COMP 9322
Software Service Design and Engineering

Lecture 2 (Part2)– Web Services/RESTful APIs

Notice

- Some of the slides are taken from COMP9321 lectures presented by Mortada Al-Banna in S2 2018 and hence if you've attended the course last year please bare with me and I promise things will get interesting…

What is World Wide Web

‘an information space where documents and other web resources are identified by Uniform Resource Locators (URLs), interlinked by hypertext links, and can be accessed via the Internet’

- For human consumption
- Formatted using Hypertext Markup Language
- Uniform API and technologies
- Rendered by client’s Web browser

The Concept of Programmable Web

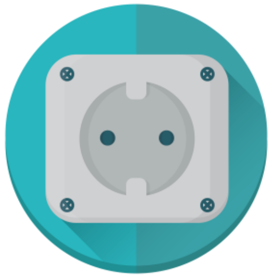
- The Programmable Web use the same technologies and communication protocols of the WWW
- Difference:
  - The data is not delivered necessarily for human consumption
  - A client can be implemented using any programming language
- Technologies
  - Services and APIs
  - Transport protocol: Hyper Text Transfer Protocol (HTTP)
  - Clients: Browser, Java, Web API, …
  - Data serialization languages

Web Services

- 'logical units with clearly defined interfaces(API):'
  - What functionality they perform
  - Which data formats they accept and produce
- They are application independent
- Services can be used by other services and applications
- Web services are not prepared to human consumption (in contrast to websites).
  - Web services require an architectural style to provide clear and unambiguous interaction (clearly defined interfaces).

Web API

- Application Programming Interfaces
  - A good analogy is the electricity wall socket
- Endpoints addressable over the Web are called Web APIs.
- How the service is exposed:
  - Protocol semantics
  - Application semantics
- We frequently use Web API instead of Web services but they are not the same
- We will be focusing on the RESTfull Web API



- Service: Electricity
- Conforms to specs: 220V, 60Hz …
- Fitting patterns are defined
- Through the standard interface all connecting equipment (consumers) work
- A layer of abstraction

Market Impact

- Making functionality available over the web changed the way software functionality delivered.
- If you needed a CRM functionality in 1990s you had to invest in hardware, software, the CRM experts, training …
- Today's CRM providers like Salesforce use cloud to deliver the functionality.
  - Multi-tennacy – sharing common infrastructure among customers.
  - Using web browsers was the norm to access this functionality
  - Today customers are granted API level access
    - Non salesforce applications can easily use the services.
- Thousands of companies are changing their strategies toward delivering functionality through Web APIs:
  - https://www.programmableweb.com/apis/directory is a good source

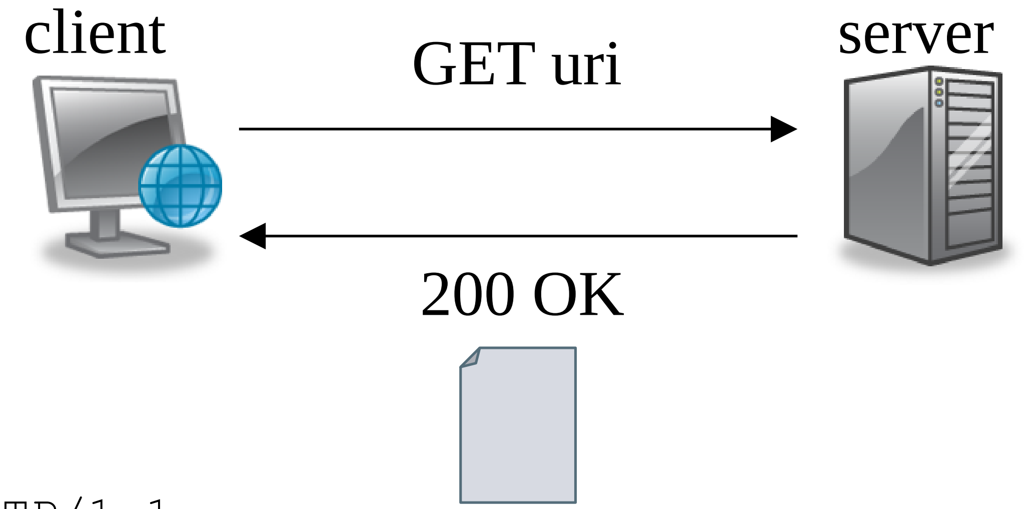
The Protocol Semantics: Hypertext Transfer Protocol (HTTP)

- ‘An application level protocol for distributed, collaborative, hypermedia information systems’
  - It takes place over TCP/IP connections.
  - De-facto application protocol in the World Wide Web.
  - Used as a transport protocol for other application protocols, such as SOAP, XML-RPC
- Allows bidirectional transfer of ‘network data objects Identified by a URI‘ representations between client and server

HTTP Messages

```
<message> ::=   ( <request> | <response> )
                <header>*
                CRLF
                <body>

<request> ::=   <method> SP <request-uri> SP
                <http-version> CRLF

<response>::=   <http-version> SP <status-code>
                SP <reason-phrase> CRLF

<header>  ::=   <field-name> : <field-value> CRLF

<body>    ::=   <sequence of bytes>
```

Typical Request Response



```
GET / HTTP/1.1
Host: www.unsw.edu.au

HTTP/1.1 200 OK
Content-Type: text/html

<html>
<head><title>UNSW Homepage</title></head>
<body><h1>Welcome to UNSW</h1> … </body>
</html>
```

HTTP Methods

- GET – Returns the resource representation
- HEAD – Like GET, but ask that only a header be returned
- POST – Create subordinate resources (no URL known beforehand) Appends information to the current resource state
- PUT – Changes the state of the resource
  - Creates a new resource when the URL is known
- DELETE – Deletes the specified resource
- (TRACE, OPTIONS, CONNECT, PATCH)

Further reading:
http://www.w3.org/Protocols/rfc2616/rfc2616.html

Representational State Transfer (REST)

- A way of providing interoperability between computer systems on the Internet.
  - REST-compliant Web services allow requesting systems to access and manipulate textual representations of Web resources using a uniform and predefined set of stateless operations.
- An architectural style of building networked systems
  - a \design guideline" for building a system (or a service in our context) on the Web
  - defines a set of architectural constraints in a protocol
- REST is built on standards:
  - HTTP, URL, XML/HTML/JPEG/ … (resource representations)
  - text/xml, text/html, image/gif, image/jpeg, … (MIME Types)
- REST itself is not an official standard specification

Resources

- Everything starts and ends with resources.
- What is a Resource?
  - *'The key abstraction of information in REST is a resource. Any information that can be named can be a resource:*

    a document or image, a temporal service (e.g. today's weather in Los Angeles), a collection of other resources, a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the denition of a resource.

    A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.'

http://www.ics.uci.edu/elding/pubs/dissertation/top.htm

What is Resource

- A thing that users want to create a link to, retrieve, annotate, or perform other operations on.
- A resource:
  - is **unique** (i.e., can be identied uniquely)
  - has at least one **representation**,
  - has one or more **attributes** beyond ID
  - has a potential **schema**, or definition
  - can provide **context**
  - is reachable within the **addressable** universe
- collections, relationships (structural, semantic)

Representational State Transfer?

- Web is comprised of resources
- UNSW can define COMP3392 as a resource
  - Students can access this resource through a URL:
  - http://www.unsw.edu.au/course/COMP3392
- Representation is returned COMP3392.html –
  - The representation place client application in a **state**
  - Client can access another resource in COMP3392.html
  - The new representation places client in another **state**
- The client application **transfer states** with each resource **representation.**

Resource Oriented Architectures

- ROA:
  - Architecture for creating Web APIs that conforms to the REST design principles
  - Base technologies: URLs, HTTP and Hypermedia
- Web Services with a ROA architecture are called RESTful Web Services (Restfull Web APIs)
- HTTP requests are used to manipulate the state of a resource

URI: Identifies the resource to manipulate

http://www.unsw.edu.au/$\overbrace{\text{course/COMP3392}}$

HTTP method: The action to be performed to manipulate the resource

ROA Properties

- Addressability
- Uniform interface
- Statelessness
- Connectedness

Addressability

- An application is addressable if it exposes the interesting aspects of its data set as resources
  - Files systems, cells in spreadsheets, Reference to a book at amazon.com.au …
  - Can you think of examples of not addressable things?
- Each resource is exposed using its URI:
- The URI can be copied, pasted and distributed
- Example:
  - http://unsw.edu.au/course/COMP3392

  refers to the information of the course
  - I can send this URI by email, and the receiver can, access this information by copying this URI into, his/her, browser

Uniform Interface

- APIs uses the same methods with the same semantics
  - Without a uniform interface, clients have to learn how each API is expected to get and send information
- ROA uses uniform interface provided by HTTP to act over the resource provided in the URI
- REST Uniform Interface Principle has 4 HTTP main operations on resources
  - GET: Retrieve a representation of a resource.
  - PUT: Create a new resource (new URI) or update a resource (existing URI)
  - DELETE: Clear a resource, after the URI is no longer valid
  - POST*: Modify the state of a resource. POST is a read-write operation and may change the state of the resource and provoke side effects on the server. Web browsers warn you when refreshing a page generated with POST.

Uniform Interface Matters

- Uniformity
  - Every service use HTTP's interface the same way.
  - It means, for example, GET does mean 'read-only' across the Web no matter which resource you are using it on.
  - It means, we do not use methods in place of GET like doSearch or getPage or nextNumber.
- But, it is not just using GET in your service, it is about using it the way it was meant to be used.

Uniform	Non Uniform
Get(URI)	getCustomer()
Put(URI, Resource)	updateCustomer(Customer)
Delete(URI)	Delete(CustomerId)

Uniform Interface Two benefits

- being Safe
  - Read-only operations: The operations on a resource do not change any server state.
  - The client can call the operations 10 times, it has no effect on the server state.
  - (like multiplying a number by 1, e.g., 4x1, 4x1x1, 4x1x1x1, ...)
- being Idempotent
  - Operations that have the same " effect" whether you apply it once or more than once.
  - An effect here may well be a change of server state. An operation on a resource is idempotent if making one request is the same as making a series of identical requests.
  - (like multiplying a number by 0, e.g., 4x0, 4x0x0, 4x0x0x0, ...)
- Together they let a client make reliable HTTP requests over an unreliable network.
  - Your GET request gets no response? make another one. It's safe.
  - Your PUT request gets no response, make another one. Even if your earlier one got through, your second PUT will have no side-eect.

Safety and Idempotence

- GET, HEAD and OPTION: safe
- PUT: - idempotent
  - If you create a resource with PUT, and then resend the PUT request, the resource is still there and it has the same properties you gave it when you created it.
  - If you update a resource with PUT, you can resend the PUT request and the resource state won't change again
- DELETE: - idempotent
  - If you delete a resource with DELETE, it's gone. You send DELETE again, it is still gone !

States in Statelessness

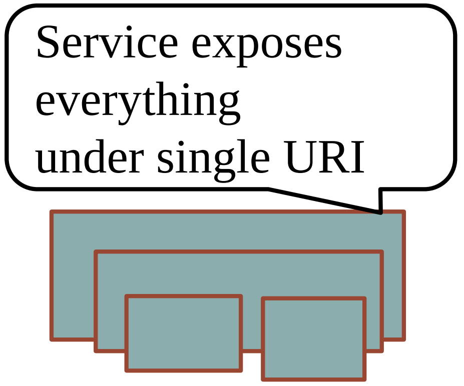
- Resource state: Kept in the server
  - The current state of a resource depends on:
    - The values of information items belonging to the resource
    - Links to related resources and future states of applications
    - The results of evaluating any application logic that relates the resource to other local resources (the state of a resource might depend on the state of other resources)
  - Same state for all the clients making simultaneous requests
  - Client and server exchange resource state representations
- Application state: Kept in the client
  - Snapshot of the entire system at a particular instant
    - What I have done till now and what can I do in the future (application workflow)
  - Application state resides at the last resource retrieved from the server

Statelessness

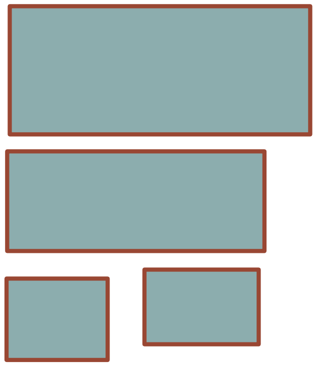
- Stateless means every HTTP request happens in a complete isolation.
  - Statelessness in REST applies to the **client application state**
  - Server never operates based on information from previous requests
    - Client includes in the HTTP request all necessary information to process a request
  - Statelessness 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

Connectedness

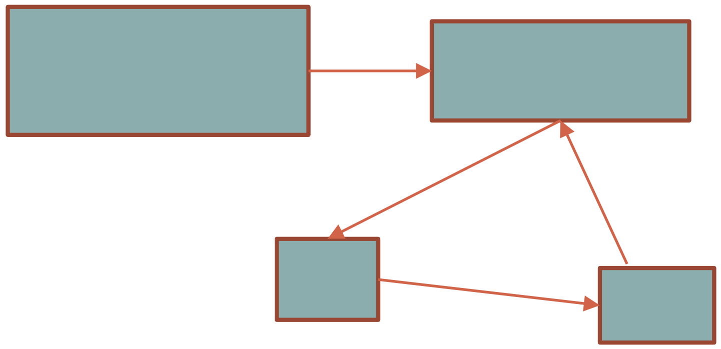
- Resource representation MUST contain reference (links) to other resources
  - Associated resources
    - Including the relation among resources
  - Future application states
    - Information on how to access those states (protocol semantics)



not addressable, not connected



Services are addressable, but not connected



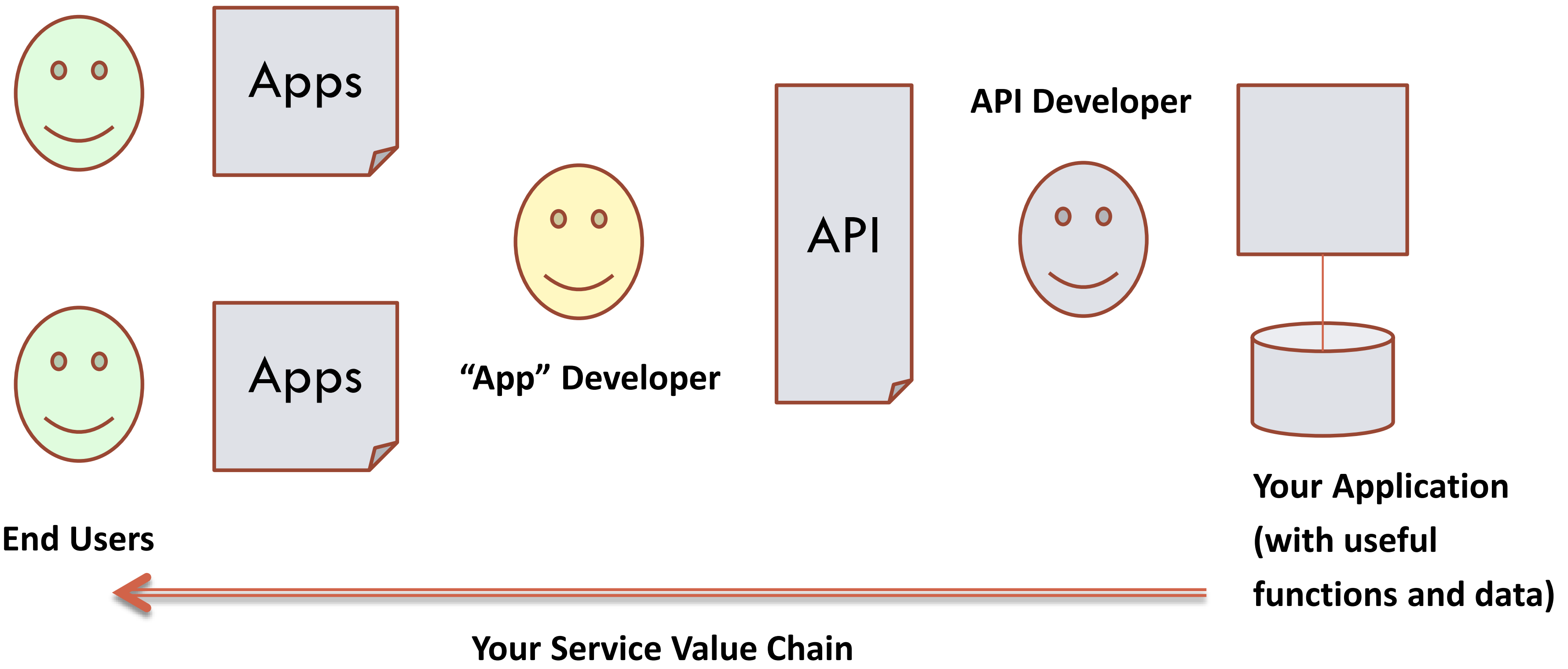
Services are addressable and connected

Architectural Constraints of REST

- Client-Server
- Uniform Interface
- Statelessness
- Caching
- Layered System
- Code on demand (optional)

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

Designing an API – who should you target?



URI design (= API endpoints)

Avoid using 'www', instead:

- https://api.twitter.com
- https://api.walmartlabs.com

Identify and "name" the resources. We want to move away from the RPC-style interface design where lots of 'operation names' are used in the URL
e.g., /getCoffeeOrders, /createOrder, /getOrder?id=123

Instead:

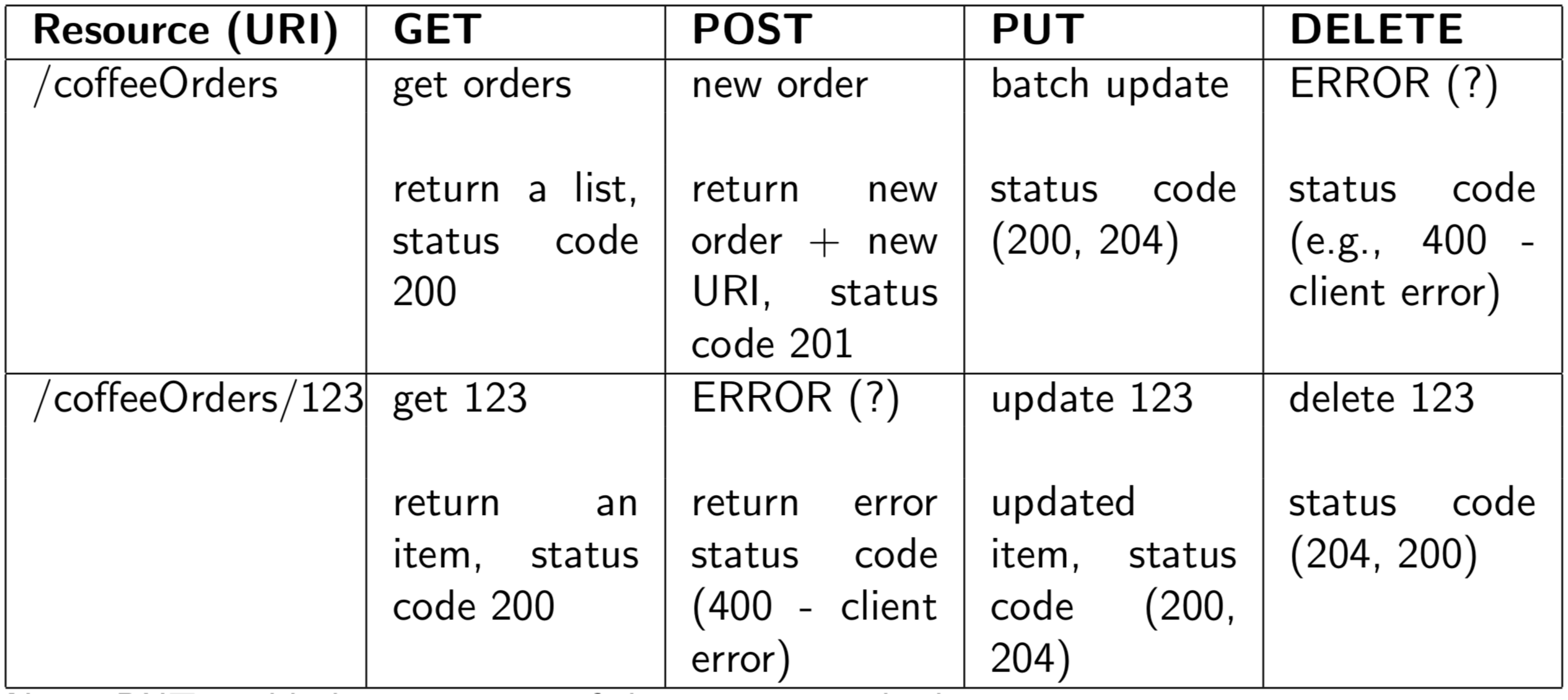
- Use nouns (preferably plurals) (e.g., orders)

•Walmart	/items	/items/{id}
•LInkedIn	/people	/people/{id}
•BBC	/programmes	/programmes/{id}

URI design …

On the resources' URIs ... we add 'actions/verbs'
Completes the endpoints of your APIs

Should it?

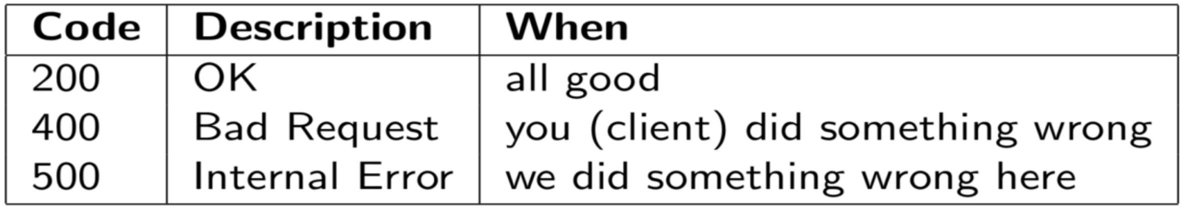


Resource (URI)	GET	POST	PUT	DELETE
/coffeeOrders	get orders return a list, status code 200	new order return new order + new URI, status code 201	batch update status code (200, 204)	ERROR (?) status code (e.g., 400 - client error)
/coffeeOrders/123	get 123 return an item, status code 200	ERROR (?) return error status code (400 - client error)	update 123 updated item, status code (200, 204)	delete 123 status code (204, 200)

Note: PUT could also return 201 if the request resulted in a new resource

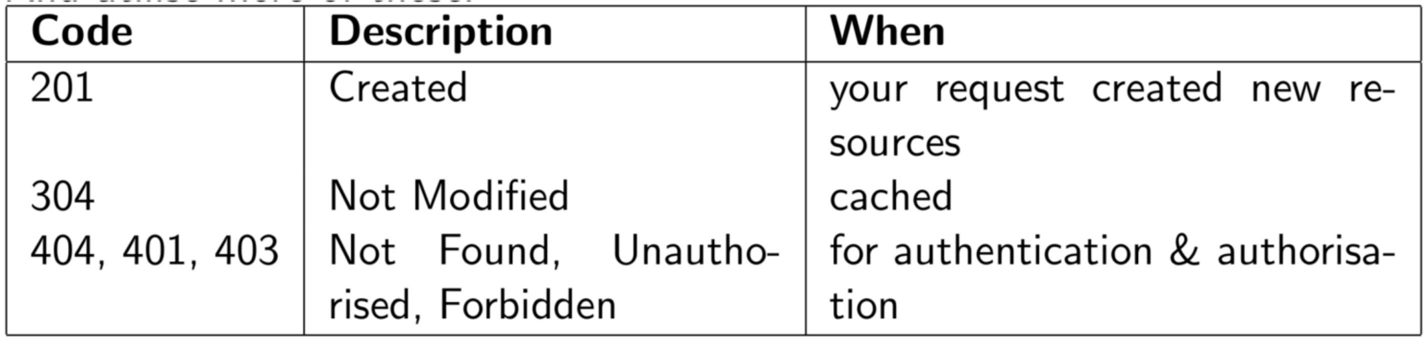
Decide How to Use the Status Codes

Using proper status codes, and using them consistently in your responses will help the client understand the interactions better.
The HTTP specification has a guideline for the codes ..., but at minimum:



Code	Description	When
200	OK	all good
400	Bad Request	you (client) did something wrong
500	Internal Error	we did something wrong here

And utilise more of these, but restrict the number of codes used by your API (clean/clear)



Code	Description	When
201	Created	your request created new re-sources
304	Not Modified	cached
404, 401, 403	Not Found, Unautho-rised, Forbidden	for authentication & authorisa-tion

RFC2616 (HTTP status codes)

https://www.w3c.org/Protocols/rfc2616/rfc2616.html

Decide your response format

Should support multiple formats and allow the client content negotiation (JSON only?)
Use simple objects.
A single result should return a single object.
Multiple results should return a collection - wrapped in a container.

```
/coffeeOrders/123

{
    "Id": "123",
    "type": "latte",
    "extra shot": "no",
    "payment": {
         "date": "2015-04-15",
         "credit card": "123457"
     },
    "served_by": "mike"
}
```

```
/coffeeOrders

{
    "resultSize": 25,
    "results": [ {
                    "id": "100",
                    "type": "latte",
                    "extra shot": "no",
                    "payment": {
                         "date": "2015-04-15",
                         "credit card": "22223"
                      },
                      "served_by": "sally"
                 },
                 { … },
               ]
}
```

Taking your API to the Next Level

HATEOAS = Hypermedia As The Engine Of Application State

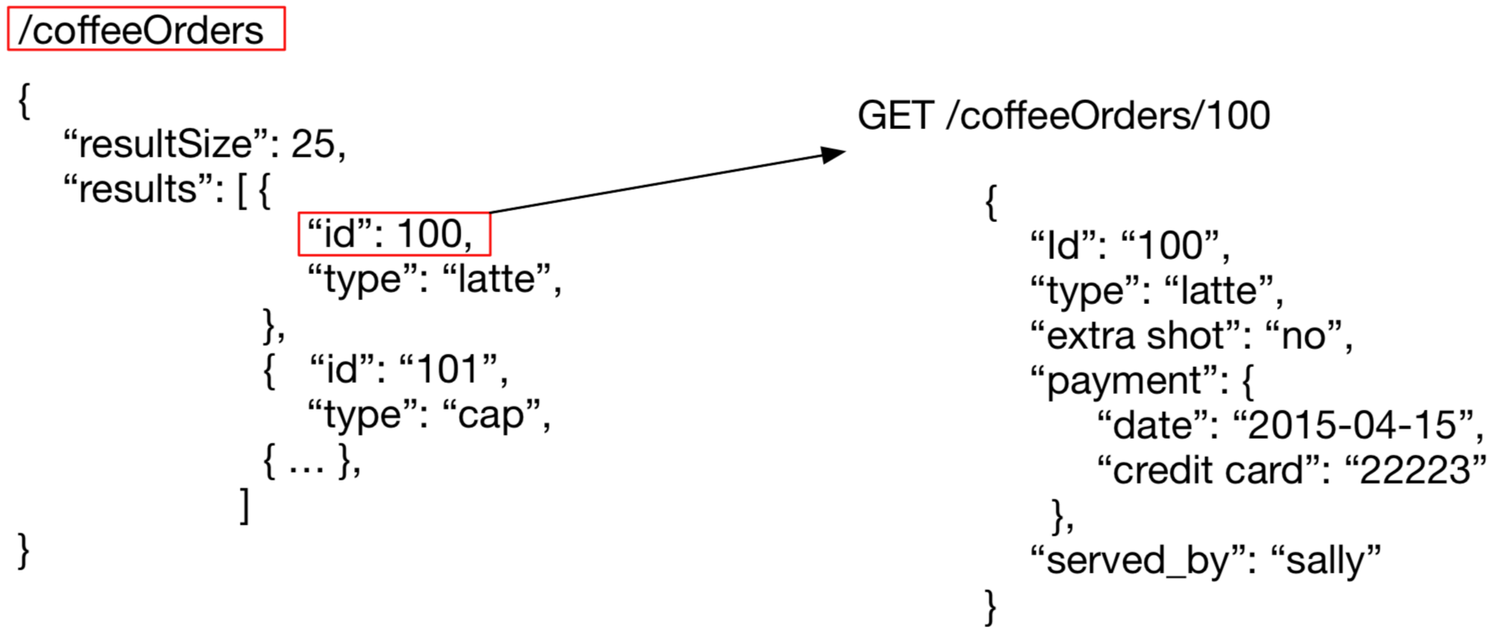
From Wikipedia: The principle is that a client interacts with a network application entirely through hypermedia provided dynamically by application servers. A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.

Think how people interact with a Web site. No one needs to look up a manual to know how to use a Web site ... Hypermedia (i.e., documents with links to other things) itself serves as a self-explanatory guide for the users.

The HATEOAS principle aims to realise this in API design.

Not Using HATEOAS

Not implementing the links in REST API would look like this:



```
/coffeeOrders

{
   "resultSize": 25,
   "results": [ {
                  "id": 100,
                  "type": "latte",
               },
               {  "id": "101",
                  "type": "cap",
               { … },
              ]
}
```

```
GET /coffeeOrders/100

   {
      "Id": "100",
      "type": "latte",
      "extra shot": "no",
      "payment": {
          "date": "2015-04-15",
          "credit card": "22223"
       },
      "served_by": "sally"
   }
```

You assume that the client knows how to construct the next request path (i.e., combine /coffeeOrders and id:100) - maybe by reading your API document?

Using HATEOAS

/coffeeOrders

```
{
   “resultSize”: 25,
   “links”: [{
           “href”: “/coffeeOrders”
           “rel”: “self”
       },
       {  “href”: “/coffeeOrders?page=1”,
           “rel”: “alternative”
       }
       {   “href”: “/coffeeOrders?page=2”,
            “rel”: “nextPage”
       }
    ],
   “results”: [ {
                   “id”: “100”,
                   “type”: “latte”,
                   “links”: [ {
                          “href”: “/coffeeOrders/100”,
                          “rel”: “details”
                    }]
                },
                { … },
    ]
```

/coffeeOrders/123

```
{
   “Id”: “123”,
   “type”: “latte”,
   “extra shot”: “no”,
   “payment”: {
       “date”: “2015-04-15”,
       “credit card”: “123457”
    },
   “served_by”: “mike”
   “links”: [ {
          “href”: “/coffeeOrders/123”,
          “rel”: “self”
          },
          {
          “href”: “/payments/123”,
          “rel”: “next”
          }
    ]
}
```

API Versioning

•When your API is being consumed by the world, upgrading the APIs with some breaking changes would also lead to breaking the existing products or services using your API.
•Try to include the version of your API in the path to minimize confusion of what features in each version.

Example:
http://api.yourservice.com/v1/stuff/34/things

Standarised API specification

OpenAPI Specification (Swagger, swagger.io)

•a standard, language-agnostic interface definition to RESTful APIs

•both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection.

•"Auto-documentation" (from the written specification)

•"Auto inspection/testing interface"

•"Auto client code generation"

An idea that is increasingly becoming attractive … and this helps RESTful client application development (every aspect of the standard is to help API consumer understand and consume API quicker/less labour intensive way)

cf. SOAP/WSDL - the calculator service (http://www.dneonline.com/calculator.asmx)

Standarised API specification

Basic structure (what is in it …)

•A quick run through:

https://swagger.io/docs/specification/basic-structure/

•Petstore.yaml

Useful resources

- Richardson and Amundsen, RESTful Web APIs, O'Reilly, 2013
- www.programmableweb.com
- Richardson and Ruby, RESTful Web Services by, O'Reilly, 2007 (http://oreilly.com/catalog/9780596529260)
- Chapter 5 of Fielding's dissertation is "Representational State Transfer (REST)“ https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- Paper “Principled Design of the ModernWeb Architecture” https://www.ics.uci.edu/~taylor/documents/2002-REST-TOIT.pdf
- RESTful API design https://www.mulesoft.com/resources/api/what-is-rest-api-design
- RESTful API Basics https://blog.fullstacktraining.com/restful-api-design-the-basics/

Questions?