REST and Express

Frank Walsh

(based on post by Stefan Tilkov)
http://www.infoq.com/articles/rest-introduction
And
http://www.ibm.com/developerworks/xml/library/wa-ajaxarch/

What is REST

- Short for **RE**presentational **S**tate **T**ransfer
- A software architecture style for distributed hypermedia systems(WWW)
- A set of principles that define how Web standards(HTTP and URIs) can be used.
    - One "incarnation" of the REST style is HTTP (and a set of related set of standards, such as URI).
- The way the Web's architecture "should" be used
- Coined by Roy Fielding in his PhD thesis
- The "right" way to implement heterogeneous application-to-application communication?...

REST Concept

- Resource Orientated
  - Resources are identified by uniform resource identifiers (URIs)
- Resources are manipulated through their representations
- Messages are self-descriptive and stateless
- Multiple representations are accepted or sent

Representation Concept

- What do you get when you request a web page?
  - A **representation** of a resource
- Resources are just “concepts”
  - i.e. list of Customers, Dept. of Computing Maths and Physics.
- A client can request a specific representation of a resource from the representations available on a server
  - http://www.wit.ie/SchoolOfScience/DeptofComputingMathsandPhysics/

State Transfer Concept

- State refers to an application/session state
- Clients initiate requests to servers; servers process requests and return appropriate responses
- A client can either be transitioning between application states or "at rest".
- The client begins sending requests when it is ready to transition to a new state.
  - (i.e. request new URI)
- While one or more requests are outstanding, the client is considered to be transitioning states.
- The representation of each application state contains links that may be used next time the client chooses to initiate a new state transition.

State Transfer Concept

- A Web-based application is a dynamically changing graph of
  - state representations (pages)
  - potential transitions (links) between states

- If it doesn’t work like that, it may be *accessible* from the Web, but it’s not really *part of the* Web

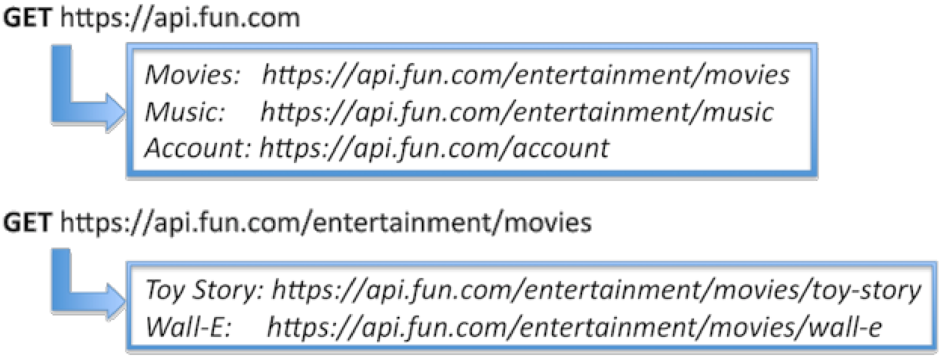
Rest Key Principles

1. Every “thing” has an identity
2. Link things together
3. Use standard set of methods
4. Resources can have multiple representations
5. Communicate statelessly

1-Identity

- Everything identifiable in an application should get a unique global ID
  - URIs
- URIs are consistent naming scheme for resources
- Universally recognised standard
- Example: companys assign unique product IDs. These can be URIs…

http://www.amazon.co.uk/gp/product/B002BWONF8/
http://example.com/customers/1234
http://example.com/orders/2007/10/776654
http://example.com/products/4554



GET https://api.fun.com

Movies: https://api.fun.com/entertainment/movies
Music: https://api.fun.com/entertainment/music
Account: https://api.fun.com/account

GET https://api.fun.com/entertainment/movies

Toy Story: https://api.fun.com/entertainment/movies/toy-story
Wall-E: https://api.fun.com/entertainment/movies/wall-e

2 – Linking Things

- Hypermedia as the engine of application state.
  - This means the links that make the Web Work
- Familiar with this from HTML but not restricted to this…

- Any application retrieving the above XML document can "follow" the links to retrieve more information.
- Links can be provided by a different application/server/company
  - naming scheme(URIs) are a global standard, all of the resources that make up the Web can be linked to each other.
- Furthermore links allow the client (the service consumer) to move the application from one state to the next by following a link.

3 – Standard Methods

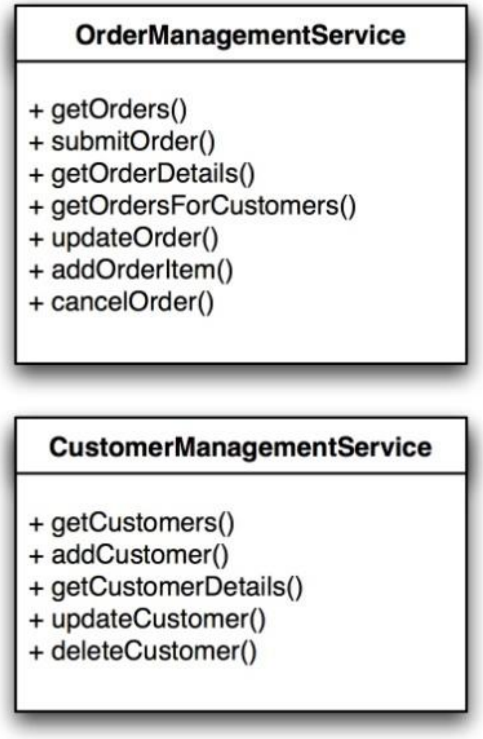
- how does your browser know what to do with the URI?
  - every resource supports the same interface, the same set of methods
  - HTTP ***verbs: GET, POST, PUT, DELETE, HEAD, OPTIONS***
  - From Object Orientated point of view, it's like each RESTful Class must extend a Resource object that contains the above methods
- Because Web resources use the same interface, you can be sure to get a representation of that resource by using the GET method.

3 – Standard Methods

- HEAD, GET, OPTIONS are defined as ”*safe*”
  - intended only for information retrieval
- POST, PUT and DELETE are intended for actions which may cause side effects either on the server
  - changing of persisted data
- HEAD, GET, OPTIONS, PUT and DELETE are defined as **Idempotent** methods
  - multiple identical requests should have the same effect as a single request
- Post is NOT defined as **Idempotent**
  - sending an identical POST request multiple times may further affect state(e.g. financial transactions, ticket purchase)
  - Ever see “only click once/wait for response/don’t click back” on a web application

3 – Standard Methods Example

- Order Management Class Models – standard design
- Client needs to be coded against these particular interfaces
- Cant use a client that was built before these interfaces were specified



OrderManagementService

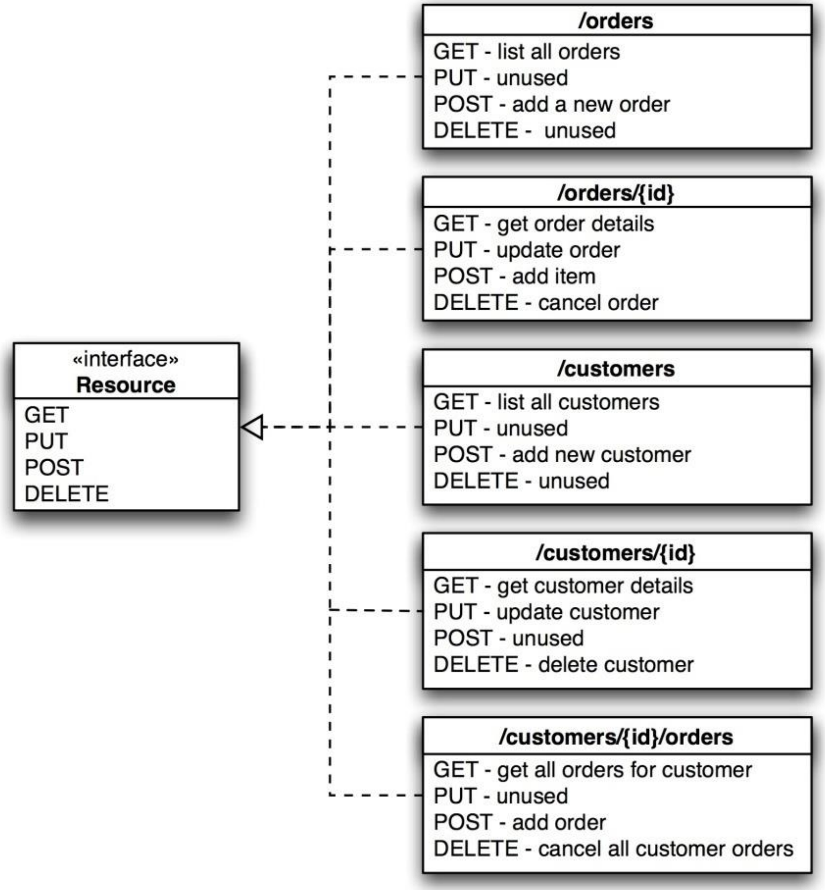
+ getOrders()
+ submitOrder()
+ getOrderDetails()
+ getOrdersForCustomers()
+ updateOrder()
+ addOrderItem()
+ cancelOrder()

CustomerManagementService

+ getCustomers()
+ addCustomer()
+ getCustomerDetails()
+ updateCustomer()
+ deleteCustomer()

RESTful HTTP Approach

- Define generic interface that makes up the *HTTP application protocol*.
- Specific operations of the services have been mapped to the standard HTTP methods.
- New set of resources created.



«interface»
Resource
GET
PUT
POST
DELETE

/orders
GET - list all orders
PUT - unused
POST - add a new order
DELETE - unused

/orders/{id}
GET - get order details
PUT - update order
POST - add item
DELETE - cancel order

/customers
GET - list all customers
PUT - unused
POST - add new customer
DELETE - unused

/customers/{id}
GET - get customer details
PUT - update customer
POST - unused
DELETE - delete customer

/customers/{id}/orders
GET - get all orders for customer
PUT - unused
POST - add order
DELETE - cancel all customer orders

Comparison to SOAP-based Services

- First approach has many operations and many kinds of data and a fixed number of Services
- RESTful approach has fixed number of operations, many kinds of data and many objects/Resources to invoke those fixed methods upon.
    - If there's 1 million orders in my database it means 1 million additional URIs on the web! So what?
- Opting for RESTful approach makes your app inherently part of the Web.
- Other approach usually involves one endpoint(URL) for each service, beyond which the methods can be accessed through some higher level protocol(e.g. SOAP)

4 - Multiple Representation

- How does a client know how to request and deal with the data it retrieves?
  - Can look at HTTP headers: *accept* and *content-type*
- HTTP allows separation of concerns between handling the data and invoking operations
  - Client can specify what data formats it can handle
  - a client can ask for a *representation* in a particular format.

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: application/json
```

5 - Stateless Communication

- REST mandates communication is Stateless
    - Does not mean that application cannot have state
- State must be:
    - A resource state
    - Kept on the client
- A server should not have to retain the communication state beyond a single request

5 – Stateless Communication

- Advantages of Stateless Comms:
  - Scalability. The server does not have to maintain state for each client
  - Isolation from changes on the server
    - not dependent on talking to the same server in two consecutive requests. Links from document returned by search engine will still work even if the search engine is shut down.

5 – What's wrong with State on Servers

- Remember, ideally software components are stateless.
  - Example: maintaining login credentials across a cluster of servers (an auto-scaled cluster in amazon).
  - If Restful, requests should not depend of the ones before
  - So what if your web server is shut down/drops HTTP connection, what happens to your laptop in your cart if your load balancer redirects next HTTP request to another server???
- Could use shared cache that all servers share.
  - Spread cache across n servers to stop imprisoned session data

Web API Design

API Design

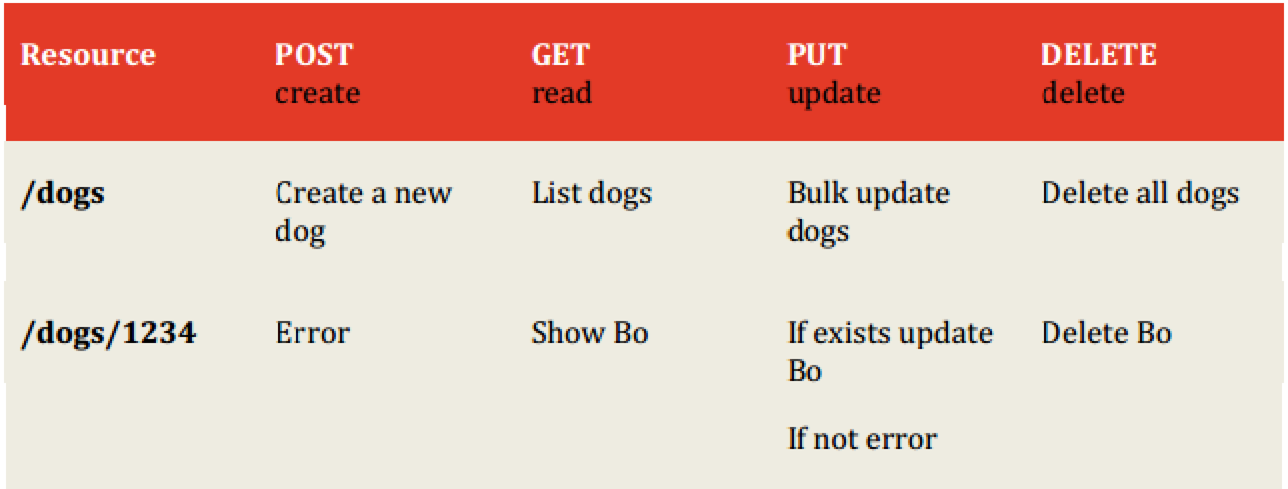
- APIs expose functionality of an application or service
- Designer must:
    - Understanding enough of the important details of the application for which an API is to be created,
    - Model the functionality in an API that addresses all use cases that come up in the real world, following the RESTful principles as closely as possible.

Nouns are good, verbs are bad

- Keep your base URL simple and intuitive
- 2 base URLs per resource
  - The first URL is for a collection; the second is for a specific element in the collection.
- Example
  - /contacts
  - /contacts/1234
- Keep verbs out of your URLS

Use the HTTP verbs

- We can use the HTTP verbs to manipulate the resources
- GET, PUT, POST, DELETE is equivalent to READ, UPDATE, CREATE, DELETE
- Rich set of intuitive capability



Resource	POST create	GET read	PUT update	DELETE delete
/dogs	Create a new dog	List dogs	Bulk update dogs	Delete all dogs
/dogs/1234	Error	Show Bo	If exists update Bo If not error	Delete Bo

Rest In Express

- Can easily implement REST APIS using express routing functionality
- Functionality usually implemented in api routing script

```
app.get('/dogs', dogs.listAllDogs)
app.post('/dogs', dogs.addADog)
app.put('/dogs/:id', dogs.updateDog)
app.delete('/dogs/:id', dogs.deleteDog)
```

Creating Route Modules (Style 1)

server.js

```
var express = require('express')
var dogs = require('./api/dogs/index');


…


app.get('/dogs', dogs.listAllDogs);
```

index.js

```
// GET the homepage
exports.listAllDogs = function(req, res){
       …);
};
```

Creating Route Modules (Style 2)

server.js

```
// Routes
require('./api/dogs/index')(app);
```

index.js

```
/*
 * Module dependencies
 */

module.exports = function(app){

    // GET home page
     app.get('/dogs', function(req, res){
         ...
     });
}
```

Express Request Object

- The **req** object represents the HTTP request.
  - □by convention, the object is always referred to as '**req',** Response is **'res'**
- Can use it to access the request query string, parameters, body, HTTP headers.
- Example:

```
app.get('/user/:id', function(req, res){
  res.send('user ' + req.params.id);
});
```

req.body

- Contains key-value pairs of data submitted in the request body.
- Need body-parsing middleware such as body-parser.

- This example shows how to use body-parsing middleware to populate req.body.

```
var app = require('express')();
var bodyParser = require('body-parser');
var multer = require('multer');

app.use(bodyParser.json()); // for parsing application/json
app.use(bodyParser.urlencoded({ extended: true })); // for
parsing application/x-www-form-urlencoded
app.use(multer()); // for parsing multipart/form-data

app.post('/', function (req, res) {
  console.log(req.body);
  res.json(req.body);
})
```

Response Object

- The res object represents the HTTP response that an Express app sends when it gets an HTTP request.

```
app.get('/user/:id', function(req,
res){ res.send('user ' +
req.params.id); });
```

Response Properties

- **res.json([body])**
  - Sends a JSON response. This method is identical to res.send() with an object or array as the parameter.

```
res.json({ user: 'tobi' })
res.status(500).json({ error: 'message' })
```

Response Properties

- **res.send([body])**
  - Sends the HTTP response.
  - The body parameter can be a String, an object, or an Array.
  - For example:

```
res.send({ some: 'json' });
res.send('<p>some html</p>');
res.status(404).send('Sorry, we cannot find that!');
res.status(500).send({ error: 'something blew up'
});
```

Response Properties

- **res.format(object)**
  - Performs content-negotiation on the Accept HTTP header on the request object

```
res.format({
  'text/plain': function(){
    res.send('hey');
  },

  'text/html': function(){
    res.send('<p>hey</p>');
  },

  'application/json': function(){
    res.send({ message: 'hey' });
  },

  'default': function() {
    // log the request and respond with 406
    res.status(406).send('Not Acceptable');
  }
});
```

Express Route Filters

```
//Catch-all
app.all('/app(/*)?',  function(req, res, next) {
  if(req.session && req.session.userName) {
    next();
  } else {
    res.redirect('/login?redir=' + req.url);
  }
});
```

Further Reference

- ExpressJS.com - Official Express Homepage
- Node and Express Tutorial