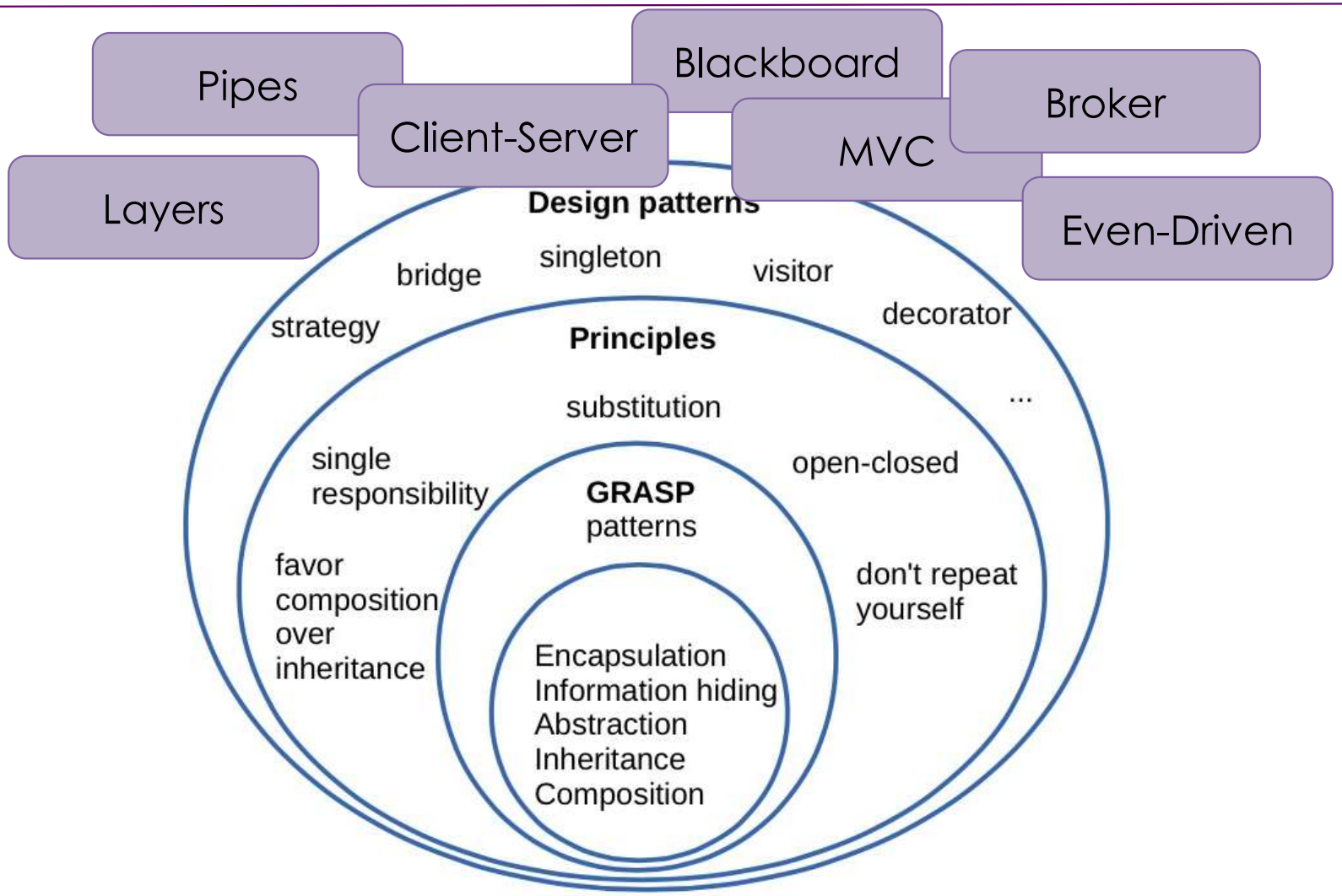
Software Architecture Patterns

UA.DETI.PDS
José Luis Oliveira

Resources

- Software Architecture Patterns, Mark Richards, O'Reilly Media, Inc., 2015
  - http://www.oreilly.com/programming/free/files/software-architecture-patterns.pdf

Software Architecture Patterns?



Software Architecture Patterns

- Layered Architecture
- Event-Driven Architecture
- Microkernel Architecture
- Microservices Architecture Pattern
- Space-Based Architecture

Software Architecture Patterns

- **Layered Architecture**
- Event-Driven Architecture
- Microkernel Architecture
- Microservices Architecture Pattern
- Space-Based Architecture

Layered Architecture

- The most common architecture pattern is the layered architecture pattern, otherwise known as the **n-tier architecture pattern**.

Layered Architecture

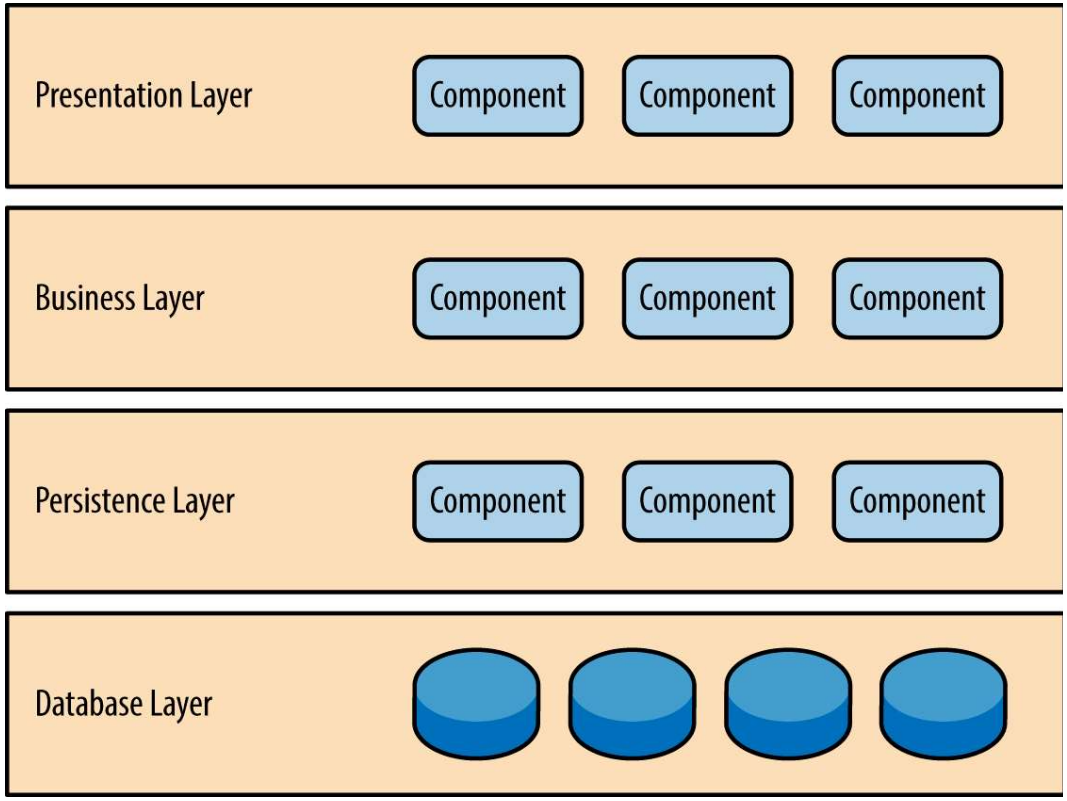
- The most common architecture pattern is the layered architecture pattern, otherwise known as the **n-tier architecture pattern**.
- This pattern is the *de facto* standard for most Java EE applications and therefore is widely known by most architects, designers, and developers.

Layered Architecture

- The most common architecture pattern is the layered architecture pattern, otherwise known as the **n-tier architecture pattern**.
- This pattern is the *de facto* standard for most Java EE applications and therefore is widely known by most architects, designers, and developers.
- The layered architecture pattern **closely matches the traditional IT communication and organizational structures** found in most companies
  - making it a natural choice for most business application development efforts.

Pattern Description

- Most layered architectures consist of four standard layers:
  - presentation, business, persistence, and database



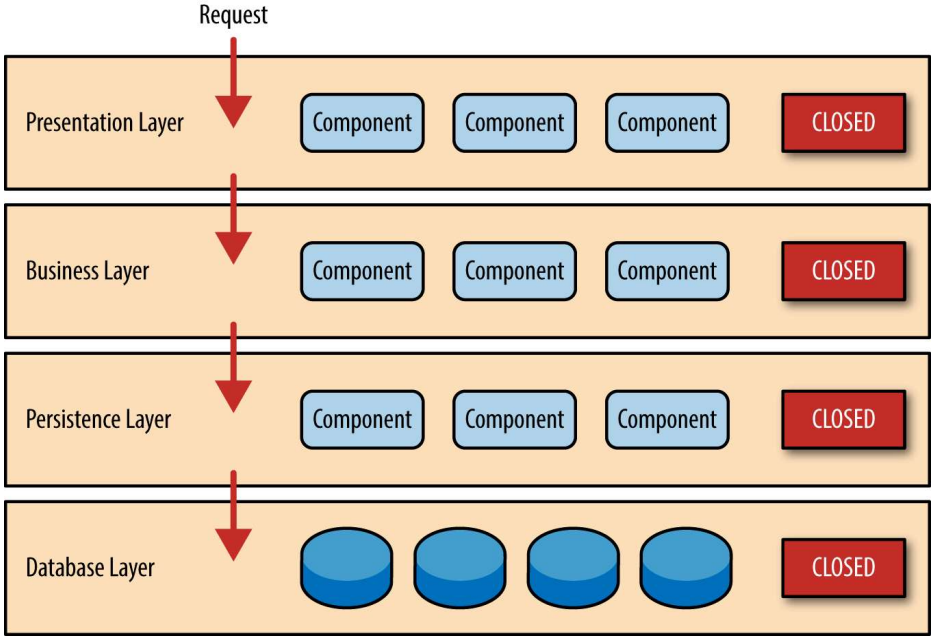
Pattern Description

- Most layered architectures consist of four standard layers:
  - presentation, business, persistence, and database

- One of the powerful features of the layered architecture pattern is the **separation of concerns** among components.
  - Each layer of the layered architecture pattern has a specific role and responsibility within the application.
  - Components within a specific layer deal only with logic that pertains to that layer.

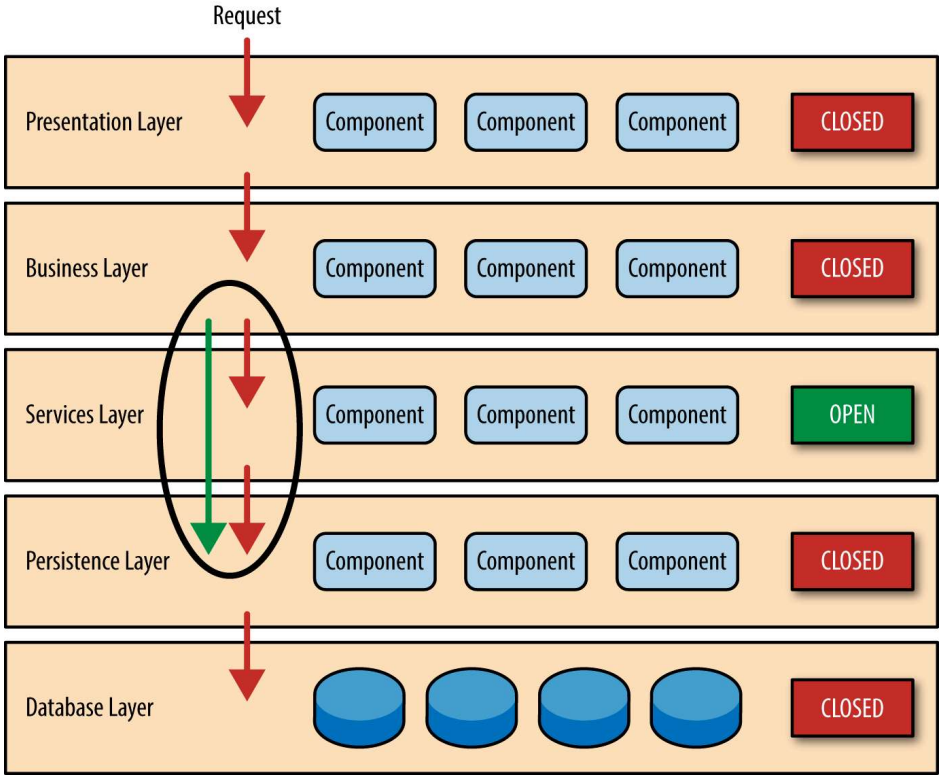
Key Concepts: Closed layers

- Request moves from layer to layer
  - it must go through the layer right below it to get to the next layer below that one - layers of isolation.
- The **layers of isolation** concept means that changes made in one layer of the architecture generally don't impact or affect components in other layers.

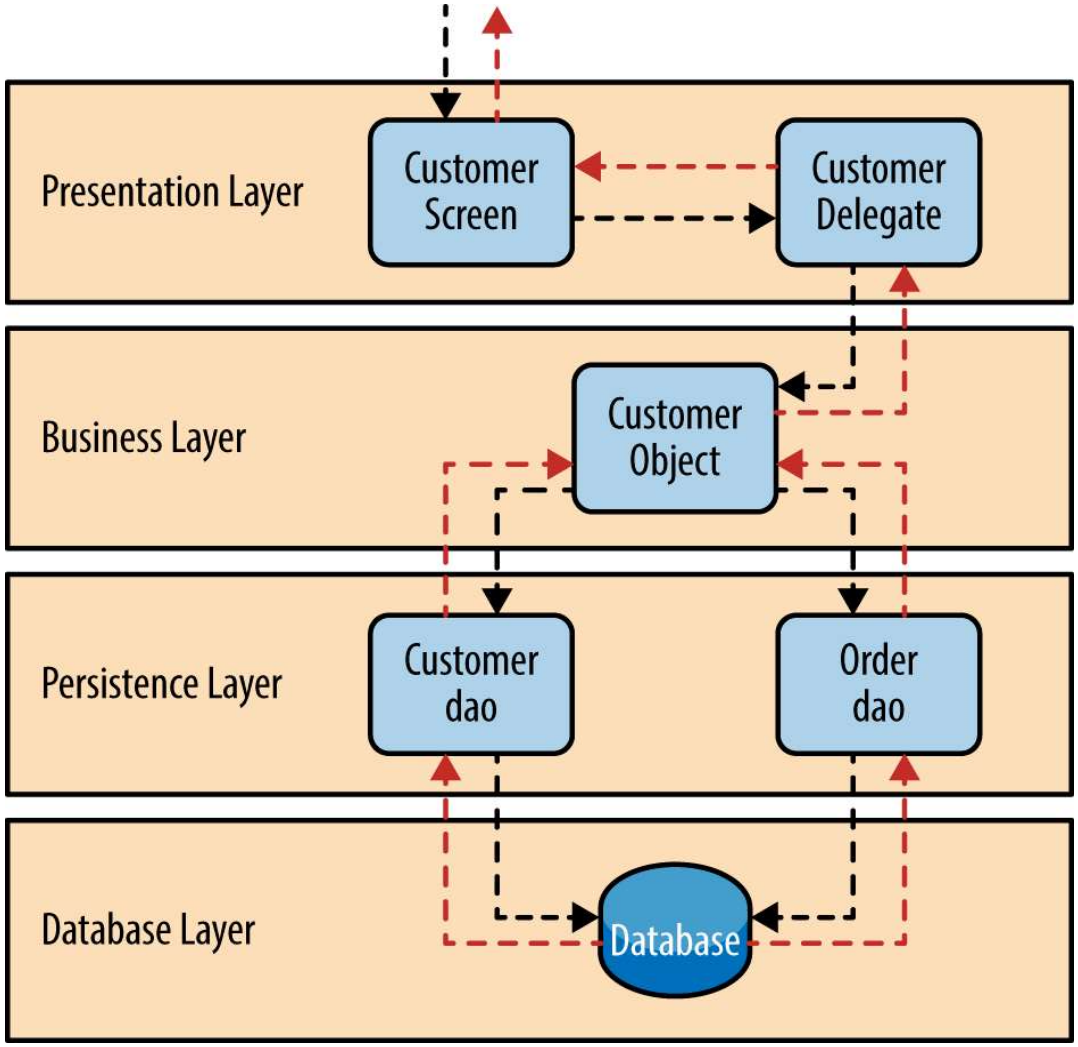


Key Concepts: Open layers

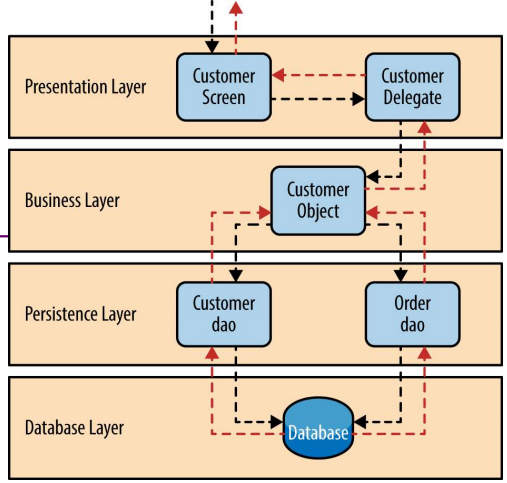
- Since the services layer is open, the business layer is now allowed to bypass it and go directly to the persistence layer



Pattern Example 1

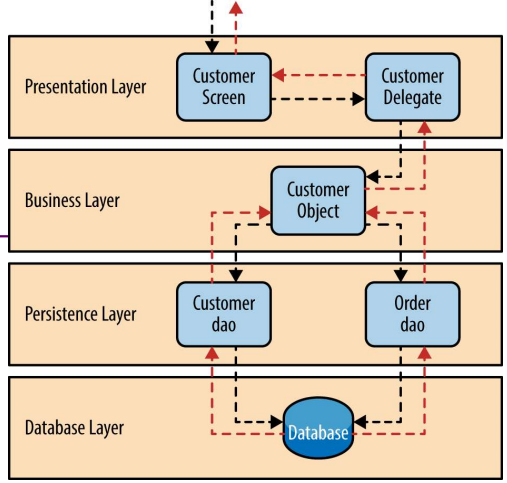


Pattern Example 1



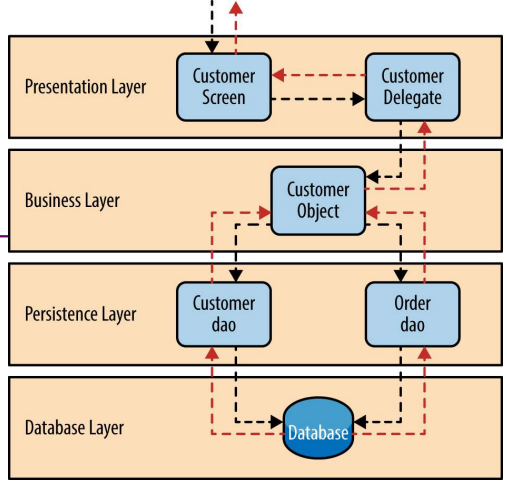
- The **customer screen** is responsible for accepting the request and displaying information.
  - does not know where the data is, how it is retrieved, which database tables must be queried
  - it forwards the request onto the **customer delegate** module.
  - This module is responsible for knowing which modules in the business layer can process that request

Pattern Example 1



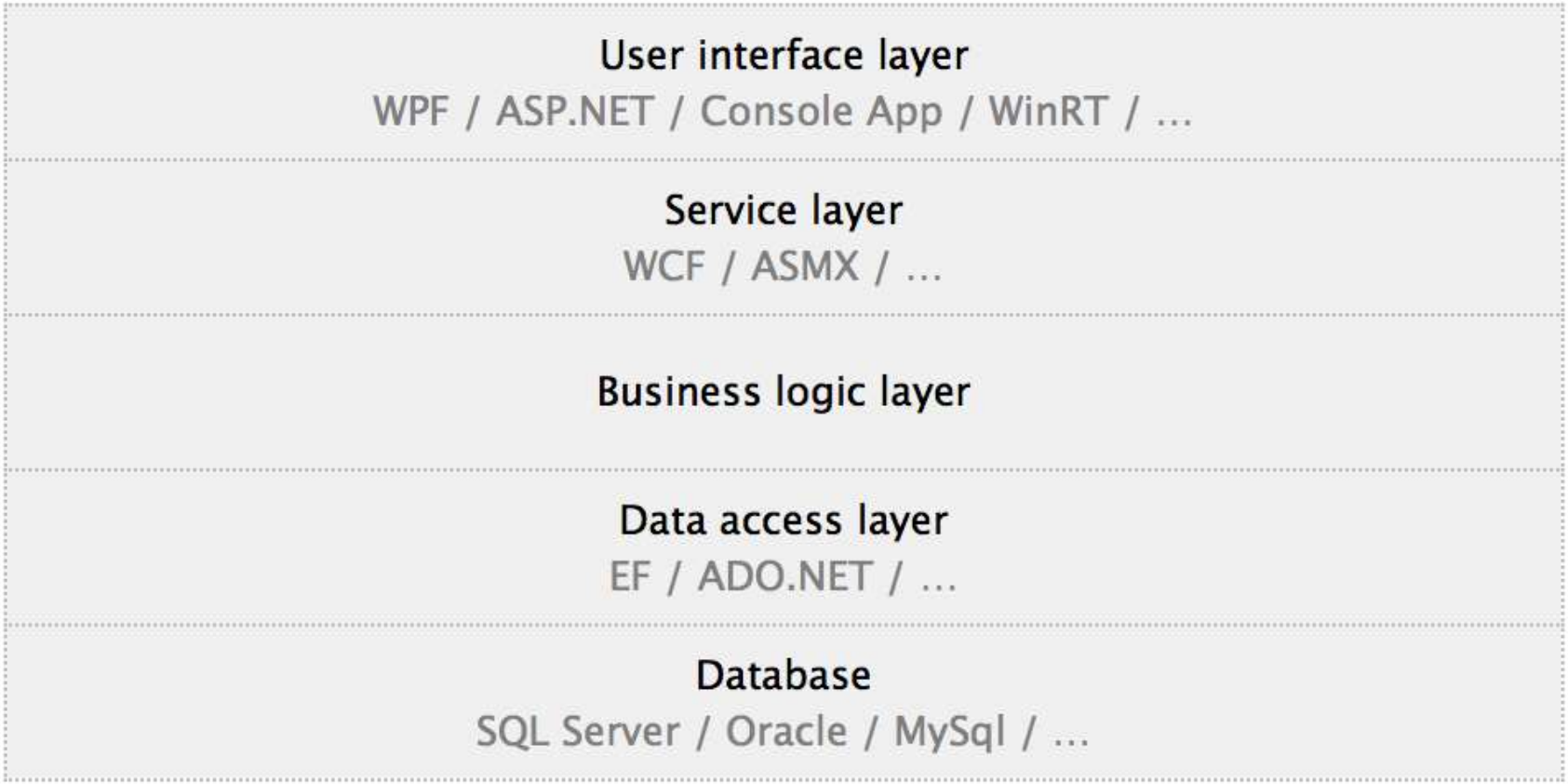
- The **customer screen** is responsible for accepting the request and displaying information.
  - does not know where the data is, how it is retrieved, which database tables must be queried
  - it forwards the request onto the **customer delegate** module.
  - This module is responsible for knowing which modules in the business layer can process that request
- The **customer object** is responsible for aggregating all of the information needed by the business request.
  - This module calls out to the **customer dao** (data access object) module in the persistence layer to get customer data, and also the **order dao** module to get order information.

Pattern Example 1



- The **customer screen** is responsible for accepting the request and displaying information.
  - does not know where the data is, how it is retrieved, which database tables must be queried
  - it forwards the request onto the **customer delegate** module.
  - This module is responsible for knowing which modules in the business layer can process that request
- The **customer object** is responsible for aggregating all of the information needed by the business request.
  - This module calls out to the **customer dao** (data access object) module in the persistence layer to get customer data, and also the **order dao** module to get order information.
- These modules in turn execute SQL statements to retrieve the corresponding data and pass it back up to the customer object in the business layer.
  - Once the customer object receives the data, it aggregates the data and passes that information back up to the customer delegate, which then passes that data to the customer screen to be presented to the user.

Pattern Example 2



Considerations

- Solid general-purpose pattern, making it a good starting point for most applications.
- But:
- Watch out for it what is known as the architecture *sinkhole anti-pattern*.
  - the situation where requests flow through multiple layers of the architecture as simple pass-through processing with little or no logic performed within each layer.
  - The 80-20 rule (20% of pass-through processes, at most) is usually a good practice to follow to determine whether or not you are experiencing the architecture *sinkhole anti-pattern*.

Pattern Analysis

- Overall agility
  - Rating: Low
- Ease of deployment
  - Rating: Low
- Testability
  - Rating: High
- Performance
  - Rating: Low
- Scalability
  - Rating: Low
- Ease of development
  - Rating: High

Software Architecture Patterns

- Layered Architecture
- **Event-Driven Architecture**
- Microkernel Architecture
- Microservices Architecture Pattern
- Space-Based Architecture

Event-Driven Architecture

- Popular distributed asynchronous architecture pattern used to produce highly scalable applications.
  - It is also highly adaptable and can be used for small applications and as well as large, complex ones.
  - Also referred to as ***message-driven*** *architecture* or ***stream processing*** *architecture*

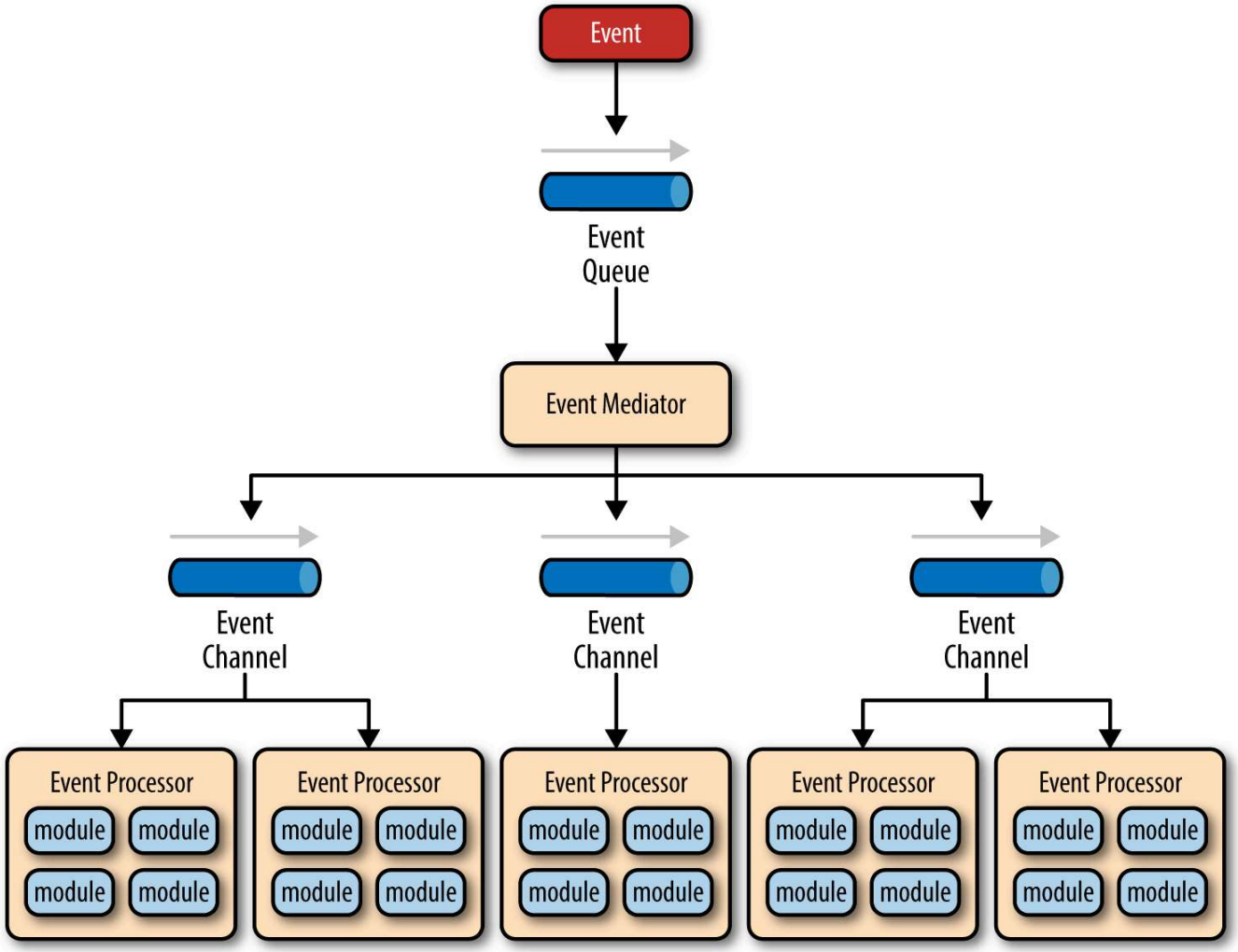
Event-Driven Architecture

- Popular distributed asynchronous architecture pattern used to produce highly scalable applications.
  - It is also highly adaptable and can be used for small applications and as well as large, complex ones.
  - Also referred to as ***message-driven*** *architecture* or ***stream processing*** *architecture*
- It is made up of highly decoupled, single-purpose event processing components that asynchronously receive and process events.
- The event-driven architecture pattern consists of two main topologies, the **mediator** and the **broker**.

Mediator Topology

- The mediator topology is useful for events that have multiple steps and require **some level of orchestration** to process the event.
  - For example, a single event to place a stock trade might require you to first validate the trade, then check the compliance of that stock trade against various compliance rules, assign the trade to a broker, calculate the commission, and finally place the trade with that broker
- There are four main types of architecture components within the mediator topology:
  - event queues, an event mediator, event channels, and event processors.

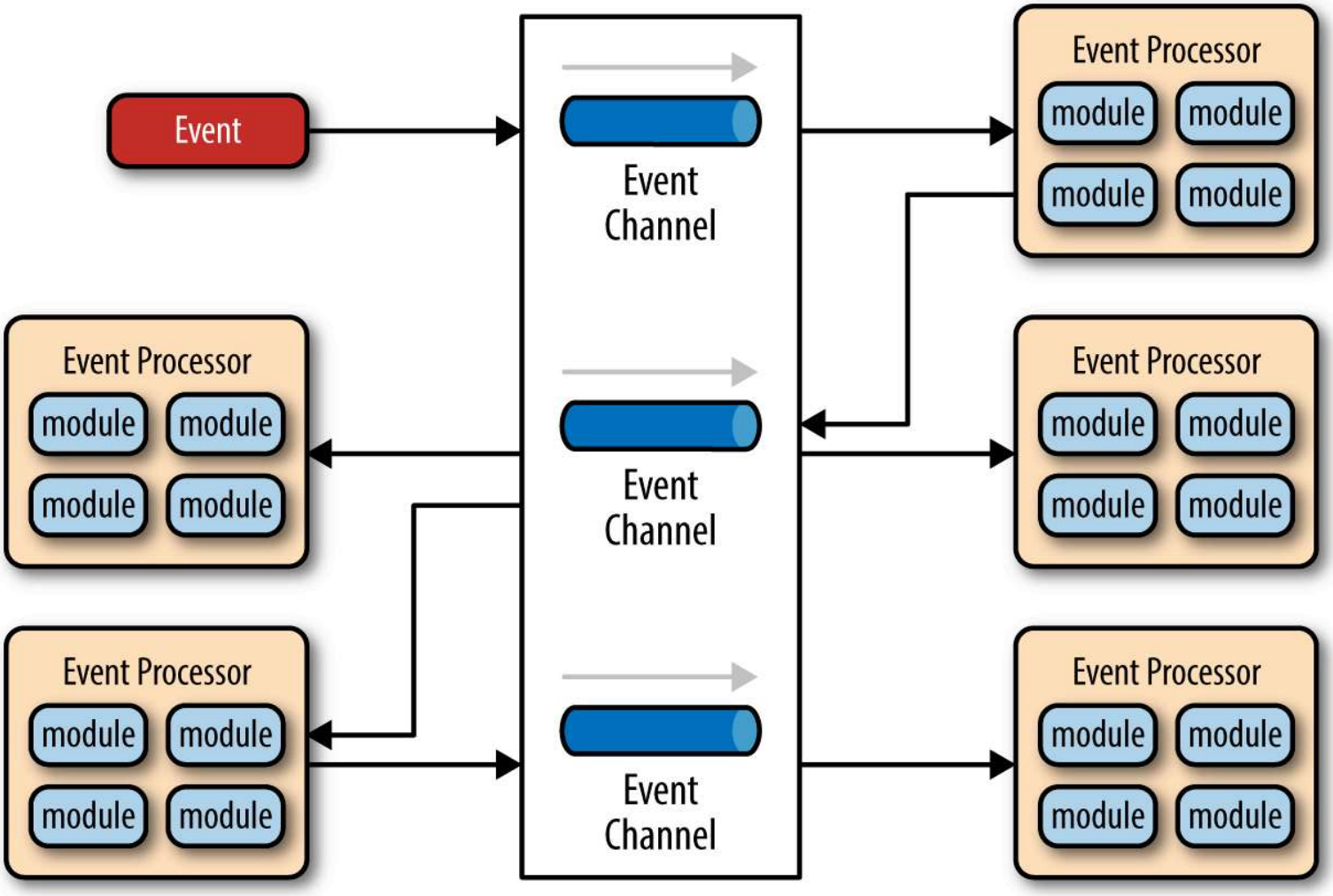
Mediator Topology



Broker Topology

- There is no central event mediator
  - the message flow is distributed across the event processor components in a chain-like fashion through **a lightweight message broker**.
- This topology is useful when you have a relatively **simple event processing flow** and you do not want (or need) central event orchestration.
- There are two main types of architecture components within the broker topology:
  - a broker component and an event processor component.

Broker Topology



Considerations

- The event-driven architecture pattern is a relatively **complex** pattern to implement, primarily due to its asynchronous distributed nature.
- **Lack of atomic transactions** for a single business process.
  - Event processor components are highly decoupled and distributed,
  - it is very difficult to maintain a transactional unit of work across them.
- A key aspect is the **creation, maintenance, and governance** of the event-processor component contracts.

Pattern Analysis

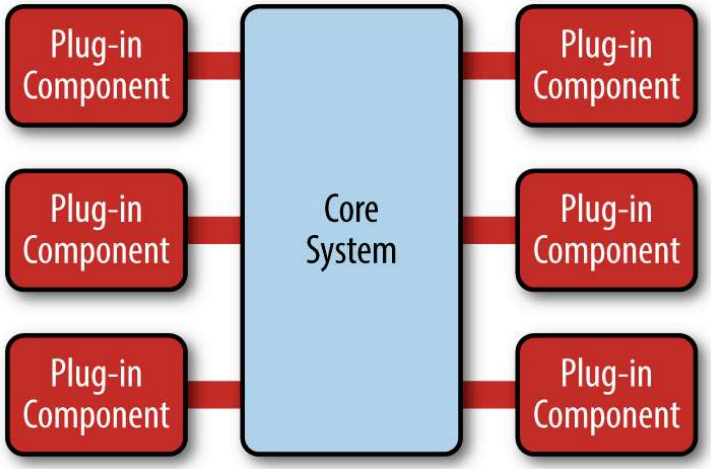
- Overall agility
  - Rating: High
- Ease of deployment
  - Rating: High
- Testability
  - Rating: Low
- Performance
  - Rating: High
- Scalability
  - Rating: High
- Ease of development
  - Rating: Low

Software Architecture Patterns

- Layered Architecture
- Event-Driven Architecture
- **Microkernel Architecture**
- Microservices Architecture Pattern
- Space-Based Architecture

Microkernel Architecture

- The microkernel architecture pattern allows you to add additional **application features as plug-ins** to the **core** application, providing extensibility as well as feature separation and isolation.
- It is also referred to as the ***plug-in architecture pattern*** and it is a natural pattern for implementing product-based applications.



- Many **operating systems** implement the microkernel architecture pattern, hence the origin of this pattern's name.

Pattern Description

- Two types of architecture components:
  - a core system and plug-in modules
- The **core system** traditionally contains only the minimal functionality required to make the system operational
- **Plug-in modules** can be connected to the core system through a variety of ways
  - OSGi (open service gateway initiative), messaging, web services, or even direct point-to-point binding (i.e., object instantiation)

Considerations

- One great thing about the microkernel architecture pattern is that it can be embedded or used as part of another architecture pattern.
- Provides great support for evolutionary design and incremental development.

Considerations

- One great thing about the microkernel architecture pattern is that it can be embedded or used as part of another architecture pattern.
- Provides great support for evolutionary design and incremental development.
- For product-based applications it should always be the first choice as a starting architecture
  - particularly for those products where we will be releasing additional features over time and want control over which users get which features.
  - we can always refactor the application to another architecture pattern better suited for your specific requirements.

Pattern Analysis

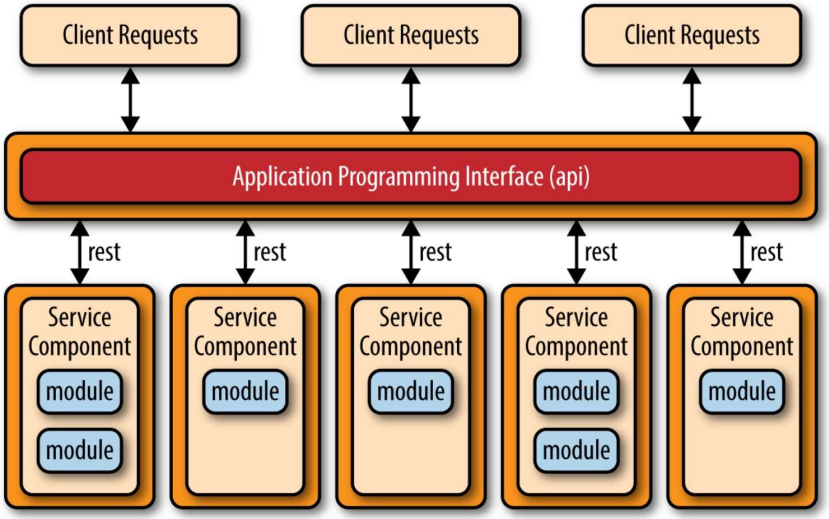
- Overall agility
  - Rating: High
- Ease of deployment
  - Rating: High
- Testability
  - Rating: High
- Performance
  - Rating: High
- Scalability
  - Rating: Low
- Ease of development
  - Rating: Low

Software Architecture Patterns

- Layered Architecture
- Event-Driven Architecture
- Microkernel Architecture
- **Microservices Architecture Pattern**
- Space-Based Architecture

Microservices Architecture Pattern

- Large and complex applications are composed of one or more smaller services.
- It is gaining ground in the industry as a viable alternative to monolithic applications and service-oriented architectures.



Pattern Description

- The microservices architecture style naturally evolved from two main sources:
  - monolithic applications developed using the layered architecture pattern and
  - distributed applications developed through the service-oriented architecture pattern.
- The first characteristic is the notion of separately deployed units.
  - Each **service component** is deployed as a separate unit, allowing for easier deployment and decoupling.
  - from a single module to a large portion of the application.
- Distributed architecture
  - all the components are fully decoupled
  - communication through JMS, AMQP, REST, SOAP, RMI, etc.

Key characteristics of a service

- Highly maintainable and testable
  - enables rapid and frequent development and deployment
- Loosely coupled with other services
  - enables a team to work independently the majority of time on their service(s) without being impacted by changes to other services and without affecting other services
- Independently deployable
  - enables a team to deploy their service without having to coordinate with other teams
- Capable of being developed by a small team
  - essential for high productivity by avoiding the high communication overhead of large teams

Microservices are …

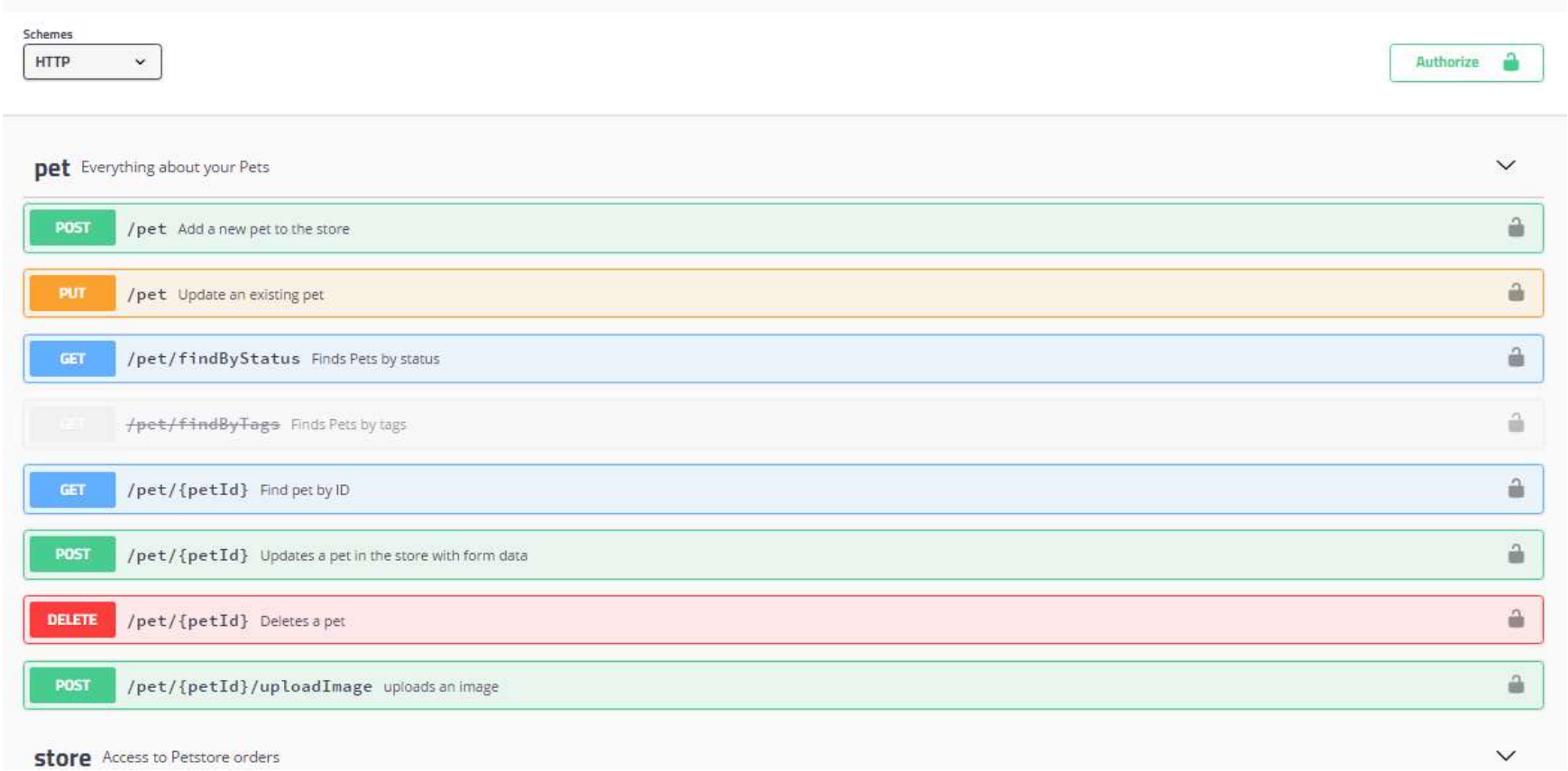
Loosely Coupled

Small & Focused

Language Neutral

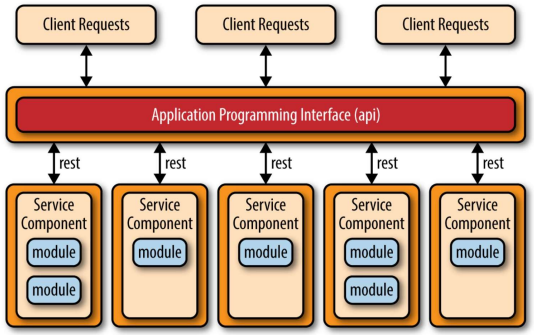
Bounded Context

Microservices examples

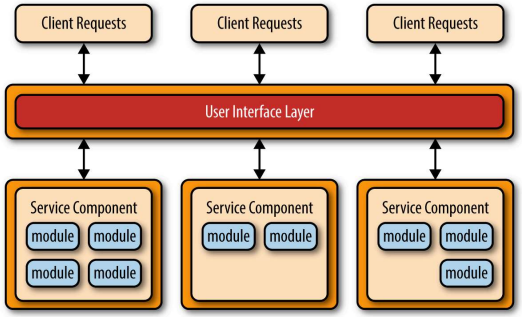


Pattern Topologies

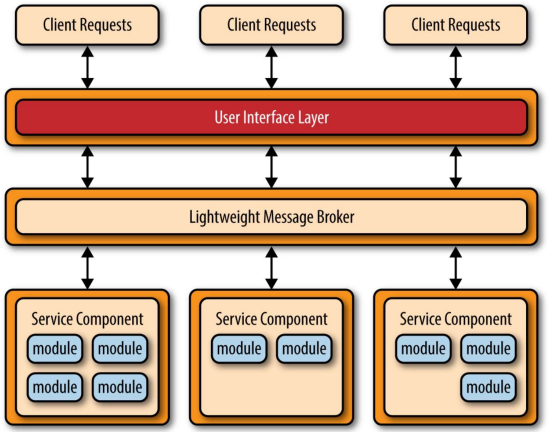
- There are many distinct ways to implement a microservices architecture pattern
- But three main topologies stand out as the most common and popular:
  - the API REST-based topology (1)
  - application REST-based topology (2)
  - the centralized messaging topology (3)



(1)



(2)



(3)

Considerations

- Applications are generally more robust, provide better scalability, and can more easily support continuous delivery.
- Capability to do real-time production deployments.
  - Only the service components that change need to be deployed.
- But .. distributed architecture
  - it shares some of the same complex issues found in the event-driven architecture pattern, including contract creation, maintenance, and government, remote system availability, and remote access authentication and authorization.

Pattern Analysis

- Overall agility
  - Rating: High
- Ease of deployment
  - Rating: High
- Testability
  - Rating: High
- Performance
  - Rating: Low
- Scalability
  - Rating: High
- Ease of development
  - Rating: High

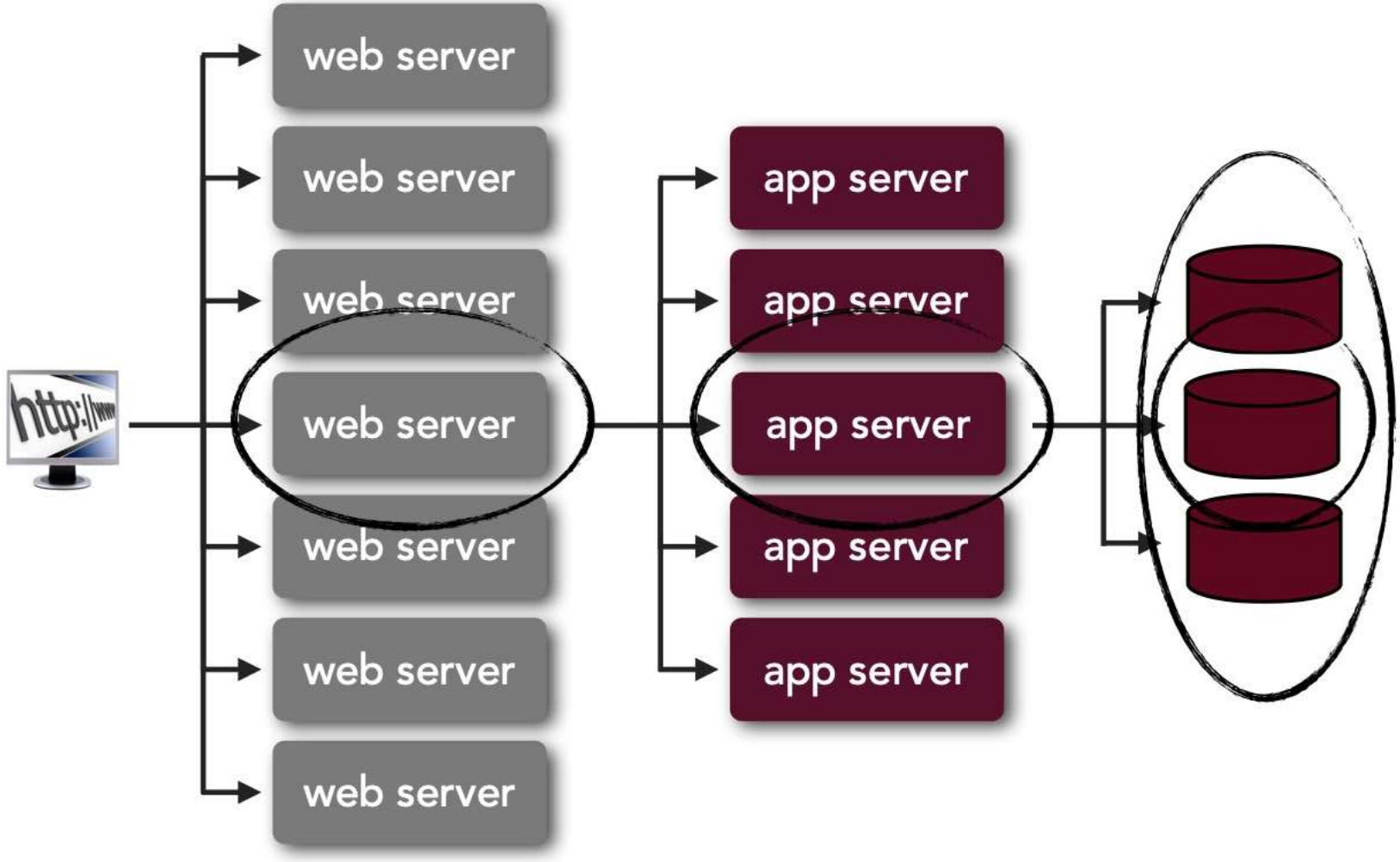
Software Architecture Patterns

- Layered Architecture
- Event-Driven Architecture
- Microkernel Architecture
- Microservices Architecture Pattern
- **Space-Based Architecture**

Space-Based Architecture

- Most web-based business applications follow the same general request flow:
  - a request from a browser hits the web server, then an application server, then finally the database server.
- Bottlenecks start appearing as the user load increases
  - first at the web-server layer, then at the application-server layer, and finally at the database-server layer.
- The space-based architecture pattern is specifically designed to address and solve **scalability and concurrency** issues.
  - It is often a better approach than trying to scale out a database or retrofit caching technologies into a non-scalable architecture.

Space-Based Architecture



Pattern Description

- This pattern gets its name from the concept of tuple space, the idea of distributed shared memory.
  - Also referred to as the *cloud architecture pattern*
  - High scalability is achieved by replacing the central database with replicated in-memory data grids.
  - Application data is kept in-memory and replicated among all the active processing units.
- Processing units can be dynamically started up and shut down as user load increases and decreases.
  - The database bottleneck is removed, providing near-infinite scalability within the application.

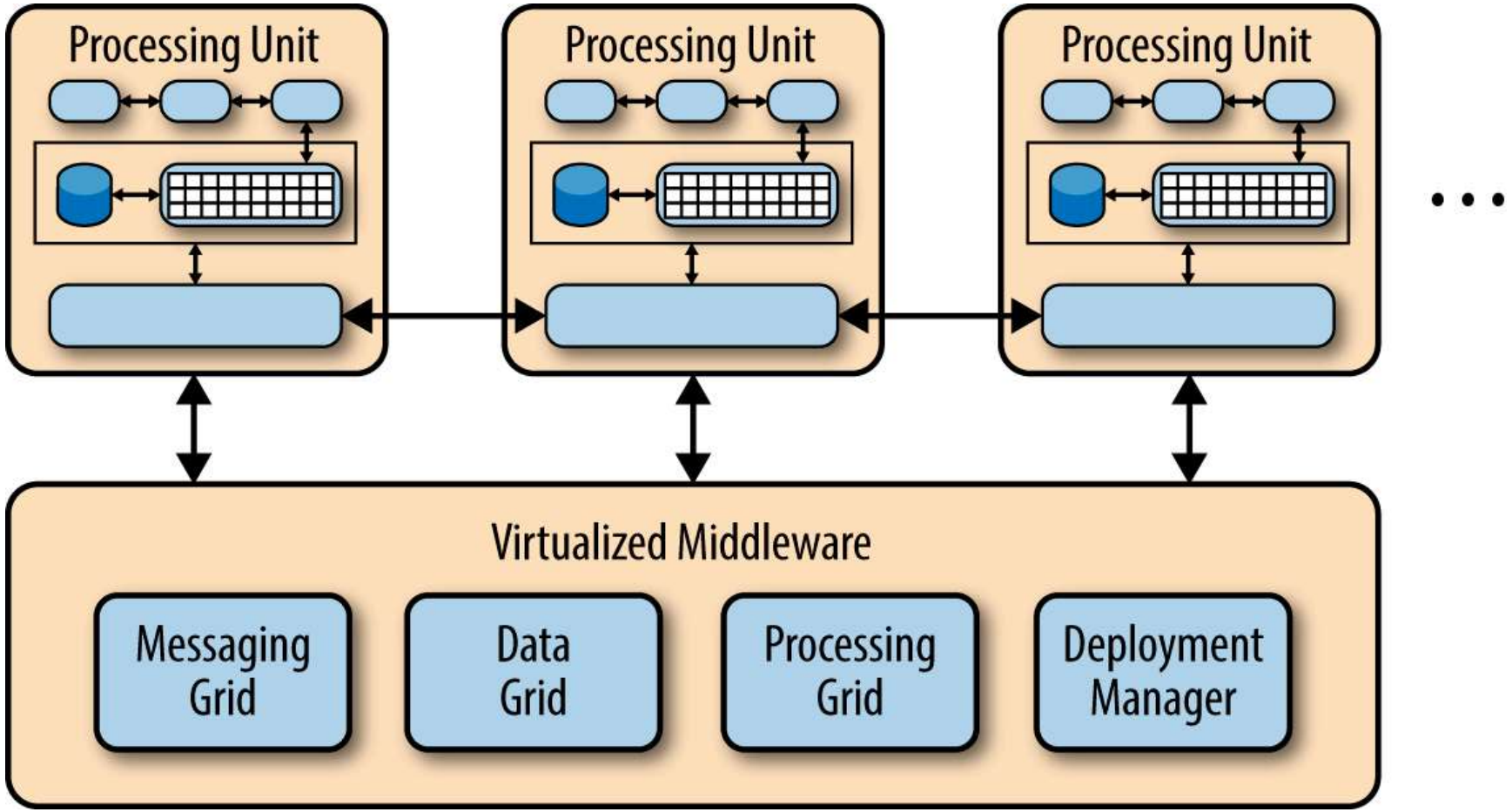
Pattern Description

- There are two primary components within this architecture pattern: a processing unit and virtualized middleware.

- The **processing-unit component** contains the application components.
  - This includes web-based components as well as backend business logic.

- The **virtualized-middleware** component handles housekeeping and communications.

Space-Based Architecture



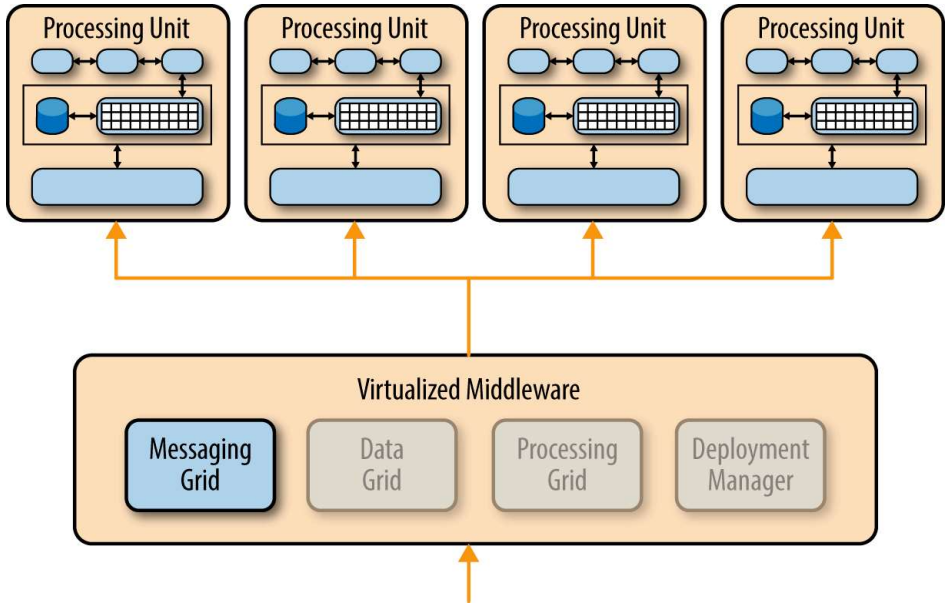
Virtualized Middleware

- The controller that manages requests, sessions, data replication, distributed request processing, and process-unit deployment.

- There are four main architecture components in the virtualized middleware:
  - messaging grid, manages input request and session information
  - data grid
  - processing grid
  - deployment manager

Messaging Grid

- Manages input request and session information
  - For each request the messaging-grid component determines which active processing components are available to receive the request
  - The strategy can range from a simple round-robin algorithm to a more complex next-available algorithm.

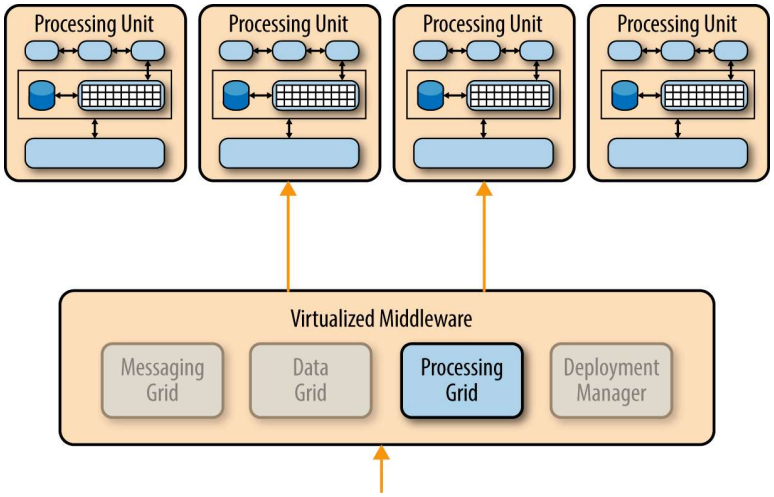


Data Grid

- The data grid interacts with the data-replication engine in each processing unit
- The goal is to manage the data replication between processing units when data updates occur.
- Each processing unit must contains exactly the same data in its in-memory data grid.
- This need to be done in parallel, asynchronously, and very quickly, sometimes in a matter of microseconds.

Processing Grid

- Optional component within the virtualized middleware
  - it manages distributed request processing when there are multiple processing units, each handling a portion of the application.
  - It is responsible for the coordination between processing unit types (e.g., an order processing unit and a customer processing unit).



Deployment Manager

- This component manages the dynamic startup and shutdown of processing units based on load conditions.
- It continually monitors response times and user loads
  - Starts up new processing units when load increase
  - Shuts down processing units when the load decreases
- It is a critical component to achieving variable scalability needs within an application.

Considerations

- The space-based architecture pattern is a complex and expensive pattern to implement.
- It is a good architecture choice for smaller web-based applications with variable load (e.g., social media sites, bidding and auction sites).
- However, it is not well suited for traditional large-scale relational database applications with large amounts of operational data.

Pattern Analysis

- Overall agility
  - Rating: High
- Ease of deployment
  - Rating: High
- Testability
  - Rating: Low
- Performance
  - Rating: High
- Scalability
  - Rating: High
- Ease of development
  - Rating: Low

Pattern Analysis Summary

	Layered	Event-driven	Microkernel	Microservices	Space-based
Overall Agility	↓	↑	↑	↑	↑
Deployment	↓	↑	↑	↑	↑
Testability	↑	↓	↑	↑	↓
Performance	↓	↑	↑	↓	↑
Scalability	↓	↑	↓	↑	↑
Development	↑	↓	↓	↑	↓

Resources

- Software Architecture Patterns, Mark Richards, O'Reilly Media, Inc., 2015
  - http://www.oreilly.com/programming/free/files/software-architecture-patterns.pdf