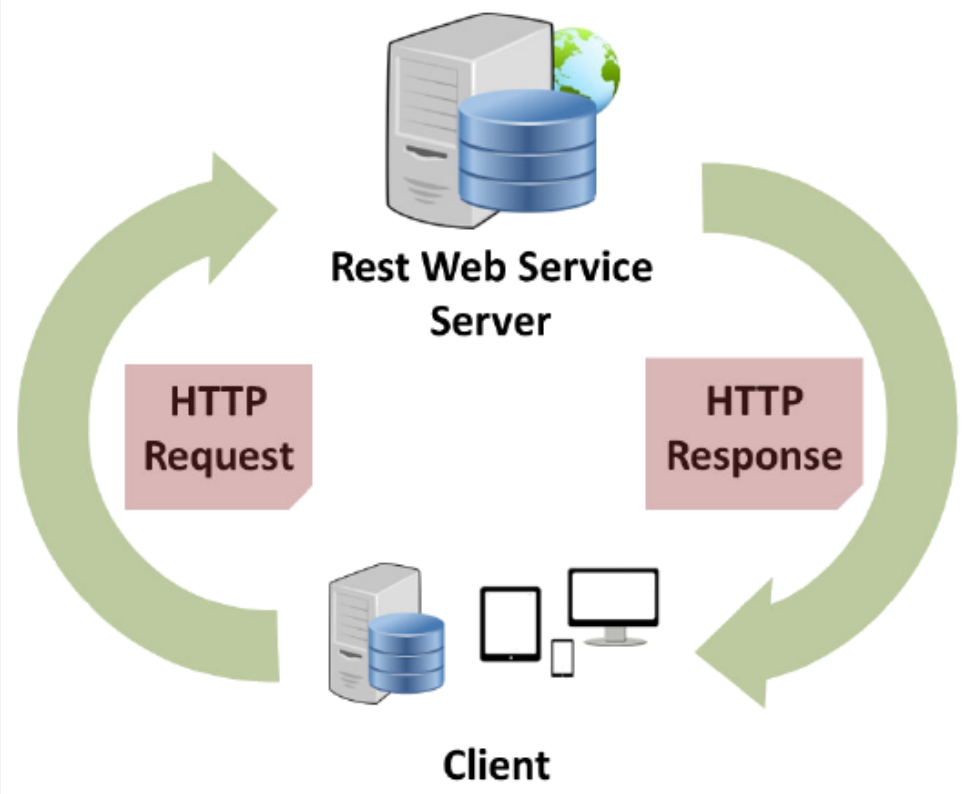
P.PORTO

POLITÉCNICO
DO PORTO
**ESMAD**

PROGRAMAÇÃO WEB II
**TSIW**

# SUMMARY

- Web Services
- REST APIs



# Web Services

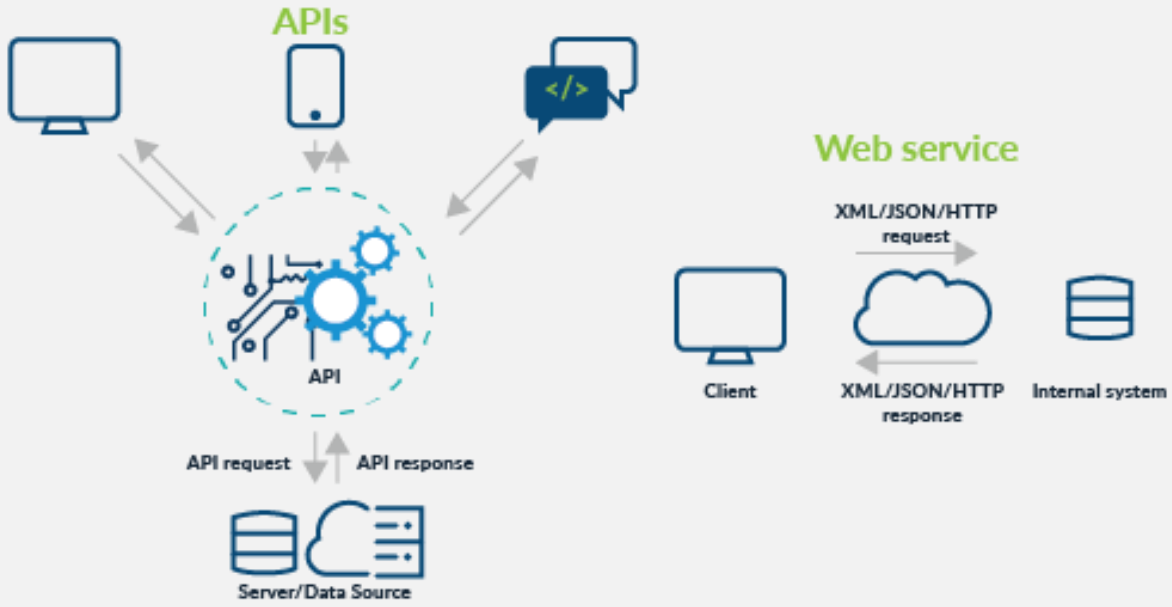
- Up to late 90's, the dominant method for delivering web applications was for **web servers to send HTML, CSS, JavaScript**, and multimedia assets **in response to every request**
  - Browsers mitigated some of this huge overhead cost by caching assets, but the pace of innovation in web applications was straining this model
- Web 2.0 arose, people were expected rich web experiences, so developers realized that **not every change in an application required information from the server**, and **not every change that required information from the server needed the entire application just to deliver a small change**
  - the page itself might change in response to user interaction, and instead of asking the server for a whole new application, **JavaScript would change the DOM directly**

# Web Services

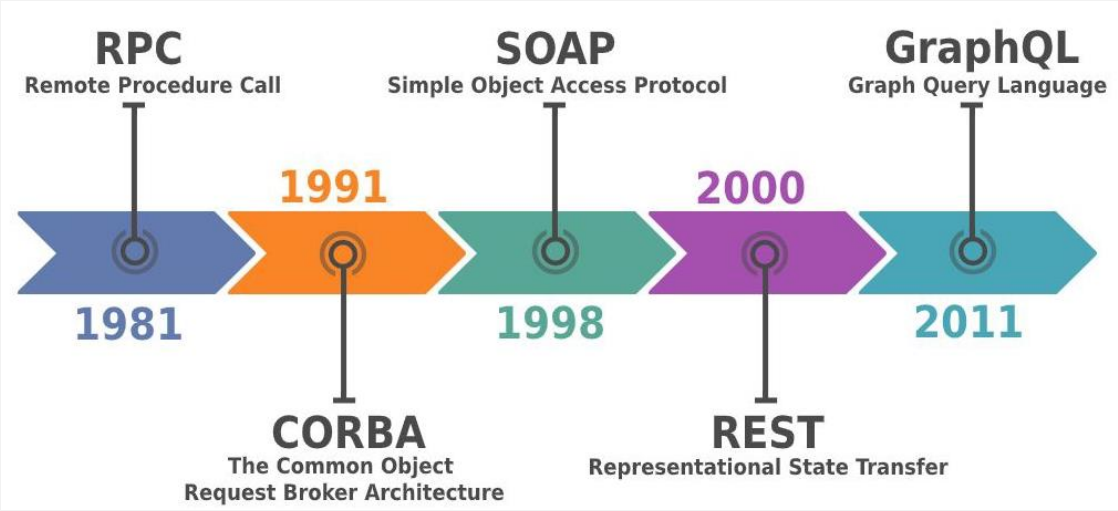
- By 2012, it was common practice to try to send as little information as possible over the network, and do as much as possible in the browser, providing the conditions for the SPA's (**Single-Page Application** techniques)
  - for any given web application, the HTML, JS, and CSS (if any) are shipped exactly once; then, it is up to the JS to make all changes to the DOM to make the user feel that they are navigating to a different page
  - of course, the server is still involved: it's still responsible for providing up-to-date data, and being the "single source of truth" in a multiuser application
- So, most modern web applications rely on receiving unstructured data (like JSON or XML), **communicating directly using HTTP requests to a Web Service or API**

# Web Services / APIs

- They are around for several years, and can be defined as a service offered by an electronic device to another, communicating with each other via the WWW
  - Meaning, it is a Web-based interface to a database server, that uses HTTP protocol to communicate using machine-readable file formats such as XML and JSON



# REST




- Before REST, protocols like RPC or SOAP were a standard when developing Web APIs
- Look at this video, that compares the 4 most common API types: RPC, SOAP, REST and GraphQL

# REST

## API ARCHITECTURAL STYLES

| | RPC | SOAP | REST | GraphQL |
|---|---|---|---|---|
| Organized in terms of | local procedure calling | enveloped message structure | compliance with six architectural constraints | schema & type system |
| Format | JSON, XML, Protobuf, Thrift, FlatBuffers | XML only | XML, JSON, HTML, plain text, | JSON |
| Learning curve | Easy | Difficult | Easy | Medium |
| Community | Large | Small | Large | Growing |
| Use cases | Command and action-oriented APIs; internal high performance communication in massive micro-services systems | Payment gateways, identity management CRM solutions financial and telecommunication services, legacy system support | Public APIs simple resource-driven apps | Mobile APIs, complex systems, micro-services |

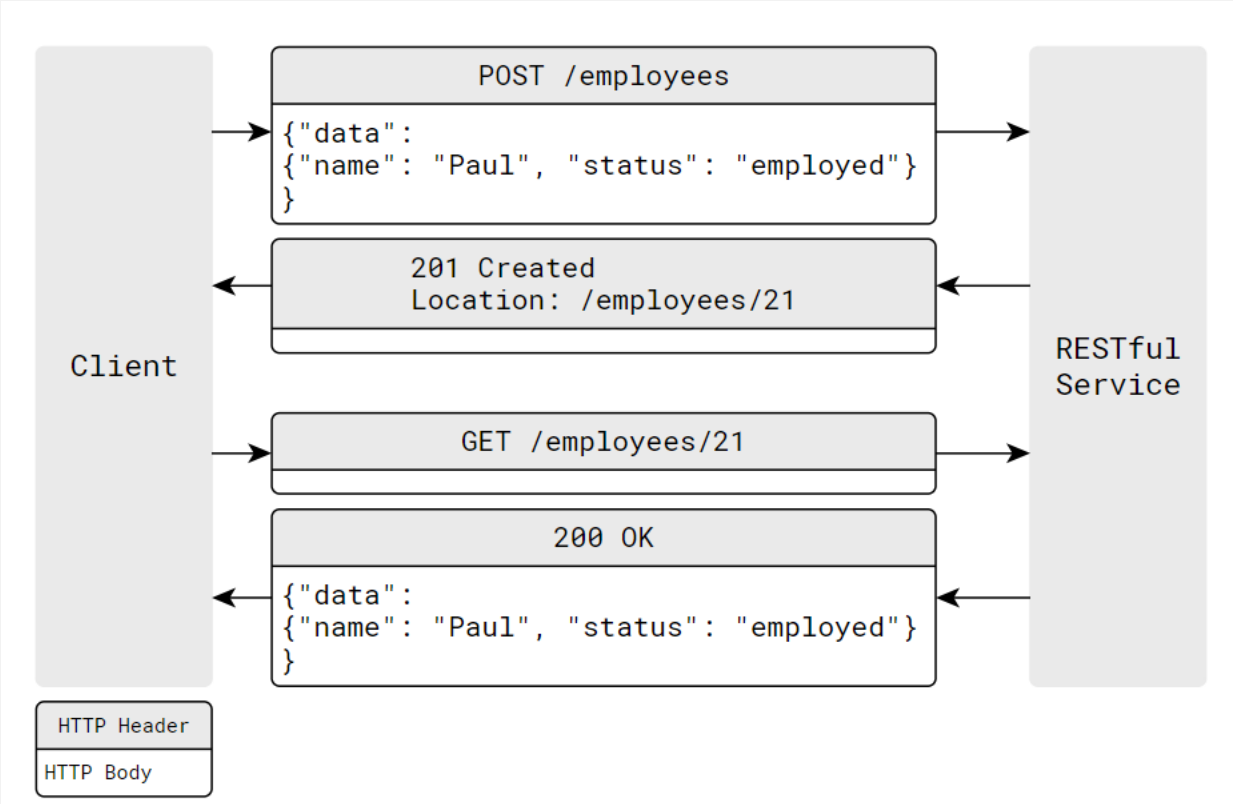
altexsoft
software r&d engineering

# REST


- **REST** stands for REpresentational State Transfer
  - When called, the server transfers to the client a representation of the state of the requested resource
- It is a set of architectural principles for distributed computer systems on the web
  - Roy Fielding first presented it in 2000 in his famous dissertation
- It revolves around resources (data the API wants to expose) where resources are accessed by endpoints (URIs used in the HTTP requests)
  - **Resources** in REST is like objects in Object Oriented Programming or entities in a Database
    - In Instagram's API, for example, a resource can be a user, a photo, a hashtag
  - **URIs** are used to access those resources
  - **HTTP verbs** are used to perform actions on those resources

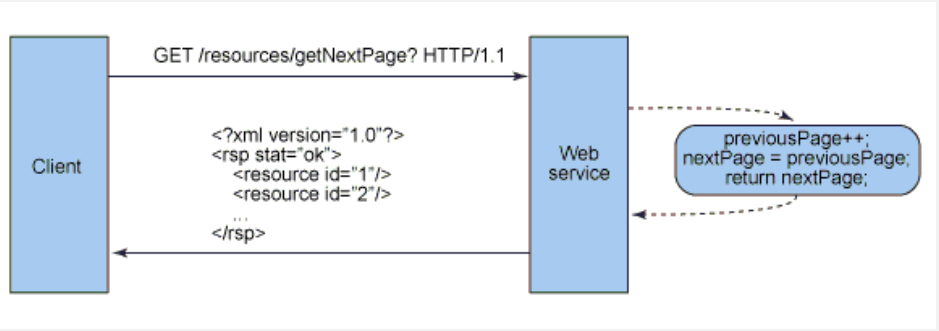
# REST

- A Web API conforming to the REST architectural style can be named as RESTfull

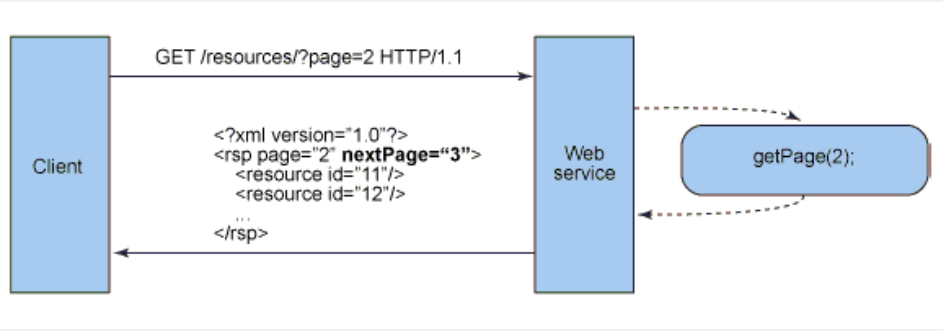


# REST principles

- **Client-server**: clear boundaries between roles of the two systems; one must function as the server that is being called, and the other is the one making the requests
- **Stateless**: each request is self-descriptive, and has enough context for the server to process that message
  - meaning no client content is stored on the server between requests




Not stateless – client state stored on the server




Stateless – client state stored on the client

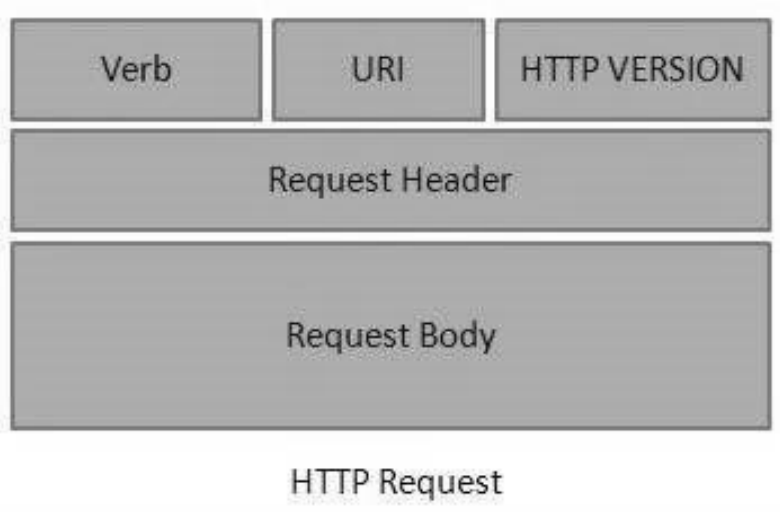
# REST principles

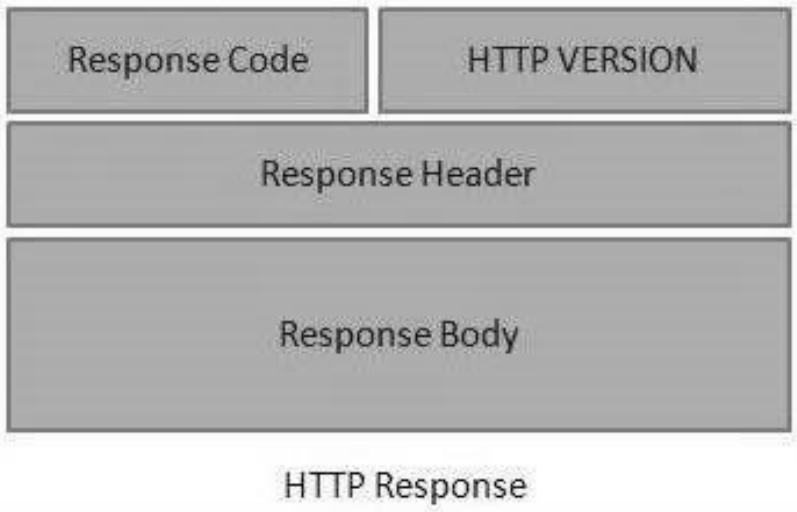
- **Uniform interface**: information is transferred in a standardized form instead of specific to an application's needs; the following 4 constraints can achieve a uniform REST interface:
  1. **Identification of resources** – a unique identifier (URI) assigned to each resource
  2. **Manipulation of resources through representations** – the response the server returns include enough information so the client can modify the resource
  3. **Self-descriptive messages** – each request contains all the information the server needs to perform the request, and each response contains all the information the client needs to understand it
  4. **Hypermedia as the engine of application state** – the server can inform the client , in a response, of the ways to change the state of the application; the client application should dynamically drive all other resources and interactions with the use of hyperlinks

# REST principles

- **Cacheable**: unless denoted, a client can cache any representation
  - REST APIs allow clients to store frequently accessed data on their side instead of requesting them again and again
  - this is possible thanks to the statelessness, since every representation is self-descriptive
- **Layered system**: client-server interactions can be mediated by hierarchical layers
  - For example, one can keep the API in a server, the database in another server and authentication system in separate server
  - Client does not know the numbers of servers that responses come through
- **Code on demand**: (optional) the server may return a piece of executable code on the client's demand

# REST messages





# REST resources

- REST does not impose any restriction on the **format** of a resource representation
  - A client can ask for JSON representation whereas another client may ask for XML representation of the same resource to the server
  - It is the responsibility of the REST server to pass the client the resource in the format that the client understands
- At present most of the web services are representing resources using either XML or JSON format; there are plenty of libraries and tools available to understand, parse, and modify XML and JSON data

```xml
<user>
   <id>1</id>
   <name>Mahesh</name>
   <profession>Teacher</profession>
</user>
```

Resource **user** in XML and JSON format

```json
{
   "id":1,
   "name":"Mahesh",
   "profession":"Teacher"
}
```

# REST API URIs

**REST API designers should create URIs that convey a REST API's resource model** to its potential client developers

- RULE #1: forward slash separator (/) must be used to indicate a hierarchical relationship between resources

  http://api.example.com/**shapes/polygons/quadrilaterals/squares**

- RULE #2: a trailing forward slash (/) should not be included in URIs
  - as the last character within a URI's path, a forward slash (/) adds no semantic value and may cause confusion

  http://api.example.com/**shapes/** **NOT**
  http://api.example.com/**shapes**

# REST API URIs

- RULE #3: hyphens (-) should be used to improve the readability of URIs
  - use the hyphen (-) character to improve the readability of names in long path segments
  - underscores (_) should not be used in URIs

    http://api.example.com/blogs/guy-levin/posts/**this_is_my_first_post** **NOT**
    http://api.example.com/blogs/guy-levin/posts/**this-is-my-first-post**

- RULE #4: lowercase letters should be preferred in URI paths

    http://api.example.com/my-folder/my-doc (1)
    HTTP://API.EXAMPLE.COM/my-folder/my-doc (2)
    http://api.example.com/My-Folder/my-doc (3)

    (1) and (2) can be considered equal; (3) is not the same as (1) or (2)

# REST API URIs

- RULE #5: file extensions should not be included in URIs
  - REST APIs should not include artificial file extensions in URIs to indicate the format of a message's entity body
  - Instead, they should rely on the media type, as communicated through the *Content-Type* header, to determine how to process the body's content
  - REST API clients should be encouraged to utilize HTTP's provided format selection mechanism, the *Accept* request header

    http://api.college.com/students/3248234/transcripts/2005/**fall.json** **NOT**
    http://api.college.com/students/3248234/transcripts/2005/**fall**

# REST API URIs

- RULE #6: use consistent subdomain names
  - full domain name of an API should add a subdomain named **api**
  - many REST APIs have an associated website, known as a developer portal, to help onboard new clients with documentation, forums, and self-service provisioning of secure API access keys; if an API provides a developer portal, by convention it should have a subdomain labeled **developer**

| | |
|---|---|
| PRODUCTION API | http://**api**.fakecompany.com |
| DEVELOPER PORTAL | http://**developer**.fakecompany.com |
| TEST API | http://**api.sandbox**.fakecompany.com |

# REST API URIs

- RULE #7: document versus collection
  - **a document resource is a singular concept** that is akin to an object instance or database record
  - a collection resource is a server-managed directory of resources; **always use plural names when referring to a collection resource**

| | |
|---|---|
| http://api.example.com/**students/** | COLLECTION |
| http://api.example.com/**students/mary** | DOCUMENT |
| | |
| http://api.example.com/**users/** | COLLECTION |
| http://api.example.com/**users/007** | DOCUMENT |
| http://api.example.com/**user/007** | **NOT** |

# REST API URIs

- RULE #8: a controller resource models a procedural concept
  - controller resources are like executable functions, with parameters and return values, inputs and outputs
  - a REST API relies on controller resources to perform **application-specific actions** that cannot be logically mapped to one of the standard methods (create, retrieve, update, and delete)
  - controller names are **verbs** and typically appear as the last segment in a URI path, with no child resources to follow them in the hierarchy

**POST** /alerts/245743/**resend**
controller resource that allows a client to resend an alert to user 245743
**POST** /students/**login**
URI to *login* a student (providing its credentials in the body request)

# REST API URIs

- RULE #9: variable path segments may be substituted with identity-based values
  - Static URI path segments: fixed names chosen by the REST API designer
  - Variable URI path segments: they are **automatically filled in with some identifier** that may help provide the URI with its uniqueness

http://api.soccer.com/leagues/**{leagueId}**/teams/**{teamId}**/players/**{playerId}**
this URI template example below has three variables: *leagueId*, *teamId*, and *playerId*

Client URI example:
http://api.soccer.com/leagues/**champions**/teams/**porto**/players/**27**

# REST API URIs

- RULE #10: CRUD function names should not be used in URIs
  - CRUD: Create, Read, Update, Delete—the four standard, storage-oriented functions
  - Instead, HTTP request methods should be used to indicate which CRUD function is performed
  - URIs should be used to uniquely identify resources

**DELETE** /users/1234

GET /deleteUser?id=1234 **NOT**
GET /deleteUser/1234 **NOT**
DELETE /deleteUser/1234 **NOT**

# REST API URIs

- RULE #11: the **query** component of a URI may be used to filter collections or get partial results

URI = "/" path [ "?" query ]

GET /users
The response message lists all the users in the collection
GET /users**?role=admin**
The response contains a **filtered list** of all the users with a "role" value of *admin*

GET /users/007?**fields**=firstname,name,address(street)
When performance is a strong concern, allow **retrieval of only partial data**

GET /orders?state=payed&id_user=007
GET /users/007/orders?state=payed
Multiple URIs can reference the same resource

# REST API URIs

- RULE #12: the **query** component of a URI should be used to **paginate** collection or **order** results

  GET /users?**limit=25&offset=2**
  the *limit* parameter specifies the maximum number of elements to return in the response (one page)
  the *offset* parameter specifies the page offset (usually zero-based)

  GET /restaurants?**sort=rating**

# REST & HTTP verbs

- Each HTTP method has specific, well-defined semantics within the context of a REST API's resource model:
  - GET: retrieves a representation of a resource's
  - PUT: updates a resource (can also create a resource if client specifies which resource it is to store)
  - POST: creates a new resource
  - DELETE: removes a resource
  - PATCH: partially updates a resource
  - HEAD: retrieve the metadata associated with the resource
  - OPTIONS: retrieve metadata that describes a resource's available interactions

# REST & HTTP verbs

| HTTP verb | Collection: /orders | Document: /orders/{id} |
|---|---|---|
| **GET** | Read all orders | Read a specific order |
| **POST** | Creates a new order | - |
| **PUT** | - | Complete update<br>Creates a specific order |
| **PATCH** | - | Partial update |
| **DELETE** | - | Deletes a specific order |

| | | |
|---|---|---|
| **REQUEST** | **GET /orders** | **GET /orders/1234** |
| RESPONSE STATUS CODE | 200 OK | 200 OK |
| RESPONSE BODY | [{"id":"1234", "state":"paid"}<br>{"id":"5678", "state":"running"}] | {"id":"1234", "state":"paid"} |

# REST & HTTP verbs

| HTTP verb | Collection: /orders | Document: /orders/{id} |
|---|---|---|
| **GET** | Read all orders | Read a specific order |
| **POST** | Creates a new order | - |
| **PUT** | - | Complete update<br>Creates a specific order |
| **PATCH** | - | Partial update |
| **DELETE** | - | Deletes a specific order |

**REQUEST URI** **POST /orders**
REQUEST BODY **{"state":"running", "id_user":"007"}**

RESPONSE STATUS CODE 201 Created
RESPONSE BODY {"location":
"https://api.fakecompany.com/orders/1234"}

# REST & HTTP verbs

| HTTP verb | Collection: /orders | Document: /orders/{id} |
|---|---|---|
| **GET** | Read all orders | Read a specific order |
| **POST** | Creates a new order | - |
| **PUT** | - | Complete update<br>Creates a specific order |
| **PATCH** | - | Partial update |
| **DELETE** | - | Deletes a specific order |

| | | |
|---|---|---|
| **REQUEST URI** | **PUT /orders/1234** | **PATCH /orders/1234** |
| REQUEST BODY | **{"state":"paid",<br>"id_user":"007"}** | **{"state":"running“}** |
| RESPONSE STATUS CODE | 200 OK | 200 OK |

# REST & HTTP response status codes

- HTTP defines forty standard status codes that can be used to convey the results of a client's request
- The status codes are divided into the five categories:

| Category | Description |
|---|---|
| 1xx: Informational | Communicates transfer protocol-level information. |
| 2xx: Success | Indicates that the client's request was accepted successfully. |
| 3xx: Redirection | Indicates that the client must take some additional action in order to complete their request. |
| 4xx: Client Error | This category of error status codes points the finger at clients. |
| 5xx: Server Error | The server takes responsibility for these error status codes. |

# REST & HTTP response status codes

- Do not try to use all of them, use the principal 12

| Category | HTTP status code | Description |
|---|---|---|
| Success | 200 OK | Basic success code. Works for most cases. Especially used in the success of the first GET request, or update with PUT / PATCH. |
| | 201 Created | Indicates that the resource has been created. Typical response to a PUT or POST request. |
| | 202 Accepted | Indicates that the request has been accepted for processing. Typical response to a call for asynchronous processing (for better UX and good performance). |
| | 204 No content | The request worked, but there is no content to return. Common response to a successful DELETE. |
| | 206 Partial content | The returned resource is incomplete. Usually used in paged resources. |

# REST & HTTP response status codes

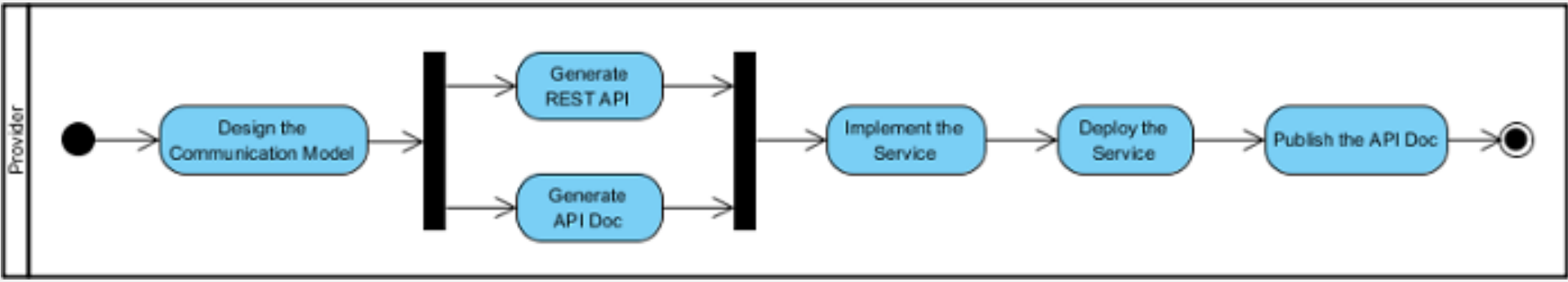
- Do not try to use all of them, use the principal 12

| Category | HTTP status code | Description |
|---|---|---|
| Client error | 400 Bad request | General error for any request<br>GET /users?payed=1<br>< 400 Bad Request<br>< {"error": "invalid_request", "error_description": “There is no 'payed' property on users."} |
| | 401 Unauthorized | Sent when the client either provided invalid credentials or forgot to send them<br>GET /users/42/orders<br>< 401 Unauthorized<br>< {"error": "no_credentials", "error_description": "This resource is under permission, you must be authenticated with the right rights to have access to it"} |
| | 403 Forbidden | Sent to deny access to a protected resource<br>GET /users/42/orders<br>< 403 Forbidden<br>< {"error": "not_allowed", "error_description": "You’re not allowed to perform this request"} |

# REST & HTTP response status codes

| Category | HTTP status code | Description |
|---|---|---|
| Client error | 404 Not Found | Sent when the client tried to interact with a URI that the REST API could not map to a resource<br>GET /users/999999<br>< 400 Not Found<br>< {"error": "not_found", "error_description": "The user with the id '999999' doesn't exist"} |
| | 405 Method not allowed | Sent when the client tried to interact using an unsupported HTTP method (e.g., POST on read-only resource) |
| | 406 Not Acceptable | Sent when the client tried to request data in an unsupported media type format<br>GET /users Accept: text/xml<br>< 406 Not Acceptable<br>< Content-Type: application/json<br>< {"error": "not_acceptable", "available_format":"json"} |
| Server error | 500 Internal server Error | Should be used to indicate API malfunction<br>GET /users<br>< 500 Internal server error<br>< {"error":"server_error", "error_description":"Oops! Something went wrong... "} |

# REST API – design



# REST API – design

- Start by defining the resources and designing the URLs mapped with the business operations

| RESTful URL | HTTP Action | Noun | Business Operation |
|---|---|---|---|
| /Accounts/Profiles/; <profileData> | POST | Profile | createAccountHolderProfile |
| /Accounts/Profiles/{profile_id} | GET | Profile | getAccountHolderProfile |
| /Accounts/Profiles/{profile_id};< profileData> | PUT | Profile | updateAccountHolderProfile |
| /Accounts/{acc_id} | GET | Account | getAccountSummary |
| /Accounts/Loans/ | GET | Loan | getLoanAccounts |
| /Accounts/ | GET | Account | getAllAccounts |
| /Accounts/Bills/; <BillData> | POST | BILL | billPayment |
| /Accounts/Payments/{paymentId} | DELETE | Payment | cancelPayment |
| /Accounts/Payees/ ;<payee data> | POST | Payee | addPayee |
| /Accounts/Payees/{payee_id};<payee data> | PUT | Payee | updatePayee |
| /Accounts/Payee/{payee_id} | DELETE | Payee | deletePayee |
| /Accounts/fd;<FD Data> | POST | FD | createFixedDeposit |
| /Accounts/fd{fd_id};<FD Data> | PUT | FD | preCloserFixedDeposit |

Best Practices for Building RESTful Web services (infosys.com)

# REST API – documentation

- Whether it's internal or external API consumers, they'll want to know about authentication, the endpoints, and what response data to expect
- Checklist:
  - The root path for this version of your API
  - The path to call each endpoint
  - Which HTTP methods can be used with each endpoint
  - Authentication and other headers required with each request
  - The request data fields and where each goes, such as path, query-string, or body
  - Explanation of what request data is required and what is optional
  - Which HTTP status codes are possible for each endpoint/method pairing
  - What each status code means in the context of each call
  - The data to expect in each response, including which responses will always be present
  - Example request and response data

# REST API – documentation

## Title

<Additional information about your API call. Try to use verbs that match both request type (fetching vs modifying) and plurality (one vs multiple).>

- URL <The URL Structure (path only, no root url)>
- Method <The request type>
- Request Params: path, body, query, header <If URL params exist (path or query), specify them in accordance with name mentioned in URL section. If making a post request, what should the body payload look like? Header parameters are included in the request header; they are usually related to authorization. Separate into optional and required. Should document data constraints.>
- Response: success and error responses <Success: what should the status code be on success and is there any returned data? This is useful when people need to to know what their callbacks should expect! Error: most endpoints will have many ways they can fail. From unauthorized access, to wrongful parameters etc. All of those should be listed here. It might seem repetitive, but it helps prevent assumptions from being made where they should be>
- Sample call and sample responses <Just a sample call to your endpoint in a runnable format ($.ajax call or a curl request) - this makes life easier and more predictable.>
- Notes <This is where all uncertainties, commentary, discussion etc. can go>

# REST API – documentation

Show user Returns JSON data about a single user. Accessible only by authenticated users.

- URL /users/:id
- Method GET
- Request Params: PATH id=[integer] (required) HEAD token=[string] (required)
- Response:

  Success: Code: 200 OK
  Content: { id : 12, name : "Michael Bloom" }
  Error: Code: 404 NOT FOUND
  Content: { error : "User doesn't exist" }
  Error: Code: 401 UNAUTHORIZED
  Content: { error : "You are unauthorized to make this request." }

- Sample call

```
curl -X 'GET' \
  'https://fakeAPI/v1/users/1' \
  -H 'accept: application/json'
  -H "Authorization: {token}"
```

# REST API – example

Consider an application to manage books in a library. Let's point out some possible requirements (feel free to add more!):

- The application provides some public and protected data. Users need to log in to have access to protected data. There are two types of users: regular users and the librarian (acts as a system administrator)
- New (regular) users can create an account to use the application
- The list of books is available publicly (searchable by author name, title,...)
- A regular user can:
  - see all information about himself
  - can modify its data
  - borrow a book / see all it’s borrows (but cannot update or delete them)
  - see the borrow historic of a book
- A librarian has access to all data and can:
  - modify/delete books (only those without borrows)
  - see all borrows and modify borrows status (can not delete them)

# REST API – example

What about the resources? First, it is necessary to define them, and then map the necessary actions to them:

- **Users** – standard actions like create, get data and update
- **Books** – standard actions like create, delete, update and search books by different criteria (through different fields of a book)
- **Borrows** – regular users can make as many borrows as they want (cannot be deleted); librarians can read and update borrows

# REST API – example

For each resource, define the routes for all actions:

- **USERS**

| Method | Endpoint | Description |
|---|---|---|
| POST | /users | Creates a new user (public route) |
| POST | /users/login | User authentication (public route) – if successfull, retrieves the user authentication token |
| GET | /users/current | Gets all logged user data (protected route) |
| GET | /users/current/borrows | Gets all logged user borrows (protected route) |
| PATCH | /users/current | Updates logged user data (protected route) |

# REST API – example

For each resource, define the routes for all actions:

- **BOOKS**

| Method | Endpoint | Description |
|---|---|---|
| POST | /books | Creates a new book<br>* only for librarians |
| GET | /books?title=…, | Gets a list of books (public route)(can be filtered) |
| GET | /books/:id | Gets one book (more detailed information) (public route) |
| GET | /books/:id/borrows | Gets all borrows of a certain book (protected route) |
| PATCH | /books/:id | Updates one book<br>* only for librarians |
| DELETE | /books/:id | Deletes one book<br>* only for librarians |

# REST API – example

For each resource, define the routes for all actions:

- **BORROWS**

| Method | Endpoint | Description |
|---|---|---|
| GET | /borrows | Gets a list of all borrows (protected route )<br>* only for librarians |
| POST | /borrows | Creates a new borrow (protected route - user and book ids in request body or user id – via token - in header and book id in request body) |
| PATCH | /borrows/:id | Updates a certain borrow (ex: mark the book as delivered)<br>* only for librarians |