Be Academy - Architecture

February 2019

Agenda

I. Architectural patterns overview

II. Architectural styles in cloud

III. Cloud design patterns

IV. Architecture of BeOnTime application

V. Exercise

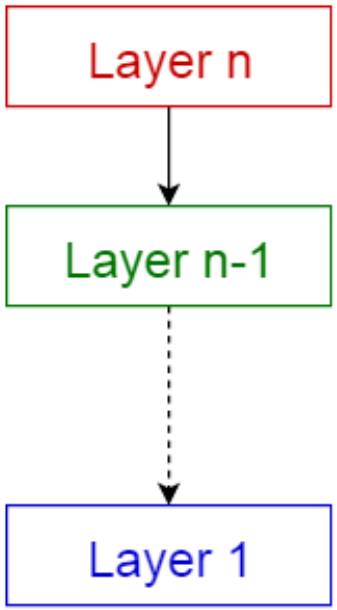
VI. Exercice Overview

Architectural patterns overview

1. Layered pattern
2. Client-server pattern
3. Master-slave pattern
4. Broker pattern
5. Event-bus pattern
6. Peer-to-peer pattern
7. Model-view-controller pattern

Architectural patterns overview

Layered pattern



Layered pattern

The most commonly found 4 layers of a general information system are as follows.

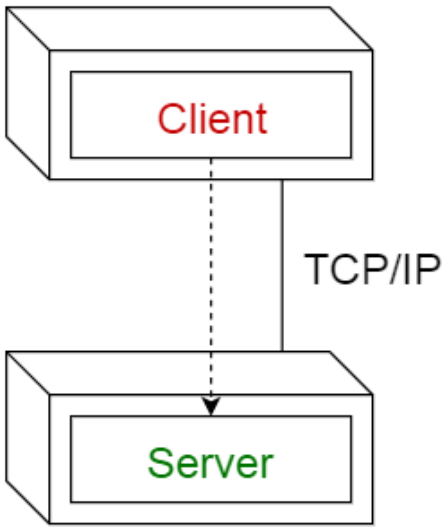
- Presentation layer (also known as UI layer)
- Application layer (also known as service layer)
- Business logic layer (also known as domain layer)
- Data access layer (also known as persistence layer)

Usage

- General desktop applications.
- E commerce web applications.

Architectural patterns overview

Client-Server pattern



Client-server pattern

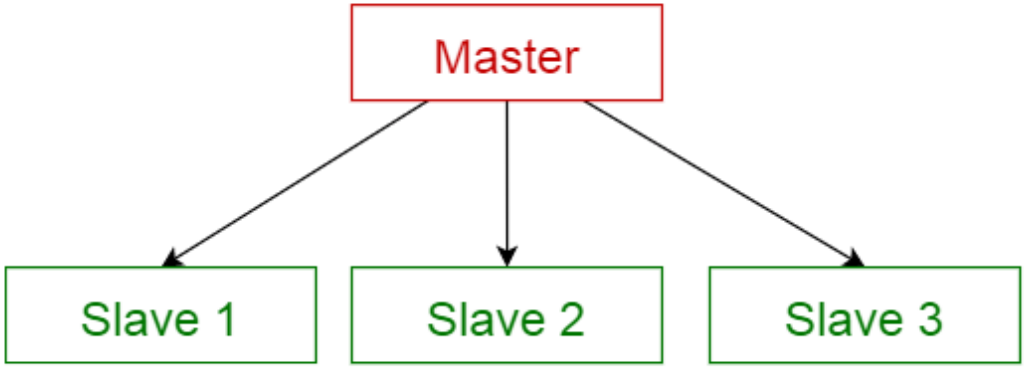
This pattern consists of two parties; a server and multiple clients. The server component will provide services to multiple client components. Clients request services from the server and the server provides relevant services to those clients. Furthermore, the server continues to listen to client requests.

Usage

- Online applications such as email, document sharing and banking.

Architectural patterns overview

Master-slave pattern



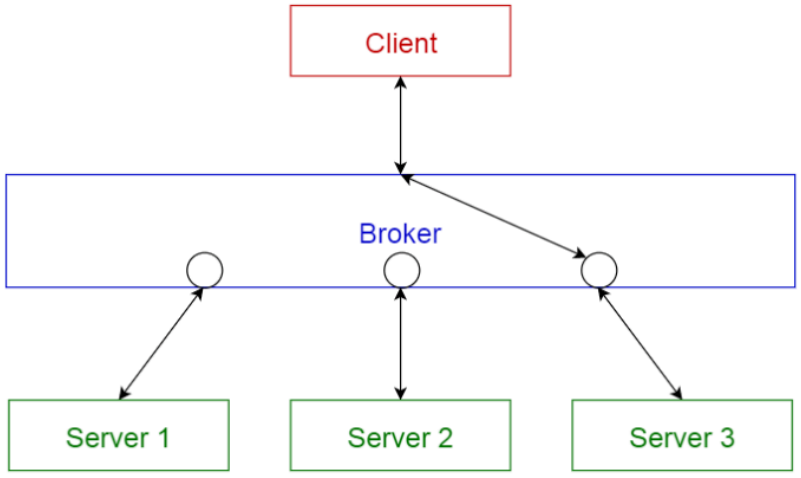
This pattern consists of two parties; master and slaves. The master component distributes the work among identical slave components, and computes a final result from the results which the slaves return.

Usage

- In database replication, the master database is regarded as the authoritative source, and the slave databases are synchronized to it.
- Peripherals connected to a bus in a computer system (master and slave drives).

Architectural patterns overview

Broker pattern



Broker pattern

This pattern is used to structure distributed systems with decoupled components. These components can interact with each other by remote service invocations. A broker component is responsible for the coordination of communication among components.

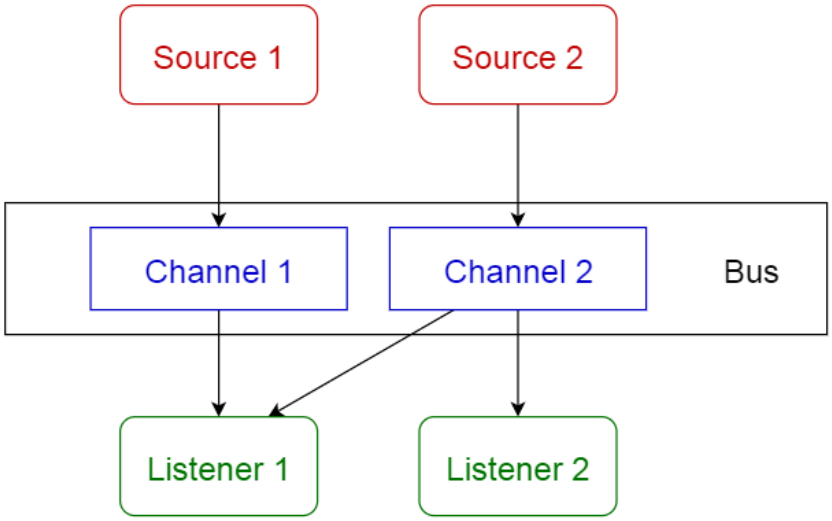
Servers publish their capabilities (services and characteristics) to a broker. Clients request a service from the broker, and the broker then redirects the client to a suitable service from its registry.

Usage

- Message broker software such as Apache ActiveMQ, Apache Kafka, RabbitMQ and JBoss Messaging.

Architectural patterns overview

Message/Event bus pattern



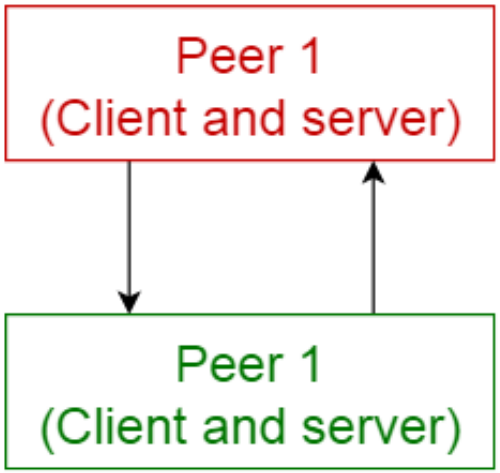
This pattern primarily deals with events and has 4 major components; event source, event listener, channel and event bus. Sources publish messages to particular channels on an event bus. Listeners subscribe to particular channels. Listeners are notified of messages that are published to a channel to which they have subscribed before.

Usage

- Android development
- Notification services

Architectural patterns overview

Peer-to-peer pattern



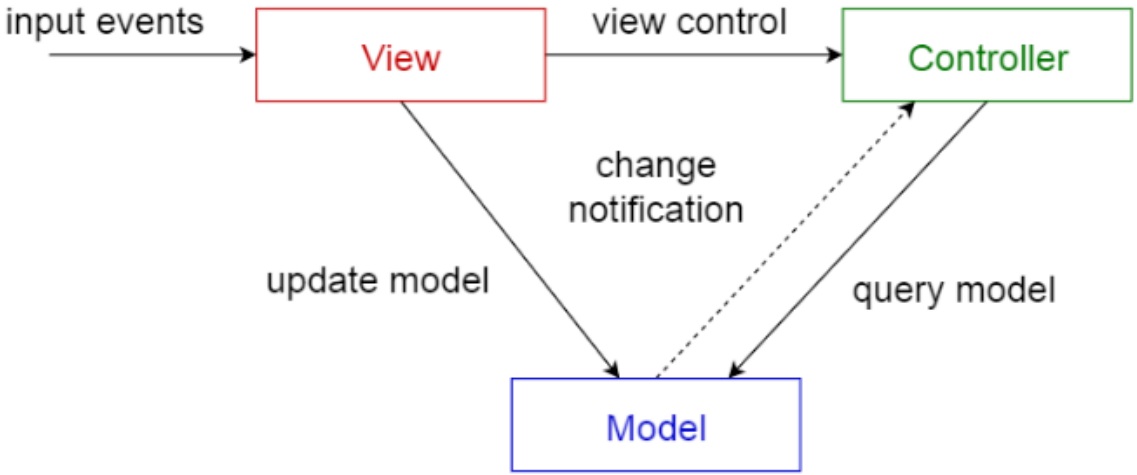
In this pattern, individual components are known as peers. Peers may function both as a client, requesting services from other peers, and as a server, providing services to other peers. A peer may act as a client or as a server or as both, and it can change its role dynamically with time.

Usage

- File-sharing networks such as Gnutella and G2)
- Multimedia protocols such as P2PTV and PDTP.

Architectural patterns overview

Model-view-controller pattern



This pattern, also known as MVC pattern, divides an interactive application in to 3 parts as,

- model — contains the core functionality and data
- view — displays the information to the user (more than one view may be defined)
- controller — handles the input from the use

Usage

- Architecture for World Wide Web applications in major programming languages.
- Web frameworks such as Django and Rails.

Agenda

I. Architectural patterns overview

II. Architectural styles in cloud

III. Cloud design patterns

IV. Architecture of BeOnTime application

V. Exercise

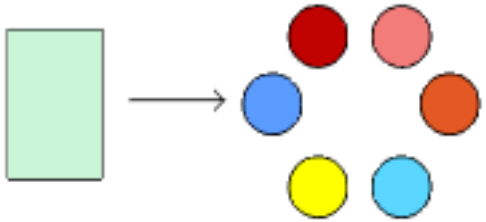
VI. Exercice Overview

Architectural styles in cloud applications

1. Microservices
2. Event driven
3. Big Data/Big Compute
4. Challenges and benefits of architectural styles

Architectural styles in cloud

Microservices

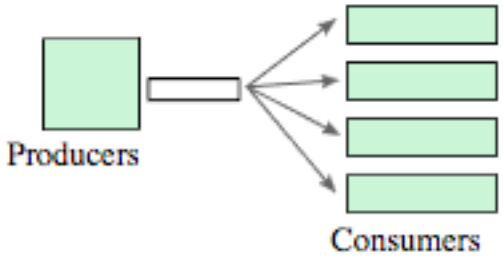


A microservices application is composed of many small, independent services. Each service implements a single business capability. Services are loosely coupled, communicating through API contracts.

Each service can be built by a small, focused development team. Individual services can be deployed without a lot of coordination between teams, which encourages frequent updates. A microservice architecture is more complex to build and manage than either N-tier or web-queue-worker. It requires a mature development and DevOps culture. But done right, this style can lead to higher release velocity, faster innovation, and a more resilient architecture.

Architectural styles in cloud

Event driven



Event-Driven Architectures use a publish-subscribe (pub-sub) model, where producers publish events, and consumers subscribe to them. The producers are independent from the consumers, and consumers are independent from each other.

Consider an event-driven architecture for applications that ingest and process a large volume of data with very low latency, such as IoT solutions. The style is also useful when different subsystems must perform different types of processing on the same event data.

Architectural styles in cloud

Big Data / Big Compute

Big Data and **Big Compute** are specialized architecture styles for workloads that fit certain specific profiles. Big data divides a very large dataset into chunks, performing paralleling processing across the entire set, for analysis and reporting. Big compute, also called high-performance computing (HPC), makes parallel computations across a large number (thousands) of cores. Domains include simulations, modeling, and 3-D rendering.

Agenda

I. Architectural patterns overview

II. Architectural styles in cloud

III. Cloud design patterns

IV. Architecture of BeOnTime application

V. Exercise

VI. Exercice Overview

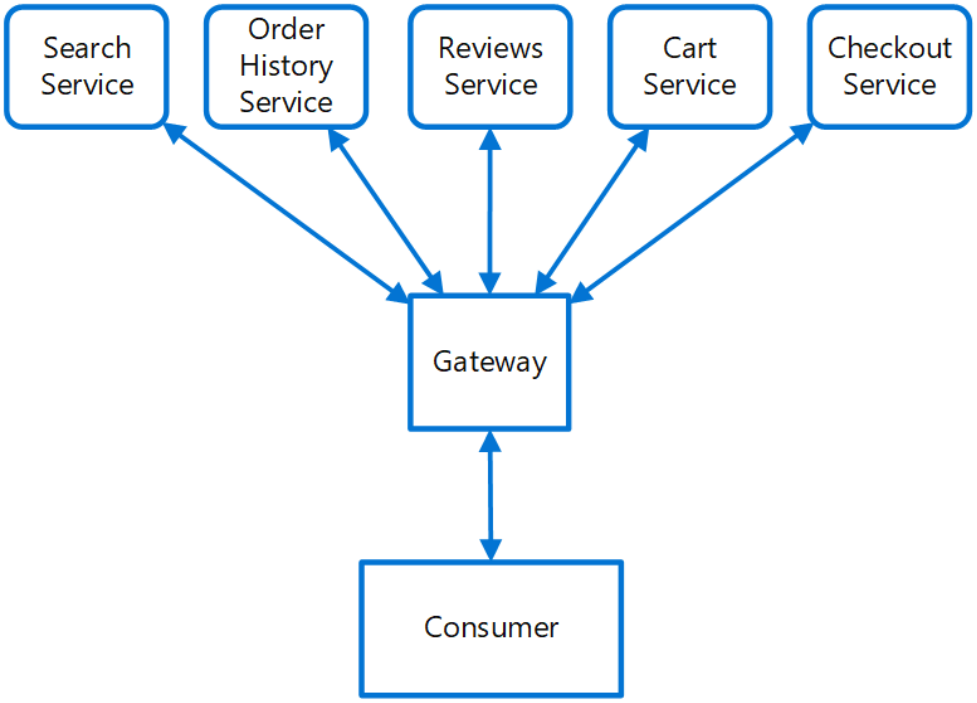
Cloud design patterns

1. Gateway routing
2. Service discovery
3. Circuit breaker
4. Retry
5. External configuration store
6. Event sourcing

Cloud design patterns

Gateway routing

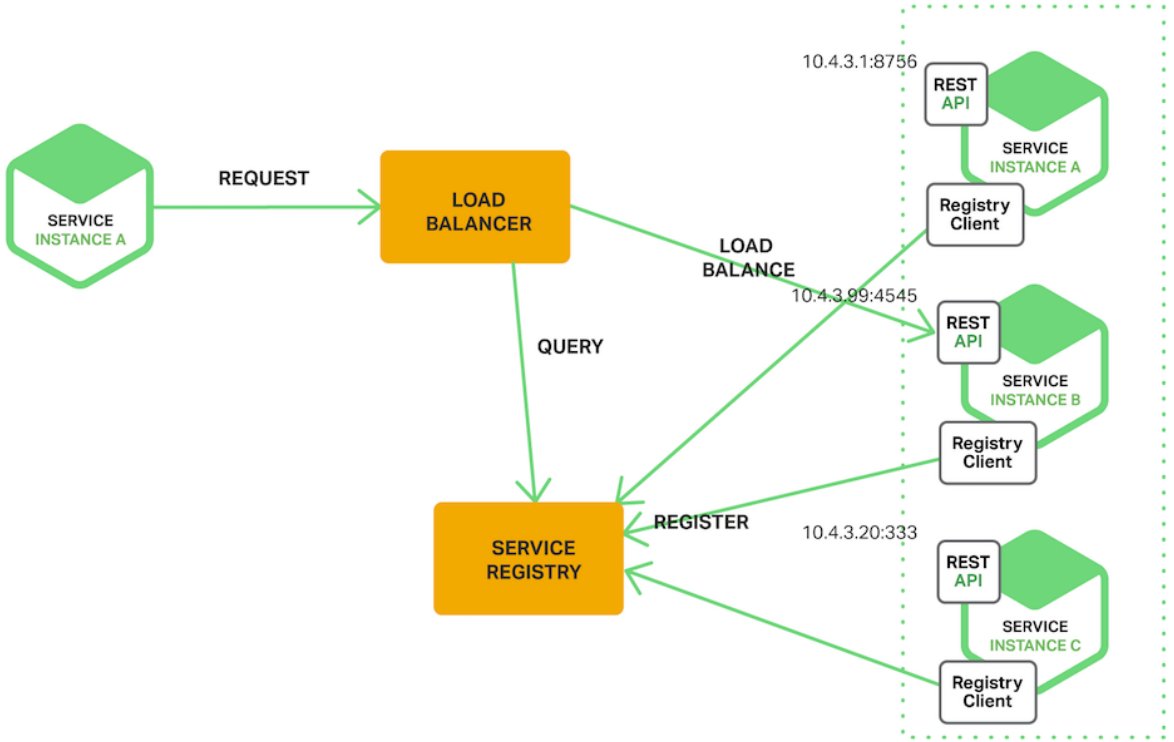
Route requests to multiple services using a single endpoint. This pattern is useful when you wish to expose multiple services on a single endpoint and route to the appropriate service based on the request.



Cloud design patterns

Service discovery

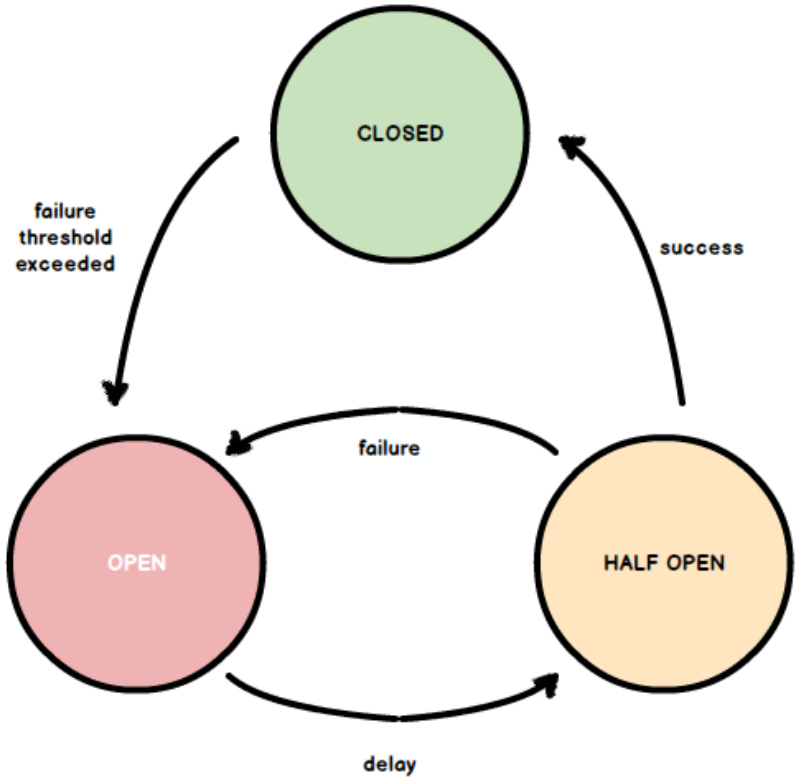
When making a request to a service, the client makes a request via a router (a.k.a load balancer) that runs at a well known location. The router queries a service registry, which might be built into the router, and forwards the request to an available service instance.



Cloud design patterns

Circuit breaker

Handle faults that might take a variable amount of time to recover from, when connecting to a remote service or resource. This can improve the stability and resiliency of an application.



Cloud design patterns

Retry

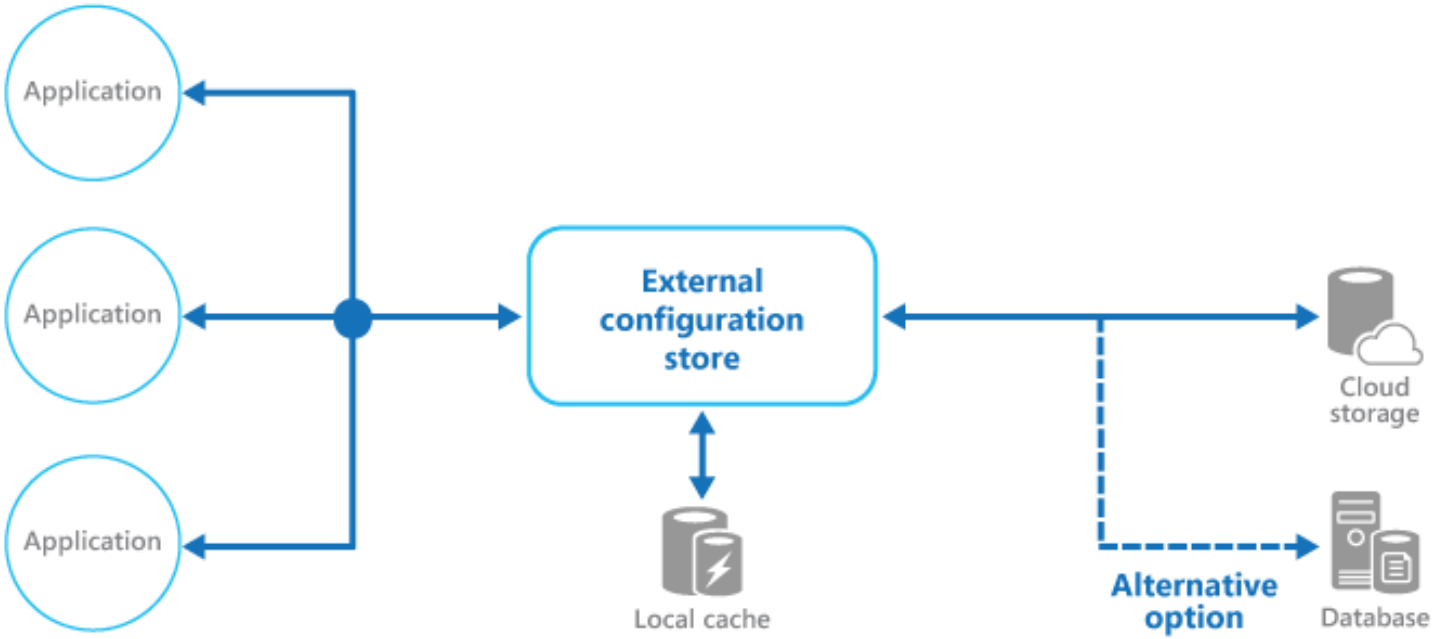
Enable an application to handle transient failures when it tries to connect to a service or network resource, by transparently retrying a failed operation. This can improve the stability of the application.

1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

Cloud design patterns

External configuration store

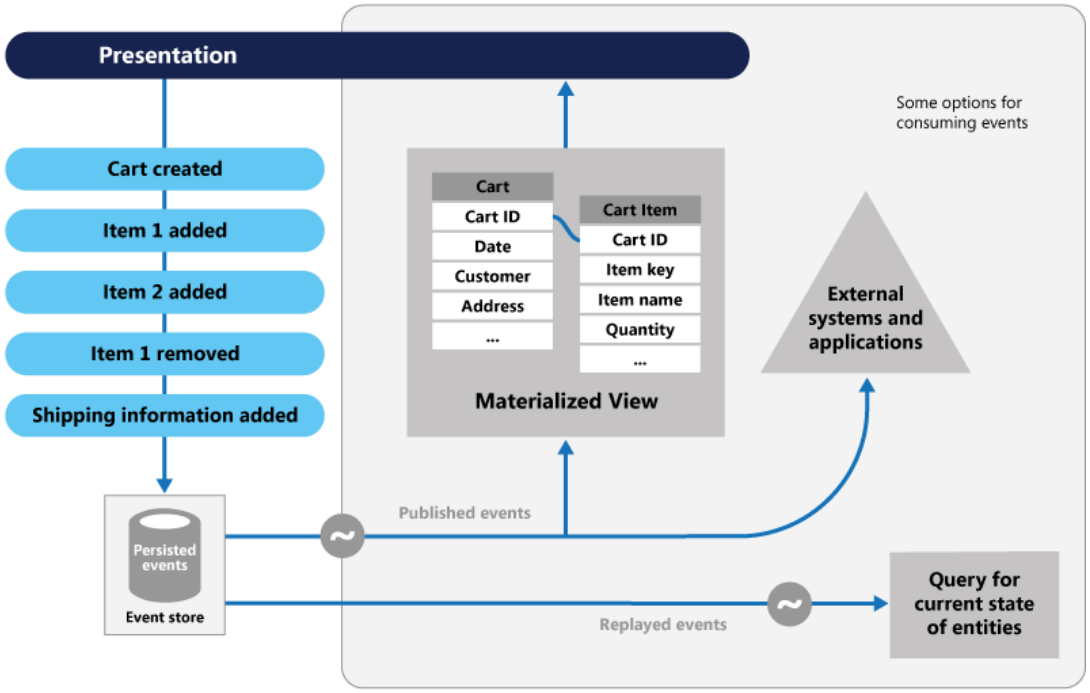
Move configuration information out of the application deployment package to a centralized location. This can provide opportunities for easier management and control of configuration data, and for sharing configuration data across applications and application instances.



Cloud design patterns

Event sourcing

Instead of storing just the current state of the data in a domain, use an append-only store to record the full series of actions taken on that data. The store acts as the system of record and can be used to materialize the domain objects. This can simplify tasks in complex domains, by avoiding the need to synchronize the data model and the business domain, while improving performance, scalability, and responsiveness. It can also provide consistency for transactional data, and maintain full audit trails and history that can enable compensating actions.



Agenda

I. Architectural patterns overview

II. Architectural styles cloud

III. Cloud design patterns

IV. Architecture of BeOnTime application

V. Exercise

VI. Exercice Overview

Architecture of BeOnTime application

1. Architecture style
    - Microservices
2. Cloud design patterns
    - Gateway routing
    - Service discovery
    - External configuration store
    - Retry
3. Containerization
    - Docker
4. Technology stack
    - Spring Framework/ Spring Cloud

Agenda

I. Architectural patterns overview

II. Architectural styles cloud

III. Cloud design patterns

IV. Architecture of BeOnTime application

V. Exercise

VI. Exercice Overview

Exercise

https://github.com/bepoland-academy/architecture-exercise

Links

https://docs.microsoft.com/en-us/azure/architecture/
http://spring.io/projects/spring-cloud
http://spring.io/projects/spring-cloud-netflix
https://docs.docker.com/get-started/

Credits and copyrights

This presentation contains materials from couple of external sources:

- *https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013*
- *https://docs.microsoft.com/en-us/azure/architecture/*

Thank you !