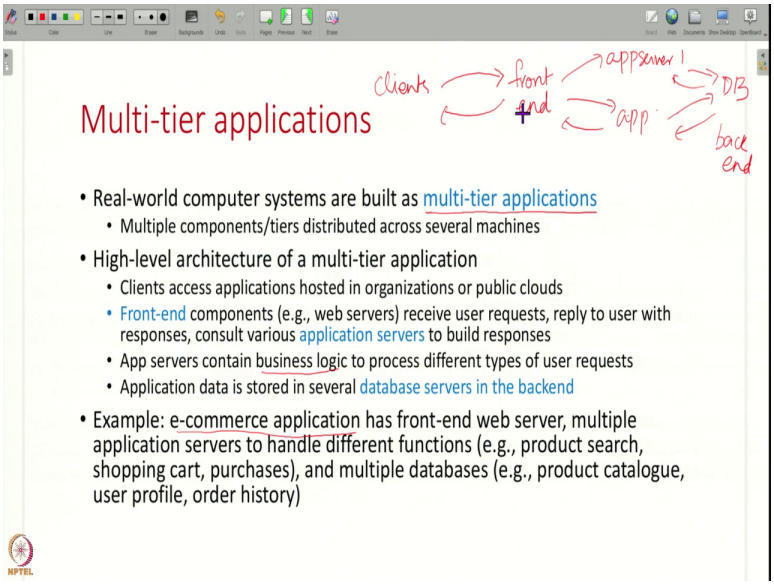
Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture 40
Multi-Tier Application Design

Hello everyone, welcome to the 28th lecture in the course Design and Engineering of Computer Systems. So, in the previous lectures, we have finally seen all the building blocks needed to design computer systems and, in this lecture, we will begin to understand how to do end to end application design. So, let us get started.

(Refer Slide Time: 00:40)



So, real world systems are built as what are called multi-tier applications that is you have multiple tiers or components that are distributed across several machines, you do not have everything built as just one big monolithic component on one system, but it is distributed across different machines.

So, for example, you might have clients, users of a computer system access the computer system that is hosted either within the organization or on some public cloud anywhere the system can be hosted. And then the first component that clients interact with are what are called front end components, like web servers or something that received the user's request.

Then these web servers may not know how to process all types of requests. So, all the logic, all the business logic of an application, you will not write in just this web server itself. Instead, there will be multiple different application servers, there need not be just one there could be many application servers, each of which can handle a certain type of request a certain functionality.

And these front-end servers will talk to these different application servers, get back the responses and send it back to the client. And these application servers also may contact, there could be other databases, which actually store application data. So, all of these app servers might be talking to databases to store and retrieve data.

So, these databases are typically called the backend. So, in this way, real computer systems have multiple tiers or multiple layers, you have a front end, then you have the application servers, where all the actual business logic resides, then you have database servers in the backend, of course, there are many different variations on the simple characterization.

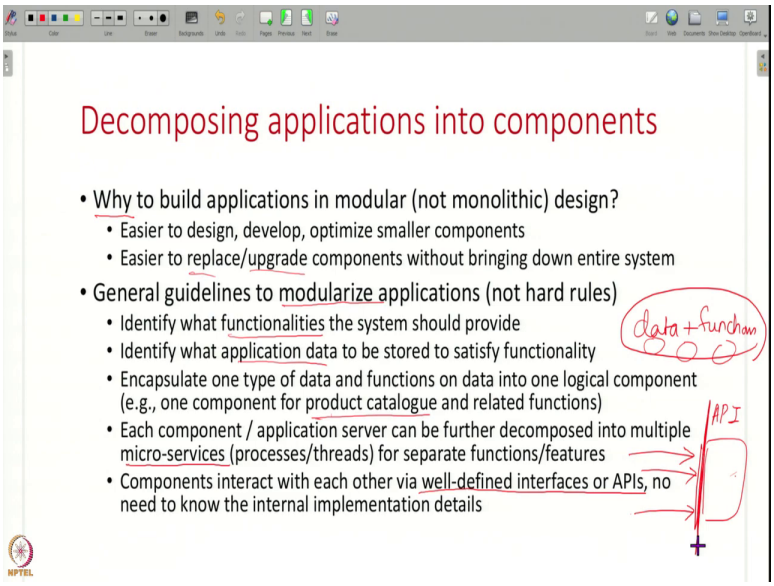
But this is at a high level, how real systems look like. So, for example, if you consider an e-commerce application, something that we have been referring to many times in this course, then you might have a front end, a web server that displays the web page that you search for products that you purchase, all of that could be the front end.

Then there could be different application servers that are managing different functionality. For example, you could, there could be one server to manage all the products, one server to manage your shopping cart, one server to do purchases, for billing, all of that all of these different functionality could be handled by different servers.

And then you could have multiple databases, one database to store the product information, one database to store the users order history or profile. So, you have different front ends, different application servers, different databases, all of which are working together to produce some functionality of say e-commerce to the user.

So, in the beginning we have seen what is a computer system, it is a set of computers that are providing some common functionality to the user and this computer system. Now, I hope you can visualize the internals of it, it has multiple different components that are together doing some common work for the user.

(Refer Slide Time: 04:12)



So, the next question comes up, how do you decompose these applications? How do you build a system composed of these multiple components? How do you design it? So, before we answer that question, let us answer the why. Why do you need to build an application in a modular fashion? By now I think the answer should be clear to you.

It is in general easier to design, develop, optimize, if you have many smaller components, each component can be built independently can be tested independently can be optimized for performance, reliability, independently and then you can put all of these together that is easier than just writing million lines of code in one program. So, all of us understand the benefits of modularity. The other reason is, it is also easier to maintain a system. As your system runs for many days, maybe some components need some replacement, upgrading something fails.

All of this is easier if your system has multiple different components instead of everything in one big file. So, therefore, modularity is critical for large systems. Now, the question comes up, how do you modularize an application? I have this large system that I want to build, how do I know which are the components? How do I split it into modules?

So, I will in this lecture, I will try to provide some very general guidelines of course, there are many different ways of doing it, these are not hard and fast rules, but at a high level, very simple ideas you can use to modularize applications, I will try and describe now. So, the first thing that you have to do is identify what are the functionalities that your system should provide. List down

all the different things that you want your system to do. And to satisfy all of these functionalities, you have to maintain some data, like if you want to, if you are an e-commerce website, you might have to maintain information about products, about users about orders, all of that you have to maintain.

Then the next thing you do is identify one type of data plus all the functions that run on that data package them together and make that into one logical component. For example, the product catalogue all the list of products in an e-commerce website and all the functions on that product like adding products, buying products, returning products, whatever it is, all of those functions on this data are implemented together in one component.

In this way, you can split your functionality into multiple components. And of course, again within a component also a component can be providing multiple functions. For example, the component managing your product catalogue could be providing many functions like adding products to it, buying products, deleting products, all of that, there could be multiple functions.

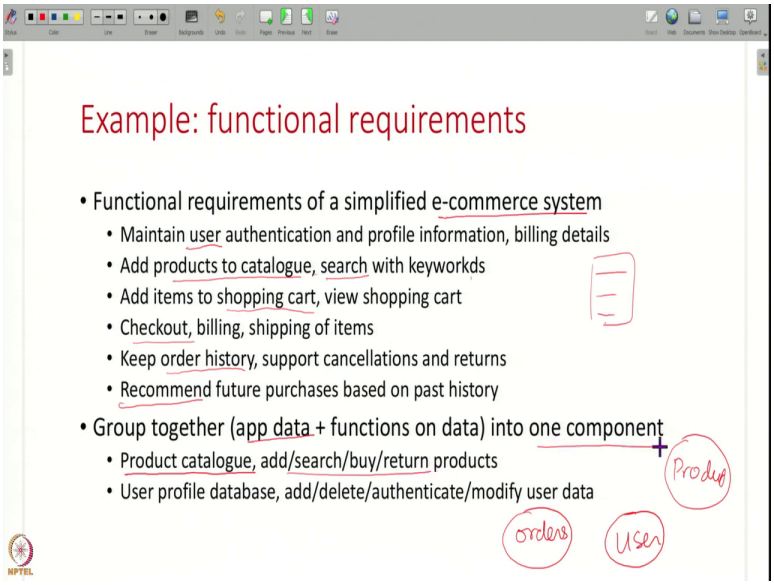
And these different functions can further be decomposed into smaller components, that is usually called micro services, each bigger logical component can have smaller micro services, providing different functionality. These micro services can be processes or threads or container something. You have a bigger component in that component a functionality to add new products, to buy products, to return products to search for products, all of these different functions can be implemented as separate sub-components or micro services within a bigger component.

And these micro services can be processes threads, whatever. So, in this way, you can hierarchically design an application. Start with the bigger functionality, split it into logical components, and each component you can split into sub components. And all of these components need to interact with each other via well defined interfaces or APIs. So, the purpose of modularity is you should not know how a component is implemented.

So, every component will just expose a certain interface and everybody else should be able to access a component using this interface or this API. For example, the component that manages your product catalogue can expose an API to add, buy, return, search products, whatever it can expose an API, it can expose different functions and other components should not go and access this product catalogue directly or mess with the code of this component in any way, you should

be able to access the functionality of the component using only this API. So, that is how you design an application, identify the logical components, identify the sub-components, and then identify the interfaces between all of these.

(Refer Slide Time: 09:06)



So, let us take a simple example of an E commerce system to actually run through this entire process of application design, how you go about modularizing a computer system? So, let us start with the functional requirements of an e-commerce application. So, suppose you want to build a simple e-commerce system, of course, real life systems have a lot of complexity, I will just consider what is the bare minimum requirements that you need to build an e-commerce system.

First, you need to have some way for users to create accounts, authenticate, log in profile information, billing information, all of that has to be managed. Then every E commerce system will have like, some bunch of products in a catalogue and you want to let suppliers or vendors add products to this catalogue then you want users to be able to search for the available products using some keywords.

And then users should be able to add whatever products that they want to buy into a shopping cart. And then check out the shopping cart pay the build for all the items do some online payment for all the items in your shopping cart and provide some shipping address and have

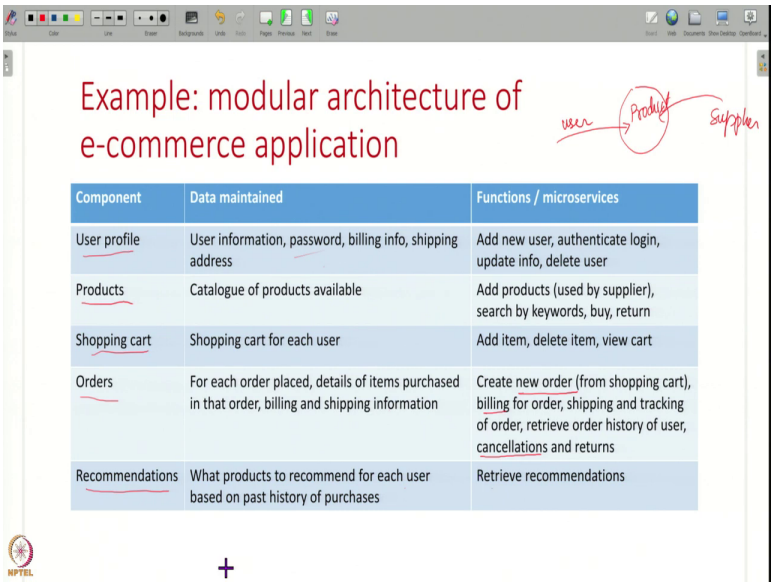
there are these products in your cart, you bill them and you ship them. All of this functionality the users want to do.

And of course, once you have placed an order, you might want to keep your history of your orders because in the future when you get a product, you may not like it, you may want to return it, all of this you have to support. And of course, the other thing that a lot of systems today do is they recommend they say oh if you have purchased these products, maybe you might want to consider looking at this product also. For their own revenue stream in the future.

So, such things also you may want to support. So, this is all the functional requirements that an e-commerce system, a very simple e-commerce system may have to support. Now, let us see how we can start modularizing the system. We should group together some data and the functions of the data. For example, you can say I will have one component for managing my product catalogue and all the functions to add products from the supplier, search, buy, return all of these functions will be handled by the product component.

Then you can say I will have a user profile data all of that is managed in another component adding users, authenticating users, all of that is taken care of by another component. Similarly, another component can be there for managing all the orders, whenever somebody purchases, the building shipping storing the order history all of that can be another component. In this way you identify what is your data, and what are the functions on that data and group them together into one logical component.

(Refer Slide Time: 11:46)



So, in this slide, I have shown you one example of how you can modularize an e-commerce application. Note that this is not the only way of course real-life systems are more complex and of course, there are many different ways of modularizing. But this is just one example that makes sense.

For example, you can decide to have components for the user profile, for the products, for the shopping cart, for orders and for recommendations. So, different components are managing different functionality. So, this user profile component might use a database to store user information, password, the your billing address, shipping address, credit card details, this component can maintain all of this data and provide functionality like creating a new user account, authenticating a user, updating user info and so on.

Then another component can manage all information related to products, it can keep a catalogue of products and provide various functions to other components or to users, like searching for products, adding products, supplier can here is your product component, the server that is maintaining product information, the supplier can talk to this component and say add some product, then the user can search for products, can buy products and so on.

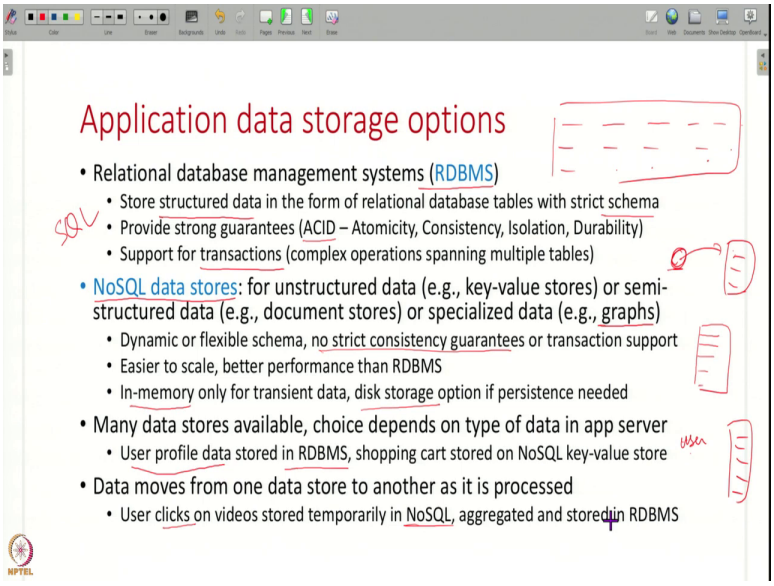
So, this component is managing all the data pertaining to products. You can have another component that is managing the shopping cart for each user. The every user's shopping cart, adding items, deleting items, viewing the shopping cart and so on. You can have another

component that is actually doing the ordering the processing of an order. It maintains these details of in every order, what are all the items purchased, all of that information is maintained. And this component will do things like when you want to buy something, check out your cart, it will create a new order, it will do the billing of that order, shipping, tracking.

In the future, if the user wants to cancel an order, do some returns, all of this order related, purchase related information can be managed by one component. Of course, you can have different kinds of modularity. You can say, one component does the billing, somebody else does the shipping, you can also split this into even finer granularity also I am not saying this is the only way, but this is one example of modularity.

And finally, you can have another component that is whose only job is to do recommendations, look at all the orders that have happened in the past based on what the user has purchased, what other users have purchased, you can recommend products for the user to buy in the future. And this can store all the recommendations and anybody can look up these recommendations. So, in this way, you can take these complex functional requirements, many different requirements of a computer system and I had split them into components each component is managing some data and doing some operations on that data that is how you can think of a computer system.

(Refer Slide Time: 15:07)



So next question comes up, how do you store these different kinds of data in a component? So, every component is managing one or more kinds of data, then how do you store them? There are

many different options here. For example, you have things like relational databases, relational database management systems. If you have taken a database course, you would have heard about these. These are databases that store that store data in the form of tables, you can create a table, every table will have a schema, different columns, a fixed format you can create tables and you can store structured data in these tables. That is the data you are storing has a certain format. Every row has a fixed number of columns, and so on.

And these databases also provide strong guarantees. When you store the data, there are certain properties called ACID properties, like atomicity, consistency, isolation, durability, and these databases also provide transaction support that is, you can do complex operations that span multiple tables, and they will ensure that all of these operations are done very consistently and correctly.

So, if you take a database course, you will study about all of these in more detail about how databases work, how they store data in tables, how they support querying of the data, how they provide all these ACID properties, transactions, all of that, you will study in a database course, we will not cover that here.

But you should simply know that if real life systems want to store such structure data with all of these consistency guarantees, they can use existing database systems. But of course, you might not always want to store such structured data, sometimes you might want to store data that does not have so much structure. For that you have different types of data stores, which are called no SQL data stores.

This is the common name because databases usually work with SQL is the query language to query all the rows and extract some rows and columns in a database. So, no SQL data stores work without any structure, they can work with unstructured data. For example, you might just have a key value store, there is some key, and there is some value corresponding to that key.

It can be anything, I do not know, what is the structure of this value, how many columns it has, is it a table, I do not care, this is called unstructured data given a key I will give you some blob of data. That no SQL data stores can store such data or you might have semi structured data, you might have some structure in your data, but not a whole lot.

Like for example, you might be storing a document. It might have some structure like author name, date or something, but that is all not very rigid structure, it can have some columns, but it can also not have some columns sometimes or you can have data stores for to store very special types of data like you want to store a graph, you want to store something else that has specific properties.

So, for all such use cases, regular traditional databases are not useful therefore you have many new no SQL systems coming up today. These systems, they are very dynamic, flexible, and they do not provide any strict consistency guarantees, but they are higher performance they give better performance and they are easier to scale to larger capacities than regular databases.

So, you when you design a system, you always have this design choice between using traditional databases that have fixed format, fixed schema with very good consistency and other guarantees or no SQL data stores that give better performance but lesser structure and lesser guarantees. Of course, this is a spectrum there are many choices in between, but many data stores are available again with respect some datastore stores only in memory, but some store on disk for persistence.

If you store only in memory, memory accesses fast you will get back responses you can read and write faster if you store in disk it will take longer, but with disk you will get persistence. So, there are many data choices available. So, if you take a database course this will become clearer as to how you design these databases and how you design this no SQL data stores.

But for the purpose of this course it is enough to know that many different options are available. And when you are designing a computer system based on your requirements, you will pick a data store. For example, if you have to store user profile data, this you have to store it securely it has a fixed format every user will have name, email, phone number, billing address, shipping address there is a fixed format to the data such data you can store in our traditional relational database.

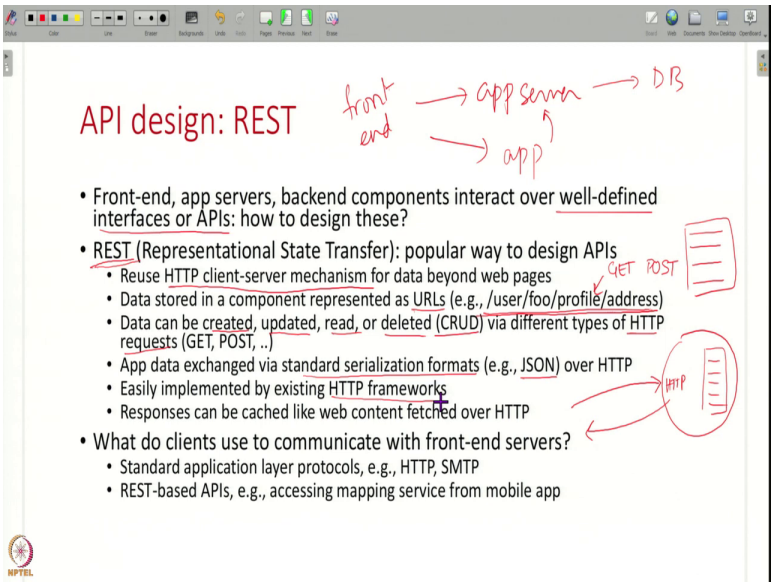
But if you want to store a shopping cart, you can store in a no SQL key value store, user ID and a list of all the items in that shopping cart can be stored as a key and a value in a key value store. Why because with shopping carts, you do not need lot of consistency, it is okay if the values are not always consistent, it is if some failures happen and one or two items goes missing. We will study this later when we study reliability and performance. We will see why it is easier to get

good performance when there is lesser structure in your data, when there are lesser consistency guarantees to be provided. Therefore, you will get better performance that is always a trade-off.

You want more consistency, you have to trade-off some performance for it. So, for things like shopping cart, it is not super important information, it is okay if it a little bit here and there happen. So, such data can be stored in a no SQL key value store. So, therefore, depending on your application requirements, each component can make the choice of which data store to use.

And also, it is also possible that the data will move from one store to the other. Initially, for example, on some video streaming website, whatever the user clicks, the user clicks on a video, they can be temporarily stored in a no SQL data store and later on this information can be aggregated and stored in a database. So, you have many different options and you can also move data from one kind of data store to the other. So, now that we have seen how to make components, how to store the data in components.

(Refer Slide Time: 21:42)



Next, we will understand how to design the interfaces between components. So, you have your front end, and you have your various application servers, there will be many servers for many different components. And each server will again have multiple databases in the backend. So, all of these components have to interact with each other, your front end has to contact the application servers. And these application servers also may have to talk to each other, to exchange some information.

So, for all of these purposes, every component needs to have a well-defined interface or API, using which you will access the functionality provided by the component. So, the question comes up, how do you design these APIs? So, there are many ways, one popular ways to design API's in the form of REST API's. So, REST stands for Representational State Transfer, I will explain what this means. So, this is a very fancy term, but the meaning is very simple. What it means is simply use HTTP client server mechanism for providing API. So, every component, it has multiple pieces of data that it is maintaining.

So, for every piece of data assigned some URL. For example, a component that is storing user data, you create a URL, user, foo, profile, the address, credit card, name, email, whatever, you create a URL like this for every piece of data that this component is managing. And then now once this URL is created, we can use HTTP to access the data. The component can handle various types of HTTP requests on the data.

For example, you can say, GET this URL, then you will be able to access the profile, some address of the user. If you say POST HTTP request, you can do many different things, you can GET, you can POST, you can do things, you can create, update, read, delete, you can do what are called CRUD operations on data using HTTP requests.

Different types of HTTP requests can be used to create, update, read, delete your data that is available. So, then now what is the component doing? It is simply handling HTTP requests, the other components or front end or users are sending HTTP requests to create, read, update, delete the data and this component is responding back with HTTP responses.

And the data all the data inside the component is represented by URLs. And of course, HTTP responses and requests need not just be text, you can also send complicated data structures we have seen, there are standard serialization formats like JSON available using which the component can send complex data also not just text.

So, every component, whatever data it has, it is storing it in the form of URLs and HTTP requests and responses are being processed in order to manipulate that data. So, this is one way of designing interfaces between components. And the advantages of the REST interface is that there are many frameworks available already to send and receive HTTP data. These frameworks

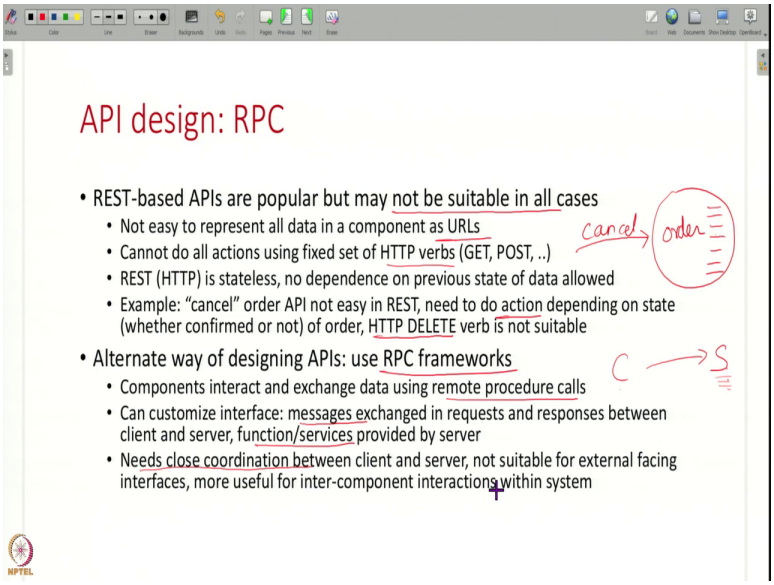
were available created for web pages, but we can re-use those same frameworks and instead of sending, web pages, you can send any information about your component.

You can send user data, or the client can update user data by doing an HTTP POST request all of these can be done just that now you are not exchanging web pages, but you are exchanging specific items of data stored in that component. And these responses can also be cached, just like web pages. For all of these reasons, it is easy to develop API's in the REST format.

And of course, these REST API's can be used between components, and also can be used by clients. So, when clients have to communicate with front end servers, they can use standard protocols, or they can use REST based APIs. You can either access a web page, you can either access a system using a web page, using HTTP or you can also, from your app, the your app can be just sending these REST based API messages to access information.

For example, a mapping service, you can be fetching information using, you are not going to a webpage, but in your app, you are just using these REST based API's to access data. So, REST based APIs are very powerful as long as you can represent your data in your application in the form of these URLs and you can manipulate these data using these HTTP, GET, POST all of these requests, then you can use a REST API to interact with other components in the system.

(Refer Slide Time: 26:40)



But sometimes, rest API's may not be suitable in all cases. For example, you may not be able to represent all the data in your component as just URLs you know your data can be very complex,

and I cannot just create like a URL hierarchy to represent all of it. And it can so happen that you cannot do all actions using only HTTP verbs.

Because HTTP is fundamentally stateless, there is no dependence on the previous state of the data allowed and all of that. So, I will make this clear with an example. Suppose, there is a component that is processing all the orders. And some user wants to cancel an order. So, this component has all the list of all the orders, and you want to cancel an order. And you want to cancel an order only if the order has not been confirmed so far.

So, this is a complicated action. How do you represent it using HTTP? It is very hard. So, HTTP, this is not creating any information. This is not reading any information. This is not deleting information, I cannot use HTTP delete request. Why? Because I do not want to delete the order. I want my order to stay in the database, the fact that I have cancelled the order, I want that information to remain.

But I just want to cancel the order. I do not want the order to be shipped to me. So, this is a very complicated action, it does not map to just putting some information or reading information. This is a very complicated action. So, for such actions, it is very hard to design a REST based API, you cannot, put this action as some URL and just do HTTP fetch, or POST or delete, you cannot do it easily like that.

Therefore, sometimes for some components for some APIs a REST based API is not suitable. In such cases, what we can do is we can use RPC. Components can interact using remote procedure calls. So, this server can expose a set of functions that the other components can remotely invoke. You can remotely execute functions on another component.

So, we have seen how RPC works before. So, these two endpoints, the client and server that are communicating over RPC, they will define their interface, your interface is no longer now limited to HTTP, GET, POST, HTTP request response. No, your interface can be anything you can exchange whatever message you want, you can provide whatever functions you want.

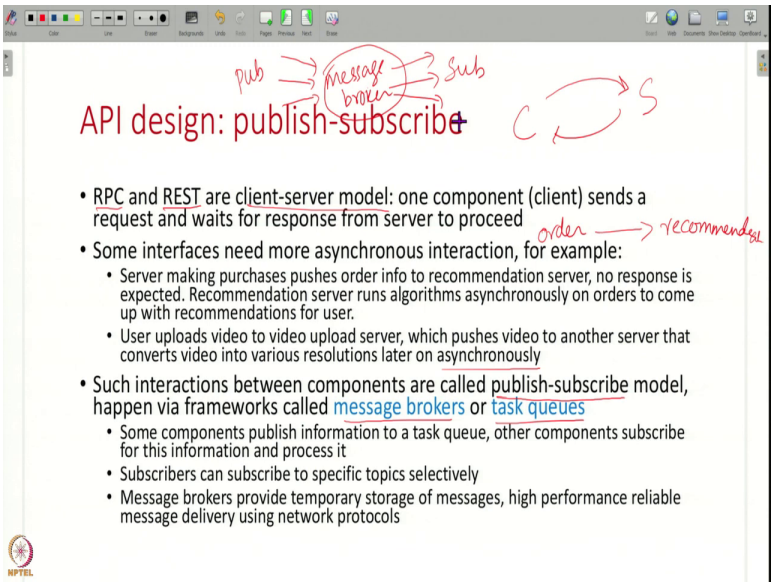
But you will define all of this and both endpoints will agree on this common interface definition. And then this component will invoke the functions on the other side. Clients will invoke functions at the server end. So, not that client and server is simply just indicating who provides this functionality and who invokes the functionality.

So, the advantage here is that you can do complicated actions, you can simply have a function for cancel and invoke that cancel, you do not have to think about, how do I do this cancel using HTTP messages? Which HTTP request, which HTTP response? You do not have to think of all of that. You can define whatever functions you want.

But that also means that you need a close coordination between your different components. You need to know what functions that the other component implements and invoke them using an RPC framework. Therefore, these RPC API's are more useful when interacting between components of a system.

If you have external facing interfaces, you have to talk to clients outside, it is very hard to do RPC because then the other side has to know what are all the functions, whereas HTTP anybody you can send an HTTP request, it is a standard format. Anybody can send you an HTTP request, anybody can use the REST API. But with RPC a more closer coordination is needed. So, this is a design choice that you have to make, whether you want to go with REST based API's or RPC based API s.

(Refer Slide Time: 30:49)



There is also a third type, which is called a publish subscribe or a Pub Sub API. So, in REST and RPC, there is a client server model. That is there is one component that needs some information from another component and it is getting it from that component. Whether using standard HTTP request response or custom request response via RPC.

But sometimes some interfaces, they are not in this client server model, it is not like you are asking something and getting some information back, or you are posting some information and getting a confirmation back. Sometimes you just want to give some work to the other guy and forget about it.

For example, you might have a server that is handling all the order information can just push this information to another server that takes care of the recommendations. So, this order server whenever an order comes in, it will just give this information to the recommendation server and there is no response needed, this guy is not waiting for any information from the recommendation server. The recommendation server will later on asynchronously process this information and build its recommendation for the user in the future.

Similarly, when the user uploads a video to one server, the server can just give this video to another server that will convert it into various other resolutions, and this will happen asynchronously in some sense, the first component is not waiting for a response from the other component.

So, such interactions are called the publish subscribe model. So, one or more components are just publishing some information to what is called a message broker or a task queue there are many names for such entities that you can just push information to this message broker and other people can just read information from this message broker or a task queue.

So, this is not a very close interaction client server model, but rather it is an asynchronous Pub Sub model. We have seen examples, we will see more examples later on also. So, with these, so these are special programs, special frameworks called message brokers or task queues, where some components can just publish information, other components can read this information and process it later on.

And the subscribers so these components that push information are called the publishers and these components that read the information or call subscribers and these subscribers can subscribe to all the information coming in a message broker or they can subscribe to specific topics only.

For example, in your recommendation server, there could be one server that is only listening for orders on books and only doing recommendations on books, another server is only doing

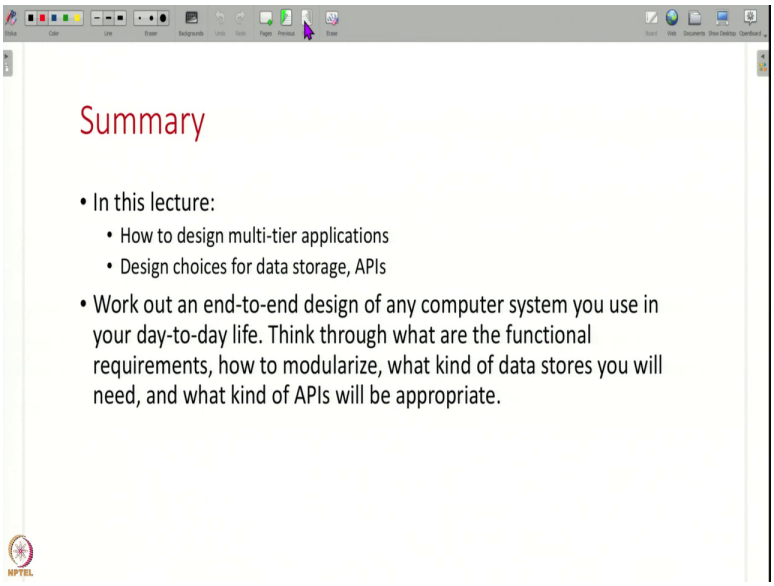
recommendations on electronic items in this way you can just get information only about specific orders or specific topics. And these message brokers what they will do is whenever one component publishes message, they will temporarily store the message until other subscriber components subscribe or receive retrieve that message later on.

And of course, these message brokers will also provide some high-performance reliable message delivery. Many publishers should be able to push messages quickly, many subscribers should be able to read messages quickly. And these publishers and the broker, the broker and the subscribers will again be using maybe something like sockets, TCP to reliably exchange messages, all of that will be there.

So, what you are doing is? You basically adding an intermediary between two components instead of two components directly interacting with each other in a client server manner, you are putting an intermediary so that one component can just push information the other can read the information later on.

So, where the interaction between components does not have to be synchronous and it is asynchronous, these Pub Sub kind of API's can be used. And there are many frameworks available today to help you build these Pub Sub API's. There are many message brokers or task queue software available to you, where you using which you can have this kind of an asynchronous interaction between components.

(Refer Slide Time: 35:44)



So, that is all I have for this lecture. In this lecture, I have given you some brief guidelines and an introduction of how you do end to end system design. So, we have considered the example of a simple e-commerce system and we have seen, how is this complicated system built as smaller components, how does each component store data, how do you design the API's interfaces between these components. So, all of this, we have seen the very basic level of information about system design.

So, I request you all to think about systems yourself and try to think about what are the requirements, how do you modularize, what kind of data stores, what kind of API's, all of these things that we have discussed in today's lecture, I request you to pick another example other than e-commerce and do it on your own. So, in the next lecture, we will also run through more examples, what we will do is we will apply all of these concepts and do an end to end design of many different kinds of computer systems in the next lecture. So, that is all I have for now. Thank you.