In today’s episode, we will be learning about how the UI makes a call to the backend api call and fetch the data.

Differences between monolith and microservice architectures:

**Monolith Architecture:**

* **Single Codebase:** Everything (UI, backend logic, APIs, databases, etc.) is bundled together in one large application.
* **A single code base for everything that is required to develop in the project.**
* **Tightly Coupled:** All parts are interconnected, meaning a change in one area might affect others.
* **Deployment:** Even if you make a small change to one module (e.g., a minor backend logic tweak), the entire application must be redeployed.
* **Scaling:** Scaling is less flexible—you need to scale the entire application rather than just the part that requires more resources.
* **Maintenance:** Can become complex as the application grows, with a higher chance of code conflicts and dependencies.

**Microservices Architecture:**

* **Separation of Concerns:** Each service is dedicated to a specific business function (e.g., user authentication, order processing, frontend), and they communicate over a network, typically via APIs.
* **Loosely Coupled:** Each service operates independently, so changes to one service don't directly affect others.
* **Deployment:** You can deploy, scale, and maintain each service independently. For instance, if you make a change to the authentication service, only that service needs to be redeployed.
* **Scaling:** Services can be scaled independently based on their load. For example, if your authentication service needs more resources, you can scale just that service.
* **Flexibility:** Different services can be built using different technologies or programming languages based on what best fits the needs of that service.

Some insights into async nature of JS:  
 **Asynchronous behavior in JavaScript** allows the code to continue executing without waiting for long-running tasks, such as fetching data from a server, file reading, or timers. This helps prevent the blocking of the main thread (like the UI) while the operation is happening in the background.

 **Callbacks, Promises, and async/await** are mechanisms to handle asynchronous operations in JavaScript:

* **Callbacks** are functions passed into other functions to be executed once an operation completes.
* **Promises** represent the eventual completion (or failure) of an asynchronous operation and its result. They allow chaining with .then() for success and .catch() for errors.
* **async/await** simplifies working with promises. It makes the asynchronous code look like synchronous code, which improves readability.

 **async functions**:

* Any function declared with the async keyword automatically returns a promise.
* If the function returns a value, that value is wrapped in a resolved promise. For example, async function() { return 5; } is equivalent to Promise.resolve(5).
* If the function throws an error, that error is wrapped in a rejected promise.

 **await** pauses the execution of the function until the promise is resolved or rejected:

* You can only use await inside an async function.
* When you use await before a promise, JavaScript pauses the execution of the async function until the promise settles (either resolves or rejects).
* Once the promise resolves, the function resumes and you can use the resolved value directly. If the promise is rejected, the function will throw an error that can be caught using try/catch.

**async function fetchMoviesJSON() {**

**const response = await fetch('/movies');**

**const movies = await response.json();**

**return movies;**

**}**

**fetchMoviesJSON().then(*movies* => {**

**movies; *// fetched movies***

**});**

 **fetch() returns a promise** to represent the ongoing HTTP request.

* If the **promise is resolved**, you get a **Response object** (which may contain success data or error status like 404).
* If the **promise is rejected** (due to a **network error** or some failure in making the request), you get an **error object**.

 You use await in front of fetch and response.json() because you want to **wait** for the promise to either resolve (into a response object) or reject (into an error object) **before moving forward** in the code. This ensures that you are working with the actual data (or a response) instead of incomplete or unresolved promise.

A shimmer UI is a temporary placeholder until the API fetches the data from the backend and renders it on the screen.