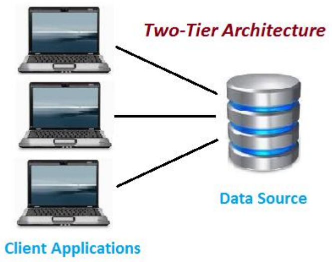
Enterprise Systems & Architecture

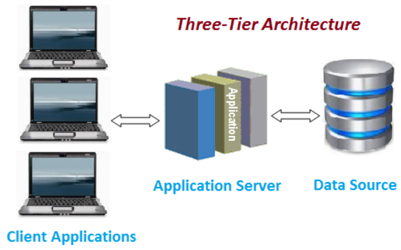
Enterprise Applications Integration (EAI)

Enterprise Architectures

- Where functionality resides
  - How network is used
  - What machines/devices are used, servers, clients, mainframes, virtual servers, “cloud”
- How the functionality is designed:
  - Individual programs: tightly coupled
  - Enterprise systems: loosely coupled

Mainframe

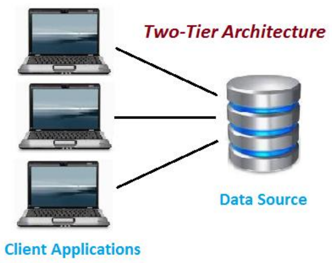


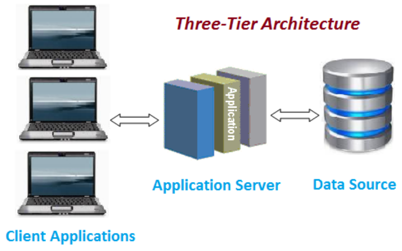


Add *Mobile Devices* & *Rich Web Applications* - gives rise to a variety of viable architectures

IBM's Z-14

Mainframe

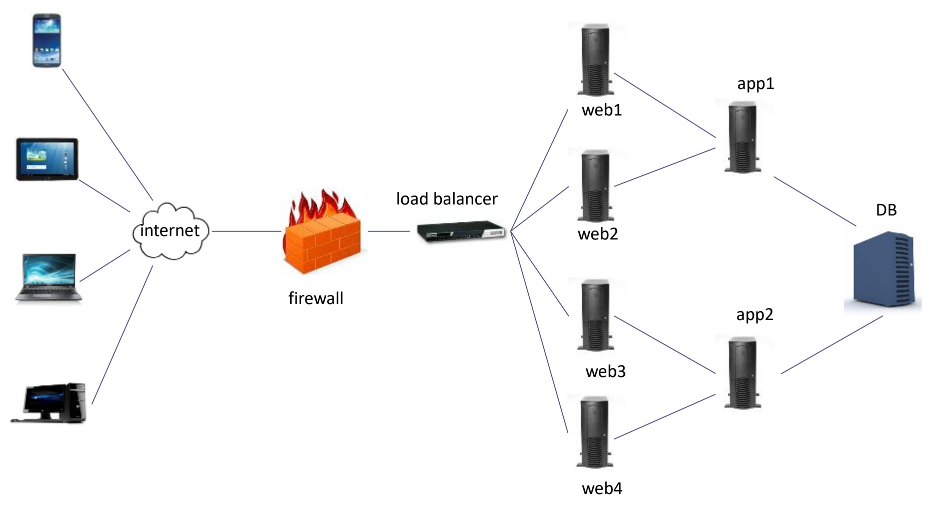




Enterprise Architecture N-Tier

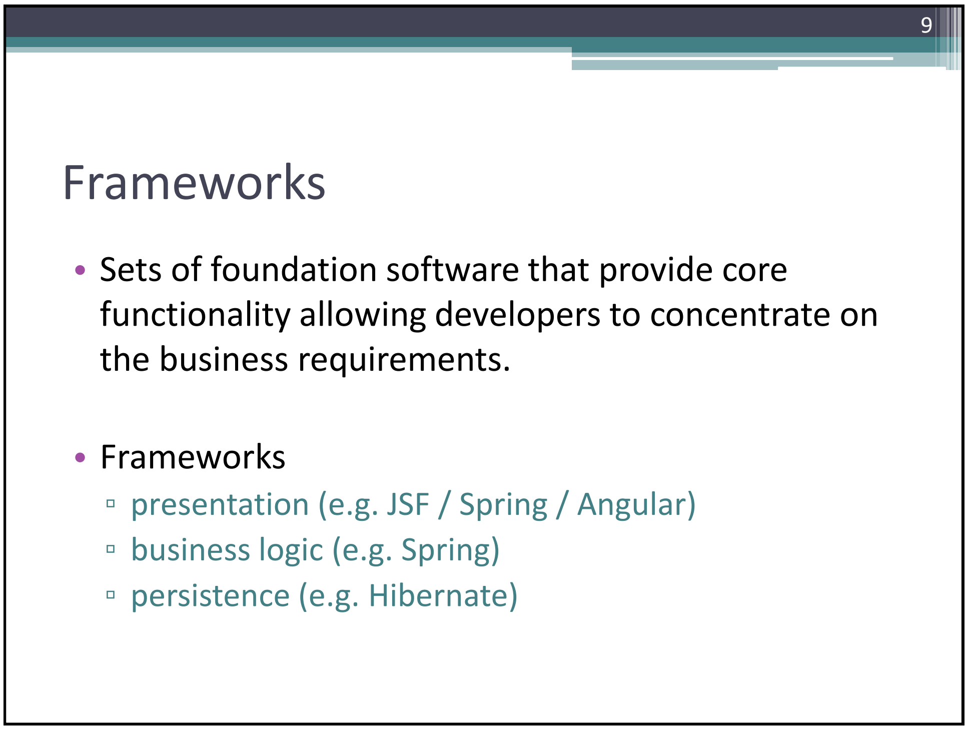
- Multi-Tier / Layered
  - Access
  - Presentation
  - Business
  - Persistence
  - Data Repository
  - Component/Service based
- Multiple client devices / systems
- Multiple datastores

A Basic Enterprise Architecture Example



Advantages of N-Tier Architecture

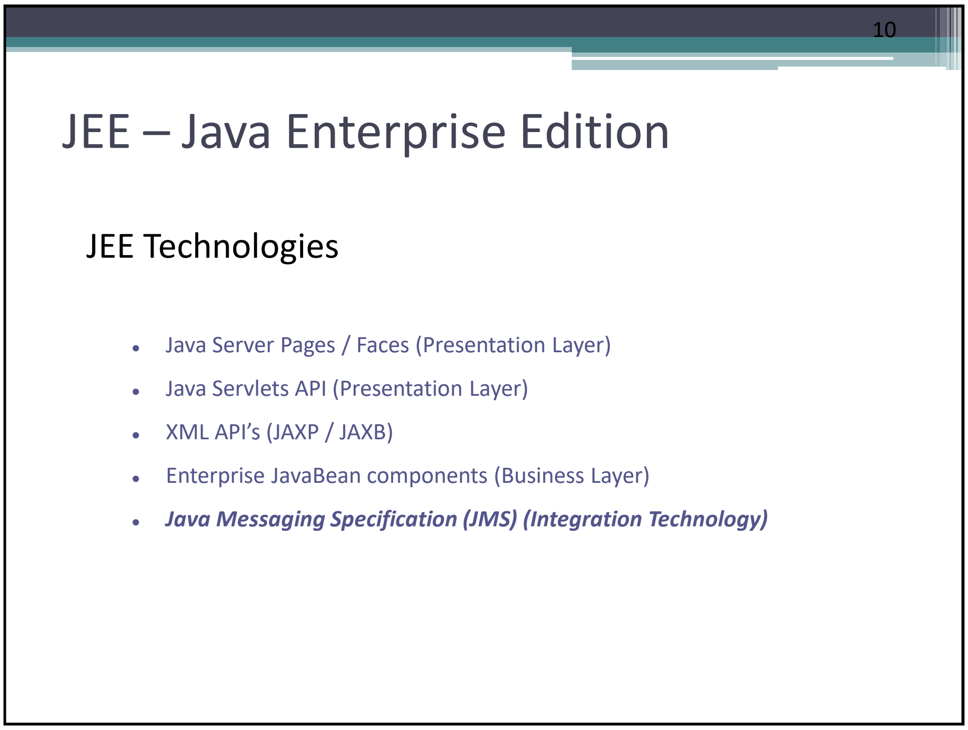
- Separation of concerns
  - Functionality is separated into its responsible parts
  - E.g. web page rendering vs business functions
- Maintainability
  - Identifying relevant implementation code easier
  - Testing specific functional concerns easier
  - Separation of developer skills possible
- Extendibility
  - Adding new code / refactoring existing code is made easier and less error prone
- Loose coupling
  - Design-time: Implementation can change with minimised impact on dependant functionality
  - Run-Time: Time/Location/Protocol independence of the functions gives a more robust system
- Functionality Access
  - Different client devices can access the same business functionality



Frameworks

- Sets of foundation software that provide core functionality allowing developers to concentrate on the business requirements.

- Frameworks
  - presentation (e.g. JSF / Spring / Angular)
  - business logic (e.g. Spring)
  - persistence (e.g. Hibernate)

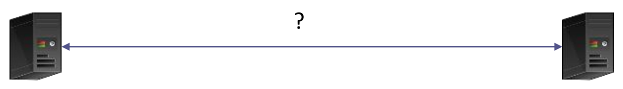


JEE – Java Enterprise Edition

JEE Technologies

- Java Server Pages / Faces (Presentation Layer)
- Java Servlets API (Presentation Layer)
- XML API's (JAXP / JAXB)
- Enterprise JavaBean components (Business Layer)
- ***Java Messaging Specification (JMS) (Integration Technology)***

Loose Coupling

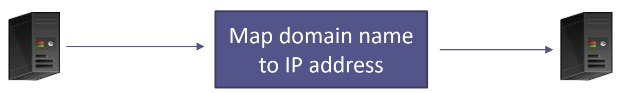


When two systems are tightly coupled there are a set of requiremnents that are forced into play:

- Data encoding used by both systems has to be the same
- The location of the system to communicate with is known
- The system that needs to be communicated with is currently up and running
- The two systems communicate using a common protocol/format

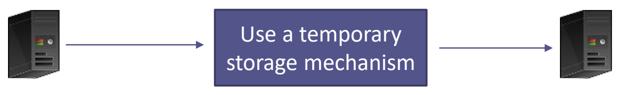
Loose Coupling

- To change from a tightly coupled situation to a loosely coupled situation we need to remove the these requirements.



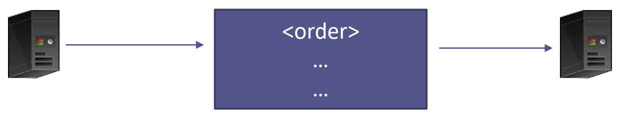
Location – introduce a level of indirection so that systems can be located / moved without the need for the other system to be aware.

Loose Coupling



Time – introduce a temporary storage mechanism so that systems can send / receive content asynchronously. Receiving system does not need to up and running at the time of the request but it will still be processed without error.

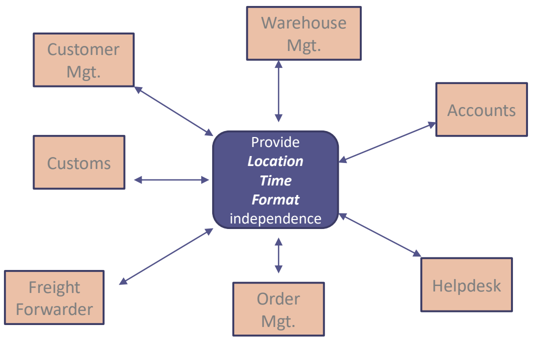
Loose Coupling



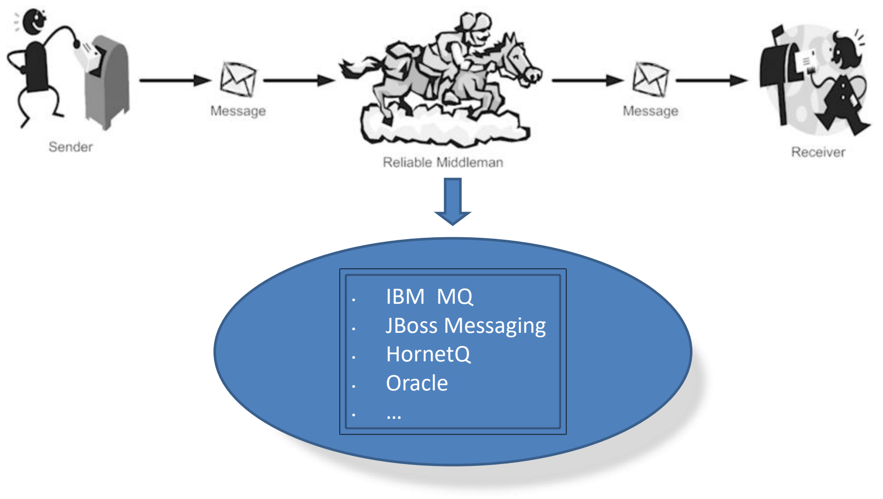
Format – introduce a mechanism to allow systems use their own data formats. Each system uses their own format but content is transformed / validated as needed without the systems themselves being aware.

Loose Coupling

- Reduces brittleness
- Allows scalability
- Can allow additional processing without the need for systems development

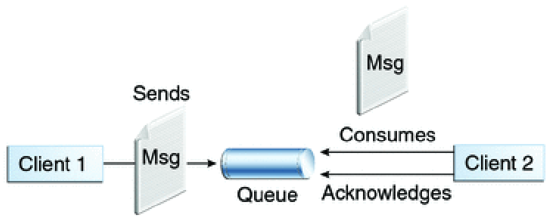


Message Oriented Middleware



Messaging Domains – Point to Point

- ***Point-to-Point (PTP)***
  - built around the concept of message queues
  - each message has only one consumer

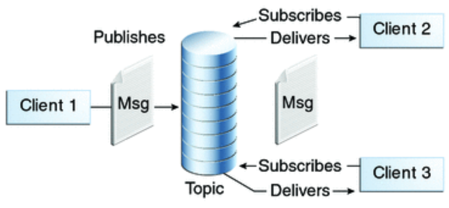


- Can have multiple clients sending messages
- Can have multiple clients as consumers
- Only *one consumer* will receive the message

Messaging Domains – Publish & Subscribe

- ***Publish-Subscribe*** systems
  - Uses a “topic” to send and receive messages
  - Each message has multiple consumers

  - A *Topic* is a type of message **destination**

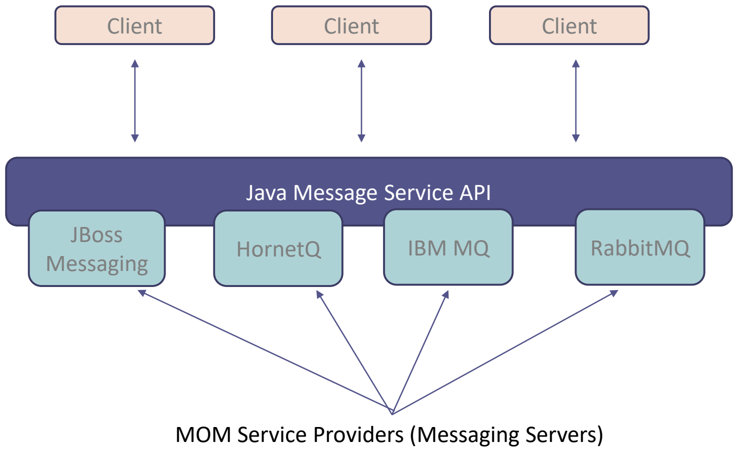


- Can have multiple clients publishing messages
- Each subscriber receives the message

Java Messaging Specification - JMS

- Provides a a standard java API to Message Oriented Middleware (MOM)
- A **specification** that describes a common way for Java programs to create, send, receive and read distributed enterprise messages
- Java clients that connect to the message broker are abstracted from the specific software implementation
- *Loosely coupled* communication
- *Asynchronous* messaging
- *Reliable* delivery - A message is guaranteed to be delivered once and only once.

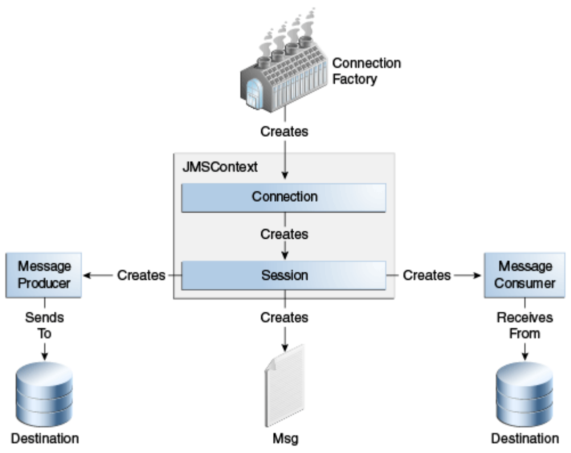
Messaging – Message Oriented Middleware *(MOM)* & JMS



Message Consumptions

- Synchronously
  - A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method.
  - The receive method can *block* until a message arrives or can time out if a message does not arrive within a specified time limit.
- Asynchronously
  - A client can register a *message listener* with a consumer.
  - Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's onMessage() method.

JMS API Programming Model

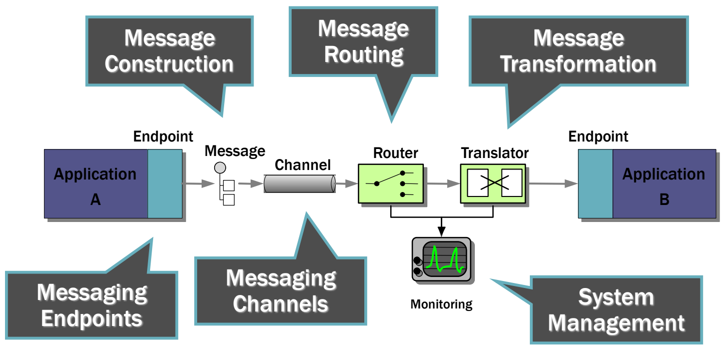


Ref: https://docs.oracle.com/javaee/7/tutorial/jms-concepts003.htm#BNCEH

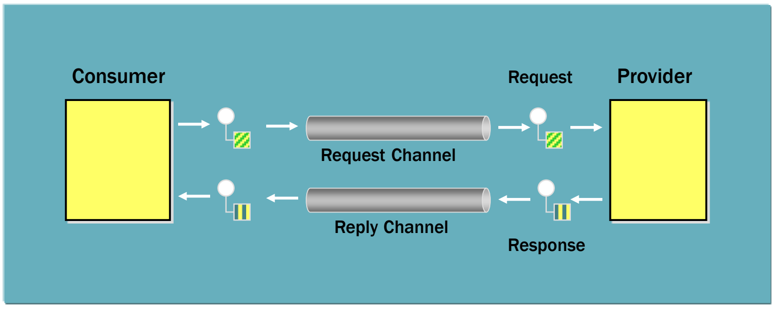
Enterprise Integration Patterns

Ref:
http://www.eaipatterns.com/toc.html
http://erik.doernenburg.com

Enterprise Integration Patterns

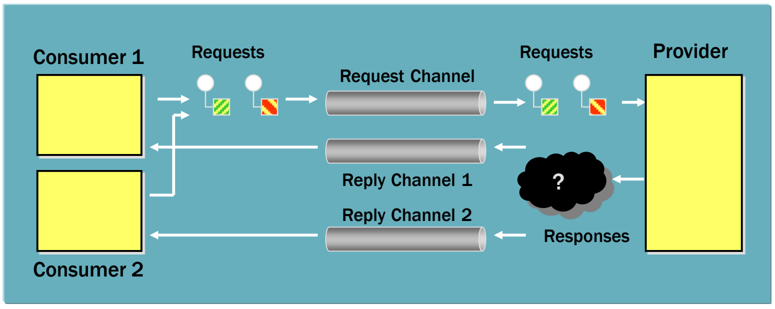


Pattern: *Request-Response*



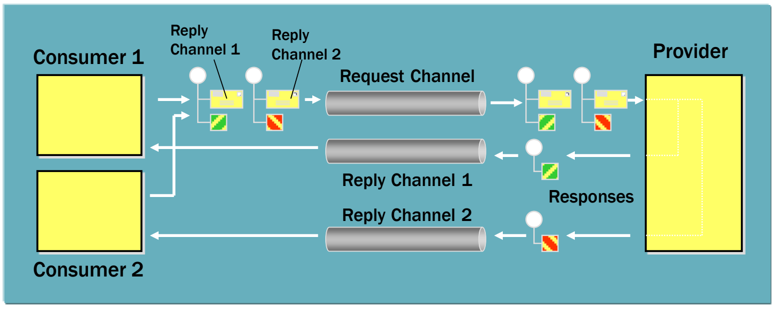
- Service Consumer and Provider (similar to RPC)
- Channels are unidirectional
- Two asynchronous point-to-point channels
- Separate request and response messages

Multiple consumers



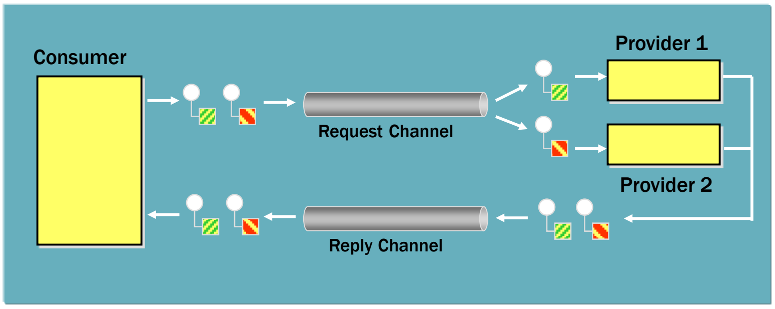
- Each consumer has its own reply queue
- But how does the provider send the response?
  - Could send to all consumers (very inefficient)
  - Hard code (violates principle of context-free service)

Pattern: *Return Address*



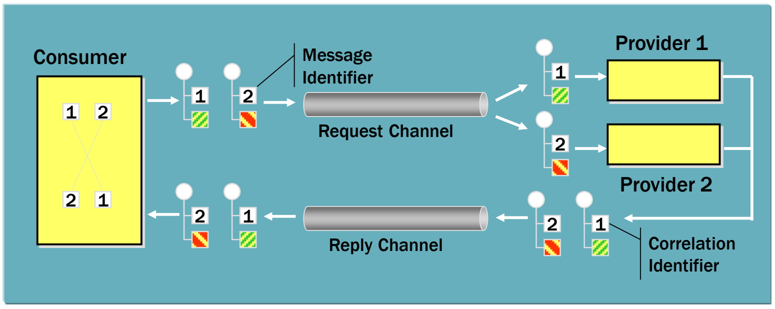
- Consumer specifies *Return Address* (the reply channel) in the request message
- Service provider sends response message to specified channel

Load-balanced service providers



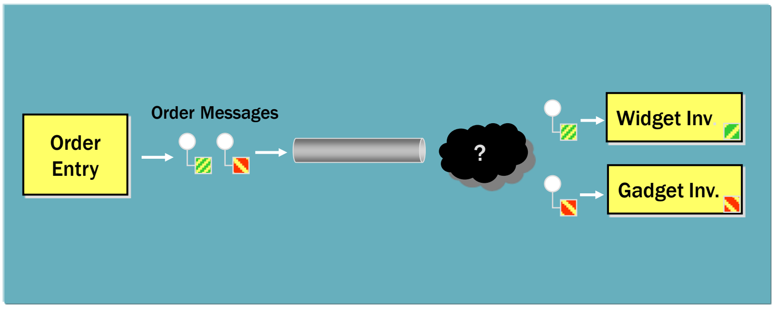
- Request message can be handled by multiple providers
- Only one service receives each request message
- But what if the response messages are out of order?

Pattern: *Correlation Identifier*



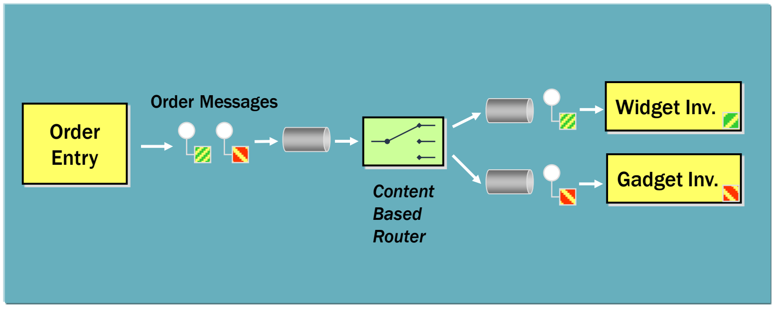
- Consumer assigns a unique identifier to each message
  - Identifier can be an arbitrary ID, a GUID, a business key
- Provider copies the ID to the response message
- Consumer can match request and response

E.g. Order Entry system - Multiple specialised providers



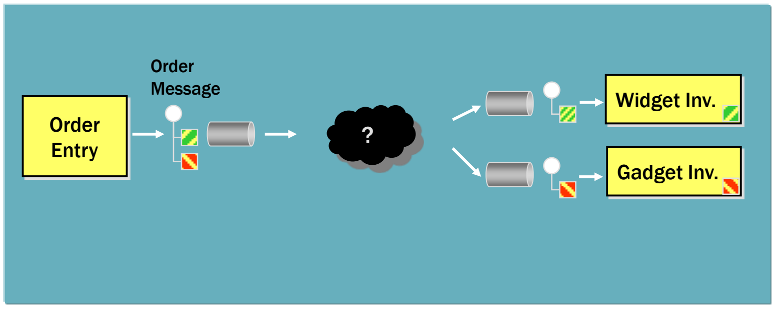
- Each provider can only handle a specific type of message
- Route the request to the "appropriate" provider. But how?
  - *Do not want to burden sender with decision*
  - *Letting providers "pick out" messages requires coordination*

Pattern: *Content-Based Router*



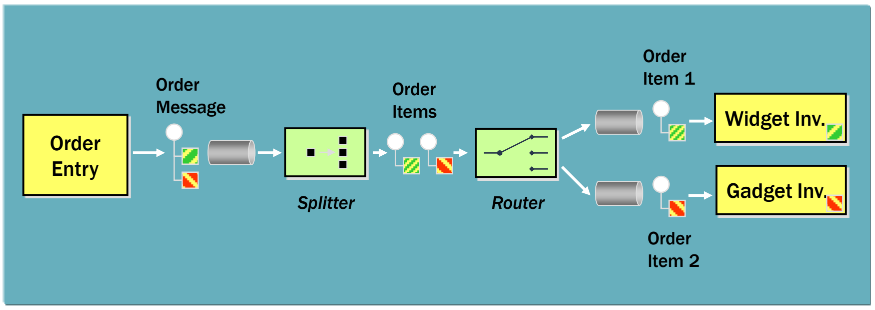
- Insert a content-based router
- Routers forward incoming messages to different channels
- Message content not changed

Composite messages



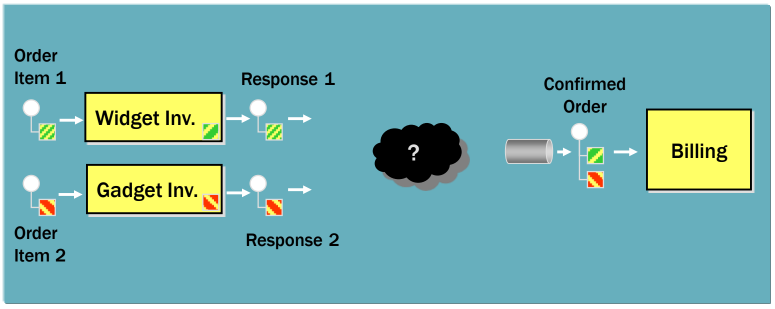
- How can we process a message that contains multiple elements?

Pattern: *Splitter & Router*



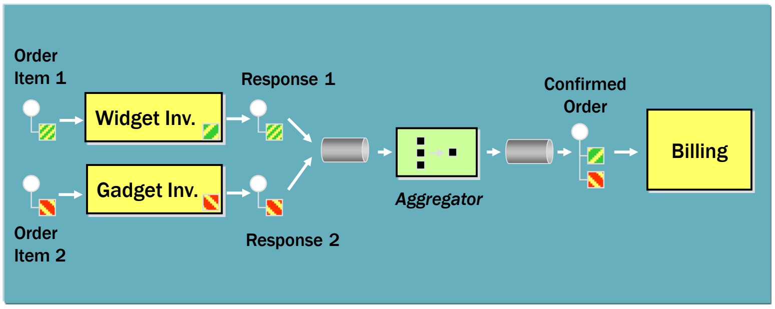
- Use a splitter to break out the composite message into a series of individual messages
- Then use a router to route the individual messages as before
- Note that two patterns are composed

Producing a single response



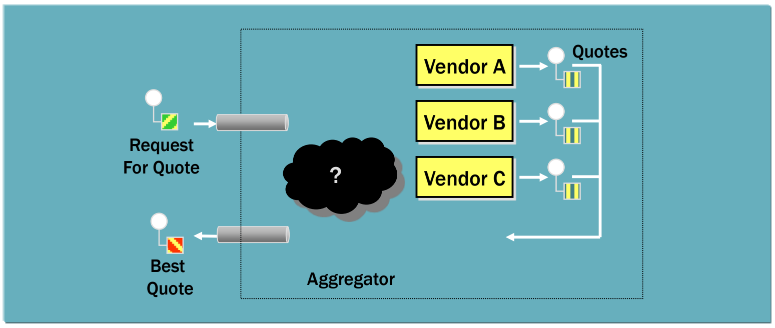
- How to combine the results of individual but related messages?
  - Messages can be out-of-order, delayed
  - Multiple conversations can be intermixed

Pattern: *Aggregator*



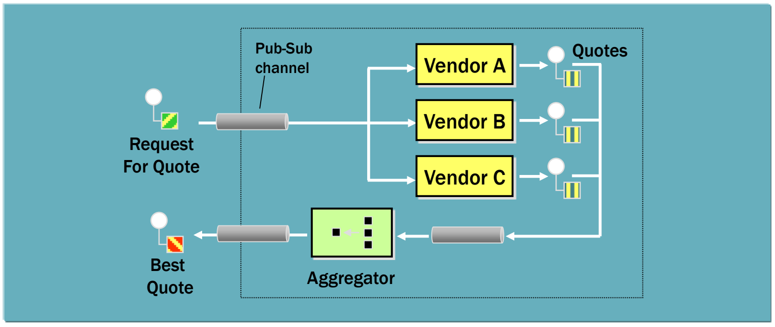
- Use a stateful filter, an Aggregator
- Collects and stores messages until a complete set has been received (completeness condition)
- Publishes a single message created from the individual messages (aggregation algorithm)

Communicating with multiple parties



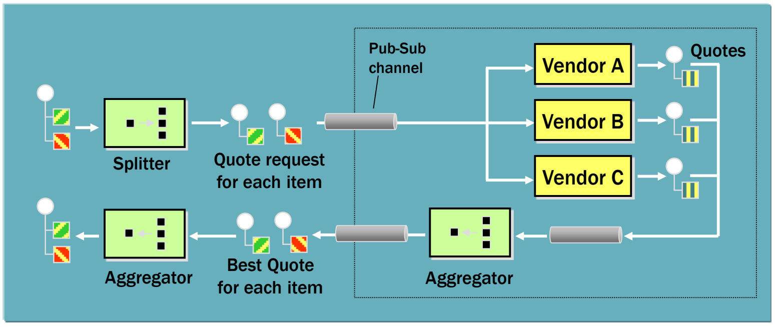
- How to send a message to a dynamic set of recipients?
- And return a single response message?

Pattern: *Scatter-Gather*



- Send message to a pub-sub channel
- Interested recipients subscribe to a "topic"
- Aggregator collects individual response messages
  - may not wait for all quotes, only returns one quote

Complex composition



- Receive an order message
- Use splitter to create one message per item
- Send to scatter/gather which returns "best quote" message
- Aggregate to create quoted order message