The Web as a Platform

The Web

- The Web has transformed how we produce and share information
- International ecosystem of applications and services that allows us to search, aggregate, combine, transform, replicate, cache, and archive the information that underpins society.
- Largest, least formal integration project ever attempted

How can we apply the Web's underlying architectural principals to develop "enterprise applications?

The REST question…

What is the Web?

- The Web is the result of millions of simple, small-scale interactions between agents and resources that use the founding technologies of HTTP and URI. [Architecture of the World Wide Web, Volume One. - w3.org]
- The Web is a set of widely commoditised servers, from web servers to proxies, caches, and content delivery networks
- The Web allows us to shift to Hypermedia with the assumption that links can break
    - We are no longer in control of the whole system all the time!!! - Dangling references?!?

Thinking in Resources

- Resources are the fundamental building blocks of Web-based systems
- Anything we can expose - *documents, video clips, devices, etc.*
- We make them present by abstracting out their useful information
- URI - *Uniform Resource Identifier*

Identifiers and Representations

- A Resource can have multiple URIs
    - There is no mechanism for evaluating wether two URIs actually refer to the same resource
- Each identifier is associated with one or more representations - *a representation is a transformation or a view of a resources state in a given instant of time*

Representations

- Access to a resource is mediated by a representation

- This separation promotes loose-coupling between backends and consumers

- It also helps with scalability, since representations can be cached and replicated

- Gives different views - *useful for different kinds of interactions*

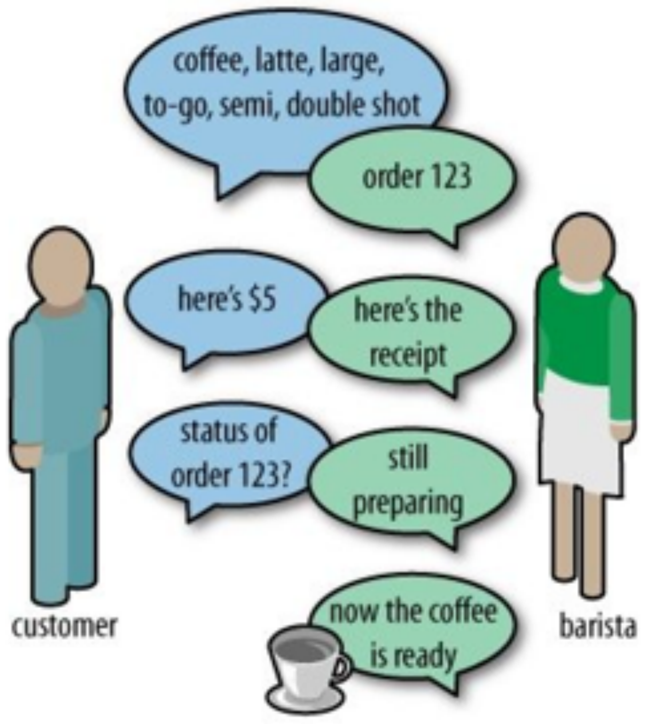
Restbucks

From a small shop to global domination

- Actors are customers, cashiers. baristas, managers, and suppliers
- Let us assume each actor is a program that interacts through the Web
- Goals: serve good coffee, take payments, keep the supply chain moving, and keep the business alive

Sample Interactions

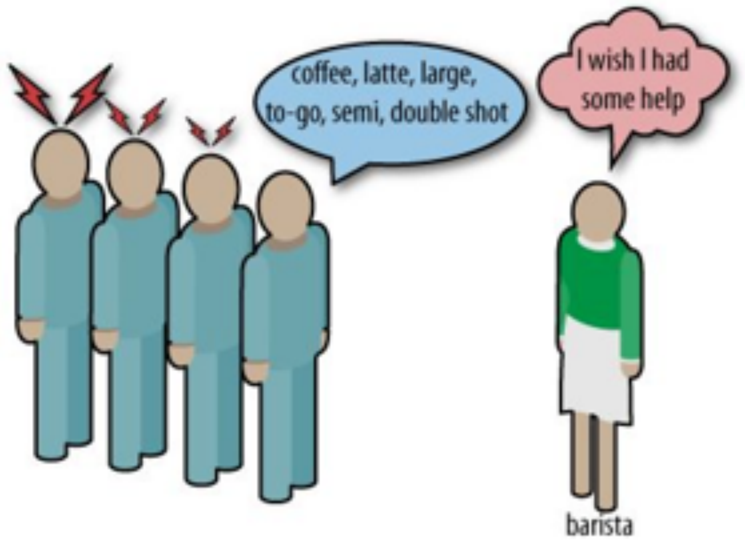
- Customer - Barista : in the beginning (small shop) the barista does everything
    - orders, preparation, payments, and receipts



Issues

- Notifications - *signal that a coffee is ready*
- Communication failure - *including timeouts*
- Transactions - *optimistically accept orders?*
- Scalability - *cope with large volumes of customers/requests*

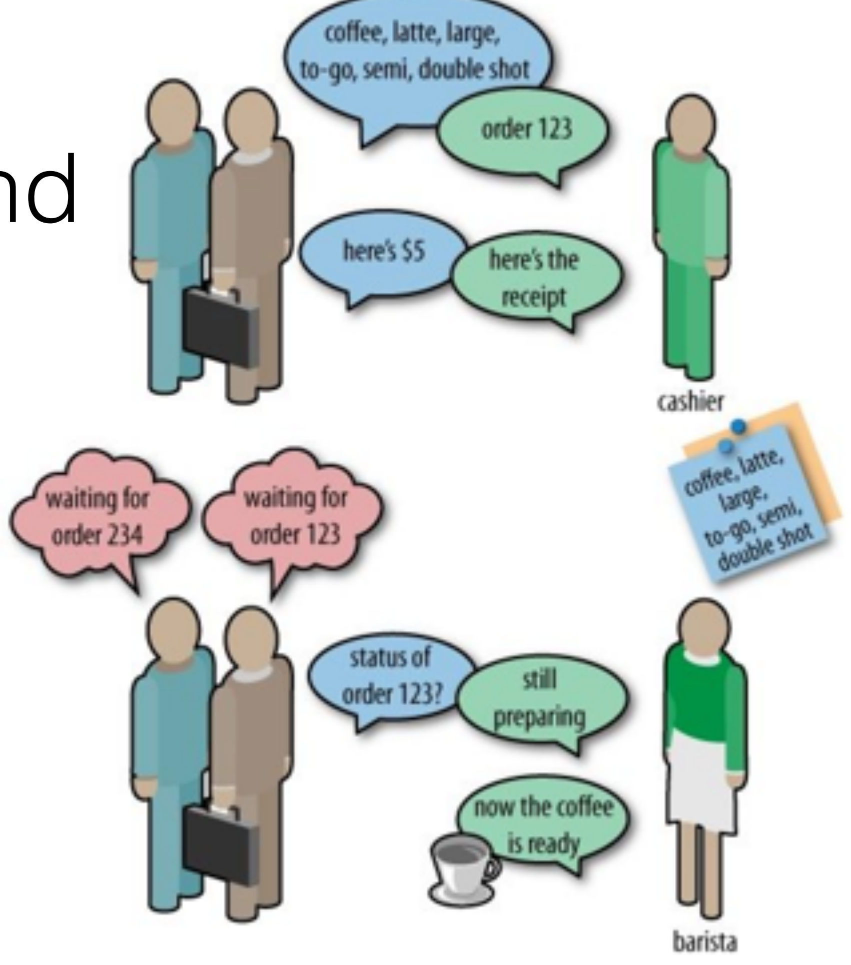
- The single barista does not scale well
- We have long lines



Customer - Cashier - Barista

- To help scale we introduce a cashier, while the barista concentrates on preparing the coffee
- Customer interactions remain the same
- We need to coordinate the cashier and the customer - *sticky notes*

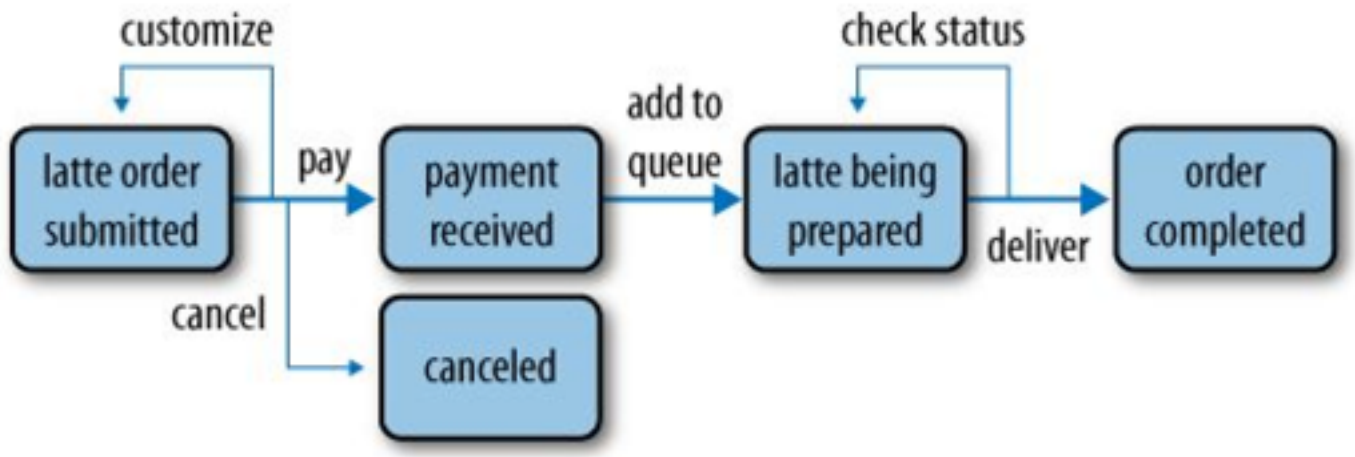
Decouple ordering from preparation



Advantages in Decoupling

- Optimise available resources - the barista can look at the queue and make decisions about preparation order
- Scale operations by adding more cashiers and baristas independently as demand increases

State machine representations



- The protocol state machine evolves through the interactions between customer, cashier, and barista

- A state machine is discovered as the interactions occur

- May change over time

Basic Web Integration

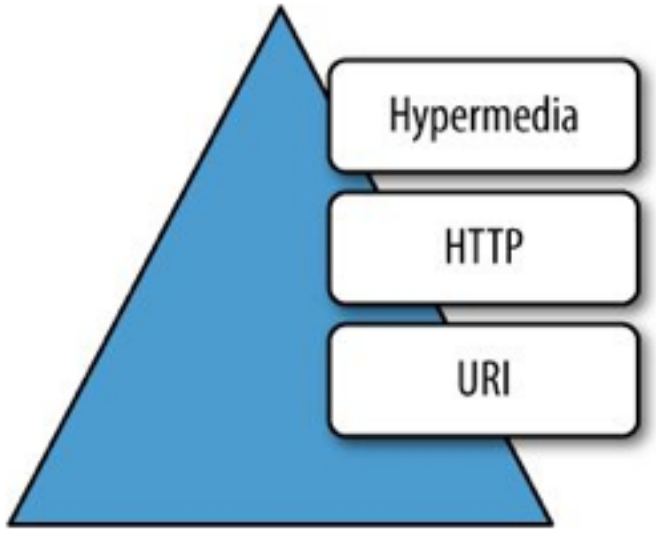
Enterprise Integration

- Many enterprise integration projects start with the identification of a middleware (reliable messaging, security, transaction, etc.)
- But do we always need all this? Sometimes it is overkill
- What about a lightweight solution? HTTP?
  - Can it reduce cost, risk, and time?

Let's look at some options!!!

Web maturity model

- Leonard Richardson provides a maturity model
  - a degree of Web friendliness
- Level 0 - a single URI and a single HTTP verb
  - SOAP is level 0
- Level 1 - many URIs but still a single HTTP verb
  - there is a notion of resources
- Level 2 - many URIs and many HTTP verbs
  - CRUD
- Level 3 - adds Hypermedia as the engine of application state

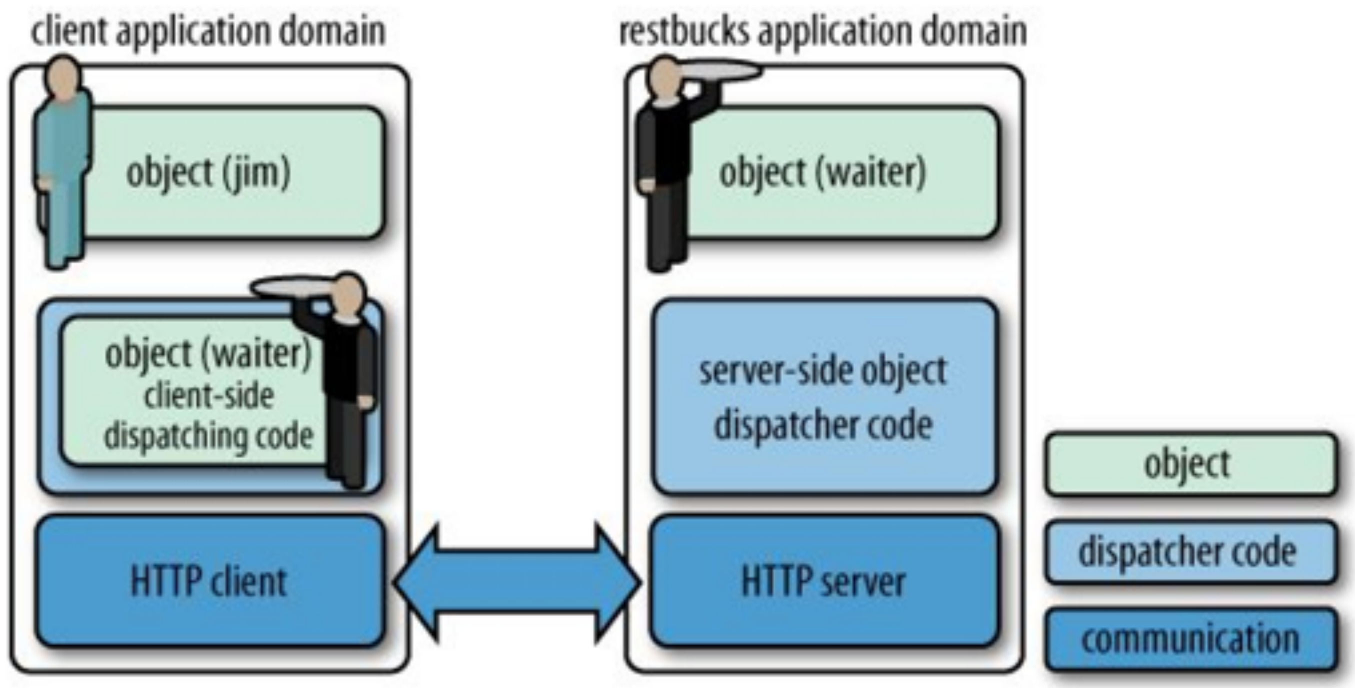


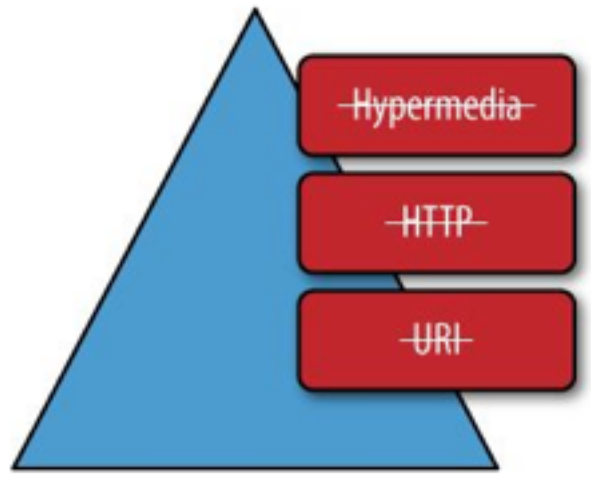
Many approaches say they are RESTful but they are only level one…

Local Facade integration

Local Facades

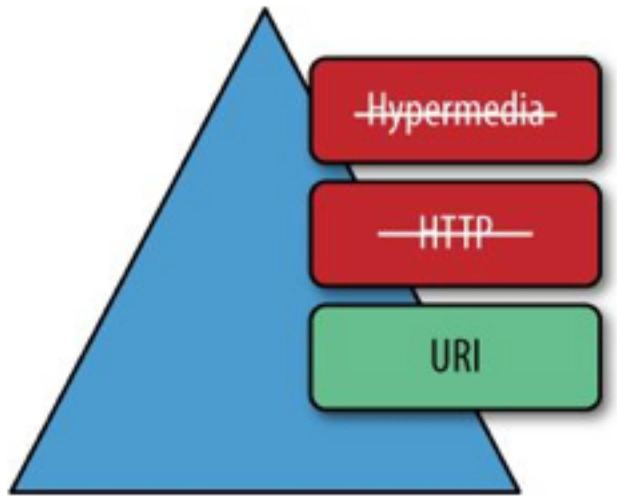
- Integration points encapsulate business processes and workflows
- They use facade to avoid exposing technical or implementation details


client application domain
restbucks application domain
object (jim)
object (waiter)
object (waiter) client-side dispatching code
server-side object dispatcher code
HTTP client
HTTP server
object
dispatcher code
communication



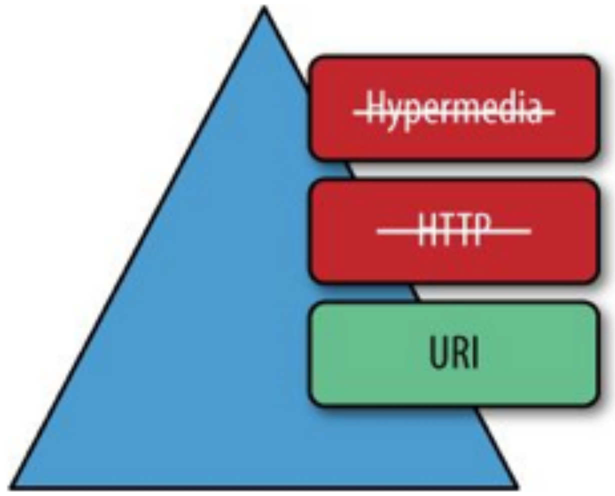
URI Templates

- Intuitive URIs
  - http://restbucks.com/order/1234
  - If I change the number will I get a different order?
- URI Templates
  - http://restbucks.com/order/{order_id}
  - http://restbucks.com/order/{year}/{month}/{day}



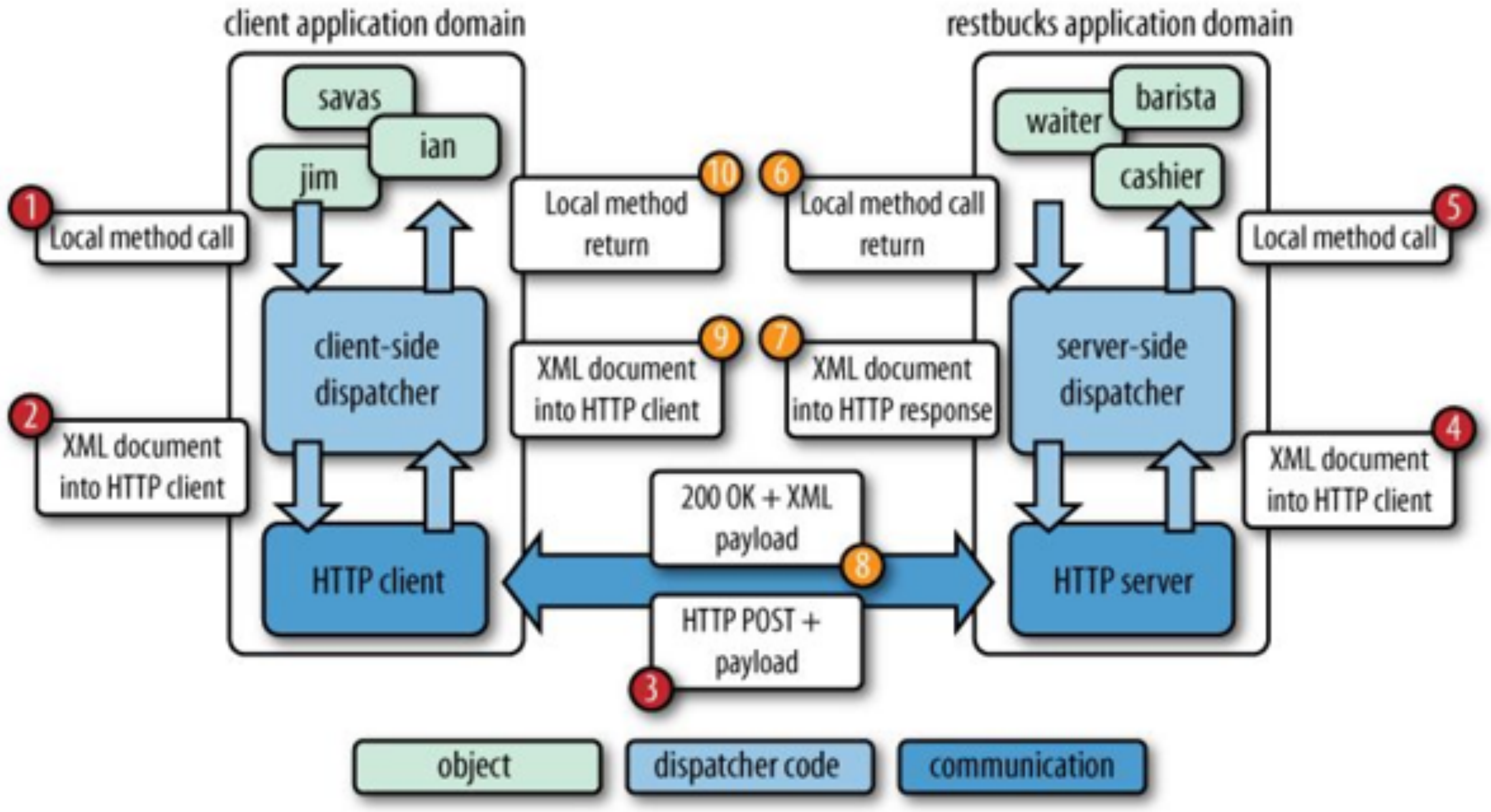
URI Tunnelling

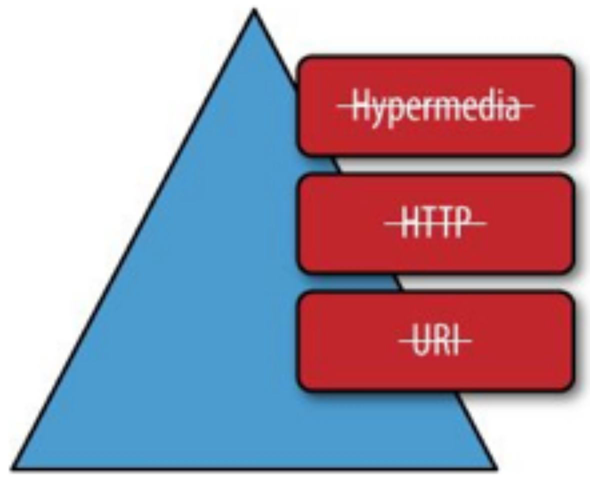
- URI Tunnelling
  - transfer information across system boundaries - encode information in the URI itself [GET/POST]
  - http:// restbucks.com/ PlaceOrder? coffee={ type}& size={size}& milk={ milk}& location ={location}
- Not Web-friendly
  - Encoding operations and not resources!!!



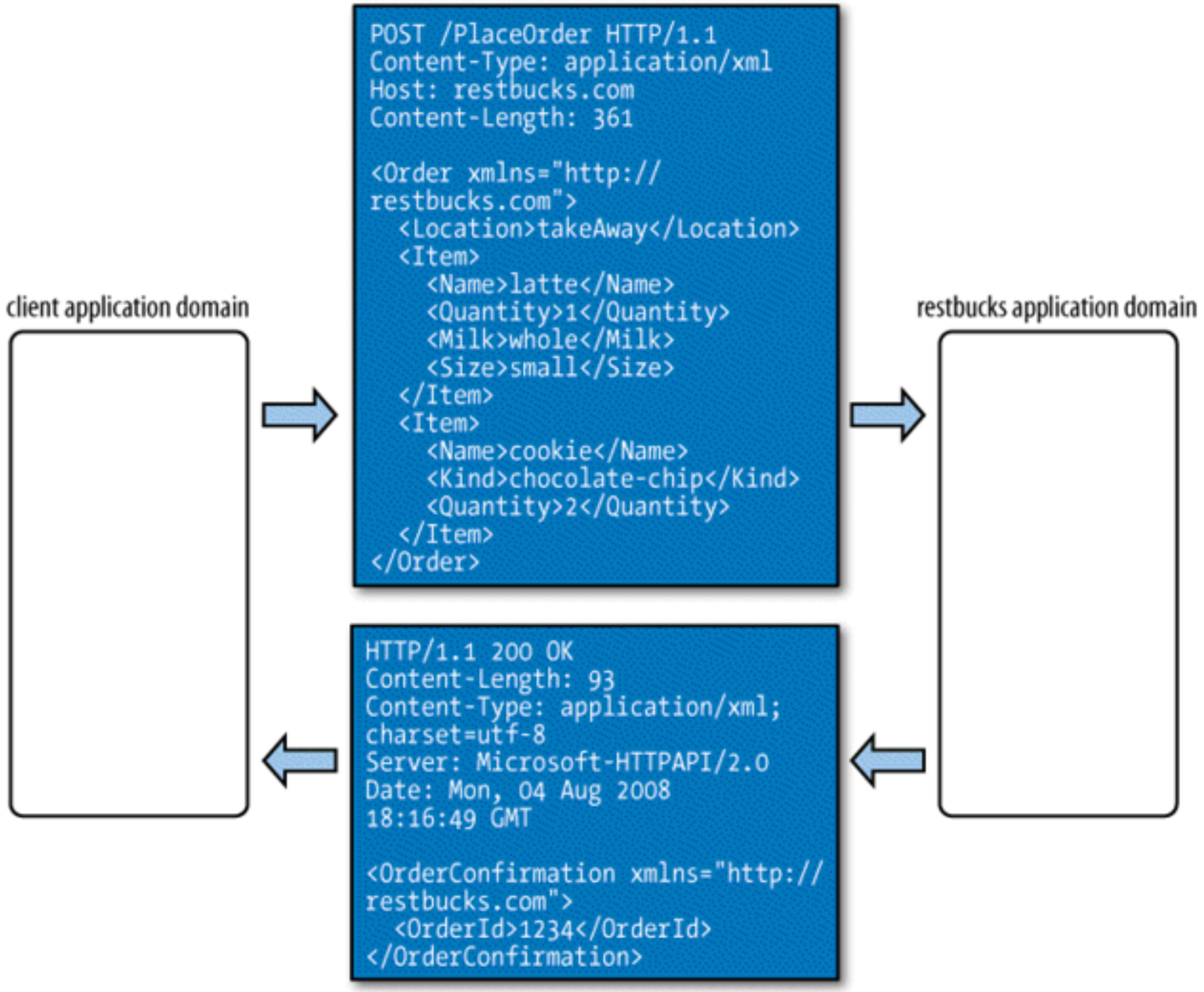
Plain Old XML over HTTP

- HTTP Requests and Responses transfer XML
  - Gives platform independence
  - More complex data structures and sophistication
- Not robust.
  - Does not support reliable messaging, transactions, etc.





Restbucks example



```
POST /PlaceOrder HTTP/1.1
Content-Type: application/xml
Host: restbucks.com
Content-Length: 361

<Order xmlns="http://
restbucks.com">
  <Location>takeAway</Location>
  <Item>
    <Name>latte</Name>
    <Quantity>1</Quantity>
    <Milk>whole</Milk>
    <Size>small</Size>
  </Item>
  <Item>
    <Name>cookie</Name>
    <Kind>chocolate-chip</Kind>
    <Quantity>2</Quantity>
  </Item>
</Order>
```

```
HTTP/1.1 200 OK
Content-Length: 93
Content-Type: application/xml;
charset=utf-8
Server: Microsoft-HTTPAPI/2.0
Date: Mon, 04 Aug 2008
18:16:49 GMT

<OrderConfirmation xmlns="http://
restbucks.com">
  <OrderId>1234</OrderId>
</OrderConfirmation>
```

CRUD Services

CRUD

- Create, read, update and delete resources
  - Create an order when a customer makes a purchase
  - Read order for preparation
  - Update order if the customer changes their mind
  - Cancel order
- These map to HTTP verbs!
  - POST/GET/UPDATE/DELETE
  - HTTP is a good platform for CRUD services

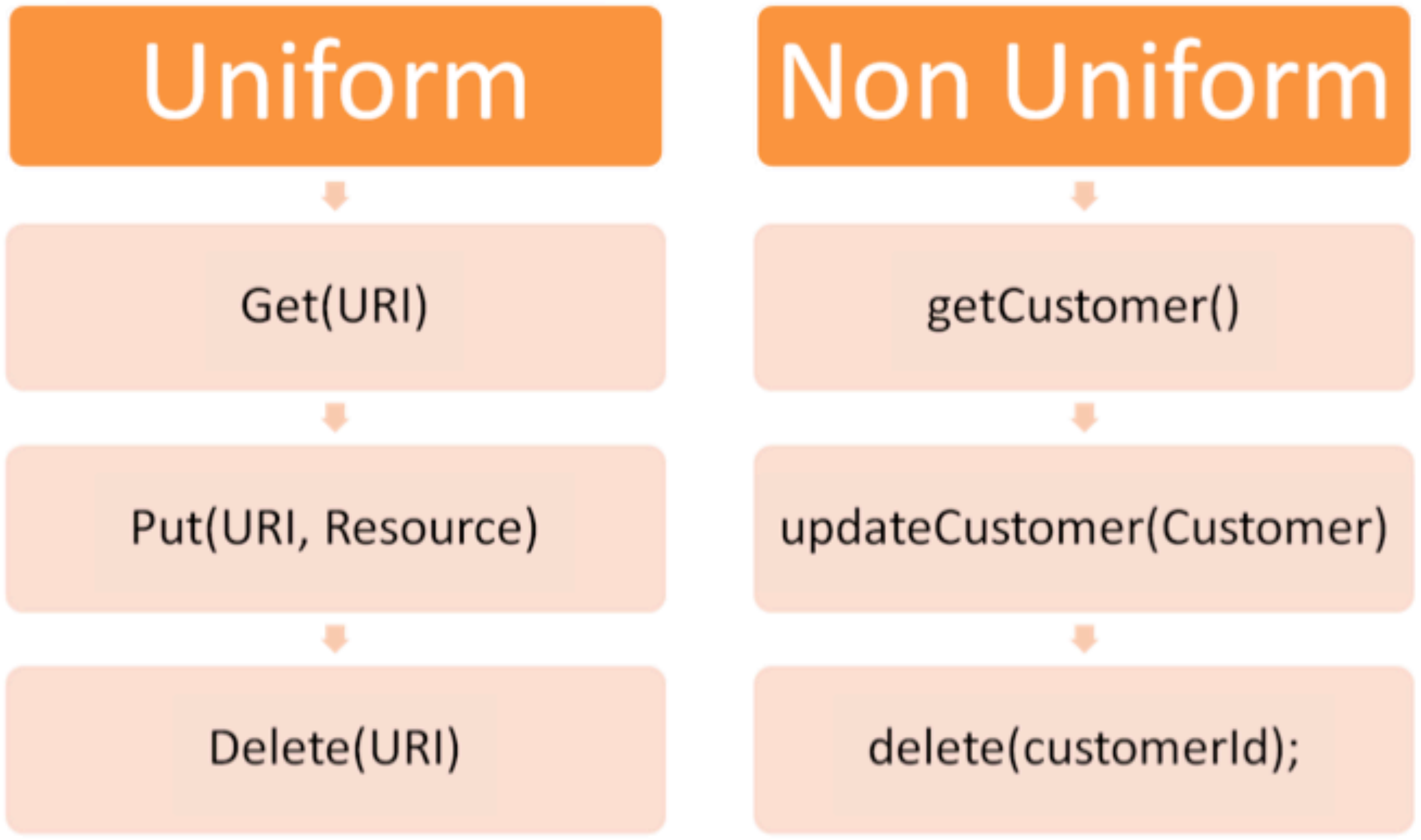
We are now starting to embrace the Web!!!

Uniform Interface

- GET
  - read-only operation
  - idempotent
    - no matter how many times you apply it, the resource is not affected
  - safe
    - its invocation does not change the state of the resource
- POST
  - like a resource upload
- DELETE
  - remove resources
- PUT
  - the only non-idempotent and unsafe operation
  - is allowed to modify the service in a unique way

- HEAD
  - HEAD is like GET except it returns only a response code
- OPTIONS
  - is used to request information about the communication options of the resource
  - It allows the client to determine the capabilities of a server and a resource without triggering actions

Uniform Interface



Why is this important?

Familiarity

If you have a URI that points to a service, you know exactly which methods are available on that resource.

Interoperability

HTTP is a very ubiquitous protocol.

Scalability

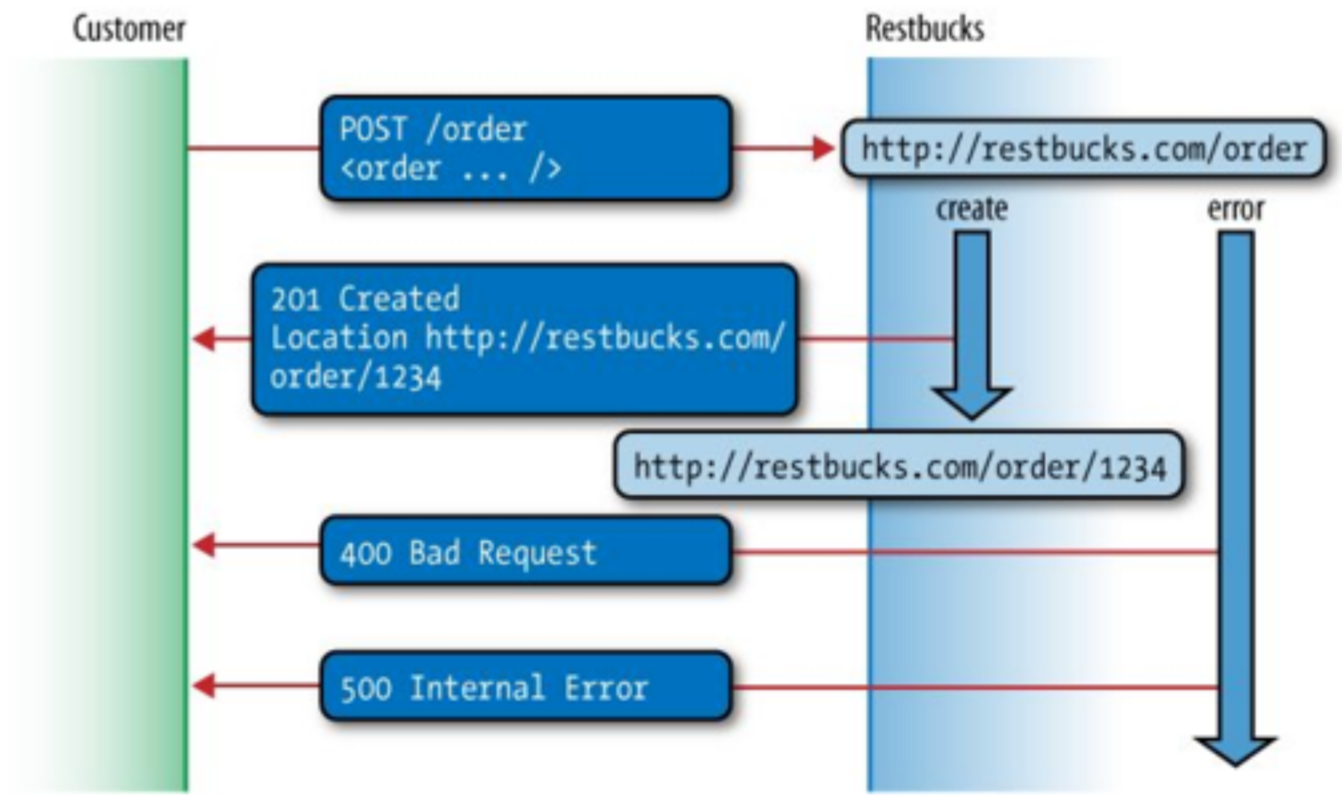
HTTP caching increases the service performance.

With PUT and DELETE neither the client nor the server has to worry about handling duplicate message delivery.

Restbucks example

POST	/order	Create a new order
GET	/order/{order_id}	Request current state
PUT	/order/{order_id}	Update the order with new full representation
DELETE	/order/{order_id}	Remove the order

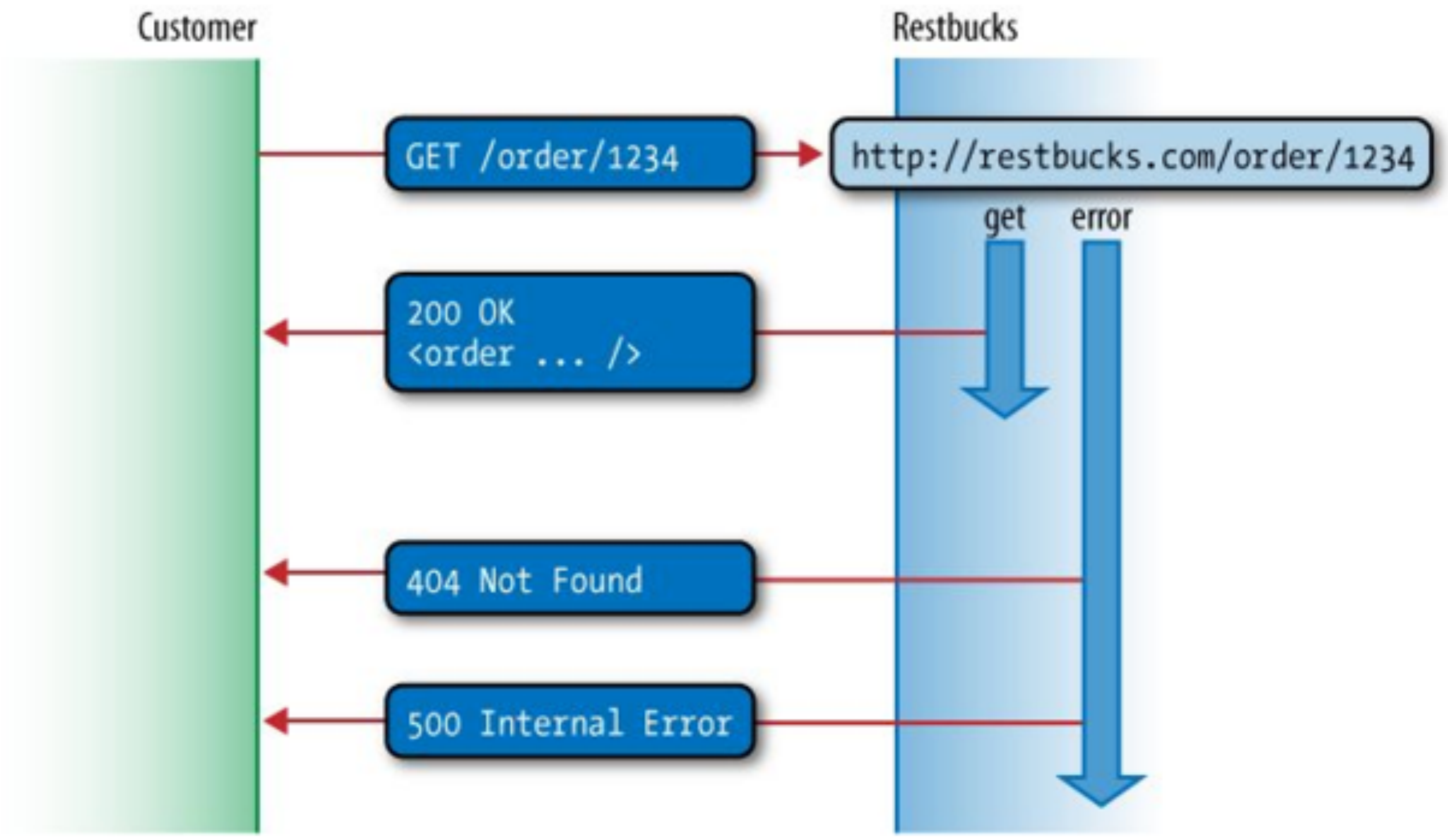
Create an order


Customer
Restbucks
POST /order
<order ... />
http://restbucks.com/order
create
error
201 Created
Location http://restbucks.com/order/1234
http://restbucks.com/order/1234
400 Bad Request
500 Internal Error

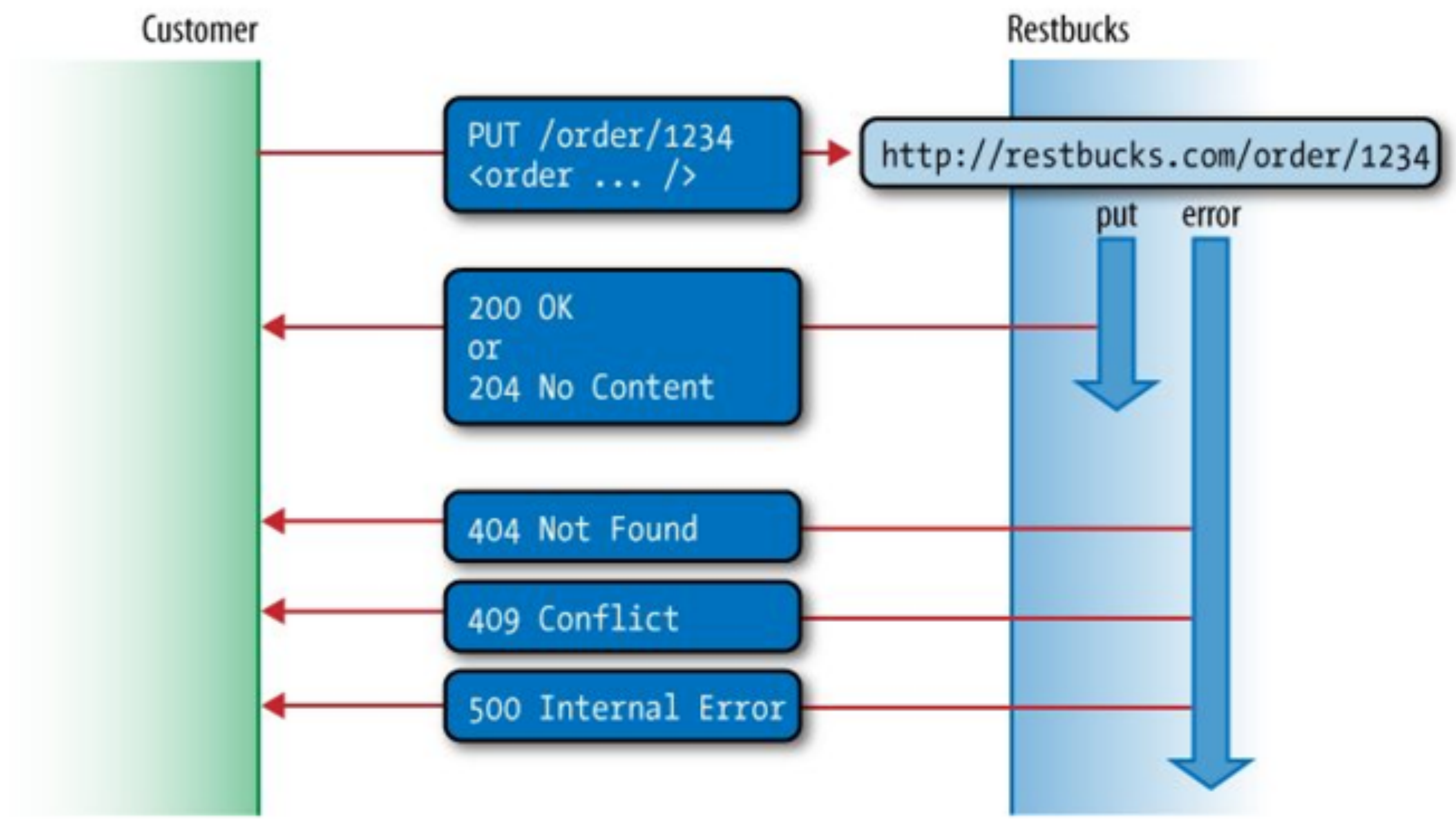
```
< order xmlns = http:// schemas.restbucks.com/ order >
        < location > takeAway </ location >
        < items >
                < item >
                        < name > latte </ name >
                        < quantity > 1 </ quantity >
                        < milk > whole </ milk >
   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

```
HTTP/ 1.1 201 Created
Content-Length: 267
Content-Type: application/ xml
Date: Wed, 19 Nov 2008 21: 45: 03 GMT
Location: http:// restbucks.com/ order/ 1234
```

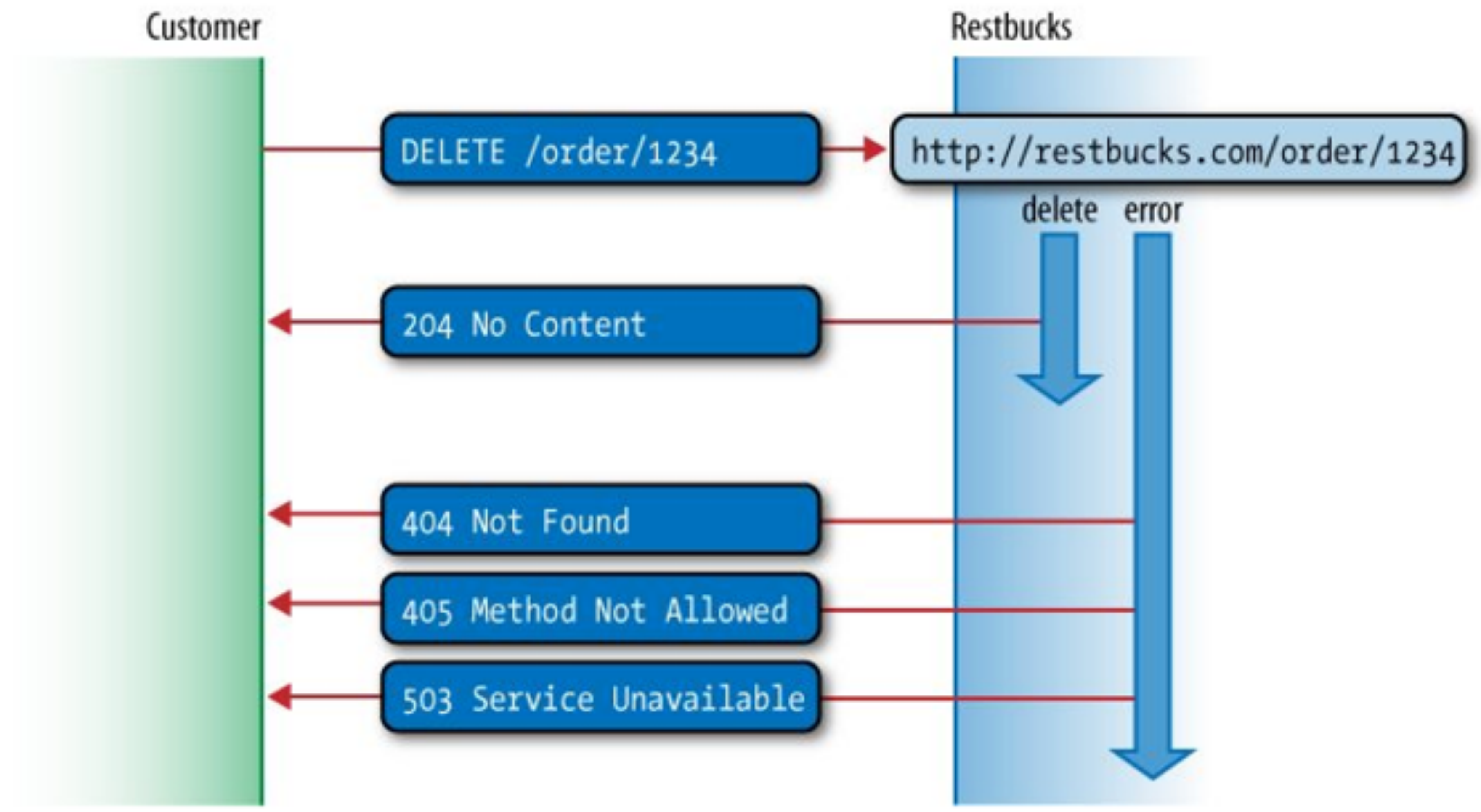
Reading an order



Updating an order

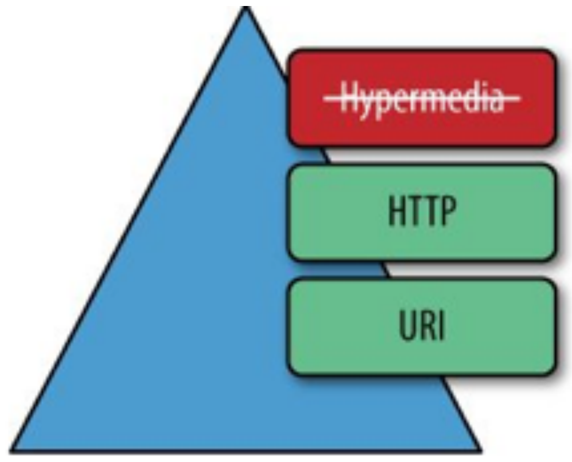


Deleting an order



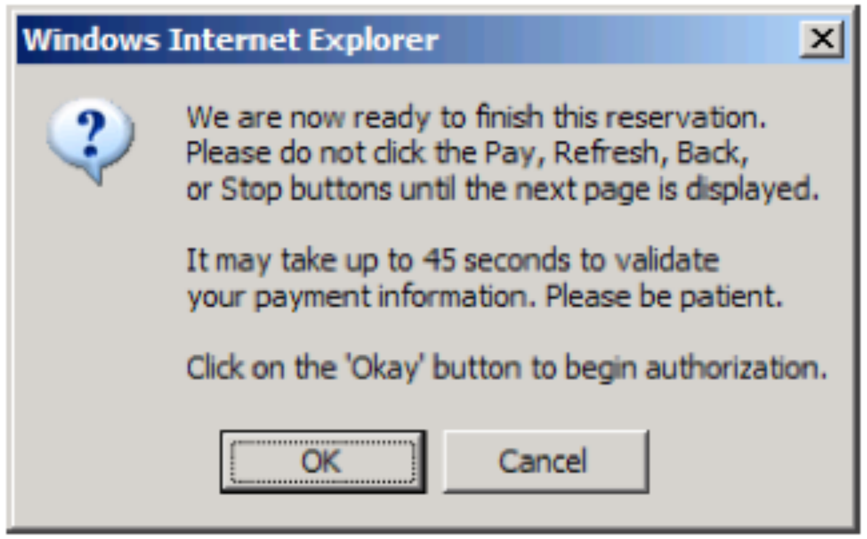
CRUD Conclusions

- Strengths
  - Viable and easy to implement
  - Good for systems that manipulate records
  - Many REST services stop here…
- Weaknesses
  - They are only suited for CRUD scenarios
  - More advanced scenarios need richer models
  - May also need stronger decoupling



POST vs GET

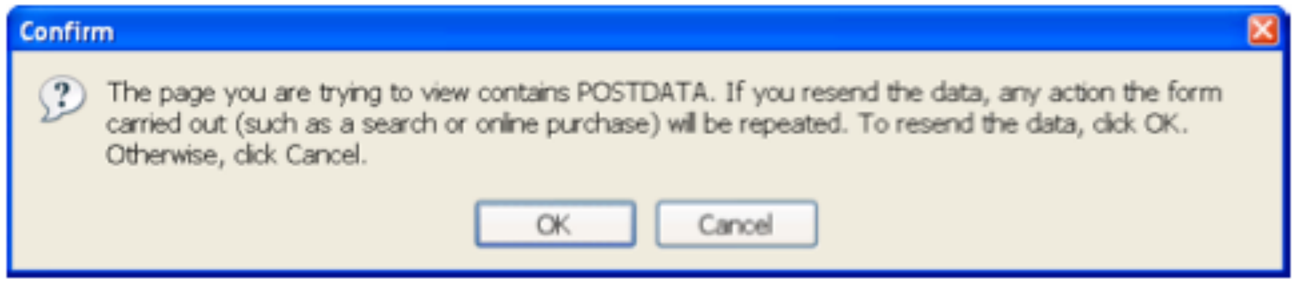
- GET
  - read-only operation - can be repeated without affecting the resource state
  - representation might not be the same
- POST
  - read-write operation - may change the state of the resource
  - browsers will warn when you refresh a page with a POST


Windows Internet Explorer

We are now ready to finish this reservation.
Please do not click the Pay, Refresh, Back,
or Stop buttons until the next page is displayed.

It may take up to 45 seconds to validate
your payment information. Please be patient.

Click on the 'Okay' button to begin authorization.


Confirm

The page you are trying to view contains POSTDATA. If you resend the data, any action the form carried out (such as a search or online purchase) will be repeated. To resend the data, click OK. Otherwise, click Cancel.

PUT vs POST

- What is the right way to create a resource?
  - PUT /resource/{id}
    - 201 created
    - The client ensures {id} is unique
  - POST /resource
    - 301 Moved Permanently - Location: /resource/{id}
    - The server computes the unique {id}

Adding Hypermedia

The essence of hypermedia

- When we browse we navigate through pages using “clicks”
- The interlinked pages realize a protocol!
    - buying books, searching for info, ordering a coffee…

- Hypermedia is part of our everyday lives
- Yet, it is not so common in computer-to-computer interactions

Hypermedia as the engine of application state

- What is application state?
- An app is a computerised behaviour that achieves a goal
- An app protocol is the set of legal interactions needed to realise the behaviour
- App state is a snapshot of an execution of such an app protocol

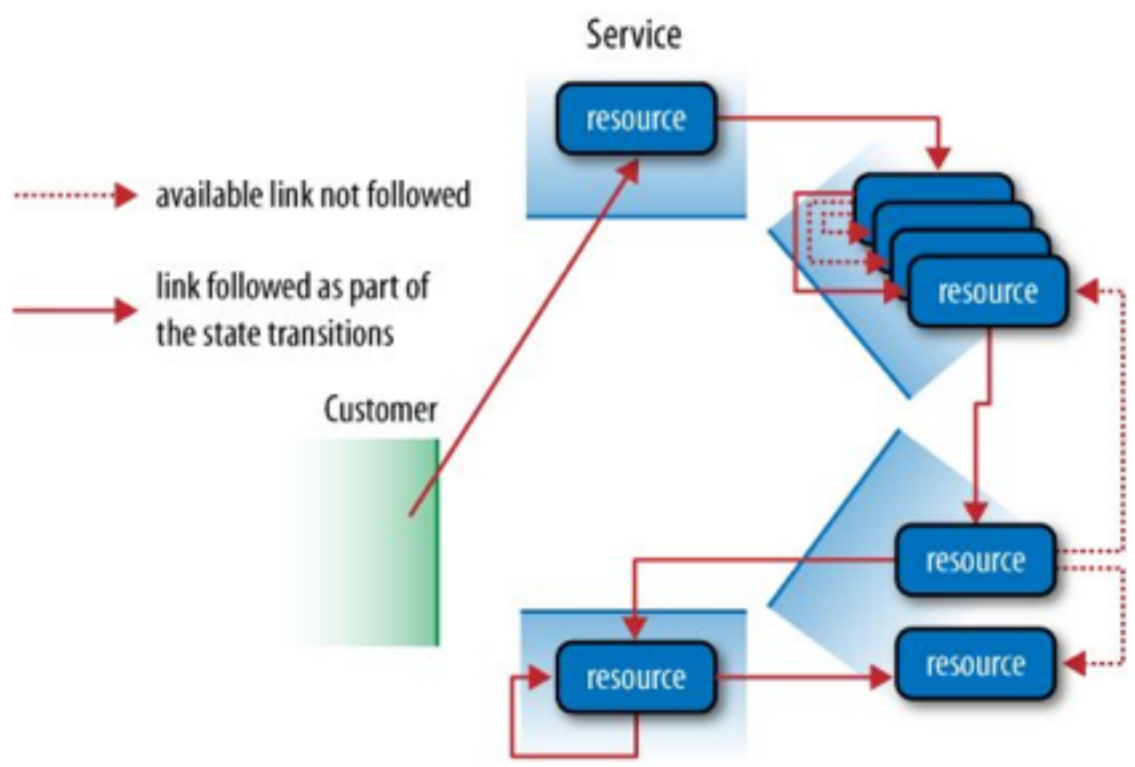
Transforming state

- A hypermedia app transfers links in the resource representations
- The links advertise the resources that participate in the app protocol

- A request receives a resource representation with a list of links
- By choosing a link the consumer gets one step closer to reaching his/her goal
    - The distributed application state changes!

- State transformation is the result of systemic behaviour of the whole

But what are we exchanging?


Service
available link not followed
link followed as part of the state transitions
Customer
resource
resource
resource
resource
resource

- On each interaction the resource and the consumer exchange resource state! NOT application state!
  - Application state is not recored in the representations
  - It is inferred by the consumer based on the state of all the resources

Loose coupling

- Decrease coupling by abstracting away implementation details
    - However, consumers have to have the information required to interact!
    - They need a way to drive the application protocol!

- The beauty of hypermedia is that it allows us to convey protocol information in a declarative and just-in-time fashion as part of the resource representations!

Computer-to-computer

- We advertise protocol information by embedding links in representations
- To describe why a link exists we annotate it!
    - We can use semantic micro-formats (XHTML)
    - We call these hypermedia controls

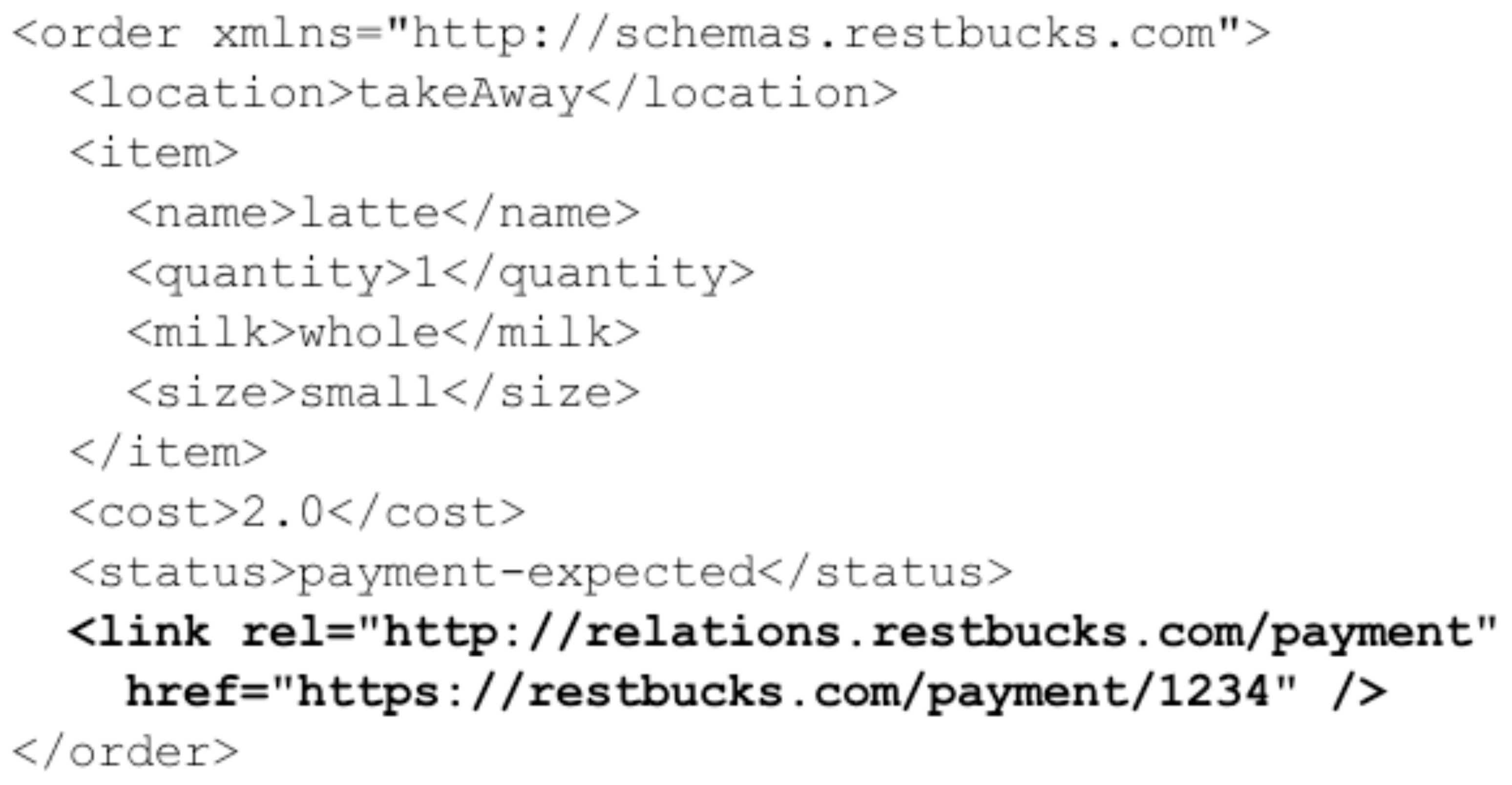
App-specific control formats

- We can enrich XML with our own control formats (closed hypermedia)

```
<order xmlns="http://schemas.restbucks.com">
  <location>takeAway</location>
  <item>
    <name>latte</name>
    <quantity>1</quantity>
    <milk>whole</milk>
    <size>small</size>
  </item>
  <cost>2.0</cost>
  <status>payment-expected</status>
  <payment>https://restbucks.com/payment/1234</payment>
</order>
```

Separation of Concerns

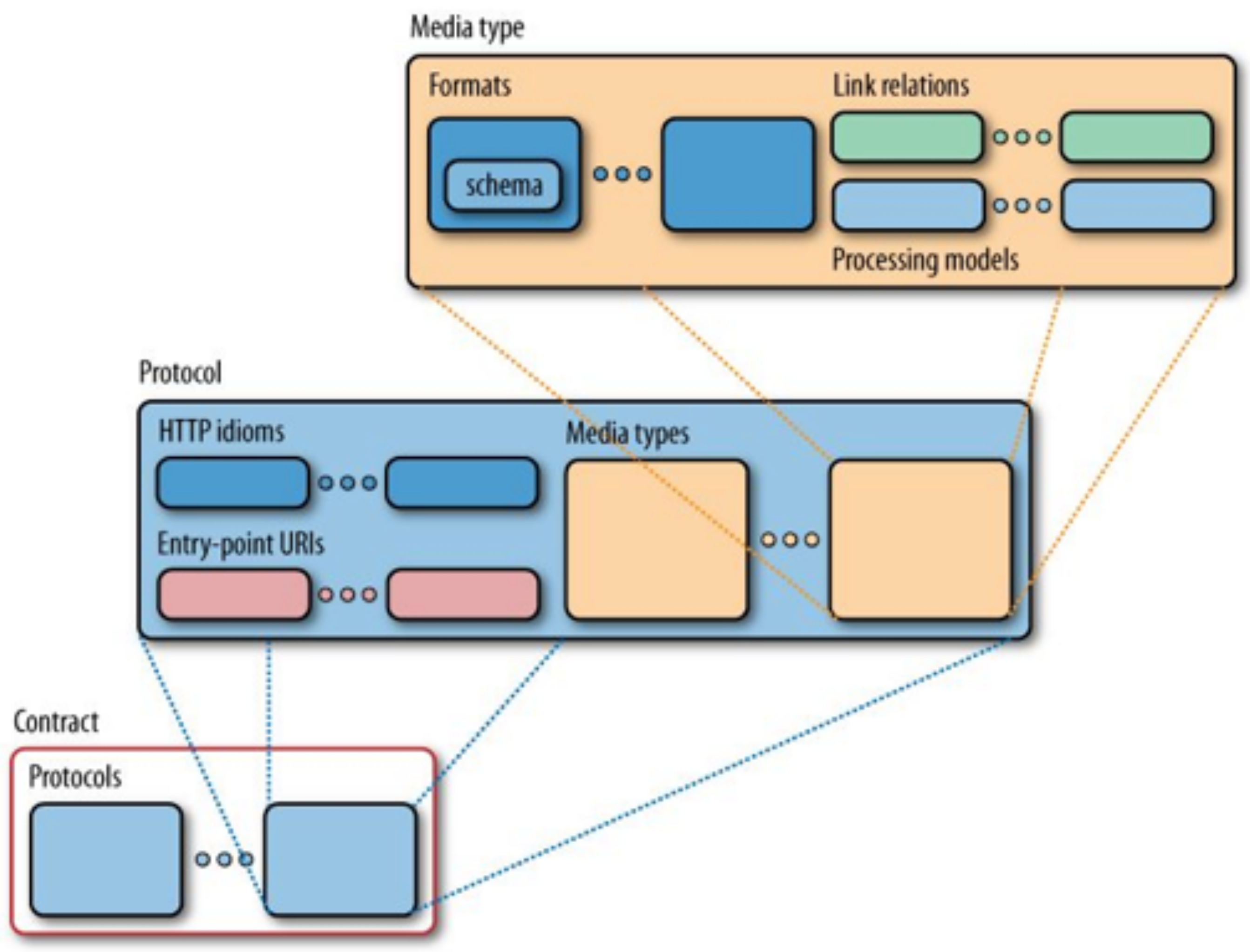
- Distinguish between linking and adding meaning to linking
    - The latter change from context to context
- <link> - domain-agnostic link function
    - @rel - application semantics



```
<order xmlns="http://schemas.restbucks.com">
  <location>takeAway</location>
  <item>
    <name>latte</name>
    <quantity>1</quantity>
    <milk>whole</milk>
    <size>small</size>
  </item>
  <cost>2.0</cost>
  <status>payment-expected</status>
  <link rel="http://relations.restbucks.com/payment"
    href="https://restbucks.com/payment/1234" />
</order>
```

Media types and Contracts

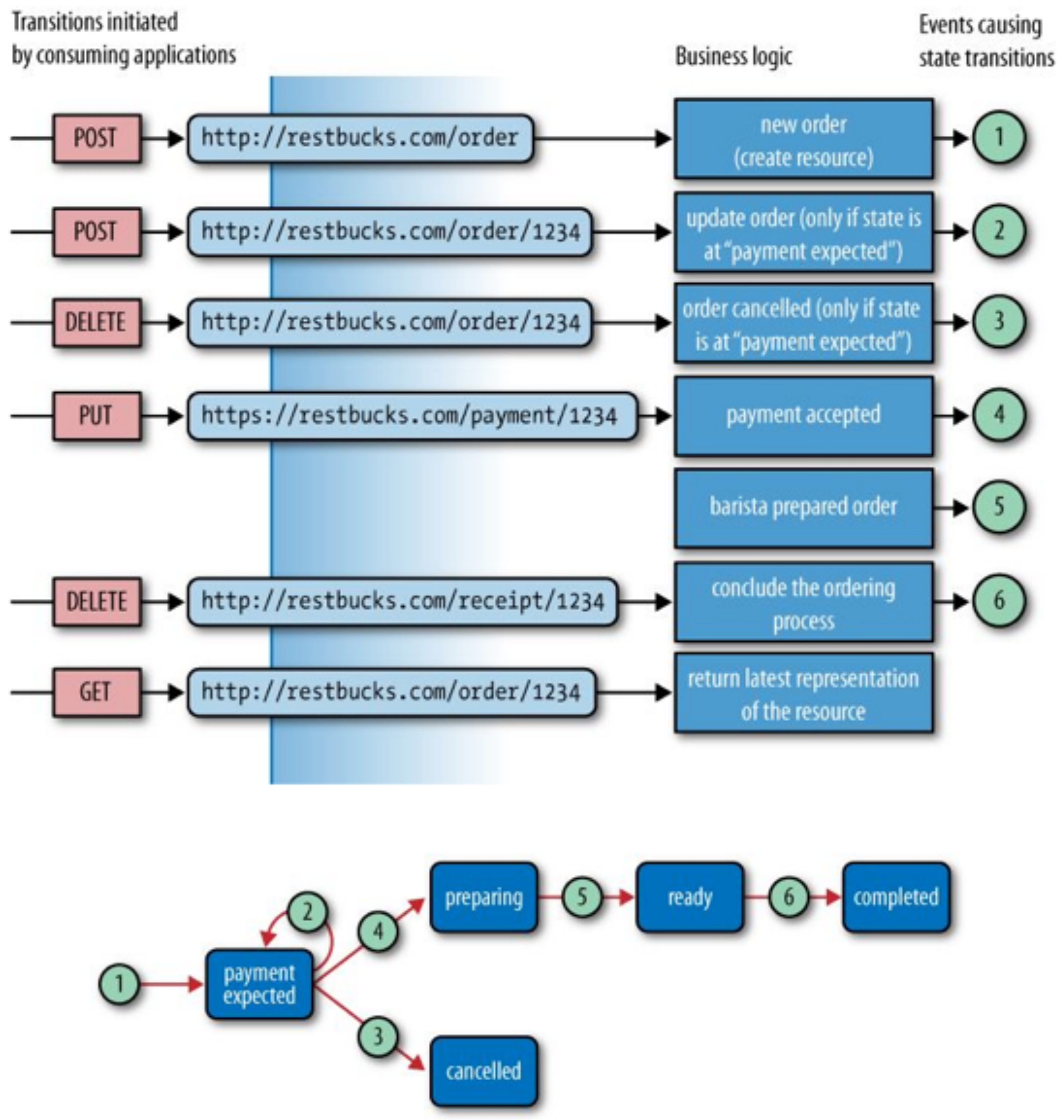
- How to process and reason about representations?
- A media type specifies an interpretative schema
    - application/vnd.restbucks + xml



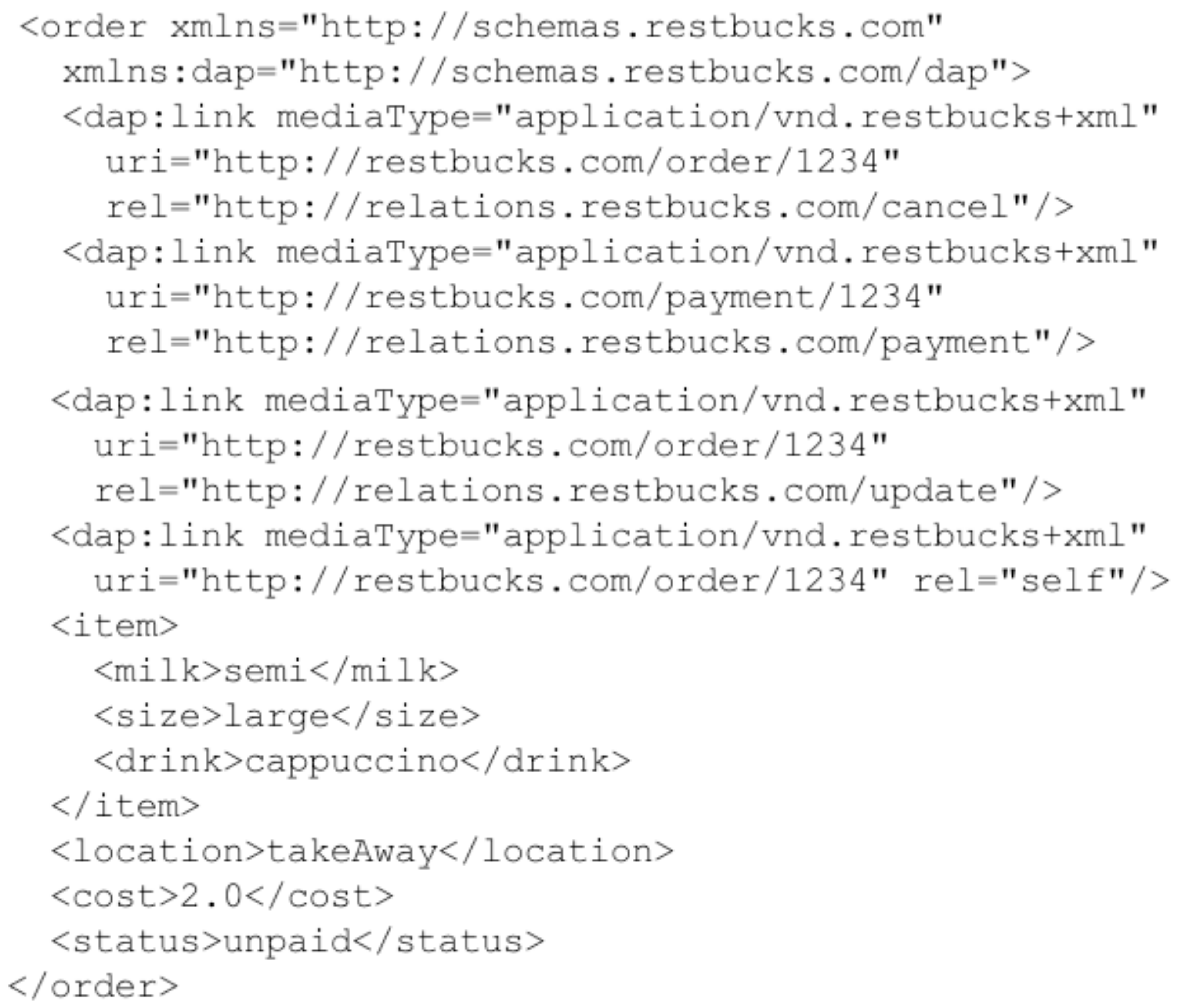
Extending Contracts with Protocols

- Protocols add new link relations and processing models
- Link relations tell us why we might want to follow a link
- The processing model augments the set of HTTP verbs

The Restbucks Example


Transitions initiated by consuming applications
Business logic
Events causing state transitions
POST
http://restbucks.com/order
new order (create resource)
1
POST
http://restbucks.com/order/1234
update order (only if state is at "payment expected")
2
DELETE
http://restbucks.com/order/1234
order cancelled (only if state is at "payment expected")
3
PUT
https://restbucks.com/payment/1234
payment accepted
4
barista prepared order
5
DELETE
http://restbucks.com/receipt/1234
conclude the ordering process
6
GET
http://restbucks.com/order/1234
return latest representation of the resource
payment expected
preparing
ready
completed
cancelled

Example of an Exchange



```
<order xmlns="http://schemas.restbucks.com"
  xmlns:dap="http://schemas.restbucks.com/dap">
  <dap:link mediaType="application/vnd.restbucks+xml"
    uri="http://restbucks.com/order/1234"
    rel="http://relations.restbucks.com/cancel"/>
  <dap:link mediaType="application/vnd.restbucks+xml"
    uri="http://restbucks.com/payment/1234"
    rel="http://relations.restbucks.com/payment"/>
  <dap:link mediaType="application/vnd.restbucks+xml"
    uri="http://restbucks.com/order/1234"
    rel="http://relations.restbucks.com/update"/>
  <dap:link mediaType="application/vnd.restbucks+xml"
    uri="http://restbucks.com/order/1234" rel="self"/>
  <item>
    <milk>semi</milk>
    <size>large</size>
    <drink>cappuccino</drink>
  </item>
  <location>takeAway</location>
  <cost>2.0</cost>
  <status>unpaid</status>
</order>
```

Scaling

Caching

- GET
    - safe and idempotent operation/verb
    - only used for retrieval and NOT modification
    - can be integrated with existing caching techniques

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

```
HTTP/ 1.1 200 OK
Content-Length: ...
Content-Type: application/ vnd.restbucks + xml
Date: Fri, 26 Mar 2010 10: 01: 22 GMT
Last-Modified: Fri, 26 Mar 2010 09: 55: 15 GMT
Cache-Control: max-age = 3600
ETag: 74f4be4b
< order xmlns = http:// schemas.restbucks.com >
        < location > takeaway </ location >
        < item >
                < drink > latte </ drink >
                < milk > whole</ milk >
                < size > large </ size >
        </ item >
</ order
```

Caching

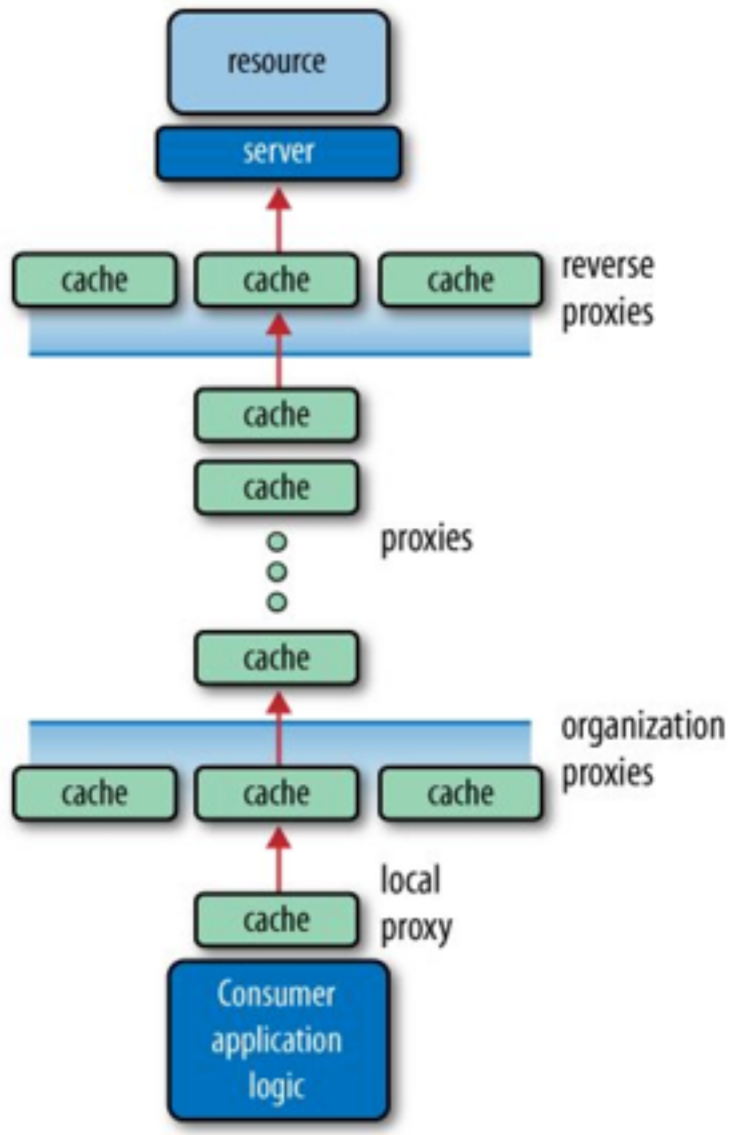
- Store copies of frequently accessed data somewhere along the request/response path
- If a cache satisfies a request it is used - the path is short-circuited
  - if a request is not captured it reaches the *origin server*
- Origin servers control the caching
  - they look at the HTTP headers to see *what* can be cached and *for how long*
  - a cache is given a *freshness lifetime*
  - an representation can become *stale* (need to be *revalidated*)
- Responses can have Age headers
  - how long a resource representation has been cached (seconds)

Benefits of Caching

- Reduce bandwidth
  - we reduce the number of network hops
- Reduce latency
  - the representations are stored nearer to the user
- Reduce load on servers
  - we reduce the number of requests that actually hit the origin server
- Hide network failures
  - we can continue to respond to GETS even if the origin server is no longer available

Types of Caches

- Local cache
  - stores representations from many origin servers on behalf of a single user
- Proxy cache
  - stores representations from many origin servers on behalf o many consumers
    - hosted inside or outside a corporate firewall
    - can prevent requests actually hitting the Internet
- Reverse proxy
  - also called an accelerator - stores representations from one origin server on behalf of many consumers
  - located in front of an application server
  - they help improve redundancy and prevent popular resources from becoming server hotspots (Squid, Varnish, Apache Traffic Server, etc.)



Consistency

- Weak consistency is a feature of Web applications
  - A resource has no way of telling a consumer that it has changed
  - Risk of acting on stale data
  - Caching makes things worse
- We can improve consistency through
  - Invalidation
  - Validation
  - Expiration

Improving Consistency

- Invalidation
    - notify consumers and caches of changes to resources
    - requires a list of who to contact - against the fact that we do not want to keep application state
- Validation
    - consumers and caches can check the local copy against the server
    - uses bandwidth and places some extra load on the server
    - servers do not need to maintain anything
    - efficient
- Expiration
    - specify a TTL for each representation

Reasons for not caching

- When GET generates server-side effects that have business impact
  - logging traffic / customer counting / etc.
- When consumers cannot tollerate any latency between what they see and the actual resource state
  - caching exacerbates weak consistency
- When a response contains sensitive or personal data
  - security and caching co-exist to a certain extent
- When the data change very frequently
  - when revalidation would introduce too much overhead

Statelessness

- Key architectural tenet
  - servers and services should not preserve application state
- Statelessness helps make distributed applications fault-tolerant and horizontally scalable
- Downsides
  - consumers must transfer id information with each exchange
    - message size and bandwidth
  - Services should forget previous clients
    - publish and subscribe is not possible
    - we need to poll our resources

Caching helps mitigate the problems of statelessness

Polling

- Polling is what allows the application to scale
    - Every request warms the caches it meets as it travels to and from the origin server
- latency/scalability trade-off
    - Making representations cacheable we get massive scale, but introduce latency between the resource changing and the changes becoming visible to the consumers

Security

Web Security

- Confidentiality
    - keep information private while in transit
- Integrity
    - prevent information from being changed
- Identity
    - authenticate the parties involved
- Trust
    - authorising a party

HTTPS

- HTTPS plays a big part
    - vast numbers of secure interactions on the Web every day
    - it is mature, widely deployed, well understood and well researched
- Instead of exchanging HTTP requests over TCP, we transmit the requests over TLS (Transport Layer Security).
    - TLS provides integrity and confidentiality
    - Three phases
        - Handshake
        - Secure Session
        - Channel setup

Costs

- Keep in mind that HTTPS costs a lot
    - As such it should not be used always
    - It can limit scalability
    - Cryptography costs
    - limits caching because intermediaries cannot see - we can only use local caching

- We should ask ourselves…
    - What is the value of the resource?
- And we should look at patterns that we can use instead

Authorization with OAuth

- Enables services and apps to interact with resources hosted securely in third-party services, without requiring the owners of the resources share their credentials
- Terminology
  - Server (S) - the service hosting the protected resources
  - Resource owner (RO) - the owner of the resources stored on the server
  - Client (C) - the service that needs to access the protected resources

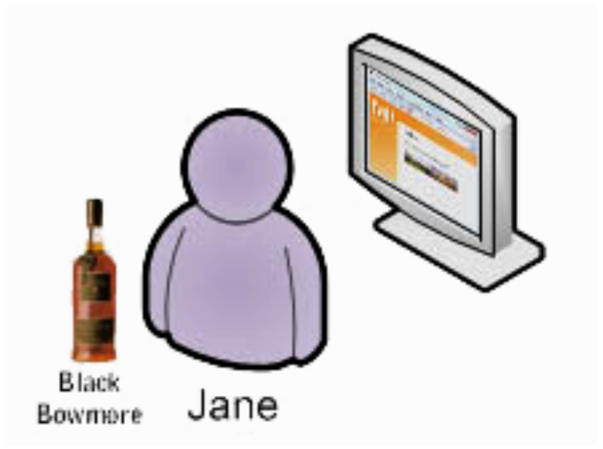
The client never sees the credentials

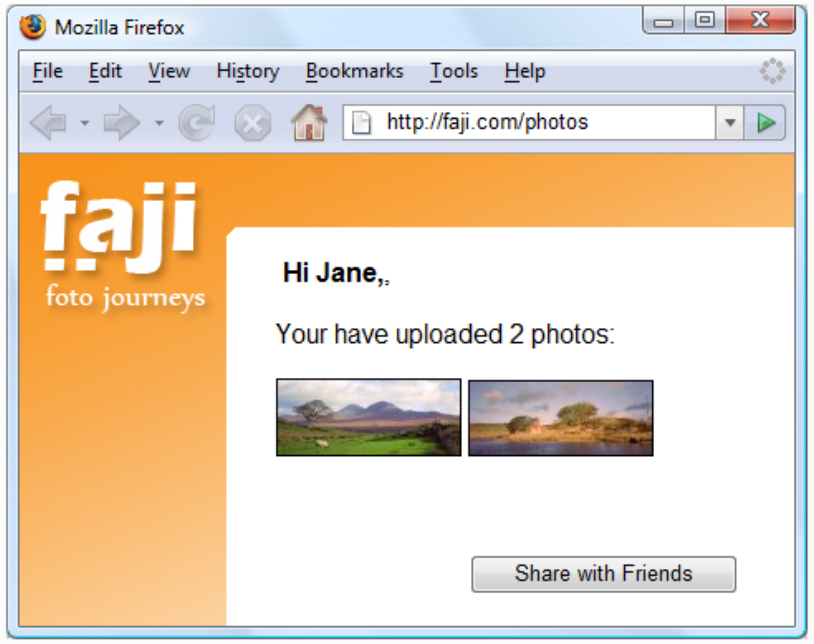
Credentials

- Client - established out-of-band between the server and the client
- Temporary - used to bootstrap the OAuth protocol
- Token - used to allow access to the protected resources after the protocol has completed successfully

Sharing Scotch

- Jane is back from her Scotland vacation. She spent 2 weeks on the island of Islay sampling Scotch. When she gets back home, Jane wants to share some of her vacation photos with her friends.
- Jane uses Faji, a photo sharing site, for sharing journey photos.
- She signs into her faji.com account, and uploads two photos which she marks private.
- Using OAuth terminology, Jane is the resource owner and Faji the server. The 2 photos are the protected resources.



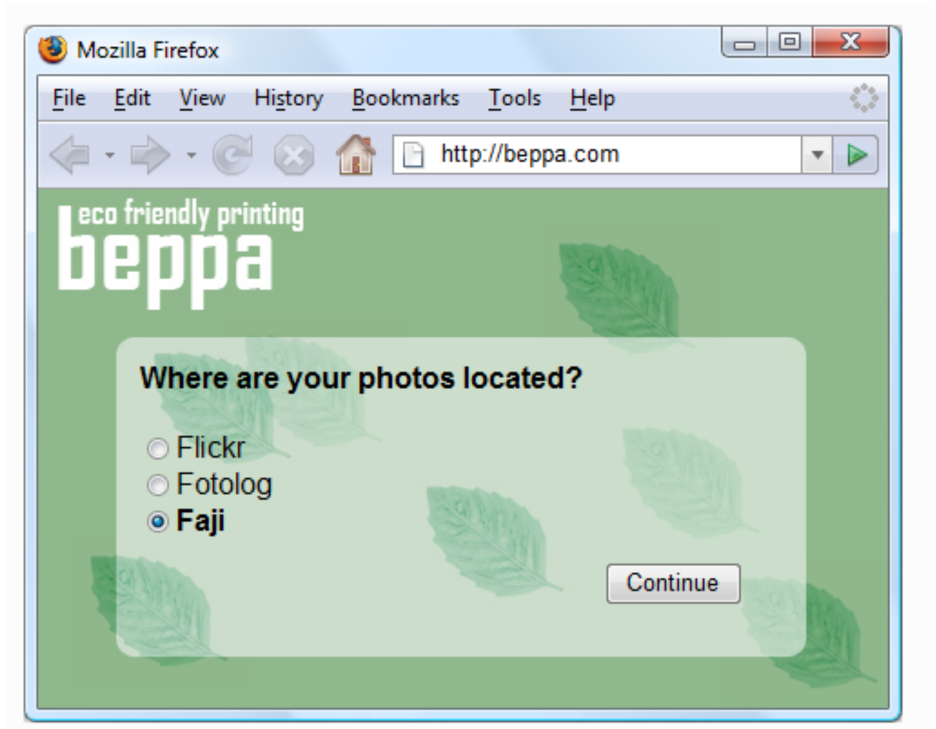


Hi Jane,,

Your have uploaded 2 photos:

Prints

- After sharing her photos with a few of her online friends, Jane wants to also share them with her grandmother.
- Being a responsible person, Jane uses Beppa, an environmentally friendly photo printing service.
- Using OAuth terminology, Beppa is the client. Since Jane marked the photos as private, Beppa must use OAuth to gain access to the photos in order to print them.

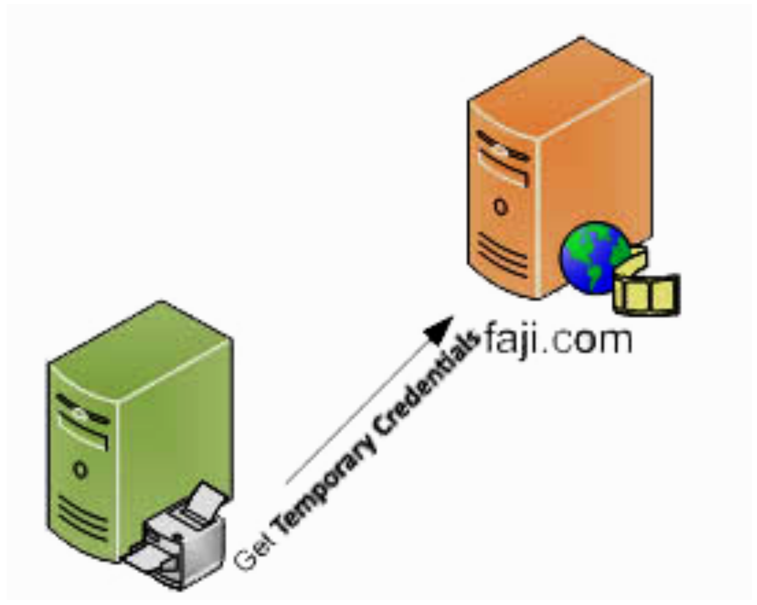


Where are your photos located?

Flickr [✓ Faji] Fotolog

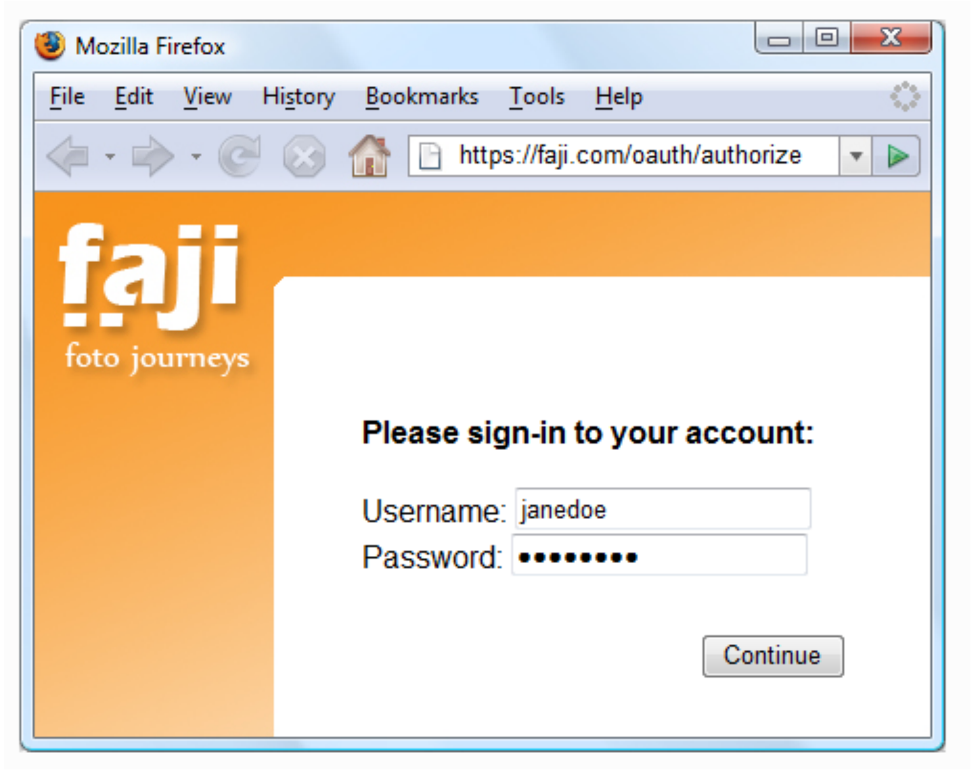
Credentials

- When Beppa added support for Faji photo import, a Beppa developer known in OAuth as a client developer obtained a set of client credentials (client identifier and secret) from Faji to be used with Faji's OAuth-enabled API.
- After Jane clicks Continue, Beppa requests from Faji a set of temporary credentials that are not resource-owner-specific, and can be used by Beppa to gain resource owner approval from Jane to access her private photos.



Redirection

- When Beppa receives the temporary credentials, it redirects Jane to the Faji OAuth User Authorization URL with the temporary credentials and asks Faji to redirect Jane back once approval has been granted to http://beppa.com/order.
- OAuth allows Jane to keep her username and password private and not share them with Beppa or any other site.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://faji.com/oauth/authorize
faji
foto journeys
Continue

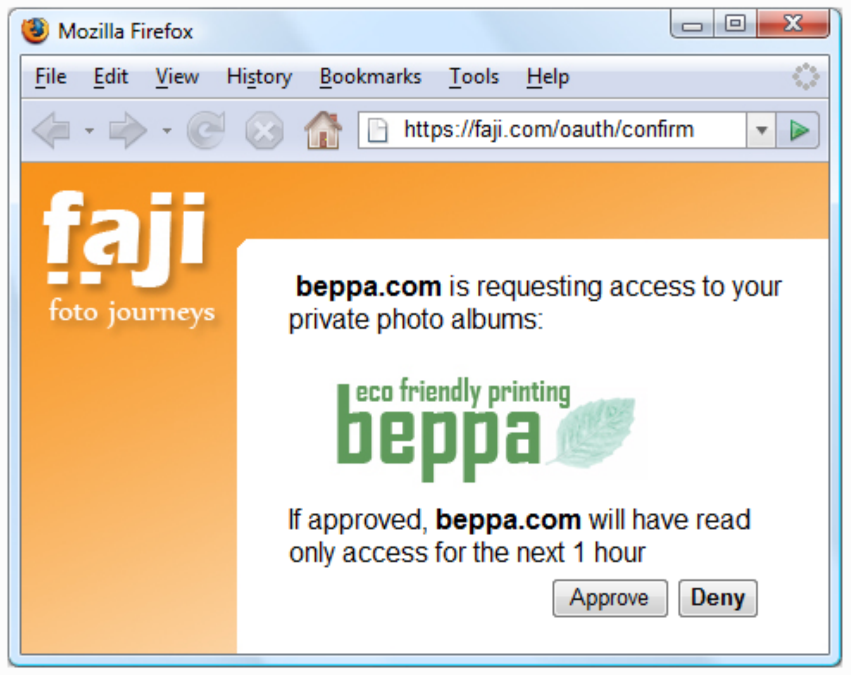
Please sign-in to your account:

Username: janedoe

Password: ••••••••

Getting Access

- Jane is asked to grant access to Beppa. Faji informs Jane of who is requesting access and the type of access being granted. Jane can approve or deny access.
- Faji marks the temporary credentials as resource-owner-authorized by Jane.
- Jane's browser is redirected back to http://beppa.com/order together with the temporary credentials identifier.
- This allows Beppa to know it can now continue to fetch Jane's photos.

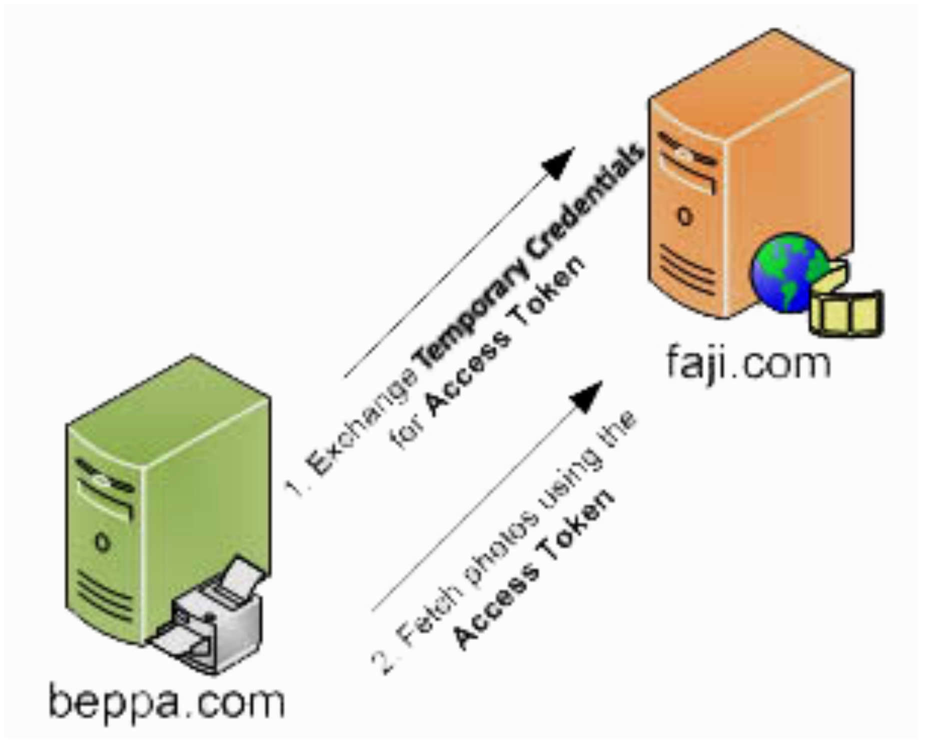

Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://faji.com/oauth/confirm
faji
foto journeys
eco friendly printing
beppa
Approve
Deny

beppa.com is requesting access to your private photo albums:

If approved, **beppa.com** will have read only access for the next 1 hour

Access Tokens

- Beppa uses the authorized Request Token and exchanges it for an Access Token.
- Request Tokens are only good for obtaining User approval, while Access Tokens are used to access Protected Resources. In the first request, Beppa exchanges the Request Token for an Access Token and in the second (can be multiple requests, one for a list of photos, and a few more to get each photo) request gets the photos.


1. Exchange **Temporary Credentials** for **Access Token**
2. Fetch photos using the **Access Token**
faji.com
beppa.com

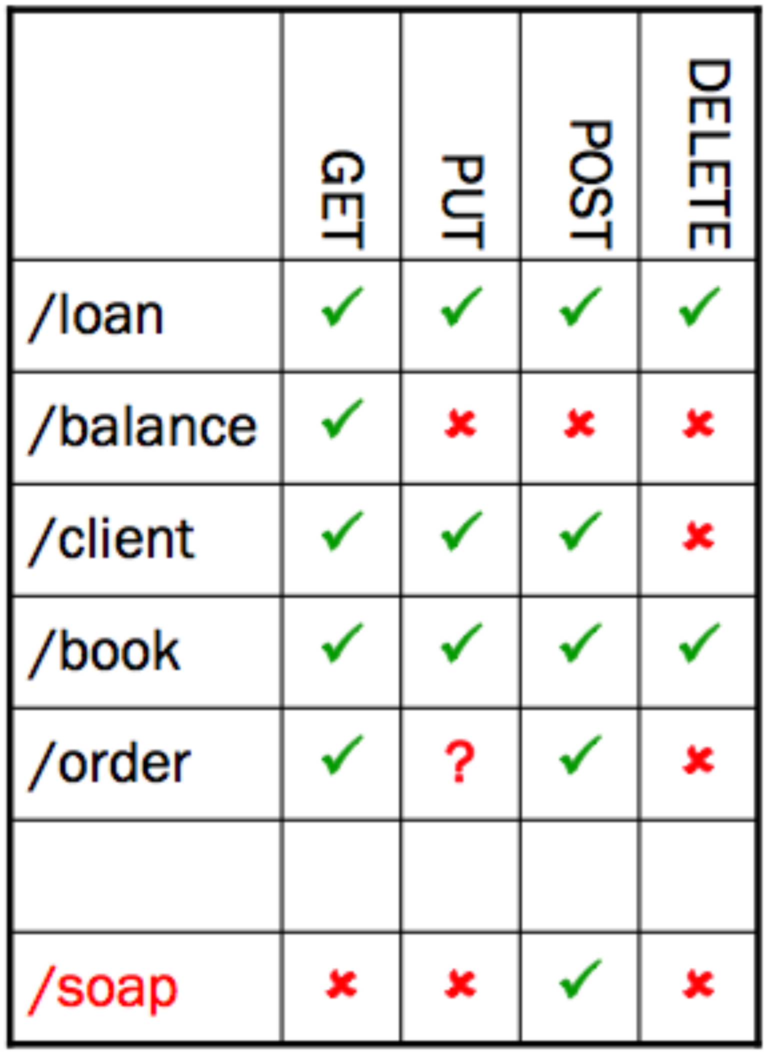
Let’s see some examples

Patterns and Anti-Patterns

Thanks to Prof. Cesare Pautasso for providing some of the following materiale

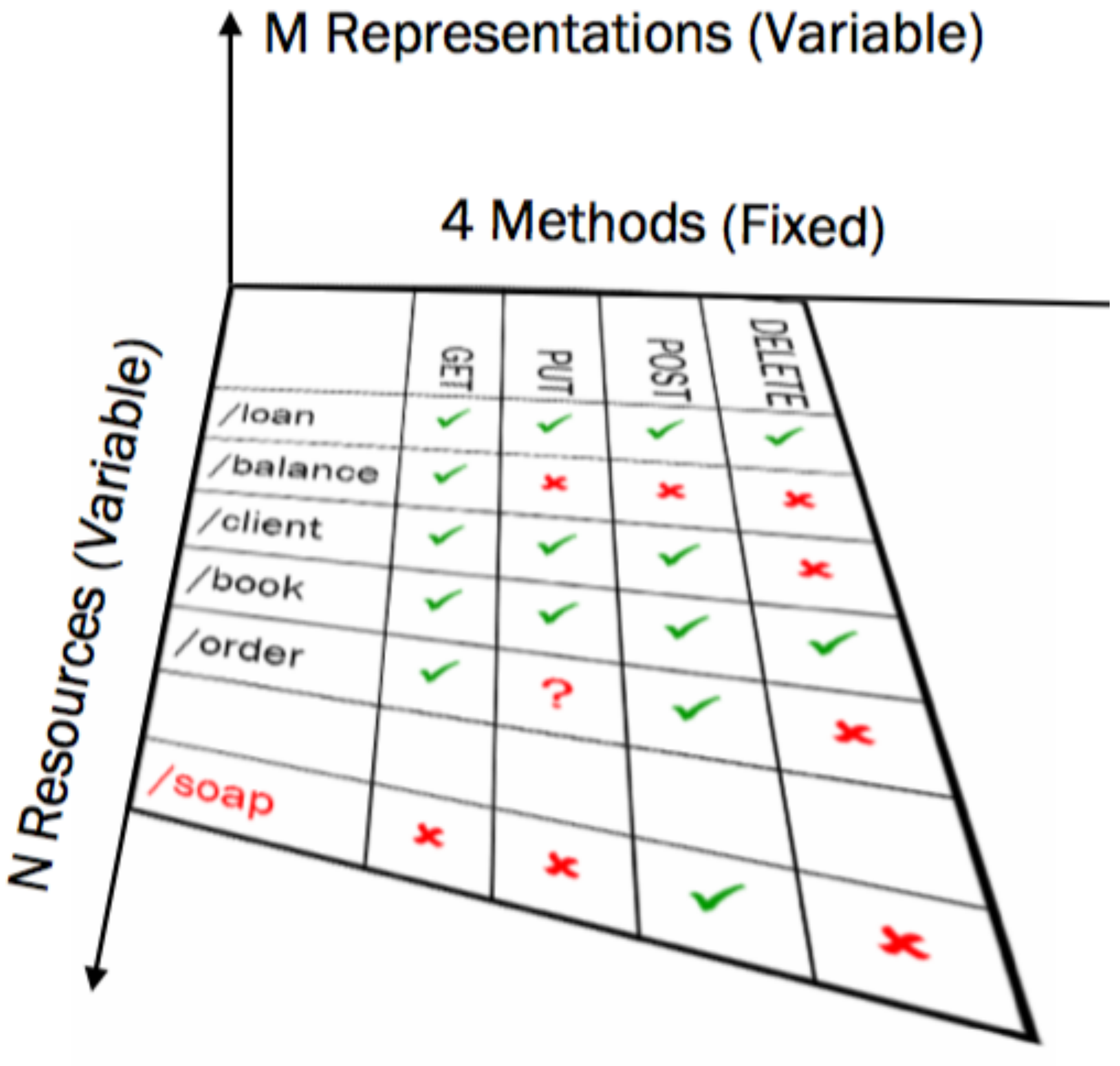
General Design Methodology

- Identify Resources
- Model relationships
- Define “nice” URIs to address the resources
- Understand what it means to GET, POST, PUT, and DELETE the resources
- Design and document the resource representations
- Implement and deploy
- Test



	GET	PUT	POST	DELETE
/loan	✓	✓	✓	✓
/balance	✓	✗	✗	✗
/client	✓	✓	✓	✗
/book	✓	✓	✓	✓
/order	✓	?	✓	✗
/soap	✗	✗	✓	✗

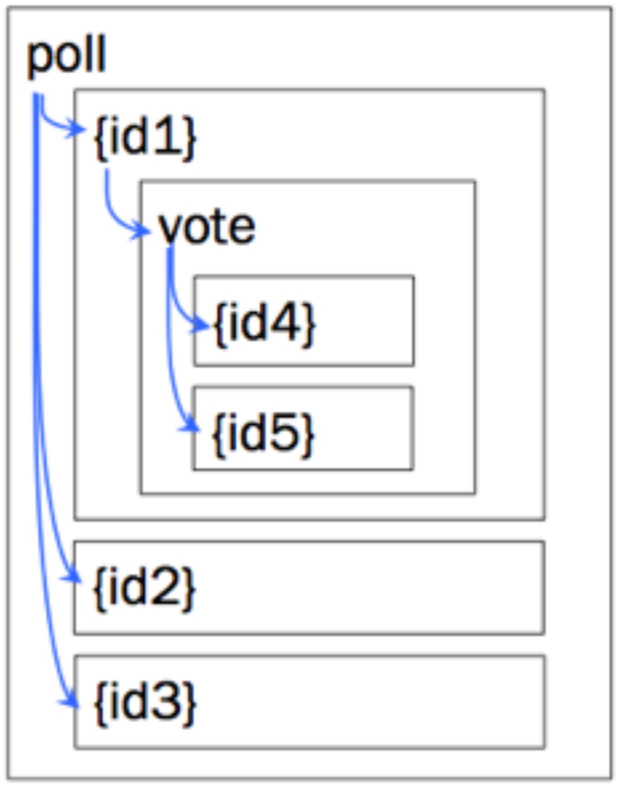
Design Space

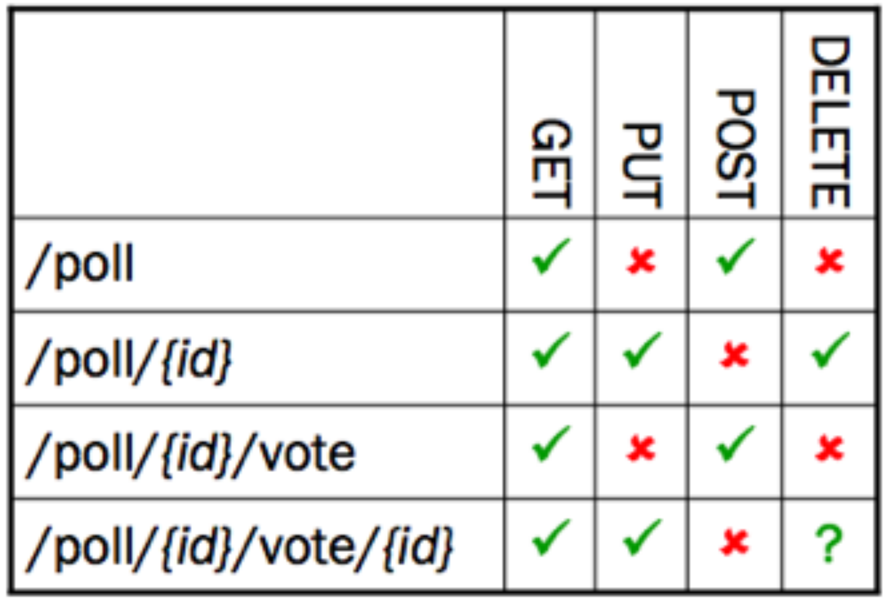


	GET	PUT	POST	DELETE
/loan	✓	✓	✓	✓
/balance	✓	✗	✗	✗
/client	✓	✓	✓	✗
/book	✓	✓	✓	✓
/order	✓	?	✓	✗
/soap	✗	✗	✓	✗

Simple Doodle API Example

- Resources -> polls and votes
- Containment Relationships
- URIs embed ids of children resources
- POST on the container is used to create
- PUT/DELETE for updates



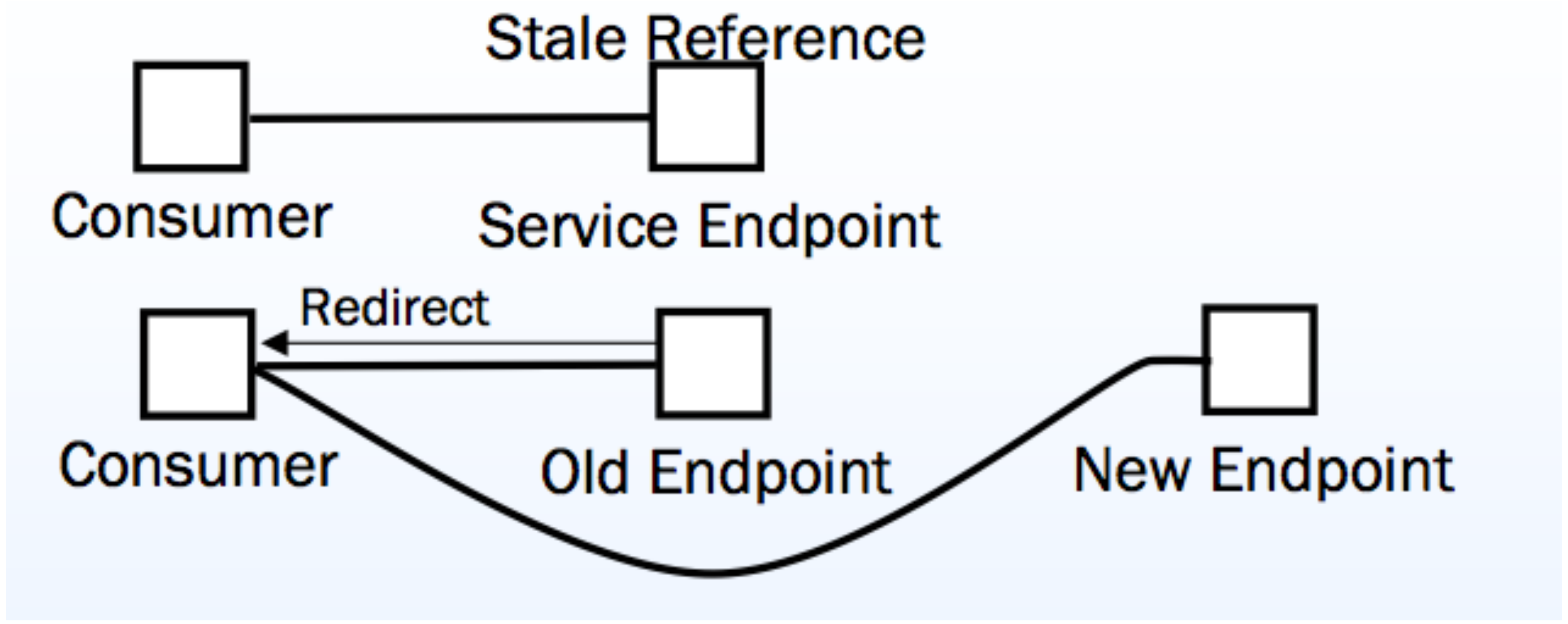


	GET	PUT	POST	DELETE
/poll	✓	✗	✓	✗
/poll/*{id}*	✓	✓	✗	✓
/poll/*{id}*/vote	✓	✗	✓	✗
/poll/*{id}*/vote/*{id}*	✓	✓	✗	?

Pattern: Endpoint Redirection

- How can consumers adapt when services inventories are restructured
  - They may change over time to accomodate business change
  - Automatically refer service consumers to the new identifier

Natively supported by HTTP



URI Design Guidelines

- Prefer nouns to verbs
- Keep URIs short
- Follow a “positional” parameter-passing scheme for algorithmic resource query string
    - Avoid key-value&p=v encoding
    - URIs are opaque and meant to be discovered through Hypermedia, not constructed by the user
- Evaluate URI postfixes for content type
    - This may break the abstraction
- Do not change URIs
    - use redirection

Pattern: Content Negotiation

- How can services support different customers without changing their contracts?
  - They may need to change and continue to support old consumers
  - Specific content and data representations should be negotiated at run time as part of the interaction - *multiple standard media types*
- Benefits - Loose Coupling, Increased Interoperability, Increased Organisational Agility

Comes for free with REST

```
GET /resource
Accept: text/html, application/xml, application/json

200 Ok
Content-type: application/json
```

The most appropriate format is chosen
or 406 is returned

Advanced Negotiation

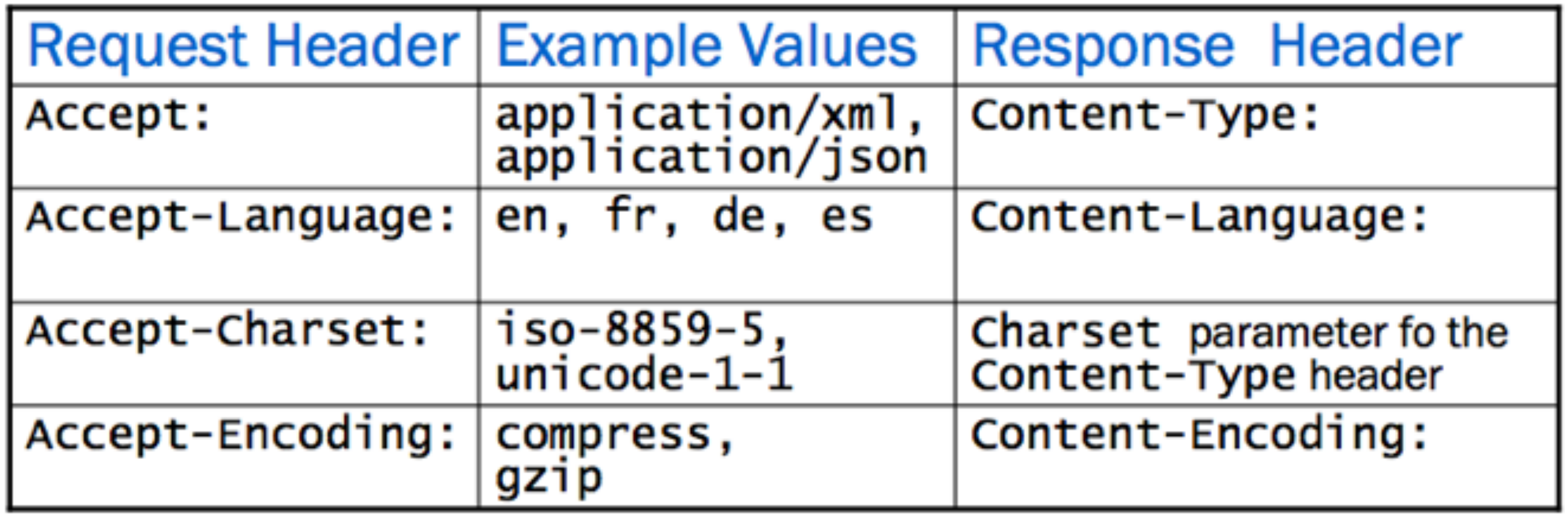
- Quality factors allow clients to specify preferences over representation type

Media/Type; q=x

Accept: application/xhtml+xml; q=0.9, text/html; q=0.5, text/plain; q=0.1

The client prefers to receive XHTML, or HTML if it is not available and will use plain text as a fall back

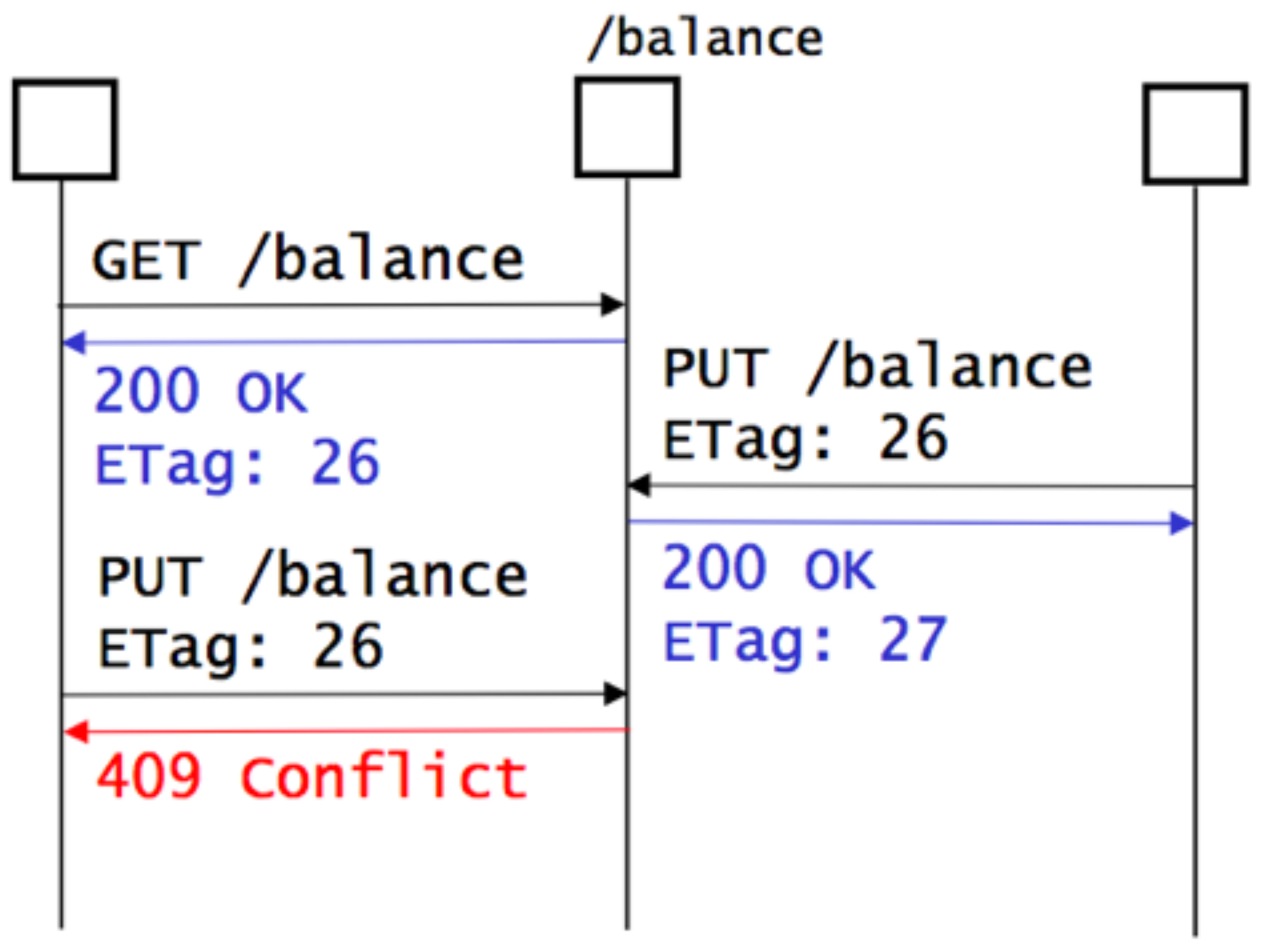
Content negotiation can be multidimensional



Request Header	Example Values	Response Header
`Accept:`	`application/xml, application/json`	`Content-Type:`
`Accept-Language:`	`en, fr, de, es`	`Content-Language:`
`Accept-Charset:`	`iso-8859-5, unicode-1-1`	`Charset` parameter fo the `Content-Type` header
`Accept-Encoding:`	`compress, gzip`	`Content-Encoding:`

Concurrency

- HTTP allows us to deal with concurrency
- The 409 code informs a client that his request would render the state of the resource inconsistent



AntiPattern: HTTP as a tunnel

- Tunnel through one HTTP method

```
GET /api?method=addCustomer&name=Guinea
GET /api?method=deleteCustomer&id=42
GET /api?method=getCustomerName&id=42
GET /api?method=findCustomers&name=Guinea
```

- Everything through GET
  - Easy to test through a browser
  - GET should only be used for idempotent actions
  - Requests can only send up to approx. 4KB
    - 414 Request-URI too long
- Everything through POST
  - Can upload/download arbitrary amount of data (useful for SOAP or XML/RPC)
  - POST is not idempotent and is unsafe to cache

AntiPattern: Ignoring Caching

- By including Cache-control: no-cache we prevent caching entirely
    - sometimes it is a default setting
- Supporting efficient caching and re-validation is one of the key benefits
    - use ETags - *mechanism for validating cache representations*

AntiPattern: Ignoring Status Codes

- HTTP has a very rich set of status codes
  - 200 (OK), 404 (not found), 500 (internal server error), etc. - *some are very common*
  - 201 (created - URI in Location header), 409 (conflict), 412 (precondition failed)
- Clients will depend on codes
- Some supposedly REST approaches only give 2xx and 5xx codes
  - some even only use 200 with error codes in body (SOAP)
- If we don't use codes we are missing an important opportunity for increased re-use, better interoperability, looser coupling

AntiPattern: Misusing Cookies

- Using cookies to propagate a key to some server-side session state is an anti-pattern
    - Cookies are never a good sign - *something is not RESTful*
    - We should not use cookies to identify conversations
    - A key idea of REST is statelessness
    - Session state is disallowed for scaling reasons
- If the cookie is used to carry something like an authentication token, but there is no session state then this is fine
    - However they should not be used if the information can be sent in some more traditional way

AntiPattern: Forgetting Hypermedia

- Adhering to a standard set of methods is already sometimes difficult
  - The result is a Web-based CRUD service
  - But it is not enough - *as we have seen*
- Hypermedia is what makes the Web a connected set of resources, where apps move from one state to the next by following links
- Indicators
  - absence of links in representations
  - link recipes on the client side or a mix of link constructing and links in representations
  - unnecessarily focusing on human readable links
  - ideally the client should only need to know one link!

AntiPattern: Ignoring MIME Types

- Content negotiation allows a client to retrieve different representations of resources based on needs - *XML, JSON, YAML, etc.*
- Using only one representation format is a missed opportunity!
- More unforeseen client will use the resource if we have multiple types
  - increase re-use
  - don’t re-invent the wheel and focus on existing types!

AntiPattern: Breaking Self-Descriptiveness

- So very common!
- Ideally a message should contain the information needed by any generic client, server, or intermediary
    - Think of how a PDF is retrieved and showed in your browser!
- Every time we invent new headers, formats, or protocols we break self-descriptiveness