Service Oriented Architecture

‘REST as a Hybrid Architectural Style’

REST

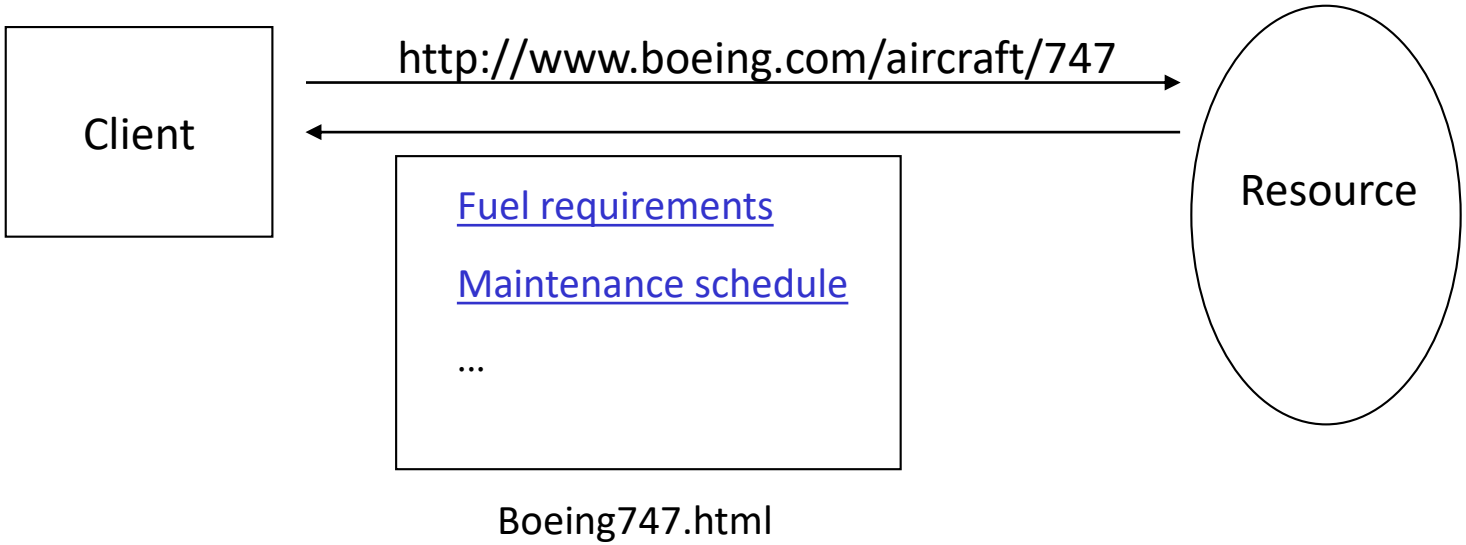
- REST is an architectural style for building distributed hypermedia systems (systems linked by hyperlinks).
- It was defined by Roy Fielding in this doctoral thesis “Architectural Styles and the Design of Network-based Software Architectures”.
- The Web (HTTP, URI and Media Types) is the best-known *instantiation* of REST and will be used to help explain REST in many of the examples.

REST

- Fielding describes REST as follows:

"The name Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through the application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use".

Representation State Transfer (REST)



*The Client references a Web resource using a URL. A **representation** of the resource is returned (in this case as an HTML document). The representation (e.g., Boeing747.html) places the client application in a **state**. The result of the client traversing a hyperlink in Boeing747.html is that another resource is accessed. The new representation places the client application into yet another state. Thus, the client application changes (**transfers**) state with each resource representation, hence Representational State Transfer"*

REST

- A resource in REST is an abstract concept, which is made concrete via a representation of that resource. For example, note that in the previous slide that the resource being accessed is logical, and not a physical object i.e. the URI finishes with “747” as opposed to “747.html”.

- The URI accessed is the abstract resource and what is returned is the HTML representation of that resource i.e. Boeing747.html. This decouples the implementation of the resource from the clients.

Resources

- REST refers to information abstractly as “resources”. “Any information that can be named can be a resource”. Examples include “today’s weather at Dublin airport” and “version 1.1 of the software”.

- A resource has an identifier and a representation. For example, on the Web, one uses a URI to identify a resource, which may have a HTML/JSON/HTML representations.

Representations

- The data represents the physical version of the abstract resource e.g. a HTML representation of the resource “Sky News home page”.

- The data format of a representation is known as a media type. On the Web, the media types are well known standard types maintained by the Internet Assigned Numbers Authority (IANA). We mostly use JSON in this course.

Resource State

- Resource state refers to the data associated with a resource. It resides on the server (in databases typically) and is sent to the client via representations.

- An example would be a post on your Facebook page. It resides in a database on the Facebook server and is transferred to the browser when you access your page.

Application State

- Application state is information about the path taken by a client though an application and is therefore different for each client i.e. client-specific.
- Application state resides on the client.
- An example of application state is the history stored by a browser, where page 1..n links are stored.
- The history enables users to go back through their browsing history (i.e. their previous application states) via the Back button.

Deriving REST

- REST is a hybrid style derived from other network-based architectural styles. In his thesis, Fielding defines an architectural style as a “coordinated set of architectural constraints that has been given a name for ease of reference”.

- The use of an architectural style applies the associated *constraints* on the system. Each *constraint* induces certain properties e.g. simplicity or scalability.

- Thus, a style applies (it’s) constraints, which induce certain properties.

Deriving REST

- Fielding defines the properties of key interest when considering the target architecture of network-based hypermedia (the Web); for example: scalability, simplicity, visibility and independent evolvability.

- He then evaluates several common network-based architectural styles (e.g. client-server) for the properties they would induce.

- He then derives REST by applying the styles that induce the properties he requires.

- https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

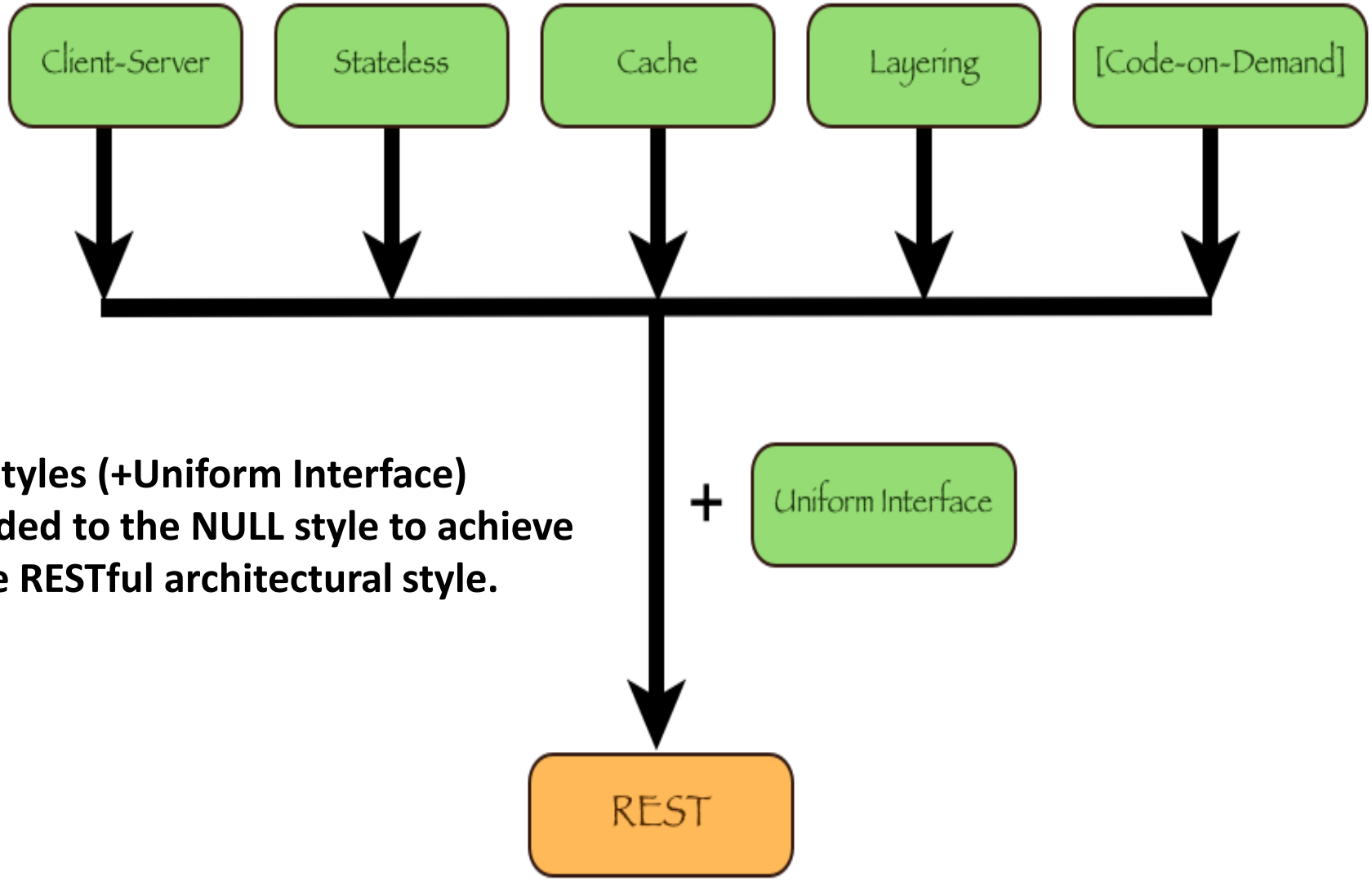
Deriving REST

- To do this, Fielding firstly defines the "null" style i.e. a style with *no constraints* at all.

- Fielding then adds certain pre-defined styles, which induce the desired properties for the target architecture of network-based hypermedia.

- This hybrid style is combined with other constraints (most notably the uniform interface constraint) to form the REST architectural style.

Deriving REST



5 Styles (+Uniform Interface) added to the NULL style to achieve the RESTful architectural style.

Client-Server Style

- Client-Server (CS) style
  - This is the first style to be added. The software engineering principle guiding this style is the “separation of concerns” principle.
  - This is fairly obvious given the distributed nature of the systems we’re dealing with. The main idea here is that the server can be changed without the client having to change.
  - This separation of concerns enables components to evolve independently - an important consideration given the scale of the Web and the large number of enterprises it hosts

Stateless Style

- Stateless constraint (CSS = Client-Stateless-Server)
  - The stateless constraint is added to the client-server interaction and states that each client request is independent of any other request i.e. each request must contain all the information necessary to understand it.
  - Session state is therefore kept on the client.

Stateless Style

- Stateless constraint (CSS = Client-Stateless-Server)
  - Statelessness does not mean that you cannot have state persisted in a database on a server – resource state (outlined earlier) is stored on the server. However, the application state is not.
  - “A RESTful service is ‘stateless’ if the server never stores any application state” [Ruby and Richardson 07].

Stateless Style

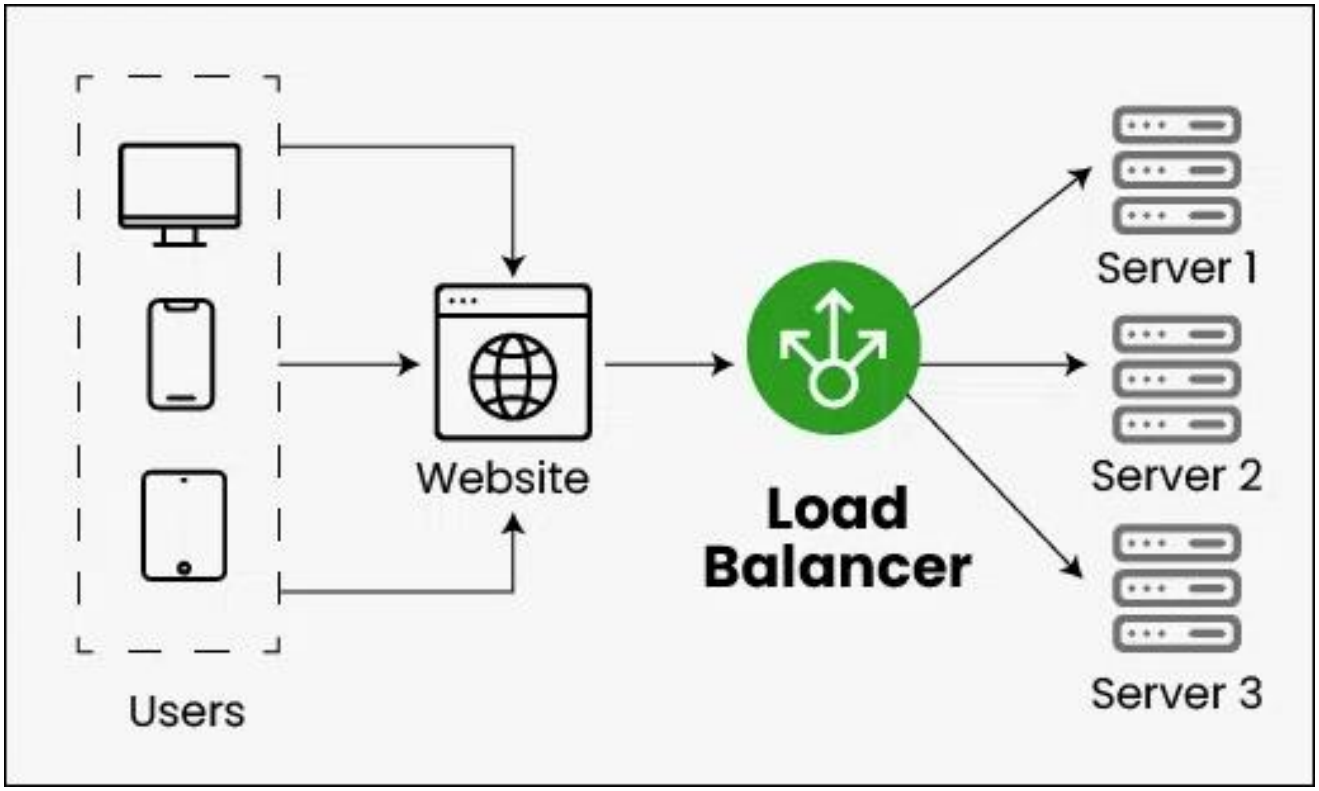
- Stateless constraint (CSS = Client-Stateless-Server)
  - The stateless constraint induces the properties of visibility and scalability.
  - Visibility is improved because an intermediary can inspect and understand the message, as all the information necessary is present in the message.

Stateless Style

- Stateless constraint (CSS = Client-Stateless-Server)
  - As no two requests depend on each other, different servers can handle them i.e. it does not matter which server you talk to.
  - Thus, improving scalability is a matter of adding extra servers to a load balancer [Ruby and Richardson 07]. Scalability is improved as the server implementation is simplified because the server does not have to store state between requests.

Stateless Style

- Adding more servers allows the application to scale; however, each request must be free to go to a new server for the load balancer to work effectively.



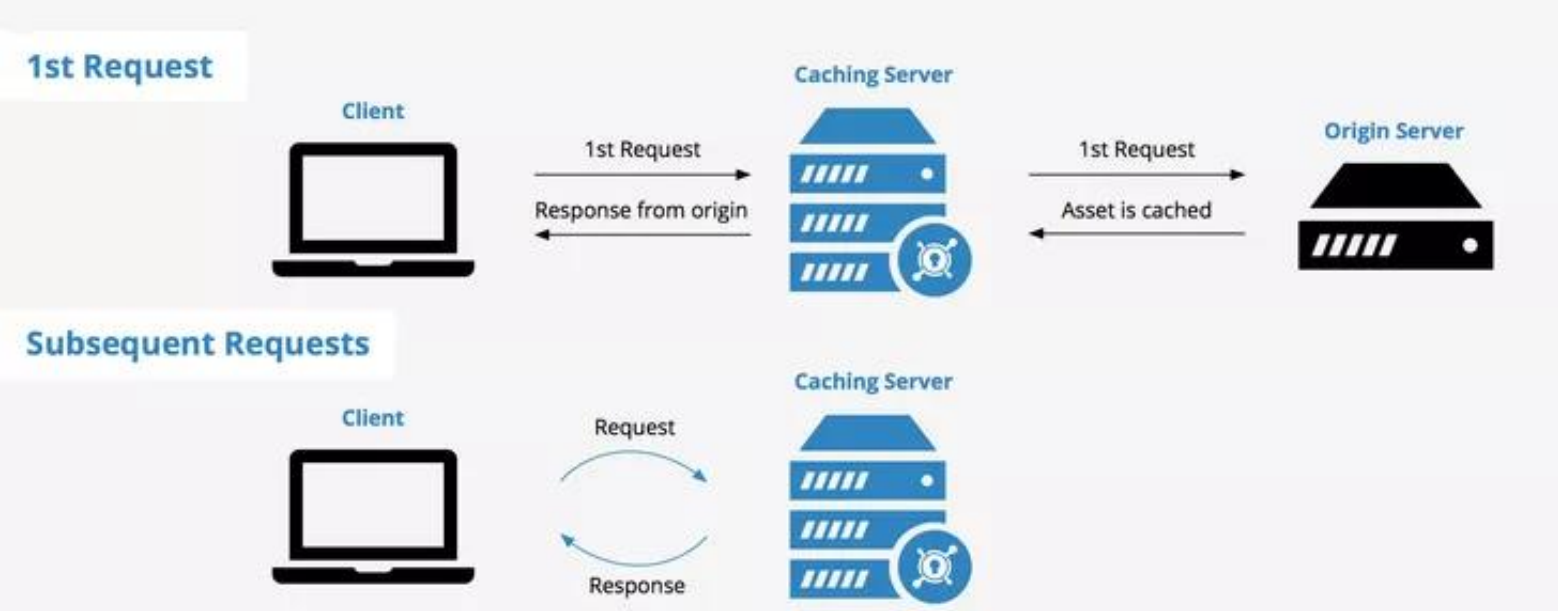
- Picture link: https://www.geeksforgeeks.org/what-is-load-balancer-system-design/

Cache Style

- Cache constraint (CCSS = Client-Cache-Stateless-Server)
  - The next constraint added is the cache constraint. This constraint is added to improve network efficiency.
  - The cache constraint specifies that a response to a request can be cached to satisfy subsequent equivalent requests.
  - Caches improve :
    - network efficiency (as the request may not travel all the way to the origin server) and
    - scalability (allows the introduction of extra client components).

Cache Style

- Concept of a cache:



- Caches improve network efficiency and as a result allow for more clients to be added.

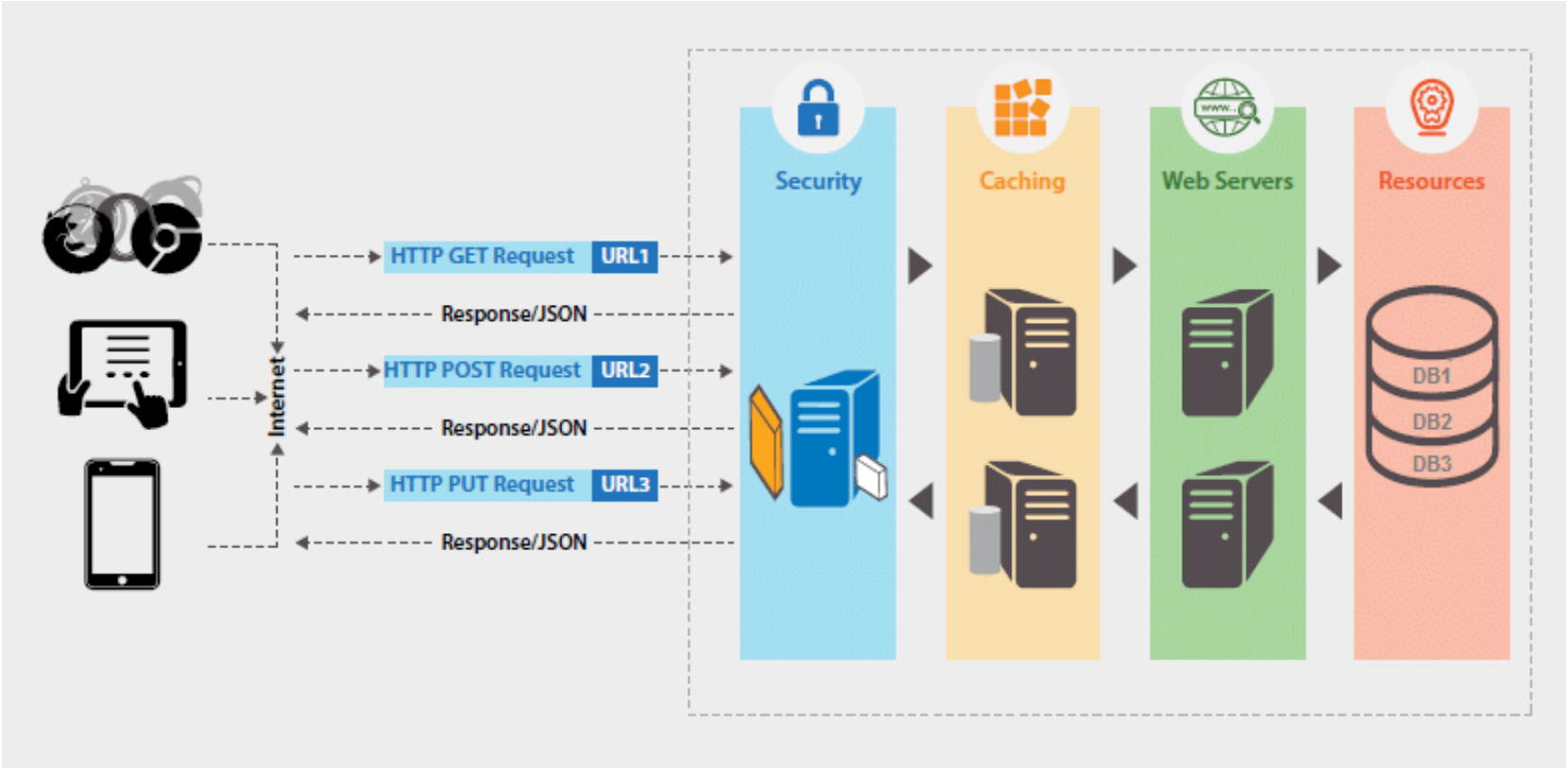
- Picture link: https://www.keycdn.com/support/cache-definition-explanation

Layered Style

- Layered system (LCCSS = Layered-Client-Cache-Stateless-Server)
  - A layered system is organised hierarchically such that each layer provides services to the layer above it and uses services from the layer below it e.g. the TCP/IP model.
  - Each layer is constrained from seeing beyond the layer with which it is communicating.
  - A layered client-server adds components such as proxies/gateways to the client-server style. Thus, an application can interact with a resource without knowing if or how many caches, proxies and firewalls exist between it and the server actually holding the information [REST Wikipedia].

Layered Style

- The layered system can also include features such as load balancing (which helps scalability)



- Picture link: https://www.researchgate.net/figure/REST-layered-style-according-to-the-model-proposed-by-Deepak-2015-p-11_fig1_316675285

Code-on-demand Style

- Code-on-demand (LCODCCSS = Layered-Code-On-Demand-Client-Cache-Stateless-Server)
  - With this style, a client downloads code from a server and executes it locally e.g. applets or scripts (JS).
  - Extensibility is improved because features can be added to pre-implemented clients.
  - The most significant limitation is the lack of visibility as the server is sending code (not simple data) and thus it is only an optional constraint in REST [Fielding 00].

Uniform Interface

- Uniform Interface constraint (REST = Layered-Code-On-Demand-Client-Cache-Stateless-Server + Uniform Interface)
  - The last constraint added to define REST is the Uniform Interface constraint.
  - The uniform interface is the most differentiating of REST's constraints [Fielding 00].

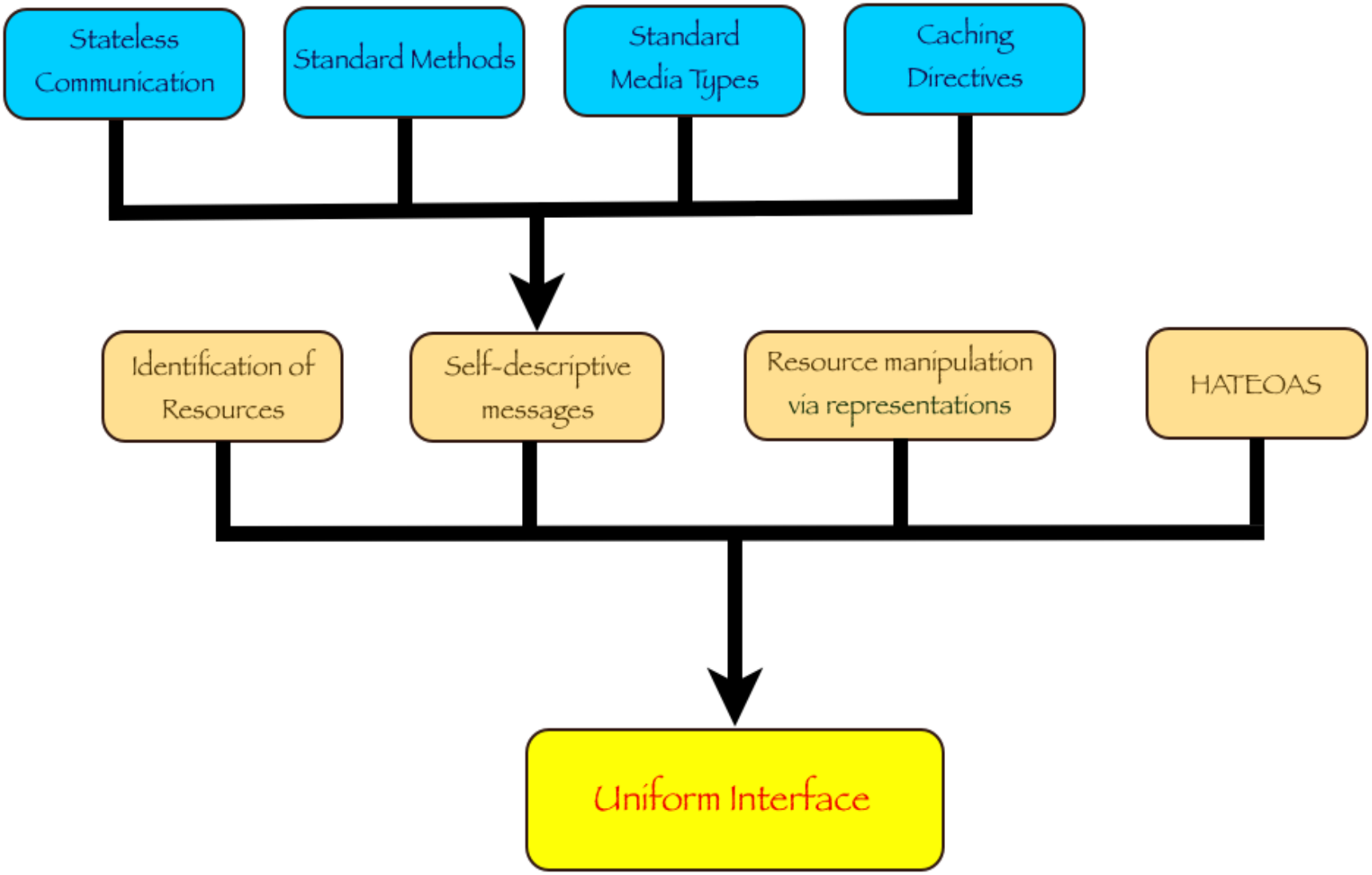
Uniform Interface

- *'Generality'* and '*Information hiding'* are the software engineering principles guiding this constraint.
  - The generality principle encourages simplicity whereas the information hiding principle ensures that implementations are decoupled from the interface thereby encouraging independent evolvability.
  - Generality is a s/w engineering principle that motivates constraints to induce the property of reusability of components e.g. s/w reuse. Definition - "while solving a problem, try to discover if it is an instance of a more general problem whose solution can be re-used in other cases". (All interactions can be CRUD operations)

Uniform Interface

- This constraint enables visibility into component interactions and is obtained with the support of the following sub-constraints [Fielding 00]:
  - Identification of resources (unique URI for each resource)
  - Self-descriptive messages (GET, POST, PUT, DELETE etc.)
  - Manipulation of resources through representations (JSON/XML)
  - Hypermedia as the engine of application state (HATEOAS)

Uniform Interface



Uniform Interface

- Identification of Resources
  - “Resources are named with URI’s”. All resources that merit identification get an identifier.
  - On the Web, URI’s make up a global namespace - using URI’s to identify your resources means they get a unique, global ID.
  - This is how the online store Amazon.com operates: every one of its products gets a unique ID (a URI).

Uniform Interface

- Identification of Resources
  - Resources are abstract and can relate to anything you merit as being identifiable, for example:
    - an individual item e.g. http://example.com/customers/1234
    - a collection of items e.g. http://example.com/customers

Uniform Interface

- Self-descriptive messages
  - “REST constrains messages between components to be self-descriptive in order to support intermediate processing of interactions”
  - This means that the message semantics are visible without looking at the message body, thus enabling intermediaries (e.g. proxies and caches) to inspect the message and act accordingly.
  - This constraint enables “critical distributed systems concepts such as proxying, caching, and monitoring”.

Uniform Interface

- Self-descriptive messages
  - For a message to be self-descriptive it requires the following:
    - stateless communication: each request contains all the information necessary to understand the request, independent of any previous request e.g. login information
    - use standard methods: in HTTP, the primary methods are GET, PUT, POST and DELETE. These methods have well defined semantics as per the HTTP specification [HTTP 1.1]. For example, a proxy understands that a HTTP GET is a retrieval message.
    - standard media types: the format of the information exchanged by components is standard and well-known e.g. on the Web, one has the Internet media types XML and JSON.
    - response messages that are to be cached must be marked accordingly

Uniform Interface

- Manipulation of resources through representations
  - Resources are manipulated via representations in the format of media types.
  - If a resource supports more than one representation, the client can use content negotiation to select the most appropriate one.
  - For example, if a resource supports both JSON and XML representations, then clients can specify which one is required.

Uniform Interface

- Manipulation of resources through representations
  - Clients can also send content to create/update the resource state by sending the resource representation with the message.
  - For example, to update a resource in HTTP, a client PUTs the new representation identified by the Content-Type header e.g. text/json, to the resource identified by the URI.

Uniform Interface

- Hypermedia as the engine of application state (HATEOAS)
  - This sub-constraint is based on the idea of hypermedia i.e. links. Given that the Web is a RESTful application it helps to refer to it when explaining this sub-constraint.
  - When one visits a Web site, for example visiting www.tus.ie, one receives back HTML, images and hyperlinks. At this point, we are in the initial state of a virtual state machine representing the Web site.
  - By clicking on a hyperlink on the page we execute a state transition whereby a new resource representation (a HTML page), is transferred to the browser. This reflects our transfer into a new application state. Thus, we move between application states in the virtual state machine by clicking on hyperlinks.

Uniform Interface

- Hypermedia as the engine of application state (HATEOAS)
  - The next slide shows an example file demonstrating how HATEOAS should be supported in a programmatic client i.e. a non-browser based client.
  - Firstly, note that the ref attribute contains a link. However this link is not just some application-specific identifier: it is a URI. As URI's are global standards, all the resources on the Web can be linked to each other i.e. using URI's as links means that the links can point to resources provided by a different application, a different server or a different company anywhere in the world.
  - Note that it is the server that is guiding the client through the application because it is the server that builds the representation which contains the links to guide the client through the application state.

Uniform Interface

- Hypermedia as the engine of application state (HATEOAS)
  - If implemented properly, a client need only be aware of the entry-point URI; the other URI's will be in the resource representations. This is similar to a Web site visit i.e. one enters the first URL in a browser and follows hyperlinks from that point on.

```
<order self='http://example.com/customers/1234' >
    <amount>23</amount>
    <product ref='http://example.com/products/4554' />
    <customer ref='http://example.com/customers/1234' />
</order>
```

Summary

- REST is an abstract architectural style that places constraints on the elements within the architecture.
- It was defined for optimal performance for the common case on the Web i.e. large-grain (large amounts) data transfer.
- Fielding identified other architectural styles that demonstrated desired properties in the Web.
- He amalgamated these styles together to obtain a hybrid style to which he added a uniform interface constraint: “to form a new architectural style that better reflects the desired properties of a modern Web architecture”.

Summary

Style/Properties	Simplicity	Scalability	Independent Evolvability	Visibility	Extensibility	Efficiency
Client-Server	✓	✓	✓			
+ Stateless		✓		✓		
+ Cache		✓				✓
+ Layering	✓	✓				
+ [Code-on-Demand]					✓	
+ Uniform Interface	✓	✓	✓	✓		
REST	✓	✓	✓	✓	✓	✓

Styles used in deriving REST, and the key properties induced by REST