

RESTFUL WEB SERVICES

HappyJack Tech Talk
April 16, 2008

OVERVIEW

- Distributed Programming
- Web Services
- HTTP Protocol
- RESTful Architecture
- Designing a REST application

DISTRIBUTED PROGRAMS

- Distributed programming is one of computing's Holy Grails
- The question is:
 - How do we split a program into pieces, each of which runs in a different computer
- This problem is still not solved!

PAINLESS DISTRIBUTION

- More importantly, how do we develop distributed programs without complexity?
- Ideally, we should be able to write a “typical” program, yet still be able to distribute it across several machines

RPC

- One way to Nirvana is to use Remote Procedure Calls (RPC)
- Simply write the program using structured programming techniques
- Then designate some procedures to run on other machines

RPC (CONT'D)

- The RPC mechanism is in charge of
 - converting arguments to a “network” representation
 - sending arguments over the network
 - invoking the called procedure
 - converting / sending result value

RPC (CONT'D)

- The programmer simply writes the procedure **as if it was a local procedure**
- Calls to the remote procedure look **just like local procedure calls**

RPC AND OBJECTS

- RPC worked reasonably well
 - though the details are a little trickier than I've led you to believe...
- Many systems in use today are based on RPC
- But RPC is defined in terms of procedures, not objects

OBJECT-ORIENTED RPC

- The OO equivalent to RPC is given by remote objects
- You can place an object on a remote machine
- Methods to this object look like local methods, but are actually sent (transparently) to the remote object

OO RPC TECHNOLOGIES

- There have been lots of remote method technologies
 - Microsoft's DCOM
 - OMG's CORBA
 - J2EE
 - .Net Remote Objects

OO RPC TECHNOLOGIES

- None of these technologies has been wildly successful
- The technologies are still too complex!
- Also, designing distributed objects is inherently different than designing traditional OOP systems

WEB SERVICES

- Web Services can be viewed as another technology for distributed objects
- The technology is based on the web:
 - HTTP protocol for sending / receiving data
 - XML (usually) for representing data

RPC WEB SERVICES

- Some of the same parties that previously proposed hideously complex standards for remote object invocation are busy at work defining Web Services
- We'll talk about this in the next tech talk

REST WEB SERVICES

- REST is a way of describing the architecture of the internet
- It can also be used as the model for a distributed programming paradigm
- This leads to a (lowercase) web services solution
- Think of it as the programmer's solution to web-based distributed programming

UNDERSTANDING HTTP

- REST is a **web technology**
- It builds on the standard HTTP protocol to transfer messages
- To design REST applications, it's important that we understand HTTP
 - what it is and how it's meant to be used....

HTTP AND THE WEB

- HTTP is a familiar protocol
- We'll use our intuition from the web to describe the protocol
- But, this can be misleading...
 - There are some key differences between HTTP (the protocol) and HTTP as it's used today in the (human) web....

HTTP OVERVIEW

- The HTTP protocol is deliberately simple
 - naive, even
- A client makes a connection to an HTTP server and sends an **HTTP Request**
- The server replies with an **HTTP Response**
- The connection is terminated

HTTP REQUESTS

- An HTTP request is made up of three components:
 - Destination – method and URI
 - Header(s) – key / value pairs, some with standard semantics, e.g., If-Modified-Since
 - Body – an optional chunk of information

HTTP REQUEST

```
GET /index.html HTTP/1.1  
User-Agent: curl/7.16.3 ...  
Host: www.cs.uwyo.edu  
Accept: */*
```

HTTP RESPONSE

- An HTTP Response is also made up of three components
 - Status – numeric and text description of the result, e.g., 200 (OK), or 404 (Not Found)
 - Header(s)
 - Body

HTTP/1.1 200 OK

Date: Wed, 16 Apr 2008 11:11:26 GMT

Server: Apache/2.2.3 (Red Hat)

X-Powered-By: PHP/5.1.6

Content-Length: 6446

Connection: close

Content-Type: text/html; charset=ISO-8859-1

<html>

...

</html

REST PHILOSOPHY

- REST was originally designed to describe the architecture of the web
 - I.e., most read-only websites are RESTful
- To build a RESTful web service, you have to embrace HTTP

EMBRACING HTTP

- Server data is organized into “resources”
- Each resource is accessed by one or more URIs
- All client requests use HTTP request methods
 - GET, HEAD, PUT, DELETE, POST

EMBRACING HTTP

- The URI specifies a resource (not an operation or extra data)
- Extra information is placed in the request body
- HTTP response codes are used appropriately
- Response body is program-friendly, e.g., XML

EMBRACING HTTP

- REST clients need only
 - an HTTP library
 - an XML parser
- Both of these are widely available
- REST servers can be written in any web platform, e.g., Ruby on Rails

WHAT IS A RESOURCE?

- Predefined, singleton object
 - all emails received
- Other objects exposed through the service
 - email with id #35
 - the most recent email received

WHAT IS A RESOURCE?

- The **result** of an algorithm
 - emails from Aunt Betty

RESOURCE != OBJECT

- Many OOP programmers create URIs that map to objects
- Each object has state, so the URI encodes some type of method invocation
- This is the wrong idea
- Resources only respond to the predefined HTTP methods!

RESOURCE != OBJECT

- So what do you do if you want your resource to respond to a new method?
- The answer is to think of the method as a new resource, associating the arguments
- For example, instead of adding a **subscribe** method to your blog, you can expose the resource of **subscriptions**

URIS

- The URI should uniquely identify a resource
- That's it!
- I.e., it should not have method information, such as `/inbox/35?action=delete`
- It is OK to have format information in the URI, e.g., `/inbox/35?format=HTML`

URIS

All emails received	<code>/inbox</code>
Message with ID #35	<code>/inbox/35</code> <code>/inbox?id=35</code>
Most recent email	<code>/inbox/latest</code>
Emails from Aunt Betty	<code>/inbox/search?from=betty</code>

HTTP METHODS: GET

- GET retrieves a representation of the object
- The representation is usually encoded as XML or an image (e.g., JPEG)
- But it can be anything else that is useful to a **program**

HTTP METHODS: GET

- GET requests should be
 - **Safe** – request no side effects from the server
 - **Idempotent** – sending two GET requests should generate the same result

HTTP METHODS: HEAD

- HEAD requests are just like GET requests, except they do not return the result body
- This is useful if all you need are the headers
- E.g., to check if a resource exists

HTTP METHODS: PUT

- PUT requests **create** or **update** a resource
- PUT is not used in the human web
- The URI specifies the request to be created or modified
 - not the actual change!
- The change is represented in the request body

HTTP METHODS: PUT

```
PUT /inbox/32 HTTP/1.1
```

```
Host: api.mail.org
```

```
Accept: */*
```

```
spam=high&priority=low
```

HTTP METHODS: PUT

- PUT requests are **idempotent** but **not safe**
- I.e., two successive PUT requests are the same as a single PUT requests
 - the first one creates and / or modifies
 - the second one performs the **same** modification

HTTP METHODS: DELETE

- DELETE method removes a resource
- This method is also not used in the human web
- DELETE is **idempotent** but **not safe**

HTTP REQUESTS: POST

- POST is often abused in the human web and in pseudo-RESTful web services
- Part of the reason is the HTTP standard describes several different roles for POST

HTTP REQUESTS: POST

- POST can be used to
 - annotate an existing resource
 - post a message to a bulletin board
 - provide a block of data to data-handling process
 - append to a database

HTTP REQUESTS: POST

- In the human web, POST is usually used to send data to a “data-handling process”
 - i.e., through a <form>
- In the RESTful world, we use POST mostly to append to a database

HTTP REQUEST: POST

- For example, suppose a resource represents a weblog entry
- PUTting to the resource alters the entry, e.g., changes the text in the entry
- POSTing to the resource can add a comment to the entry

HTTP REQUEST: POST

- If you add a comment to a weblog entry, is the comment a new resource?
- Maybe – if you want to allow users to modify comments with PUT (or moderators to DELETE them)
- So POST can be used to create resources

POST VS. PUT

- Do you use POST or PUT to create resources?
- That depends on whether the client or the server should create the URI
- If the client creates the URI, then a PUT to a new URI should be used
- If the server creates it, then use a POST to the resource that manages the entries

POST VS. PUT

- This is a subtle distinction, but it becomes clearer with practice
- E.g., Amazon's S3 service uses a PUT to the resource `/bucket/file` to create a file
 - The contents of the file are in the body
- But a POST to `/weblog/entry` makes more sense for adding a comment to the weblog entry

POST VS. PUT

- Some frameworks will make the choice for you
- Ruby on Rails only supports resources that can be thought of as belonging to a master list
 - E.g., records in a table
- The table is a resource, as are the rows
- To add a row, POST to the table resource

HTTP REQUESTS: POST

- Finally, notice that POST requests are **not safe** and **not idempotent**
- That is why we prefer to use PUT to create resources, when possible

HTTP RESPONSE CODES

- Response codes are critical in the RESTful web
 - 200 – OK
 - 301 – Moved Permanently
 - 400 – Bad Request

HTTP RESPONSE CODES

- 404 – Not Found
- 409 – Conflict
- 410 – Gone
- 500 – Internal Server Error

WHO USES REST?

- The programmable web is moving towards REST
- Amazon provides two APIs to interact with their store
 - the REST API is much more popular than their SOAP API
- Other Amazon web services (e.g., S3) use REST

WHO USES REST?

- Google is a strong REST supporter
 - and they recently “deprecated” their SOAP API
- Del.icio.us has a RESTful API
- And many more....
 - ...but, most REST APIs are pseudo-RESTful

CONCLUSION

- REST is an architecture for writing web services
- The overall philosophy is to embrace HTTP
- It's a different way of organizing the server, but it leads to easy of use for the client
- The world today is a mixture of REST and RPC over REST – and moving to pure REST