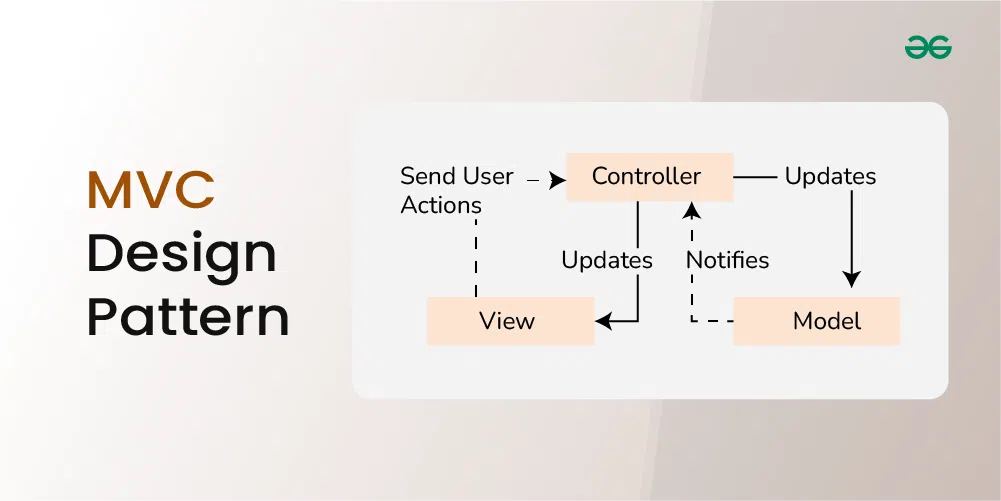
DAY-2

ASSIGNMENT -1

1.MVC(model view controller): The MVC design pattern is a software architecture pattern that separates an application into three main components: Model, View, and Controller, making it easier to manage and maintain the codebase. It also allows for the reusability of components and promotes a more modular approach to software development.

The **Model View Controller** (MVC) design pattern specifies that an application consists of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects.

* The MVC pattern separates the concerns of an application into three distinct components, each responsible for a specific aspect of the application’s functionality.
* This separation of concerns makes the application easier to maintain and extend, as changes to one component do not require changes to the other components.
* **1. Model**
* The Model component in the MVC (Model-View-Controller) design pattern represents the data and business logic of an application. It is responsible for managing the application’s data, processing business rules, and responding to requests for information from other components, such as the View and the Controller.
* **2. View**
* Displays the data from the Model to the user and sends user inputs to the Controller. It is passive and does not directly interact with the Model. Instead, it receives data from the Model and sends user inputs to the Controller for processing.
* **3. Controller**
* Controller acts as an intermediary between the Model and the View. It handles user input and updates the Model accordingly and updates the View to reflect changes in the Model. It contains application logic, such as input validation and data transformation.

ASSIGNMENT-2

Applications of microservices and Event driven Architecture in E Commerce App:

1. **Scalability**
2. **Fault Isolation**
3. **Real-time Updates**
4. **Flexibility and Innovation**
5. **Order Processing**
6. **Inventory Management**
7. **User Activity Tracking**
8. **Payment Processing**
9. **Notifications and Alerts**
10. **Promotions and Campaigns**

In designing an e-commerce application, adhering to SOLID principles can help create a robust, maintainable, and scalable system. The SOLID principles, coined by Robert C. Martin, are a set of five design principles that promote good software design practices:

1. **Single Responsibility Principle (SRP)**:
   * Each class or module should have only one reason to change.
   * In an e-commerce app, separate modules should handle specific responsibilities such as product management, user authentication, order processing, and payment handling. This ensures that each module has a clear and focused purpose, making the system easier to understand, maintain, and extend.
2. **Open/Closed Principle (OCP)**:
   * Software entities (classes, modules, functions) should be open for extension but closed for modification.
   * In the context of an e-commerce app, this principle suggests designing components in a way that allows for adding new features or functionality without modifying existing code. For example, using interfaces and abstract classes to define contracts between modules enables easy extension through inheritance or composition.
3. **Liskov Substitution Principle (LSP)**:
   * Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.
   * In e-commerce app design, this means that derived classes (e.g., specific types of products or payment methods) should be substitutable for their base class (e.g., generic product or payment interface) without introducing unexpected behavior or breaking the system's functionality.
4. **Interface Segregation Principle (ISP)**:
   * Clients should not be forced to depend on interfaces they do not use.
   * When designing interfaces in an e-commerce app, it's important to keep them cohesive and focused on specific sets of behaviors required by clients. This prevents clients from being burdened with unnecessary dependencies and reduces the risk of interface pollution.
5. **Dependency Inversion Principle (DIP)**:
   * High-level modules should not depend on low-level modules. Both should depend on abstractions.
   * Abstractions should not depend on details; details should depend on abstractions.
   * In the context of e-commerce app design, this principle advocates for designing modules to depend on abstract interfaces rather than concrete implementations. For example, decoupling components such as database access, external services, and business logic through dependency injection allows for easier testing, maintainability, and flexibility.

**DRY (Don't Repeat Yourself)**:

* DRY principle emphasizes that each piece of knowledge or logic within a system should have a single, unambiguous representation.
* In an e-commerce application, DRY can be applied by avoiding code duplication. For instance:
  + Ensure that common functionalities, such as user authentication, product listing, and order processing, are implemented in reusable modules or services.
  + Use templates or components for user interface elements that are repeated across multiple pages to avoid duplicating HTML/CSS/JavaScript code.
  + Centralize data access and business logic to avoid redundant database queries or repetitive code blocks.

**KISS (Keep It Simple, Stupid)**:

* KISS principle advocates for simplicity in design and implementation. It suggests that systems should be kept as simple as possible while still meeting the requirements.
* In the context of an e-commerce application:
  + Prioritize simplicity and clarity in the user interface to ensure ease of use for customers. Complex user journeys or overly cluttered interfaces can lead to confusion and frustration.
  + Keep the architecture and design of the application straightforward and easy to understand. Avoid over-engineering or adding unnecessary complexity that can increase development time and introduce potential bugs.
  + Focus on essential features and functionalities that directly address the needs of users and the business. Avoid adding features just for the sake of it, as they can add unnecessary complexity and maintenance overhead.