

# **GET Connected:**

## **A Companion Tutorial on Web-based Services**

ICSOC Tutorial - 6 December 2008

Dr. Jim Webber  
<http://jim.webber.name>

Dr. Halvard Skogsrud  
[halvard@skogsrud.com](mailto:halvard@skogsrud.com)

## Roadmap

- Introduction and Motivation
- The Web Architecture
- Simple Web Integration
- CRUD Services
- Hypermedia
- Scalability
- ATOM and ATOMPUB
- Security
- Conclusions and further thoughts

## Introduction

- This is a tutorial about the Web
- It's very HTTP centric
- But it's not about Web pages!
- The Web is a middleware platform which is...
  - Globally deployed
  - Has reach
  - Is mature
  - And is a reality in every part of our lives
- Which makes it interesting for distributed systems geeks

## Motivation

- This follows the plot from a book called “GET /Connected” which is currently being written by:
  - Jim Webber
  - Savas Parastatidis
  - Ian Robinson
- The book deals with the Web as a distributed computing platform
  - The Web as a whole, not just REST
- And so does this tutorial...

A brief history of the Web

# **BACKGROUND**

## Why the Web was Inevitable

Tim Berners-Lee is a physicist



(Sir Tim is also a knight, but that's not important right now)



## Why the Web was Inevitable

He lived in a hole in the ground



Underneath a big mountain  
(in Switzerland)

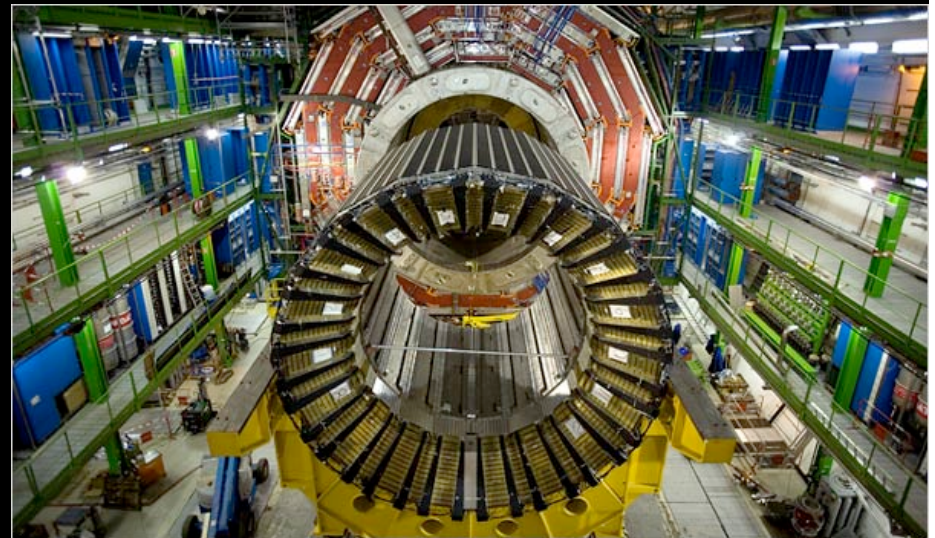


## Why the Web was Inevitable



And because he was a physicist (and not yet a knight)...

...he only had a big atom-smashing thing for company





## Why the Web was Inevitable



And for a lonesome physicist stuck underground with smashed up atoms for company...

...gopher just wasn't going to cut it!

```

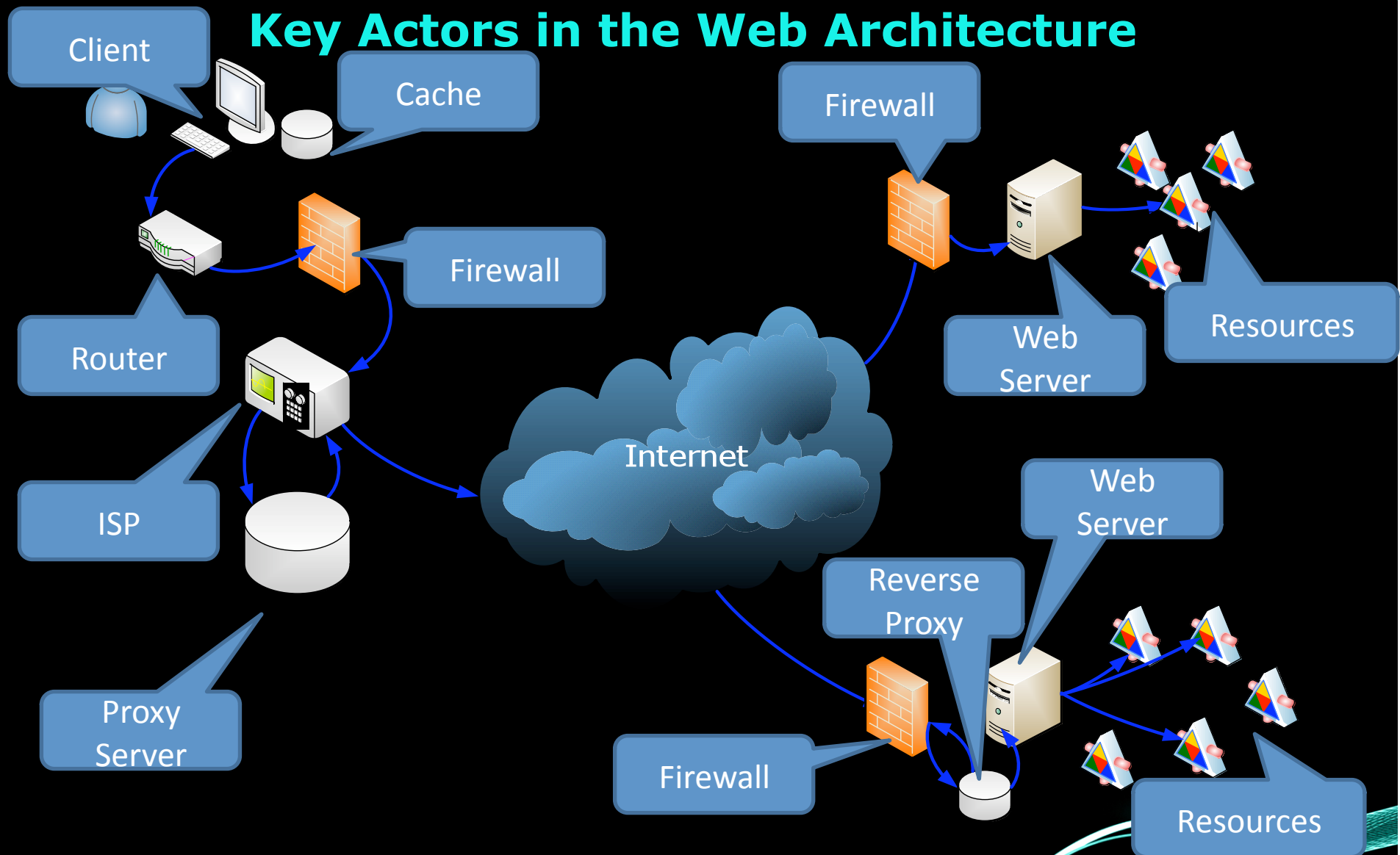
wsgopher 1.2 - Reorganization of Information Technology Support at un
File Edit Bookmark Configure Window Help
href="http://geocities.com/dava183230duz/?a=wjRYlwIgmALfsGr
<FONT STYLE=font-size:4px font-family:Courier font-weight:b
cpre>
hgw7      5ai8      pu5877ogqu    55919z69ykq705mw
va6i      2cp0      ca4s      44k5      p26r
1966      88w4      g4q8      d99g      hx3g
wda1      070y      713q      8332      2o8b
gq703h1i95k31504    hooi      sqf8      c7na
2th5      5468      321v      e14t      8u55
7o48      3uy8      f5sf      9a8c      7so2
kz14      c456      w5d9      3z17      kuy4
f4ab      566x      ocz4z44rf7    652q

k2g3p7sicx    o8nt2f7x    xvy9d3yd5z6    0cne
fc68      fs      vl93      g616      wk60      4ncm
3h2      phaw      ya1u      n6fp      22v8
xqf      4a32      ad41      d350      y586
a5z      wz8      85f64g0d698h    54v5
h84      s1r368xe    vr5a      y7g0      ge7h      72a4
gg0      cwfo      i479      7t51      5a02      40ow
noqp      75gl      719p      4a59      cqqw      9167
c442z151bgbe    56b512w4    3085      wqoq      512219cn87
  
```

## Web Fundamentals

- To embrace the Web, we need to understand how it works
- The Web is a distributed hypermedia model
  - It doesn't try to hide that distribution from you!
- Our challenge:
  - Figure out the mapping between our problem domain and the underlying Web platform

## Key Actors in the Web Architecture



## Web Characteristics

- Scalable
  - Fault-tolerant
  - Recoverable
  - Secure
  - Loosely coupled
- 
- Precisely the same characteristics we want in business software systems!



## Scalability

- Web is truly Internet-scale
  - Loose coupling
    - Growth of the Web in one place is not impacted by changes in other places
  - Uniform interface
    - HTTP defines a standard interface for all actors on the Web
    - Replication and caching is baked into this model
      - Caches have the same interface as real resources!
  - Stateless model
    - Supports horizontal scaling

## Fault Tolerant

- The Web uses a stateless model
  - All information required to process a request must be present in that request
    - Sessions are still available, but must be handled in a Web-consistent manner
- Statelessness also means easy replication
  - One Web server is replaceable with another
  - Easy fail-over, horizontal scaling

## Recoverable

- The Web places emphasis on repeatable information retrieval
  - GET is idempotent
  - In failure cases, can safely repeat GET on resources
- HTTP verbs plus rich error handling help to remove guesswork from recovery
  - HTTP statuses tell you what happened!

## Secure

- HTTPs is a mature technology
  - Based on SSL for secure point-to-point information retrieval
- Isn't sympathetic to Web architecture
  - Can't cache!
- As we shall see, higher-order protocols like Atom are starting to change this...



## Loosely Coupled

- Adding a Web site to the WWW does not affect any other existing sites
- All Web actors support the same, uniform interface
  - Easy to plumb new actors into the big wide web
    - Caches, proxies, servers, resources, etc

Web != REST...

## **SIMPLE WEB INTEGRATION**

## Web Tunnelling

- Web Services tunnel SOAP over HTTP
  - Using the Web as a transport only
  - Ignoring many of the features for robustness the Web has built in
- Many Web people do the same!
  - URI tunnelling, POX approaches are the most popular styles on today's Web
  - Worse than SOAP!
    - Less metadata!

But they claim to be  
“lightweight” and  
RESTful

## URI Tunnelling Pattern

- Web servers understand URIs
- URIs have structure
- Methods have signatures
- Can match URI structure to method signature
- E.g.

– `http://example.com/addNumbers?p1=10&p2=11`

– `int addNumbers(int i, int j) { return i + j; }`



## URI Tunnelling Strengths

- Very easy to understand
- Great for simple procedure-calls
- Simple to code
  - Do it with the servlet API, HttpListener, IHttpHandler, RAILS, whatever!
- Interoperable
  - It's just URIs!

## URI Tunnelling Weaknesses

- It's brittle RPC!
- Tight coupling, no metadata
  - No typing or "return values" specified in the URI
- Not robust – have to handle failure cases manually
- No metadata support
  - Construct the URIs yourself, map them to the function manually
- You typically use GET (prefer POST)
  - OK for functions, but against the Web for procedures with side-effects

## POX Pattern

- Web servers understand how to process requests with bodies
  - Because they understand forms
- And how to respond with a body
  - Because that's how the Web works
- POX uses XML in the HTTP request and response to move a call stack between client and server

## POX Strengths

- Simplicity – just use HTTP POST and XML
- Re-use existing infrastructure and libraries
- Interoperable
  - It's just XML and HTTP
- Can use complex data structures
  - By encoding them in XML

## POX Weaknesses

- Client and server must collude on XML payload
  - Tightly coupled approach
- No metadata support
  - Unless you're using a POX toolkit that supports WSDL with HTTP binding (like WCF)
- Does not use Web for robustness
- Does not use SOAP + WS-\* for robustness



## Web Abuse

- Both POX and URI Tunnelling fail to take advantage of the Web
  - Ignoring status codes
  - Reduced scope for caching
  - No metadata
  - Manual crash recovery/compensation leading to high development cost
  - Etc
- They're useful in some situations
  - But they're not especially robust patterns

Moving on up...

## **CRUD WEB SERVICES**

## Using the Web

- URI tunnelling and POX use the Web as a transport
  - Just like SOAP without metadata support
- CRUD services begin to add coordination support
- But the Web is more than just a transport
  - Transport, plus
  - Metadata, plus
  - Fault model, plus
  - Component model, plus
  - Runtime environment, plus...

Status Codes

Uniform

Caches, proxies,  
servers, etc

## Resources

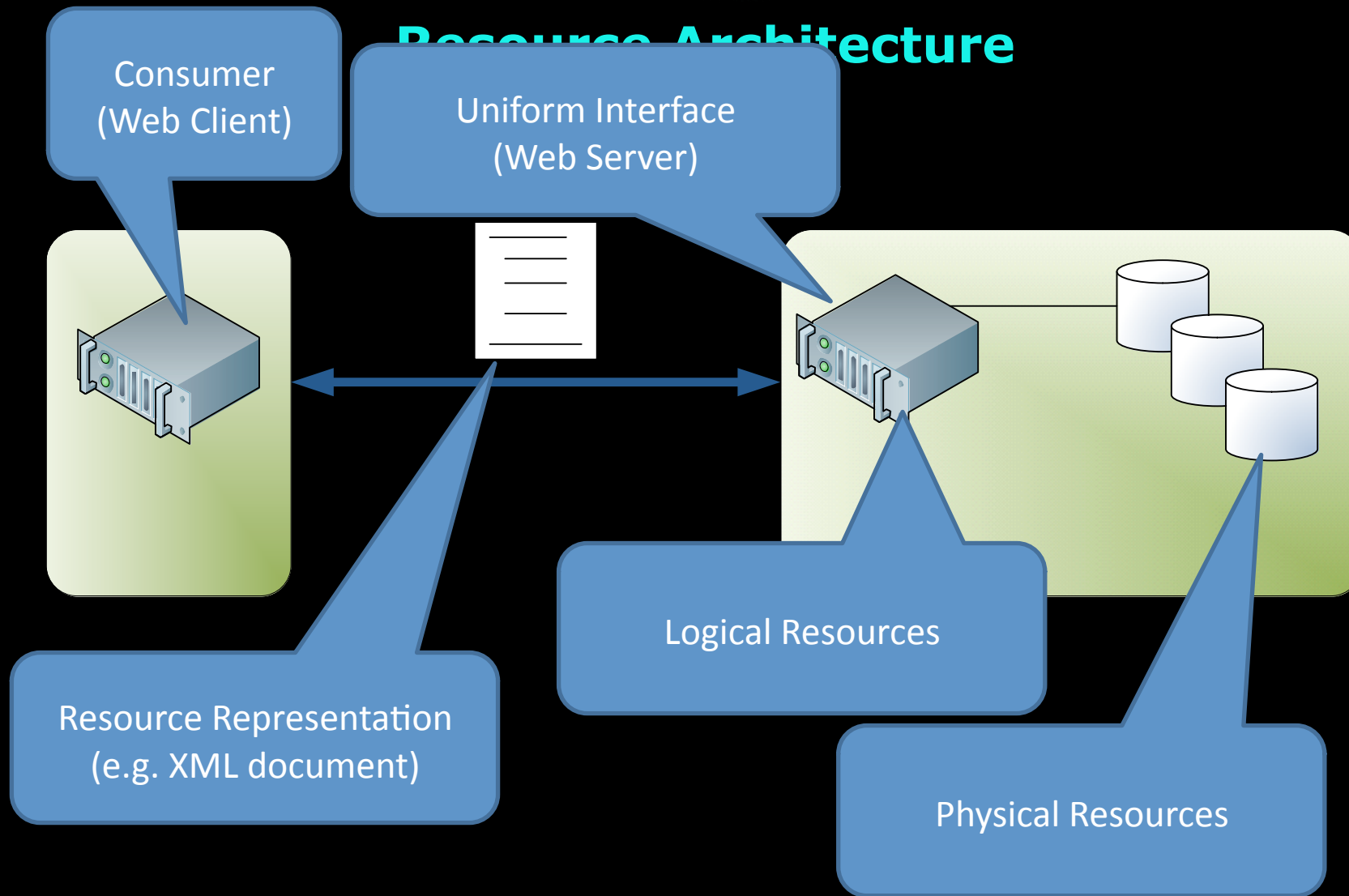
- A resource is something “interesting” in your system
- Can be anything
  - Spreadsheet (or one of its cells)
  - Blog posting
  - Printer
  - Winning lottery numbers
  - A transaction
  - Others?

## Interacting with Resources

- We deal with representations of resources
  - Not the resources themselves
    - Pass-by-value: data in message bodies
    - Pass-by-reference: exchange URIs
  - Representation can be in any format
    - Any media type
- Each resource implements a standard uniform interface
  - The HTTP interface
- Resources have names and addresses (URIs)
  - HTTP URIs (aka URLs)



## Resource Architecture



## Resource Representations

- Making your system Web-friendly increases its surface area
  - You expose many resources, rather than fewer endpoints
- Each resource has one or more representations
  - Representations like JSON or XML good for the programmatic Web
- Moving representations across the network is the way we transact work in a Web-native system

## URIs

- URIs are addresses of resources in Web-based systems
  - Each resource has at least one URI
- They identify “interesting” things
  - i.e. Resources 😊
- Any resource implements the same (uniform) interface
  - Which means we can access it programmatically!

## CRUD Resource Lifecycle

- The resource is created with POST
- It's read with GET
- And updated via PUT
- Finally it's removed using DELETE

## POST Semantics

- POST creates a new resource
- But the server decides on that resource's URI
- Common human Web example: posting to Web log
  - Server decides URI of posting and any comments made on that post
- Programmatic Web example: creating a new employee record
  - And subsequently adding to it



## POST Request

POST / HTTP/1.1

Verb, path, and HTTP  
version

Content-Type: text/xml

Host: localhost:8888

Content type (XML)

Content-Length: ....

Connection: Keep-Alive

<buy>

<symbol>ABCD</symbol>

<price>27.39</price>

Content (again XML)

</buy>

## POST Response

201 CREATED

Location: /orders/halvards/ABCD/2007-07-080-13:50:53

## When POST goes wrong

- We may get 4xx or 5xx errors
  - Client versus server problem
- We turn to GET!
- Find out the resource states first
  - Then figure out how to make forward or backward progress
- Then solve the problem
  - May involve POSTing again
  - May involve a PUT to rectify server-side resources in-place

## Safety and Idempotency

- A safe operation is one which changes no state at all
- An idempotent operation is one which updates state in an absolute way, and the result of executing the operation more than once is the same as executing it once
  - E.g.  $x = 4$  rather than  $x += 2$
- Web-friendly systems scale because of safety
  - Caching!
- And are fault tolerant because of idempotent behaviour
  - Just re-try in failure cases

## GET Semantics

- GET retrieves the representation of a resource
- Should be safe
  - Shared understanding of GET semantics
  - Don't violate that understanding!



## GET Exemplified

```
GET /employees?id=1234 HTTP/1.1
```

```
Accept: text/xml
```

```
Host: crm.example.com
```

## When GET Goes wrong

- Simple!
  - Just 404 – the resource is no longer available
- Are you sure?
  - GET again!
- GET is safe (and hence also idempotent)
  - Great for crash recovery scenarios!

## PUT Semantics

- PUT creates a new resource but the client decides on the URI
  - Providing the server logic allows it
- Also used to update existing resources by overwriting them in-place

Why not POST?

Not idempotent!

## PUT Request

```
PUT /orders/halvards/ABCD/2007-07-08 13:50:53 HTTP/1.1  
Content-Type: text/xml  
Host: localhost:8888  
Content-Length: ....  
Connection: Keep-Alive
```

Verb, path and HTTP version

```
<buy>  
  <symbol>ABCD</symbol>  
  <price>27.44</price>  
</buy>
```

Updated content  
(higher buy price)

## PUT Response

200 OK

Location: /orders/halvards/ABCD/2007-07-080-13:50:53

Content-Type: text/xml

<nyse:priceUpdated .../>

Minimalist response might contain  
only status and location

## When PUT goes wrong

- If we get 5xx error, or some 4xx errors simply PUT again!
  - PUT is idempotent
- If we get errors indicating incompatible states (409, 417) then do some forward/backward compensating work
  - And maybe PUT again



## DELETE Semantics

- Stop the resource from being accessible
  - Logical delete, not necessarily physical

- Request

```
DELETE /user/halvards HTTP 1.1
Host: example.org
```

- Response

```
200 OK
Content-Type: application/xml
<admin:userDeleted>
  halvards
</admin:userDeleted>
```

This is important for  
decoupling  
implementation details  
from resources

## When DELETE goes wrong

- DELETE again!
  - Delete is idempotent!
  - DELETE once, DELETE 10 times has the same effect: one deletion
- Some 4xx responses indicate that deletion isn't possible
  - The state of the resource isn't compatible
  - Try forward/backward compensation instead

## CRUD is Good?

- CRUD is good
  - But it's not great
- CRUD-style services use some HTTP features
- But the application model is limited
  - Suits database-style applications
- CRUD has limitations
  - CRUD ignores hypermedia
  - CRUD encourages tight coupling through URI templates
  - CRUD encourages server and client to collude
- The Web supports more sophisticated patterns than CRUD!

A little detour...

## **SEMANTICS**

## Microformats

- Microformats are an example of little “s” semantics
- Innovation at the edges of the Web
  - Not by some central design authority (e.g. W3C)
- Started by embedding machine-processable elements in Web pages
  - E.g. Calendar information, contact information, etc
  - Using existing HTML features like `class`, `rel`, etc

## Microformats and Resources

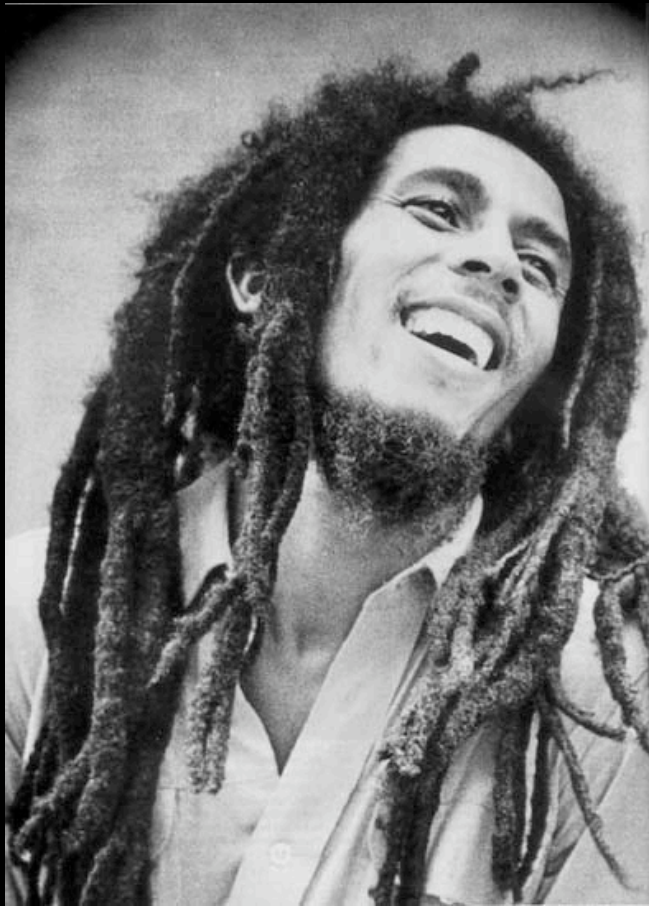
- Use Microformats to structure resources where formats exist
  - I.e. Use hCard for contacts, hCalendar for data
- Create your own formats (sparingly) in other places
  - Annotating links is a good start
  - `<link rel="withdraw.cash" .../>`
  - `<link rel="service.post" type="application/atom+xml" href="{post-uri}" title="some title">`
- The `rel` attribute describes the semantics of the referred resource



The HATEOAS Constraint...

**HYPERMEDIA**

## RESTafarians?



## HEAD Semantics

- HEAD is like GET, except it only retrieves metadata
- Request

```
HEAD /user/halvard HTTP 1.1  
Host: example.org
```

- Response

```
200 OK  
Content-Type: application/xml  
Last-Modified: 2007-07-08T15:00:34Z  
Etag: aabd653b-65d0-74da-bc63-4bca-ba3ef3f50432
```

Useful for caching,  
performance

## OPTIONS Semantics

- Asks which methods are supported by a resource
  - Easy to spot read-only resources for example

- Request

```
OPTIONS /user/halvard HTTP 1.1  
Host: example.org
```

- Response

```
200 OK  
Allowed: GET, HEAD, POST
```

You can only read and add to  
this resource



## Conditional GET Pattern

- Bandwidth-saving pattern
- Requires client and server to work together
- Server sends `Last-Modified` and/or `ETag` headers with representations
- Client sends back those values when it interacts with resource in `If-Modified-Since` and/or `If-None-Match` headers
- Server responds with a 200 an empty body if there have been no updates to that resource state
- Or gives a new resource representation (with new `Last-Modified` and/or `ETag` headers)

## HTTP Headers

- Headers provide metadata to assist processing
  - Identify resource representation format (media type), length of payload, supported verbs, etc
- HTTP defines a wealth of these
  - And like status codes they are our building blocks for robust service implementations



## Must-know Headers

- Authorization
  - Contains credentials (basic, digest, WSSE, etc)
  - Extensible
- Etag/If-None-Match
  - Opaque identifier – think “checksum” for resource representations
  - Used for conditional GET
- Content-Type
  - The resource representation form
    - E.g. application/xml, application/xhtml+xml

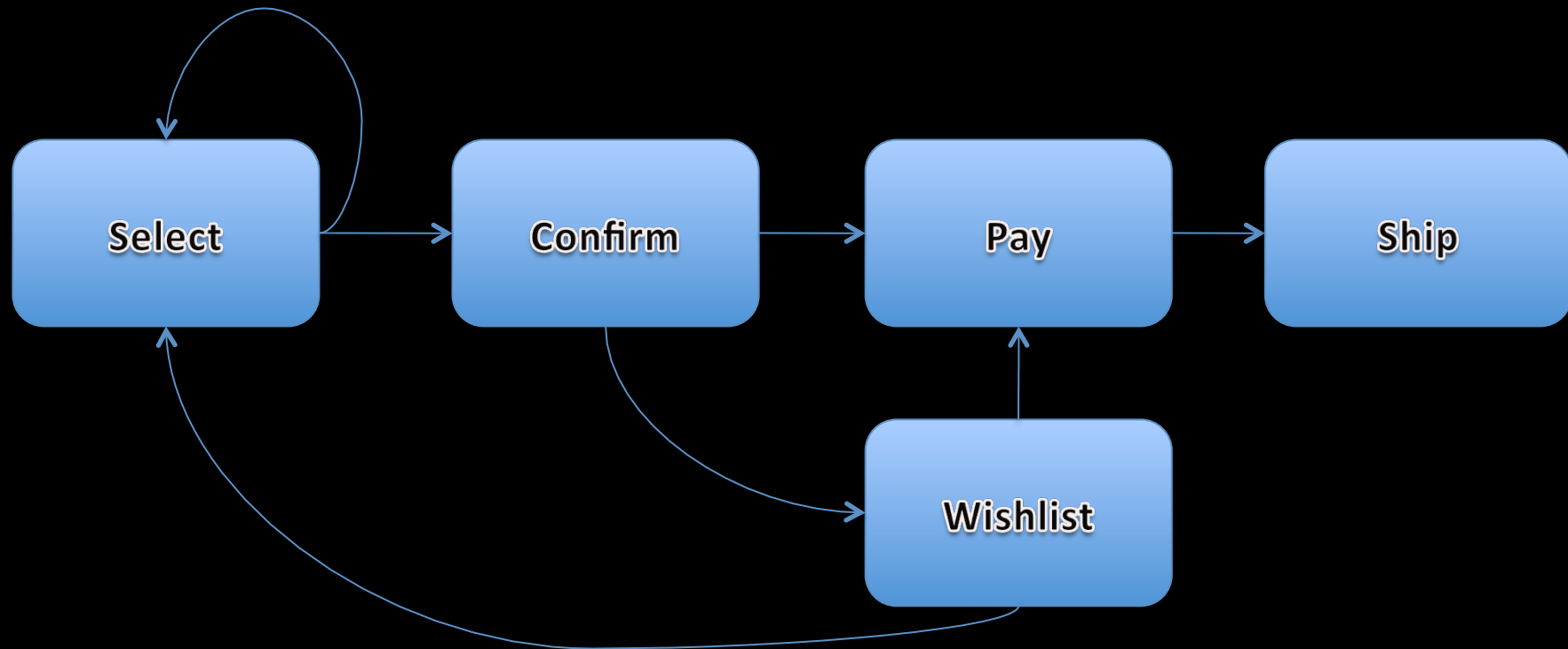
## Yet More Must-Know Headers

- If-Modified-Since/Last-Modified
  - Used for conditional GET too
- Location
  - Used to flag the location of a created/moved resource
  - In combination with:
    - 201 Created, 301 Moved Permanently, 302 Found, 307 Temporary Redirect, 300 Multiple Choices, 303 See Other
- WWW-Authenticate
  - Used with 401 status
  - Informs client what authentication is needed

## Describing Contracts with Links

- The value of the Web is its “linked-ness”
  - Links on a Web page constitute a contract for page traversals
- The same is true of the programmatic Web
- Resource representations can contain other URIs
- Links act as state transitions
- Application (conversation) state is captured in terms of these states
- Use Links to describe state transitions in programmatic Web services
  - By navigating resources you change application state

## Links are State Transitions

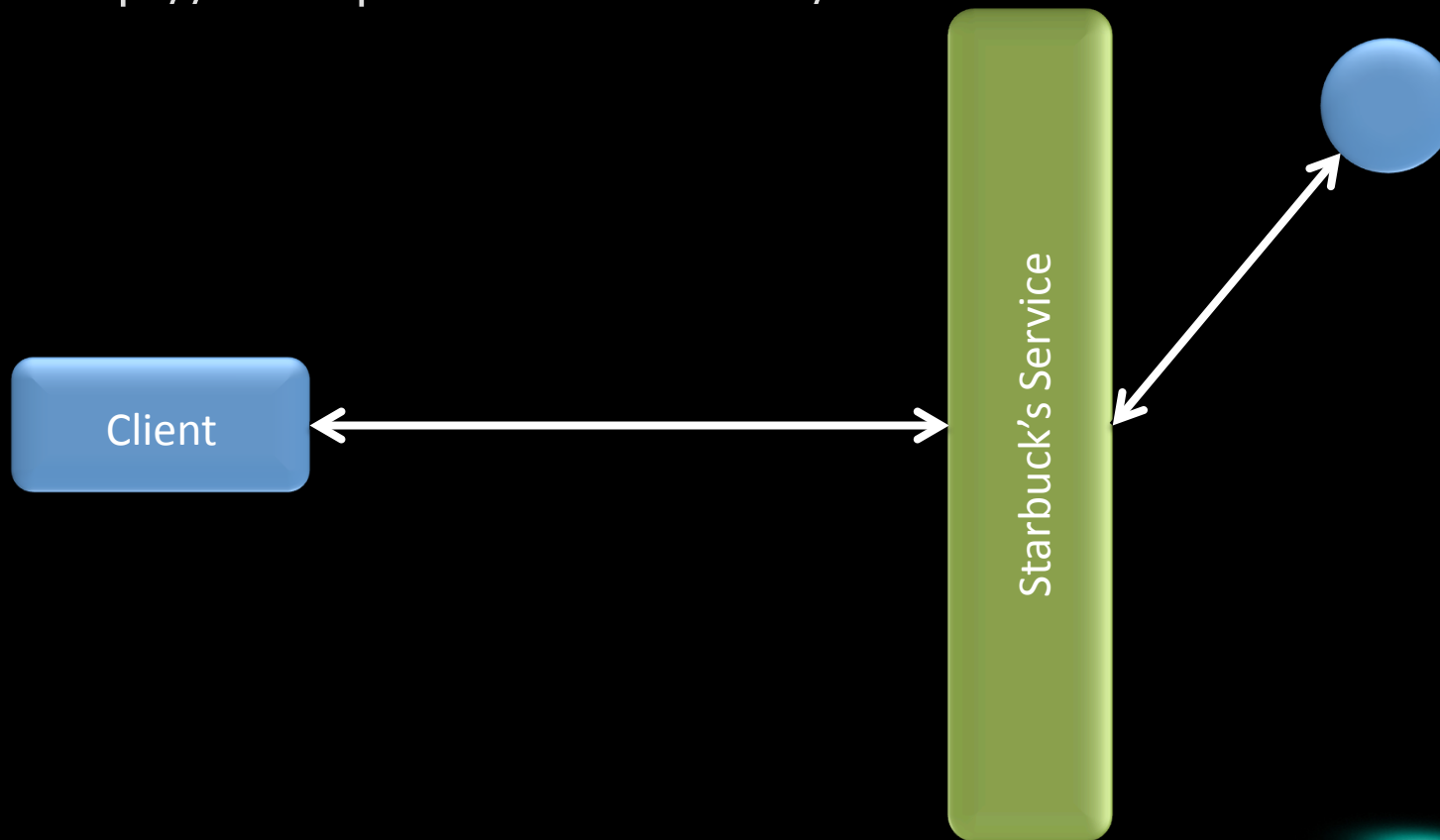


## Workflow

- How does a typical enterprise workflow look when it's implemented in a Web-friendly way?
- Let's take Starbucks as an example, the happy path is:
  - Make selection
    - Add any specialities
  - Pay
  - Wait for a while
  - Collect drink

## Placing an Order

- Place your order by POSTing it to a well-known URI
  - `http://example.starbucks.com/order`





## Placing an Order: On the Wire

- Request

```
POST /order HTTP 1.1
Host: starbucks.example.org
Content-Type: application/xml
Content-Length: ...
```

```
<order xmlns="urn:starbucks">
<drink>latte</drink>
</order>
```

A link! Is this the start  
of an API?

- Response

```
201 Created
Location: http://
    starbucks.example.org/order?
    1234
Content-Type: application/xml
Content-Length: ...
```

```
<order xmlns="urn:starbucks">
<drink>latte</drink>
<link rel="payment"
    href="https://
    starbucks.example.org/
    payment/order?1234"
    type="application/xml"/>
</order>
```

## Whoops! A mistake

- I like my coffee to taste like coffee!
- I need another shot of espresso
  - What are my OPTIONS?

### ⦿ Request

```
OPTIONS /order?1234 HTTP 1.1
```

```
Host: starbucks.example.org
```

### ⦿ Response

```
200 OK
```

```
Allow: GET, PUT
```

Phew! I can  
update my order,  
for now

## Optional: Look Before You Leap

- See if the resource has changed since you submitted your order
  - If you're fast your drink hasn't been prepared yet

### ⦿ Request

```
PUT /order?1234 HTTP 1.1  
Host: starbucks.example.org  
Expect: 100-Continue
```

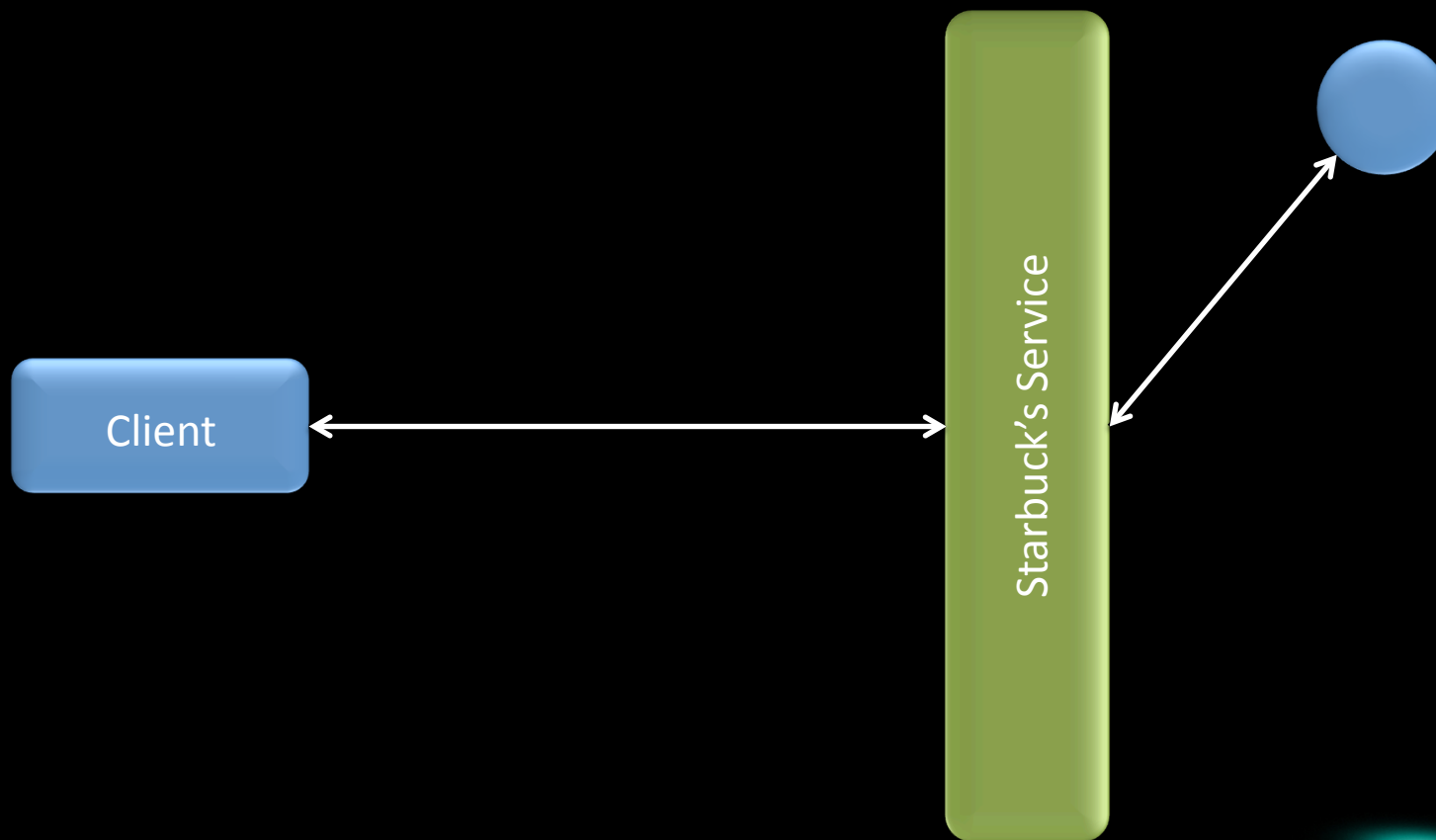
### ⦿ Response

```
100 Continue
```

I can still PUT this resource, for now.  
(417 Expectation Failed otherwise)

## Amending an Order

- Add specialities to you order via PUT
  - Starbucks needs 2 shots!



## Amending an Order: On the Wire

- Request

```
PUT /order?1234 HTTP 1.1
Host: starbucks.example.org
Content-Type: application/xml
Content-Length: ...

<order xmlns="urn:starbucks">
  <drink>latte</drink>
  <additions>shot</additions>
  <link rel="payment"
    href="https://
      starbucks.example.org/payment/
      order?1234"
    type="application/xml"/>
</order>
```

- Response

```
200 OK
Location: http://
  starbucks.example.org/order?
  1234
Content-Type: application/xml
Content-Length: ...

<order xmlns="urn:starbucks">
  <drink>latte</drink>
  <additions>shot</additions>
  <link rel="payment"
    href="https://
      starbucks.example.org/payment/
      order?1234"
    type="application/xml"/>
</order>
```

## Statelessness

- Remember interactions with resources are stateless
- The resource “forgets” about you while you’re not directly interacting with it
- Which means race conditions are possible
- Use `If-Unmodified-Since` on a timestamp to make sure
  - Or use `If-Match` and an ETag
- You’ll get a `412 PreconditionFailed` if you lost the race
  - But you’ll avoid potentially putting the resource into some inconsistent state



## Warning: Don't be Slow!

- Can only make changes until someone actually makes your drink
  - You're safe if you use `If-Unmodified-Since` or `If-Match`
  - But resource state can change without you!

### Request

```
PUT /order?1234 HTTP 1.1
Host: starbucks.example.org
...
```

### Request

```
OPTIONS /order?1234 HTTP 1.1
Host: starbucks.example.org
```

### Response

409 Conflict

Too slow! Someone else has  
changed the state of my order

### Response

Allow: GET

## Order Confirmation: On the Wire

- Request

```
GET /order?1234 HTTP 1.1
Host: starbucks.example.org
Content-Type: application/xml
Content-Length: ...
```

- Response

```
200 OK
Location: http://
          starbucks.example.org/order?
          1234
Content-Type: application/xml
Content-Length: ...
```

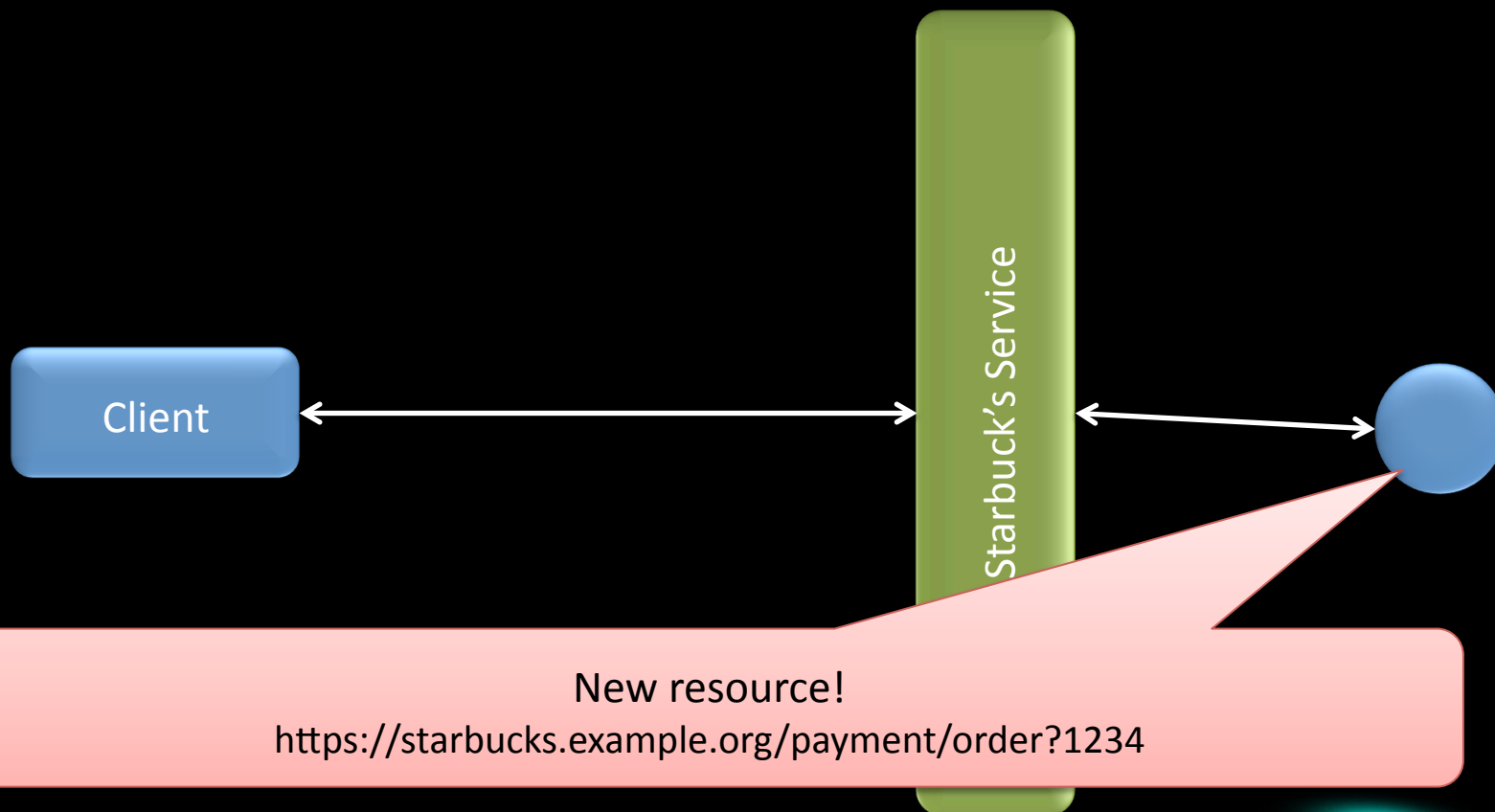
```
<order xmlns="urn:starbucks">
  <drink>latte</drink>
  <additions>shot</additions>
  <link rel="payment" href="https://
    starbucks.example.org/payment/
    order?1234"
    type="application/xml"/>
</order>
```

Are they trying to tell me something with hypermedia?

## Order Payment

- PUT your payment to the order resource

`https://starbucks.example.org/payment/order?1234`



## How did I know to PUT?

- The client knew the URI to PUT to from the link
  - PUT is also idempotent (can safely re-try) in case of failure
- Verified with OPTIONS
  - Just in case you were in any doubt 😊

### ⦿ Request

```
OPTIONS /payment/order?1234 HTTP 1.1
```

```
Host: starbucks.example.org
```

### ⦿ Response

```
Allow: GET, PUT
```

## Order Payment: On the Wire

- Request

```
PUT /payment/order?1234 HTTP 1.1
Host: starbucks.example.org
Content-Type: application/xml
Content-Length: ...
```

```
<payment xmlns="urn:starbucks">
<cardNo>123456789</cardNo>
<expires>07/07</expires>
<name>John Citizen</name>
<amount>4.00</amount>
</payment>
```

- Response

```
201 Created
Location: https://
    starbucks.example.org/
    payment/order?1234
Content-Type: application/xml
Content-Length: ...
```

```
<payment xmlns="urn:starbucks">
<cardNo>123456789</cardNo>
<expires>07/07</expires>
<name>John Citizen</name>
<amount>4.00</amount>
</payment>
```

## Check that you've paid

- Request

```
GET /order?1234 HTTP 1.1  
Host: starbucks.example.org  
Content-Type: application/xml  
Content-Length: ...
```

My "API" has changed,  
because I've paid enough  
now

- Response

```
200 OK  
Content-Type: application/xml  
Content-Length: ...
```

```
<order xmlns="urn:starbucks">  
  <drink>latte</drink>  
  <additions>shot</additions>  
</order>
```



## Finally drink your coffee...



Source: [http://images.businessweek.com/ss/06/07/top\\_brands/image/starbucks.jpg](http://images.businessweek.com/ss/06/07/top_brands/image/starbucks.jpg)

## What did we learn from Starbucks?

- HTTP has a header/status combination for every occasion
- APIs are expressed in terms of links, and links are great!
- XML is fine, but we could also use formats like Atom, JSON or even default to XHTML as a sensible middle ground
- Form encoding still works
  - application/x-www-form-urlencoded media type
- State machines (defined by links) are important
  - Just as in Web Services...

How on Earth does a text-based, synchronous, client-server protocol scale?

**SCALABILITY**

## Statelessness

- Every action happens in isolation
  - This is a good thing!
- In between requests the server knows nothing about you
  - Excepting any state changes you caused when you last interacted with it.
- Keeps the interaction protocol simpler
  - Makes recovery, scalability, failover much simpler too
  - Avoid cookies!



## Application vs Resource State

- Useful services hold persistent data – Resource state
  - Resources are buckets of state
  - What use is Google without state?
- Brittle implementations have application state
  - They support long-lived conversations
  - No failure isolation
  - Poor crash recovery
  - Hard to scale, hard to do fail-over fault tolerance
- Recall stateless Web Services – same applies in the Web too!

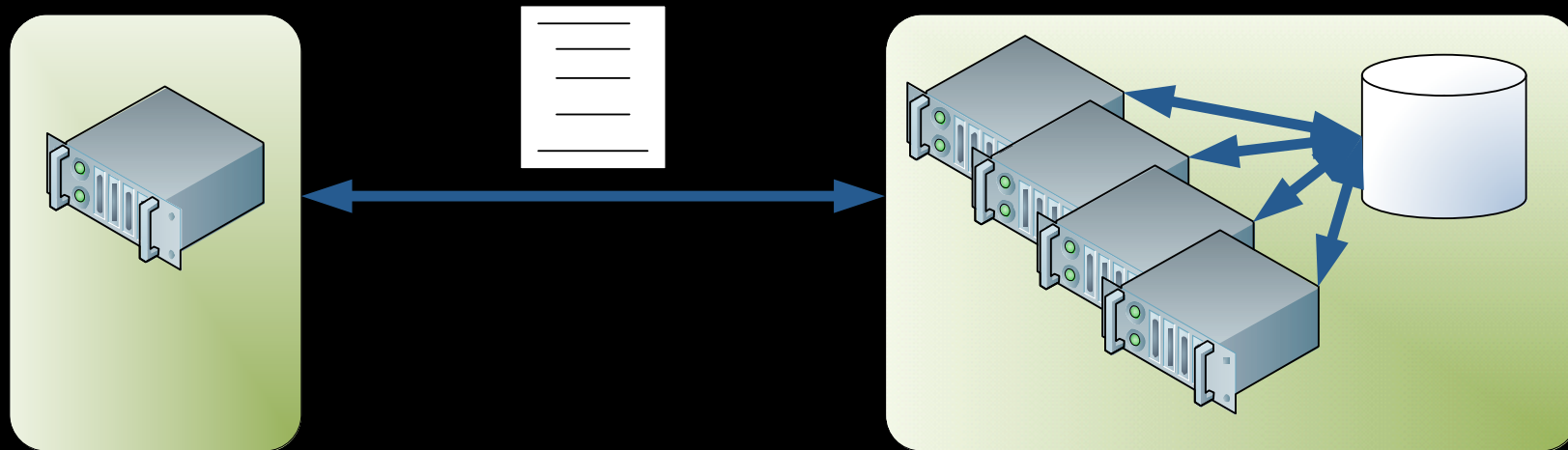
## Scaling Horizontally

- Web farms have delivered horizontal scaling for years
  - Though they sometimes do clever things with session affinity to support cookie-based sessions
- In the programmatic Web, statelessness enables scalability
  - Just like in the Web Services world



## Scalable Deployment Configuration

- Deploy services onto many servers
- Services are stateless
  - No sessions!
- Servers share only back-end data

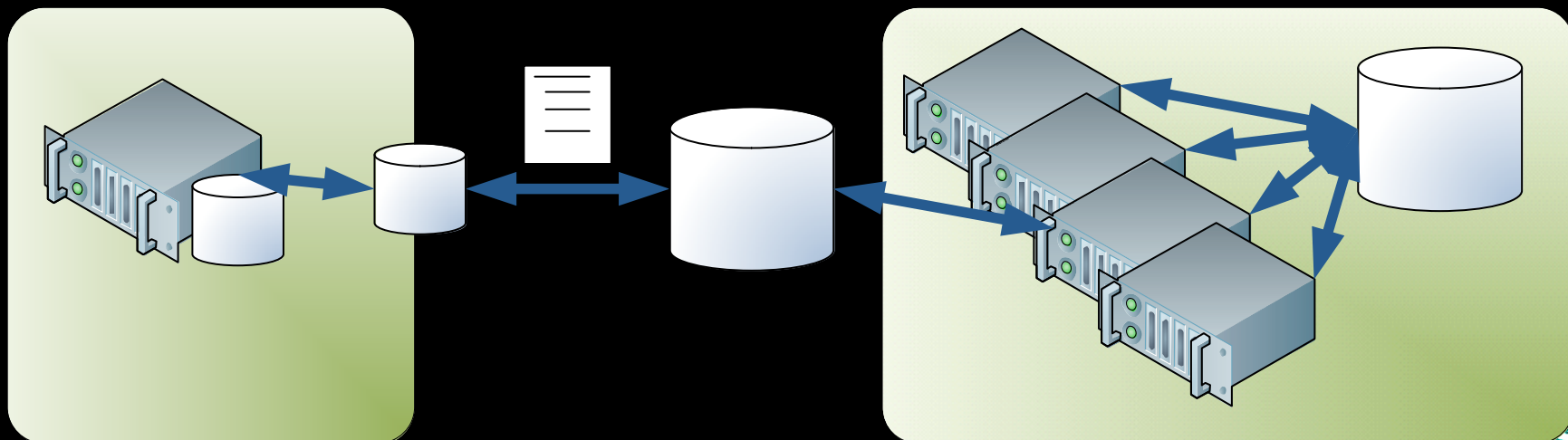


## Scaling Vertically... without servers

- The most expensive round-trip:
  - From client
  - Across network
  - Through servers
  - Across network again
  - To database
  - And all the way back!
- The Web tries to short-circuit this
  - By determining early if there is any actual work to do!
  - And by caching

## Caching in a Scalable Deployment

- Cache (reverse proxy) in front of server farm
  - Avoid hitting the server
- Proxy at client domain
  - Avoid leaving the LAN
- Local cache with client
  - Avoid using the network



## Being workshy is a good thing!

- Provide guard clauses in requests so that servers can determine easily if there's any work to be done
  - Caches too
- Use headers:
  - If-Modified-Since
  - If-None-Match
  - And friends
- Web infrastructure uses these to determine if its worth performing the request
  - And often it isn't
  - So an existing representation can be returned

## Retrieving a Resource Representation

- Request

```
GET /transactions/debit/1234 HTTP 1.1
Host: bank.example.org
Accept: application/xml
If-Modified-Since: 2007-07-08T15:00:34Z
If-None-Match: aabd653b-65d0-74da-bc63-4bca-ba3ef3f50432
```

- Response

```
200 OK
Content-Type: application/xml
Content-Length: ...
Last-Modified: 2007-07-08T15:10:32Z
Etag: abbb4828-93ba-567b-6a33-33d374bcad39
<t:debit xmlns:t="http://bank.example.com">
  <t:sourceAccount>12345678</t:sourceAccount>
  <t:destAccount>987654321</t:destAccount>
  <t:amount>299.00</t:amount>
  <t:currency>GBP</t:currency>
</t:debit>
```

## Not Retrieving a Resource Representation

- Request

```
GET /transactions/debit/1234 HTTP 1.1
Host: bank.example.org
Accept: application/xml
If-Modified-Since: 2007-07-08T15:00:34Z
If-None-Match: aabd653b-65d0-74da-bc63-4bca-
ba3ef3f50432
```

- Response

```
200 OK
Content-Type: application/xml
Content-Length: ...
Last-Modified: 2007-07-08T15:00:34Z
Etag: aabd653b-65d0-74da-bc63-4bca-ba3ef3f50432
```

Client's representation of  
the resource is up-to-date



Living life on the wild, wild, Web  
**SECURITY**

## Good Ole' HTTP Authentication

- HTTP Basic and Digest Authentication: IETF RFC 2617
- Have been around since 1996 (Basic)/1997 (Digest)
- Pros:
  - Respects Web architecture:
    - stateless design (retransmit credentials)
    - headers and status codes are well understood
  - Does not prohibit caching (set `Cache-Control` to `public`)
- Cons:
  - Basic Auth must be used with SSL/TLS (plaintext password)
  - Not ideal for the human Web – no standard logout
  - Only one-way authentication (client to server)

## HTTP Basic Auth Example

1. Initial HTTP request to protected resource

```
GET /index.html HTTP/1.1  
Host: example.org
```

2. Server responds with

```
HTTP/1.1 401 Unauthorized  
WWW-Authenticate: Basic realm="MyRealm"
```

3. Client resubmits request

```
GET /index.html HTTP/1.1  
Host: example.org  
Authorization: Basic Qm9iCnBhc3N3b3JkCg==
```

Further requests with same or deeper path can include the additional `Authorization` header preemptively

## HTTP Digest Difference

- Server reply to first client request:

```
HTTP/1.1 401 Unauthorized
```

```
WWW-Authenticate: Digest
```

```
    realm=myrealm@example.org,
```

```
    qop="auth,auth-int",
```

```
    nonce="a97d8b710244df0e8b11d0f600bfb0cdd2",
```

```
    opaque="8477c69c403ebaf9f0171e9517f347f2"
```

- Client response to authentication challenge:

```
Authorization: Digest
```

```
    username="bob",
```

```
    realm=myrealm@example.org,
```

```
    nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
```

```
    uri="/index.html",
```

```
    qop=auth, nc=00000001, cnonce="0a6f188f",
```

```
    response="56bc2ae49393a65897450978507ff442",
```

```
    opaque="8477c69c403ebaf9f0171e9517f347f2"
```

## Unhealthy Cookies?

- Form-based authentication on the human Web uses cookies
- Can be used on the programmatic Web – POST to the authentication URL
- Server can (should!) inform client about intended cookie lifetime
- Cookie value often used as key to server session state
  - Breaks stateless constraint
  - Solution that does not require server side session state:  
<http://cookies.lcs.mit.edu/pubs/webauth:tr.pdf>

## SSL / TLS

- “Strong” server and optional client authentication, confidentiality and integrity protection
- The only feasible way to secure against man-in-the-middle attacks
- Not broken! Even if some people like to claim otherwise
- Not cache friendly, even using ‘null’ encryption mode
- Performance *and* security becomes difficult

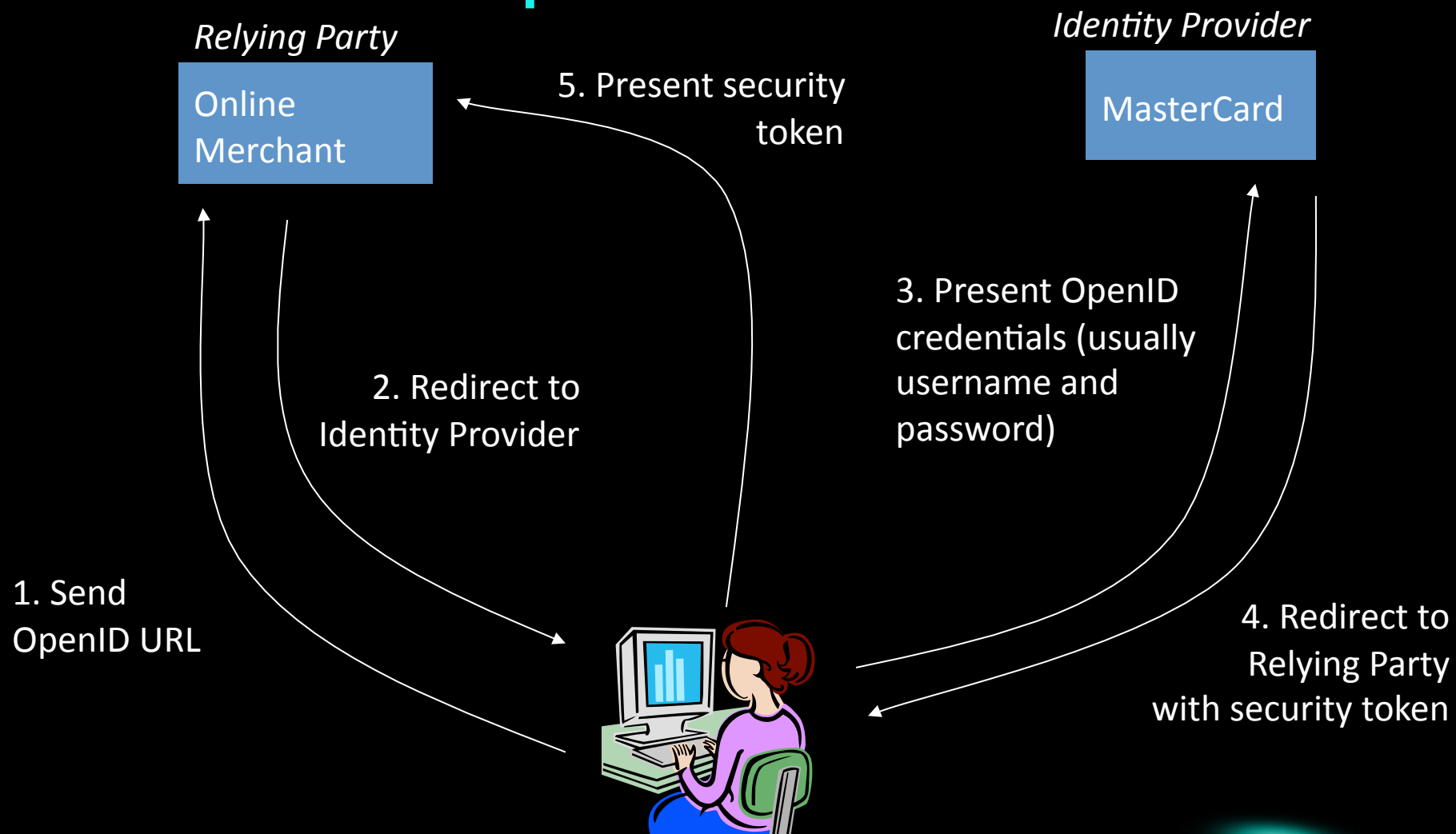


## OpenID

- OpenID is a decentralised framework for digital identities
  - Not trust, just identity!
- You have an OpenID provider or one is provided for you
  - It has a URI
- Services that you interact with will ask for that URI
- Your OpenID provider will either
  - Accept the request for processing
  - Ask whether you trust the request (e.g. with hyperlinks)
- Once your OpenID server OK's the login, then you are authenticated against the remote service
  - With your canonical credentials

Authenticating doesn't mean you're authorised to do anything!  
This is not a trust system!

## OpenId Workflow



## OAuth

- Web-focused access delegation protocol
- Give other Web-based services access to some of your protected data without disclosing your credentials
- Simple protocol based on HTTP redirection, cryptographic hashes and digital signatures
- Extends HTTP Authentication as the spec allows
  - Makes use of the same headers and status codes
  - These are understood by browsers and programmatic clients
- Not dependent on OpenID, but can be used together

## Why OAuth?

### Find people you know on Facebook

Your friends on Facebook are the same friends, acquaintances and family members that you communicate with in the real world. You can use any of the tools on this page to find more friends.



#### Find People You Email

[Upload Contact File](#)

Searching your email address book is the fastest and most effective way to find your friends on Facebook.

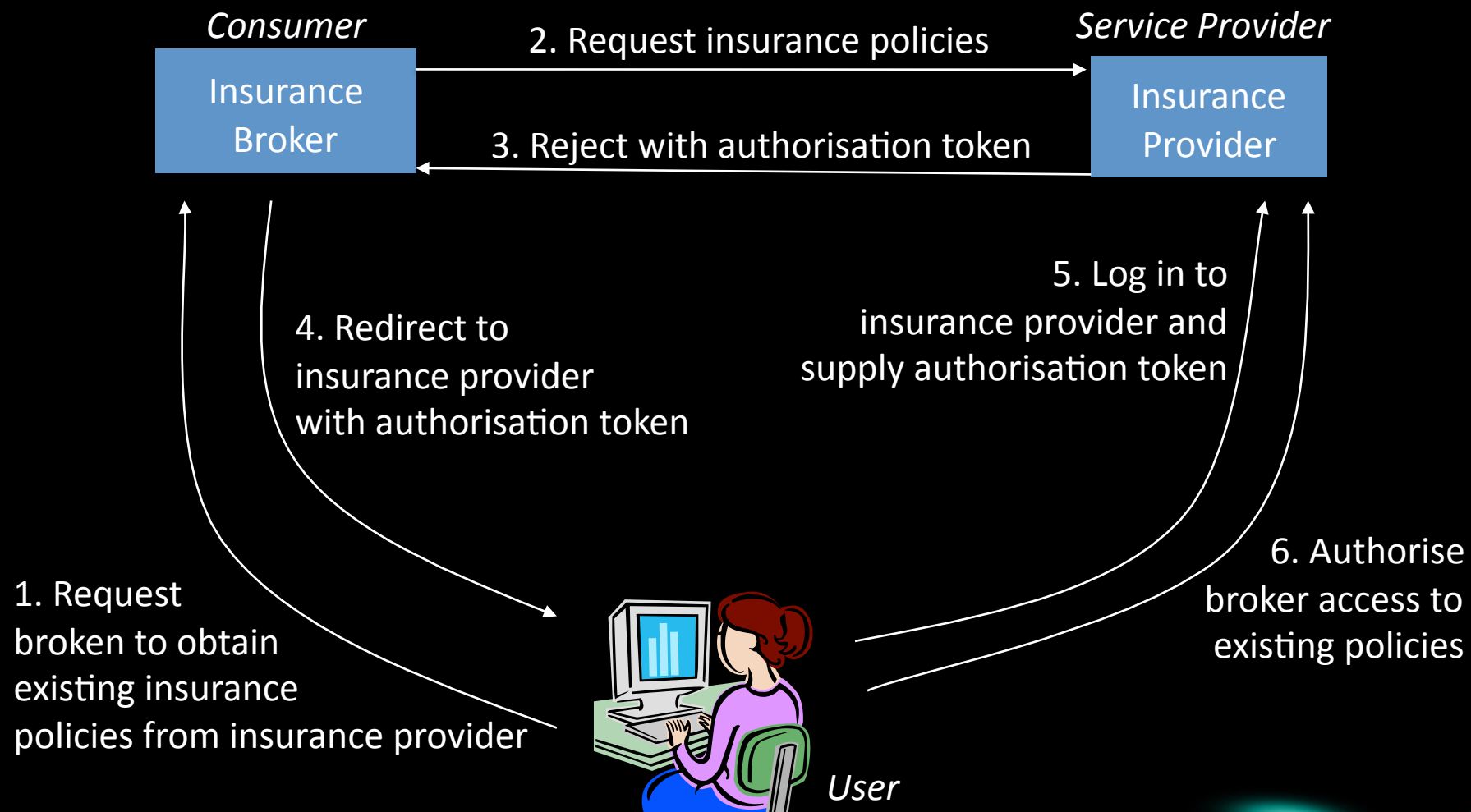
Your Email:

Password:

**Find Friends**

We won't store your password or contact anyone without your permission.

## OAuth Workflow



## OAuth Messages (1)

1. Alice (the User) has accounts on both the insurance broker and provider's Web sites
2. The insurance broker (Consumer) has registered itself at the insurance company and has a Consumer Key and Secret
3. Alice logs in to the broker and requests it to obtain her existing policies from the provider
4. Broker request to Insurance Provider:  
`GET /alice/policies HTTP 1.1`  
`Host: insurance.org`
5. Insurance provider's response:  
`401 Unauthorized`  
`WWW-Authenticate: OAuth realm="http://insurance.org/"`



## OAuth Messages (2)

6. Broker requests authorisation token from Provider:  
`POST /request_token`  
`oauth_consumer_key=abc&oauth_nonce=39kg&oauth_ ...`
7. Provider sends authorisation token in response body:  
`200 Success`  
`oauth_token=xyz&oauth_token_secret=abc`
8. Broker redirects Alice to Provider in response to her request:  
`302 Redirect`  
`Location: http://insurance.org/authorise?oauth_token=`  
`xyz&oauth_callback=http%3A%2F%2Fbroker.org&...`
9. Alice logs in to Insurance Provider using her credentials at that site (the Broker never sees these) and authorises the Broker to access her existing policies for a defined period of time.

## OAuth Messages (3)

10. Insurance Provider redirects Alice to the callback URL:

302 Redirect

Location: `http://broker.org/token_ready?oauth_token=xyz`

11. Broker knows Alice approved, it asks Provider for Access Token:

GET `/accesstoken?oauth_consumer_key=abc&oauth_token=xyz`

Host: `insurance.org`

12. The Insurance Provider sends back the Access Token:

200 Success

`oauth_token=zxcvb`

13. Broker creates hash or signature using access token, nonce, timestamp, Consumer Key and Secret (and more):

GET `/alice/policies` HTTP 1.1

Host: `insurance.org`

Authorization: OAuth realm="`http://insurance.org/`",  
`oauth_signature=...`, `oauth_consumer_key="abc"`, ...

How Atom and AtomPub make a mockery of your JMS

# **SYNDICATION**

## Syndication History

- Originally syndication used to provide feeds of information
  - Same information available on associated Web sites
- Intended to be part of the "push" Web
  - And allow syndication etc
- RSS was the primary driver here
  - Several versions, loosely described
    - Simple!
- ATOM followed
  - Format and protocol
  - Richer than RSS and now being used for the programmatic Web

## Atom

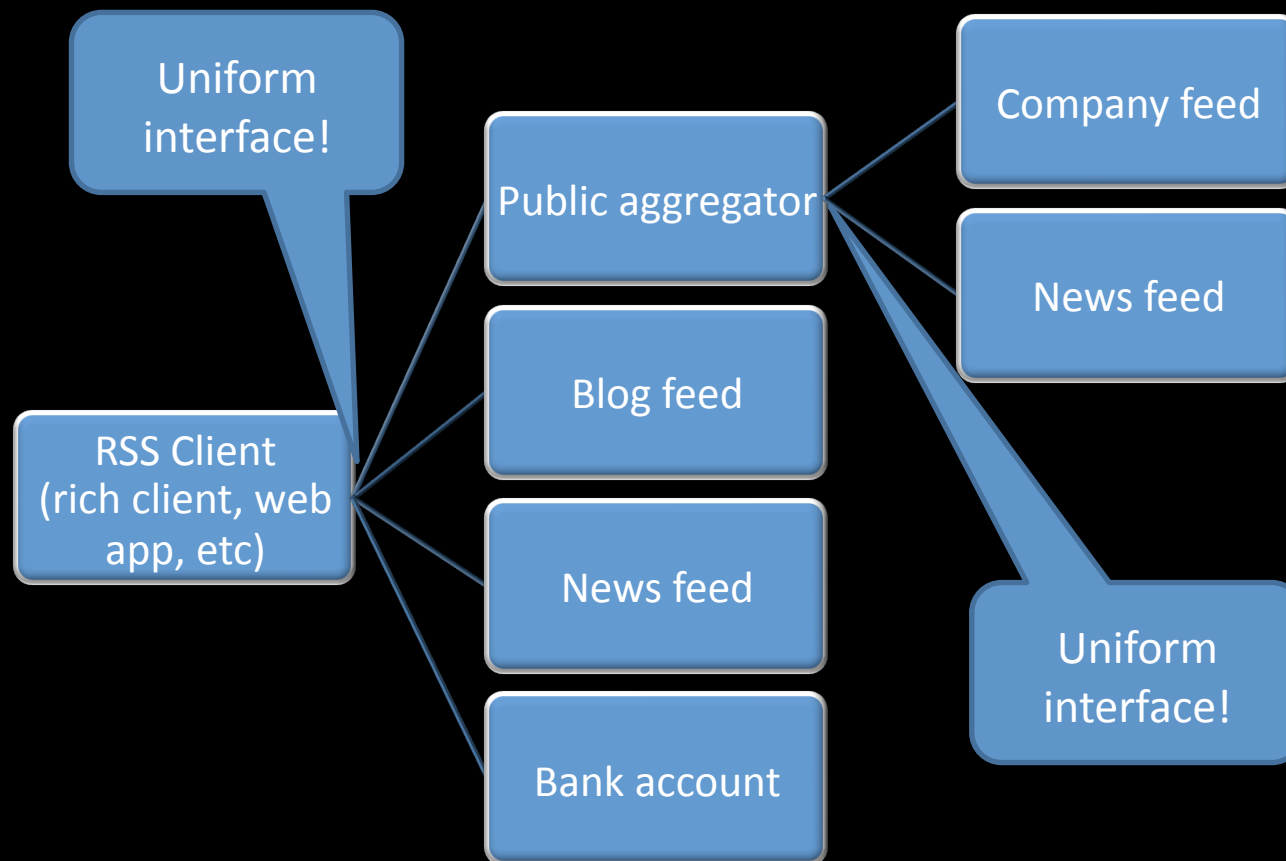
- Atom comes in two parts
  - XML vocabulary for lists of (time-stamped) entries
    - Aka feeds
  - Publishing protocol
    - A uniform interface that layers atop HTTP's uniform interface

## Atom Feeds

- Atom feeds contain useful information aimed at supporting *publishing*
  - Its primary domain is weblogs, syndication, etc
- Atom lists are known as *feeds*
- Items in Atom lists are known as *entries*



## Feed Architecture



## Anatomy of an Atom Feed

HTTP metadata

- **Media type:** application/atom+xml

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

```
<feed xmlns="http://www.w3.org/2005/Atom">
```

```
  <title>Webber, Parastatidis, and Robinson</title>
```

```
  <link rel="alternate"
```

```
    href="http://jim.webber.name/web.integration/" />
```

```
  <updated>2007-07-01T13:00:44Z</updated>
```

```
  <author><name>Jim Webber</name></author>
```

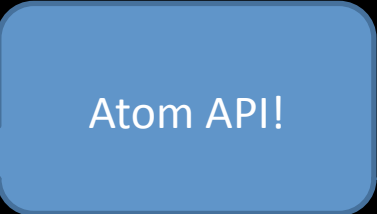
```
  <contributor><name>Savas Parastatidis</name></contributor>
```

```
  <id>urn:ab45fe7e-7ff3-886c-11d2-7da3fe465322</id>
```

Feed metadata

## More Anatomy of an Atom Feed

```
<entry>
  <title>Chapter 10 complete, says Webber</title>
  <link rel="service.edit" type="application/x.atom+xml"
    href="http://jim.webber.name/c10.aspx"/>
  <link rel="service.post" type="application/x.atom+xml"
    href="http://jim.webber.name/c10.aspx">
  <id>urn:dd64ef10-975d-23de-13fa-33d32117acb432</id>
  <updated>2007-07-01T13:00:44Z</updated>
  <summary>Chapter 10 deals with the comparison of Web and Web
  Services approaches to building distributed applications.
  </summary>
  <category scheme="http://jim.webber.name/categories/books"
    term="local" label="book news"/>
</entry>
</feed>
```



Atom API!



## Atom Feeds and Resources

- Atom is just a resource representation
- An Atom feed is a good resource representation for returning resources in response to a query
  - As is XML, XHTML, RSS, JSON and others



## Atom Extensibility

- Q: What if your resource representations don't fit in Atom entries directly?
- A: Use your own data!

```
<entry>
  <title>Chapter 10 complete, says Webber</title>
  ...
  <jw:openIssues xmlns:jw="http://jim.webber...">
    <jw:issue title="Colour diagrams degraded in BW format">
      <jw:status>closed</jw:status>
      <jw:actionTaken date="2007-06-28T16:44:12Z">
        <jw:takenBy>savas@parastatidis.name</jw:takenBy>
        <jw:description>re-drew all diagrams</jw:description>
      </jw:actionTaken>
    </jw:issue>
    ...
  </jw:openIssues>
</entry>
```

This will be ignored if  
your client application  
doesn't know the  
namespace



## Atom Publishing Protocol

- APP defines a set of resources that handle *publishing* Atom documents
  - Four kinds of resources
    - Collection
    - Member
    - Service Document
    - Category Document
  - And their representations on the wire
- Another uniform interface atop the HTTP uniform interface

## APP: Collections

- Collection's representation is an Atom feed
- APP defines semantics for the collection representation
  - GET – retrieve the collection/feed
  - POST – adds a new member to the collection
    - Adds a new entry to the feed
  - PUT and DELETE undefined by APP
    - But probably should delete a collection or update a collection in place respectively

## Consuming Feeds in Applications

- Feeds on the Internet have so-far been used to optimise the human Web
  - Site summaries, blog posting, etc
- However feeds are a data structure
  - And so potentially machine-processable
- Embedding machine-readable payloads means we have a vehicle for computer-computer interaction