REpresentational State Transfer

Service-oriented Architectures

REST Concepts

REST Levels

REST + AJAX

These notes are combined from

1) NSF-funded CCLI RIT on SOP
2) H. Koehnemann, 321Gang
3) V. Paliath, InfusionSoft
4) Rajendran, Doran, Balasubramanian Student presentation

Principles of SOA

Stateless interactions

- Services are self-contained & do not store state between invocations
- State should be managed in the business process (e.g. choreography) and provided to individual services

2 Modern Flavors

1. Communicate through normalized message broker (ESB)
   - SOAs provide a independent protocol layer (normalized messages)
   - Adapters may be required to convert legacy applications to participate
   - Message-oriented, stateless communication; Focus is on the message, not the communication protocol
2. Use REST or similar services with JSON
   - Rely on established design patterns and best practices instead of heavy standards-based technology stacks
   - Ease the burden of deployment and troubleshooting on your ops staff
   - Scale-up/down (elasticity, cloud) on demand

Trends

The Recent Past

Application Containers

Components behind Services

Assembly and Composition

Ops: Virtualization

Values

- Flexibility (dev & ops)
- Scalability
- Decomposition

The Next Wave

YAGNI

Single-process, Single-server

CCC – Commoditization, the Cloud, & Containerization

Lean Configuration

Values

- Speed (DevOps)
- Scalability
- Decomposition – yes, but inversion of the process space (microservices)

We join this show already in progress…

REST

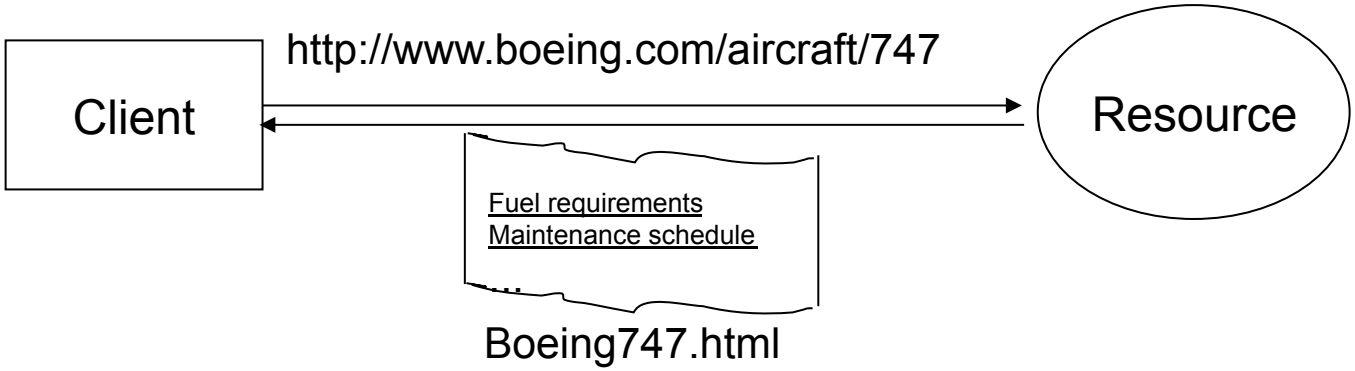
- Introduced and defined by Roy Fielding in his Ph.D. dissertation (UC Irvine 2000)
  - At the time he was working w/ others on HTTP 1.1 & URI standards
- REST defined a set of architectural constraints to make a system *anarchically stable*
  - Uh, what does that mean? Fielding 2002 (emphasis mine):

    "Most software systems are created with the implicit assumption that the entire system is under the control of one entity, or at least that *all entities participating within a system are acting towards a common goal and not at cross-purposes. Such an assumption cannot be safely made when the system runs openly on the Internet*. Anarchic scalability refers to the need for architectural elements to continue operating when subjected to an unanticipated load, or when given malformed or maliciously constructed data, since they may be communicating with elements outside their organizational control. The architecture must be amenable to mechanisms that enhance visibility and scalability.

Stable operation in the face of anarchy! Even better than graceful degradation!

REST (REpresentational State Transfer)

- The Client references a Web resource using a URL
- A resource representation is returned (an HTML document)
- Representation (e.g., Boeing747.html) puts client in new state
- When client selects hyperlink in Boeing747.html, it accesses another resource
- New representation places client into yet another state
- Client transfers state with each resource representation



REST Data Elements

Resource

- any object or concept that can be named
- the resource does not have to exist (yet)
- examples: books, animals, phones

Resource Identifier

- logical name and address for a resource
- implemented as a URI
- example: www.myexample.com/papers/rest.doc

Representation

- contains information about a resource's state
- in a specific format (Mime media types like text/html)
- examples: html, xml, mp4

Rajendran, Doran, Balasubramaniam, 2009

Resource Representation

- Each resource is represented as a distinct Uniform Resource Identifier (URI – you have seen these)
  - Uniform Resource Name (URN); e.g., isbn-10: 3642078885
  - Uniform Resource Locator (URL)

    e.g. http://www.imdb.com/title/tt0068646/?ref_=fn_al_tt_1
- Create a resource for every service
- Uniquely identify each resource with a logical URL
- Design your information to link to other information
  - That is, the information that a resource returns to a client should link to other information in a network of related information
- Most web interactions are done using HTTP and just four operations:
  - Retrieve information (HTTP GET)
  - Create information (HTTP PUT)
  - Update information (HTTP POST)
  - Delete information (HTTP DELETE)

An Example of RESTful Web Service

Service: Get a list of parts

- Web service makes an available URL to a parts list resource
- A client uses URL http://www.parts-depot.com/parts to get the parts list
  - How web service generates the parts list is completely transparent to the client
  - This is loose coupling
- Each resource is identified as a URL
  - Parts list has links to get each part's detailed info
- Key feature of REST design pattern
  - Client transfers from one state to next by examining and choosing from alternative URLs in the response document

```
<?xml version="1.0"?>
<Parts>
    <Part id="00345" href="http://www.parts-depot.com/parts/00345"/>
    <Part id="00346" href="http://www.parts-depot.com/parts/00346"/>
    <Part id="00347" href="http://www.parts-depot.com/parts/00347"/>
    <Part id="00348" href="http://www.parts-depot.com/parts/00348"/>
</Parts>
```

Response Formats of RESTful Web Services

XML or JSON

- JSON is more popular now, as the primary use case for RESTful services is consumption in a browser application
- http://www.omdbapi.com/?s=titanic[&r=xml]

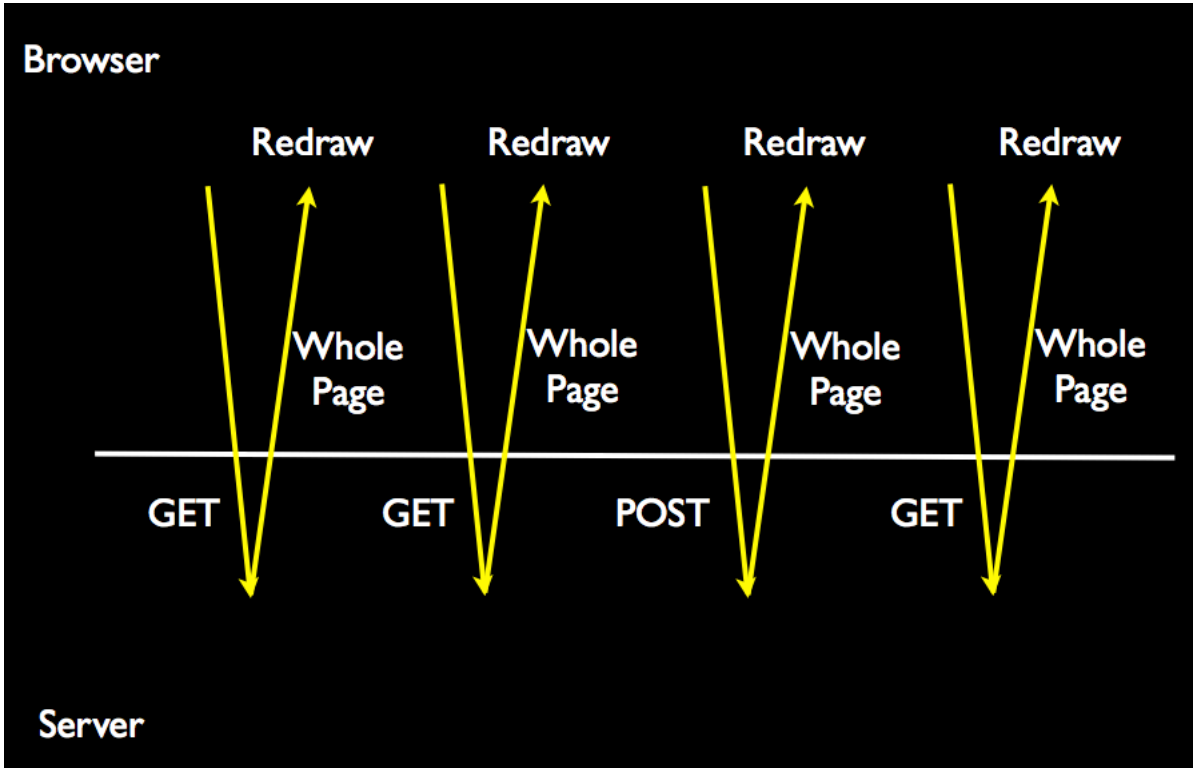
```
<root response="True">
<Movie Title="Titanic" Year="1997" imdbID="tt0120338" Type="movie"/>
<Movie Title="Titanic II" Year="2010" imdbID="tt1640571" Type="movie"/>
<Movie Title="Titanic: The Legend Goes On..." Year="2000" imdbID="tt0330994" Type="movie"/>
<Movie Title="Titanic" Year="1953" imdbID="tt0046435" Type="movie"/>
<Movie Title="Titanic" Year="1996" imdbID="tt0115392" Type="movie"/>
<Movie Title="Raise the Titanic" Year="1980" imdbID="tt0081400" Type="movie"/>
<Movie Title="Titanic" Year="2012" imdbID="tt1869152" Type="series"/>
<Movie Title="The Chambermaid on the Titanic" Year="1997" imdbID="tt0129923" Type="movie"/>
<Movie Title="Titanic: Blood and Steel" Year="2012" imdbID="tt1695366" Type="series"/>
<Movie Title="Titanic" Year="1943" imdbID="tt0036443" Type="movie"/>
</root>
```

```
{"Search":[{"Title":"Titanic","Year":"1997","imdbID":"tt0120338","Type":"movie"},
{"Title":"Titanic II","Year":"2010","imdbID":"tt1640571","Type":"movie"},{"Title":"Titanic:
The Legend Goes On...","Year":"2000","imdbID":"tt0330994","Type":"movie"},
{"Title":"Titanic","Year":"1953","imdbID":"tt0046435","Type":"movie"},
{"Title":"Titanic","Year":"1996","imdbID":"tt0115392","Type":"movie"},{"Title":"Raise the
Titanic","Year":"1980","imdbID":"tt0081400","Type":"movie"},
{"Title":"Titanic","Year":"2012","imdbID":"tt1869152","Type":"series"},{"Title":"The
Chambermaid on the Titanic","Year":"1997","imdbID":"tt0129923","Type":"movie"},
{"Title":"Titanic: Blood and Steel","Year":"2012","imdbID":"tt1695366","Type":"series"},
{"Title":"Titanic","Year":"1943","imdbID":"tt0036443","Type":"movie"}]}
```

So what does REST have to do with web apps?

Déjà vu: Consider our server-side web app pattern:

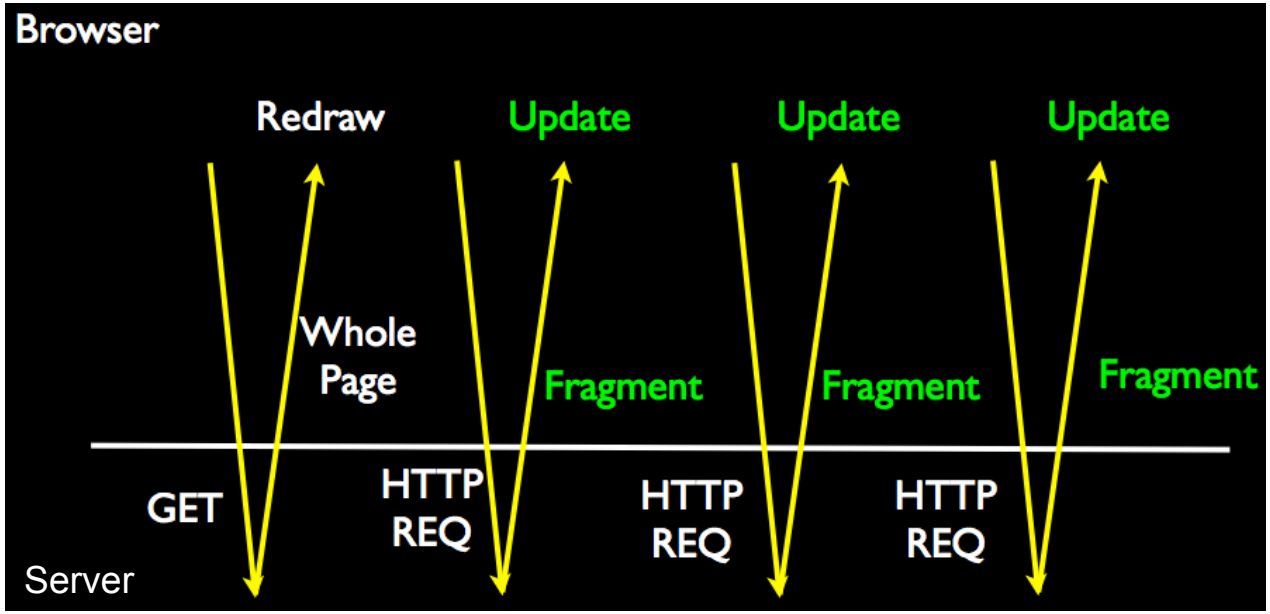
- A user takes some action like a click on a link or button
- The browser makes a TCP/IP connection to the web server
- The browser sends a POST or GET request
- The server sends back a rendered page to display to the user
- Repeat the Request-Response Cycle...



Déjà vu: XMLHttpRequest and AJAX

By 1999, Microsoft wanted to move some of the processing of web pages from the web server to the web browser

- The idea was instead of sending whole pages of HTML to the browser, send out the data to be displayed as XML and then produce presentation in JavaScript in the browser
- Originally a Microsoft innovation - other browsers soon adopted the idea and it became a defacto standard with a little variation between browsers
- It soon became clear that this could send *anything* - not just XML back and forth between a browser and client



REST and AJAX Summary

AJAX:

- Each action sends data and receives results in the background.
- The browser typically gets back a fragment of HTML, XML, or JSON which is used to update a portion of the screen using the browser document model

REST + AJAX:

- REST is useful w/out AJAX, but together they are “like peas & carrots”
- REST, AJAX, and HTML5 mean this is the app pattern moving forward
- Server-side developers need to think RESTfully or “service-ly”
  - Avoid making the API stateful
  - Create APIs that are HTTP-centric but not client-device centric
    - Make that the CSS problem (Responsive Web Design)

HTTP:

- To work properly as a RESTful client, we have to correctly use HTTP
  - HTTP Verbs and HTTP response codes

HTTP Verbs

GET: List a collection or retrieve a resource representation

POST: Create a new resource in a collection of resources

PUT: Replace a resource in a collection, or create a new one

DELETE: Delete a resource or resource collection

PATCH: Modify ("partial update of") an existing resource

HEAD and **OPTIONS** are not normally used in REST

- → **PATCH** is used sparingly for specific situations
- → **PUT** and **DELETE** are *idempotent* - Return the same thing over & over
- → **PUT** and **POST** can be difficult to separate. Use PUT when you know the endpoint (INSERT_or_UPDATE operation).
  - → Use **POST** when the resource URL may be created
  - → restcookbook.com/HTTP%20Methods/put-vs-post/

Use these verbs! If you are creating URIs that embed some sort of action in the URI or a query parameter, then you are not doing it correctly!

HTTP Response Codes

…and when to use

200: Everything “OK”. Typically on GET/HEAD

201: Resource created (PUT or POST). Location header should be set to URI of new resource. Body may be empty

204: No content. May be used on successful DELETE

400,404: Improper client request / bad URI

401,403: Unauthorized/Forbidden – The former means no or incorrect authentication information was provided, latter means client is forbidden no matter the authentication

409: Conflict. Completing the request would leave the server resource in an unstable state. Most often on PUT/PATCH

5xx: As before, server-side errors; not really specific to REST

See http://www.restapitutorial.com/httpstatuscodes.html

Summary

Is that it? Where is the *stuff* Dr. Gary?

- You already have the programming concepts to consume REST services and APIs – AJAX, HTTP, JSON, XML, etc.
- REST is the current in-vogue flavor of building API-driven apps
- While technologies you know, the key is acting like a good RESTful client
  - Knowing what verb to send in an HTTP
  - Knowing how to handle to various response codes

Isn't there more to it? There is a lot of hullabaloo about REST

- Well OK yes there is
- REST has a number of extended API principles supporting its conceptual formulation as a web of resources that get acted on by "verbs" and project "representations"
- As a server-side dev you worry about the extended nature of representations, HATEOS, navigability, versionability…