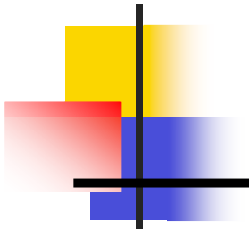
Put Your Feet Up and Take a RESTful Approach

Denise Hatzidakis
Perficient, Inc
Director, Chief Technologist
denise.hatzidakis@perficient.com

Colorado Software Summit

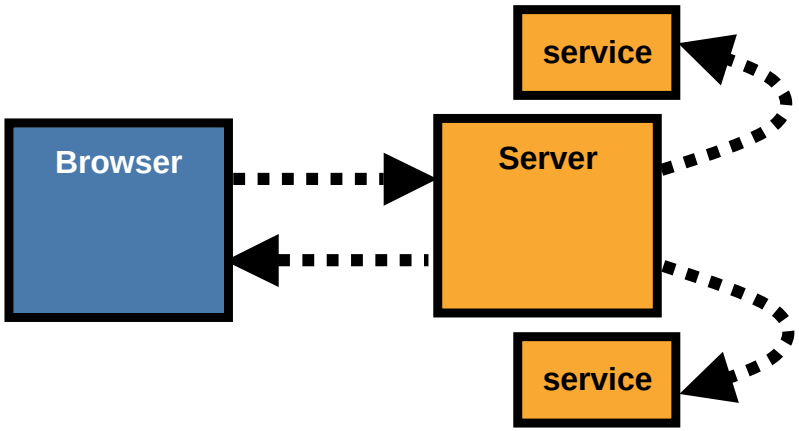


Overview

- The Web and Web 2.0
- Where did REST come from
- What is REST
- Where does REST fit
- RESTful HTTP
- REST Example
- Advantages of REST
- Why use REST
- RESTful SOA

Web Applications and SOA

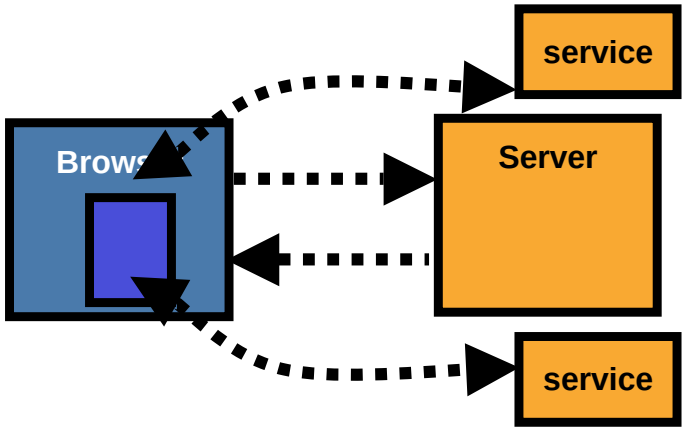
Classic web application patterns promoted server side service access



Existing web services standards (WS_*, WSDL, SOAP) are typically used in this model

Focus is on access from multiple programming languages using a variety of communication protocols

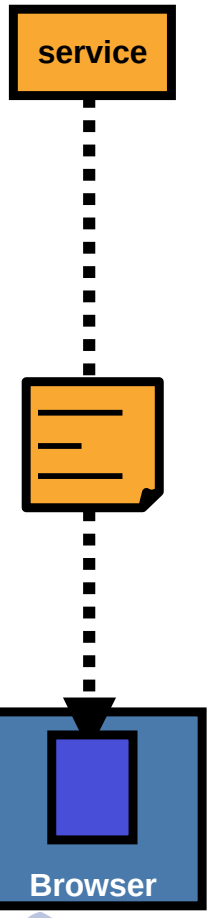
Web 2.0 application patterns promote client side (browser) access to services



Emerging web services approaches leveraging REST and JSON are typically used in this model

Optimized for *single language* (JavaScript), *single protocol* (HTTP) access via Ajax in a browser

Web 2.0 style services – key concepts



REST

- **RE**presentational **S**tate **T**ransfer
- Server side architectural style relying on HTTP semantics to access services or resources
- Easily invoked by browsers via Ajax

JSON

- **J**ava**S**cript **O**bject **N**otation
- Data format used to exchange information between browser and a service
- Directly consumable by JavaScript clients

Ajax

- **A**synchronous **J**avaScript **A**nd **X**ML
- Browser based technology to provide highly interactive and responsive web pages
- Enables the browser to invoke services directly from the client

The scope of the problem…

Web's major goal was to be a shared information space through which people and machines could communicate… Consider this…

Computer A in Colorado …

… tells computer B in Outer Mongolia …

… about a resource available on Computer C in Timbuktu

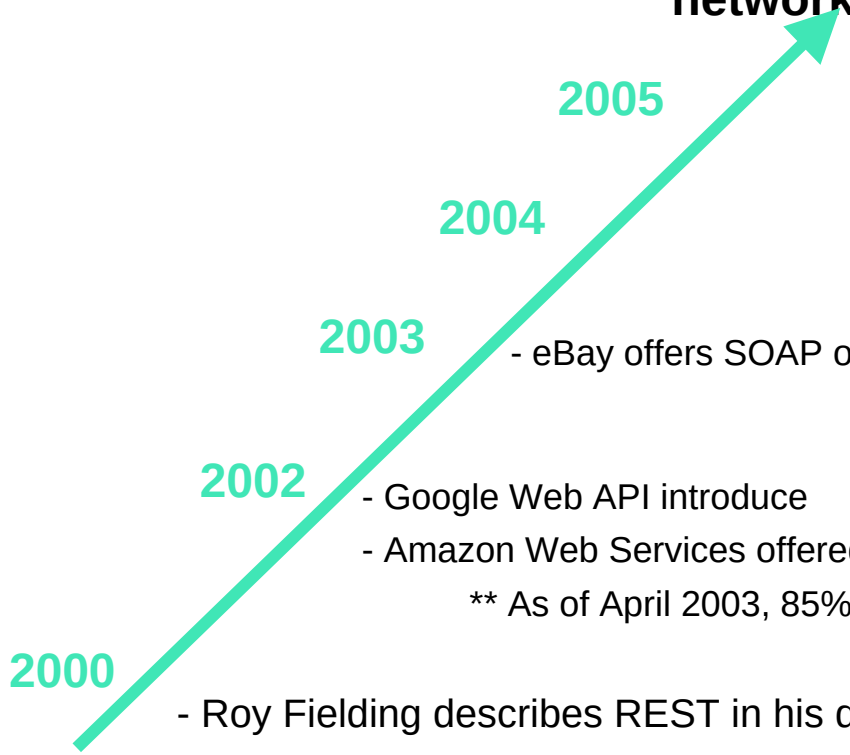
But… None of them belongs to the same trust domain

REST

(REpresentational State Transfer)

Where did all this start?

" REST " was coined by Roy Fielding in his Ph.D. dissertation [1] to describe a design pattern for implementing networked systems.



2005

- Google Maps
- Jesse James Garrett coins the term AJAX**
- Tim O'Reilly coins the term Web 2.0

2004

2003

- eBay offers SOAP or REST Web Service

2002

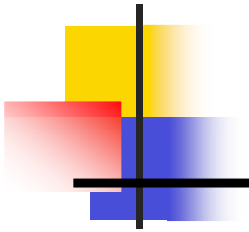
- Google Web API introduce
- Amazon Web Services offered via SOAP or REST

** As of April 2003, 85% of the traffic was REST**

2000

- Roy Fielding describes REST in his doctoral dissertation

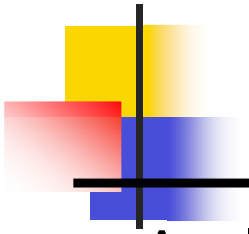
[1] http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm



What is REST ?

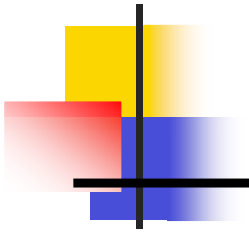
REST is the acronym for
Representational State Transfer

- It is the architectural model on which the World Wide Web is based
- The term is often used in a looser sense to describe any simple interface which transmits domain-specific data over HTTP
  - without an additional messaging layer such as SOAP or session tracking via HTTP cookies



REST Principals

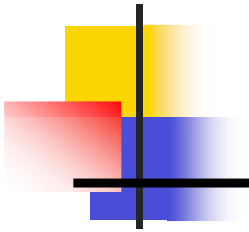
- Application state and functionality are abstracted into resources
- Every resource is uniquely addressable using a universal syntax for use in hypermedia links
- All resources share a uniform interface for the transfer of state between client and resource, consisting of
  - A constrained set of well-defined operations
  - A constrained set of content types, optionally supporting code on demand
- A protocol which is:
  - Client-server
  - Stateless
  - Cacheable
  - Layered



What is/is not REST?

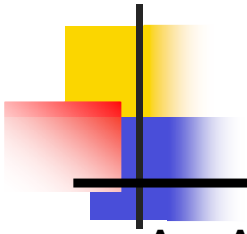
- It is possible to
  - design any large software system in accordance with Fielding's REST architectural style
    - without using HTTP and without interacting with the World Wide Web

- It is also possible to
  - design simple XML+HTTP interfaces which do not conform to REST principles,
    - and instead follow a model of remote procedure call



What is REST ?

- Principles of REST
  - Resource centric approach
    - An important concept in REST is the existence of resources (sources of specific information)
  - All relevant resources are addressable *via* a global identifier
    - *e.g.*, a URI in HTTP
  - Uniform access via standardized interfaces
    - *e.g.*, HTTP actions – GET, POST, PUT, DELETE
  - Content type negotiation allows retrieving alternative representations from same URI.

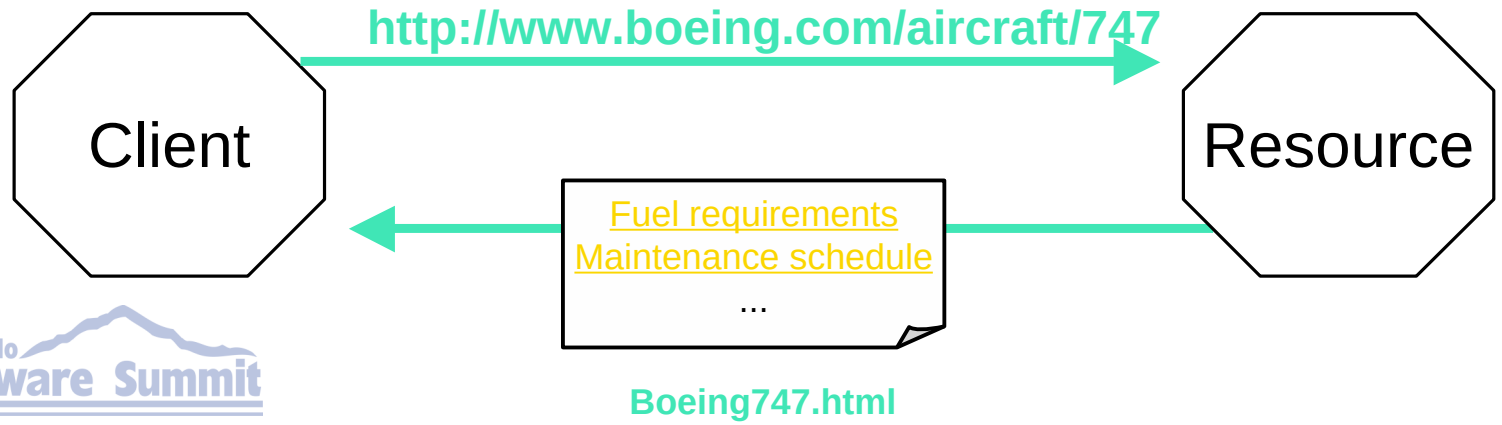


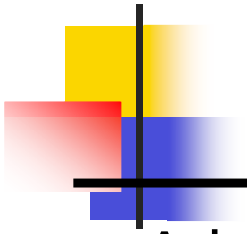
What is REST ?

- An Application can interact with a resource by knowing two things:
  - the identifier of the resource, and
  - the action required
- Each message contains all the information necessary to understand the request when combined with state at the resource
- The application does not need to
  - Know whether there are caches, proxies, gateways, firewalls, tunnels, or
  - anything else between it and the server actually holding the information
- The application does need to
  - Understand the format (representation) of the information
    - typically an HTML, XML or JSON document of some kind, although it may be an image, plain text, or any other content.
- **REST style services**
  - are easy to access from code running in web browsers, any other client or servers
  - can serve multiple representations of the same resource

Why is it called Representational State Transfer?

- The Client references a Web resource using a URL.
- A **representation** of the resource is returned (in this case as an HTML document).
- The representation (*e.g.*, Boeing747.html) places the client in a new **state**.
- When the client selects a hyperlink in Boeing747.html, it accesses another resource.
- The new representation places the client application into yet another state.
- Thus, the client application **transfers** state with each resource representation



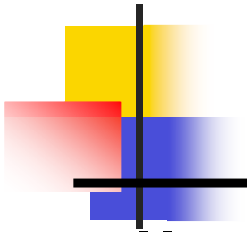


What is REST?

- A high-level style that could be
  - implemented using many different technologies, and
  - instantiated using different values for its abstract properties
- For Example
  - REST DOES include the concepts of resources and uniform interfaces
  - REST DOES NOT say which interfaces (methods) they should be
- However
  - One “incarnation” of the REST style is HTTP
  - The Web, HTTP and URIs are the only major, certainly the only relevant instance of the REST style as a whole

REST – Not a Standard

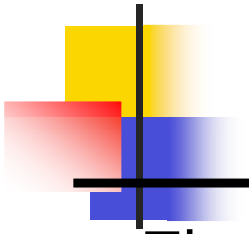
- REST is an Architectural Style, Not a Standard.
  - You will not see the W3C putting out a REST specification.
  - You will not see IBM or Microsoft or Sun selling a REST developer's toolkit.
- REST is just a design pattern
  - You can't bottle up a pattern.
  - You can only understand it and design your Web services to it.
- REST does prescribe the use of standards:
  - HTTP
  - URL
  - XML/HTML/GIF/JPEG/etc. (Resource Representations)
  - text/xml, text/html, image/gif, image/jpeg, *etc*. (Resource Types, MIME Types)



REST Key Concepts

All REST interactions are stateless

- It removes any need for the connectors to retain application state between requests,
  - thus reducing consumption of physical resources and improving scalability;
- It allows interactions to be processed in parallel without requiring that the processing mechanism understand the interaction semantics
- It allows an intermediary to view and understand a request in isolation, which may be necessary when services are dynamically rearranged
- It forces all of the information that might factor into the reusability of a cached response to be present in each request

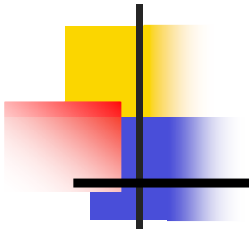


REST Key Concepts

- The key abstraction of information in REST is a resource.
  - The definition of resource in REST is based on a simple premise: identifiers should change as infrequently as possible.
- REST components perform actions on a resource by using a representation to capture the current or intended state of that resource and transferring that representation between components
- REST uses various connector types, to encapsulate the activities of accessing resources and transferring resource representations

Key REST Principals

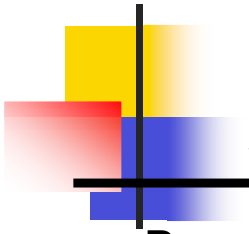
- Give every "thing" an ID
  - Everything identifiable gets an ID
  - Use URIs to identify everything that merits being identifiable, specifically, all of the "high-level" resources that your application provides,
    - whether they represent individual items, collections of items, virtual and physical objects, or computation results
- Link things together
  - Hyperlinking – Use links to refer to identifiable things (resources) wherever possible.
- Use standard methods
  - For clients to be able to interact with your resources, they should implement the default application protocol (HTTP) correctly, *i.e.* make use of the standard methods GET, PUT, POST, DELETE.
- Resources with multiple representations
  - Provide multiple representations of resources for different needs.
- Communicate statelessly
  - The statelessness constraint isolates the client against changes on the server as it is not dependent on talking to the same server in two consecutive requests



Why REST?

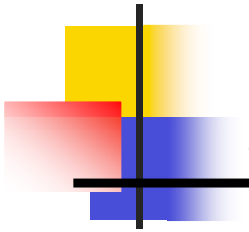
WS-* is complex in certain scenarios

- Great for
  - Machine to machine solutions
  - Process engines and BPM solutions
- Not so great for
  - Human to Computer
  - Developer to Computer
  - Browser to server
- For rich internet applications we need something far simpler and easier to use and have a wider impact on every developer skill level.



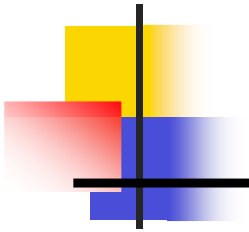
Advantages of REST

- Provides improved response time and reduced server load due to its
  - support for the **caching** of representations
- Improves server scalability by reducing the need to maintain session state.
  - This means that different servers can be used to handle different requests in a session
- Requires less client-side software to be written than other approaches,
  - because a single browser can access any application and any resource
- Depends less on vendor software and mechanisms which layer additional messaging frameworks on top of HTTP
- Provides equivalent functionality when compared to alternative approaches to communication



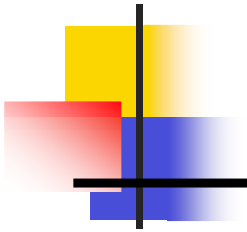
Advantages of REST

- Provides better long-term compatibility and evolvability characteristics than RPC. This is due to:
  - The capability of document types such as HTML to evolve without breaking backwards- or forwards-compatibility
  - The ability of resources to add support for new content types as they are defined without dropping or reducing support for older content types.
- Portability
  - a ReSTful implementation allows a user to bookmark specific "queries" (or requests) and allows those to be conveyed to others across e-mail, instant messages, or to be injected into wikis, *etc*.



Using REST…

- When building Widgets for use in Mashups or RIA's, or building feeds (ATOM or RSS) for use in a browser or by an aggregator
- When you want to make assets available to the web
  - In a form that it can be parsed by the widest range of technologies available
  - Where it may be consumed on either the client or server side
- When the asset you are exposing is naturally resource-oriented



Protocols and Standards

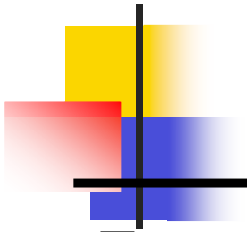
- Transport & Invocation
  - HTTP
  - REST
  - COMET

} **The focus of this presentation**

- Protocols Formats :
  - JSON
  - JSON – RPC
  - Bayeux
  - XML (POX)
  - RSS
  - ATOM

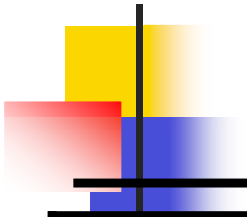
HTTP

The transport foundation of the Internet



HTTP Overview

- Textual protocol, based on MIME
- Request-response exchanges
  - One connection per exchange
- Request message contains
  - Method: GET, PUT, POST, DELETE, ...
  - Address, HTTP version
  - Headers, request entity
- Response message contains
  - HTTP version
  - Status code and reason phrase
  - Headers, response entity



HTTP methods

Method	Description
GET	Retrieves information specified in the URI.
HEAD	Acts in the same manner as the GET method, but it only returns the headers. Used for testing validity, accessibility, and recent modification.
OPTIONS	Returns the HTTP methods supported by the resource.
POST	Sends an arbitrary request with additional information in the body of the request message. Often used to send form data to the application server.
PUT	Publishes or updates a resource on the server. Most Web servers disable this command.
DELETE	Removes a resource on the server. Most Web servers disable this command.
TRACE	Returns the request message via an application-level loopback for testing purposes. Most Web servers disable this command.
CONNECT	Reserved keyword for switching a connection to a tunnel.

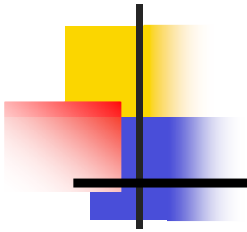
HTTP Core Operations for REST
Simple Set of Operations, *via* the HTTP API

- HTTP provides a simple set of operations.

The HTTP API is CRUD (Create, Retrieve, Update, and Delete)

Amazingly, all Web exchanges are done using this simple HTTP API:

- PUT
  - creates a new resource at an identified location (URI)
  - "here's some new info" (**Create**)
- GET
  - returns a state representation of the identified resource
  - "give me some info" (**Retrieve**)
- POST
  - performs some form of application-specific update to the identified resource
  - "here's some update info" (**Update**)
- DELETE
  - destroys a resource at the identified location (URI).
  - "delete some info" (**Delete**)



An analogy for REST

- A RESTful Web service is formed like a sentence
  - The **Noun** is the URI of the Service (the document)
  - The **Verb** is the HTTP Action performed on the document (GET, POST, PUT, DELETE)
  - An **Adjective** is the MIME type of the resulting document

Sentence	URI (Noun)	Action (Verb)	MIME Type (Adjective)
List all photo albums	../album	GET	JSON
Show a photo	../photo	GET	JPEG
Delete a photo	../photo	DELETE	JPEG
Add a photo	../photo	POST	JPEG

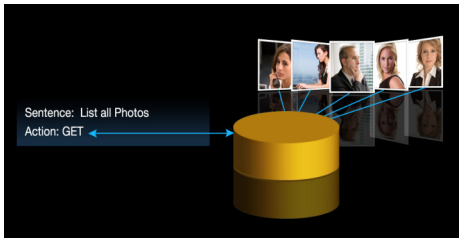
REST and HTTP Methods

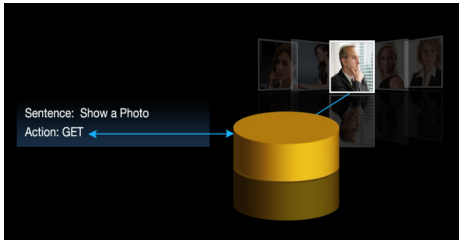
http://<host>/customer

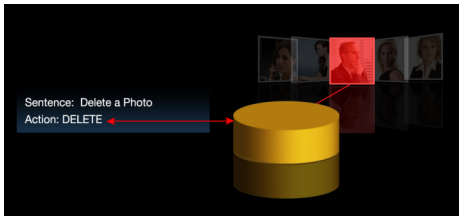
- GET
  - Returns list of customers
- POST
  - Creates Customer Record

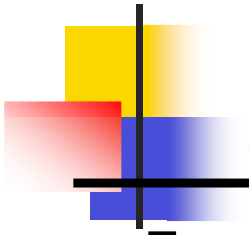
http://<host>/customer/customerID

- GET
  - Returns customer record identified by customerID
- PUT
  - Updates record identified by customerID
- DELETE
  - Delete Record identified by customer









An Example

- For example,
  - a request to /resources/people/1
    (Translates to a request for the resource named “people” with the identifier “1”)
  - Has the collection name "people" and member identifier "1“
  - The collection name identifies the resource handler
    - <apphome>/app/resources/people.groovy
  - and the value of member identifier is provided as a request parameter with name "<collection name>Id"
    - zget("/request/params/peopleId") == 1.

An Example

HTTP Method	URI	….Invokes method in - app/resources/people.groovy	… with event Data
Get	/resources/people	onList()	
Post	/resources/people	onCreate()	

```
def onList() {
      // Get configured DataManager for data access
        def data = zero.data.groovy.Manager.create('peopleDB')
       def result = data.queryArray('SELECT * FROM people')
    // Serialize list to JavaScript Object Notation format
       request.view = 'JSON'
       request.json.output = result
   render()
   }
```

```
def onCreate() {
    // Convert entity to JSON object
       def emp = zero.json.Json.decode(request.input[])
       def data = zero.data.groovy.Manager.create('peopleDB')
       def memberId = data.insert(
              """INSERT INTO people (firstname, lastname)
              VALUES ($emp.firstname, $emp.lastname)""", ['id'])
    // Set a Location header with URI to the new record
       locationUri = getRequestedUri(false) + '/' + memberId
        request.headers.out.Location = locationUri
        request.status = HttpURLConnection.HTTP_NO_CONTENT
}
```

app/resources/people.groovy

An Example

HTTP Method	URI	….Invokes method in - app/resources/people.groovy	… with event Data
Get	/resources/people/1/accounts	onRetrieve()	zget("/request/params/peopleId")==1 zget("/event/pathInfo")==/accounts

```
def onRetrieve() {
    // Member id is parsed from the path
     String id = request.params.peopleId[]
     def data = zero.data.groovy.Manager.create('peopleDB')
     def person = data.queryFirst("SELECT * FROM people WHERE id=$id")
     if(person!= null) {
         // Use ViewEngine JSON rendering
           request.view='JSON'
           request.json.output = person
          render()
      } else {
           // Error handling
         request.status = HttpURLConnection.HTTP_NOT_FOUND
         request.error.message = "username $username not found."
         request.view = 'error'
        render()
      }
}
```

app/resources/people.groovy

An Example

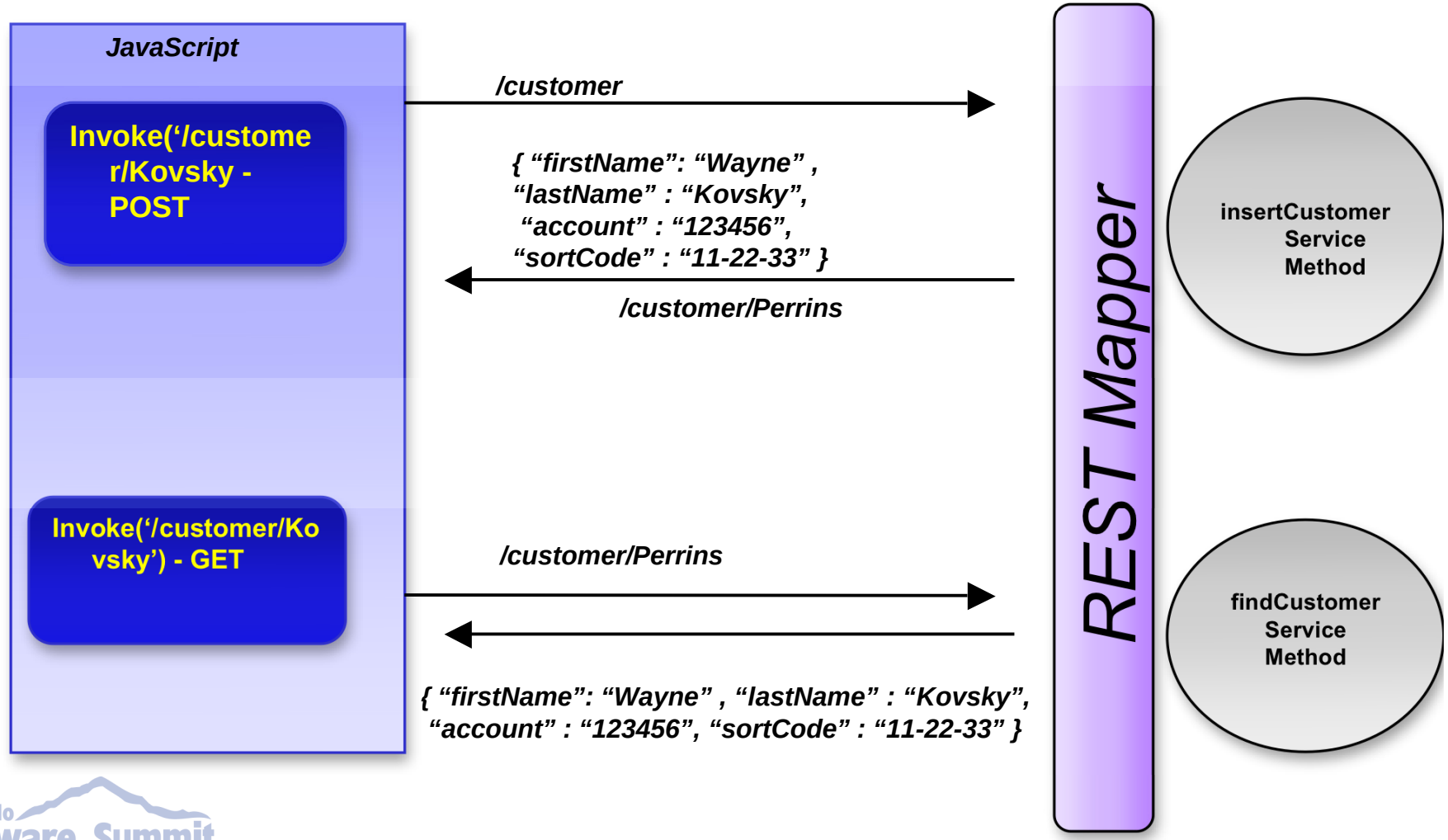
HTTP Method	URI	….Invokes method in - app/resources/people.groovy	… with event Data
Put	/resources/people/1	onUpdate()	zget("/request/params/peopleId")==1
Delete	/resources/people/1	onDelete()	zget("/request/params/peopleId")==1

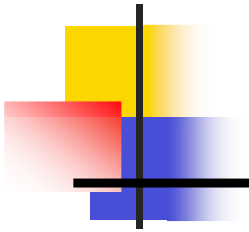
```
def onUpdate() {
        def emp = zero.json.Json.decode(request.input[])
        def data = zero.data.groovy.Manager.create('personDB')
        data.update("UPDATE employees “ + "SET firstname=$emp.firstname,
            lastname=$emp.lastname “ + "WHERE id=$emp.id")
         request.status = HttpURLConnection.HTTP_NO_CONTENT
}
```

app/resources/people.groovy

```
def onDelete() {
      def id = request.params.peopleId[]
      def data = zero.data.groovy.Manager.create('personDB')
      data.update("DELETE FROM people WHERE id=$id");
       request.status = HttpURLConnection.HTTP_NO_CONTENT
}
```

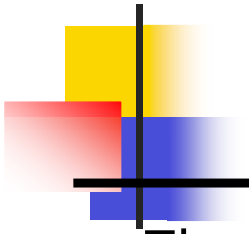
RESTful JSON





Return Values

- Data types
  - Data is encoded as JSON or XML, and ATOM
- Return values are based on HTTP Return codes
  - More than just 200, 404 and 500
  - 2xx response status codes are success
    - and others 3xx, 4xx, 5xx report some kind of error
  - For Example,
    - 204 - No Content
    - 206 - Partial Content
    - 203 - Non-Authoritative Information
    - 202 - Accepted
    - 201 - Created
    - 200 - OK
    - 301 – Moved Permanently
    - 302 – Found
    - *etc*.



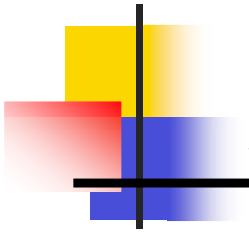
REST *vs* RPC

- The key motivator of a RESTful SOA is its simplicity and its
- ubiquity
  - It's about delivering content in the simplest possible way
  - HTTP is available everywhere; it's like the air around us

- With WS-* It's not the body that matters, it's the headers
  - WS-Security is about choice in the decision of encryption, identity tokens and digital signatures
  - WS-Addressing is about transport-neutral mechanisms of describing addresses
  - Even the WS-I standards are about allowing maximum flexibility within a fully agreed-upon framework of standards

- You might want to choose WS* where you don't have direct

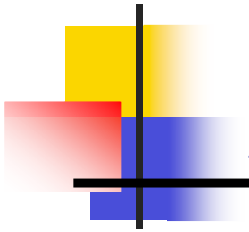
- control of all the pipes.
  - WS-ReliableMessaging and SOAP over JMS are about choice in how you obtain qualities of service



Advantages so far

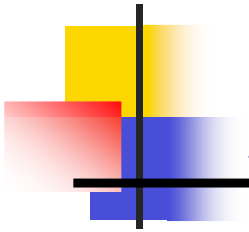
We have a simple, scaleable REST api to work against a wide variety of resources

- Easy to discover
- Easy to transmit
- Easy to script
- Easy to cache
- Easy to secure
- Low risk to adopt



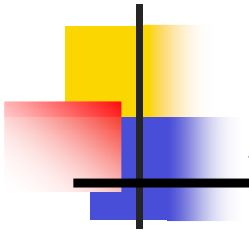
Attributes and advantages of a REST

- Simplicity
  - Many decisions pre-made, constrained choices
    - Fixed protocol (HTTP)
    - Fixed encryption model (HTTPs)
    - Fixed identity token exchange (Basic-Auth or standard HTTP schemes)
- Ubiquity
  - Use the HTTP infrastructure and other technologies like JavaScript that already exist
- Effortless use of services
  - Single, well-understood programming model (Javascript)
  - Lots of examples on the web
    - Copy-cut-and-paste programming to use services
  - Someone should be able to use a RESTful SOA Service without knowing they're doing it!



Attributes and advantages of a REST

- Cacheability (HTTP GET)
  - Unique URI per resource
  - Stateless interactions; response is not a function of how user reaches the URI
  - Can be cached with existing web caching technologies
- Scalability
  - Unique URI per resource enables simple partitioning; leverage distributed data
    POST /foo/{user}/bar – [a-l]* to server1
    [m-z]* to server2
  - Leverages standard HTTP infrastructure for scalability
    - Routers and sprayers
- Testability
  - Reuse your existing HTTP testing infrastructure for functionality and performance
  - Very easy to develop test scripts



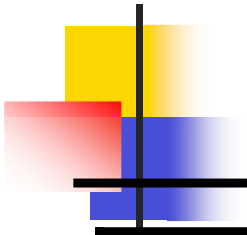
Attributes and advantages of a REST

- Secureability
  - Unique URI per resource; straightforward to set policy on URIs
  - Uses standard HTTP encryption and identity token exchange
  - Can be secured with the existing web security infrastructure

- Navigability
  - Resources can be navigated via hyperlinks
    - Think browser clients
  - *e.g.* GET on a collection returns a list of member URIs and optional paging links (next/prev/first/last)

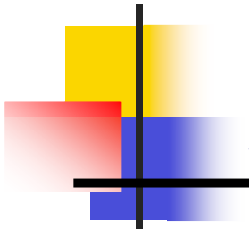
REST *vs* RPC

- REST
  - Resources—Commands are defined in simple terms: resources to be retrieved, stored / get, set—difficult to do many joins
- RPC
  - Commands—Commands are defined in methods with varying complexity: depending on “standard”—easier (?) to hide complex things behind a method
- REST
  - Nouns—Exchanging resources and concepts
- RPC
  - Verbs—Exchanging methods



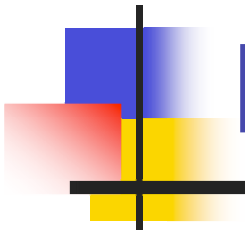
REST and RPC Styles

REST	RPC
Document oriented; Loose typing	Function oriented; Generally strong typing
Unique URI per resource	Overloaded URI as service endpoint**
HTTP methods operate upon resources (GET, POST, PUT, DELETE)	Arbitrary methods, generally specified in the body
Designed for HTTP	Generally designed to hide the network



Advantage of RPC

- "Developability"
  - Simple extension of normal programming constructs.
  - Invoke Web Service/EJB/SCA
  - Easy to consume
  - Rigid - type checking, adheres to a contract
  - Development tool support



REST and SOA

RESTful SOA

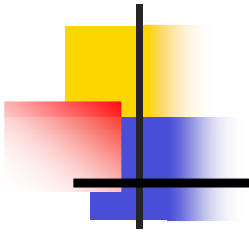
- A RESTful SOA is an instance of SOA that uses concepts from the Web as the primary service architecture
  - REST to represent and access services
    - Entities are addressed via URL
    - GET, POST, PUT, DELETE are the actions
  - Data is encoded as JSON or XML, and ATOM
  - Rich User Interfaces built using AJAX

- Key aspects of building an effective RESTful SOA
  - UI runs in any commodity web-server / browser
  - Make content simple and human readable
  - Use well-established, ubiquitous technologies for scalability, performance and security

- The key motivator of a RESTful SOA is its simplicity and its ubiquity
  - It's about delivering content in the simplest possible way
  - HTTP is available everywhere; it's like the air around us

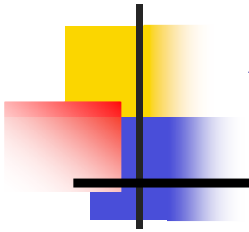
Where can a RESTful SOA apply?

- When building Widgets for use in
  - Mashups or
  - RIA's, or
  - Building feeds (ATOM or RSS) for use in a browser or by an aggregator
- When you want to make assets available to the web
  - In a form that it can be parsed by the widest range of technologies available
  - Where it may be consumed on either the client or server side
- When the asset you are exposing is naturally resource-oriented



RESTful SOA

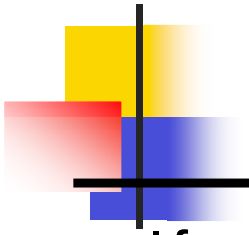
- With WS-* It's not the body that matters, it's the headers
  - WS-Security is about choice in the decision of encryption, identity tokens and digital signatures
  - WS-Addressing is about transport-neutral mechanisms of describing addresses
  - Even the WS-I standards are about allowing maximum flexibility within a fully agreed-upon framework of standards
- You might want to choose WS* where you don't have direct control of all the pipes.
  - WS-ReliableMessaging and SOAP over JMS are about choice in how you obtain qualities of service



Attributes and advantages of a RESTful SOA

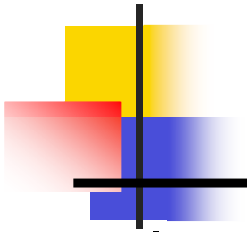
Building on the advantages of REST....

- Simplicity
- Ubiquity
- Effortless use of services
- Cacheability
- Scalability
- Testability
- Secureability
- Navigability



Design for REST first

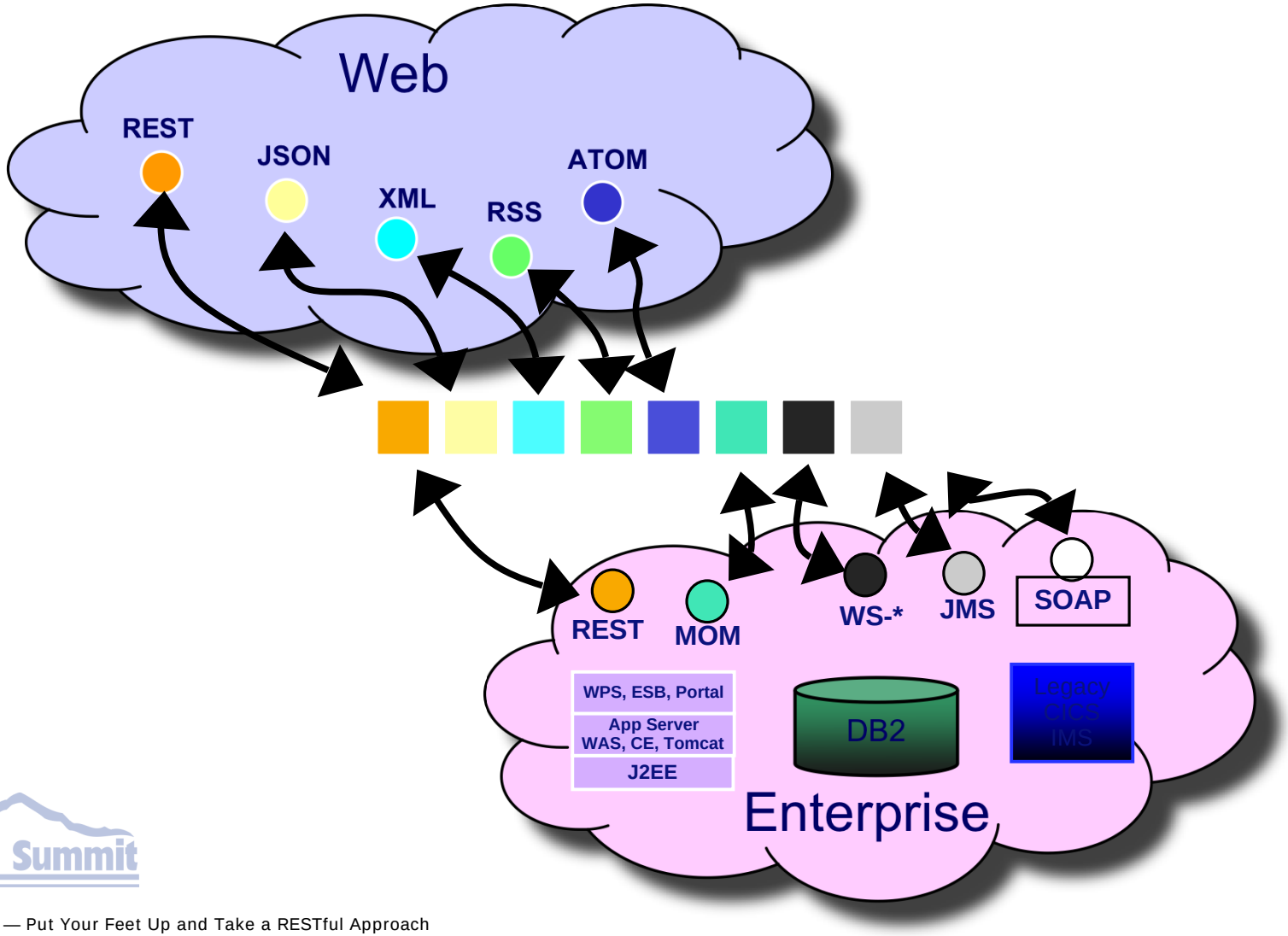
- If you view the Enterprise Web Services as an extension of a simpler REST web service, then you can see some guidelines in constructing your services
  - Focus on developing simple, human-readable XML content
  - Make the content available as both a REST service and as an Enterprise Web Service
  - A REST service can be easily mapped to four (Create/Read/Update/Delete) corresponding Enterprise Web Services that have the same content
  - Add Quality of Service attributes (WS-Security, WS-Addressing, WS-AtomicTransaction) as necessary
- Consider the different Semantics of a Web Extended SOA service from an Enterprise SOA Service
  - Exposing an internal service may not be the best way to reach the masses with your service – the external service may need to represent a unique viewpoint of your enterprise

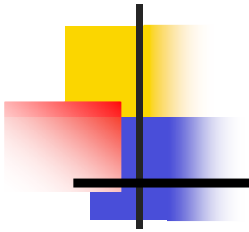


Get the Semantics Right

- Imagine you're extending your Enterprise SOA onto the Web for a worldwide shipping company (DHL, FedEx, UPS, *etc*.)
  - There are two different foci for how you build your services
- The *internal* focus is about optimizing your resources to maximize profitability
  - How full are your planes?
  - How do you minimize fuel costs?
  - What's the cheapest way to get X packages from point A to point B?
- But the *external* focus is only about **MY** package
  - Where is it and when will it get here?

Bridging "RESTful SOA" and "Enterprise SOA"

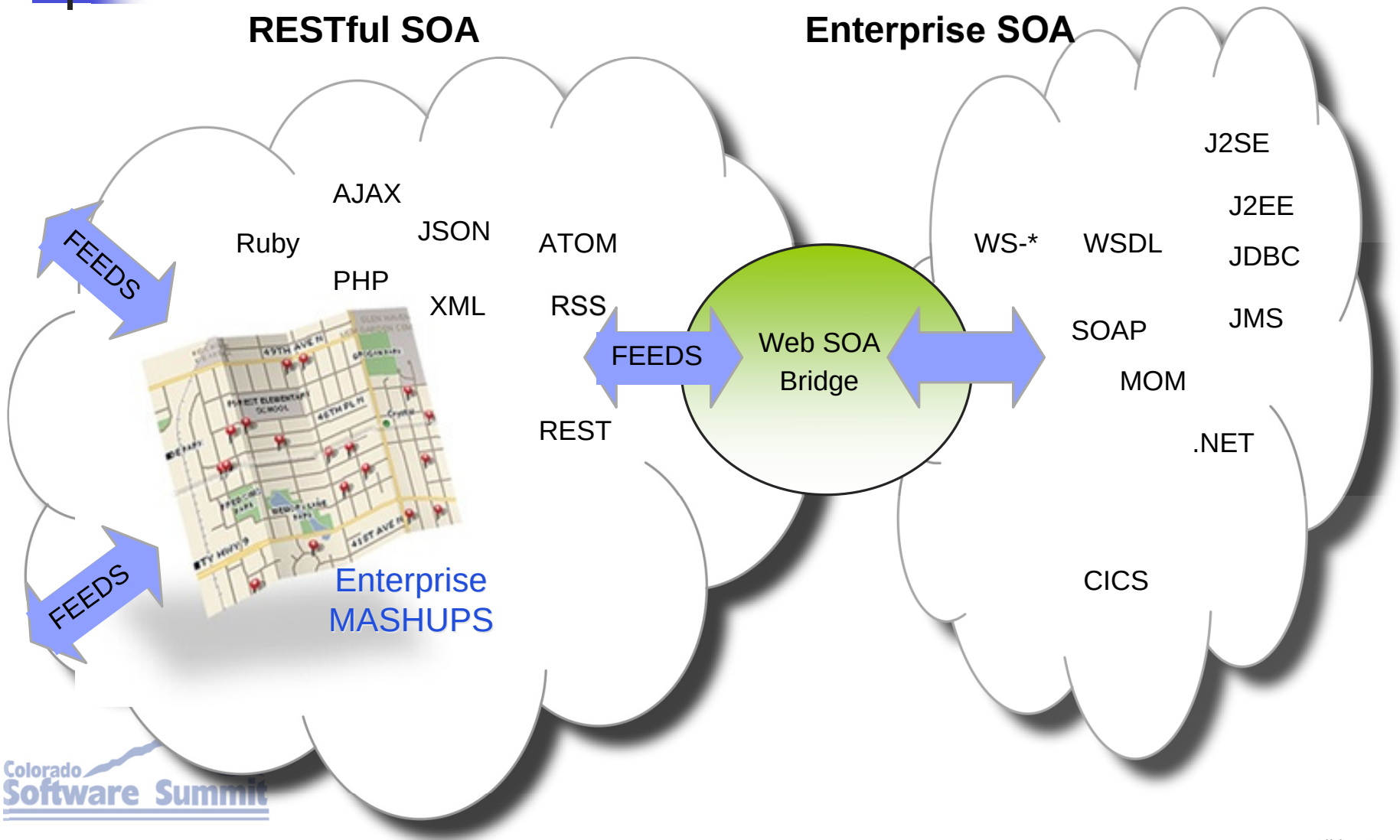




Don't Reinvent Enterprise SOA

- There is often a tendency to reinvent Enterprise SOA in REST
  - Adding additional facilities for security, transactions, *etc*. or
  - trying to run a REST service over a non-HTTP protocol
- These are all the things that an Enterprise Web Service provides
  - Stick within the limitations of the REST approach
  - Use the facilities provided by the HTTP infrastructure
  - If you find you're fighting it too much, reconsider if REST is appropriate in this instance

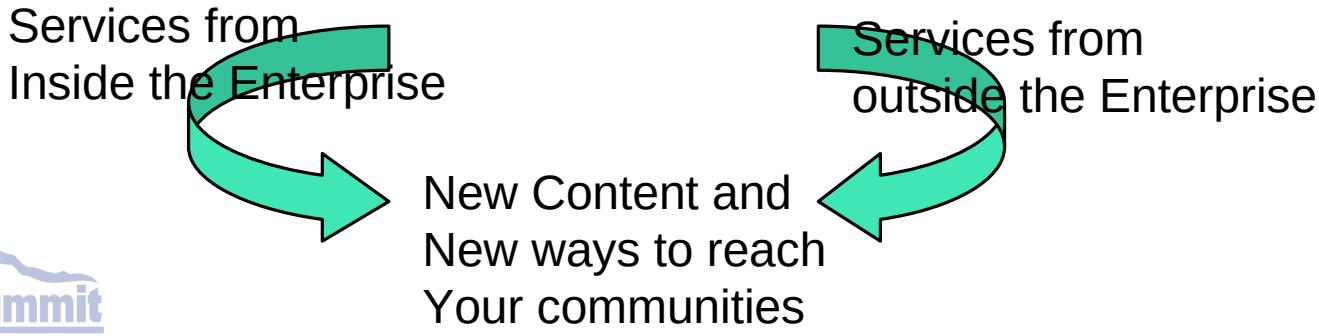
Reducing the complexity

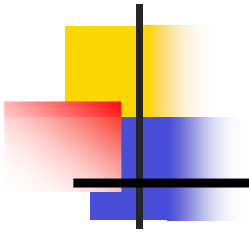


Merging Enterprise SOA and RESTful SOA

- If you take the route of developing for both Enterprise SOA and RESTful SOA then you can take advantage of two separate content pools
  - Services generated inside your enterprise
  - Services generated outside the enterprise

- This gives you the best of both worlds and allows you to take advantage of all the communities served by your business





REST is out there already

- Syndication using RSS
- Bla-bla List
  - Secure, simple, sharable to-do lists.
- AJAX
  - Asynchronous JavaScript and XML
- The blogosphere
  - the universe of weblogs
- Openomy
  - online file storage system
- Amazon
  - offers its developer interface in both REST and SOAP versions
  - with the REST version accounting for most of the traffic
- eBay
  - offers a REST developer interface
- Yahoo
  - offers a REST developer interface