IN5320

RESTful Web Services

Outline

- The REST Architectural Style
- HTTP - REST in practice
- RESTful web services
- RESTful web services compared to other web services and tools

Why REST?

- Already worked with a RESTful API in the first assignment, probably more in the second
- Platform literature: *interfaces* are a critical component of a platform ecosystem. Interfaces are often Web APIs, and often (claim to be) RESTful
- Group project with DHIS2 platform based on working with a REST API

REpresentational **S**tate **T**ransfer

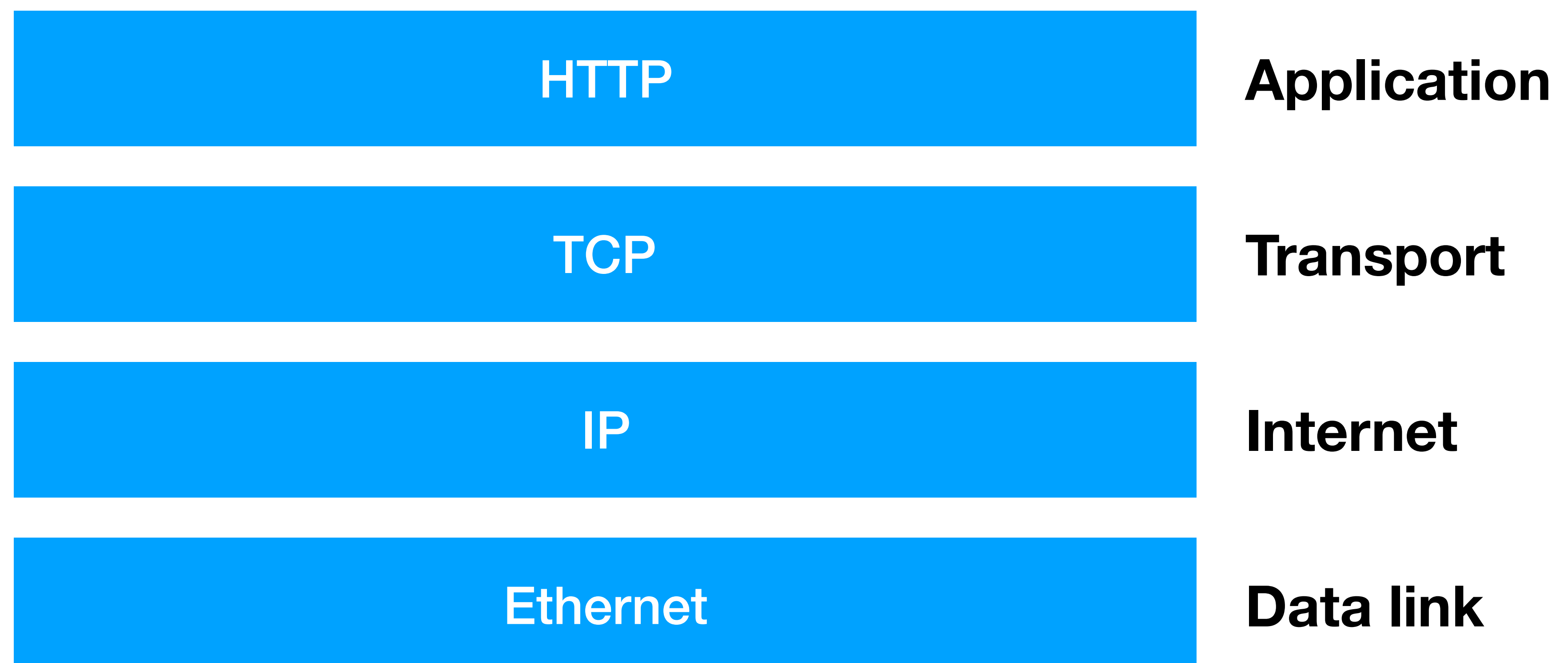
- REST is an *architectural style*
- An architectural style is a set of *architectural constraints*
- Simply put: *architectural constraints* can be thought of as "rules" for what is allowed within an architecture

REST and HTTP

- REST architectural constraints guided the development of HTTP
- HTTP is a *standard* - REST is not!

HTTP

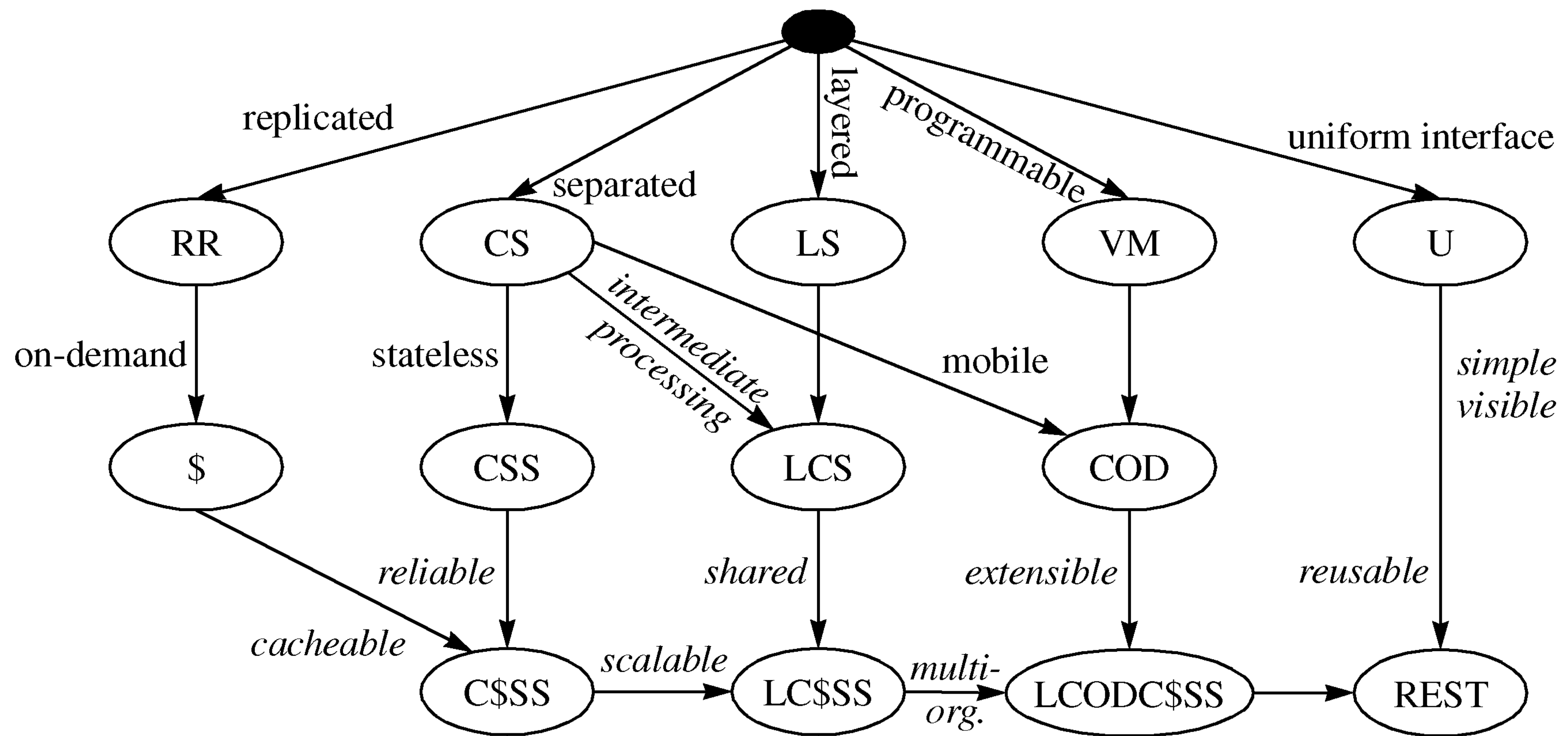
- HyperText Transfer Protocol
- Application layer protocol - foundation for data communication on the Web



HTTP Background

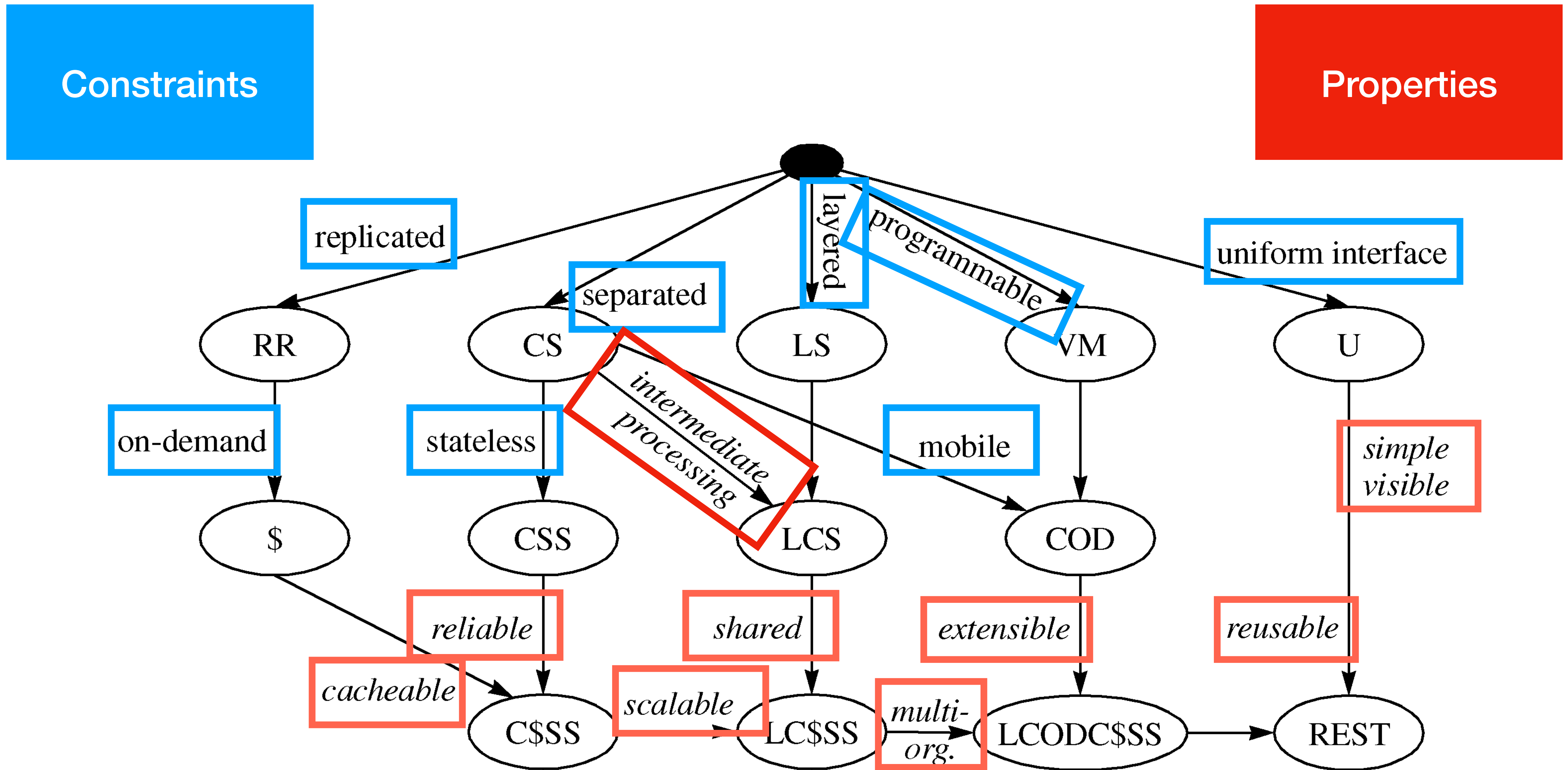
- Work on HTTP protocol started in 1989
- HTTP/1.1 first released in 1997 - updated since
- Protocol for the Web, which meant it needed:
  - Low entry-barrier to enable adoption => simple
  - Preparedness for change over time => extensible
  - Usability of hypermedia (links) => minimal network interactions
  - Deployed on internet scale => built for unexpected load and network changes
- Dissertation on REST first published in 2000

REST Architectural constraints



Source: Fielding and Taylor (2002)

REST Architectural constraints



Source: Fielding and Taylor (2002)

Style = null

- Starting point: No constraints
- Adding constraints that result in desirable properties
- Goal: architecture with minimal latency and network communication, and maximum scalability and independence of components

Style += Client-Server

- Client-server architecture
- Separation of concerns - interface from data storage

+ Simplifies the server component
+ Components can evolve separately
+ Improves UI portability

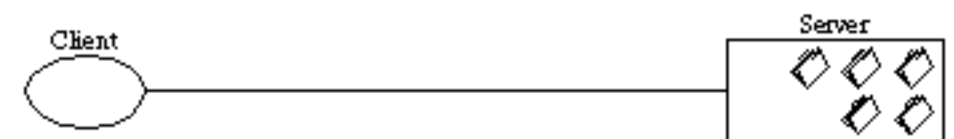


Figure 5-2. Client-Server

Style += Stateless

- Communication/interactions must be stateless:
  - Each request must be self-descriptive
  - Session state is kept by client

\+ Improves visibility, reliability and scalability

\+ Simplified server

\- Decrease network performance due t

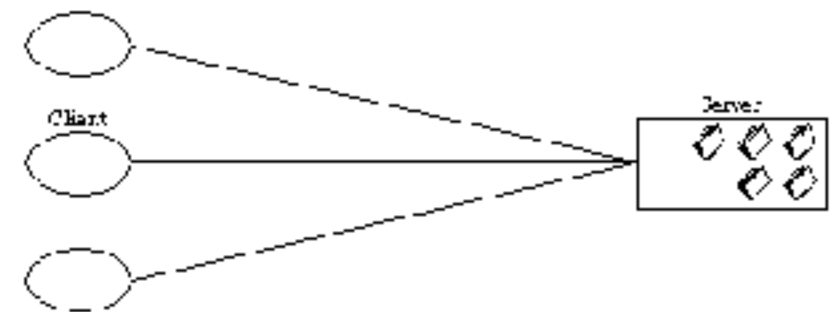


Figure 5-3. Client-Stateless-Server

Style += Cacheable

- Clients and intermediaries can cache response
- Data within a response must be labeled cacheable (or not)

\+ Improves network performance and reduces interaction

\- Can decrease reliability

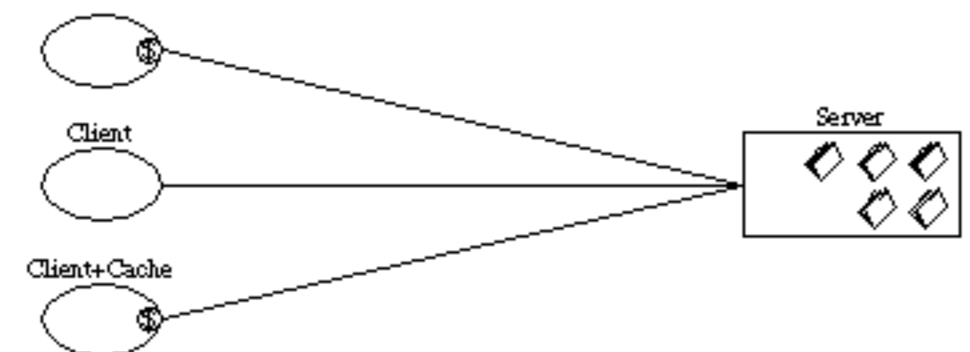


Figure 5-4. Client-Cache-Stateless-Server

Style += Layered system

- The architecture can consist of hierarchical levels
- Components only communicate with their "neighbours"

\+ Reduce system complexity

\+ Intermediaries can improve efficiency, e.g. provide caching

\- Adds overhead and latency

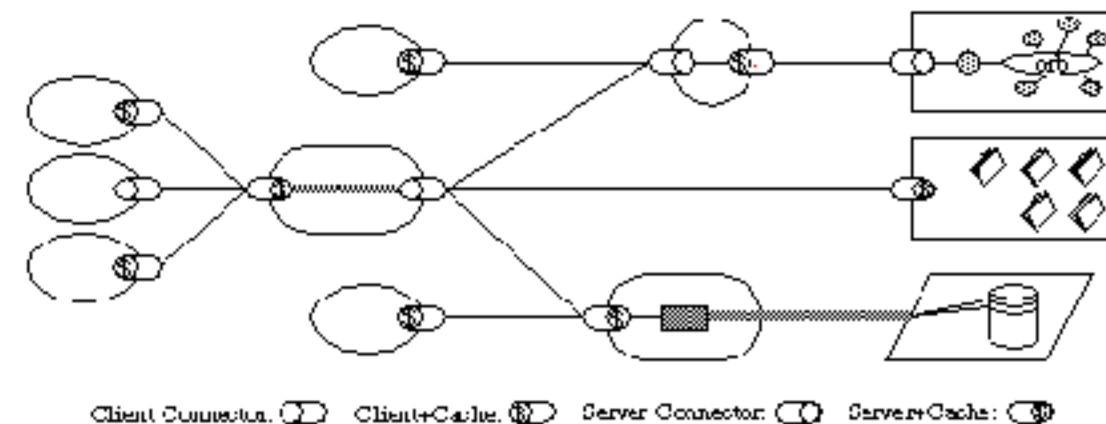


Figure 5-7. Uniform-Layered-Client-Cache-Stateless-Server

Style += Uniform interface

- There is a uniform interface for interacting with resources
- Five interface constraints:
  - *Addressability* - all resources are identified by one identifier mechanism
  - *Universal semantic* - a small set of standard methods support all interactions and apply to all resources
  - *Resource representations* - resources are manipulated through their representations
  - *Self-describing messages* - interactions happen through request and response message that contain both data and metadata
  - *Hypermedia as engine of application state (HATEOAS)* - resources include links to related resources, enabling decentralised discovery. Application state is kept on client, resource state on server.

Style += Uniform interface

- \+ Decouples implementations from services that are provided

- \- Can decrease efficiency - information is transferred in a standard format rather than optimised to the application

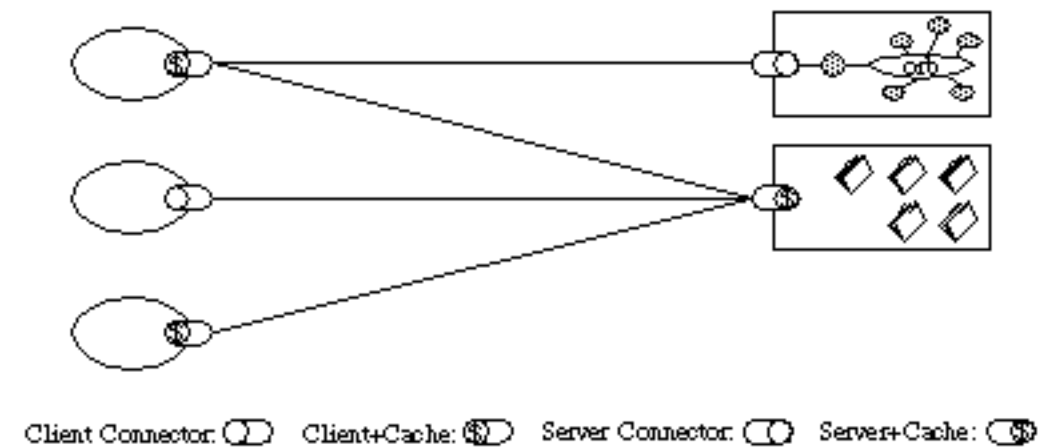


Figure 5-6. Uniform-Client-Cache-Stateless-Server

Style += Code-on-demand

- Clients can download and execute code to extend functionality

+ Simplifies clients and improves extensibility

- Reduces visibility

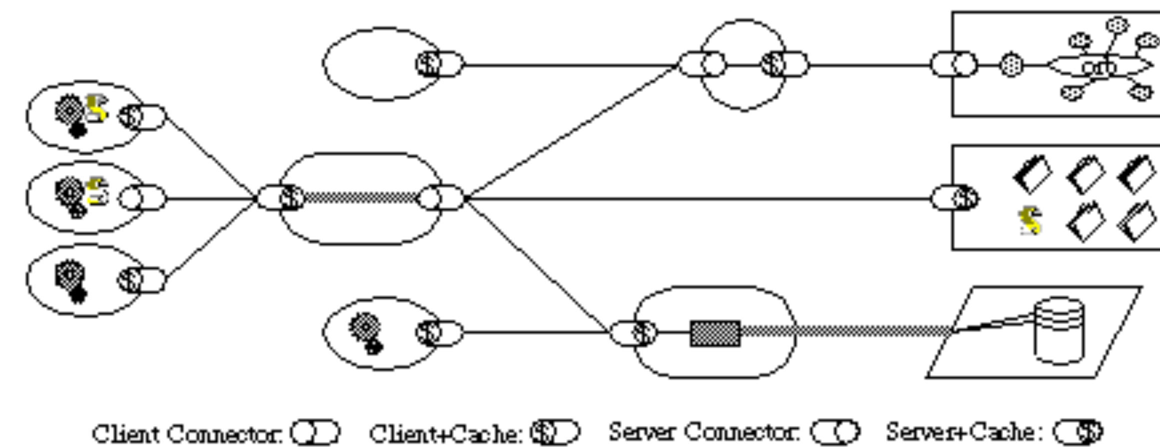
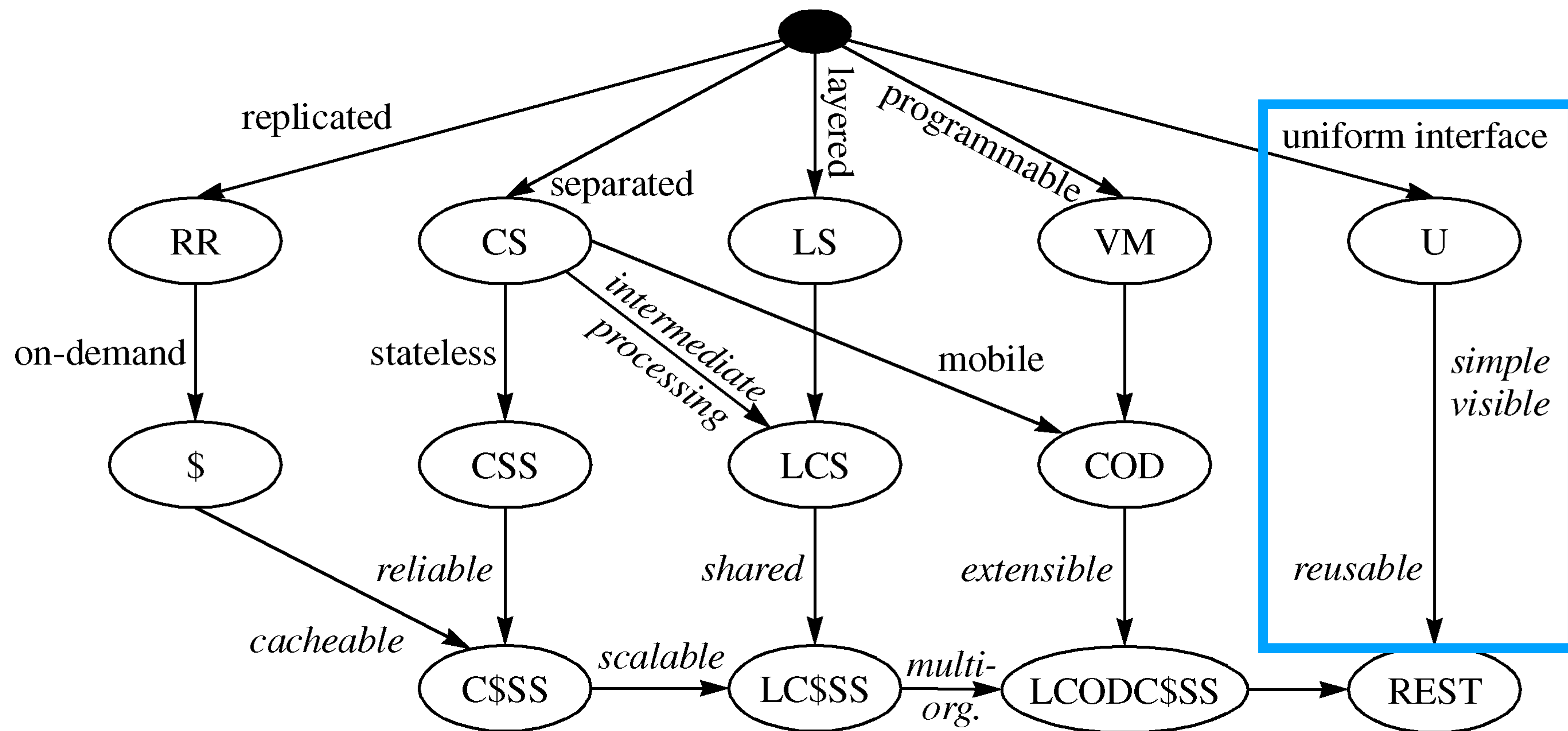


Figure 5-8. REST

Note on REST definitions

- Often, only the *Uniform interface* constraint(s) are listed, e.g. in Pautasso
- *Uniform interface* for Pautasso is the limited set of methods for manipulation (GET, POST etc)
- *Uniform interface* for Fielding consists of five (four in Fielding 2002) constraints above

REST Architectural constraints



Source: Fielding and Taylor (2002)

REST architectural elements

- Data elements
- Components
- Connectors

REST data elements

Data element	Example
resource	conceptual target of reference, e.g. today’s weather
resource identifier	URL
representation	HTML document, XML document, image file
representation metadata	media type, last-modified
resource metadata	source link, alternates
control data	cache-control

Resources

- Resources are the key information elements in REST
- Any information that can be named can be a resource - image, service, document
- Resources refer to conceptual mappings, not particular entities or values
- Abstract definition of resources enables:
  - generality - information is not divided by type, implementation
  - late binding to representation - representation (format) can be decided based on request
  - we can refer/link to (persistent) concepts rather than specific instances of a concept

Resources - example

https://www.yr.no/sted/Norge/Oslo/Oslo/Oslo_(Blindern)_målestasjon/varsel.xml

- "6-hour forecast for Oslo" is a resource
- Values/content changes regularly, but we can refer to the *resource* over time

Resource identifiers

- Each resource needs an unique identifier - URI
- Identifier is defined by the "author" of the resource, not centralised
- For the Web: URL

Representations

- *Resources* are not transferred between components in the architecture, but *representations* of resources
- Representations consists of both data and metadata describing the data
- Resource metadata provide information about the resource not specific to the representation
- Control data provides information about the message, such as for caching

Representations - example

JSON

```
GET /2.30/api/organisationUnits/ImspTQPwCqd?fields=name,id HTTP/1.1
> Host: play.dhis2.org
> Accept: application/json

HTTP/1.1 200
< Content-Type: application/json;charset=UTF-8
{"name":"Sierra Leone","id":"ImspTQPwCqd"}
```

XML

```
GET /2.30/api/organisationUnits/ImspTQPwCqd?fields=name,id HTTP/1.1
> Host: play.dhis2.org
> Accept: application/xml

HTTP/1.1 200
< Content-Type: application/xml;charset=UTF-8
<?xml version='1.0' encoding='UTF-8'?><organisationUnit xmlns="http://dhis2.org/
schema/dxf/2.0" name="Sierra Leone" id="ImspTQPwCqd"/>
```

REST components

Component	Example
origin server	apache, MS IIS
gateway/reverse proxy	squid, cgi, nginx
proxy	
user agent	Chrome, Firefox, curl

REST connectors

Connector	Example
client	libwww, libcurl
server	libwww, Apache API
cache	browser, cache networks
resolver	bind
tunnel	SOCKS

REST connectors

- Connectors handle communication for the components
- Because interactions are stateless and requests self-descriptive:
  - Connectors can handle requests independently and in parallel
  - Intermediaries can understand requests in isolation
  - Information relevant for caching is part of each request

REST in Practice - HTTP

- Anatomy of HTTP requests and responses
- HTTP methods
- Content negotiations
- Status codes

HTTP requests

- HTTP requests consists of header and body
- Body - the data/payload
- Header - different types:
  - General header that can apply to both request and response - Date, Cache-Control
  - Request header - Accept, User-Agent, Referer
  - Response header - Age, Location, Server
  - Entity header is metadata about the body (MIME, content length etc)

```
~>curl google.com -v
* Rebuilt URL to: google.com/
*   Trying 216.58.209.142...
* TCP_NODELAY set
* Connected to google.com (216.58.209.142) port 80 (#0)
> GET / HTTP/1.1
> Host: google.com
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 302 Found
< Cache-Control: private
< Content-Type: text/html; charset=UTF-8
< Referrer-Policy: no-referrer
< Location: http://www.google.no/?gfe_rd=cr&dcr=0&ei=mEu4WbXAL4ir8we1o4a4Dg
< Content-Length: 268
< Date: Tue, 12 Sep 2017 21:03:20 GMT
<
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.no/?gfe_rd=cr&amp;dcr=0&amp;ei=mEu4WbXAL4ir8we1o4a4Dg">here</A>.
</BODY></HTML>
* Connection #0 to host google.com left intact
```

Request header

Response header

Response body

```
curl -X PATCH "https://play.dhis2.org/demo/api/dataElements/FTRrcoaog83" -u admin:district -H
"Content-type: application/json" -d '{"domainType": "BLABLA"}' -vv
*   Trying 52.30.174.183...
* TCP_NODELAY set
* Connected to play.dhis2.org (52.30.174.183) port 443 (#0)
* TLS 1.2 connection using TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
* Server certificate: play.dhis2.org
* Server certificate: RapidSSL SHA256 CA - G3
* Server certificate: GeoTrust Global CA
* Server auth using Basic with user 'admin'
> PATCH /demo/api/dataElements/FTRrcoaog83 HTTP/1.1
> Host: play.dhis2.org
> Authorization: Basic YWRtaW46ZGlzdHJpY3Q=
> User-Agent: curl/7.54.0
> Accept: */*
> Content-type: application/json
> Content-Length: 24
>
* upload completely sent off: 24 out of 24 bytes
< HTTP/1.1 500 Internal Server Error
< Server: nginx/1.4.6 (Ubuntu)
< Date: Tue, 12 Sep 2017 21:15:09 GMT
< Content-Type: application/json;charset=UTF-8
< Content-Length: 408
< Connection: keep-alive
< X-XSS-Protection: 1; mode=block
< X-Frame-Options: SAMEORIGIN
< X-Content-Type-Options: nosniff
< Set-Cookie: JSESSIONID=62886259EE13F8F9A3A9BFFAAA5E8077; Path=/demo/; HttpOnly
< Cache-Control: no-cache, private
<
* Connection #0 to host play.dhis2.org left intact
{"httpStatus":"Internal Server Error","httpStatusCode":500,"status":"ERROR","message":"Can not
construct instance of org.hisp.dhis.dataelement.DataElementDomain from String value (\"BLABLA\"):
value not one of declared Enum instance names: [TRACKER, AGGREGATE]\n at [Source: {\"domainType\":
\"BLABLA\"}; line: 1, column: 16] (through reference chain:
org.hisp.dhis.dataelement.DataElement[\"domainType\"])"}
```

Request header

Response header

Response body

HTTP methods

- GET - request representation of a resource
- POST - create an entity based on the payload (body)
- PUT - update an entity based on the payload
- PATCH - partially update an entity based on the payload
- DELETE - delete the resource
- HEAD, TRACE, OPTIONS, CONNECT

Details: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

HTTP methods

- GET - safe, idempotent, cacheable
- POST
- PUT - idempotent
- PATCH - *can* be idempotent
- DELETE - idempotent
- Idempotent methods can be called multiple times without changing the result/outcome

Content negotiation

- Content negotiation is the process of determining the *representation* of the resource
- Clients specify desired representation through:
  - HTTP header **Accept** field - Accept: application/json
  - URL extension - http://localhost/api/cars**.json**
- If the requested representation is not available the server should:
  - Respond with status code 406 not acceptable
  - Include a list of available representations

HTTP status codes

- HTTP status codes are divided into classes:
  - 1XX - informational
  - 2XX - success
  - 3XX - redirection
  - 4XX - client error
  - 5XX - server error

HTTP status codes

- Each class is extensible with additional codes
- Clients do not need to understand *all* codes
- Unknown codes default to the X00 code (100, 200 etc)

- https://tools.ietf.org/html/rfc7231#section-6

REST and RESTful

- REST is an architectural style
- RESTful web services (or REST APIs) are used to describe web services designed according to the REST architecture style

REST Maturity Model

- Whether a web service is RESTful is not either or is not binary
- Richardson’s maturity model define levels of adherence to the REST architecture:

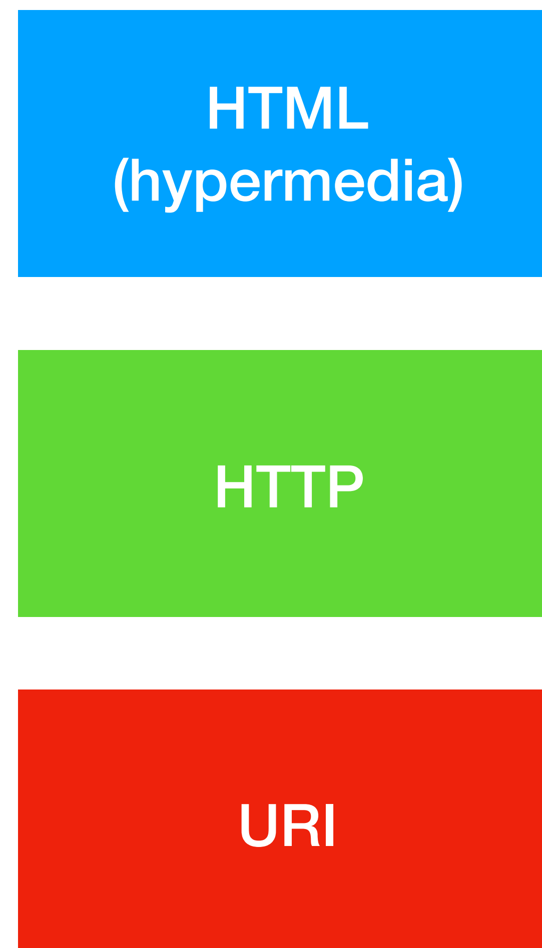
Level 0 - HTTP as a tunnel

Level 1 - Use of multiple identifiers and resources

Level 2 - Use of HTTP verbs

Level 3 - Use of hypermedia to model relationships

REST Maturity Model

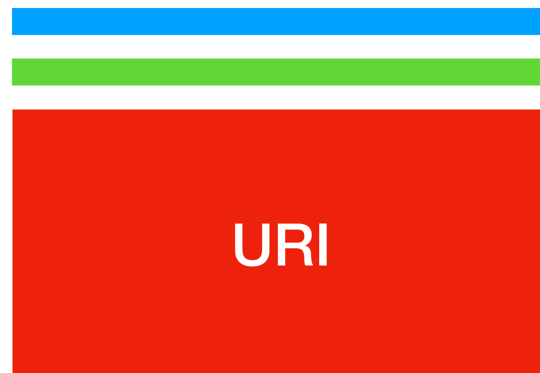


Source: https://www.crummy.com/writing/speaking/2008-QCon/act3.html

Level 0

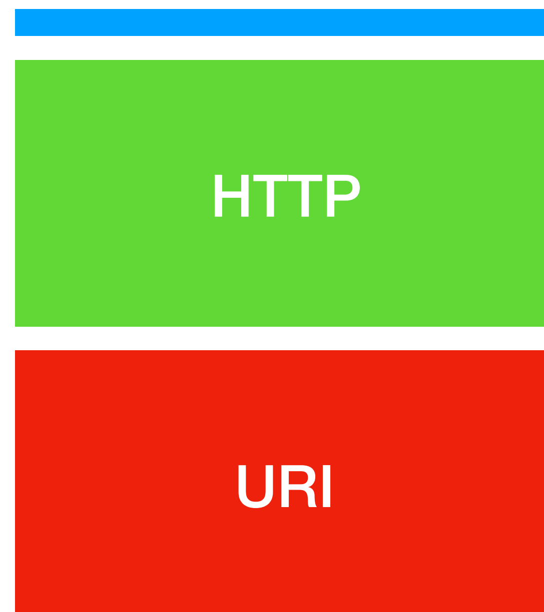
- Single resource/endpoint/URI
- Example: RPC-XML

Level 1



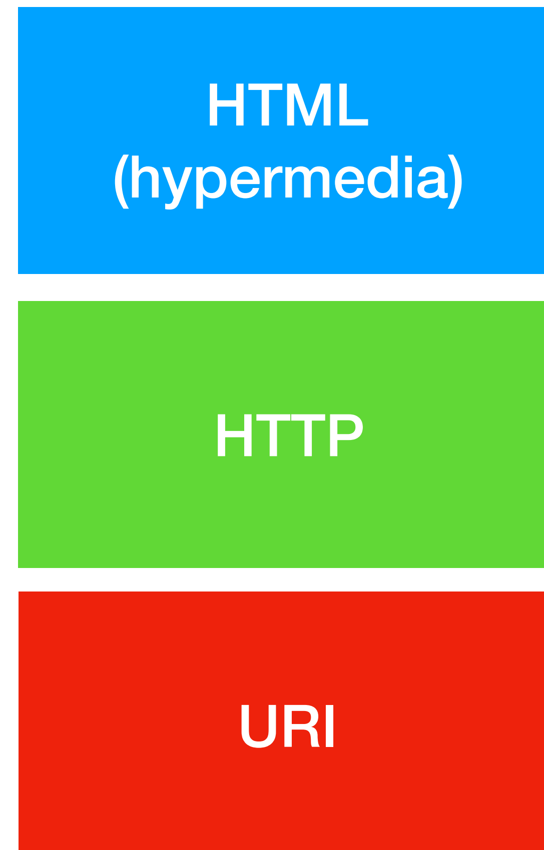
- Multiple endpoints/resources/URIs -"modularisation"
- Limited HTTP methods, e.g. only using POST

Level 2



- Multiple endpoints/resources/URIs -"modularisation"
- Multiple HTTP methods, e.g. GET and POST

Level 3



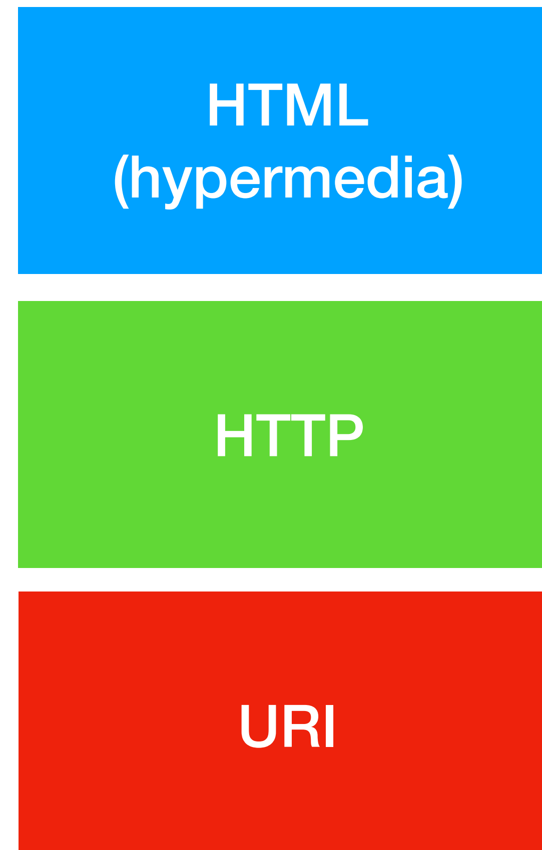
- Resources include information about related resources, i.e. links
- Relation between objects described dynamically by the service rather than in separate documentation

Example - PayPal


```
{
  "links": [{
    "href": "https://api.paypal.com/v1/payments/sale/36C38912MN9658832",
    "rel": "self",
    "method": "GET"
  }, {
    "href": "https://api.paypal.com/v1/payments/sale/36C38912MN9658832/refund",
    "rel": "refund",
    "method": "POST"
  }, {
    "href": "https://api.paypal.com/v1/payments/payment/PAY-5YK922393D847794YKER7MUI",
    "rel": "parent_payment",
    "method": "GET"
  }]
}
```


https://developer.paypal.com/docs/api/overview/#hateoas-links

Level 3



- Resources include information about related resources, i.e. links

- Relation between objects described dynamically by the service rather than in separate documentation

REST Maturity Model

Level 0 - HTTP as a tunnel

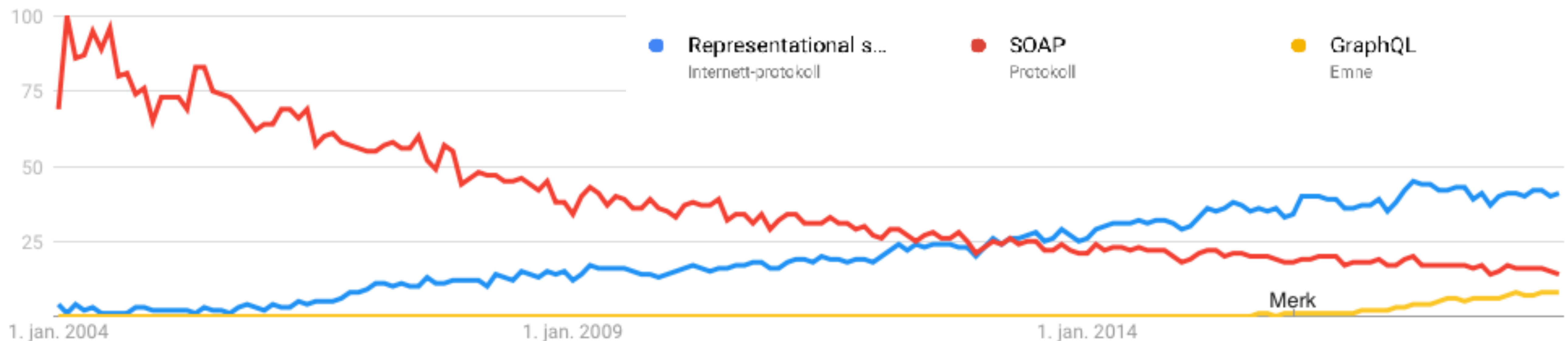
Level 1 - Use of multiple identifiers and resources

Level 2 - Use of HTTP verbs

Level 3 - Use of hypermedia to model relationships

RESTful vs other WS

- RESTful web services make full use of the HTTP protocol
- "Traditional" web services (XML-RPC etc) use HTTP primarily for transport
- GraphQL is gaining popularity as alternative/supplement to REST



"Traditional" web services

- Typically RPC (Remote Procedure Call)-type protocols
- A number of standards, such as:
  - XML-RPC, which evolved into SOAP (Simple Object Access Protocol) - messaging standard
  - WSDL (Web Services Description Language) - XML format for describing/defining the web service
  - Various WS-* standards built on SOAP messaging

"Traditional" web services

- Based on interacting with *services* e.g. through remote procedure calls (RPCs)
- All operations are typically POSTed to one/few endpoint(s)
- Operations to be performed is based on content of SOAP (or similar) message rather than an HTTP verb
- Extensions to SOAP for specific functionality - WS-Security, WS-Policy, WS-Addressing etc

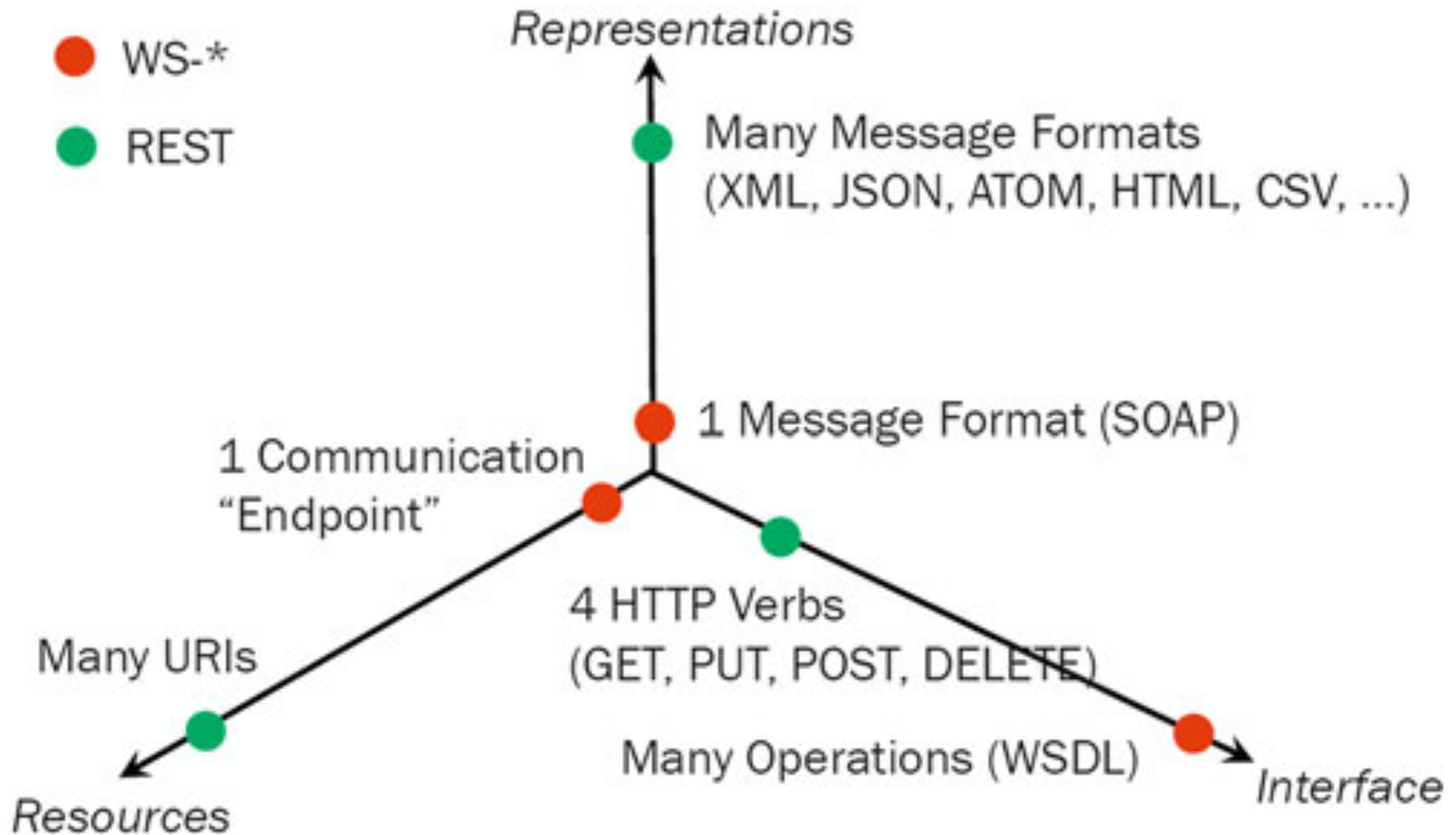
SOAP example

```
POST http://somedomain.com/service
<?xml version="1.0" encoding="utf-8" ?>
<soap:="http://www.w3.org/2003/05/soap-envelope/">
  <soap:Body>
    <FindCustomerByNum xmlns="urn:OrderSvc:OrderInfo">
      <CustomerNumber>3</CustomerNumber>
    </FindCustomerByNum>
  </soap:Body>
</soap:Envelope>
<?xml version="1.0" encoding="UTF-8" ?>
<soap:http://www.w3.org/2003/05/soap-envelope/>
  <soap:Body>
    <FindCustomerByNumResponse xmlns="urn:OrderSvc:OrderInfo">
      <CustomerName>Hoops</CustomerName>
    </FindCustomerByNumResponse>
  </soap:Body>
</soap:Envelope>
```

```
GET http://somedomain.com/api/customers/3
{
    id: 3,
    name: Hoops
}
```

Rest equivalent

RESTful vs other WS



GraphQL

- API query language, Open Sourced in 2015 by Facebook
- Language specifications and runtime backend
- Also supports writing and subscribing to changes
- Clients define data structure of data being requested, in order to:
    - Reduce number of requests
    - Reduce "unneeded" data

GraphQL

```
{
  hero {
    name
    # Queries can have comments!
    friends {
      name
    }
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
```

```
{
  human(id: "1000") {
    name
    height
  }
}
```

```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 1.72
    }
  }
}
```

Source: http://graphql.github.io/learn/queries/

GraphQL vs REST

- Network usage
- Evolvability
- One vs multiple requests
    - RESTful APIs can emulate some GraphQL functionality

Literature

- Curriculum:
  - Fielding and Taylor. 2002.
  - Pautasso. 2014.
  - Fowler. 2010.

- More on REST:
  - Erenkrantz et al 2007. Section 2 gives more concise definition of REST.
  - Fielding et al 2017. Sections 1-2 discusses different definitions of REST over the years.
- More on "Big" web services vs RESTful web services: http://www2008.org/papers/pdf/p805-pautassoA.pdf
- More on GraphQL: https://graphql.org/learn/

Other sources

- Roy Fielding presentations:
  - https://www.slideshare.net/royfielding/a-little-rest-and-relaxation
  - https://www.slideshare.net/royfielding/rest-in-aem?next_slideshow=1