National College of Ireland

Topic: Software Architecture & Case Study

Course: H9EEAI- Engineering and Evaluating Artificial Intelligence Systems

Outline

Definition: What is Software Architecture?

Architectural Thinking

- Architecture vs Design
- Technical Breadth
- Analysing tradeoffs
- Understanding business drivers

Important Software Architecture Concepts

- Client Server Model , API and Modularity

Case Study: Notflix a hypothetical system

- Monolethic Architecture
- Vertical and Horizontal Scaling
- Microservices
- Back-end For Front-end (BFF)
  Hosting & Folder Structure

National College of Ireland

What is software architecture?

Definition of SA is hard??

Some architects refer to software architecture as the *blueprint* of the system,

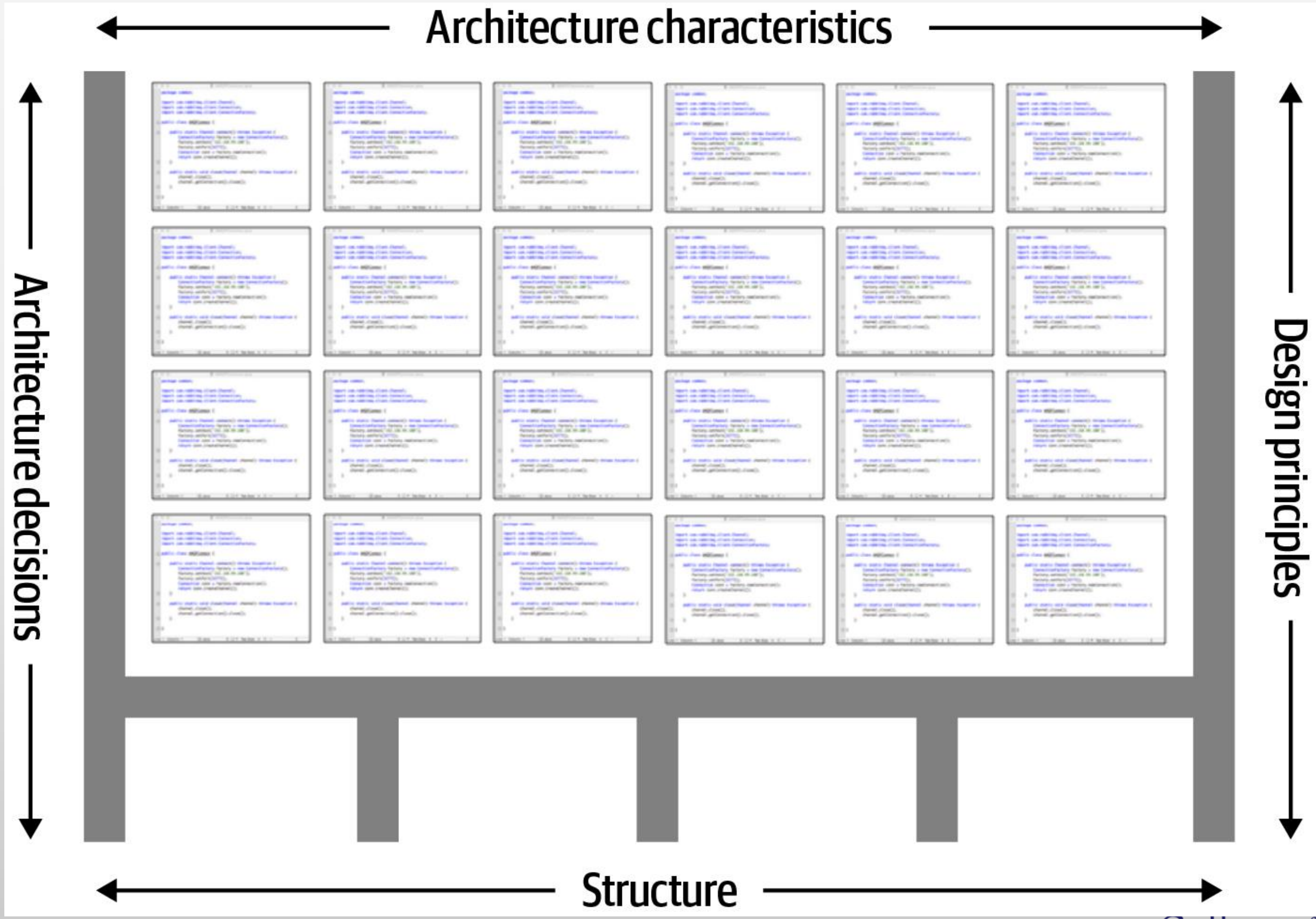
While others define it as the *roadmap* for developing a system.

So, then the confusion arise, What blueprint or roadmap contains? For example, what is analyzed when an architect *analyzes* an architecture?

National College of Ireland

Software Architecture

Architecture consists of the structure combined with architecture characteristics ("-ilities"), architecture decisions, and design principles

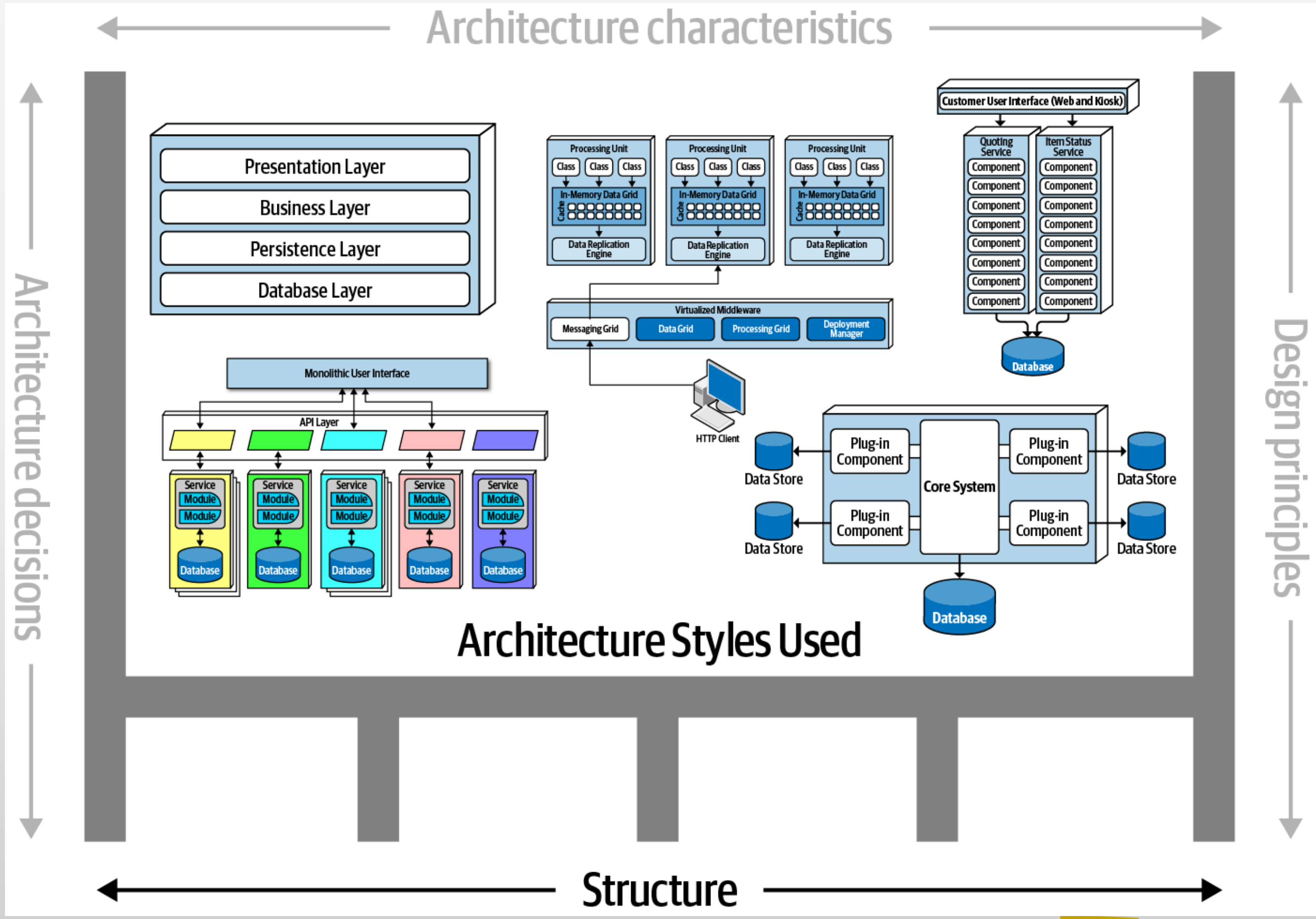


College of Ireland

The *structure* of the system

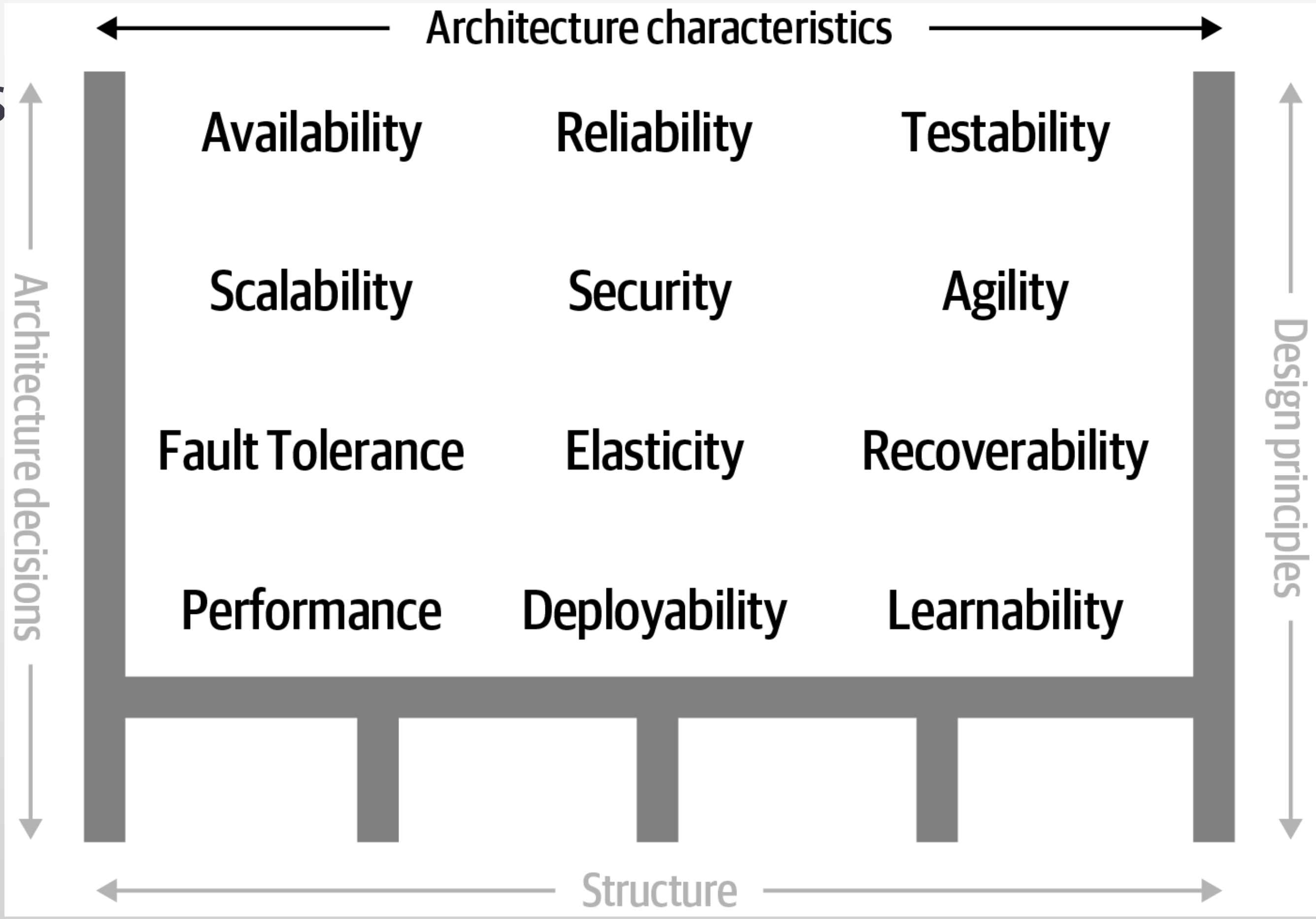
- Architecture style (or styles) – microservices, layered, monolithic or microkernel

- Describing an architecture solely by the structure does not wholly elucidate an architecture.



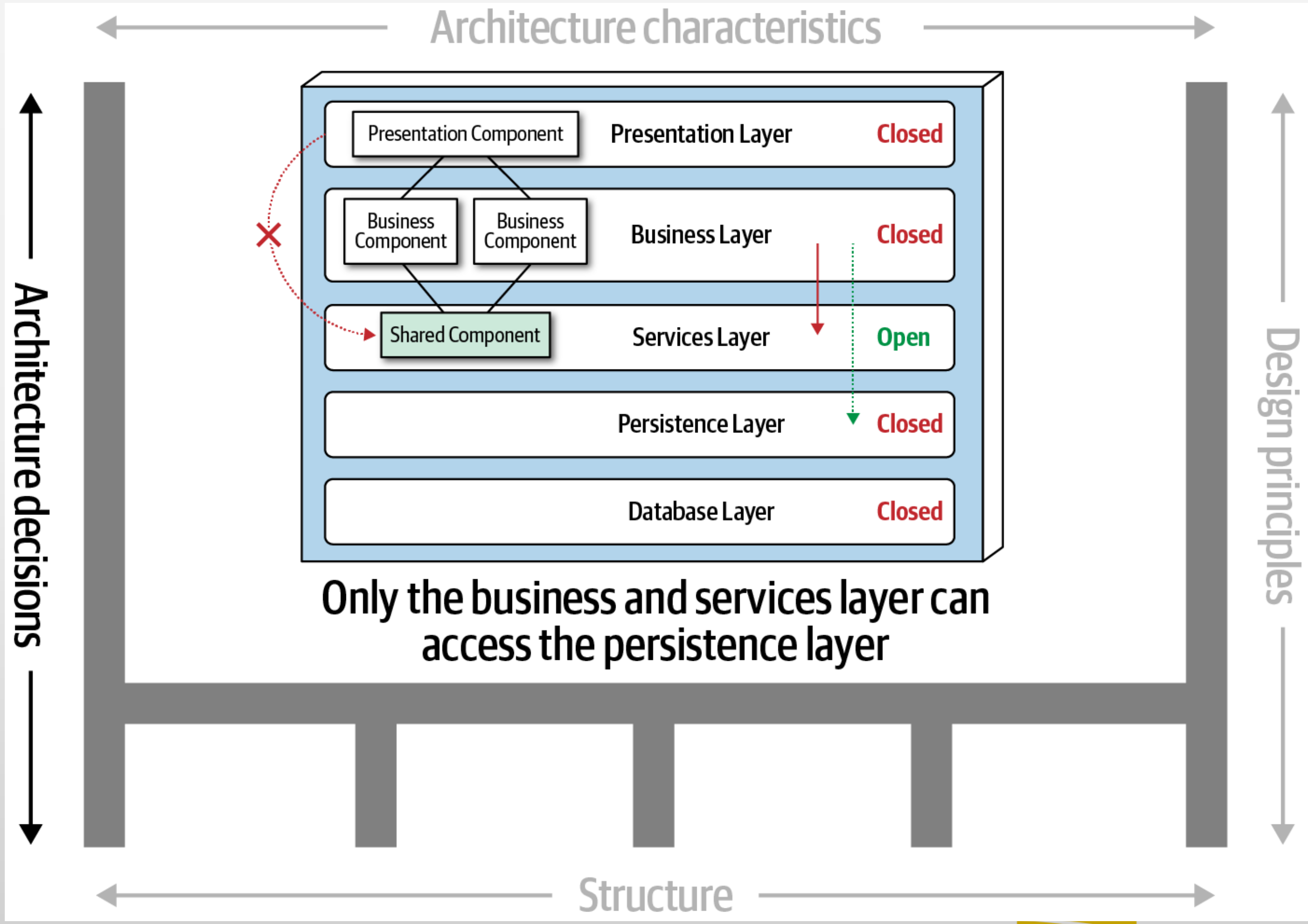
Architecture characteristics

- The architecture characteristics define the success criteria,
- Generally orthogonal to the functionality.



Architecture decisions

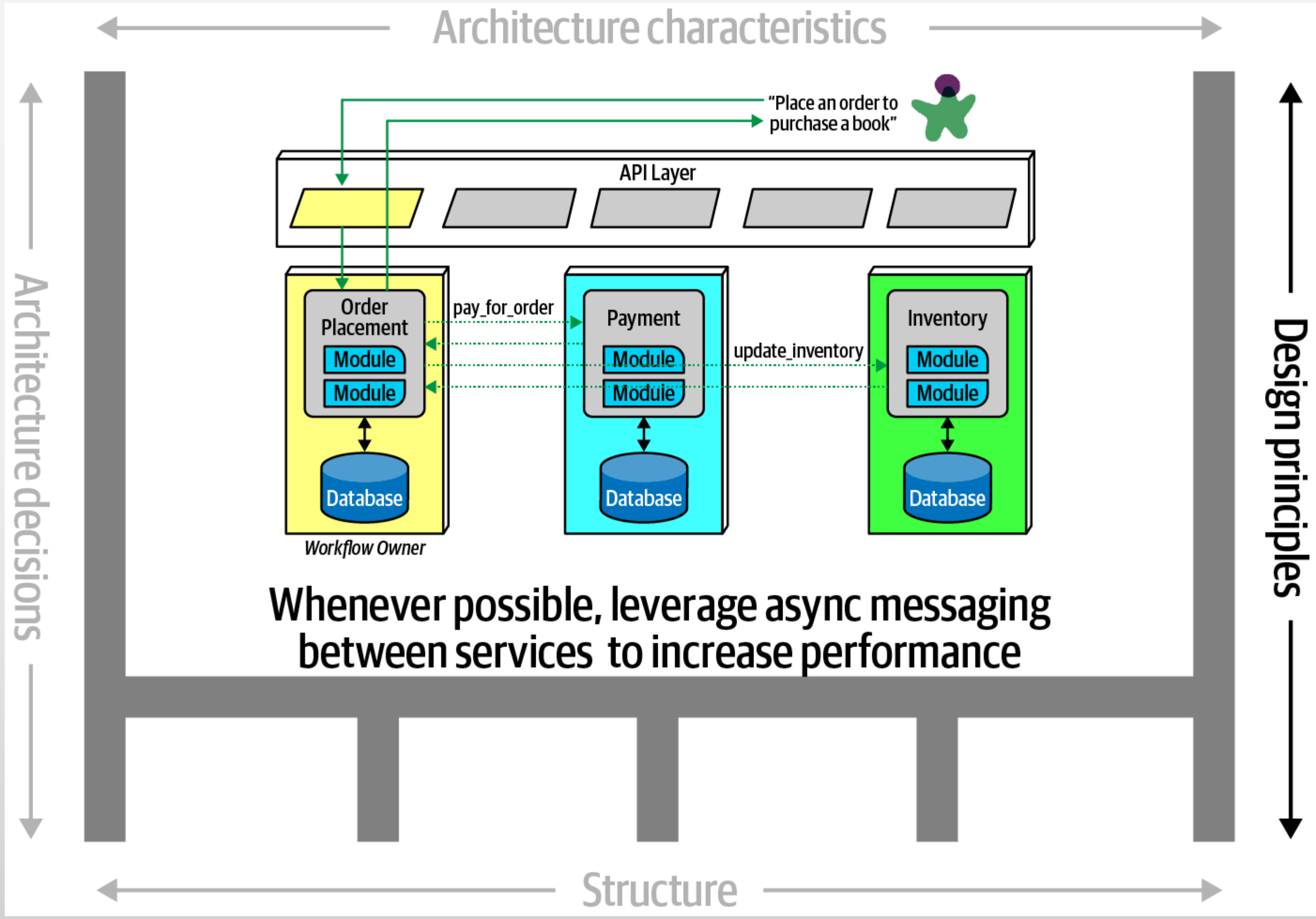
- Architecture decisions define the rules for how a system should be constructed.
- Architecture decisions form the constraints of the system and direct the development teams on what is and what isn't allowed.



Design principles

A design principle differs from an architecture decision in that a design principle is a guideline rather than a hard-and-fast rule.

For example: Development teams should leverage asynchronous messaging between services within a microservices



Expectations of an Architect

There are eight core expectations placed on a software architect, irrespective of any given role, title, or job description:

- Make architecture decisions
- Continually analyze the architecture
- Keep current with latest trends
- Ensure compliance with decisions
- Diverse exposure and experience
- Have business domain knowledge
- Possess interpersonal skills
- Understand and navigate politics

Architectural Thinking

An architect sees things differently from a developer's point of view, much in the same way a meteorologist might see clouds differently from an artist's point of view.

There are four main aspects of thinking like an architect.

- Understanding the difference between architecture and design and knowing how to collaborate with development teams to make architecture work.
- Wide breadth of technical knowledge while still maintaining a certain level of technical depth, allowing the architect to see solutions and possibilities that others do not see.
- It's about understanding, analysing, and reconciling trade-offs between various solutions and technologies.
- About understanding the importance of business drivers and how they translate to architectural concerns

Architecture Versus Design

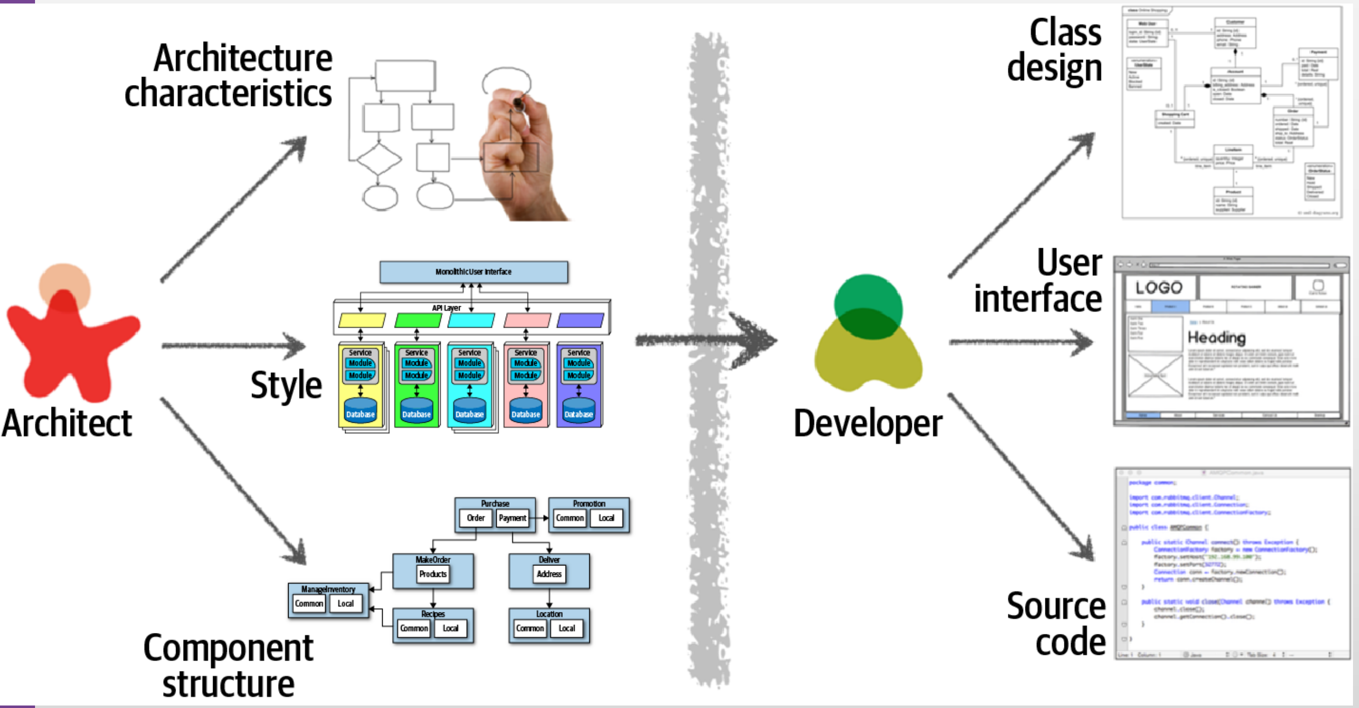
The difference between architecture and design is often a confusing one.

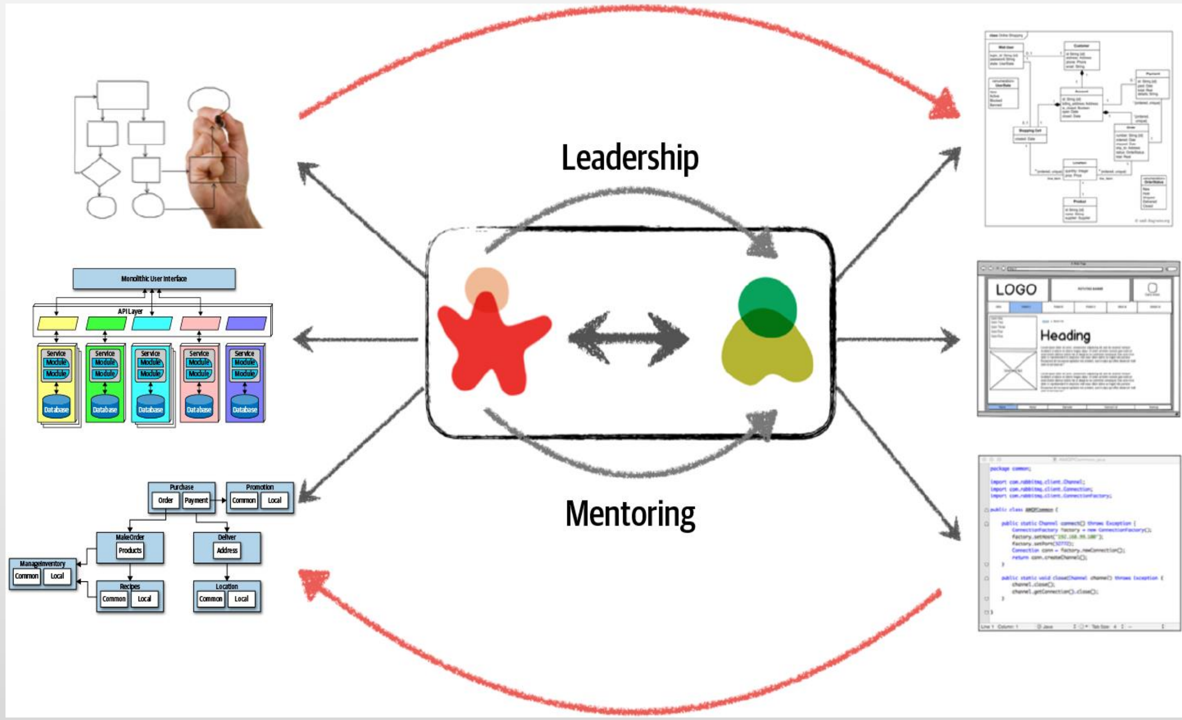
Where does architecture end and design begin?

What responsibilities does an architect have versus those of a developer?

Thinking like an architect is knowing the difference between architecture and design and seeing how the two integrate closely to form solutions to business and technical problems.

Traditional view vs Ideal View of architecture versus design



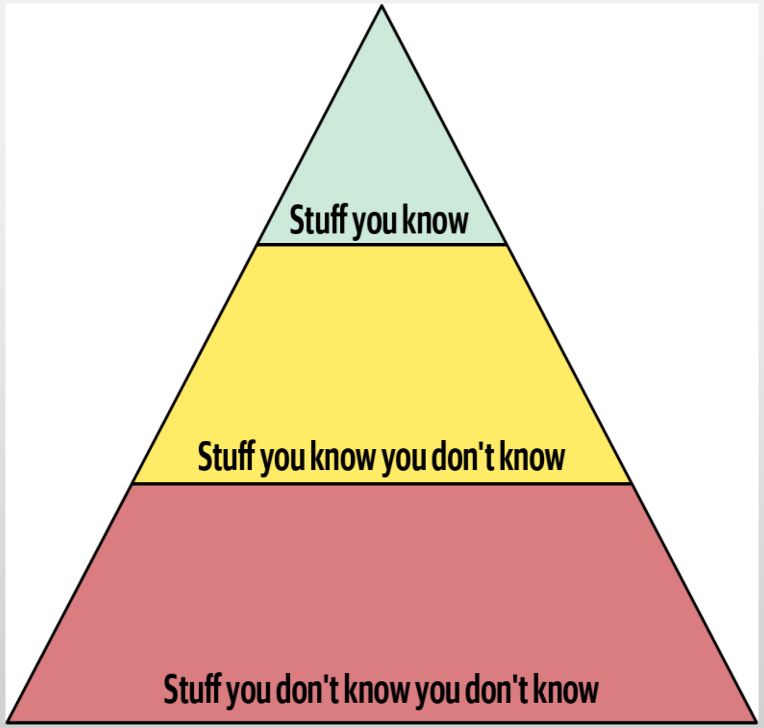


Technical Breadth

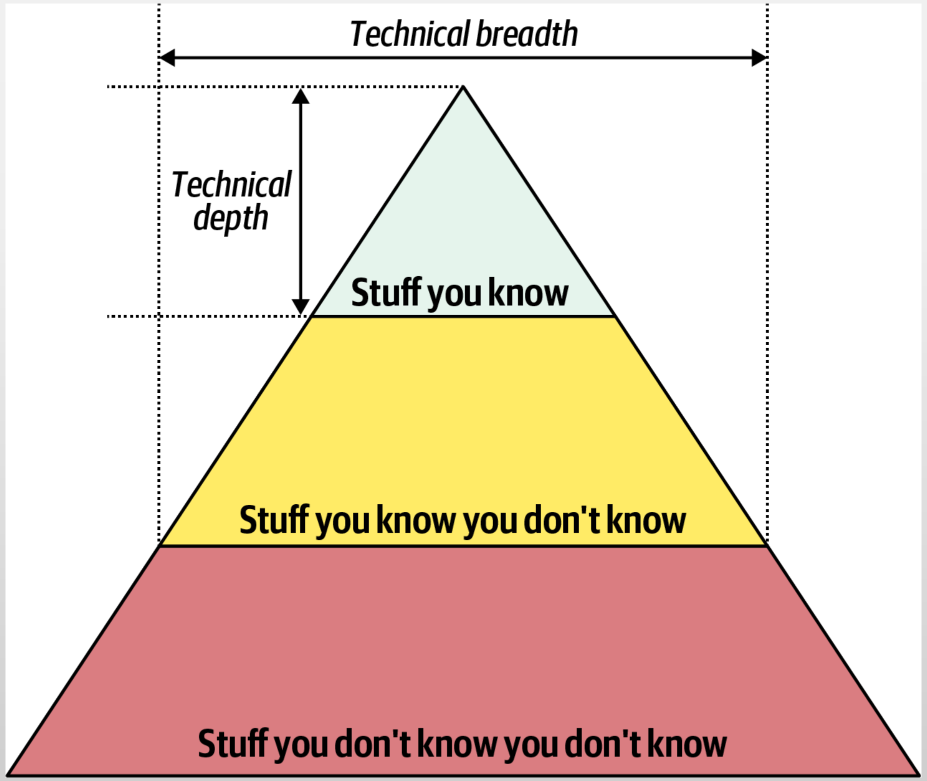
The scope of technological detail differs between developers and architects.

Unlike a developer, who must have a significant amount of **technical depth** to perform their job, a software architect must have a significant amount of **technical breadth** to think like an architect and see things with an architecture point of view.

Technical Breadth


Stuff you know
Stuff you know you don't know
Stuff you don't know you don't know

The pyramid representing all knowledge


Technical breadth
Technical depth
Stuff you know
Stuff you know you don't know
Stuff you don't know you don't know

National College of Ireland

Analysing Trade-Offs

Thinking like an architect is all about seeing trade-offs in every solution, technical or otherwise, and analyzing those trade-offs to determine what is the best solution.

Mark (the author of the book – you can find in reference) says:

“Architecture is the stuff you can’t Google.”

Everything in architecture is a trade-off, which is why the famous answer to every architecture question in the universe is “it depends.”

However, it is true because: It depends on the deployment environment, business drivers, company culture, budgets, timeframes, developer skill set, and dozens of other factors.

“There are no right or wrong answers in architecture—only trade-offs”

Understanding Business Drivers

Understanding the business drivers that are required for the success of the system and translating those requirements into architecture characteristics (such as scalability, performance, and availability).

Software Architecture

https://www.freecodecamp.org/news/an-introduction-to-software-architecture-patterns/

What is software architecture?

“The software architecture of a system represents the design decisions related to overall system structure and behavior.”

“software architecture refers to how you organize stuff in the process of creating software. And "stuff" here we can refer to:”

- **Implementation details** (that is, the folder structure of your repo)
- **Implementation design** decisions (Do you use server side or client side rendering? Relational or non-relational databases?)
- The **technologies** you choose (Do you use REST or GraphQl for your API? Python with Django or Node with Express for your back end?)
- **System design** decisions (like is your system a monolith or is it divided into microservices?)
- **Infrastructure** decisions (Do you host your software on premise or on a cloud provider?)

National College of Ireland

Important Software Architecture Concepts to Know

What's the client-server model?

- **Client-server** is a model that structures the tasks or workloads of an application between a resource or **service provider** (server) and a service or **resource requester** (client).
- Clients are normally represented by front-end applications
- Servers are usually back-end applications.
- Another important concept to know is that clients and servers are part of the same system, but each is an application/program on its own. Meaning they can be developed, hosted, and executed separately.

National College of Ireland

What are APIs?

Client and Server usually communicate through an API (application programming interface).

An API is nothing more than a set of defined rules that establishes how an application can communicate with another. It's like a contract between the two parts that says "If you send A, I'll always respond B. If you send C, I'll always respond D..." and so on.

There're different ways in which an API can be implemented. The most commonly used are REST, SOAP and GraphQL.

Regarding how APIs communicate, most often the HTTP protocol is used, and the content is exchanged in JSON or XML format. But other protocols and content formats are perfectly possible.

National College of Ireland

Jersey HelloWorld

- URL http://localhost:8080/REST_Lab/webapi/hello

```java
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
public class HelloResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHello() {
        return "Hello World!";
    }
}
```

Jersey Hello {name}

- ❑ URL http://localhost:8080/REST_Lab/webapi/hello/name/Patrick

```java
@Path("/hello")
public class HelloResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHello() {
        return "Hello World!";
    }

    @GET
    @Path("/name/{value}")
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHello(@PathParam("value") String value) {

        return "Hello " + value;
    }
}
```

XML

XML, a language that lets one write structured documents with a user-defined vocabulary.

A typical representation in xml

```
<book>
	<title>
		Nonmonotonic Reasoning: Context-Dependent Reasoning
	</title>
	<author>A. Mike</author>
	<author>M. Jhon</author>
	<publisher>Springer</publisher>
	<year>1993</year>
	<ISBN>0387976892</ISBN>
</book>
```

National College of Ireland

XML data model

All tags in XML must be closed (for example, for an opening tag <title> there must be a closing tag </title>), while some tags may be left open in HTML (such as <br />).

Also referred as Element

The enclosed content, together with the corresponding opening and closing tag,

XML allows to represent information that is also machine-accessible.

XML separates content from use and presentation.

What is Modularity?

- Modularity means the practice of dividing big things into smaller pieces.
- This practice of breaking things down is performed to simplify big applications or codebases.
- Cohesion
  - Cohesion refers to what extent the parts of a module should be contained within the same module.
  - In other words, it is a measure of how related the parts are to one another.
  - Ideally, a cohesive module is one where all the parts should be packaged together.
- Coupling
  - Coupling refers to the degree of interdependence or connectivity between software components or modules within a system.
  - It measures how closely two or more components are linked or reliant on each other.

Modularity

Customer Maintenance

- add customer
- update customer
- get customer
- notify customer
- get customer orders • cancel customer orders

Should the last two entries reside in this module or should the developer create two separate modules, such as:

Customer Maintenance

• add customer • update customer • get customer • notify customer

Order Maintenance

• get customer orders • cancel customer orders

Which is the correct structure?

As always, it depends:

Are those the only two operations for Order Maintenance? If so, it may make sense to collapse those operations back into Customer Maintenance.

Is Customer Maintenance expected to grow much larger, encouraging developers to look for opportunities to extract behavior?

Does Order Maintenance require so much knowledge of Customer information that separating the two modules would require a high degree of coupling to make it functional?

National College of Ireland

Advantages of Modularity

- It's good for dividing concerns and features, which helps with the visualization, understanding, and organization of a project.
- The project tends to be easier to maintain and less prone to errors and bugs when it's clearly organized and subdivided.
- If your project is subdivided into many different pieces, each can be worked on and modified separately and independently, which is often very useful.

National College of Ireland

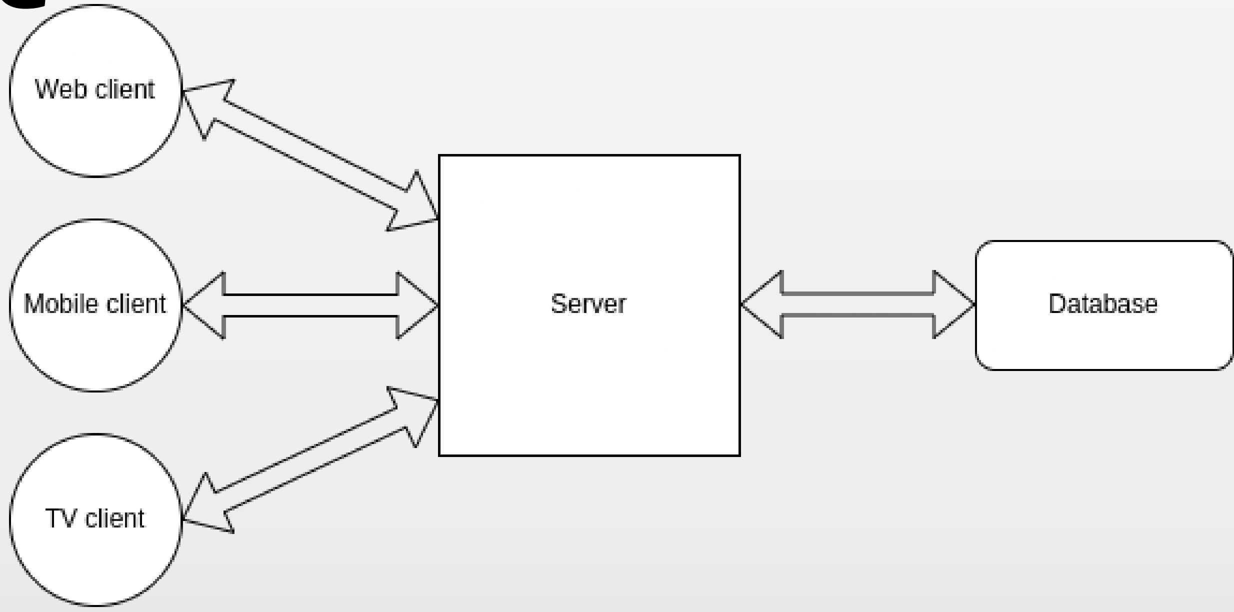
Case Study: hypothetical application Notflix

- Notflix will be a typical video streaming application, in which the user will be able to watch movies, series, documentaries and so on. The user will be able to use the app in web browsers, in a mobile app, and on a TV app, too.
- The main services included in our app will be
  - **Authentication** (so people can create accounts, login, and so on),
  - **Payments** (so people can subscribe and access the content... because you didn't think this was all for free, right?) and
  - **Streaming** of course (so people can watch what they're paying for).

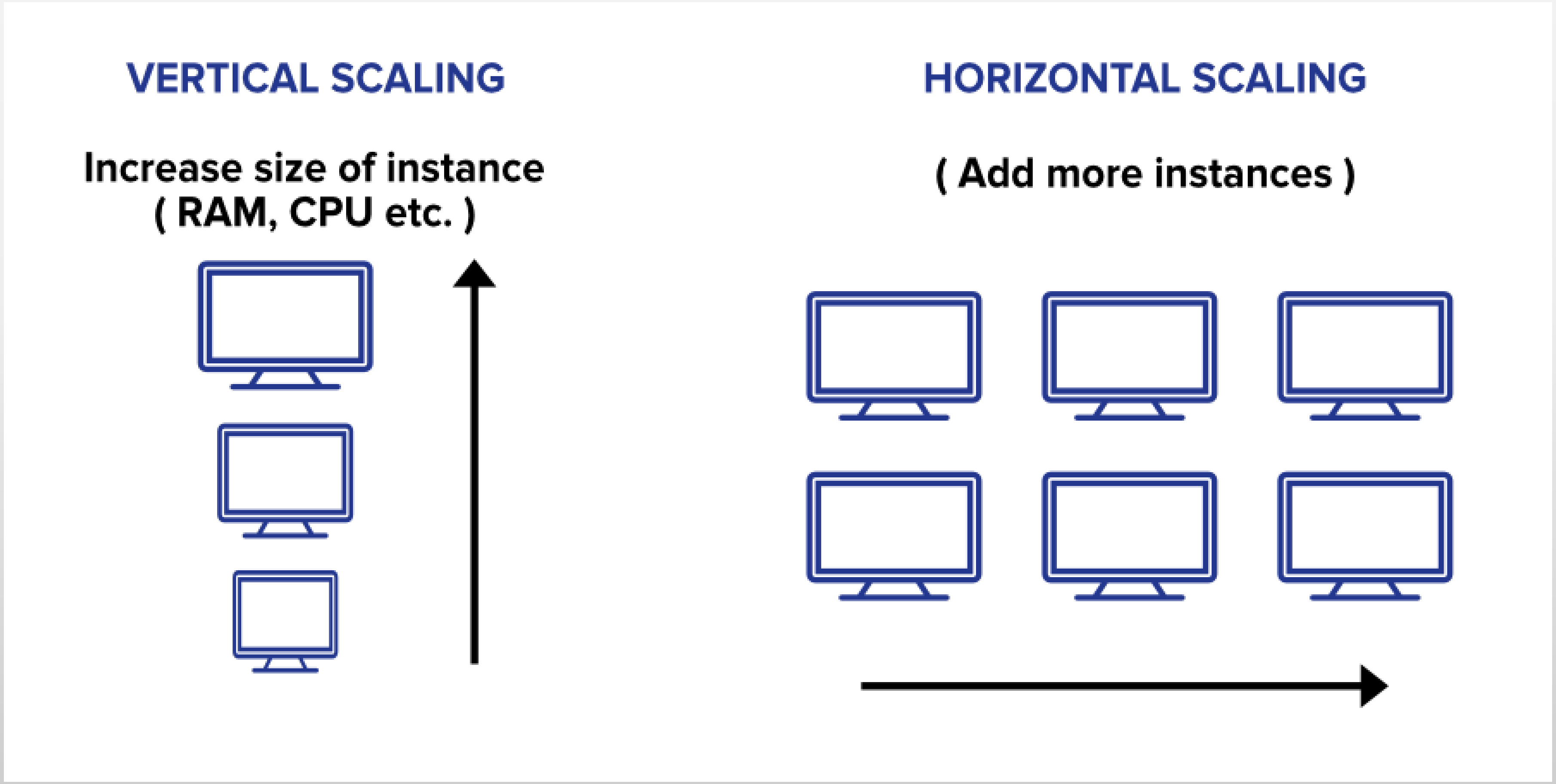
National College of Ireland

Monolithic Architecture

This kind of architecture is called a **monolithic** because there's a single server application that is responsible for all the features of the system. In our case, if a user wants to authenticate, pay us, or watch one of our movies, all the requests are going to be sent to the same server application.



Vertical scaling and Horizontal Scaling



https://www.geeksforgeeks.org/system-design-horizontal-and-vertical-scaling/

Notflix Success

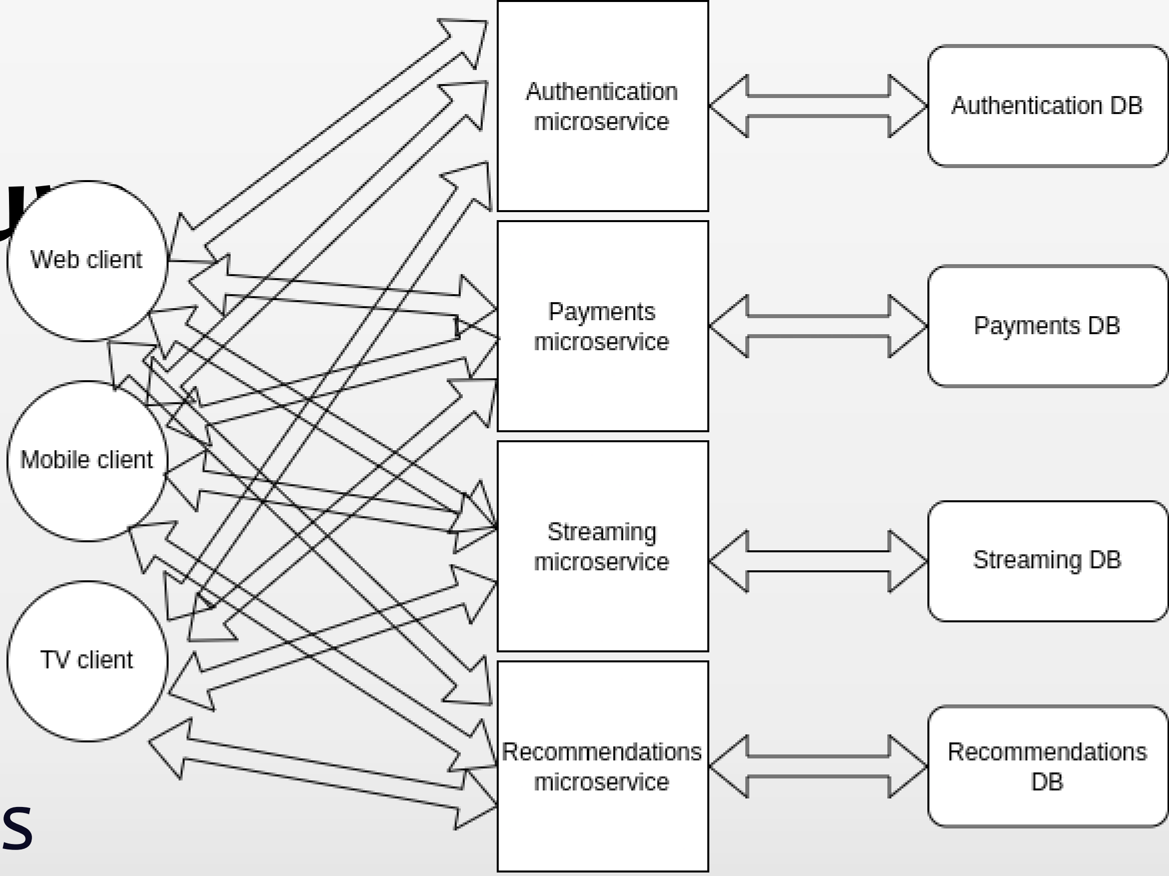
- Additionally developing new features into our system (such as a recommendation tool that reads the user's preferences and recommends movies that suit the user profile) and **our codebase is starting to look huge and very complex** to work with.
- Analyzing this problem in depth, we've found the feature that takes the most resources is streaming, while other services such as authentication and payments don't represent a very big load.

Microservices Architectu

Dividing server-side features into many small servers that are responsible for only one or a few specific features.

After implementing microservices

- a server responsible for **authentication,**
- another responsible for **payments,**
- another for **streaming,**
- and the last one for **recommendations.**



Benefits of Microservices

- You can **scale particular services as needed**

  Now that we have the streaming feature separated into a single server, we can scale only that one and leave the rest alone as long as they keep working right.

- Features will be more **loosely coupled**, which means we'll be able to develop and deploy them independently.
- The **codebase** for each server will be much smaller and **simpler**.

  Which is nice for the dev folks that have been working with us from the start, and also easier and quicker for new developers to understand.

Communication Issue with Microservices

Communication with front-end apps gets more complex.

Now we have many servers responsible for different things, which means front-end apps would need to keep track of that info to know who to make requests to.

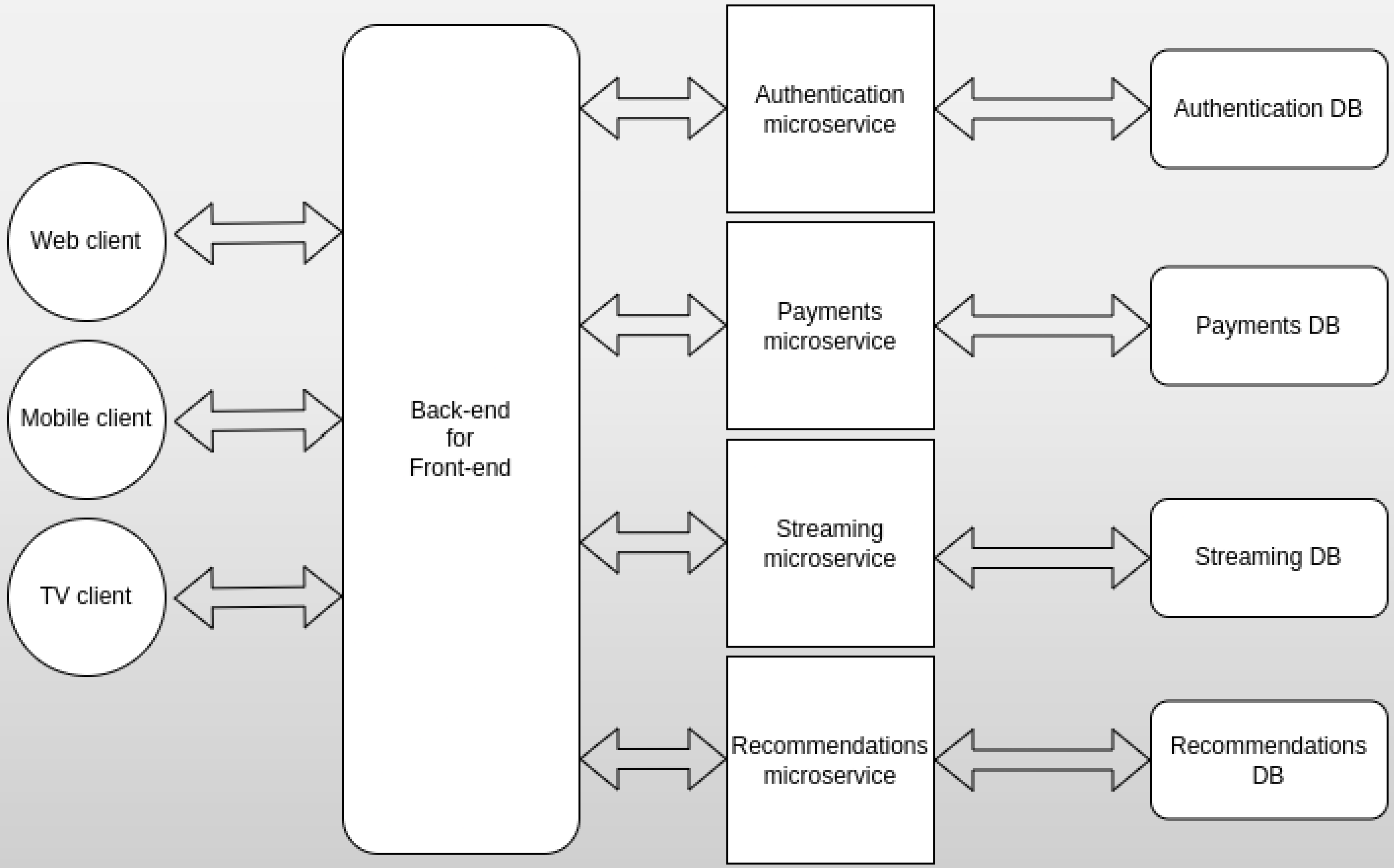
Solution:

This problem gets solved by implementing an **intermediary layer** between the front-end apps and the microservices.

This layer will receive all the front-end requests, redirect them to the corresponding microservice, receive the microservice response, and then redirect the response to the corresponding front-end app.

National College of Ireland

Back-end For Front-end (BFF)

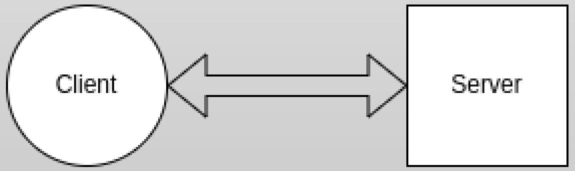


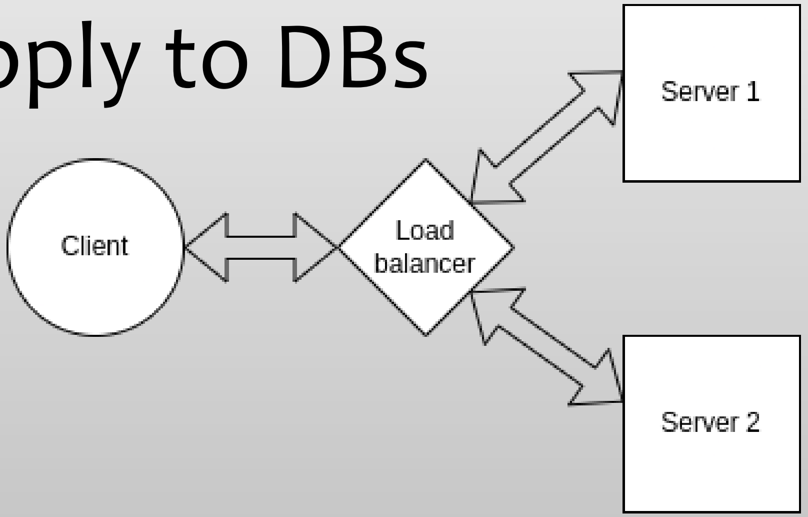
How to use load balancers and horizontal scaling?

Horizontal Scaling works but the there we need load balancers.

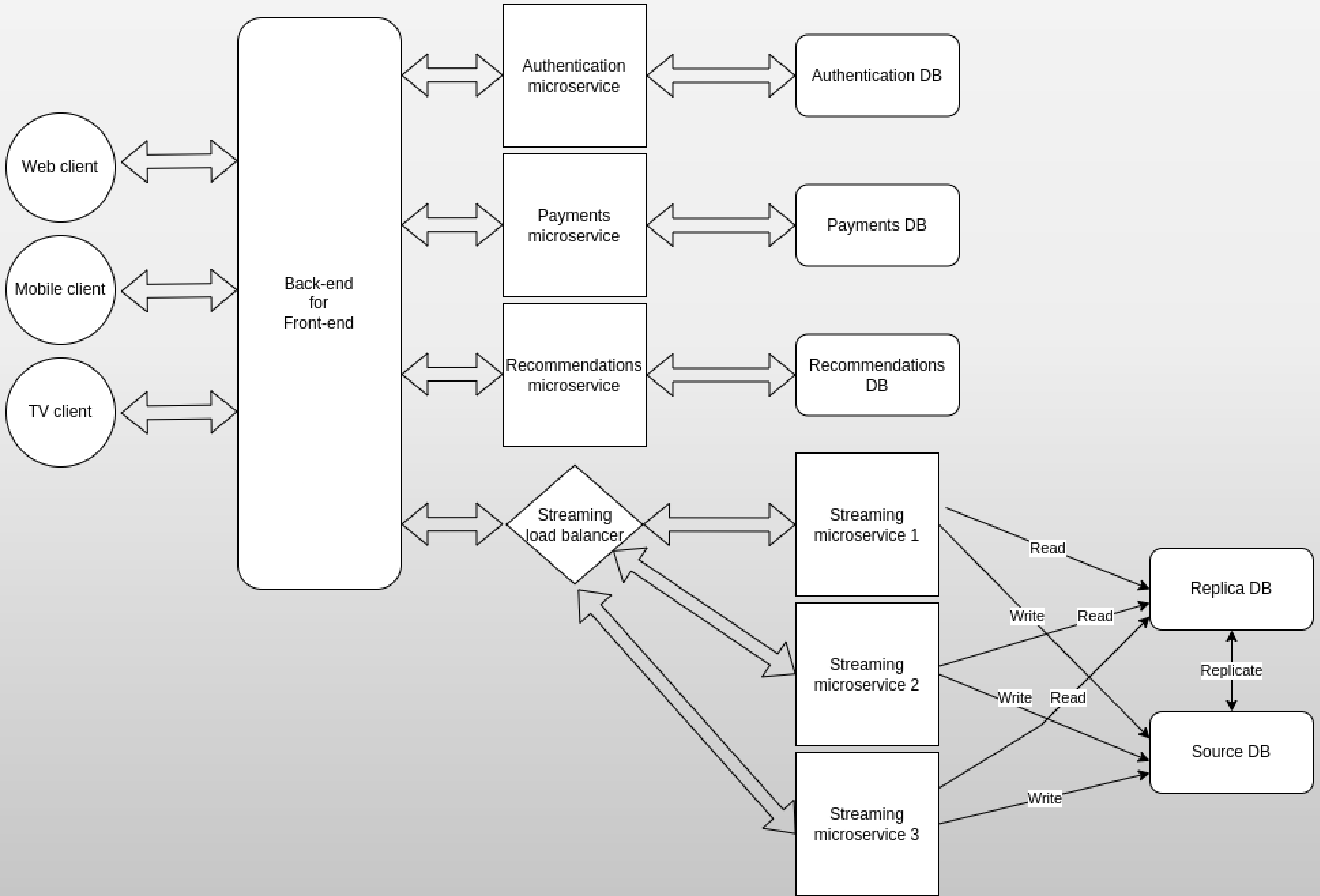
Horizontally scaling means setting up more servers to perform the same task. Load balancers act as **reverse proxys** to our servers.

Horizontal scaling may also apply to DBs





Horizontally scaled architecture



Where Your Infrastructure Lives?

Now that we have a basic idea of how an application infrastructure might be organized, the next thing to think about is where we're going to put all this stuff.

On-Premise Hosting

On premise means you own the hardware in which your app is running.

The good thing about this option is that the company gets total control over the hardware.

However

Horizontal scaling and later returning of the equipment’s won’t be possible

Think of usage???

One situation in which on premise servers still make sense is when dealing with very delicate or private information.

National College of Ireland

Traditional Server Providers

A more comfortable option for most companies are traditional server providers.

These are companies that have servers of their own and they just rent them.

What's great about this option is that you don't need to worry about anything hardware-related anymore.

Another cool thing is that scaling up or down is easy and risk free.

If you need more hardware, you pay for it. And if you don't need it anymore, you just stop paying.

National College of Ireland

Hosting on the Cloud

The hardware of big tech still represents a cost whether they're using it or not, the clever thing to do is to commercialize that computing power to others.

AWS (Amazon web services), **Google Cloud**, or Microsoft **Azure**

Traditional

The first way is to use them in a similar way you'd use a traditional server provider. You select the kind of hardware you want and pay exactly for that on a monthly basis.

Elastic

The second way is to take advantage of the "elastic" computing offered by most providers. "Elastic" means that the hardware capacity of your application will automatically grow or shrink depending on the usage your app has.

Serverless

Another way in which you can use cloud computing is with a serverless architecture.

Following this pattern, you won’t have a server that receives all requests and responds to them. Instead, you'll have individual functions mapped to an access point (similar to an API endpoint).

National College of Ireland

Different Folder Structures to Know

All in One Place Folder Structure

So, what's the problem with this? Nothing it works just fine.

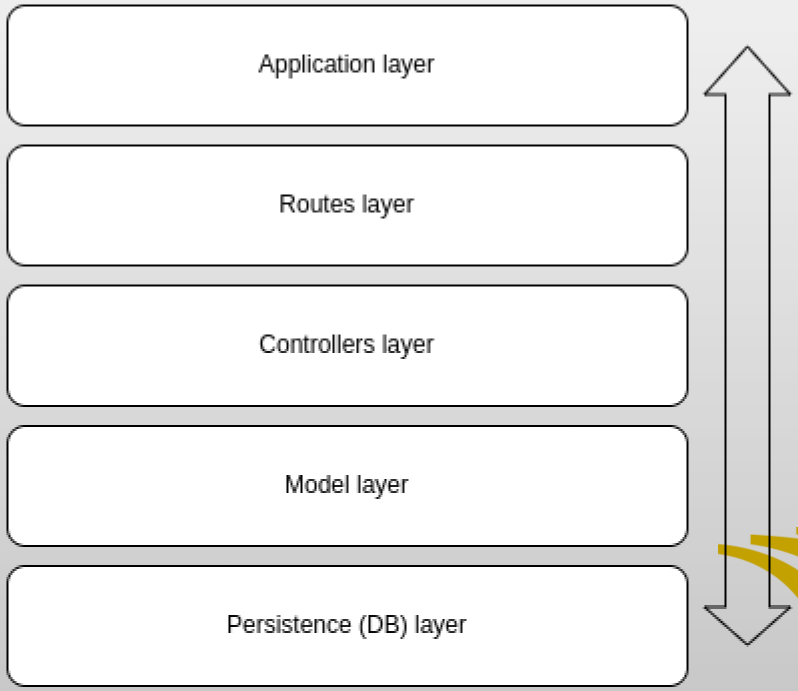
The problem will only arise when the codebase gets bigger and more complex, and we start adding new features to our API.

Following the modularity principle, a better idea is to have different folders and files for the different responsibilities and actions we need to perform.

Layers Folder Structure

Layers architecture is about dividing concerns and responsibilities into different folders and files, and allowing direct communication only between certain folders and files.

- The application layer will have the basic setup of our server and the connection to our routes.
- The routes layer will have the definition of all of our routes and the connection to the controllers (the next layer).
- The controllers layer will have the actual logic we want to perform in each of our endpoints and the connection to the model layer
- The model layer will hold the logic for interacting with database.
- Finally, the persistence layer is where database will be.

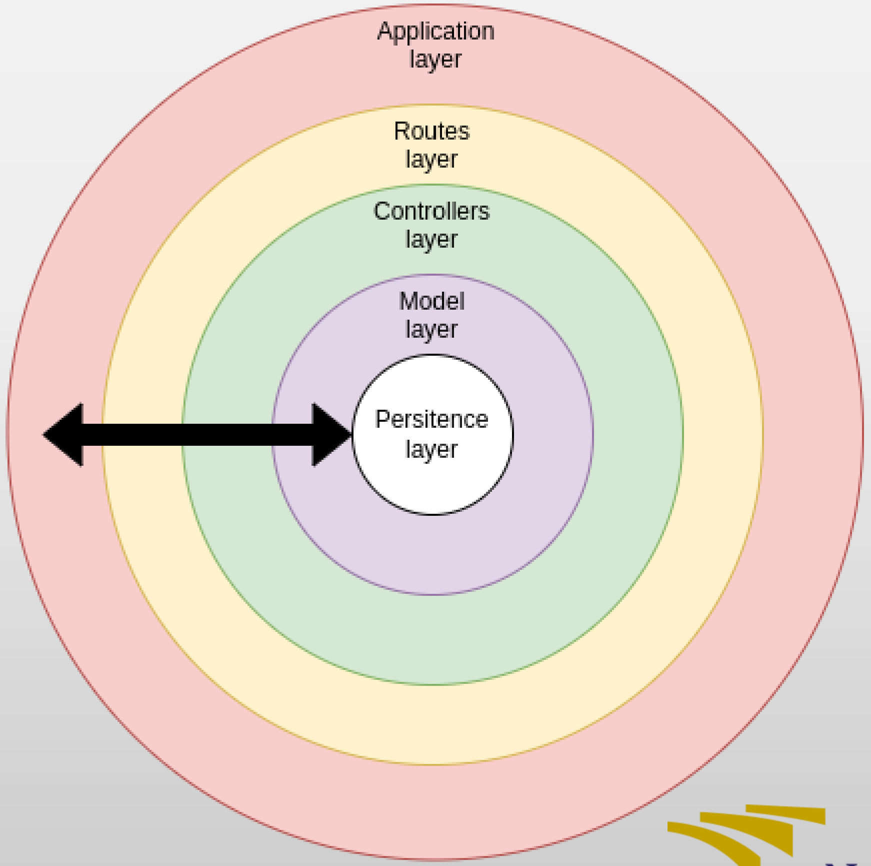


Layers Folder Structure

There's a defined communication flow between the layers.

This means that a request first
has to go through the first layer,
then the second, and so on.

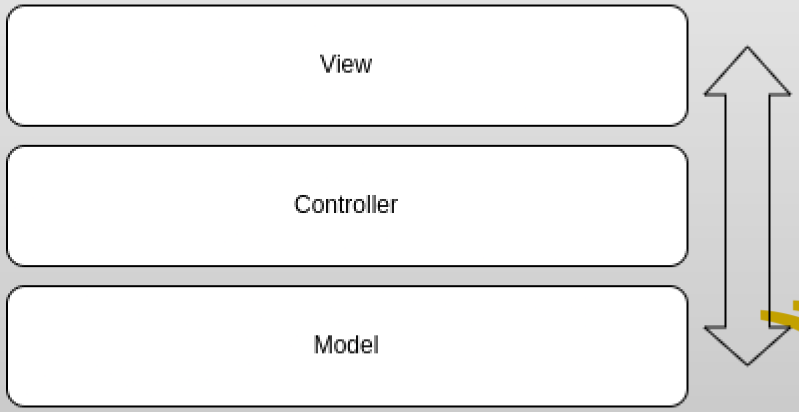
No request should skip layers because
that would mess with the logic of
the architecture and the benefits
of organization and modularity it gives us.



MVC Folder Structure

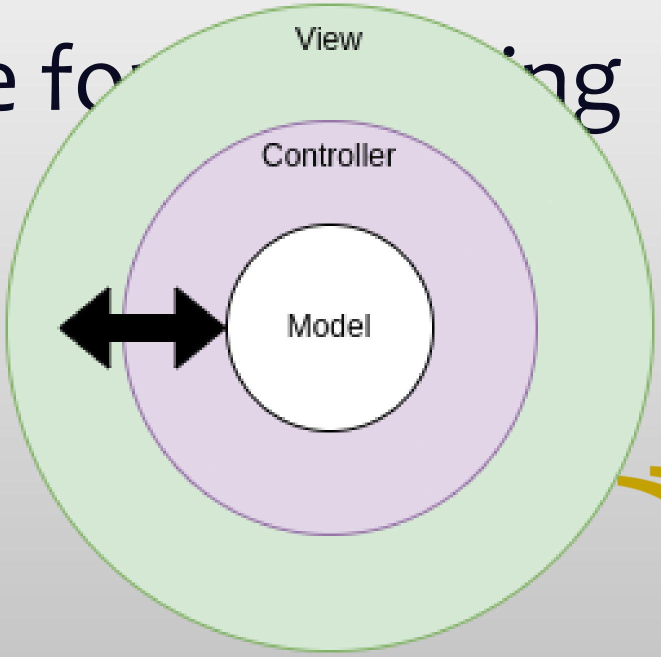
MVC is an architecture pattern that stands for **Model View Controller**.

We could say the MVC architecture is like a simplification of the layers architecture, incorporating the front-end side (UI) of the application as well.



MVC

- The view layer will be responsible for rendering the UI.
- The controller layer will be responsible for defining routes and the logic for each of them.
- The model layer will be responsible fo ing with our database.



Frameworks

There're many frameworks that allow you to implement MVC architecture out of the box (like Django)

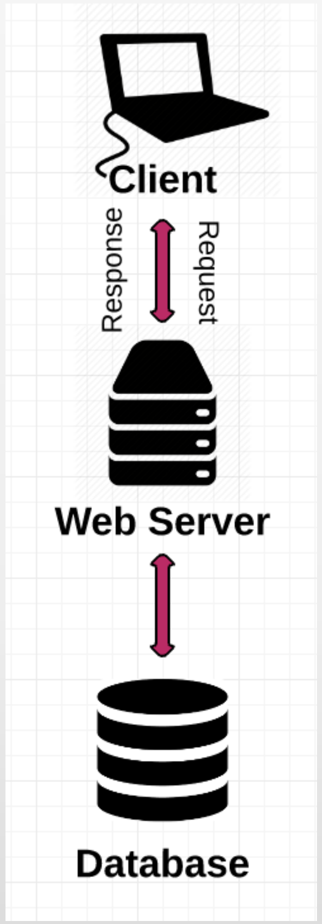
Revising a simple web application

When this application grows,

a single server won’t have the power to handle thousands — if not millions — of concurrent requests from visitors.

In order to scale to meet these high volumes,

one thing we can do is distribute the incoming traffic across a group of back-end servers.



How to Scale a Simple Web Application?

This is where things gets interesting. You have multiple servers, each with its own IP address. So how does the Domain Name Server (DNS) know which instance of your application to send your traffic to?

The simple answer is that it doesn't. The way to manage all these separate instances of your application is through something called a **load balancer (reverse proxy)**.

The load balancer acts as a traffic cop that routes client requests across the servers in the fastest and most efficient manner possible.

How to Scale a Simple Web Application?

Since you can’t broadcast the IP addresses of all your server instances, you create a **Virtual IP address**, which is the address you publicly broadcast to clients.

This **Virtual IP address points to your load balancer.** So, when there’s a DNS lookup for your site, it’ll point to the load balancer. Then the load balancer jumps in to distribute traffic to your various back-end servers in real-time.

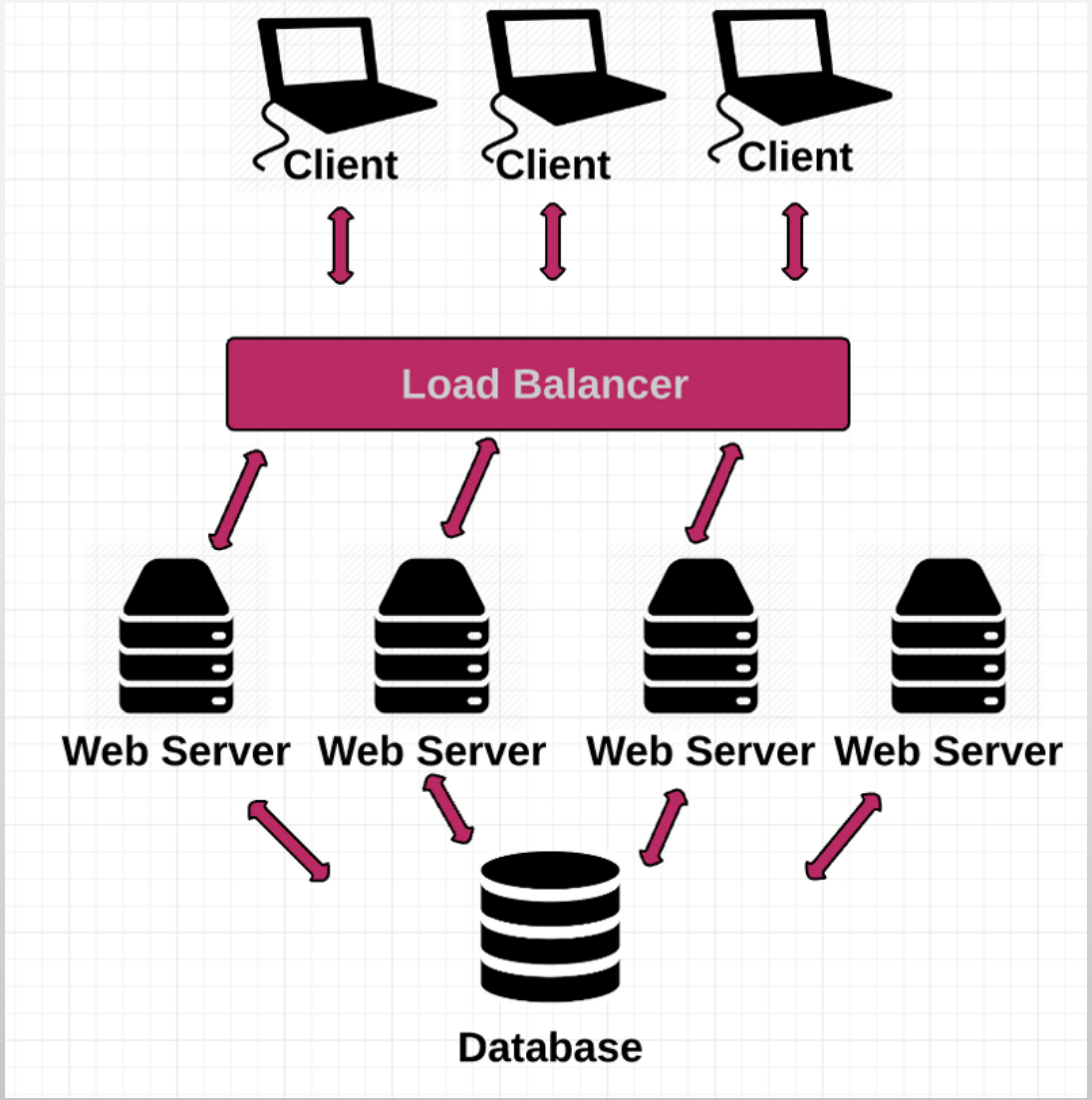
You might be wondering how the load balancer knows which server to send traffic to. **The answer: algorithms**.

One popular algorithm, Round Robin, involves evenly distributing incoming requests across your server farm (all your available servers). You would typically choose this approach if all of your servers have similar processing speed and memory.

With another algorithm, Least Connections, the next request is sent to the server with the least number of active connections.

National College of Ireland

How to Scale a Simple Web Application?

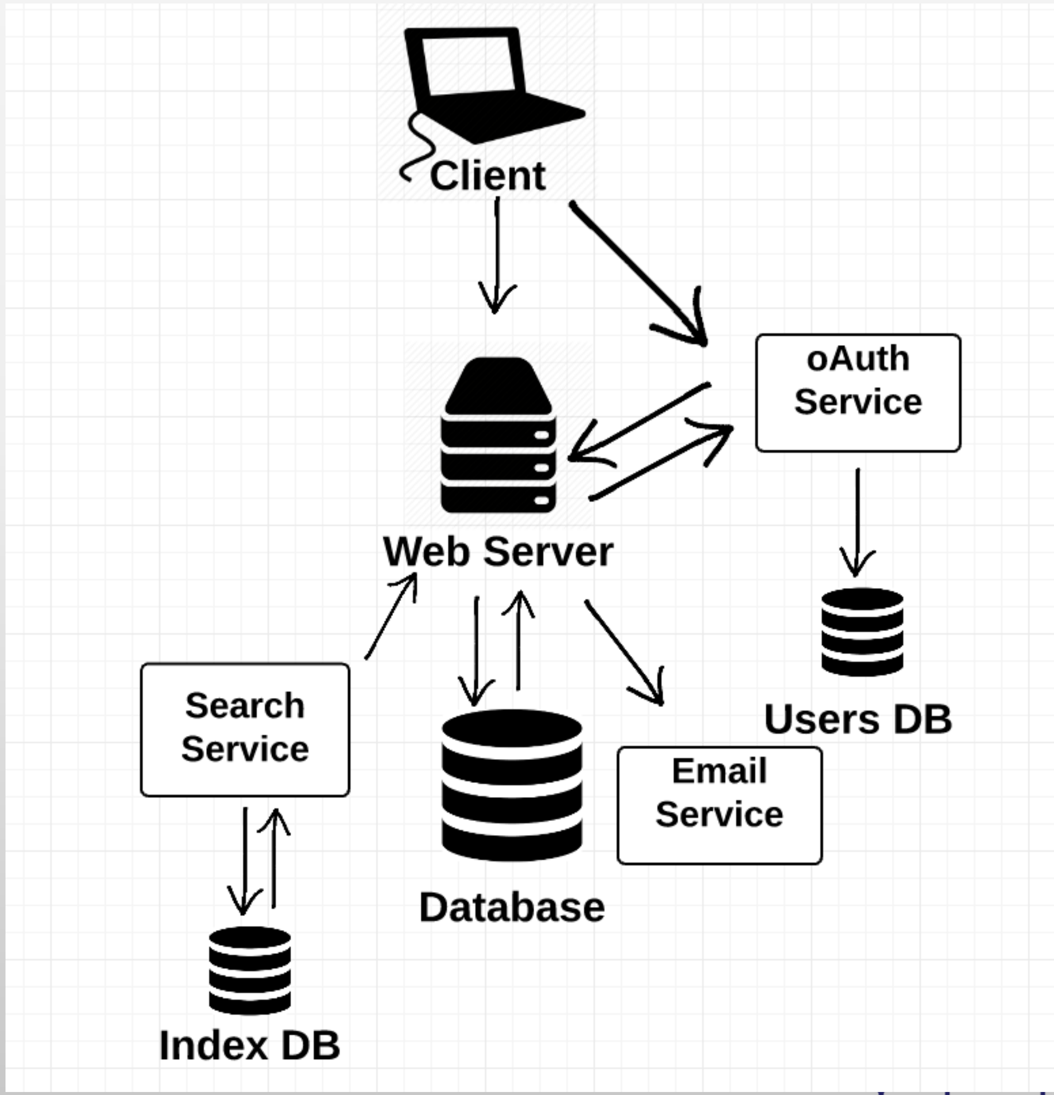


Another solution - Services

Replicating a bunch of servers could still lead to problems as your application grows.

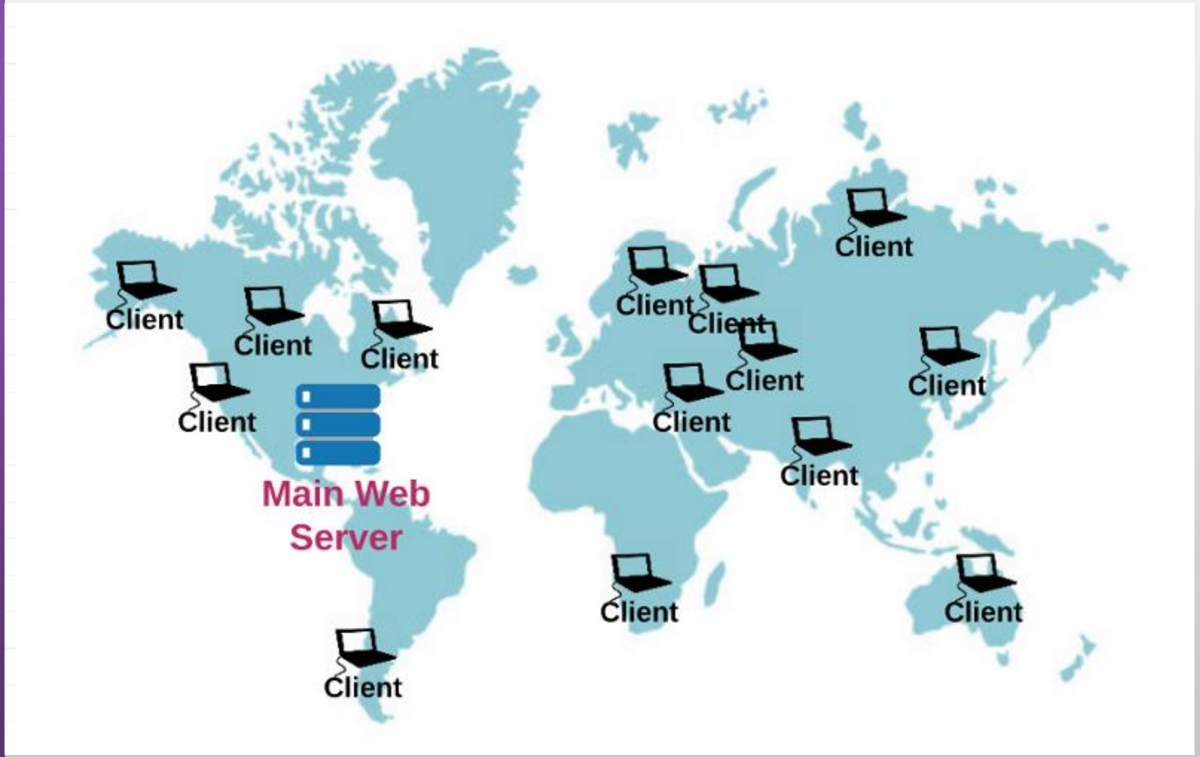
Services allow you to break up your single web server into multiple services that each performs a discrete functionality.

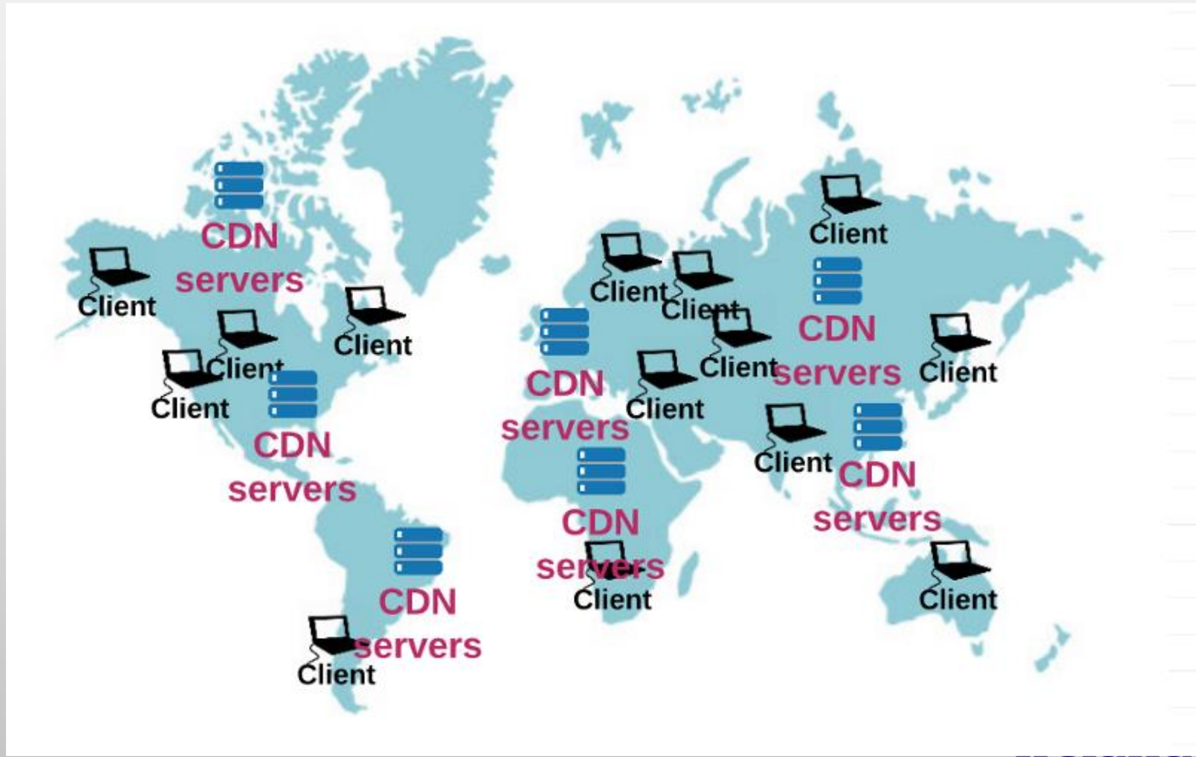
Each service has a self-contained unit of functionality, such as authorizing users or providing a search functionality.



Another Solutions could be - **Content Delivery Networks**

All of the above works great for scaling traffic, but your application is still centralized in one location. A popular tactic to solve this is using a Content Delivery Network (CDN).





Resources

Fundamentals of Software Architecture
by Mark Richards and Neal Ford

Software Architecture Patterns
by Mark Richards

The Software Architecture Handbook
Germán Cocca

https://www.freecodecamp.org/news/how-the-web-works-part-ii-client-server-model-the-structure-of-a-web-application-735b4b6d76e3/

National College of Ireland