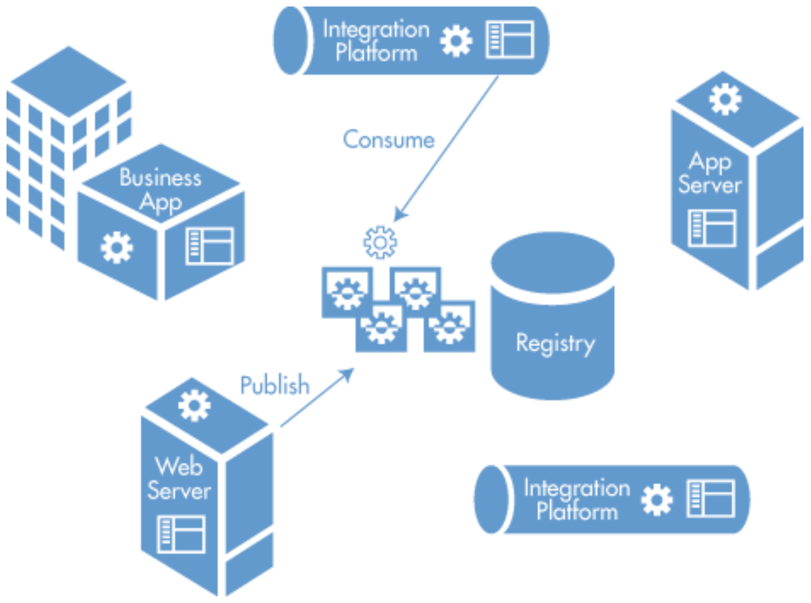
UNIVERSITÀ DI PARMA

Service-Oriented Architectures

RESTful Services



Tecnologie Internet

a.a. 2022/2023

REST

REST stands for **Re**presentational **S**tate **T**ransfer.

REST relies on a stateless, client-server, cacheable communications protocol.

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 an 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.

Roy Fielding. PhD thesis, 2000

Any state management tasks must be performed by the client.

REST

Roy Fielding has never mentioned any recommendation around which method to be used in which condition.

However, rather than using complex protocols like SOAP to connect between machines, simple **HTTP** is most frequently used to make calls between machines, with standard Web servers (e.g., Apache httpd) playing the part of REST servers.

RESTful applications may use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data.

Thus, RESTful applications may use HTTP for all four **CRUD** (Create/Read/Update/Delete) operations.

REST

REST is a lightweight solution for implementing services, alternative to Web Service technologies (SOAP, WSDL, etc.).

- Despite being simple, REST is fully-featured; there's basically nothing you can do in Web Services that cannot be done with a RESTful architecture.

- REST has no "standard specification". There will never be a W3C recommendation for REST, because it is an **architectural style**, not a protocol.

REST architecture components

- **Resources** are the key element of a true RESTful design. Individual resources are identified in requests, for example using URLs in web-based REST systems. The resources themselves are conceptually separate from the **representations** that are returned to the client. For example, the server may send data from its database as XML or JSON documents, none of which reflecting the server's internal representation.

- **A web of resources**, meaning that a single resource should not be overwhelmingly large and contain too fine-grained details. Whenever relevant, a resource should contain *links* to additional information -- just as in web pages.

REST architecture components

- There is **no connection state**; interaction is stateless (although the servers and resources can of course be stateful). Each new request should carry all the information required to complete it, and must not rely on previous interactions with the same client.

- Responses should be **cacheable** whenever possible (with an expiration date/time). The protocol must allow the server to explicitly specify which responses may be cached, and for how long.
  - HTTP cache-control headers are used for this purpose.
  - Clients must respect the server's cache specification for each resource.

- **Proxy servers** can be used as part of the architecture, to improve performance and scalability. Any standard HTTP proxy can be used.

RESTful services

Much like a Web Service, a RESTful service is:

- **platform-independent** (you don't care if the server is Unix, the client is a Mac, or anything else);
- **language-independent** (C# can talk to Java, etc.);
- **standards-based** (may run on top of HTTP, CoAP, etc.);
- can easily be used in the presence of firewalls.

Like Web Services, REST offers **no built-in security features, encryption, session management, QoS guarantees, etc.** But also as with XML-based services, these can be added by building on top of HTTP:

- for security, username/password tokens are often used;
- for encryption, REST can be used on top of HTTPS (secure sockets);
- etc.

Example: phonebook application

The query will probably look like this:

http://www.acme.com/phonebook/user-details/12345

Note how the URL's "method" part is not called "get-user-details", but simply "user-details". It is a common convention in REST design to use *nouns* rather than *verbs* to denote simple *resources*.

It is just a URI, which is sent to the server using a simple HTTP GET request. The HTTP reply is the raw result data - not embedded inside anything, just the data you need in a way you can directly use.

- With REST, a simple network connection is all you need. You can even test the API directly, using your browser.
- Still, REST libraries (for simplifying things) do exist.

More complex REST requests

REST can easily handle more complex requests, including multiple parameters. In most cases, you'll just use HTTP GET parameters in the URL. For example:

`http://www.acme.com/phonebook/user-details?name=John&surname=Doe`

If you need to pass long parameters, or binary ones, you'd normally use HTTP POST requests, and include the parameters in the POST body.

REST *requests* rarely use XML. As shown above, in most cases, request parameters are simple, and there is no need for the overhead of XML.

- One advantage of using XML is type safety. However, you should *always* verify the validity of your input, XML or otherwise!

REST server responses

A server response in REST is often an XML file. However, other formats can also be used; unlike SOAP services, REST is *not* bound to XML in any way.

Possible formats include **CSV** (comma-separated values) and **JSON** (JavaScript Object Notation).

Each format has its own advantages and disadvantages. XML is easy to expand (clients should ignore unfamiliar fields) and is type-safe; CSV is more compact; and JSON is trivial to parse in JavaScript clients (and easy to parse in other languages, too).

REST server responses

One option is not acceptable as a REST response format, except in very specific cases:

HTML, or any other format which is meant for human consumption and is not easily processed by clients.

The specific exception is, of course, when the REST service is documented to return a human-readable document; and when viewing the entire WWW as a RESTful application, we find that HTML is in fact the most common REST response format.

Routes vs. Endoints

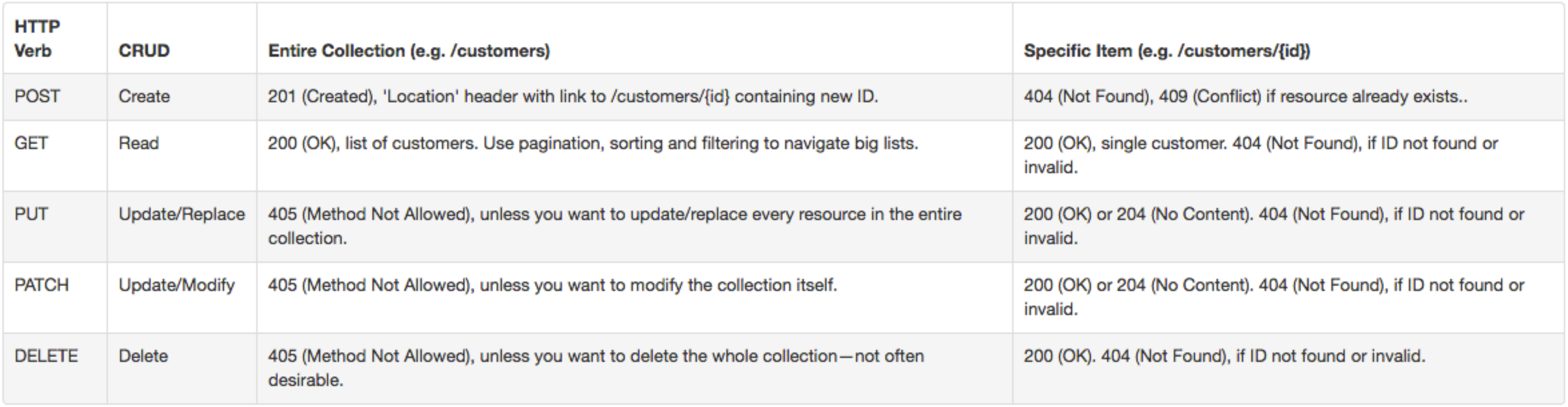
Endpoints are functions available through the API.

A **route** is the “name” you use to access endpoints, used in the URL. A route can have multiple endpoints associated with it, and which is used depends on the specified verb.

Example with the URL http://example.com/wp-json/wp/v2/posts/123

- The route is wp/v2/posts/123 – The route doesn’t include wp-json because wp-json is the base path for the API itself.
- This route has 3 endpoints:
    - GET triggers a get_item method, returning the post data to the client.
    - PUT triggers an update_item method, taking the data to update, and returning the updated post data.
    - DELETE triggers a delete_item method, returning the now-deleted post data to the client.

REST design guidelines



HTTP Verb	CRUD	Entire Collection (e.g. /customers)	Specific Item (e.g. /customers/{id})
POST	Create	201 (Created), 'Location' header with link to /customers/{id} containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists..
GET	Read	200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single customer. 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
PATCH	Update/Modify	405 (Method Not Allowed), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable.	200 (OK). 404 (Not Found), if ID not found or invalid.

http://www.restapitutorial.com/lessons/httpmethods.html

REST design guidelines

1. Do not use "physical" URLs.
A physical URL points at something physical - e.g., an XML file: "http://www.acme.com/inventory/product003.xml".
A *logical* URL does not imply a physical file: "http://www.acme.com/inventory/products/003".
Sure, even with the .xml extension, the content could be dynamically generated. But it should be "humanly visible" that the URL is logical and not physical.

2. Queries should not return an overload of data.
If needed, provide a paging mechanism. For example, a "product list" GET request should return the first *n* products (e.g., the first 10), with next/prev links.

REST design guidelines

3. Even though the REST response can be anything, make sure it is well documented, and do not change the output format lightly (since it will break existing clients).

- Remember, even if the output is human-readable, your clients are not human users.
- If the output is in XML, make sure you document it with a schema or a DTD.

4. Rather than letting clients construct URLs for additional actions, include the actual URLs with REST responses.
For example, a "product list" request could return an ID per product, and the specification says that you should use http://www.acme.com/products/(product-id) to get additional details. That's bad design. Rather, the response should include the actual URL with each item: http://www.acme.com/products/001263, etc.

Famous REST APIs

- Twitter: https://developer.twitter.com/en/docs
- Facebook: https://developers.facebook.com/docs/graph-api
- Amazon Simple Storage Service (S3): http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
- Google Developers API: https://developers.google.com/oauthplayground/
- YouTube API: https://developers.google.com/youtube/v3/docs/
- WordPress API: https://developer.wordpress.org/reference/
- Dropbox API: https://www.dropbox.com/developers

Consuming a RESTful service with jQuery

Suppose there is a service accepting requests at

```
http://rest-service.guides.spring.io/greeting
```

The service will respond with a JSON representation of a greeting:

```
{"id":1,"content":"Hello, World!"}
```

Consuming a RESTful service with jQuery

jQuery controller for consuming the REST service:

```
public/hello.js

$(document).ready(function() {
    $.ajax({
        url: "http://rest-service.guides.spring.io/greeting"
    }).then(function(data) {
       $('.greeting-id').append(data.id);
       $('.greeting-content').append(data.content);
    });
});
```

Documentation for jQuery ajax() method:

https://www.w3schools.com/jquery/ajax_ajax.asp

Consuming a RESTful service with jQuery

HTML page that will load the client:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello jQuery</title>
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.1/
jquery.min.js"></script>
        <script src="hello.js"></script>
    </head>

    <body>
        <div>
            <p class="greeting-id">The ID is </p>
            <p class="greeting-content">The content is </p>
        </div>
    </body>
</html>
```

Express

http://expressjs.com/

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

Express provides a thin layer of fundamental web application features, without obscuring Node.js features that you know and love.

Creating a robust API with Express is quick and easy.

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

References

R. T. Fielding
Architectural Styles and the Design of Network-based Software Architectures
Ph.D. Thesis, University of California, Irvine, 2000.

http://www.restapitutorial.com/lessons/httpmethods.html