UNSW SYDNEY | Australia's Global University

COMP9321

Data Services Engineering

Term 1, 2021

Week 5 REST Services Part 2

Architectural Constraints of REST

1. Client-Server
2. Uniform Interface
3. Statelessness
4. Caching
5. Layered System
6. Code on demand (optional)

If your design satisfies the first five, you can say your API is ‘RESTful’

On Uniform Interface – Linked Resources

Representations are hypermedia: resource (data itself) + links to other resources

e.g., Google Search representation:

Jellyfish
Jellyfish are most recognised because of their jelly like appearance and this is where they get their name. They are also recognised for their bell-like ...
www.reefed.edu.au › ... › Corals and Jellyfish - Cached - Similar

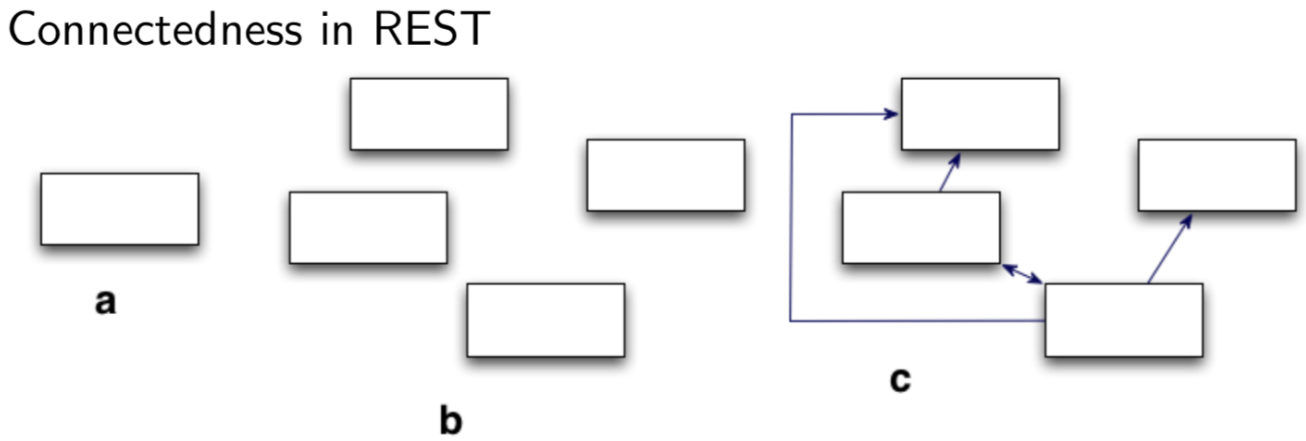
Jellyfish - Wikipedia, the free encyclopedia
Jellyfish (also known as jellies or sea jellies) are free-swimming members of the phylum Cnidaria. **Jellyfish** have several different morphologies that ...
Terminology - Anatomy - Jellyfish blooms - Life cycle
en.wikipedia.org/wiki/**Jellyfish** - Cached - Similar

Searches related to **jellyfish**

jellyfish **facts**	**types of** jellyfish	jellyfish **pictures**	**blue bottle** jellyfish
jellyfish **stings**	jellyfish **photos**	jellyfish **reproduction**	jellyfish **life cycle**

Goooooooooo gle ▶
1 2 3 4 5 6 7 8 9 10 **Next**

On Uniform Interface – Linked Resources



All three services expose the same functionality, but their usability increases towards the right

- Service A is a typical remote-function style service, exposing everything through a single URI. Neither addressable, nor connected
- Service B is addressable but not connected; there are no indication of the relationships between resources. A hybrid style ...
- Service C is addressable and well-connected; resources are lined to each other in ways that make sense. A fully RESTful service

On Uniform Interface – Linked Resources

Example: Pagination

```
HTTP/1.1 200 OK
{
  "href" : "https://api.mycompany.com/v1/users?offset=50&amp;limit=50"
  "offset": 50,
  "limit": 50,
  “first”: {
      “href”: "https://api.mycompany.com/v1/users"
  },
   “prev”: {
      “href”: "https://api.mycompany.com/v1/users"
  },
  “next”: {
      “href”: "https://api.mycompany.com/v1/users?offset=100&amp;limit=50"
  },
  “last”: {
      “href”: "https://api.mycompany.com/v1/users?offset=50&amp;limit=50"
  },
  "items": [
    {
      … user 51 name/value pairs …
    },
    …,
    {
      … user 100 name/value pairs …
    }
  }
```


https://stormpath.com/blog/linking-and-resource-expansion-rest-api-tips

Statelessness

1. Client-Server
2. Uniform Interface
3. **Statelessness**
4. Caching
5. Layered System

REST API must be stateless

All calls from clients are independent

Stateless means every HTTP request happens in a complete isolation.

Stateless is good !! - scalable, easy to cache, addressable URI can be bookmarked (e.g., 10th page of search results)

HTTP is by nature stateless. We do something to break it in order to build applications

the most common way to break it is ’HTTP sessions’

the first time a user visits your site, he gets a unique string that identifies his session with the site

http://www.example.com/forum?PHPSESSIONID=27314962133
the string is a key into a data structure on the server which contains what the user has been up to. All interactions assume the key to be available to manipulate the data on the server

On Statelessness

What counts as ‘state’ exactly?

Think a Flickr.com-like web site ... you will add photos, rename photos, share them with friends, etc. – what would ‘being stateless’ mean here?

KEY notion: **separation of client application state and RESTful resource state.**

- consider the application state as data that could vary by client, and per request.
- consider the resource state as data that could is centrally managed by the server. It is the same for every client.
- resource states live on the server
- individual client application states should be kept off the server

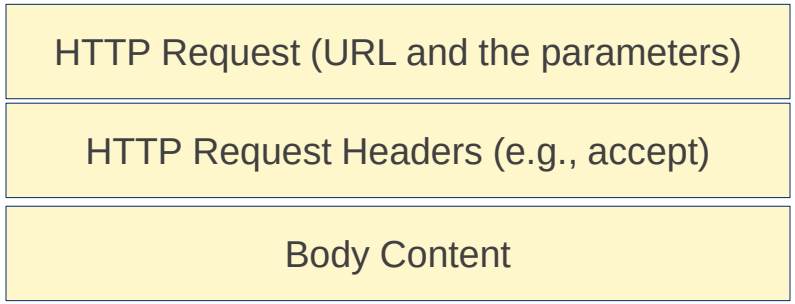
Consider a scenario: a little photo edit app + Flickr and other APIs ...

On Statelessness

Statelessness in REST applies to the client application state (from the server's view point)

What does this mean to the client application?

- every REST request should be totally self-descriptive
- client transmit the state to the server for every request that needs it.



a RESTful service requires that the application state to stay on the client side. Server does not keep the application state on behalf of a client

What about OAuth? Is it conflicting with the Statelessness of REST?

Caching

1. Client-Server
2. Uniform Interface
3. Statelessness
4. **Caching**
5. Layered System

Responses must be marked ‘cachable’ or ‘non-cachable’

Well-managed caching partially or completely eliminates some client–server interactions, improving scalability and performance.

Being Stateless: every action happens in isolation:

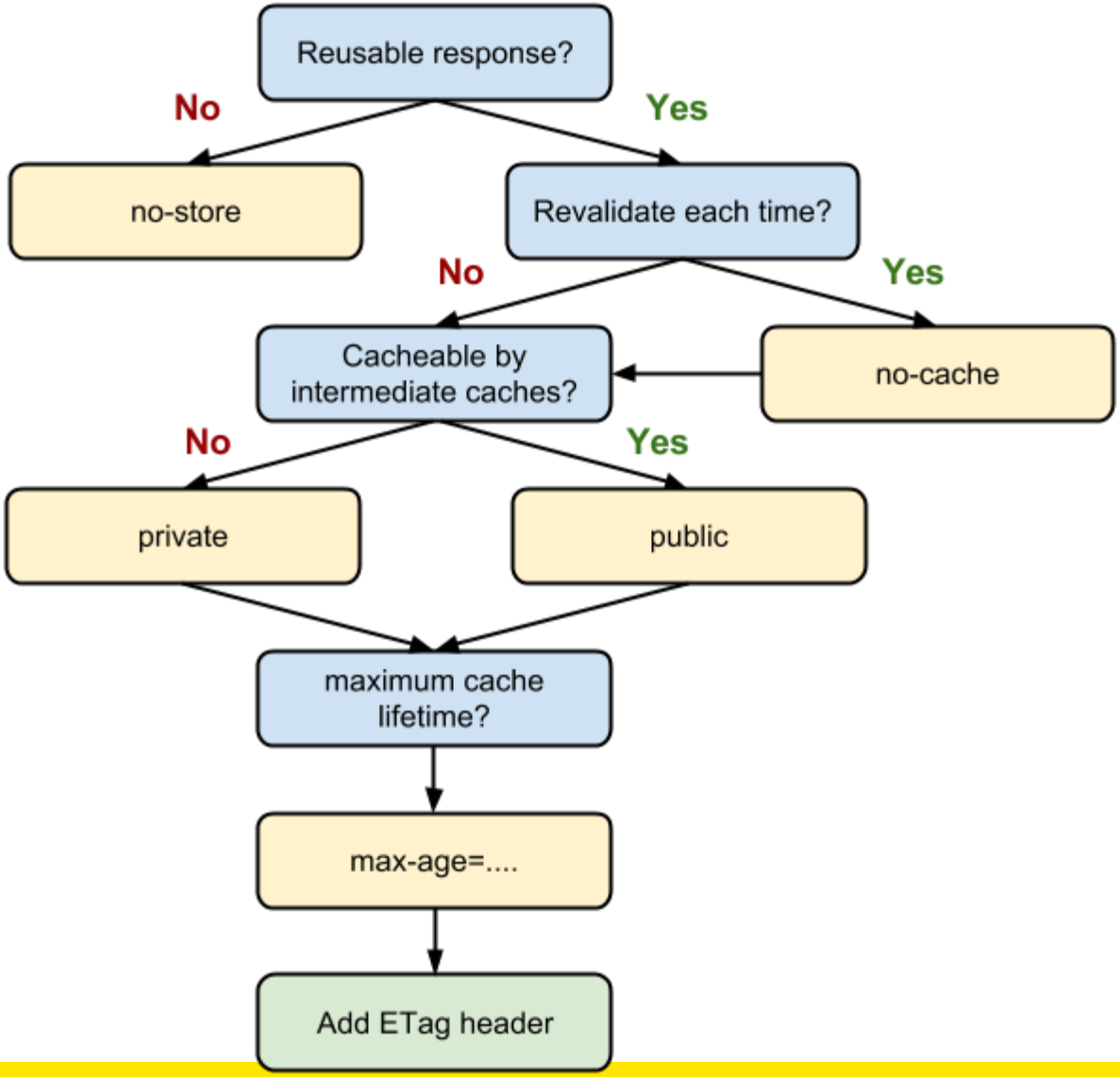
- Keeps the interaction protocol simpler
- But the interactions may become ‘chattier’

To scale, RESTful API must be work-shy (only generate data traffic when needed, other times use cache)

This requires ’server-client’ collaboration:

- Client provide guard clauses in requests so that servers can determine easily if there’s any work to be done
- If-Modified-Since, Last Modified, If-None-Match/ETag

Caching (Defining optimal Cache-Control policy)



https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching?hl=en

Caching

Request

```
GET /transactions/debit/1234 HTTP 1.1
Host: bank.example.org
Accept: application/xml
If-None-Match: aabd653b-65d0-74da-bc63-4bca-ba3ef3f50432
```

Response

```
200 OK
Content-Type: application/xml
Content-Length: ...
Last-Modified: 2010-21-04T15:10:32Z
Etag: abbb4828-93ba-567b-6a33-33d374bcad39
<t:debit xmlns:t="http://bank.example.com">
<t:sourceAccount>12345678</t:sourceAccount>
<t:destAccount>987654321</t:destAccount>
<t:amount>299.00</t:amount>
<t:currency>GBP</t:currency>
</t:debit>
```

Caching

Request

```
GET /transactions/debit/1234 HTTP 1.1
Host: bank.example.org
Accept: application/xml
If-Modified-Since: 2010-21-04T15:00:34Z
```

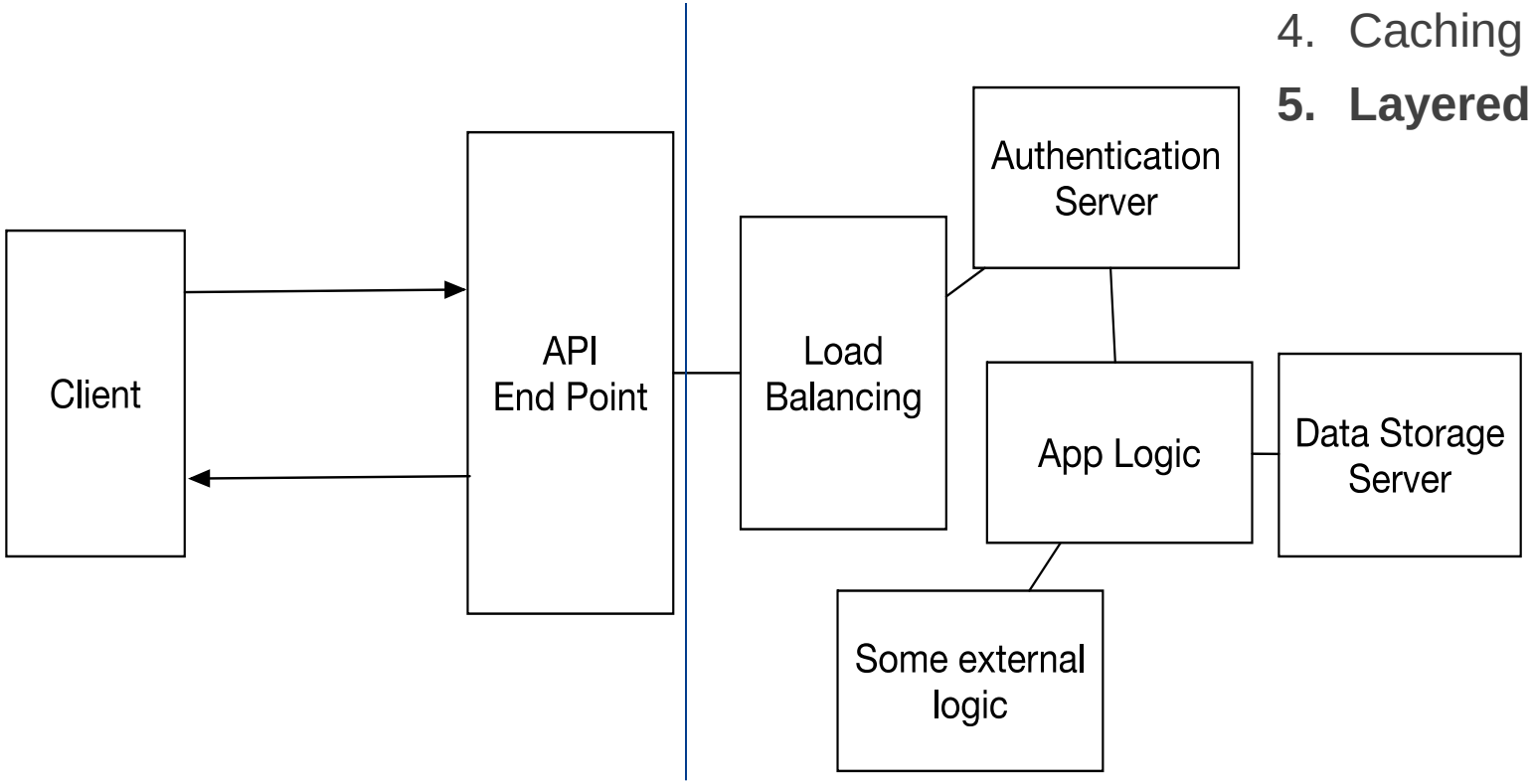
Response

```
200 OK
Content-Type: application/xml
Content-Length: ...
Last-Modified: 2010-21-04T15:00:34Z
```

Server needs cache management policy and implementation …

Layered System

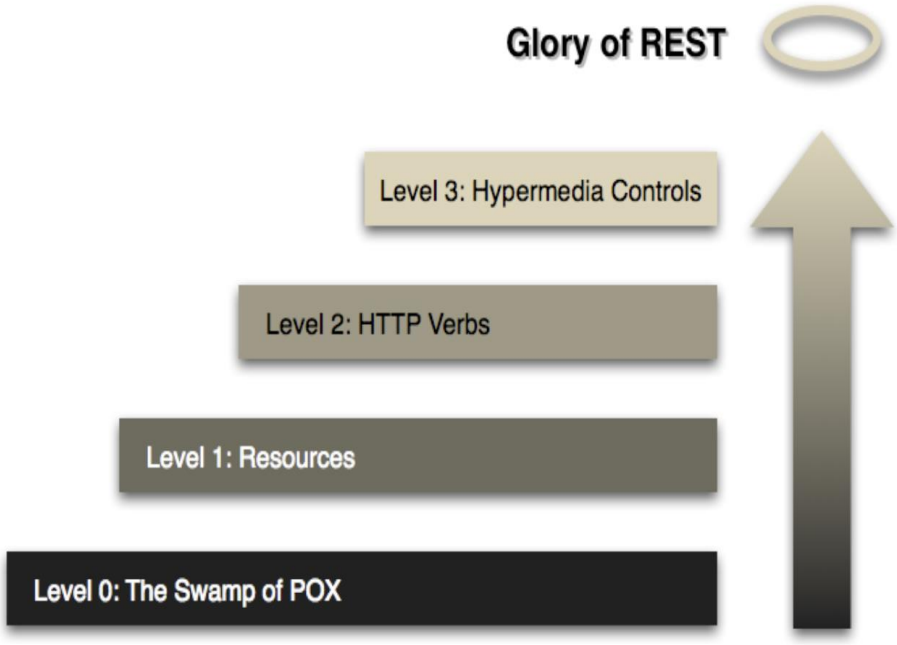
1. Client-Server
2. Uniform Interface
3. Statelessness
4. Caching
5. **Layered System**



A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.

Again, de-coupling allows the components in the architecture to evolve independently

The Richardson Maturity Model



Leonard Richardson: can we measure to what level your service is RESTful?

Level 0: One URI (single endpoint) exposed, requests contain operation details

Level 1: Expose resource URIs - individual URIs for each resource. Requests could still contain some operation details

Level 2: HTTP Methods - use the standard HTTP methods, status codes with the resource URIs,

Level 3: HATEOAS - self-documenting responses, responses include links that the client can use

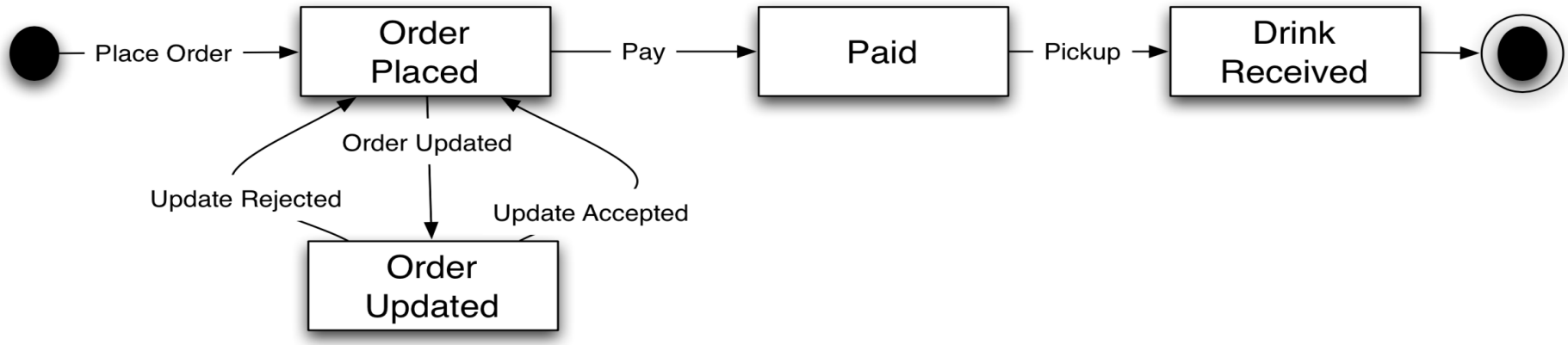
A good read: https://martinfowler.com/articles/richardsonMaturityModel.html

Interacting with RESTful API (workflow)

What would it be like to have a stateless conversation with API vs. stateful conversation with API? (e.g., POST -> return ID, forget vs. POST-> return no ID, plant Cookies)

Take the Coffee Order Process from Jim Webber as example ...

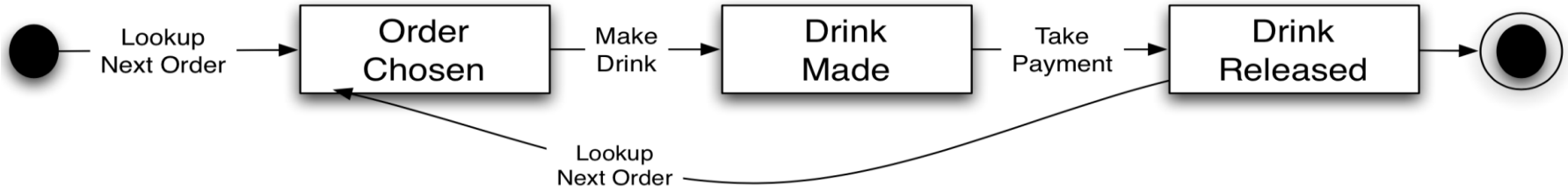
The customer workflow:



customers advance towards the goal of drinking some coffee by interacting with the Starbucks service, the customer orders, pays, and waits for the drink, between ‘order’ and ‘pay’, the customer can update (asking for skimmed milk)

Interacting with RESTful API (workflow)

The barista workflow:



the barista loops around looking for the next order to be made, preparing the drink, and taking the payment,

The outputs of the workflow are available to the customer when the barista finishes the order and releases the drink

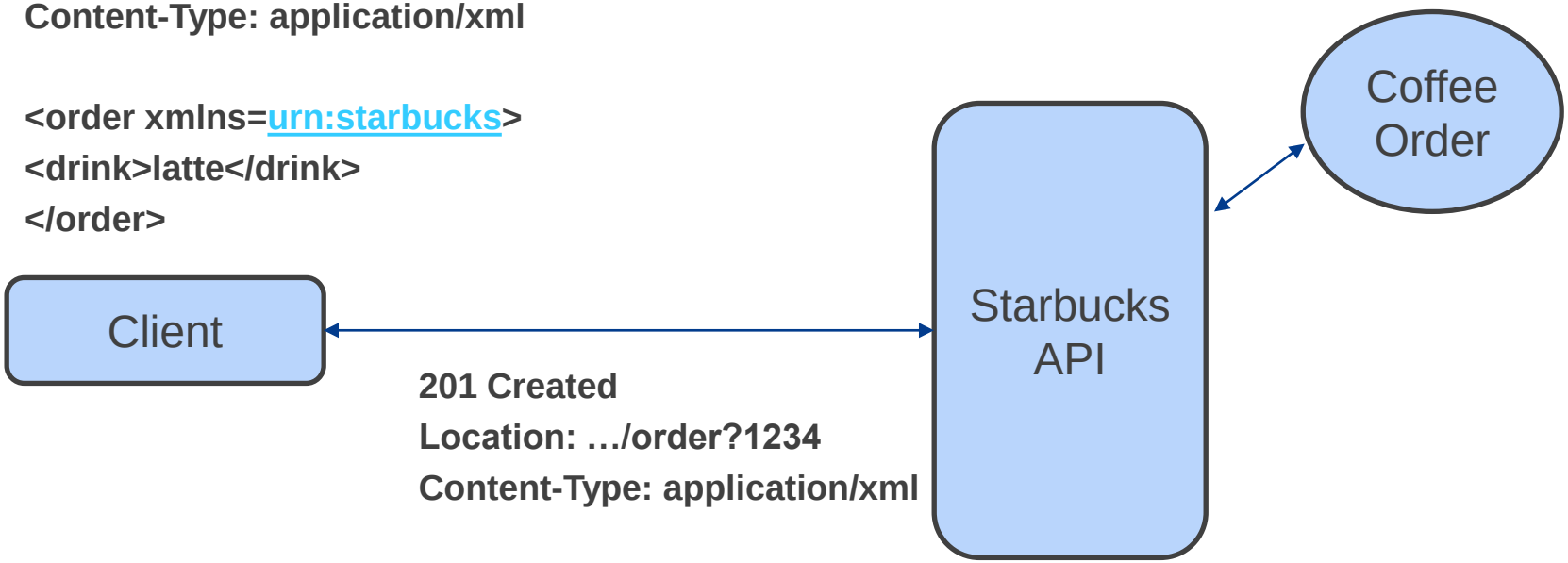
Points to Remember: We will see how each transition in two state machines is implemented as an interaction with a Web resource. Each transition is the combination of a HTTP verb on a resource via its URI causing state changes.

Customer's View Point

Place an order: POST-ing on http://api.starbucks.com/orders

POST /orders HTTP/1.1
Host: xxx
Content-Type: application/xml

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



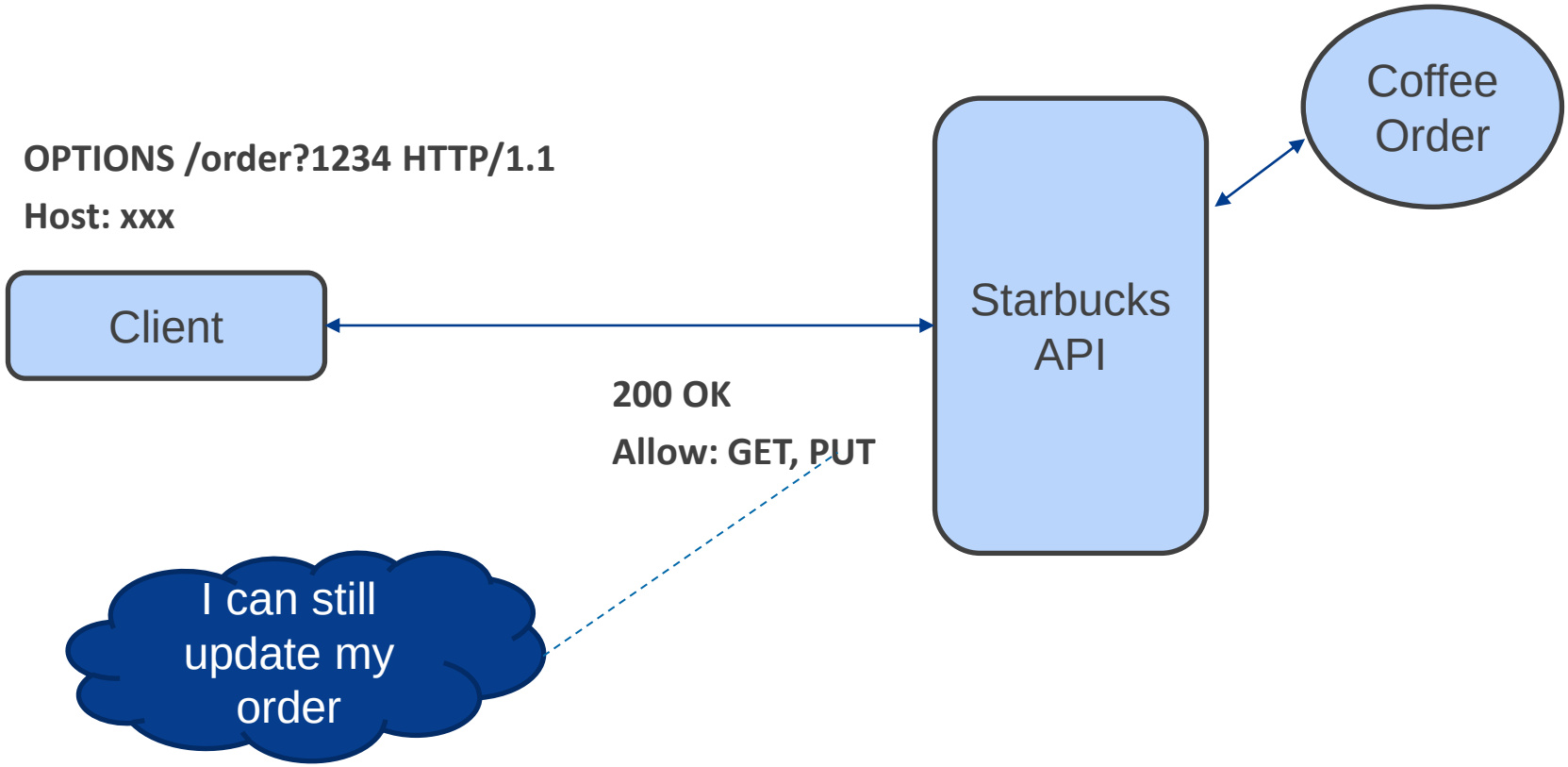
201 Created
Location: …/order?1234
Content-Type: application/xml

<order xmlns=urn:starbucks>
<drink>latte</drink>
<link rel="payment" href="…/payment/order?1234>
</order>

Oops ... A mistake!

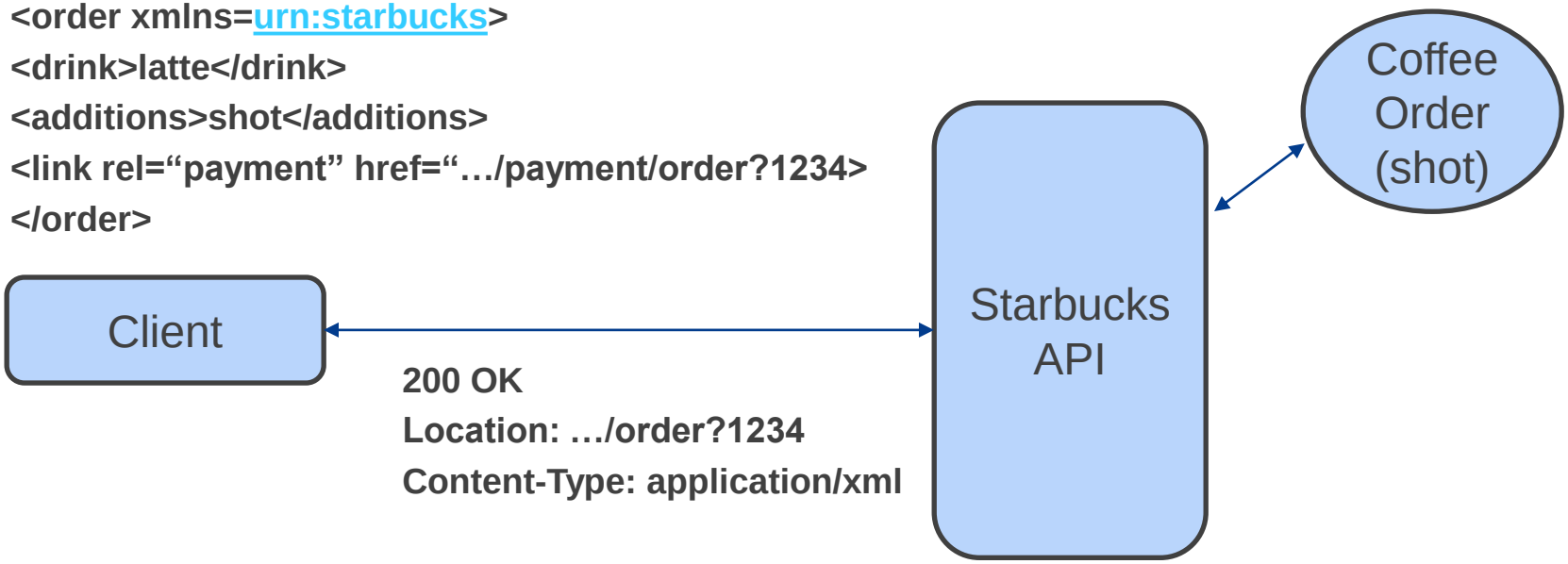
I like my coffee to be strong

Need another shot of espresso, what are my OPTIONS?



Update the order

```
PUT /order?1234  HTTP/1.1
Host: xxx
Content-Type: application/xml

<order xmlns=urn:starbucks>
<drink>latte</drink>
<additions>shot</additions>
<link rel="payment" href="…/payment/order?1234>
</order>
```



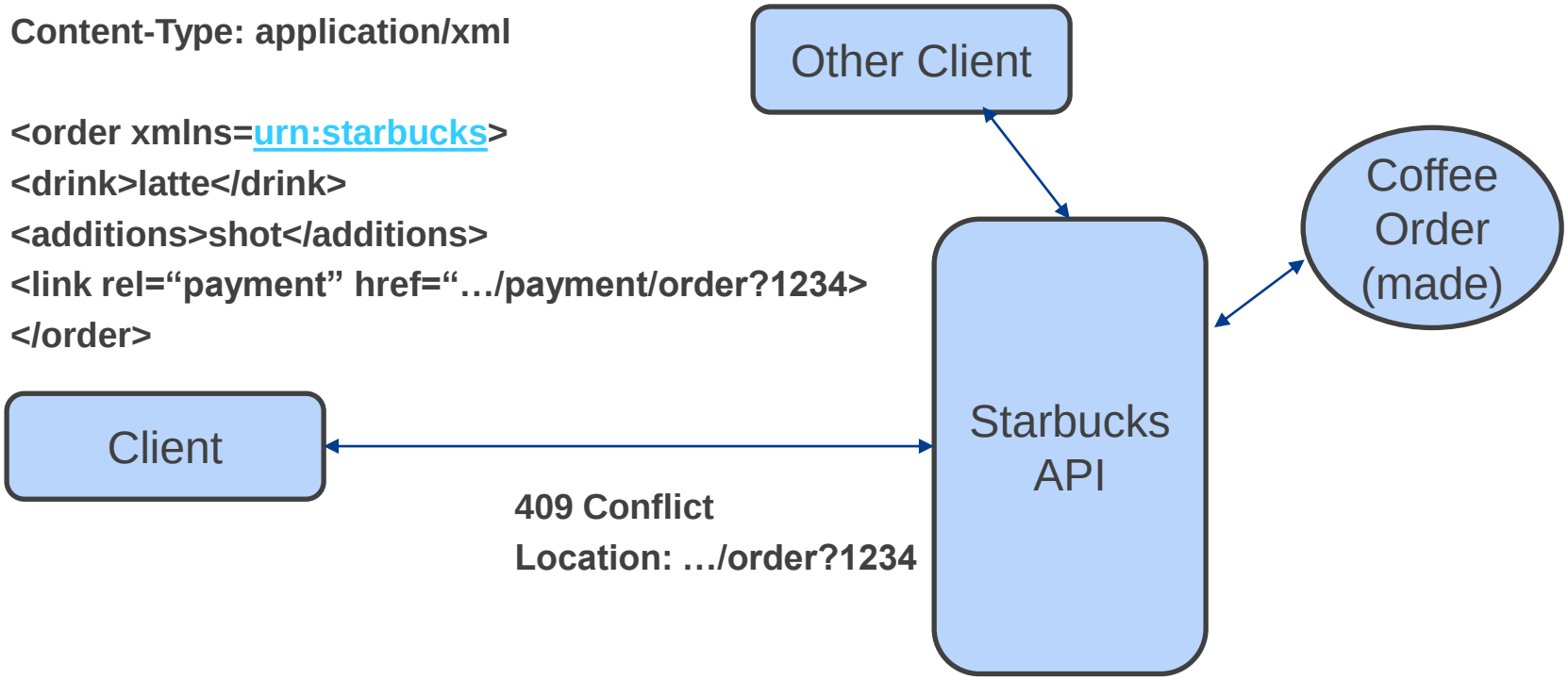

```
200 OK
Location: …/order?1234
Content-Type: application/xml

<order xmlns=urn:starbucks>
<drink>latte</drink>
<additions>shot</additions>
<link rel="payment" href="…/payment/order?1234>
</order>
```

Possible conflict with another workflow

The resource state can change without you … (before your PUT-ing getting to the server)

PUT /order?1234 HTTP/1.1
Host: xxx
Content-Type: application/xml

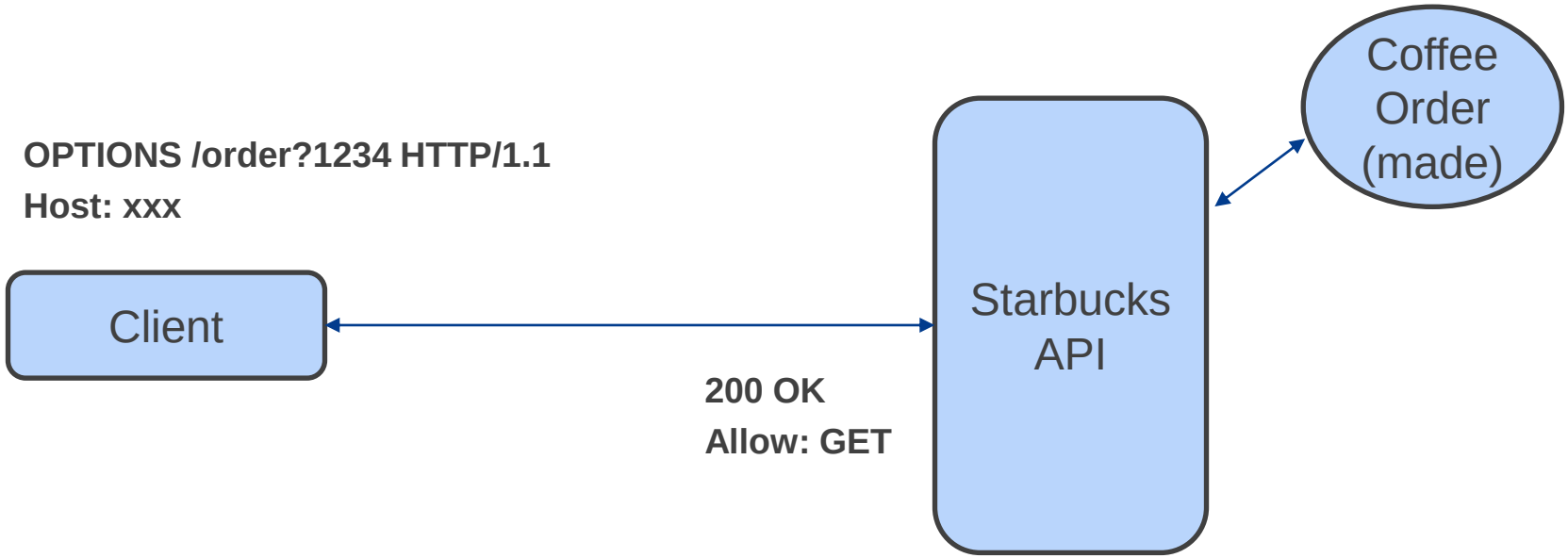
<order xmlns=urn:starbucks>
<drink>latte</drink>
<additions>shot</additions>
<link rel="payment" href="…/payment/order?1234>
</order>



Possible conflict with another workflow

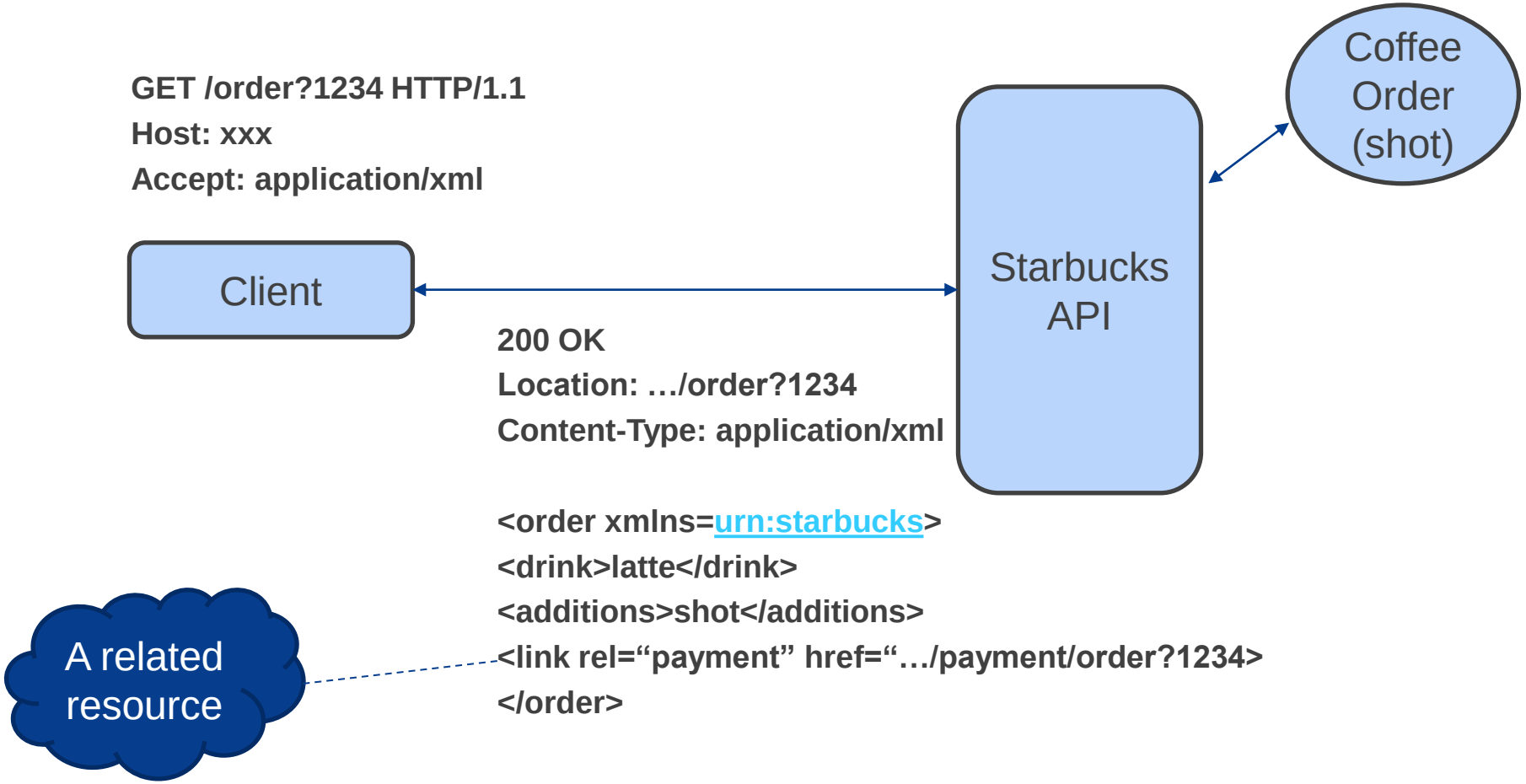
What are my OPTIONS now?

How do I recover?



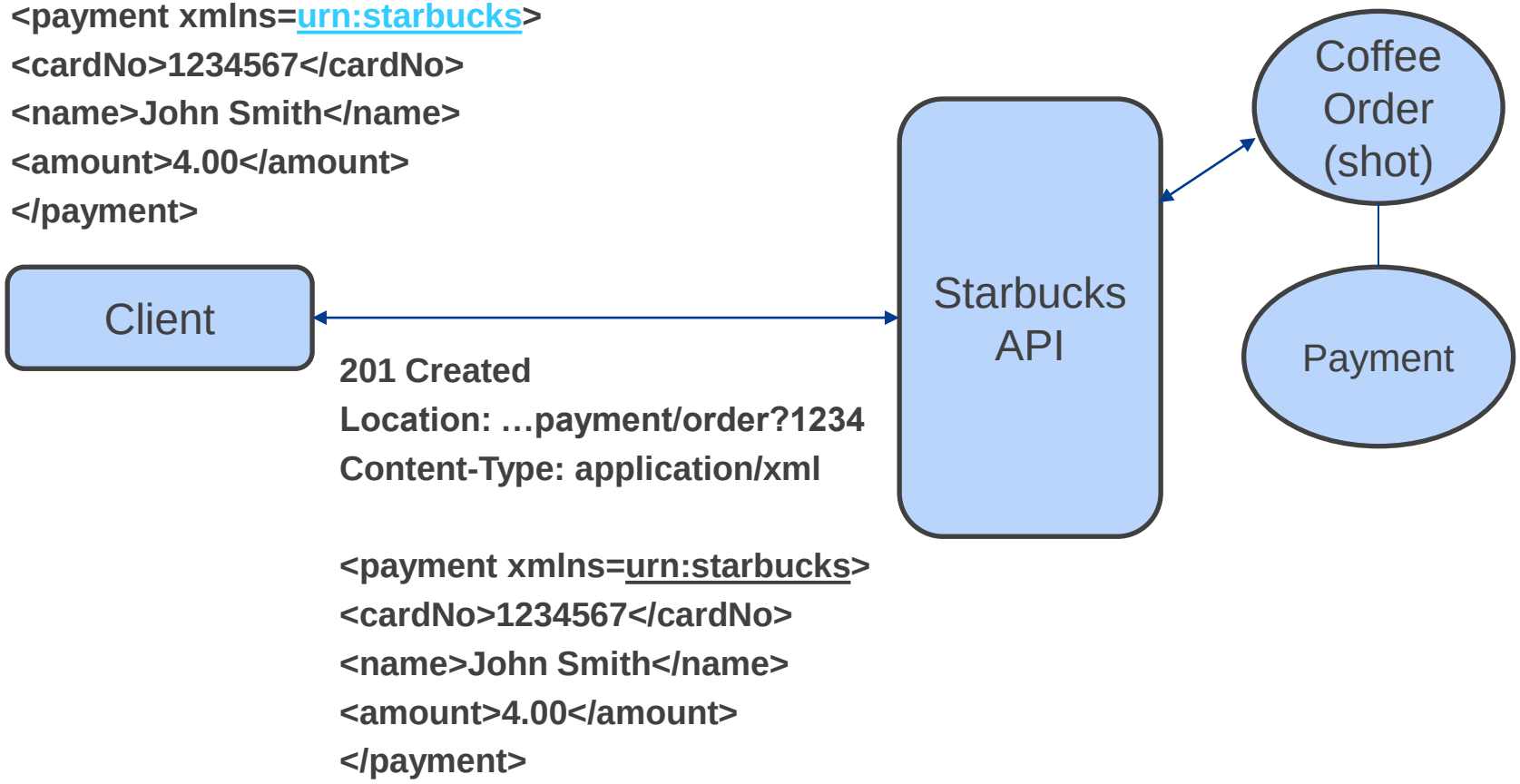
OK, update successful, what now?

Idea floated here is … FOLLOW THE LINK.

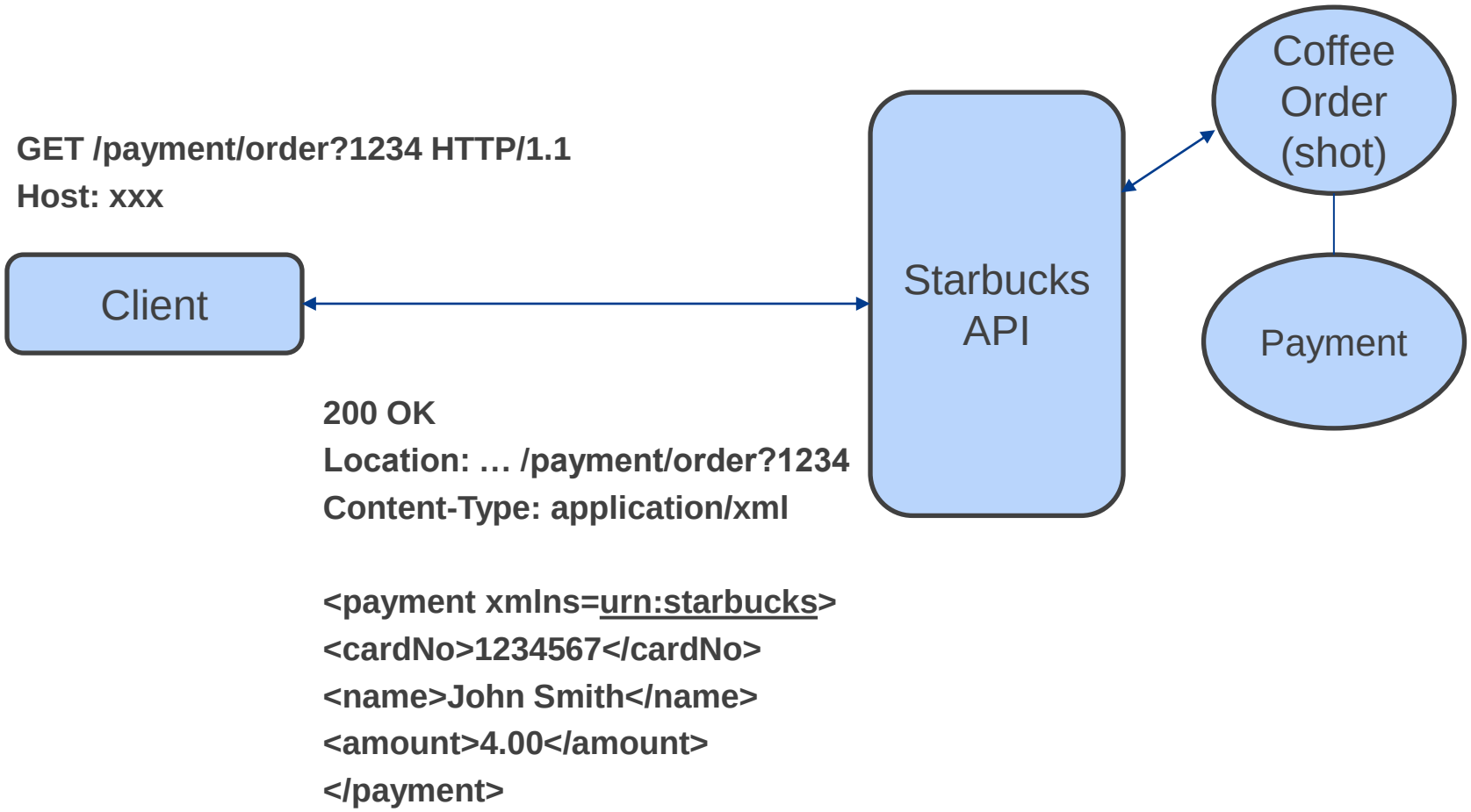


Pay for the order (PUT = idempotent)

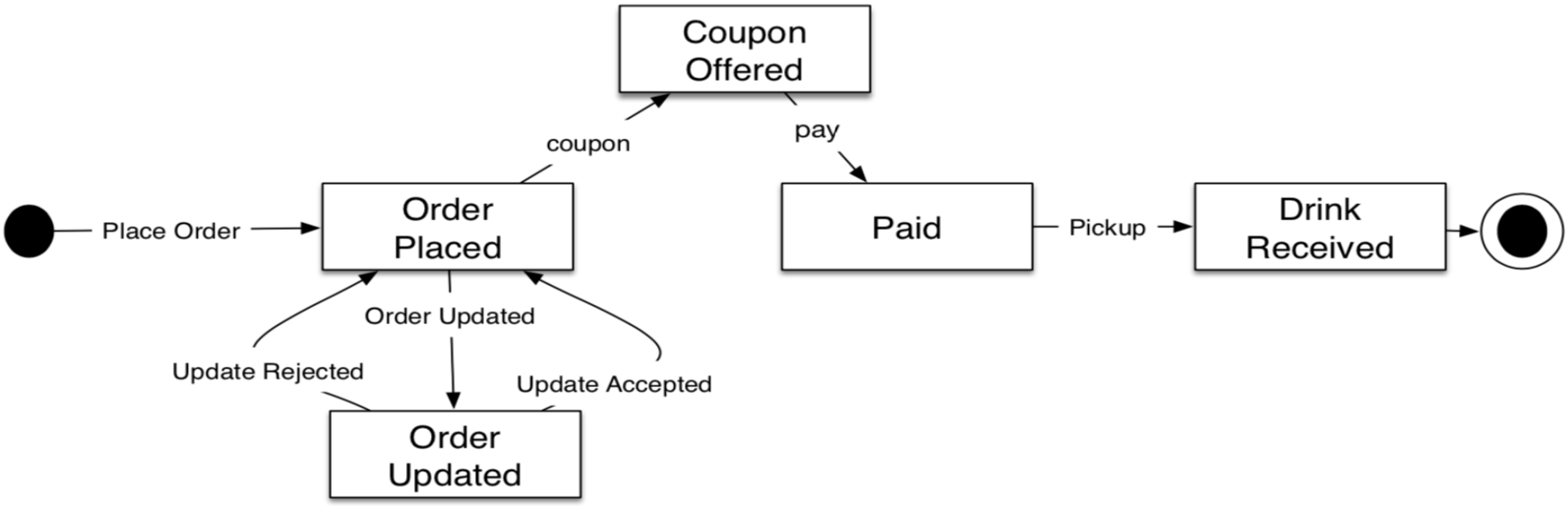
```
PUT /payment/order?1234  HTTP/1.1
Host: xxx
Content-Type: application/xml

<payment xmlns=urn:starbucks>
<cardNo>1234567</cardNo>
<name>John Smith</name>
<amount>4.00</amount>
</payment>
```



Check that you have paid?



Changing the API conversation?



If the self-describing nature of the workflow (i.e., links) is well-respected, the client should not be surprised by the changes !

Tools of the Trade

- Python
- Pandas
- Flask RESTX
- Swagger

Flask and Flask RESTX

- **Flask** is a Python Micro Web Framework. It allows you to build light-weight Web Apps, but it has good capabilities because it support extensions (there are many)

- **Flask RESTX** is an extension for Flask that adds support for quickly building REST APIs. Flask-RESTX encourages best practices with minimal setup. It provides a coherent collection of decorators and tools to describe your API and expose its documentation properly (using Swagger). Flask RESTX is a forked project of Flask RESTplus. So far they are 100% compatible.

Swagger

- When using Flask RESTPlus, A Swagger API documentation is automatically generated and available on your API root but you need to provide some details with the Api.doc() decorator

- Swagger (now the "Open API Initiative) is a framework for describing your API using a common language that everyone can understand. Think of it as a blueprint for a house. You can use whatever building materials you like, but you can't step outside the parameters of the blueprint.

- Let’s have a look (http://editor.swagger.io)

Questions