SOFTWARE ARCHITECTURE PATTERNS

INGEGNERIA DEL SOFTWARE

Università degli Studi di Padova
Dipartimento di Matematica

Corso di Laurea in Informatica, A.A. 2016 – 2017

rcardin@math.unipd.it

SUMMARY

- Introduction
- Layered architecture
- Event-driven architecture
- Microservices architecture

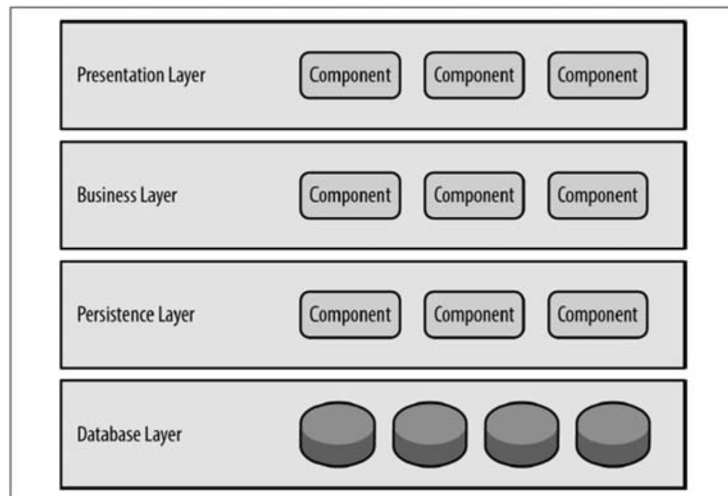
INTRODUCTION

- Applications lacking a formal architecture are generally coupled, brittle, difficult to change
  - Modules result in a collection of unorganized
  - *Big ball of mud* antipattern
  - Deployment and maintenance problems
    - Does the architecture scale? How does application response to changes? What are the deployment characteristics?
- Architecture patterns
  - Helps to manage these aspects, knowing the characteristics, strengths and weakness

LAYERED ARCHITECTURE

- Most common architecture pattern
  - *N-tier* architecture pattern
  - Standard *de facto* for most JEE applications
    - Widely known by most architects and developers
  - Reflects the organizational structure found in most IT companies
- Components are organized into horizontal layers
  - Each layer performs a single and specific role
    - The most common implementation consists of four layers
      - Presentation, business, persistence and database

LAYERED ARCHITECTURE



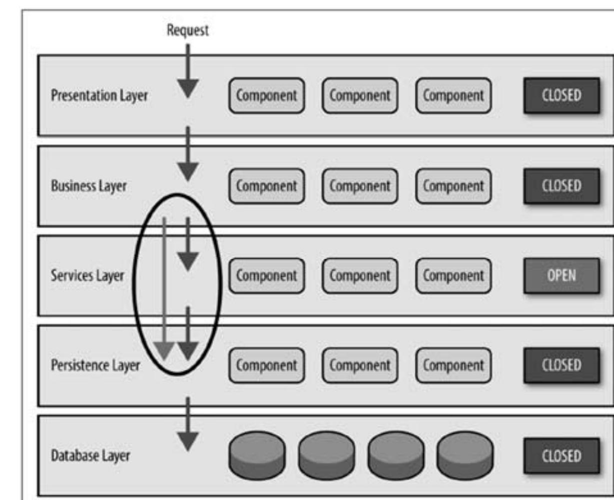
SEPARATION OF CONCERNS

- Every layer forms an abstraction over a particular business request
  - Components within a specific layer deal only with logic that pertains to that layer
    - i.e. Presentation layer does not need to know how to get customer data
- Component classification makes easy to build effective roles and responsibility models
  - Limited component scopes make easy to develop, test and govern , maintain such applications
- Well defined component interfaces

KEY CONCEPTS

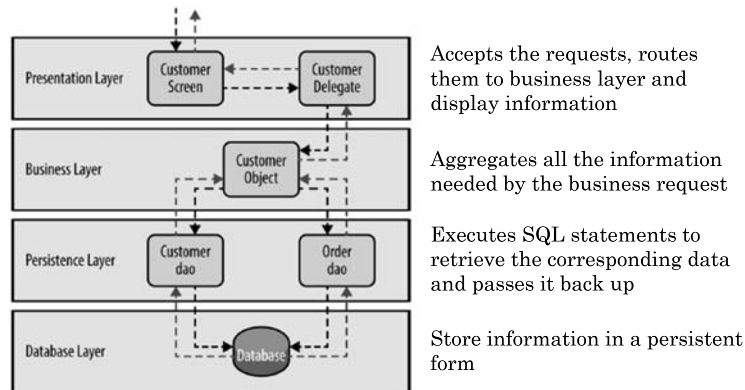
- Layers should be closed (layer isolation)
  - A request move from one layer to the layer right below it
  - Changes made in one layer generally don't impact or effect components in other layers.
- Layers can be open, too
  - Imagine a service layer below the business layer that offers commong services.
    - The business layer should go through the service layer to access the persistence layer.
    - Making the service layer open resolve the problem
      - Open layers should be very well documented

KEY CONCEPTS



EXAMPLE

> Example
> Consider a request from a business user to retrieve customer information for a particular individual



Accepts the requests, routes them to business layer and display information

Aggregates all the information needed by the business request

Executes SQL statements to retrieve the corresponding data and passes it back up

Store information in a persistent form

CONSIDERATIONS

- A good starting point for most application
  - It's a solid general purpose pattern
- *Architecture sinkhole anti-pattern*
  - Requests flow through multiple layers as simple pass-through
    - 80-20 proportion of good paths wrt sinkhole path
    - Open some layer (but be aware!)
- Tends to lend itself toward monolithic applications
    - It could be an issue in terms of deployment, robustness and reliability

PATTERN ANALYSIS

Characteristic	Rating	Description
Overall agility	⬇	Small changes can be properly isolated, big one are more difficult due to the monolithic nature of the pattern
Ease of deployment	⬇	Difficult for larger applications, due to monolithic deployments (that have to be properly scheduled
Testability	⇧	Very easy to mock or stub layers not affected by the testing process
Performance	⬇	Most of requests have to go through multiple layers
Scalability	⬇	The overall granularity is too bread, making it expensive to scale
Ease of development	⇧	A well know pattern. In many cases It has a direct connection with company's structure

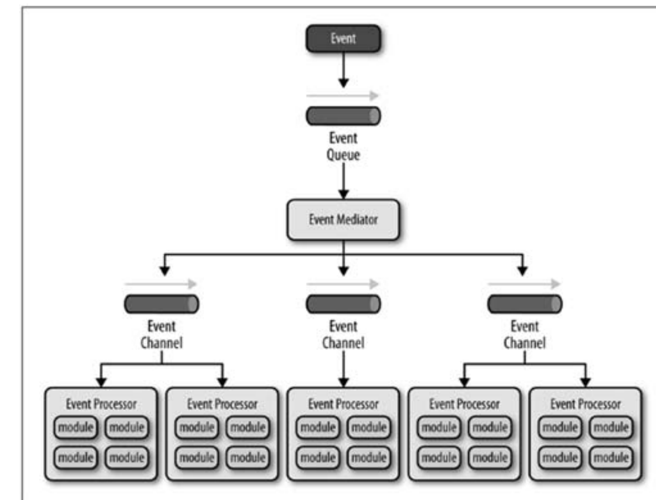
EVENT-DRIVEN ARCHITECTURE

- Popular asynchronous architeture pattern
  - It produces high scalable applications
    - Very adaptable: from small to very large applications
  - Single purpose, highly decoupled event processing modules
    - Process asynchronously events
- Mediator topology
  - A central mediator is used to orchestrate events throug multiple steps
- Broker topology

MEDIATOR TOPOLOGY

- Multiple steps orchestration
  - Events have multiple ordered steps to execute
  - Four main types of architecture components
    - Event queues
      - It is common to have anywhere from a dozen to hundreds
        - Message queue, web service point, ...
    - Event mediator
    - Event channels
    - Event processors
- Two types of main events
  - Initial event / processing event

MEDIATOR TOPOLOGY



MEDIATOR TOPOLOGY

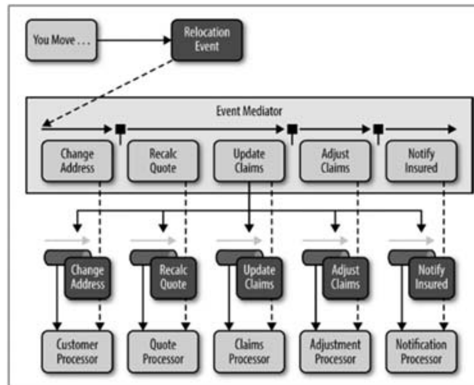
- Event mediator
  - Orchestrates the steps contained in the initial event
    - For each step it sends a specific processing event to an event channel
  - Does not perform any business logic
    - It knows only the step required to process the initial event
  - Implementation through open source hubs
    - Spring Integration, Apache Camel, Mule ESB
    - More sophisticated, using Business Process Execution Language (BPEL) engines or Business Process Managers (BPM)
      - BPMN allows to include human tasks

MEDIATOR TOPOLOGY

- Event channel
  - Asynchronous communication channel
    - Message queues
    - Message topic
      - An event can be processed by multiple specialized event processors
- Event processor
  - Contains business logic to process any event
  - Self conteined, independent, highly decoupled components
    - Fine-grained / Coarse-grained

EXAMPLE

Example
Suppose you are insured through a insurance company and you decide to move.



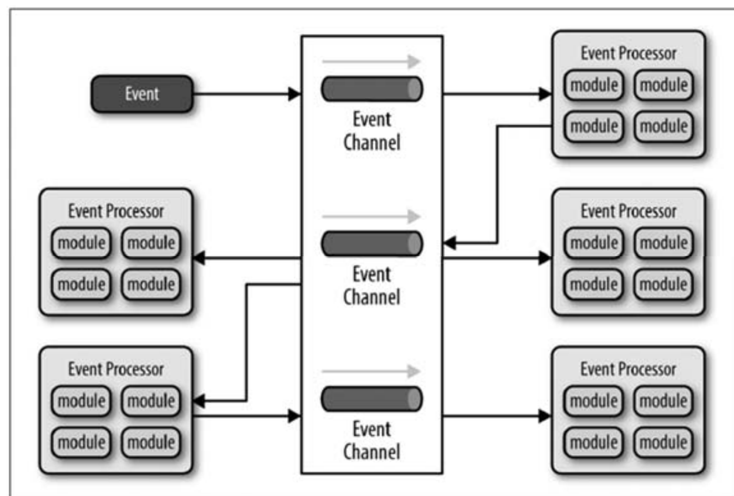
The initial event is a *relocation event*. Steps are contained inside the Event mediator.

For each event sends a processing event (*change*, *address*, *recalc quote*) to each event channel, and waits for the response.

BROKER TOPOLOGY

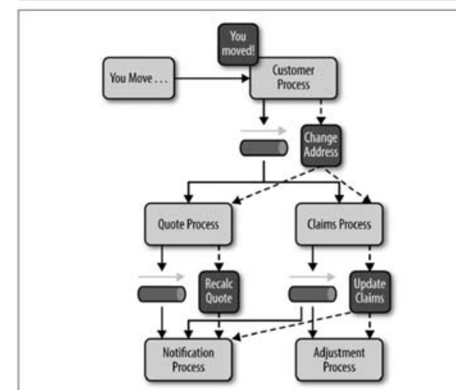
- There is no central mediator
  - The message flow is distributed across the event processors components
    - Chain-like fashion, lightweight message broker
      - Useful when the processing flow is very simple
  - Two main types of component
    - Broker: contains all event channels (queues, topics or both)
  - Event-processor
    - Contains the business logic
    - Responsible for publishing a new event
      - The event indicates the action is just performed. Some can be created only for future development

BROKER TOPOLOGY



EXAMPLE

Example
Suppose you are insured through a insurance company and you decide to move.



Customer process component receives the event directly. Then, it sends out and event *change address*.

Two processors are interested in this event. Both elaborate it, and so on...

The event chain continues until no more events are published.

CONSIDERATIONS

- Event-driven is complex to implement
  - Completly asynchronous and distributed
    - Remote process availability, lack of responsiveness, reconnection logic
  - Lack of atomic transactions
    - Which event can be run independently? Which granularity?
  - Strict need of contracts for event-processors
    - Standard data format (XML, JSON, Java Object, ...)
    - Contract versioning poilicy

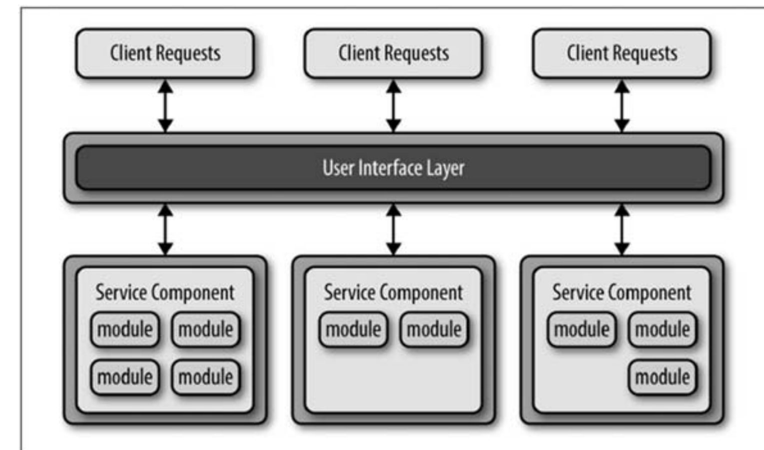
PATTERN ANALYSIS

Characteristic	Rating	Description
Overall agility	⇧	Changes are generally isolated and can be made quickly with small impacts
Ease of deployment	⇧	Ease to deploy due to the decoupled nature of event-processor components. Broker topology is easier to deploy
Testability	⬇	It requires some specialized testing client to generate events
Performance	⇧	In general, the pattern achieves high performance through its asynchronous capabilities
Scalability	⇧	Scaling separately event-processors, allowing for fine-grained scalability
Ease of development	⬇	Asynchronous programming, requires hard contracts, advanced error handling conditions

MICROSERVICES ARCHITECTURE

- A still evolving pattern
  - A viable alternative to monolithic and service-oriented architecutures
  - *Separately deployed unit*
    - Easier deployment, increased scalability, high degree of decoupling
- Service components
  - From a single module to a large application's portion
    - Choose the right level of service component granularity is one of the biggest challenges
  - Distributed: *remote access protocol*
    - JMS, AMQP, REST, SOAP, RMI, ...

MICROSERVICES ARCHITECTURE



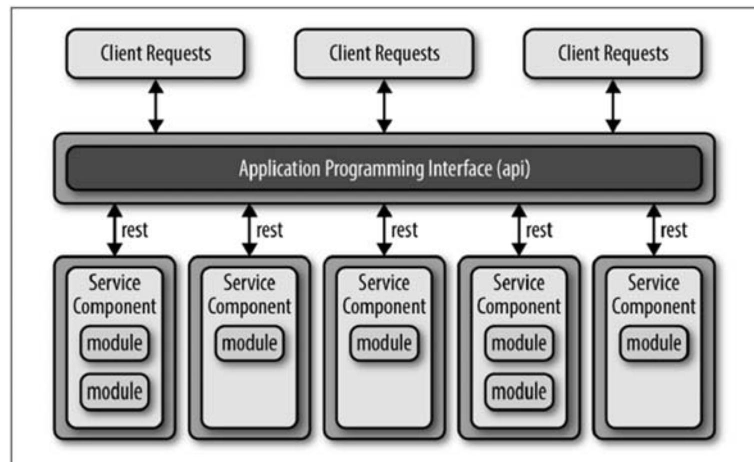
EVOLUTIONARTY PATHS

- Evolved from issues associated with other architectures
  - From monolithic: open to continous delivery
    - Avoid the «monthly deployment» cycles due to tightly coupling between components
    - Every service component is independent developed, tested and deployed
  - From SOA: simplification of the service notion
    - SOA is a powerful pattern, that promises to align business goals with IT capabilities
      - Expensive, ubiquitous, difficult to understand / implement
    - Eliminates orchestration needs, simplyfing connectivity

API REST-BASED TOPOLOGY

- Useful for websites that expose small services
  - Service components are very fine-grained
    - Specific business function, independet from the rest
      - Only one or two modules
    - *Microservice*
- These services are accessed through and API
  - REST-based interface
    - Separately deployed web-based API layer
    - Google, Amazon, Yahoo cloud-based RESTful web services

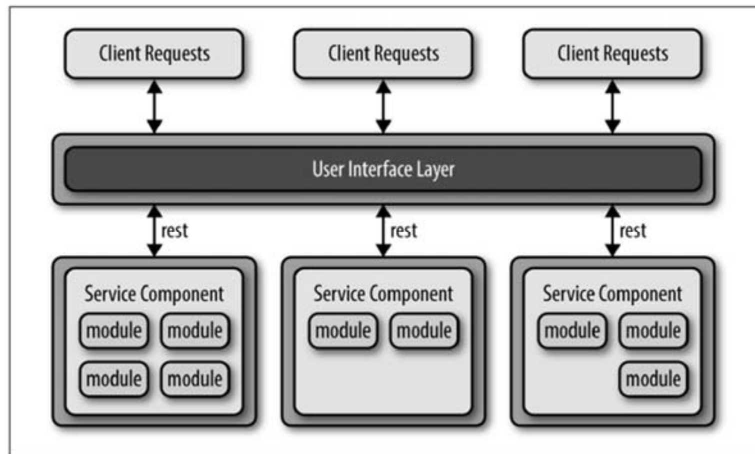
API REST-BASED TOPOLOGY



REST-BASED TOPOLOGY

- Accessed directly by fat / web based clients
  - User interface is deployed separately
  - REST-based interface
    - No middle API layer required
- Larger and coarse-grained
  - Represent a small portion of the overall business application
    - Common for small to medium-sized business applications

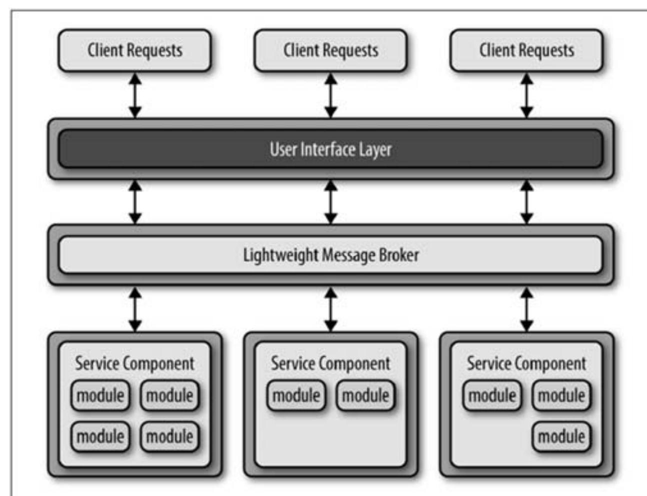
REST-BASED TOPOLOGY



CENTRALIZED MESSAGE TOPOLOGY

- Lightweight centralized message broker
  - No trasformation, orchestration or complex routing
    - Not to confuse with Service-oriented application
  - No REST-based access required
  - Found in larger business applications
- Sophisticated control over the transport layer
  - Advanced queuing mechanisms, asynchronous messaging, monitoring, error handling, ...
  - Broker clustering and federation
    - Avoid the architectural single point of failure and bottleneck

CENTRALIZED MESSAGE TOPOLOGY



SERVICES GRANULARITY

- The main challenge is to defined the right granularity of service components
  - Coarse-grained services
    - Not easy to deploy, scale, test and are not loose couples
  - Too fine-grained services
    - Require orchestration, turning into SOA application
    - Require inter-service communication to process a single request
- Use database communication
  - Avoid service-to-service communication

SERVICE GRANULARITY

- Violation of the DRY principle
  - Replicate some functionalities to keep independency across services
    - No share of business logic
- Is it the right pattern for your architecture?
  - NO, if you still cannot avoid service-component orchestration
  - No definition of transactional unit of work
    - Due to the distributed nature of the pattern
    - Using transaction framework adds too much complexity

CONSIDERATIONS

- Robustness, better scalability, continous delivery
  - Small application component, separately deployed
    - Solve many problems of monolithic and SOA architectures
- Real-time production deployments
  - Only changed service components can be deployed
    - Redirection to an error / waiting page
    - Continous availability (hotdeploy)
- Distributed architeture problems
  - Contract creation and maintanance, remote system availability, remote access authentication, ...

PATTERN ANALYSIS

Characteristic	Rating	Description
Overall agility	⇧	Changes are generally isolated. Fast and easy deployment. Loose cooupling
Ease of deployment	⇧	Ease to deploy due to the decoupled nature of service components. Hotdeploy and continous delivery
Testability	⇧	Due to isolation of business functions, testing can be scoped. Small chance of regression
Performance	⬇	Due to distributed nature of the pattern, performance are not generally high
Scalability	⇧	Each service component can be separately scaled (fine tuning)
Ease of development	⇧	Small and isolated business scope. Less coordination needed among developers or development teams

RIFERIMENTI

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