Any clod can have the facts...

having opinions is an *art*

-- Charles McCabe
Outline

• Big picture
  • API design guidelines
  • AJAX
  • REST
  • API as instance of design problem
Disclaimer

• Today we’re talking about AJAX and REST
• Using them together is a common idiom in current practice
• Nothing ties them together:
  – Can have AJAX programs talking to non-REST APIs
  – Can have non-AJAX programs calling REST APIs
Our story so far

Taking out the browser and talking to the server...

Seems like this should be *easier* than what we’ve done before...

What’s the catch?
Application vs. API

- Meal vs. restaurant
- Song vs. recording studio
- Student vs. university

- Skills with item on left are *necessary* but not *sufficient*
Common API uses

• Alternative UI for existing app
• New app or hybrid app (mashup)
• Indexing / spidering / robotic data collection
Different kinds of program

Program

Functions in real world

Programming Product

Plays well with others

Programming System

Functions in real world

Programming Systems Product

After Brooks, *The Mythical Man-Month*, 1975
Different kinds of program

After Brooks, *The Mythical Man-Month*, 1975
Different kinds of API

Private or Enterprise API

Functions in real world

Public API

Plays well with others

Flexible multi-language API

Functions in real world

Public flexible Multi-language API

Plays well with others

Hypothesis: Cost increases similar to program vs. programming system product
Demo: Google API Explorer
Outline

• Big picture
  • API design guidelines
• AJAX
• REST
• API as instance of design problem
Lots of this is not new

- Modules
- Model vs. implementation
- Interfaces
- Normal vs. Exceptions
- Designing for others
Good vs. bad areas of focus

**Good**
- Solve right problem(s)
- Avoid disaster

**Bad**
- Achieve ultimate performance
- Achieve ultimate expressiveness
Designing for programmers is new

• Security
  – Accidents
  – Attacks

• Narrow vs. wide, Deep vs. shallow
• Filtering / limits
• Versioning

We’re deferring this to next week
An important if disturbing truth

• Arguably the most important elements for API success are:
  – Documentation quality
  – Error message quality
  – Language/format support

• These are not typical “design issues”
  – You can make them worse with a bad design
  – You can’t overcome them with a good design
Aside: Cross-language API tools

• Apache Thrift
• Alpaca

*Handy for improving “real-world” adoption, probably not relevant to our simpler projects*
A narrow deep API

• Do_stuff(string): *Takes a program explaining the computation to be run and runs it. Returns many different kinds of things.*
A wide shallow API

• Set_bit0()
• Clear_bit0()
• Read_bit0() returns(boolean)
• ...
• Set_bitFFFFFFFF()
• Clear_bitFFFFFFFF()
• Read_bitFFFFFFFF() returns(boolean)
Sensible API constraints

• Narrow enough to be comprehensible and usable
• Wide enough to distinguish things that are distinct
• Deep enough so operations are *meaningful*
• Shallow enough so operations are *units*
Filtering/limiting

• How big is an object?
• How big is a collection?
• Neither one is limited by the protocol!
Big stuff

• Allow creation/manipulation of big objects*
• Allow creation/manipulation of big collections*

*assuming it makes sense for your domain!
Small clients and big stuff

• Allow client to limit how much they get
  /dogs/?limit=5

• Have sensible default limit
• Allow client to override limit
  /dogs/?limited=no
Versioning

• How do you support “old” clients when you do a “new” API?

• Won’t happen in 6.170 but crucial in real world

• Goal: never version your API
  – Definitely don’t want every change to be a new version!

• Note: Versioning problem is key issue motivating HATEOAS (later, in REST)
Usual approaches

• Extension is OK
  – Liskov substitution principle!

• Replacing incorrect behavior with correct behavior is OK
  – But need to be sure that no-one depends on incorrect behavior

• Otherwise on dangerous ground

• Fortunately lots of development is extension and correction
Where to put the version number?

- v1.api.example.com/foo/bar
- api.example.com/v1/foo/bar
- example.com/api/v1/foo/bar

- No single right answer for all cases
- Smart to think about it before you need it!
Design criteria

• Versioning
  – When does a function change need to appear as a new API version?

• Extension
  – Do other parties add to the base API?
XML vs. JSON

• JSON easier esp. with JavaScript
  – Parsing/unparsing easy

• XML better for extensibility
  – See examples...
XML vs. JSON (1 of 3)

Adapted from: David Zuelke, “Designing HTTP Interfaces and RESTful Web Services”
XML vs. JSON (2 of 3)

XML

```xml
<?xml version="1.0" encoding="utf-8" ?>
<products xmlns="http://www.acme.com/shop/products">
  <product id="123">
    <name>Bacon</name>
    <price>5.99</price>
    <price currency="EUR">4.49</price>
  </product>
</products>
```

JSON

```
{products: [
  {id: "123", name: "Bacon", price: "5.99"}
]}
```

Adapted from: David Zuelke, “Designing HTTP Interfaces and RESTful Web Services”
XML vs. JSON (3 of 3)

<?xml version="1.0" encoding="utf-8" ?>
<products xmlns="http://www.acme.com/shop/products">
  <product id="123">
    <name xml:lang="en">Bacon</name>
    <name xml:lang="de">Speck</name>
    <price>5.99</price>
  </product>
</products>

{products: [  
  {id: "123", name: "Bacon", price: "5.99"}
]}

Adapted from: David Zuelke, “Designing HTTP Interfaces and RESTful Web Services”
Evil URLs

• /getAccount
• /createDirectory
• /updateGroup
• /verifyAccountEmailAddress

Source: Lee Hazlewood, “Design Beautiful REST + JSON APIs”
Evil URLs continued...

• /getAccount
• /getAllAccounts
• /searchAccounts
• /createDirectory
• /createLDAPDirectory
• /updateGroup
• /updateGroupByName
• /findGroupsByDirectory
• /searchGroupsByName
• /verifyAccountEmailAddress
• /verifyAccountEmailAddressByToken
• ...

Source: Lee Hazlewood, “Design Beautiful REST + JSON APIs”
What’s gone wrong?

• Thinking in terms of operations
  – We already know that data is the more powerful basis for modularity

• Single flat namespace for operation names

• “Smells like bad RPC. Don’t do it”
Bad URLs example

- /product
- /product/filter/cats/desc
- /product/1234
- /photos/product/1234
- /photos/product/1234/new
- /photos/product/1234/5678

One element or a group?
Encoding query in URL
I guess this is a product?
Is this photo 1234 or product 1234?
New photo or new product?
Encoding operation in URL, yuk
What the heck is this?

Adapted from: David Zuelke, “Designing HTTP Interfaces and RESTful Web Services”
Better URLs example

- /products/
- /products/?filter=cats&sort=desc
- /products/1234
- /products/1234/photos/
- /products/1234/photos/?sort=latest
- /products/1234/photos/5678

Adapted from: David Zuelke, “Designing HTTP Interfaces and RESTful Web Services”
# Operations

<table>
<thead>
<tr>
<th>Collection</th>
<th>POST</th>
<th>GET</th>
<th>PUT</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>.../products/</td>
<td>Create new product in collection (return its URI)</td>
<td>List the collection</td>
<td>Bulk update products</td>
<td>Delete the collection</td>
</tr>
<tr>
<td>Element</td>
<td>Error</td>
<td>Show product</td>
<td>If product exists, replace. Otherwise error</td>
<td>Delete product</td>
</tr>
<tr>
<td>.../products/1234</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After Brian Mulloy, “RESTful API design, 2nd edition”
Errors

• Status code for programs
  – Use HTTP error codes
  – Remember that programs typically can’t/won’t discriminate on any other field

• Message for humans
  – Short readable tag
  – Calm suggestion of problem and way(s) to fix
  – More than one field OK
  – URL to documentation is nice
  – Avoid blaming caller!
API design slogans

- Only 2 kinds of URL: collection, element
- Nouns not verbs
- Plurals for collections
- Concrete better than abstract
- Sweep complexity behind “?”
- Anticipate versioning, filtering, limiting

Adapted from Brian Mulloy, “RESTful API design, 2nd edition”
Outline

• Big picture
• API design guidelines
  • AJAX
  • REST
• API as instance of design problem
AJAX

• “Old Web” Multi-page apps
  – Hypertext-ish: click to follow link
  – Not very app-lish
    • Unresponsive, heavy

• Single-page apps
  – Asynchronous JavaScript
  – And XML... actually mostly And JSON, but AJAJ isn’t as catchy
Easy AJAX examples

- Google search box
- Google Maps
Understanding AJAX

• Client side: UI events $\rightarrow$ event loop, callbacks
• Server side: requests $\rightarrow$ event loop, callbacks
• Client calling server: why should that be synchronous?
  – Client calls server $\rightarrow$ async, callback with result
MBTA data returned

• Ask for all routes, get back:
• `{mode: [
    {..., mode_name: string,
    route: [
        {..., route_name: string,
        route_id: string, ...
    },
    ...
    ]
], ...
}`

Need this to make further queries
Demo: AJAX in raw JavaScript

• Can process partial information
  – Useful for performance
• Simple code needs to know when done
## XMLHttpRequest changes state

<table>
<thead>
<tr>
<th>readyState</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UNSENT</td>
</tr>
<tr>
<td>1</td>
<td>OPENED</td>
</tr>
<tr>
<td>2</td>
<td>HEADERS_RECEIVED</td>
</tr>
<tr>
<td>3</td>
<td>LOADING</td>
</tr>
<tr>
<td>4</td>
<td>DONE</td>
</tr>
</tbody>
</table>
Demo: AJAX with JQuery
MBTA API

• Good:
  – We could solve the problem
  – The right info was available

• Bad:
  – Awkward packaging of data
    • Array of modes???
  – Operator/implementor view
    • No real “Red Line” – 3 distinct routes
Google Maps API

• Good:
  – We could solve the problem
  – The right tools were available
  – Lots of sample code, good documentation
  – Well organized

• Sort-of-bad:
  – API is huge... can be daunting
Outline

• Big picture
• API design guidelines
• AJAX
  • REST
• API as instance of design problem
Understanding REST

• REST = REpresentational State Transfer
• Principled approach to Web
  – Tidying up afterward – not as invented
  – “Faking a rational design process”
  – Roy Fielding’s dissertation
• Easy to take for granted
  – “That’s all obvious” – not really
Two layers of REST

• Understanding the “purist position”
  – So you can decide if you care

• Understanding the “pragmatic position”
  – So you can respond sensibly to common questions
Purist position

• Need all REST features to be RESTful
  – Extensibility and long-term evolution maximized
  – Might not be very efficient

• aka “RESTafarian” 😊
Pragmatic position

- RESTful is when it’s more like REST than anything else
- Style follows Web, rather than being some other paradigm with an HTTP transport
Discovery within the familiar

- Fish are probably not aware of the qualities of water
- “For forty years now I’ve been speaking in prose without knowing it!” (Molière)
- Something of the same quality of looking at the architectural style of the Web
Web requirements

• Low entry barrier
• Extensibility
• Distributed hypermedia
• Internet scale
  – Anarchic
  – Independent deployment
Richardson Maturity Model (RMM)

• Level 0: HTTP as transport (only) – single URI
• Level 1: Resources – multi URIs
• Level 2: HTTP verbs
• Level 3: HATEOAS / Hypermedia
Elements

- Replicated repository (RR)
- Cache ($)
- Client/server (CS)
- Layered system (LS)
- Stateless (S)
- Virtual machine (VM)
- Code on Demand (COD)
- Uniform interface (U)
Uniform interface?

- Identification of resources
- Manipulation of resources through representations
- Self-describing messages
- Hypertext as the engine of application state (HATEOAS)
Client/Server

Benefit: Separation of concerns
Client-Stateless-Server

Benefit: Scale
Client-Cache-Stateless-Server

Benefit: Scale, performance
Constraints at this point

- Stateless: each request must contain all information
- Stateless: Server does not maintain context
- Cache: Client is free to reuse responses
- This was basically the model of the early Web
Uniform-Client-Cache-Stateless-Server

Figure 5-6. Uniform-Client-Cache-Stateless-Server

Benefit: Separation of concerns
Why Uniformity?

• Decouple implementation from services provided
• Reduces efficiency but improves evolvability
Uniform-Layered-Client-Cache-Stateless-Server

Benefit: Separation of concerns, scale
Why layers?

• Each layer provides some kind of service
• Layer cannot interact beyond its immediately adjacent layers
• Each layer can encapsulate special knowledge or legacy behavior
REST

Benefit: Separation of concerns

Figure 5-8. REST
How & why it all comes together

REST core

• Key distinction between **resource** and **representation**
  – Resource is at server, responds to methods
  – Representations transmitted over network, self-describing (media-type header)

• Resources are the distributed virtual machines that do interesting things

• Representations are values that are transmitted

• No client ever “has” a resource
Hypertext as the engine...

- Link includes URI
- URI names resource
- Representation contains links
- Resource supports verbs (interaction)
Web UI vs. Web API

• You all understand how to use HATEOAS in a web-browsing context
  — Even though you’ve probably never called it that
• Why doesn’t the same paradigm apply to APIs?
  — Purists (RESTafarians) say it does!
Hypermedia APIs

• HATEOAS: Hypertext As The Engine Of Application State
• Programs use APIs “like humans do”
• Navigate from root, discover API function
• Win: Flexibility
• Lose: Chatty (mitigate with caching?)
Hypertext as the engine...

Link includes URI

URI names resource

Representation contains links

Resource supports verbs (interaction)
Outline

• Big picture
• API design guidelines
• AJAX
• REST
  • API as instance of design problem
<table>
<thead>
<tr>
<th></th>
<th>Functionality</th>
<th>Speed</th>
<th>Fault-tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Completeness</strong></td>
<td>Normal vs. worst case</td>
<td>Safety first Shed load End-to-end</td>
<td>End-to-end</td>
</tr>
<tr>
<td><strong>Interface</strong></td>
<td>Do one thing well</td>
<td>Make it fast Split resources Static analysis Dynamic translation</td>
<td>End-to-end Log updates Atomic actions</td>
</tr>
<tr>
<td></td>
<td>Don’t generalize</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Get it right</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Don’t hide power</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Leave it to the client</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Keep basic interfaces stable</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>Keep secrets</td>
<td>Cache answers Use hints Use brute force Compute in background Batch processing</td>
<td>Atomic actions Use hints</td>
</tr>
<tr>
<td></td>
<td>Use a good idea again</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Divide and conquer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Design concepts

• Hide secrets
  – Parnas “On the Criteria to be Used in Decomposing Systems into Modules.” 1971

• Allow for expansion/contraction
  – Parnas “Designing Software for Ease of Extension and Contraction.” 1979

• Tell the story