**Context Diagram**

**Diagram

Description automatically generated**

**Database Diagram**



**Solution Architecture**

high-level solution architecture for a Store Management app that was built using .NET Core API + Angular:

.NET Core API:

1. ASP.NET Core was used to build the RESTful API.
2. Entity Framework Core was utilized for data access and management.
3. The Repository pattern was implemented to decouple data access from the API and business logic.
4. Dependency injection mechanism was used to decouple higher level module to lower level module.
5. The API can be deploy on a cloud platform such as Azure or AWS.

Angular:

1. Angular was used as the front-end framework for building the client-side application.
2. Angular Router was used for navigation and routing.
3. Reactive Forms were implemented for building forms.
4. Angular HttpClient was used to call the .NET Core API.
5. Observables and RXJS were implemented for handling asynchronous data and events.
6. Bootstrap was used for UI components.

**Design Decisions**

Here are some design decisions that may have been considered while building the Store Management app using .NET Core API + Angular:

**.**NET Core API:

1. Choosing ASP.NET Core as the framework for building the API: This choice was likely made due to its lightweight, high-performance, and cross-platform capabilities.
2. Using Entity Framework Core for data access and management: This decision may have been made to simplify data access and management, as Entity Framework Core is a widely used ORM tool in the .NET ecosystem.
3. Implementing Repository pattern: This decision may have been made to separate the data access logic from the API business logic, making it easier to test and maintain the code.

Angular:

1. Choosing Angular as the front-end framework: Angular was likely chosen for its powerful and scalable features, as well as its popularity and large community.
2. Using Angular Router for navigation and routing: This decision may have been made to simplify navigation and routing within the application.
3. Implementing Reactive Forms: This decision may have been made to simplify form creation and validation in the Angular application.
4. Using Angular HttpClient to call the .NET Core API: This choice was likely made to simplify API calls from the Angular application.
5. Implementing observables and RXJS: This decision may have been made to handle asynchronous data and events in a clean and efficient manner.

**Non-functional requirements considered and how the design addresses them**

1. The Country, State and City Id are introduced in the store table. So by this we can preserve the address of retail chains and identify which store belongs to which country.
2. Async programming concepts are used while developing the API. Async programming with the async and await API in .NET Core improves performance, resource utilization, scalability, user experience, and efficient use of resources by allowing simultaneous execution of multiple tasks and releasing resources when not in use.
3. Pagination is used in the grid which displays the pricing information across the stores. By this way the large amount of data can be handled and without affecting the performance of the page.

**Assumption**

1. The product details will be same across the different store.
2. The SKU and price of product can be different across different store.
3. The data storage needs of the application can be fulfilled using a relational database, such as Microsoft SQL Server.