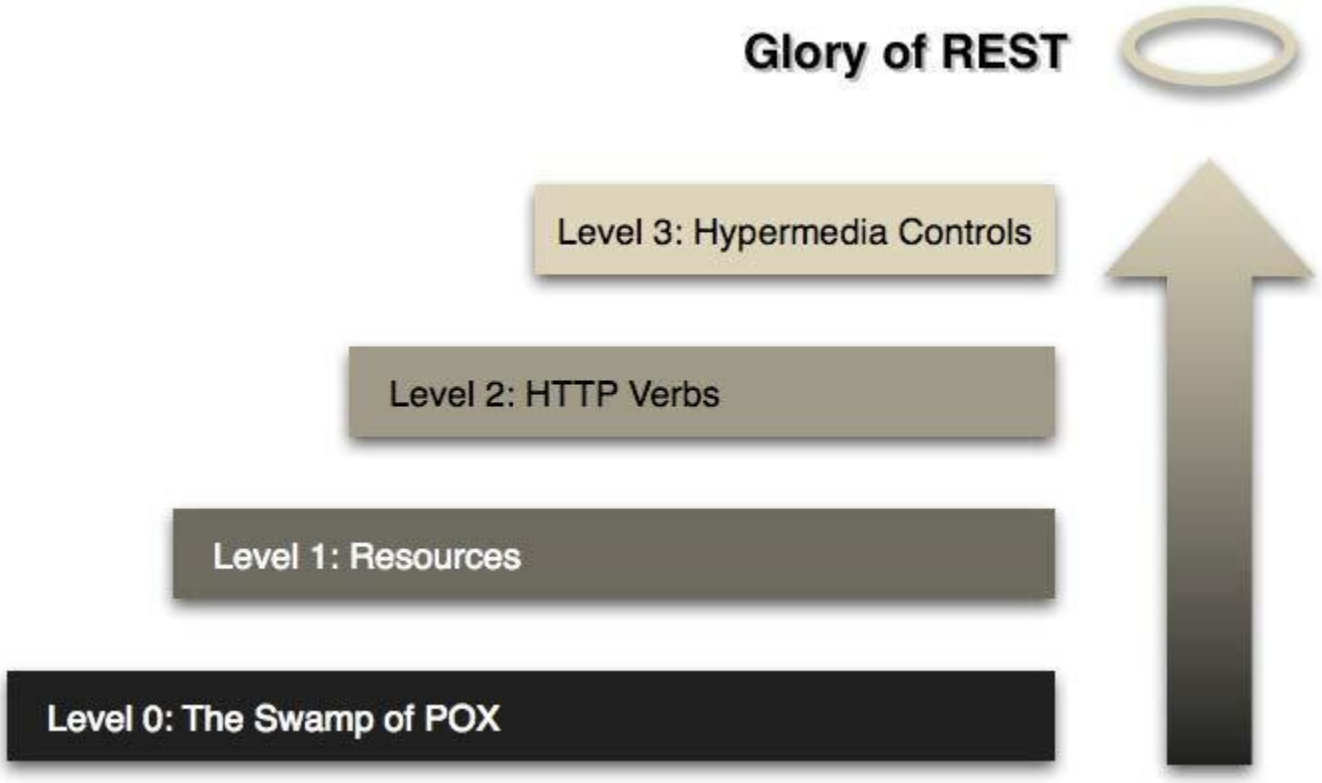
Best Practices for writing RESTful APIs

March, 2015 (v1.1)

WHAT IS REST?

- **Representational State Transfer (REST)**
  - Introduced in 2000 by Roy Fielding in his doctoral dissertation at UC Irvine
  - An architectural style for creating distributed hypermedia systems a.k.a. network-based applications
  - Describes the software engineering principles guiding the design and implementation of such applications
  - It is a philosophy, not a library or a framework for implementation!

RESTFUL MATURITY MODEL



A model developed by Leonard Richardson that breaks down the key elements of REST into three levels

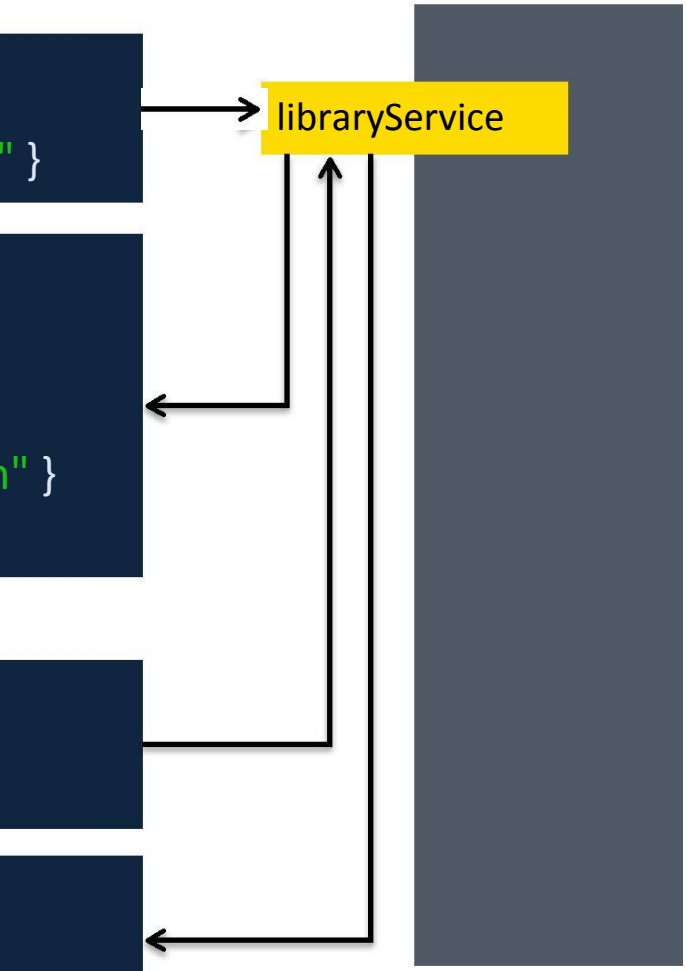
LEVEL 0: REMOTE PROCEDURE CALLS (RPC)

```
POST /libraryService HTTP/1.1

{ "method": "searchBooks", "titleContains": "amazon" }
```

```
HTTP/1.1 200 OK

[
   { "id": 1735, "title": "The Amazon River" },
   { "id": 5288, "title": "The Discovery of the Amazon" }
]
```

```
POST /libraryService HTTP/1.1

{ "method": "reserveBook", "bookId": 5288 }
```

```
HTTP/1.1 200 OK

{ "reservationId": 36827 }
```



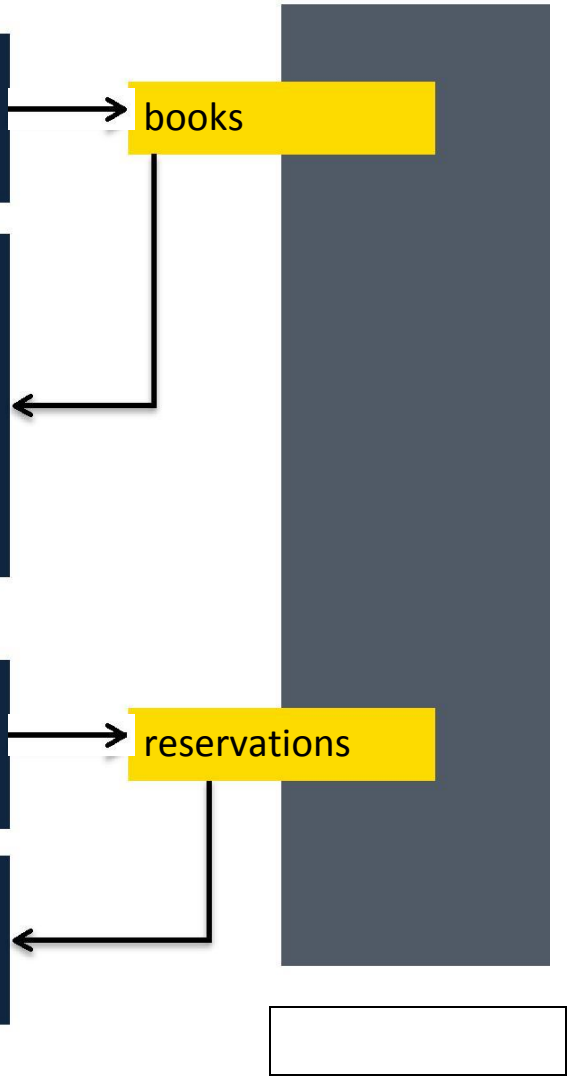
LEVEL 1: RESOURCES

```
POST /books HTTP/1.1

{ "method": "searchBooks", "titleContains": "amazon" }
```

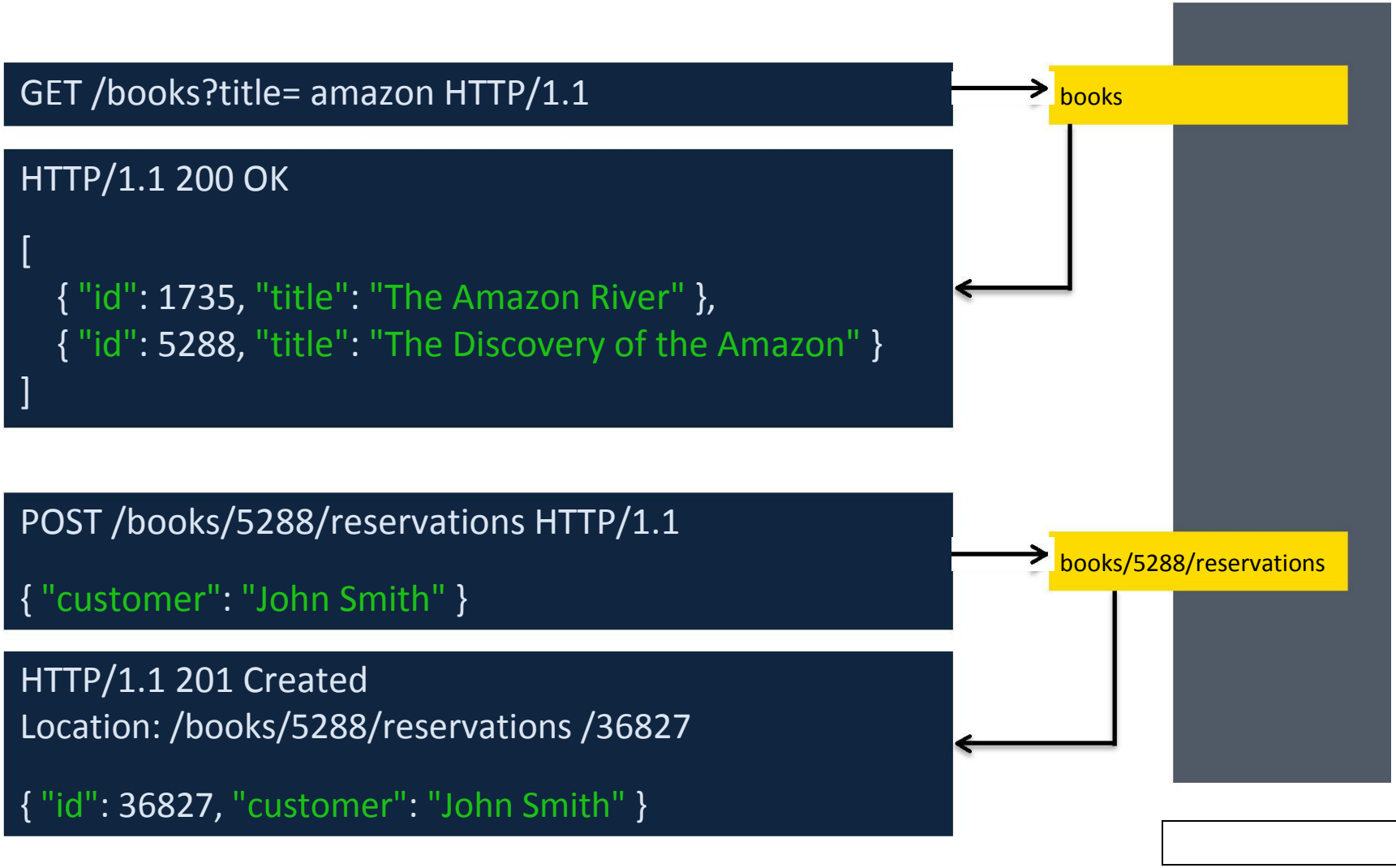
```
HTTP/1.1 200 OK

[
   { "id": 1735, "title": "The Amazon River" },
   { "id": 5288, "title": "The Discovery of the Amazon" }
]
```

```
POST /reservations HTTP/1.1

{ "method": "reserveBook", "bookId": 5288 }
```

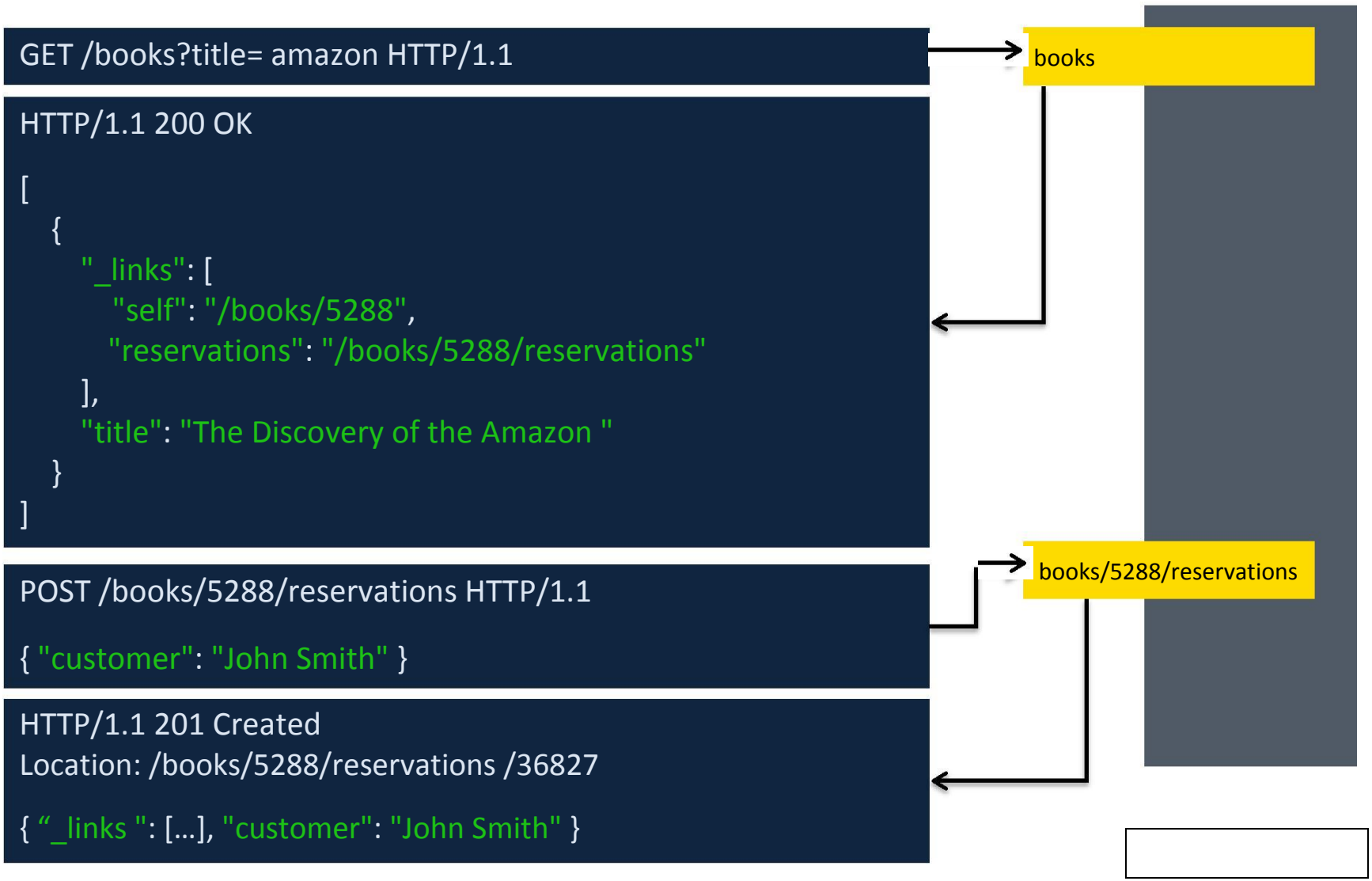
```
HTTP/1.1 200 OK

{ "reservationId": 36827 }
```



LEVEL 2: HTTP VERBS



LEVEL 3: HYPERMEDIA CONTROLS[1]



1: HATEOAS (Hypertext As The Engine Of Application State)

SHOULD I IMPLEMENT LEVEL 3?

- Ideally, YES! However depending on your platform, support for hyperlinks may vary. Make your decision carefully as it will directly affect your effort.
- Here's a pragmatic point-of-view from a blog article, but use it in conjunction with other evaluation criteria for your project:

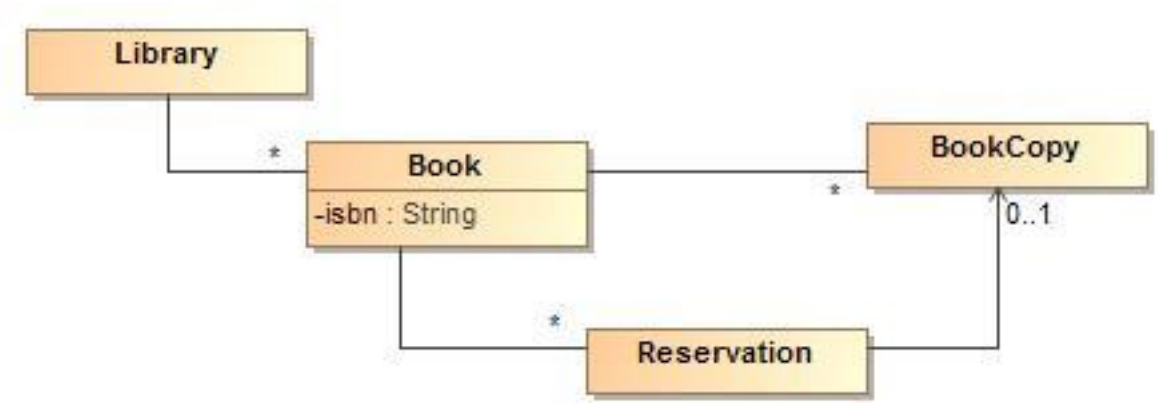
  "Although the web generally works on HATEOAS type principles (where we go to a website's front page and follow links based on what we see on the page), I don't think we're ready for HATEOAS on APIs just yet. When browsing a website, decisions on what links will be clicked are made at run time. However, with an API, decisions as to what requests will be sent are made when the API integration code is written, not at run time. Could the decisions be deferred to run time? Sure, however, there isn't much to gain going down that route as code would still not be able to handle significant API changes without breaking. That said, I think HATEOAS is promising but not ready for prime time just yet. Some more effort has to be put in to define standards and tooling around these principles for its potential to be fully realized."

DEMO – MOBILE OXFORD

- **Provides help with day-to-day tasks related to University of Oxford and the City of Oxford**
  - Finding library books
  - Checking the next bus
  - Nearest post office box
- Application: http://m.ox.ac.uk
- RESTful API: http://api.m.ox.ac.uk/browser

- Github is another example of a RESTful API
  - It supports HATEOAS
  - See http://developer.github.com/v3/

UNDERSTAND YOUR DOMAIN *Before designing the API*



1: HATEOAS (Hypertext As The Engine Of Application State)

DOCUMENT YOUR API

"An API is only as good as its documentation. The docs should be easy to find and publicly accessible. Most developers will check out the docs before attempting any integration effort. When the docs are hidden inside a PDF file or require signing in, they're not only difficult to find but also not easy to search."

- Best Practices for Designing a Pragmatic RESTful API

"Honestly, if you don't conform 100% to the criteria in this guide, your API will not necessarily be horrible. However, if you don't properly document your API, nobody is going to know how to use it, and it WILL be a horrible API.."

- Principles of good RESTful API Design

We recommend using Postman to document your REST API. Postman has a feature called "Collections" which allows you to organize requests into a nice folder structure. Each request can be supplemented with one or more JSON responses to show different scenarios. These collections can also be used to automate the testing of your server.

NAME YOUR RESOURCES CORRECTLY

- Resources should nouns, not verbs
  - Good: /books
  - Bad: /searchBook, /reserveBook
- There should only be two types of resources
  - Collection resources, e.g. /books (note that these are plural)
  - Instance resources, e.g. /books/1234

DON'T CONFUSE HTTP VERBS WITH CRUD

- GET, PATCH and DELETE are straightforward:
  - GET = *Read*
  - PATCH = *Update*
  - DELETE = *Delete*
- PUT and POST are not so obvious
  - Both can be used for *Create* and *Update*
  - POST can do *Create* and *Update* (including partial updates) because it doesn't have to be idempotent (an operation that produces the same result regardless of how many times it is called)
  - PUT can do both *Create* and *Update*, but only full updates. Also it must supply the id of the resource in both cases. These restrictions are due to the requirement that PUT must be idempotent.
  - P.S. GET, HEAD, PUT and DELETE are all idempotent operations

USE THE RIGHT HTTP VERB TO ADD BEHAVIOR

- `GET /books`
  - Retrieve all books
- `GET /books/16`
  - Retrieve book #16
- `POST /books`
  - Create a book
- `PUT /books/16`
  - Update book #16 (full update)
- `PATCH /books/16`
  - Update book #16 (partial update)
- `DELETE /books/16`
  - Delete book #16

DIFFERENCE BETWEEN URI, URL and URN

- Based on W3C recommendations:
  - A Uniform Resource Identifier (URI) is a string of characters used to identify a resource. Different schemes can be used for identification.
  - "http:" is a URI scheme for identifying resources by their location. It is used primarily for identifying web resources, e.g. http://archfirst.org. The URI is called a URL because it is location-based.
  - "urn:" is a URI scheme for identifying resources by their name. For example, `urn:isbn:0-486-27557-4` unambiguously identifies a specific edition of Shakespeare's play *Romeo and Juliet*. Here "isbn" is called a namespace. To gain access to this object, you will need a URL.

CHOOSE A SIMPLE BASE URL

- Two approaches:
  - http://api.archfirst.com (preferred, short and sweet)
  - http://www.archfirst.com/api (slightly longer, but allows both web content and the API to be served from the same server)

PAY ATTENTION TO URL DESIGN

- How to deal with relations?
  - If the relation can only exist within another resource, it may be better to locate it via the parent, e.g. `/books/12/reservations/56`
  - Github example: `/repos/:owner/:repo/notifications`
  - Use the same location for all HTTP verbs (GET, PUT, POST, DELETE)
  - It is ok to have an alias representing the same resource, for example: `/notifications`
- For many-to-many relationships, it may be better to introduce a meaningful intermediate resource
  - Project to Person is many-to-many
  - Introduce a `/members` resource
  - Allows adding relationship specific properties, e.g. `memberSince`

USE QUERY PARAMS TO OPTIMIZE GET RESPONSES

- Filter a collection
  - `GET /books/1234/reservations?filter=dueDate lt $today`
- Sort a collection
  - `GET /books/1234/reservations?sort=dueDate`
- Fetch related resources
  - `GET /books/1234?expand=reservations`
- Fetch a subset of fields
  - `GET /books?select=isbn,title`

VERSION YOUR API

- Example: http://api.archfirst.com/v1
- Do not use minor version numbers (v1.1 etc.)
  - Doesn't mean much for a data API
  - Always increment to the next major version number (v2, v3 etc.)
- Don't change versions too often
  - Painful to adapt to breaking changes
  - Your clients will not be very happy with you!
- Note that adding attributes and resources does not introduce breaking changes

FORMATTING REQUESTS AND RESPONSES

- Blank fields should be returned as null instead of being omitted
- Use the ISO 8601 standard to represent dates and times and always represent time in UTC, e.g.

```
{
  "startTime": "2015-04-01T09:00:00Z"
}
```

- Return the resource in response to a POST, PUT and PATCH
  - Because some properties may have changed (e.g. timestamp)
  - Exception: when resource is very large (say a 2MB video), it doesn't make sense to send back the same resource

RESOURCES

- Roy Fielding's doctoral dissertation introducing REST
- Richardson Maturity Model – Martin Fowler
- Beautiful REST + JSON APIs – Les Hazlewood, Stormpath
  - Video presentation
  - Textual Summary
- The RESTful CookBook – FAQ and recipes for RESTful APIs
- Hypertext Application Language (HAL) – a specification for implementing HATEOAS
  - halberd – HAL implementation for Node.js
  - Spring Data REST – HAL implementation for Java
- OData – Another specification for implementing HATEOAS (good .NET support)
- Domain-Driven Design – Naresh Bhatia