Software Engineering -2-

Lecture -2-

Dr. Inas Laila

الفصل الدراسي الاول ٢٠٢٥ / ٢٠٢٦

Modularization:

ما المقصود بـ **Modularization** ولماذا نقوم به؟؟

Modularization is the process of dividing a software system into multiple independent modules where each module works independently.

There are many advantages of Modularization in software engineering. Some of these are given below:

- Easy to understand the system.
- A module can be used many times as their requirements. No need to write it again and again (reuse).
- System maintenance is easy.

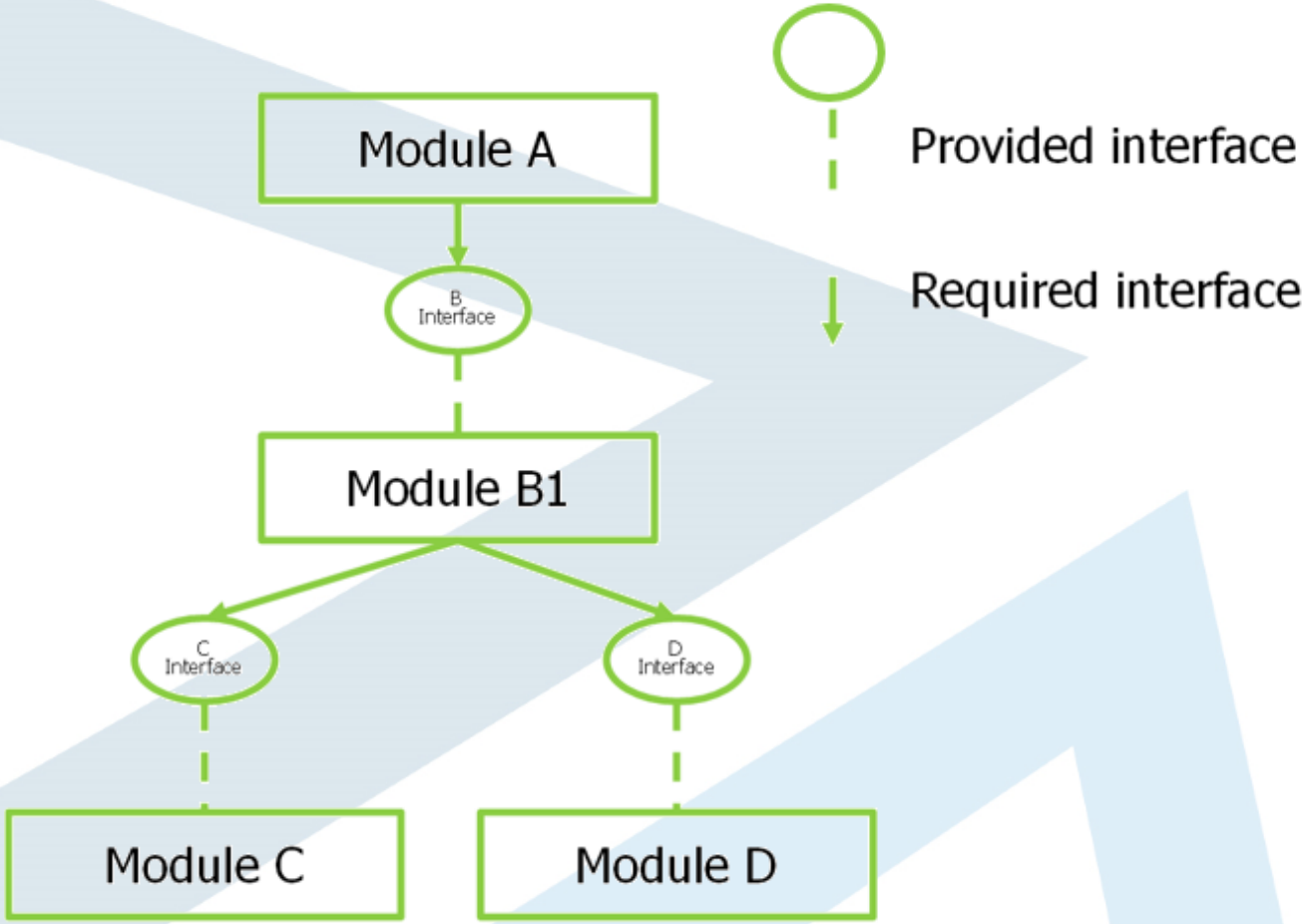
Modularity: Breaks the "big problem" into smaller, more manageable pieces, which are easier to change without affecting other modules

There are some important properties of a software module , A module…

1.Encapsulates functionality and data: Encapsulation is about hiding the implementation details of a module.. Other modules can only access functionality and data via the provided interface. Generally, it is good to encapsulate as much as possible.

2.Provides an interface to other modules for accessing its functionality and data: The software module interface should be as abstracted and as small as possible.

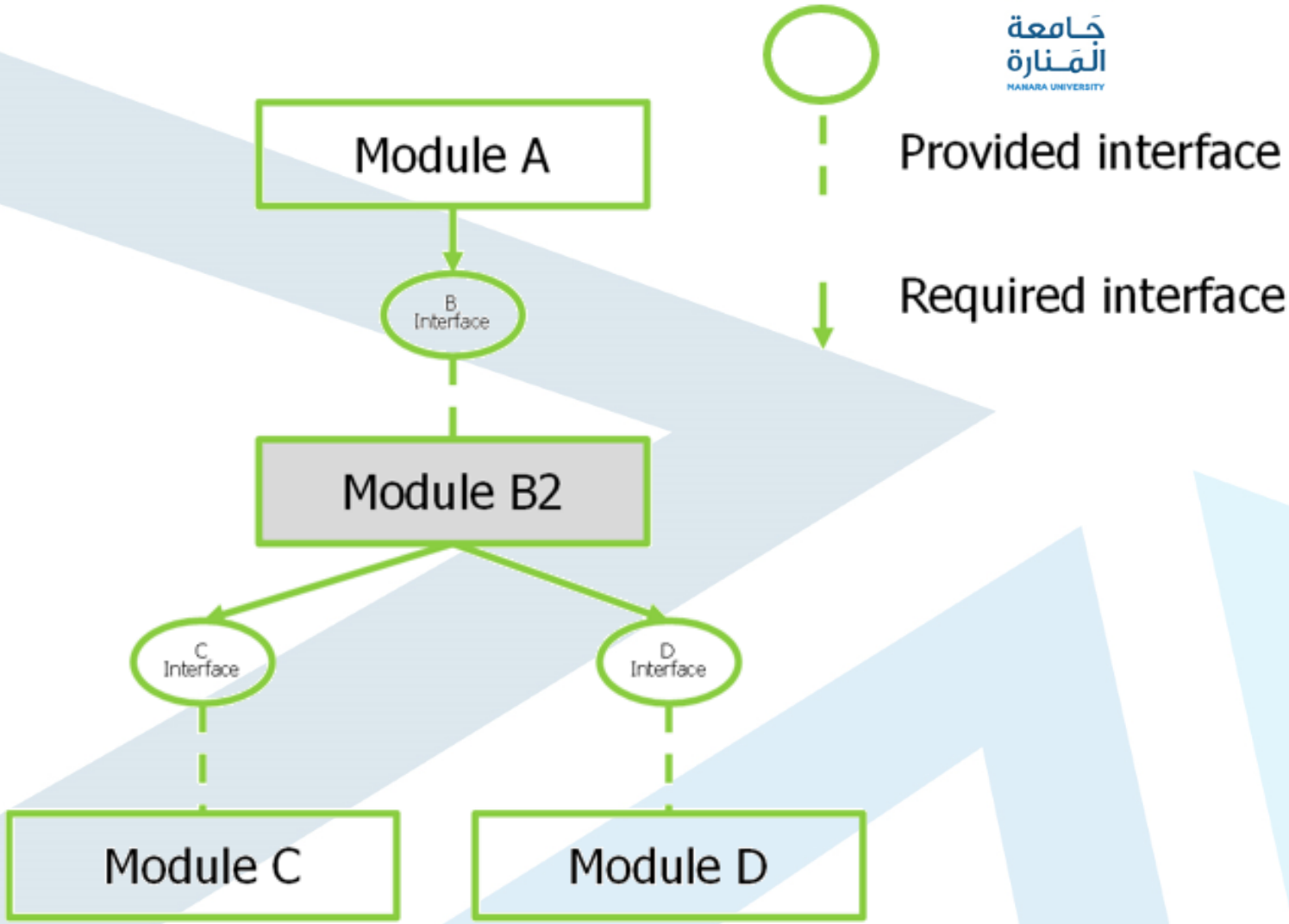
3.Requires interfaces provided by other software modules: To realize the functionality of a software module, it must sometimes use the provided interfaces of other modules. **A software module that uses another module through its interface has a dependency on that other module. The fewer dependencies, the better.**



Module B1 provides Interface B to Module A and requires Interfaces C and D.

Assume we have a new Module B2 with improved performance.

If Module B2 provides the same interface and requires the same interfaces as B1, then it is interchangeable with B1.


Module A
B
Interface
Module B2
C
Interface
D
Interface
Module C
Module D
Provided interface
Required interface

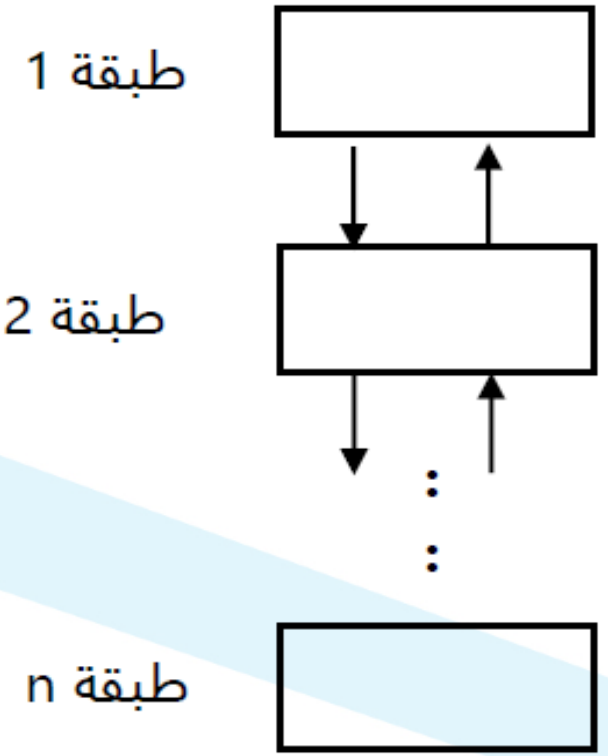
What are software architecture patterns in Software Development?

1. Layered Architecture
2. Client-Server Architecture
3. Repository architecture
4. Pipe and filter architecture
5. MVC Architecture
6. Message Broker Architectural

……

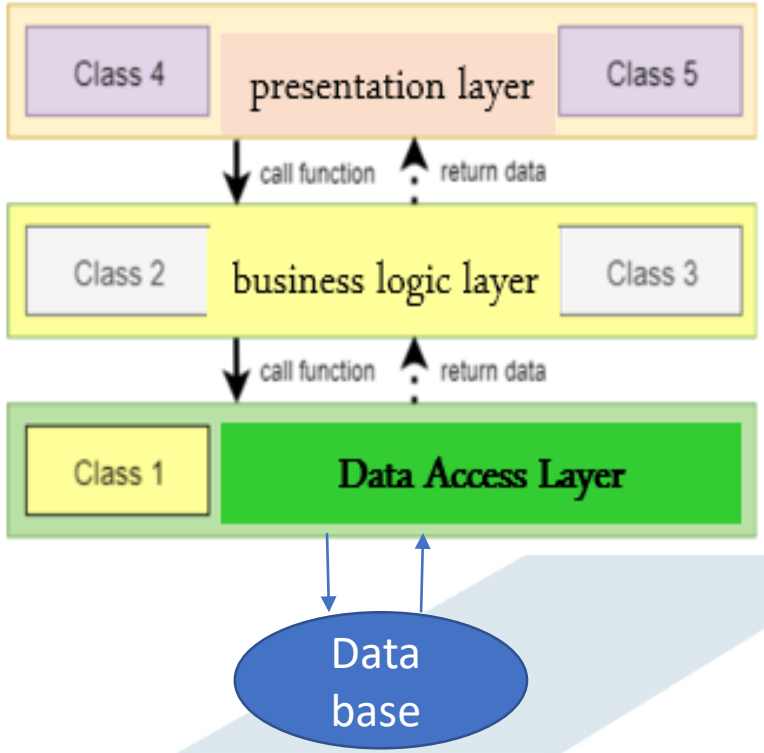
Layered Architecture:

الهندسة المعمارية الطبقية هي نمط تصميم برمجي يُنظّم البرمجية إلى طبقات أفقية، تُركّز كل طبقة على مهمة مُحدّدة مما يُعزز فصل المهام. تعمل الطبقات معًا لمعالجة البيانات، وإدارة منطق العمل، والتفاعل مع المستخدمين، مما يضمن عدم تحمّل أي طبقة بمفردها جميع المسؤوليات. يُشار إلى هذا النمط المعماري أيضًا باسم الهندسة المعمارية متعددة الطبقات، ويُستخدم على نطاق واسع في مختلف المجالات، بما في ذلك تطبيقات الويب، وبروتوكولات الشبكة، والأنظمة الموزعة. الهدف الرئيسي هو تغليف جوانب مُختلفة من التطبيق - مثل واجهة المستخدم، ومنطق العمل، وإدارة البيانات - في طبقات مُستقلة تتفاعل عادةً بطريقة تنازلية فقط مع الطبقات المُجاورة.



تشمل أفضل الممارسات تحديد مسؤوليات واضحة لكل طبقة، ووضع قواعد اتصال بين الطبقات المتجاورة، وتوثيق تفاعلات الطبقات لضمان الاتساق والوضوح أثناء التطوير والصيانة.

مثال : 3 layer architecture



ينقسم التطبيق إلى ثلاث طبقات أساسية على الأقل في بنية الطبقات النموذجية.

طبقة العرض presentation layer: مسؤولة عن التعامل مع جميع تفاعلات المستخدم وهو بمثابة الواجهة الأمامية للتطبيق، حيث يوفر واجهة للمستخدم ويجمع مدخلات المستخدم.

Focuses on how data is displayed and how users interact with the system.
Examples: Web pages, mobile app interfaces, or desktop GUIs.

الطبقة التالية هي طبقة منطق الأعمال business logic layer، وهي الطبقة التي تعالج البيانات، وتُنفذ قواعد وعمليات العمل الأساسية، وتتخذ القرارات. هذه الطبقة هي قلب عمليات التطبيق

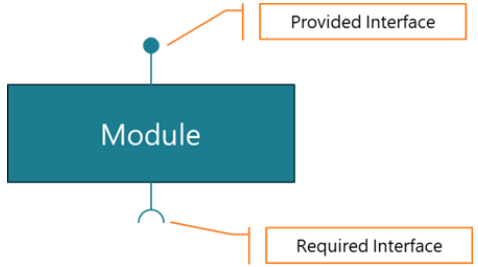
Concentrates on the application's business logic and rules.
Examples: Validation logic

الطبقة الثالثة هي طبقة الوصول data access layer وهي الطبقة التي تُدير عملية تخزين البيانات واسترجاعها وهي المسؤولة عن التواصل مع قاعدة البيانات.

Manages data access and interaction with storage systems
Examples: Database queries, caching, and APIs.

يساعد التحديد الواضح للمسؤوليات في الحفاظ على الفصل النظيف بين الاهتمامات، مما يجعل إدارة التطبيق وتوسيع نطاقه أسهل.

The actual data storage system where information is saved.
Examples: Relational databases (SQL Server, PostgreSQL) or NoSQL solutions (MongoDB)



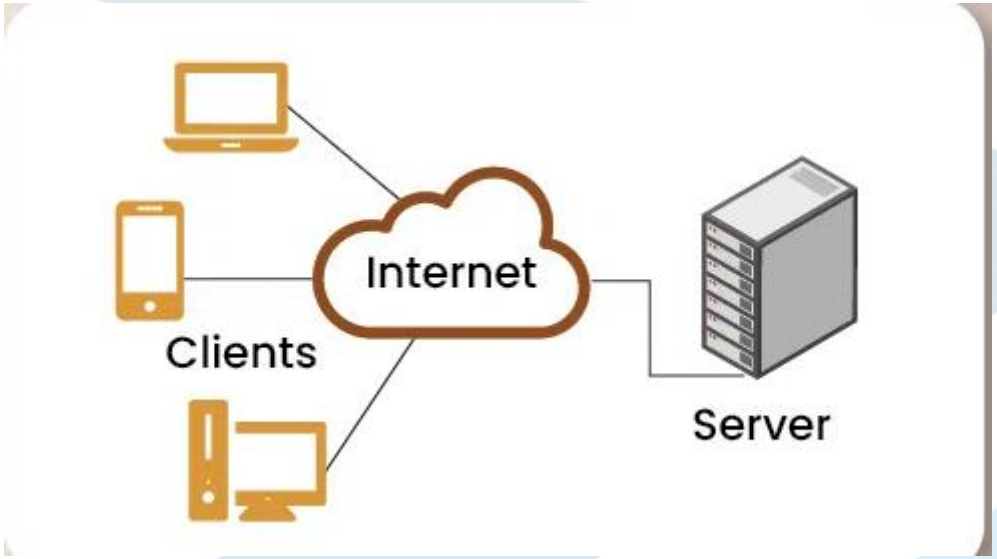
name	Layered architecture
Description	**Separation of concerns** is a foundational principle of layered architecture. Each layer focuses on a single aspect of the application ينظّم النظام في طبقات، ولكل طبقة وظائف محددة. تُقدّم كل طبقة خدمات للطبقة التي تعلوها، بحيث تُمثّل الطبقات الأدنى مستوىً الخدمات الأساسية التي يُحتمل استخدامها في جميع أنحاء النظام. **Dependencies are minimized**: Changes in one layer impact only adjacent layers. **Loose Coupling**: Each layer interacts with others through defined interfaces, reducing interdependencies.
Advantages	يسمح باستبدال الطبقات بأكملها طالما تم الحفاظ على الواجهة بين الطبقات.
Disadvantages	عمليًا، غالبًا ما يكون توفير فصل واضح بين الطبقات أمرًا صعبًا، على الرغم من مزاياها، قد تُسبب البنية الطبقية عبئًا إضافيًا على الأداء بسبب مرور الطلب عبر طبقات متعددة، خاصةً إذا كان عدد الطبقات كبيرا(مشكلةً البطء الناتج عن تعدد مستويات تفسير طلب الخدمة أثناء معالجته في كل طبقة). Each request must pass through multiple layers. This sequential flow can introduce latency and inefficiencies. Increases response time for simple queries. Can lead to unnecessary resource consumption.

الإيجابيات :

- زيادة مستوى الأمان النّاتج عن عزل الـ database في طبقة مستقلّة بحيث لا يمكن الوصول لها إلّا عبر المرور على الطبقات الأعلى.
- سهولة الصّيانة والتعديل النّاتج عن تقسيم الصّفوف حسب الاختصاص، فلو وُجد عطل ما في إحدى الطبقات يتم عزلها وإجراء الصّيانة اللازمة عليها وإعادتها لمكانها.
- تدعم إعادة الاستخدام reusability بحيث يمكن إحضار بعض الطبقات الجاهزة واستخدامها ضمن النّظام.

يُعد مبدأ فصل الاهتمامات جوهريا في البنية الطبقية، إذ يُمكّن من تطوير كل طبقة واختبارها وصيانتها بشكل مستقل.
تُعزز هذه الوحدات النمطية قابلية الاختبار Testability حيث يُمكن اختبار كل طبقة بشكل مُنعزل،
كما تدعم قابلية الصيانة Maintainability مما يسمح بتحديث الطبقات الفردية دون التأثير على النظام بأكمله.
كما تدعم قابلية التوسع Scalability من خلال استخدام نسخ متعدد من طبقة ما في حال وجود حمل زائد عليها
Layers can be scaled independently by creating additional instances (e.g., scaling the Persistence Layer to handle high data traffic).

Client-Server Architecture - System Design



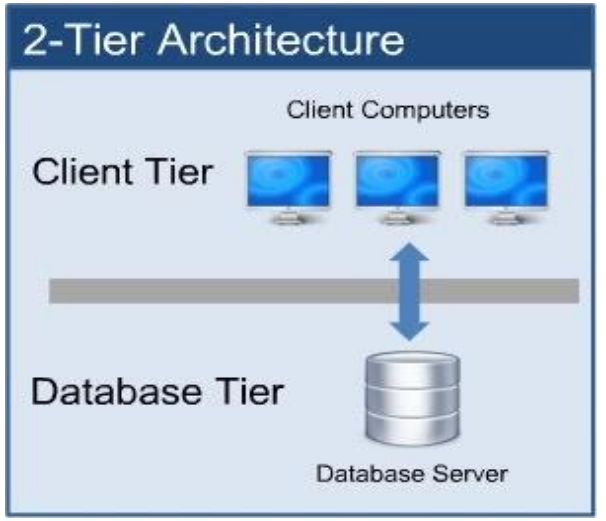
نموذج العميل والخادم في هندسة البرمجيات هو بنية موزعة حيث يطلب العميل، مثل متصفح الويب أو تطبيق الهاتف المحمول أو تطبيق سطح المكتب، خدمات أو موارد من خادم، والذي يقوم بمعالجة الطلب وإرجاع الاستجابة، تتضمن الشبكة عدة عملاء وخادمًا واحدًا او اكثر. العملاء هم أجهزة أو برامج تطلب خدمات أو موارد، بينما الخادم هو جهاز قوي يوفر هذه الموارد أو الخدمات. تتيح هذه البنية إدارة بيانات ومشاركة موارد فعّالة، مما يجعلها شائعة في تطبيقات الويب وقواعد البيانات وغيرها من الأنظمة الشبكية. تُحسّن بنية العميل-الخادم الأداء وقابلية التوسع والأمان وذلك من خلال فصل الأدوار وتوزيع المهام.

In this model, Communication between clients and the server follows a request-response protocol, such as HTTP/HTTPS for web services or SQL for database queries.

This model is fundamental to many modern applications and systems. For example, **when a user visits a website like Wikipedia, their browser (the client) sends an HTTP request to a web server, which then responds by sending back the HTML, CSS, and JavaScript files needed to display the page**

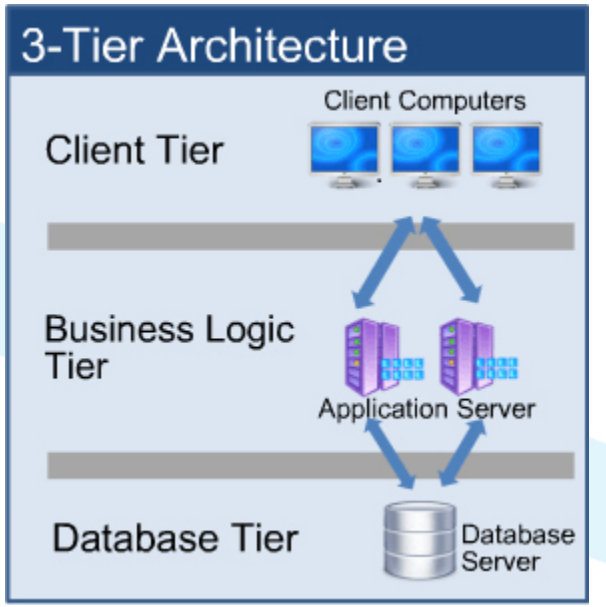
Types of Client-Server Architecture

Two-Tier Architecture (Client-Server): In this type, the client directly communicates with the server to request services and retrieve data.



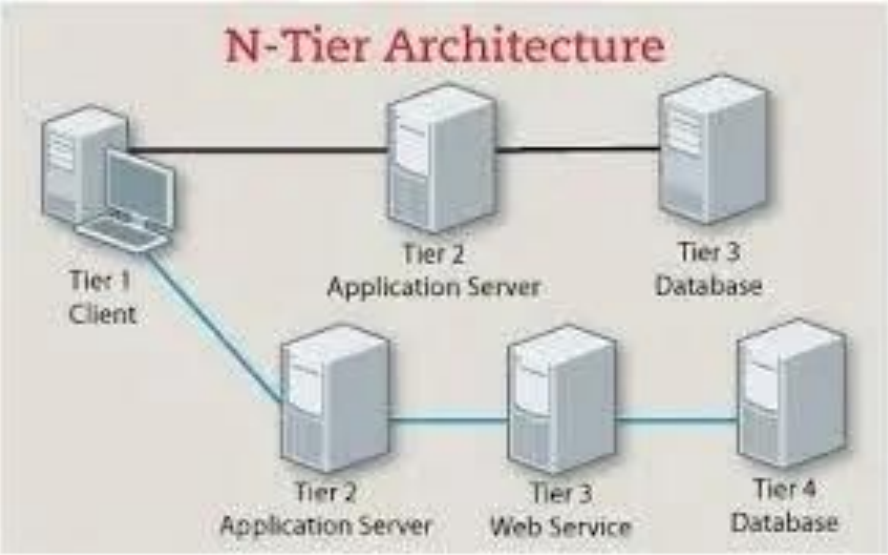
Three-Tier Architecture: In this type, the client interacts with an intermediate layer called the "application server" or "middleware," which acts as a bridge between the client and the database server.

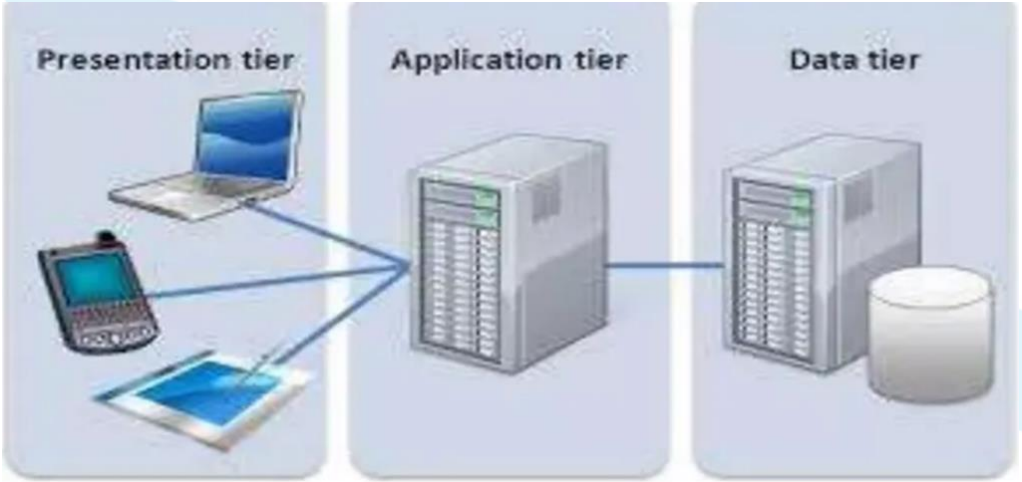
- The three tiers are:
  - Presentation Tier (Client): Responsible for the user interface and capturing user input.
  - Application Tier (Middleware): Handles the business logic and processing of requests. It communicates with the database server to retrieve and store data.
  - Data Tier (Server): This is the database server that stores and manages the data.
- This architecture enhances scalability, as the workload can be distributed between the application server and the database server.
- Example: Web applications where the web browser is the client, the web server is the application server, and the database server stores the data.



N-Tier Architecture:

- N-Tier architecture is an extension of the three-tier architecture and allows for more layers or tiers to be added for specific purposes.
- Additional tiers may include security servers, caching servers, load balancers, etc., depending on the complexity and requirements of the application.
- This architecture provides better modularity, flexibility, and performance optimization.
- Example: Large-scale enterprise applications with multiple layers for security, caching, and load balancing.





Design Principles for Effective Client-Server Architecture

- **Scalability:** Client-server architecture supports scalability. As the number of clients grows, additional servers can be added (**Horizontal Scalability**), or existing server capacities can be expanded(**Vertical Scalability**) to handle increased demand without significantly altering the overall system architecture.
- **Reliability and Availability:** With robust server infrastructure, client-server systems can ensure high reliability and availability. Redundancies such as multiple servers, to ensure the system remains operational in case of failures, backups, and load balancing techniques can be implemented on the server side to minimize downtime and ensure continuous service availability.
- **Enhanced Security:** Centralized servers enable better security controls and data protection measures. Sensitive data can be securely stored on servers, and access can be tightly controlled and monitored. Encryption and authentication mechanisms can be more effectively implemented.
- **Centralized Management:** By centralizing resources and services on a server, this architecture simplifies maintenance, updates, and security management. Administrators can efficiently monitor and manage data, apply updates, and enforce security policies from a single location.
- **Interoperability: Platform Independence:** Design the system to support multiple platforms and devices, allowing various clients to interact with the server.

الإيجابيات :

- دعم أكثر من مستخدم بشكل متزامن.
- تحقيق التواصل بين عدّة أجهزة مختلفة المواقع بنفس الوقت.

السّلبيات :

- إمكانية حدوث heavy-load على الـ server عند وجود الكثير من الـ query.
- عطل الـ server يؤدّي إلى توقف التطبيق عند كل العملاء.
- الأداء ليس متعلّق بالنّظام فقط وإنّما بالشبكة أيضاً.

risk of a single point of failure: if the server goes down, services become unavailable.
Servers can also be vulnerable to denial-of-service attacks if overwhelmed by excessive requests. Additionally, the architecture can introduce latency, especially in complex multi-tier systems.

معمارية المخزن Repository architecture

تتألّف بعض الأنظمة من مجموعة من الأنظمة الفرعية الأخرى التي تتبادل الـ data فيما بينها أحياناً، وتحقيق التواصل بينهم ممكن بطريقتَين :

1- أن توضع الـ data المشتركة في مكان واحد مركزي يمكن لكل sub-systems أن تصل له.

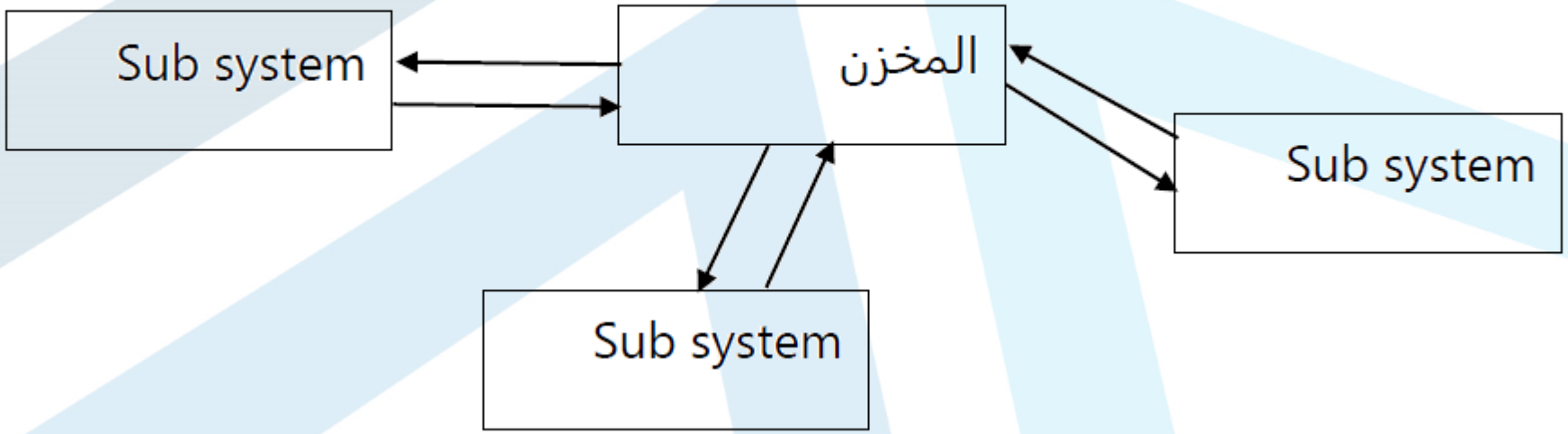
2- أن يكون لكل database sub-system خاصة به ويتم تمرير الـ data من قِبَل كل نظام جزئي لباقي الأنظمة.

في حال كان حجم الـ data المشتركة بين هذه الأنظمة كبير يتم استخدام الطريقة الأولى لأنّها أكثر فعالية، وتُدعى هذه الطريقة بالـ repository model، والـ repository هو المخزن المركزي الذي يحوي الـ data المشتركة.

لاحظ أنّه في هذا النّموذج مَن يصل للـ data ليس بالضرورة أن يكون طبقة control وإنّما application آخر بحدّ ذاته.

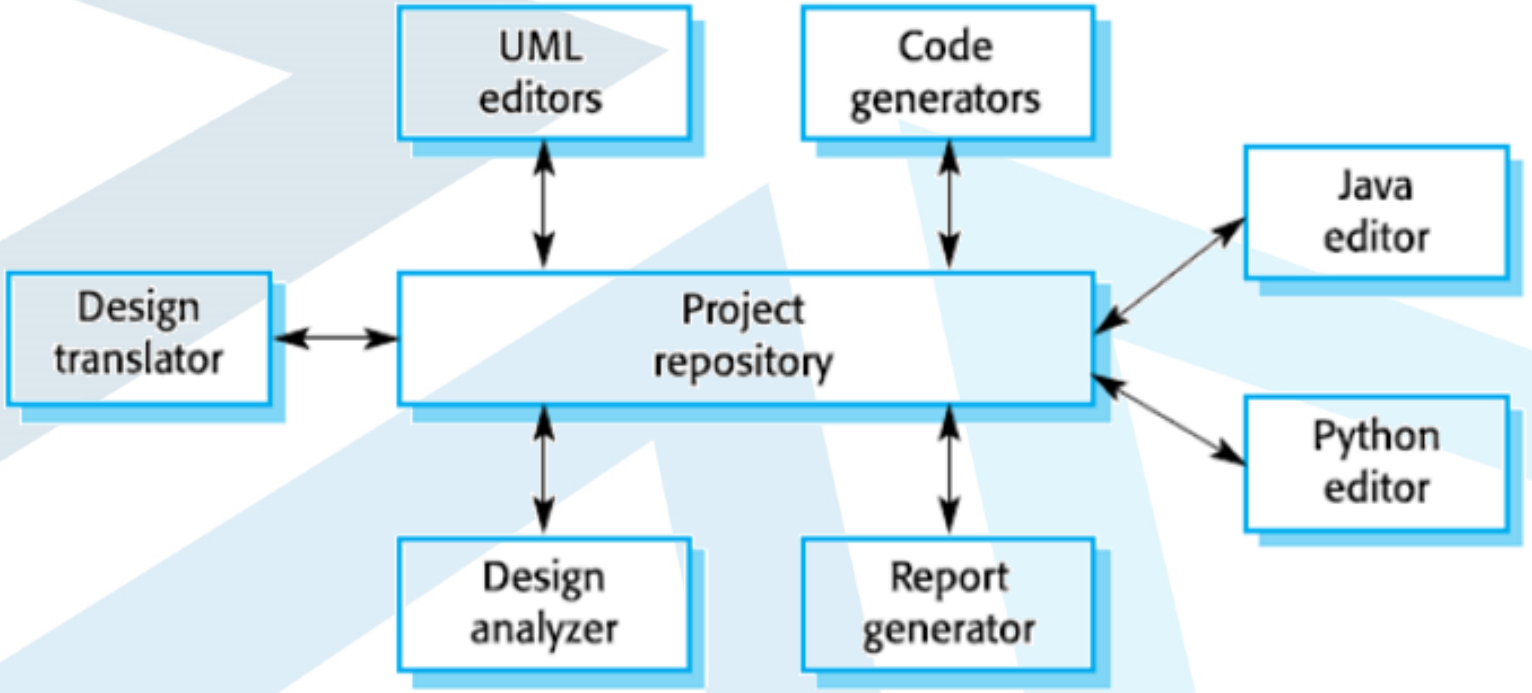
تستخدم هذه المعمارية عندما يكون لدينا data معينة مركزية و كل ال component و ال sub system تريد استخدام هذه ال data لذلك نقوم بوضعها بمخزن و نجعل جميع ال component تتصل مع هذا المخزن .

و يكون شكلها كالتالي :



مثال :

IDEs (بيئة التطوير المتكاملة) مصمّمة بطريقة الـ repository، حيث يمكن لتطبيقات منفصلة ومختلفة كالـ programming languages editors و compiler و graphical packages أن تصل لنفس الكود البرمجي الذي يتم العمل عليه، وهذا الكود هو الـ repository بالنّسبة لهذا النّظام.



-Components can be independent--they do not need to know of the existence of other components.

-All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.

-Changes made by one component can be propagated to all components.

الإيجابيات :

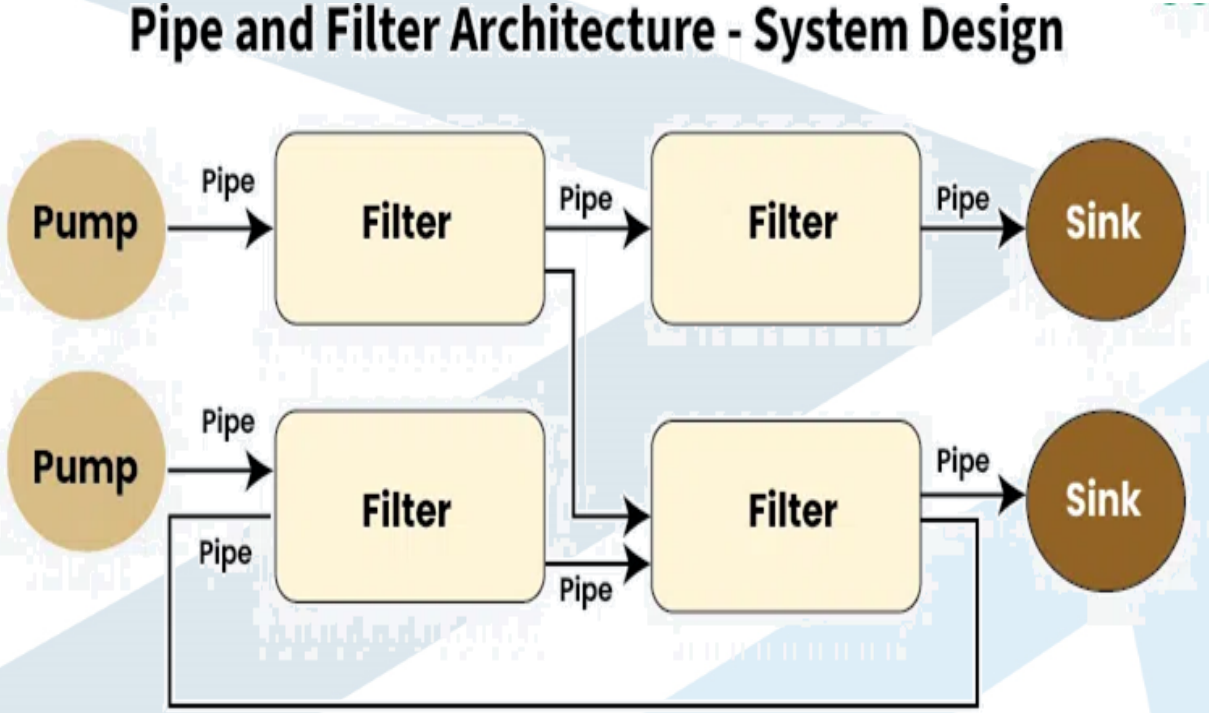
- عزل المكوّنات عن بعضها واستقلالها (كل نظام جزئي هو تطبيق بحدّ ذاته).
- مركزية المعلومات ممّا يسهّل عملية إدارة استعادة الـ data.
- التعديل الذي يقوم به أحد المكوّنات يصل لباقي المكوّنات.

السّلبيات :

- عطل المخزن سيؤدّي إلى إيقاف كل التطبيقات التي تعتمد عليه.

The repository is a single point of failure so problems in the repository affect the whole system

Pipe and filter architecture



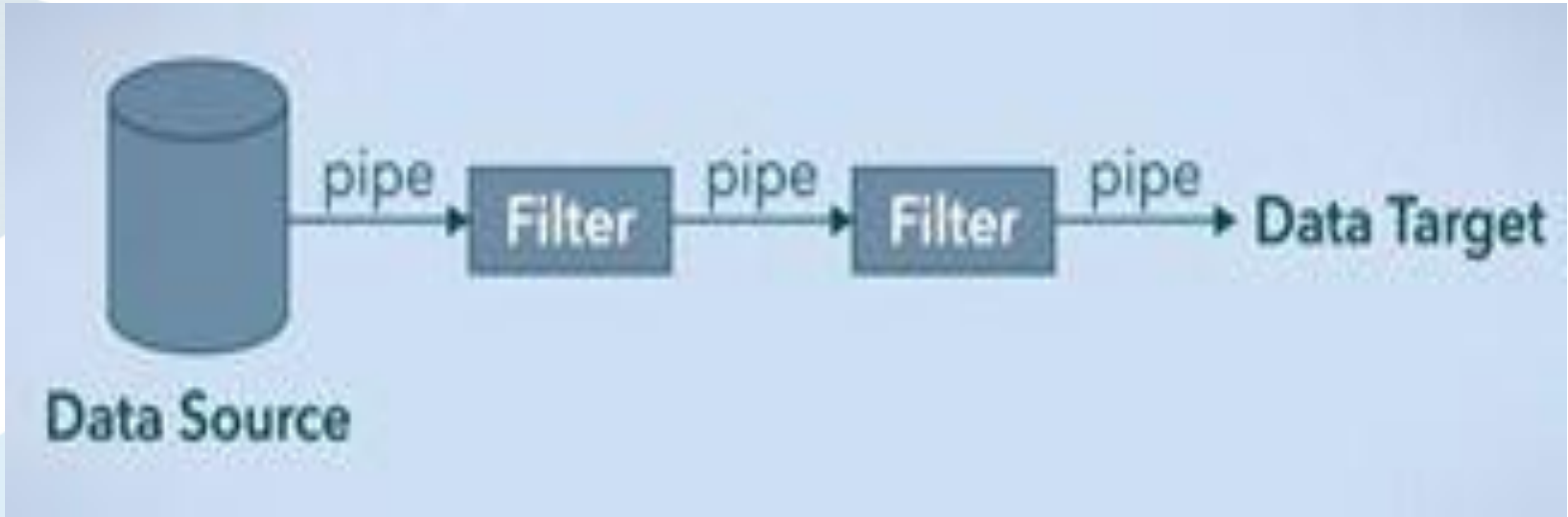
بنية **Pipe and filter architecture**

هي نمط تصميم يُنظّم الأنظمة بتقسيم العملية إلى سلسلة من خطوات المعالجة المستقلة والمتميزة، تُعرف باسم المرشحات(الفلاتر)، وتتصل هذه الفلاتر بقنوات تُسمى الأنابيب.

يؤدي كل مرشح مهمة واحدة محددة، مثل تحويل البيانات أو التحقق من صحتها أو تجميعها، ويعمل على معالجة تدفق بيانات الإدخال وانتاج مخرجات . تتميز المرشحات باستقلاليتها، حيث تستقبل المدخلات من أنبوب وارد وتنشر المخرجات إلى أنبوب صادر. يعزز هذا التصميم التجزئة الى وحدات نمطية(فلاتر)، حيث لا يتفاعل المرشح مع المرشحات الأخرى، انما يتفاعل فقط مع مخططات الإدخال والإخراج المباشرة الخاصة به، مما يسمح بمعاملتها كصناديق سوداء.

تُستخدم هذه البنية على نطاق واسع في أنظمة معالجة البيانات (data processing applications) وهو غير مُناسب للأنظمة التفاعلية.
يُعد ترتيب المرشحات أمرًا بالغ الأهمية، إذ يجب أن يتوافق مُخرج كل مرشح مع تنسيق الإدخال المُتوقع من المرشح التالي.

the processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.



Characteristics of Pipe and Filter Architecture مميزات بنية الأنابيب والمرشحات

The Pipe and Filter architecture in system design possesses several key characteristics that make it an effective and popular design pattern for many applications. Here are its main characteristics:

Modularity: كل مرشح هو مكون مستقل يؤدي مهمة محددة. يتيح هذا الفصل سهولة فهم المرشحات الفردية وتطويرها وصيانتها دون التأثير على النظام بأكمله.

Reusability قابلية إعادة الاستخدام: يمكن إعادة استخدام المرشحات عبر أنظمة مختلفة أو داخل أجزاء مختلفة من نفس النظام. وهذا يقلل من ازدواجية الجهود ويعزز الاستخدام الفعال للموارد.

Composability القابلية للتركيب: يمكن تكوين المرشحات بتسلسلات مختلفة لإنشاء خطوط أنابيب معالجة معقدة. تتيح هذه المرونة للمصممين إنشاء مسارات عمل مخصصة ببساطة عن طريق إعادة ترتيب المرشحات أو دمجها.

Scalability قابلية التوسع: تدعم البنية المعالجة المتوازية عن طريق تشغيل مثيلات متعددة من المرشحات. وهذا يعزز قدرة النظام على التعامل مع كميات أكبر من البيانات ويحسن الأداء.

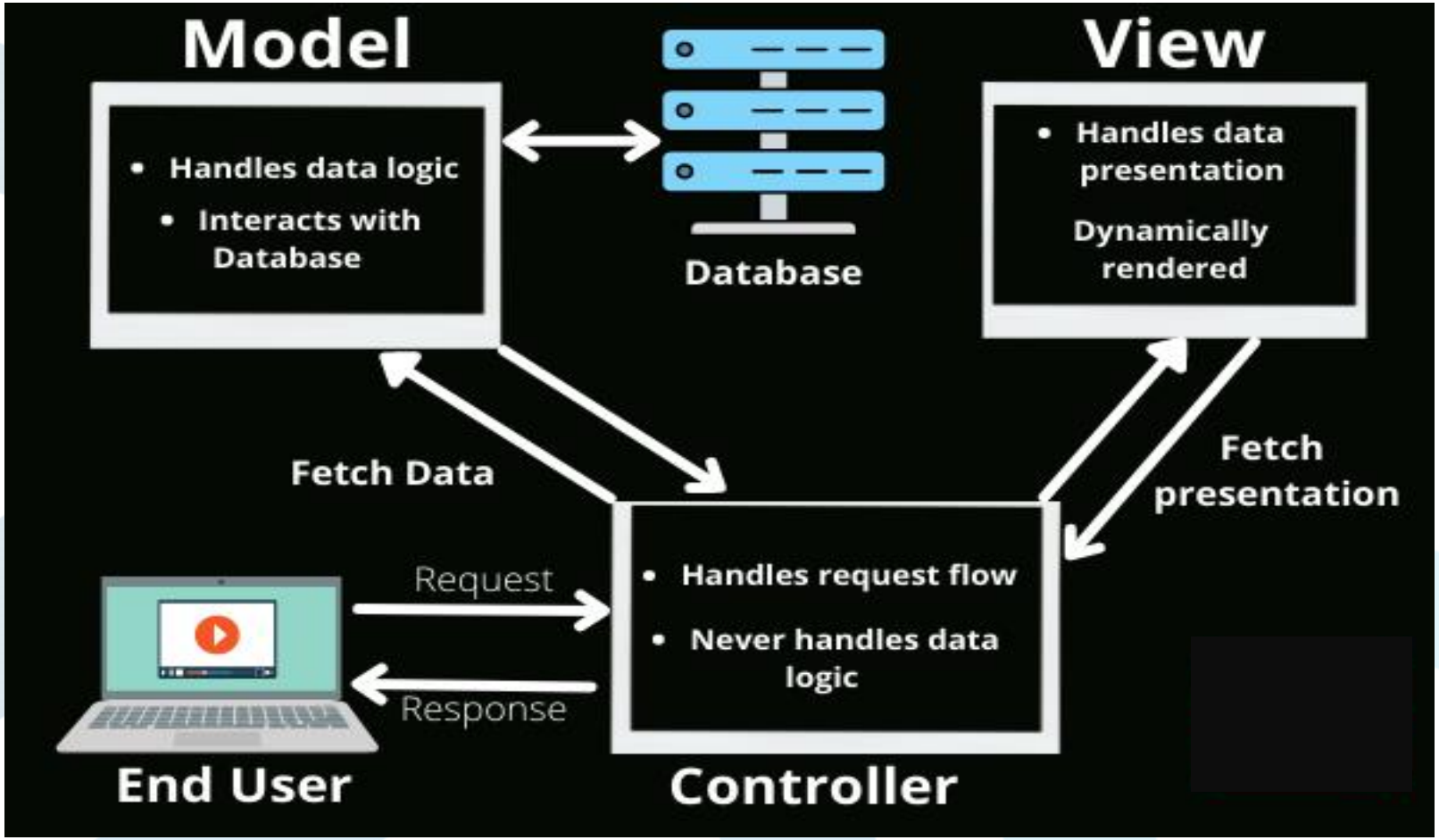
Maintainability قابلية الصيانة: يؤدي عزل الوظائف إلى مرشحات منفصلة إلى تبسيط عملية التصحيح والصيانة. لا تؤثر التغييرات التي يتم إجراؤها على مرشح واحد على المرشحات الأخرى، مما يجعل إدارة التحديثات وإصلاحات الأخطاء أسهل.

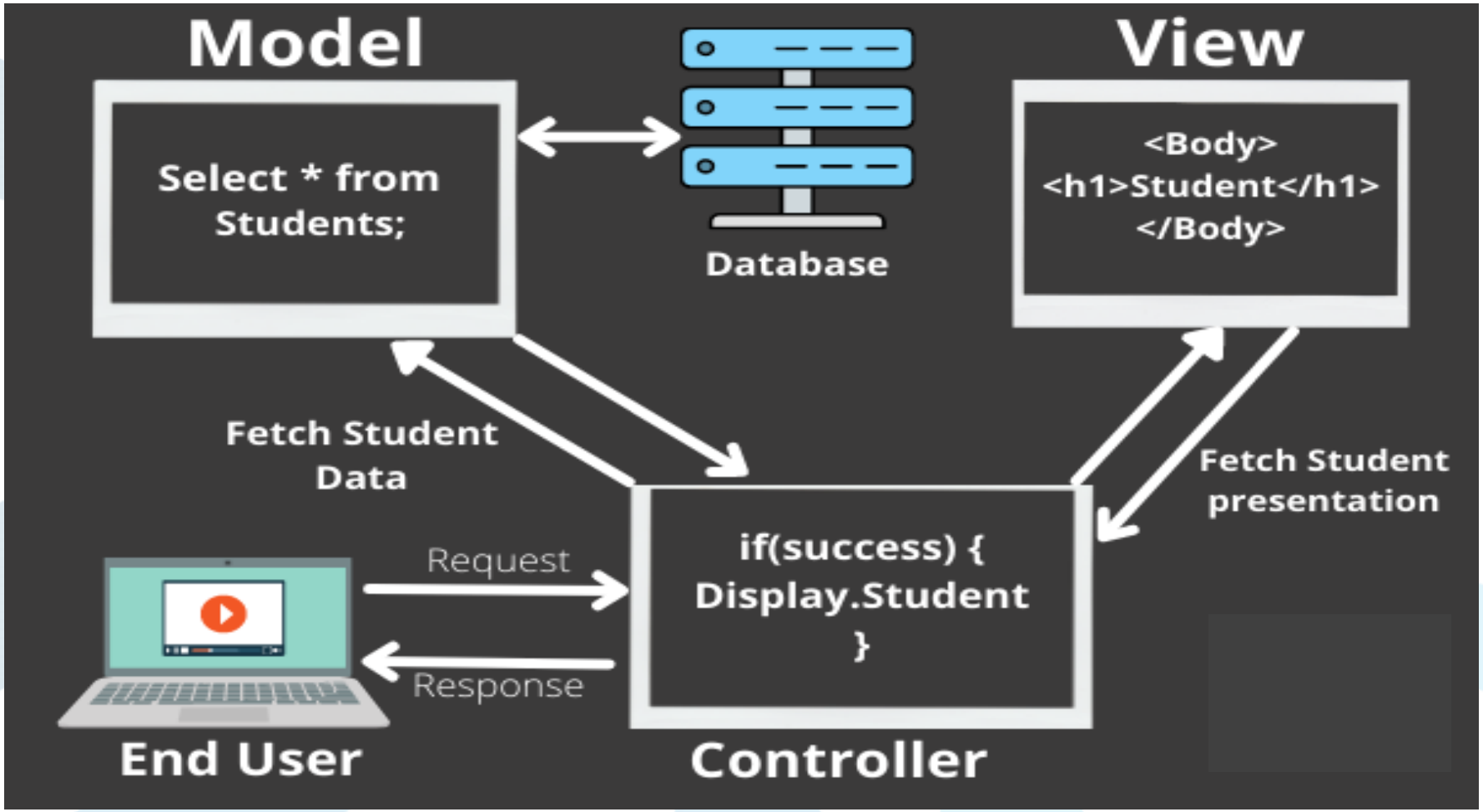
Model-View-Controller (MVC)

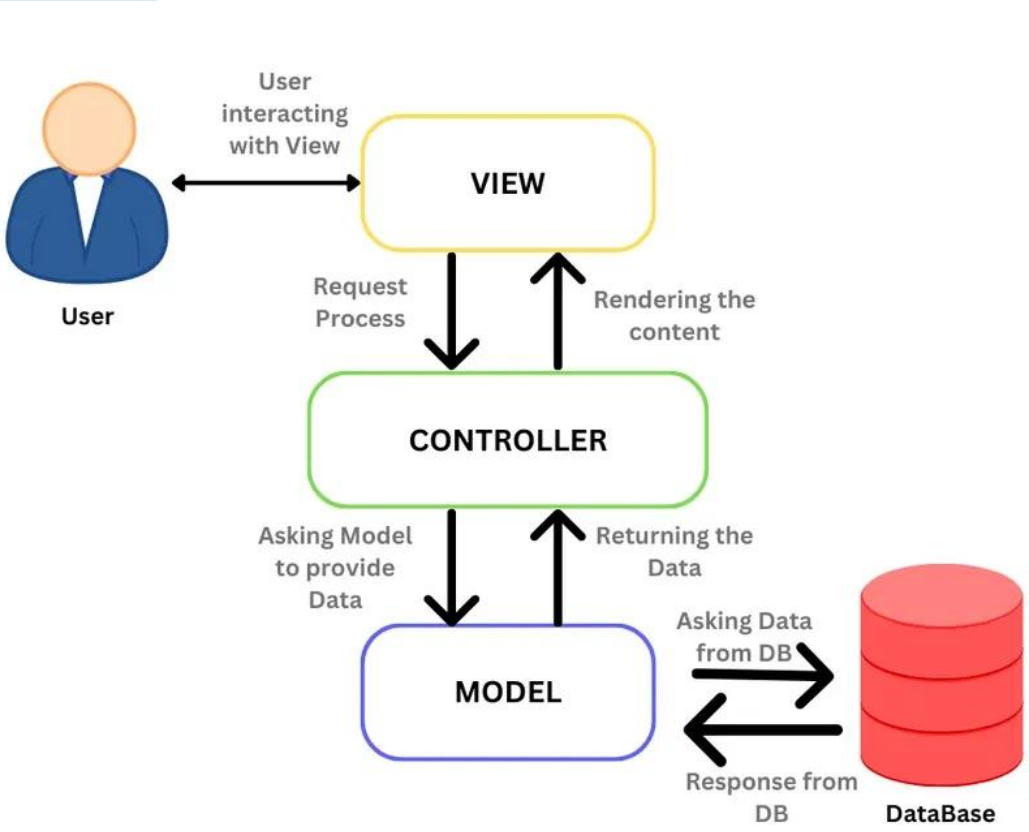
MVC is abbreviated as Model View Controller is a design pattern created for developing applications specifically web applications. As the name suggests, it has three major parts. The traditional software design pattern works in an "Input - Process - Output" pattern whereas MVC works as "Controller -Model - View" approach.

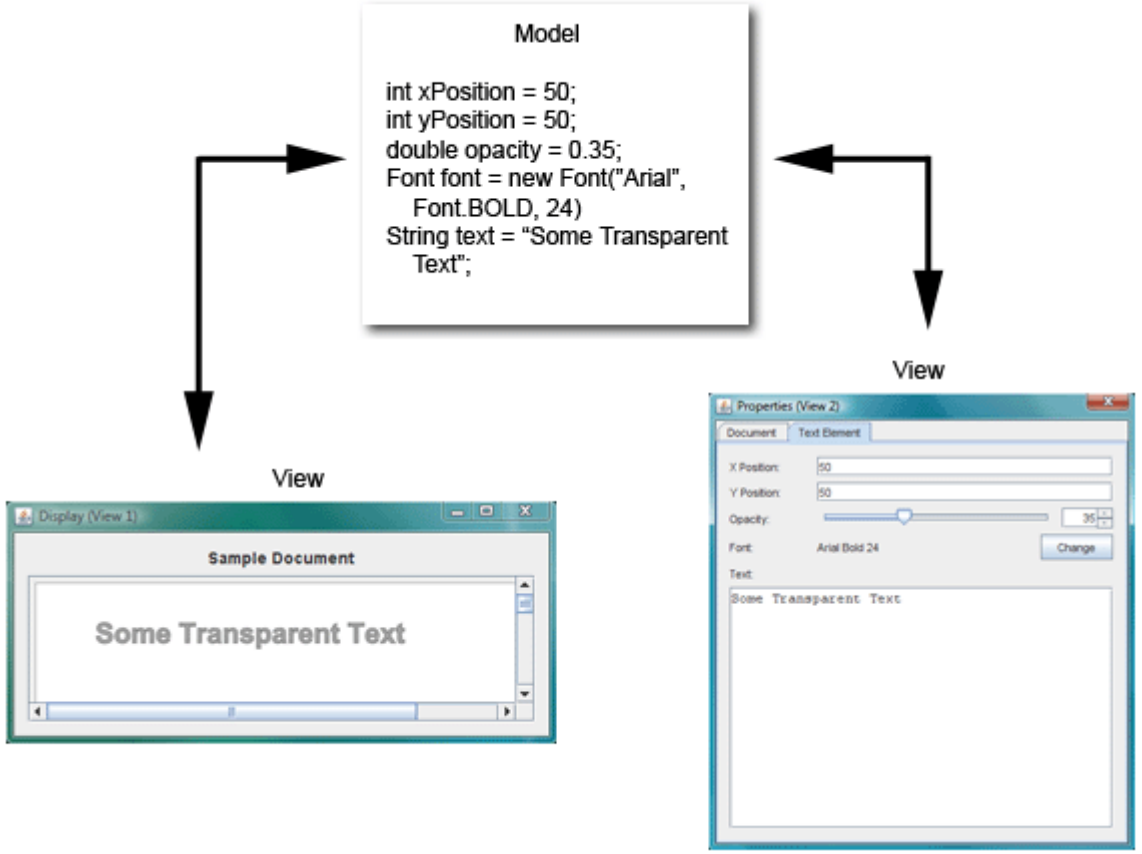
- **The Controller** acts as an intermediary between the Model and the View, receiving user input, processing it, and updating the Model or View accordingly. It routes commands to the Model and View, ensuring that user actions are properly handled and that the application state is updated
- **The View** The View element is used for presenting the data of the model to the user , displaying data to the user in a visual format such as charts, diagrams, or tables, and may also accept user input.
- **The Model** The central component of the pattern. The model is responsible for managing the data of the application. It receives user input from the controller. It directly manages the data, logic and rules of the application. It is responsible for maintaining the data and ensuring its consistency and integrity.

UI Logic
Input logic
Business Logic




Model
Select * from Students;
Database
View
<Body>
<h1>Student</h1>
</Body>
Fetch Student Data
Fetch Student presentation
Request
Response
if(success) {
Display.Student
}
End User
Controller

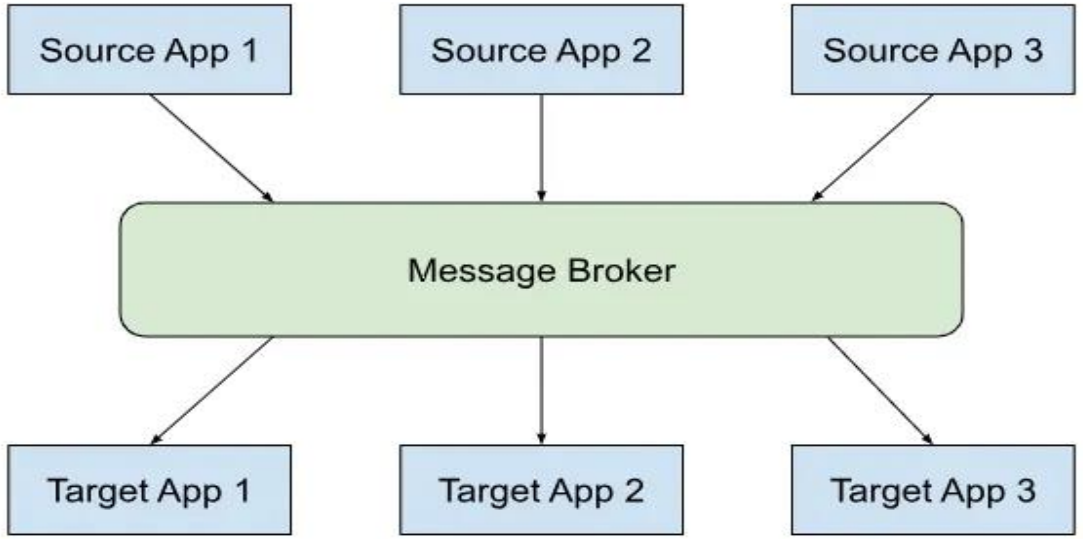




In MVC, models can have numerous views

Message Brokers in System Design

A message broker is a key architectural component responsible for facilitating communication and data exchange between different parts of a distributed system or between heterogeneous systems. It acts as an intermediary or middleware that receives messages from producers (senders) and delivers them to consumers (receivers) based on predefined routing rules and patterns. The broker acts as a "middleman" between the components, allowing them to communicate without being aware of each other's existence.



Terminology

Broker

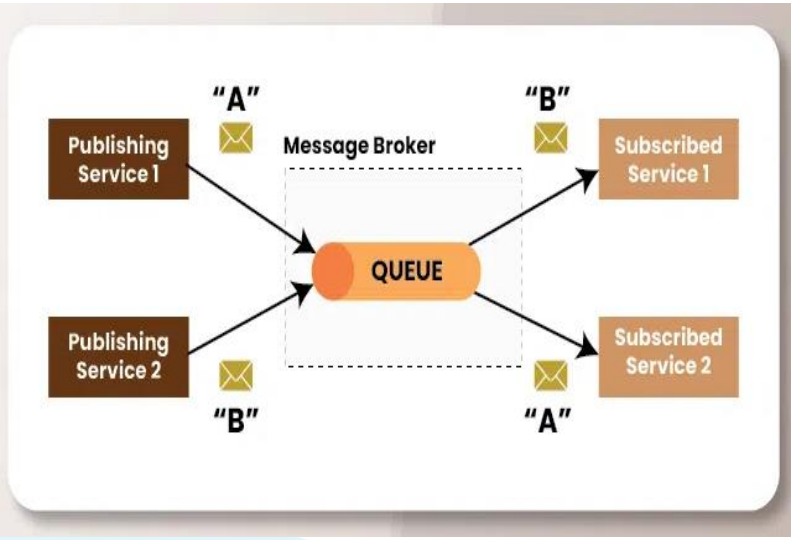
- Maintain a routing table of registered software components. And transiting messages to the right software components.
- May assure additional functionalities such as information security and quality of service.

Server

- Software components responsible for sending a message out.
- It is also referred to as a *publisher*.

Client

- Software components that subscribed and await a specific message.
- It can also be referred to as *a consumer* or *subscriber.*

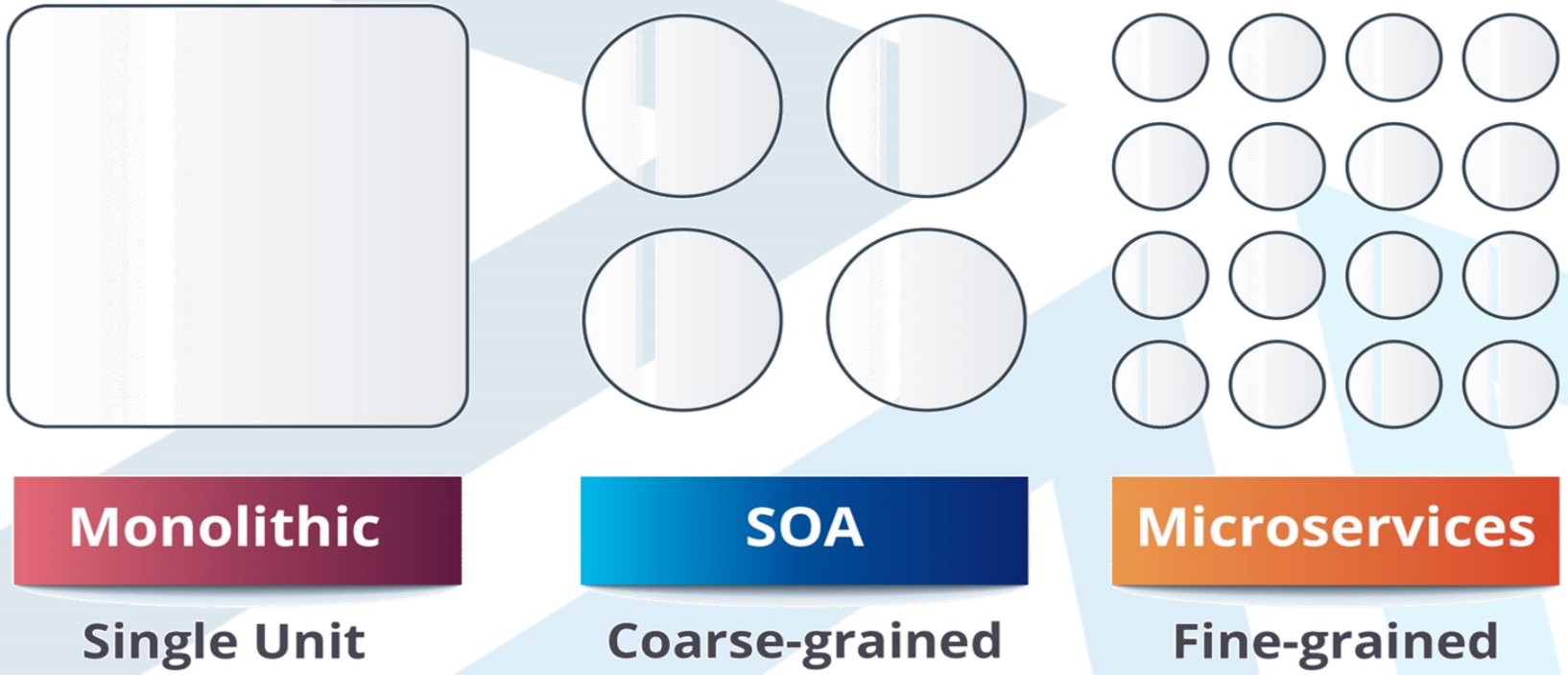


Advantages: Dynamic changes, additions, deletions, and relocations of components possible.

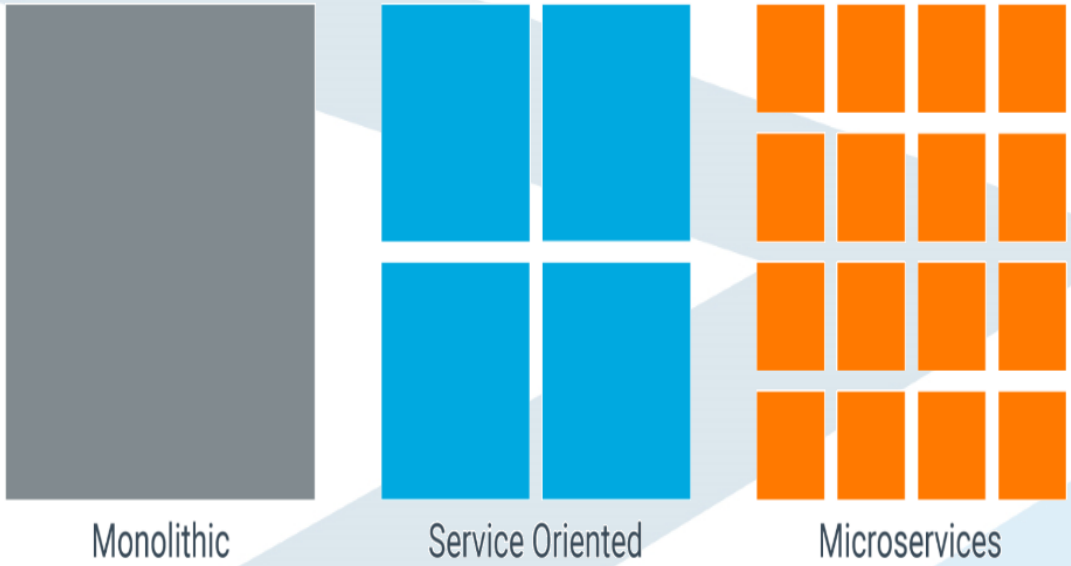
Components do not need to know each other: The broker pattern allows the components to remain decoupled and focused on their own responsibilities, while still being able to communicate and collaborate with other components in the system. It can also be used to reduce the number of dependencies between components, making the system more flexible and easier to maintain.

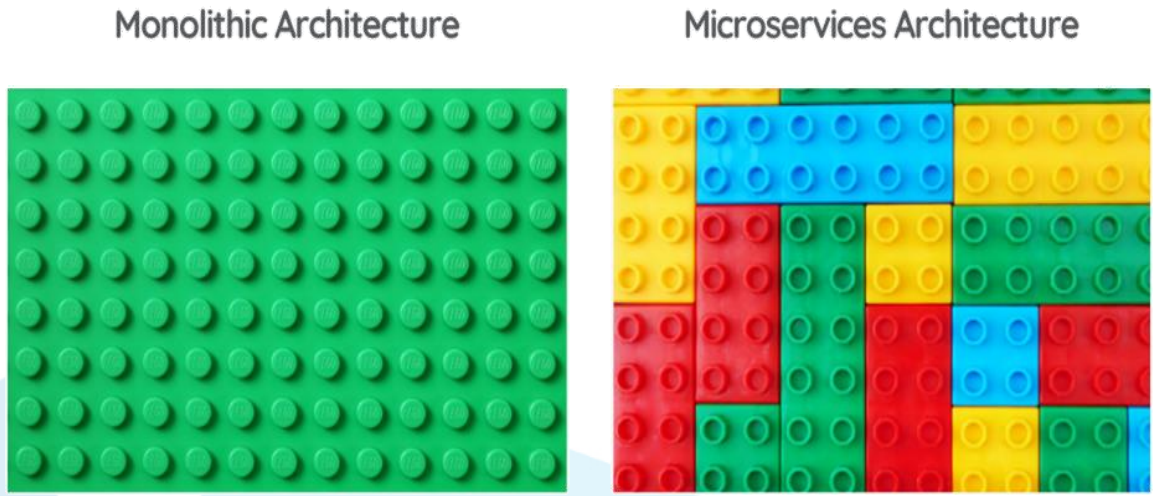
Disadvantages: One central component that needs to be robust and efficiently written.

Monolithic vs. SOA vs. Microservices

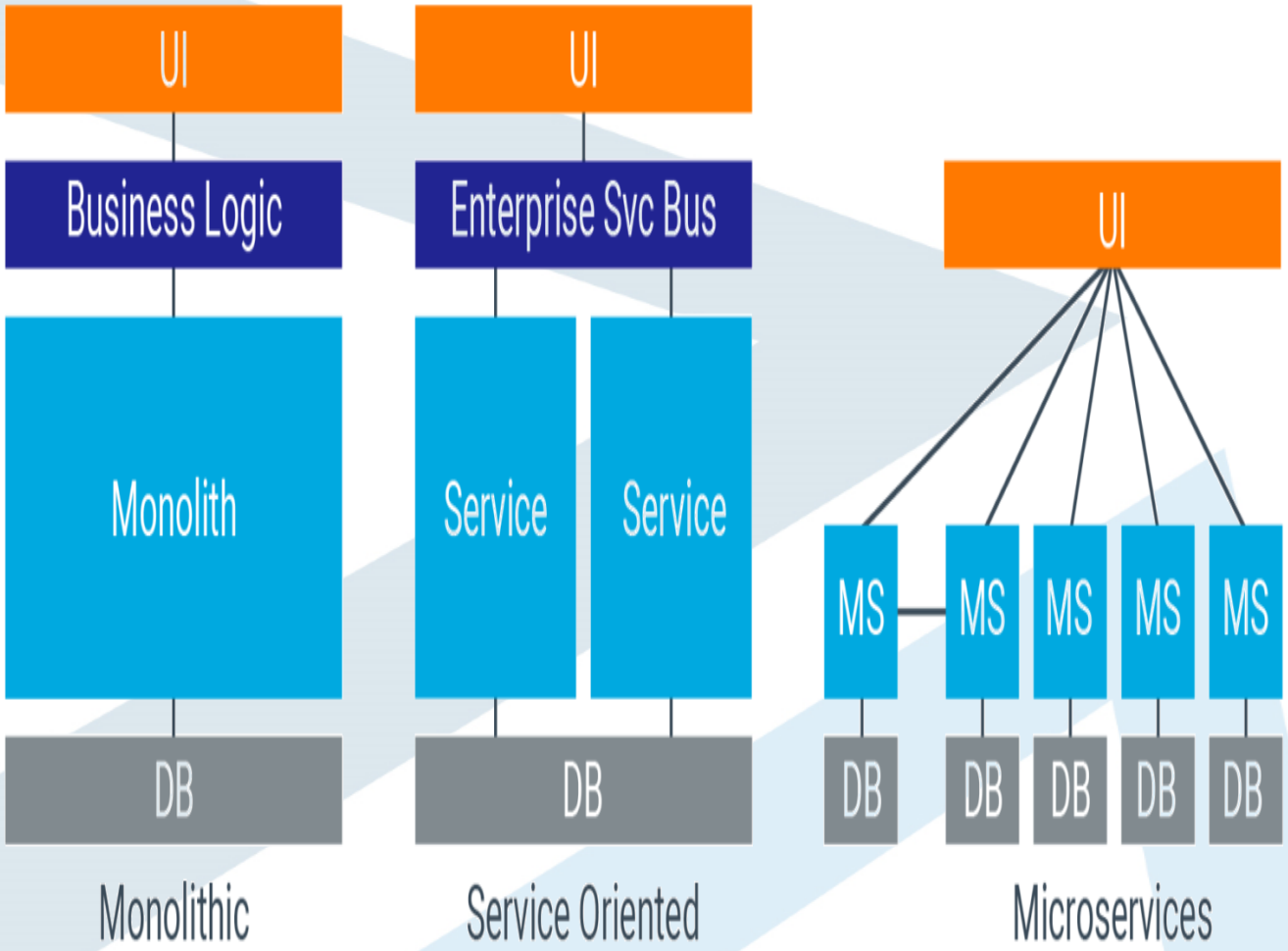


In software engineering, a **monolithic application** is a single unified software application that is self-contained and independent from other applications, but typically lacks flexibility. There are advantages and disadvantages of building applications in a monolithic style Monolith applications are relatively simple and have a low cost but their shortcomings are lack of elasticity, fault tolerance and scalability.Alternative styles to monolithic applications include multitier architectures, distributed computing and microservices. Monolithic architectures require the entire program to be recompiled and deployed every time a change is made





In software engineering, service-oriented architecture (SOA) is an architectural style that focuses on discrete services instead of a monolithic design. SOA is a good choice for system integration. In this architecture, services are provided to form applications, through a network call over the internet.



Microservices architecture is **a software engineering approach that structures an application as a collection of small, independent, and loosely coupled services, each designed to perform a specific business function.** Unlike monolithic architectures, where all components are tightly integrated into a single codebase, microservices break down applications into smaller, self-contained units that can be developed, deployed, and scaled independently.

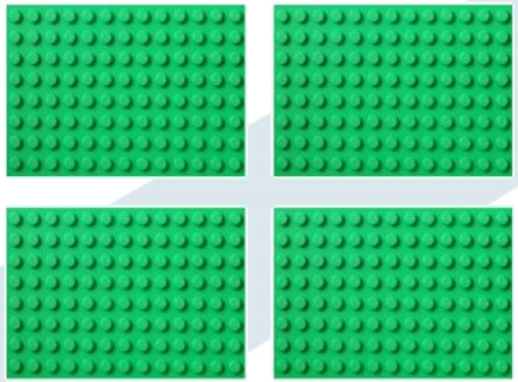
Each microservice typically runs in its own process and communicates with others through lightweight mechanisms such as REST APIs.

مثال:

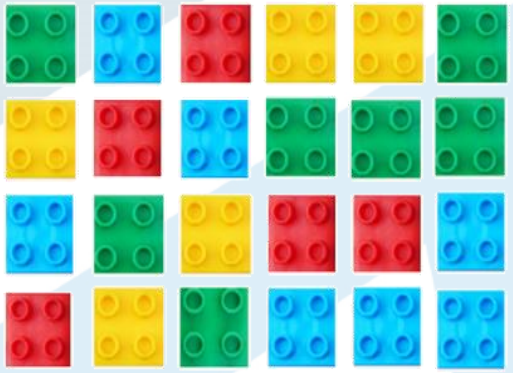
لو كان تطبيقنا من وظائفه الغير وظيفية ال availability ماذا نفعل؟؟

المتطلب غير الوظيفي availability تعني أنه في حال حدوث أي عطل في component لا يجب على الخدمة أن تتوقف عن التنفيذ بل يجب أن يكون تطبيقنا متاح دائما و لذلك يجب أن يتم طلب الخدمة من component ثاني أي يجب أن يصبح لدينا كل component مكرر مرتين أي عدد ال component أكبر

لو أردنا maintainability (سهولة صيانة) نستخدم ال fine grain .



Scalability in
Monolithic Architecture



Scalability in
Microservices Architecture

وبعد مرحلة التصميم المعماري هناك مرحلة **التصميم التفصيلي** وهي عبارة عن الدخول إلى كل الـ Sub System التي ظهرت معنا في مرحلة التصميم المعماري ونضع الخوارزميات للوظائف التي تقدمها الـModels التي تكون المجتزئ الواحد Sub System ، بمعني آخر هي:

1) كتابة الخوارزميات لتنجيز الخدمات التي يقدمها sub System معين (وهذه المرحلة لن نتحدث عنها)
2) تصميم قاعدة المعطيات أو مخططات الـ ERD للنظام : حيث معظم التطبيقات وخاصة Information System لها قاعدة معطيات لتخزين المعلومات واستخراج المعلومات (وأيضاً لن ندخل بهذه المرحلة)
3) تصميم واجهات النظام


HAVE ANY
QUESTIONS AND DOUBTS