RESTful web services

Frictionless Flow of Information

REST Web Services

- REST = **RE**presentational **S**tate **T**ransfer
- REST is an architectural style consisting of a coordinated set of architectural constraints
- First described in 2000 by Roy Fielding in his doctoral dissertation at UC Irvine
- RESTful is typically used to refer to web services implementing a REST architecture
- Alternative to other distributed-computing specifications such as SOAP
- Simple HTTP client/server mechanism to exchange data
- Everything – the UNIVERSE is available through a URI
- Utilizes HTTP: GET/POST/PUT/DELETE operations

RESTful compared to SOAP Services

RESTFUL

- No significant tools required to interact with theWeb service
- Smaller learning curve
- Efficient (SOAP uses XML for all messages, REST can use smaller message formats)
- Fast (no extensive processing required)
- Closer to otherWeb technologies in design philosophy
- RPC style – short burst messages
- Explosive growth in commercial end user applications

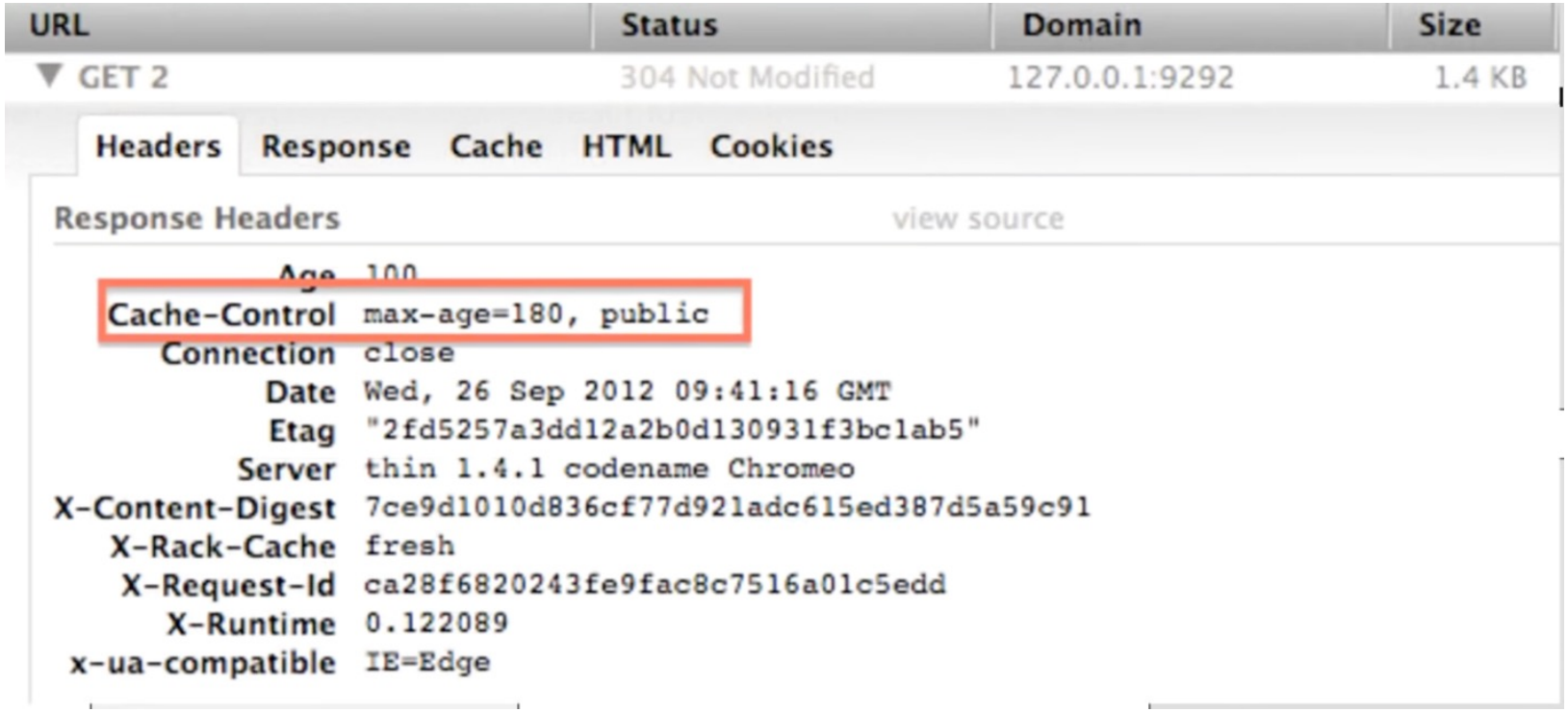
SOAP

- Language, platform, and transport independent (REST requires use of HTTP)
- Works well in distributed enterprise
- environments (REST assumes direct point-to-point comm.)
- Standardized
- Built-in error handling
- Provided document style to better represent domain model
- Strong enterprise adoption – B2B

Unlike SOAP-based web services, there is no "official" standard for RESTful web APIs

Architectural Principles

- Uniform interface
  - Individual resources are identified in requests, i.e., using URIs in web-based REST systems. Resource, URI, HTTP methods (CRUD)
- Client-server
  - Separation of concerns. A uniform interface separates clients from servers.
- Stateless
  - The client-server communication is further constrained by no client context being stored on the server between requests.
- Cacheable
  - Basic WWW principle: clients can cache responses.
- Layered system
  - A client cannot necessarily tell whether it is connected directly to the end server, or to an intermediary along the way.
- Code on demand (optional)
  - REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.


URL
Status
Domain
Size
GET 2
304 Not Modified
127.0.0.1:9292
1.4 KB
Headers
Response
Cache
HTML
Cookies
Response Headers
view source
Age 100
Cache-Control max-age=180, public
Connection close
Date Wed, 26 Sep 2012 09:41:16 GMT
Etag "2fd5257a3dd12a2b0d130931f3bc1ab5"
Server thin 1.4.1 codename Chromeo
X-Content-Digest 7ce9d1010d836cf77d921adc615ed387d5a59c91
X-Rack-Cache fresh
X-Request-Id ca28f6820243fe9fac8c7516a01c5edd
X-Runtime 0.122089
x-ua-compatible IE=Edge

RESTful API HTTP methods

Resource	GET	PUT	POST	DELETE
Collection URI, such as http://example.com/resources	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.	**Delete** the entire collection.
Element URI, such as http://examples.com/resources/item17	**Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	**Replace** the addressed member of the collection, or if it doesn't exist, **create** it.	Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry in it.	**Delete** the addressed member of the collection

POST means "create new" as in "Here is the input for creating a user".
PUT means "insert, replace if already exists" as in "Here is the data for user 5".
PUT is Idempotent

Idempotent means that multiple calls with the same operation doesn't change the representation

Spring MVC REST-style Controller

- Essentially means receive & send the content directly as the message body instead of structuring HTML pages.
- We are **NOT** using HTML
- We are using well-formed XML OR JSON
- Spring support is based on the
  - `@RequestBody` & `@ResponseBody` annotations
  - `@ResponseStatus(value = HttpStatus.NO_CONTENT)`
    - ***For deletes, creates, updates…***
  - `@RestController = @Controller + @ResponseBody`

RequestBody & ResponseBody

- **`@ResponseBody`**
  - Spring framework uses the "`Accept`" header of the request to decide the media type to send to the client.
- **`@RequestBody`**
  - Spring framework will use the " `Content-Type`" header to determine the media type of the Request body received.

- To get XML, MIME media type = "`application/xml`"
- To get JSON, MIME media type = "`application/json`"

JSON (JavaScript Object Notation)


```
{
    "cartId": "1234",
    "cartItems": {
        "P1234": {
            "product": {
                "productId": "P1234",
                "name": "iPhone 5s",
                "unitPrice": 500,
                "description": "Apple iPhone 5s smartphone with 4.00-inch 640x1136 display and 8-megapixel rear camera",
                "manufacturer": "Apple",
                "category": "Smart Phone",
                "unitsInStock": 1000,
                "unitsInOrder": 0,
                "discontinued": false,
                "condition": "NEW"
            },
            "quantity": 1,
            "totalPrice": 500
        }
    },
    "grandTotal": 500
}
```

RESTful Web Service CartRestController

```
@Controller
public class CartRestController {
    @Autowired
    private CartService cartService;

    @RequestMapping(method = RequestMethod.POST)
    public @ResponseBody Cart create(@RequestBody Cart cart) {
        return cartService.create(cart);
    }

    @RequestMapping(value = "/{cartId}", method = RequestMethod.GET)
    public @ResponseBody Cart read(@PathVariable(value = "cartId") String cartId) {
        return cartService.read(cartId);
    }

    @RequestMapping(value = "/add/{productId}", method = RequestMethod.PUT)
    @ResponseStatus(value = HttpStatus.NO_CONTENT)
    public void addItem(@PathVariable String productId, HttpServletRequest request) {
        //Code omitted
    }
}
```

See demo: webstore_rest

HATEOAS

- **HATEOAS (Hypermedia as the Engine ofApplication State)** is a constraint of the REST application architecture that keeps the RESTful style architecture unique from most other network application architectures.
- "***hypermedia***": refers to any content that contains links to other forms of media such as images, movies, and text.
- lets you use hypermedia links in the response contents so that the client can dynamically navigate to the appropriate resource by traversing the hypermedia links.


```
{
   "id": 1,
   "name":"Computing",
   "description":null,
   "_links":{
      "all-categories":{
         "href":"http://localhost:9999/categories"
      },
      "all-categories2":{
         "href":"http://localhost:9999/categories"
      }
   }
}
```

Versioning

- Media type versioning (a.k.a“content negotiation” or“accept header”
  - Github
- (Custom) header versioning
  - Microsoft
- URI versioning
  - Twitter
- Request Parameter versioning
  - Amazon

- Factors
  - URI Pollution
  - Misuse of HTTP Headers
  - Caching
  - Execute the request on the browser
  - API Documentation

Documentation

- REST API Elements
  - Resource descriptions
  - Endpoints and methods
  - Parameters
  - Request example
  - Response example

- Swagger
  - An open spec for describing REST APIs [github]
  - Tools for auto-generating
    - Documentation
    - Code for your API
  - Donated to the OpenAPI initiative and renamed OpenAPI in 2015
- Mashery
  - An open source project [github]
  - Tools for generating
    - Documentation
    - An exploration interface for your API
- Apiary and API Blueprint
  - Write the API description in a DSL within markdown
  - Tools for auto-generating
    - Documentation
    - Mock server
  - *Seems to be focused on ruby+mac devs*
- RAML
  - A spec for describing REST APIs [github]
- WADL
  - A spec for writing discoverable API docs with XML
  - Some discussion comparing WSDL and WADL
  - Java specific

Richardson Maturity Model

- Level 0
  - Expose SOAPWeb Services in REST Style
    - http://server/getPosts
    - http://server/deletePosts
    - http://server/doThis
- Level 1
- Expose Resources with Proper URI
  - Note: Improper use of HTTP Methods
    - http://server/accounts
    - http://server/accounts/10
- Level 2
  - Level 1 + HTTP Methods
- Level 3
  - Level 2 + HATEOAS
  - Data + Next Possible Actions

https://programmerspub.com/blog/general/richardson-maturity-model

Main Point

- REST is defined by architectural constraints. It is able to access information through the ubiquitous URI. Everything on the web is available through a URI.

- *Everything in creation is known through understanding and experience of the Unified Field of Consciousness*

Web 2.0

- WWW sites that emphasize user-generated content, usability, and interoperability.
- Update selected regions of the page area without undergoing a full page reload.
- **Ajax and JavaScript frameworks:**
  - YUI Library - DojoToolkit - MooTools - **jQuery**
  - Ext JS - Prototype JavaScript Framework
  - **Ember.js, React.js, AngularJS, Backbone.js**
- Payload - typically formatted in XML or JSON
- **NOTE: we will use Spring MVC REST technology to create the Web 2.0 payload**

AJAX

- **A**synchronous **J**avascript **A**nd **X**ML
- Web applications are able to make quick, incremental updates to the client without reloading the entire browser page
- The use of XML is not required; JSON is often used instead (AJAJ)
- Examples :
  - Validate values of form fields before saving
  - Dynamically load dropdown values from database
  - Load table data and paginations
  - Live Search
- ***Ajax* -** a broad group of Web technologies that communicates with a server in the background, without interfering with the current state of the page.

JQuery

- **SLOGAN:** The Write Less, Do More, JavaScript Library.

- Fast, small, and feature-rich JavaScript library.
- HTML document traversal and manipulation, event handling, animation, and Ajax.

cart.js

- RESTful services:

- addToCart - Add Item
- removeFromCart - Remove Item

cart.html consumes cart.js example

- Include js files

```
<script type="text/javascript“ src="http://code.jquery.com/jquery-1.10.1.min.js"></script>
<script type="text/javascript" th:src="@{/js/cart.js}"></script>
```

- Bind data for adding item function

```
<a href="#" class="btn btn-warning btn-large order-btn"
th:data="${product.productId}">
    <span class="glyphicon-shopping-cart glyphicon"></span> Order Now </a>
```

- Bind data for removing item function

```
<a href="#" class="label label-danger product-remove-btn"
    th:data="${item.value.product.productId}">
        <span class="glyphicon glyphicon-remove" /></span> Remove</a>
```

See demo: `webstore_rest`

cart.js – register adding item function

```
$('.order-btn').click(function(event){
   event.preventDefault();
   var productId = $(this).attr("data");

   $.ajax({
      url: '/rest/cart/add/' + productId,
      type: 'PUT',
      dataType: "json",
      success: function(response){
        alert("Product Successfully added to the Cart!");
      },
      error: function(){
        alert('Error while request..');
      }
   });
});
```

Replace

```
var contextRoot = "/" + window.location.pathname.split('/')[1];

url: contextRoot + '/rest/cart/add/' + productId,
```

cart.js – register removing from cart

```
$('.product-remove-btn').click(function(event){
   event.preventDefault();
   var productId = $(this).attr("data");

   $.ajax({
      url: '/rest/cart/remove/'+ productId,
      type: 'PUT',
      dataType: "json",
      success: function (response) {
        location.reload(true);
      },
      error: function(){
        alert('Error while request..');
      }
   });
});
```

RESTful input Validation

- How does THAT work?

- **XML - Schema validation is generally not a good idea in a REST service**.
- A major goal of REST is to decouple client and server so that they can evolve separately.

- **What about JSON validation/consistency?**
- API producers have frequently developed their own JSON response formats in the absence of well-defined standards.

ALTERNATIVE OPTION: JSR-303 BeanValidation

category.html

```
<form id="categoryForm" method="post">
  <p>
    <label for="name"> Name : </label>
    <input type="text" name="name" id="name" value="" />
  </p>
  <p>
    <label for="description"> Description: </label>
    <input id="description" name="description"
      type="text" />
  </p>
  <input type="button" id="categorySubmitBtn" value="Add
    Category"/>
</form>
```

AJAX Call

The **JSON.stringify()** method converts a JavaScript value/object to a JSON string EXAMPLE:
JSON.stringify({ x: 5, y: 6 }); yields '{"x":5,"y":6}'

```
$(document).ready(function() {
    $("#categorySubmitBtn").click(function() {

        let data = JSON.stringify($("#categoryForm").serializeFormJSON());
        $.ajax({
            type : "POST",
            url : "http://localhost:8080/BookRestValidation_JavaConfig/api/addCategory",
            data : data,
            contentType: "application/json",
            dataType : "json",
            success : function(data){
                $("#categoryForm")[0].reset();
                $("#result").empty();
                $('#result').append('<H3 align="center"> OK!! <H3><p>').show();
            },
            error: function(XMLHttpRequest, textStatus, errorThrown){
                console.log(XMLHttpRequest.responseJSON);
                $("#result").empty();

                if (XMLHttpRequest.responseJSON.errorType == "ValidationError") {
                    let errorMsg = '<h3> Error(s)!! </h3>';
                    errorMsg += "<p>";
                    var errorList = XMLHttpRequest.responseJSON.fieldErrors;
                    $.each(errorList, function(i, error) {
                        errorMsg = errorMsg +error.message + '<br>';
                    });
                    errorMsg += '</p>';
                    $('#result').append(errorMsg);
                    $('#result').show();
                } else {
                    alert(errorObject.responseJSON.errors(0)); // "non" Validation
                }
            }
        });
    });
});
```

`serializeFromJSON` – custom function to take form data and structure it as a JSON object e.g. [{ x:5,y:6 }]

RequestBody & ResponseBody

```
@RestController
public class CategoryRestController {

   @Autowired
   private CategoryService categoryService;

   // @Valid - but NO BindResult! an exception will be thrown...
   @CrossOrigin(origins = { "http://localhost:9080", "http://localhost:9000"
   },
      maxAge = 6000)
   @PostMapping(value="/api/addCategory", produces = "application/json")
   public Category saveCategory(@Valid @RequestBody Category category) {
      categoryService.save(category);
      return category;
   }
}
```

if NO BindingResult **bindingResult** in Signature,

MethodArgumentNotValidException will be thrown if Category fails validation...

Form Validation Exception Handling

```
@ControllerAdvice
public class RestErrorHandler {
    @Autowired
    private MessageSourceAccessor messageSourceAccessor;

    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ResponseBody
    public ValidationErrorDTO processValidationError(MethodArgumentNotValidException ex) {
        BindingResult result = ex.getBindingResult();
        List<FieldError> fieldErrors = result.getFieldErrors();
        return processFieldErrors(fieldErrors);
    }

    private ValidationErrorDTO processFieldErrors(List<FieldError> fieldErrors) {
        ValidationErrorDTO dto = new ValidationErrorDTO("ValidationError");

        for (FieldError fieldError : fieldErrors) {
           dto.addFieldError(fieldError.getField(), messageSourceAccessor.getMessage(fieldError));
        }
    return dto;
```

See demo:
Bookrestserver
Bookrestclient

```
<bean id="messageSourceAccessor" class="org.springframework.context.support.MessageSourceAccessor">
    <constructor-arg  ref="messageSource"/>
</bean>
```

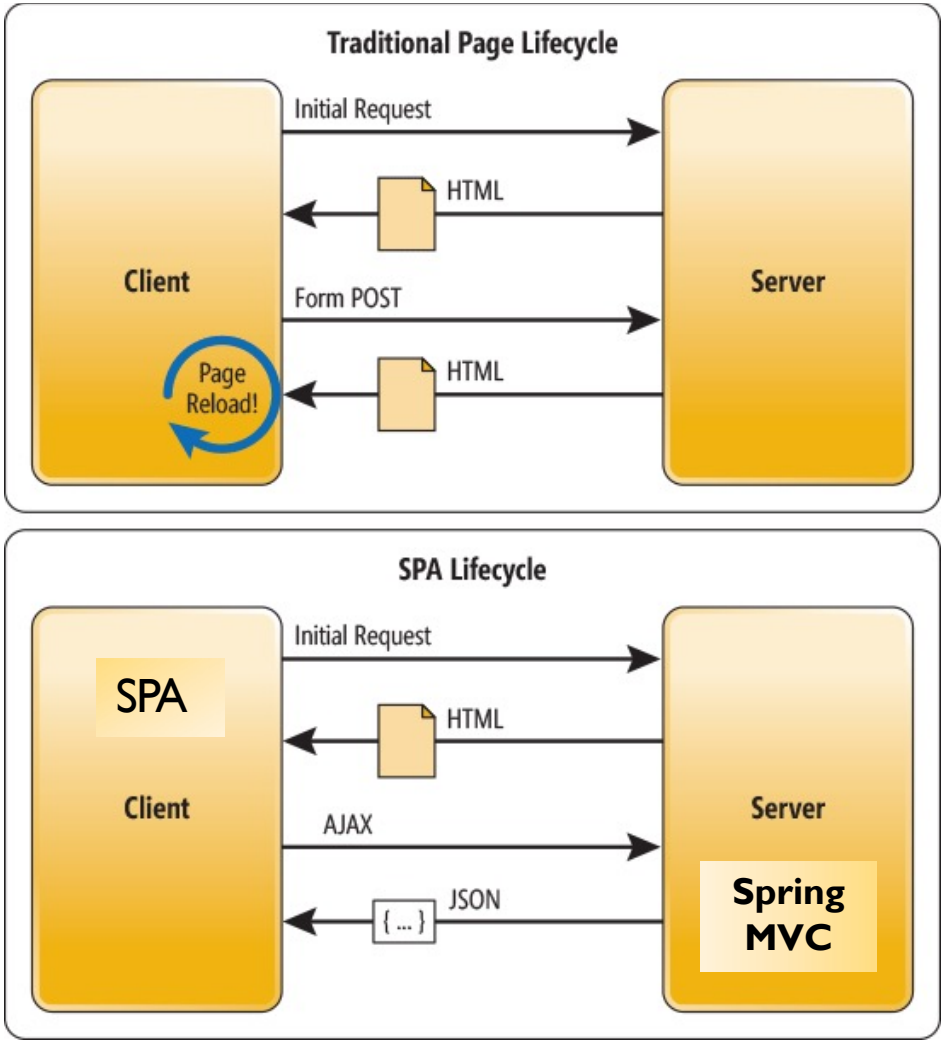
Validator Error DTOs

```
/**
 * This DTO contains the information of a single validation error.
*/
public class FieldErrorDTO {
   private String field;
   private String message;
}
/**
 * This DTO wraps those validation errors together
*/
public class ValidationErrorDTO {
   private String errorType;
   private List<FieldErrorDTO> fieldErrors = new ArrayList<>();

   public void addFieldError(String path, String message) {
      FieldErrorDTO error = new FieldErrorDTO(path, message);
      fieldErrors.add(error);
   }
}
```

Spring MVC Rest Controller & SPA

- Web application that fits on a single to provide a more fluid user experience.
- All the "heavy lifting" presentation-wise is done on the client side, in JavaScript.
- Server interaction with a SPA involves dynamic communication with the web server behind the scenes [RE:Ajax].
- Spring RESTful Controller is a perfect fit for Server side support of SPA.



Main Point

Ajax uses Javascript in a browser to access data or functionality residing on a server and then quickly and efficiently selectively update the browser.

Nature is maximally efficient.