

# Internet Applications Design and Implementation

## (Lecture 2 - Software Architecture)

**MIEI - Integrated Master in Computer Science and Informatics  
Specialization block**

**João Costa Seco ([joao.seco@fct.unl.pt](mailto:joao.seco@fct.unl.pt))**

(with previous participations of Jácome Cunha ([jacome@fct.unl.pt](mailto:jacome@fct.unl.pt)) and João Leitão ([jc.leitao@fct.unl.pt](mailto:jc.leitao@fct.unl.pt)))

# Outline

---

- Software Architecture - Introduction
- Software Architecture for Internet Applications
  - Three-tier architecture
  - Service-based architectures
  - Microservice-based architectures
- Frameworks at the service of Software Architecture
- The architectural style REST to instantiate webservices

# Internet Applications Design and Implementation

## (Lecture 2, Part 1 - Software Architecture - Introduction)

**MIEI - Integrated Master in Computer Science and Informatics  
Specialization block**

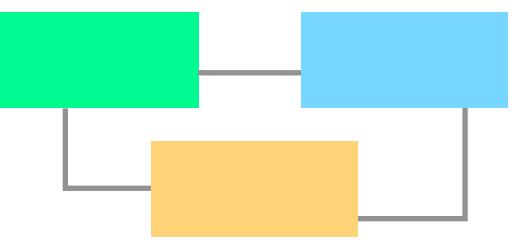
**João Costa Seco ([joao.seco@fct.unl.pt](mailto:joao.seco@fct.unl.pt))**

(with previous participations of Jácome Cunha ([jacome@fct.unl.pt](mailto:jacome@fct.unl.pt)) and João Leitão ([jc.leitao@fct.unl.pt](mailto:jc.leitao@fct.unl.pt)))

# Software Architecture

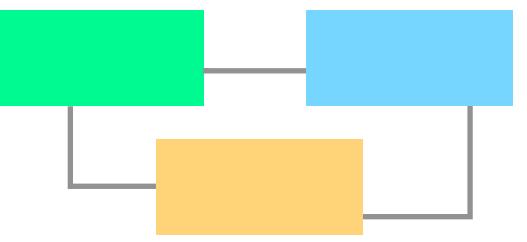
Introduction

based on the book “Software Architecture in Practice”



# Software Architecture

- What is software architecture?
- What are the benefits of using one?

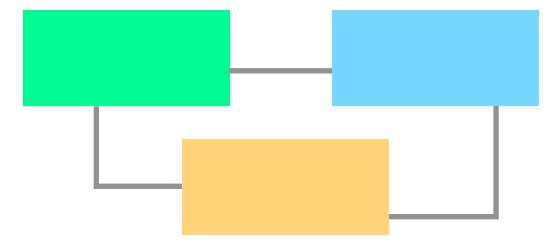


# Software Architecture

- What is software architecture?

“The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.” (in Software Architectures in Practice)

- Structures can be:
  - the module decomposition structure, which divide computational responsibilities and work assignment (among teams). Identify modules or components
  - runtime structures (connectors, e.g. HTTP, bus, mailboxes) that deal with the communication between components (e.g. services)
  - organisational structures. How components are developed, tested, deployed



# Software Architecture...

... is a form of Abstraction  
(different views over the same system)

... is in every software system  
(from caos to tidy)

... includes Behaviour  
(names and connectors have semantics)

**Not All Architectures Are Good Architectures**

# Software Architecture

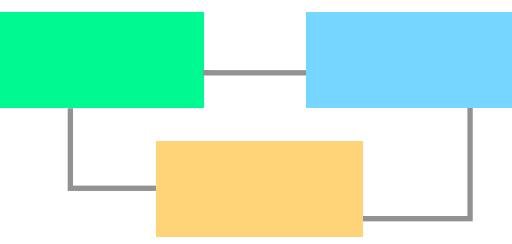
## What is it? Why do we need it?

Good organisation of the internal structure of systems leading to:

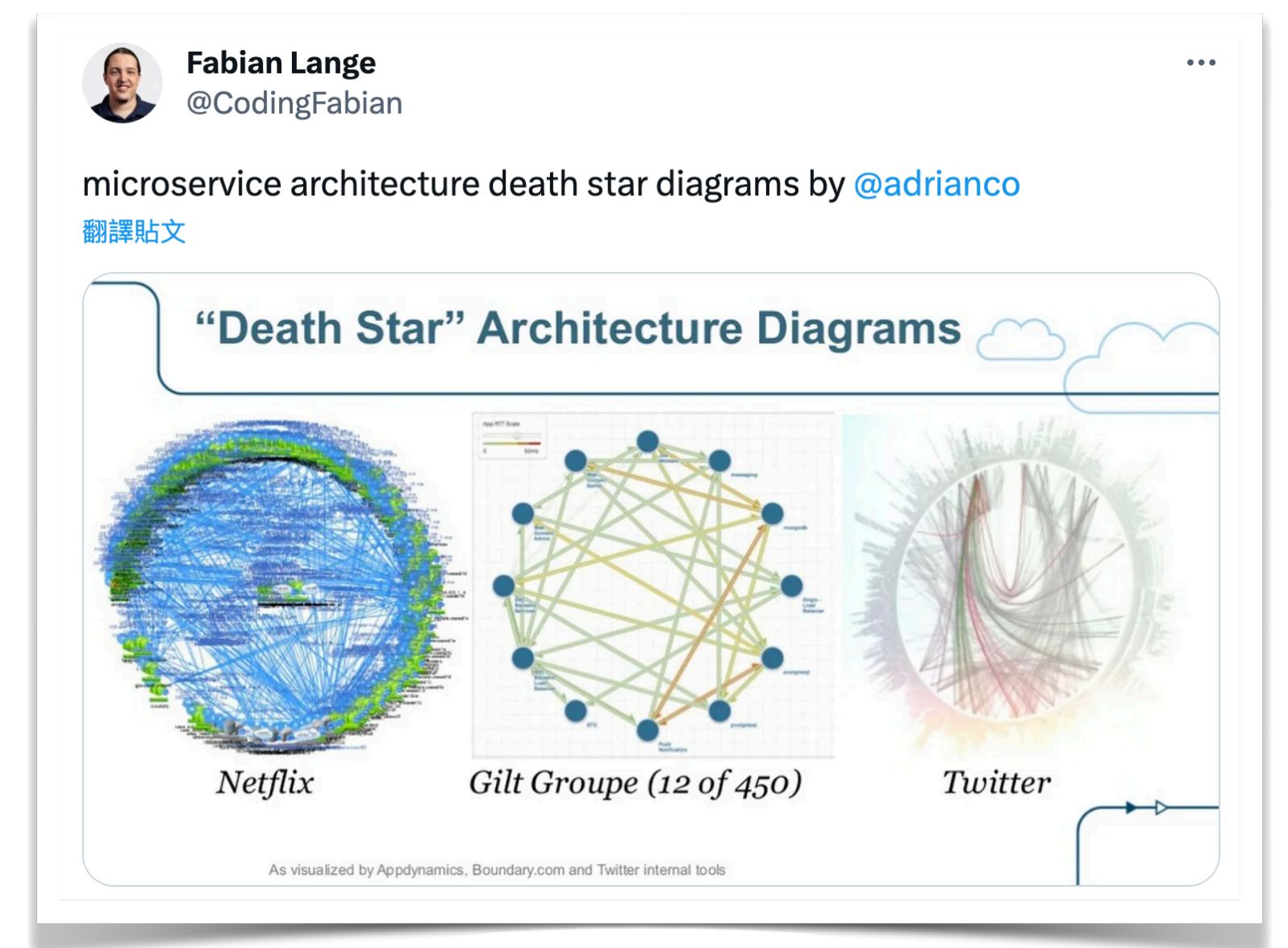
- Productive development
- Easy evolution
- Easy maintenance
- Trustworthiness  
(Functional Correctness, Security)
- High-Performance

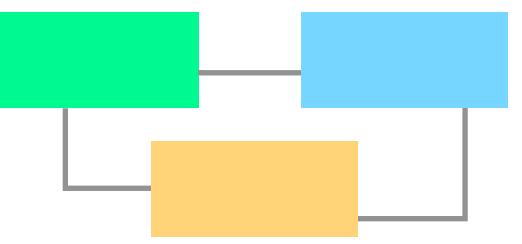


# Software Architecture (Structures)

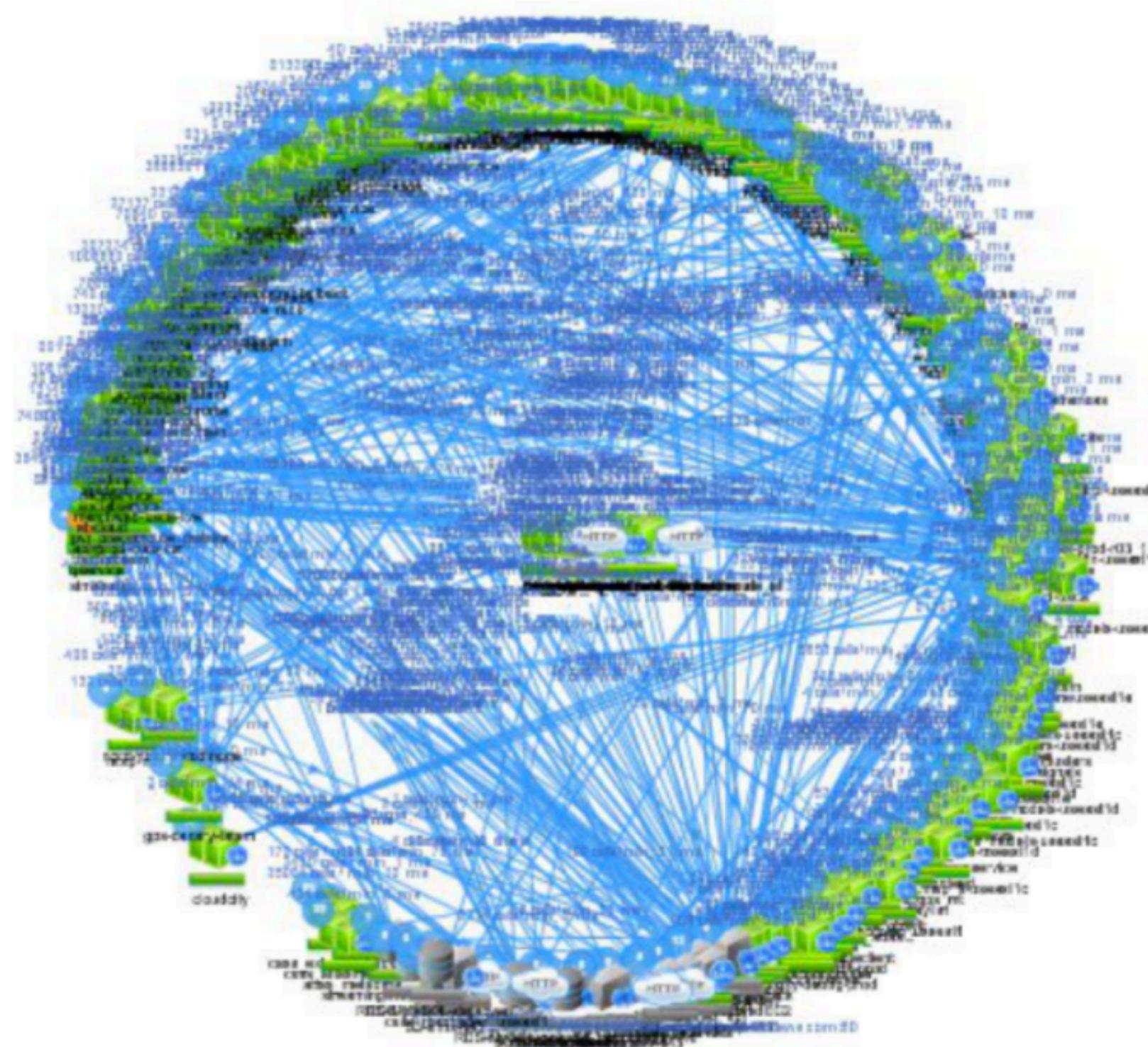


- Module structures
  - What is the primary functional responsibility assigned to each module?
  - What are the dependencies to other software elements?
  - What modules are each module related to? by generalisation or specialisation.
- Component and connector structures
  - How do modules interact?
  - What are the shared data stores?
  - What is the data flow in the system?
  - Can the structure change? how?
  - What are the security requirements? performance bottlenecks?
- Allocation structures
  - What is the hardware/cloud infrastructure? module ownership? which are the regressions tests? etc.

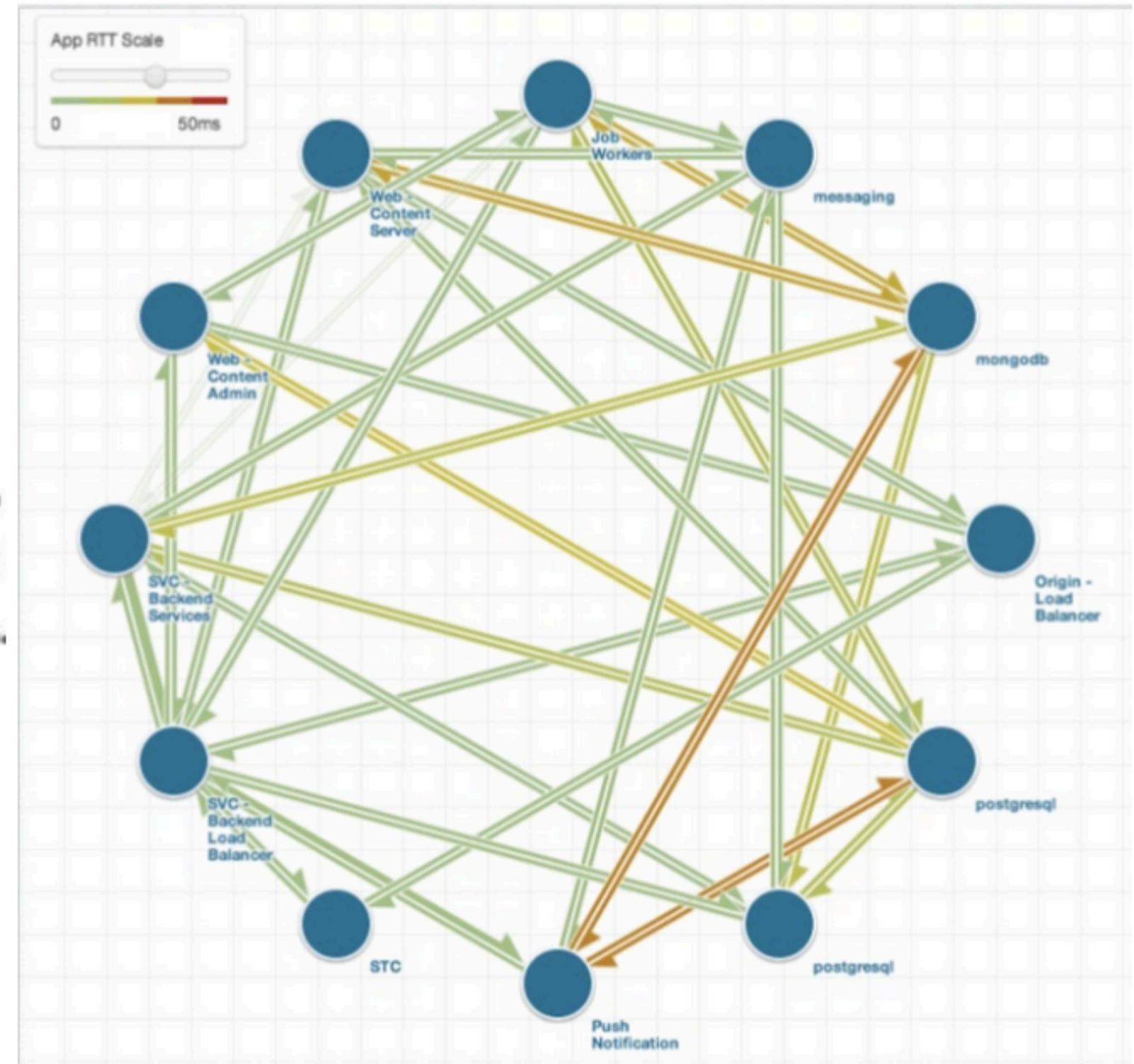




# Death-start architecture diagrams



*Netflix*

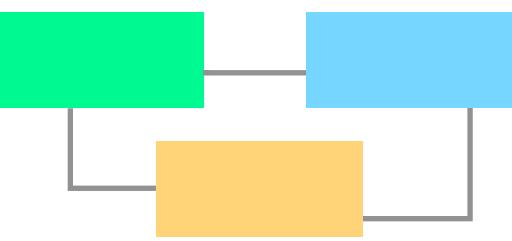


*Gilt Groupe (12 of 450)*

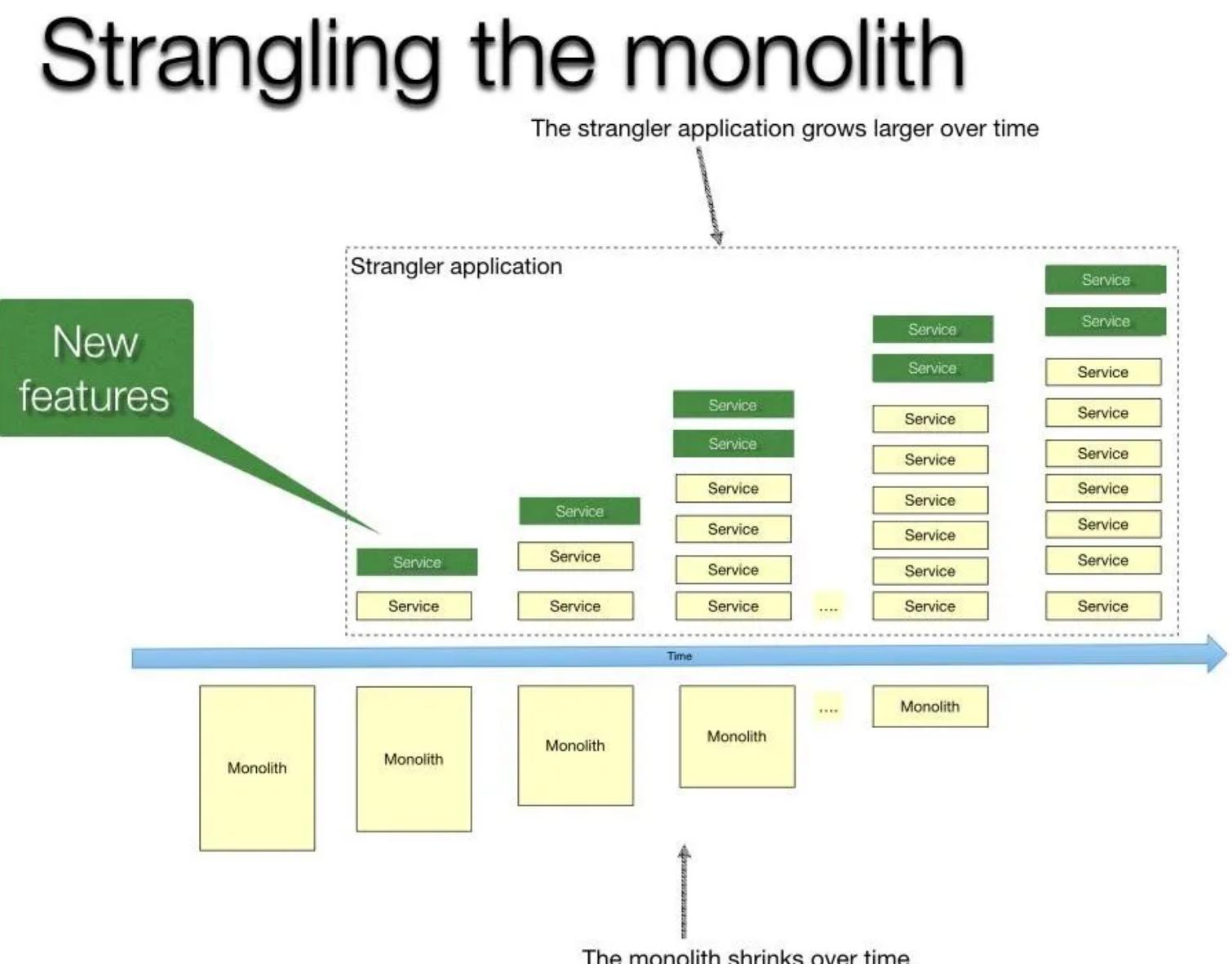
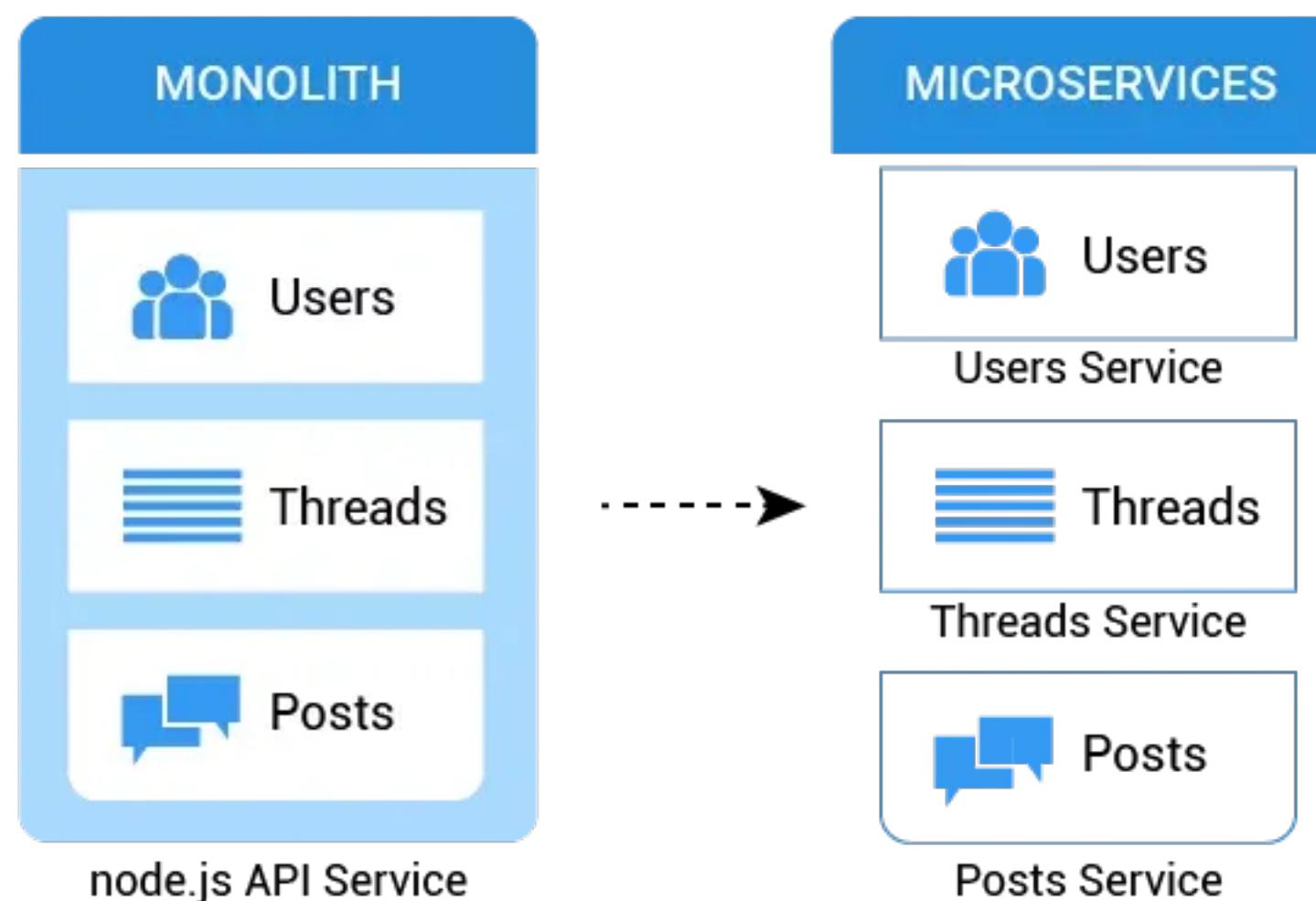


*Twitter*

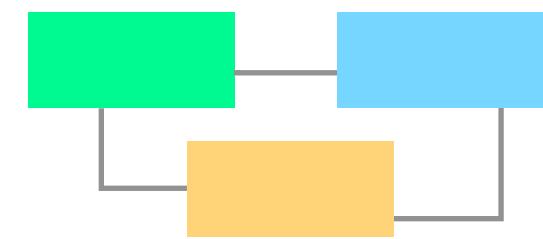
# From Monoliths to MicroServices



- Design from scratch, aim for separation of data domains.
- Changing existing applications requires a careful step-by-step process.
- Tension between database keys and API IDs (use natural keys)
- Loss of data consistency and (maybe) synchronicity



# Evolution is hard in MicroServices



- Any change to an interface may be a breaking change. Many are adaptable.
- Versioning is usually manual

**Type-Safe Evolution of Web Services**

Publisher: IEEE [Cite This](#) [PDF](#)

João Campinhos ; João Costa Seco ; Jácome Cunha [All Authors](#)

2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)

## Type-Safe Evolution of Web Services

João Campinhos, João Costa Seco, Jácome Cunha  
NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa, Portugal  
j.campinhos@campus.fct.unl.pt, joao.seco@fct.unl.pt, jacome@fct.unl.pt

**Abstract**—Applications based on micro or web services have had significant growth due to the exponential increase in the use of mobile devices. However, using such kind of loosely coupled interfaces provides almost no guarantees to the developer in terms of evolution. Changes to service interfaces can be introduced at any moment, which may cause the system to fail due to mismatches between communicating parts.

In this paper, we present a programming model that allows the development of web service applications, server end-points and their clients, in such a way that the evolution of services' implementation does not cause the disruption of the client. Our approach is based on a type based code slicing technique that ensures that each version only refers to type compatible code, of the same version or of a compatible version, and that each client request is redirected to the most recent type compatible version implemented by the server.

We abstract the notion of version and parametrize type compatibility on the relation between versions. The relation between versions is tagged with compatibility levels, so to capture the common conventions used in software development. Our implementation allows multiple versions of a service to be deployed simultaneously, while reusing code between versions in a type safe way. We describe a prototype framework, based on code transformation, for server-side JavaScript code, and using

approaches above, we generalize the kind of safe variability presented in [4] in a language based approach.

To avoid the kind of disruption described above we introduce a novel programming model that supports the smooth evolution of service interfaces and corresponding implementations. We adopt a language based approach that makes it possible for several versions of a service to coexist in the same running system, to be defined by the same source code, to share and reuse functionality, and yet enjoy type soundness as separate slices of code. This approach extends the classic notions of software variability, where the same source can be used to produce different versions of the same program, but only one can be running at a given time [7], [9], [4]. Technically, version tag modifiers are attached to service interface declarations (in the routing component in an application server) and also to program declarations (e.g. variables, functions), which are used to support a version sensitive method dispatching and execution mechanism. The system's code is pre-compiled using the prescribed versioning policies, where binding of identifiers is statically resolved,

**Robust Contract Evolution in a TypeSafe MicroService Architectures**

João Costa Seco<sup>a</sup>, Paulo Ferreira<sup>b</sup>, Hugo Lourenço<sup>b</sup>, Carla Ferreira<sup>a</sup>, and Lúcio Ferrão<sup>b</sup>

<sup>a</sup> NOVA LINCS, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa, Portugal  
<sup>b</sup> OutSystems, Portugal

**Abstract** Microservice architectures allow for short deployment cycles and immediate effects, but offer no safety mechanisms for service contracts when they need to be changed. Maintaining the soundness of microservice architectures is an error-prone task that is only accessible to the most disciplined development teams. The strategy to evolve a producer service without disrupting its consumers is often to maintain multiple versions of the same interface and dealing with an explicitly managed handoff period and its inherent disadvantages.

We present a microservice management system that statically verifies service interface signatures against

**★ A Language-Based Version Control System for Python**

We extend prior work, on a language-based approach to versioned software development, to add support for versioned programs with mutable state and evolving method interfaces.

Unlike the traditional approach of mainstream version control systems, where each evolution step is represented by a textual diff, we treat versions as programming elements.

Each evolution step, merge operation, and version relationship, is represented explicitly in code. This provides static guarantees for safe code reuse from previous versions, as well as forward and backwards compatibility between versions, allowing clients to use newly introduced code without needing to manually refactor their program.

By lifting the versioning to the language level, we pave the way for tools that interact with software repositories to have more insight regarding the evolution of a system's behaviour.

We instantiate our work in the Python programming language and demonstrate its applicability in regards to common evolution and refactoring patterns found in different versions of popular Python packages.

 **Luís Carvalho**  
NOVA School of Science and Technology

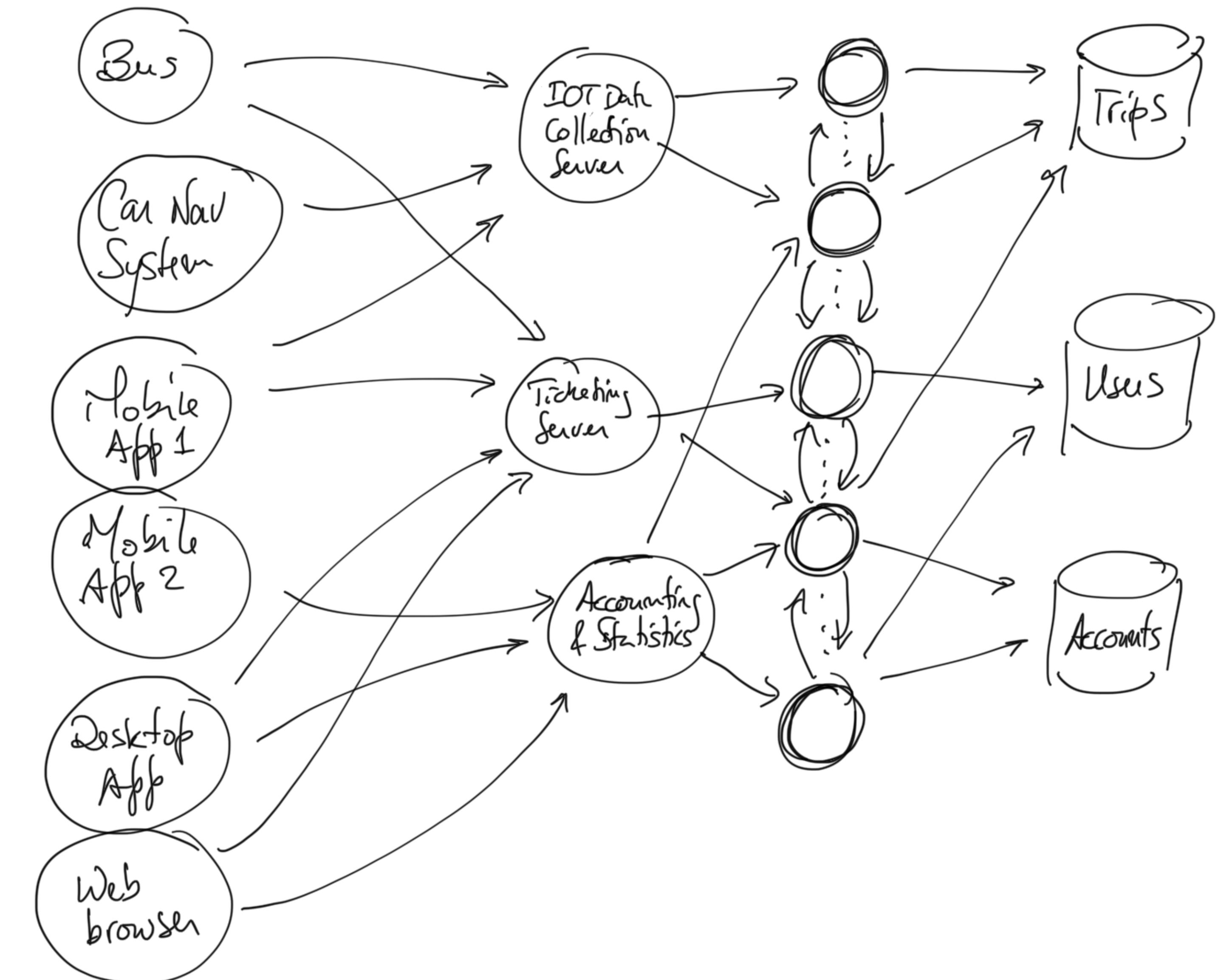
 **João Costa Seco**  
NOVA-LINCS; Nova University of Lisbon  
Portugal

Clients ← → Web Servers ← → Services ← → Data Storage

# Break Down

## Partition is related to

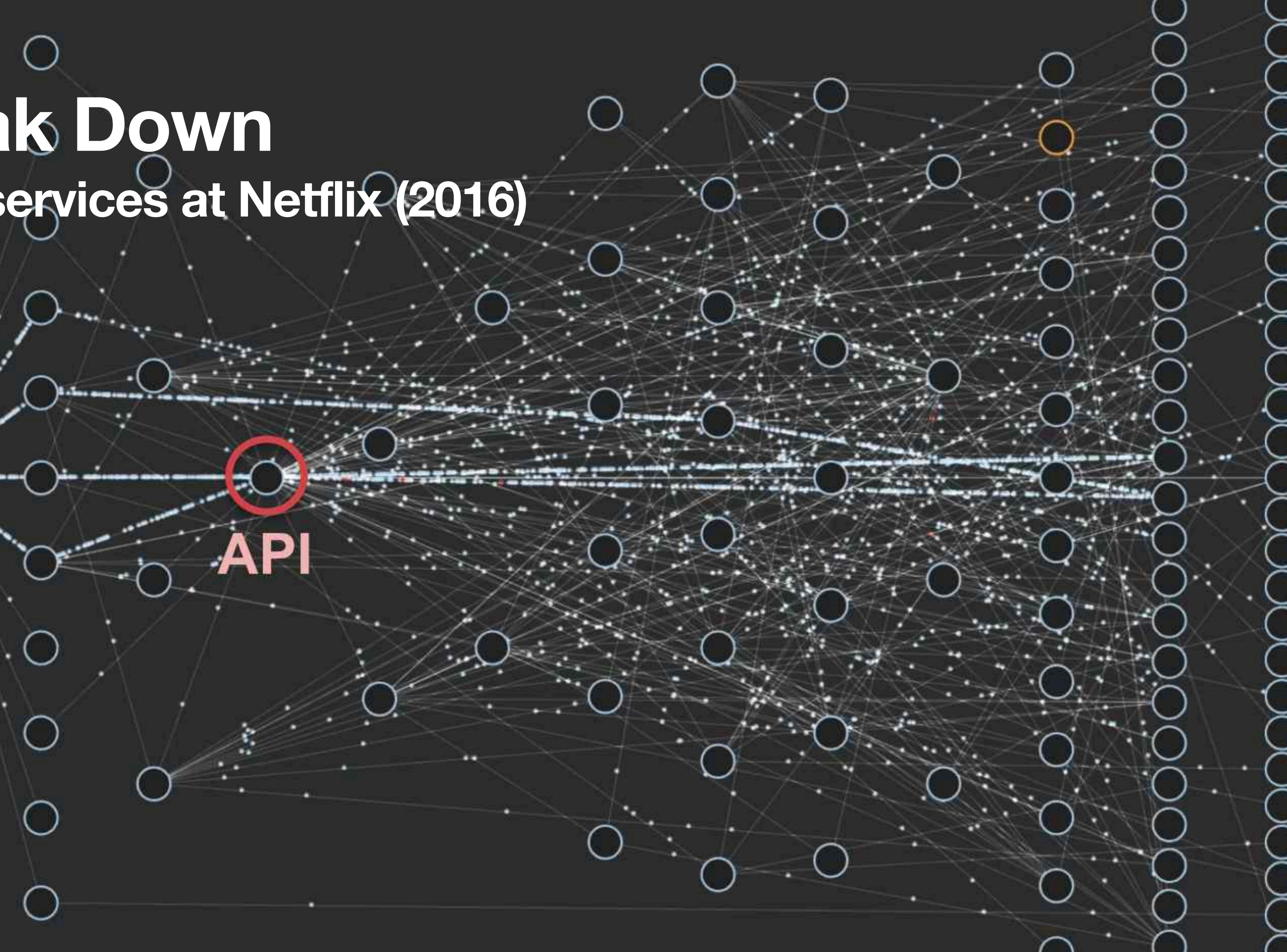
- Abstraction/Reuse
- Different Concerns
- Complexity
- Technology
- Ownership
- Performance

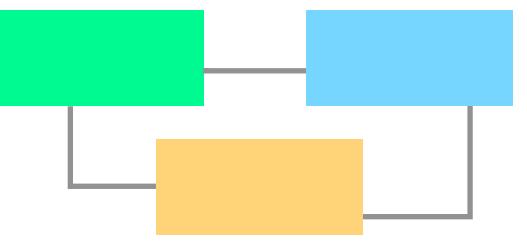


# Break Down Microservices at Netflix (2016)

ELB

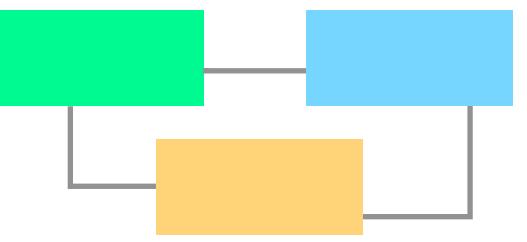
API





# Architectural Patterns

- Pre-determined compositions of architectural elements provide strategies for solving common problems in software systems.
- Examples:
  - Layered pattern — Linear (unidirectional) dependencies between multiple elements
  - Shared-data pattern — Components and connectors to create and manipulate persistent data, connectors are languages like SQL.
  - Client-server pattern — Components: Clients and servers; Connectors: protocols and languages
  - Multi-tier pattern — A generic deployment structure of components in different infrastructures
  - Competence center — Work assignment division by expertise (eg. departments)



# More Architectural Patterns

"The architectural pattern captures the design structures of various systems and elements of software so that they can be reused. During the process of writing software code, developers encounter similar problems multiple times within a project, within the company, and within their careers. One way to address this is to create design patterns that give engineers a reusable way to solve these problems, allowing software engineers to achieve the same output structurally for a given project."

The screenshot shows a website layout with the Red Hat logo at the top. Below it, there are two main article cards.

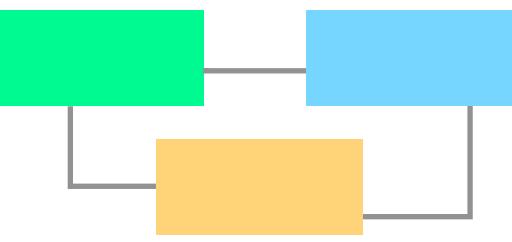
**Left Article:**

- Title:** 5 essential patterns of software architecture
- Description:** Software is essential. What are the main architectural patterns used to create the software we all rely on daily?
- Posted:** December 16, 2020 | Anand Butani

**Right Article:**

- Title:** 14 software architecture design patterns to know
- Description:** Architectural patterns increase your productivity: These reusable schemes address common software design challenges.
- Posted:** March 16, 2022 | Vicki Walker (Editorial Team, Red Hat)

# 5 architectural Patterns of software architecture



- Model-view-controller pattern
- Microservices pattern (by [Martin Fowler](#))  
(Aggregator pattern, API gateway design pattern, chain of responsibility pattern, branch pattern, and asynchronous messaging design pattern)
- Client-server pattern
- Controller-responder pattern  
“The controller component distributes the input or work among identical responder components and generates a composite result from the results generated from each responder.”  
- components may replicate data from the controller (writer)
- Layered pattern (the most common among developers)

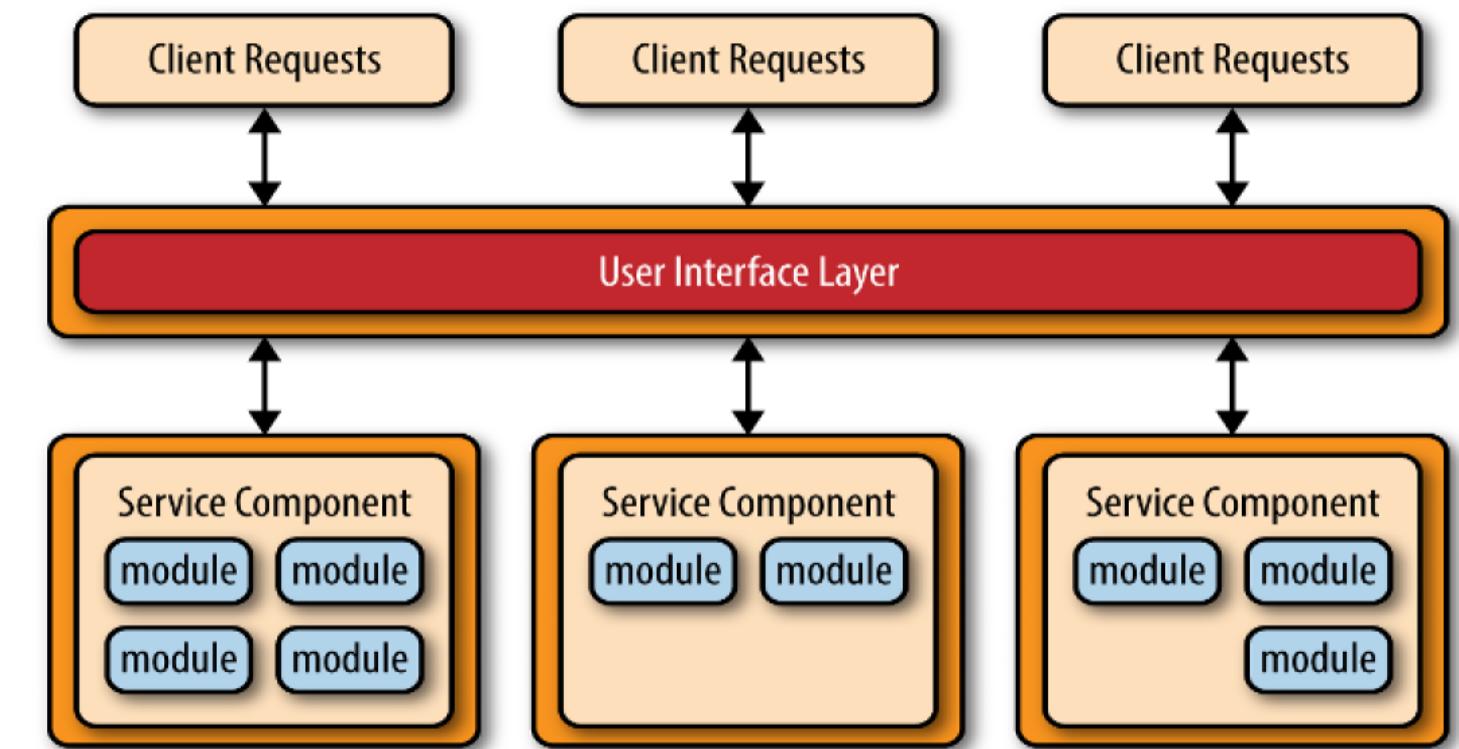
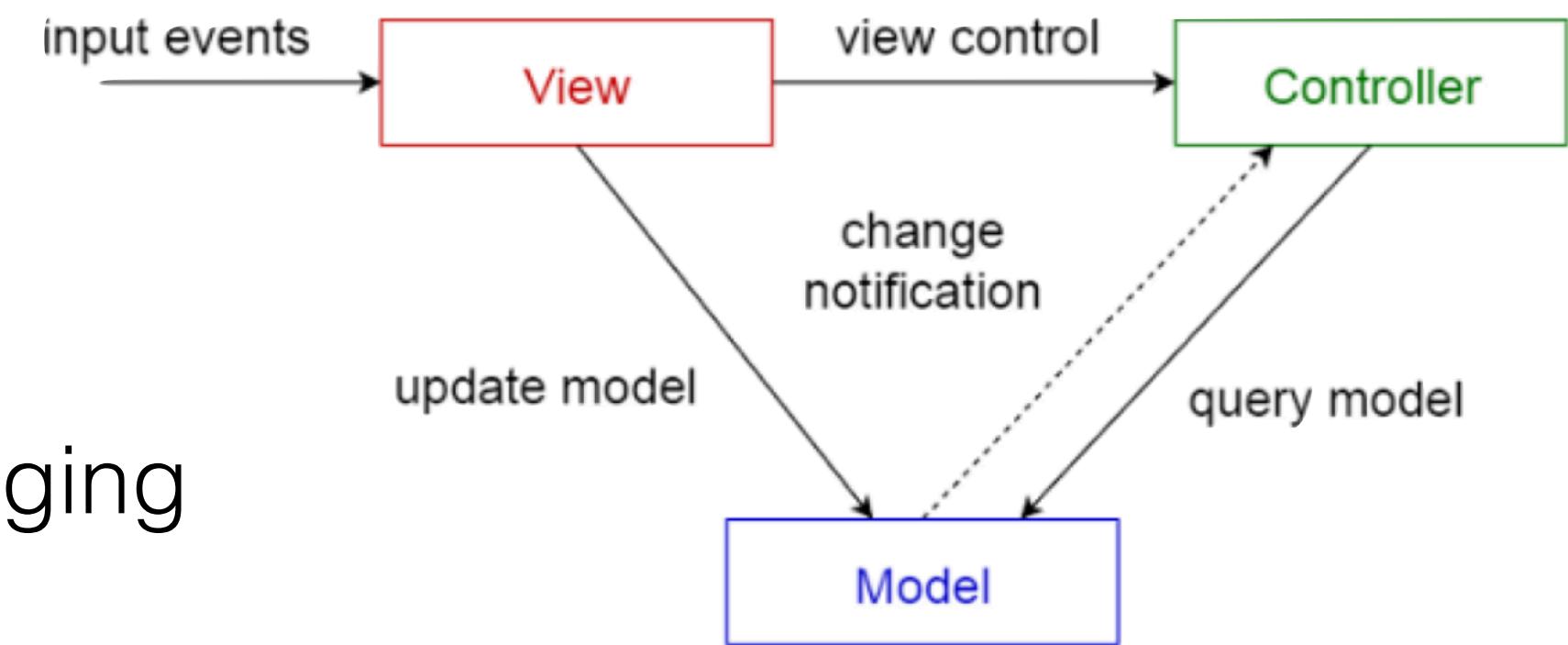
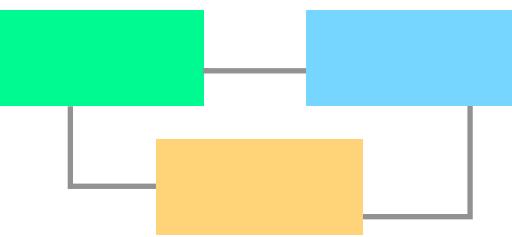
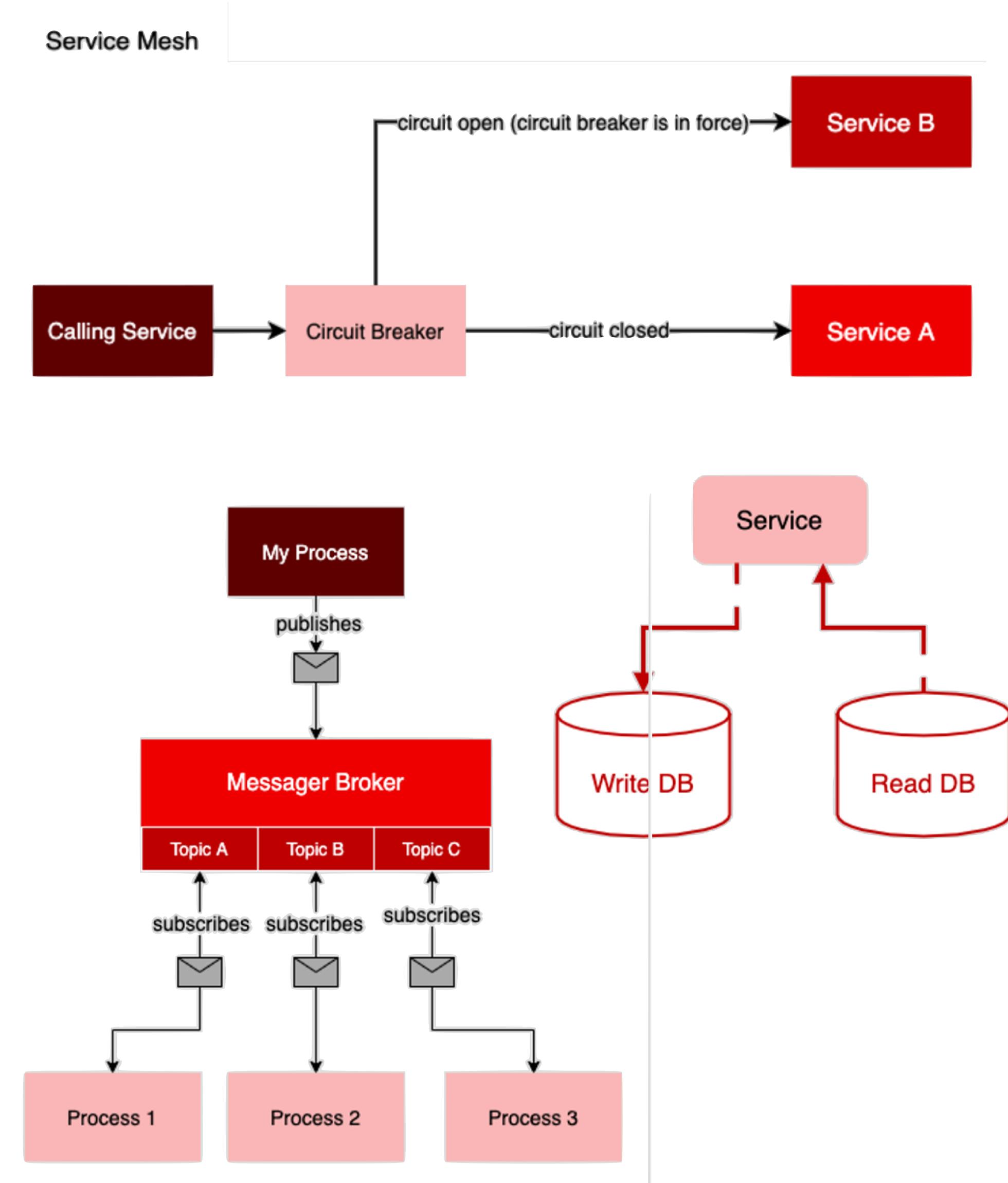
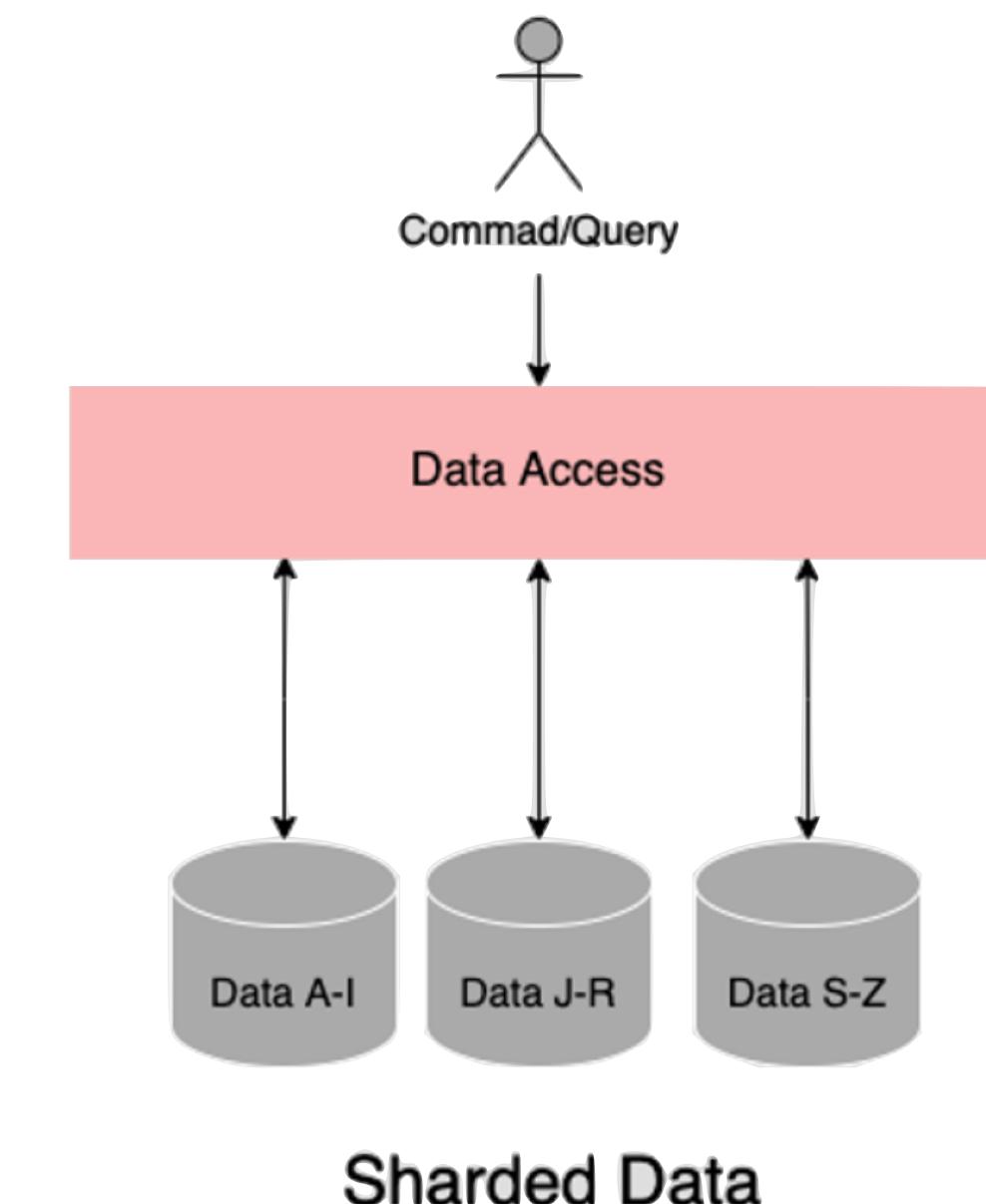


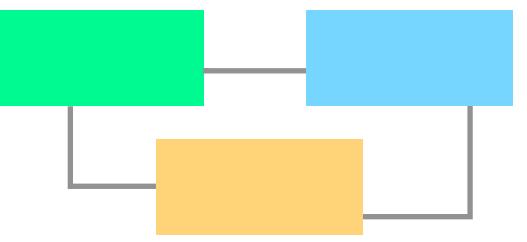
Image Credits: RedHat



# Extra software architecture patterns

- Circuit breaker
  - Prevents effects of faults by rerouting to other services in case of failure. It needs to be tested thoroughly.
- Command query responsibility segregation
  - Separates reads from writes when convenient. Hard to ensure consistency.
- Pub-Sub (Asynchronous)
- Sharding
- ...





# The Layered Architecture

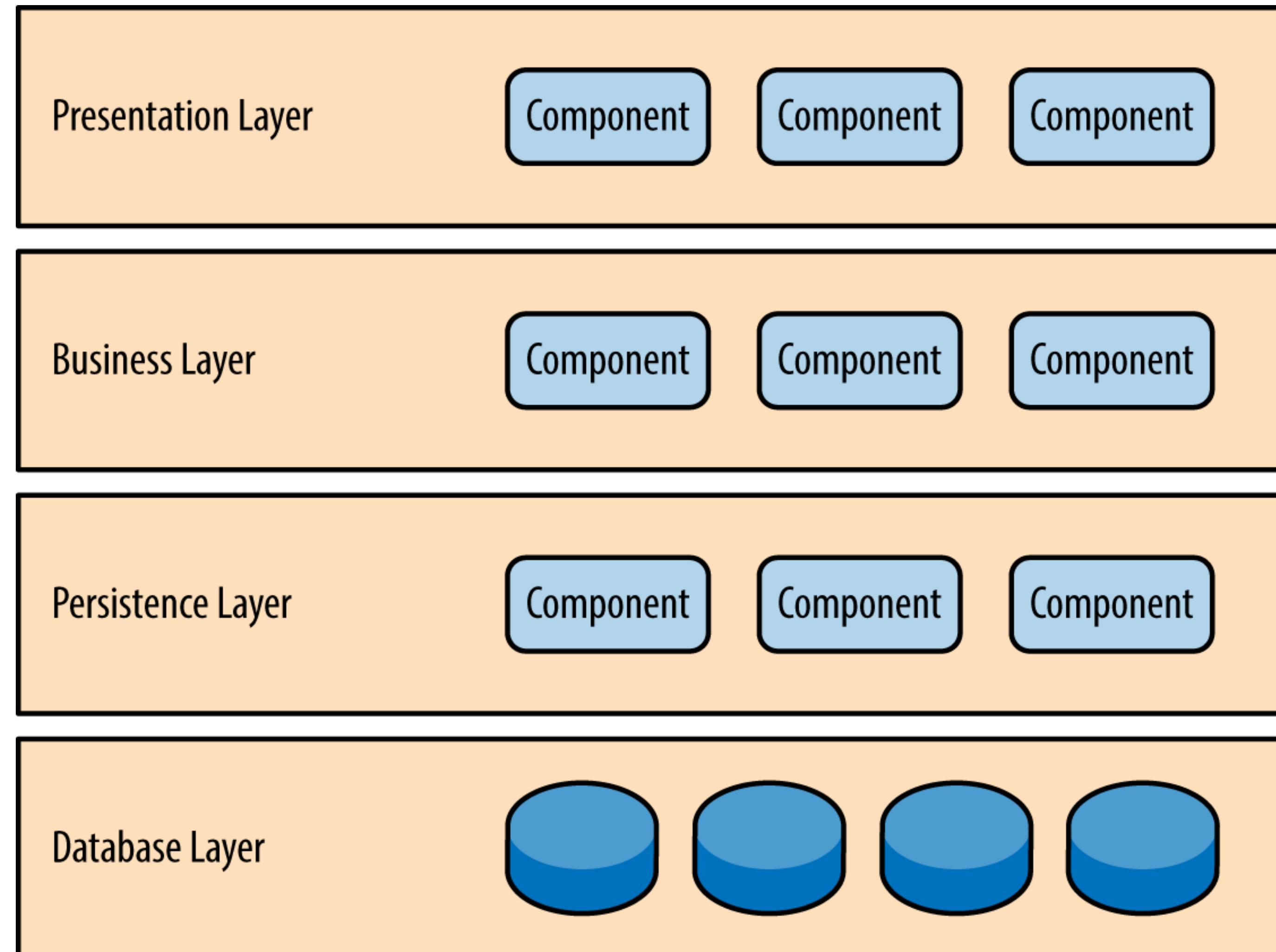


Image credits: O'Reilly

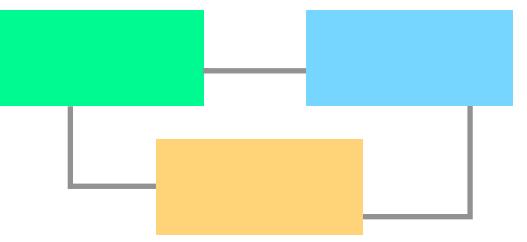
# Internet Applications Design and Implementation

## (Lecture 2, Part 2 - Software Architecture - Internet Applications)

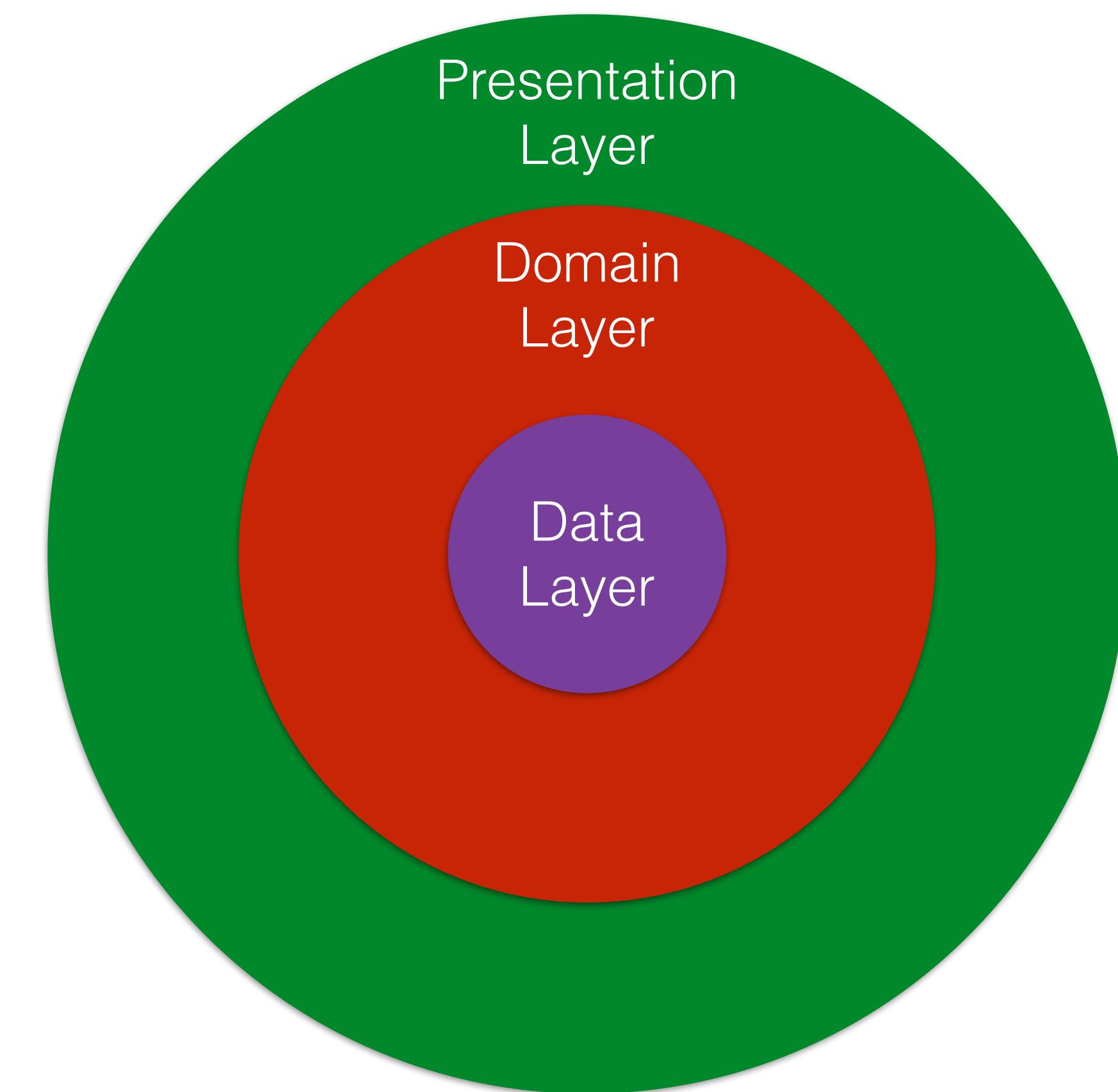
**MIEI - Integrated Master in Computer Science and Informatics  
Specialization block**

**João Costa Seco ([joao.seco@fct.unl.pt](mailto:joao.seco@fct.unl.pt))**

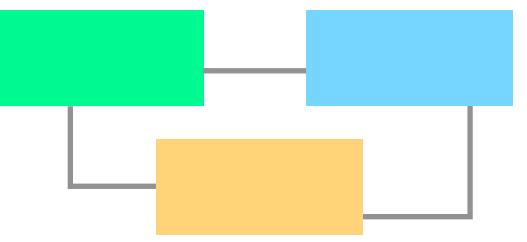
(with previous participations of Jácome Cunha ([jacome@fct.unl.pt](mailto:jacome@fct.unl.pt)) and João Leitão ([jc.leitao@fct.unl.pt](mailto:jc.leitao@fct.unl.pt)))



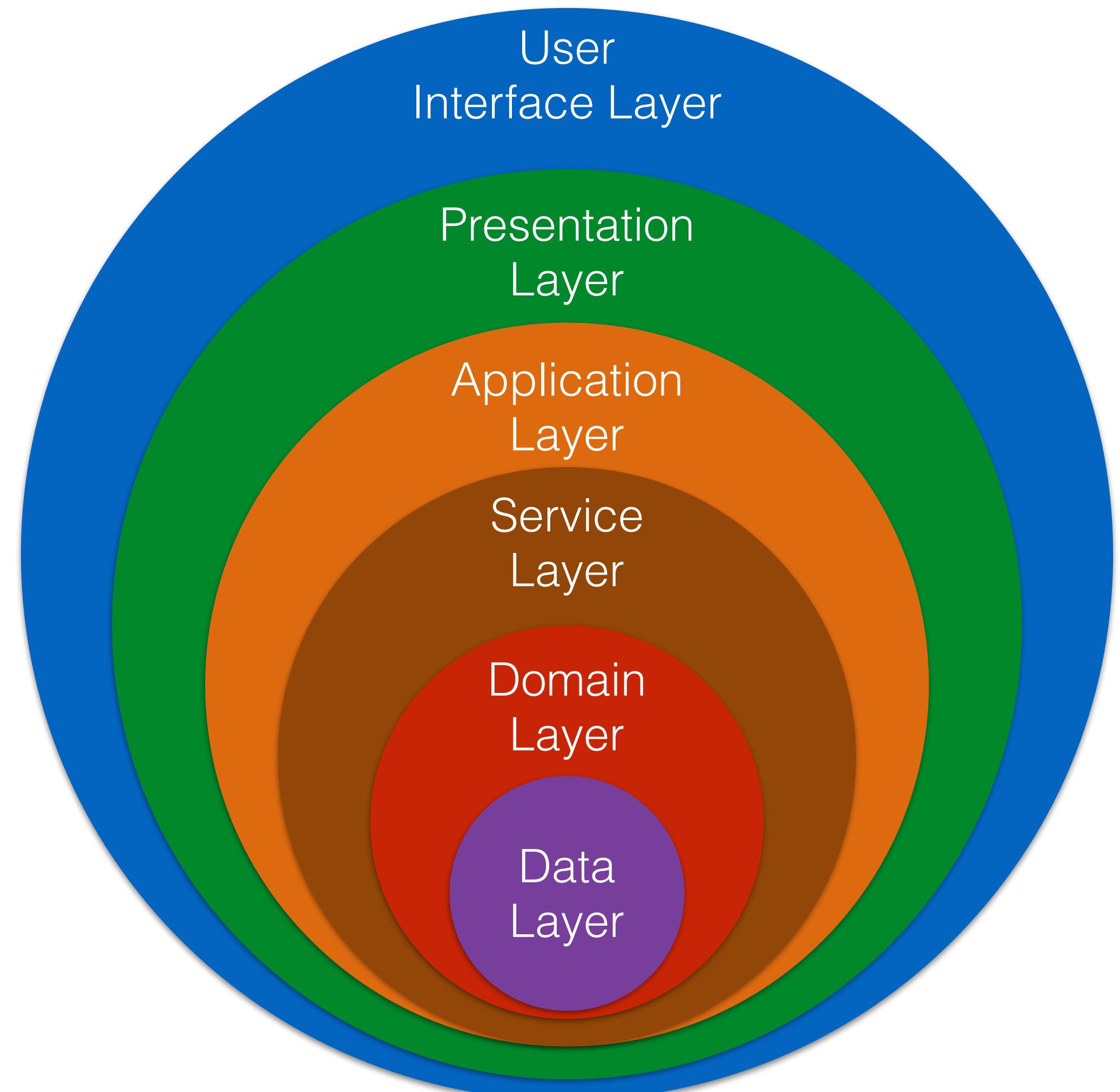
# Internet Applications are Data-Centric



Patterns of Enterprise Application Architecture

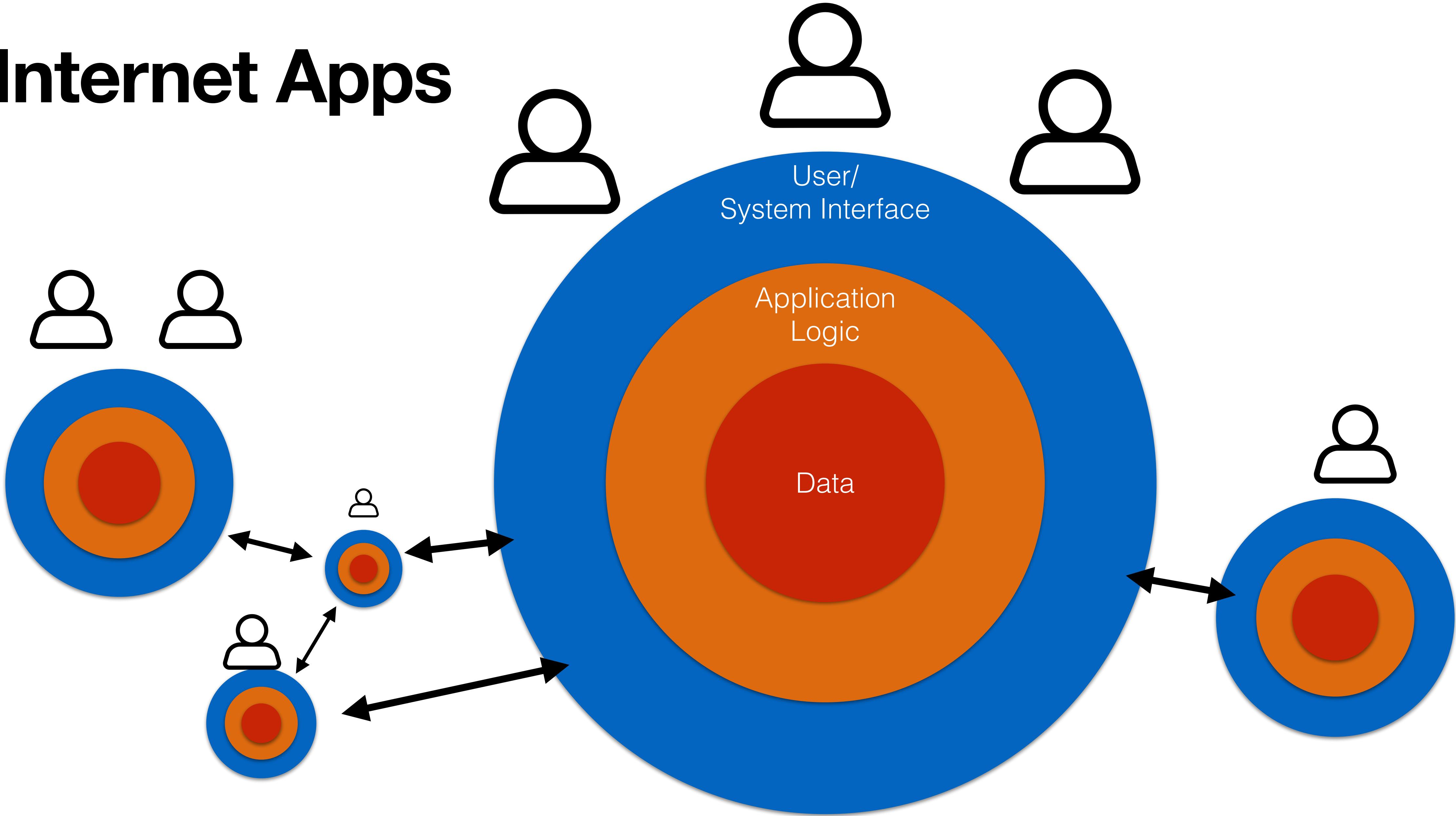


# Internet Applications are Data-Centric



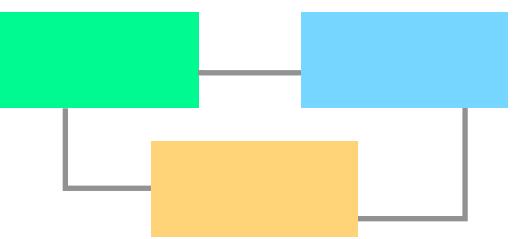
<https://dzone.com/articles/layered-architecture-is-good>

# Internet Apps

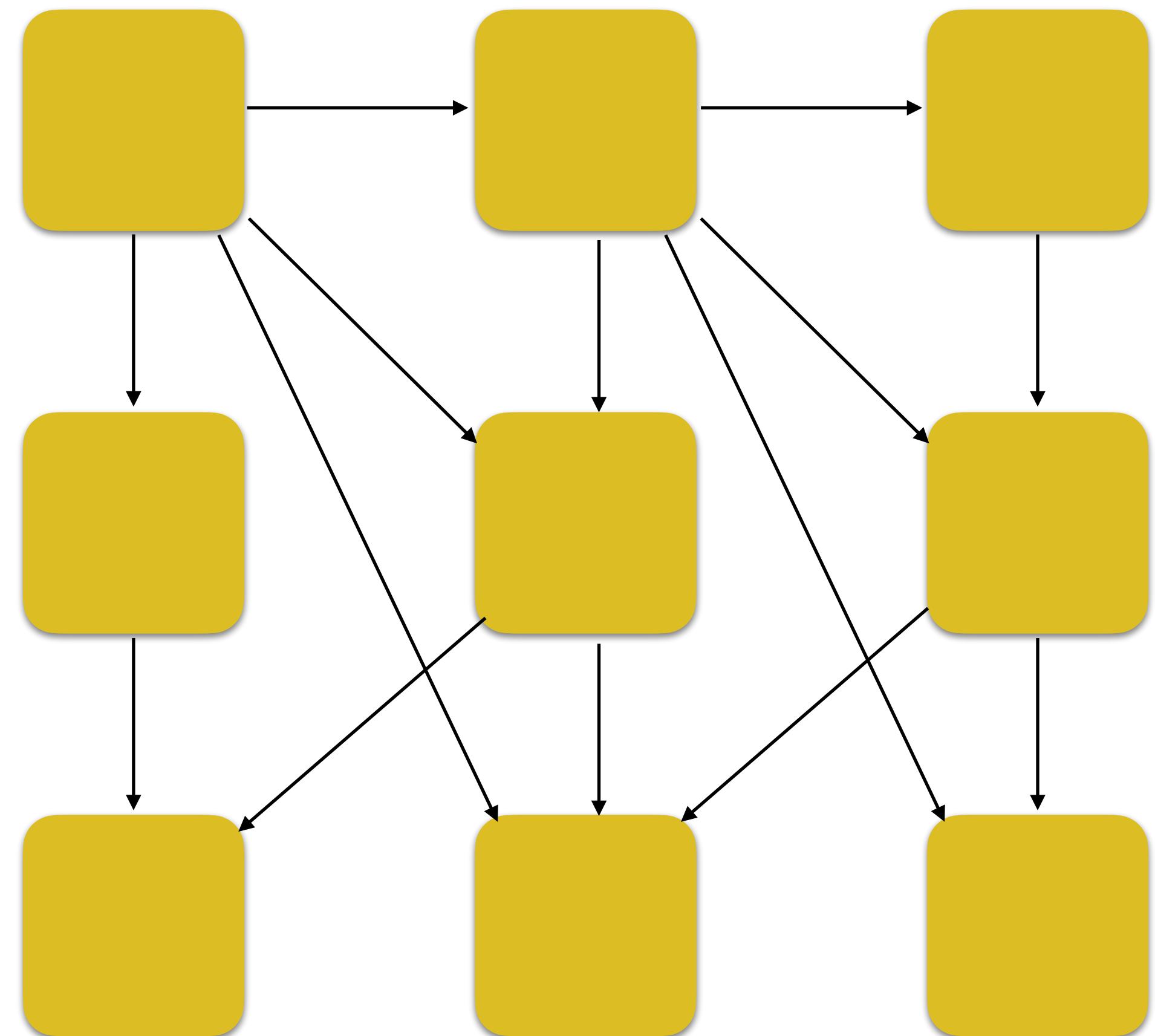


# Monoliths can be agile if well structured



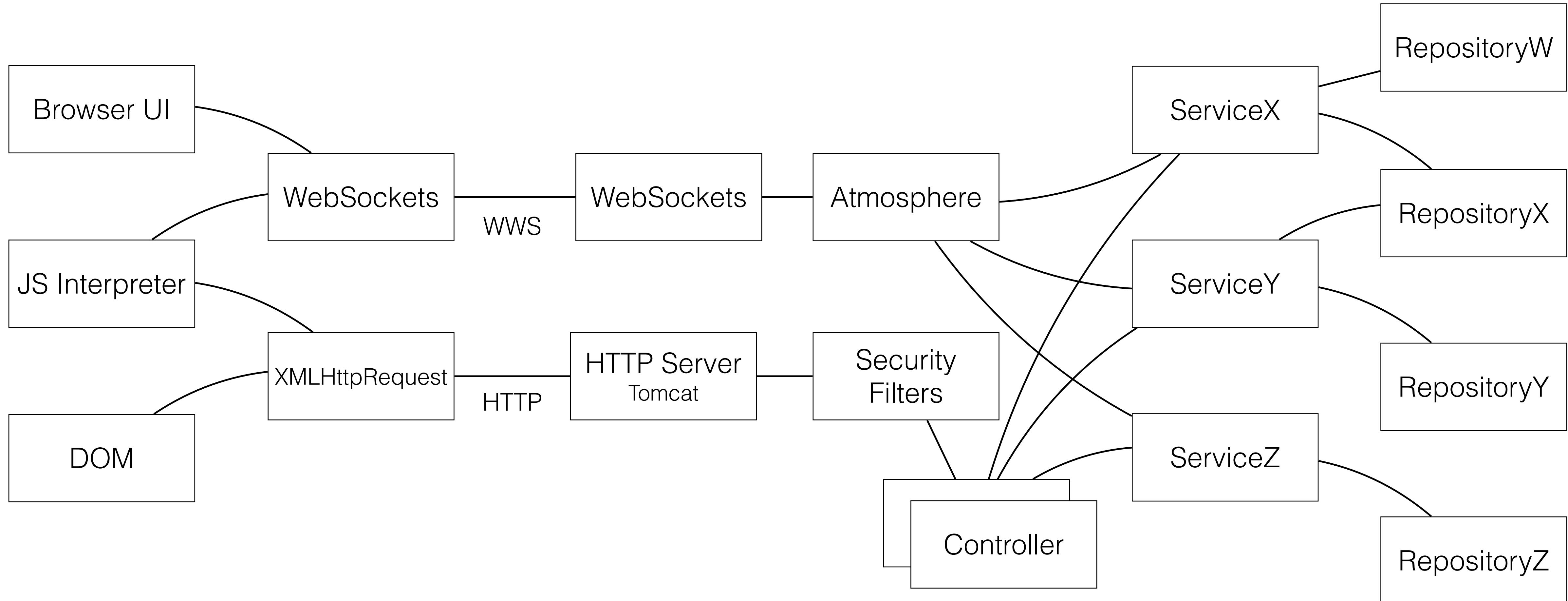
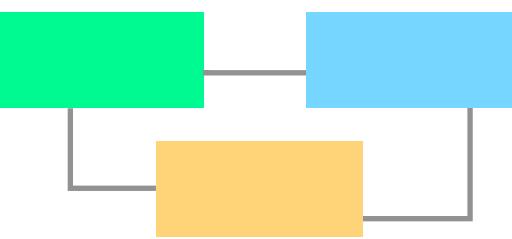


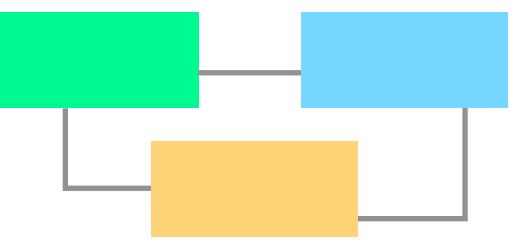
# Internet Applications are also Decentralised



<https://dzone.com/articles/introduction-to-microservices-part-1>

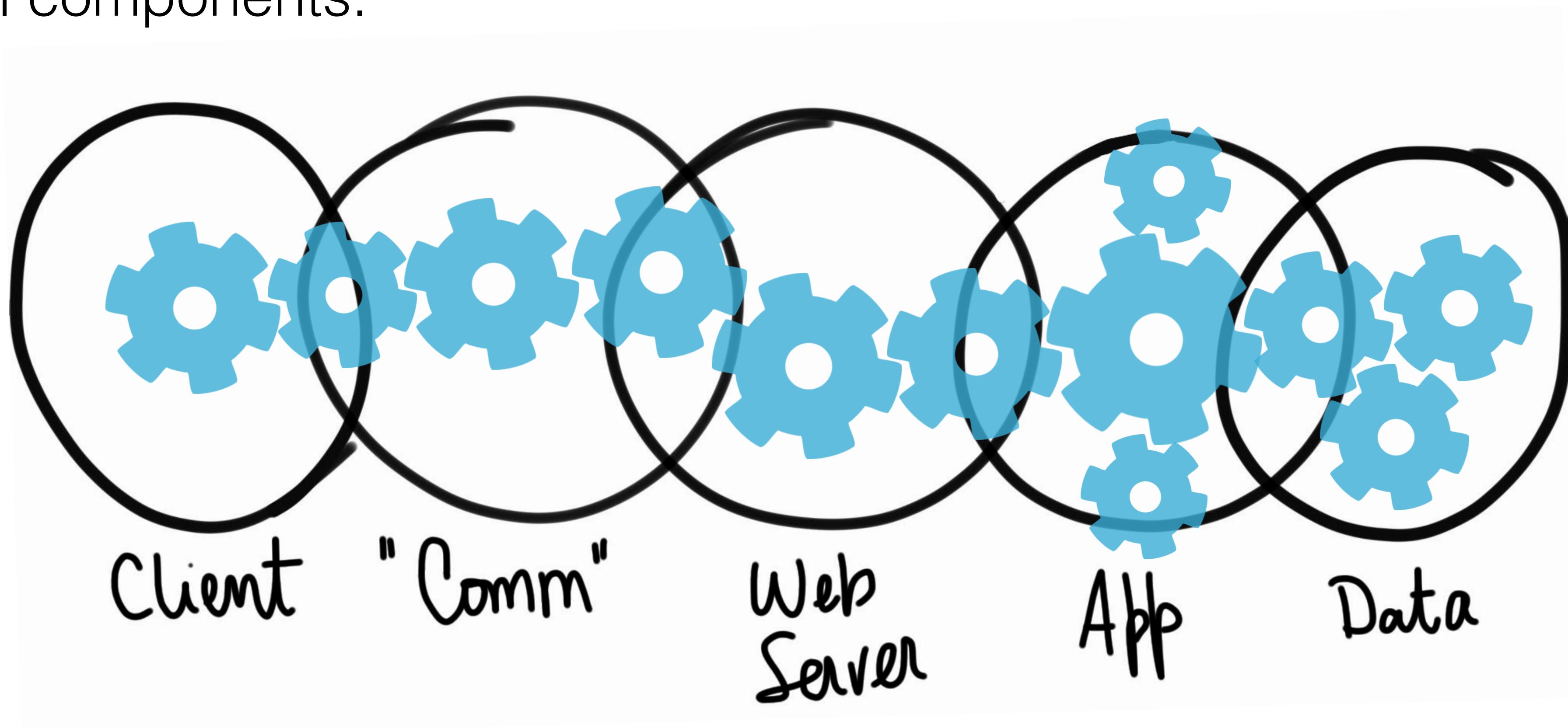
# Architecture of Web applications

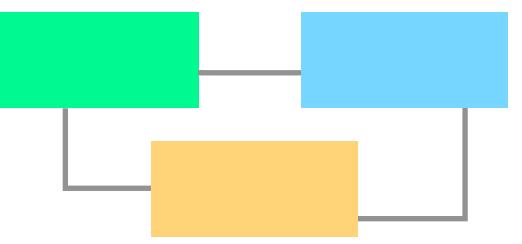




# Architecture of Web applications - The Big Picture

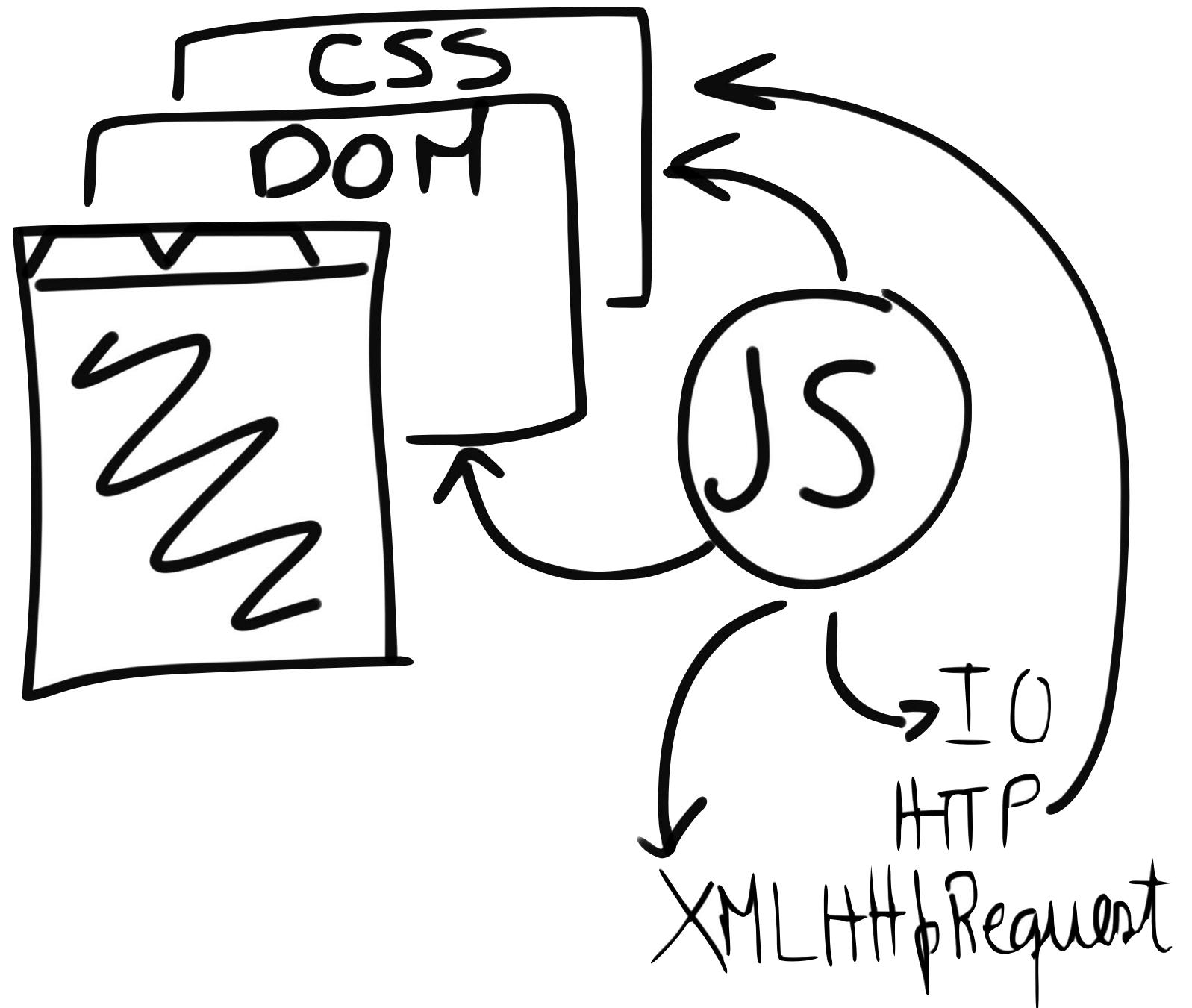
- A web application is made of code deployed to the independent and different physical components.

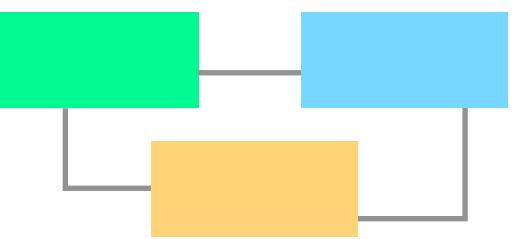




# Web client architecture

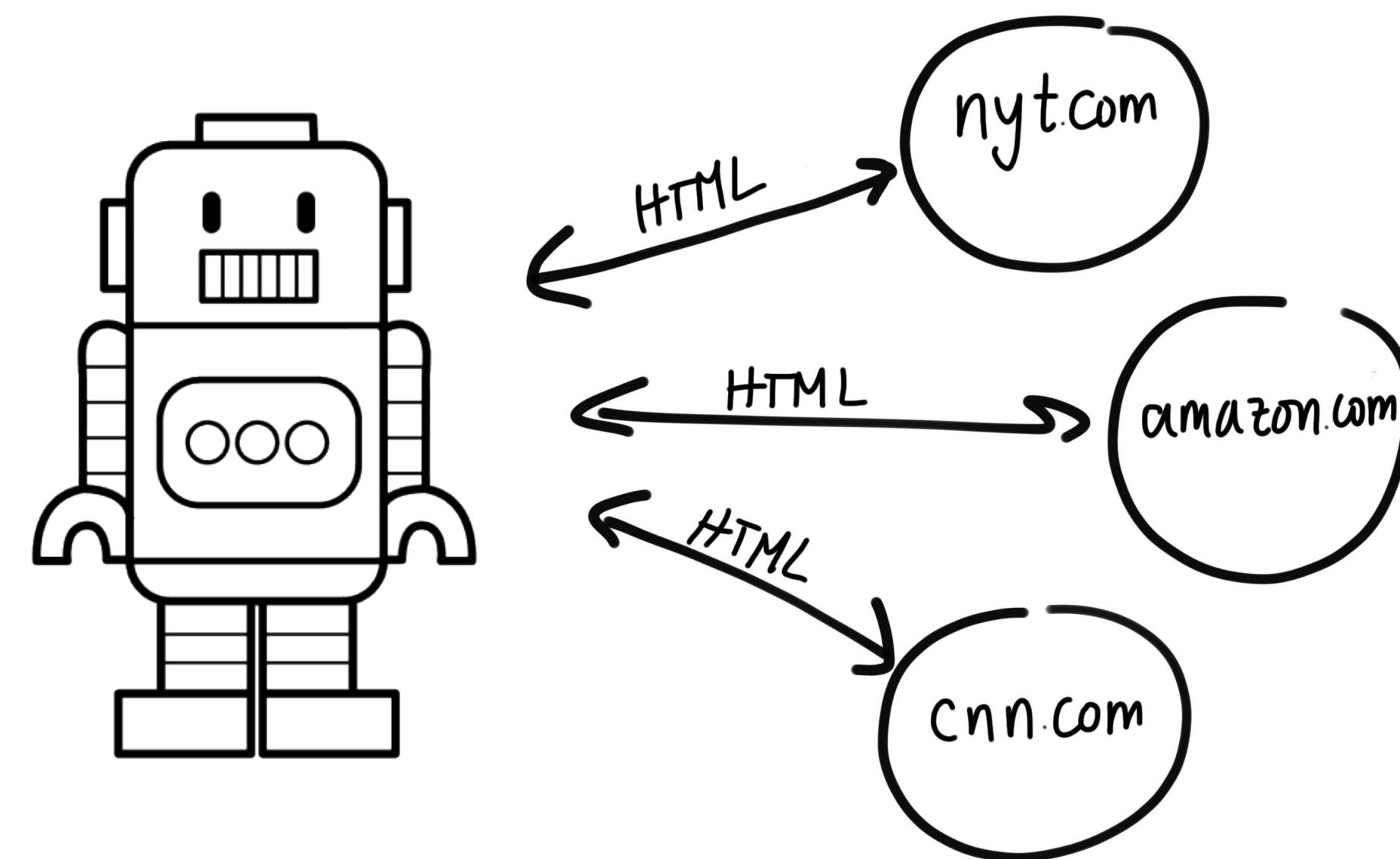
- Browser (HTML5)
  - HTML (structure and semantics)
  - CSS (style and UI behaviour)
  - JS + AJAX,  
Socket interfaces (behaviour)
  - DOM  
(the supporting data structure)
  - UI Events & callbacks  
(mechanism for dynamic structuring of behaviour)

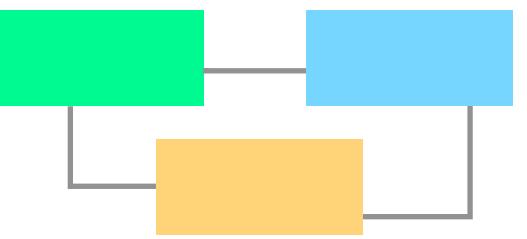




# Web client architecture

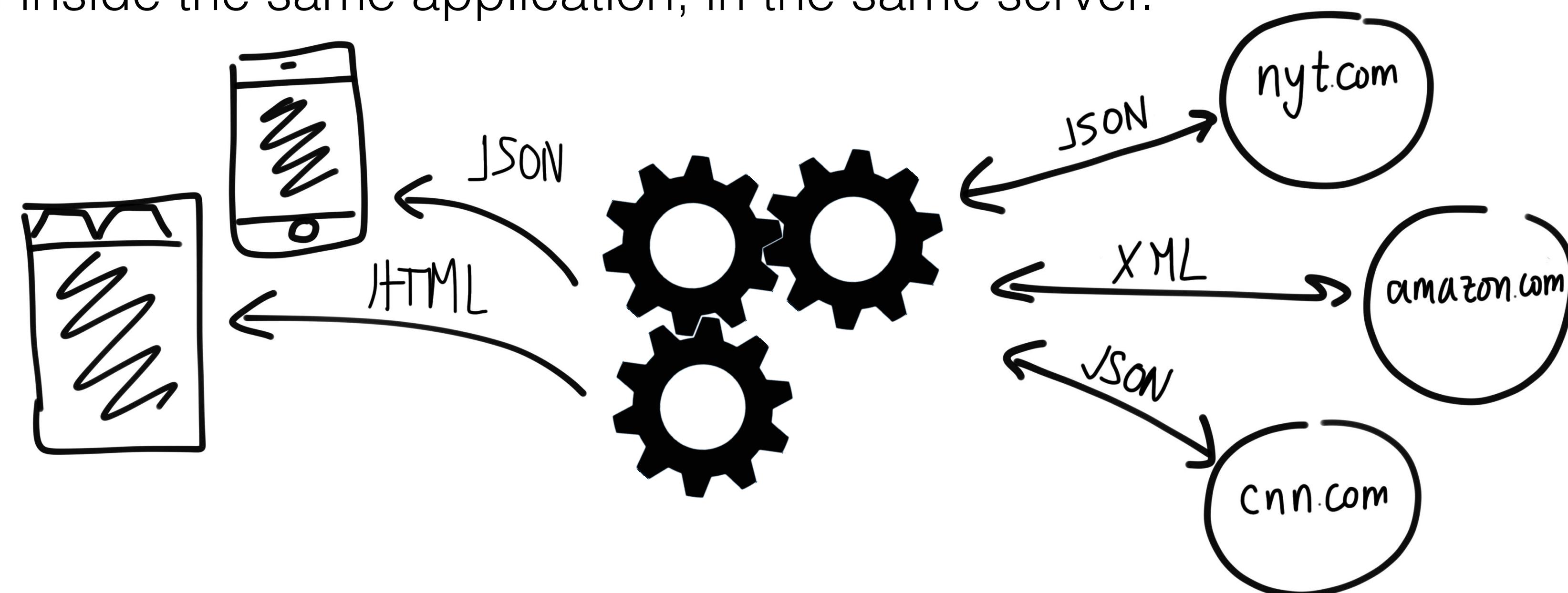
- Other kind of web clients (asking for HTML)
  - simulate web requests and sessions with web-servers
  - parse/crawl the results to extract information
  - should be built with web-services instead (if possible).

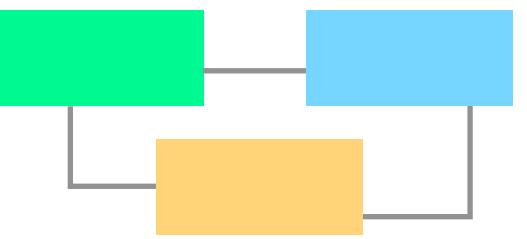




# Service based web client architecture

- Provide web content based on data providing services
  - Data can be exchanged in “machine-friendly” formats.  
(e.g. JSON, XML)
  - Combined and refurbished in HTML or data formats.
  - Even reused inside the same application, in the same server.





# Interconnection layer (low-level support)

- HTTP/1.1 (*Hypertext Transfer Protocol*) Protocol
  - Method: GET, HEAD, POST, PUT, DELETE (3 more)
  - Arguments (query string and body)
  - Multi-typed message body
  - Cookies
  - Return codes: 1XX, 2XX, 3XX, 4XX, 5XX
- Websockets (standard RFC6455 2011 - ws:// wss://)
  - supported by browsers and web servers to allow two way data communication between client and servers
- HTTP/2 approved May 2015.
  - Faster, compressed, and cyphered transmission of data
- TLS (successor of SSL)
  - Provides cryptographic support for web communications (handshake+symmetric crypto)

1xx Informational  
**100** Continue  
**101** Switching Protocols  
**102** Processing  
2xx Success  
**200** OK  
**201** Created  
**202** Accepted  
**203** Non-authoritative Information  
**204** No Content  
**205** Reset Content  
**206** Partial Content  
**207** Multi-Status  
**208** Already Reported  
**226** IM Used  
3xx Redirection  
**300** Multiple Choices  
**301** Moved Permanently  
**302** Found  
**303** See Other  
**304** Not Modified  
**305** Use Proxy  
**307** Temporary Redirect  
**308** Permanent Redirect  
4xx Client Error  
**400** Bad Request  
**401** Unauthorized  
**402** Payment Required  
**403** Forbidden  
**404** Not Found  
**405** Method Not Allowed  
**406** Not Acceptable  
**407** Proxy Authentication Required  
**408** Request Timeout  
**409** Conflict  
**410** Gone  
**411** Length Required  
**412** Precondition Failed  
**413** Payload Too Large  
**414** Request-URI Too Long  
**415** Unsupported Media Type  
**416** Requested Range Not Satisfiable  
**417** Expectation Failed  
**418** I'm a teapot  
**421** Misdirected Request  
**422** Unprocessable Entity  
**423** Locked  
**424** Failed Dependency  
**426** Upgrade Required  
**428** Precondition Required  
**429** Too Many Requests  
**431** Request Header Fields Too Large  
**444** Connection Closed Without Response  
**451** Unavailable For Legal Reasons  
**499** Client Closed Request  
5xx Server Error  
**500** Internal Server Error  
**501** Not Implemented  
**502** Bad Gateway  
**503** Service Unavailable  
**504** Gateway Timeout  
**505** HTTP Version Not Supported  
**506** Variant Also Negotiates  
**507** Insufficient Storage  
**508** Loop Detected  
**510** Not Extended  
**511** Network Authentication Required  
**599** Network Connect Timeout Error

<https://httpstatuses.com/>

# Web server / App server architecture

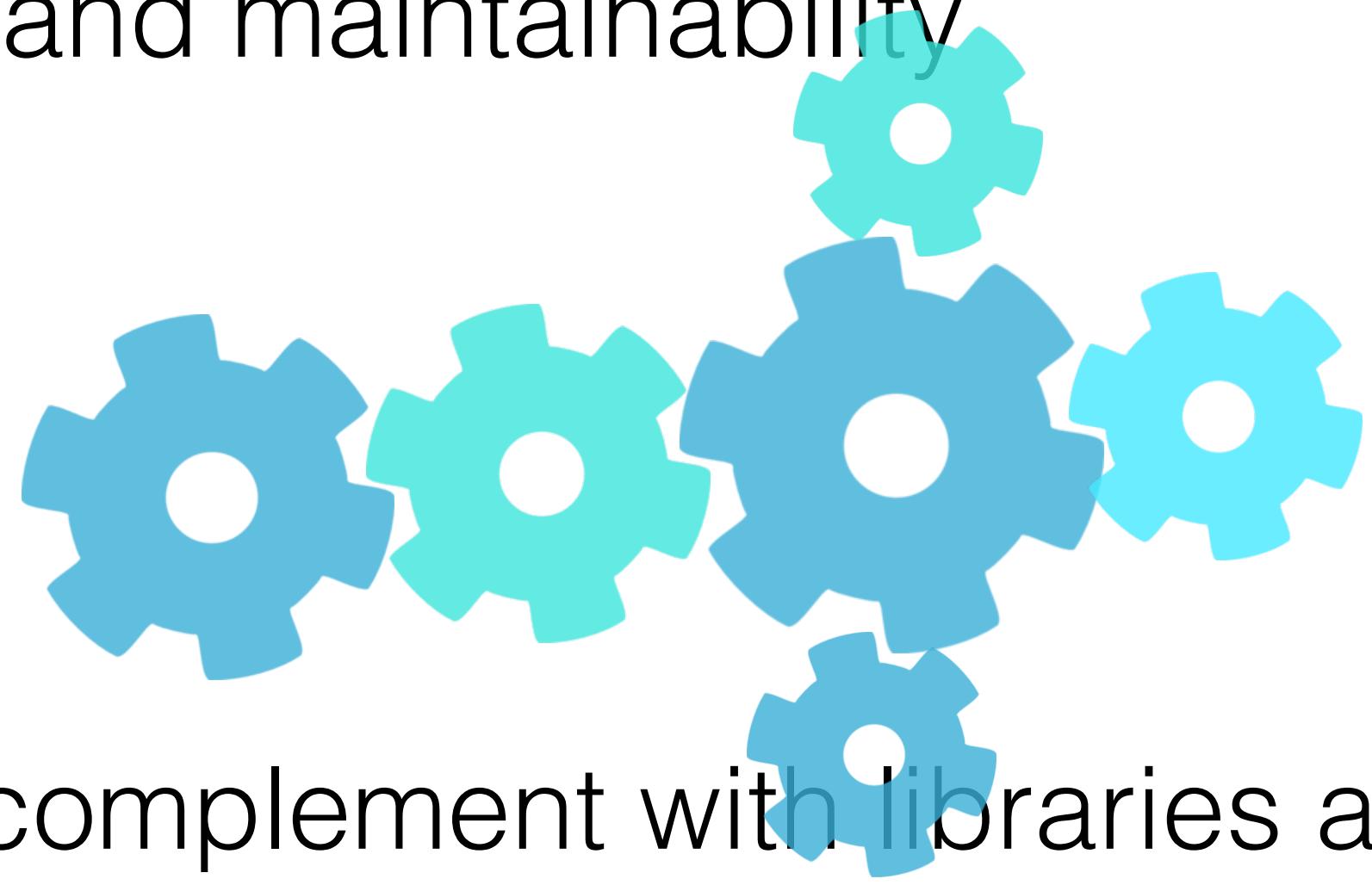
---

- Web servers handle HTTP requests, map URLs to local files, execute local scripts (e.g. CGI, PHP), or locally bound code (e.g. Servlets).
- App Servers modularly manage bound code, and associated resources (e.g. sessions, context, connections).
  - One web server, many applications.
  - Allows the assembly of components of applications (controllers, views, models)

# Web architectures, patterns and styles

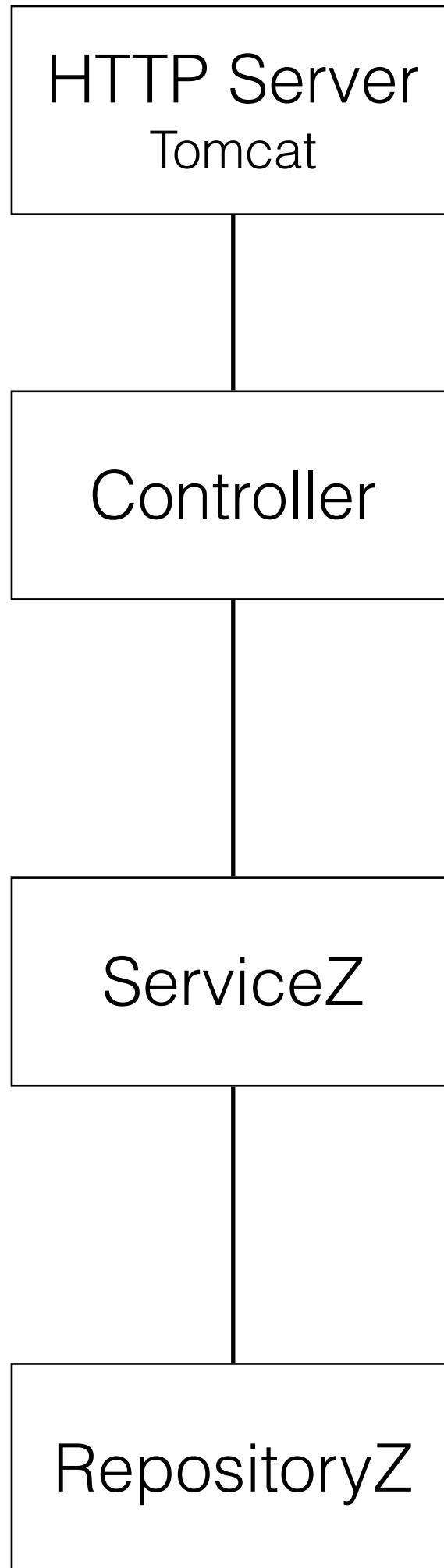
---

- Increase the level of abstraction, reusability, and maintainability
  - Software Architectures
  - Architectural and design patterns
  - Architectural styles
- Software frameworks implement some, and complement with libraries and tools.
- User-defined pieces are required to specify the “core” logic, and configure general purpose code.
- All implement the “inversion of control” pattern.



An architecture built with Spring

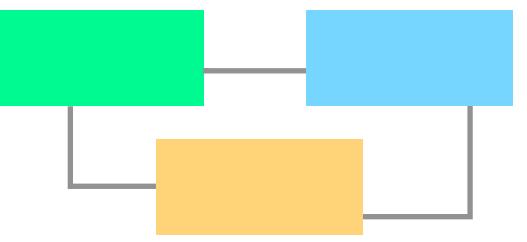
# Example of an Architecture built with Spring



- Spring is a component framework
- Resolves component dependencies by dependency injection
- Uses annotations to configure components

```
@RestController  
@RequestMapping("/")  
class EmpController(val employees:EmployeeService) {  
  
    // http GET :8080/api/projects/2/team  
    @GetMapping( "/api/projects/{id}/team")  
    fun teamMembersOfProject(  
        @PathVariable id:String  
    )  
        = employees.teamMembersOfProject(id)  
  
    }  
  
    @Service  
    class EmployeeService(val employees:EmployeeRepository) {  
        fun teamMembersOfProject(id:String) = employees.findAll()  
    }  
  
    interface EmployeeRepository : CrudRepository<Employee, Long>
```

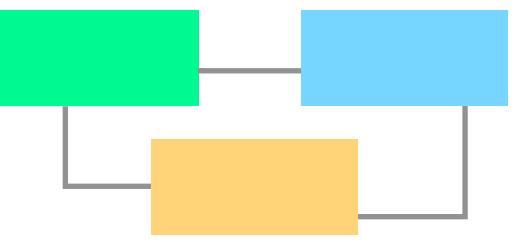
# Service Based Architectures



# Service Oriented Architectures

- Are the technological and methodological basis for building **open-ended Internet Applications**.
- Provide a **model of distributed computation** based on **loosely coupled interactions**.
- Define **heterogeneous ecosystems** of service implementations.
- Use implementation-independent data formats (**JSON, XML**)
- Allows **independent development** of services by different vendors and technologies
- A service:
  - Is a logical representation of **a repeatable business activity** that has a specified outcome (e.g., check customer credit, provide weather data, consolidate drilling reports)
  - Is self-contained
  - May be composed of other services
  - Is a “black box” to consumers of the service

<https://web.archive.org/web/20160819141303/http://opengroup.org/soa/source-book/soa/soa.htm>



# Web architectures, patterns, and styles

---

- Web services are usually defined over HTTP protocol
- **SOAP** (*Simple Object Access Protocol*)
  - Operation based protocol (on HTTP) to implement web services (XML as format)
- **REST** (*Representational State Transfer*)
  - Resource based architectural style to implement web services over HTTP (or another connection protocol)

# Micro Service Architectures

---

- Are an extreme interpretation of service based architectures.
- Have smaller grained services and interfaces.
- Provide independent and lightweight deployment.
- Allow flexible management (e.g. replicas).
- Based on a clear service ownership model  
(team responsible for all stages of dev&ops).
- Isolated persistent state (**sometimes bad**).

<https://martinfowler.com/microservices/>

# Amazon's API Mandate (by Jeff Bezos, 2002)

---

- All teams will henceforth expose their data and functionality through service interfaces.
- Teams must communicate with each other through these interfaces.
- There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
- It doesn't matter what technology you use.
- All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
- The mandate closed with: Anyone who doesn't do this will be fired. Thank you; have a nice day!

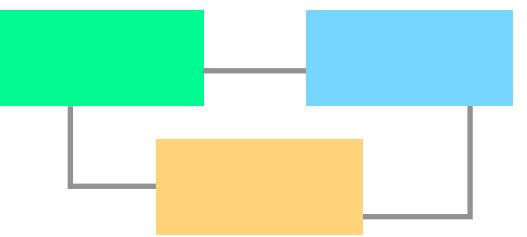
# Internet Applications Design and Implementation

## (Lecture 2, Part 3 - Software Architecture - Frameworks)

**MIEI - Integrated Master in Computer Science and Informatics  
Specialization block**

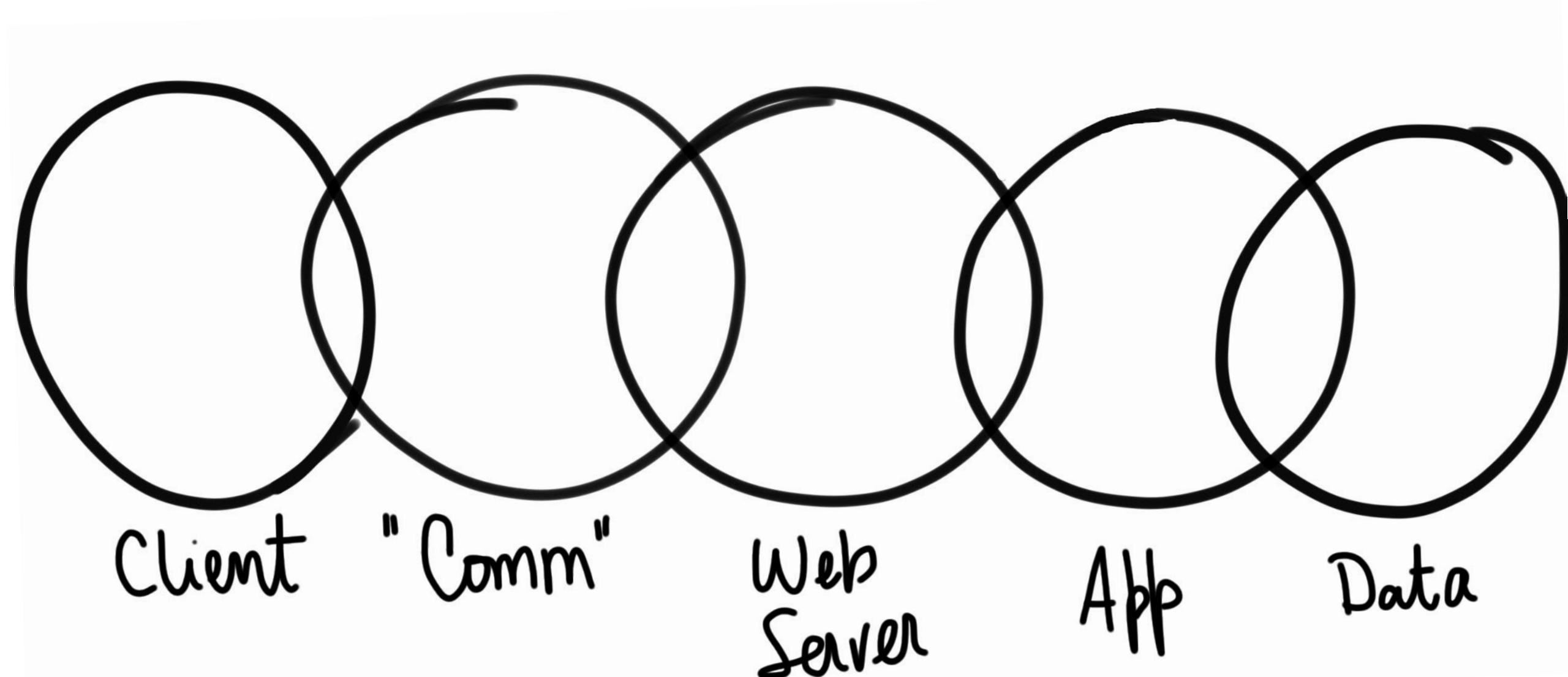
**João Costa Seco ([joao.seco@fct.unl.pt](mailto:joao.seco@fct.unl.pt))**

(with previous participations of Jácome Cunha ([jacome@fct.unl.pt](mailto:jacome@fct.unl.pt)) and João Leitão ([jc.leitao@fct.unl.pt](mailto:jc.leitao@fct.unl.pt)))

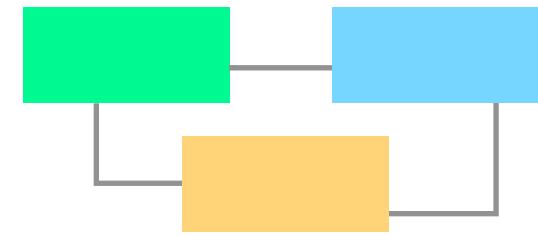


# Web architectures, patterns and styles

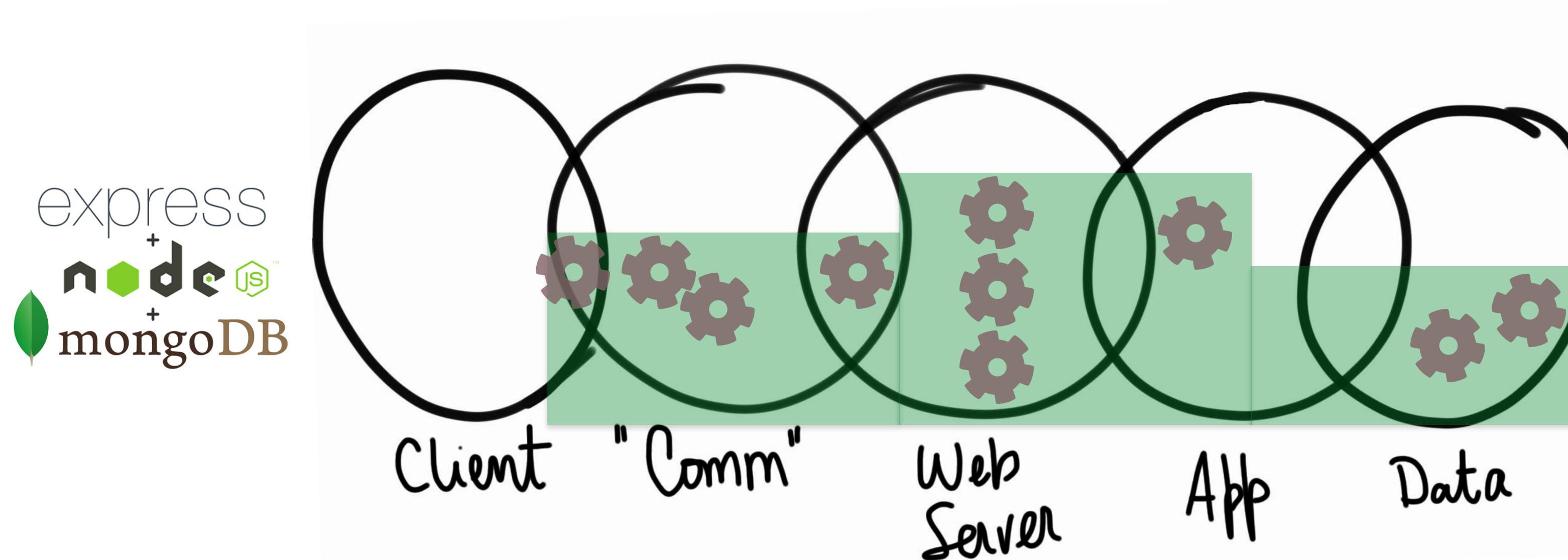
- Most common web applications follow the MVC architectural pattern.
  - Model layer - isolate the representation of persistent data and its operations, validations and conditions
  - Controller - contains the core application logic implementing the application interface (e.g. ad-hoc URL mapping, REST convention)
  - View - defines the way in which responses are formed (e.g. HTML, JSON)



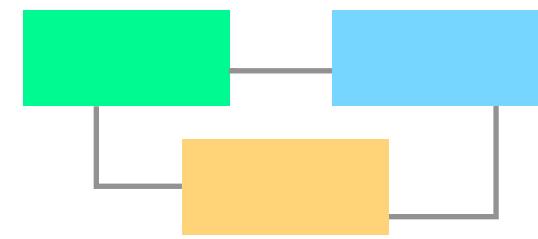
# Web architectures, patterns and styles



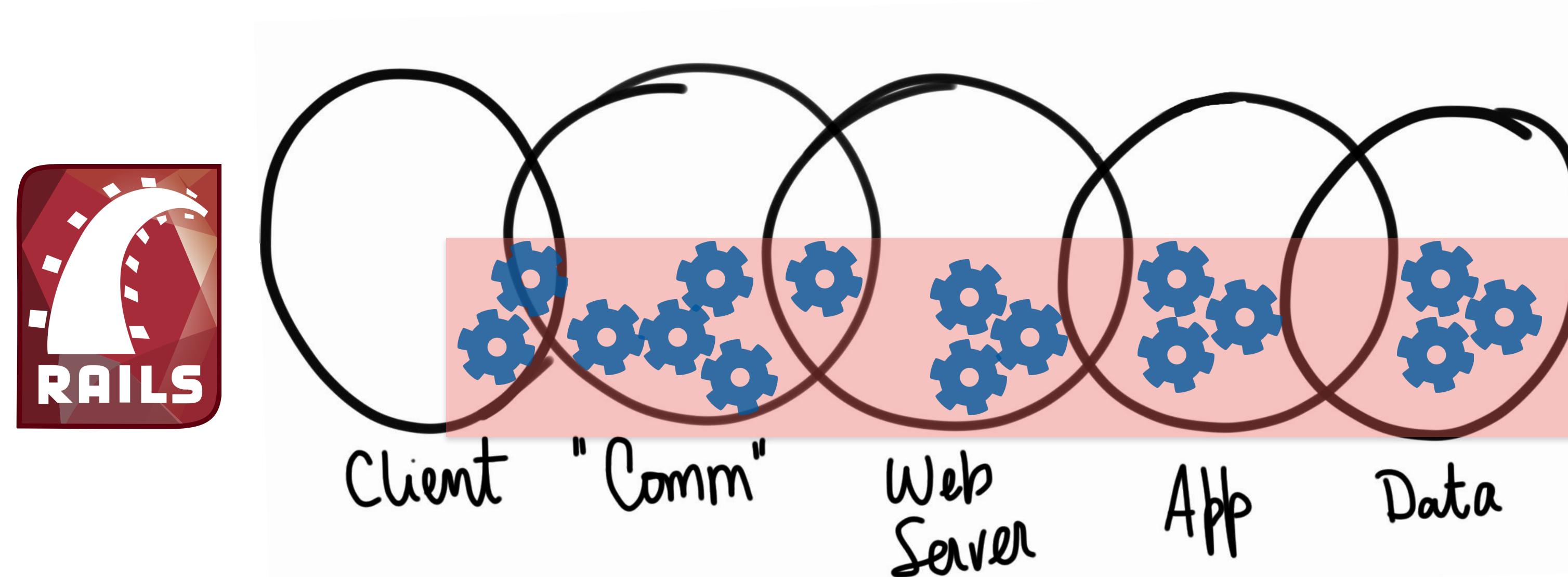
- Most common web applications follow the MVC architectural pattern.
  - Model layer - isolate the representation of persistent data and its operations, validations and conditions
  - Controller - contains the core application logic implementing the application interface (e.g. ad-hoc URL mapping, REST convention)
  - View - defines the way in which responses are formed (e.g. HTML, JSON)



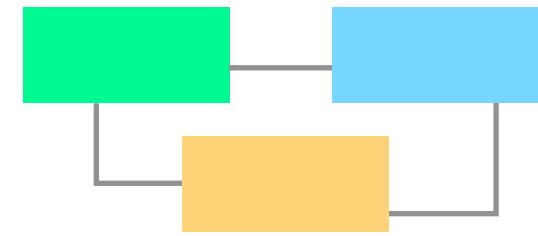
# Web architectures, patterns and styles



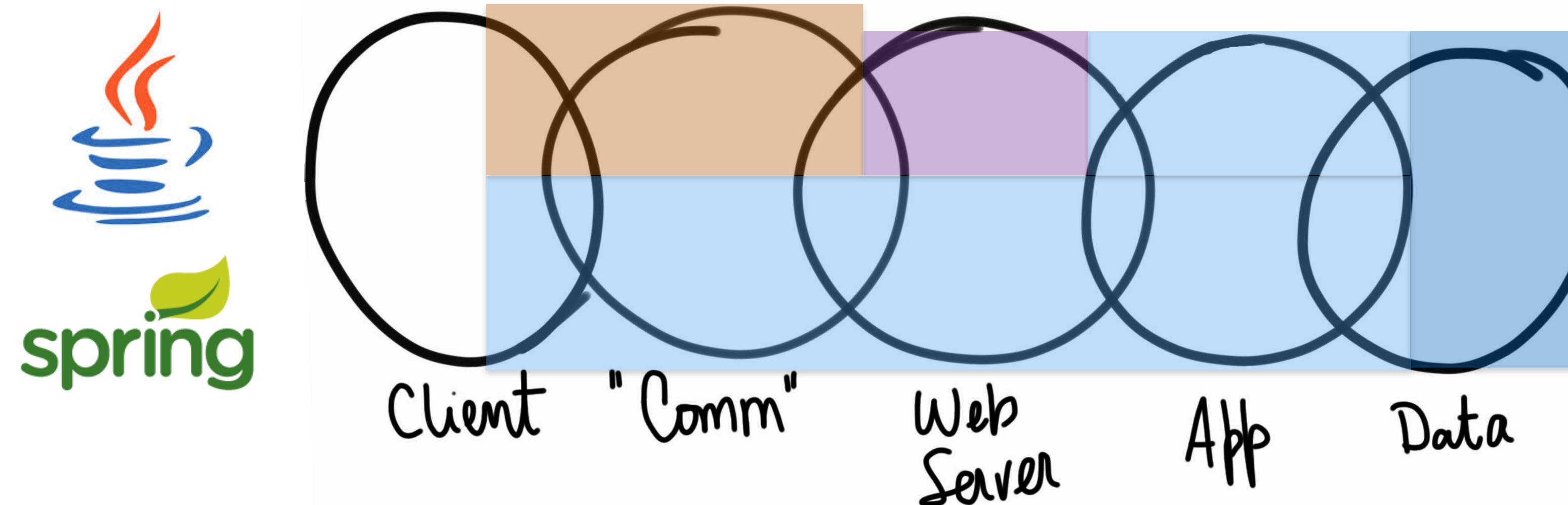
- Most common web applications follow the MVC architectural pattern.
  - Model layer - isolate the representation of persistent data and its operations, validations and conditions
  - Controller - contains the core application logic implementing the application interface (e.g. ad-hoc URL mapping, REST convention)
  - View - defines the way in which responses are formed (e.g. HTML, JSON)

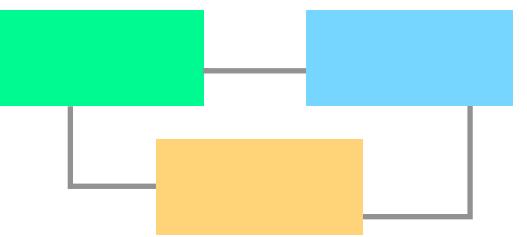


# Web architectures, patterns and styles



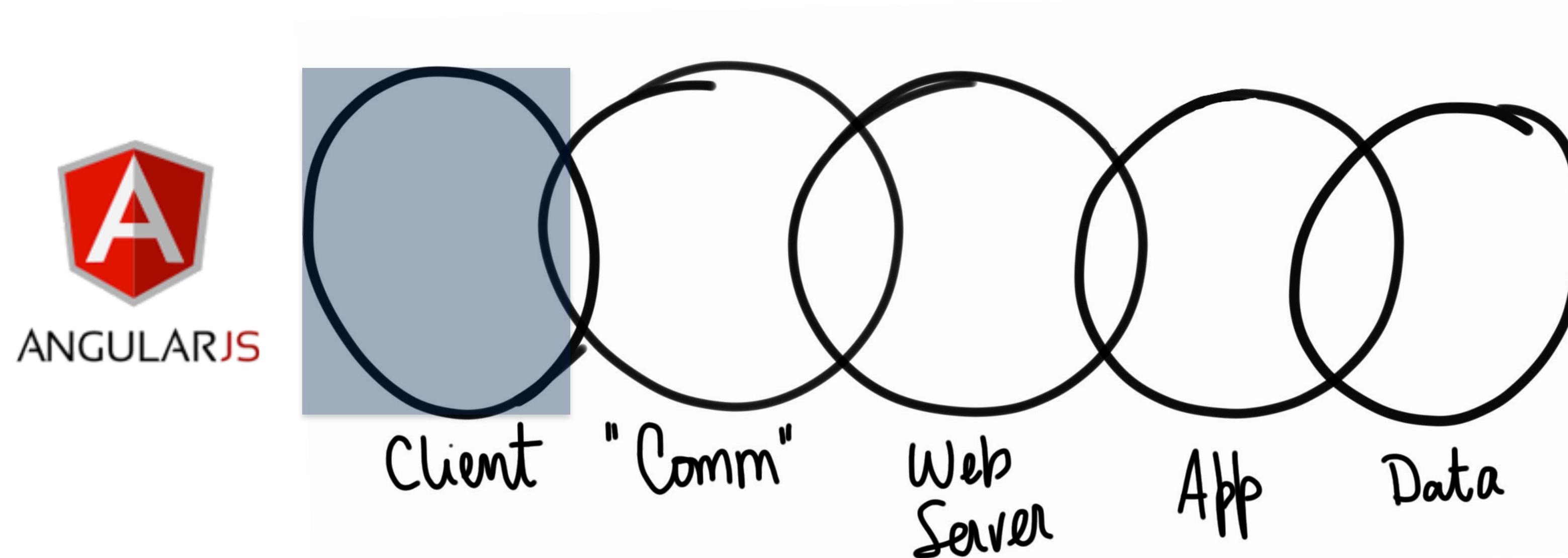
- Most common web applications follow the MVC architectural pattern.
  - Model layer - isolate the representation of persistent data and its operations, validations and conditions
  - Controller - contains the core application logic implementing the application interface (e.g. ad-hoc URL mapping, REST convention)
  - View - defines the way in which responses are formed (e.g. HTML, JSON)



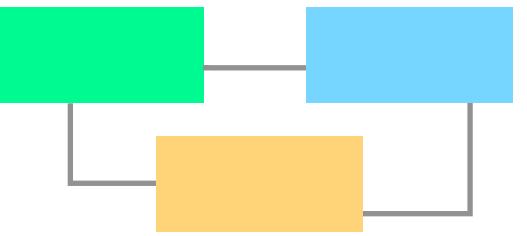


# Web architectures, patterns and styles

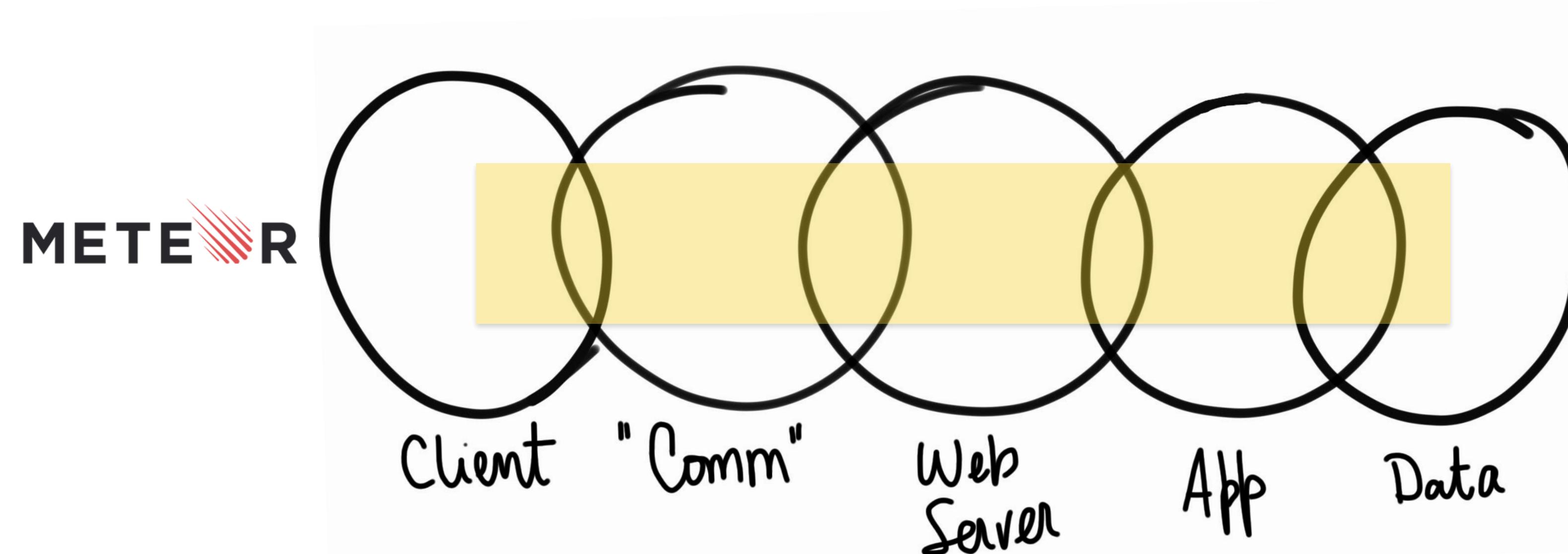
- Most common web applications follow the MVC architectural pattern.
  - Model layer - isolate the representation of persistent data and its operations, validations and conditions
  - Controller - contains the core application logic implementing the application interface (e.g. ad-hoc URL mapping, REST convention)
  - View - defines the way in which responses are formed (e.g. HTML, JSON)

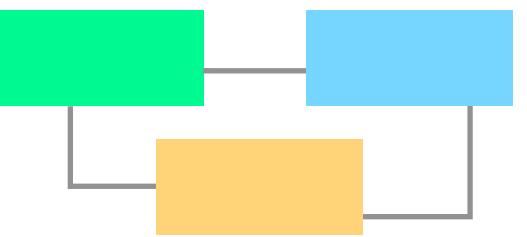


# Web architectures, patterns and styles



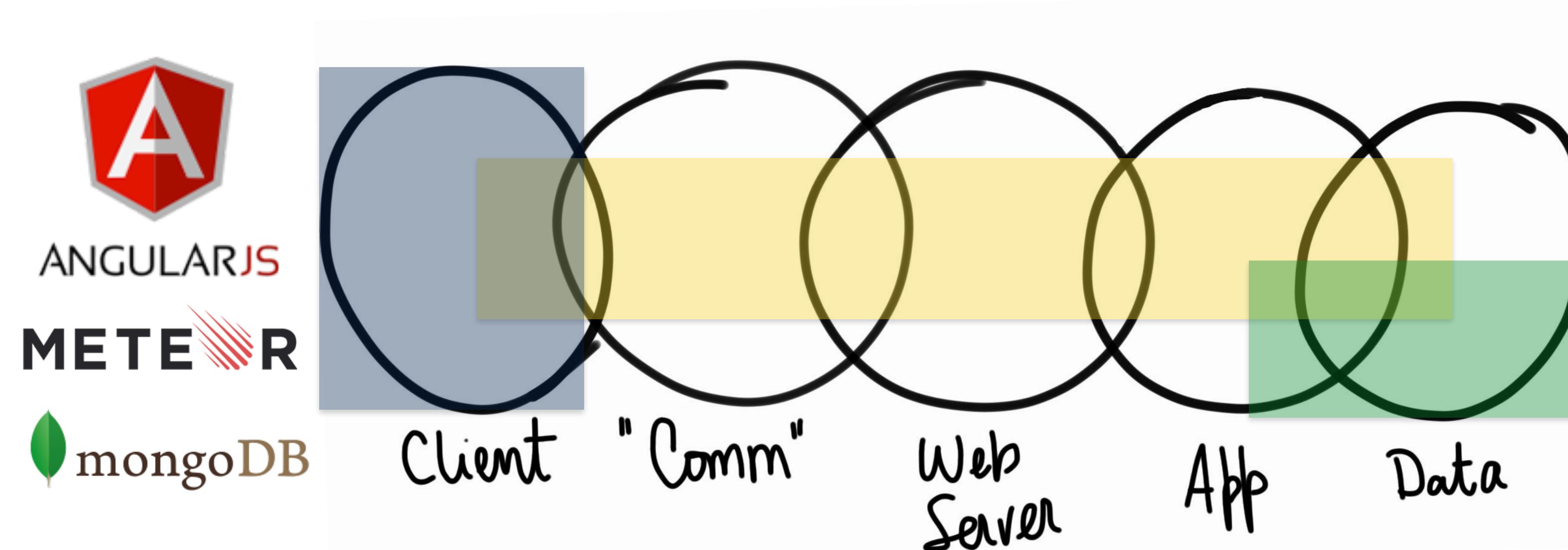
- Most common web applications follow the MVC architectural pattern.
  - Model layer - isolate the representation of persistent data and its operations, validations and conditions
  - Controller - contains the core application logic implementing the application interface (e.g. ad-hoc URL mapping, REST convention)
  - View - defines the way in which responses are formed (e.g. HTML, JSON)





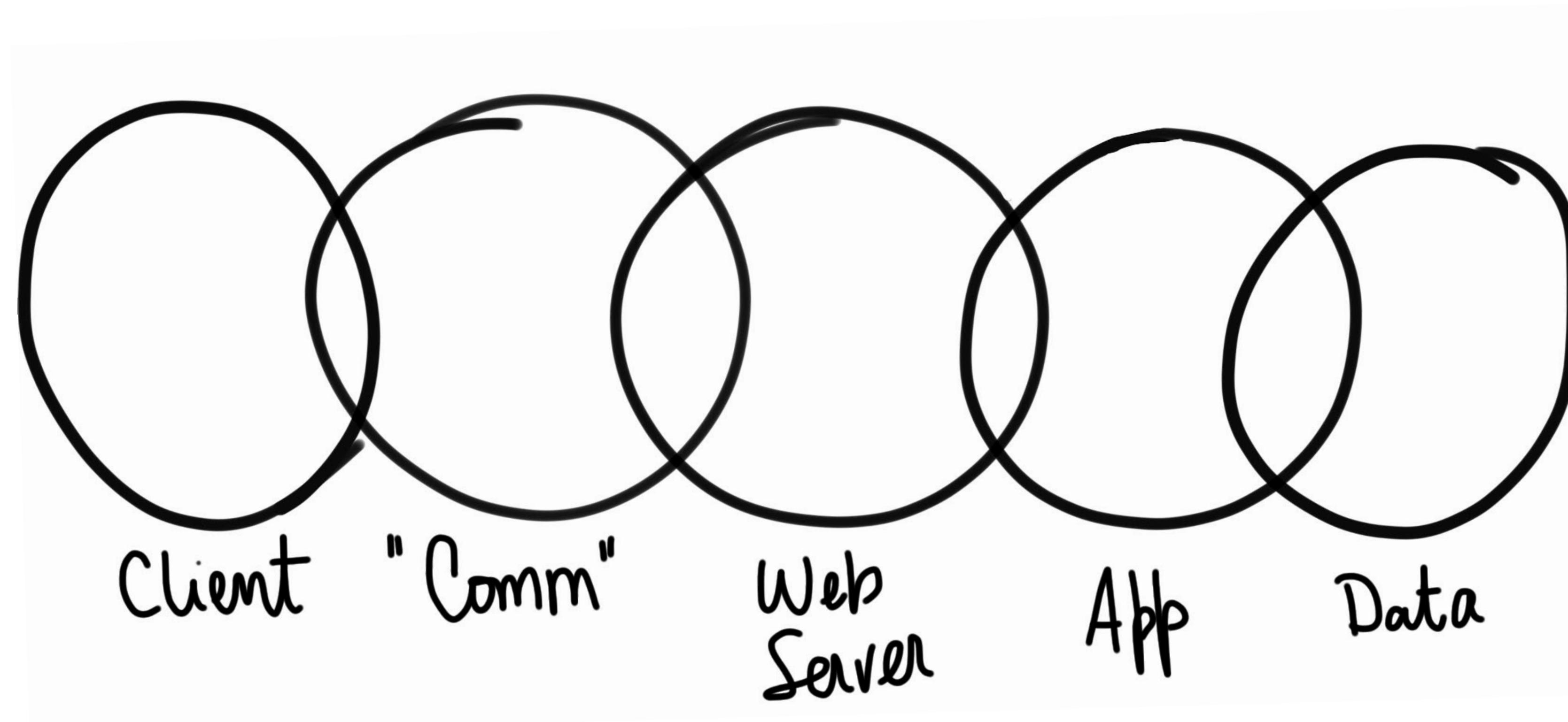
# Web architectures, patterns and styles

- Most common web applications follow the MVC architectural pattern.
  - Model layer - isolate the representation of persistent data and its operations, validations and conditions
  - Controller - contains the core application logic implementing the application interface (e.g. ad-hoc URL mapping, REST convention)
  - View - defines the way in which responses are formed (e.g. HTML, JSON)



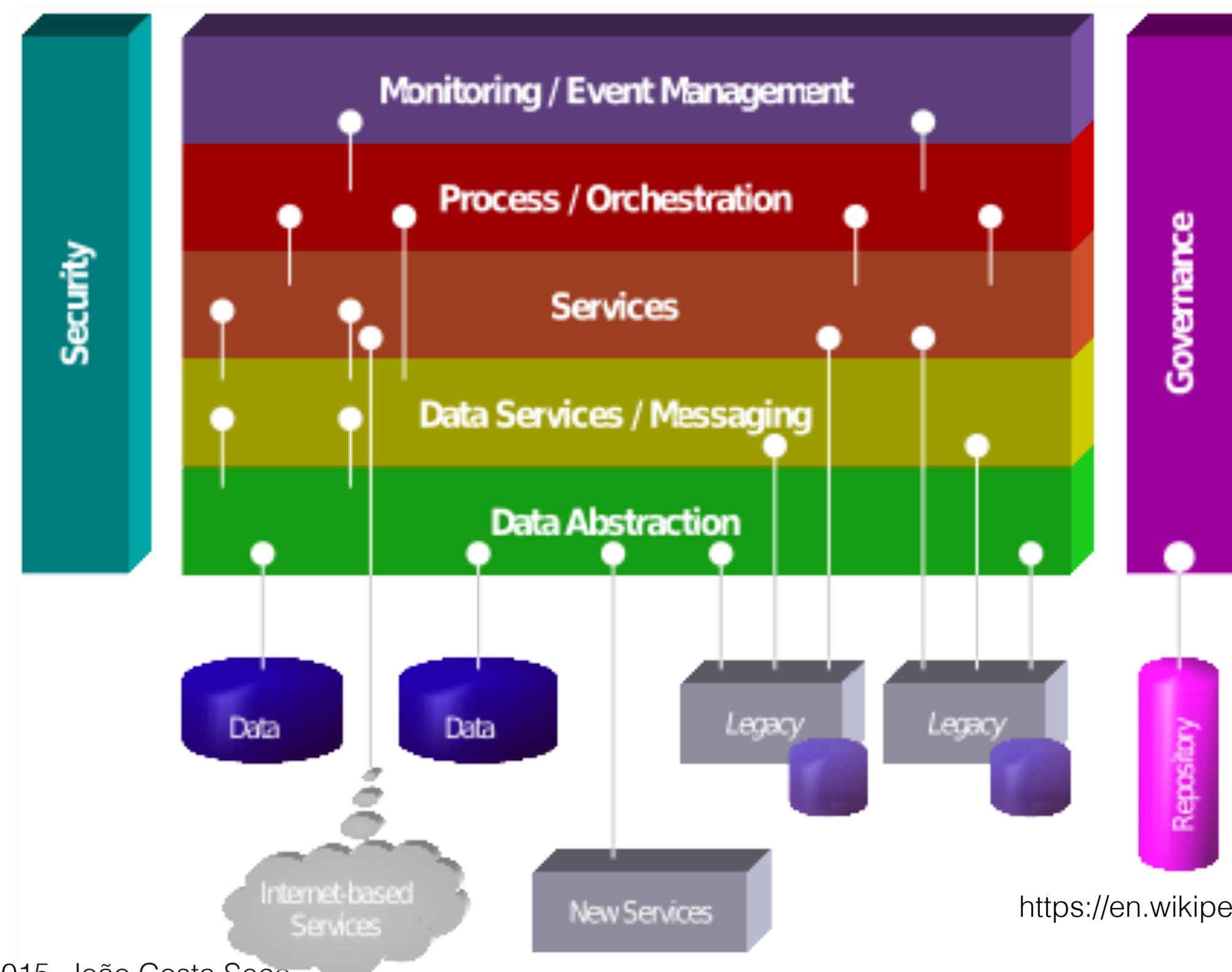
# Summary - Web Frameworks

- Web Frameworks are “languages” that carry libraries and abstractions that get compiled to run on the “web virtual machine”.



# (web & local) Services

- Service oriented architectures are a way of decoupling implementation from use.
- Services are implemented based on a “contract” or interface and provided by a broker.



[https://en.wikipedia.org/wiki/Service-oriented\\_architecture](https://en.wikipedia.org/wiki/Service-oriented_architecture)

# Service Orchestration Languages

- Spring Web Flow

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                           http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd"
```

```
<start-state="welcome">

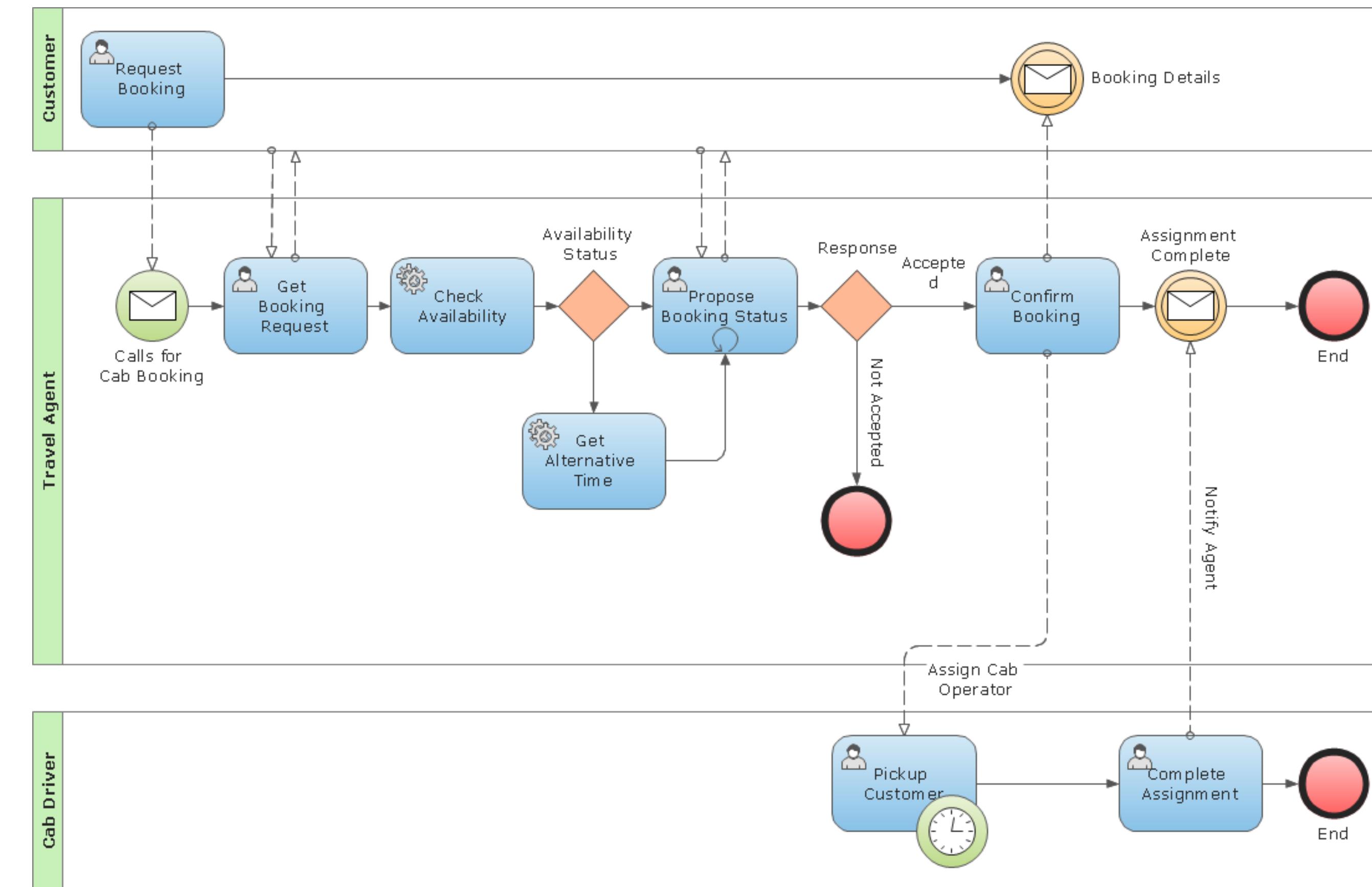
    <view-state id="welcome" view="/welcome.jsp">
        <transition on="continue" to="finish"/>
        <transition on="cancel" to="cancelled"/>
```

- Jolie



# Process Specification Languages

- From processes to code...



# Internet Applications Design and Implementation

## (Lecture 2, Part 4 - Software Architecture - RESTful applications)

**MIEI - Integrated Master in Computer Science and Informatics  
Specialization block**

**João Costa Seco ([joao.seco@fct.unl.pt](mailto:joao.seco@fct.unl.pt))**

(with previous participations of Jácome Cunha ([jacome@fct.unl.pt](mailto:jacome@fct.unl.pt)) and João Leitão ([jc.leitao@fct.unl.pt](mailto:jc.leitao@fct.unl.pt)))

# Restful interface design (Recap)

---

- Follows an architectural style (convention)
  - Architectural style that promotes a simpler and more efficient way of providing and connecting web services. Built on top of basic HTTP
- Promotes the decoupling from Data-centric server side applications and client user-centric applications
- Implementations provide (convenient) flavours
  - Web-service style pure JSON/XML Data
  - Complete/partial HTML view responses
  - Javascript code responses (e.g. Rails AJAX responses)
- Fielding, Roy Thomas (2000). "Chapter 5: Representational State Transfer (REST)".  
Architectural Styles and the Design of Network-based Software Architectures (Ph.D.).  
University of California, Irvine

# REST - Representational State Transfer

---

- Resource Based
- Representation
- Uniform Interface
- Stateless
- Cacheable
- Client-Server
- Layered System
- Code on Demand (optional)

# Representational State Transfer

---

- Resource Based
  - vs Action Based
  - Nouns and not verbs to identify data in the system
  - Identified (represented) by URI
  - Aliasing is admissible
- Representation
- Uniform Interface
- Stateless
- Cacheable
- Client-Server
- Layered System
- Code on Demand (not talking about it)

# Representational State Transfer

---

- Resource Based
- Representation
  - JSON or XML representation of the state of a given resource transferred between client and server at a given verb in a given URL.
  - Well identified interface (the information retrieved at an URL — the type)
- Uniform Interface
- Stateless
- Cacheable
- Client-Server
- Layered System
- Code on Demand (not talking about it)

# Representational State Transfer

---

- Resource Based
- Representation
- 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
- Cacheable
- Client-Server
- Layered System
- Code on Demand (not talking about it)

# Representational State Transfer

---

- Resource Based
- Representation
- Uniform Interface
- Stateless
  - Server does not hold session state
  - Messages are self contained
- Cacheable
- Client-Server
- Layered System
- Code on Demand (not talking about it)

# Representational State Transfer

---

- Resource Based
- Representation
- Uniform Interface
- Stateless
- Cacheable
  - Responses can be tagged as cacheable (in the server)
  - (also) Bookmarkable
- Layered System
- Code on Demand (not talking about it)

# Representational State Transfer

---

- Resource Based
- Representation
- Uniform Interface
- Stateless
- Cacheable
- Layered System
  - Establishes an API between a client and a “database”
  - Code on Demand (not talking about it)



## 6. Real REST Examples

Here's a very partial list of service providers that use a REST API. Note that some of them also support a WSDL (Web Services) API, in addition, so you can pick which to use; but in most cases, when both alternatives are available, REST calls are easier to create, the results are easier to parse and use, and it's also less resource-heavy on your system.

So without further ado, some REST services:

- The Google Glass API, known as "[Mirror API](#)", is a pure REST API. Here is [an excellent video talk](#) about this API. (The actual API discussion starts after 16 minutes or so.)
- Twitter has a **REST API** (in fact, this was their original API and, so far as I can tell, it's still the main API used by Twitter application developers),
- [Flickr](#),
- [Amazon.com](#) offer several REST services, e.g., for their [S3 storage solution](#),
- [Atom](#) is a RESTful alternative to RSS,
- [Tesla Model S](#) uses an (undocumented) REST API between the car systems and its Android/iOS apps.

in ... <http://rest.elkstein.org/2008/02/real-rest-examples.html>

```
interface
procedure ClrScr;
procedure SetColor(x,y:integer);
procedure SetTextColor(TxtColor,
                      BackColor:integer);
procedure SetX(X:integer);
procedure SetY(Y:integer);
function ReadKey:char;
function KeyPressed:boolean;
procedure CtrlC;
```

# Mirror API - Google Glasses

## Contacts

For Contacts Resource details, see the [resource representation](#) page.

Method	HTTP request	Description
URIs relative to <a href="https://www.googleapis.com/mirror/v1">https://www.googleapis.com/mirror/v1</a> , unless otherwise noted		
<a href="#">delete</a>	DELETE /contacts/ <i>id</i>	Deletes a contact.
<a href="#">get</a>	GET /contacts/ <i>id</i>	Gets a single contact by ID.
<a href="#">insert</a>	POST /contacts	Inserts a new contact.
<a href="#">list</a>	GET /contacts	Retrieves a list of contacts for the authenticated user.
<a href="#">patch</a>	PATCH /contacts/ <i>id</i>	Updates a contact in place. This method supports <a href="#">patch semantics</a> .
<a href="#">update</a>	PUT /contacts/ <i>id</i>	Updates a contact in place.

in ... <https://developers.google.com/glass/v1/reference/>

# Mirror API - Google Glasses

## Timeline

For Timeline Resource details, see the [resource representation](#) page.

Method	HTTP request	Description
URIs relative to <a href="https://www.googleapis.com/mirror/v1">https://www.googleapis.com/mirror/v1</a> , unless otherwise noted		
<a href="#">delete</a>	DELETE /timeline/ <i>id</i>	Deletes a timeline item.
<a href="#">get</a>	GET /timeline/ <i>id</i>	Gets a single timeline item by ID.
<a href="#">insert</a>	POST <a href="https://www.googleapis.com/upload/mirror/v1/timeline">https://www.googleapis.com/upload/mirror/v1/timeline</a> and POST /timeline	Inserts a new item into the timeline.
<a href="#">list</a>	GET /timeline	Retrieves a list of timeline items for the authenticated user.
<a href="#">patch</a>	PATCH /timeline/ <i>id</i>	Updates a timeline item in place. This method supports <a href="#">patch semantics</a> .
<a href="#">update</a>	PUT <a href="https://www.googleapis.com/upload/mirror/v1/timeline/">https://www.googleapis.com/upload/mirror/v1/timeline/</a> <i>id</i> and PUT /timeline/ <i>id</i>	Updates a timeline item in place.

# Mirror API - Google Glasses

## Timeline.attachments

For Timeline.attachments Resource details, see the [resource representation](#) page.

Method	HTTP request	Description
URIs relative to <a href="https://www.googleapis.com/mirror/v1">https://www.googleapis.com/mirror/v1</a> , unless otherwise noted		
delete	DELETE /timeline/ <i>itemId</i> /attachments/ <i>attachmentId</i>	Deletes an attachment from a timeline item.
get	GET /timeline/ <i>itemId</i> /attachments/ <i>attachmentId</i>	Retrieves an attachment on a timeline item by item ID and attachment ID.
insert	POST <a href="https://www.googleapis.com/upload/mirror/v1/timeline/itemId/attachments">https://www.googleapis.com/upload/mirror/v1/timeline/<i>itemId</i>/attachments</a>	Adds a new attachment to a timeline item.
list	GET /timeline/ <i>itemId</i> /attachments	Returns a list of attachments for a timeline item.

in ... <https://developers.google.com/glass/v1/reference/>

# Tesla API

 Tesla JSON API (Unofficial)

[GitHub](#) [Tesla](#)

Introduction	
API BASICS	
Authentication	
Vehicles	
VEHICLE	
State	>
Commands	▼
Wake	
Alerts	
<b>Remote Start</b>	
Homelink	
Speed Limit	
Valet Mode	
Sentry Mode	
Doors	
Frunk/Trunk	
Windows	
Surfboard	

## Remote Start

**POST**

**/api/1/vehicles/{id}/command/remote\_start\_drive**

Enables keyless driving. There is a two minute window after issuing the command to start driving the vehicle.

### Parameters

Parameter	Example	Description
password	edisonsux	The password used to log in to the vehicle.

### Response

```
1  {
2    "reason": "",
3    "result": true
4 }
```

```
49 - request:
50   method: post
51   uri: https://owner-api.teslamotors.com/api/1/vehicles/1514029006966957156/command/actuate_trunk
52   body:
53     encoding: UTF-8
54     string: which_trunk=front
55   headers:
56     Authorization:
57       - Bearer <TESLA_ACCESS_TOKEN>
58   response:
59     status:
60       code: 200
61       message: OK
62     headers:
63       Server:
64         - nginx
65       Date:
66         - Sun, 05 Aug 2018 17:04:59 GMT
67       Content-Type:
68         - application/json; charset=utf-8
```

master ▾ [tesla-api / spec / cassettes /](#)

timdorr Handle an invalid passcode

..

[client-login-mfa-invalid.yml](#) Handle an invalid passcode

[client-login-mfa.yml](#) Add MFA detection/support to login

[client-login.yml](#) Update specs and cassettes for new

[client-refresh.yml](#) Update specs and cassettes for new

[client-vehicles.yml](#) Update specs.

[vehicle-activate\\_speed\\_limit.yml](#) Revamp Client.

[vehicle-auto\\_conditioning\\_start.yml](#) Test the remaining action endpoints.

[vehicle-auto\\_conditioning\\_stop.yml](#) Test the remaining action endpoints.

[vehicle-cancel\\_software\\_update.yml](#) Revamp Client.

# RESTful design

---

- Resource = object or representation of something
- Collection = a set of resources
- URI = a path identifying **resources** and allowing actions on them
- URL methods represents standardised actions
  - GET = request resources
  - POST = create resources
  - PUT = update or create resources
  - DELETE = deletes resources
- HTTP Response codes = operation results
  - 20x Ok
  - 3xx Redirection (not modified)
  - 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found
  - 5xx Server Error
- Searching, sorting, filtering and pagination obtained by query string parameters
- Text Based Data format (JSON, or XML)

<https://hackernoon.com/restful-api-designing-guidelines-the-best-practices-60e1d954e7c9>

# Example

---

- Application to manage contacts of partner companies (e.g. for security clearance in events)
- Resources
  - Companies (name, address, email, list of contacts (employees))
  - Contact/Employee (name, email, job, company)
- Operations (CRUD)
  - List, add, 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

---

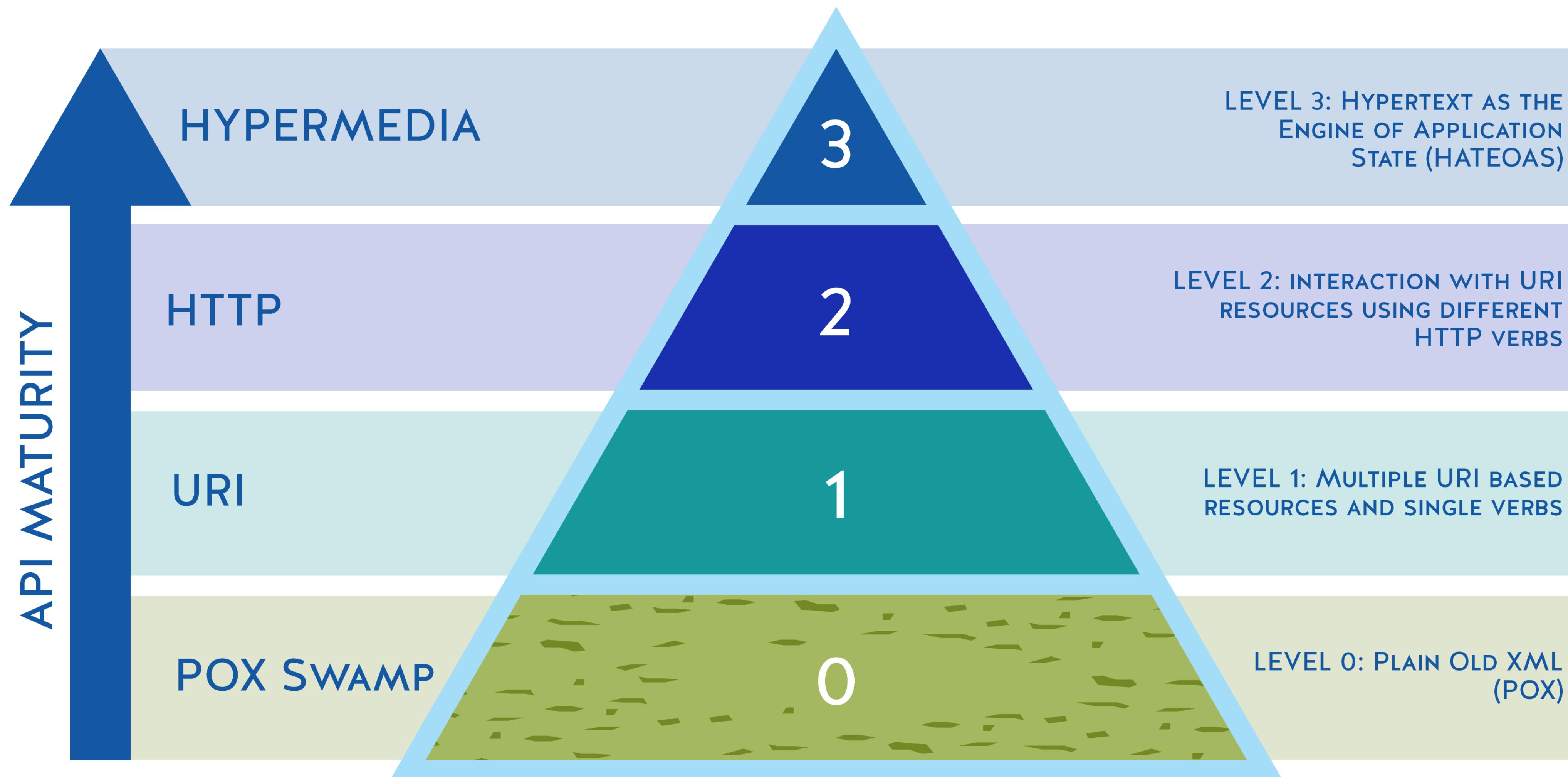
- **GET /contacts** - List all the contacts
- **GET /contacts?search=<criteria>** - List all the contacts that contain the substring **<criteria>**
- **POST /contacts** - Create a contact described in the payload. The request body must include all the necessary attributes.
- **GET /contacts/{id}** -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

# THE RICHARDSON MATURITY MODEL



# Example: Contacts in a Spring Controller and Java



```
@RestController
@RequestMapping("/people")
public class PeopleController {

    @Autowired
    PeopleRepository people;

    @Autowired
    PetRepository pets;

    @GetMapping("")
    Iterable<Person> getAllPersons(@RequestParam(required = false) String search)
    {
        if( search == null )
            return people.findAll();
        else
            return people.searchByName(search);
    }

    @PostMapping("")
    void addNewPerson(@RequestBody Person p) {
        p.setId(0);
        people.save(p);
    }

    @GetMapping("{id}")
    Optional<Person> getOne(@PathVariable long id) {
        return people.findById(id);
    }
}
```

# JAX-RS: A standard for API declaration

- A lightweight specification method with (Java) annotations
- Implemented by RESTEasy and Jersey
- Similar to Spring annotations
- Official Java Specification
- [//jcp.org/en/jsr/detail?id=339](http://jcp.org/en/jsr/detail?id=339)

```
@Path("/notifications")
public class NotificationsResource {
    @GET
    @Path("/ping")
    public Response ping() {
        return Response.ok().entity("Service online").build();
    }

    @GET
    @Path("/get/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getNotification(@PathParam("id") int id) {
        return Response.ok()
            .entity(new Notification(id, "john", "test notification"))
            .build();
    }

    @POST
    @Path("/post/")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response postNotification(Notification notification) {
        return Response.status(201).entity(notification).build();
    }
}
```