Architectural Patterns

Please read Chapter 16

Dr. Abdelkarim Erradi

Computer & Engineering Science Dept.

QU

Outline

- Architectural Patterns
  - Layered Architecture
  - Model-View-Controller (MVC)
  - Message Bus
  - Service-Oriented Architecture (SOA)
  - Pipes and Filters
  - Other patterns
- Architecture Design Example
- Project Milestone 3 Tips

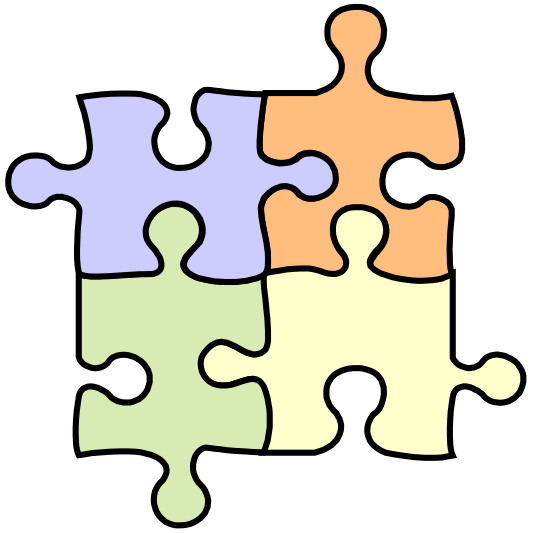
What Is an Architectural pattern?

- You can think of architecture pattern = a re-usable template solution of how to organize a system into components and connectors to solve frequently recurring design problems.
  - Proven best practices that can provide guidelines for assembling elements in some form
  - **Improves partitioning** and **promotes design reuse**
- Choose a architecture pattern/patterns that suit requirements
  - No magic formula
  - Analyze requirements and quality attributes supported by each pattern
- Complex architectures require creative blending of multiple patterns.

Architectural patterns

A pattern is a generally repeatable solution to a commonly occurring problem in software design

- Codifies knowledge collected from experience in a domain
- Represents distilled reusable experience
- Simplifies and speeds-up architecture and design
- Reduces risk
- Facilitates communication between practitioners



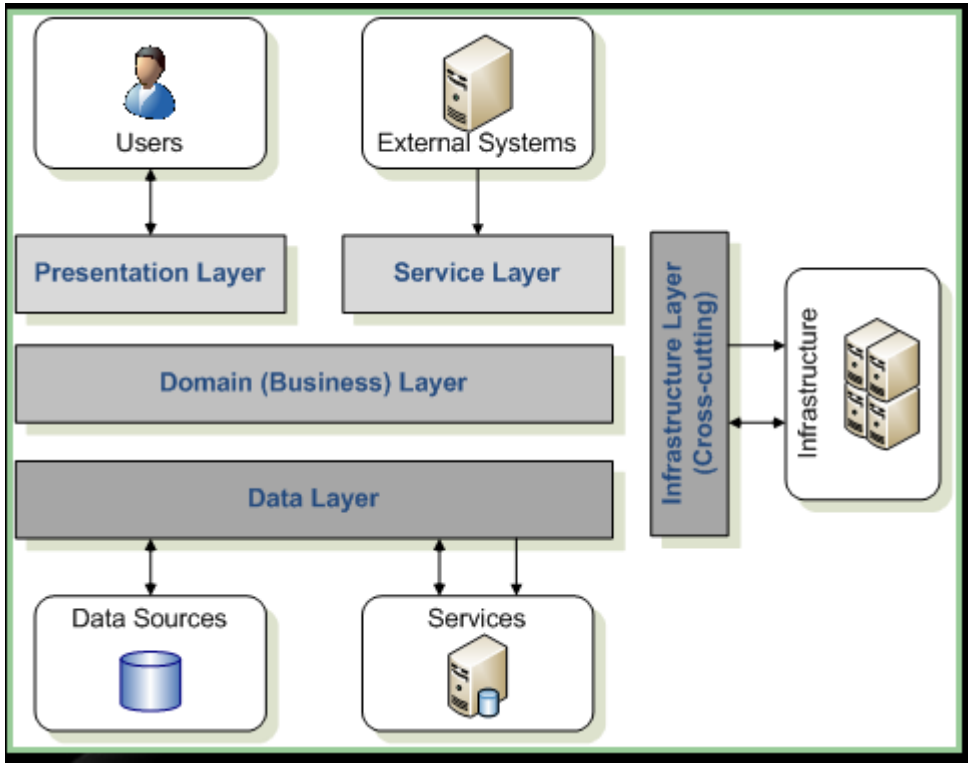
Common Architectural Styles

Category	Architecture styles
Structure	Layered Architecture
Interaction	Model-View-Controller (MVC)
Communication	Message Bus Service-Oriented Architecture (SOA) Pipes and Filters

Layered Architecture

Layered Architecture

- The high level design solution is decomposed into Layers with a unique role per layer:
  - Structurally, each layer provides a *related set* of services
  - Dynamically, each layer may only use the layers below it
- Cross-Cutting Concerns
  - Isolate domain logic from infrastructure concerns such as Authentication, Authorization, Logging
- Business logic can be used by multiple presentations as well as the service layer

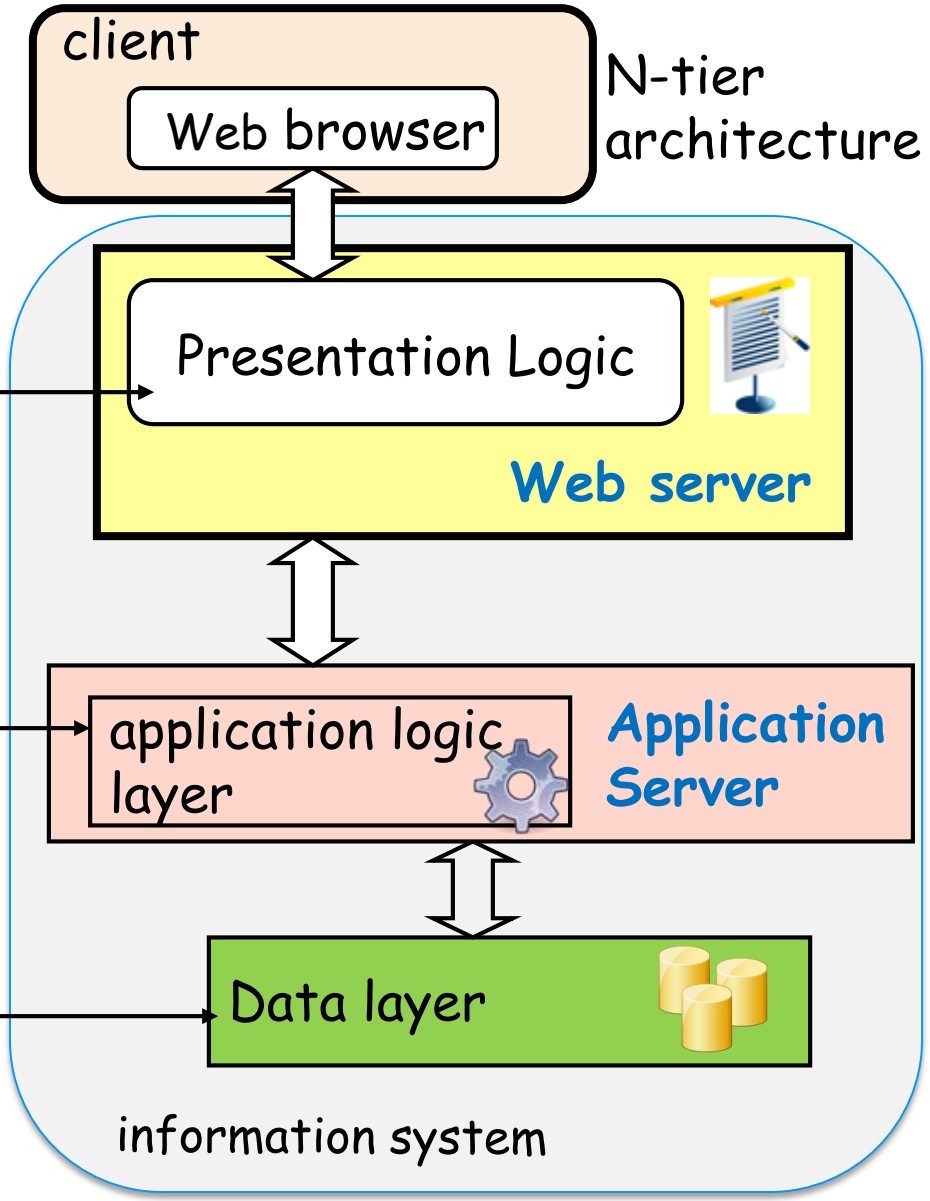


Typical Layered Architecture Pattern

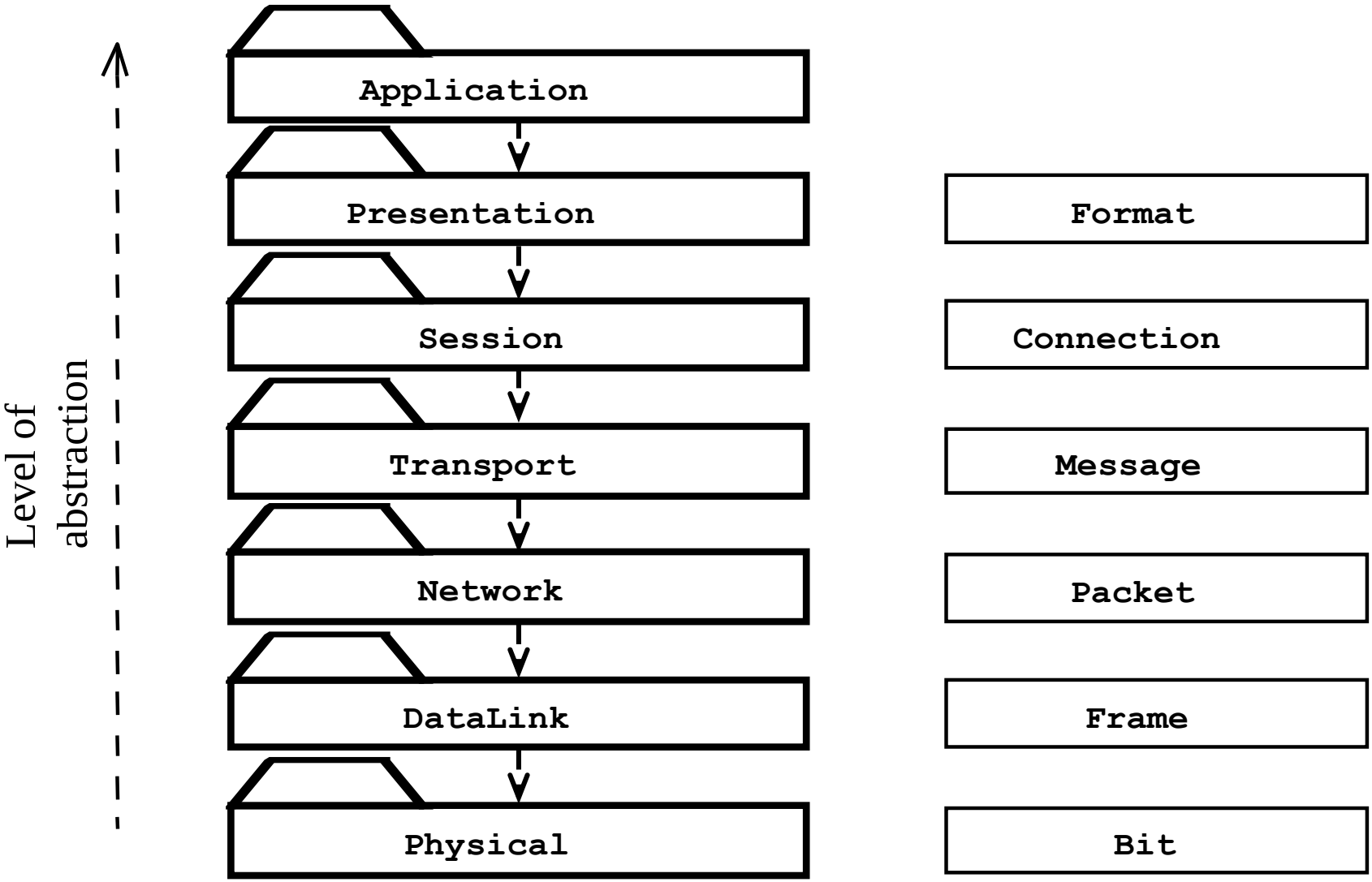
1. Takes care of user interfaces. Clients interact with the system through a presentation layer.

2. Implements computation, Enforces business rules and coordinates business processes

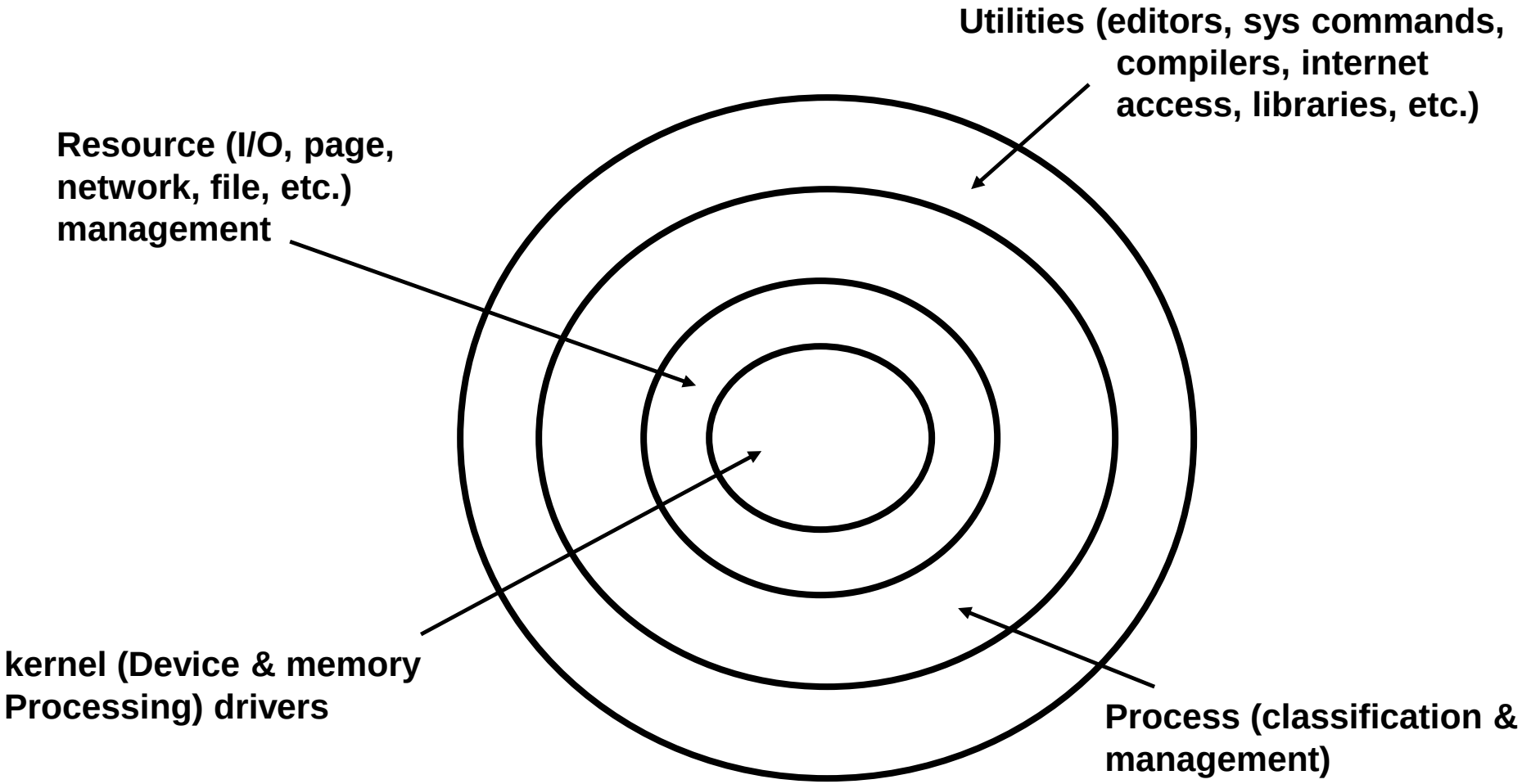
3. define the data sources and data organization needed to implement the application logic



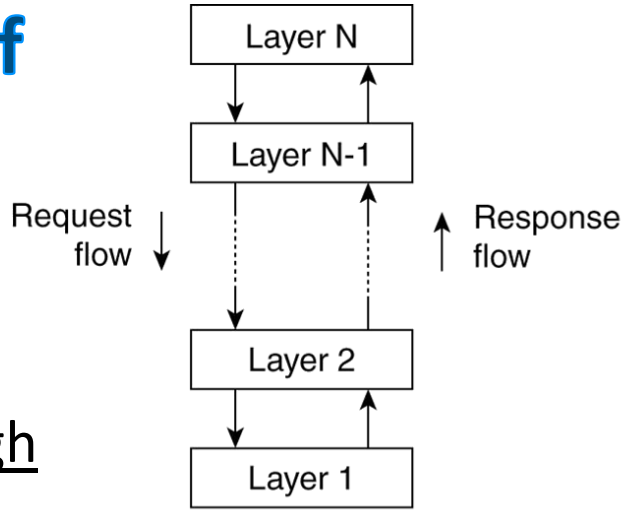
Example - OSI 7 Layers Architecture



Example - Layered Architecture for OS



Advantages and Disadvantages of Layered Architecture



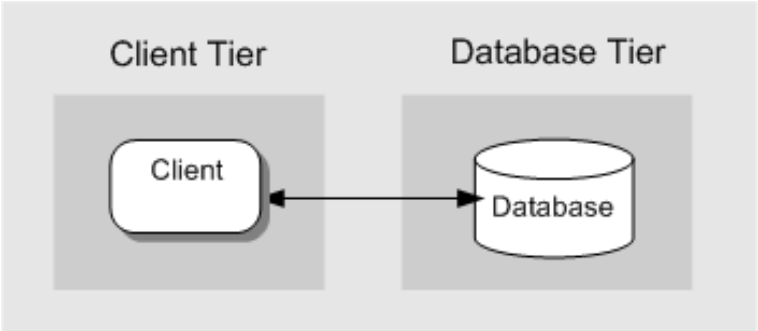
- Advantages:
  - Each layer is selected to be a set of related services; thus the architecture provides high degree of cohesion within the layer.
  - Each layer hides complexity from other layers
  - Layers may use only lower layers hence reducing coupling.
  - Each layer, being cohesive and is coupled only to lower layers, makes it easier for reuse and easier to be replaced
  - Flexible deployment: all layers could run on the same machine, or each tier may be deployed on its own machine.
- Disadvantages:
  - Layered Style may cause performance problem depending on the number of layers

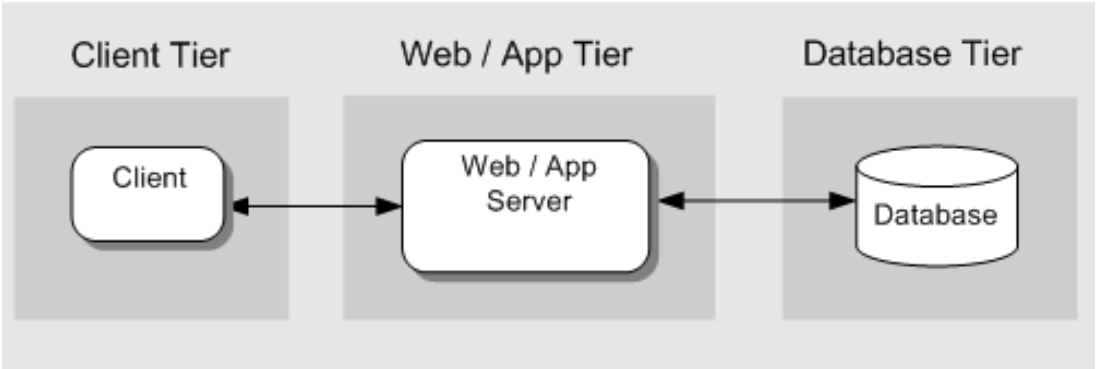
Layered Architecture – Quality Attribute Analysis

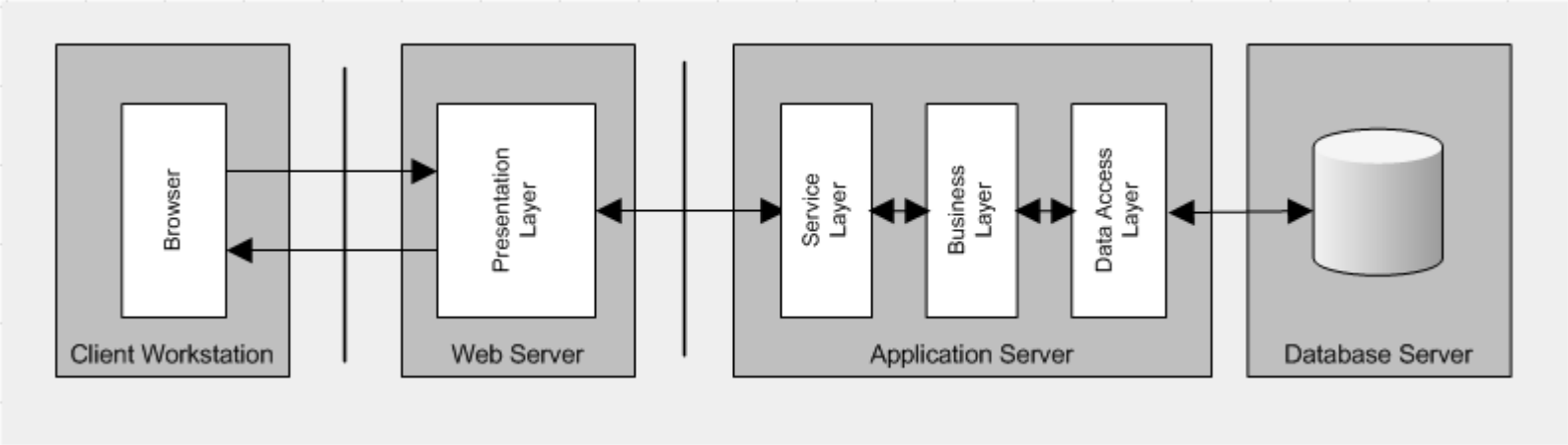
Quality Attribute	Issues
Availability	Servers in **each tier can be replicated**, so that if one fails, others remain available. This means a client request is, without its knowledge, redirected to a live replica server that can satisfy the request. Overall the application will provide a lower quality of service until the failed server is restored.
Modifiability	**Separation of concerns enhances modifiability**, as the presentation, business and data management logic are all clearly encapsulated. Each can have its internal logic modified in many cases without changes rippling into other layers.
Performance	**Performance maybe slightly degraded**. Key issues to consider are the speed of connections between tiers and the amount of data that is transferred. As always with distributed systems, it makes sense to minimize the calls needed between tiers to fulfill each request.
Scalability	As servers in **each tier can be replicated** the architecture scales well. In practice, the data management tier often becomes a bottleneck on the capacity of a system.

Deployment Patterns: Tiers (2-Tier, 3-Tier, N-Tier)

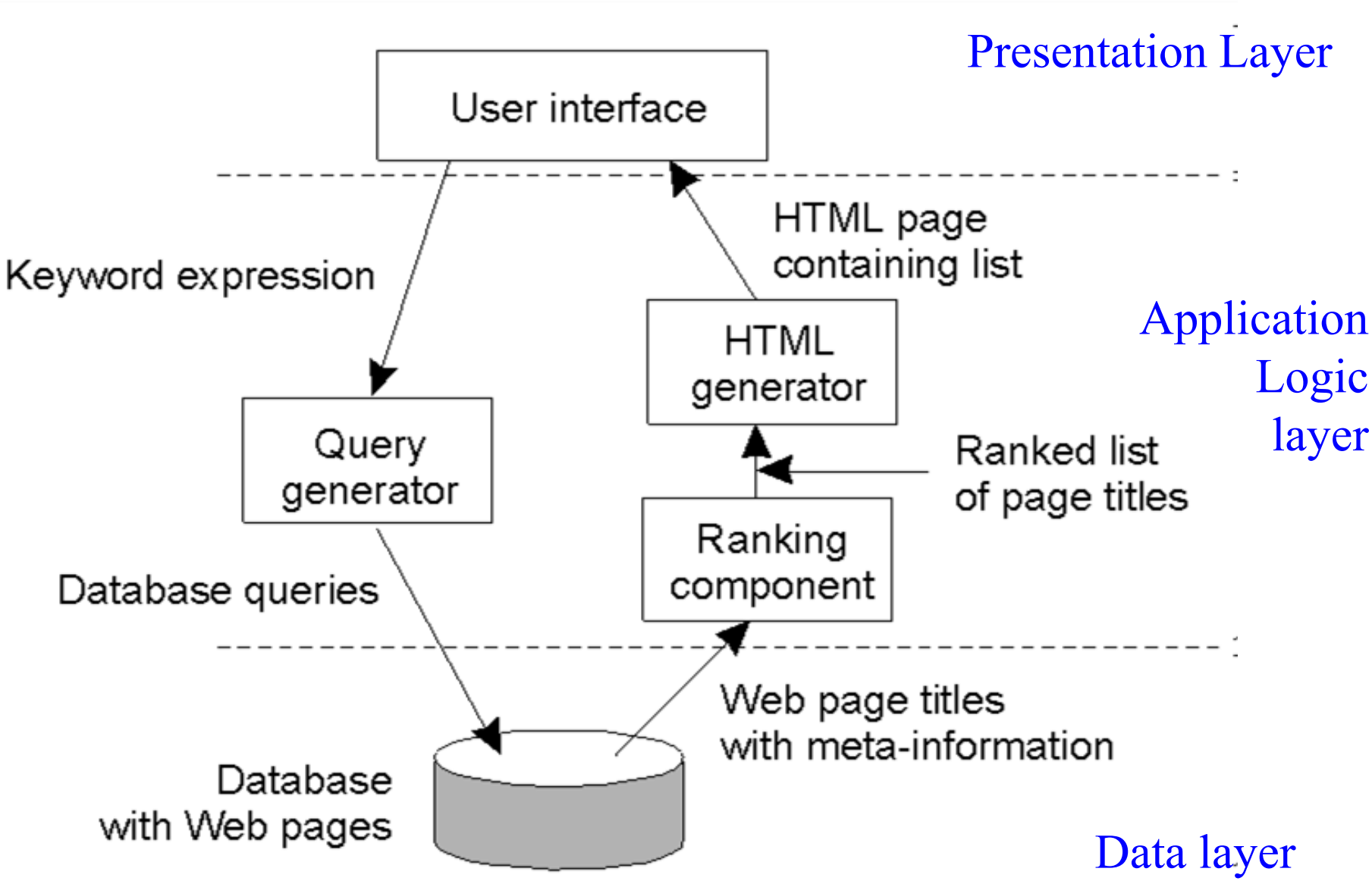
Layered Architecture provides flexible deployment: There are no restrictions on how a multi-layer application is deployed. All layers could run on the same machine, or each tier may be deployed on its own machine.







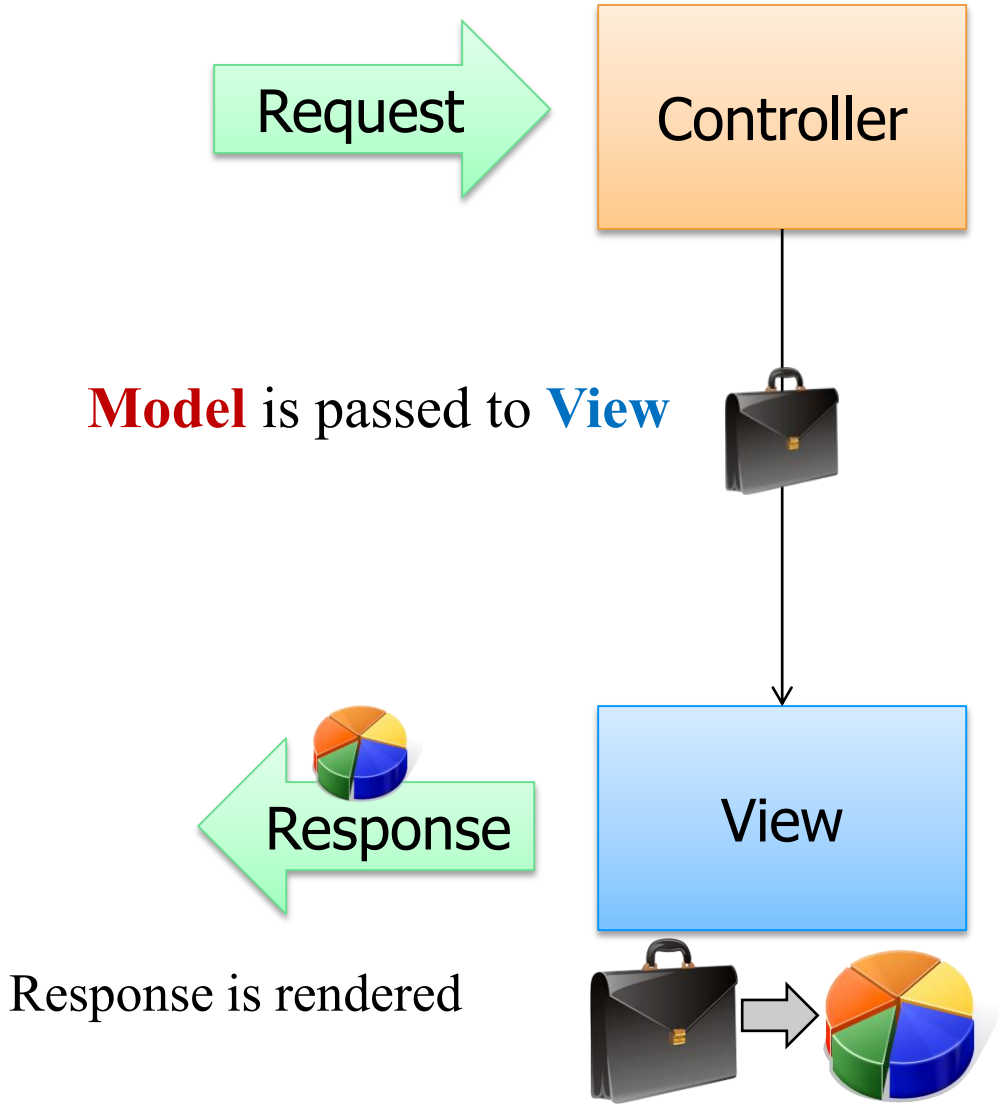
Example - Internet search engine



MVC

The MVC pattern is intended to allow each part to be changed independently of the others.

How MVC Works



Controller

- Incoming request directed to **Controller**
- A controller accepts input from the user and instructs the model to perform actions based on that input

e.g. the controller adds an item to the user's shopping cart

- Results objects are then passed to the View

View

Collects user input and displays results

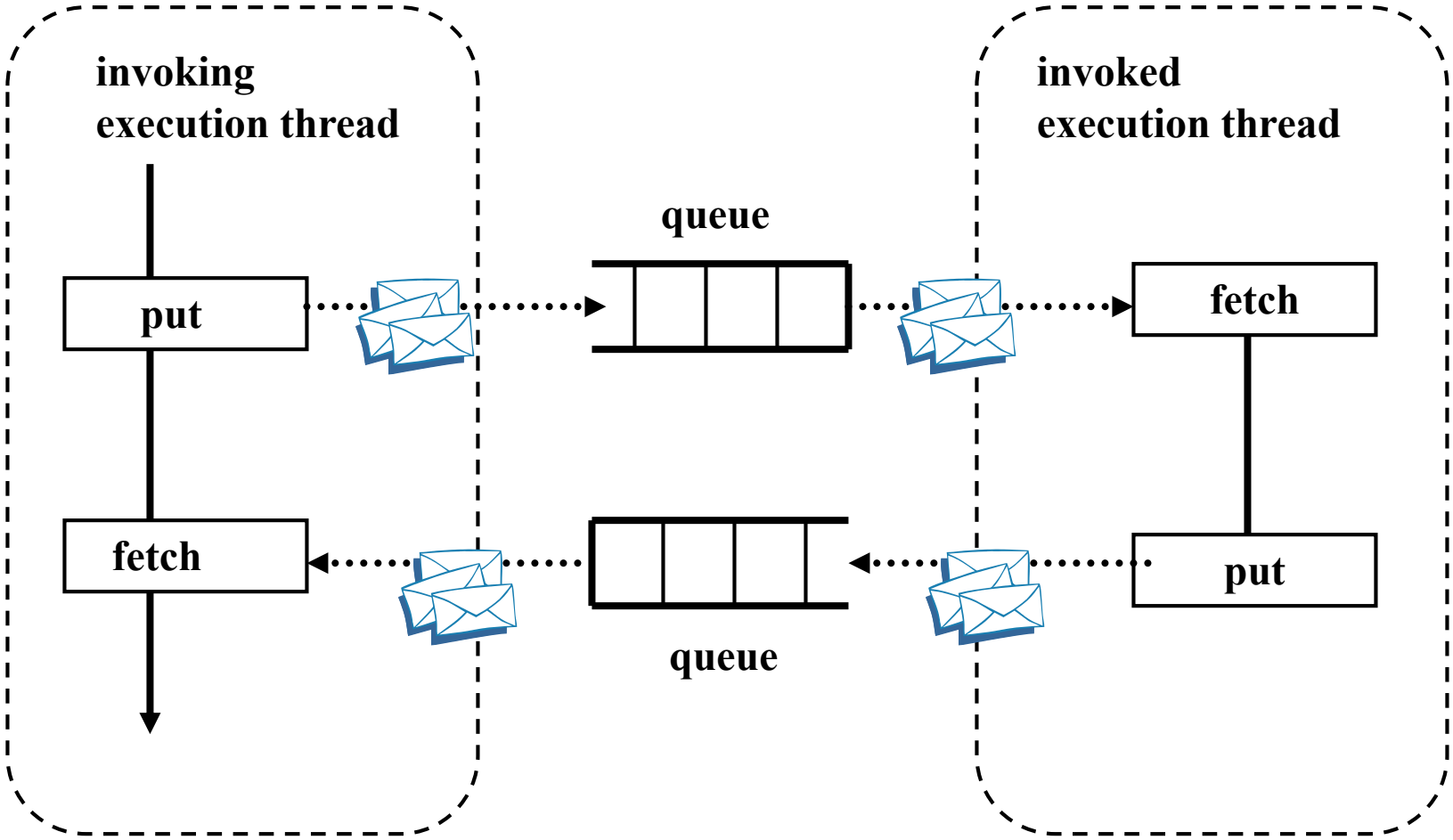
Advantages of MVC

- ***Separation of concerns***
  - Views, controller, and model are separate components. This allows modification and change in each component without significantly disturbing the other.
    - Computation is not intermixed with Presentation. Consequently, code is cleaner and easier to understand and change.
- **Flexibility**
  - The view component, which often needs changes and updates to keep the users continued interests, is separate
    - The UI can be completely changed without touching the model in any way
- **Reusability**
  - The same model can used by different views (e.g., Web view and mobile view)
- **Disadvantages:**
  - Heavily dependent on a framework and tools that support the MVC architecture (e.g. ASP.Net MVC, Ruby on Rails)

MVC is widely used and recommended particularly for interactive web-applications.

Message Bus

Message Bus (basic version)
- uses a Message Oriented Middleware

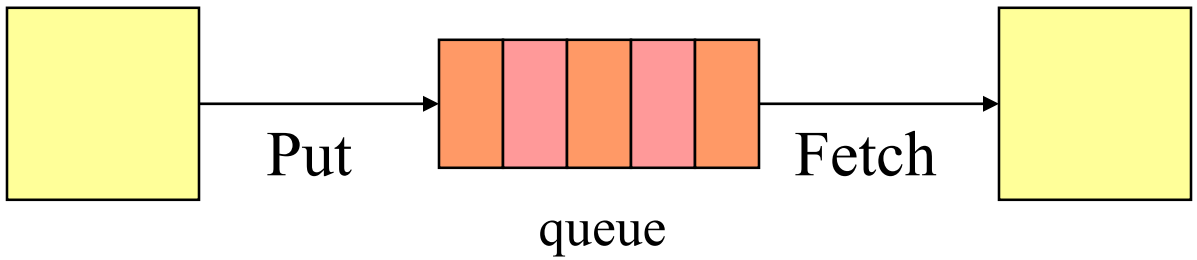


Aimed at achieving *decoupling*
and reliability

= Messages

Implicit Invocation Style

- Usually facilitated by a Message Oriented Middleware (MOM)
  - **Put** (queue, message) – Write message onto queue
  - **Fetch** (queue, message) – Read message from queue
- Sender places a message in a queue instead of method invocation
  - "Listeners" read message from queue and process it



MOM Advantages and Disadvantages

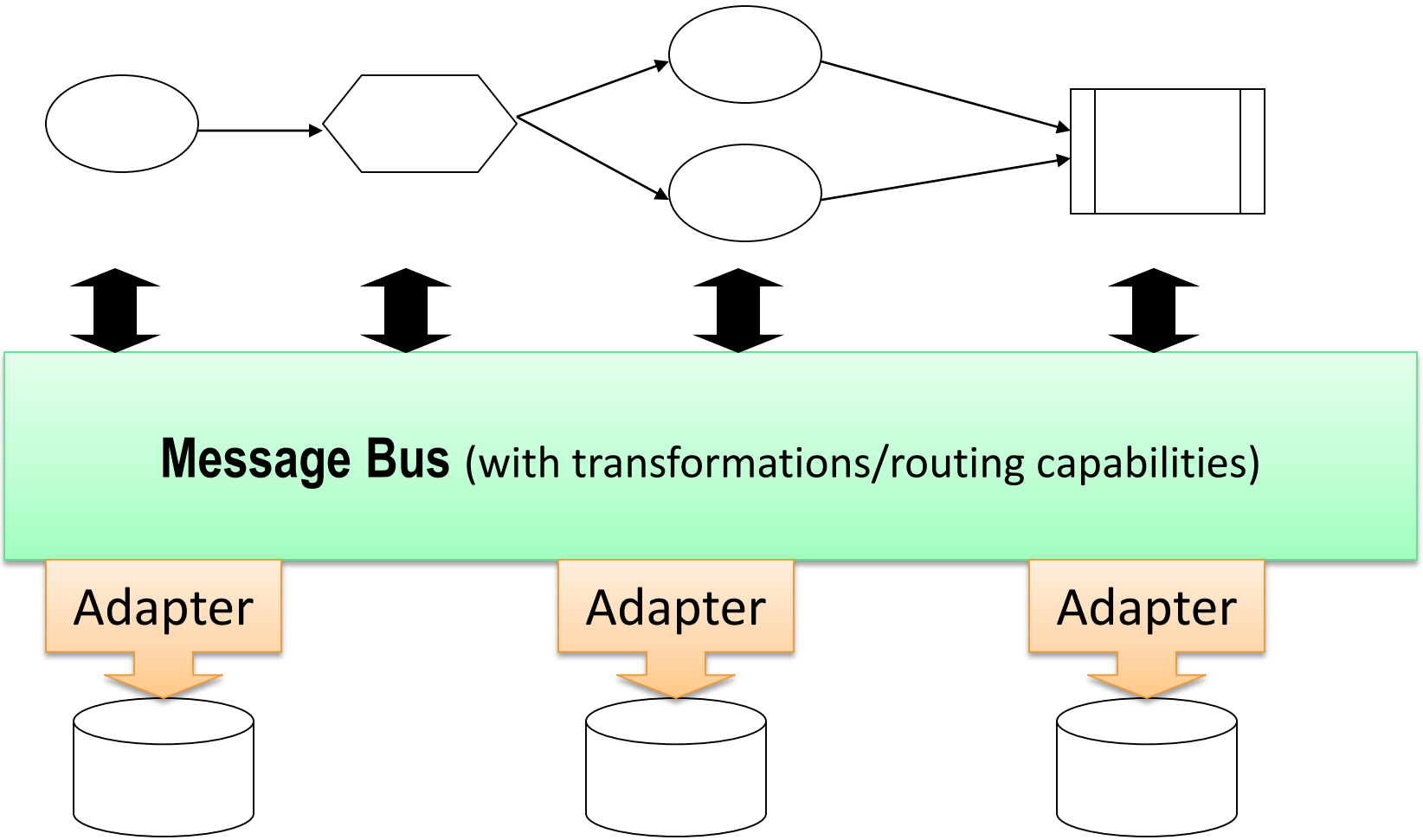
- **Advantages**
  - **Lower coupling between components**: the message senders and the message processors are separate
    - Easier system evolution: e.g., a component can be easily replaced by another one
    - Any sender or processor malfunction will not affect the other senders and message processors
  - Higher component reuse
- **Disadvantages**
  - MOM malfunction will bring the whole system down
  - MOM can be a single point of failure
  - Lower system understandability:
    - No knowledge of what components will respond to event
    - No knowledge of order of responses

Messaging – Quality Attribute Analysis

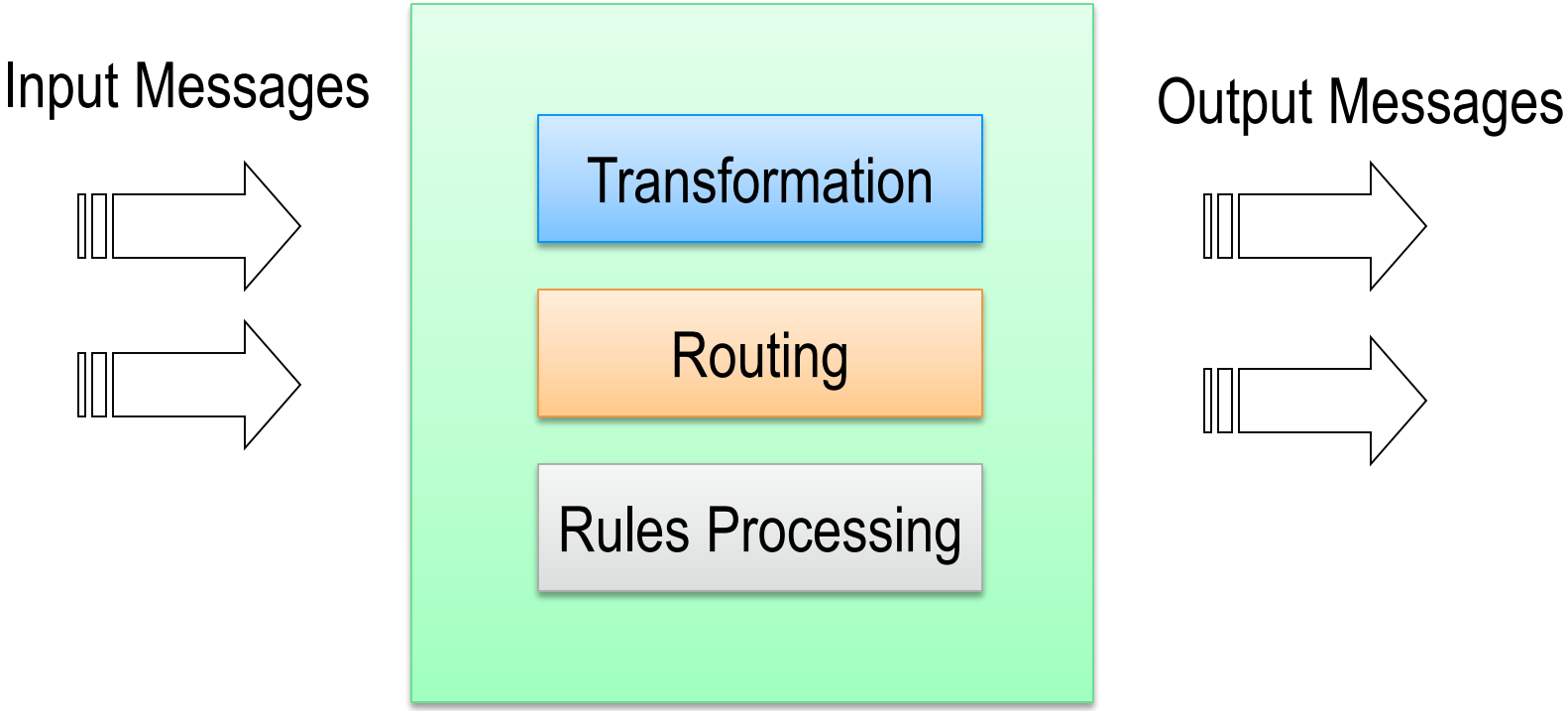
Quality Attribute	Issues
Availability	Physical **queues with the same logical name can be replicated across different messaging server instances**. When one fails, clients can send messages to replica queues.
Modifiability	Messaging is inherently loosely coupled, and this promotes high modifiability as clients and servers are **not directly bound through an interface**. Changes to the format of messages sent by clients may cause changes to the server implementations => **dependency on message structure and format**
Performance	Message queuing technology can deliver thousands of messages per second.
Scalability	**Queues can be replicated** across clusters of messaging servers hosted on multiple server machines. This makes messaging a highly scalable solution.

Message Bus Architecture

Message Bus uses Adapters to communicate with applications



Message Bus Services



Message Bus Features

- Message transformation – transform between different source/target formats
  - Graphical message format definition and mapping tools
  - High performance transformation engines
- Intelligent routing
  - Route messages based on message content
- Rules Engine
  - For rule-based routing and transformations
  - Using a scripting language with built-in functions

Message Bus - Quality Attribute Analysis

Quality Attribute	Issues
Availability	To build high availability architectures, **the Message Bus must be replicated**.
Failure handling	Message Bus has **typed** input ports to **validate and discard any messages that are sent in the wrong format**. With replicated bus, senders can fail over to a live bus should one of the replicas fail.
Modifiability	Message Bus separates the transformation and message routing logic from the senders and receivers. This **enhances modifiability**, as changes to transformation and routing logic can be made without affecting senders or receivers.
Performance	Message Bus **can potentially become a bottleneck**, especially if they must service high message volumes and execute complex transformation logic.
Scalability	**Clustering Message Bus instances** makes it possible to construct systems scale to handle high request loads.

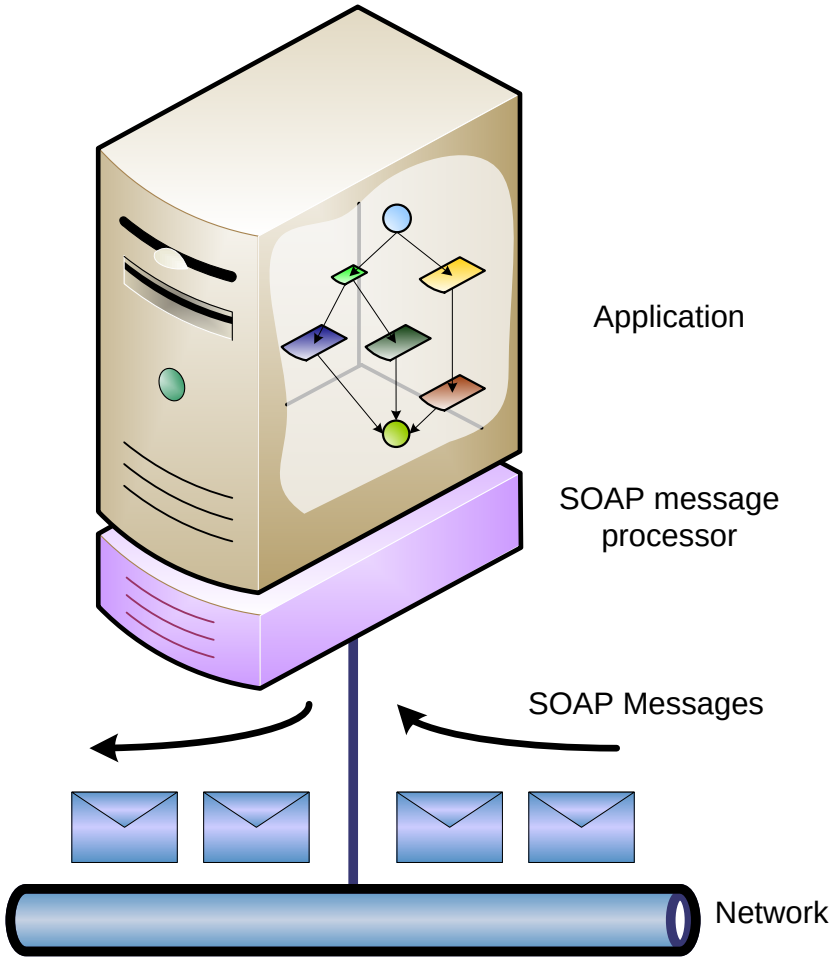
Service Oriented Architecture

Service-Oriented Architecture (SOA)

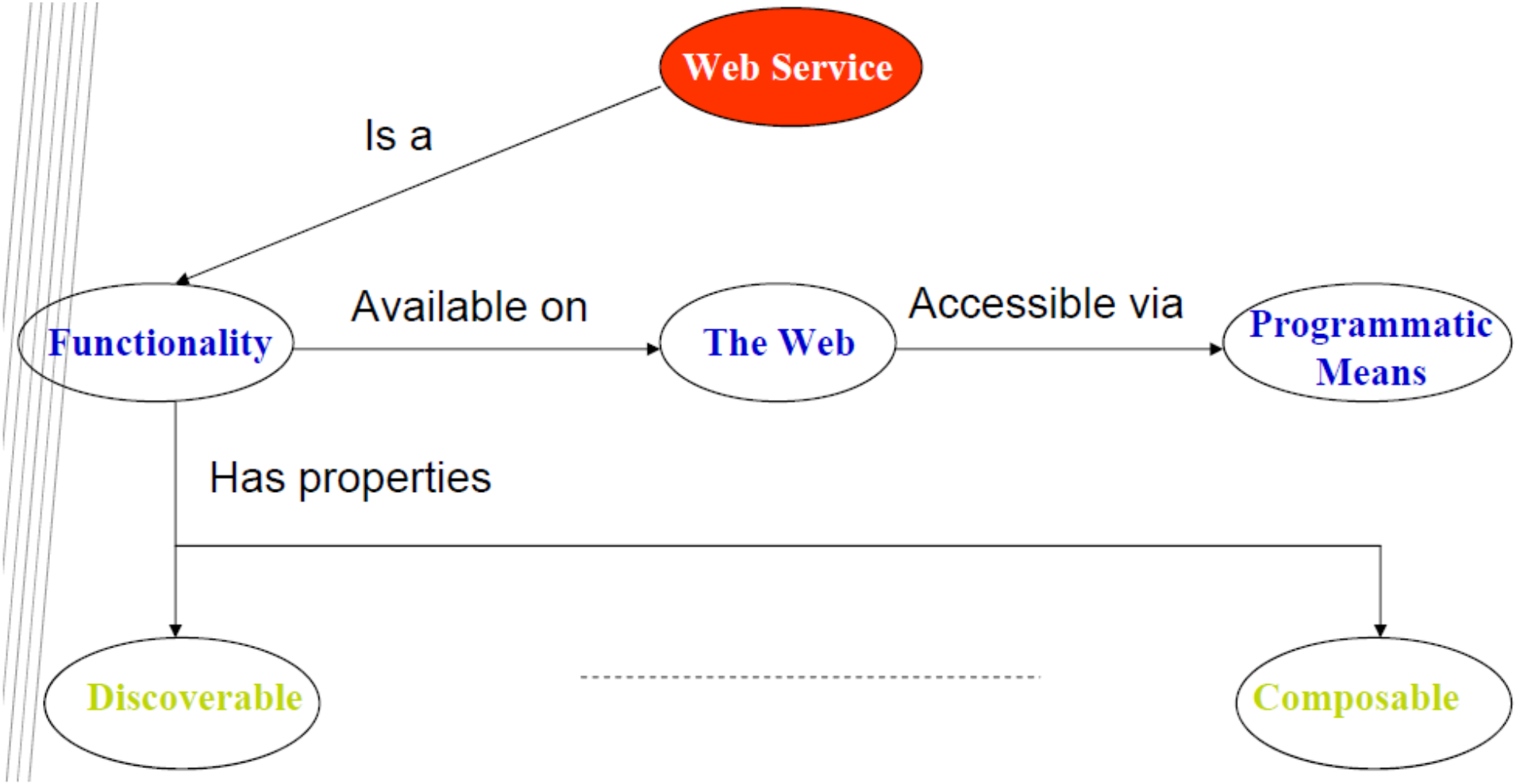
- Refers to applications that **expose and consume functionality as a service** using standard message formats and communication protocols
- Services are well-defined business functionalities that are built as loosely-coupled software components that can be reused for different purposes
- Web services provides a new paradigm for program to program communication
- Interoperability is the key goal of SOA

What is a Web Service?

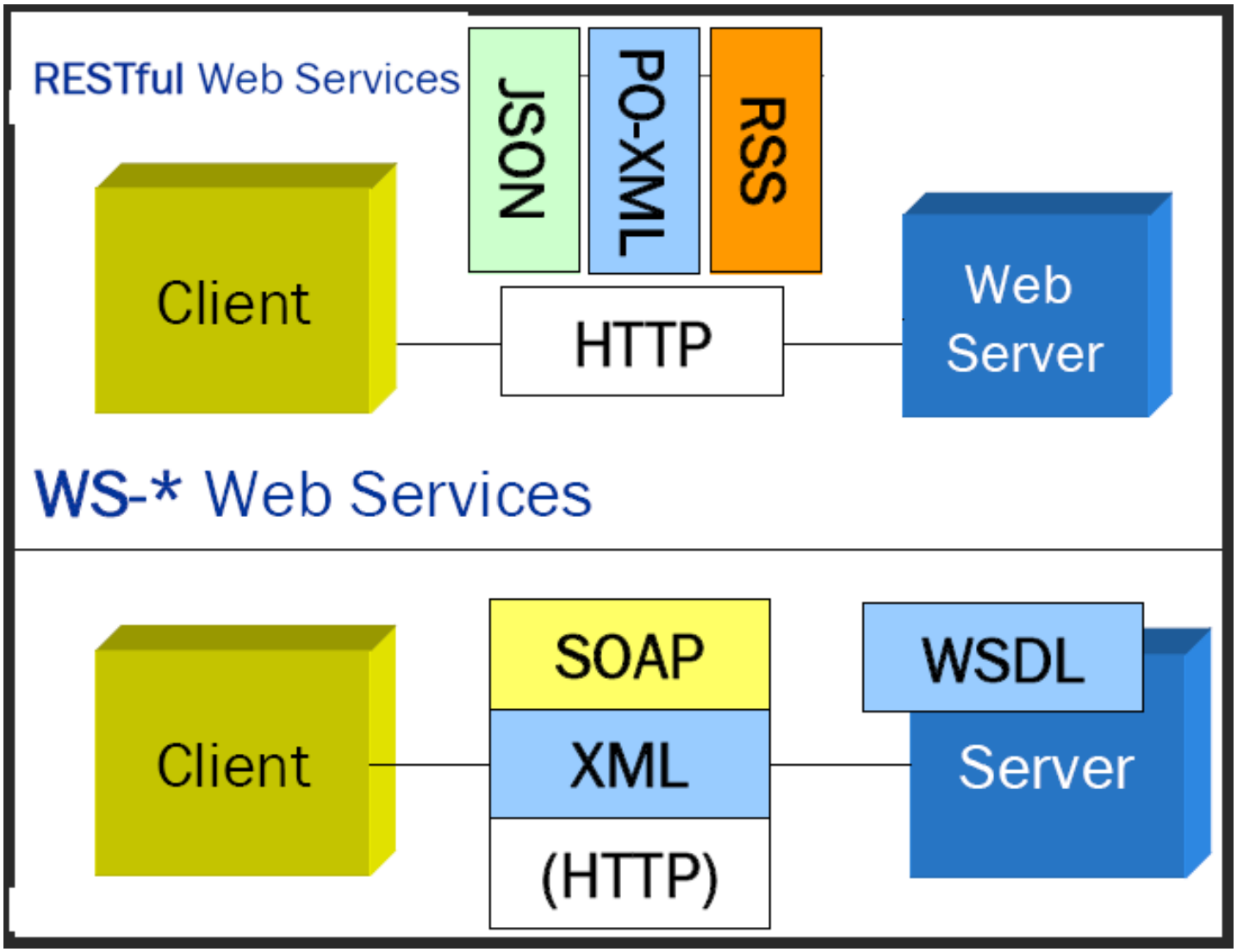
- Web services = latest fashion for building and integrating applications/components
- Software component provided through a **network-accessible endpoint**
  - Someone else may own the service and is responsible for its operation
- Services exchange standard XML messages
- Major design goal = **interoperability between heterogeneous systems**



What is Web Service?



REST vs. SOAP Services



REST = The Web of Services should work the way the Web of Pages works

JSON and XML Simple Example

JSON = Hierarchical key/value pairs

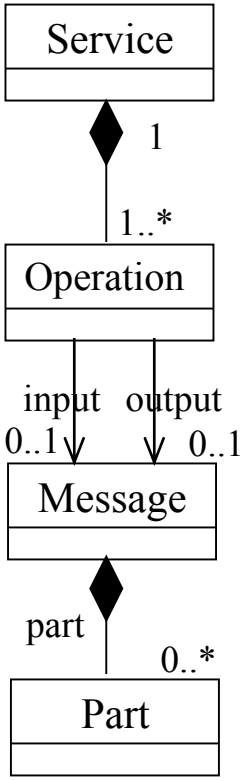
```
{ firstName: "Amir",
  lastName: "Mahdi",
  address: {
  streetAddress: "5 Qu Rd",
     city: "Doha",
     state: "Qatar",
     postalCode: 2713
  },
  phoneNumbers: [
     "06 555-4444",
     "05 111-2222" ]
}
```

XML = Tree of nested nodes

```
<Student>
 <firstName>Amir</firstName>
 <lastName>Mahdi</lastName>
  <address>
    <streetAddress>
      5 Qu Rd
    </streetAddress>
     <city>Doha</city>
     <state>Qatar</state>
     <postalCode>2713</postalCode>
    <phoneNumbers>
      <phoneNumber>06555-4444
      </phoneNumber>
      <phoneNumber>05111-2222
      </phoneNumber>
    </phoneNumbers>
  </address>
</Student>
```

Web Service Architecture

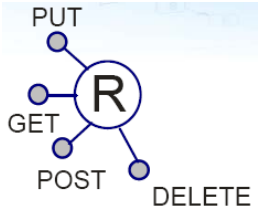
- The service is described in a machine-processable format called Web Services Description Language (WSDL).
  - XML-based language for describing the **functional interface** of a Web Service + the **mechanism for interacting** with that service
- Services exchange standard XML messages usually in 'Simple Object Access Protocol' (SOAP) format
  - XML as Data Serialization
- SOAP Engine at two sides of interaction Serialize and Deserialize the message content and route it to the appropriate implementation.
- **Benefits:**
  - Good support for security, routing, reliable messaging, etc.
- **Drawbacks:**
  - Requires heavier, more specific infrastructure



What are REST Services?

Websites designed for *computers* instead of people.

REST Principles



- **Addressable Resources** (nouns): Identified by a URI

(e.g., http://example.com/customers/123)

- **Uniform Interface** (verbs): GET, POST, PUT, and DELETE

  -Use verbs to exchange application state and representation

  -Embracing HTTP as an Application Protocol

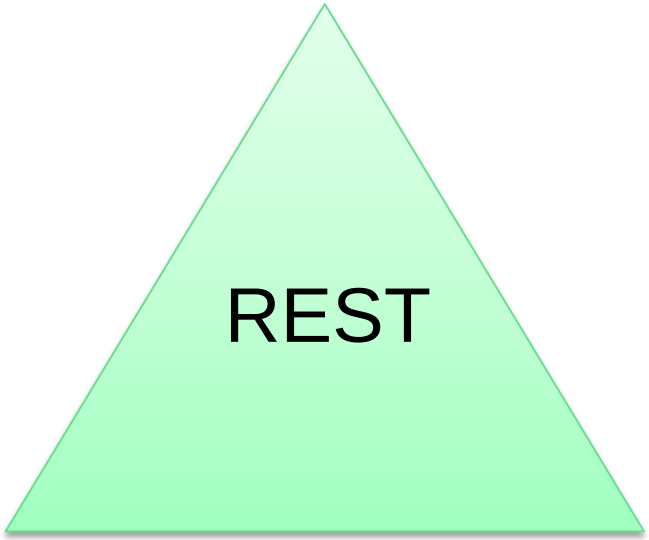
- **Representation-oriented**

  -**Representation of the resource state** transferred between client and server in a variety of data formats: XML, JSON, (X)HTML, RSS..

- **Hyperlinks** define relationships between resources and valid state transitions of the service interaction

REST Services Main Concepts

Nouns (Resources)
e.g., http://example.com/employees/12345



Verbs
e.g., GET, POST

Representations
e.g., XML, JSON

Naming Resources

- A resource is a conceptual mapping to a set of entities

- REST uses URI to identify resources
  - http://localhost/books/
  - http://localhost/books/ISBN-0011
  - http://localhost/books/ISBN-0011/authors

  - http://localhost/classes
  - http://localhost/classes/cmps356
  - http://localhost/classes/cs356/students

- As you traverse the path from more generic to more specific, you are navigating the data

Representations

- Specify the data format used when returning a resource representation to the client

- Two main formats:
  - JavaScript Object Notation (JSON)
  - XML

- It is common to have multiple representations of the same data

Representations

- XML

```
<course>
   <id>cmps356</id>
   <name>Enterprise Application
   Development</name>
</course>
```

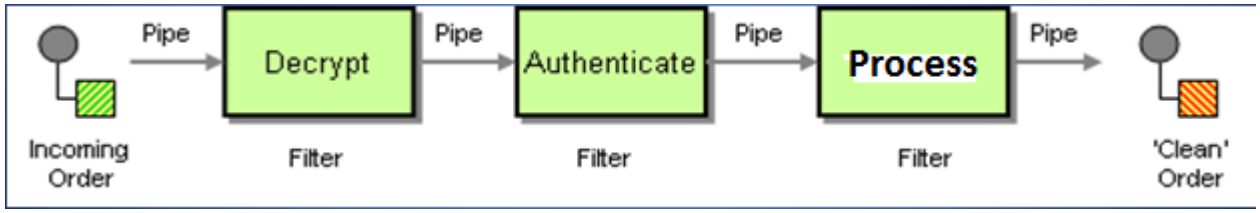
- JSON

```
{
  id: ‘cmps356’,
   name: ‘Enterprise Application Development’
}
```

Pipe and Filter Architecture

Pipe and Filter Architecture

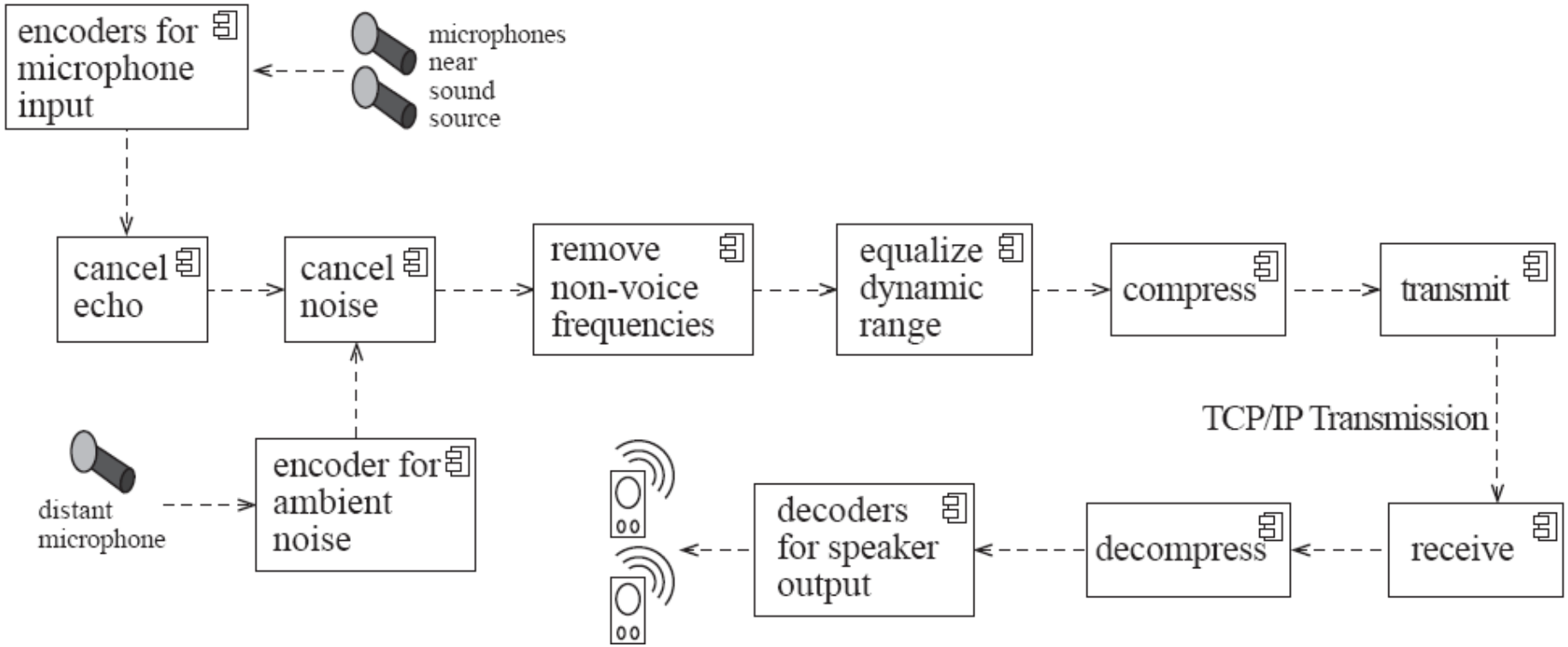
- **The solution design is decomposed into filters and pipes:**
  - Filter is a service that transforms a stream of input data into a stream of output data
  - Pipe is a mechanism or conduit through which the data flows from one filter to another
  - Allows developer to divide larger processing tasks into smaller, independent tasks
- Components are filters and Connectors are pipes



- Examples: UNIX shell, Signal processing

Problems that require batch file processing seem to fit this e.g., payroll and compilers

Example of a pipe-and-filter system



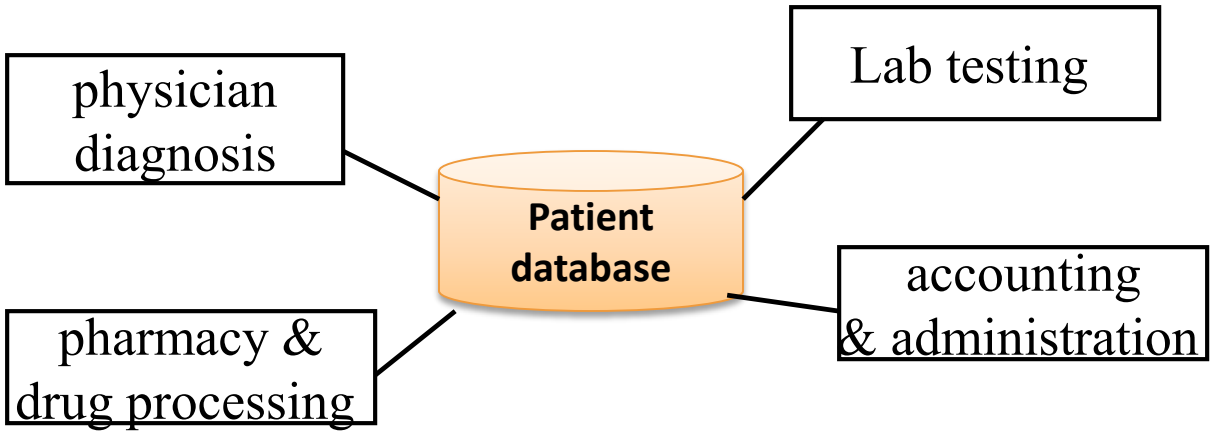
Advantages and Disadvantages of Pipe-Filter

- **Advantages:**
  - Filters are self containing processing service that performs a specific function thus it is fairly cohesive
  - Easier filter addition, replacement, and reuse
  - Filters communicate (pass data most of the time) through pipes only, thus it is constrained in coupling
- **Disadvantages:**
  - Filters processes and sends streams of data over pipes is a solution that fits well with heavy batch processing, but may not do well with any kind of user-interaction.

Other styles

Shared (Central) Data Store

- The high level design solution is based on a *shared data-store* which acts as the "central command" with 2 variations:
  - *Blackboard style*: the data-store alerts the participating parties whenever there is a data-store change
  - *Repository style*: the participating parties check the data-store for changes



Very Common In Business where Data is central

Problems that fit this style such as *patient processing*, *tax processing system*, *inventory control system*; etc. have the following properties:

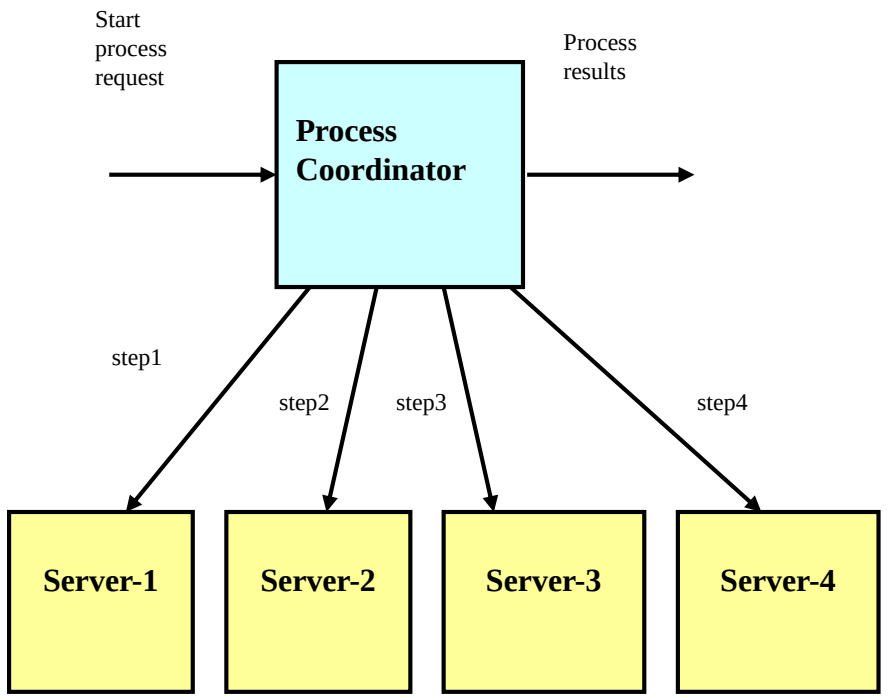
1. All the functionalities work off a single data-store.
2. Any change to the data-store may affect all or some of the functions
3. All the functionalities need the information from the data-store

Advantages and Disadvantages of Shared Data

- **Advantages:**
  - Higher component cohesion and low coupling: the **coupling is restricted to the shared data**
  - Single data-store makes the maintenance of data in terms of back-up recovery and security easier to manage
- **Disadvantages:**
  - **High coupling to shared data:** any data format change in the shared data requires agreement and, potentially, changes in all or some the functional areas.
  - **Data store can become a single point of failure**: if the data-store fails, all parties are affected and possibly all functions have to stop (may need to have redundant database for this architecture style; also, should have good back up- and recovery procedures.)

Process Coordinator Pattern

- **Process encapsulation:** The process coordinator coordinates the execution of a **sequence of steps** needed to fulfill the business process.
- **Loose coupling:** The server components are unaware of their role in the overall business process, and of the order of the steps in the process.
- **Flexible communications:** Communications between the coordinator and servers can be synchronous or asynchronous.



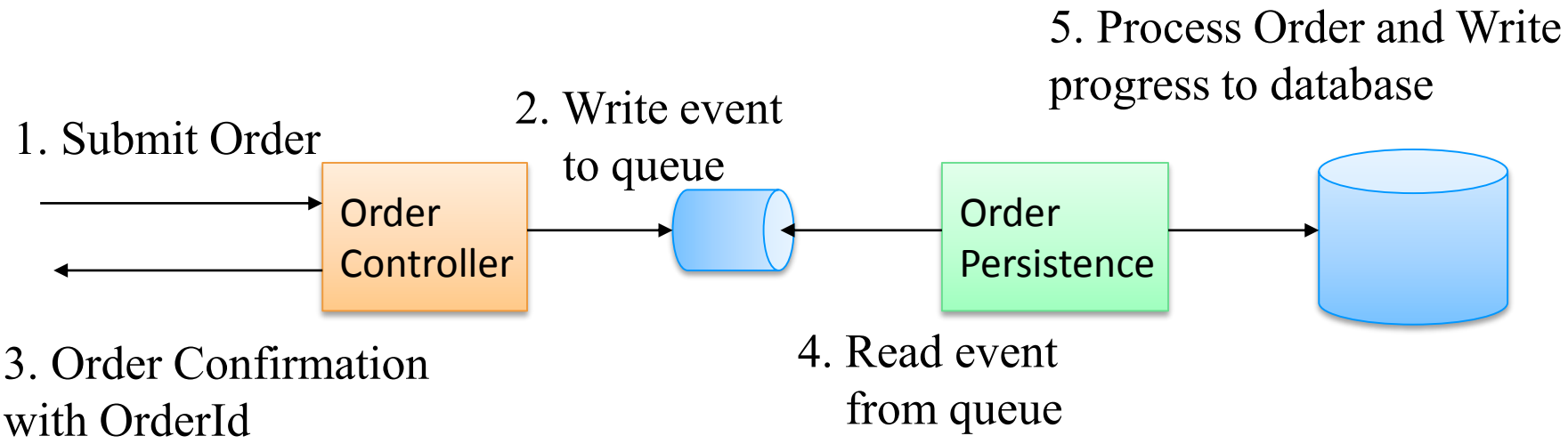
Process Coordinator – Quality Attribute Analysis

Quality Attribute	Issues
Availability	The **coordinator is a single point of failure**. Hence it needs to be replicated to create a high availability solution.
Failure handling	Failure handling is complex, as it can occur at any stage in the business process coordination. Failure of a later step in the process may require earlier steps to be undone using compensating transactions. Handling failures needs careful design to ensure the data maintained by the servers remains consistent.
Modifiability	Process modifiability is enhanced because the process definition is encapsulated in the coordinator process. Servers can change their implementation without affecting the coordinator or other servers, as long as their external service interface doesn't change.
Performance	To achieve high performance, the coordinator must be able to handle multiple concurrent requests and manage the state of each as they progress through the process. Also, the performance of any process will be limited by the slowest step, namely the slowest server in the process.
Scalability	The coordinator can be replicated to scale the application both up and out.

Architecture Design Example

Architecture Design Example
Stock Trading System

- **Response time goal:**
  - Users should be able to submit orders within 15 seconds
- **Solution :**
  - Decouple user Order Creation from Order Processing using a queue



Order Processing Architecture

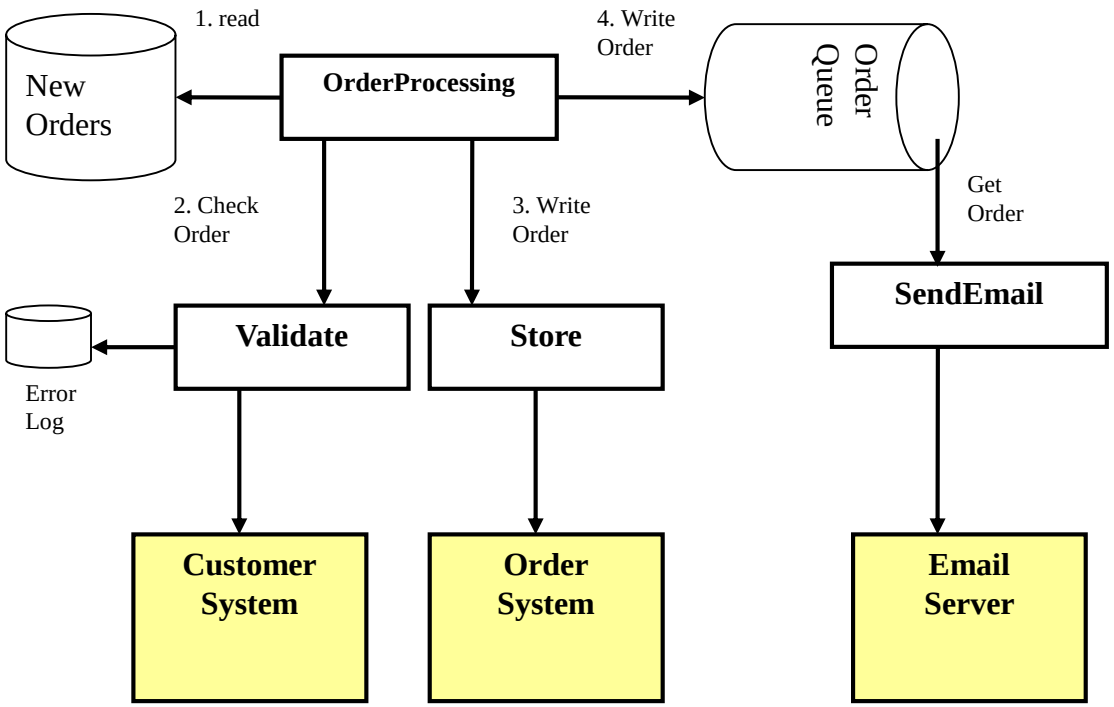


Figure Key

Existing Component

New Component

Dependency

Database

Persistent Queue

Order Processing Components

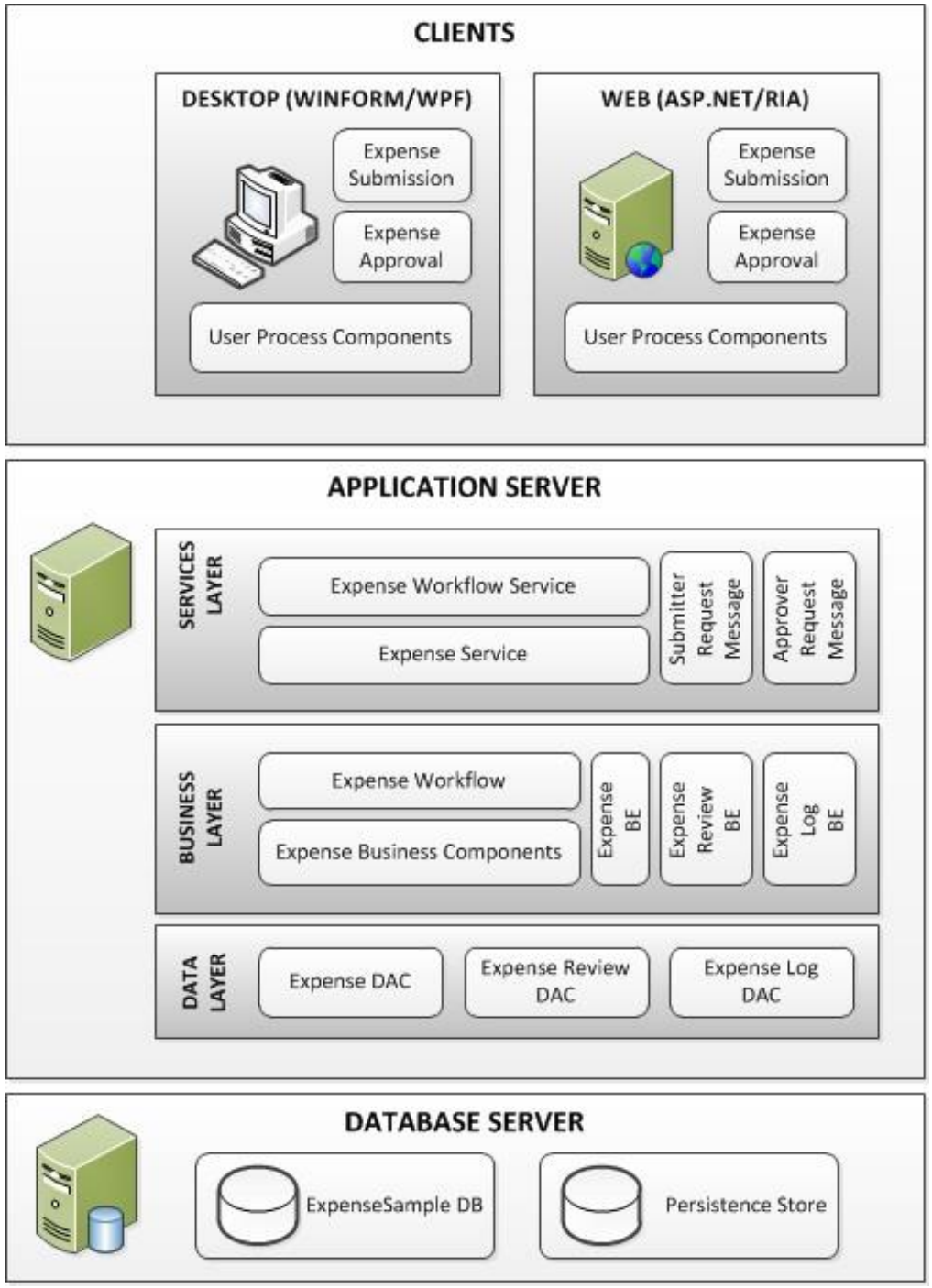
- Messaging based architecture
- Application components are:
  - **OrderProcessing:** responsible for accessing the new orders database, encapsulating the order processing logic, and writing to the queue.
  - **Validate:** encapsulates the responsibility of interacting with the customer system to carry out validation, and writing to the error logs if an order is invalid.
  - **Store:** responsibility of interacting with the order system to store the order data.
  - **SendEmail:** removes a message from the queue, formats an email message and sends it via an email server. It encapsulates all knowledge of the email format and email server access.
- Clear responsibilities and dependencies

Architecture Design Guidelines

- **Minimize dependencies between components**. Strive for a loosely coupled solution in which changes to one component do not ripple through the architecture, propagating across many components.
  - Remember, every time you change something, you have to re-test it.
- Design components that **encapsulate a highly "cohesive" set of responsibilities**. Cohesion is a measure of how well the parts of a component fit together.
- Isolate dependencies on any third party components.
- Minimize calls between components, as these can prove costly if the components are distributed.
- Designing an architecture is **iterative**, involving initial formulations and revisions

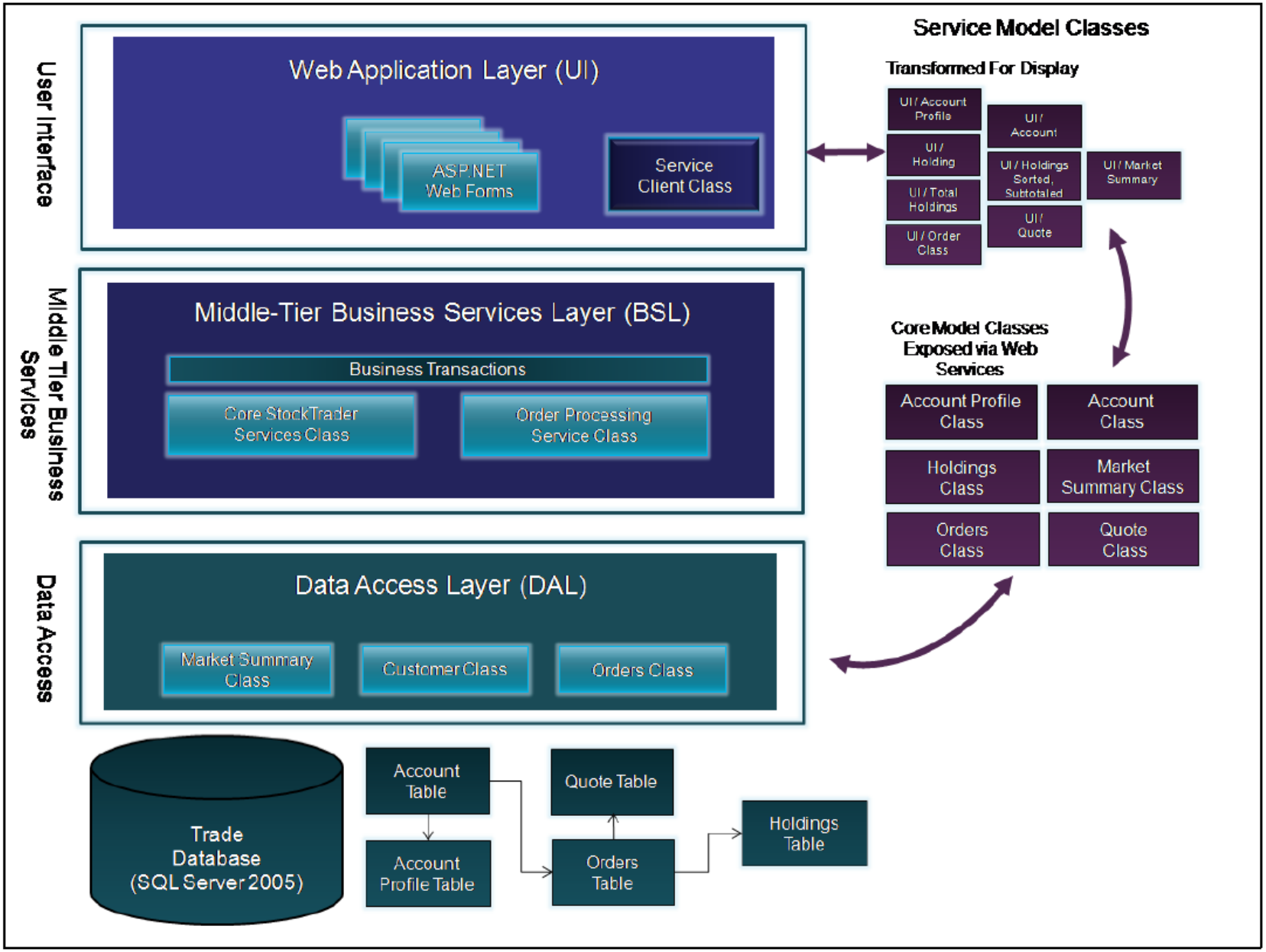
Project Milestone 3 Tips

Layered Architecture Sample



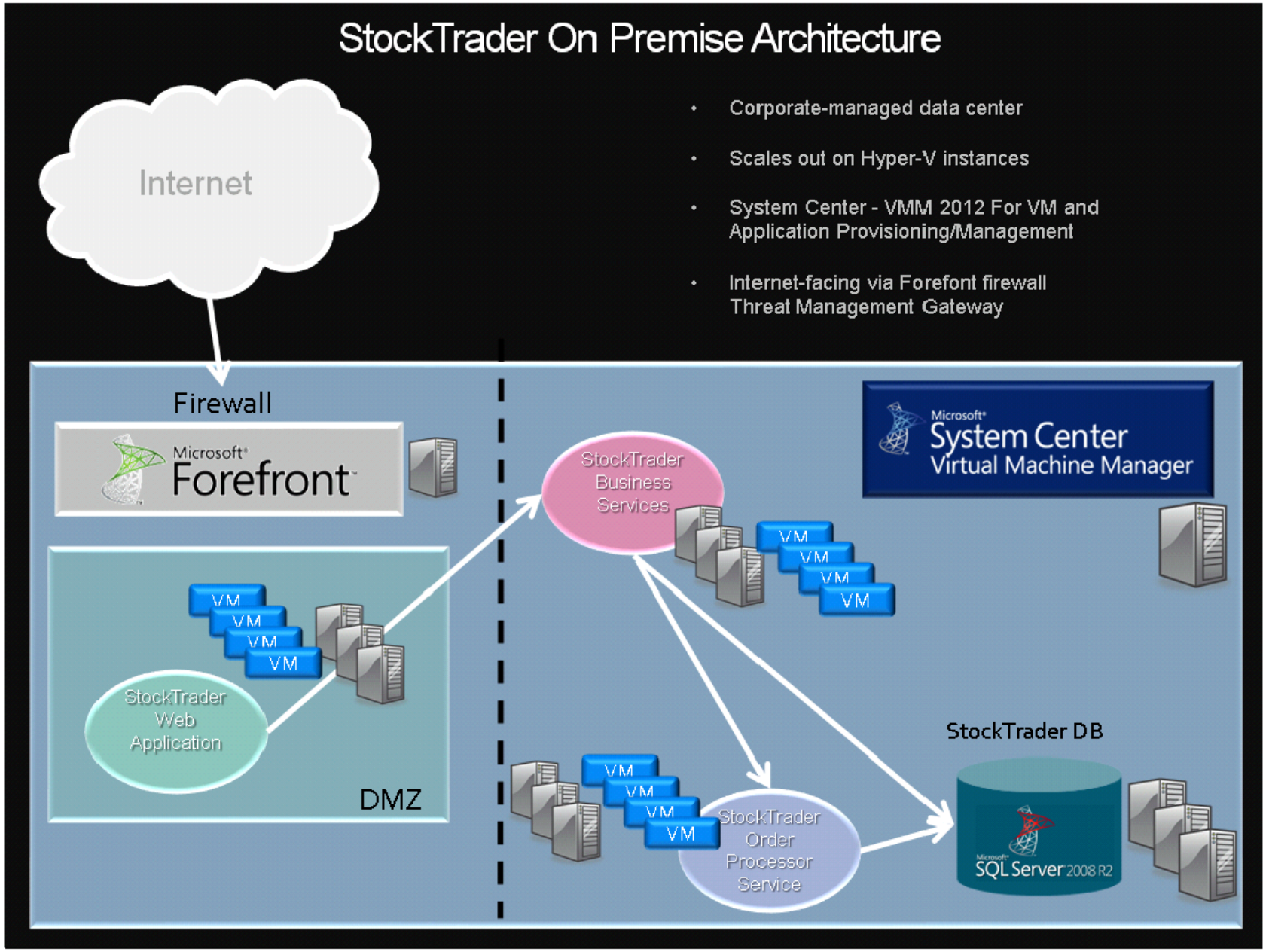
- More info @ http://layersample.codeplex.com/

Logical Architecture for Stock Trading

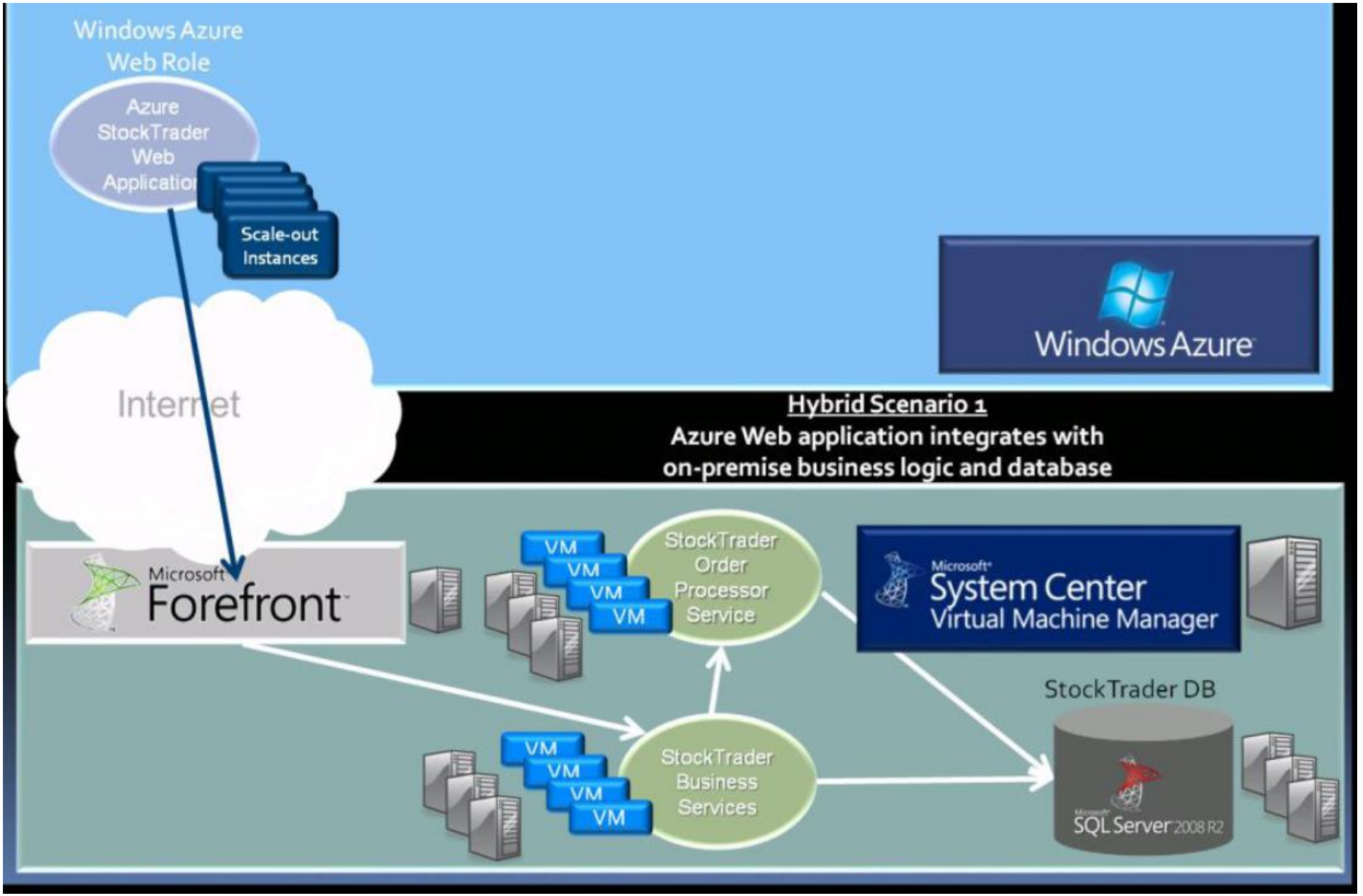


http://msdn.microsoft.com/en-us/netframework/bb499684.aspx

Deployment Architecture – On Premise



Deployment Architecture - Hybrid



Deployment Architecture – Cloud Deployment