N-Tier Applications

In this lesson, we will go over the n-tier applications and their components.

We'll cover the following

- N-tier applications
- Why the need for so many tiers?
- Single responsibility principle
- Separation Of concerns
- Difference between layers & tiers

N-tier applications#

> An *n-tier* application is an application that has more than three components involved.

What are those components?

- *Cache*
- *Message queues for asynchronous behavior*
- *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*

services etc.

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

> **Note:** There is another name for n-tier apps, "**distributed applications**." But, I don't think it's safe to use the word "*distributed*" yet, as the term *distributed* brings along a lot of complex stuff with it. At this point, it would confuse rather than help us. Although I will discuss *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* and the *separation of concerns*.

Single responsibility principle#

Single responsibility principle simply means giving only one responsibility to a component and letting it execute it perfectly, 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 and makes management easier, like when upgrading a database server.

When installing a new *OS* or a patch, this approach wouldn't impact the other components of the service running. Additionally, 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 and only the features requiring the database would be impacted.

We can also have dedicated teams and code repositories for every component, which keeps things cleaner.

The *single responsibility principle* is then 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 the future, we want to plug in a different database? Where do we take the business logic? Do we take it to the new database? Or do we try to refactor the application code and 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 and stop worrying about the rest of the stuff.

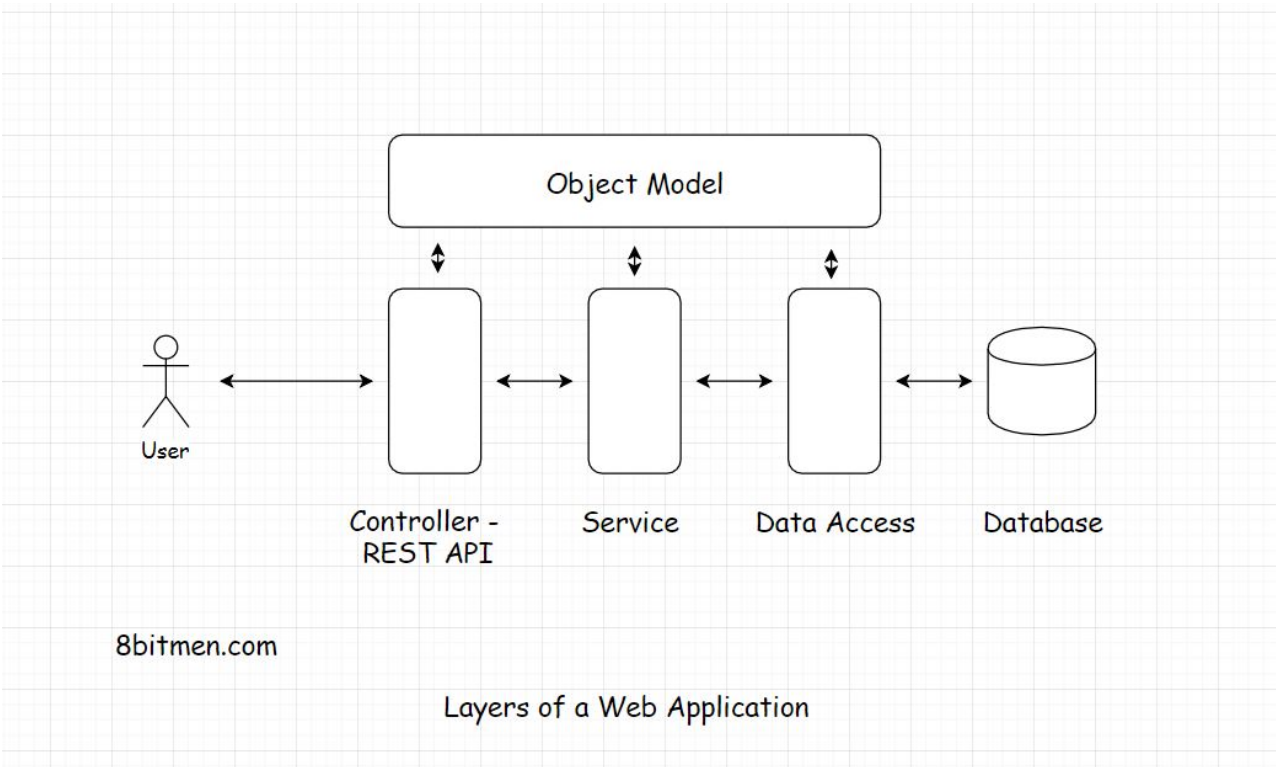
These principles act at all the levels of the service, both at the tier level and the code level.

Keeping the components separate makes them reusable. Different services can use the same database, 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 the 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. However, in the industry, layers of an application typically means the *user interface layer*, *business layer*, *service layer*, or the *data access layer*.



Layers of a Web Application

The layers mentioned in the illustration are at the code level. The difference between *layers* and *tiers* is that layers represent the conceptual organization of the code and its components, whereas, tiers represent the 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 and focus on web architecture.

← Back

Three-Tier Applications

Next →

Different Tiers in Software Architectur...

Completed

Report an Issue