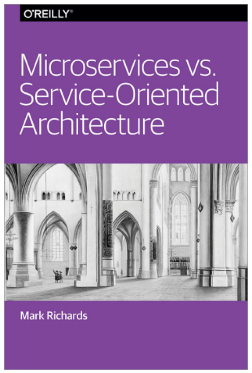
Enterprise architecture patterns

UA.DETI.IES - 2019/20

Resources & Credits

- Ian Sommerville,
  Software Engineering, 10th Edition,
  Pearson, 2016
  - (chapters 6, 17, 18)



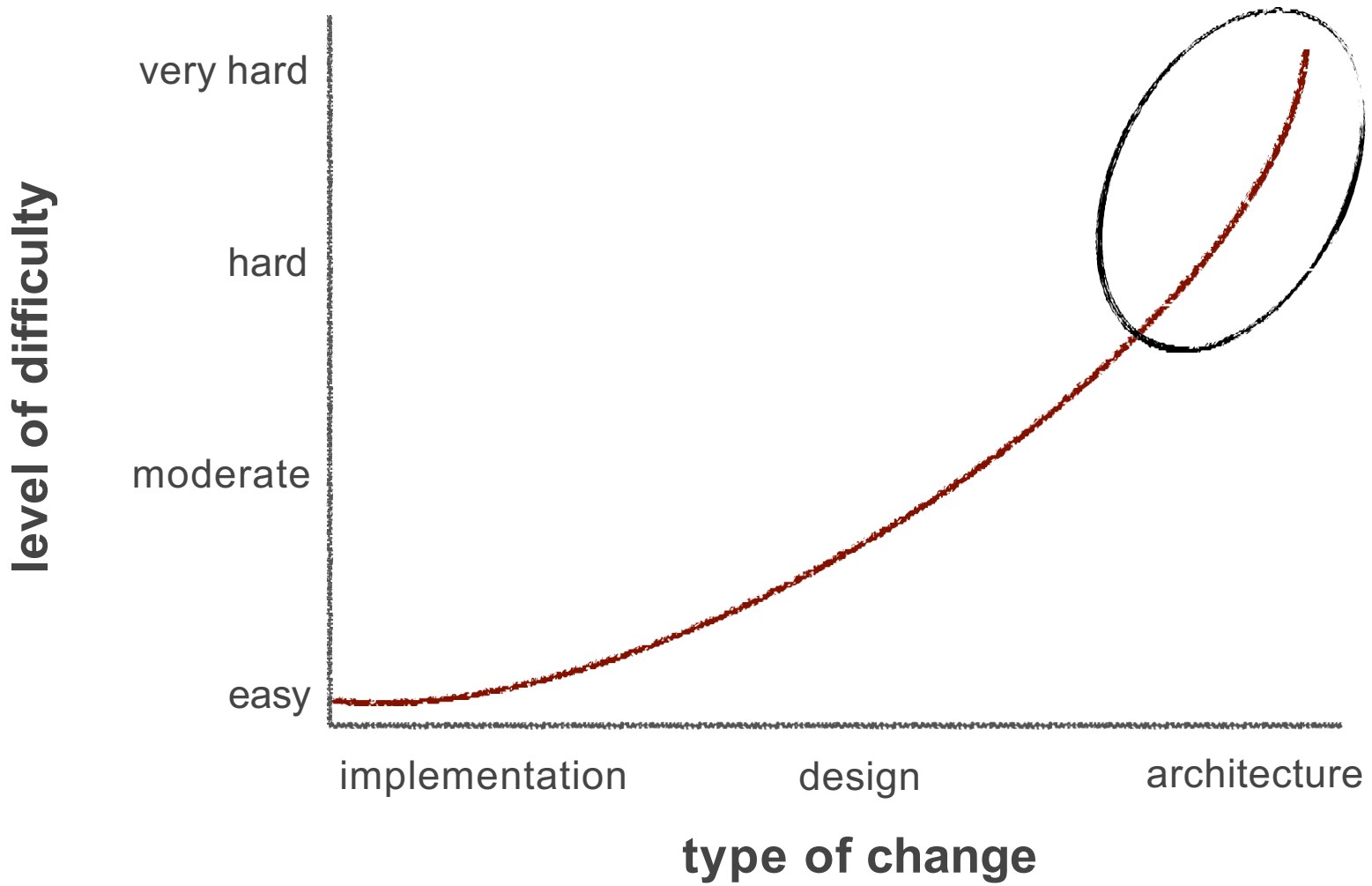
- Mark Richards, Microservices vs. Service-Oriented Architecture, O'Reilly, 2015

- Mark Richards,
  Software Architecture Fundamentals Workshop
  https://conferences.oreilly.com/software-architecture

Software architecture?

- Architecture is the highest level concept of the expert developers.
  - "In most successful software projects, the expert developers working on that project have a shared understanding of the system design. This shared understanding is called 'architecture.' This understanding includes how the system is divided into components and how the components interact through interfaces. These components are usually composed of smaller components, but the architecture only includes the components and interfaces that are understood by all the developers."
  - http://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf

Architecture decisions



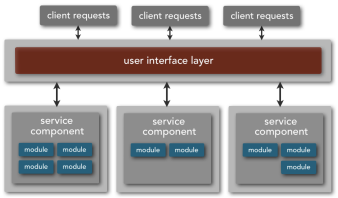
Architecture decisions

- the decision to use java server faces as your web framework
  - vs.

- the decision to use a web-based user interface for your application

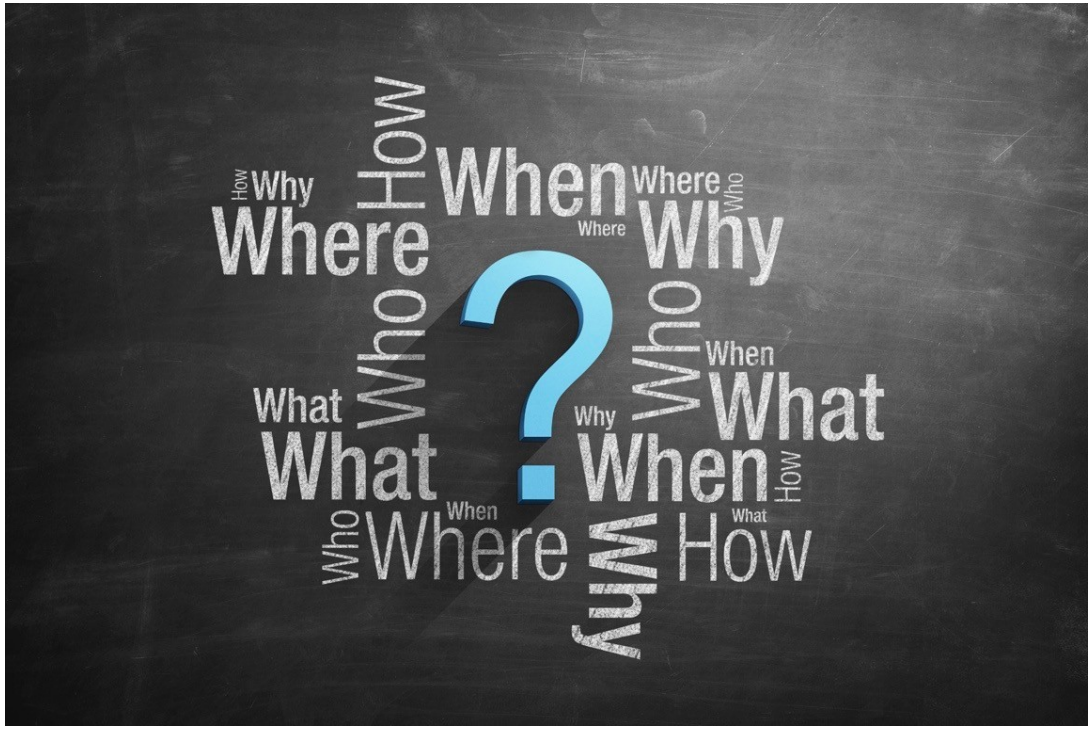
- the decision to use rest to communicate between distributed components
  - vs.

- the decision that components should be distributed remotely for better scalability



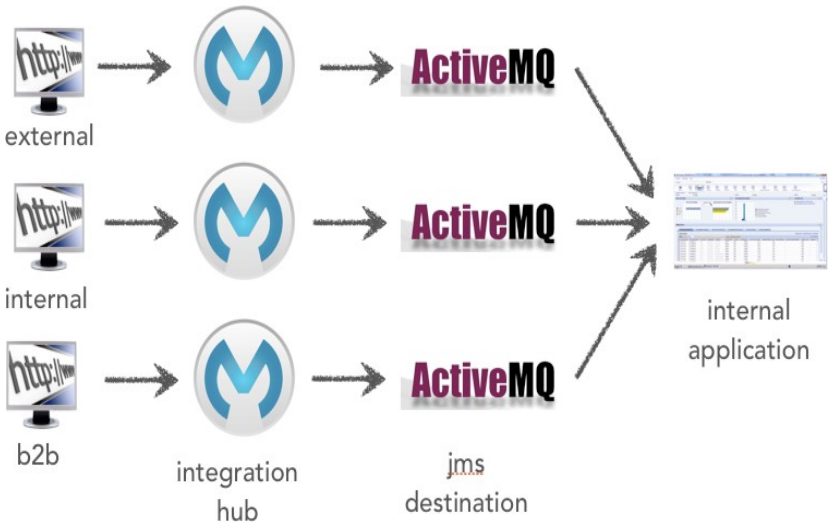
Justifying architecture decisions

- **Groundhog day** anti-pattern
  - no one understands why a decision was made so it keeps getting discussed over and over and over...

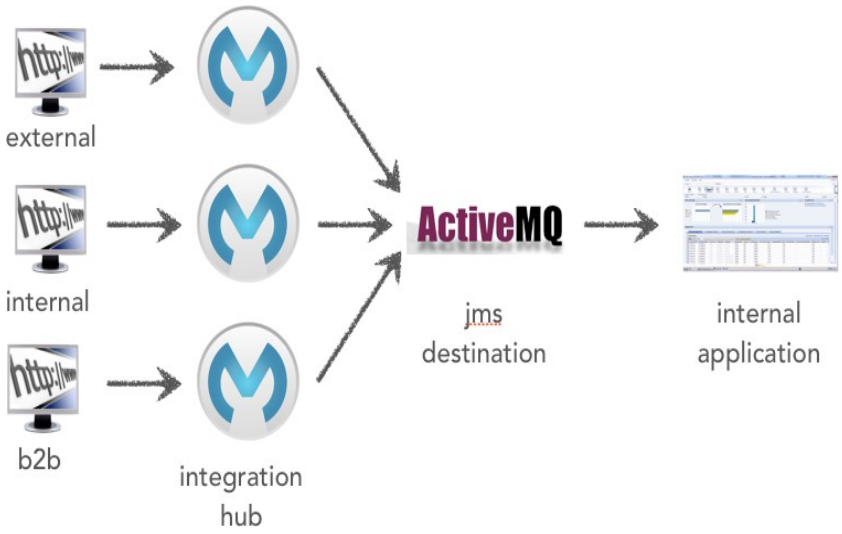


Justifying decisions - Example

- dedicated broker instances?



- or a centralized broker?



Justifying decisions - Example

- **Identify the conditions and constraints**
  - broker only used for hub access
  - low transaction volumes expected
  - application logic may be shared between different types of client applications (e.g., internal and external)

- **Analyze each option based on conditions**
  - broker usage and purpose
  - overall message throughput
  - internal application coupling
  - single point of failure
  - performance bottleneck

Justifying decisions - Example

- Architecture **decision**:
  - centralized broker

- **Justification**:
  - the internal applications should not have to know from which broker instance the request came from.
  - only a single broker connection is needed, allowing for the expansion of additional hub instances with no application changes.
  - due to low request volumes the performance bottleneck is not an issue; single point of failure can be addressed through failover nodes or clustering.

Documenting and communicating decisions

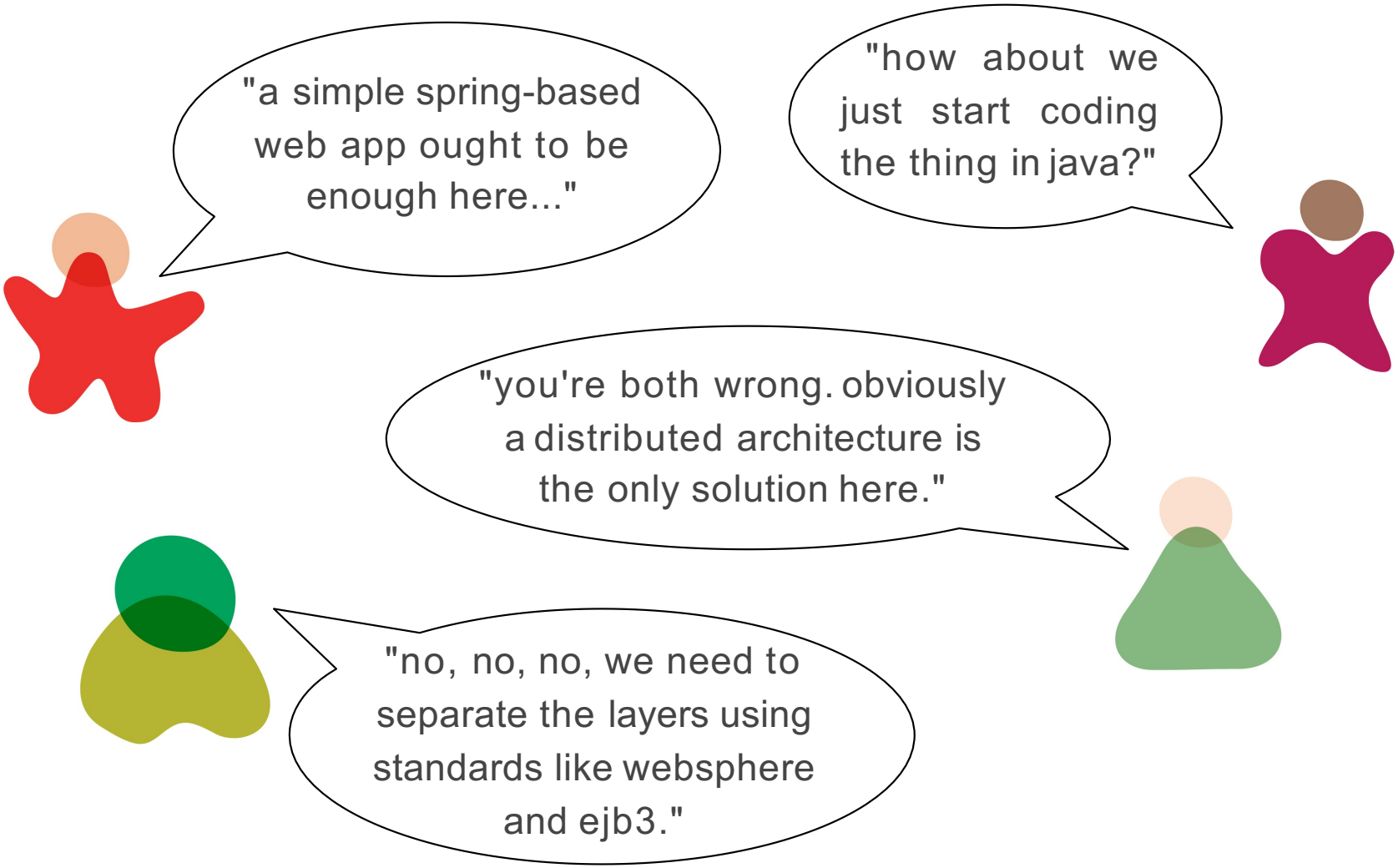
- Establish early on where your decisions will be documented and make sure every team member knows where to go to find them
  - In a central document or wiki, rather than multiple files spread throughout a crowded shared drive

- Email-driven solutions (.. not 😐)
  - people forget, lose, or don't know an architecture decision was made, and therefore don't implement the architecture correctly

Avoid Witches brew architecture

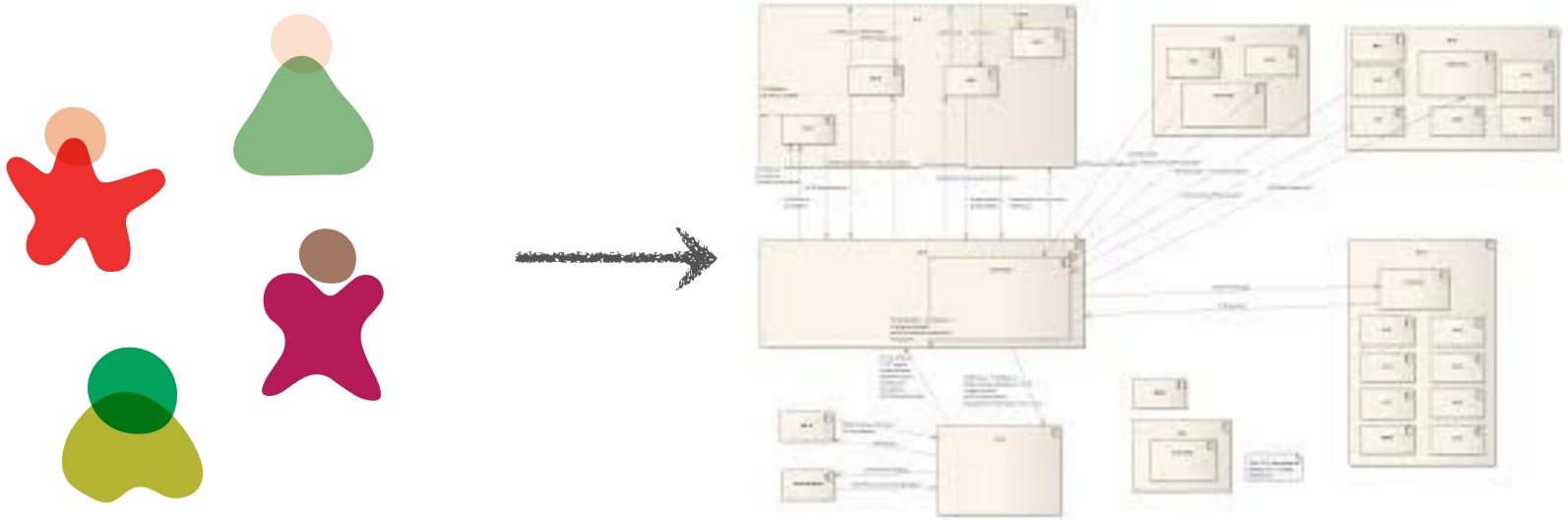
- Architectures designed by groups resulting in a complex mixture of ideas and no clear vision

The problem

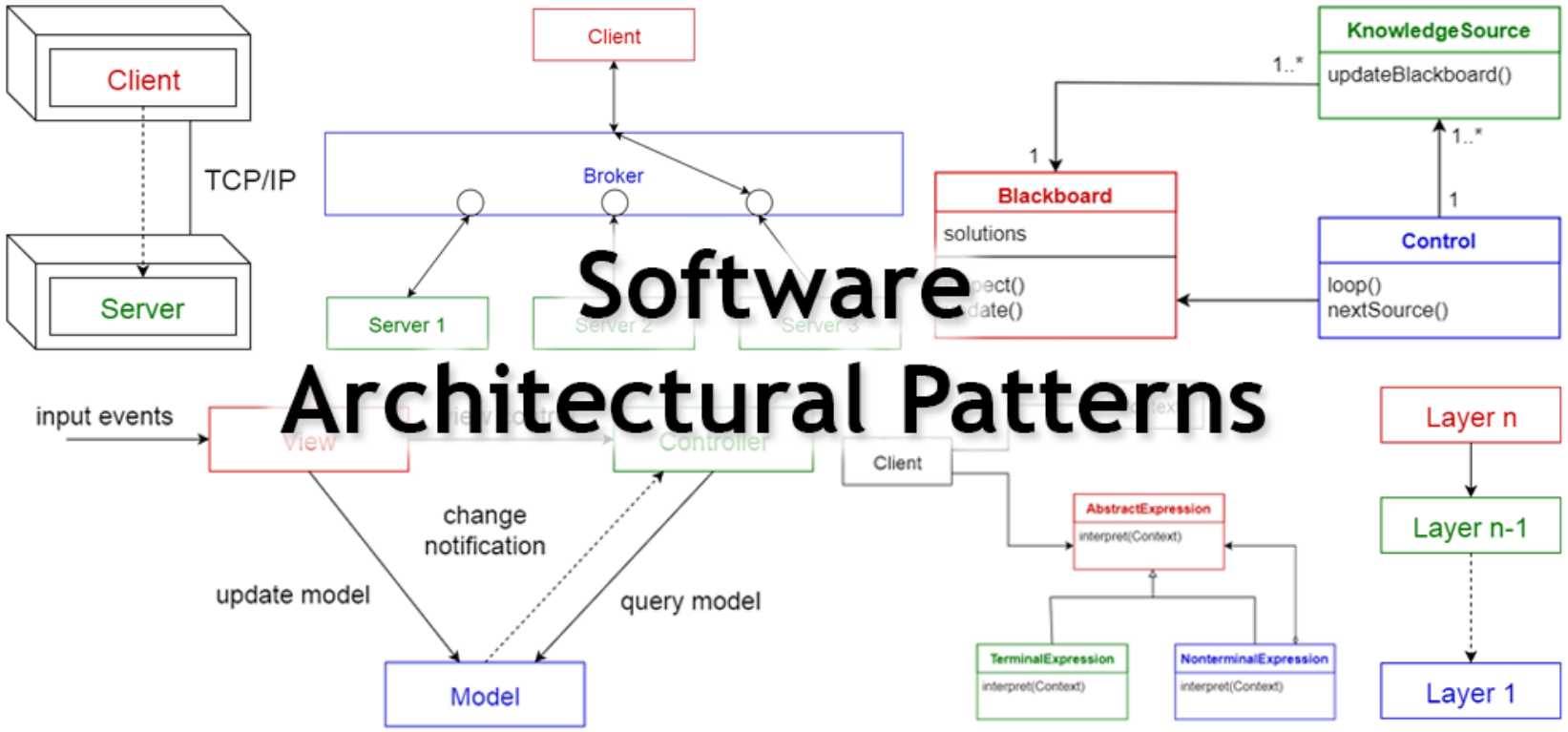

"a simple spring-based web app ought to be enough here..."
"how about we just start coding the thing in java?"
"you're both wrong. obviously a distributed architecture is the only solution here."
"no, no, no, we need to separate the layers using standards like websphere and ejb3."

The goal

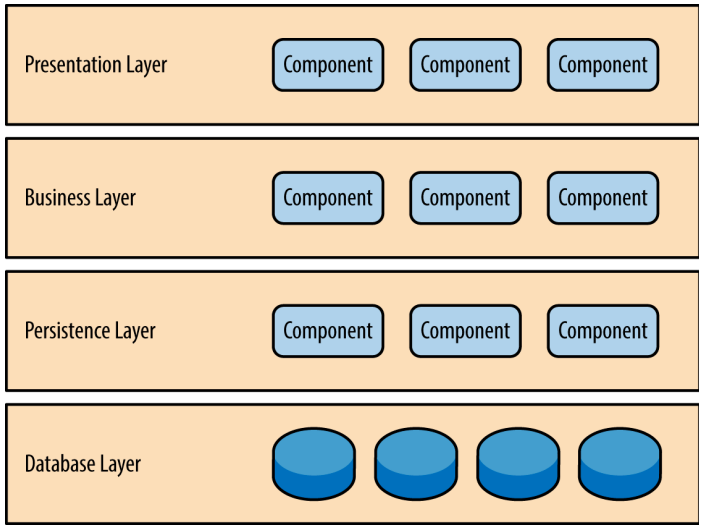
- Using collective knowledge and experience to arrive at a unified vision for the architecture

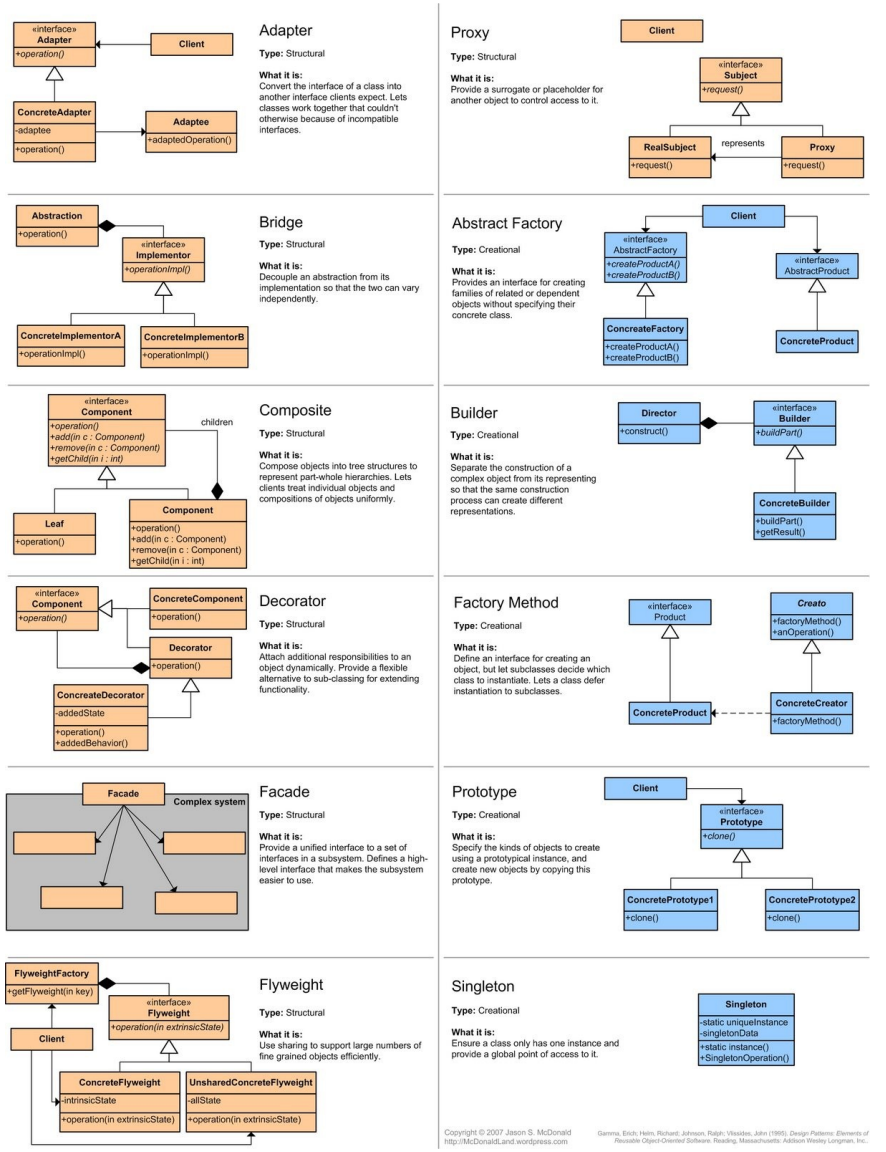


Architecture Patterns



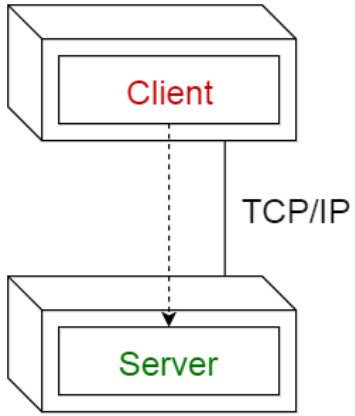
Architectures, Components, Classes



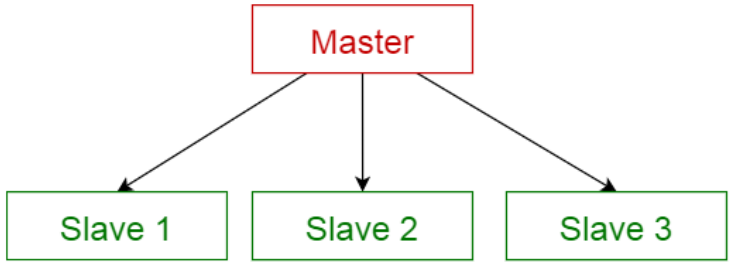


(Simpler) Architecture Patterns

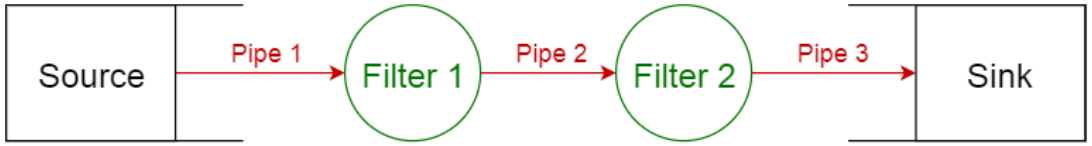
- Client-server pattern



- Master-slave pattern



- Pipe-filter pattern

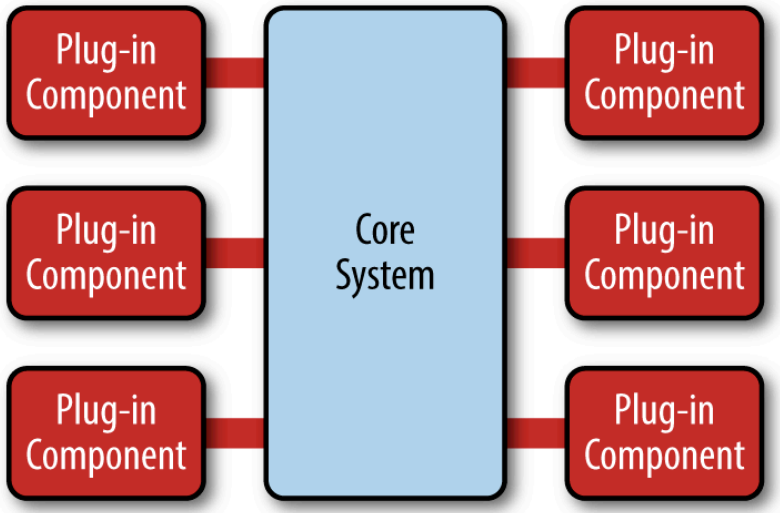


Architecture Patterns

- Microkernel Architecture
- Layered Architecture
- Event-Driven Architecture
- Service-oriented Architecture
- Microservices Architecture
- Space-Based Architecture

Microkernel Architecture

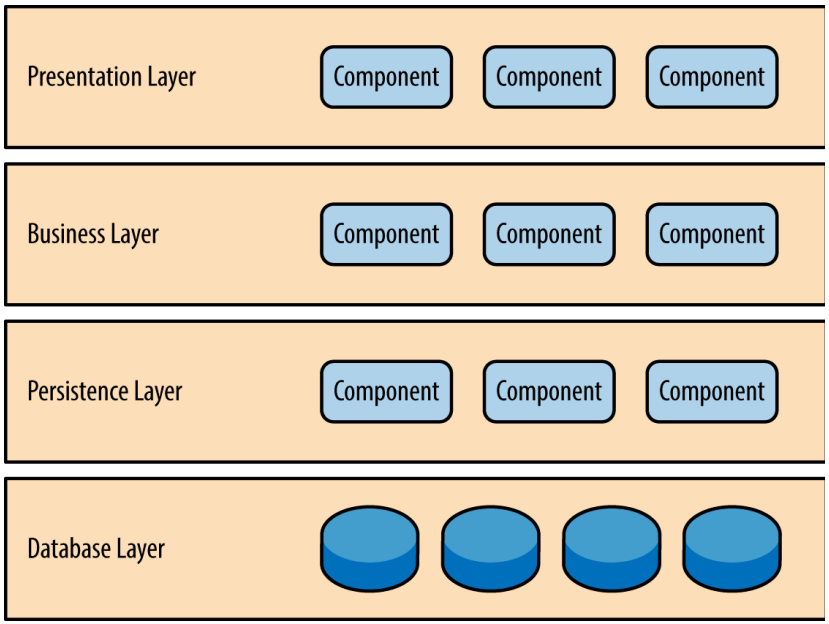
- **Core** application (stable) + **plugins** (dynamic)
  - New application features are added as plug-ins
  - Extensibility, feature separation and isolation.



- It can be embedded or used as part of another architecture pattern.
- A good first choice for product-based applications.

Layered Architecture

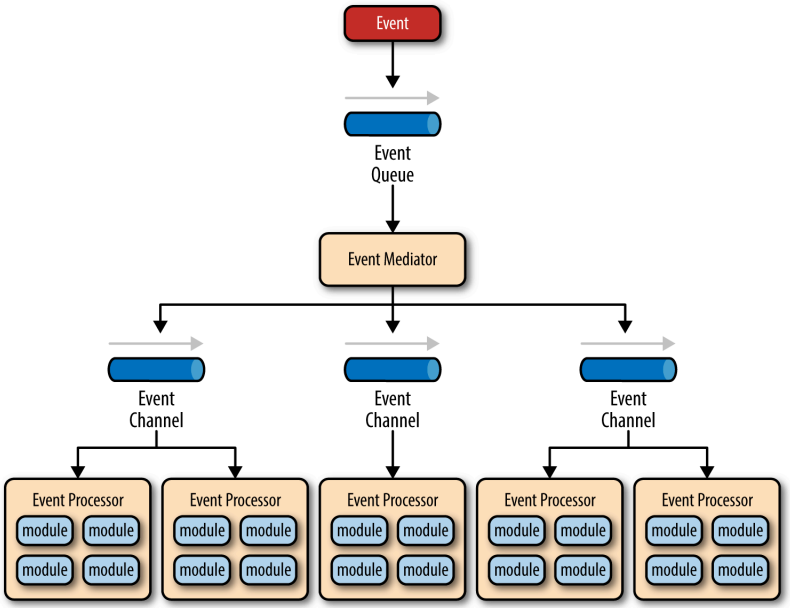
- The most common n-tier architecture pattern.
- A *de facto* standard for most Java EE applications.
- Each layer has a specific role and responsibility.
  - Separation of concerns
- Closely matches the organizational structures found in most companies.
- Solid general-purpose pattern
  - making it a good starting point for most applications



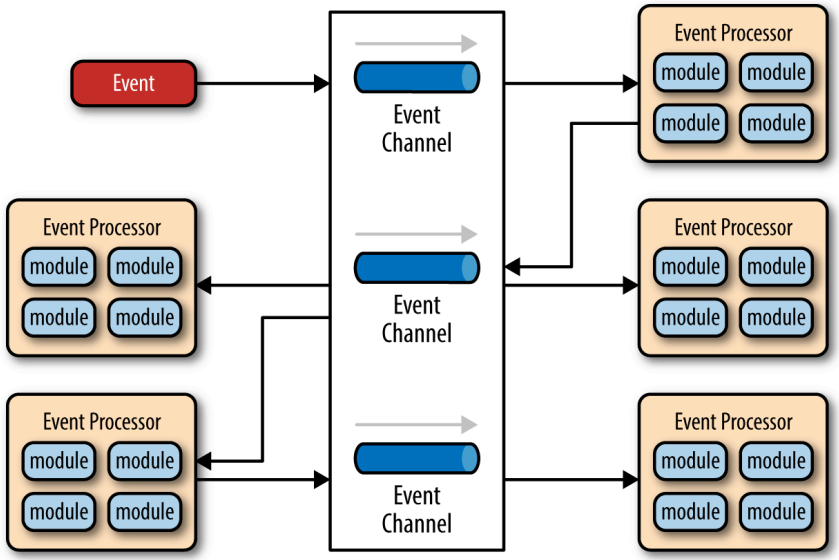
Event-driven Architecture

- Distributed and asynchronous
  - used to produce highly scalable applications.
  - AKA, message-driven or stream processing

Mediator topology



Broker topology



Event-driven Architecture

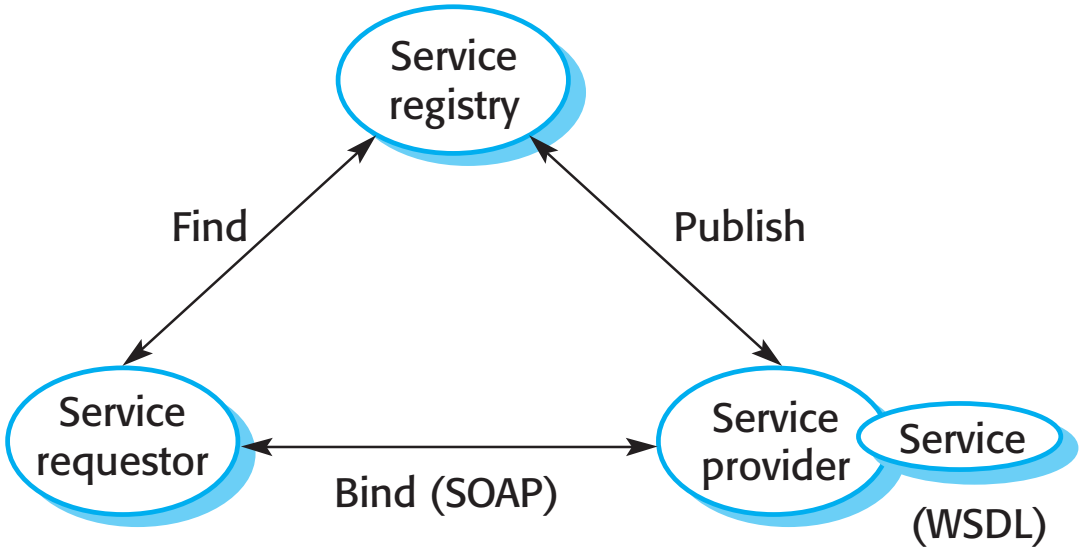
- The event-driven architecture pattern is a relatively **complex** pattern to implement, primarily due to its asynchronous distributed nature.
- **Lack of atomic transactions** for a single business process.
  - Event processor components are highly decoupled and distributed,
  - it is very difficult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

Architecture Patterns (*recap*)

- Microkernel Architecture
- Layered Architecture
- Event-Driven Architecture
- **Service-oriented Architecture**
- Microservices Architecture
- Space-Based Architecture

Service-Oriented Architecture (SOA)

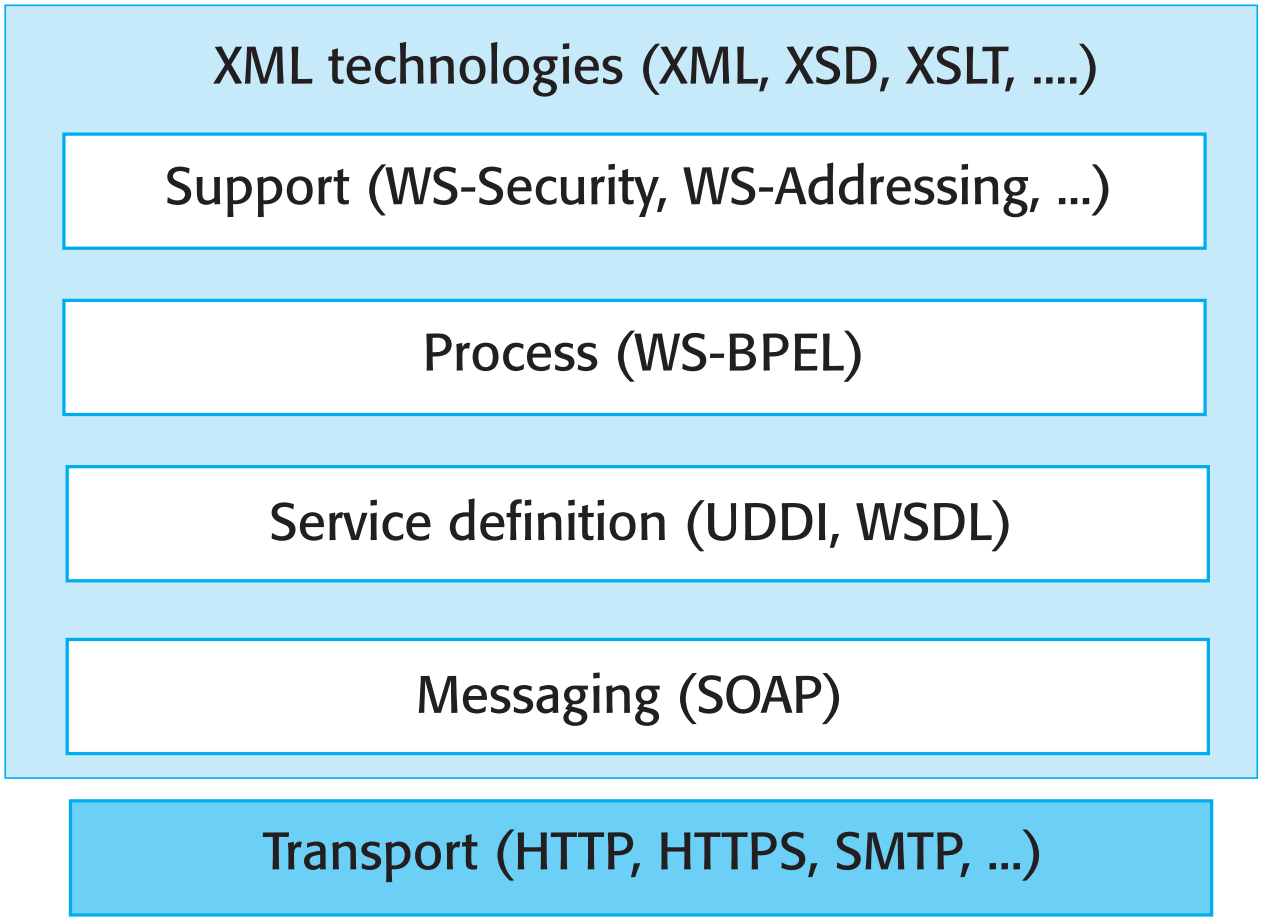
- A means of developing distributed systems where the components are stand-alone services
- Key characteristics
  - Standardized Service Contracts
  - Language-independent
  - Loose Coupling
  - Abstraction
  - Reusability
  - Autonomy
  - Statelessness
  - Discoverability
  - Composability



Key standards

- SOAP
  - A message exchange standard that supports service communication
- WSDL (Web Service Definition Language)
  - This standard allows a service interface and its bindings to be defined
- WS-BPEL
  - A standard for workflow languages used to define service composition

Web service standards



SOAP (Simple Object Access Protocol)

- A protocol based on XML language.
  - Platform and language independent communication.
- The structure of SOAP messages:
  - An Envelope element that identifies the XML document as a SOAP message
  - A Header element that contains header attributes
  - A Body element that contains call and response information
  - A Fault element containing errors and status information

SOAP (Simple Object Access Protocol)

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Header>
...
</soap:Header>

<soap:Body>
...
    <soap:Fault>
      ...
    </soap:Fault>
</soap:Body>

</soap:Envelope>
```

RESTful web services

- Current web services standards have been criticized as 'heavyweight' standards that are over-general and inefficient.
- REST (REpresentational State Transfer) is an architectural style based on transferring representations of resources from a server to a client.
- This style underlies the web as a whole and is simpler than SOAP/WSDL for implementing web services.
- RESTful services involve a lower overhead than so-called 'big web services' and are used by many organizations implementing service-based systems.

REST (REpresentational State Transfer)

- When REST is used, requests can be sent to various endpoints through GET, PUT, POST and DELETE HTTP requests.
- Provides a stateless, simpler and lightweight way of communicating with a system.
- It uses JSON format to send and receive data.
  - This format is a simple text containing a series of attribute-value pairs

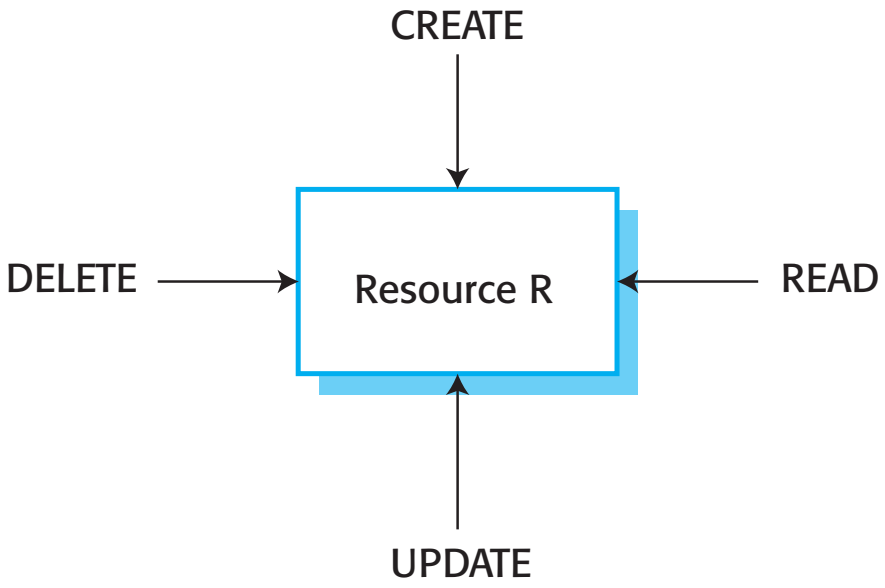
```
{ "trade":
   {
       "cust_id": "12345",
       "cusip": "037833100",
       "shares": "1000",
       "trade_dt": "10/12/15"
   }
}
```

Resources

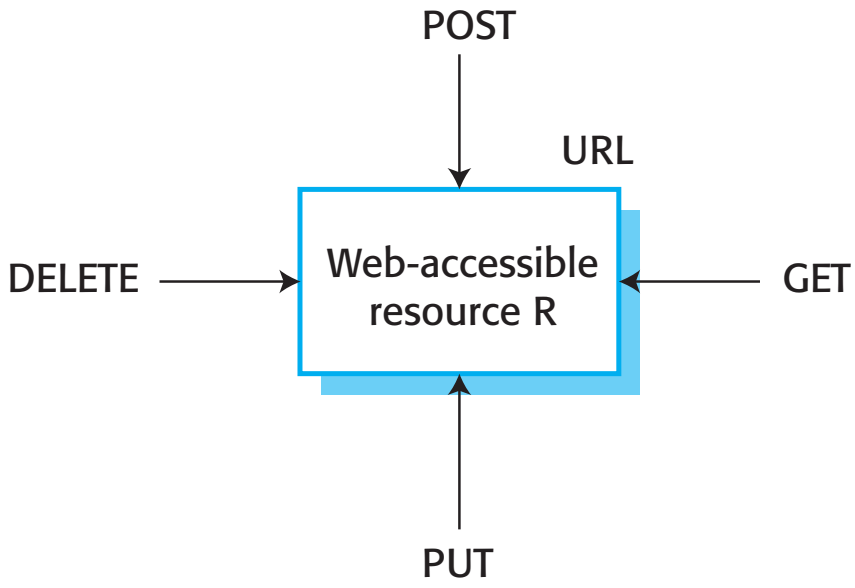
- The fundamental element in a RESTful architecture is a resource.
- Essentially, a resource is simply a data element such as a catalog, a medical record, or a document, such as this book chapter.
  - http://api.ipma.pt/file-example/1110600.json
  - http://api.ipma.pt/open-data/distrits-islands.json
- In general, resources may have multiple representations i.e. they can exist in different formats.
  - MS WORD
  - PDF
  - HTML

Resource operations

- Create – bring the resource into existence.
- Read – return a representation of the resource.
- Update – change the value of the resource.
- Delete – make the resource inaccessible.



(a) General resource actions

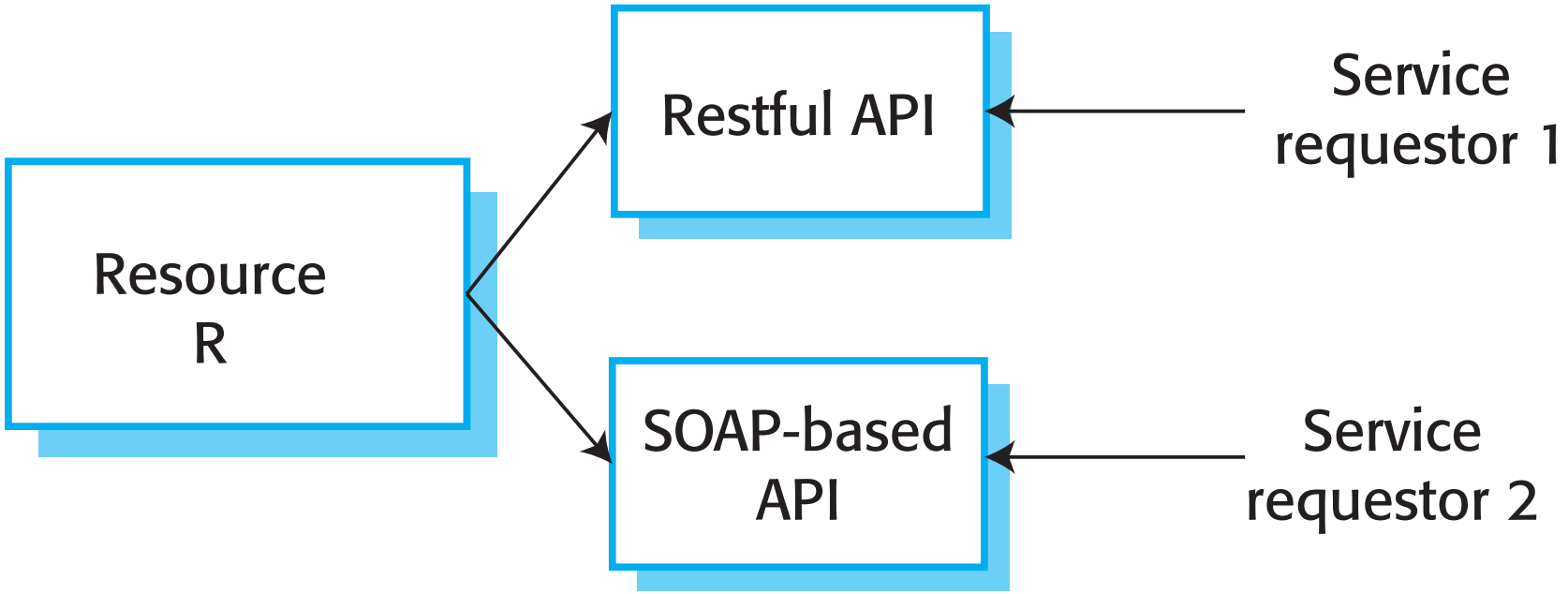


(b) Web resources

Disadvantages of RESTful approach

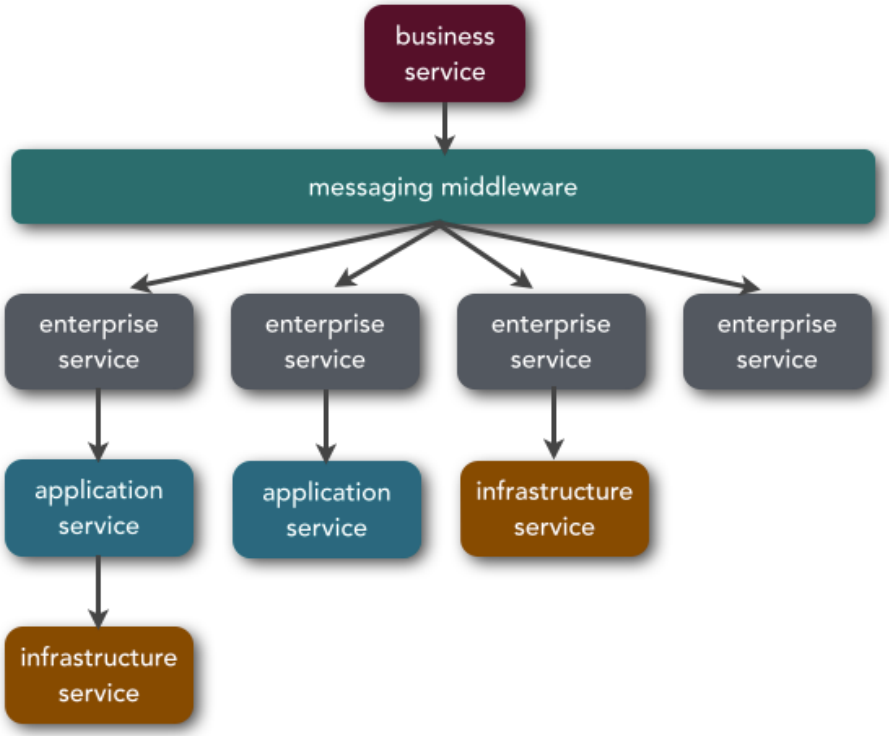
- When a service has a complex interface and is not a simple resource, it can be difficult to design a set of RESTful services to represent this.
- There are no standards for RESTful interface description so service users must rely on informal documentation to understand the interface.
- When you use RESTful services, you have to implement your own infrastructure for monitoring and managing the quality of service and the service reliability.

RESTful and SOAP-based APIs



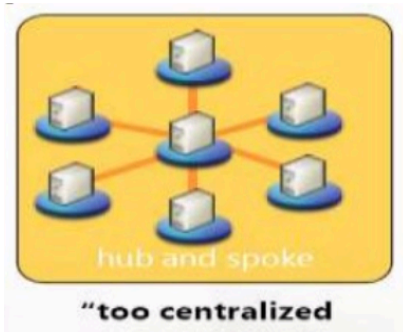
SOA topology

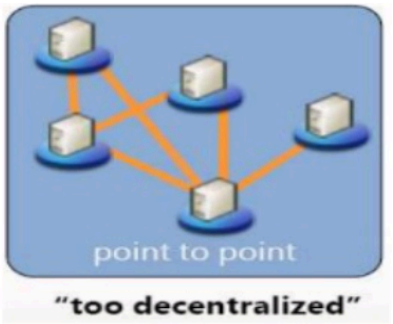
- SOA was primarily used to develop Monolithic application, though it has evolved from there
- Many organizations use SOA to resolve integration complexities.
  - Organizations in this case heavily depend on Enterprise Service Bus (ESB) technologies (messaging middleware).
  - All services are accessed through the ESB.

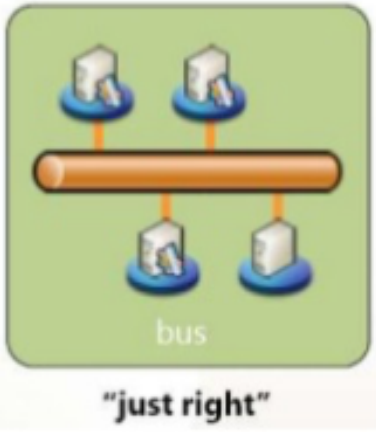


Key Characteristics of ESB

- Streamlines development
- Supports multiple binding strategies
- Performs data transformation
- Intelligent routing
- Real time monitoring
- Exception handling
- Service security





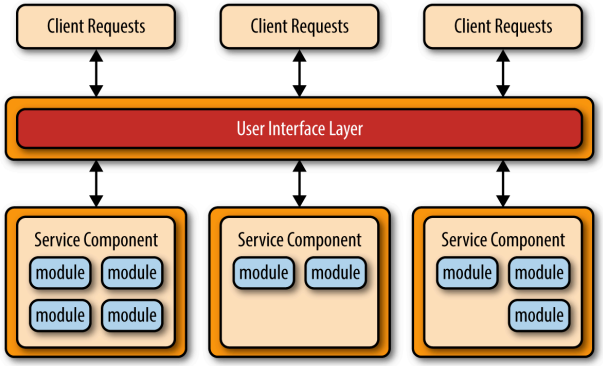


Architecture Patterns (*recap*)

- Microkernel Architecture
- Layered Architecture
- Event-Driven Architecture
- Service-oriented Architecture
- Microservices Architecture
- Space-Based Architecture

Microservices Architecture

- It evolved from two main sources:
  - applications developed using the layered pattern
  - distributed applications developed using service-oriented architecture.



- 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.
- Distributed architecture
  - all the components are fully decoupled
  - communication through JMS, AMQP, REST, SOAP, RMI, etc.

Microservices Architecture

- 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.

SOA vs. Microservices

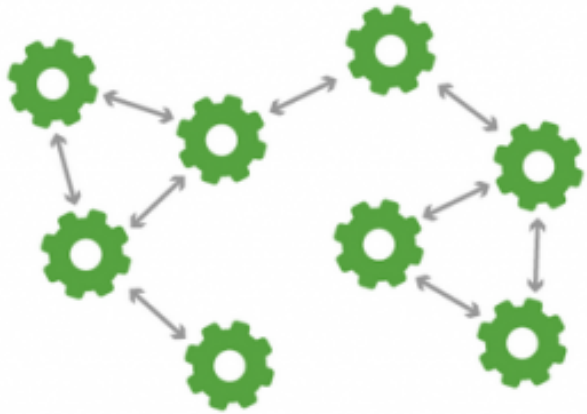
2000's

SERVICE ORIENTED ARCHITECTURE

SOA based applications are compromised of more loosely coupled components that use an Enterprise Services Bus messaging protocol to communicate between themselves.

2010's

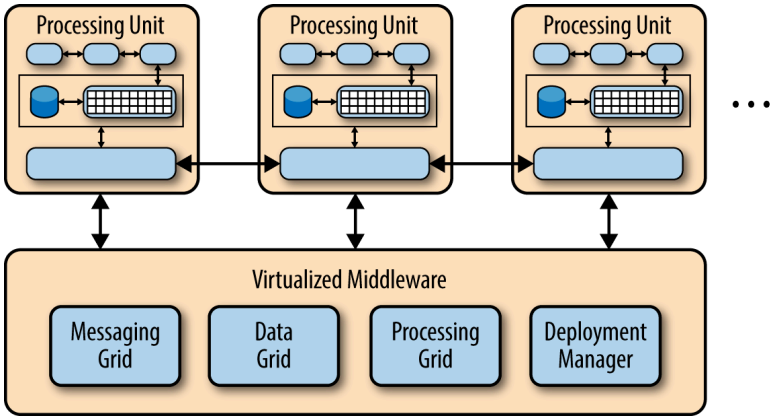
MICROSERVICES ARCHITECTURE



Microservices are a number of independent application services delivering one single functionality in a loosely connected and self-contained fashion, communicating through light-weight messaging protocols such as HTTP, REST or Thrift API.

Space-Based Architecture

- **Web-based applications**
  - 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.
- Specifically designed to address **scalability and concurrency** issues
  - It is often a better approach than trying to scale out a database into a non-scalable architecture.
- A good choice for applications with variable load
  - (e.g., social media sites, bidding and auction sites).



Extra resources

- Rick Kazman, Paul Clements, Len Bass, Software Architecture in Practice, Third Edition, 2012

- Comparing architetures in java ecosystem https://www.dineshonjava.com/software-architecture-patterns-and-designs/

Summary

- A software architecture is a description of how a software system is organized.
  - Properties of a system such as performance, security, and availability are influenced by the architecture used.
- Architectural design decisions include decisions on:
  - the type of application, the distribution of the system.
- Architectures may be documented from several different perspectives.
  - Possible views include a conceptual view, a logical view, a process view, a development view, and a physical view.
- Architectural patterns are a means of reusing knowledge about generic system architectures.
  - They describe the architecture, explain when it may be used, and point out its advantages and disadvantages.