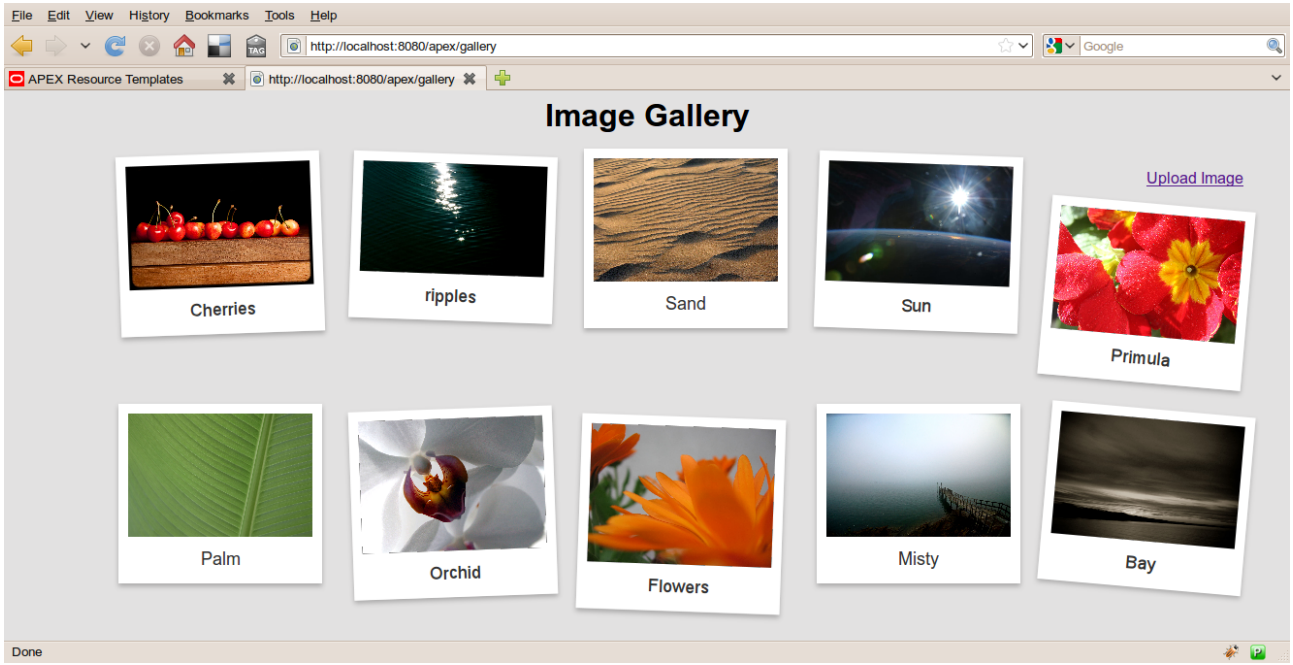
When you have created the Resource Template edit the `gallery` Resource Template and add the following in the `<head>` section:

```
<link rel="stylesheet" href="gallery.css" type="text/css"/>
```

and save your changes. Now view the gallery again:

```
http://host:port/apex/gallery
```

Now it should look something like this:



gallery.js

Create and save a new Resource Template with the following data:

URI Template	`gallery.js`
HTTP Method	`GET`
Handler Type	`Media Resource`

and the following SQL Query:

```
select 'application/javascript', '
$(function() {
 $(".buttons a").button();
 $(".buttons button").button();
});
' from dual
```

- This script uses the JQuery UI framework to transform the Upload Image link to a clickable button

The RESTful API

This tutorial has created both a web application and a RESTful API. We have seen the web application now it is time to explore the RESTful API.

The structured semantic data that clients need to interact with the API is encoded in the HTML as Microdata.

Clients have two choices for processing this Microdata, they can either parse it directly from the HTML (browser based applications will be able to use the HTML5 Microdata JavaScript APIs) or they can process the alternate JSON representation that the APEX Listener generates automatically.

The alternate representation is identified via the `<link>` element in the `<head>` section of the `gallery` resource. The link element has a `rel` attribute with the value `alternate` and a `type` attribute of `application/json`.

Below is a sample of the JSON representation:

```
{
 "items": [
  {
   "properties": {
    "gallery": "http://localhost:8080/apex/gallery"
   }
  },
  {
   "properties": {
    "image": "http://localhost:8080/apex/gallery/images/13",
    "title": "Cherries"
   }
  },
...
 {
   "properties": {
    "image": "http://localhost:8080/apex/gallery/images/4",
```

```
    "title": "Bay"
   }
  }
 ]
}
```

- Clients need only a small set of knowledge to sucessfully navigate the API
  - New Images can be uploaded by `POST`ing data to the URI indicated by the `gallery` property
  - The location of an image is indicated by the `image` property
  - The title of an image is indicated by the `title` property

The important thing to note is that the client knows only about a single URI (`gallery`) and discovers all other URIs from this initial URI. It doesn't know anything about the structure of the URIs, instead it just knows the names of properties that identify certain link types, this means that as the application evolves URIs are free to change and clients will not be affected.