# Web as a Platform

## Web History

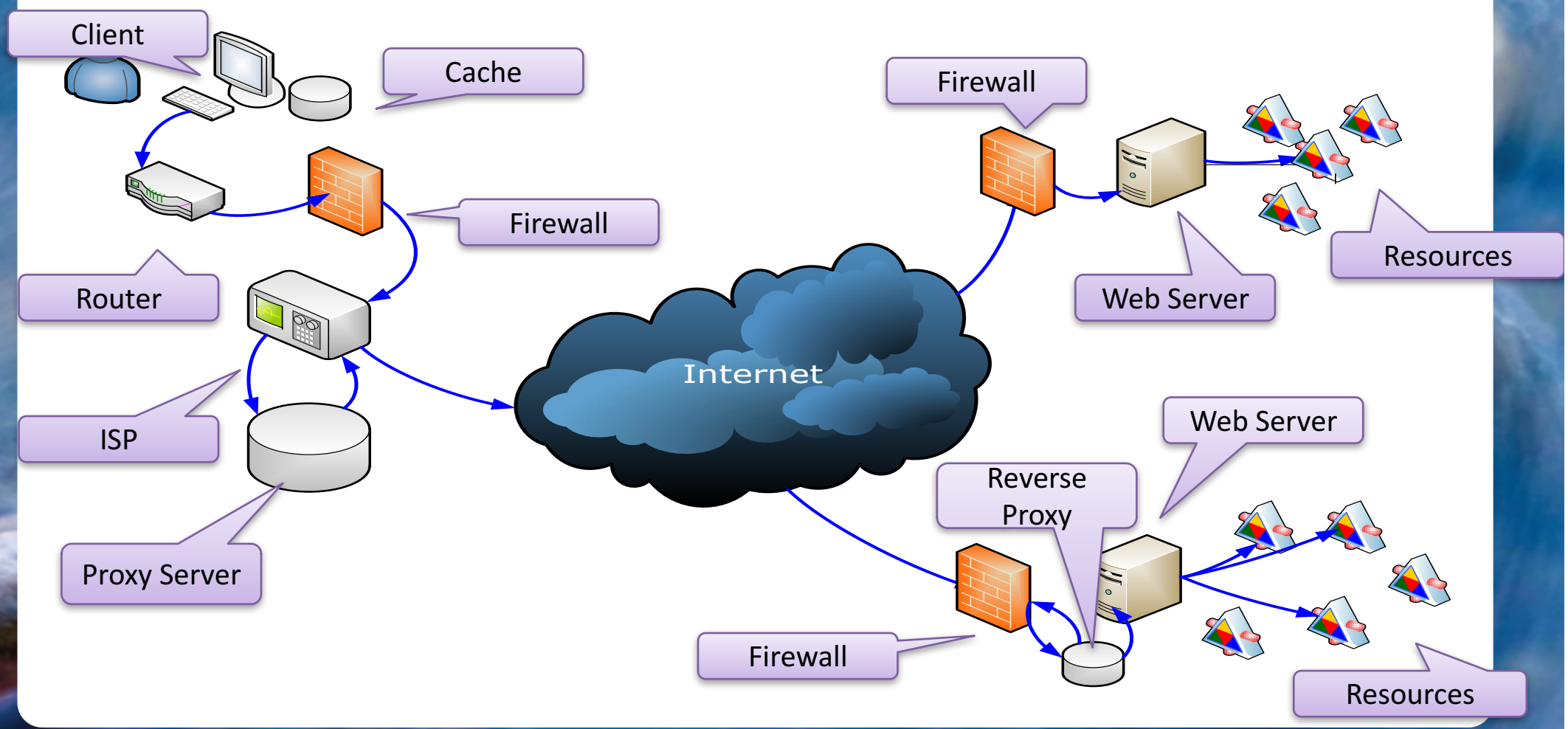
- Started as a distributed hypermedia platform
  - CERN, Berners-Lee, 1990
- Revolutionised hypermedia
  - Imagine emailing someone a hypermedia deck nowadays!
- Architecture of the Web largely fortuitous
  - W3C and others have since retrofitted/captured the Web's architectural characteristics

# The Web broke the rules

## 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



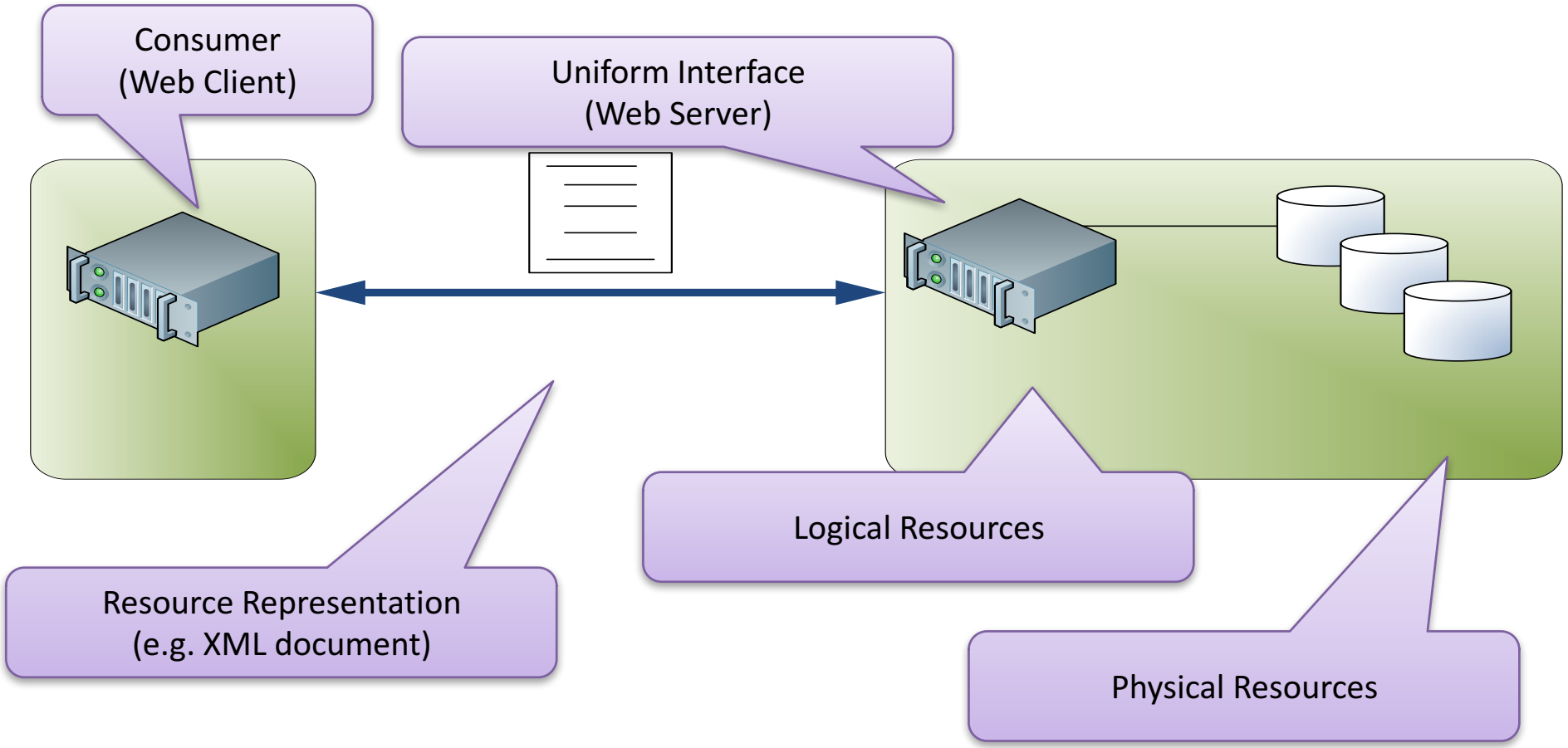
## Resources

- 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" semantics
  - Representation can be in any format
    - Any media type
- Each resource implements a standard uniform interface
  - Typically the HTTP interface
- Resources have names and addresses (URIs)
  - Typically 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 or good for the programmatic Web
- Moving representations across the network is the way we transact work in a Web-native system

## URIs

| Part | Description |
| --- | --- |
| Identity | Distinguishes one entity from another |
| Addressing | Means to interact/communicate with entity |

## Identity

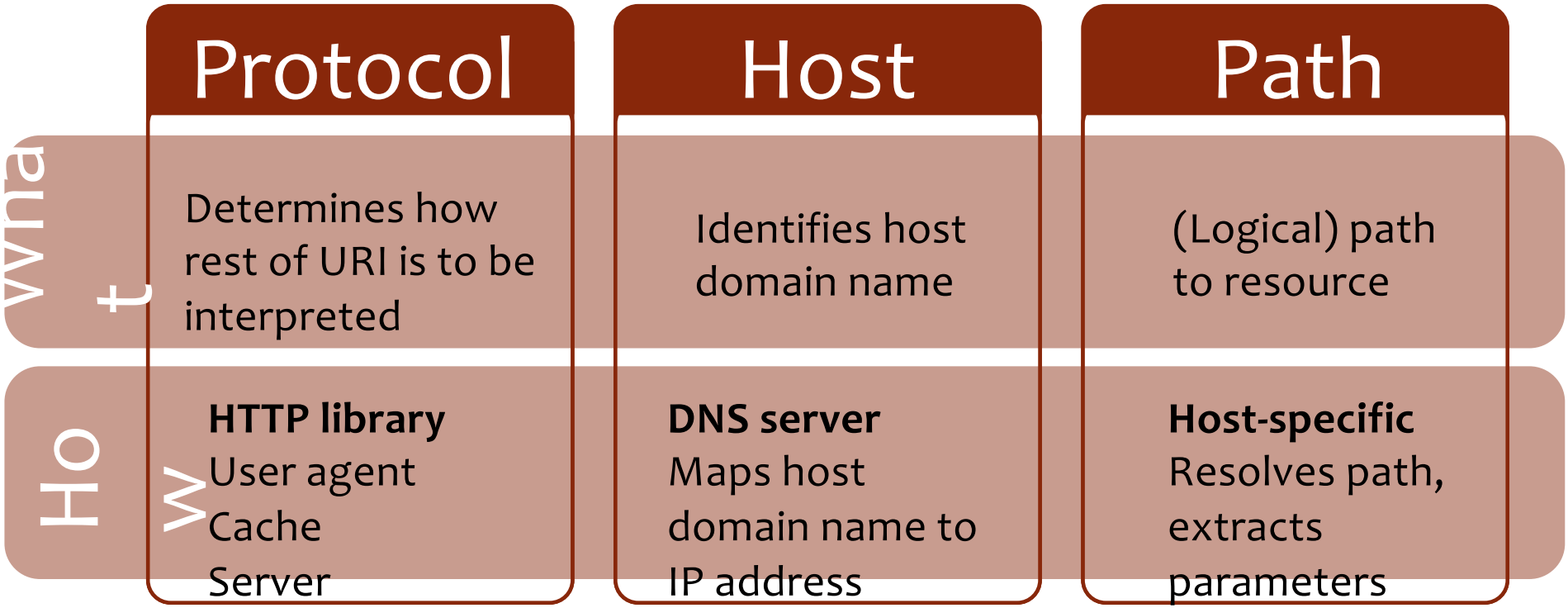
***An*** identifier
not
***The*** identifier

**Bounded in time**
Cool URIs don't change
URIs change
Get over it

```
<entry>
    <id>urn:uuid:b436fda6-93f5-4c00-98a3-06b62c3d31b8</id>
    <title type="text">promotion cancelled</title>
    <updated>2008-09-10T13:57:14+01:00</updated>
    <link rel="self"
      href="http://restbucks.com/notifications/b436fda6"/>
    <content type="application/xml">
     ...
    </content>
</entry>
```

## Addressing: portable, portmanteau, declarative structure

```
<protocol>://<host>:<port>/<path>
http://restbucks.com:8080/order/123
```



## 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 is stateless
  - All information required to process a request must be present in that request
    - Sessions are still plausible, but must be handled in a Web-consistent manner
      - Modelled as resources!
- Statelessness 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
    - Library of Congress found this the hard way!
  - 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!
  - Some verbs (e.g. PUT, DELETE) are safe to repeat

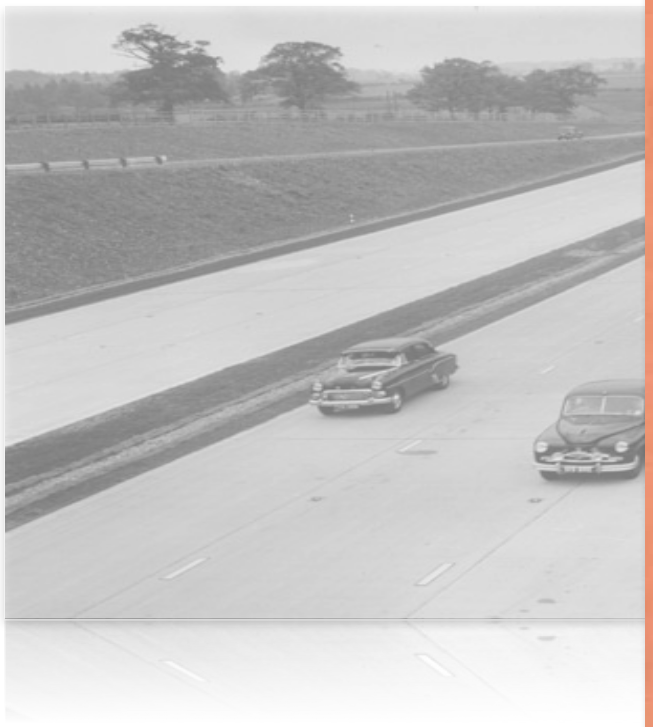
## Secure

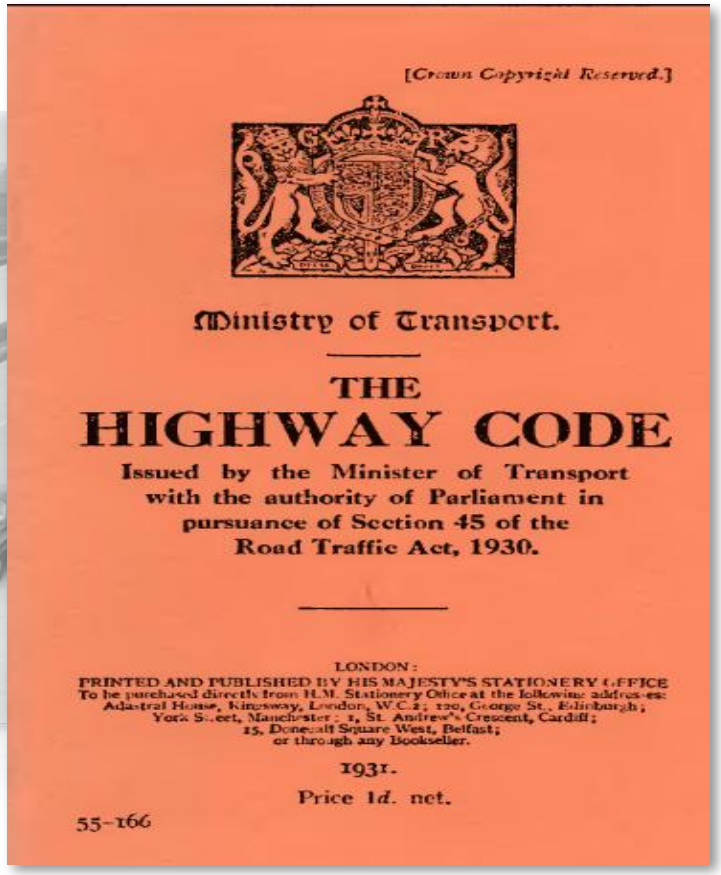
- HTTPs is a mature technology
  - Based on SSL for secure point-to-point information retrieval
- Isn't sympathetic to Web architecture
  - Restricted caching opportunities
- Higher-order protocols like Atom are starting to change this...
  - Encrypt parts of a resource representation, not the transport channel
  - OK to cache!

## 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

# HTTP = ???





[Crown Copyright Reserved.]

Ministry of Transport.

THE
HIGHWAY CODE

Issued by the Minister of Transport
with the authority of Parliament in
pursuance of Section 45 of the
Road Traffic Act, 1930.

LONDON:
PRINTED AND PUBLISHED BY HIS MAJESTY'S STATIONERY OFFICE
To be purchased directly from H.M. Stationery Office at the following addresses:
Adastral House, Kingsway, London, W.C.2; 120, George St., Edinburgh;
York Street, Manchester; 1, St. Andrew's Crescent, Cardiff;
15, Donegall Square West, Belfast;
or through any Bookseller.

1931.

Price 1d. net.

55-166

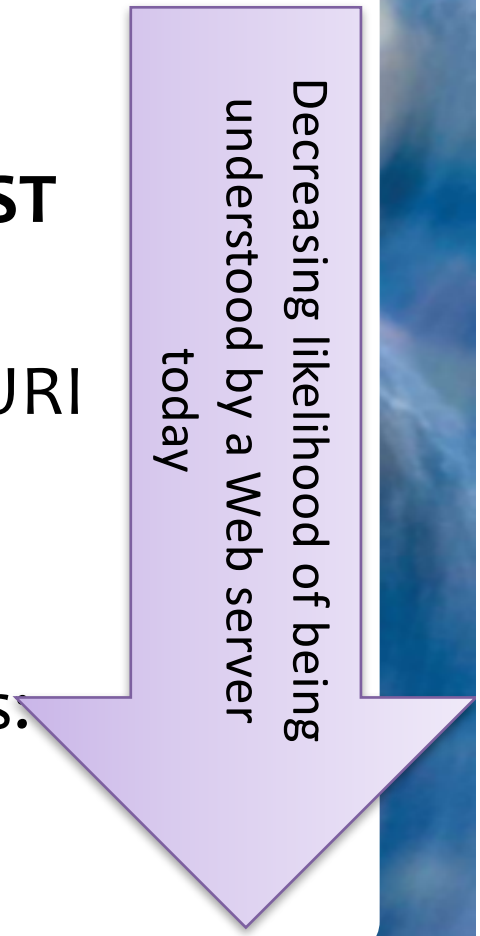

HTTP Fundamentals

## HTTP: an application transfer protocol

| Term | Description |
| --- | --- |
| Application | Transacted at application layer of network stack, in user agent and server software |
| Transfer | For the purpose of coordinating the transfer of documents |
| Protocol | According to rules that determine legitimate interactions between client and server |

## The HTTP Verbs

- Retrieve a representation of a resource: **GET**
- Create a new resource: **PUT** to a new URI, or **POST** to an existing URI
- Modify an existing resource: **PUT** to an existing URI
- Delete an existing resource: **DELETE**
- Get metadata about an existing resource: **HEAD**
- See which of the verbs the resource understands: **OPTIONS**



## HEAD Semantics

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

```
HEAD /order/1234 HTTP 1.1
Host: restbucks.com
```

- Response

```
200 OK
Content-Type:
  application/vnd.restbucks+xml
Last-Modified: 2
:00:34Z
Etag: 77fe653b
```

Useful for caching, performance

## OPTIONS Semantics

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

- Request

```
OPTIONS /o                    .1
Host: rest
```

- Response

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

You can only read and add to this resource, may change over time

## HTTP Status Codes

- The HTTP status codes provide metadata about the state of resources
- They are part of what makes the Web a rich platform for building distributed systems
- They cover five broad categories
  - 1xx - Metadata
  - 2xx – Everything's fine
  - 3xx – Redirection
  - 4xx – Client did something wrong
  - 5xx – Server did a bad thing
- There are a handful of these codes that we need to know in more detail

## 1XX

- 100 – Continue
  - The operation will be accepted by the service
  - The “look before you leap” pattern
    - Use with the Expect header

- Request

```
POST /orders HTTP/1.1
Host: restbucks.com
Content-Type: application/vnd.restbucks+xml
Content-Length: 10216
Expect: 100-continue
```

- Response

```
HTTP/1.1 100 Continue
```

OR

```
HTTP/1.1 417 Expectation Failed
```

## 2xx

- 200 – OK
  - The server successfully completed whatever the client asked of it
- 201 – Created
  - Sent when a new resource is created at the client's request via POST
  - `Location` header should contain the URI to the newly created resource
- 202 – Accepted
  - Client's request can't be handled in a timely manner
  - `Location` header should contain a URI to the resource that will eventually be exposed to fulfil the client's expectations

# More 2xx Codes

- 203 – Non-Authoritative Information
  - Much like 200, except the client knows not to place full trust in any headers since they could have come from 3rd parties or be cached etc.
- 204 – No Content
  - The server declines to send back a representation
    - Perhaps because the associated resource doesn't have one
  - Used like an "ack"
    - Prominent in AJAX applications
- 206 – Partial Content
  - Optimisation used in failure cases to support partial GETs
  - Request `Content-Range` header must specify the byte range of the resource representation it wants
  - Response headers must contain `Date`;
    - `ETag` and `Content-Location` headers must be consistent with the original request not the current values

## 3xx

- 301 – Multiple Choices
  - Response Location header should contain the preferred URI
  - Message body can contain list of URIs
    - In XHTML possibly
  - In general avoid being ambiguous!
- 302 – Moved Permanently
  - `Location` header contains the new location of the resource
- 303 – See Other
  - `Location` header contains the location of an alternative resource
  - Used for redirection

## More 3xx

- 304 – Not Modified
  - The resource hasn't changed, use the existing representation
  - Used in conjunction with conditional GET
  - Client sends the `If-Modified-Since` header
  - Response `Date` header must be set
  - Response `Etag` and `Content-Location` headers must be same as original representation
- 307 – Temporary Redirect
  - The request hasn't been processed, because the resource has moved
  - Client must resubmit request to the URI in the response `Location` header

## 4xx

- 400 – Bad Request
  - The client has PUT or POST a resource representation that is in the right format, but contains invalid information
- 401 – Unauthorized
  - Proper credentials to operate on a resource weren't provided
  - Response `WWW-Authenticate` header contains the type of authentication the server expects
    - Basic, digest, WSSE, etc
  - Don't leak information!
    - Consider 404 in these situations

## More 4xx

- 403 – Forbidden
  - The client request is OK, but the server doesn't want to process it
    - E.g. Restricted by IP address
  - Implies that resource exists, beware leaking information
- 404 – Not Found
  - The standard catch-all response
  - May be a lie to prevent 401 or 403 information leakage

## Even more 4xx

- 405 – Method Not Allowed
  - The resource doesn't support a given method
  - The response `Allow` header lists the verbs the resource understands
    - E.g. `Allow: GET, POST, PUT`
- 406 – Not Acceptable
  - The client places too many restrictions on the resource representation via the `Accept-*` header in the request
  - The server can't satisfy any of those representations

## Yet More 4xx

- 409 – Conflict
  - Tried to change the state of the resource to something the server won't allow
    - E.g. Trying to DELETE something that doesn't exist
- 410 – Gone
  - The resource has gone, permanently.
  - Don't send in response to DELETE
    - The client won't know if it was deleted, or if it was gone and the delete failed
- 411 – Length Required
  - If a request (POST, PUT) contains a representation, it should set the `Content-Length` header
  - The server *might* demand this, and interrupt the client request

## Still more 4xx

- 412 – Precondition Failed
  - Server/resource couldn't meet one or more preconditions
    - As specified in the request header
  - E.g. Using `If-Unmodified-Since` and PUT to modify a resource provided it hasn't been changed by others
- 413 – Request Entity Too Large
  - Response comes with the `Retry-After` header in the hope that the failure is transient

## Final 4xx Codes

- 414 – Request URI Too Long
  - You've got a hopeless HTTP server, or ridiculous URI names!
- 415 Unsupported Media Type
  - E.g. If the server resource expects JSON but the client sends XML

## 5xx Codes

- 500 – Internal Server Error
  - The normal response when we're lazy ☺
- 501 – Not Implemented
  - The client tried to use something in HTTP which this server doesn't support
    - Keep HTTP use simple, get a better HTTP server
- 502 – Bad Gateway
  - A proxy failed
  - Doesn't help us much ☹

## More 5xx Codes

- 503 – Service Unavailable
  - The HTTP server is up, but not supporting resource communication properly
  - Server may send a `Retry-After` header, assuming the fault is transient

## 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
- Content-Length
  - Length of payload, in bytes
- Content-Type
  - The resource representation form
    - E.g. application/vnd.restbucks+xml, application/xhtml+xml

## More Must-Know Headers

- Etag: `Etag: "bne3t6s"`
  - Opaque identifier – think "checksum" for resource representations
- If-None-Match
  - Execute conditional operation if resource ***has*** changed
    - Conditional GET: 304 Not Modified
    - Other conditional operations: 412 Precondition Failed
- If-Match
  - Mainly other conditional operations (PUT, POST, DELETE)
  - Execute conditional operation if resource ***hasn't*** changed
  - 412 Precondition Failed

## Some More Must-Know Headers

- Last-Modified: `Last-modified: Wed, 03 Mar 2010 20:58:03 GMT`
  - Timestamp
- If-Modified-Since
  - Execute conditional operation if resource ***has*** changed
    - Conditional GET: 304 Not Modified
    - Other conditional operations: 412 Precondition Failed
- If-Not-Modified-Since
  - Mainly other conditional operations (PUT, POST, DELETE)
  - Execute conditional operation if resource ***hasn't*** changed
  - 412 Precondition Failed

## Yet More Must-Know Headers

- 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
- User-Agent
  - Tells the server side what the client-side capabilities are
  - Should not be used in the programmable Web!

## Final Must-Know Headers

- WWW-Authenticate
  - Used with 401 status
  - Informs client what authentication is needed
- Date
  - Mandatory!
  - Timestamps on request and response
- Host
  - Contains the domain-name part of the URI

## Useful Headers

- Accept
  - Client tells server what formats it wants
  - Can externalise this in URI names in the general case
- Accept-Encoding
  - Client tells server that it can compress data to save bandwidth
  - Client specifies the compression algorithm it understands
- Content-Encoding
  - Server-side partner of Accept-Encoding

## More Useful Headers

- Allow
  - Server tells client what verbs are allowed for the requested resource (used in combination with OPTIONS)
- Cache-Control
  - Metadata for caches, tells them how to cache (or not) the resource representation
  - And for how long etc.
- Content-MD5
  - Cryptographic checksum of body
  - Useful integrity check, has computation cost

## Yet More Useful Headers

- Expect
  - A conditional – client asks if it's OK to proceed by expecting 100-Continue
  - Server either responds with 100 or 417 – Expectation Failed
- Expires
  - Server tells client or proxy server that representation can be safely cached until a certain time
- If-Match
  - Used for ETag comparison
  - Opposite of If-None-Match

## Final Useful Headers

- If-Unmodified-Since
  - Useful for conditional PUT/POST
    - Make sure the resource hasn't changed while you're been manipulating it
  - Compare with If-Modified-Since
- Range
  - Specify part of a resource representation (a byte range) that you need – aka partial GET
  - Useful for failure/recovery scenarios

## Less Often-Used Headers

- Retry-After
  - Resource or server is out of action for the specified time
  - Usually associated with 413 – Request Entity Too Large, or one of the 5xx server error statuses
- Content-Location
  - Header gives the canonical URI of the resource
  - Client might be using a different URI to access that resource

## HTTP RCF 2616 is Authoritative

- The statuses and headers here are a sample of the full range of headers in the HTTP spec
- They spec contains more than we discuss here
- It is authoritative about usage
- And it's a good thing to keep handy when you're working on a Web-based distributed system!


The Platform is the Web

## Web as a Platform

- The Web is a reliable integration platform
  - Not just for human content
- HTTP is a very well-thought-out protocol for manipulating objects at a distance
  - Concurrency control, caching, fault tolerance, resume, coordination all baked in
- This session takes a deep look at what's on offer

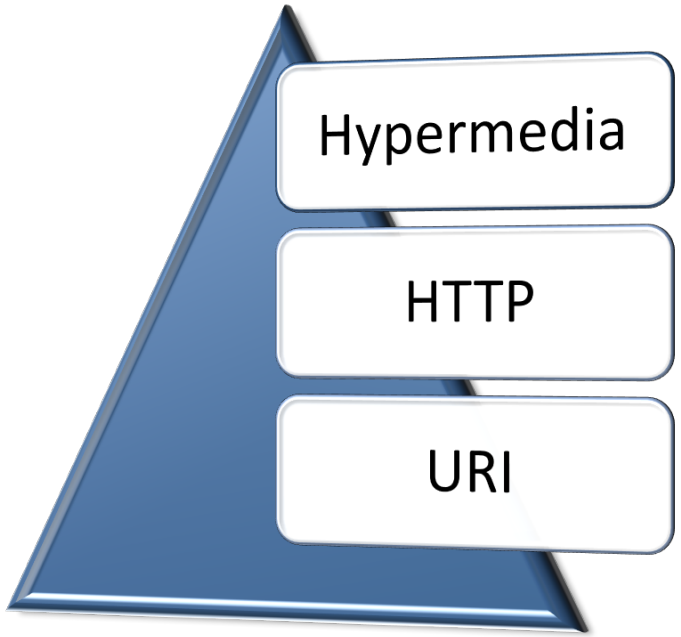
## The Web is middleware

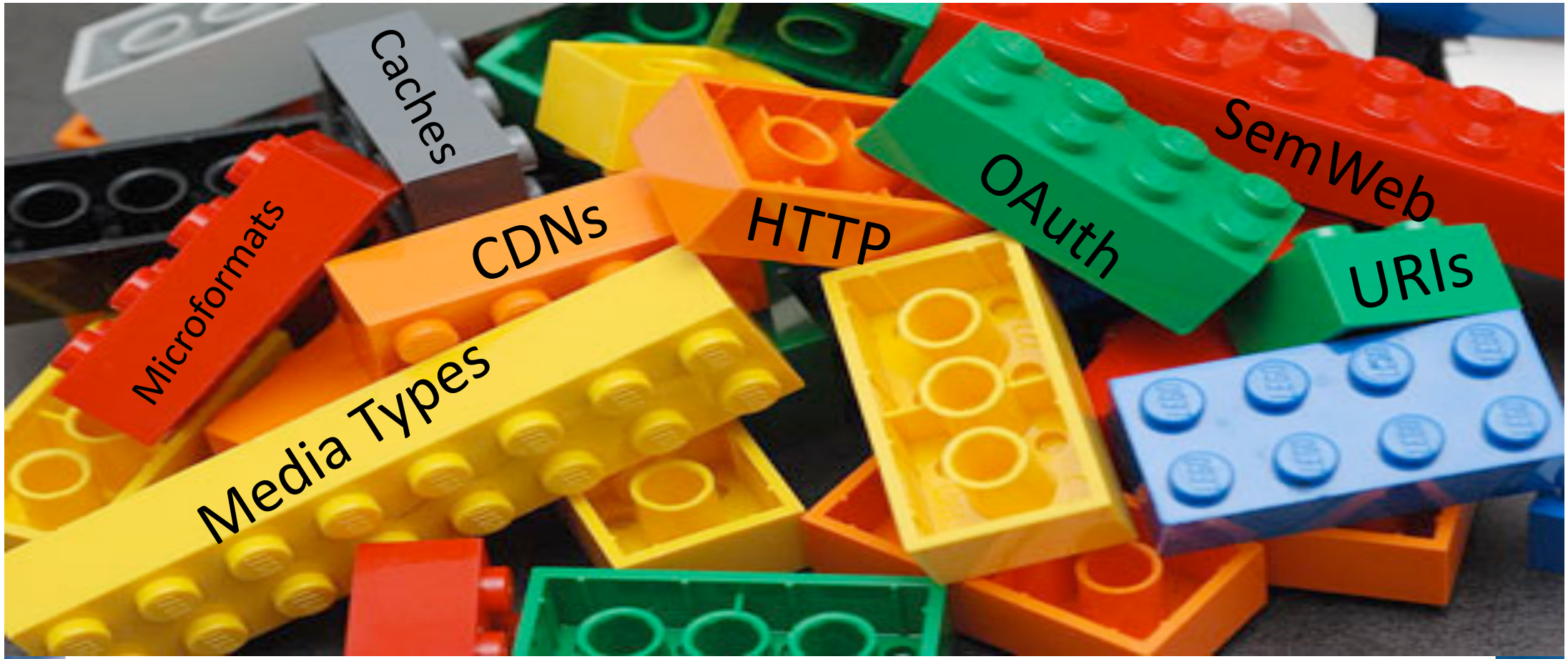
- The Web is a middleware platform which is…
  - Globally deployed
  - Has reach
  - Is mature
  - And is a reality in every part of our lives

- Which makes it interesting for distributed systems like microservices

# Leonard Richardson's Web service maturity heuristic

- REST is cool
- But the Web is permissive
- There's a spectrum of maturity of service styles
  - From completely bonkers to completely RESTful
- We'll use the Richardson maturity model to frame these kinds of discussions
  - Level 0 to Level 3
  - Web-ignorant to RESTful!






REST is an Architectural style suited to the Web…

…but it is not mandatory

## Why the Web? Why be RESTful?

- Scalable
- Fault-tolerant
- Recoverable
- Secure
- Loosely coupled

- Precisely the same characteristics we want in business software systems!

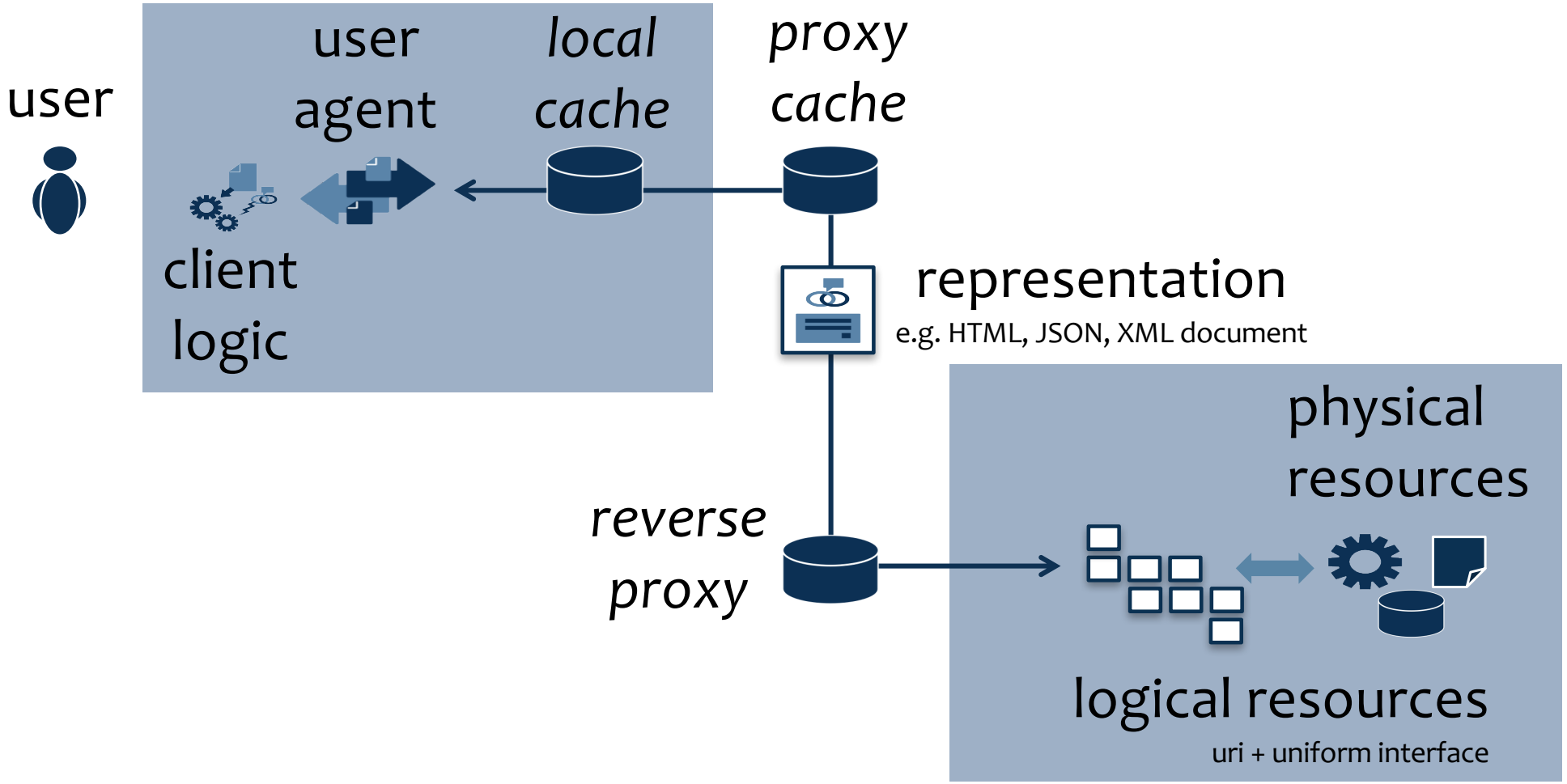
## REST isn't...

- Just pretty URIs
- Just XML or JSON
- URI templates
- AJAX applications

But you could be forgiven for thinking it is!


Web Architecture

# Web Architecture



## REST: A Formal Description of the Architecture of the Web

- Roy Fielding, Chapter 5, "Architectural Styles and the Design of Network-based Software Architectures"
  - https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- ***Re***presentational ***S***tate ***T***ransfer
  - Transfer representations of state (documents) between client and server in order to progress the state of an application
- An architectural ***style***
  - Application of design constraints
    - For network-based applications
    - To induce the architectural properties of the Web

# REST's Architectural Constraints (1)

| Constraint | Induces | Tradeoff |
|---|---|---|
| Client-Server | Separation of concerns<br>UI portability<br>Simplified server<br>Multiple organizational domains | |
| Stateless | Visibility<br>Reliability<br>Scalability | Decreased network performance<br>Server does not control application state |
| Caching | Reduced latency<br>Efficiency<br>Scalability | Stale data reduces *reliability* |

# REST's Architectural Constraints (2)

| Constraint | Induces | Tradeoff |
|---|---|---|
| Uniform Interface | Visibility<br>Decoupled implementation<br>Independent evolution | Standardized rather than application-specific format degrades efficiency |
| Layered System | Reduced complexity<br>Substrate independence<br>Encapsulate legacy components<br>Shared intermediaries<br>Load balancing | Increased latency |
| (Optional) Code-on-Demand | Simplified clients<br>Extensibility | Reduced visibility |

# REST's Uniform Interface Constraints

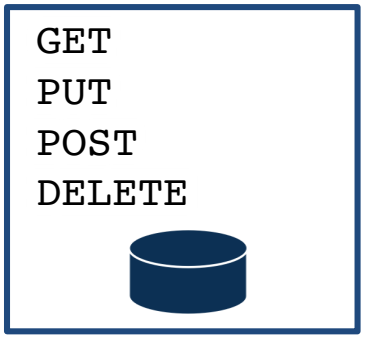
| Constraint | Description |
|---|---|
| Identification of resources | Each important resource is uniquely *identifiable*<br>One resource identifier mechanism: e.g. URI |
| Manipulation of resources through representations | Request – via document exchange – rather than tell |
| Uniform access semantics | Uniform document exchange protocol: e.g. HTTP |
| Self-descriptive messages | Intention "written on the wire"<br>e.g. HTTP + media-type processing model |
| Hypermedia as the engine of application state | Hypermedia systems transform application state |

http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

# What is a Resource?

- Address + identity (URI)
- Encapsulates state (data)
- Manipulated through representations (e.g. HTML, JSON, XML)
- Supports HTTP uniform interface

```
<shop xmlns="http://schemas.restbucks.com/shop"
  xmlns:rb="http://relations.restbucks.com/">
  <items>
    <item>
      <description>Costa Rica Tarrazu</description>
      <amount>250g</amount>
      <price currency="GBP">4.40</price>
    </item>
  </items>
  <link rel="rb:order-form"
    href="http://restbucks.com/order-forms/1234"/>
</shop>
```




`http://restbucks.com/quotes/1234`

## Web Technologies

- URIs
- Media Types
- HTTP

## URIs are Names for Resources

```
http://restbucks.com:8080/order/123
```

### Identity

Differentiates one resource from another

### Addressing

Means for accessing and manipulating resource representations
Each resource has ***at least one*** address
Every resource implements the same uniform interface – HTTP

### *Not* Semantics

```
<link href="http://restbucks.com:8080/order/123" rel="order"/>
```

# URI – Declarative Structure Targeting Different Parts of Network

```
<protocol>://<host>:<port>/<path>
http://restbucks.com:8080/order/123
```

| | Protocol | Host | Path |
|---|---|---|---|
| What | Determines how rest of URI is to be interpreted | Identifies host domain name | (Logical) path to resource |
| How | **User agent**<br>Client software | **DNS server**<br>Maps host domain name to IP address | **Host-specific**<br>Resolves path, extracts parameters |

## Media Types

- Used for labelling and formatting resource representations
- Some types are special because they work in harmony with the Web
  - We call these “hypermedia formats”
- A media type defines a processing model
  - If the client understands the type, it can process the content
- Processing model may include:
  - Representation format (may include schemas)
  - Supported operations (methods, headers and status codes)
  - Definitions of hypermedia controls (links, forms)
  - Application semantics (link relations)

# Hypermedia Formats

- **JSON Hypertext Application Language**
  - Supports links and embedded resources
  - No intrinsic support for forms/operations
- **JSON for Linked Documents**
  - Can be used to enhance existing JSON
  - Use (unofficial) HYDRA vocabulary for operations
- **Collection+JSON**
  - Query templates
  - semantic profiles (application semantics for message body)
- **HTML**
  - Intrinsic support for links and forms
  - Application semantics using e.g. microformats
- **Atom**
  - Atom Syndication Format for structure
  - Atom Publication Protocol for operations
  - Supports nested representations using other media types

# JSON

```
{
    "playerId": "1234567890",
    "alias": "soofaloofa",
    "displayName": "Kevin Sookocheff",
    "profilePhotoUrl": "https://api.example.com/player/1234567890/avatar.png"
}
```

http://sookocheff.com/post/api/on-choosing-a-hypermedia-format/

# JSON-LD

```
{
    "@context": {
        "name": "https://schema.org/name",
        "alternateName": "https://schema.org/alternateName",
        "image": {
            "@id": "https://schema.org/image",
            "@type": "@id"
        },
        "friends": {
            "@container": "@set"
         }
    },
    "@id": "https://api.example.com/player/1234567890",
    "playerId": "1234567890",
    "name": "Kevin Sookocheff",
    "alternateName": "soofaloofa",
    "image": "https://api.example.com/player/1234567890/avatar.png",
    "friends": [
        {
            "@id": "https://api.example.com/player/1895638109"
        },
        {
            "@id": "https://api.example.com/player/8371023509"
        }
    ]
}
```

http://sookocheff.com/post/api/on-choosing-a-hypermedia-format/

# HAL

```
{
    "_links": {
        "self": { "href": "https://api.example.com/player/1234567890" },
        "https://api.example.com/docs/rels/friends": {
                  "href": "https://api.example.com/player/1234567890/friends" }
    },
    "playerId": "1234567890",
    "name": "Kevin Sookocheff",
    "alternateName": "soofaloofa",
    "image": "https://api.example.com/player/1234567890/avatar.png"
}
```

http://sookocheff.com/post/api/on-choosing-a-hypermedia-format/

# Collection+JSON

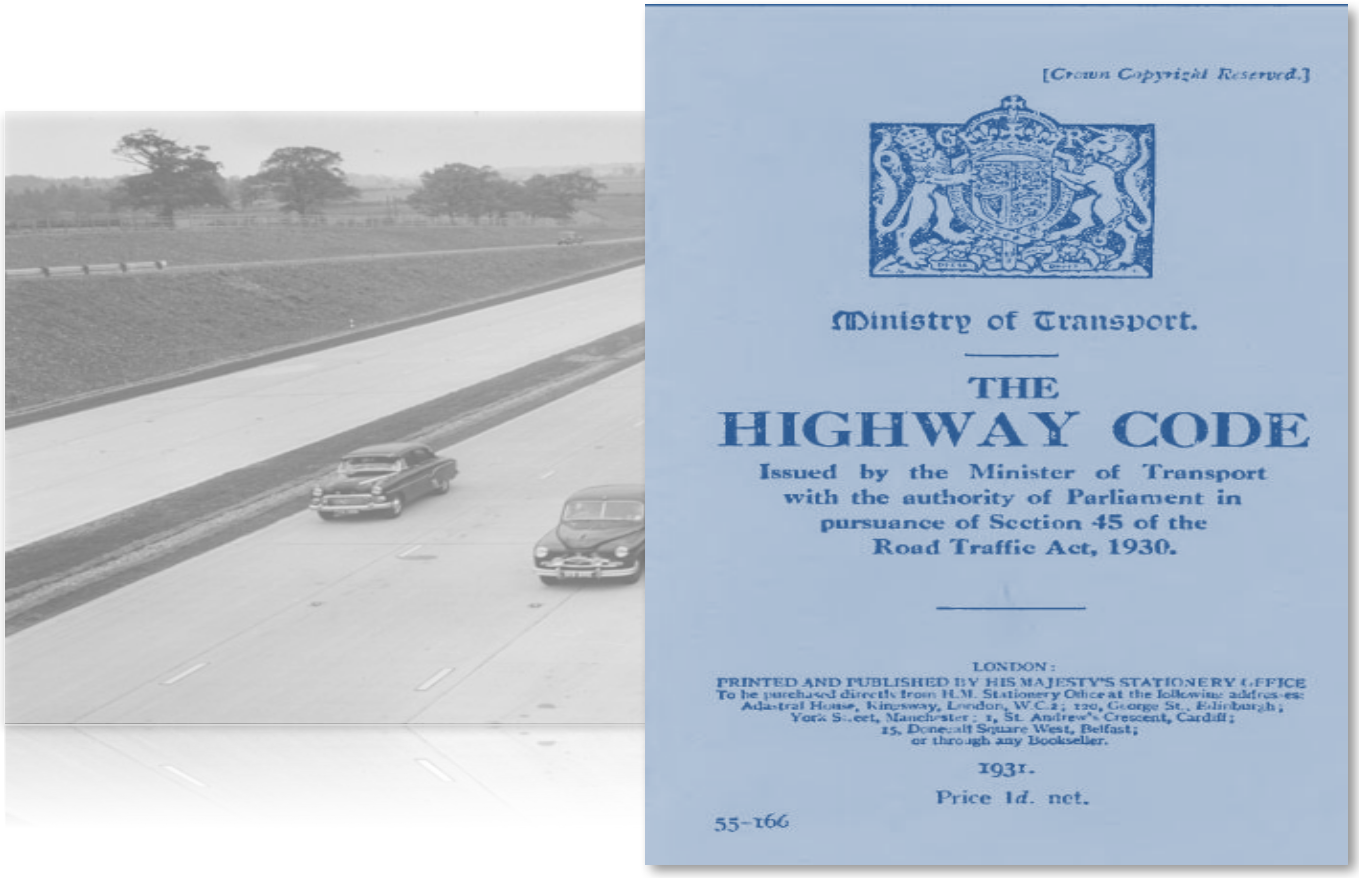
```
{
    "collection": {
        "version": "1.0",
        "href": "https://api.example.com/player",
        "items": [
            {
                "href": "https://api.example.com/player/1234567890",
                "data": [
                      {"name": "playerId", "value": "1234567890", "prompt": "Identifier"},
                      {"name": "name", "value": "Kevin Sookocheff", "prompt": "Full Name"},
                      {"name": "alternateName", "value": "soofaloofa", "prompt": "Alias"}
                ],
                "links": [
                    {"rel": "image",
                     "href": "https://api.example.com/player/1234567890/avatar.png",
                     "prompt": "Avatar",
                     "render": "image" },
                    {"rel": "friends",
                     "href": "https://api.example.com/player/1234567890/friends",
                     "prompt": "Friends" }
                ]
            }
        ]
    }
}
```

http://sookocheff.com/post/api/on-choosing-a-hypermedia-format/

# Decision Criteria

- Plain text
  - Ideal for very simple data payloads
- HAL, JSON-LD, Collection+JSON
  - Best choice for JavaScript (e.g. browser-based) clients
  - HAL & JSON-LD preserve domain/document structure; Collection+JSON is message-oriented
  - *http://sookocheff.com/post/api/on-choosing-a-hypermedia-format/*
- HTML
  - Links and forms
  - Viewable in the browser
- Atom
  - Nested media types
  - Viewable in a feed reader
- Remember, you can mix-and-match!
  - Use content-negotiation to service different clients
  - E.g. plain text, HTML and Atom

# HTTP is a (Document Transfer) Application Protocol



[Crown Copyright Reserved.]

Ministry of Transport.

THE
HIGHWAY CODE

Issued by the Minister of Transport
with the authority of Parliament in
pursuance of Section 45 of the
Road Traffic Act, 1930.

LONDON:
PRINTED AND PUBLISHED BY HIS MAJESTY'S STATIONERY OFFICE
To be purchased directly from H.M. Stationery Office at the following addresses:
Adastral House, Kingsway, London, W.C.2; 120, George St., Edinburgh;
York Street, Manchester; 1, St. Andrew's Crescent, Cardiff;
15, Donegall Square West, Belfast;
or through any Bookseller.

1931.

Price 1d. net.

55-166

# HTTP's Uniform Interface

Every resource accessed and manipulated in the same way using:

## Status codes

Coordination

## Headers

Message processing context
Availability and consistency

## Methods

Reliability

## Status Codes Coordinate Exchanges

```
HTTP/1.1 200 OK

HTTP/1.1 201 Created
Location: http://restbucks.com/quotes/1234

HTTP/1.1 202 Accepted
Location: http://restbucks.com/orders/9876

HTTP/1.1 303 See Other
Location: http://restbucks.com/orders/9876

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

# Even 404 Provides Useful Coordination Information

## Client logic (pseudocode)

```
if (GET(/orders/1234) == 200 &&
    GET(/orders/1234/payment) == 200 &&
    GET(/orders/1234/cancellation) == 404 &&
    OPTIONS(/orders/1234/payment).Contains(POST)) {
      applicationState = AwaitingPayment
  }

if (POST(/orders/1234/payment) == 405 &&
    GET(/orders/1234/cancellation) == 404) {
      applicationState = Paid
  }
```

## HTTP Headers

**Parameterize the request**

`Host`

**Information about the client**

`Accept`

**Message metadata**

`Cache-Control, Date`

**Information about the server**

`Server`

**Response generation**

`Vary`

**Further access to the resource**

`ETag`

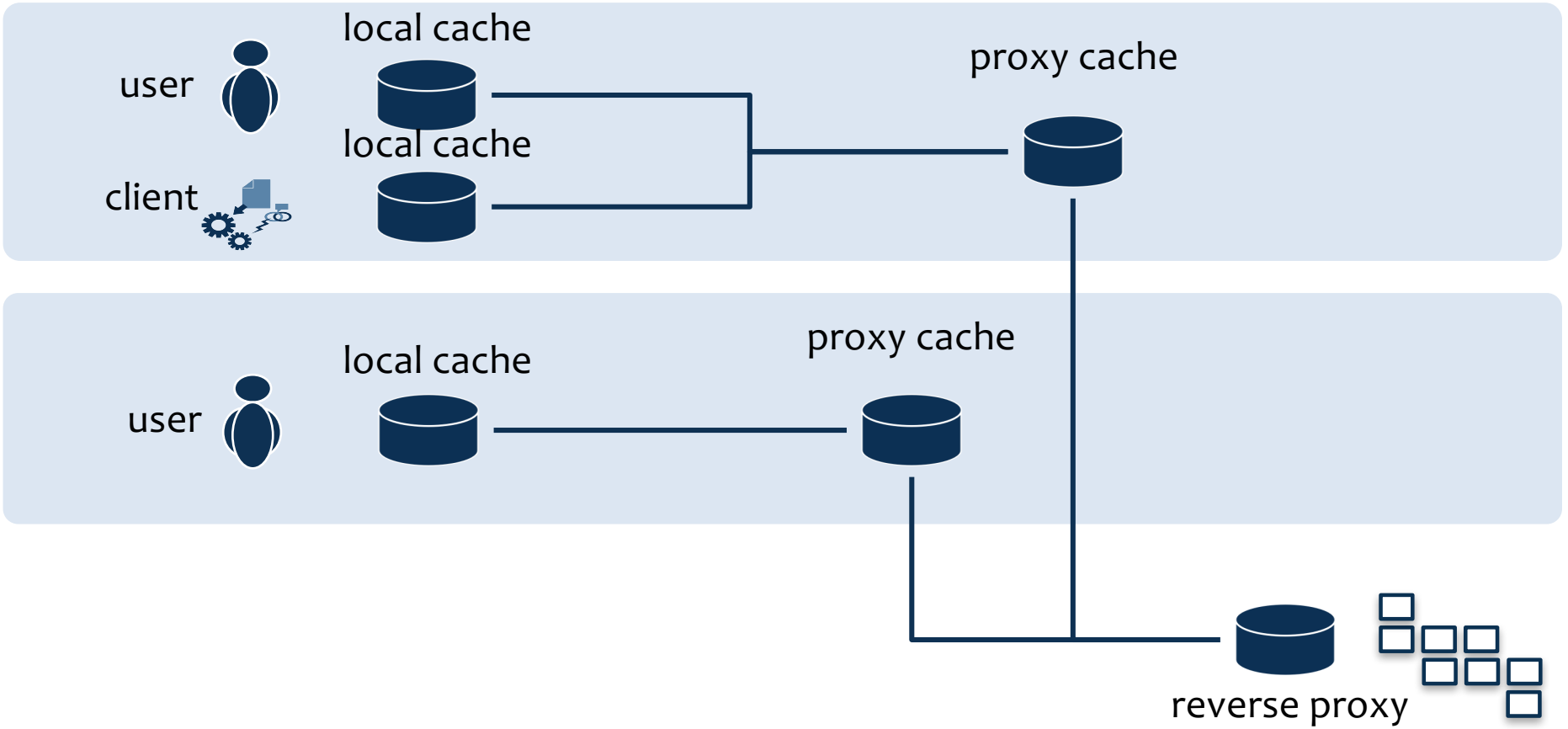
**Entity body metadata**

`Content-Type`

**Resource metadata**

`Last-Modified`

# Caching for Availability and Federated Load




```
Cache-Control: public, max-age=86400
```

## ETags for Consistency

### ETag

Opaque "checksum" of resource state

When resource changes, entity tag changes

### Conditional GET

`If-None-Match: <etag value>`

"Return a full response only if the resource *has* changed"

`304 Not Modified`

### Conditional POST/PUT

`If-Match: <etag value>`

"Process this request only if the resource *hasn't* changed"

`412 Precondition Failed`

# Etag Example – First Request

## Request

```
GET /orders/9876 HTTP/1.1
Host: restbucks.com
```

## Response

```
HTTP/1.1 200 OK
ETag: "4d3e88c9"
Content-Type: application/restbucks+xml

<shop xmlns="http://schemas.restbucks.com/shop"
  xmlns:rb="http://relations.restbucks.com/">
   <status>Awaiting Payment</status>
</shop>
```

# Etag Example – Second Request

## Request

```
GET /orders/9876 HTTP/1.1
Host: restbucks.com
If-None-Match: "4d3e88c9"
```

## Response

```
HTTP/1.1 304 Not Modified
```

# Etag Example – Third Request

## Request

```
GET /orders/9876 HTTP/1.1
Host: restbucks.com
If-None-Match: "4d3e88c9"
```

## Response

```
HTTP/1.1 200 OK
ETag: "5612cfaa"
Content-Type: application/restbucks+xml

<shop xmlns="http://schemas.restbucks.com/shop"
  xmlns:rb="http://relations.restbucks.com/">
   <status>Paid</status>
</shop>
```

# Server-driven content negotiation

Server chooses best representation based on information in the request

## Request

```
GET /updates HTTP/1.1
Host: restbucks.com
Accept: application/atom+xml,application/xhtml+xml
```

# Server-driven content negotiation

Server chooses best representation based on information in the request

## Response

```
HTTP/1.1 200 OK
Cache-Control: max-age=10
Date: Fri, 29 Oct 2010 14:40:45 GMT
Server: Microsoft-IIS/6.0
Vary: Accept
ETag: "081fcddc"
Content-Location: http://restbucks.com/updates.atom
Content-Type: application/atom+xml
Content-Length: 5348
Last-Modified: Fri, 29 Oct 2010 14:40:44 GMT

<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  ...
</feed>
```

# Server-driven content negotiation (continued)

## Selection criteria

`Accept`, `Accept-Charset`, `Accept-Encoding`, `Accept-Language`
Server can use other information in the request (e.g. `User-Agent`)

## `Vary` header

Indicates which parameters were used to select the representation
Caches can use `Vary` to match cached responses with future requests
Beware of varying on headers with wildly differing values (e.g. `Cookie`)

## `Accept` *q* parameter

Indicates client weightings

```
Accept: application/xml; q=0.5, application/atom+xml
```

## Identifying variants

Sometimes useful to identify variants with `Content-Location` header

# Agent-driven content negotiation

Client chooses best representation based on server-supplied list

## Request

```
GET /documentation HTTP/1.1
Host: restbucks.com
```

## Response

```
HTTP/1.1 300 Multiple Choices
Cache-Control: public
ETag: "c549995b"
Content-Length: 0
Link: </documentation.txt>;type=text/plain,
      </documentation.html>;type=text/html,
      </documentation.pdf>;type=application/pdf
```

## Request

```
GET /documentation.html HTTP/1.1
Host: restbucks.com
```

## Response

```
HTTP/1.1 200 OK
...
```

## Reliability by convention

**Safe**

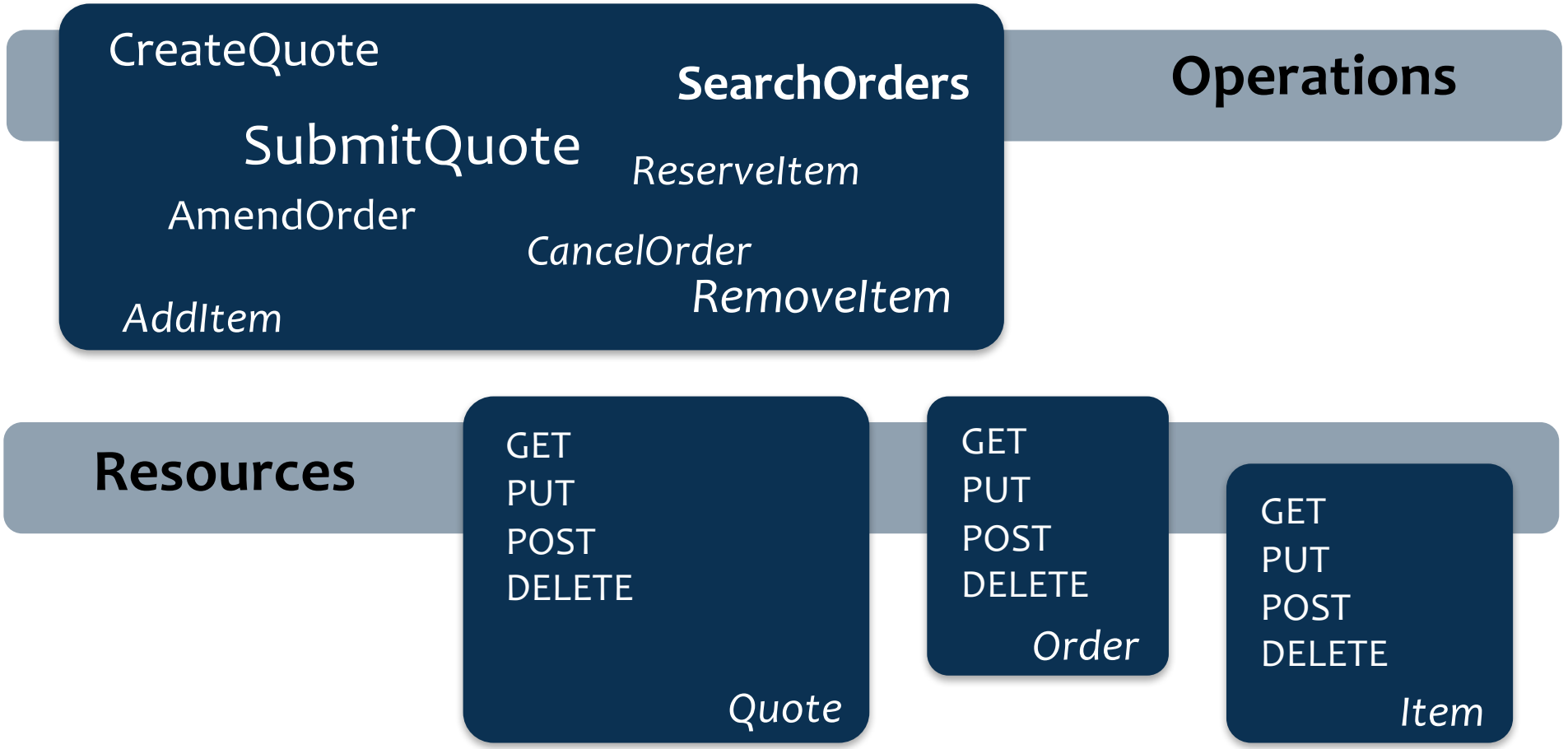
No side-effects for which client is responsible

**Idempotent**

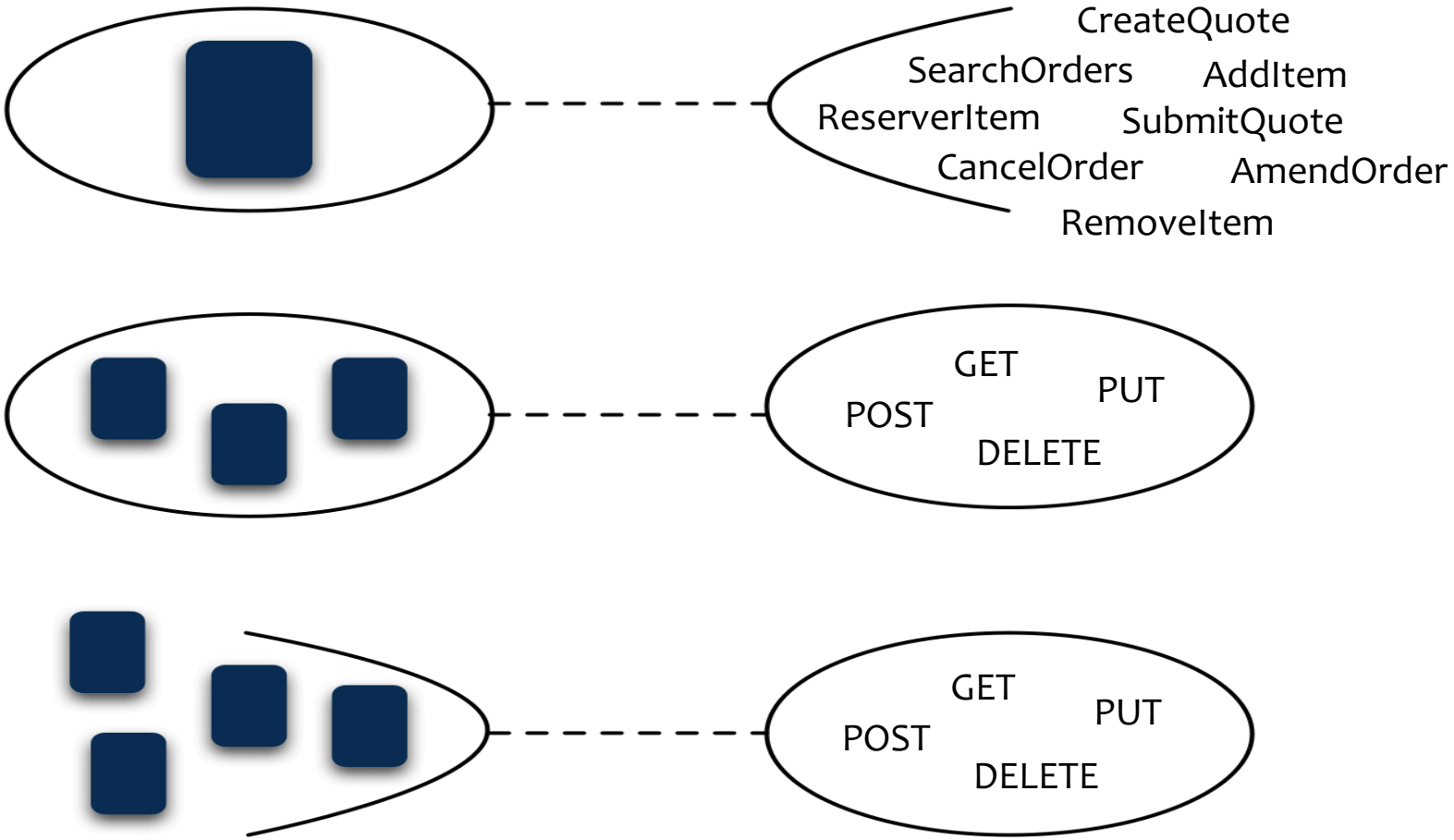
OK to repeat in event of failures

| Verb | Safe? | Idempotent? | Description |
|---|---|---|---|
| `GET` | ■ | ■ | Retrieve representation |
| `PUT` | | ■ | Store representation |
| `POST` | | | Create resource, handle data, annotate resource |
| `DELETE` | | ■ | Delete resource |

# Operation-Oriented vs. Resource-Oriented Design



# Specialization and Innovation Depend on an Open Set



## HTTP/2

- Binary rather than textual protocol
  - compact, less error prone than textual
- Fully multiplexed, instead of ordered and blocking
  - Eliminates "head-of-line blocking" with large responses
- One connection for parallelism
  - Mitigate against buffers overflows in network, congestion and retransmits
- Header compression
  - Reduce number of TCP round trips to get headers onto the wire
- Server push
  - Allows servers to “push” responses proactively into client caches

## Summary

- Web is a stateless, client-server, layered architecture
  - Global reach and evolvability achieved through uniform, coarse-grained interactions (rather than fine-grained, application-specific operations)
- Clients manipulate resources by exchanging documents (resource representations)
  - Resources identified and addressed using URIs
  - Resource representations use standardised media types
- Document coordination achieved using a document transfer application protocol (HTTP)


Tech Interlude
URI Templates

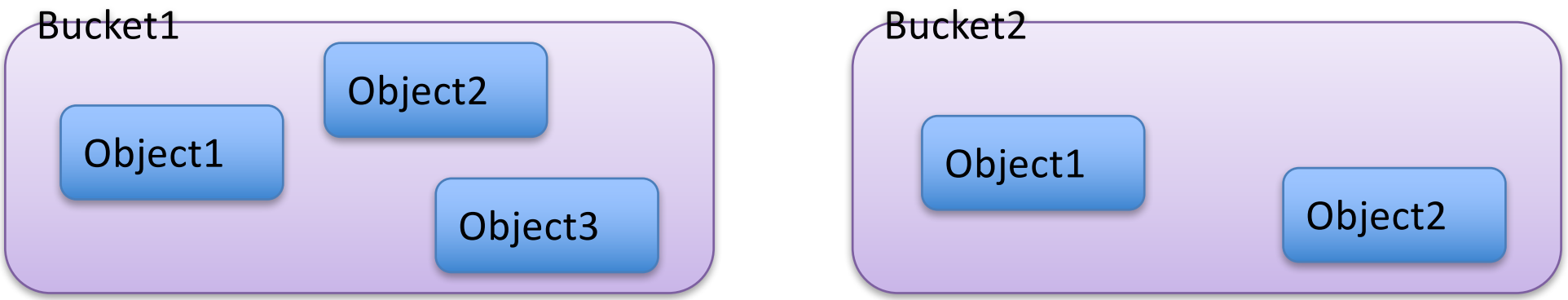
## Conflicting URI Philosophies

- URIs should be descriptive, predictable?
  - http://spreadsheet/cells/a2,a9
  - http://jim.webber.name/2007/06.aspx
    - Convey some ideas about how the underlying resources are arranged
      - Can infer http://spreadsheet/cells/b0,b10 and http://jim.webber.name/2005/05.aspx for example
    - Nice for programmatic access, but may introduce coupling
- URIs should be opaque?
  - http://tinyurl.com/6
  - TimBL says "opaque URIs are cool"
    - Convey no semantics, can't infer anything from them
      - Don't introduce coupling

## URI Templates, in brief

- Use URI templates to make your resource structure easy to understand
- For Amazon S3 (storage service) it's easy:

```
http://s3.amazon.com/{bucket-name}/{object-name}
```



## URI Templates are Easy!

- Take the URI:

```
http://restbucks.com/orders?{order_id}
```

- You could do the substitution and get a URI:

```
http://restbucks.com/orders?1234
```

- Can easily make more complex URIs too
  - Mixing template and non-template sections

```
http://restbucks.com/{orders}/{shop}/{year}/{month}.xml
```

- Use URI templates client-side to compute server-side URIs
  - But beware this introduces coupling!

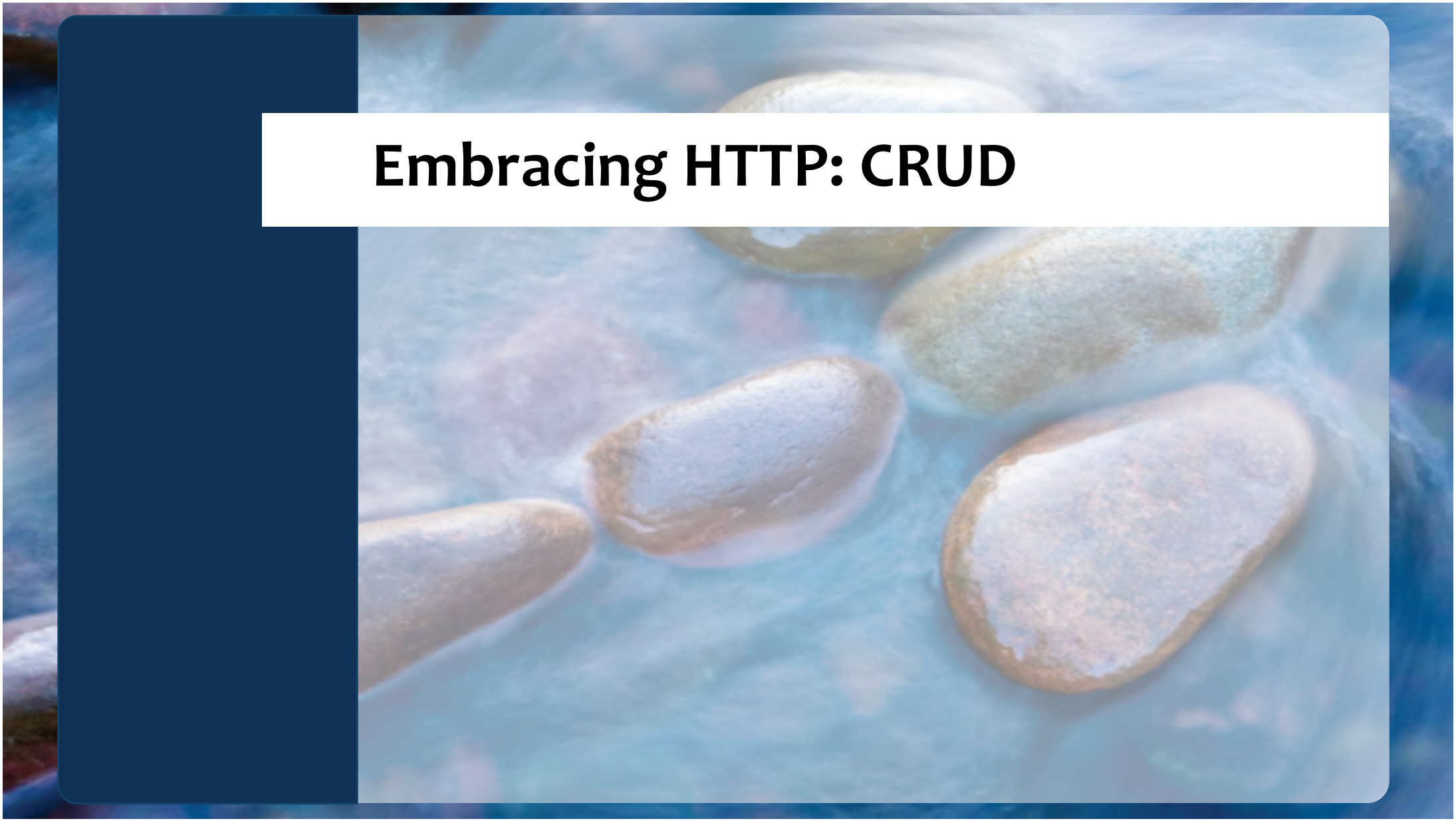
## Why URI Templates?

- Regular URIs are a good idiom in Web-based services
  - Helps with understanding, self documentation
- They allow users to infer a URI
  - If the pattern is regular
- URI templates formalise this arrangement
  - And advertise a template rather than a regular URI

## URI Templates Pros and Cons

- Everything interesting in a Web-based service has a URI
- Remember, two schools of thought:
  - Opaque URIs are cool (Berners-Lee)
  - Transparent URIs are cool (everyone else)
- URI templates present two core concerns:
  - They invite clients to invent URIs which may not be honoured by the server
  - They increase coupling since servers must honour forever any URI templates they've advertised
- Use URI templates sparingly, and with caution
  - Entry point URIs only is a good rule of thumb

# Embracing HTTP: CRUD



## Domain Overview: Restbucks

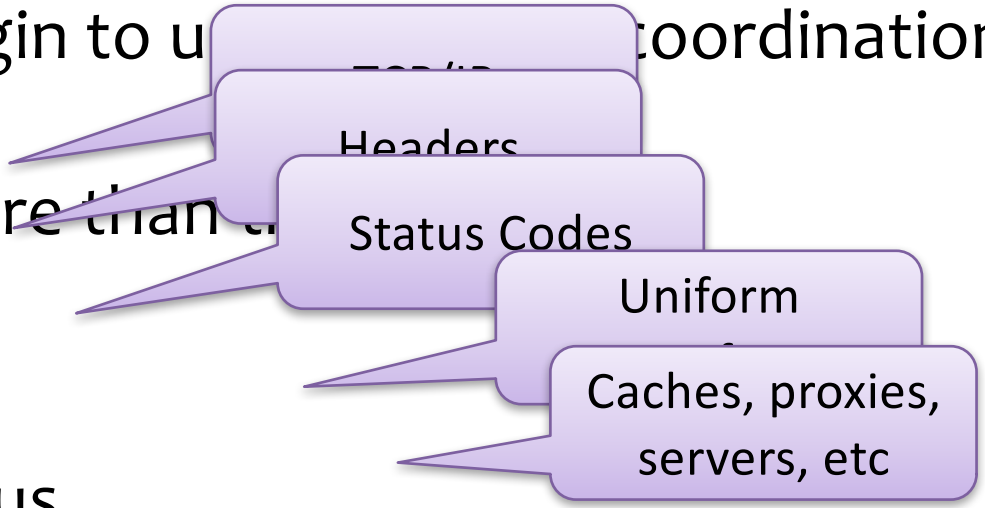
- Throughout *REST in Practice* we used an example drawn from conventional commerce
- A coffee shop called RESTbucks*
  - Influenced by Gregor Hohpe's work "Startbucks does not use Two Phase Commit"
- Everybody knows how a coffee shop works, so the domain doesn't get in the way
- Although the example is a little contrived, it has plenty of scope for dealing with real-world systems-of-microservices
- Let's start with the protocol for order management…

## CRUD APIs

- This session is about CRUD/HTTP fundamentals
- We see how HTTP is used to manipulate an object in a remote service
- But we also consider the limitations:
  - Coupling (design time, runtime)
  - Limited protocol expressivity
  - We'll explore these with a hands-on exercise

## Using the Web

- SOAP services use the Web as a transport
- CRUD services begin to u coordination support
- But the Web is more than t
  - Transport, plus
  - Metadata, plus
  - Fault model, plus
  - Component model, plus
  - Runtime environment, plus…

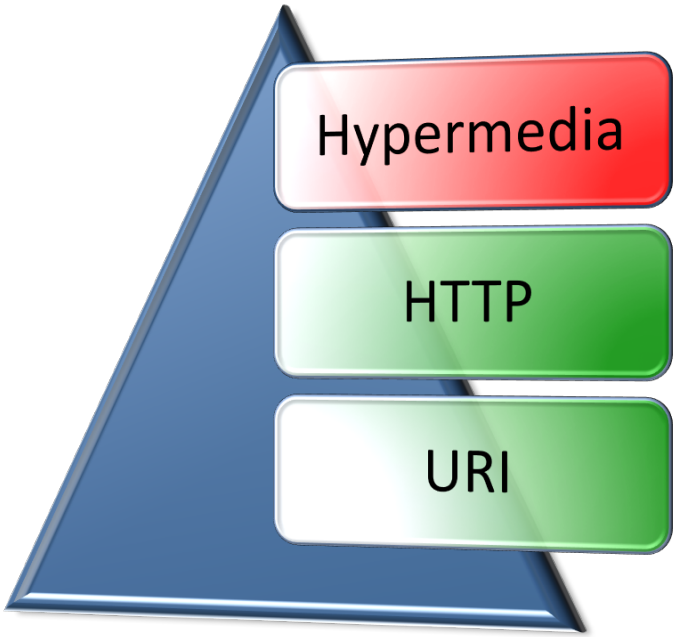


## CRUD Resource Lifecycle

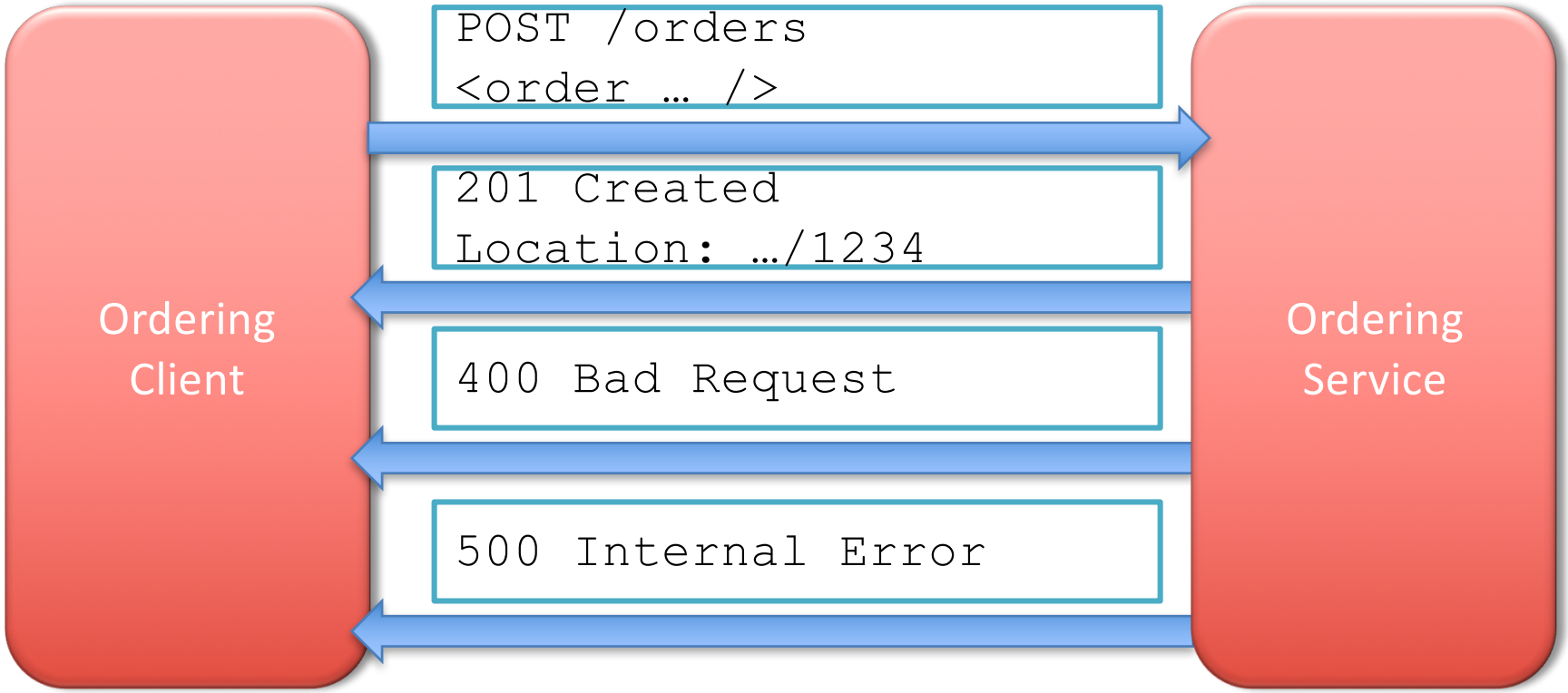
- The resource is created with `POST`
- It's read with `GET`
- And updated via `PUT`
- Finally it's removed using `DELETE`

# Richardson Model Level 2

- Lots of URIs
- Understands HTTP!
- No hypermedia



# Create with POST



## 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 /orders HTTP/1.1
Host: restbucks.example.com
Content-Type:application/vnd.restbucks+xml
Content-Length: 225

<order xmlns="http://schemas.restbucks.com">
  <location>takeAway</location>
  <item>
    <drink>latte</drink>
    <milk>whole</milk>
    <size>small</size>
  </item>
</order>
```

Verb, path, and HTTP version

Restbucks-specific XML content

Content (again Restbucks XML)

## POST Response

```
HTTP/1.1 201 Created
Location: /orders/1234
```

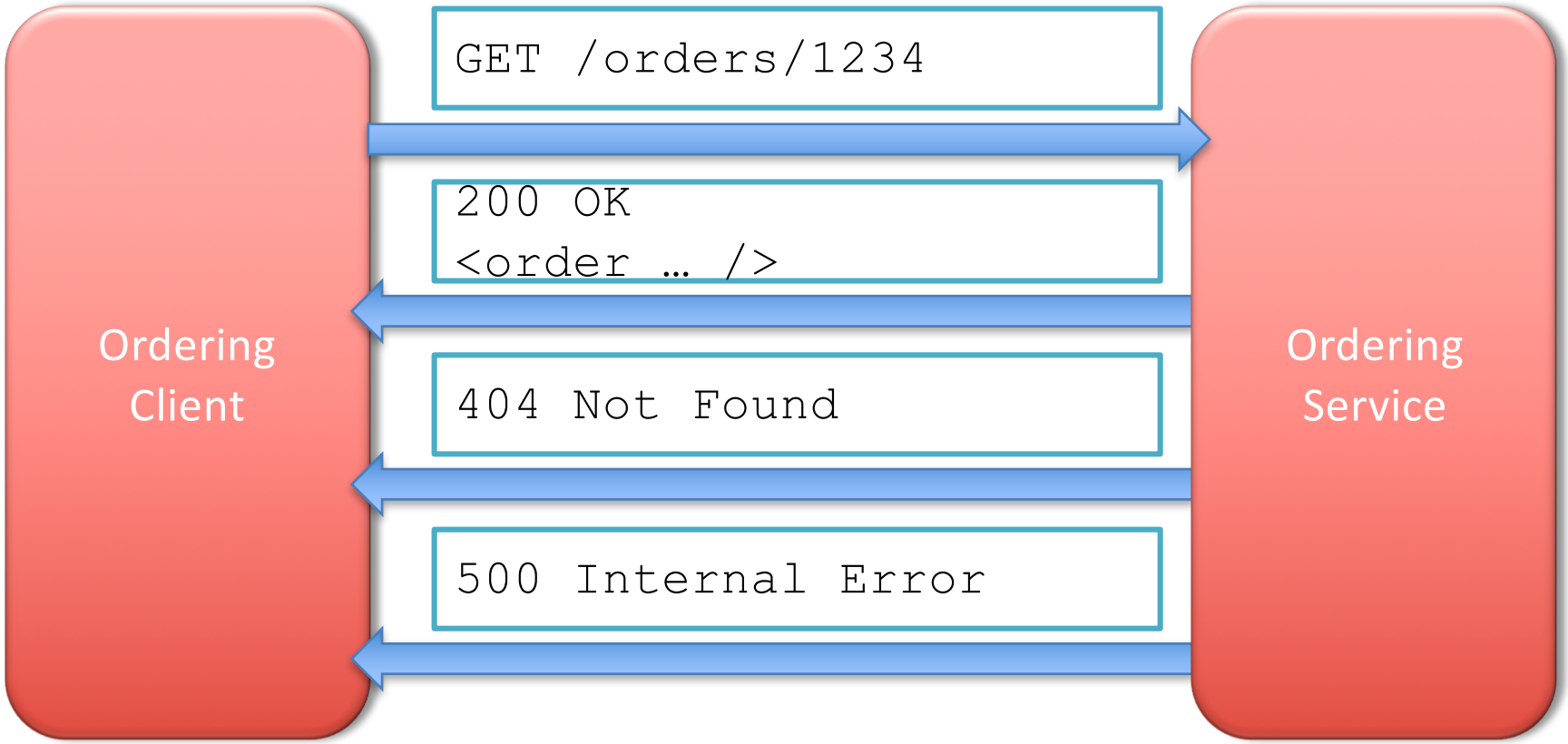
## When `POST` goes wrong

- We may be 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 `POST`ing again
  - May involve a `PUT` to rectify server-side resources in-place
    - `PUT` coming soon

# POST Implementation with a Servlet

```
protected void doPost(HttpServletRequest request,
                      HttpServletResponse response) {
  try {
    Order order = extractOrderFromRequest(request);
    String internalOrderId = OrderDatabase.getDatabase().saveOrder(order);
    response.setHeader("Location", computeLocationHeader(request,
                       internalOrderId));
    response.setStatus(HttpServletResponse.SC_CREATED);
  } catch(Exception ex) {
    response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
  }
}
```

# Read with GET



## GET Semantics

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

Library of congress catalogue incident!

## GET Exemplified

```
GET /orders/1234 HTTP/1.1
Accept: application/vnd.restbucks+xml
Host: restbucks.com
```

# GET Response

```
HTTP/1.1 200 OK
Content-Length: 232
Content-Type: application/vnd.restbucks+xml
Date: Wed, 19 Nov 2008 21:48:10 GMT

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <items>
    <item>
      <name>latte</name>
      <quantity>1</quantity>
      <milk>whole</milk>
      <size>small</size>
    </item>
  </items>
  <status>pending</status>
</order>
```

## When GET Goes wrong

- Simple!
  - Just 404 – the resource is no longer available

```
HTTP/1.1 404 Not Found
Content-Type: application/vnd.restbucks+xml
Content-Length: 952
Date: Sat, 20 Dec 2008 19:01:33 GMT
```

- Are you sure?
  - `GET` again!
- `GET` is safe and idempotent
  - Great for crash recovery scenarios!

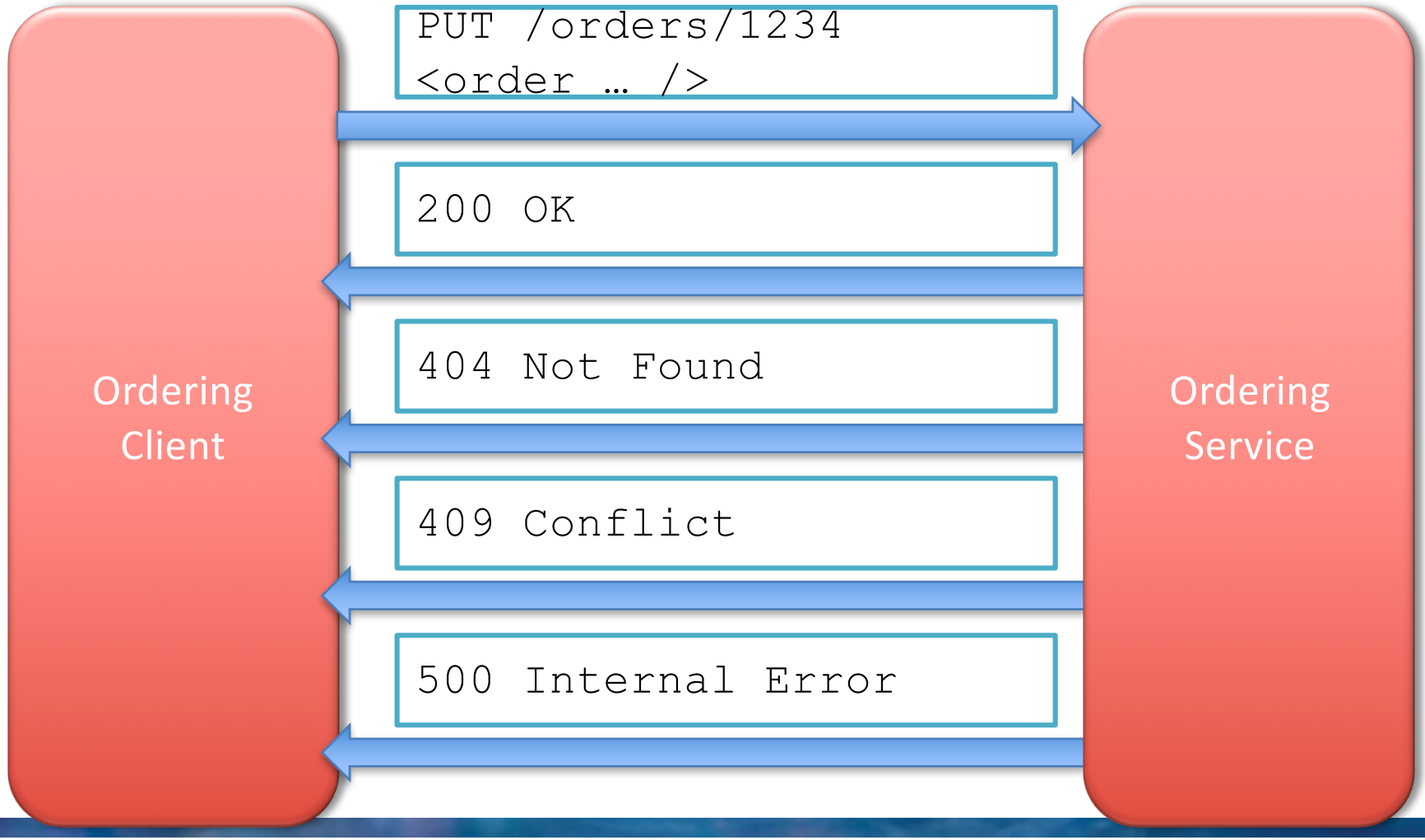
## Idempotent Behaviour

- An action with no side affects
  - Comes from mathematics
- In practice means two things:
  - A safe operation is one which changes no state at all
    - E.g. HTTP `GET`
  - An idempotent operation is one which updates state in an absolute way
    - 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 JAX-RS Implementation

```
@Path("/")
public class OrderingService {
  @GET
  @Produces("application/xml")
  @Path("/{orderId}")
  public String getOrder(@PathParam("orderId") String orderId) {
    try {
      Order order = OrderDatabase.getDatabase().getOrder(orderId);
      if (order != null) {
        return xstream.toXML(order);
      } else {
        throw new WebApplicationException(404);
      }
    } catch (Exception e) {
      throw new WebApplicationException(500);
    }
  }
  // Remainder of implementation omitted for brevity
}
```

# Update with PUT



## `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
- `PUT` is idempotent
  - Makes absolute changes
- But is not safe
  - It changes state!

# PUT Request

```
PUT /orders/1234 HTTP/1.1
Host: restbucks.com
Content-Type: application/xml
Content-Length: 386

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <items>
    <item>
      <milk>whole</milk>
      <name>latte</name>
      <quantity>2</quantity>
      <size>small</size>
    </item>
    <item>
      <milk>whole</milk>
      <name>cappuccino</name>
      <quantity>1</quantity>
      <size>large</size>
    </item>
  </items>
  <status>preparing</status>
</order>
```

Updated content

## PUT Response

```
HTTP/1.1 200 OK
Date: Sun, 30 Nov 2008 21:47:34 GMT
Content-Length: 0
```

Minimalist response contains no entity body

# 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

```
HTTP/1.1 409 Conflict
Date: Sun, 21 Dec 2008 16:43:07 GMT
Content-Length:382

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <items>
    <item>
      <milk>whole</milk>
      <name>latte</name>
      <quantity>2</quantity>
      <size>small</size>
    </item>
    <item>
      <milk>whole</milk>
      <name>cappuccino</name>
      <quantity>1</quantity>
      <size>large</size>
    </item>
  </items>
  <status>served</status>
</order>
```

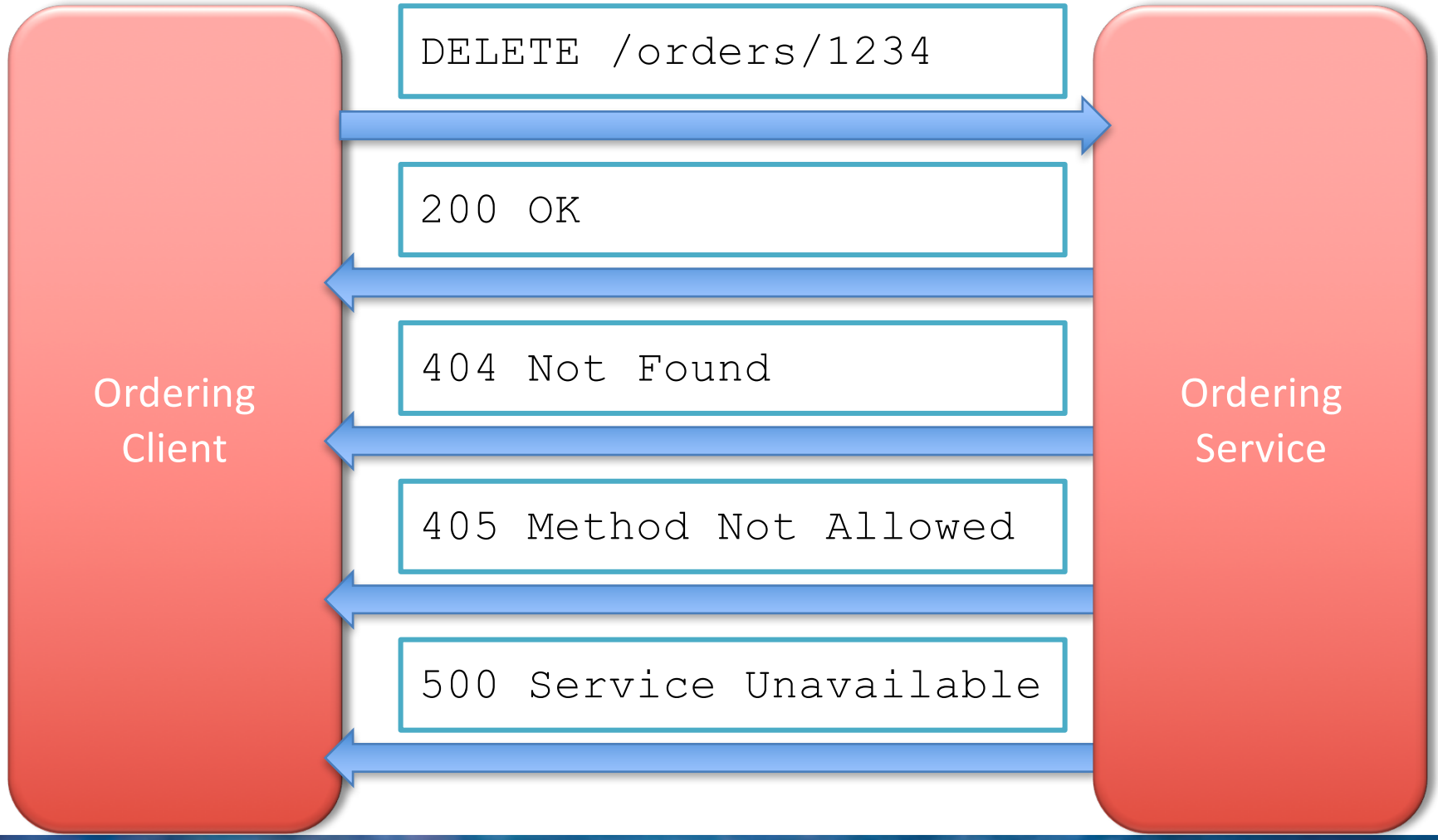
# WCF Implementation for PUT

```
[ServiceContract]
public interface IOrderingService
{
  [OperationContract]
  [WebInvoke(Method = "PUT", UriTemplate =
  "/orders/{orderId}")]
  void UpdateOrder(string orderId, Order order);

  // …
}
```

# WCF Serializable Types

```
[DataContract(Namespace = "http://schemas.restbucks.com/order", Name = "order")]
public class Order
{
  [DataMember(Name = "location")]
  public Location ConsumeLocation
  {
    get { return location; }
    set { location = value; }
  }

  [DataMember(Name = "items")]
  public List<Item> Items
  {
    get { return items; }
    set { items = value; }
  }

  [DataMember(Name = "status")]
  public Status OrderStatus
  {
    get { return status; }
    set { status = value; }
  }
  // …
}
```

# Remove with DELETE



## DELETE Semantics

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

```
DELETE /orders/1234 HTTP/1.1
Host: restbucks.com
```

- Response

```
HTTP/1.1 200 OK
Content-Length: 0
Date: Tue, 16 Dec 2008 17:40:11 GMT
```

This is important for decoupling implementation details from resources

## When DELETE goes wrong

- Simple case, DELETE again!
  - DELETE is idempotent!
  - DELETE once, DELETE 10 times has the same effect: one deletion

```
HTTP/1.1 404 Not Found
Content-Length: 0
Date: Tue, 16 Dec 2008 17:42:12 GMT
```

# When DELETE goes Really Wrong

- Some 4xx responses indicate that deletion isn't possible
  - The state of the resource isn't compatible
  - Try forward/backward compensation instead

```
HTTP/1.1 409 Conflict
Content-Length: 379
Date: Tue, 16 Dec 2008 17:53:09 GMT

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <items>
    <item>
      <name>latte</name>
      <milk>whole</milk>
      <size>small</size>
      <quantity>2</quantity>
    </item>
    <item>
      <name>cappuccino</name>
      <milk>skim</milk>
      <size>large</size>
      <quantity>1</quantity>
    </item>
  </items>
  <status>served</status>
</order>
```

Can't delete an order that's already served

**CRUD does not mean Worthless**

## Exercise

- Write a client for the service hosted on this laptop
- It's an ordering service and you can CRUD the state of an order
- Here are some URI templates you can use:

```
/
/order
/order/{order_id}
/receipt/{order_id}
/payment
/payment/{payment_id}
```

## Network

- There's an ad-hoc network called RESTinPractice
- Join it
- Then see if you can point your browser at:
  - `http://169.254.147.226:8080`

# Exercise

- Order XML* template you can use:

```
<order xmlns="http://schemas.restbucks.com">
  <item>
    <milk>{whole | semi | skim | none}</milk>
    <size>{small | large | medium}</size>
    <drink>
        {espresso | latte |cappuccino | americano}
    </drink>
  </item>
  <location>takeaway</location>
</order>
```

*

**application/vnd.restbucks+xml**

# Exercise

- Payment template

```
<payment xmlns="http://schemas.restbucks.com">
  <amount>{x.y}</amount>
  <cardholderName>{cardholder name}</cardholderName>
  <cardNumber>{card number no spaces}</cardNumber>
  <expiryMonth>{mm}</expiryMonth>
  <expiryYear>{yy}</expiryYear>
</payment>
```

## Curl (if you don't want to "code")

- Windows users:
  - `https://curl.haxx.se/download.html`
- Single-line example:

```
curl -X POST -d @order.xml
http://localhost:8080/order
-H "Content-Type:
application/vnd.restbucks+xml"
-H "Accept:
application/vnd.restbucks+xml" -v
```


THOUGHT WE WERE DOING JSON FOR MICROSERVICES
GOT XML

## 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
  - Hence frameworks like Microsoft's Astoria
- 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!

# Tech Interlude

## Hypermedia Formats

## Media Types Rule!

- The Web's contracts are expressed in terms of media types and link relations
  - If you know the type, you can process the content
- Some types are special because they work in harmony with the Web
  - We call these "hypermedia formats"

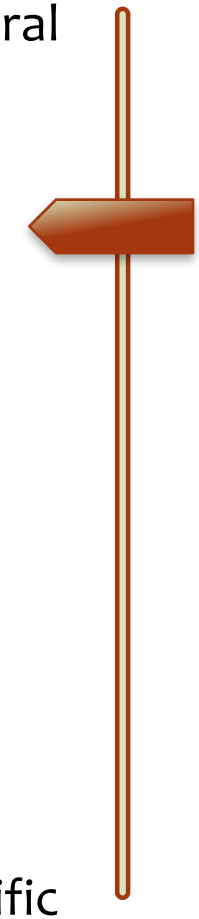
# (Hyper) media types

Standardised media type

**Processing model**

- Hypermedia controls (links and forms)
- Supported operations (methods, headers and status codes)
- Representation formats (may include schemas)

General



Specific

Compose application-specific behaviours on top of the handling of standardised media types

## Other Resource Representations

- Remember, XML is not the only way a resource can be serialised
  - Remember the Web is based on **RE**presentational **S**tate **T**ransfer
- The choice of representation is left to the implementer
  - Can be a standard registered media type
  - Or something else
- But there is a division on the Web between two families
  - Hypermedia formats
    - Formats which host URIs and links
  - Regular formats
    - Which don't

# Plain Old XML is not Hypermedia Friendly

```
HTTP/1.1 200 OK
Content-Length: 227
Content-Type: application/xml
Date: Wed, 19 Nov 2008 21:48:10 GMT

<order xmlns="http://schemas.restbucks.com/order">
  <location>takeAway</location>
  <items>
    <item>
      <name>latte</name>
      <quantity>1</quantity>
      <milk>whole</milk>
      <size>small</size>
    </item>
  </items>
  <status>pending</status>
</order>
```

Where are the links?
Where's the protocol?

## So what?

- How do you know the next thing to do?
- How do you know the resources you're meant to interact with next?
- In short, how do you know the service's protocol?
  - Turn to WADL? Yuck!
  - Read the documentation? Come on!
  - URI Templates? Tight Coupling!

## URI Templates are NOT a Hypermedia Substitute

- Often URI templates are used to advertise all resources a service hosts
  - Do we really need to advertise them all?
- This is verbose
- This is out-of-band communication
- This encourages tight-coupling to resources through their URI template
- This has the opportunity to cause trouble!
  - Knowledge of “deep” URIs is baked into consuming programs
  - Services encapsulation is weak and consumers will program to it
  - Service will change its implementation and break consumers

## Bad Ideas with URI Templates

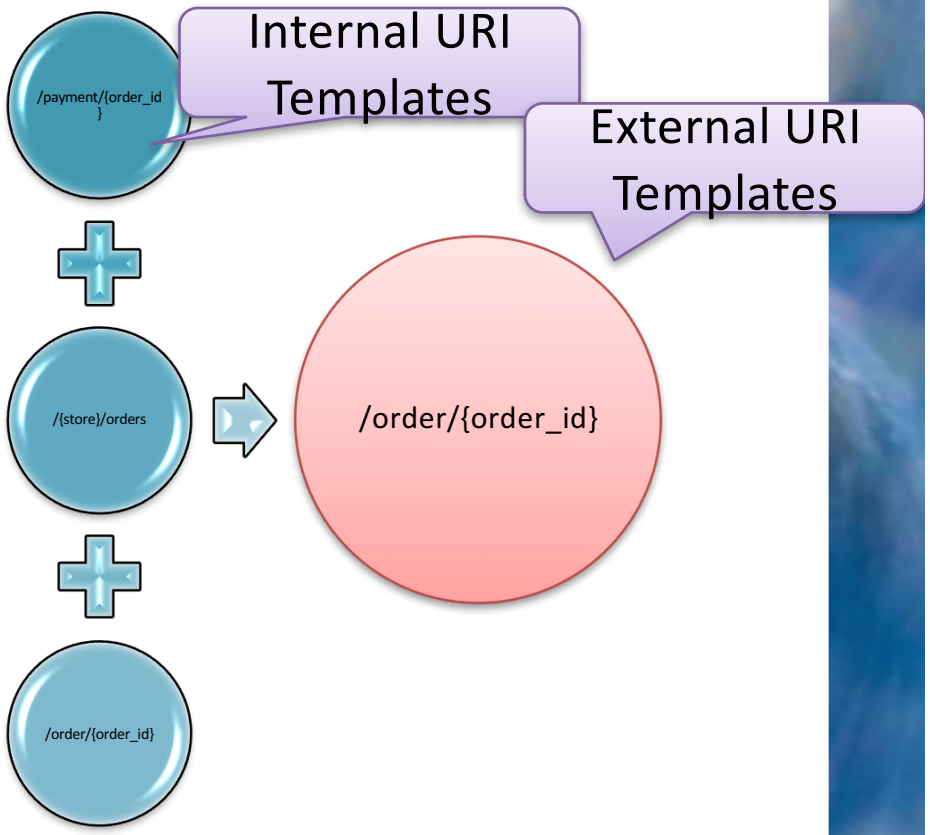
- Imagine we're created an order, what next?
- We could share this URI template:
  - `http://restbucks.com/payment/{order_id}`
- The `order_id` field should match the order ID that came from the restbucks service
  - Sounds great!
- But what if Restbucks outsources payment?
  - Change the URI for payments, break the template, break consumers!
  - D'oh!
- Be careful what you share!

## Better Ideas for URI Templates: Entry Points

- Imagine that we have a well-known entry point to our service
  - Which corresponds to a starting point for a protocol
- Why not advertise that with a URI template?
- For example:
  - `http://restbucks.com/signIn/{store_id}/{barista_id}`
- Changes infrequently
- Is important to Restbucks
- Is transparent, and easy to bind to

# Better Ideas for URI Templates: Documentation!

- Services tend to support lots of resources
- We need a shorthand for talking about a large number of resources easily
- We can use a URI template for each “type” of resource that a service (or services) supports
- But we don’t share this information with others
  - Don’t violate encapsulation!



## `application/xml` or `application/json` not enough

- Remember that HTTP is an application protocol
  - Headers and representations are intertwined
  - Headers set processing context for representations
- Remember that `application/xml` and `application/json` have specific processing models
  - Which doesn't include understanding the semantics of links
- Remember if a representation is declared in the `Content-Type` header, you must treat it that way
  - HTTP is an application protocol – did you forget already? ☺
- We need real hypermedia formats!

## Hypermedia Formats

- Standard
  - Wide “reach”
  - Software agents already know how to process them
  - But sometimes need to be shoe-horned
- Self-created
  - Can craft specifically for domain
  - Semantically rich
  - But lack reach

## Two Common Hypermedia Formats: XHTML and ATOM

- Both are commonplace today
- Both are hypermedia formats
  - They contain links
- Both have a processing model that explicitly supports links
- Which means both can describe *protocols*…

## XHTML

- XHTML is just HTML that is also XML
- For example:

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xml:lang="en" lang="en"
  xmlns:r="http://restbucks.org">
  <head>
    <title> XHTML Example </title>
  </head>
  <body>
    <p>
      ...
```

## What's the big deal with XHTML?

- It does two interesting things:
  1. It gives us document structure
  2. It gives us links
- So?
  1. We can understand the format of those resources
  2. We can discover other resources!
- How?
  1. Follow the links!
  2. Encode the resource representation as "normal" XML in your XHTML documents
- Contrast this with the Atom and APP approach...similar!

# XHTML in Action

```
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
   <div class="order">
      <p class="location">takeAway</p>
      <ul class="items">
        <li class="item">
          <p class="name">latte</p>
          <p class="quantity">1</p>
          <p class="milk">whole</p>
          <p class="size">small</p>
        </li>
      </ul>
      <a href="http://restbucks.com/payment/1234"
         rel="payment">payment</a>
   </div>
</body>
</html>
```

Business data

"Hypermedia Control"

## `application/xhtml+xml`

- Can ask which verb the resource at the end of the link supports
  - Via HTTP OPTIONS
- No easy way to tell what each link actually does
  - Does it buy the music?
  - Does it give you lyrics?
  - Does it vote for a favourite album?
- We lack semantic understanding of the linkspace and resources
  - But we have microformats for that semantic stuff!
- Importantly XHTML is a hypermedia format
  - It contains hypermedia controls that can be used to describe protocols!

# Atom Syndication Format

- We'll study this in more depth later, but for now…
- The application/atom+xml media type is hypermedia aware
- You should expect links when processing such representa
- And be pre with them!

```
HTTP/1.1 200 OK
Content-Length: 342
Content-Type: application/atom+xml
Date: Sun, 22 Mar 2009 17:04:10 GMT

<entry xmlns="http://www.w3.org/2005/Atom">
  <title>Order 1234</title>
  <link rel="payment"
    href="http://restbucks.com/payment/1234"/
    >
  <link rel="special-offer"
    href="http://restbucks.com/offers/freeCoo
    kie"/>
  <id>http://restbucks.com/order/1234</id>
  <updated>2009-03-22T16:57:02Z</updated>
  <summary>1x Cafe Latte</summary>
  </try>
```

Links to other resources, a nascent protocol

## `application/atom+xml`

- No easy way to tell what each link actually does
  - But look at the way the `rel` attribute is being used
  - Can we inject semantics there?
- Atom is a hypermedia format
  - Both feeds and entries contains hypermedia controls that can describe protocols

# Tech Interlude

## Semantics

## Microformats

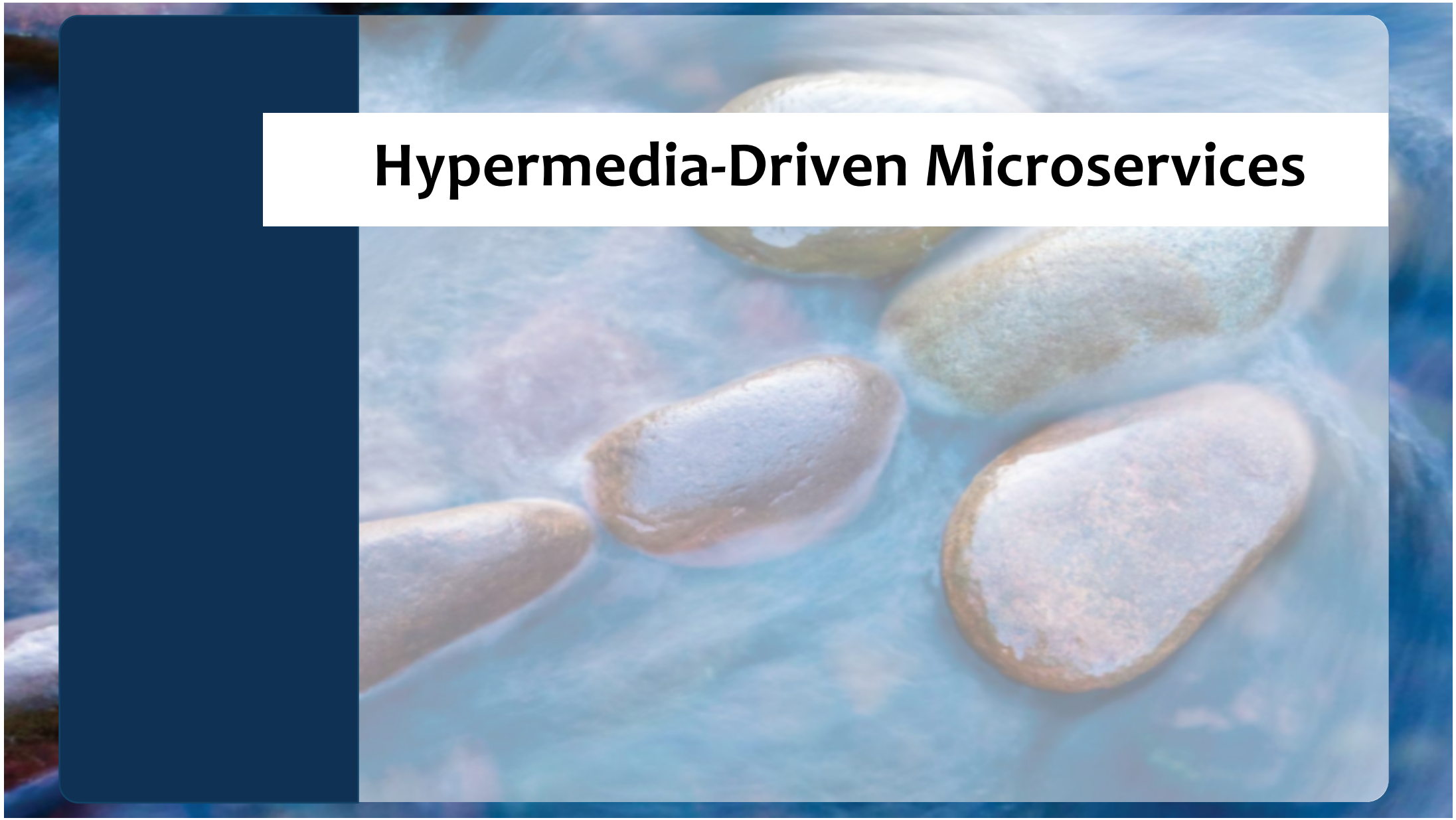
- Microformats are an example of little “s” semantics
- Innovation at the edges of the Web
  - Not by some central design authority (e.g. W3C)
- Started by embedding machine-processable elements in Web pages
  - E.g. Calendar information, contact information, etc
  - Using existing HTML features like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

## Semantic versus semantic

- Semantic Web is top-down
  - Driven by the W3C with extensive array of technology, standards, committees, etc
  - Has not currently proven as scalable as the visionaries hoped
    - RDF tripples have been harvested and processed in private databases
- Microformats are bottom-up
  - Little formal organisation, no guarantee of interoperability
  - Popular formats tend to be adopted (e.g. hCard)
  - Easy to use and extend for our systems
  - Trivial to integrate into current and future programmatic Web systems

## Microformats and Resources

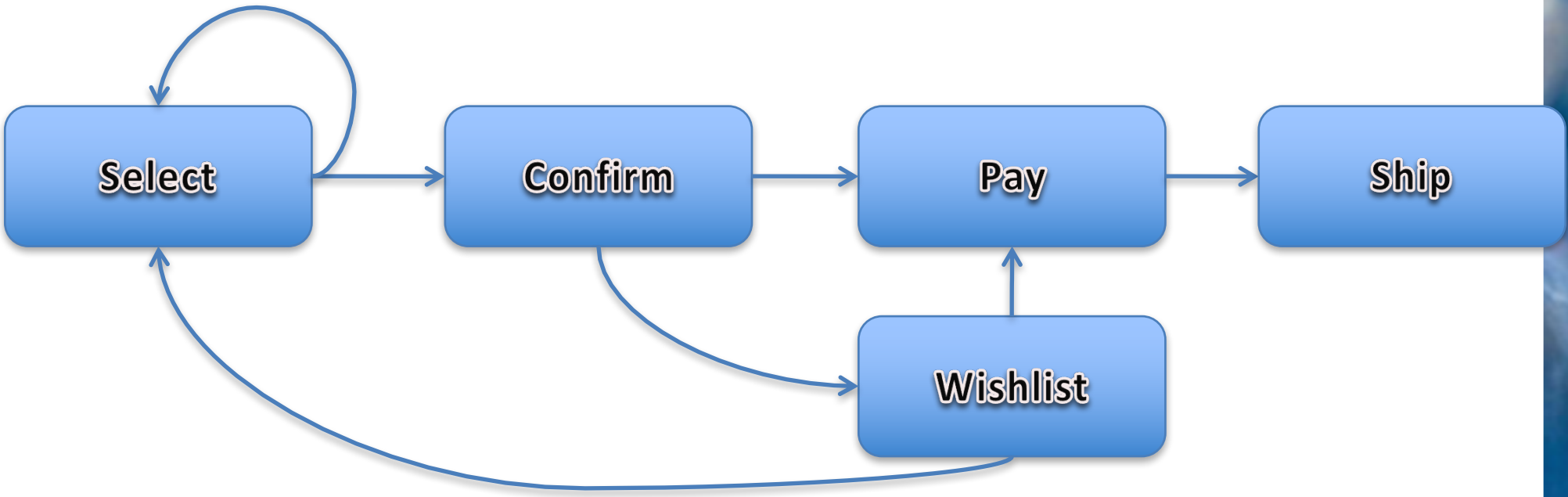
- Use Microformats to structure resources where formats exist
  - I.e. Use hCard for contacts, hCalendar for data
- Create your own formats (sparingly) in other places
  - Annotating links is a good start
  - `<link rel="withdraw.cash" .../>`
  - `<link rel="service.post" type="application/atom+xml"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


Hypermedia-Driven Microservices

## Links

- Connectedness is good in Web-based systems
- Resource representations can contain other URIs
- Links advertise opportunities for state transitions
- Application (conversation) state is captured in terms of these states

# Links advertise State Transitions



## Links for APIs

```
<confirm xmlns="...">
<link rel="payment"
   href="https://pay"
  type="application/xml"/>
<link rel="postpone"
   href="https://wishlist"
  type="application/xml"/>
</confirm>
```

- Following a link causes an action to occur
- This is the start of a state machine!
- Links lead to other resources which also have links
- Can make this stronger with semantics
  - Microformats

# Anatomy of a link

```
<drink>
  <name>latte</name>
  <milk>whole</milk>
  <size>small</size>
  <link
    href="http://restbucks.com/payment/1234"
    rel="http://relations.restbucks.com/payment"
    type="application/vnd.restbucks+xml"/>
</drink>
```

**Address**
WHERE does the linked resource reside?

**Information processing**
WHAT form does the linked resource take?

**Semantic context**
WHY access the linked resource?

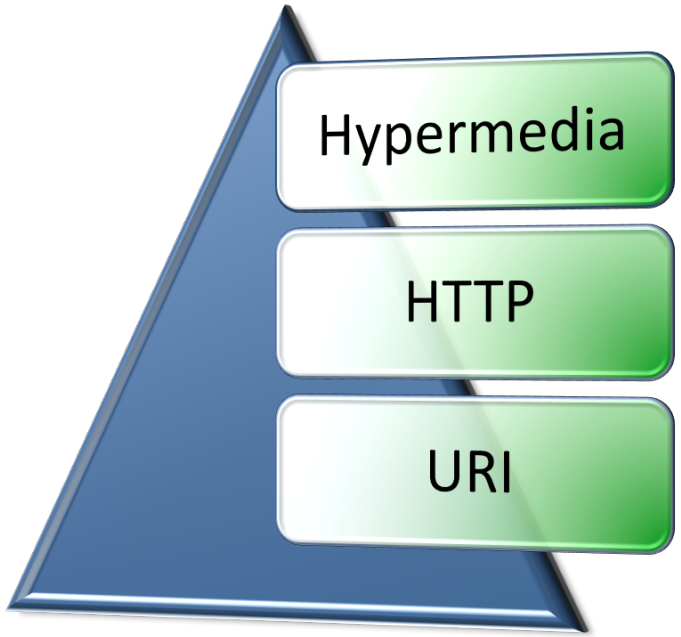
## We have a framework!

- The Web gives us a processing and metadata model
  - Verbs and status codes
  - Headers
- Gives us metadata contracts or Web “APIs”
  - URI Templates
  - Links
- Strengthened with semantics
  - Little “s”


The Web is a platform
Microformats
Verbs
hypermedia
Status codes
Semantics
Media Types

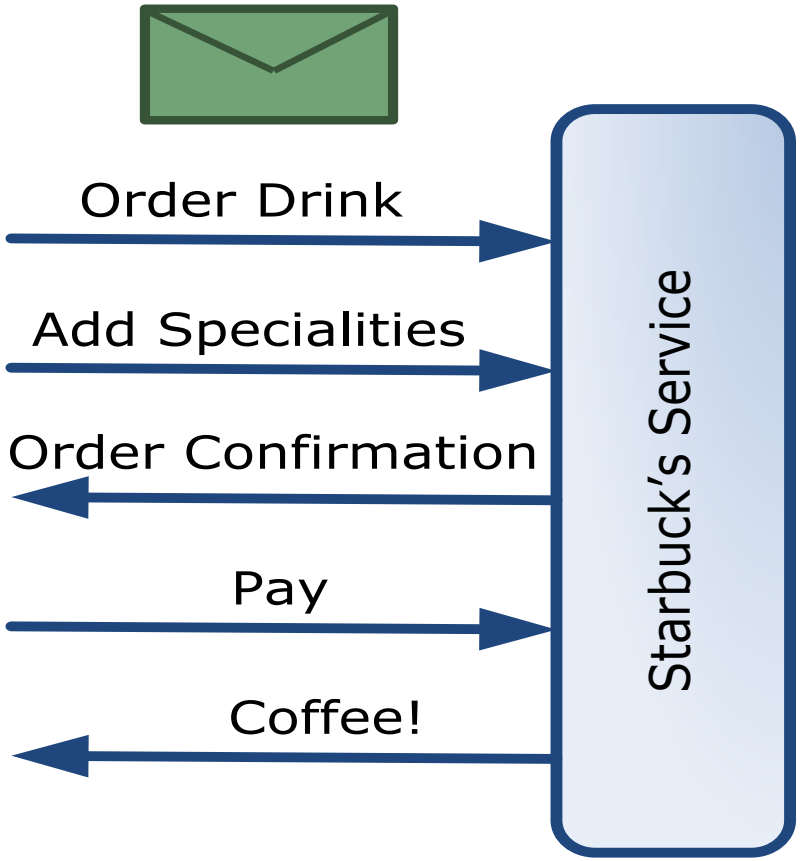
## Richardson Model Level 3

- Lots of URIs that address resources
- Embraces HTTP as an application protocol
- Resource representations and formats identify other resources
  - Hypermedia at last!



## Workflow and MOM

- With Web Services we exchange messages with the service
- Resource state is hidden from view
- Conversation state is all we know
  - Advertise it with SSDL, BPEL
- Uniform interface, roles defined by SOAP
  - No “operations”



## Hypermedia Describes Protocols!

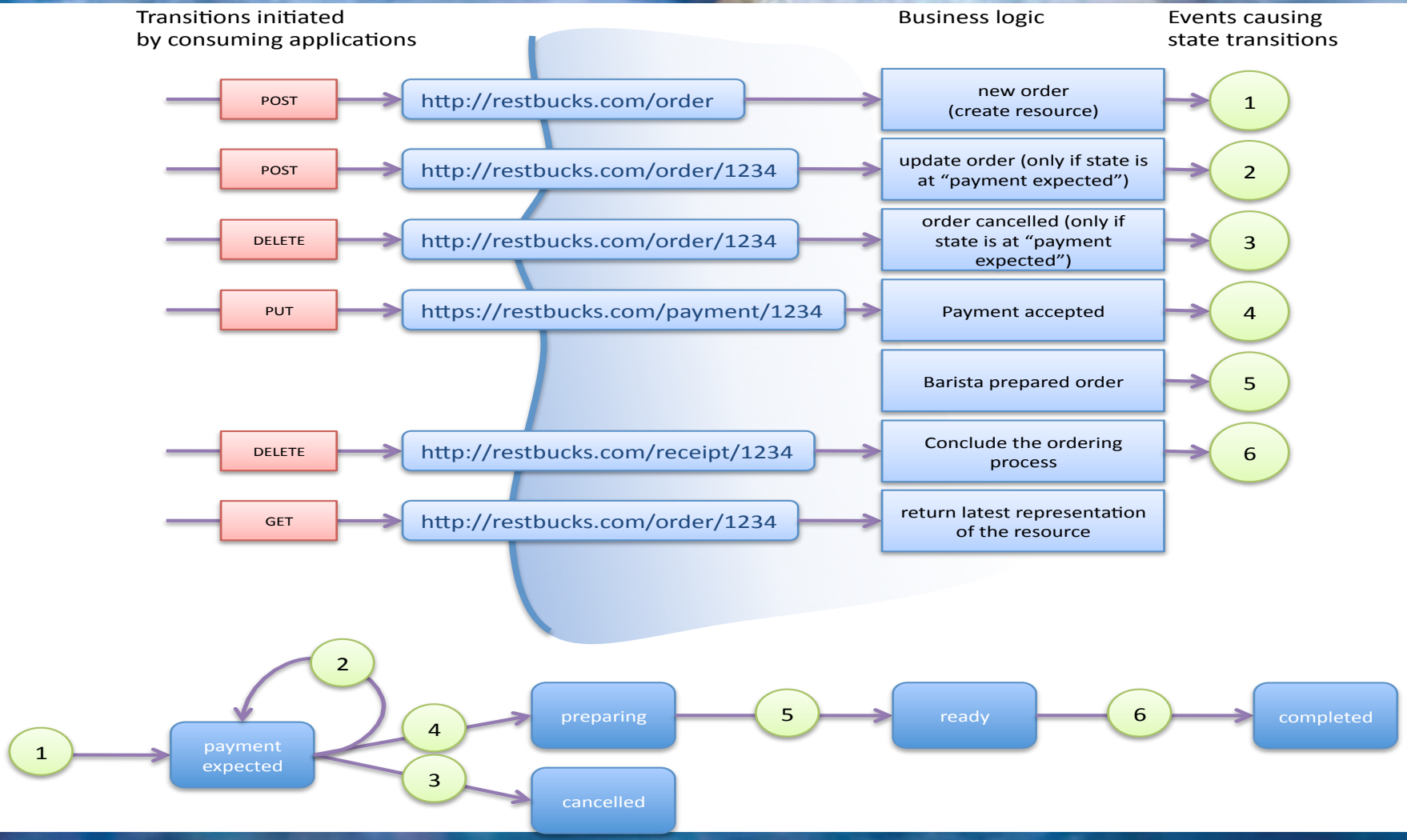
- Links declare next valid steps
- Following links and interacting with resources changes *application state*
- Media types and links define contracts
  - Media type defines processing model
  - Links (with microformats) describe state transitions
- Don't need a static contract description
  - No WSDL, no WADL
- This is HATEOAS.
- So let's see how we order a coffee at Restbucks.com…
  - Based on:
    http://www.infoq.com/articles/webber-rest-workflow

Domain Application protocols!

## Workflow

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

# Static Interface and State Transitions



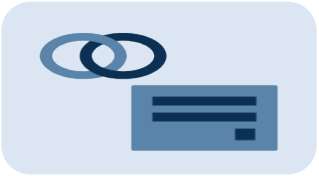
# `application/vnd.restbucks+xml`

- What a mouthful!
- The `vnd` namespace is for proprietary media types
  - As opposed to the IANA-registered ones
- Restbucks own XML is a hybrid
  - We use plain old XML to convey information
  - And `link` elements to convey protocol
- This is important, since it allows us to create RESTful, hypermedia aware services

# application/vnd.restbucks+xml

## Schemas

`http://schemas.restbucks.com`



## Hypermedia controls

`<link>` element, borrowed from Atom

## Link relations

| | |
|---|---|
| `order` | `update` |
| `cancel(lation)` | `payment` |
| `receipt` | **`self`** |

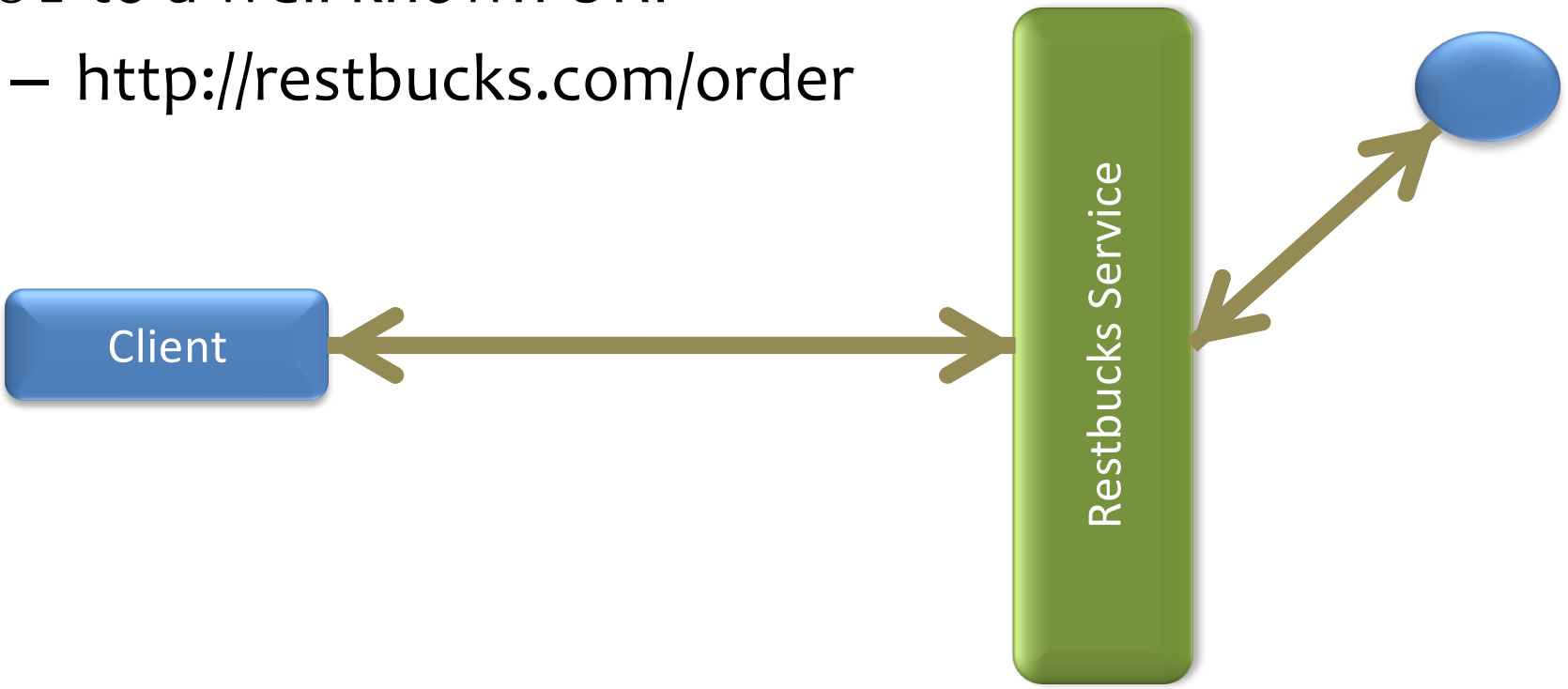
## Processing model

Atom-like, links indicate HTTP idioms to apply

## Place an Order

`POST` to a well-known URI

- http://restbucks.com/order



# Placing an Order

## Request

```
POST /order HTTP/1.1
Content-Type: application/vnd.restbucks+xml
Accept: application/vnd.restbucks+xml
Host: restbucks.com
Connection: keep-alive
Content-Length: 283

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:order xmlns:ns2="http://schemas.restbucks.com">
  <ns2:item>
    <ns2:milk>semi</ns2:milk>
    <ns2:size>large</ns2:size>
    <ns2:drink>latte</ns2:drink>
  </ns2:item>
  <ns2:location>takeaway</ns2:location>
</ns2:order>
```

# Placing an Order

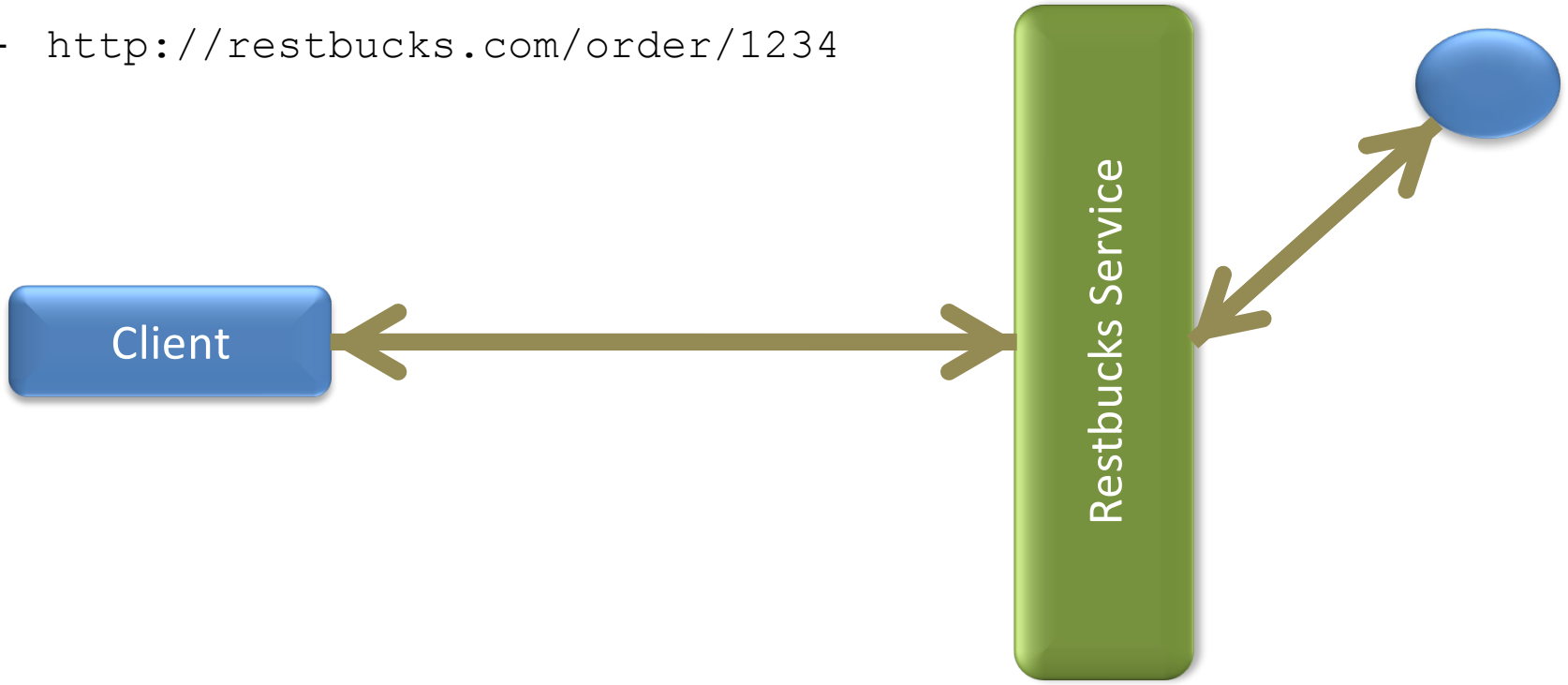
## Response

```
HTTP/1.1 201 Created
server: grizzly/1.8.1
Location: http://restbucks.com/order/1234
Content-Type: application/vnd.restbucks+xml
Content-Length: 1039
Date: Wed, 03 Mar 2010 20:58:03 GMT

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:order xmlns:ns2="http://schemas.restbucks.com" xmlns:ns3="http://schemas.restbucks.com/dap">
  <ns3:link mediaType="application/vnd.restbucks+xml" uri="http://restbucks.com/order/1234"
    rel="http://relations.restbucks.com/cancel"/>
  <ns3:link mediaType="application/vnd.restbucks+xml" uri="http://restbucks.com/payment/1234"
    rel="http://relations.restbucks.com/payment"/>
  <ns3:link mediaType="application/vnd.restbucks+xml" uri="http://restbucks.com/order/1234"
    rel="http://relations.restbucks.com/update"/>
  <ns3:link mediaType="application/vnd.restbucks+xml" uri="http://restbucks.com/order/1234" rel="self"/>
  <ns2:item>
    <ns2:milk>semi</ns2:milk>
    <ns2:size>large</ns2:size>
    <ns2:drink>latte</ns2:drink>
  </ns2:item>
  <ns2:location>takeaway</ns2:location>
  <ns2:cost>2.0</ns2:cost>
  <ns2:status>unpaid</ns2:status>
</ns2:order>
```

# Confirm the Order

`GET` from the rel="self" URI

- `http://restbucks.com/order/1234`



## Confirm the Order

### Request

```
GET /order/1234 HTTP/1.1
Accept: application/vnd.restbucks+xml
Host: restbucks.com
Connection: keep-alive
```

# Confirm the Order

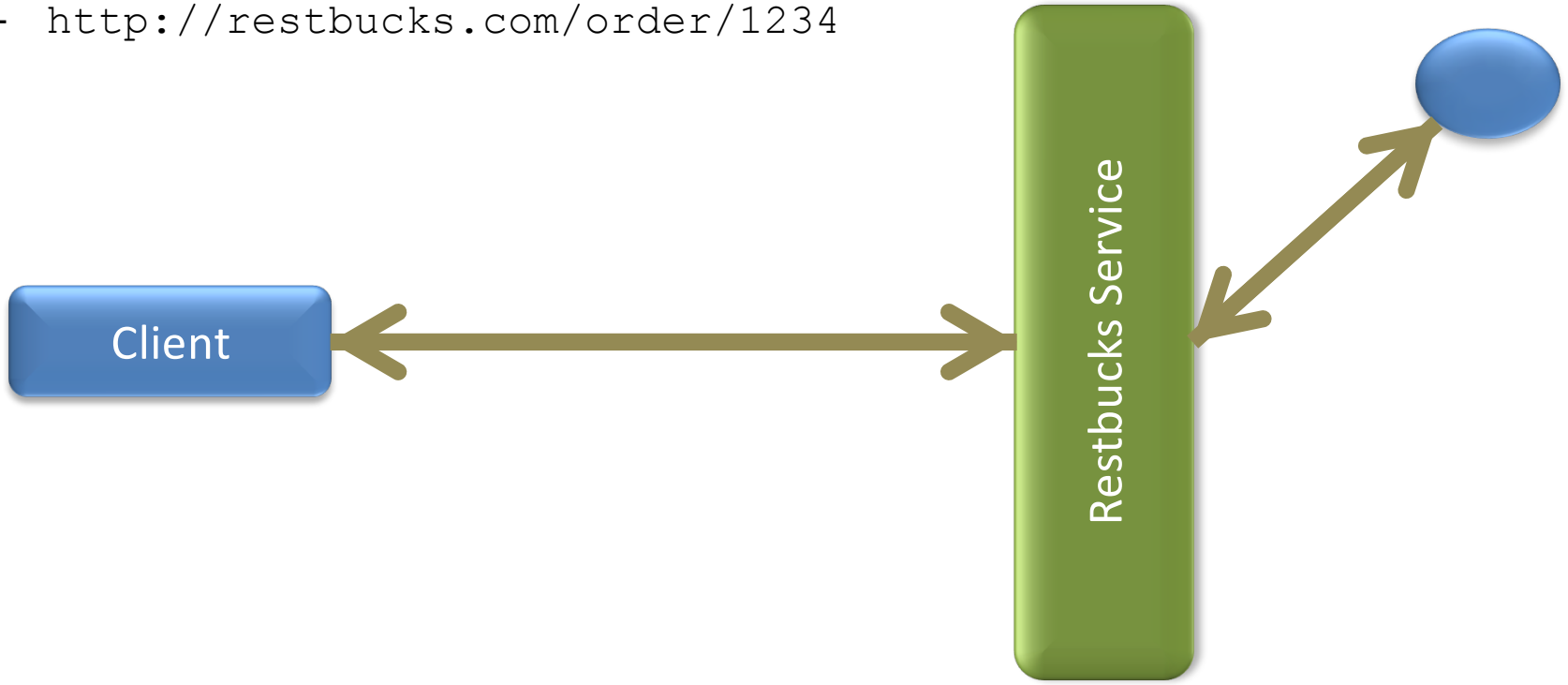
## Response

```
HTTP/1.1 200 OK
Location: http://restbucks.com/order/1234
Content-Type: application/vnd.restbucks+xml
Content-Length: 911
Date: Sun, 06 Sep 2009 06:51:22 GMT

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:order xmlns:ns2="http://schemas.restbucks.com" xmlns:ns3="http://schemas.restbucks.com/dap">
  <ns3:link mediaType="application/vnd.restbucks+xml" uri="http://restbucks.com/order/1234"
     rel="http://relations.restbucks.com/cancel"/>
  <ns3:link mediaType="application/vnd.restbucks+xml" uri="http://restbucks.com/payment/1234"
     rel="http://relations.restbucks.com/payment"/>
  <ns3:link mediaType="application/vnd.restbucks+xml" uri="http://restbucks.com/order/1234"
     rel="http://relations.restbucks.com/update"/>
  <ns3:link mediaType="application/vnd.restbucks+xml" uri="http://restbucks.com/order/1234" rel="self"/>
  <ns2:item>
    <ns2:milk>semi</ns2:milk>
    <ns2:size>large</ns2:size>
    <ns2:drink>latte</ns2:drink>
  </ns2:item>
  <ns2:location>takeaway</ns2:location>
  <ns2:cost>2.0</ns2:cost>
  <ns2:status>unpaid</ns2:status>
</ns2:order>
```

# Change the Order

`POST` new order to service-generated URI

- `http://restbucks.com/order/1234`



## POST? Not PUT?

- `PUT` expects the whole resource state to be presented in the request
  - But our clients aren't responsible for generating cost or status elements
- `PATCH` would be better
  - It allows us to send diffs, but is new to the family of HTTP verbs and not widely supported yet
- So `POST` is our only option
  - Compare this to our CRUD protocol earlier

**Can we better decompose the problem space to avoid this?**

# Change the Order

## Request

```
POST order/1234 HTTP/1.1
Content-Type: application/vnd.restbucks+xml
Accept: application/vnd.restbucks+xml
Host: restbucks.com
Connection: keep-alive
Content-Length: 288

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:order xmlns:ns2="http://schemas.restbucks.com">
  <ns2:item>
    <ns2:milk>semi</ns2:milk>
    <ns2:size>large</ns2:size>
    <ns2:drink>cappuccino</ns2:drink>
  </ns2:item>
  <ns2:location>takeaway</ns2:location>
</ns2:order>
```

No service-generated content

# Change the Order

## Response

```
HTTP/1.1 200 OK
Location: http://restbucks.com/order/1234
Content-Type: application/vnd.restbucks+xml
Content-Length: 916
Date: Sun, 06 Sep 2009 06:52:22 GMT

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:order xmlns:ns2="http://schemas.restbucks.com" xmlns:ns3="http://schemas.restbucks.com/dap">
  <ns3:link mediaType="application/vnd.restbucks+xml" uri="http://restbucks.com/order/1234"
     rel="http://relations.restbucks.com/cancel"/>
  <ns3:link mediaType="application/vnd.restbucks+xml" uri="http://restbucks.com/payment/1234"
     rel="http://relations.restbucks.com/payment"/>
  <ns3:link mediaType="application/vnd.restbucks+xml" uri="http://restbucks.com/order/1234"
     rel="http://relations.restbucks.com/update"/>
  <ns3:link mediaType="application/vnd.restbucks+xml" uri="http://restbucks.com/order/1234" rel="self"/>
  <ns2:item>
    <ns2:milk>semi</ns2:milk>
    <ns2:size>large</ns2:size>
    <ns2:drink>cappuccino</ns2:drink>
  </ns2:item>
  <ns2:location>takeaway</ns2:location>
  <ns2:cost>2.0</ns2:cost>
  <ns2:status>unpaid</ns2:status>
</ns2: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: restbucks.com
...
```

- Request

```
OPTIONS /order/1234 HTTP 1.1
Host: restbucks.com
```

- Response

```
409 Conflict
```

- Response

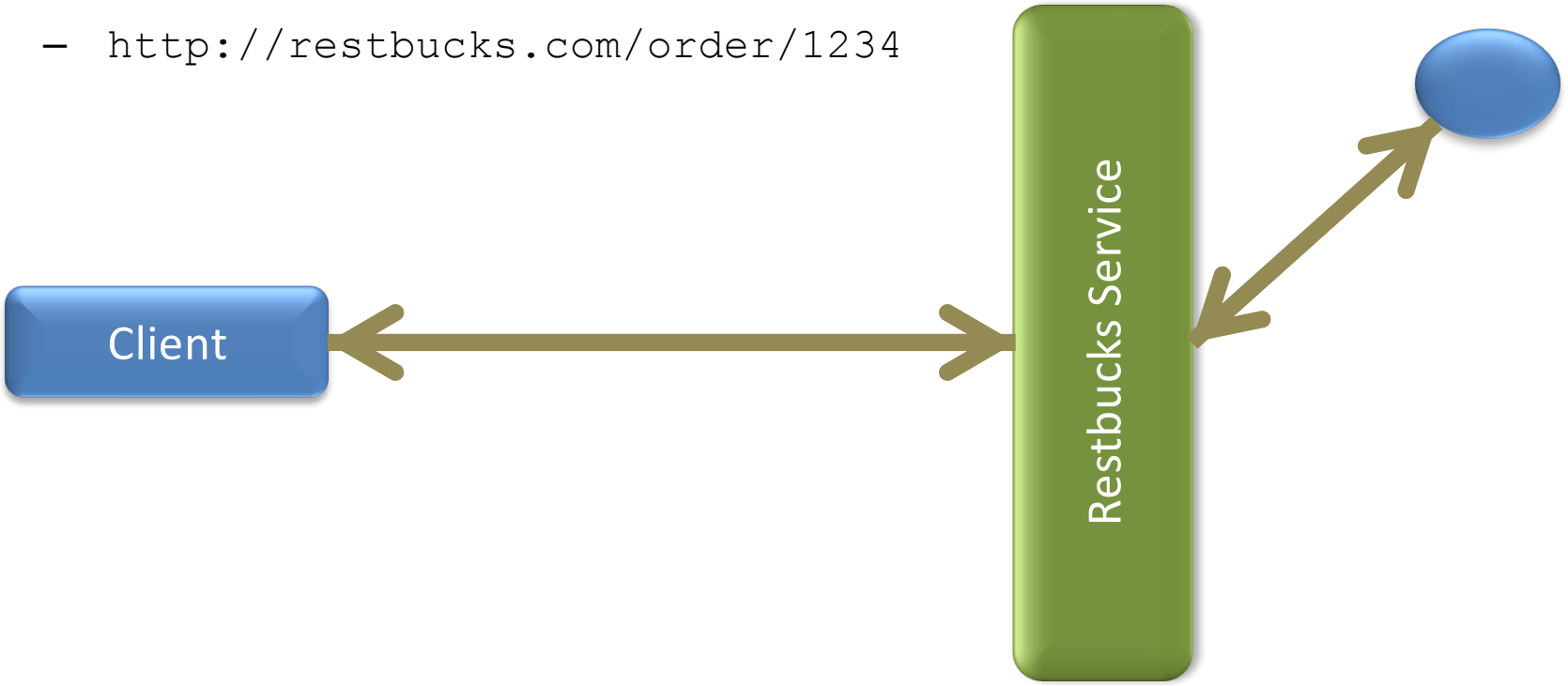
```
Allow: GET
```

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

# What if we want to cancel?

`DELETE` order at rel="self" URI

- `http://restbucks.com/order/1234`



# What if we want to cancel?

## Request

```
DELETE /order/1234 HTTP/1.1
Host: restbucks.com
Connection: keep-alive
```

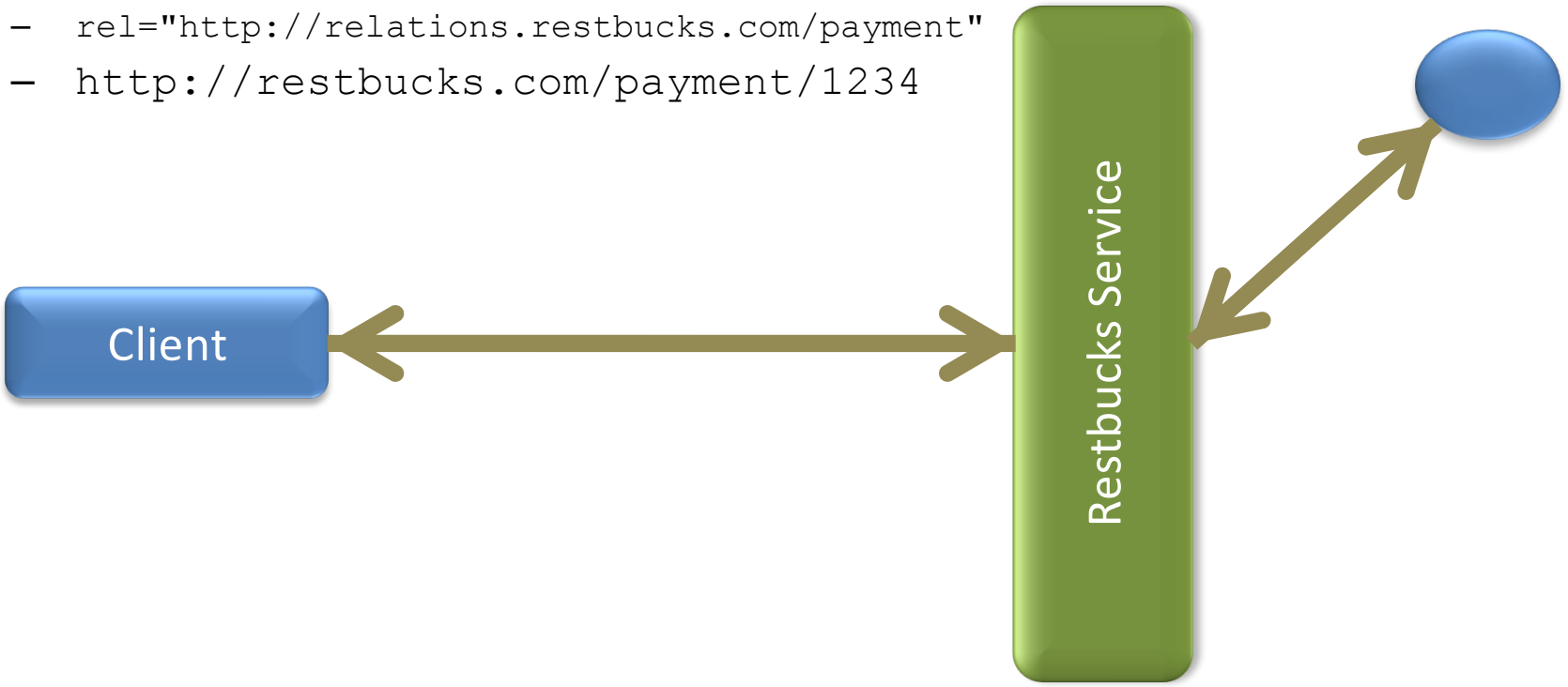
## Response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.restbucks+xml
Date: Sun, 06 Sep 2009 06:53:22 GMT
```

# Order Payment

PUT to service-generated payment URI

- `rel="http://relations.restbucks.com/payment"`
- `http://restbucks.com/payment/1234`



## Why `PUT` this time?

- It's idempotent
  - Safe to repeat in the event of failures
- For all $ transactions, prefer idempotent verbs

# Order Payment

## Request

```
PUT /payment/1234 HTTP/1.1
Content-Type: application/vnd.restbucks+xml
Accept: application/vnd.restbucks+xml
Host: restbucks.com
Connection: keep-alive
Content-Length: 287

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<payment xmlns="http://schemas.restbucks.com">
  <amount>2.0</amount>
  <cardholderName>Michael Farraday</cardholderName>
  <cardNumber>11223344</cardNumber>
  <expiryMonth>12</expiryMonth>
  <expiryYear>12</expiryYear>
</payment>
```

# Order Payment

## Response

```
HTTP/1.1 201 Created
server: grizzly/1.8.1
Location: http://restbucks.com/payment/1234
Content-Type: application/vnd.restbucks+xml
Content-Length: 650
Date: Wed, 03 Mar 2010 20:58:03 GMT

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:payment xmlns="http://schemas.restbucks.com/dap" xmlns:ns2="http://schemas.restbucks.com">
  <link mediaType="application/vnd.restbucks+xml" uri="http://restbucks.com/order/1234"
    rel="http://relations.restbucks.com/order"/>
  <link mediaType="application/vnd.restbucks+xml" uri="http://restbucks.com/receipt/1234"
    rel="http://relations.restbucks.com/receipt"/>
  <ns2:amount>2.0</ns2:amount>
  <ns2:cardholderName>Michael Farraday</ns2:cardholderName>
  <ns2:cardNumber>11223344</ns2:cardNumber>
  <ns2:expiryMonth>12</ns2:expiryMonth>
  <ns2:expiryYear>12</ns2:expiryYear>
</ns2:payment>
```

# What if we want to cancel now?

## Request

```
DELETE /order/1234 HTTP/1.1
Host: restbucks.com
Connection: keep-alive
```

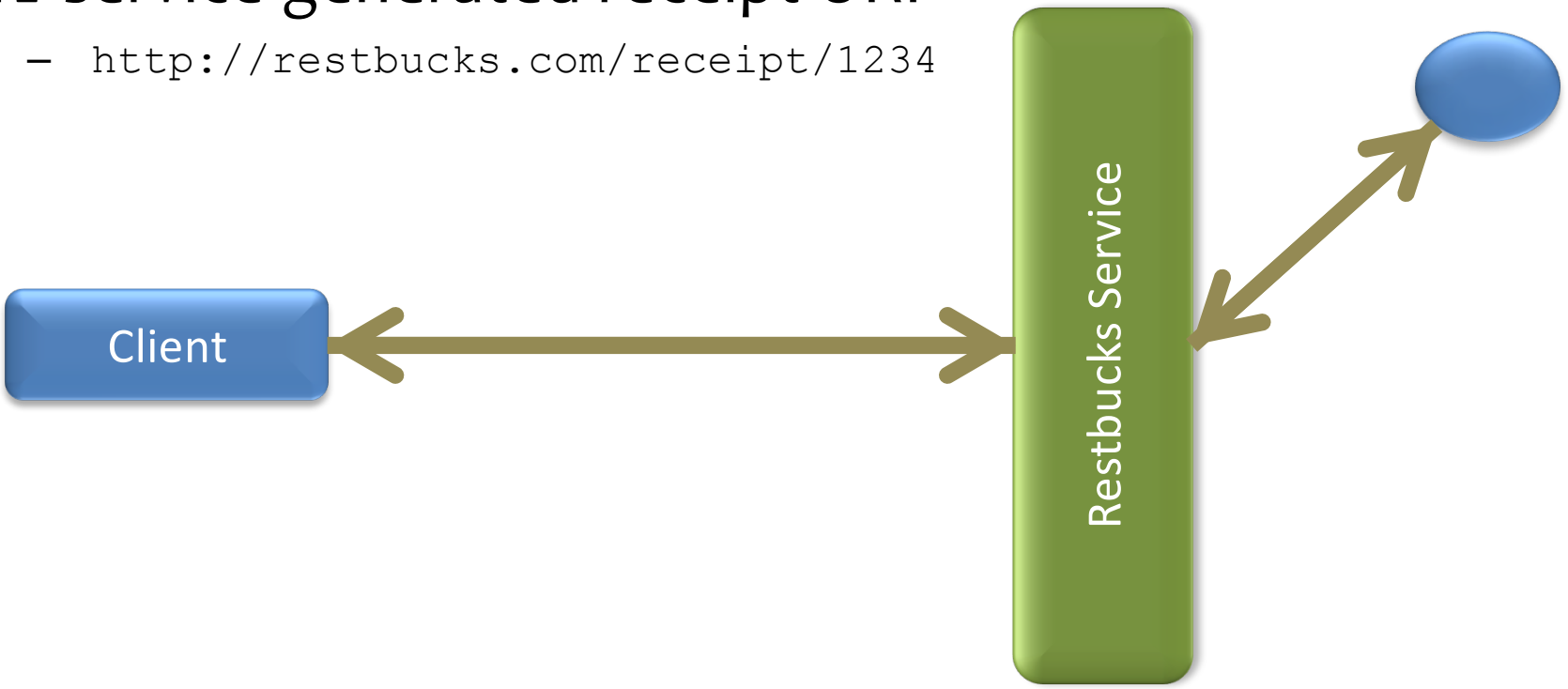
## Response

```
HTTP/1.1 405 Method Not Allowed
Allow: GET
Content-Type: application/vnd.restbucks+xml
Content-Length: 0
Date: Sun, 06 Sep 2009 07:43:29 GMT
```

# Grab a Receipt

`GET` service-generated receipt URI

- `http://restbucks.com/receipt/1234`



# Grab a Receipt

## Request

```
GET /receipt/1234 HTTP/1.1
Accept: application/vnd.restbucks+xml
Host: restbucks.com
```

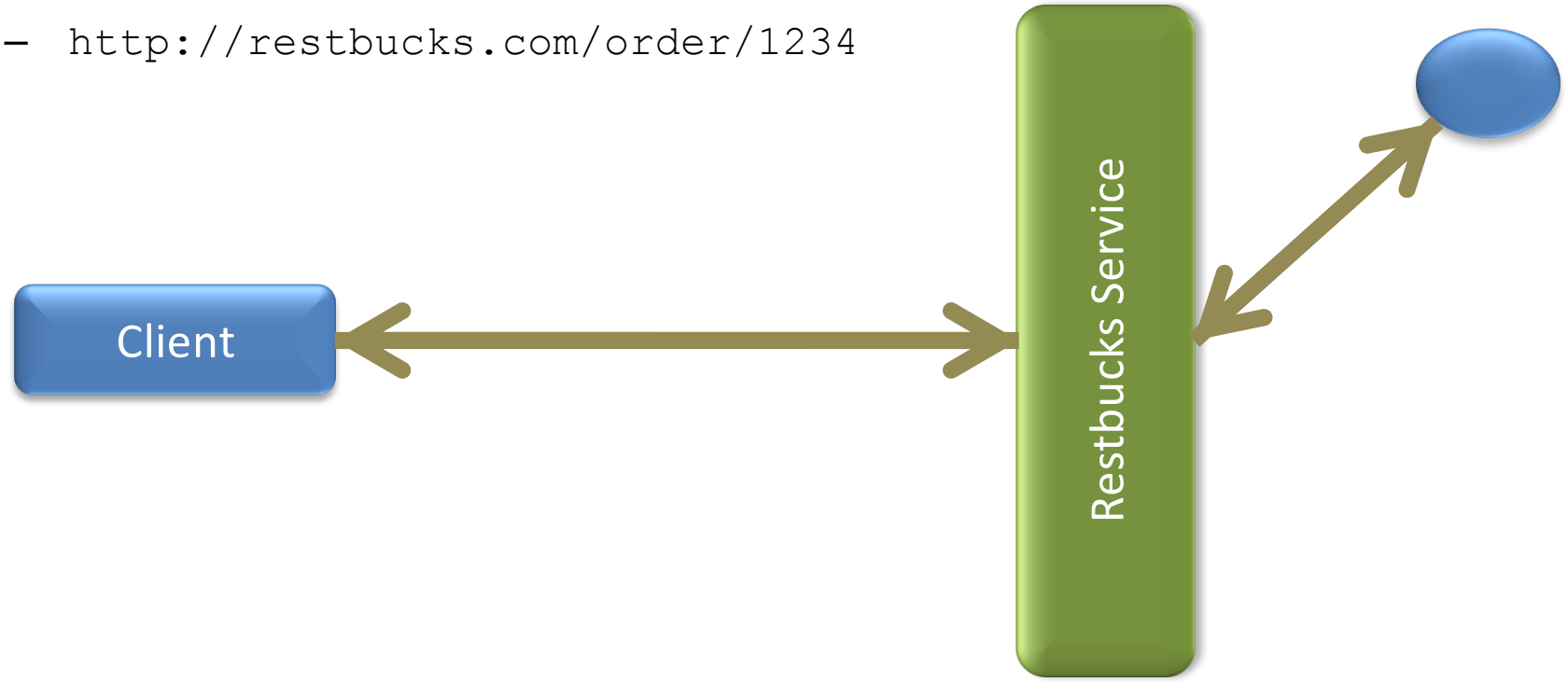
## Grab a Receipt

- Request

```
HTTP/1.1 200 OK
server: grizzly/1.8.1
Content-Type: application/vnd.restbucks+xml
Content-Length: 384
Date: Wed, 03 Mar 2010 20:58:03 GMT

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:receipt xmlns="http://schemas.restbucks.com/dap"
   xmlns:ns2="http://schemas.restbucks.com">
   <link mediaType="application/vnd.restbucks+xml"
    uri="http://restbucks.com/order/1234"
    rel="http://relations.restbucks.com/order"/>
   <ns2:amount>2.0</ns2:amount>
   <ns2:paid>2010-03-03T21:58:03.834+01:00</ns2:paid>
</ns2:receipt>
```

# Poll until the order's ready

- GET from `rel="http://relations.restbucks.com/order"` URI
  - `http://restbucks.com/order/1234`



## Polling for a ready order on the wire

Request

```
GET /order/1234 HTTP/1.1
Host: restbucks.com
Connection: keep-alive
```

# Order's ready, take the receipt and walk away

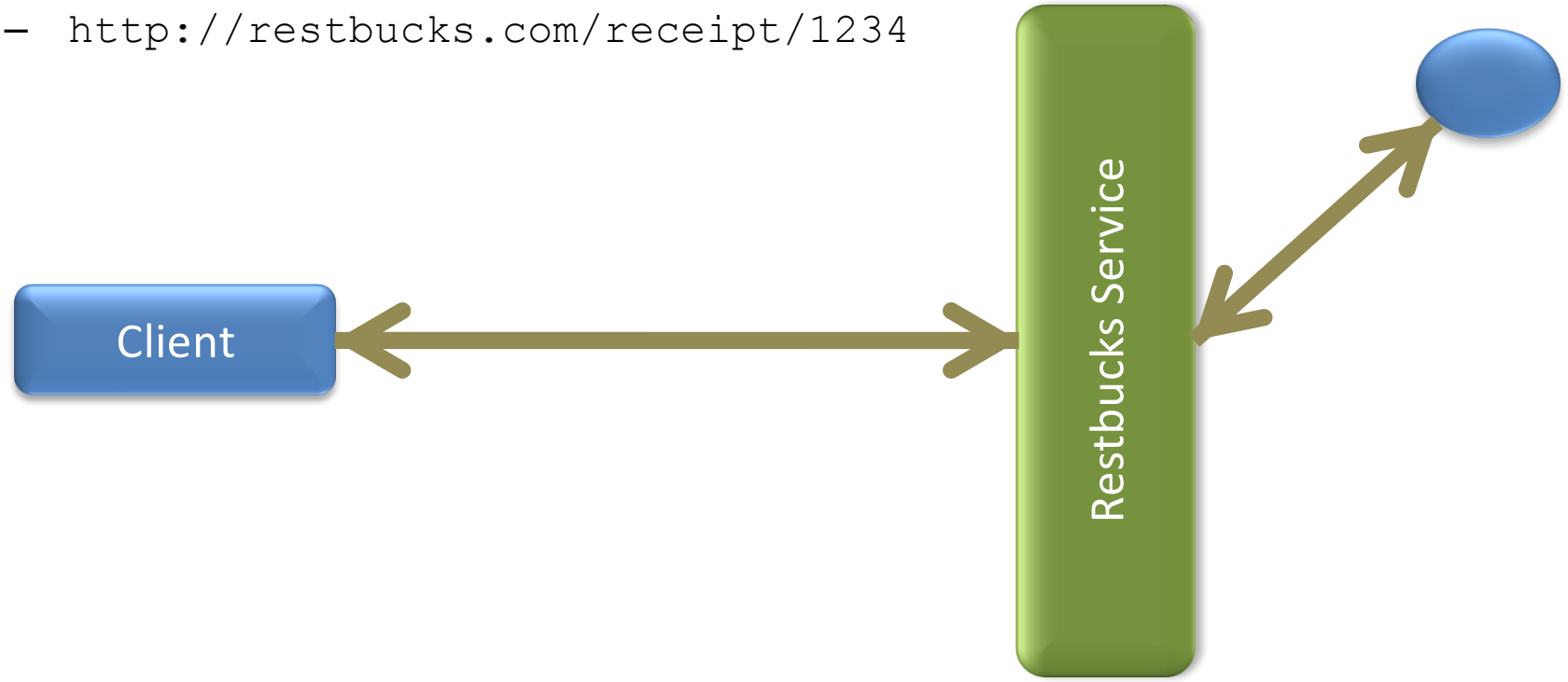
## Response

```
HTTP/1.1 200 OK
server: grizzly/1.8.1
Content-Type: application/vnd.restbucks+xml
Content-Length: 534
Date: Wed, 03 Mar 2010 20:58:03 GMT

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:order xmlns:ns2="http://schemas.restbucks.com"
    xmlns:ns3="http://schemas.restbucks.com/dap">
  <ns3:link mediaType="application/vnd.restbucks+xml"
    uri="http://restbucks.com/receipt/1234"
    rel="http://relations.restbucks.com/reciept"/>
  <ns2:item>
    <ns2:milk>semi</ns2:milk>
    <ns2:size>large</ns2:size>
    <ns2:drink>cappuccino</ns2:drink>
  </ns2:item>
  <ns2:location>takeaway</ns2:location>
  <ns2:cost>2.0</ns2:cost>
  <ns2:status>ready</ns2:status>
</ns2:order>
```

# Complete the Order

DELETE order at `rel="http://relations.restbucks.com/receipt"` URI

- `http://restbucks.com/receipt/1234`



## Complete the Order

### Request

```
DELETE/receipt/1234 HTTP/1.1
Host: restbucks.com
Connection: keep-alive
```

# Complete the Order

## Response

```
HTTP/1.1 200 OK
server: grizzly/1.8.1
Content-Type: application/xml
Content-Length: 393
Date: Wed, 03 Mar 2010 20:58:03 GMT

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:order xmlns:ns2="http://schemas.restbucks.com"
    xmlns:ns3="http://schemas.restbucks.com/dap">
  <ns2:item>
    <ns2:milk>semi</ns2:milk>
    <ns2:size>large</ns2:size>
    <ns2:drink>cappuccino</ns2:drink>
  </ns2:item>
  <ns2:location>takeaway</ns2:location>
  <ns2:cost>2.0</ns2:cost>
  <ns2:status>taken</ns2:status>
</ns2:order>
```

No Hypermedia Controls

# Finally drink your coffee…


Source: http://images.businessweek.com/ss/06/07/top_brands/image/starbucks.jpg

## Ordering and Payments Protocol Weaknesses

- Lacks HTTP idioms
  - ETags
  - Cache metadata
  - Etc
- No circuit breaker
  - E.g. 413 at busy times

## Exercise

- How on Earth did you pay for a coffee in the CRUD exercise?
- Some of you picked out URIs from the response entity
- But that's a job that can be done by a computer, so…
- Let's properly order a coffee starting at `/order`

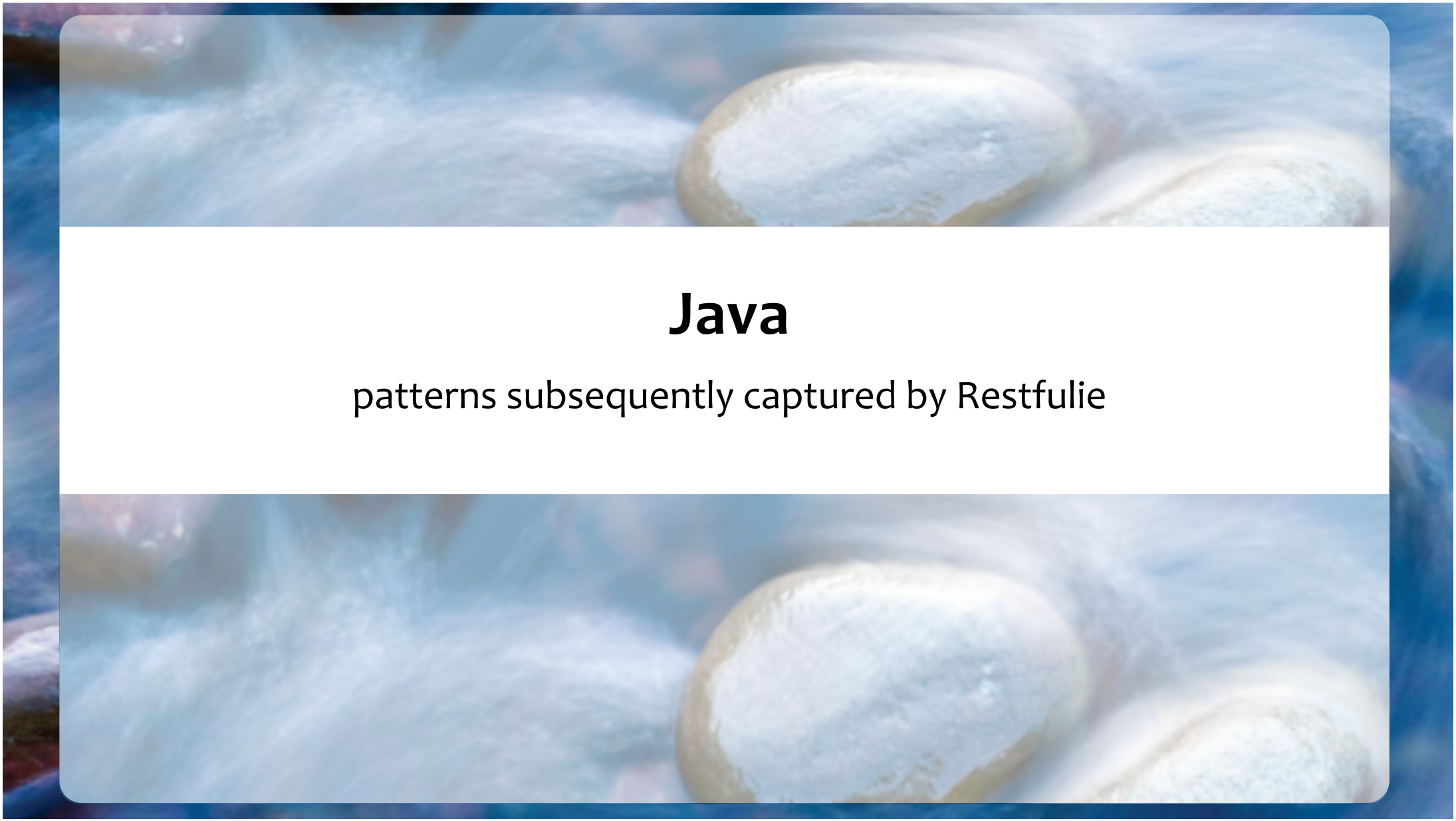

THOUGHT WE'D FINISHED XML EXERCISES
GOT EVEN MOREXML

## Domain Application Protocol

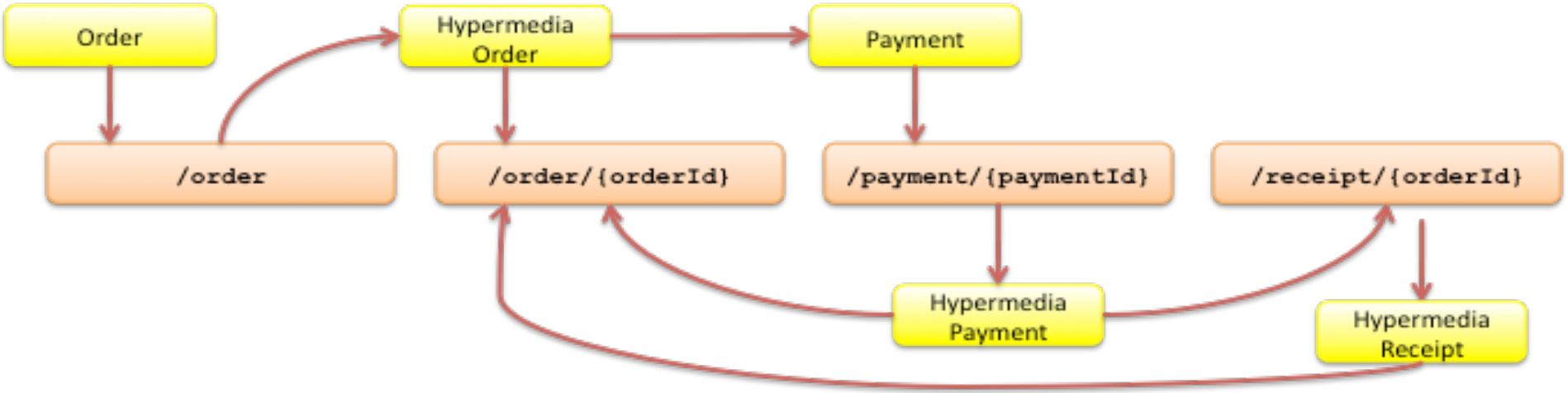
- The XML contains links which describe* a protocol
- If you bind to those link names, then the URIs don't matter
- Treat them as "things you'd like to do next"
- Let's code/curl

# Tech Interlude

## Implementing the Restbucks Ordering Service


Java
patterns subsequently captured by Restfulie

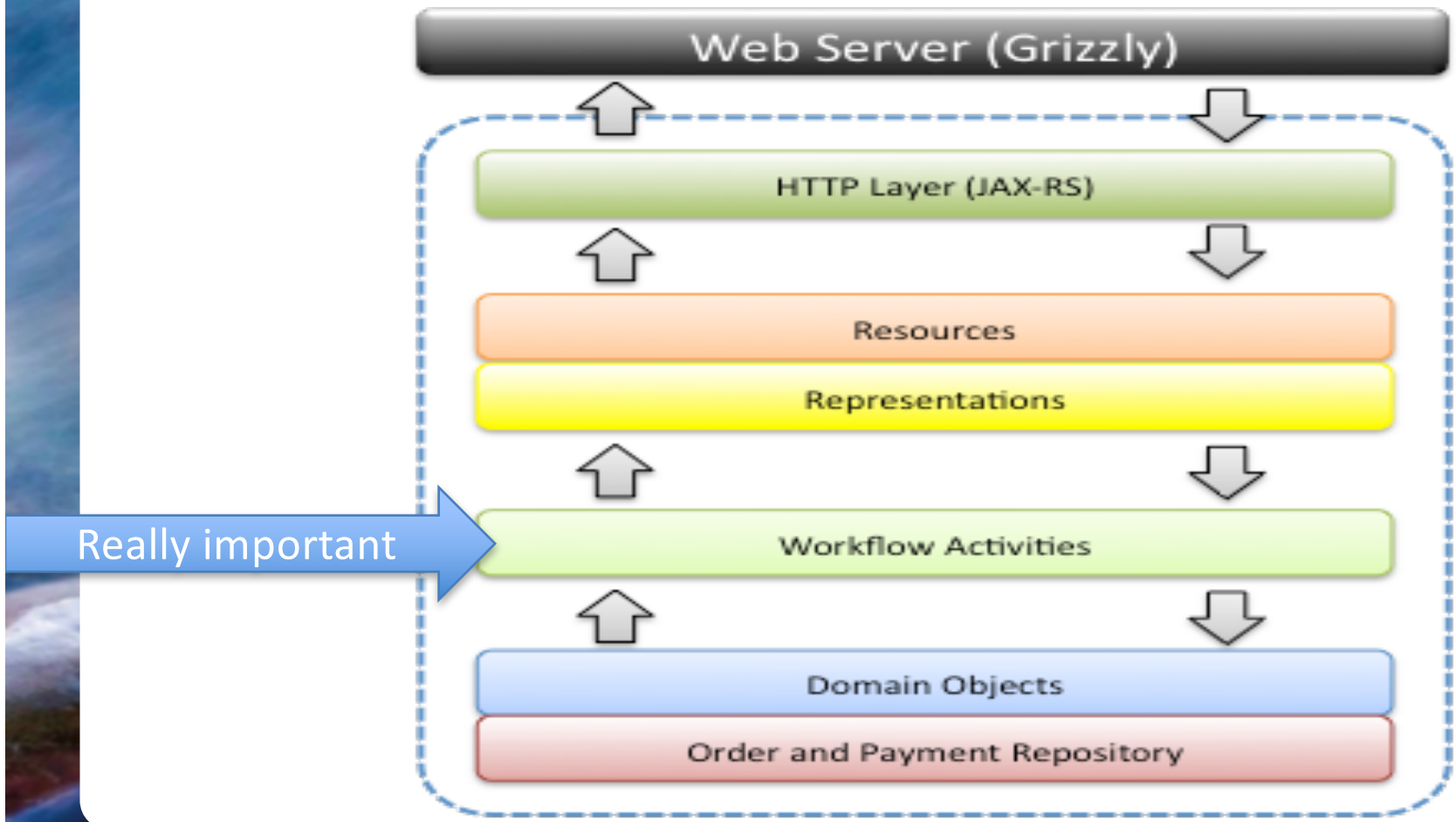
# Protocol/Resource Mapping



## Server's easy

- We picked Jersey (JAX-RS)
- Somewhat able to TDD it, with effort
  - Some of the server-side return types don't invite easy inspection sadly

# Java Server-side Architecture



# Typical JAX-RS Plumbing

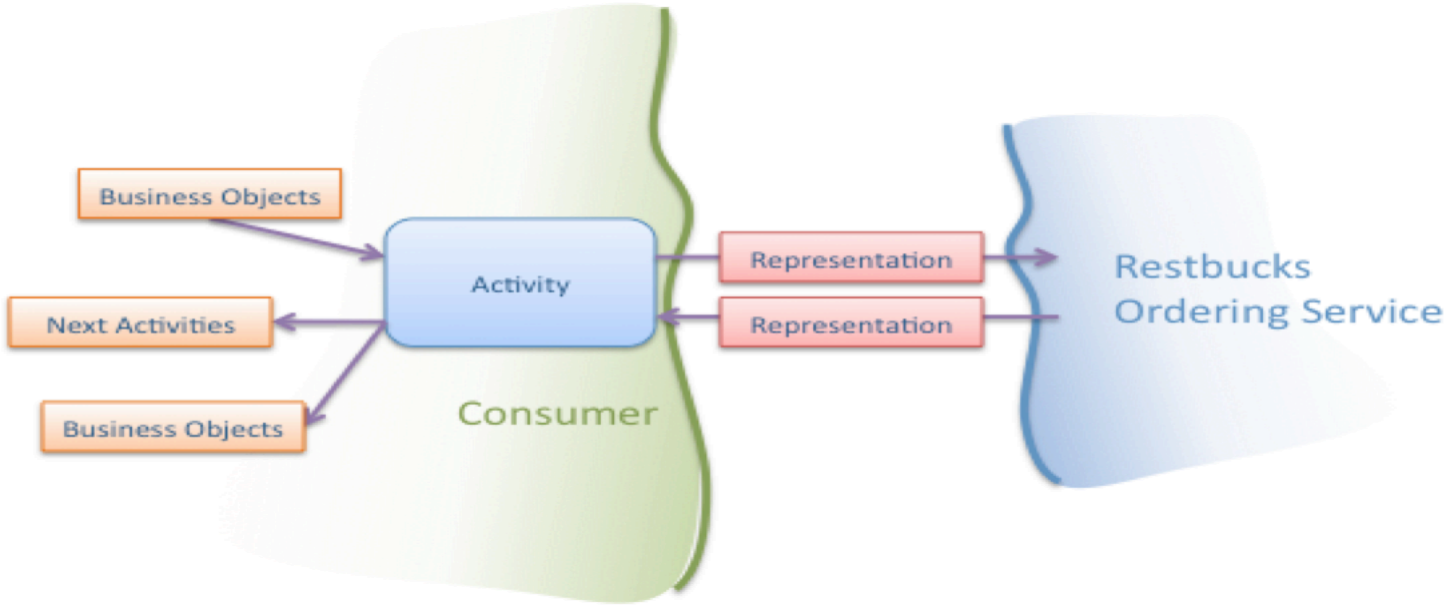
```
@POST
 @Path("/{orderId}")
 @Consumes("application/vnd.restbucks+xml")
 @Produces("application/vnd.restbucks+xml")
 public Response updateOrder(String orderRepresentation) {
   try {
     OrderRepresentation responseRepresentation = new UpdateOrderActivity()
                        .update(
                         OrderRepresentation.fromXmlString(
                          orderRepresentation)
                         .getOrder(), new RestbucksUri(uriInfo.getRequestUri()));
     return Response.ok().entity(responseRepresentation).build();
   } catch (InvalidOrderException ioe) {
     return Response.status(Status.BAD_REQUEST).build();
   } catch (NoSuchOrderException nsoe) {
     return Response.status(Status.NOT_FOUND).build();
   } catch(UpdateException ue) {
     return Response.status(Status.CONFLICT).build();
   } catch (Exception ex) {
     return Response.serverError().build();
   }
  }
```

# Generating a Response Representation

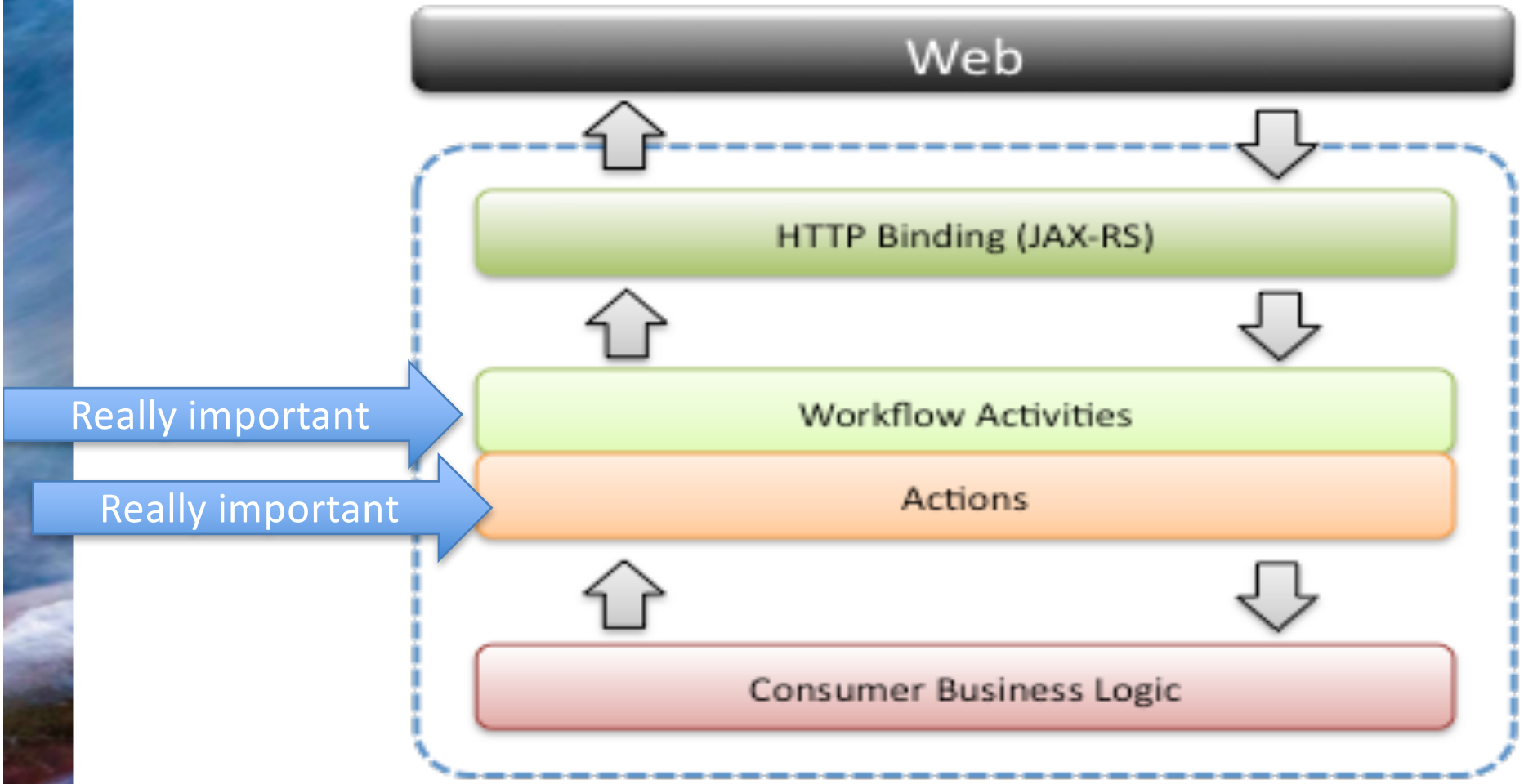
```
public class CreateOrderActivity {
  public OrderRepresentation create(Order order,
                                    RestbucksUri requestUri) {
    order.setStatus(OrderStatus.UNPAID);

    Identifier identifier = OrderRepository.current().store(order);

    RestbucksUri orderUri = new RestbucksUri(requestUri.getBaseUri() +
                                "/order/" + identifier.toString());

    RestbucksUri paymentUri = new RestbucksUri(requestUri.getBaseUri() +
                                 "/payment/" + identifier.toString());
    return new OrderRepresentation(order,
        new Link(Representation.RELATIONS_URI + "cancel", orderUri),
        new Link(Representation.RELATIONS_URI + "payment", paymentUri),
        new Link(Representation.RELATIONS_URI + "update", orderUri),
        new Link(Representation.SELF_REL_VALUE, orderUri));
  }
}
```

# Consumers are harder

- We chose Jersey client
- Code to the media type, be declarative
- Separation of business data and next activities



# Java Consumer-side Architecture



# Consumer-side workflow

```
public void orderAndPay(Order order, URI entryPointUri) {

  PlaceOrderActivity placeOrderActivity = new PlaceOrderActivity();
  placeOrderActivity.placeOrder(order, entryPointUri);

  // Order processing ommitted for brevity…

  Actions actions = placeOrderActivity.getActions();

  if(actions.has(PaymentActivity.class)) {
    PaymentActivity paymentActivity = actions.get(PaymentActivity.class);
    paymentActivity.payForOrder(
      payment().withAmount(readOrderActivity.getOrder().getCost())
                               .build());
    actions = paymentActivity.getActions();
  }
  // Remainder of workflow ommitted for brevity
}
```

## Summary

- Web basics
- HTTP fundamentals
- Web-as-platform
- URI templates are not a good idea
- Crud is good
- Hypermedia is better!
- Getting a coffee