RESTful Services Design

Design Methodology

1. Identify and name resources to be exposed by the service
   - actors, movies
2. Model relationships between resources that can be followed to get more details
   - same actor in different movies, different actors in same movie
3. Define "nice" URIs to address the resources
4. Map HTTP verbs to resources
   - GET movie, POST movie, …
5. Design and document resource representations
   - we want to serve JSON (and XML)
   - the JSON mime-type is application/json (and application/xml)
6. Implement and deploy Web Service
7. Test with cURL or browser developer tools

REST API Design

1. Who will use the APIs?
2. What are we trying to achieve with the API?

- Make application developers as successful as possible
- Keep things simple
- Take the developer's point of view!

Simple Nouns!

- REST URIs are opaque identifiers that are meant to be discovered by following hyperlinks and not constructed by the client
- Simple and intuitive base URLs
  - GOOD: /actors
  - BAD: /peopleplayingin80iesmovies
- 2 base URLs per resource
  - GOOD: /actors (collection)
  - GOOD: /actors/1234 (specific element in collection)
- Keep verbs out of your base URLs
  - BAD: /getAllActors

Simple Nouns!

- Using plural nouns might be more intuitive
  - GOOD: /movies
  - GOOD: /actors
- Singular nouns are OK, but avoid mixed model
  - GOOD: /movie /actor
  - BAD: /movies /actor
- Prefer a manageable number (12-24) of concrete entities over abstraction
  - GOOD: /movie /actor /producer /cinema
  - BAD: /item

HTTP Verbs

Resource	POST (create)	GET (read)	PUT (update)	DELETE (delete)
/actors	Create a new actors	List actors	Bulk update actors	Delete all actors
/actors/1234	Error	Show actor 1234	If exists update actor 1234 else error	Delete actor 1234

Handle Errors

- Use HTTP status codes
  - over 70 are defined; most APIs use only subset of 8-10
- Start by using
  - 200 OK (… everything worked)
  - 400 Bad Request (… the application did something wrong)
  - 500 Internal Server Error (… the API did something wrong)
- If you need more, add them
  - 201 Created
  - 304 Not Modified
  - 401 Unauthorized
  - 403 Forbidden
  - …

HTTP Status Codes

100 Continue
200 OK
201 Created
202 Accepted
203 Non-Authoritative
204 No Content
205 Reset Content
206 Partial Content
300 Multiple Choices
301 Moved Permanently
302 Found
303 See Other
304 Not Modified
305 Use Proxy
307 Temporary Redirect

4xx Client's fault

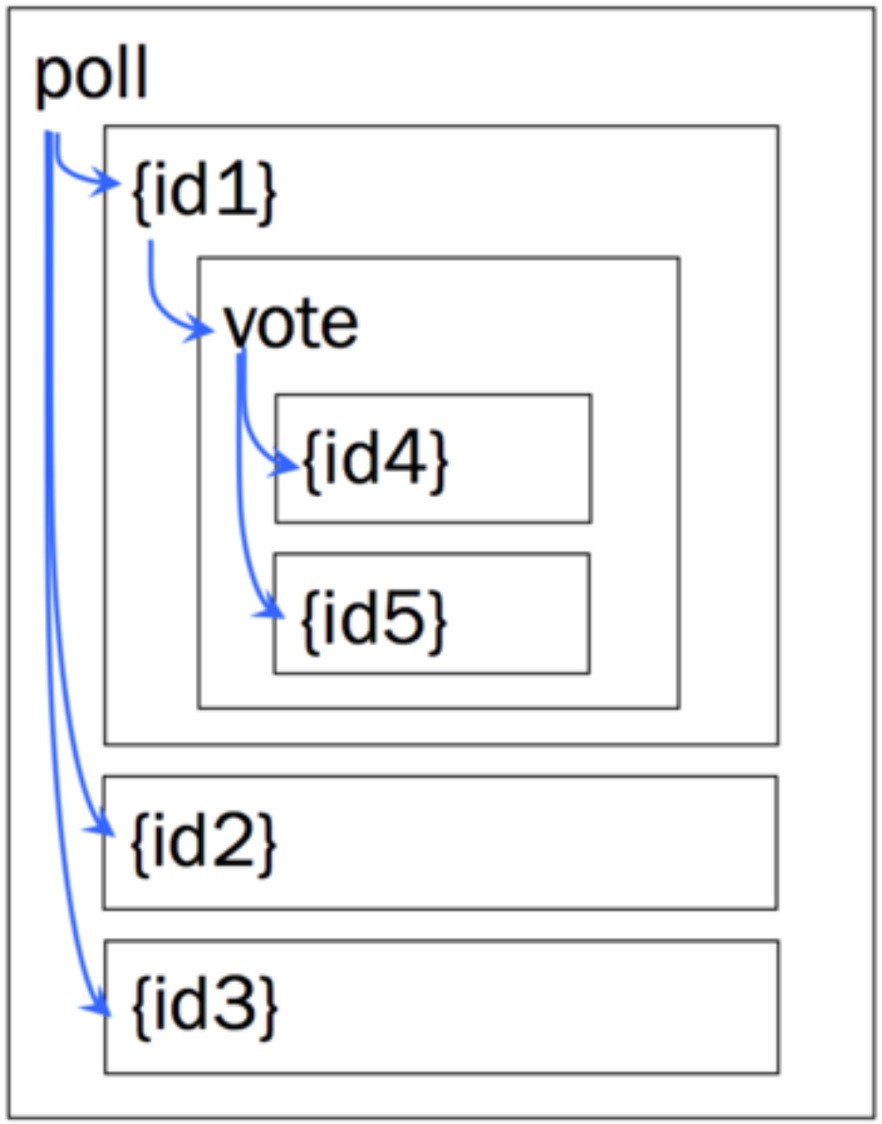
400 Bad Request
401 Unauthorized
402 Payment Required
403 Forbidden
404 Not Found
405 Method Not Allowed
406 Not Acceptable
407 Proxy Authentication Required
408 Request Timeout
409 Conflict
410 Gone
411 Length Required
412 Precondition Failed
413 Request Entity Too Large
414 Request-URI Too Long
415 Unsupported Media Type
416 Requested Range Not Satisfiable
417 Expectation Failed

500 Internal Server Error
501 Not Implemented
502 Bad Gateway
503 Service Unavailable
504 Gateway Timeout
505 HTTP Version Not Supported

5xx Server's fault

Simple DOODLE

1. Resources: polls and votes
2. Containment relationships:



3. URIs embed ids of child instance resources
4. POST on the container is used to create child resources
5. PUT/DELETE for updating and removing child resources

	GET	PUT	POST	DELETE
/poll	✓	✗	✓	✗
/poll/*{id}*	✓	✓	✗	✓
/poll/*{id}*/vote	✓	✗	✓	✗
/poll/*{id}*/vote/*{id}*	✓	✓	✗	?

Simple DOODLE APIs Example

1. Creating a poll (transfer the state of a new poll on the service)

```
POST /poll
<options>A,B,C</options>

201 Created
Location: /poll/090331x
```

2. Reading a poll (transfer the state of the poll from the service)

```
GET /poll/090331x

200 OK
<options>A,B,C</options>
<votes href="/vote"/>
```

Simple DOODLE APIs Example

3. Creating a vote (Participating in a poll)

```
POST /poll/090331x/vote
<name>N. Tonellotto</name>
<choice>B</choice>
```

```
201 Created
Location: /poll/090331x/vote/1
```

4. Reading a poll (with votes)

```
GET /poll/090331x
```

```
200 OK
<options>A,B,C</options>
<votes>
  <vote id="1">
    <name>N. Tonellotto</name>
    <choice>B</choice>
  </vote>
</votes>
```

Simple DOODLE APIs Example

5. Updating a vote (Changing a vote)

```
PUT /poll/090331x/vote/1
<name>N. Tonellotto</name>
<choice>C</choice>
```

```
200 OK
```

6. Reading a poll (with votes changes)

```
GET /poll/090331x
```

```
200 OK
<options>A,B,C</options>
<votes>
  <vote id="1">
    <name>N. Tonellotto</name>
    <choice>C</choice>
  </vote>
</votes>
```

Simple DOODLE APIs Example

3. Deleting a poll

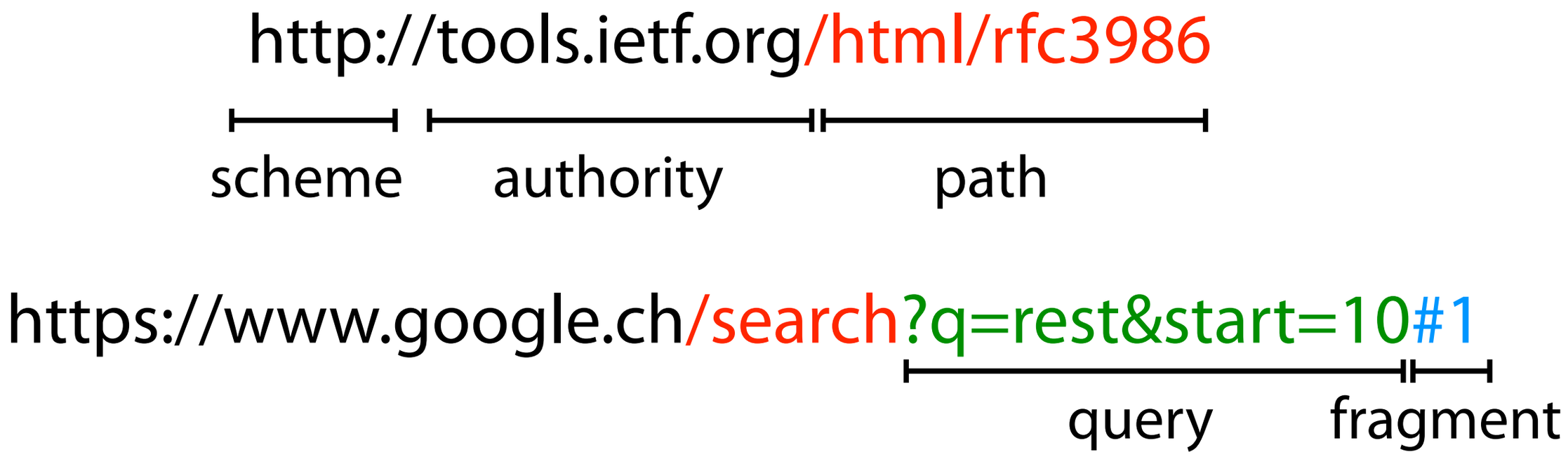
```
DELETE /poll/090331x
200 OK
```

4. Reading a poll (delete)

```
GET /poll/090331x
404 Not found
```

Info on the real DOODLE APIs: http://doodle.com/xsd1/RESTfulDoodle.pdf

URI Design



- REST does not advocate the use of "nice" URIs
- In most HTTP stacks URIs cannot have arbitrary length (4Kb)
- #Fragments are not sent to the server
- Do not hardcode URIs in the client!

URI Templates

- URI Templates specify how to construct and parse parametric URIs.
  - On the service they are often used to configure "routing rules"
  - On the client they are used to instantiate URIs from local parameters

- Do not hardcode URI templates in the client!
- Reduce coupling by fetching the URI template from the service dynamically and fill them out on the client

URI Examples

- URI Template:

  http://www.myservice.com/order/{oid}/item/{iid}

- Example URI:

  http://www.myservice.com/order/XYZ/item/12345

- URI Template:

  http://www.google.com/search?{-join|&|q,num}

- Example URI:

  http://www.google.com/search?q=REST&num=10

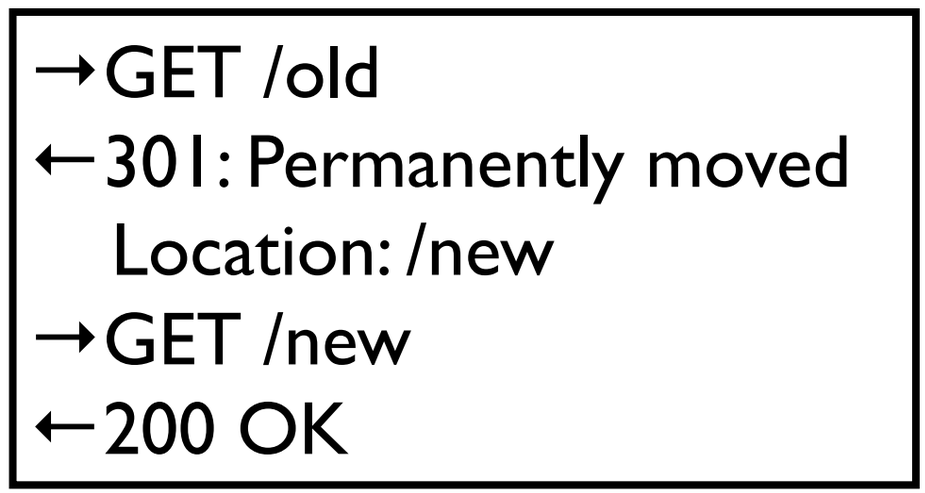
Uniform Interface Constraints

CRUD	REST	Action	Safe	Idempotent
CREATE	POST	Create a (sub)resource	NO	NO
READ	GET	Retrieve the *current state* of the resource	YES	YES
UPDATE	PUT	Initialize or update the state of a resource at a given URI	NO	YES
DELETE	DELETE	Clear a resource, after the URI is no longer valid	NO	YES

Redirection

- Problem:
  - URIs may change over time for business or technical reasons.
  - It may not be possible to replace all references to old links simultaneously risking to introduce broken links.
  - How can consumers of a RESTful service adapt when service locations and URIs are restructured?

- Solution:
  - HTTP Redirection
  - HTTP natively supports redirection using a combination of 3xx status codes and standard headers:
    - 301 Moved Permanently
    - 307 Temporary Redirect
    - Location: /newURI



Content Negotiation

- Problem:
  - Service consumers may make different assumptions about the messaging format
  - A service may have to support both old and new consumers without having to introduce a specific interface for each kind of consumer.
- Solution:
  - Specific content and data representation formats to be accepted or returned by a service capability is negotiated at runtime as part of its invocation.
  - The service contract refers to multiple standardized "media types".

```
→GET /resource
  Accept: text/html, application/xml, application/json
←200 OK
  Content-Type: application/json
```