CHAPTER - 6
SYSTEM DESIGNING

6.1 LOW LEVEL DESIGNING

6.1.1 SOLID Principle

SOLID principles are fundamental guidelines in object-oriented design, emphasizing principles like single responsibility and dependency inversion. Applying SOLID principles in ASP.NET Web API development ensures that each component has a clear purpose, promoting modularity and maintainability.

The Single Responsibility Principle advocates for each class or module to have only one responsibility, enhancing code clarity and testability.

The Open/Closed Principle suggests that software entities should be open for extension but closed for modification, encouraging developers to use abstraction and inheritance to accommodate change.

The Liskov Substitution Principle emphasizes the importance of maintaining substitutability of derived classes for their base classes, ensuring consistency and predictability in the system.

The Interface Segregation Principle advises against creating large, monolithic interfaces, instead favouring smaller, more focused interfaces to prevent clients from depending on methods they do not use.

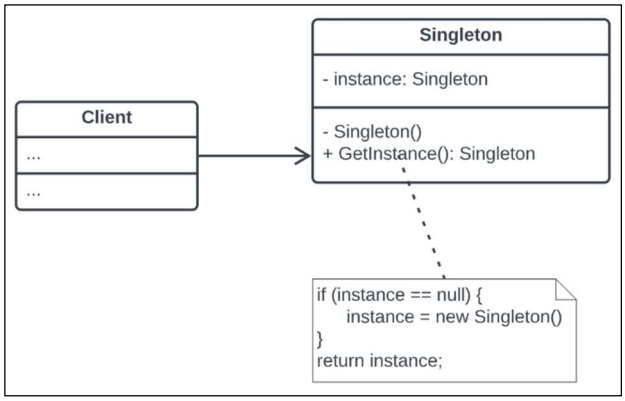
The Dependency Inversion Principle encourages designing components to depend on abstractions rather than concrete implementations, promoting loose coupling and facilitating easier maintenance. Adhering to SOLID principles in ASP.NET Web API development fosters modularity and maintainability, leading to robust and scalable APIs that are easier to maintain and extend over time.

6.1.2 DbFactory Design Pattern

The dbFactory design pattern abstracts database connection management, offering a centralized factory for creating database-specific objects. By encapsulating database creation logic, the dbFactory pattern simplifies database interactions in ASP.NET Web API development, facilitating seamless integration with different database providers.

6.1.3 Singleton Design Pattern

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance.



[Fig 6.1.3.1 Singleton Design Pattern]

In ASP.NET Web API, the Singleton pattern can be applied to manage resources that need to be shared across multiple requests, such as caching objects or configuration settings. Care must be taken when implementing Singletons in a multi-threaded environment to avoid race conditions and ensure thread safety.

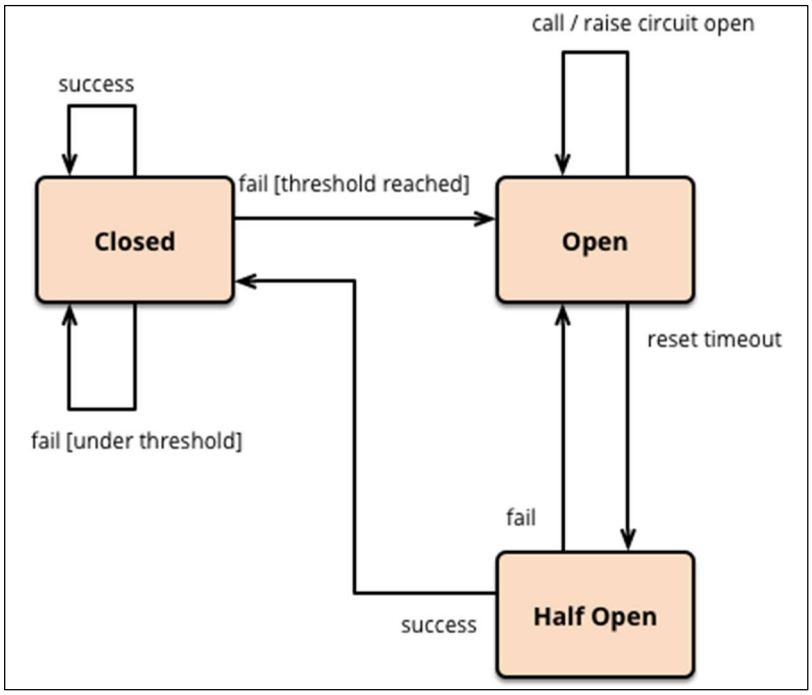
6.1.4 Flyweight Design Pattern

The Flyweight pattern minimizes memory usage by sharing common state between multiple objects. In the context of ASP.NET Web API, the Flyweight pattern can be applied to optimize memory usage when dealing with large datasets or frequently accessed resources. For example, in a Web API serving static content, the Flyweight pattern can be used to cache and reuse common resources like images or CSS files across multiple requests.

6.1.5 Circuit Breaker Design Pattern

The Circuit Breaker pattern provides a mechanism to handle faults in distributed systems by temporarily blocking requests to a failing component.

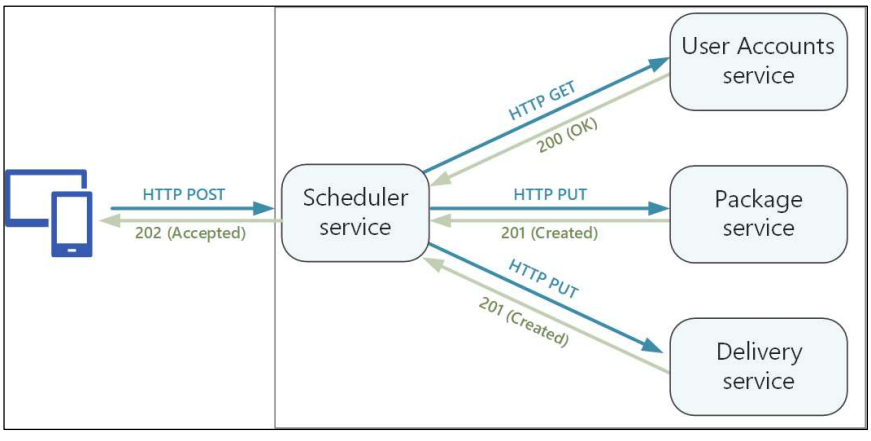
In ASP .NET Web API development, the Circuit Breaker pattern can be used to improve resilience and prevent cascading failures when calling external services or APIs. By monitoring the health of external dependencies and selectively opening or closing the circuit based on predefined thresholds, the Circuit Breaker pattern helps maintain system stability and performance.



[Fig 6.1.5.1 Circuit Breaker Design Pattern]

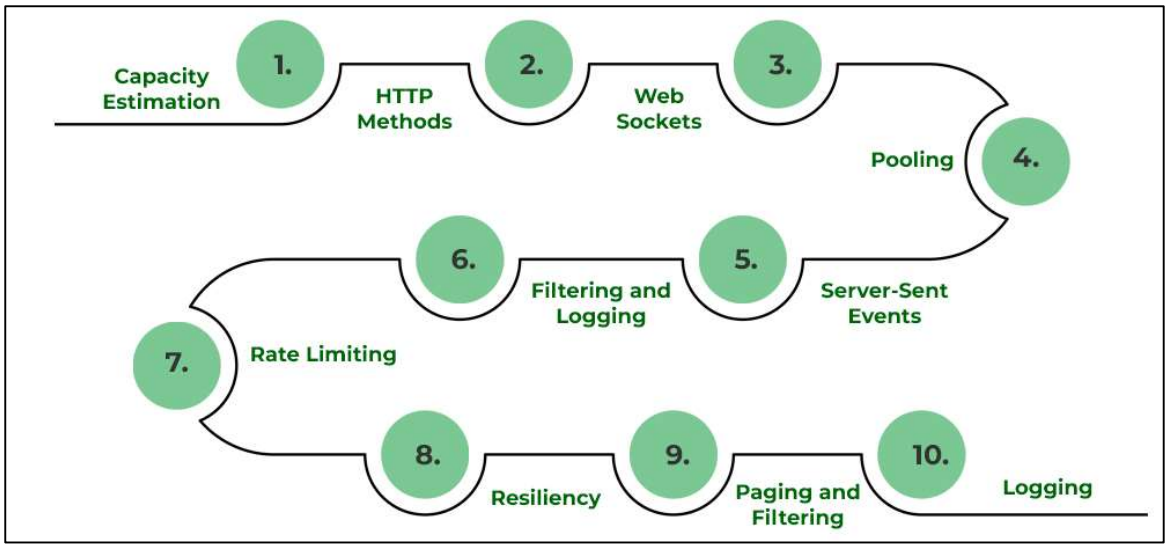
6.2 HIGH LEVEL DESIGNING

High-level designing of a web API within a microservice architecture involves decomposing the system into small, independently deployable services. Utilizing asynchronous messaging patterns enhances scalability and responsiveness. Incorporating Redis as a cache layer improves performance by storing frequently accessed data in memory. Security measures such as authentication and encryption ensure data protection. Monitoring and observability tools enable real-time monitoring of service health. Containerization simplifies deployment and scaling. Overall, these practices facilitate the development of robust and scalable APIs that meet modern application demands.



[Fig 6.2.1 Microservice HLD]

Example of HLD design with services is referenced above which is prepared by Microsoft for understanding purpose which insists to use dependency injection to follow LLD rules properly.



[Fig 6.2.2 Checkpoints of HLD]

Correct implementation & HLD designing of any application leads to the most user friendly and efficient application.