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 developin 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.

**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.