

# RESTFul Design

Jon Flanders

Instructor

Pluralsight

[jon.flanders@gmail.com](mailto:jon.flanders@gmail.com)

<http://www.rest-ful.net/>

# About Me

- Jon Flanders –  
<http://www.rest-ful.net/>
- Independent consultant/  
Instructor
- MS MVP
- Don't fret –  
technology in this sess



# Overview

- **What is REST?**
- **The constraints of REST**
- **Why REST?**
- **URI Design**
- **URI “issues” : “processing”, query string, caching**
- **Verbs**
- **Media Types**
- **Using HTTP to your advantage**
- **HATEOAS**

# REST

- **Representational State Transfer**
  - introduced by Roy Fielding
- **Its an architectural style**
- **A set of constraints distilled from the architecture of the Web**

# Architectural Constraints

Applications are modeled using resources (things)

Every resource is uniquely identified by a URI

Resources are self-descriptive (media types)

Clients interact with a resource through its URI and the uniform interface (limited verbs)

Resources contain links to other resources and can inform the client of the current state of the application

# REST Advantages

- **GET responses can be cached**
- **Using URIs build on experience using the Web**
- **Uniform interface simplifies building and using services**
- **Statelessness constraint eases scalability**
- **GET is safe, can be called N times without causing change**
- **PUT and DELETE are idempotent (same effect no matter how many times called)**
- **Wide-spread interoperability**

# Building a RESTful service

- **Design your resource(s)**
- **Determine the URI for each resource**
- **Determine what part of the uniform interface each resource should implement**

# Using URIs is key, “good” URIs are not

- **URIs on the web are supposed to be opaque**
  - `http://example.org/995B6BB1-B20C-4a73-A206-658734A77F31` (opaque) versus `http://example.org/users/jon/info` (transparent)
  - Both could point to same resource
- **Users of the human web prefer “pretty” and “hack-able” URIs**
  - Some developers prefer this as well
  - Flickr.com is a good example of this
- **Implementation details sometimes dictate part of the URI**
  - Generally a good practice to design URIs first, and then redirect to implementation specific URIs
- **This is an “art” not a “science”, and can be subjective**

# Mapping resources to URIs

- **Often URIs will flow naturally based on your resources**
  - Especially if you have pure hierarchal resources
  - <http://example.org/users>
  - <http://example.org/users/jon>
  - <http://example.org/users/aaron>
  - <http://example.org/users/jon/connections>
- **Sometimes URI design is more difficult**

# Common sense URI design rules

- Each part of your resource hierarchy should be a separate path segment

<http://example.org/orders/5542>

- If you have information about the resource that isn't part of a hierarchy

- Use a semicolons to separate if order doesn't matter

<http://example.org/albumsby/Johnny%20Cash;Emmylou%Harris>

- Use a comma to separate if order does matter

<http://example.org/color/FF,FF,DC>

- Use a query string to indicate execution of an algorithm

<http://example.org/orders/search?name=Jon>

# Uniform interface

GET

- Retrieves a resource
- Guaranteed not to cause side-effects (SAFE)
- Results are cacheable

POST

- Creates a new resource
- Unsafe: effect of this verb isn't defined by HTTP

PUT

- Updates an existing resource
- Used for resource creation when client knows URI
- Can call N times, same thing will always happen (idempotent)

DELETE

- Removes a resource
- Can call N times, same thing will always happen (idempotent)

# To POST or not to POST

- **POST is the default verb for creating a new resource**
- **If the user-agent has the ability to specify the URI of the new resource use PUT**

# URIs + Verbs

- For each resource you determine what verb(s) it will support
- This is usually a fairly mechanical mapping

URI	Verb	Description	Output	Input
/users	GET	All users	List of users	n/a
/users	POST	Creates a new user	user	user (id not specified)
/users/{id}	GET	Returns the user based on id	user	n/a
/users/{id}	PUT	Modifies the user resource	user	user
/users/{id}	DELETE	Deletes the resource	n/a	n/a

# “Processing”

- **In a RESTful system everything should be considered a “resource”**
  - No “operations” (e.g. “ProcessOrder”)
  - <http://example.org/ProcessOrder> == bad URI
- **Generally the correct answer is to create a new resource**
  - <http://example.org/order> - use POST to create a new resource
  - <http://example.org/transaction> - use POST to create a new resource

# Query string “verbs”

- **Often you see people “slip” verb/method as a query string parameter**
  - Bad design, but the purpose is to get around user agent limitations around verb limitation
  - Browser s/RIA are the culprits
- **Better to keep URIs “pure” and use X-HTTPMethod-Override**

# URI templates

```
http://example.org/orders/{orderid}
```

```
http://example.org/orders/{orderid}.{format}
```

```
http://example.org/orders/search?name={name}
```

- **Useful to parameterize the variable path segments of URIs**
  - Variable paths can be substituted with variables
  - Query string parameters as well
- **Simplifies parsing and concatenating of URIs on the client**
  - There is a argument that this violates the HATEOAS constraint
  - Standardization movement started
    - <http://bitworking.org/projects/URI-Templates/>
- **Useful for declaring routing rules for server implementations**
  - Substituted values can be passed to your code

# Media Types

- **Using media types is essential to REST**
  - The self-describing constraint
- **There is some disagreement on how to use them correctly**
- **General consensus – stay away from standard opaque media types**
  - application/xml etc
- **Use standard transparent media types when you can**
  - application/atomsvc+xml etc
  - Check out all options before going to next step - <http://microformats.org/>
- **Sometimes that doesn't work**
  - Register new media types with IANA (takes along time)
  - Use application/vnd.XXXXXX (reduces interoperability)

# URI and media types

- Should your URI indicate media type?

`http://example.org/orders/5542.json`

- What about the Accept header?

```
HTTP /orders/5542 GET
Host: http://example.org
Accept: application/xml
```

- **Generally you want to use an extension on the URI**
  - OSes (especially Windows) use the extension to determine resource disposition (important only for human web really)
- **Recommendation: support both Accept or URI extension**
  - Frameworks can make this difficult

# Media Types and versioning

- **One interesting way to version is by using media types**
  - application/vnd.myformat
  - applicaiton/vnd.myformatv2
- **In general URIs shouldn't change**
  - Is version 2 of your resource a different resource? If so you can change URI
  - Is version 2 of your resource just slightly different
- **Using vendor specific media types can reduce interoperability**
  - Sometimes you don't care

# HTTP

- **REST of course can work over any protocol**
- **Most of the advantages work because of the nature of HTTP**
- **Learning about HTTP == RESTful goodness**

# Caching

- **One benefit of using REST is taking advantage of caching**
  - Web servers can cache
  - Proxies can cache
  - Browsers can cache
- **Caching is really one of the main reasons the web scales**
  - Web would screech to a halt if caching was turned off tomorrow

# Conditional GET

- **Using Conditional GET is one really important part of caching**
  - Other caching can be turned off by security
- **Conditional GET can be accomplished with either**
  - ETags
  - Last-Modified header
- **Reduces processing and bandwidth**
- **User agent will make conditional GET request**
  - Either If-None-Match (for ETags) or If-Modified-Since (for Last-Modified)
  - Server returns 304 to indicate client has the most recent version of the resource

# URIs and caching

- **If you use query string**
  - Typically intermediary won't cache
- **Sometimes you want this behavior**
- **General rule – avoid query string parameters if you want to maximize caching**

# Status codes

- **Sending correct status codes should be part of your standard operating procedures**
- **201 with Location header for resource creation**
- **404 if resource not found**
  - Many toolkits return 500 if code has exception
- **304 for condition GETs**

# Hypertext for application state

- **A.K.A. HATEOAS**
  - Hypertext as the engine of application state
- **Simple idea**
  - Same as human web, following links from “home page”
- **RESTful clients shouldn't have pre-defined knowledge of URIs**
  - Some people argue against UriTemplates as well

# Summary

- **Having each resource identified by a unique URI is essential to REST**
- **Having “nice” URIs is a value judgment**
- **Defining a set of design rules for your URIs is a good practice**
- **Don’t forget to plan for caching**
- **Use HTTP to your advantage**
- **Use standard transparent media types when possible**
- **Use HATEOAS to be more “pure”**

# References

- Cool URIs don't change <http://www.w3.org/Provider/Style/URI>
- URL as UI <http://www.useit.com/alertbox/990321.html>
- URI templates project <http://bitworking.org/projects/URI-Templates/>