designing web APIs

Daniel Jackson
old web apps & their structure
standard web app, circa 2000

- browser
- session state
- HTML pages
- get, post
- web server
- scripts
- templates
- session state
- database

like dialog boxes of an app
no clean MVC separation

cookies index into server session state
eg, what user just did
URLs

URLs look like this
› http://store.com/shop.asp?cmd=nextPage

exposes implementation
› names should be designed for clients, not services

meaning is contextual
› depends on session state
› can’t go back, replay or bookmark
› what will refresh do?
data encoding

consequences of server sending HTML?

hard to modify app
  › presentation and content are intertwined
  › how to make a new app?

formatting and layout on server side
  › computational burden on shared resource
  › extra latency
session state

server stores per user session state

when does session end?
› too early: times out before user is done
› too late: user returns and is stuck in mode
› server-side app needs garbage collection

scalability
› where is session data stored?
› load balancing: split requests must access shared DB

inflexible navigation
› user must follow narrow path
cool new web apps
from an infamous rant

So one day Jeff Bezos issued a mandate... that was so out there, so huge and eye-bulgingly ponderous... something along these lines:

1) All teams will henceforth expose their data and functionality through service interfaces....

Over the next couple of years, Amazon transformed internally into a service-oriented architecture...

—Steve Yegge, a former Amazon employee [October 12, 2011]

see https://plus.google.com/112678702228711889851/posts/eVeouesVvaVX
a new (old?) model

- browser
- minimal session state

- server sends HTML, XML, JSON

- data

- get, post

- web server
- controller
- service
- minimal session state

- client sends XML, JSON

- only for authentication

- looks like service layer of traditional app
URLs

URLs look like this
› http://store.com/items/123/images/4

each URL is an operation
› eg, show an item; add an item to cart; purchase

URLs can be replayed
› can go back, bookmark, etc

no HTTP requests for small steps
› updating display, incremental data
› so reduce server-side burden
data encoding

structured data in both directions
› like a conventional procedure call
› JSON, XML, microformats, etc

where to render?
› on client side, using JavaScript eg
› on server side, using Flask templates

accept header in request
› lets user agent specify response type
› server can provide both HTML and XML, eg

now basic functions & UI decoupled
session state

limited session state
› just for authentication
› can limit to client side cookies

scalability
› session expiry, session storage no longer concerns
› load balancing: can split requests across servers
› caching: can process GET in local store
interleaving & statelessness

what do you lose with stateless interaction?
› transactions!
› must allow all interleavings

unexpected orderings
› for your network stickies project
   what if user modifies sticky that has been deleted?
operations on a movie queue

POST http://movies.com/users/123/queue/items/4?action=top
DEL http://movies.com/users/123/queue/items/1
ReSTful web apps
ReST

‘representational state transfer’
› architectural assumptions behind HTTP
› according to Roy Fielding’s 2000 PhD
› but now a meaningless buzzword?

meaning 1:
› transfer whole resources (eg, reviews, sales orders)
› minimize session state

meaning 2:
› pass data as GET queries and simple JSON/XML
› don’t bother with interface def languages (WSDL, SOAP)

meaning 3:
› standardize the syntax of URLs
SOAP

originally Simple Object Access Protocol
› unrelated to Service Oriented Architecture

what is it
› a message protocol, on top of HTTP (or SMTP, etc)
› a language- & platform-indep XML syntax for messages

WSDL: Web Services Description Language
› machine-readable description of a web service interface
› client can read WSDL to know what ops are available
› can use to translate messages (eg, from Java calls)

often used together
› WSDL + SOAP
SOAP example

POST /reviews HTTP/1.1
Host: www.tipster.com
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
 xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
 <soap:Body
 xmlns:m="http://www.tipster.com/review">
   <m:GetReviews>
     <m:SubjectName>Peets Coffee</m:SubjectName>
     <m:ReviewAuthor>Chloe Closure</m:ReviewAuthor>
   </m:GetReviews>
 </soap:Body>
</soap:Envelope>

request body
› XML document with rich structure
› XML namespace makes field names globally unique
POST /reviews HTTP/1.1
Host: www.tipster.com
Content-Type: application/json; charset=utf-8
Content-Length: nnn

{subjectName: Peets Coffee, reviewAuthor: Chloe Closure}

JavaScript Object Notation
› JavaScript’s object literal syntax (but language indep too)
› popularized by Doug Crockford, starting 2001

advantages
› usually smaller than XML
› better integrated with JavaScript
standardizing URLs

URL names
› a resource or a collection of resources

elements
› /subjects
› /subjects/4
› /subjects/4/reviews
› /subjects/4/reviews/3
ReSTful names from object models

URL is a path in the OM
what’s nice about standard URLs?

- friendly to humans
  - guess what they mean
  - remember them
  - derive new from old
    - /subjects/4/reviews/3
- stable
  - good for bookmarks
  - links won’t die so soon
- systematic
  - encourage structure in app
  - brings OM to fore
disciplined use of HTTP methods

old approach
› request is GET, with query string giving op and args
› or request is POST, with form data for args
› eg, GET /reviews.php?action=update&reviewid=23&rating=5

ReSTful approach
› request is resource name + HTTP method
› eg, PUT /subject/3/review/2

standard interpretations
› GET: return resource(s)
› POST: add new resource
› PUT: replace resource
› DELETE: replace resource
operation properties

properties of operations
› ‘safe’: don’t modify state
› idempotent: >0 requests same as exactly one

rationale
› results can be cached for safe operations
› browser can warn about unsafe operations (eg resubmit)
› idempotent operations can be safely repeated

<table>
<thead>
<tr>
<th>operation</th>
<th>safe?</th>
<th>idempotent?</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>POST</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>PUT</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>DELETE</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
examples of ReSTful requests

get reviews of subject 3
› GET /subjects/3/reviews

create new review of subject 3
› POST /subjects/3/reviews

update review 2 of subject 3
› PUT /subjects/3/reviews/2

delete review 2 of subject 3
› DELETE /subjects/3/reviews/2
problems with ReSTful URLs

obtaining forms: not such a nice fit
› GET /subjects/3/reviews/new
› GET /subjects/3/reviews/2/edit

non-CRUD operations
› PUT /subjects/3/reviews/2?action=approve
› POST /subjects/3/reviews/2/approvals

binary operations
› POST /subjects/3/merge?subject=4
problems with HTTP method dogma

where does the argument go?
› GET: query string
› POST: request body

what methods are available?
› for forms, only POST and GET
› PUT and DEL supported only in Ajax

example: search form submission
› ReST says use GET, but then messes up URL

example: edit form submission
› ReST says use PUT, but can’t do in plain HTML
Yahoo weather: not like the ReST

For the Weather RSS feed there are two parameters:

- \( w \) for WOEID.
- \( u \) for degrees units (Fahrenheit or Celsius).

The WOEID parameter \( w \) is required. Use this parameter to indicate the location for the weather forecast as a WOEID.

http://weather.yahooapis.com/forecastrss?w=location

For example, to get weather for Yahoo! Headquarters in Sunnyvale, CA, use the WOEID code for Sunnyvale (2502265):

http://weather.yahooapis.com/forecastrss?w=2502265

Would a ReSTful URL scheme help?
exercise: designing a ReSTful scheme

registrar’s system
› classes, students, terms
› faculty add and remove classes, assign grades
› students register for and drop classes

how would you encode these operations?
self-describing data
self-describing data

when a response comes back
› what do you do with the data?

traditional approach
› API doc tells you what to do with it

self-describing data
› response carries its own description
› tells you how to display it, what it means, what to do next

how does the web support this idea?
› media types (eg, as MIME)
› human-readable fields (in XML or JSON)
› links & link relations
a DNS resource record

```plaintext
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
|                                               |
|                      NAME                     |
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      TYPE                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     CLASS                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     TTL                      |
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                   RDLENGTH                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                     RDATA                     /
    /                                               /
    |                                               |
```

The diagram represents a DNS resource record with fields for NAME, TYPE, CLASS, TTL, RDLENGTH, and RDATA.
Google directions response message

```json
{
  "status": "OK",
  "routes": [
    {
      "summary": "I-40 W",
      "legs": [
        {
          "steps": [
            {
              "travel_mode": "DRIVING",
              "start_location": {
                "lat": 41.8507300,
                "lng": -87.6512600
              },
              "end_location": {
                "lat": 41.8525800,
                "lng": -87.6514100
              },
              "polyline": {
                "points": "a~l~Fjk~uOwHJy@P"
              },
              "duration": {
                "value": 19,
                "text": "1 min"
              }
            }
          ]
        }
      ]
    }
  ]
}
```
example: atom feed

```xml
<?xml version="1.0" encoding="utf-8"?>

<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Example Feed</title>
  <subtitle>A subtitle.</subtitle>
  <link href="http://example.org/feed/" rel="self"/>
  <link href="http://example.org/"/>
  <id>urn:uuid:60a76c80-d399-11d9-b91c-0003939e0af6</id>
  <updated>2003-12-13T18:30:02Z</updated>
  <author>
    <name>John Doe</name>
    <email>johndoe@example.com</email>
  </author>
  <entry>
    <title>Atom-Powered Robots Run Amok</title>
    <link href="http://example.org/2003/12/13/atom03"/>
    <link rel="alternate" type="text/html" href="http://example.org/2003/12/13/atom03.html"/>
    <link rel="edit" href="http://example.org/2003/12/13/atom03/edit"/>
    <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
    <updated>2003-12-13T18:30:02Z</updated>
    <summary>Some text.</summary>
  </entry>
</feed>
```

› note use of link relations, how server returns self id too
HATEOAS

a trendy (ugly) name for this idea
› Hypermedia as the Engine of Application State

some people say
› essential part of ReST

allows stateful interaction
› iterators awkward? article/page/5
› hypertext to rescue: can use nextpage link relation
› where’s the state being stored?

decouples client from server
› don’t need to reload client app when server changes
› simpler and cleaner code in client
duplicate form submission

problems
› pressing submit twice may cause double billing
› refreshing response may have same effect

disabling submit button with JS
› handles double press, even before response

Post-Redirect-Get (PRG) pattern
› handles refresh resubmission

server provides shopping cart id
› used in payment request
› server stops double payment on single cart
takeaways

API design matters
› bad API = bad service
› clients rely on an API, so hard to change

always
› keep interactions as stateless as possible
› support stable URLs, and define systematically

probably
› choose JSON over XML
› avoid SOAP, WSDL
› use ReSTful URLs for CRUD apps