REST API Design Principles

Verbs

Response Codes

Resource Representations

Being Navigational

API Versioning

Being Intentional

REST API Design Principles

1. Use the right verbs
   - Keep your URIs consistent across verbs
2. Use the right response codes!
3. Resource Representations: When in doubt, use JSON
   - But if you can, return appropriate media (Content) types
4. Try to be *navigational*
5. Version your API
6. Be *intentional*

Let's talk about each of these some more

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!

More on HTTP Verbs

Some hints for best practices:

- Name the business objects (collections) of your domain
  - e.g. http://myapi.org/asu/campuses
    - GET on this returns { ‘Poly’, ‘West’, ‘Downtown’, ‘Tempe’, ‘online’ }
    - POST creates a new campus
    - PUT replaces a campus
    - PATCH (such as ‘East’ to ‘Poly’) changes the campus name attribute
    - DELETE deletes a campus
- URIs should infer relationships that make sense to the consumer
  - e.g. http://myapi.org/courses/ser422/instructors
    - GET on this returns a collection (list) of instructors for ser422
    - POST on this creates a new instructor for ser422
    - PUT replaces the instructor of SER422
    - DELETE deletes the instructor/course relationship
- Relationships may also be obtained through the response
  - GET http://myapi.org/courses/ser322/instructors
    - May return { {‘Gary’, ‘http://myapi.org/courses/ser322/instructors/kgary’}, {‘Bansal’, ‘http://myapi.org/courses/ser322/instructors/Bansal’}}

HTTP Response Code

…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

Resource Representations

1. Return JSON _and_ XML
   - Use a query string param (like OMDB), *Accept* header, or file extension
     - e.g. http://myapi.org/asu/campuses/poly.json or poly?format=json
   - The Accept header is most HTTP-ish, but then you get into returning custom file extensions (or a whole bevy of them) w/ implied priority
   - XML can be used largely for legacy reasons and because it doesn't really cost you much extra usually, but prefer JSON
2. Wrap responses (I am not a big fan of this)
   - Some frameworks (client and server side) make it difficult to get at underlying HTTP information like response codes
   - A technique is to *wrap* a response so this info is included in payload
     - Ex: {'code':200 'status':'success','data': { <some stuff> }}
     - Ex: {'code':401 'status':'error','msg':'Invalid token':'data':'Unauthorized'}
3. HATEOS: elegant concept with no practical implementation?
   - HATEOS has some big implications for implementation
     - Semantically appropriate media types – custom → coupling?
     - Navigable API with a single endpoint? - This implies a lot of round trips

We will talk HATEOS a bit more in our next discussion on semantic formats

Navigation Best Practices

1. Use the Location header for POSTs
   - When creating a new resource, the response should use response code 201 and a “Location:” header with link to new resource URI
2. Return limited set of links and rely on the consistency of your API for consumer to know how to get what s/he wants
   - i.e. an id that the consumer can stick on the URI to a collection
3. Support CORS
   - What if your return links go to another server? JS may think its XSS
     - You can use JSONP (hack) or CORS
   - Cross-Origin Resource Sharing – web spec (http://www.w3.org/TR/cors/)
     - Use header “Access-Control-Allow-Origin: <domains> in the *response*
4. Choose a semantic payload format that supports information linking
   - *Stay tuned, this is the subject of our next discussion…*

Versioning your API

IMHO, this is kind of a mess out there right now

- A number of organizations, in their rush to be RESTful, implemented APIs that have needed refactoring
  - Some used incorrect design principles
  - Some just didn't have the bandwidth to up the maturity level
  - Many evolved their APIs from specific use cases, and thus lacked the *anarchically stable* concept
  - And well, evolution of software is just natural
- So how then, do you deal with putting out a new API version, and not breaking all your existing clients?
  - Require a version number in your Accept request header, and expect one in the Content-type of the response
  - Require a version number in your URI
  - If absent, assume the *oldest supported version*
  - Provide a never-to-change endpoint that *handshakes* a version

Intentional API Design

What does this mean?

Ensure your API is only invoked in ways that your service intends; prevent accidental misuse

Intentional API Design Principles

- Some just part of REST – consistency, proper use of verbs
- Consider how APIs may be *choreographed*
  - That is, how your API may be invoked in sequence
  - This is a bit contrary to our “individual behavior” viewpoint
- Conceal internals of implementation
  - If you expose *how* you do something, or any notion of implementation, then your client may “game the system” and become tightly coupled to you
- Provide excellent documentation
- Assume limited query results, make caller provide explicit parameters to request more information
  - e.g. GET http://myapi.org/v2/parts/adapters?resultsize=1000 (default 10)
  - May also use a *Range* header for this (Content-range on response)