- It grades APIs by their Restful maturity
- There are 4 levels. Only when the APIs reach the level 4, we can talk about a Restful API. The levels are:
  - Level 0
  - Level 1
  - Level 2
  - Level 3

Level 0

To get the data and to post the data we send request to the same URL, Only POST request may be used.

To get the data,
POST http://localhost:400/customer

To post the data,
POST http://localhost:400/customer

we send info in the body base of the info we get the data or cretate the data

Level 0 (The Swamp of POX)

HTTP protocol is used for remote interaction

... the rest of the protocol isn't used as it should be

```
POST (info on data)
http://host/myapi

POST (author to create)
http://host/myapi
```

LEVEL 1

- In level 0, there was only 1 URI.
- In level 1, each resource is mapped to a specific URI.
  For example: the URI to get products
  http://localhost:2300/products

  To get URI for a specific product
  http://localhost:2300/products/1
- Still, only one HTTP method (POST) is still used for retrieving & creating data

Level 1 (Resources)

Each resource is mapped to a URI

HTTP methods aren't used as they should be

```
POST
http://host/api/authors

POST
http://host/api/authors/{id}
```

LEVEL 2

- At Level 2, correct HTTP verbs are used with each request
- For each of those requests, correct HTTP response code is provided.
- Example,
  GET http://localhost:2300/customers
  200 OK

Level 2 (Verbs)

Correct HTTP verbs are used

Correct status codes are used

```
GET
http://host/api/authors
200 Ok (authors)


POST (author representation)
http://host/api/authors
201 Created (author)
```

LEvel 3

- At level 3, the APIs support HATEOS
- Example,
  GET http://localhost:2300/customers
  For this, the response that we get is
  Customer JSON + Self-documenting Hypermedia
- Helps in Self-documentation

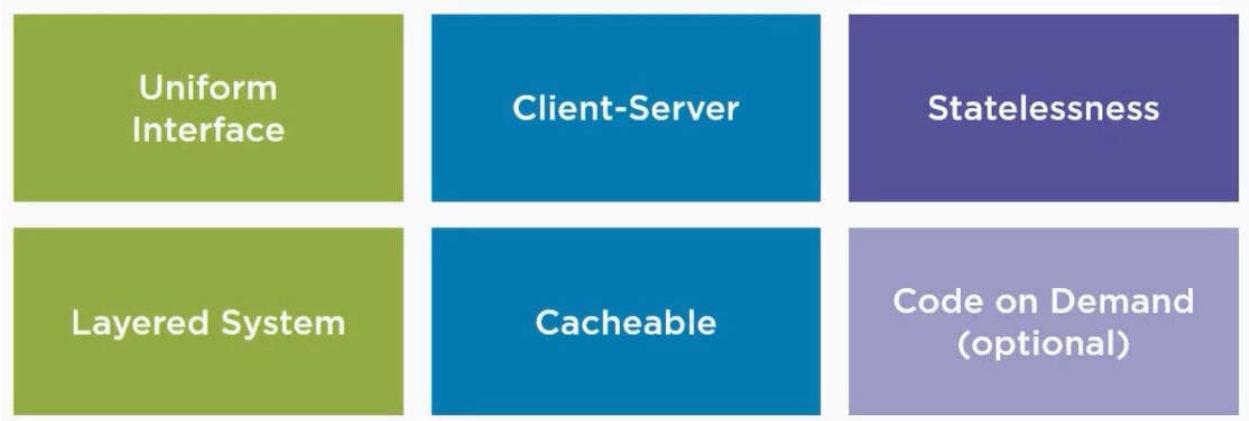
Level 3 (Hypermedia)

The API supports Hypermedia as the Engine of Application State (HATEOAS)

Introduces discoverability

```
GET
http://host/api/authors
200 Ok (authors + links that
drive application state)
```

Rest is defined by 6 constrains



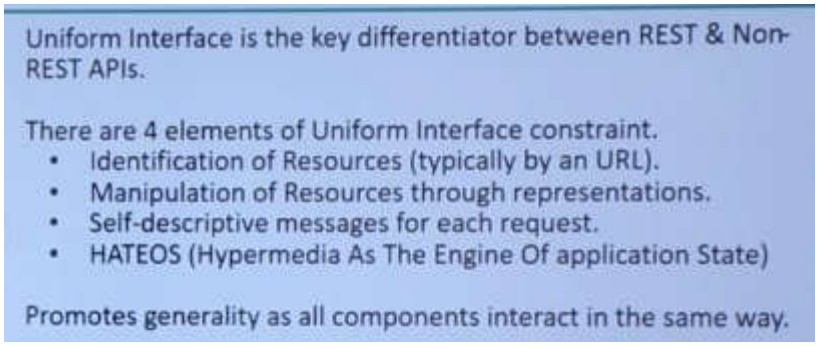
Uniform interface (key constrained)

This constraint has 4 parts:

1. The request to the server has to include a resource identifier
2. The response the server returns include enough information so the client can modify the resource
3. Each request to the API contains all the information the server needs to perform the request, and each response the server returns contain all the information the client needs in order to understand the response.
4. Hypermedia as the engine of application state — this may sound a bit cryptic, so let's break it down: by application we mean the web application that the server is running. By hypermedia we refer to the hyperlinks, or simply links, that the

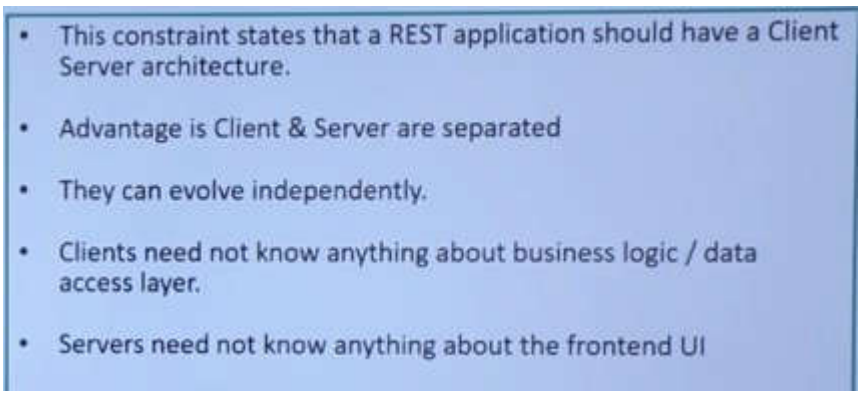
server can include in the response. The whole sentence means that the server can inform the client , in a response, of the ways to change the state of the web application. If the client asked for a specific user, the server can provide not only the state of that user but also information about how to change the state of the user, for example how to update the user's name or how to delete the user. It is easy to think about the way it's done by thinking about a server returning a response in HTML format to a browser (which is the client). The HTML will include tags with links (this is the hypermedia part) to another web page where the user can be updated (for example a link to a 'profile settings' page). To put all of this in perspective, most web **pages** do implement hypermedia as the engine of application state, but the most common web **APIs** do not adhere to this constraint. To further understand this concept, I highly recommend watching this 30 minutes YouTube video.

The result of the uniform interface is that requests from different clients look the same, whether the client is a chrome browser, a linux server, a python script, an android app or anything else.



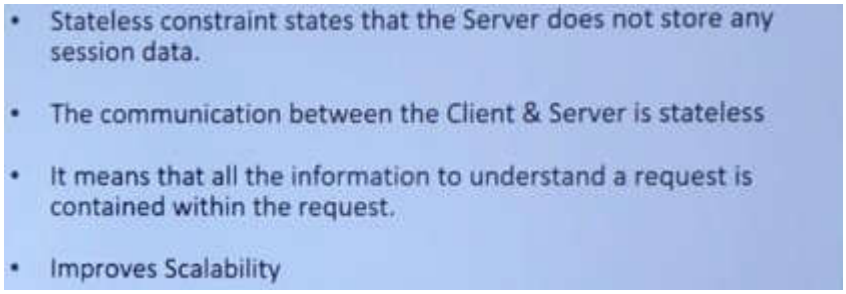
Client — server separation

The client and the server act independently, each on its own, and the interaction between them is only in the form of requests, initiated by the client only, and responses, which the server send to the client only as a reaction to a request. The server just sits there waiting for requests from the client to come. The server doesn't start sending away information about the state of some resources on its own.



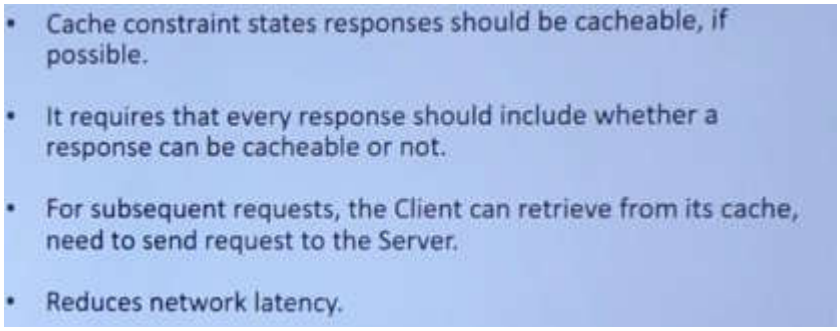
Stateless

Stateless means the server does not remember anything about the user who uses the API. It doesn't remember if the user of the API already sent a GET request for the same resource in the past, it doesn't remember which resources the user of the API requested before, and so on.Each individual request contains all the information the server needs to perform the request and return a response, regardless of other requests made by the same API user.



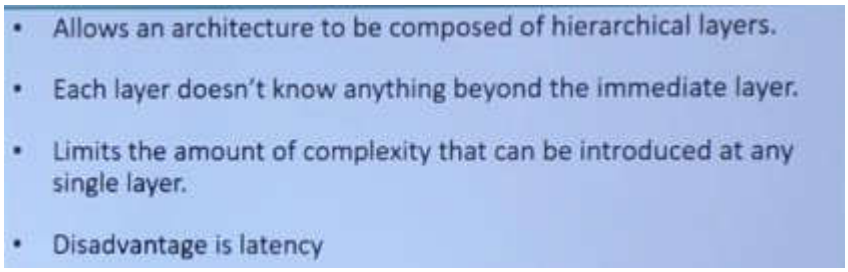
Cacheable

This means that the data the server sends contain information about whether or not the data is cacheable. If the data is cacheable, it might contain some sort of a version number. The version number is what makes caching possible: since the client knows which version of the data it already has (from a previous response), the client can avoid requesting the same data again and again. The client should also know if the current version of the data is expired, in which case the client will know it should send another request to the server to get the most updated data about the state of a resource.



Layered system

Between the client who requests a representation of a resource's state, and the server who sends the response back, there might be a number of servers in the middle. These servers might provide a security layer, a caching layer, a load-balancing layer, or other functionality. Those layers should not affect the request or the response. The client is agnostic as to how many layers, if any, there are between the client and the actual server responding to the request.



Code-on-demand

This constraint is optional — an API can be RESTful even without providing code on demand.

The client can request code from the server, and then the response from the server will contain some code, usually in the form of a

script, when the response is in HTML format. The client then can execute that code.