Web and Cloud Computing EXAM

Class 1

Web cloud applications, not disjoint:
Linked to the user
web services is what you don't see, doesn't have UI

1. **internet applications**: any applications that uses the internet to consume and provide services and data
   phones (so people are an internet application), IoT: largest creations of data
2. web applications: things that run on a browser, written in HTML.
   HTML is portable: the same on every device. Web assembly is alternative, is more powerful but will not be the same on every browser
3. mobile application could be a web application, runs on a mobile device. Depend on online data sources.
4. **Web service**: computational procedure available on the Internet to provide services and data to other apps and services. Usually associated to binding technologies like SOAP and REST.
5. **Cloud applications**: internet application deployed on an independent hosting service (data centre), providing computation and with certain properties, like on demand resources. Companies don't like to commit to hardware set-up, therefore put everything on cloud and pay as you go. Once you are connected, subscription price goes up, so people leave this idea more and more.
6. **Service-based architecture**: interconnected web services, implemented using orchestration languages and tools. Different companies can do different parts of the system and everything can be integrated trough google.
7. **Data-centric applications**: applications that are developed primarily based on resource-based rules, making their data available to others through well defined interfaces. Everything above is one, since everything goes around a database. Persistent data is needed for software. E.g. CRM systems, BI tools

Cloud computing is:

- On-demand remote computational recourses, without user intervention
- Remote network access to the assigned resources (heterogeneous client platforms)
- Resources can be allocated to multiple clients/services
- Elasticity (you pay as you use): you don't need to know in advance amount of compute you need
- Secure

3 categories on which it depends that the application you have will be more safe in the cloud or on premises:

- Infrastructure as a service: regular computer. You manage the operating system, and everything that comes with it, such as security. Advantage is that it is cheap: AWS, Azure
- Platform as a service, has a lot already installed: e.g. DB or data lake, security is taken care of and you just upload the source code.
- Software as a Service: contract to customize and use a particular product in the cloud. The code is no longer yours, you want an ERP: invoices to payments an application is hired for, you don't have access to the database, you just use the application.

Opportunities Cloud Computing:

- User base is the world
- Flexible computing power: geolocation of your data possible
- Data gathering mechanism
- Data analytics & visualization
- Short delivery cycles (pushed to users): Application for mobile phones: always running on new code to debug, e.g. facebook pushes new software every two weeks. Every hour for Netflix. Don't wait for people to install new versions.

Challenges Cloud Computing:

- Security: exposed to the world (attacks and leaks)
- Robust quality assurance method
- Scalability and performance
- User experience and user engagement are critical: first experience is important!
- Systematic development process: high code takes more time than low code

Cloud computing demands:

- Flexible services
- Modularity in the development
- Loosely coupled components (few dependencies)

Client <– HTTP –> Server (UI + data)
= Hypertext transfer protocol, you can also have servers communicating to other servers. In the end of the chain you have databases.

Partition is related to:

1. Abstraction/reuse:
   Partitioning involves breaking down a system into smaller, more manageable components or modules. This process often leads to the identification of common functionalities or patterns, which can be abstracted into reusable components. By partitioning a system effectively, developers can promote code reuse and modularity, enhancing the maintainability and scalability of the software.
2. different concerns:
   Partitioning helps in addressing different concerns or aspects of a system separately. For example, in the context of software architecture, partitioning allows separating concerns such as data storage, presentation, business logic, and user interface. This separation enables developers to focus on specific aspects of the system independently, leading to better organization and understanding of the software.complexity
3. technology:
   each component can be developed, tested, and maintained independently
   running on other tech-stacks is fine
4. ownership:
   department responsibilities
5. performance:
   By distributing workload across multiple partitions or nodes, parallelism and concurrency can be achieved, leading to improved performance and scalability.

data collection system needs to be real time while accounting system can stop over night and take backup

Elastic load balancing component ELB:
= component that automatically distributes incoming application traffic across multiple targets, such as EC2 instances, containers, or IP aadresses. The main purpose of an Elastic Load Balancer is to ensure that incoming traffic is distributed evenly across backend resources to improve availability, fault tolerance, and scalability of applications.
→ flexible well organized structure with good software architecture: ELB is key you need in order to organize all the components in order for application not to break down

Software Architecture:
Good organization of the internal structure of systems leading to:

- Productive development
- Easy evolution
- Easy maintenance: if it breaks it has to be easy to find where the problem is
- Trustworthiness: push towards a trustworthy solution, good software architecture gives you framework to avoid mistakes, since mistakes are made by humans
- High performance level

Technology stack:

3 tier architecture:
web client-side applications (HTML, JavaScript) – server-side applications (JavaScript, Express) – data storage and manipulation (WebServices, relational and noSQL, spark)

The spectrum of client-side applications:

- website / static HTML pages
  static data, poor behaviour, efficient response times (w/caching), dynamic websites are sometimes expanded for more efficiency.
- website / dynamic server rendered HTML pages
  poorer loading times, poorer development features (distance between code and result). Poor modularity, high usage of templating languages.
- website / asynchronous communication (AJAX)
  better loading times, development closer to the result (no templating needed). Better modularity and testing conditions.
- web applications (Single Page Application)
  complete application context, service based back-ends, richer behaviour and fluid UI
- progressive web applications (PWA)
  device agnostic, and yet, closer to native client applications (storage, resources, responsive UI, notifications, etc.)
- mobile applications
  more device resources, many times PWA in a shell that links to the device native features.

<u>Web technologies: practical part</u>

WWW= HTTP+ HTML + Server + Browser
= resources (often files, but also pictures and videos) interlinked

HTTP: hypertext transfer protocol
HTML: hypertext markup language

The HTTP protocol:
HTTP Request = URL + method + body

- The target resource, a URL: uniform resource locator
- The HTTP method, a verb that describes an action/operation that I want to do with that resource: GET, POST (data to you) , PUT, DELETE. If you type URL and enter, it is a GET (data to me).
- The body of the request (optional): one or more files containing a sequence of name/value pairs, a json object, a binary file, etc.
  = set of parameters for that operations, data that comes from the server towards the client
  Body can contain various types of data formats:
  HTML(from server to client),
  JSON (sending data between a server and a client),
  XML (exchanging structured data between different systems such as webservices),
  Form-data (submitting data from web forms),
  any type of textual or binary data that needs to be exchanged between server and client
- The kind of content in the request (application/json, multi-part/form-data, etc.)
- Security information (using cookies and tokens)
- Other headers indicating the client of the request (user-agent) and what kind of answer is expected (text/html, application/json, etc.

The HTTP protocol
HTTP response = code + body

- The Status code of the response = actual response: yes/no/maybe
- The kind of content in the response (text/html, application/json, etc.)
- The body of the response (optional) containing data. One or more resources.

URL: uniform resource locator = target resource
URI: uniform resource identifier

URL: http://serveraddress/pathtoresource:port#id?querystring

1. **Protocol (http, ssh, git, mailto,..)**: This specifies the communication protocol to be used for accessing the resource.

2. **Server Address or IP Address**: This part identifies the server where the resource is located. It can be either a domain name (like "example.com") or an IP address (like "192.168.1.1"). This is the destination where the client wants to connect to retrieve the resource.

3. **Path to Resource**: This is the specific path on the server where the desired resource is located. It resembles a file path on a disk, with slashes ("/") separating different directories or folders. This part is optional, but if present, it helps to locate a specific resource on the server. The same path components on different servers may refer to entirely different resources. The server address (domain or IP) in the URL is crucial in determining the full resource identifier.

4. **Port**: Ports are used to differentiate between different services running on the same server. While the server address directs the request to the correct machine, the port number ensures that the request reaches the correct service or application running on that machine. If no port is specified, the default port for the protocol is used (e.g., 80 for HTTP, 443 for HTTPS).

5. **Fragment Identifier**: This part of the URL, denoted by the "#" symbol, specifies a specific section within the resource itself. In the context of an HTML document, it usually refers to an element with a particular ID attribute.

6. **Query String**: The query string, if present, comes after the "?" symbol. It consists of key-value pairs separated by "&" symbols. This part is commonly used to pass parameters to the server, which can then be processed to tailor the response. The parameters in the query string shape the operation or specify additional information about the request.
According to the REST convention, the query part of the URL should only be used to refine the resource identified by the URL, for example, filtering a collection, breaking the results into pages, or specifying the format of the return value.

Everything except for the query string is the root of the research.
According to the REST convention, the URL is NOT sufficient to identify an operation applied to a resource available at an API, as it specifies the resource, but not the operation to be applied to that resource. The operation is determined by the HTTP method (GET, POST, PUT, DELETE). Query parameters in the URL given through the query string, can provide additional information, but are typically used for refining or filtering the resource.

URL usage in HTML form: by default the usage is a GET method if not specified otherwise. The fetch function in JavaScript defaults to using the GET method if not specified otheriwe.

example URL:

```http://example.com/path/to/resource:8080#section1?param1=value1&param2=value2```

This URL uses the HTTP protocol to connect to the server "example.com" on port 8080. It requests the resource located at "/path/to/resource", specifically targeting the section with the ID "section1". Additionally, it includes a query string with two parameters: "param1" with the value "value1" and "param2" with the value "value2".

Data exchange formats:

- JSON: Javascript object notation (lists, records & basic types)
- XML: Extensible Markup Language (Tree-like representation of data): tree structure

Parsing: understanding the structure of the sequence of sentences into something meaningful, like tokenization in NLP

%20 is a white space
32-> ASCII for ‘ ’
each character has a code and there are characters not allowed in body or method
browser inspector: F12
```

HTML:

- Derives from the XML format
- HTML elements may have attributes:
    - generic: set on any HTML element (id, class, hidden)
    - particular: img/scr, option/value, etc.
    - events: dependent on the kind of element body / onload, button / onclivk, etc.

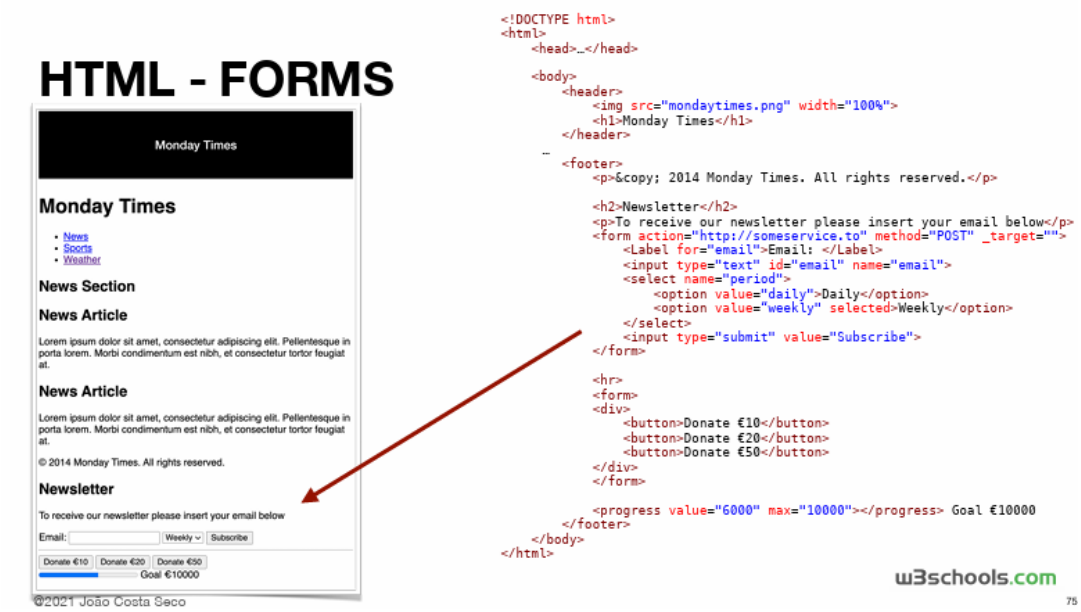
<!DOCTYPE html> defines the kind of document
<html></html> encloses all elements, defines the root element
<head></head> defines meta-information, stylesheets and scripts
<body></body> encloses the elements that define the content
<p> <h1> define block elements
  <i> define inline elements
&lt, &gt, &#233 escaped characters

Attributes:
Src= source: links an element to a file, img or script
id = identifies a single element in a page. Manipulation and styling defined using idientifiers.
class: identifies a group of elements
name: used in forms to identify data sent to a server

a is an anchor: it is a link to another page
href=news is an attribute that scrolls down the page until it finds an a and puts it there
Image is a URL that can be stored anywhere on the internet
Style sheet: font-family: sans-serif

HTML 5 semantic elements (part of)

Tag	Description
<article>	Defines an article in the document
<figcaption>	Defines a caption for a <figure> element
<figure>	Defines self-contained content, like illustrations, diagrams, photos, code listings, etc.
<footer>	Defines a footer for the document or a section
<header>	Defines a header for the document or a section
<main>	Defines the main content of a document
<menuitem>	Defines a command/menu item that the user can invoke from a popup menu
<nav>	Defines navigation links in the document
<section>	Defines a section in the document




```html
<!DOCTYPE html>
<html>
    <head>
        <!-- Head defines invisible information about the page -->
        <!-- Meta information, only read by the machine -->
        <meta charset="utf8">
        <meta name="description" content="My personal web page">
        <meta name="keywords" content="HTML, CSS, XML, Javascript">
        <meta name="author" content="João Costa Seco">

        <style type="text/css">
            * { font-family: sans-serif; }
        </style>

    </head>

    <body>
        <header>
            <img src="mondaytimes.png" width="100%">
            <h1>Monday Times</h1>

        </header>

        <nav>
            <ul>
                <li><a href="#news">News</a></li>
                <li><a href="#sports">Sports</a></li>
                <li><a href="http://weather.com">Weather</a></li>
            </ul>
        </nav>
```

```
        <section>
            <h2 id="news">News Section</h2>
            <article>
                <h2>News Article</h2>
                <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Pellentesque in porta lorem. Morbi condimentum est
                    nibh, et consectetur tortor feugiat at.</p>
            </article>
            <article>
                <h2>News Article</h2>
                <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Pellentesque in porta lorem. Morbi condimentum est
                    nibh, et consectetur tortor feugiat at.</p>
            </article>
        </section>

        <footer>
            <p>&copy; 2014 Monday Times. All rights reserved.</p>

            <h2>Newsletter</h2>
            <p>To receive our newsletter please insert your email below</p>
            <form action="http://someservice.to" method="POST" _target="">
                <Label for="email">Email: </Label>
                <input type="text" id="email" name="email">
                <select name="period">
                    <option value="daily">Daily</option>
                    <option value="weekly" selected>Weekly</option>
                </select>
                <input type="submit" value="Subscribe">
            </form>

            <hr>
            <form>
            <div>
                <button>Donate €10</button>
                <button>Donate €20</button>
                <button>Donate €50</button>
            </div>
            </form>
            <progress value="6000" max="10000"></progress> Goal €10000
        </footer>
    </body>
</html>
```

Attributes class and id:

Let's "div"vy up the tag

```
<div id="unique-id" class="some class">
    ..div element contents..
</div>
```

- id attribute should be unique
- class attribute doesn't need to be unique

a div tag can belong to more than one class, in this case the classes are separated by a space, e.g. here div tag belongs to classes "some" and "class"

item.classList.add("yellowstar")

"a" tag is the specific tag for hyperlinks: link we click on within a website to redirect us somewhere.
most important attribute: href indicates the url where the a hyperlink redirects to

Class 3

Software architecture

= way the highest level components are wired together, components are webserver, services, etc. Human deals with complexity by divide and conquer, therefore the different components are handled separately and then software architecture is needed to wire them together.

These components are defined by: pieces of these components, relations, properties, and quality attributes

The organization of structures and views (views on software architecture):

- Module structures (code or data units): represent functionality
- Component/Connector structures
    - Components: servers, clients, filters, etc.
    - Connectors: call-return, pipes, sync operators, etc. (enable communication and interaction between components)
- Allocation structures: resources that support components and connectors. Describe how the system's components and connectors are mapped to the underlying hardware resources, such as processors, memory, networks, etc. Allocation structures ensure that the system's architectural design is compatible with the available resources and can be effectively deployed and managed.

Software architecture affects:

- Performance: identify bottlenecks, isolate and optimize locally.
- Maintainability: localized bugs, less impact on other components: divide in parts and give each part an owner in order to make sure the system is correct. Maintainability is about evolution (more functionality) or correction. Because slide here has separate bullet point 'evolution' we see this one as 'maintainability correction'
- Scalability: replication of components, not the system: run different versions of the same code to satisfy different types of users: each student in this school is connected to different server of Netflix (doesn't have to do with location)
- Evolution: add more components and improve the "hub"

Architectures:

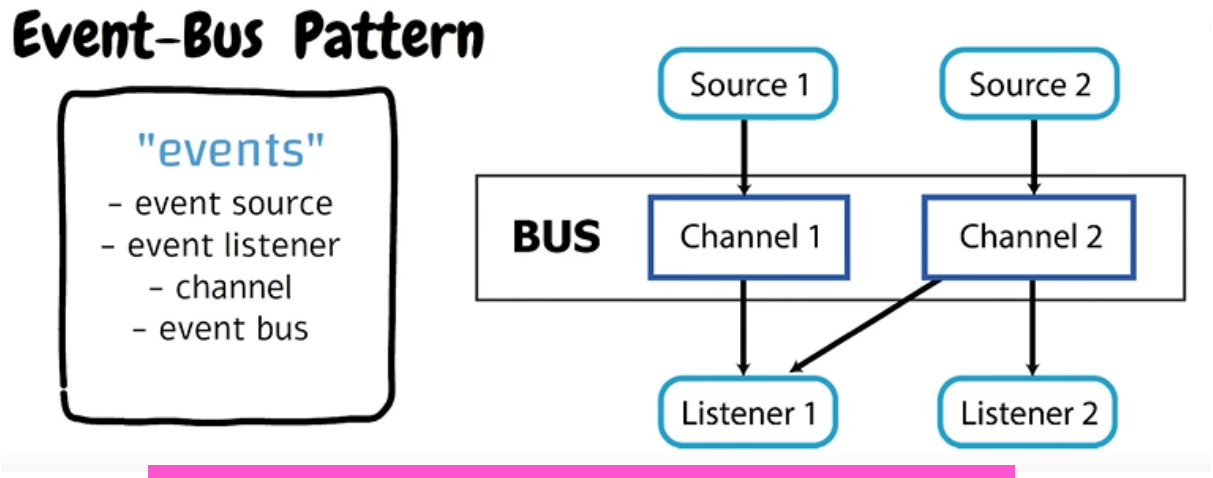
Module/architectural patterns:

- Layered:
  Divide system into layers and each lower layer is less complex in terms of the details:
    - presentation layer (user interface handling)
    - application layer
    - business logic layer
    - data access layer

  used in e-commerce web applications
  Lower layer can be used by all higher layers, but to fulfill a business request you need to go through multiple layers.
  When the uses relation among software elements is strictly unidirectional, a system of layers emerges. A layer is a coherent set of related functionality. In a strictly layered structure, a layer can only use the services of the layer immediately below it. Layers are often designed as

abstractions (virtual machines) that hide implementation specifics below from the layers above, engendering portability.

- Shared-data:
  Data is common, but components act on that data differently/with different purposes. In this architecture, data is considered a central, and shared resource that can be accessed and modified by different parts of the system.
  This pattern comprises components and connectors that create, store, and access persistent data. The repository usually takes the form of a (commercial) database. The connectors are protocols for managing the data, such as SQL.
  used in enterprise resource planning systems, customer relationship management systems, e-commerce platforms etc.
- Client-server:
  call to obtain data (request-response) : one server and multiple clients
  DOESN'T MEAN: SMALL SYSTEM - BIG SYSTEM, MEANS: ONLY ONE NEEDS INFORMATION THAT THE OTHER HAS.
  The components are the clients and the servers, and the connectors are protocols and messages they share among each other to carry out the system's work. Communication between clients and servers occurs over a network, using communication protocols such as HTTP. Servers listen for incoming requests on specific ports and respond accordingly.
  used in online/web applications like email, banking, document sharing, etc.
- Event-driven:
  sources publish messages on a particular channel on an event bus and listeners subscribe to channels



  listeners are notified of messages on a channel to which they have subscribed before
  e.g. andoid development

Allocation patterns:

1. Deployment: multi-tier, services, microservices
   This allocation pattern relates to the deployment of software components within a system. It involves organizing the components into multiple tiers or layers, each responsible for specific functionalities or services. The layers are typically arranged hierarchically, with each layer performing different tasks and communicating with adjacent layers through well-defined interfaces. This pattern helps in achieving modularity, scalability, and maintainability in software systems.

- Multi-tier Architecture :
  Logical and physical division of the system is possible. Multi-tier divides an application into logical layers or tiers, each responsible for a certain functionality: presentation tier (client tier), application tier(middle tier) and data tier(backend tier) → separation of concerns, scalability and maintainability.
  The presentation layer is responsible for user interface interactions, the application layer processes business logic, and the data layer manages data storage and retrieval. Multi-tier pattern, which describes how to distribute and allocate the components of a system in distinct subsets of hardware and software, connected by some communication medium. This pattern specializes the generic deployment (software-to-hardware allocation) structure.
- Services:
  Services architecture involves breaking down the system into smaller, self-contained services that communicate with each other over a network. Each service performs a specific function or provides a particular feature. Services are typically designed to be independent, loosely coupled, and reusable. This pattern enables flexibility, scalability, and interoperability in distributed systems.
- µServices:
  = smaller services that move a lot
  Microservices architecture takes the concept of services further by decomposing the application into even smaller and more granular services, known as microservices. Each microservice is responsible for a single business capability and can be developed, deployed, and scaled independently. Microservices communicate with each other through lightweight protocols such as HTTP/REST or messaging queues. This pattern enhances agility, scalability, and resilience in modern cloud-native applications

2. Human resources
   Competence center:
   A competence center is a specialized team or department within an organization that possesses expertise in a specific domain or area of knowledge. The competence center serves as a centralized hub of knowledge, skills, and resources related to its designated domain → ownership important.
   Owner is the one who pays the bill for the server, not where the server is anymore because everything is in the cloud.
   Competence center and platform, which are patterns that specialize a software system's work assignment structure. In competence center, work is allocated to sites depending on the technical or domain expertise located at a site. For example, user-interface design is done at a site where usability engineering experts are located. In platform, one site is tasked with developing reusable core assets of a software product line, and other sites develop applications that use the core assets.

Software Architectures:

Monoliths (1 rock of code)

= single process running on the same machine
Everything ends up being tied up and you have a blob where everything is connected: one server, one database and many clients.

Pros:

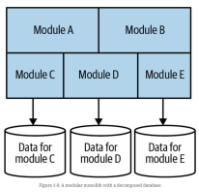
- Errors are detected earlier in the process
- Flexibility in terms of multiple purposes
- Modules connect via coupled direct calls
- Simpler deployment topology: everything needs to be ready before deployed
- Ease of coding
- Most used architecture!

Cons:

- Not flexible in terms of technology: multiple programming languages not allowed
- Not flexible in terms of ownership: You cannot have developers working on a part of the code without sharing the whole code
- Commit all or nothing, not one feature release possible
- Scales as a whole; scaling independent servers (because they are used more you want to give them more compute) not possible
- Latency issues possible/ Longer deployment times:
  Monolithic architectures could lead to higher latency compared to more modern architectures like microservices. Additionally, any changes or updates to the monolith may require redeploying the entire application, leading to longer deployment times and potentially increased latency.

Types:

1. Single-Process Monolith
   All of the code is deployed as a single process. You may have multiple instances of this process for robustness or scaling reasons, but fundamentally all the code is packed into a single process. In reality, these single-process systems can be simple distributed systems in their own right because they nearly always end up reading data from or storing data into a database, or presenting information to web or mobile applications.
2. Modular Monolith
   As a subset of the single-process monolith, the *modular monolith* is a variation in which the single process consists of separate modules. Each module can be worked on independently, but all still need to be combined together for deployment. Database can be decomposed along the same lines as the modules



3. Distributed Monolith
   A *distributed monolith* is a system that consists of multiple services, but for whatever reason, the entire system must be deployed together. A distributed monolith might well meet the definition of an SOA, but all too often, it fails to deliver on the promises of SOA. In my experience, a distributed monolith has all the disadvantages of a distributed system, *and* the disadvantages of a single-process monolith, without having enough of the upsides of either.

Service-oriented architecture SOA (multiple services on 1 DB):

Service-Oriented Architecture (SOA) is a software design approach that structures an application as a collection of loosely coupled, independent services. Each service performs a specific business function and can be independently deployed, scaled, and maintained. SOA aims to make it easier to maintain or rewrite software, as theoretically we can replace one service with another without anyone knowing, as long as the semantics of the service don't change too much. These services communicate with each other over a network using standardized protocols, such as REST or SOAP HTTP or messaging queues, or brokers:
- One request-One response
- Broker (I need something of this kind, I don't care who answers (server), the broker connects you to a particular server): can be used for load balancing, but flexibility is main purpose
- Message queues (place request in database and wait for response to be delivered there: asynchronous.

Pros:

- Modules are loosely coupled and can be independently deployed: very little communication between features. SOA has even fewer dependencies than microservices.
- Services need to be self-contained: few dependencies to other services.
- Services are stateless: give the whole answer to the request or wait until you have it because we don't want to remember which part we already gave
- black-box components: consumer only interacts through the API
- Services execute independently of one another: if one server crashes there is not necessarily a cascade of crashes, if accounting server crashes, Netflix isn't bothered
- Services provide an API and may depend on other services. Communication between these services occurs via calls across a network, rather than method calls within a process boundary.
- Services are described by metadata that specify how they interact.
- Services are modelled around a business domain.
- Services are designed with as a goal reusability/interoperability/scalabilty
- Flexibility and Agility: Services can be quickly adapted or replaced to meet changing business requirements without disrupting the overall system architecture.

Cons:

- Large financial investments
- Loads get very heavy
- Response times can be slow as loads increase

- Components need to interact with each other, which can create a lot of messages
- Enterprise service bus is a single point of failure
- Each service has a broad scope: if one service fails, the entire business workflow will be affected

The typical SOA framework relies on five horizontal layers:

1. Consumer interface layer
   users can access the features they need, set and save preferences: interface that separates users from the many SOA components
2. Business processes layer
   Creating a separate business processes layer makes SOA more agile. Instead of putting the features necessary for business processes at the core of an app's design, those features can sit independently between the services layer and consumers.
3. Services layer
   SOA's services layer includes components that fulfill critical functions, such as:
   - Access control
   - Policy management
   - Service clustering
   - Service runtime enablement
   - Service definition
4. Service component layer
   The service component layer contains the actual services that users want to access. Giving these components an independent layer separates technical business logic and IT needs from the user's experience. This layer has up to five types of ABBs:
   - Service realization and implementation
   - Service publication and exposure
   - Service deployment
   - Service invocation
   - Service binding

   Every SOA layer does something critical to the user's experience, but it's fair to see this as the core of functionality. Without these essential SOA components, nothing would happen.
5. Operational systems layer
   The operational systems layer includes the hardware and software necessary for SOA solutions to function. The layer has ABBs that manage service delivery, runtime environments, and virtualization and infrastructure services.
   The operational systems layer is also where your SOA connects to other systems, including databases and legacy software. Digitalization is an ongoing process, so you might need to continue relying on some aspects of these systems while you transition your technologies.

Microservice (each service coupled to its own DB)

Microservices is an architectural style for developing software applications as a collection of small, independent, and loosely coupled services. Each service is focused on a specific business function and can be developed, deployed, and scaled independently of other services. Microservices architecture promotes modularity, flexibility, and scalability, making it popular for building complex and distributed systems.

Pros:

- Scalable
- Domain-driven design: modeled around a business domain
- Distributed systems
- Technology heterogeneity
- Robustness: A component of a system may fail, but as long as that failure doesn't cascade, you can isolate the problem, and the rest of the system can carry on working.
- Easy scaling: With smaller services, we can scale just those services that need scaling, allowing us to run other parts of the system on smaller, less powerful hardware
- Owning their own state: if microservice wants to access data held by another microservice, it should go and ask that second microservice for the data: in that way the microservice can decide what is shared and what is hidden, which allows us to clearly separate functionality that can change freely (our internal implementation) from the functionality that we want to change infrequently (the external contract that the consumers use
- API calls to communicate -> network latency
- KPI monitoring
- Parallel deployment = independent deployability is the idea that we can make a change to a microservice, deploy it, and release that change to our users, without having to deploy any other microservices: limit backward-incompatible changes to our microservices. Hiding internal state in a microservice is analogous to the practice of encapsulation in object-oriented (OO) programming.
- Information hiding means hiding as much information as possible inside a component and exposing as little as possible via external interfaces. This allows for clear separation between what can change easily and what is more difficult to change.
- Size: it should be as big as your head: you should be able to handle it at all time.
- Flexibility: microservices buy you options
- Organizational alignment: better align architecture to organization + smaller teams working on smaller codebases → more productive
- Composability: One of the key promises of distributed systems and service-oriented architectures is that we open up opportunities for reuse of functionality. With microservices, we allow for our functionality to be consumed in different ways for different purposes.
- Microservices are designed with as a goal of integration and scalabilty
- Containerized
  Microservice architectures are usually containerized and managed by frameworks. This is no longer a distinguishing feature because everything is containerized these days: everything is on the cloud. A container is a package that you make on your computer: a package of code, necessary packages etc. and then you put it in the cloud. Hosting mechanism runs using all the configurations of the dependencies installed of the package. Container is a virtual environment only with the configurations. Often you sent 3 files: for database, for server, for configuration. Docker is thé framework for

containerization. Containers are a lightweight way to easily provision several instances. Containers are often managed by orchestration platforms like Kubernetes.

Cons:

- Developer experience, a collective ownership model is not possible
- Technology overload: Microservices give you the *option* for each microservice to be written in a different programming language, to run on a different runtime, or to use a different database
- Increase in cost (at least in short time): you'll likely need to run more things—more processes, more computers, more network, more storage, and more supporting software (which will incur additional license fees), changes that slow down your programming team
- Reporting: you might simply need to publish data from your microservices into central reporting databases (or perhaps less structured data lakes) to allow for reporting use cases.
- Monitoring and troubleshooting hard
- Security: more information flows over networks between our services
- Testing: the scope of our end-to-end tests becomes *very* large
- Latency: With a microservice architecture, processing that might previously have been done locally on one processor can now end up being split across multiple separate microservices. Information that previously flowed within only a single process now needs to be serialized, transmitted, and deserialized over networks that you might be exercising more than ever before. All of this can result in worsening latency of your system.
- Data consistency: The use of distributed transactions in most cases proves to be highly problematic in coordinating state changes.

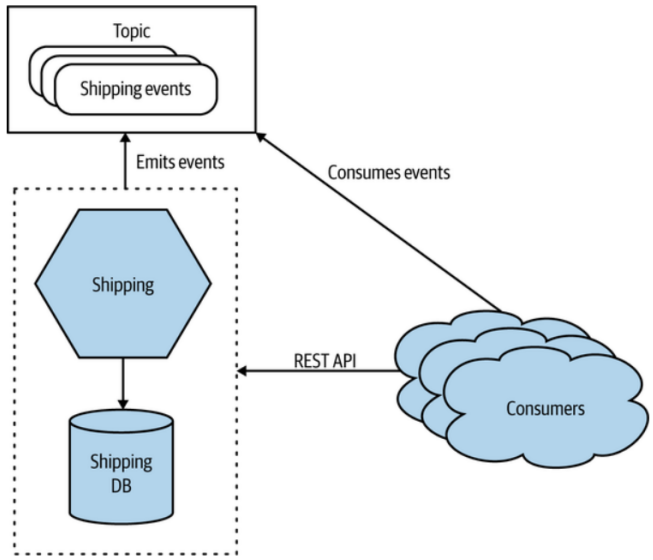


Figure 1-1. A microservice exposing its functionality over a REST API and a topic

Microservices allow efficient setup: change from horizontally layered architecture to vertical business lines

Horizontally layered architecture: Team is structured around capabilities

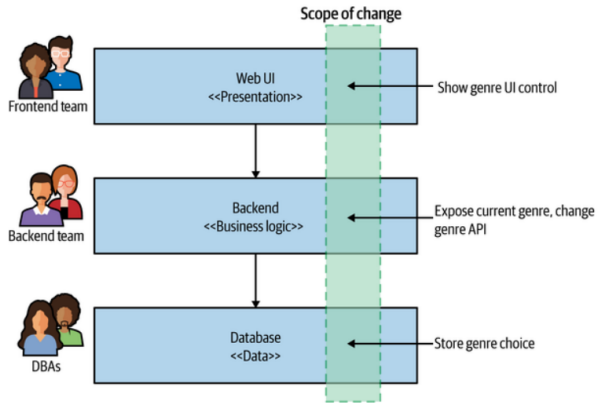
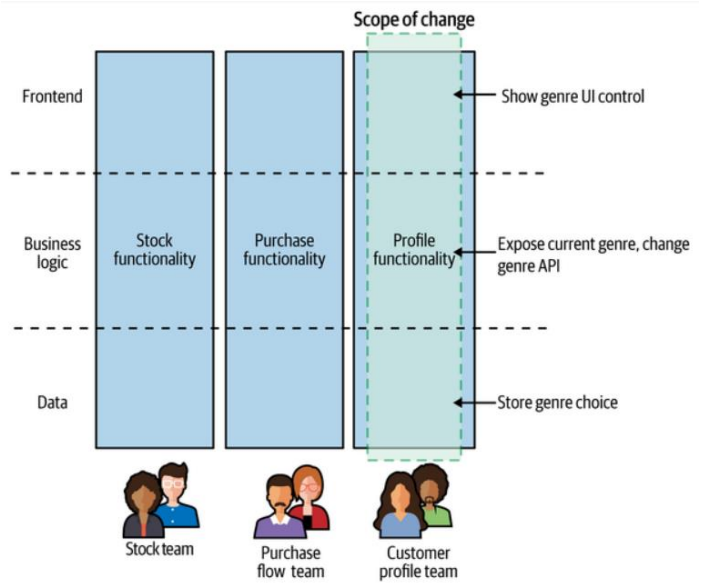


Figure 1-3. Making a change across all three tiers is more involved

To: vertical business lines

Our business domain becomes the primary force driving our system architecture, hopefully making it easier to make changes, as well as making it easier for us to align our teams to lines of business within the organization



Microservices use cases

- **Real-time data processing and analysis.** Microservices can handle real-time data processing and analysis, providing insights faster.
- **Batch processing.** By breaking down the task into smaller independent services, you can process multiple batches simultaneously.
- **Load balancing.** Microservices enable load balancing by scaling up only the services that experience heavy traffic, which is more resource-efficient.
- **Machine learning.** Different models or different parts of a model can be deployed as individual services, making the system more manageable and scalable.
- **Internet of Things (IoT).** Microservices can help handle this IoT data effectively by dividing tasks among multiple services.

When to use: SOA vs. microservices

Service-oriented architecture (SOA) and microservices provide different ways for organizations to migrate from a monolithic architecture to cloud environments. Depending on certain factors, one might be more suitable than the other in practical use cases.

SOA

Organizations with legacy or stand-alone enterprise applications benefit from the SOA architecture. SOA simplifies conventional software programs into smaller modular parts. It also pools shared resources to streamline business functionalities. Rather than building overlapping and redundant services, developers can reuse existing SOA services to implement more business solutions.

Microservices

Microservices architecture is the better option to support agile development teams. Developers can make rapid and incremental code changes without affecting the application's stability by using continuous integration and continuous delivery (CI/CD) tools. Microservices are better when developers have these goals:

- Use different programming languages, libraries, or frameworks to build a single application
- Combine individual services built with different software frameworks
- Provision compute resources and scale individual services in real time

With microservices, companies can benefit from modern cloud capabilities and deploy hundreds of microservices with ease.

Class 4

Interface technology

REST: REpresentational State Transfer

HTTP is a communication protocol. Rest is HTTP protocol but with convention on top of it. It follows an architectural style that promotes a simple and efficient way of providing and connecting web services.
Promotes the decoupling between data-centric server side applications and client user-centric applications
Simple, lightweight, runs almost on meta (no fluf, only necessary information)
Server has an internal state: schema that defines the structure and organization of its data. When accessing data from such a server, the interaction is typically operation-based, as seen in the SOAP (Simple Object Access Protocol) protocol. In a SOAP-based interaction, clients send requests to servers to perform specific operations or actions on the data stored on the server.
REST is a convention, so it is impossible to impose. It is an architectural style.
Restfull = REST as intended in the beginning, all that is REST is HTTP requests. If API uses HTTP it is not necessarily restful.

SOAP: Simple Object Access Protocol
= standard for communication based on XML: XML file stated what you wanted to do, the parameters and this is send to a server on the internet that unpacked the envelop and do the task, reply.
request should contain a lot of things
one or multiple entry points

Android phones supported SOAP: made in java, but handling SOAP requests required a lot of energy from the battery, therefore phones were big. 2007: Iphone came and didn't want SOAP on it, because envelopes too big: world had to change. Everybody was forced to support REST, even banks. REST technology appeared that took lightweight requests, answering was faster.

REST - Representational State Transfer
• Resource Based:
based on things/resources, not actions (like SOAP is): objects, files, services
use nouns instead of verbs to form the URL: it is a resource, not an action so don't call it 'open' or 'add'
resources are uniquely identified by URI (Uniform resource identifier) and can be manipulated using HTTP methods (get, post, put, delete)

• Representation:
Resources are represented in different formats, such as JSON, XML, HTML, or plain text. Clients interact with resources by exchanging representations, which convey the current or desired state of the resource. Representations allow clients and servers to communicate effectively and decouple the resource's internal structure from its external presentation. you can read source independently from the machine that is reading it, no binary format are within the convention of REST

• Uniform Interface:

- standard HTTP verbs (GET, PUT, POST, DELETE)
- standard HTTP response (status code, info in the response body)
- Uniform structure of URIs with a name, identifying the resource
- References inside responses must be complete

• Stateless
Each message must be self-contained in one request, server does not hold session state.
If system would allow you to do request half, it has to remember you, therefore it is more complicated; the idea of REST is to keep things as simple as possible

• Cacheable
Responses from servers can be explicitly marked as cacheable or non-cacheable using standard HTTP cache control mechanisms. Caching allows for improved performance and reduced network latency by serving cached responses to subsequent identical requests.
URL and request is the only thing needed to cache: all needs to be public. Bookmark is private: remember link for a resource.
In things that are not REST you have to reload. Do you want to resend the request/resubmit the file? That is because no GET is used.
POST requests are not cachable and bookmarkable.

• Client-Server
REST architectures separate the concerns of clients and servers, allowing them to evolve independently. Clients are responsible for the user interface and user experience, while servers are responsible for storing and manipulating resources and enforcing business logic.

• Layered System
RESTful systems are organized into layers, where each layer is responsible for a specific aspect of the architecture. Layers can add functionality such as security, caching, or load balancing without affecting the overall system architecture.
establishes an API between a client and a database. API on the outside can be used to create a database.

• Code on Demand (optional):
This constraint is optional in REST architectures. It allows servers to transmit executable code (such as JavaScript) to clients, which can then execute the code to extend or customize their behavior. This constraint is rarely used in practice due to security concerns and interoperability issues.

Twitter example:

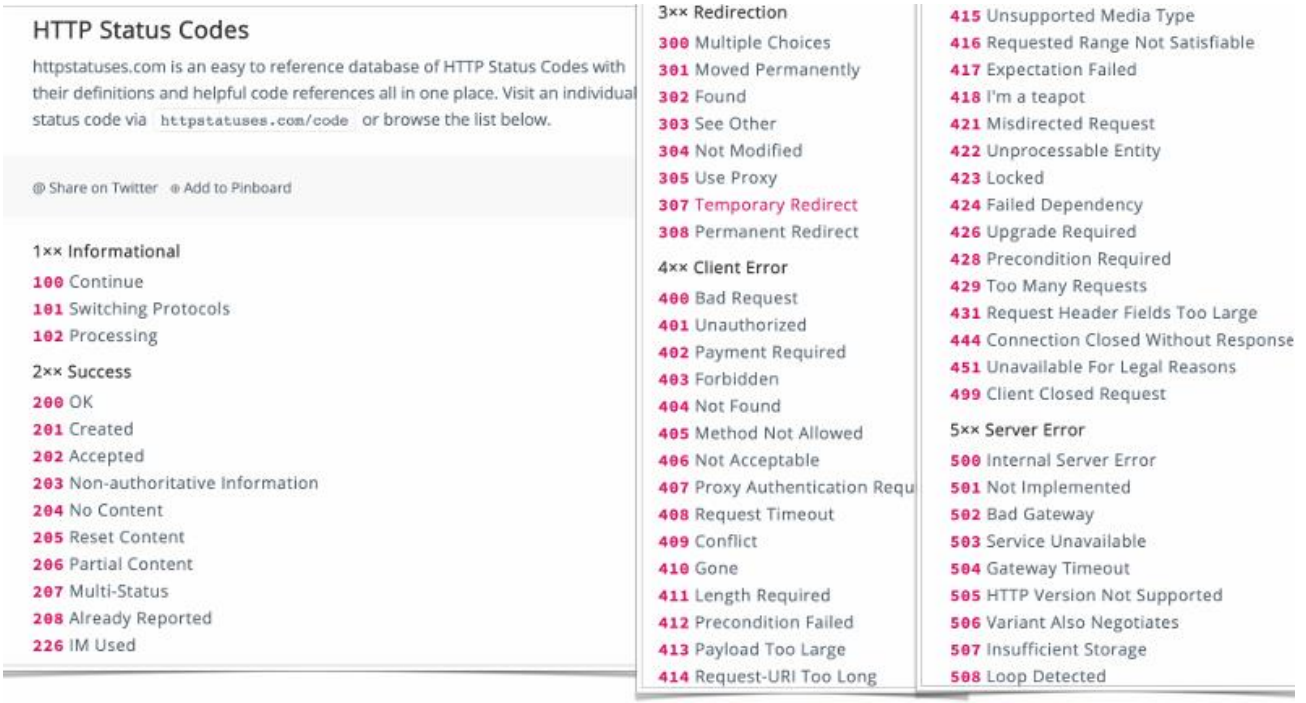
'tweets/search/stream/rules' = resource being requested:

- tweets: type of resource being accessed
- search: functionality related to searching or filtering the tweets
- sample: request is for a random sample of tweets. The Twitter API provides this feature to allow developers to access a representative subset of all tweets for analysis or monitoring purposes.
- stream: request is to stream real-time data
- rules: request is specifically related to managing rules for filtering the stream of tweets

GET = method
/2 = version of API being accessed
:id: is a placeholder representing the unique identifier of the tweet that will be hidden or unhidden
'/hidden': specifies the action performed on the tweet, here updating the status to hidden

RESTful designs are:

- based on a collection of resources/ a resource: you shall use nouns for those. Each URL corresponds to A COLLECTION of resources.
- URI = a path identifying resources and allowing actions on them
- you have URL methods representing the standardized actions: GET, POST (to create), DELETE, PUT (to update)
    1. GET = request resources: retrieve data from a specific resource
    2. POST = create resources
    3. PUT = update or create resources
    4. DELETE = deletes resource: means turn of a flag (Boolean value) saying this is deleted, since people who deleted something will often reconsider their decision and get it back.
- you have HTTP response codes = operation results
    - 10x informational: you don't see it because everything went well
    - 20x OK: you don't see it because everything went well
      203 means the thing is created: a convention
    - 30x redirection (not modified, without seeing it)
    - 40x client error
      400 bad request: invalid request body
      401 unauthorized: I don't know who you are
      403 Forbidden: I know who you are but you shouldn't be here
      404 not found
    - 5xx server error
- Searching, sorting, filtering and pagination obtained by query string parameters
- Text based data format (json or xlm)



Semi Structured data modelling:

- The data model of document-based NoSQL databases (e.g. MongoDB)
  Document-based NoSQL databases organize data in a document-oriented manner. In these databases, data is stored as JSON-like documents, which are self-contained data structures that can contain nested fields, arrays, and key-value pairs
- Flavours collections of hierarchical data (objects) where ownership is well represented
  Each document represents a single entity, and these documents can be grouped into collections based on common characteristics or types. For example, a collection may contain documents representing users, products, or orders.
- Ownership representation
- Relations between objects can be achieved with keys (identifiers)
- Collections can be indexed for better performance

```
const initialCompanies =
    [
        {name:"EDP",
         contacts:
            [
                {name:"one at edp", email:"one@edp"}
                 , { name:"two at edp", email:"two@edp"}
                 , { name:"three at edp", email:"three@edp"}
            ]
        }
    ,   {name:"RTP",
         contacts:
            [
                {name:"one at rtp", email:"one@rtp"}
            ,   {name:"two at rtp", email:"two@rtp"}
            ,   {name:"three at rtp", email:"three@rtp"}
            ]
        }
    ]
```

Semistructured Data modelling in REST:
Why? JSON: different levels of information: list of things inside list. Semi-structured requests important because tables might not be flat such as in SQL

• Collection- and Object- based data types
Collection-based data types represent a group of similar entities, such as a list of users or products, while object-based data types represent individual entities with their own attributes, such as a single user or product.

• Ownership relations by inclusion
In semistructured data modeling, ownership relations between entities are often represented by inclusion, where an object contains or includes other objects or collections. For example, a user object may contain a collection of posts or comments that the user has created. This inclusion-based approach allows for hierarchical relationships between entities.

• Other relations may use direct object references
While ownership relations are typically represented by inclusion, other types of relationships between entities may use direct object references. For example, a comment object may include a reference to the user who posted the comment, rather than embedding the entire user object within the comment object. This allows for more flexible and efficient representation of relationships.

- Flexible structure of data schema

Semistructured data modeling allows for a flexible structure of data schema, where entities can have varying attributes and relationships. Unlike traditional relational databases with rigid schemas, RESTful APIs for semistructured data can accommodate changes and variations in data structure over time without requiring modifications to the schema. This flexibility is particularly useful in scenarios where data is evolving or where different clients may have different data requirements.

EXAM QUESTION:
Example of data modelling in REST:
Application to manage contracts of partner companies (e.g. for security clearance in events)
Resources: with their attributes like Name, address, email,…

- Companies: name, address, email, list of contracts (employees))
- Employees (contacts) : name, email, job, company

Operations (CRUD): what do you want to do? Add, list, update, and delete resources

Partner companies

- `GET /companies` - List all the companies
- `GET /companies?search=<criteria>` - List all the companies that contain the substring `<criteria>`
- `POST /companies` - Create a company described in the payload. The request body must include all the necessary attributes.
- `GET /companies/{id}` - Shows the company with identifier {id}
- `PUT /companies/{id}` - Updates the company with {id} having values in the payload. The updatable items may vary (name, email, etc.)
- `DELETE /companies/{id}` - Removes the company with {id}

Partner contacts

- `GET /contacts` - List all the contacts
- `GET /contacts?search=<criteria>` - List all the contacts that contain the substring `<criteria>`
- `POST /contacts` - Create a contact described in the payload. The request body must include all the necessary attributes.
- `GET /contacts/{id}` - Shows the contact with identifier {id}
- `PUT /contacts/{id}` - Updates the contact with {id} having values in the payload. The updatable items may vary (name, email, etc.)
- `DELETE /contacts/{id}` - Removes the contact with {id}

Partner contacts

- `GET /contacts` - List all the contacts
- `GET /contacts?search=<criteria>` - List all the contacts that contain the substring `<criteria>`
- `POST /contacts` - Create a contact described in the payload. The request body must include all the necessary attributes.

! Then hierarchy not clear

- `GET /contacts/{id}` - Shows the contact with identifier {id}
- `PUT /contacts/{id}` - Updates the contact with {id} having values in the payload. The updatable items may vary (name, email, etc.)
- `DELETE /contacts/{id}` - Removes the contact with {id}

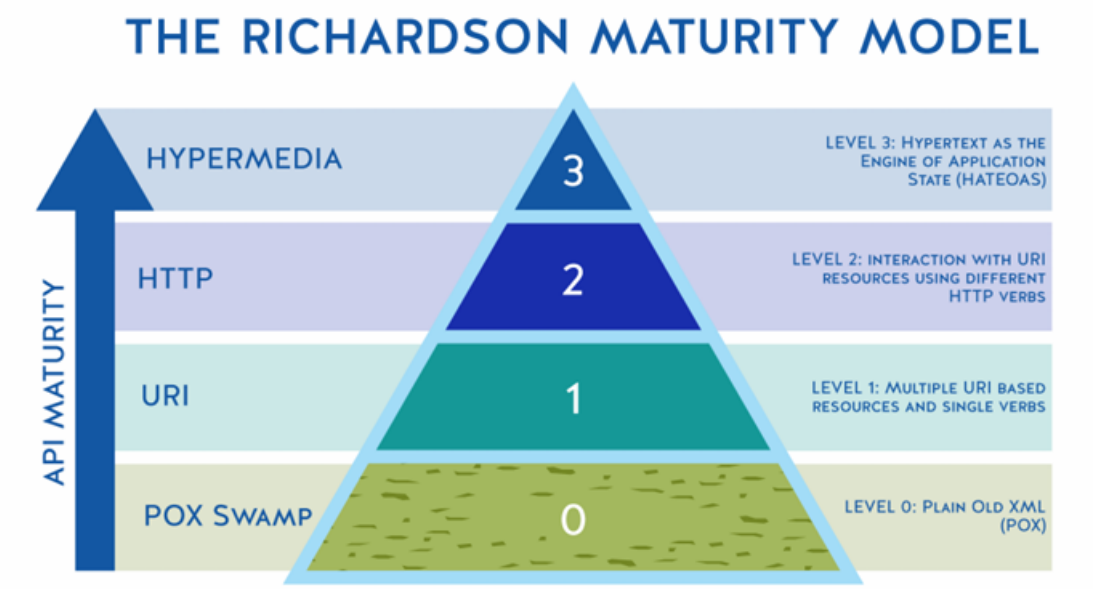
Partner contacts of companies

- `GET /companies/{id}/contacts` - List all the contacts of a company
- `GET /companies/{id}/contacts?search=<criteria>` - List all the contacts of a company that contain the substring `<criteria>`
- `POST /companies/{id}/contacts` - Create a contact of company {id} described in the payload. The request body must include all the necessary attributes.
- `GET /companies/{id}/contacts/{cid}` - Shows the contact of company {id} with identifier {cid}
- `PUT /companies/{id}/contacts/{cid}` - Updates the contact with {cid} of company {id} having values in the payload. The updatable items may vary (name, email, etc.)
- `DELETE /companies/{id}/contacts/{cid}` - Removes the contact with {id}

↳ still want to access contakt directly

the three in other color:
GET /companies/{id}/contacts/{cid} is not necessary: you know which contact you want to get, so you can go directly instead of through the company
GET /contacts{cid}

Richardson maturity model



Scale:
0 is not REST: **POX Swamp** = plain old XML kind of requests : everything packed in same XML, not different endpoints used
1: **URI** different endpoints but not cachable: breaks half of the rules of the conventions. Multiple URI based resources and single verbs
2: **HTTP** what we have seen so far: interaction with URI resources using different HTTP verbs
3 **HYPERMEDIA** is pure REST: navigation of the responses possible: links in the files so going from one page to another is like clicking on a link. Hypertext as the engine of application state (HATEOAS)

GraphQL

GraphQL is a query language for APIs that offers a more fine-grained approach to data retrieval compared to traditional RESTful APIs. In a RESTful architecture, clients typically retrieve data through fixed endpoints by making separate requests for each piece of information they need. This can lead to inefficiencies, as clients may need to make multiple requests to gather all the necessary data, resulting in increased network overhead and latency. Additionally, if the server-side implementation lacks awareness of the client's specific data requirements, it may return more data than needed, leading to overfetching.

In contrast, GraphQL allows clients to specify exactly what data they need by sending a query to the GraphQL endpoint. With GraphQL, clients can tailor their queries to request only the fields and relationships they require, avoiding the need for multiple round-trip requests. This results in more efficient data retrieval, as clients receive precisely the data they requested, minimizing unnecessary data transfer and improving performance.

Furthermore, GraphQL is not tied to any specific database or storage engine, giving developers the flexibility to use a variety of data sources and integrate with existing systems seamlessly. This makes GraphQL suitable for a wide range of use cases and environments, from simple applications to complex enterprise systems.

GraphQL is supported by a server-side runtime that executes queries and resolves the requested data from the underlying data sources. This runtime provides the necessary infrastructure for processing queries efficiently and delivering the requested data in a timely manner.

Overall, GraphQL offers a more flexible, efficient, and tailored approach to data retrieval compared to RESTful APIs, making it a compelling choice for modern API development. While REST remains a steady and widely adopted architectural style, GraphQL addresses some of the limitations and challenges associated with traditional RESTful APIs, particularly in scenarios where fine-grained control over data retrieval is essential.

Example:I want a hero, with a name and a list of friends (this is the query that is semi-structured)

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

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