goto; conference

Web API
DOs and DON'Ts

Oliver Wolf
@owolf

BERLIN INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE

gotocon.com

innoQ

Oliver Wolf
@owolf

www.innoQ.com
@innoQ

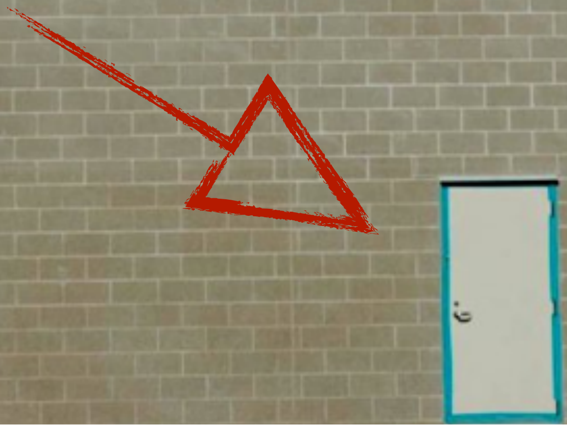
Disclaimer

- Some of the discussions around REST and Web APIs are merely a matter of taste and personal preference – I think the topics I’m going to bring up are not, but I’m as biased as everyone else.
- This is by no means a complete compilation and doesn’t even claim to be one.
- And, as always, your mileage may vary.

Don’t think in terms of “endpoints”.

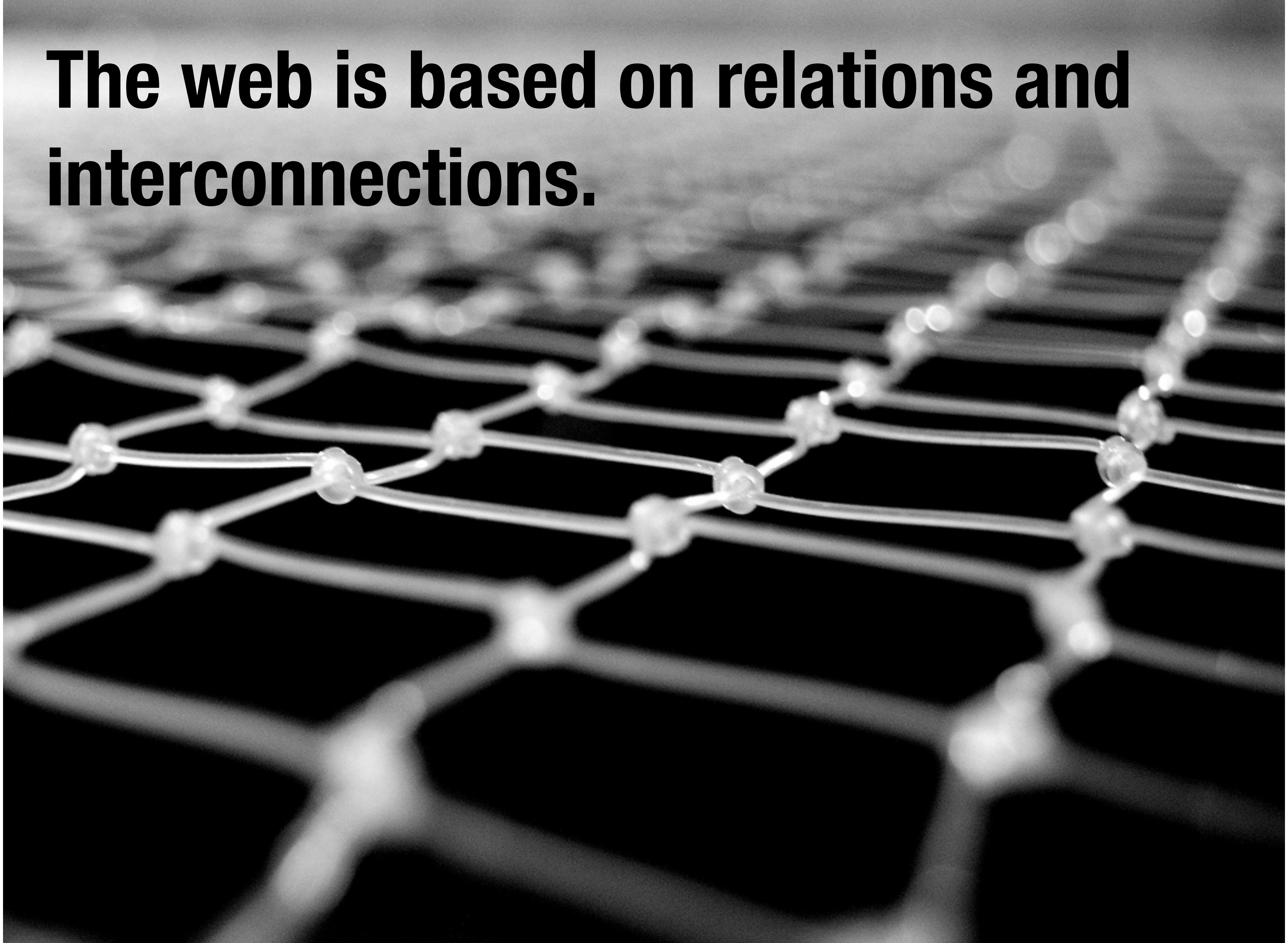
SOAP:
Facade with a single entry point

```
POST /soap/customer_service
<soap:envelope>
  <soap:body>
    <cs:create_customer>
      <cs:customer>
        <cs:name>John Doe</cs:name>
        ...
  </soap:body>
</soap:envelope>
```



The Web:
Lots of facades with lots of doors

Do you really want the web to end at your doorstep?



The web is based on relations and interconnections.

Don’t let your API be like a black hole with one way in and no way out

- Use hypermedia controls to link your resource representations together in ways that are meaningful for your audience.
- If your resource representations contain references to concepts and resources outside your domain, use hyperlinks whenever possible. That’s what they’re meant for!
- Make potential state transitions that apply to your resources visible and navigable via hypermedia controls rather than relying on out-of-band documentation.

Don’t just expose your domain model.

Many real-world domain models happen to be anemic.

If you just expose them as-is, you'll inevitably end up with bunch of CRUD resources

- This doesn't necessarily have to be bad thing, but it often is.
- A client that consumes a web API based on an anemic domain model needs to have intimate knowledge about the resources, their relations and the actions that can be performed on them – tight coupling ensues.
- It's almost always better to design APIs for intent rather than slavishly following the domain model.

Designing for intent means that you need to understand how clients will use the API

- That often requires a trade-off between flexibility vs. clarity and conciseness.
- Of course clients could request a list of the top 10 customers based on revenue like so:

```
GET /customers?sortBy=grossMargin&order=desc&pageSize=10
```

- But if that’s a frequent and meaningful use case for your API, why not introduce a new resource that explicitly conveys the intent:

```
GET /most_profitable_customers
```

Don’t overuse GET and POST.

```
GET /blog/entries/42&action=delete

POST /blog/entries/42/delete

POST /customer/123
  <customer>
    <status>Preferred</status>
  </customer>

GET /api/create_customer?name=...
```

The HTTP verbs are there for a reason – they have complementary qualities.

	Safe?	Idempotent?	Semantics
GET	✓	✓	retrieve resource representation
PUT	✗	✓	modify resource state or create resource identified by URL
POST	✗	✗	create new resource, leave assigning identifier to server
DELETE	✗	✓	delete resource

You gain a lot by using HTTP as it’s intended to be used.

- Using HTTP verbs correctly unambiguously communicates intent.
- The client knows excatly what to expect from the server:
  - Which actions can be safely retried in case of errors?
  - Which results can potentially be cached?
  - Which actions mutate server-side resource state?
- Coupling between client and server is limited to the HTTP contract, no out-of-band knowledge is required.

Don’t limit your choice
of error codes
to 200 and 500.

Life lesson:
Pretending everything’s good
when in fact it isn’t
is rarely a good idea.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    success:false,
    severity:100,
    error_message:"Everything's FUBAR!"
}
```

Srsly?

There are more than 60 error codes for you to choose from.

100 Client should continue with request
101 Server is switching protocols
102 Server has received and is processing the request
103 resume aborted PUT or POST requests
122 URI is longer than a maximum of 2083 characters

200 standard response for successful HTTP requests
201 request has been fulfilled; new resource created
202 request accepted, processing pending
203 request processed, information may be from another source
204 request processed, no content returned
205 request processed, no content returned, reset document view
206 partial resource return due to request header
207 XML, can contain multiple separate responses
208 results previously returned
226 request fulfilled, reponse is instance-manipulations

300 multiple options for the resource delivered
301 this and all future requests directed to the given URI
302 temporary response to request found via alternative URI
303 permanent response to request found via alternative URI
304 resource has not been modified since last requested
305 content located elsewhere, retrieve from there
306 subsequent requests should use the specified proxy
307 connect again to different URI as provided
308 resumable HTTP requests

400 request cannot be fulfilled due to bad syntax
401 authentication is possible but has failed
402 payment required, reserved for future use
403 server refuses to respond to request
404 requested resource could not be found
405 request method not supported by that resource
406 content not acceptable according to the Accept headers
407 client must first authenticate itself with the proxy
408 server timed out waiting for the request
409 request could not be processed because of conflict
410 resource is no longer available and will not be available again
411 request did not specify the length of its content
412 server does not meet request preconditions
413 request is larger than the server is willing or able to process
414 URI provided was too long for the server to process
415 server does not support media type
416 client has asked for unprovidable portion of the file
417 server cannot meet requirements of Expect request-header field
418 I'm a teapot
420 Twitter rate limiting
422 request unable to be followed due to semantic errors
423 resource that is being accessed is locked
424 request failed due to failure of a previous request
426 client should switch to a different protocol
428 origin server requires the request to be conditional
429 user has sent too many requests in a given amount of time
431 server is unwilling to process the request
444 server returns no information and closes the connection
449 request should be retried after performing action
450 Windows Parental Controls blocking access to webpage
451 The server cannot reach the client's mailbox.
499 connection closed by client while HTTP server is processing

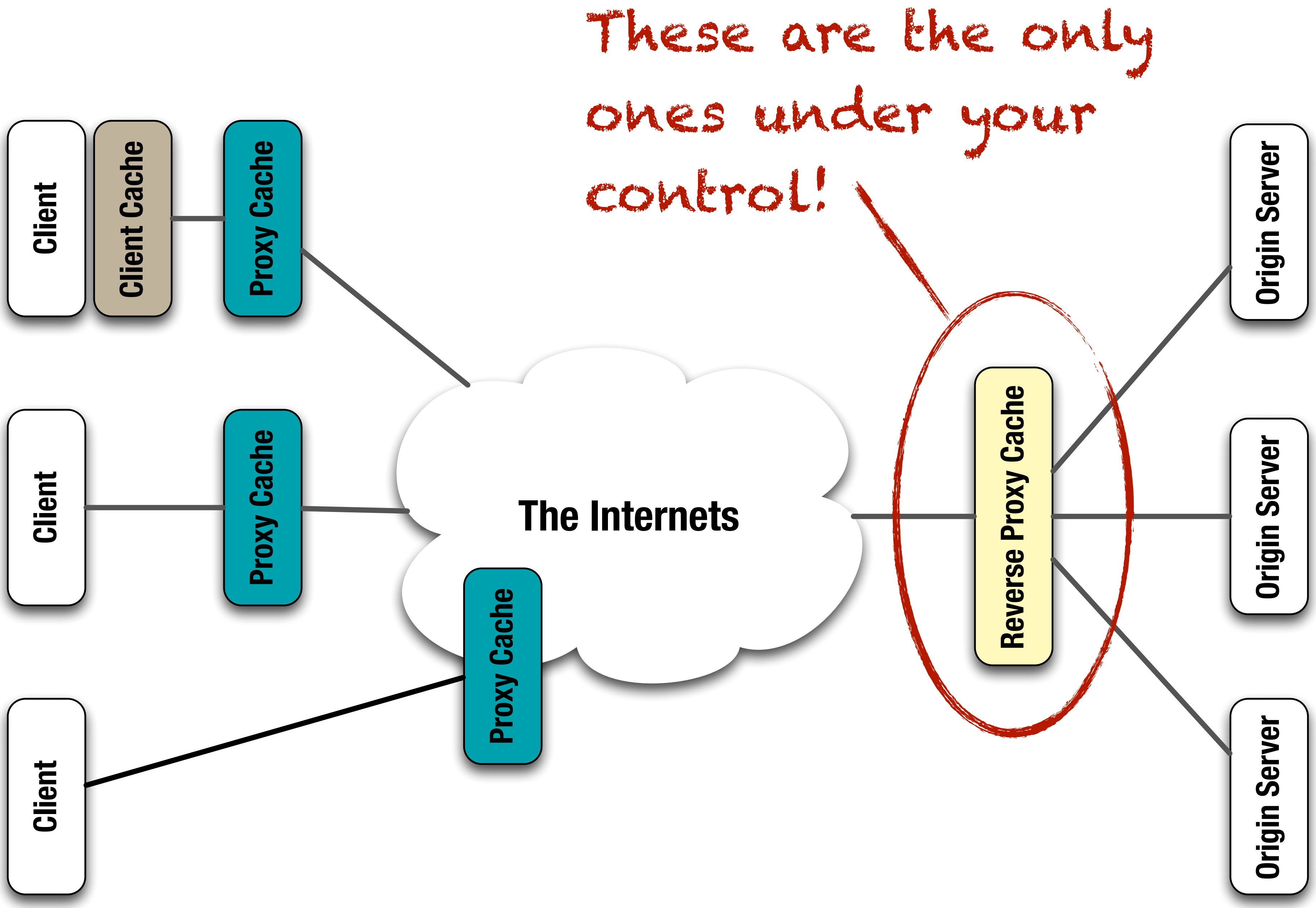
500 generic error message
501 server does not recognise method or lacks ability to fulfill
502 server received an invalid response from upstream server
503 server is currently unavailable
504 gateway did not receive response from upstream server
505 server does not support the HTTP protocol version
506 content negotiation for the request results in a circular reference
507 server is unable to store the representation
508 server detected an infinite loop while processing the request
509 bandwidth limit exceeded
510 further extensions to the request are required
511 client needs to authenticate to gain network access
598 network read timeout behind the proxy
599 network connect timeout behind the proxy

Using the right error category is key to finding the appropriate recovery strategy.

- Even if you’re not always sure about the subtleties of using one code over another, at least make sure you get the error category right:
  - 2xx codes indicate successful completion
  - 3xx codes are redirections
  - 4xx codes indicate error caused by faulty behavior on the client side – these are usually recoverable (just check the request and try again)
  - 5xx codes indicate server-side errors which may or may not be recoverable

Don’t ignore caching.

Fact:
There will be caches involved, no matter what.


These are the only ones under your control!
Client
Client Cache
Proxy Cache
Client
Proxy Cache
Client
Proxy Cache
The Internets
Reverse Proxy Cache
Origin Server
Origin Server
Origin Server

You can just ignore them, of course.

- If you don't include any caching headers in your responses, well-behaved caches will just do nothing.
- If you want to really make sure that no cache interferes with communication in any way, use

```
    Cache-Control: no-store
```

- But is this really what you want?

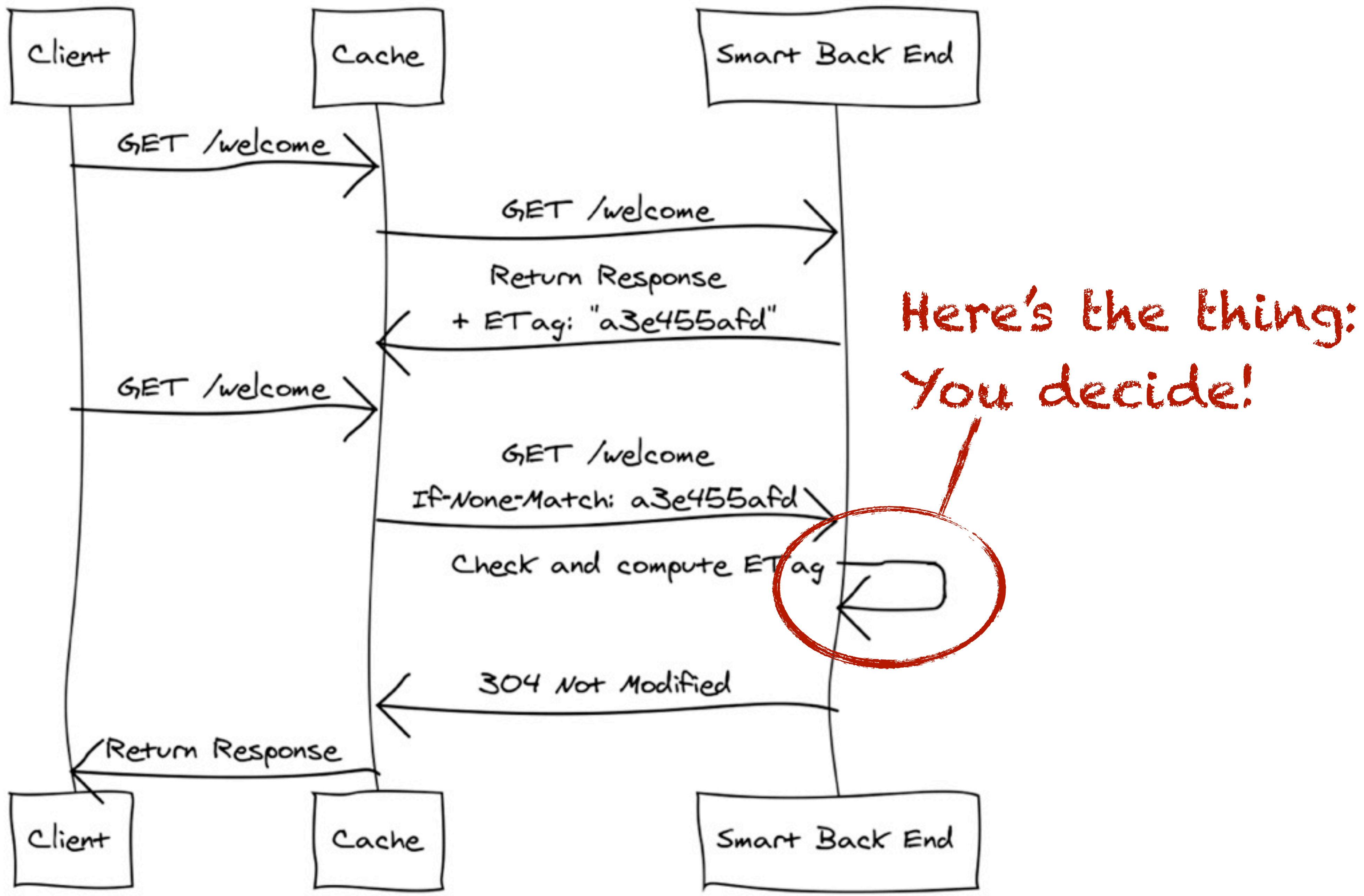
They're there to help!

(And they come for free.)

Help them so they can help you!

- The least you can do is include either an `Expires` header or a `Cache-Control: max-age=...` with a reasonable freshness period for data that changes rarely and/or at regular intervals.
- Better yet, use validators:
  - Include `Last-Modfied` in responses and honor `If-Modified-Since` in requests.
  - Include `ETag` in responses and honor `If-None-Match` in requests.

ETags are powerful beasts!



The cool thing about ETags

- ETags are opaque to proxies, so they can be just about anything:
  - hashes (not so cool if you need to create the representation to calculate the hash and then throw it away if it’s unchanged – no computation effort saved!)
  - timestamps
  - version numbers
  - or anything else that allows your server logic to decide if a representation can still be considered “fresh”, which means you can be fuzzy here!

There are some caveats to keep in mind, though.

- Be careful if your resources support multiple representations. You might want to include a `Vary: Accept` header.
- If a resource has both stable and highly volatile state, it can be useful to split it into two separate (sub-)resources (which should be hyperlinked, of course).
- Try to avoid excessive precision in query parameters as it can lead to cache misses. Consider if

  ```
    GET /weather?location=52.497N13.428E
  ```

  is really that much better than

  ```
    GET /weather?location=Berlin
  ```

Don’t see versioning
as a requisite.

As software engineers, we’ve internalized that versioning is essential to control change.

But a web API is fundamentally different from a piece of installed software.

- Web APIs are singletons – there’s only one instance at a time.
- Once a public-facing API is published and starts to gain traction, it becomes increasingly difficult to change.
- Clients are rarely under your control and it’s almost impossible for you to enforce version updates.

Often, when you think you’re changing a resource what you’re actually changing is just the representation.

- **In many real-world cases `/v1/customers` and `/v2/customers` still refer to the same "thing" (business concept, domain object, whatever). Why should it be identified by two distinct URLs?**
- **If the representation has changed, consider versioning the media type instead of introducing version information into the URL:**

```
Content-Type: application/vnd.myapi.v2
```

Better yet, try to get by without any versioning whatsoever.

- If you design your representations with extensibility in mind, you'll probably end up not needing versioning at all.
- Most JSON implementations' default mustIgnore behaviour make that easier to do in JSON than in XML.
- If backwards compatibility is not possible or adds too much of additional complexity, consider introducing an entirely new API (as Facebook did with the Graph API, for instance).

Don’t mix up searching and identifying.

Searching for resources and identifying resources are fundamentally different things.

- It's often good practice to provide more than one way to search for things, based on clients' intent:

```
/countries/germany/states/berlin/cities/berlin

/cities/berlin

/cities?name=berlin&state=berlin&country=germany
```

- Identity, however, should be unique:

```
/cities/3874
```

Try not to mix up these two concepts in your API.

- If possible, identify and refer to resources by their canonical URL.
- Use redirection:

```
GET /countries/germany/states/berlin/cities/berlin

HTTP/1.1 303
Location: http://api.example.org/cities/3874
```

Don’t obsess over URL naming – but don’t ignore it either.

Fact:
There is no such thing as a RESTful URL.
All URLs are created equal – they’re just identifiers after all.

Fact:
With proper use of hypermedia controls, URLs are irrelevant from a technical standpoint.

but

Which of the two logs will help you best with tracking down the problem if things go wrong?

```
[16/Oct/2013:13:55:36] "GET /customers” 200
[16/Oct/2013:13:56:01] "GET /customer/42” 200
[16/Oct/2013:13:56:47] "PUT /customer/42” 200
[16/Oct/2013:13:56:58] "POST /customer/42/orders” 200
[16/Oct/2013:14:11:13] "POST /orders/4711/items” 200
```

or

```
[16/Oct/2013:13:55:36] "GET /xz66fgt5” 200
[16/Oct/2013:13:56:01] "GET /ahgt67ft/42” 200
[16/Oct/2013:13:56:47] "PUT /ahgt67ft/42” 200
[16/Oct/2013:13:56:58] "POST /ahgt67ft/42/jh77hg87” 200
[16/Oct/2013:14:11:13] "POST /bn87xcws/4711/lw33mn45” 200
```

Machines don’t care.

Humans do.

Don’t use extensions as the only means of content negotiation.

Name extensions are a convenient way to select media types for representations.

- They're especially useful for testing in a browser (which doesn't provide an easy way to do content negotiation).
- But they introduce multiple URL aliases for the same resource that can lead to confusion and ambiguities when used to link to the resource in hypermedia representations.
- Prefer to use a canonical URL with "proper" content negotiation as the primary reference:

```
/customer/42          (Canonical)
/customer/42.xml      (Alias)
/customer/42.json     (Alias)
```

Recap

Don’t think in terms of “endpoints”.

Don’t just expose your domain model.

Don’t overuse GET and POST.

Don’t limit your choice of error codes to 200 and 500.

Don’t ignore caching.

Don’t see versioning as a requisite.

Don’t mix up searching and identifying.

Don’t obsess over URL naming – but don’t ignore it either.

Don’t use extensions as the only means of content negotiation.

That’s all I have.
Feel free to ask me anything!

@owolf

innoQ