Decomposition Strategies

Contents

[What is the software architecture and why is it important ? 2](#_Toc55932336)

[A definition of software architecture 2](#_Toc55932337)

[The 4+1 View Model of Software Architecture 2](#_Toc55932338)

[Why architecture matters 3](#_Toc55932339)

[Overview of architecture styles 3](#_Toc55932340)

[The Layered Architecture Style 3](#_Toc55932341)

[About the Hexagonal Architecture Style 4](#_Toc55932342)

[The microservice architecture is an architectural style 5](#_Toc55932343)

[What is a service ? 5](#_Toc55932344)

[What is loose coupling ? 6](#_Toc55932345)

[The role of shared libraries 6](#_Toc55932346)

[The size of a service is mostly unimportant 6](#_Toc55932347)

[Defining an application’s microservice architecture 7](#_Toc55932348)

[Identifying the system operations 8](#_Toc55932349)

[Defining services by applying the Decomposes by business capability pattern 8](#_Toc55932350)

[Business Capabilities define what an organization does 8](#_Toc55932351)

[Identifying Business Capabilities 8](#_Toc55932352)

[From Business Capabilities to Services 8](#_Toc55932353)

[Defining services by applying the Decompose by sub-domain pattern 8](#_Toc55932354)

[Decomposition guidelines 8](#_Toc55932355)

[Single Responsibility Principle 8](#_Toc55932356)

[Common Closure Principle 8](#_Toc55932357)

[Obstacles to decomposing an application into services 8](#_Toc55932358)

[Network Latency 8](#_Toc55932359)

[Synchronous Interprocess Communication Reduces Availability 8](#_Toc55932360)

[Maintaining Data Consistency Across Services 8](#_Toc55932361)

[Obtaining A Consistent View of The Data 8](#_Toc55932362)

[God Classes Prevent Decomposition 8](#_Toc55932363)

[Defining service APIs 8](#_Toc55932364)

[Assigning System Operations To Services 8](#_Toc55932365)

[Determining The Apis Required To Support Collaboration Between Services 8](#_Toc55932366)

# What is the microservice architecture exactly ?

## What is the software architecture and why is it important ?

### A definition of software architecture

“The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.”

### The 4+1 View Model of Software Architecture

The 4+1 View Model of Software Architecture defines four different views of a software architecture. Each describes a particular aspect of the architecture and consists of particular set of software elements and relationships between them.

The purpose of each view is as follows:

* *Logical View*:
  + The software elements that are created by developers.
  + In OOP, these elements are classes and packages.
  + The relations between them are the relationships between classes and packages, including inheritance, associations and depend-on.
* *Implementation View:*
  + The output of the build system.
  + This view consists of modules, which represent packaged code, and components, which are executable or deployable units consisting of one or more modules.
  + The relations between them include dependency relationships between modules and composition relationships between components and modules.
* *Process view:*
  + The components at runtime.
  + Each element is a process, and the relations between processes represent interprocess communication.
* *Deployment:*
  + How the processes are mapped to machines.
  + The elements in this view consist of machines and the processes.
  + The relations between machines represent networking.
  + This view also describes the relationship between processes and machines.

### Why architecture matters

An application has two categories of requirements.

The first category includes the *functional requirements*, which define what the application must do – *use cases,* or *user stories*. Architecture has little to do with functional requirements.

The second category is the *quality of service* requirements, or *quality attributes*, which define the runtime qualities such as scalability and reliability and development time qualities include maintainability, testability and deployability.

## Overview of architecture styles

“An architectural style, then, defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined.”

### The Layered Architecture Style

A layered architecture organizes software elements into layers. Each layer has a well-defined set of responsibilities. It also constraints the dependencies between the layers. A layered architecture also constraints the dependencies between the layers.

It organizes the application’s classes into the following tiers or layers:

* *Presentation layer* – Contains code the implements the user interface or external APIs
* *Business logic layer* – Contains the business logic
* *Persistence layer* – Implements the logic of interacting with the database

But this does have significant drawbacks:

* *Single presentation layer* – It doesn’t represent the fact that an application is likely to be invoked by more than just a single system.
* *Single persistence layer –* It doesn’t represent the fact that an application is likely to interact with more than just a single database.
* *Defines the business logic layer as depending on the persistence layer –* In theory, this dependency prevents you from testing the business logic without the database.

Also, the layered architecture misrepresents the dependencies in a well-designed application. The business logic typically defines an interface or a repository of interfaces that define data access methods.

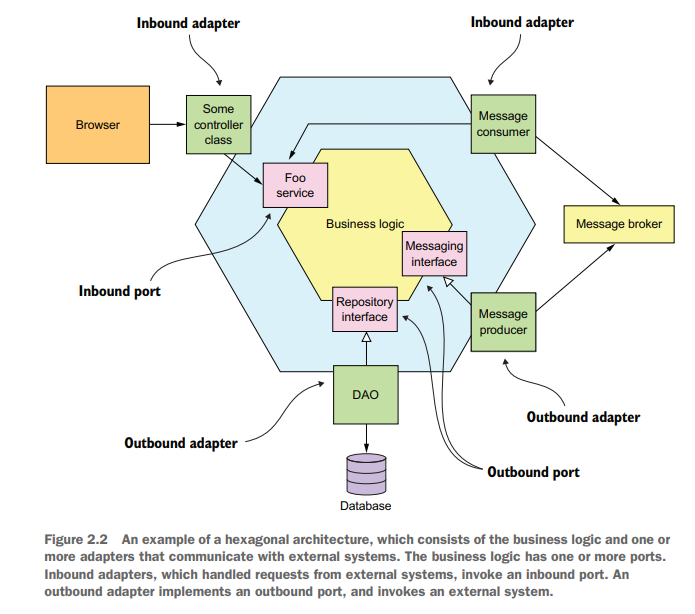
### About the Hexagonal Architecture Style

The *hexagonal architecture style* organizes the logical view in a way that places the business logic at the centre.

Instead of the presentation layer, the application has one or more *inbound adapters* that handle requests from the outside by invoking the business logic.

Similarly, instead of a data persistence tier, the application has one or more *outbound adapters* that are invoked by the business logic and invoke external applications.

A key characteristic feature and benefit of this architecture is that the business logic doesn’t depend on the adapters. Instead, they depend upon it.

The business logic has one or more ports. A *port* defines a set of operations and is how the business logic interacts with what’s outside of it.

* In Java, for example, a port is often a Java interface.
* There are two kinds of ports: inbound and outbound ports.
* An inbound port is an API exposed by the business logic, which enables it to be invoked by external applications. **An example of an inbound port is a *service interface*, which defines a service’s public methods.** An inbound adapter handles requests from the outside world by invoking an inbound port. An example of an inbound adapter is a Spring MVC Controller that implements either a set of REST endpoints or a set of web pages. Another example is a message broker client that subscribes to messages. Multiple inbound adapters can invoke the same inbound port.
* An outbound port is how the business logic invokes external systems. **An example of an output port is a *repository interface*, which defines a collection of data access operations.** An outbound adapter implements an outbound port and handles requests from the business logic by invoking an external application or service. An example of an outbound adapter is a data access object (DAO) class that implements operations for accessing a database. Another example would be a proxy class that invokes a remote service. Outbound adapters can also publish events.

**Benefits:**

1. It decouples the business logic from the presentation and data access logic in the adapters. The business logic doesn’t depend on either the presentation logic or the data access logic.
2. Another benefit is that it more accurately reflects the architecture of a modern application. The business logic can be invoked via multiple adapters, each of which implements a particular API or UI. The business logic can also invoke multiple adapters, each one of which invokes a different external system.

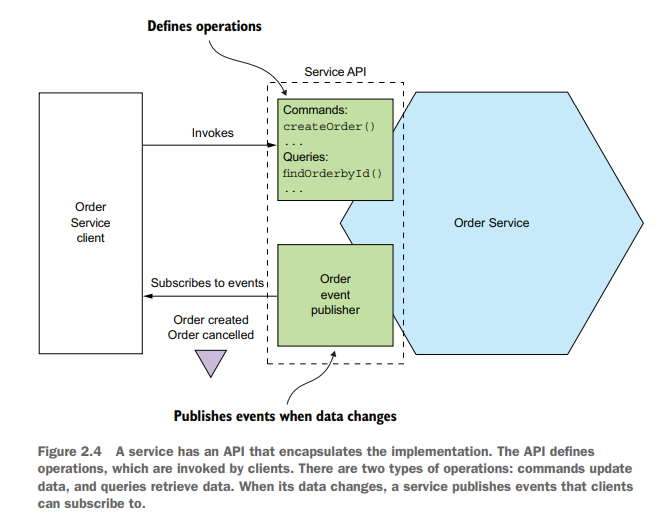
## The microservice architecture is an architectural style

**Monolithic architecture** is an architectural style that structures the implementation view as a single component: a single executable or WAR file.

The **microservice architecture** is also an architectural style. It structures the implementation view as a set of multiple components: executables or WAR files. The components are services, and the connectors are the communication protocols that enable those services to collaborate. Each service has its own logical view architecture, which is typically a hexagonal architecture.

A key constraint imposed by the microservice architecture is that the services are loosely coupled.

### What is a service ?

A service is a standalone, independently deployable software component that implements some useful functionality.

### What is loose coupling ?

All interaction with a service happens via its API, which encapsulates its implementation details. This enables the implementation of the service to change without impacting its clients.

The requirement for services to be loosely coupled and to collaborate only via APIs prohibits services from communicating via a database.

### The role of shared libraries

You should strive to use libraries for functionality that’s unlikely to change. Instead, you should create a library that’s used by the services.

### The size of a service is mostly unimportant

A much better goal is to define a well-designed service to be a service capable of being developed by a small team with minimal lead time and with minimal collaboration with other teams.

# Defining an application’s microservice architecture

A *system operation* is an abstraction of a request that the application must handle. It’s either a command, which updates data, or a query, which retrieves data. The behaviour of each command is defined in terms of an abstract domain model, which is also derived from the requirements. The system operations become the architectural scenarios that illustrate how the services collaborate.

Obstacles to decomposition:

1. Network latency due to too many round-trips between services
2. Synchronous communication between services reduces availability (solution is self-contained services)
3. Maintain data consistency across services (solution is sagas)
4. God classes (solution is concepts from domain-driven design)

## Identifying the system operations

### Creating A High-Level Domain Model

## Defining services by applying the Decomposes by business capability pattern

### Business Capabilities define what an organization does

### Identifying Business Capabilities

### From Business Capabilities to Services

## Defining services by applying the Decompose by sub-domain pattern

## Decomposition guidelines

### Single Responsibility Principle

### Common Closure Principle

## Obstacles to decomposing an application into services

### Network Latency

### Synchronous Interprocess Communication Reduces Availability

### Maintaining Data Consistency Across Services

### Obtaining A Consistent View of The Data

### God Classes Prevent Decomposition

## Defining service APIs

### Assigning System Operations to Services

### Determining the APIs Required To Support Collaboration Between Services