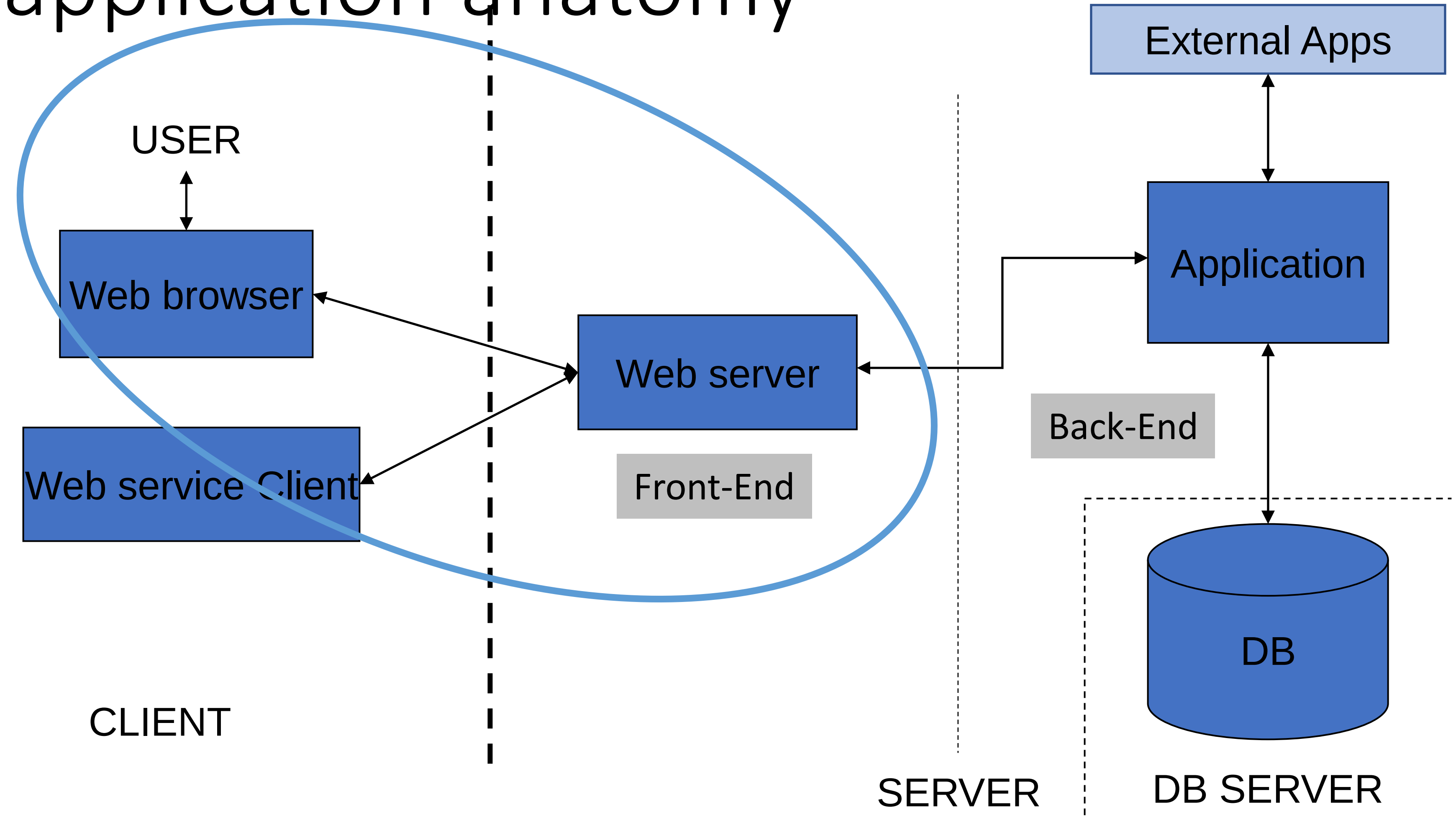
Web, Web applications & APIs

DBW

Outline

- Web basics
  - HTTP servers and browsers
  - Languages
  - Software
- Concept and types of Web applications. Web services
- Languages involved
- CGI Protocol.
- Time issues
- Authentication/authorization. Cookies & session persistence
- Hints
- Web services & APIs
  - REST
  - Languajes
  - Examples

Web application anatomy



Web (HTTP) Servers & Clients

Servers

- Applications **listening** a TCP port (80 typically), and **understand HTTP requests.**
- Information served are text or binary files (*resources*) stored locally in the server.
- HTTP servers that implements the appropriate protocols, can run **server-side applications** according to the request.
- Example SimpleHTTPServer (Perl, Python)

Clients

- Applications **making requests** to a server at a given TCP port (typically 80) **using HTTP protocol**
- Simple (command-line) browsers request for files (using HTTP) in a similar way to FTP. Resources are identified by URLs (i.e. wget, curl)
- Normal browsers “understand” the contents of the obtained files and **combine information** from one or more servers interpreting a given language (usually HTML/CSS) in **graphical output**.
  - Most **browsers can execute applications** (client-side) obtained from the information server
  - Modern Web sites rely heavily **on client-side applications** to deliver dynamics content (mostly with Javascript) → “**single-page applications**”

Languages involved (web interfaces)

- **HTML: Contents management language**
  - Defines contents and structure of the page, includes the necessary links to all elements
  - Tag formatted language `(…<p>Some text</p>…)`

- **CSS: Formatting language**
  - Defines how the contents is represented in the user browsers
  - `P {font-family:Times; font-family: 10pt; display:block; background-color:black}`

- **Javascript: Task execution language**
  - Used for client-side applications
  - Plain or in richer variants like Typescript, and frameworks like Angular, React, …

Languages involved (web services)

XML

Most traditionally used by web applications

- Same structure as HTML, but with no fixed tags
- Requires XML-schema to specify tags and check coherence

```
<Course id="DBW">
   <Acronym>DBW</Acronym>
   <Title>Databases and Web applications</Title>
   <Students>
      <Student id="1">
         <name>Josep</name>
         ...
      </Student>
      …
   </Students>
</Course>
```

JSON

Increasing replacing XML

- Natively understood by Javascript
- Can be validated using JSON-Schemas (not mandatory)

```
Course: {
   "id": "DBW",
   "Acronym": "DBW",
   "Title": "Databases and Web applications",
   "Students": [
      {"id": 1, "name": "Josep", …},…
   ]
}
```

YAML (kind of friendly JSON)

Used mainly for configuration files.

```
Course:
   id: DBW
   Acronym: DBW
   Title: Databases and Web applications
   Students:
      - 1 :
        name: Josep
      - …
```

Software to build Web pages

- A simple text editor is enough (Notepad, vi, nano, ...)

- Syntax checking editors are more useful (vim, gedit, vscode, ...)

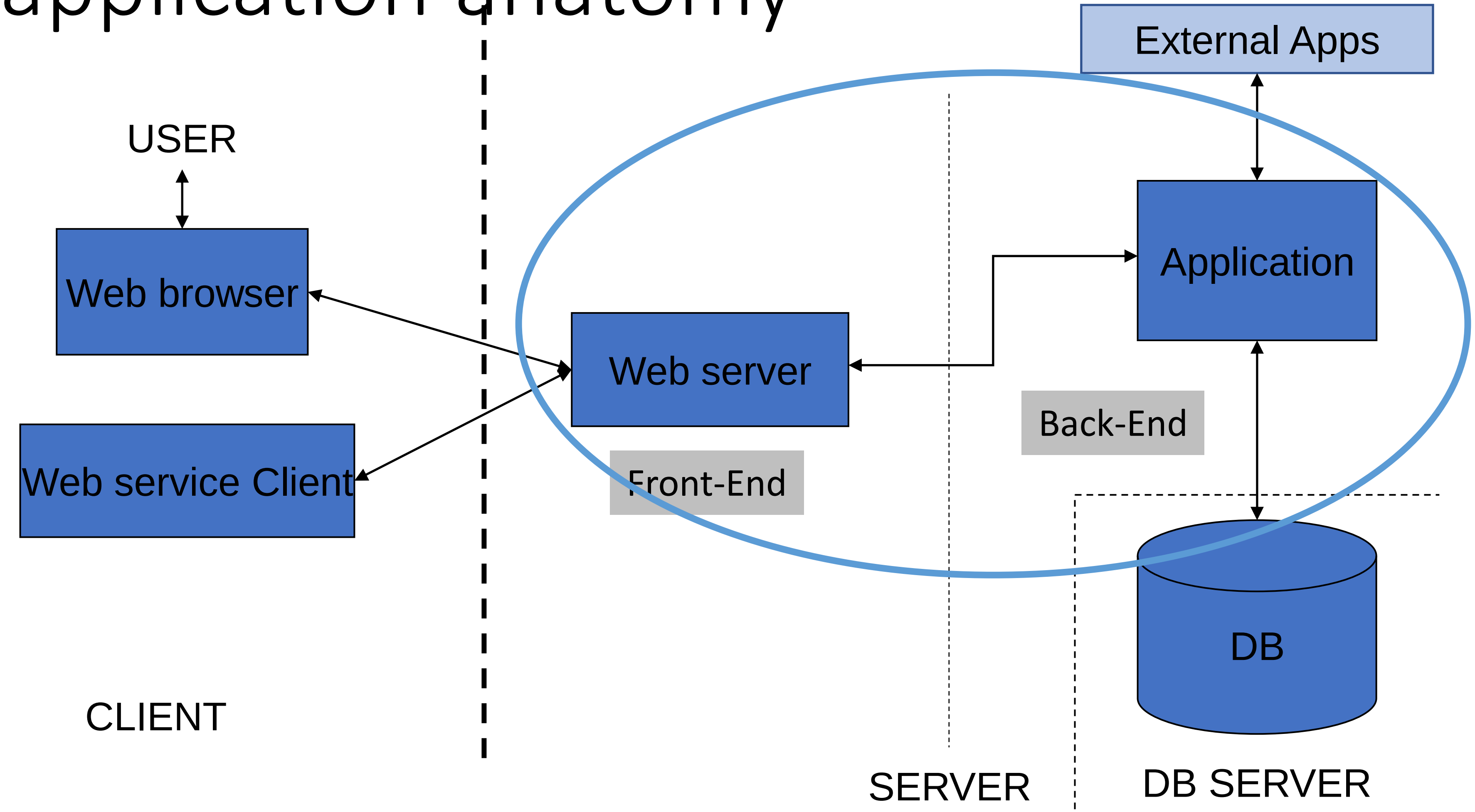
- WYSWYG editors are common (Dreamweaver, OpenOffice...).
  - However, they MUST allow to check HTML manually!

- Content Manager Systems (CMSs)
  - Integrated environments to build web sites, general or specialized
  - Can include some useful functionality (user management, email, …)
  - Very useful to build static sites, but difficult to include applications
    - However, web structure and layout made by the CMS can be used

  - Drupal, Joomla, Wordpress, …

- Pre-build environments (CSS & JS)
  - Jquery, Bootstrap, …

Web application anatomy



Definition & types

- A Web application is a dynamic extension of a Web server.
  - Adapts to user input
  - Can serve non-static information (generated in real-time)
  - Uses standard protocols (HTTP, SMTP)
  - Users interact with the application mainly using Web browsers

- Presentation-oriented
  - Generates dynamic Web pages (HTML/CSS/JS) responding to user queries
  - Usual way to provide bioinformatics results

- Service-oriented
  - Interacts with other applications (XML/SOAP, REST)
  - Allows to build automatic workflows for complex analyses

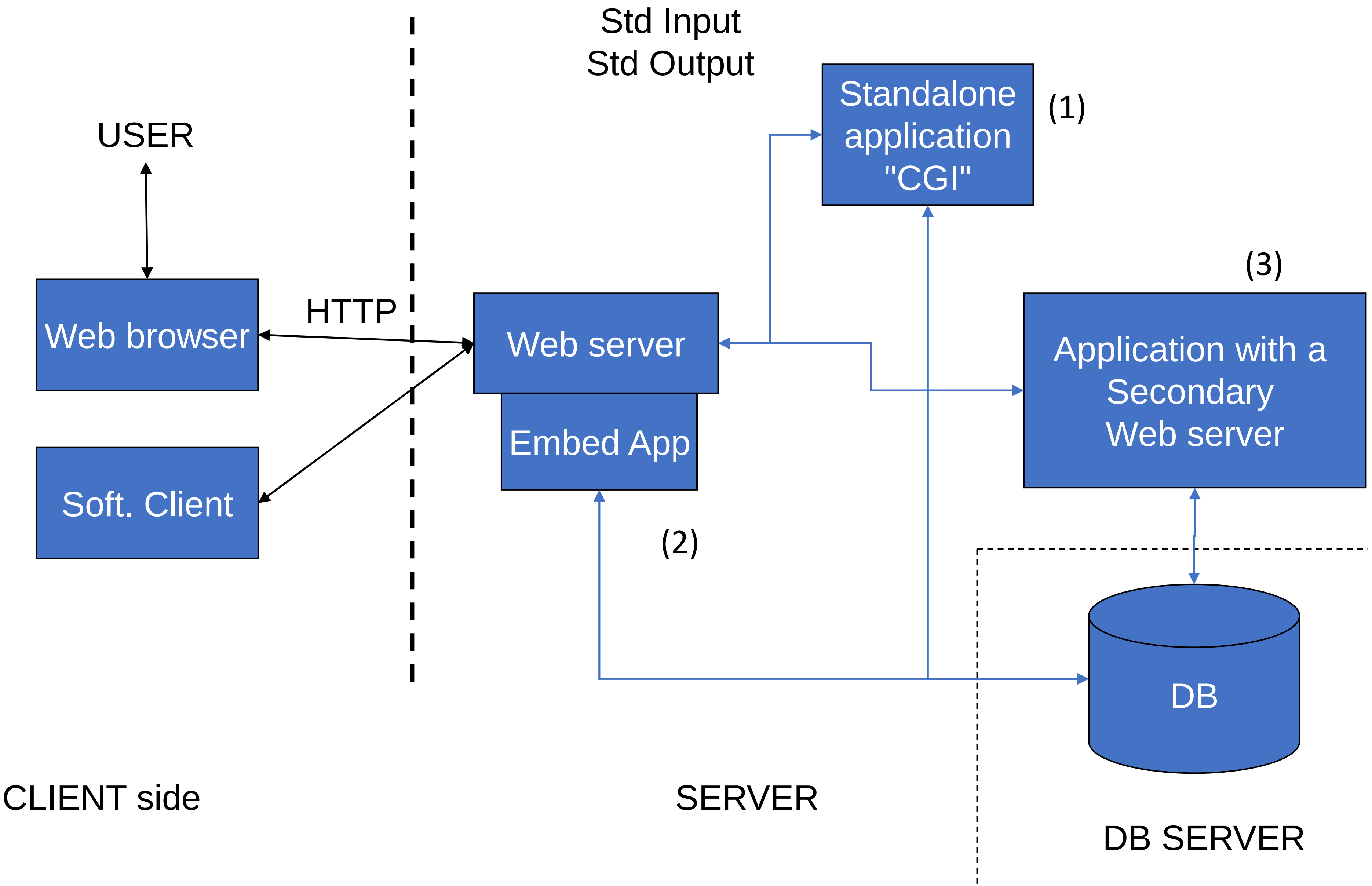
Frontend

- Application must be compatible with standard web browsers
  - HTTP protocol: GET, POST, (PUT)
- User input comes from URL's or HTML forms
- Output must be in known languages: HTML, CSS, Javascript, XML, JSON
- Output may invoke other programs (plug-ins) though MIME
  - Almost obsolete, fully replaced by HTML v5
- HTML v5 include a variety of native functionalities
  - Audio/video, SVG graphics, MathML, GeoLocalization, parallel processing, …

- Modern browsers are able to run **client-side apps** (Java applets, and Javascript)
  - **Client-side applications** are fully qualified applications, served as static files, and **run in the browser**
  - Java applets are almost obsolete (still seen in bioinformatics) due to security issues
  - **Javascript** is behind dynamic behaviour of modern web sites.
    - Asynchronous interaction with server (new requests do not require reload)
    - JsMol, NGLview (molecular structures), Jbrowse (genomic data)
    - JS Frameworks: Jquery, AngularJS, ReactJS, VueJS, …
    - Component libraries for bioinformatics start to be available (https://ebi-webcomponents.github.io/nightingale/#/)
- Client-side apps are generated on the server and sent to the browser as part (or all) of the output.

Backend

- An application is invoked by web server on receiving the request
  - **External application (CGI)**
    - Executable running in the server machine.
      Can be written in any language.
    - Get input from standard input and writes in the standard output. Web server redirects both.
  - **Server embedded.**
    - Web server is able to execute the application as a child process (may require a driver)
    - Usual languages: Python, NodeJS, Perl, Java
    - “Web oriented” languages: PHP, ASP, .NET, JSP
  - Java, Python, JS server-side applications require special servers
    - Installed normally as a secondary web server (port != 80)
    - The main server acts as a “proxy”

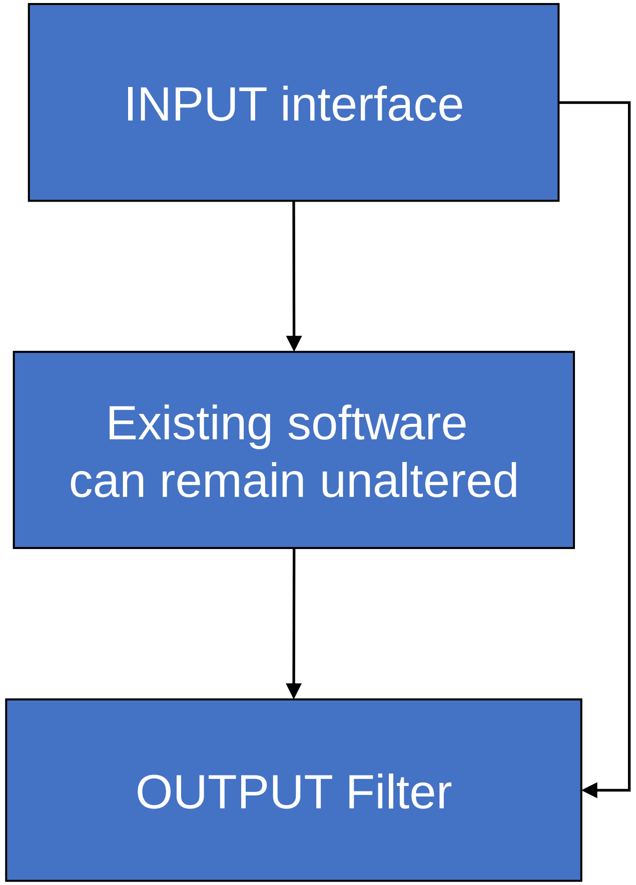

Std Input
Std Output
USER
Standalone application "CGI"
(1)
(3)
HTTP
Web browser
Web server
Embed App
Application with a Secondary Web server
Soft. Client
(2)
DB
CLIENT side
SERVER
DB SERVER

CGI Protocol & strategies

- **Common Gateway Interface (CGI)**
  - **Formal interface** between Web server and external applications
  - CGI interface provides
    - **Environment variables** including information from the browser-server conversation
    - **POST input data**, as standard input
    - **Redirection** of application standard output & error to Web stream.
- **For External applications**
  - Read Input information from Environment variables, and standard input
  - Provides results and error as standard output
  - **Are executed by the operative system** as usual command-line executables (security issues!!)
- **For Embedded applications**
  - Still use input and output standard and CGI variables, but **data is processed internally** in a web server's subprocess

- **Practical CGI use**
  - Languages have specific extensions to deal (Hide) CGI from the development process
    - Input GET and POST, and CGI variables available at predefined variables/objects
    - No need to process HTTP
      - Cookies, authentication, etc.
  - Web programming frameworks
    - Slim, Symphony (PHP), Flask, Django (Python)
    - Programming helpers acting as an interface between CGI and the programming language
    - Provide "easy" web applications

The simplest application



HTML Forms
Data validation
Error handling

Output to HTML. EMail

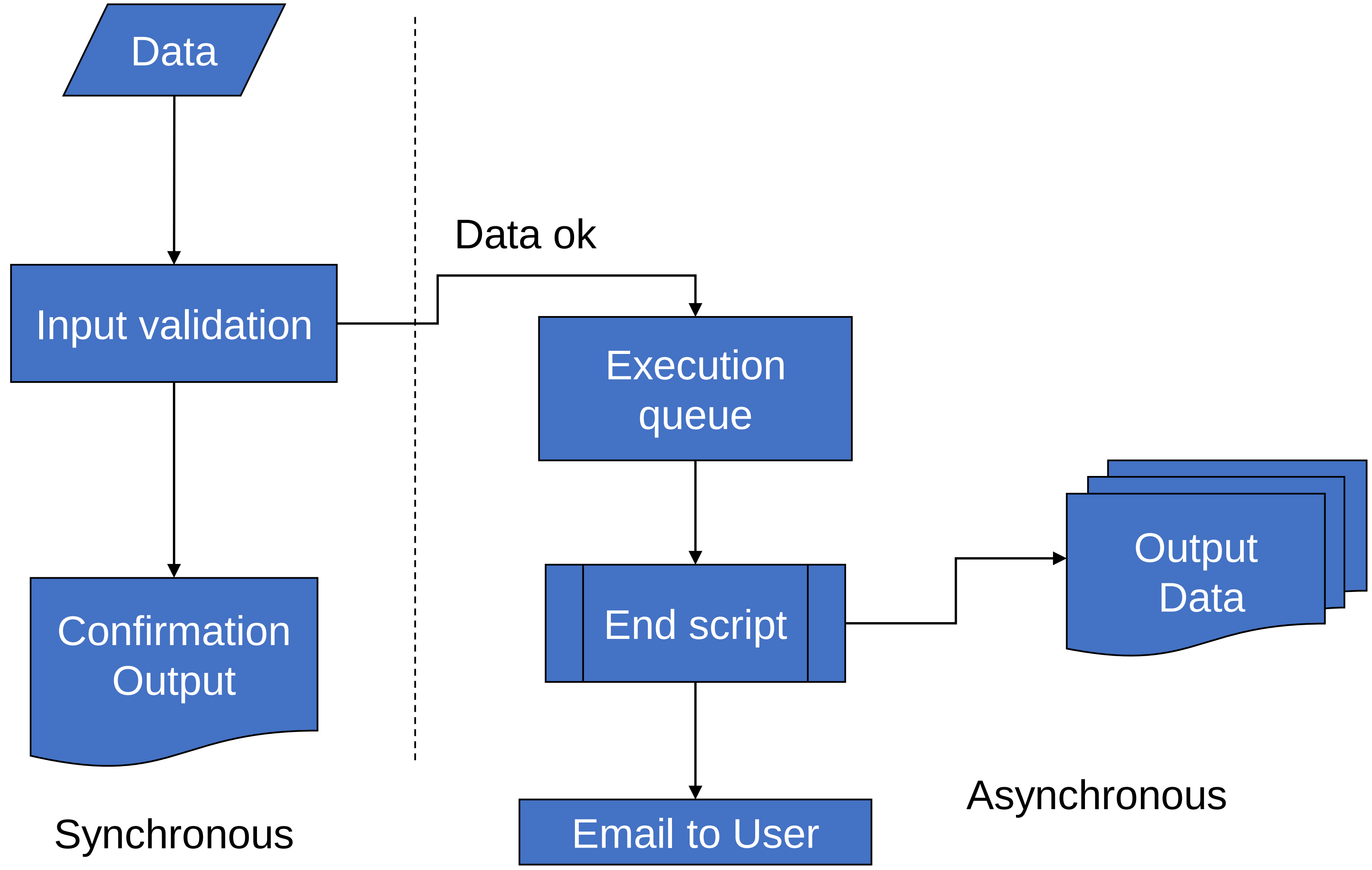
Known issues in web applications

- **Time issues**
  - Web users require “instant” responses
  - Most web browsers (and network helpers) may have a short “timeout”
  - Application that lasts more that 1-2 mins must be asynchronous

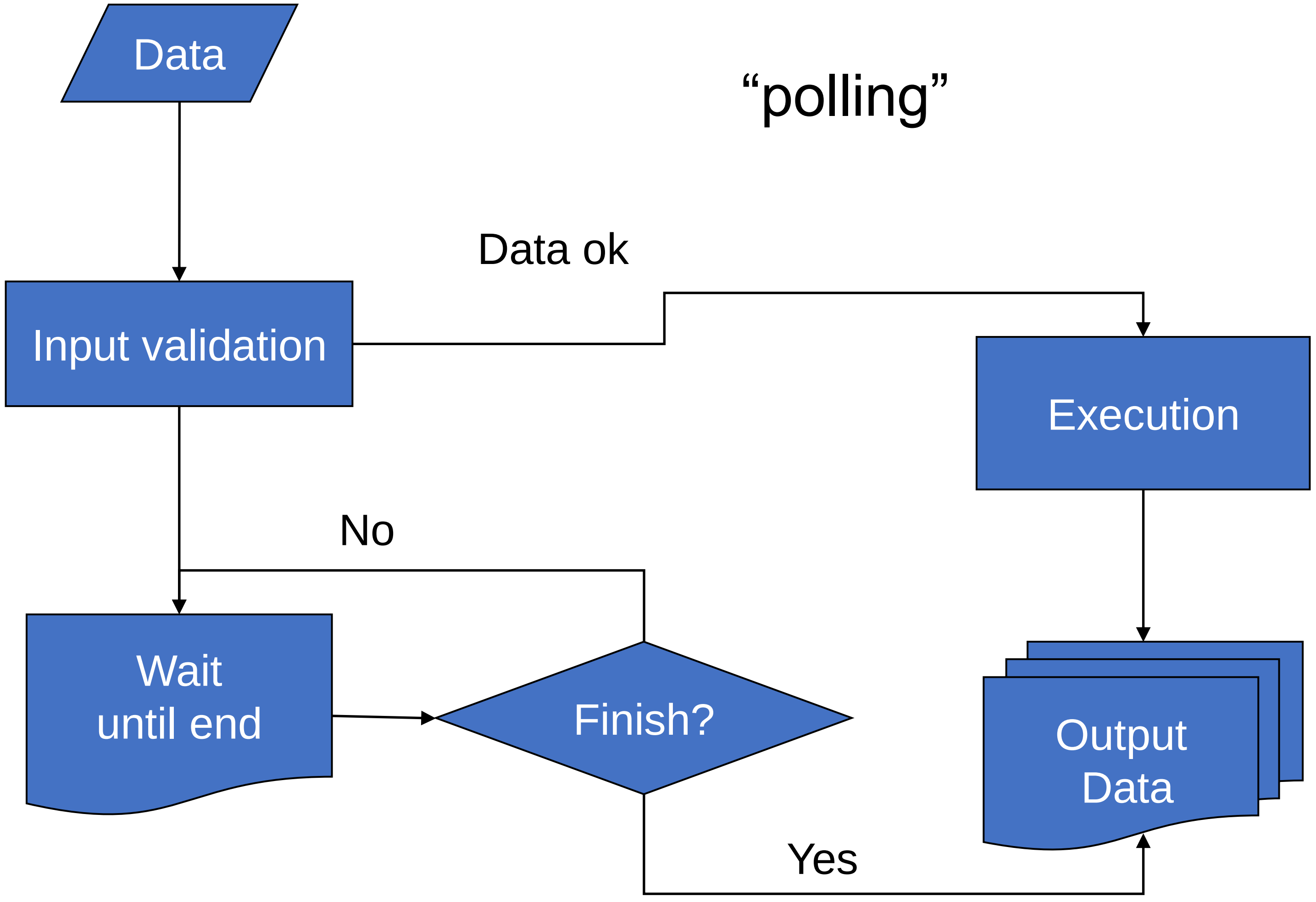
- **Persistence, and User recognition**
  - HTTP protocol is **not persistent**: Connection closes short time (~20s) after the server answers

  - Applications need to **recognize returning users**
    - Authentication (user only must write the login/password once)
    - Keep personal preferences, and private data
    - Grant access to given resources according to previous requests
    - Avoid requesting known data more than once
    - Avoid “reloads”

Time issues. Usual strategies (1)

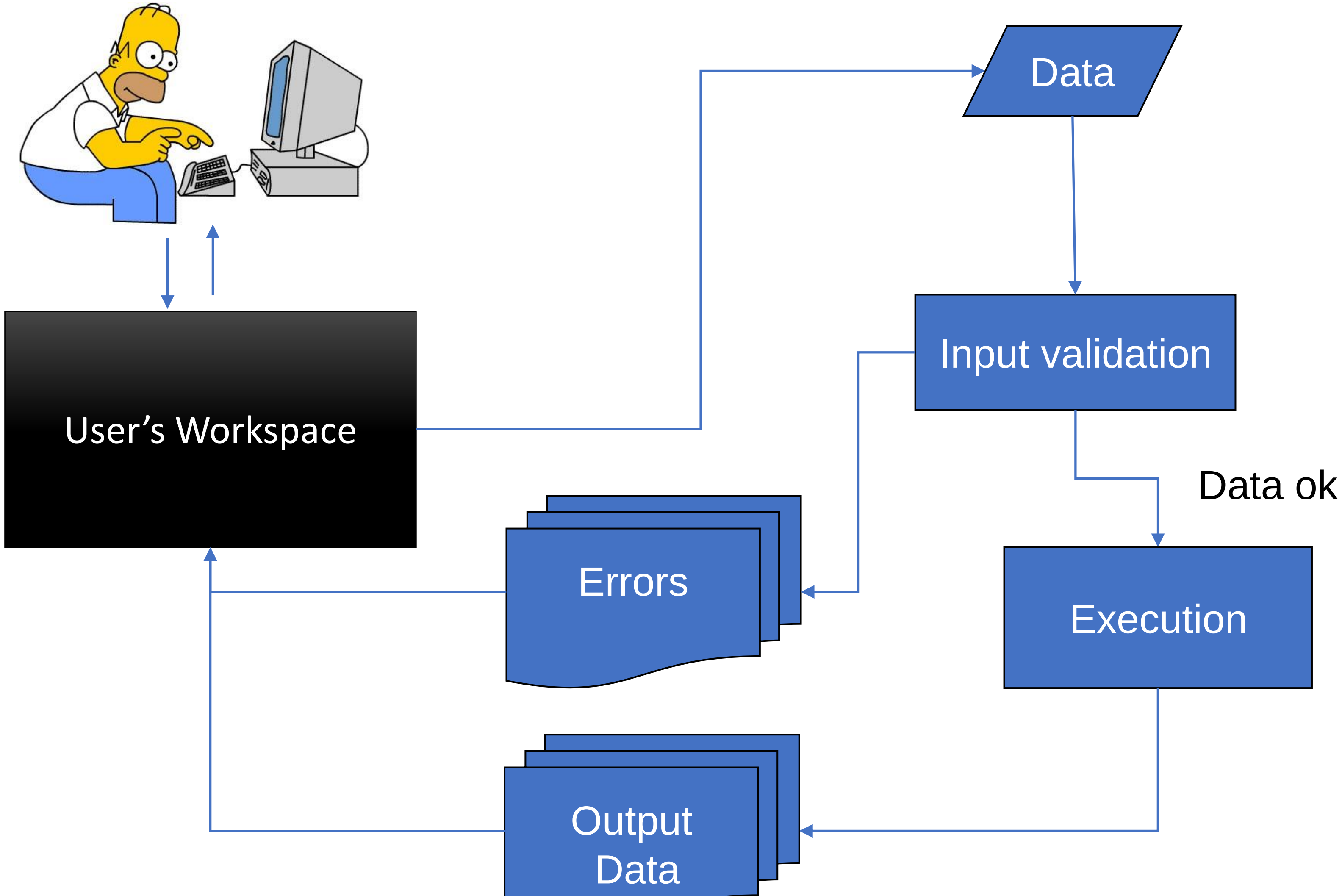


Time issues. Usual strategies (2)



“polling”

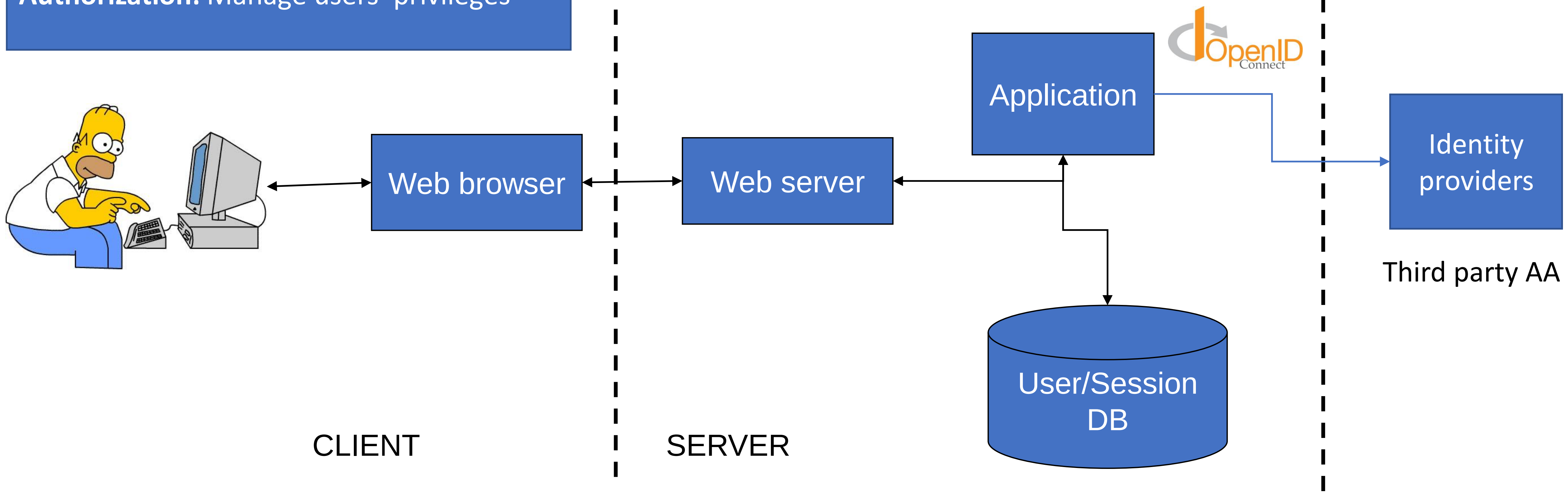
Time issues. Usual strategies (3)



Personalization (Authentication/Authorization)

Authentication: Identify Users

Authorization: Manage users’ privileges



A DB stores history of user connections and activities

Authentication/Authorization schemes

- **Server based authentication**
    - Based on unix-like passwd files (login / passwd)
    - Protects folders and sub-folders (.htaccess files)
    - Identity via CGI variable (REMOTE_USER)
    - May **require access** to **server configuration**
    - Environment managed by local DB

- **Application based (both authentication and authorization)**
    - Do not require access to server configuration
    - Authentication and environment managed by the application itself (via local DB)
    - Full control from the application (login / passwd, SSL Keys, …)
    - Persistence via Cookies or language specific constructs (PHPSessionID / Session)

- **Third party authentication (single sign-on, SSO)**
    - Authentication is done by identity providers (Google, openID, ELIXIR, ….), or other apps (eGroupWare, Drupal, …)
    - **Users can have a single point for authentication (SSO)**
    - Protocols
        - OAuth, OpenID Connect
    - Bioinformatics world
        - *Bonda fide* Researchers
        - European Life Sciences Id / ELIXIR

User identification from the application: cookies

- Small amount of text information stored by the server in the users' web browser as key /value pairs
- Do not require user/password (user do not need to be aware of)
- Limited to 4Kb
- Retrieved automatically as part of CGI handshake
- Cookies do not identify persons but browsers!!

- ID: a unique ID generated by the server
- Origin: server URL. Browsers send back cookies to the servers that created them (no other servers can get the data)
- Expiration date. Cookies can last for a single session or till a specified date

Web application layout **hints**

- Static contents (text, images, etc.) stored as normal web resources
  - For optimization, some servers keep them separated from scripts
- Dynamic pages managed by server scripts
  - No general rule, depends on language and programming style
  - The easy way: Each different screen is managed by a specific script.
  - Web frameworks use normally a “routing” mechanism to associate code to incoming URLs
  - **Single-Page-Interfaces** (http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php)
    - Becoming popular due to JS frameworks.
- Global variables
  - Each script acts in a separated HTTP transaction!
  - All scripts should load the **same global environment**, usually included from a single file
- Protected/public data
  - Protected data should be stored **outside of the web directory tree**, and be accessed only programmatically
  - Output data may be placed inside the web tree if it is already HTML/CSS

Web application layout **hints**

- Temporary data
  - Can be stored anywhere
  - Most languages provide automatic temporary directories and file names.
  - Be aware that applications can be executed by the “web” user, check permissions.
  - Should be deleted after use!!

- Beware of multiple concurrent users
  - Use request-specific file names for temporary data and results
  - Use user-based or process-based directories
  - Think in a queueing system for lengthy operations

- Collect statistics of use!!

DBW - APIs

APIs – Programmatic access

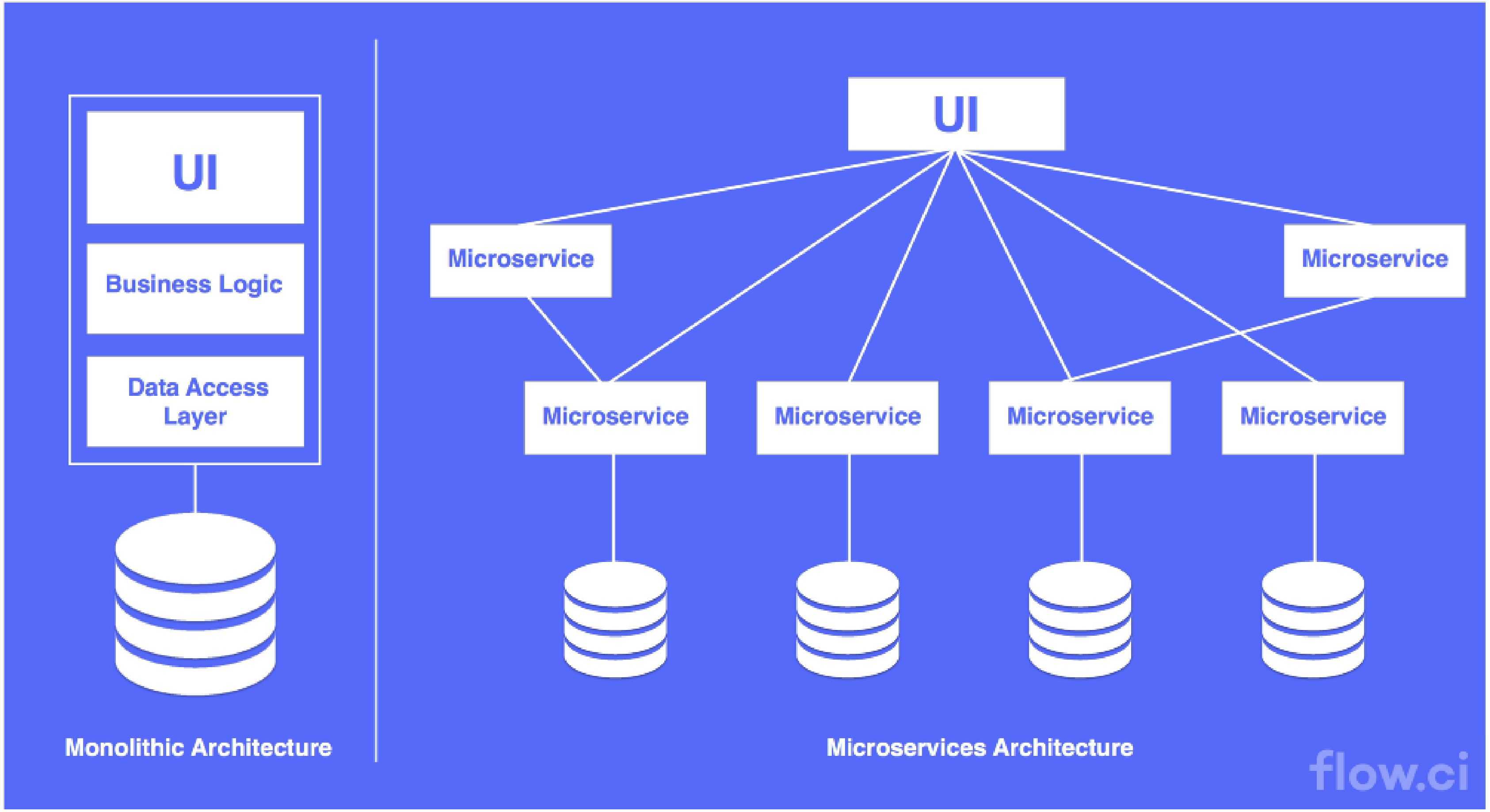
- API = “Application programming interface”
    1. Set of routines, functions, or procedures/methods, offered as a library to be used inside other software (API = “software library”)
    2. Any web service providing remote functionalities to be used inside other software or most usually as data provider (API = “web service”)

- In bioinformatics the approach is generally used to access data or to allow communication among application components (“microservices architecture”)

- Strategies: SOAP/XML, XML-RPC (remote executions), **REST**

- Authentication implicit via Oauth2 / openIDConnect

Microservices architecture



REST (REpresentational State Transfer)

- REST-ful web services
  - Used primarily to serve data
  - Data can be pre-processed at the server-side (so becoming a kind or RPC)
  - Controlled through HTTP and called using standard URLs
    - /api/{store}/{id}/option.format?option
  - HTTP interfaces to Data repositories

- HTTP based (GET, PUT, POST, DELETE)
  - Allow to GET, PUT (update), POST (insert), and DELETE a resource in the data portal (i.e. a DB)

- Using REST APIs, application components can be independent (and distributed) as long they communicate using HTTP and a known format

Data exchange languages

- Data exchange formats
  - XML: Most traditionally used by web services (SOAP, RPC)
    - Same structure as HTML, but with no fixed tags
    - Requires XML-schema to specify tags and check coherence
  - JSON: Data interchange format replacing XML (most popular)
    - Natively understood by Javascript
- Both require “schemas” to validate data model

Programmatic Access (client side)

- Perl

```
use LWP:Simple;
use JSON;
my $content = decode_json(get(‘http://...’));
```

- PHP

```
$data = json_decode(file_get_contents(“http://....”));
```

- Python

```
import requests
data = requests.get(‘http://...’).json()
```

Web service : server side

- Usual web applications but…
    - Output is not meant to be shown in browsers (no HTML, CSS, JS)
    - Headers required
        - `Content-type: text/xml | text/plain | application/json | application/x-gzip | image/png`
            - Define the type of data being sent
        - `Content-Disposition: attachment; filename=file_name`
            - Force download (when seen from a browser)
        - `Access-Control-Allow-Origin: *`
            - Allow access from any client (to avoid security checks on JS/AJAX)
    - Formats can change
        - In theory should be requested via HTTP (Accept…) but normally are included in the URL
    - Error handled via HTTP codes
        - 200 ok, 404 not found, …
    - Prevent caching
        - Header: `Cache-Control: no-cache`
- Programming frameworks are very useful here due to the complex routing

- A “quite complex” backend : http://mmb.pcb.ub.es/gitlab/MMBData/MMBApi

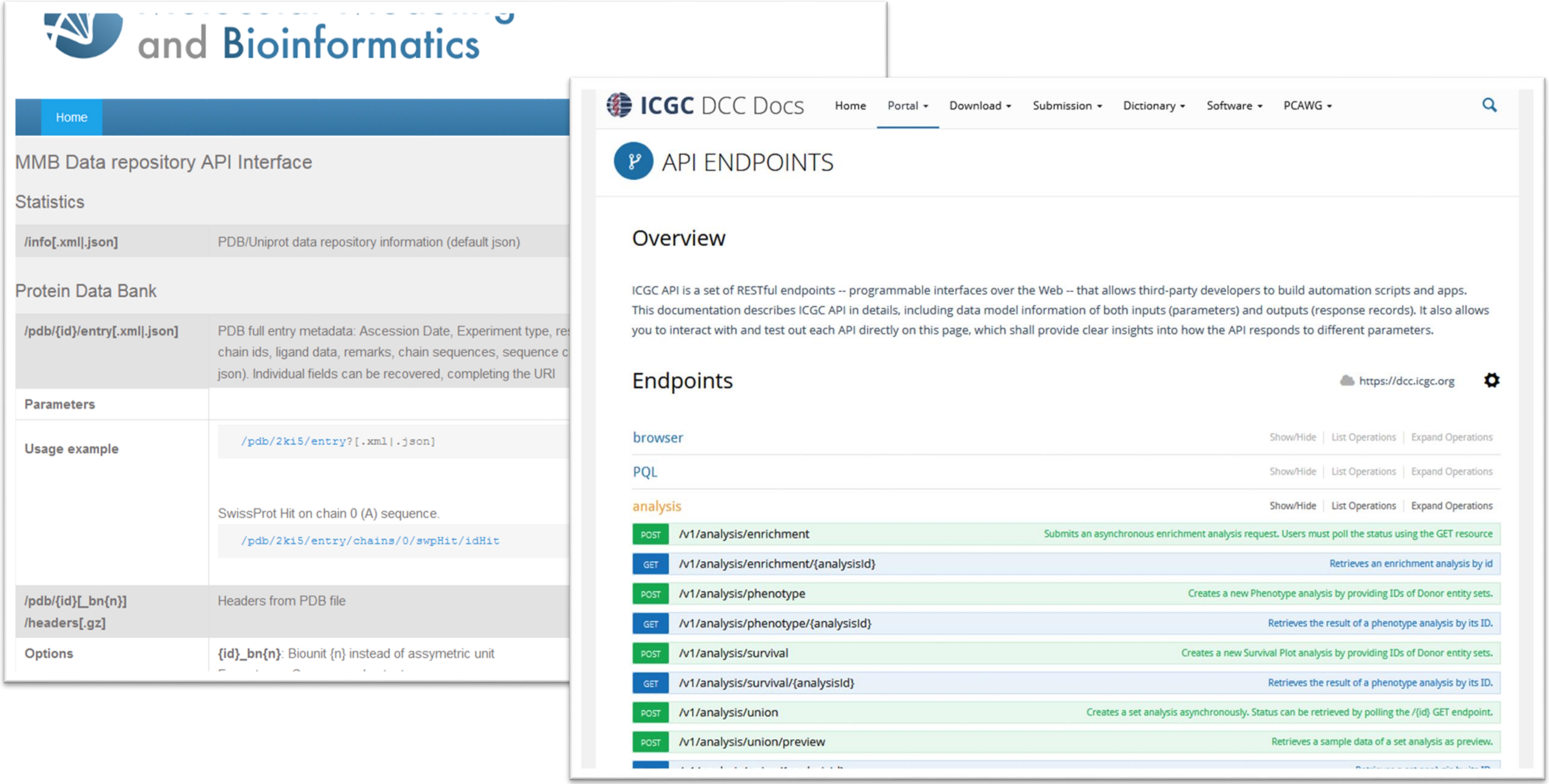
RESTful URLs

- No standard
  - A typical schema is

    /api/{store}/{id}/option.format?options

  ex. `http://mmb.pcb.ub.es/api/pdb/2ki5/entry.json`

- Documented via
  - *Ad-hoc* help pages
  - OpenAPI (a.k.a. Swagger) (recommended)
    - ICGC example



and Bioinformatics

MMB Data repository API Interface

Statistics

/info[.xml|.json]	PDB/Uniprot data repository information (default json)

Protein Data Bank

/pdb/{id}/entry[.xml|.json]	PDB full entry metadata: Ascession Date, Experiment type, res chain ids, ligand data, remarks, chain sequences, sequence c json). Individual fields can be recovered, completing the URI
Parameters	

Usage example	`/pdb/2ki5/entry?[.xml|.json]`
	SwissProt Hit on chain 0 (A) sequence.
	`/pdb/2ki5/entry/chains/0/swpHit/idHit`

/pdb/{id}[_bn{n}] **/headers[.gz]**	Headers from PDB file
Options	**{id}_bn{n}**: Biounit {n} instead of assymetric unit

API ENDPOINTS

Overview

ICGC API is a set of RESTful endpoints -- programmable interfaces over the Web -- that allows third-party developers to build automation scripts and apps. This documentation describes ICGC API in details, including data model information of both inputs (parameters) and outputs (response records). It also allows you to interact with and test out each API directly on this page, which shall provide clear insights into how the API responds to different parameters.

Endpoints

https://dcc.icgc.org

browser — Show/Hide | List Operations | Expand Operations

PQL — Show/Hide | List Operations | Expand Operations

analysis — Show/Hide | List Operations | Expand Operations

POST	/v1/analysis/enrichment	Submits an asynchronous enrichment analysis request. Users must poll the status using the GET resource
GET	/v1/analysis/enrichment/{analysisId}	Retrieves an enrichment analysis by id
POST	/v1/analysis/phenotype	Creates a new Phenotype analysis by providing IDs of Donor entity sets.
GET	/v1/analysis/phenotype/{analysisId}	Retrieves the result of a phenotype analysis by its ID.
POST	/v1/analysis/survival	Creates a new Survival Plot analysis by providing IDs of Donor entity sets.
GET	/v1/analysis/survival/{analysisId}	Retrieves the result of a phenotype analysis by its ID.
POST	/v1/analysis/union	Creates a set analysis asynchronously. Status can be retrieved by polling the /{id} GET endpoint.
POST	/v1/analysis/union/preview	Retrieves a sample data of a set analysis as preview.

Full entries and sequences

- /api/pdb/{id}/entry/ /api/pdbMonomer/{id}/entry/
  - Full data in XML or JSON

- /api/pdb/{id}.fasta

- /api/uniprot/{id}/entry

- /api/uniprot/{id}.fasta

PDB search options

- /api/pdb/ Search on PDB
  - **resmin=value, resmax=value** Min Max for resolution (XRAY only)
  - **qcompType=(prot, nuc, prot-nuc, carb, other)** Compound types.
  - **qexpType=(ELECTRON_CRYSTALLOGRAPHY, ELECTRON_MICROSCOPY, FLUORESCENCE_TRANSFER, INFRARED, NEUTRON_DIFRACCTION, NMR, SOLID-STATE_NMR, X-RAY)** Type of Experiment.
  - **query=txt** Text query
    queryOn=(header, compound, sources, authors)
  - **sequence=seq** Sequence match
    - **molTy=(protein | na)** Sequence type
      seqType=(exact | regex) Type of sequence match (exact | regular expression)

PDB options for structures

- /api/pdb/{id}/ /api/pdbMonomers/{id}/
  - Default: standard PDB coordinates(possible .gz)
  - Available filters
    - **bunit={n}** Show Biounit n instead of the Assimetric Unit
    - **noheaders=1** Skip PDB headers (implicit in the following filters)
    - **group=(ATOM | HETATM)** PDB label selection. (HETATM includes CONECT)
    - **groupRes=[!](POLAR | APOLAR | NUC | PROT)** Residue type selection. "!" negates
    - **groupAt=[!](POLAR | APOLAR | NOH | BACK | NABACK)** Atom type selection, "!" negates
    - **filter=[!][RES]nres:chain.atom/model** Atom filter using J(s)Mol format ("!" negates selection)