N Tier Applications

In this lesson, we will go over the N Tier applications and its components.

WE'LL COVER THE FOLLOWING

- N-Tier Application
- Why The Need For So Many Tiers?
- Single Responsibility Principle
- Separation Of Concerns
- Difference Between Layers & Tiers

N-Tier Application

> An *N-tier* application is an application which has more than three components involved.

What are those components?

- *Cache*
- *Message queues for asynchronous behaviour*
- *Load balancers*
- *Search servers for searching through massive amounts of data*
- *Components involved in processing massive amounts of data*
- *Components running heterogeneous tech commonly known as web services* etc.

All the social applications like *Instagram*, *Facebook*, large scale industry services like *Uber*, *Airbnb*, online massive multiplayer games like *Pokemon Go*, applications with fancy features are *n-tier* applications.

> **Note:** There is another name for n-tier apps, the "**distributed applications**". But, I think it's not safe to use the word "*distributed*" yet, as the term *distributed* brings along a lot of complex stuff with it. It would rather confuse us than help. Though I will discuss the *distributed architecture* in this course, for now, we will just stick with the term **N-tier applications**.

So, why the need for so many tiers?

Why The Need For So Many Tiers?

Two software design principles that are key to explaining this are the *Single Responsibility Principle* & the *Separation of Concerns*.

Single Responsibility Principle

Single Responsibility Principle simply means giving one, just one responsibility to a component & letting it execute it with perfection. Be it saving data, running the application logic or ensuring the delivery of the messages throughout the system.

This approach gives us a lot of flexibility & makes management easier.

For instance, when upgrading a *database* server. Like when installing a new *OS* or a patch, it wouldn't impact the other components of the service running & even if something amiss happens during the OS installation process, just the database component would go down. The application as a whole would still be up & would only impact the features requiring the database.

We can also have dedicated teams & code repositories for every component, thus keeping things cleaner.

Single responsibility principle is a reason, why I was never a fan of *stored procedures*.

Stored procedures enable us to add business logic to the database, which is a big no for me. What if in future we want to plug in a different database?

Where do we take the business logic? To the new database? Or do we try to refactor the application code & squeeze in the stored procedure logic somewhere?

A database should not hold business logic, it should only take care of persisting the data. This is what the *single responsibility principle* is. And this is why we have separate *tiers* for separate components.

Separation Of Concerns

Separation of concerns kind of means the same thing, be concerned about your work only & stop worrying about the rest of the stuff.

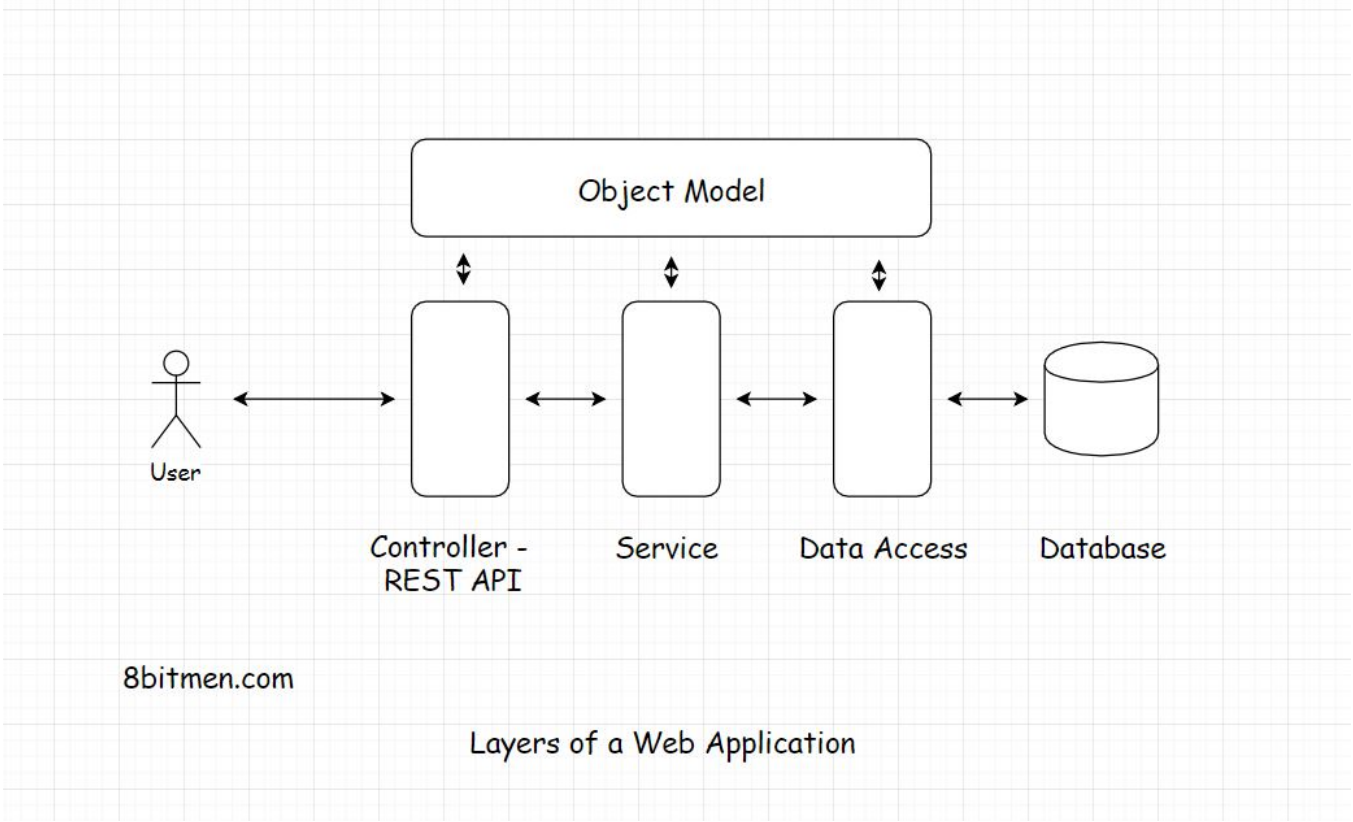
These principles act at all the levels of the service, be it at the tier level or the code level.

Keeping the components separate makes them reusable. Different services can use the same database, the messaging server or any component as long as they are not tightly coupled with each other.

Having loosely coupled components is the way to go. The approach makes scaling the service easy in future when things grow beyond a certain level.

Difference Between Layers & Tiers

> **Note:** Don't confuse tiers with the layers of the application. Some prefer to use them interchangeably. But in the industry layers of an application typically means the *user interface layer*, *business layer*, *service layer*, or the *data access layer*.



The layers mentioned in the illustration are at the code level. The difference between *layers* and *tiers* is that the layers represent the organization of the code and breaking it into components. Whereas, tiers involve physical separation of components.

All these layers together can be used in any tiered application. Be it single, two, three or N-tier. I'll discuss these layers in detail in the course ahead.

Alright, now we have an understanding of tiers. Let's zoom-in one notch & focus on web architecture.