Lecture 4 - Software Architecture

Basics of how you might organize your code, how different parts of the system interact with each other, and the myriad of decisions you might make in each part of the system.
Examples:

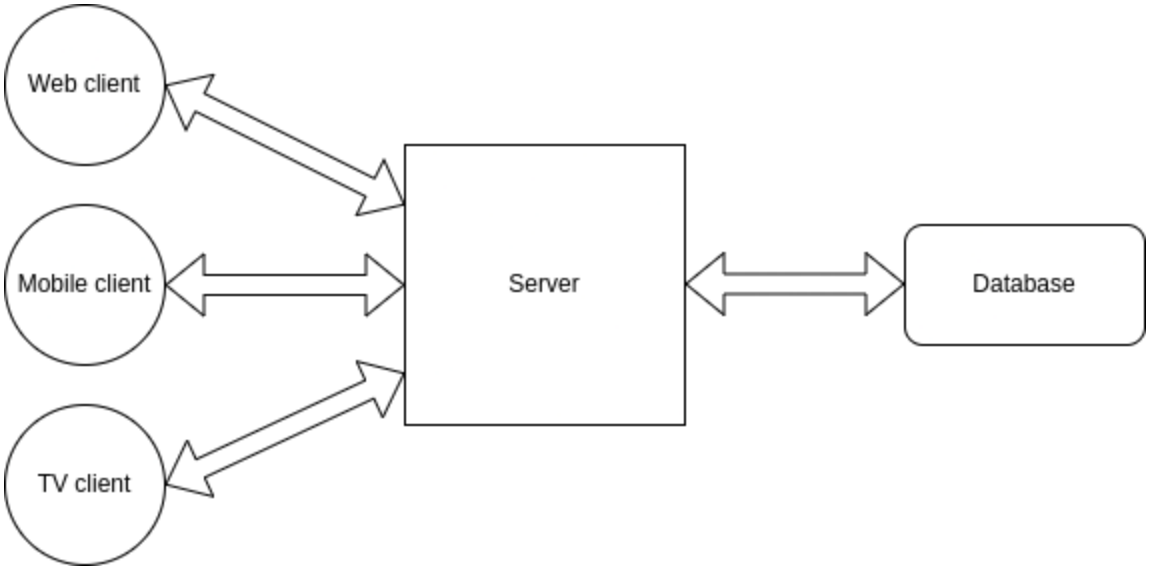
- Which technology do you use? REST or GraphQL for API? Django or Express framework for backend? React or Angular for FE?
- Are your system a monolith or divided into microservices?
- Do you use cloud provider, do you use managed service or do you roll your own services on servers?

Client-server model / what is an API

- Most of you are building web services that communicates over an internet from client (web browser, mobile devices, embedded devices) to servers (typically more powerful computers hosted in a data center).
- Alternative model might include peer to peer, RPC etc
- API stands for Application Programming Interface, think of it like a contract between different parties using the service. For example, the interface might say “if you send a request using this format, I will always respond with data in y format”.
- APIs can be used for both client-server communication, or backend communication between your service and an external service. An example might be if you are integrating payment into your service, and you are using Stripe’s API to accept payment.

Monolithic

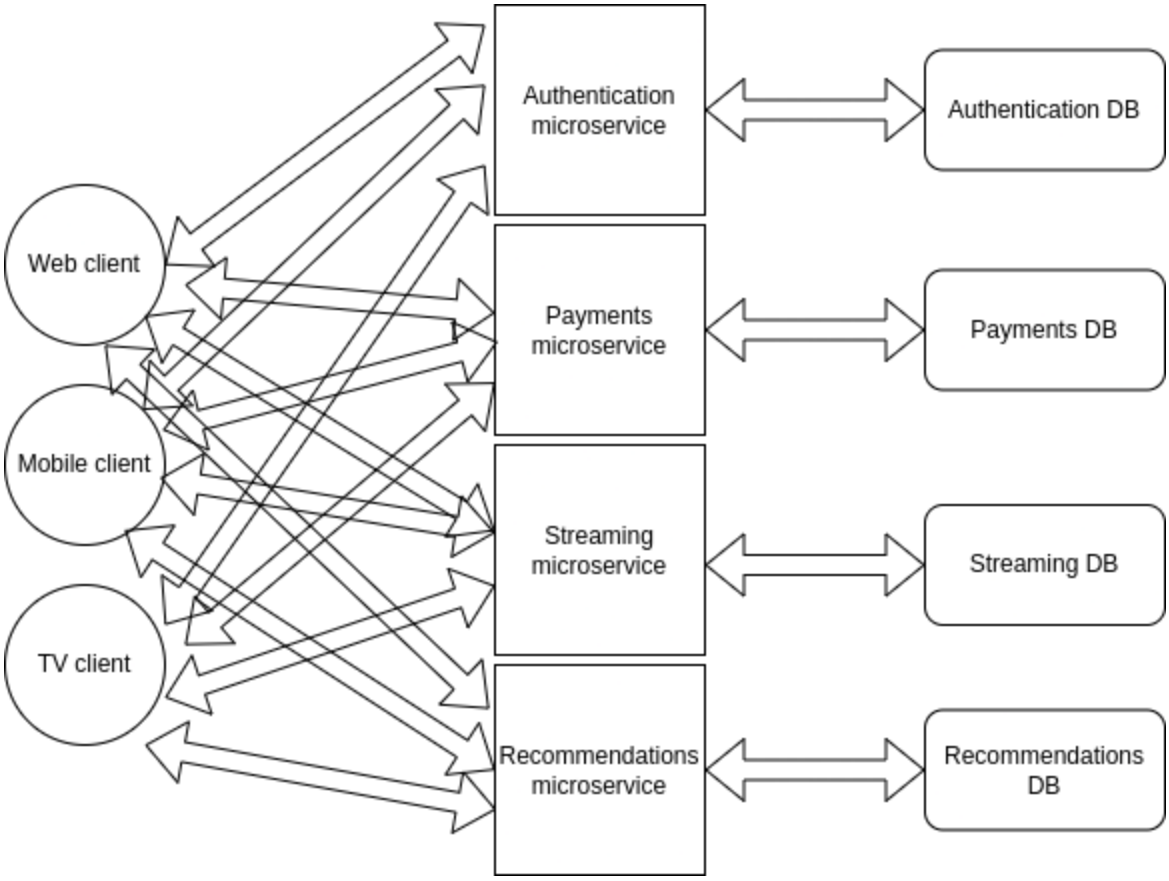
- Let’s use an example. Imagine you are designing a simplified version of Netflix. There are multiple clients with vastly different capabilities and requirements. You may want to allow users to watch TV on their computer in a browser, or on their phone using a mobile app, or on their TV using the built in TV app. On the backend, you might need to talk to a database that handles authentication, recommendation engine, and looking up movies database. You might also want to talk to a blob store and CDN in order to grab the actual media files in order to stream it to users



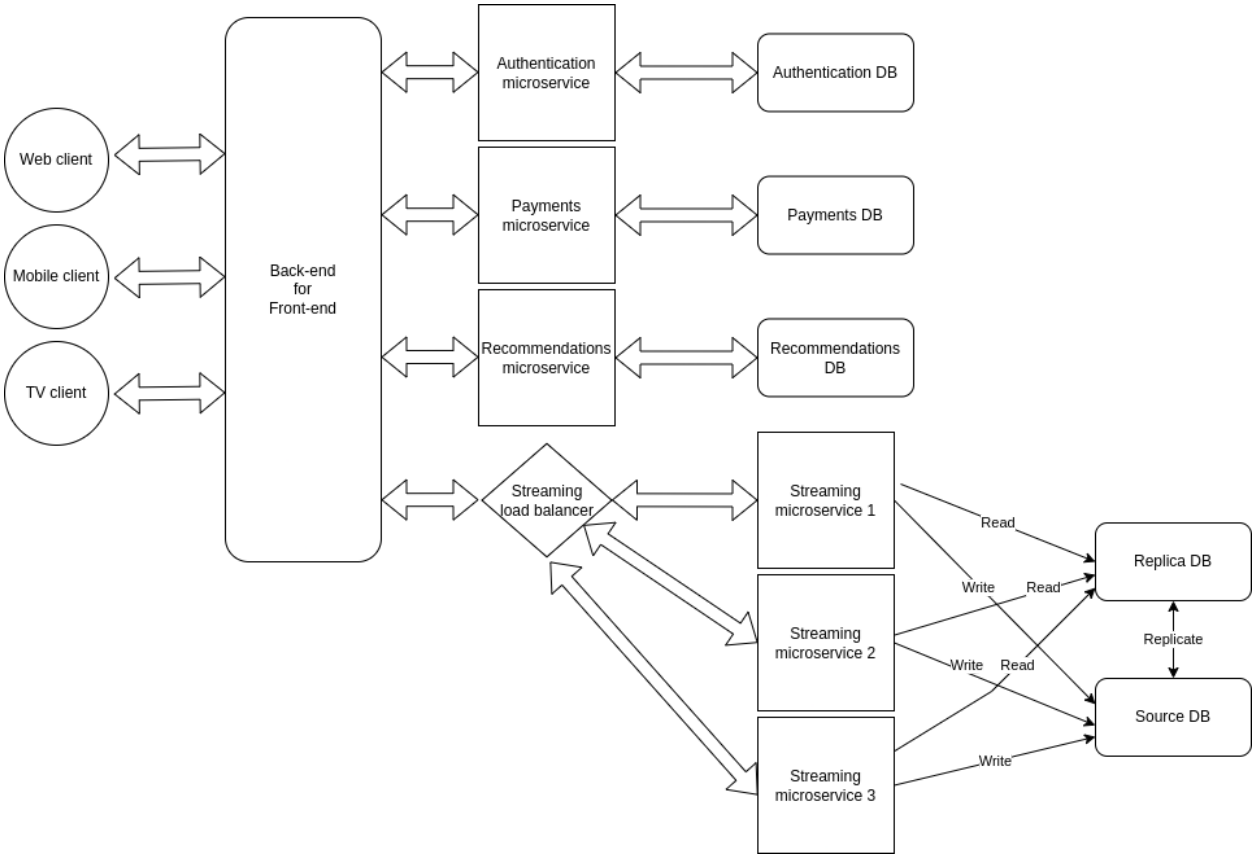
- Here the **Server** component is the monolith. It's a single codebase and service that is responsible for servicing the entire application. Regardless of if you are calling an API to authenticate the user, or getting movie recommendation, they are all being sent to the same server application
- This is a great starting point, and a path I recommend for everyone to follow

Microservices

- Okay if monolith is so great, why do we need microservices, and why are all the huge companies like Uber and Amazon all using microservices architecture?
- It's all about what you are **optimizing for**.
- For example, if you are optimizing for iteration speed, ease of starting out, and you have less than 50 engineers on the team and/or low scale (less than 1000 RPS). Monolith is the way to go.
- However, imagine now you are growing, you have 100, 200, 500 engineers on the team. Since the application is a monolith, there are interdependencies everywhere and it's becoming really hard to maintain.
- Microservices is a way to break up the dependency and a way of enforcing different parts of the codebase to be separate and only depend on each other through specific APIs.



- Another aspect to consider is how much you are scaling. For example, recommendation services might need more GPUs in order to run the machine learning algorithms at scale. However, if you attach a GPU to every server running a monolith, it's not very cost effective. Therefore, you might naturally break up recommendation into its own sub service, when you start doing that with everything, now you have a microservice architecture.
- Another benefit is if you have multiple teams of engineers, it makes sense for each team to maintain one or more of their own services. They can be hyper focused on understanding and the health of their own services, and make the best possible decision for their own services, scaling and developing each service effectively in isolation.
- IF you use PostgresQL with maybe a message queue, you are already to a certain extend using multiple services. It's just extending the ideas a bit further to build your own service into multiple services.



How to structure your code

MVC

- A fairly traditional pattern, largely used in the backend
- Stands for Model - View - Controller
- The concept you want to think about for MVC is that typically in a large application, the two aspects that are prone to change is model (aka the data store) and the view (aka the GUI, or ways of interacting with the software)
- For example, for a model, maybe instead of connecting to an externally hosted Postgres instance, you might want to deploy a smaller version of your app to a client's site and servers. For that purpose, maybe you want to use a simpler database like SQLite.
- As another example, perhaps you would like to test your app and have predictable results. You may mock all the methods/functions in your model into a dummy database and test things in isolation.
- For the view part, you may want to target different types of output. For example, you may want to have two different views targeting either HTML or JSON. HTML is for server side rendering for lite usage, and JSON would power a full featured mobile experience.
- Controller is where you would keep all of the business logic and coordinate communication between models and views of your application.

- Key thing to note is that MVC is just an organizational mental framework, and not necessarily hard and fast rules about how things are done. For example, Django is what they claim to be an MTV framework, where view = controller and template = view. The general idea is that you should keep business logic (controller) as much separated from model (data) and view (presentation) as possible. Allowing each component to be independently change as possible.

MVVM

- More of a frontend pattern. Stands for Model - View - View Model. Key difference is data binding.
- Used by a lot of frontend frameworks like Angular and React, but originally from knockout.js
- One of the key issues on the frontend is how do you update the view and keep it in sync with the data.
- The main idea is that the model is designed to store data in as a convenient format for manipulation as possible, and View Model is specifically for representing data that is shown to the user. Typically once the model is ready to be displayed, and the data is transformed into view model and it automatically gets shown to the user
- Example: https://knockoutjs.com/documentation/observables.html
- One of the key aspects to building robust UIs is that it's declarative. Instead of some piece of data is updated and now you have to figure out the imperative way of step by step how to update your UI. You basically associate a set of data with a UI and tell it, "here is some data, make sure the UI is updated as the data is updated". Put another way, declarative languages allow you to define the end result, and leave the transformation to someone else to do.
- It's instructive for example to use CSS:

```
nav a:hover {
        background-color: red;
}
```

-
- To do the same in javascript, you would need to do:

```
el.onmouseover = function () {
        this.style.backgroundColor = 'red'
}
el.onmouseout = function () {
        this.style.backgroundColor = '';
}
```

-
- Which is fine, but once you have 50 lines of CSS, you can imagine how much work it is to keep everything in sync and making sure you have a reasonable idea about the state everything is in.

- When is it better to use javascript? In fact you probably wrote something like that above, why do you do it?
- Control, if you are looking for more complex logic than what you can express in CSS, you probably will do the same thing in Javascript, but with more lines of code controlling exactly when things happen and when it doesn't happen. However the vast majority of time, you will use CSS to control basic to moderately complex interactions.
- Another way to think about declarative vs imperative is to use your own body as an example. If you think about running, you do not think about all the biomechanical interactions from your arms to your legs to how your heart needs to pump more blood. All of the imperative actions are handled for you, and your body knows how to react to the end result and get you running from point A to point B.

Architectural diagrams

CRC diagrams

- You should have learnt this in CSC207
- Good refresher: https://wwwcgi.teach.cs.toronto.edu/~csc207h/winter/lectures/L0201/w5/CRC.pdf
- Specifically Class - Responsibilities - Collaborations
- It relates different classes with other classes, and specify what they do (responsibilities) and which class does this one relate to (its collaborators).

Sequence diagram

- Arguably more useful than CRC diagrams to get an overall sense of how data and request flows in a web based system.
- Time flows down, and requests flow horizontally back and forth. Each column is different systems that you may have in the system
- Good example: https://stripe.com/docs/payments/run-custom-actions-before-confirmation


Customer
Client
Server
Stripe
Go to payment page
Send order information
POST /v1/payment_intents secret_key_confirmation =required
Creates new PaymentIntent
Return PaymentIntent client_secret
Return new PaymentIntent
Provide billing details and finalize payment
Call stripe.updatePaymentIntent()
Creates PaymentMethod and attaches it to the PaymentIntent
Return PaymentIntent
Send updated PaymentIntent to server
Confirm PaymentIntent
Attempts payment
Returns PaymentIntent with status succeeded No authentication required
Send webhook event with payment status

Database UML diagrams

- Mostly used for modeling database schemas. This diagram is a simple visual representation of how different tables relates to each other. This is similar to CRC diagrams, where instead of doing it for your classes and objects, you are thinking more in terms of data representation

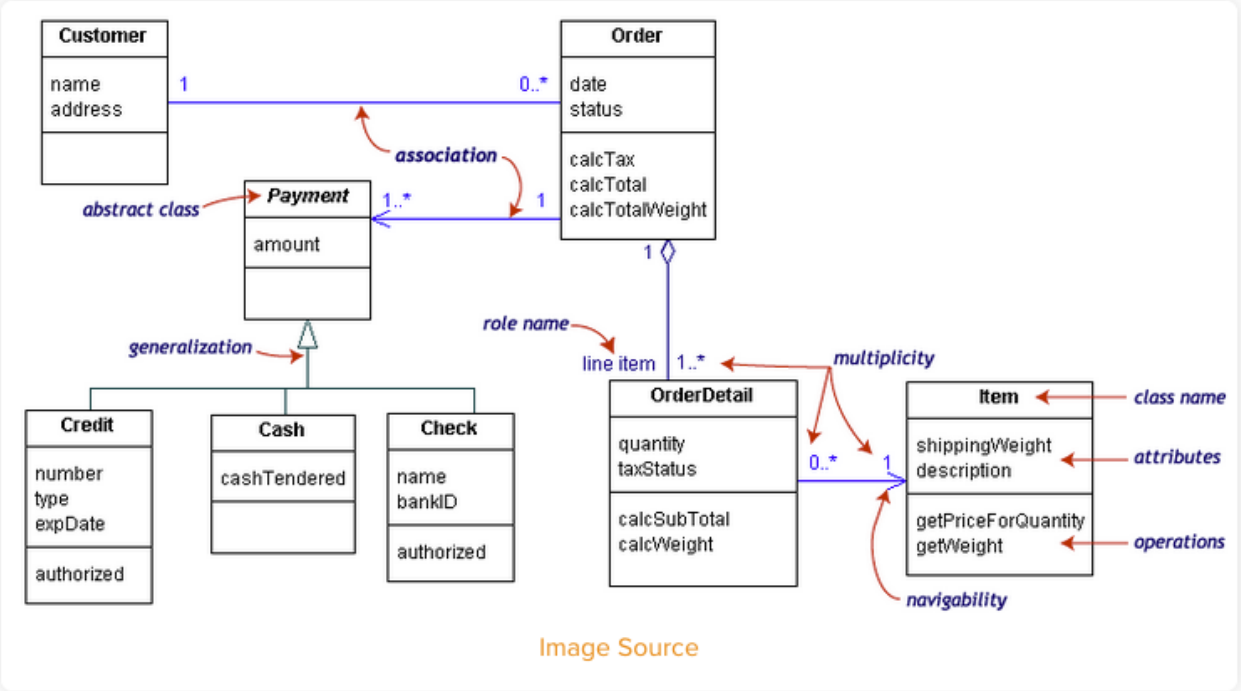


Image Source