##### ****What challenges do you face when developing distributed systems with ASP.NET Core?****

* ****Complexity:**** Increased complexity in managing multiple services, inter-service communication, and data consistency.
* ****Deployment:**** Coordinating deployment across multiple services.
* ****Monitoring and Logging:**** Centralizing logs and monitoring from disparate services.
* ****Latency:**** Increased latency due to network calls between services.

##### ****Explain the concept of versioning APIs in ASP.NET Core. How do you version your APIs?****

API versioning in ASP.NET Core allows you to support multiple versions of an API simultaneously, ensuring backward compatibility while allowing for new features and changes. Versioning can be achieved through different methods, such as URL path, query string parameters, header values, or content negotiation. ASP.NET Core supports API versioning through the Microsoft.AspNetCore.Mvc.Versioning package, which provides attributes and services to define and manage API versions easily.

##### ****Explain the Repository pattern. How do you implement it with Entity Framework Core in ASP.NET Core?****

The Repository pattern abstracts the data layer, providing a collection-like interface for accessing domain objects. It helps decouple the application’s business logic from data access logic. To implement it with Entity Framework Core in an ASP.NET Core application, you define repository interfaces and classes that use an EF Core context to perform CRUD operations. These repositories are then injected into your services or controllers, allowing for cleaner, more maintainable code by separating concerns.

When multiple requests come at the same time to an ASP.NET Core Web API, there are several ways to handle them effectively. Here’s a simplified guide:

### 1. ****Use Asynchronous Code****

* When your API does tasks like talking to a database, make sure to use async and await keywords. This lets the server handle other requests while waiting for a response, instead of being stuck.

### 2. ****Limit Simultaneous Requests****

* You can set a limit on how many requests the server will handle at once. If too many requests come in, others will have to wait.

3. **Prevent Blocking the Server**

* Avoid blocking threads by using non-blocking code. Synchronous (blocking) code can tie up server resources, reducing performance.

### 4. ****Control Heavy Requests****

* Use strategies like **rate limiting** to control how many requests a single user or client can make within a time period.
* You can also add **circuit breakers** to stop certain actions if they’re failing too often.

### 5. ****Use Connection Pooling****

* When working with databases or external APIs, reuse connections instead of opening new ones all the time. This reduces overhead and improves performance.

### 6. ****Use Load Balancing****

* If you expect a lot of traffic, distribute the requests across multiple servers using a load balancer.

### 7. ****Caching****

* Store frequently requested data in memory or in a cache so the API doesn’t need to keep fetching the same information over and over.

### 8. ****Request Throttling****

* Implement a feature to slow down or reject requests when the server is too busy.

### 9. ****Separate Long-Running Tasks****

* Move time-consuming tasks (like sending emails or generating reports) to the background, so your API can quickly respond to the user while processing heavy tasks separately.

### 10. ****Scale Horizontally****

* If your server is overwhelmed by traffic, consider adding more servers or instances to handle the load. This is called **horizontal scaling**.

### 11. ****Monitor Performance****

* Keep an eye on how your server is doing (CPU, memory usage, etc.). You can use tools like Application Insights to set up alerts if things are going wrong.

By following these steps, your API will be better equipped to handle a large number of requests coming in at the same time without slowing down