### 1. Singleton Pattern

Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the Java virtual machine.

**2. Factory Pattern**

The factory design pattern is used when we have a superclass with multiple sub-classes and based on input, we need to return one of the sub-class. This pattern takes out the responsibility of the instantiation of a class from the client program to the factory class.

### 3. Abstract Factory Pattern

Abstract Factory pattern is similar to Factory pattern and it’s a factory of factories. If you are familiar with the factory design pattern in java, you will notice that we have a single Factory class that returns the different sub-classes based on the input provided and the factory class uses if-else or switch statements to achieve this. In Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class and then an Abstract Factory class that will return the sub-class based on the input factory class.

### 4. Builder Pattern

This pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes. Builder pattern solves the issue with a large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object.

### 5. Prototype Pattern

The prototype pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing. So this pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. This pattern uses java cloning to copy the object. Prototype design pattern mandates that the Object which you are copying should provide the copying feature. It should not be done by any other class. However whether to use the shallow or [deep copy](https://www.digitalocean.com/community/tutorials/java-deep-copy-object) of the Object properties depends on the requirements and it’s a design decision.

### 1. Adapter Pattern

The adapter design pattern is one of the structural design patterns and it’s used so that two unrelated interfaces can work together. The object that joins these unrelated interfaces is called an Adapter. As a real-life example, we can think of a mobile charger as an adapter because the mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between the mobile charging socket and the wall socket.

### 2. Composite Pattern

Composite pattern is one of the Structural design patterns and is used when we have to represent a part-whole hierarchy. When we need to create a structure in a way that the objects in the structure have to be treated the same way, we can apply the composite design pattern. Let’s understand it with a real-life example – A diagram is a structure that consists of Objects such as Circle, Lines, Triangle, etc and when we fill the drawing with color (say Red), the same color also gets applied to the Objects in the drawing. Here drawing is made up of different parts and they all have the same operations.

### 3. Proxy Pattern

Proxy pattern intent is to “Provide a surrogate or placeholder for another object to control access to it”. The definition itself is very clear and proxy pattern is used when we want to provide controlled access of a functionality. Let’s say we have a class that can run some command on the system. Now if we are using it, it’s fine but if we want to give this program to a client application, it can have severe issues because the client program can issue a command to delete some system files or change some settings that you don’t want.

### 4. Flyweight Pattern

The flyweight design pattern is used when we need to create a lot of Objects of a class. Since every object consumes memory space that can be crucial for low memory devices, such as mobile devices or embedded systems, the flyweight design pattern can be applied to reduce the load on memory by sharing objects. String Pool implementation in java is one of the best examples of Flyweight pattern implementation.

### 5. Facade Pattern

Facade Pattern is used to help client applications to easily interact with the system. Suppose we have an application with a set of interfaces to use MySql/Oracle database and to generate different types of reports, such as HTML report, PDF report, etc. So we will have a different set of interfaces to work with different types of databases. Now a client application can use these interfaces to get the required database connection and generate reports. But when the complexity increases or the interface behavior names are confusing, the client application will find it difficult to manage it. So we can apply Facade pattern here and provide a wrapper interface on top of the existing interface to help client application.

### 6. Bridge Pattern

When we have interface hierarchies in both interfaces as well as implementations, then the bridge design pattern is used to decouple the interfaces from implementation and hiding the implementation details from the client programs. Like the Adapter pattern, it’s one of the Structural design patterns. The implementation of bridge design pattern follows the notion to prefer Composition over inheritance.

### 7. Decorator Pattern

The decorator design pattern is used to modify the functionality of an object at runtime. At the same time, other instances of the same class will not be affected by this, so individual object gets the modified behavior. The decorator design pattern is one of the structural design patterns (such as Adapter Pattern, Bridge Pattern, Composite Pattern) and uses abstract classes or interface with the composition to implement. We use inheritance or composition to extend the behavior of an object but this is done at compile-time and it’s applicable to all the instances of the class.

### 1. Template Method Pattern

Template Method is a behavioral design pattern and it’s used to create a method stub and deferring some of the steps of implementation to the subclasses. The template method defines the steps to execute an algorithm and it can provide a default implementation that might be common for all or some of the subclasses. Suppose we want to provide an algorithm to build a house. The steps that need to be performed to build a house are – building a foundation, building pillars, building walls, and windows. The important point is that we can’t change the order of execution because we can’t build windows before building the foundation. So, in this case, we can create a template method that will use different methods to build the house.

### 2. Mediator Pattern

The mediator design pattern is used to provide a centralized communication medium between different objects in a system. The mediator design pattern is very helpful in an enterprise application where multiple objects are interacting with each other. If the objects interact with each other directly, the system components are tightly coupled with each other which makes maintainability cost higher and not flexible to extend easily. The mediator pattern focuses on to provide a mediator between objects for communication and help in implementing lose-coupling between objects. Air traffic controller is a great example of a mediator pattern where the airport control room works as a mediator for communication between different flights. The mediator works as a router between objects and it can have it’s own logic to provide a way of communication.

## **3.Chain of Responsibility Design Pattern**

Chain of responsibility pattern is used to achieve loose coupling in software design where a request from client is passed to a chain of objects to process them. Then the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

# 4. Observer Design Pattern in Java

**Observer Pattern** is one of the **behavioral design pattern**. Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change. In observer pattern, the object that watch on the state of another object are called **Observer** and the object that is being watched is called **Subject**.

# 5. Strategy Design Pattern in Java

Strategy design pattern is one of the **behavioral design pattern**. Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.

## 6. Command Pattern

Command Pattern is one of the Behavioral Design Pattern. Command design pattern is used to implement **loose coupling** in a request-response model.

In command pattern, the request is send to the invoker and invoker pass it to the encapsulated command object. Command object passes the request to the appropriate method of Receiver to perform the specific action. The client program create the receiver object and then attach it to the Command. Then it creates the invoker object and attach the command object to perform an action. Now when client program executes the action, it’s processed based on the command and receiver object.

## 7. State Design Pattern

State design pattern is one of the behavioral design pattern. State design pattern is used when an Object change its behavior based on its internal state.

If we have to change the behavior of an object based on its state, we can have a state variable in the Object. Then use **if-else** condition block to perform different actions based on the state. State design pattern is used to provide a systematic and loosely coupled way to achieve this through Context and State implementations. State Pattern **Context** is the class that has a State reference to one of the concrete implementations of the State. Context forwards the request to the state object for processing

## 8. Visitor Design Pattern

Visitor pattern is used when we have to perform an operation on a group of similar kind of Objects. With the help of visitor pattern, we can move the operational logic from the objects to another class. For example, think of a Shopping cart where we can add different type of items (Elements). When we click on checkout button, it calculates the total amount to be paid. Now we can have the calculation logic in item classes or we can move out this logic to another class using visitor pattern

### 9. Interpreter Pattern

The interpreter pattern is used to define a grammatical representation of a language and provides an interpreter to deal with this grammar.

### 10. Iterator Pattern

The iterator pattern is one of the behavioral patterns and is used to provide a standard way to traverse through a group of objects. The iterator pattern is widely used in [Java Collection Framework](https://www.digitalocean.com/community/tutorials/collections-in-java-tutorial) where the iterator interface provides methods for traversing through a Collection. This pattern is also used to provide different kinds of iterators based on our requirements. The iterator pattern hides the actual implementation of traversal through the Collection and client programs use iterator methods.

### 11. Memento Pattern

The memento design pattern is used when we want to save the state of an object so that we can restore it later on. This pattern is used to implement this in such a way that the saved state data of the object is not accessible outside of the Object, this protects the integrity of saved state data.

Memento pattern is implemented with two Objects – originator and caretaker. The originator is the Object whose state needs to be saved and restored, and it uses an inner class to save the state of Object. The inner class is called “Memento”, and it’s private so that it can’t be accessed from other objects.

## Microservices Design Patterns:

## 1. Decomposition Patterns

### a. Decompose by Business Capability

#### **Problem**

Microservices is all about making services loosely coupled, applying the single responsibility principle. However, breaking an application into smaller pieces has to be done logically. How do we decompose an application into small services?

#### **Solution**

One strategy is to decompose by business capability. A business capability is something that a business does in order to generate value. The set of capabilities for a given business depend on the type of business. For example, the capabilities of an insurance company typically include sales, marketing, underwriting, claims processing, billing, compliance, etc. Each business capability can be thought of as a service, except it’s business-oriented rather than technical.

## 2. Integration Patterns

### a. API Gateway Pattern

#### **Problem**

When an application is broken down to smaller microservices, there are a few concerns that need to be addressed:

1. How to call multiple microservices abstracting producer information.
2. On different channels (like desktop, mobile, and tablets), apps need different data to respond for the same backend service, as the UI might be different.
3. Different consumers might need a different format of the responses from reusable microservices. Who will do the data transformation or field manipulation?
4. How to handle different type of Protocols some of which might not be supported by producer microservice.

#### **Solution**

An API Gateway helps to address many concerns raised by microservice implementation, not limited to the ones above.

1. An API Gateway is the single point of entry for any microservice call.
2. It can work as a proxy service to route a request to the concerned microservice, abstracting the producer details.
3. It can fan out a request to multiple services and aggregate the results to send back to the consumer.
4. One-size-fits-all APIs cannot solve all the consumer's requirements; this solution can create a fine-grained API for each specific type of client.
5. It can also convert the protocol request (e.g. AMQP) to another protocol (e.g. HTTP) and vice versa so that the producer and consumer can handle it.
6. It can also offload the authentication/authorization responsibility of the microservice.

### b. Aggregator Pattern

#### **Problem**

We have talked about resolving the aggregating data problem in the API Gateway Pattern. However, we will talk about it here holistically. When breaking the business functionality into several smaller logical pieces of code, it becomes necessary to think about how to collaborate the data returned by each service. This responsibility cannot be left with the consumer, as then it might need to understand the internal implementation of the producer application.

#### **Solution**

The Aggregator pattern helps to address this. It talks about how we can aggregate the data from different services and then send the final response to the consumer. This can be done in two ways:

1. A **composite microservice** will make calls to all the required microservices, consolidate the data, and transform the data before sending back.

2. An **API Gateway** can also partition the request to multiple microservices and aggregate the data before sending it to the consumer.

It is recommended if any business logic is to be applied, then choose a composite microservice. Otherwise, the API Gateway is the established solution.

## 3. Database Patterns

### a. Database per Service

#### **Problem**

There is a problem of how to define database architecture for microservices. Following are the concerns to be addressed:

1. Services must be loosely coupled. They can be developed, deployed, and scaled independently.

2. Business transactions may enforce invariants that span multiple services.

3. Some business transactions need to query data that is owned by multiple services.

4. Databases must sometimes be replicated and sharded in order to scale.

5. Different services have different data storage requirements.

#### **Solution**

To solve the above concerns, one database per microservice must be designed; it must be private to that service only. It should be accessed by the microservice API only. It cannot be accessed by other services directly. For example, for relational databases, we can use private-tables-per-service, schema-per-service, or database-server-per-service. Each microservice should have a separate database id so that separate access can be given to put up a barrier and prevent it from using other service tables.

### c. Command Query Responsibility Segregation (CQRS)

#### **Problem**

Once we implement database-per-service, there is a requirement to query, which requires joint data from multiple services — it's not possible. Then, how do we implement queries in microservice architecture?

#### **Solution**

CQRS suggests splitting the application into two parts — the command side and the query side. The command side handles the Create, Update, and Delete requests. The query side handles the query part by using the materialized views. The **event sourcing pattern** is generally used along with it to create events for any data change. Materialized views are kept updated by subscribing to the stream of events.

### d. Saga Pattern

#### **Problem**

When each service has its own database and a business transaction spans multiple services, how do we ensure data consistency across services? For example, for an e-commerce application where customers have a credit limit, the application must ensure that a new order will not exceed the customer’s credit limit. Since Orders and Customers are in different databases, the application cannot simply use a local ACID transaction.

#### **Solution**

A Saga represents a high-level business process that consists of several sub requests, which each update data within a single service. Each request has a compensating request that is executed when the request fails. It can be implemented in two ways:

1. Choreography — When there is no central coordination, each service produces and listens to another service’s events and decides if an action should be taken or not.
2. Orchestration — An orchestrator (object) takes responsibility for a saga’s decision making and sequencing business logic.

## 4. Observability Patterns

### c. Distributed Tracing

#### **Problem**

In microservice architecture, requests often span multiple services. Each service handles a request by performing one or more operations across multiple services. Then, how do we trace a request end-to-end to troubleshoot the problem?

#### **Solution**

We need a service which

* Assigns each external request a unique external request id.
* Passes the external request id to all services.
* Includes the external request id in all log messages.
* Records information (e.g. start time, end time) about the requests and operations performed when handling an external request in a centralized service.

Spring Cloud Slueth, along with Zipkin server, is a common implementation.

## 5. Cross-Cutting Concern Patterns

### a. External Configuration

#### **Problem**

A service typically calls other services and databases as well. For each environment like dev, QA, UAT, prod, the endpoint URL or some configuration properties might be different. A change in any of those properties might require a re-build and re-deploy of the service. How do we avoid code modification for configuration changes?

#### **Solution**

Externalize all the configuration, including endpoint URLs and credentials. The application should load them either at startup or on the fly.

Spring Cloud config server provides the option to externalize the properties to GitHub and load them as environment properties. These can be accessed by the application on startup or can be refreshed without a server restart.

### b. Service Discovery Pattern

#### **Problem**

When microservices come into the picture, we need to address a few issues in terms of calling services:

1. With container technology, IP addresses are dynamically allocated to the service instances. Every time the address changes, a consumer service can break and need manual changes.
2. Each service URL has to be remembered by the consumer and become tightly coupled.

So how does the consumer or router know all the available service instances and locations?

#### **Solution**

A service registry needs to be created which will keep the metadata of each producer service. A service instance should register to the registry when starting and should de-register when shutting down. The consumer or router should query the registry and find out the location of the service. The registry also needs to do a health check of the producer service to ensure that only working instances of the services are available to be consumed through it. There are two types of service discovery: client-side and server-side. An example of client-side discovery is Netflix Eureka and an example of server-side discovery is AWS ALB.

### c. Circuit Breaker Pattern

#### **Problem**

A service generally calls other services to retrieve data, and there is the chance that the downstream service may be down. There are two problems with this: first, the request will keep going to the down service, exhausting network resources and slowing performance. Second, the user experience will be bad and unpredictable. How do we avoid cascading service failures and handle failures gracefully?

#### **Solution**

The consumer should invoke a remote service via a proxy that behaves in a similar fashion to an electrical circuit breaker. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period, all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker resumes normal operation. Otherwise, if there is a failure, the timeout period begins again.

Netflix Hystrix is a good implementation of the circuit breaker pattern. It also helps you to define a fallback mechanism which can be used when the circuit breaker trips. That provides a better user experience.

### d. Blue-Green Deployment Pattern

#### **Problem**

With microservice architecture, one application can have many microservices. If we stop all the services then deploy an enhanced version, the downtime will be huge and can impact the business. Also, the rollback will be a nightmare. How do we avoid or reduce downtime of the services during deployment?

#### **Solution**

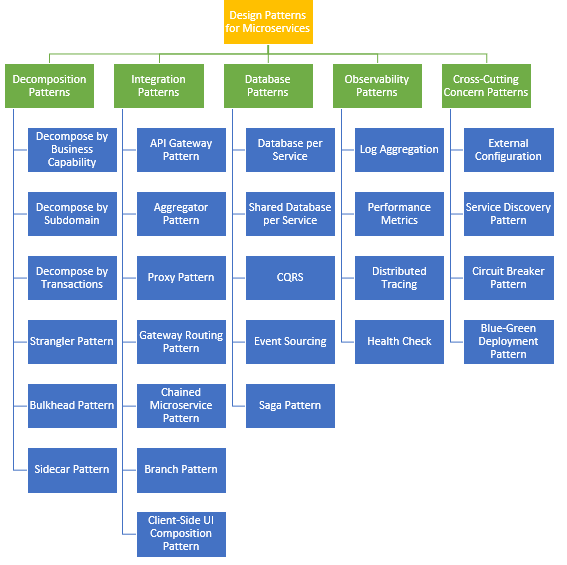
The blue-green deployment strategy can be implemented to reduce or remove downtime. It achieves this by running two identical production environments, Blue and Green. Let's assume Green is the existing live instance and Blue is the new version of the application. At any time, only one of the environments is live, with the live environment serving all production traffic. All cloud platforms provide options for implementing a blue-green deployment.

1. Reduce Cost: MSA will reduce the overall cost of designing, implementing, and maintaining IT services.
2. Increase Release Speed: MSA will increase the speed from idea to deployment of services.
3. Improve Resilience: MSA will improve the resilience of our service network.
4. Enable Visibility: MSA support for better visibility on your service and network.

You need to understand what principles microservice architecture has been built

* Scalability
* Availability
* Resiliency
* Flexibility
* Independent, autonomous
* Decentralized governance
* Failure isolation
* Auto-Provisioning
* Continuous delivery through DevOps

Add hearing to the above principles, brings several challenges and issues while bring your solution or system to live. Those problems are common for many solutions. Those can overcome with using correct and matching design patterns. There are design patterns for microservices and those can divide into five Patterns. Each many contains many patterns.



Design Patterns for Microservices

# **Decomposition Patterns**

**Decompose by Business Capability**

Microservices is all about making services loosely coupled, applying the single responsibility principle. It decomposes by business capability. Define services corresponding to business capabilities. A business capability is a concept from business architecture modeling [2]. It is something that a business does in order to generate value. A business capability often corresponds to a business object, e.g.

* Order Management is responsible for orders
* Customer Management is responsible for customers

**Decompose by Subdomain**

Decomposing an application using business capabilities might be a good start, but you will come across so-called “God Classes” which will not be easy to decompose. These classes will be common among multiple services. Define services corresponding to Domain-Driven Design (DDD) subdomains. DDD refers to the application’s problem space — the business — as the domain. A domain is consists of multiple subdomains. Each subdomain corresponds to a different part of the business.

Subdomains can be classified as follows:

* Core — key differentiator for the business and the most valuable part of the application
* Supporting — related to what the business does but not a differentiator. These can be implemented in-house or outsourced
* Generic — not specific to the business and are ideally implemented using off the shelf software

The subdomains of an Order management include:

* Product catalog service
* Inventory management services
* Order management services
* Delivery management services

**Decompose by Transactions / Two-phase commit (2pc) pattern**

You can decompose services over the transactions. Then there will be multiple transactions in the system. One of the important participants in a distributed transaction is the transaction coordinator [3]. The distributed transaction consists of two steps:

* Prepare phase — during this phase, all participants of the transaction prepare for commit and notify the coordinator that they are ready to complete the transaction
* Commit or Rollback phase — during this phase, either a commit or a rollback command is issued by the transaction coordinator to all participants

The problem with 2PC is that it is quite slow compared to the time for operation of a single microservice. Coordinating the transaction between microservices, even if they are on the same network, can really slow the system down, so this approach isn’t usually used in a high load scenario.

**Strangler Pattern**

Above three, design patterns that you go through were decomposing applications for Greenfield, but 80% of the work you do is with brownfield applications, which are big, monolithic applications (legacy codebase). The Strangler pattern comes to the rescue or solution. This creates two separate applications that live side by side in the same URI space. Over time, the newly refactored application “strangles” or replaces the original application until finally, you can shut off the monolithic application. The Strangler Application steps are transform, coexist, and eliminate [4]:

* Transform — Create a parallel new site with modern approaches.
* Coexist — Leave the existing site where it is for a time. Redirect from the existing site to the new one so the functionality is implemented incrementally.
* Eliminate — Remove the old functionality from the existing site.

**Bulkhead Pattern**

Isolate elements of an application into pools so that if one fails, the others will continue to function. This pattern is named Bulkhead because it resembles the sectioned partitions of a ship’s hull. Partition service instances into different groups, based on consumer load and availability requirements. This design helps to isolate failures, and allows you to sustain service functionality for some consumers, even during a failure.

**Sidecar Pattern**

Deploy components of an application into a separate processor container to provide isolation and encapsulation. This pattern can also enable applications to be composed of heterogeneous components and technologies. This pattern is named Sidecar because it resembles a sidecar attached to a motorcycle. In the pattern, the sidecar is attached to a parent application and provides supporting features for the application. The sidecar also shares the same lifecycle as the parent application, is created and retired alongside the parent. The sidecar pattern is sometimes referred to as the sidekick pattern and is the last decomposition pattern that we show in the post.

# Integration Patterns

**API Gateway Pattern**

When an application is broken down to smaller microservices, there are a few concerns that need to be addressed

* There are multiple calls for multiple microservices by different channels
* There is a need for handling different type of Protocols
* Different consumers might need a different format of the responses

An API Gateway helps to address many concerns raised by the microservice implementation, not limited to the ones above.

* An API Gateway is the single point of entry for any microservice call.
* It can work as a proxy service to route a request to the concerned microservice.
* It can aggregate the results to send back to the consumer.
* This solution can create a fine-grained API for each specific type of client.
* It can also convert the protocol request and respond.
* It can also offload the authentication/authorization responsibility of the microservice.

**Aggregator Pattern**

When breaking the business functionality into several smaller logical pieces of code, it becomes necessary to think about how to collaborate the data returned by each service. This responsibility cannot be left with the consumer.  
The Aggregator pattern helps to address this. It talks about how we can aggregate the data from different services and then send the final response to the consumer. This can be done in two ways [6]:

1. A composite microservice will make calls to all the required microservices, consolidate the data, and transform the data before sending back.
2. An API Gateway can also partition the request to multiple microservices and aggregate the data before sending it to the consumer.

It is recommended if any business logic is to be applied, then choose a composite microservice. Otherwise, the API Gateway is the established solution.

**Proxy Pattern**

API gateway we just expose Microservices over API gateway. I allow to get API features such as security and categorizing APIs in GW. In this example, the API gateway has three API modules:

* Mobile API, which implements the API for the FTGO mobile client
* Browser API, which implements the API to the JavaScript application running in the browser
* Public API, which implements the API for third-party developers

**Gateway Routing Pattern**

The API gateway is responsible for request routing. An API gateway implements some API operations by routing requests to the corresponding service. When it receives a request, the API gateway consults a routing map that specifies which service to route the request to. A routing map might, for example, map an HTTP method and path to the HTTP URL of service. This function is identical to the reverse proxying features provided by web servers such as NGINX.

**Chained Microservice Pattern**

There will be multiple dependencies of for single services or microservice eg: Sale microservice has dependency products microservice and order microservice. Chained microservice design pattern will help you to provide the consolidated outcome to your request. The request received by a microservice-1, which is then communicating with microservice-2 and it may be communicating with microservice-3. All these services are synchronous calls.

**Branch Pattern**

A microservice may need to get the data from multiple sources including other microservices. Branch microservice pattern is a mix of Aggregator & Chain design patterns and allows simultaneous request/response processing from two or more microservices. The invoked microservice can be chains of microservices. Brach pattern can also be used to invoke different chains of microservices, or a single chain, based your business needs.

**Client-Side UI Composition Pattern**

When services are developed by decomposing business capabilities/subdomains, the services responsible for user experience have to pull data from several microservices. In the monolithic world, there used to be only one call from the UI to a backend service to retrieve all data and refresh/submit the UI page. However, now it won’t be the same. With microservices, the UI has to be designed as a skeleton with multiple sections/regions of the screen/page. Each section will make a call to an individual backend microservice to pull the data. Frameworks like AngularJS and ReactJS help to do that easily. These screens are known as Single Page Applications (SPA). Each team develops a client-side UI component, such an AngularJS directive, that implements the region of the page/screen for their service. A UI team is responsible for implementing the page skeletons that build pages/screens by composing multiple, service-specific UI components.

Database Patterns

Defining the database architecture for microservices we need to consider below points.

1. Services must be loosely coupled. They can be developed, deployed, and scaled independently.
2. Business transactions may enforce invariants that span multiple services.
3. Some business transactions need to query data that is owned by multiple services.
4. Databases must sometimes be replicated and shared in order to scale.
5. Different services have different data storage requirements.

**Database per Service**

To solve the above concerns, one database per microservice must be designed; it must be private to that service only. It should be accessed by the microservice API only. It cannot be accessed by other services directly. For example, for relational databases, we can use private-tables-per-service, schema-per-service, or database-server-per-service.

**Shared Database per Service**

We have talked about one database per service being ideal for microservices. It is anti-pattern for microservices. But if the application is a monolith and trying to break into microservices, denormalization is not that easy. Later phase we can move to DB per services pattern, Till that we make follow this.A shared database per service is not ideal, but that is the working solution for the above scenario. Most people consider this an anti-pattern for microservices, but for brownfield applications, this is a good start to break the application into smaller logical pieces. This should not be applied for greenfield applications.

**Command Query Responsibility Segregation (CQRS)**

Once we implement database-per-service, there is a requirement to query, which requires joint data from multiple services. it’s not possible. CQRS suggests splitting the application into two parts — the command side and the query side.

* The command side handles the Create, Update, and Delete requests
* The query side handles the query part by using the materialized views

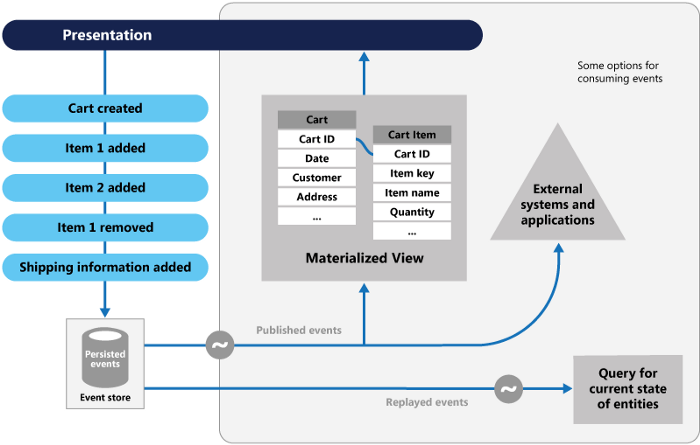
The event sourcing pattern is generally used along with it to create events for any data change. Materialized views are kept updated by subscribing to the stream of events.

**Event Sourcing**

Most applications work with data, and the typical approach is for the application to maintain the current state. For example, in the traditional create, read, update, and delete (CRUD) model a typical data process is to read data from the store. It contains limitations of locking the data with often using transactions.

The Event Sourcing pattern [8] defines an approach to handling operations on data that’s driven by a sequence of events, each of which is recorded in an append-only store. Application code sends a series of events that imperatively describe each action that has occurred on the data to the event store, where they’re persisted. Each event represents a set of changes to the data (such as AddedItemToOrder).

The events are persisted in an event store that acts as the system of record. Typical uses of the events published by the event store are to maintain materialized views of entities as actions in the application change them, and for integration with external systems. For example, a system can maintain a materialized view of all customer orders that are used to populate parts of the UI. As the application adds new orders, adds or removes items on the order, and adds shipping information, the events that describe these changes can be handled and used to update the materialized view. The figure shows an overview of the pattern.

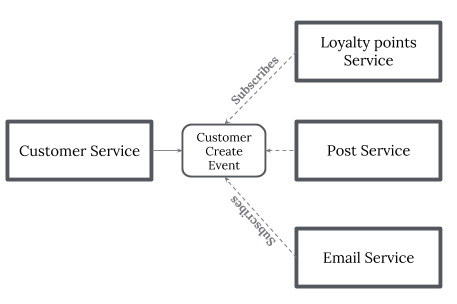


Event Sourcing pattern[8]

**Saga Pattern**

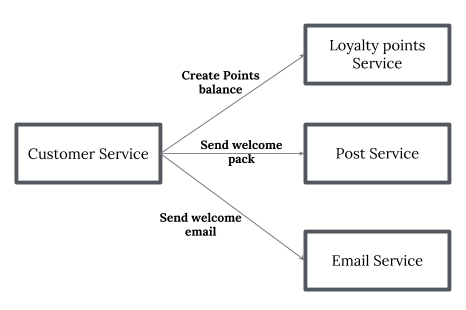
When each service has its own database and a business transaction spans multiple services, how do we ensure data consistency across services. Each request has a compensating request that is executed when the request fails. It can be implemented in two ways:

* Choreography — When there is no central coordination, each service produces and listens to another service’s events and decides if an action should be taken or not. Choreography is a way of specifying how two or more parties; none of which has any control over the other parties’ processes, or perhaps any visibility of those processes — can coordinate their activities and processes to share information and value. Use choreography when coordination across domains of control/visibility is required. You can think of choreography, in a simple scenario, as like a network protocol. It dictates acceptable patterns of requests and responses between parties.



Saga pattern — Choreography

* Orchestration — An orchestrator (object) takes responsibility for a saga’s decision making and sequencing business logic. when you have control over all the actors in a process. when they’re all in one domain of control and you can control the flow of activities. This is, of course, most often when you’re specifying a business process that will be enacted inside one organization that you have control over.



Sage pattern — Orchestration

# Observability Patterns

**Log Aggregation**

Consider a use case where an application consists of multiple services. Requests often span multiple service instances. Each service instance generates a log file in a standardized format. We need a centralized logging service that aggregates logs from each service instance. Users can search and analyze the logs. They can configure alerts that are triggered when certain messages appear in the logs. For example, PCF does have Log aggregator, which collects logs from each component (router, controller, diego, etc…) of the PCF platform along with applications. AWS Cloud Watch also does the same.

**Performance Metrics**

When the service portfolio increases due to a microservice architecture, it becomes critical to keep a watch on the transactions so that patterns can be monitored and alerts sent when an issue happens.

A metrics service is required to gather statistics about individual operations. It should aggregate the metrics of an application service, which provides reporting and alerting. There are two models for aggregating metrics:

* Push — the service pushes metrics to the metrics service e.g. NewRelic, AppDynamics
* Pull — the metrics services pulls metrics from the service e.g. Prometheus

**Distributed Tracing**

In a microservice architecture, requests often span multiple services. Each service handles a request by performing one or more operations across multiple services. While in troubleshoot it is worth to have trace ID, we trace a request end-to-end.

The solution is to introduce a transaction ID. Follow approach can be used;

* Assigns each external request a unique external request id.
* Passes the external request id to all services.
* Includes the external request id in all log messages.

**Health Check**

When microservice architecture has been implemented, there is a chance that a service might be up but not able to handle transactions. Each service needs to have an endpoint which can be used to check the health of the application, such as /health. This API should o check the status of the host, the connection to other services/infrastructure, and any specific logic.

# **Cross-Cutting Concern Patterns**

**External Configuration**

A service typically calls other services and databases as well. For each environment like dev, QA, UAT, prod, the endpoint URL or some configuration properties might be different. A change in any of those properties might require a re-build and re-deploy of the service.

To avoid code modification configuration can be used. Externalize all the configuration, including endpoint URLs and credentials. The application should load them either at startup or on the fly. These can be accessed by the application on startup or can be refreshed without a server restart.

**Service Discovery Pattern**

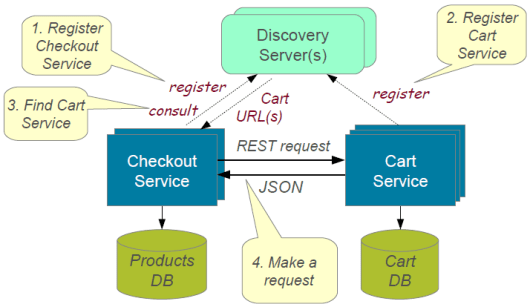
When microservices come into the picture, we need to address a few issues in terms of calling services.

With container technology, IP addresses are dynamically allocated to the service instances. Every time the address changes, a consumer service can break and need manual changes.

Each service URL has to be remembered by the consumer and become tightly coupled.

A service registry needs to be created which will keep the metadata of each producer service and specification for each. A service instance should register to the registry when starting and should de-register when shutting down. There are two types of service discovery:

* client-side : eg: Netflix Eureka
* Server-side : eg: AWS ALB.

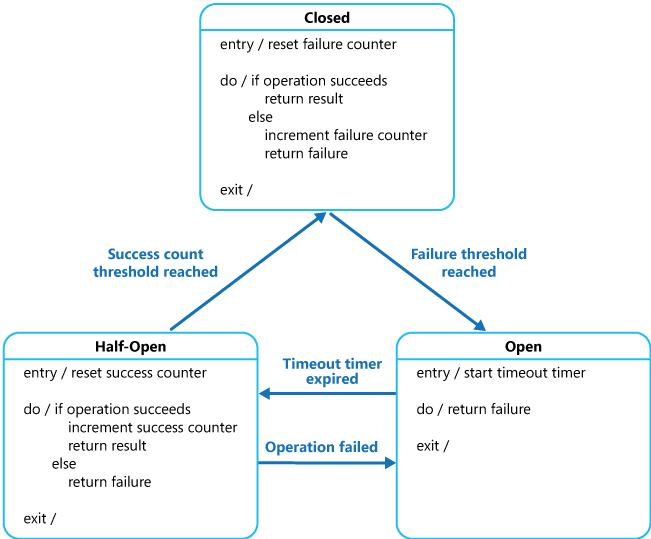


service discovery [9]

**Circuit Breaker Pattern**

A service generally calls other services to retrieve data, and there is the chance that the downstream service may be down. There are two problems with this: first, the request will keep going to the down service, exhausting network resources, and slowing performance. Second, the user experience will be bad and unpredictable.

The consumer should invoke a remote service via a proxy that behaves in a similar fashion to an electrical circuit breaker. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period, all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker resumes normal operation. Otherwise, if there is a failure, the timeout period begins again. This pattern is suited to, prevent an application from trying to invoke a remote service or access a shared resource if this operation is highly likely to fail.

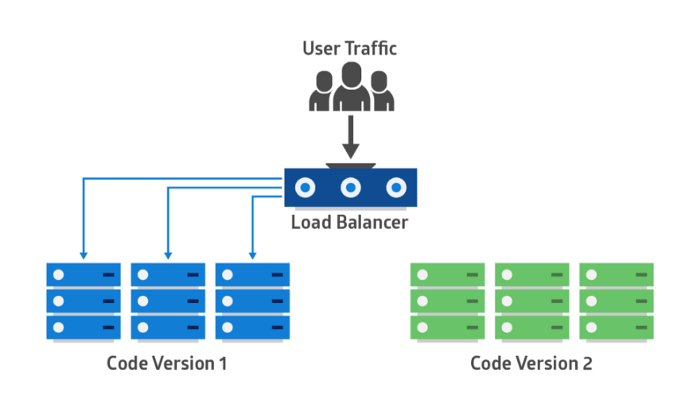


Circuit Breaker Pattern [10]

**Blue-Green Deployment Pattern**

With microservice architecture, one application can have many microservices. If we stop all the services then deploy an enhanced version, the downtime will be huge and can impact the business. Also, the rollback will be a nightmare. Blue-Green Deployment Pattern avoid this.

The blue-green deployment strategy can be implemented to reduce or remove downtime. It achieves this by running two identical production environments, Blue and Green. Let’s assume Green is the existing live instance and Blue is the new version of the application. At any time, only one of the environments is live, with the live environment serving all production traffic. All cloud platforms provide options for implementing a blue-green deployment.



12 factor Principles

# Codebase

The twelve-factor app suggests that we need to maintain **one codebase** for all environments.

With one codebase, we don’t have to recompile and package the application for each environment. Any unique environment-specific things related to the codebase should live outside of it. So, it is built once and deploy to each environment.

# Dependencies

If the codebase has dependencies, those should be isolated and publish those in some artifactory so that dependency managers like maven can download it while building the source code.

we declare the dependencies in the pom.xml file in the source code and dependency manager like Maven downloads all the dependencies while building and compiling the source code before deploying it into multiple environments.

# Backing Services

Every application has some database, restful web services to consume, etc.. Treat each of these as a separate resource that can be attached or detached from the app depending on the environment.

We should easily swap database or web service without touching the application code. When we deploy the code, we should easily attach or detach these services.

# Build, Release, Run

Separate the Build, Release and Run stages. In this way, the only way to change the application code either initiates the new build process or rollback to the previous version.

**Build:**In this phase, all the source code is compiled and packaged and referred to as build.

**Release:**In this phase, we take the build and combine with configuration and release it with some versioning. We usually put this release in some directory so that the release management tool takes it from there. These releases are ready to run in the execution environment.

**Run:**In this phase, one of the releases is selected and run it in the execution environment.

# Processes

All the twelve-factor apps have to be stateless which means any information that we want to store/save has to be done with backend resources and these resources are attached to the app at runtime. Any information should not be saved in the file or in the runtime environment.

We can conclude the app is stateless if we are able to tear down the app and get it running again without the loss of any data.

# Port Binding

All twelve-factor apps are self-contained and run on some port in the execution environment. Some routing mechanism will take care of redirecting requests to these self-contained apps.

# Concurrency

Since every twelve-factor app is self-contained and stateless, we can scale these horizontally on a demand basis. When there is a lot of traffic to these apps, we should be able to do horizontal scaling.

# Disposability

All twelve-factor apps should be designed to be torn down at any time which means disposal happens on the fly, Shutdowns should be handled gracefully. In an automated deployment environment, we should bring down and bring up the apps as quickly as possible.

# Dev/Prod Parity

We should keep all the environments as same as possible. This is useful because of the following reasons

* **Time:** we should move our changes from development to production as soon as possible. It is possible only because of similar environments
* **Personal:** All the developers are associated with deployments nowadays because of DevOps. it would be easier if we have similar environments.
* **Tools:** All the environments should use the same tools and frameworks to avoid inconsistencies.

# Logs

All the logs should be saved into an external centralized repository at least in test and prod environments. This is because when we deploy the app in the cloud, These I/O operations cause bottlenecks which leads to performance issues of the app. So, we should stream these logs to some centralized repository like Splunk or kibana, etc.. These tools expose some endpoints which would be called from our apps to log all the data.

# Admin processes

Every application has some admin processes to run while deployment. For example database migration or some scripts that should execute before you run your app. All these processes should be checked into the source code repository to have consistency across environments.

**SOLID principles**

**S — Single Responsibility Principle (known as SRP)**

The name itself suggest that the “class should be having one and only one responsibility”. What does it mean? Well let’s take the class A which does the following operations.

* Open a database connection
* Fetch data from database
* Write the data in an external file

The issue with this class is that it handles lot of operations. Suppose any of the following change happens in future.

* New database
* Adopt ORM to manage queries on database
* Change in the output structure

So in all the cases the above class would be changed. Which might affect the implementation of the other two operations as well. So ideally according to SRP there should be three classes each having the single responsibility.

We should always maintain high cohesion in a class.

Loose coupling(inter dependency) helps in maintaining SRP.

Frequency of changes with in a class may introduce bugs.

**O — Open/Closed Principle**

This principle suggests that “classes should be open for extension but closed for modification”. What is means is that if the class A is written by the developer AA, and if the developer BB wants some modification on that then developer BB should be easily do that by extending class A, but not by modifying class A.

The easy example would be the RecyclerView.Adapter class. Developers can easily extend this class and create their own custom adapter with custom behaviour without modifying the existing RecyclerView.Adapter class.

New features getting added to the software component should not have to modify the existing component.

A software component should be extendible to add a new feature or add a new behaviour to it.

**L — Liskov’s Substitution Principle**

This principle suggests that “parent classes should be easily substituted with their child classes without blowing up the application”. Let’s take following example to understand this.

Let’s consider an Animal parent class.

**public class** Animal {  
 **public void** makeNoise() {  
 System.***out***.println(**"I am making noise"**);  
 }  
}

Now let’s consider the Cat and Dog classes which extends Animal.

**public class** Dog **extends** Animal {  
 @Override  
 **public void** makeNoise() {  
 System.***out***.println(**"bow wow"**);  
 }  
}  
  
**public class** Cat **extends** Animal {  
 @Override  
 **public void** makeNoise() {  
 System.***out***.println(**"meow meow"**);  
 }  
}

Now, wherever in our code we were using Animal class object we must be able to replace it with the Dog or Cat without exploding our code. What do we mean here is the child class should not implement code such that if it is replaced by the parent class then the application will stop running. For ex. if the following class is replace by Animal then our app will crash.

**class** DumbDog **extends** Animal {  
 @Override  
 **public void** makeNoise() {  
 **throw new** RuntimeException(**"I can't make noise"**);  
 }  
}

Objects should be replaceable with their subtypes without affecting the correctness of the program.

**I — Interface Segregation Principle**

This principle suggests that “many client specific interfaces are better than one general interface”. This is the first principle which is applied on interface, all the above three principles applies on classes. Let’s take following example to understand this principle.

In android we have multiple click listeners like OnClickListener as well as OnLongClickListener.

*/\*\*  
 \* Interface definition for a callback to be invoked when a view is clicked.  
 \*/*public interface OnClickListener {  
 */\*\*  
 \* Called when a view has been clicked.  
 \*  
 \** ***@param*** *v The view that was clicked.  
 \*/* void onClick(View v);  
}*/\*\*  
 \* Interface definition for a callback to be invoked when a view has been clicked and held.  
 \*/*public interface OnLongClickListener {  
 */\*\*  
 \* Called when a view has been clicked and held.  
 \*  
 \** ***@param*** *v The view that was clicked and held.  
 \*  
 \** ***@return*** *true if the callback consumed the long click, false otherwise.  
 \*/* boolean onLongClick(View v);  
}

Why do we need two interfaces just for same action with one single tap and another with long press. Why can’t we have following interface.

public interface MyOnClickListener {  
 void onClick(View v);boolean onLongClick(View v);  
}

If we have this interface then it will force clients to implement onLongClick even if they don’t even care about long press. Which will lead to overhead of unused methods. So having two separate interfaces helps in removing the unused methods. If any client want both behaviours then they can implement both interfaces.

No client should be forced to depend methods it does not use.

**D — Dependency Inversion Principle**

This principle suggest that “classes should depend on abstraction but not on concretion”. What does it mean that we should be having object of interface which helps us to communicate with the concrete classes. What do we gain from this is, we hide the actual implementation of class A from the class B. So if class A changes the class B doesn’t need to care or know about the changes.

1. The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules; both should depend on abstractions.

Abstractions should not depend on details. Details should depend upon abstractions.

1. Code that doesn't follow this principle can be too coupled, which means you will have a hard time managing the project.
2. Ex:
3. budget reporting system that reads from a database. code violates the dependency inversion principle because our high-level module BudgetReport concretely depends on the low-level module MySQLDatabase.
4. To fix this is pretty simple, instead of concretely relying on the database class, we should use an abstraction. We will create a DatabaseInterface which will implement any kind of database we need and finally we can inject our database through the constructor.

Top 10 Java Coding Standards

## 1. Variable Scopes, Readability, and Lambda Expression

In Java, every variable declared has a scope. This means that the visibility and use of the variables must be restricted within the scope only. In case of a local variable, it is visible from the point of its declaration to the end of the method or code block it is declared in. It is a good practice to declare a variable close to the point of its possible use. This not only enhances readability of the code but also makes debugging simpler.

## 2. Class Fields

In Java, methods either belong to a class or to an interface. Therefore, there is a chance that local variables may be given the same name as a class member variable unintentionally by the programmer. The Java compiler, however, is able to pick the right one from the scope. Also, modern IDEs are sophisticated enough to identify the conflict. In any case, programmers themselves should be responsible enough to avoid such conflicts because the result can be quite disastrous. The following code illustrates how we get a very different result if we do not take care of conflicting variable names.

Another way to avoid conflict with the field names of the class with local variables is by using the this keyword. For example this.commission would always refer to the class field only. But, experience shows it’s better to use different names altogether in methods other than constructors to avoid field name conflict.

## 3. Treating Method Arguments as Local Variables

In Java, a variable once declared can be reused. Therefore, the non-final local variables declared in the method arguments can also be reused with a different value. However, this is not a good idea because the variable sends as a method argument what is supposed to hold the value it has brought, the original value that the method with work upon. If we change the value, we’ll completely lose the original content that is brought forth in the method. Instead, what we must do is copy the value to another variable and do the necessary processing. We, however, can completely restrict and make the argument variable a constant by using the final keyword as follows.

## 4. Boxing and Unboxing

In Java, boxing and unboxing are the flipside of the same technique called conversion between primitive types and their corresponding wrapper classes. Wrapper classes are nothing, but are the “class” version of primitive types such as int to Integer, float to Float, double to Double, char to Character, byte to Byte, Boolean to Boolean, and so on. Boxing is converting, say, int to Integer and unboxing is converting back from Integer to int. The compiler often performs the conversion behind the scenes; this is called autoboxing. But, sometimes, this autoboxing may produce an unexpected result and we must be aware of that.

## 5. Interfaces

Because interfaces are not tied to any method implementation (except default methods), we must use them adequately where we do not need concrete classes. Interfaces are like contracts with ample freedom and flexibility to lay out the contract. Concrete classes can avail themselves of the flexibility to enrich and replenish its business agenda. This is particularly true and can often be seen when the implementation involves an external system or services.

## 6. Strings

No other types in Java have be extended as much as String. Java strings are represented in UTF-16 format and are immutable objects. This means that every time we perform an operation like concatenation, that needs modification of the original string, a new string object is created. This intermediate string object, created only to perform the needed operation, is a waste and inefficient. This is because intermediary object creation is extraneous; it involves garbage collection, although we can avoid it. There are two companion string classes, called StringBuffer and StringBuilder, which can aptly facilitate the kind of string manipulation we may need. These two classes are built for that. The only difference between StringBuffer and StringBuilder is that the former is thread-safe. We can use these two classes whenever need extensive string manipulation rather than using the immutable String instance.

7. Naming Conventions

Java naming conventions are set of rules to make Java code look uniform across Java projects and the library. They are not strict rules, but a guideline to adhere to as a good programming practice. It is, therefore, not a good idea to violate the sanctity of the code uniformity either due to haste or rebellion. The rules are pretty simple.

* Package names are types in lowercase: *javax.sql, org.junit, java.lang*.
* Class, enum, interface, and annotation names are typed in uppercase: *Thread, Runnable, @Override*.
* Methods and field names are typed in a camel case: *deleteCharAt, add, isNull*.
* Constants are types in uppercase separated by an underscore: SIZE, MIN\_VALUE.
* Local variables are typed in camel case: *employeeName, birthDate, email.*

These conventions have become so much apart of Java that almost every programmer follows them religiously. As a result, any change in this looks outlandish and wrong. An obvious benefit of following these conventions is that the style of code aligns with the library or framework code. It also helps other programmers to quickly pick up the code when they have to, therefore leveraging overall readability of the code.

8. Standard Libraries

Java is known for its rich set of libraries. However, it is not that every library is perfectly designed but for the most part of it they are optimal. Java releases new libraries every now and then and improves the existing ones actively. Therefore, one should always use classes, methods, interfaces, enums, and annotations from the library as much as possible. This can reduce production time considerably. Moreover, the features included are well tested. It is better to use a them from the library whenever required rather than reinvent the wheel.

Because the Java platform evolves frequently, it is extremely important that we keep an eye on the features which we include from the third-party libraries or frameworks are already present in the Java Standard Library. The rule of thumb is to first exhaust the in-house resource before culling support from external sources.

9. Immutability

The concept of immutability is very important. We must decide whether the classes we intend to design can be made immutable or not, because immutability guarantees that it could be used almost everywhere without any trouble from concurrent modification. Unfortunately, not all classes can be designed as immutable. But, make sure we must do so whenever we can. This makes life much easier. You can thank me later.

10. Testing

Test driven practices (TDD) are a symbol of quality code among the Java community. Testing is a part of modern-day Java development, and the Java Standard Library has the Junit framework to assist in that direction. So, a budding Java programmer should not shy away from writing test cases with the code. Try to keep tests simple and short, focusing on one thing at a time. There can be hundreds of test cases in production environment. An obvious benefit of writing test cases is that they provide immediate feedback of the features under development.

Clean code

Clean code principles lead to source code that's highly modular and thus easier to read and test. If you think of these practices as part of a house, clean code is the foundation. Implementing clean code principles is a foundational skill that pays off especially well when it's time to refactor code or bring code under test.

Clean code

Code which is easy to read and understand easily.

Reduce cognitive load

Should be concise and to the point

Should avoid unintuitive names, complex nesting and big code blocks.

Should follow best practices and patterns.

Should be easy to maintainable and extensible.

Places to look

Names

Structure and comments

Functions

Conditionals & error handling

Classes and data structures

## Why Should We Care About Clean Code?

* *Maintainable Codebase*: Any software that we develop has a productive life and during this period will require changes and general maintenance. Clean code **can help develop software that is easy to change and maintain** over time.
* *Easier Troubleshooting*: Software can exhibit unintended behavior due to a variety of internal or external factors. It may often require a quick turnaround in terms of fixes and availability. Software developed with clean coding principles **is easier to troubleshoot for problems**.
* *Faster Onboarding*: Software during its lifetime will see many developers create, update, and maintain it, with developers joining at different points in time. This requires **a quicker onboarding to keep productivity high**, and clean code helps achieve this goal.

## Characteristics of Clean Code

* *Focused*: A piece of **code should be written to solve a specific problem**. It should not do anything strictly not related to solving the given problem. This applies to all levels of abstraction in the codebase like method, class, package, or module.
* *Simple*: This is by far the most important and often ignored characteristic of clean code. The software **design and implementation must be as simple as possible**, which can help us achieve the desired outcomes. Increasing complexity in a codebase makes them error-prone and difficult to read and maintain.
* *Testable*: Clean code, while being simple, must solve the problem at hand. It must be **intuitive and easy to test the codebase, preferably in an automated manner**. This helps establish the baseline behavior of the codebase and makes it easier to change it without breaking anything.

## Clean Coding in Java

### Project Structure

While Java doesn't enforce any project structure, **it's always useful to follow a consistent pattern to organize our source files, tests, configurations, data, and other code artifacts**. Maven, a popular build tool for Java, [prescribes a particular project structure](https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html). While we may not use Maven, it's always nice to stick to a convention.

### Naming Convention

Following **naming conventions can go a long way in making our code readable and hence, maintainable**.

Java [prescribes a set of rules](https://www.oracle.com/technetwork/java/codeconventions-135099.html) to adhere to when it comes to naming anything in Java. A well-formed name does not only help in reading the code, but it also conveys a lot about the intention of the code. Let's take some examples:

* Classes: Class in terms of object-oriented concepts is a blueprint for objects which often represent real-world objects. Hence it's meaningful to use nouns to name classes describing them sufficiently:
* *Variables*: Variables in Java capture the state of the object created from a class. The name of the variable should describe the intent of the variable clearly:
* Methods: Methods in Java are always part of classes and hence generally represent an action on the state of the object created from the class. It's hence [**useful to name methods using verbs**](https://www.baeldung.com/java-pojo-class#javabeans):

### Source File Structure

A source file can contain different elements. While Java **compiler enforces some structure, a large part is fluid**. But adhering to a specific order in which to places elements in a source file can significantly improve code readability. There are multiple popular style-guides to take inspiration from, like one by [Google](https://google.github.io/styleguide/javaguide.html) and another by [Spring](https://github.com/spring-projects/spring-framework/wiki/Code-Style).

A typical ordering of elements in a source file look:

* Package statement
* Import statements
  + All static imports
  + All non-static imports
* Exactly one top-level class
  + Class variables
  + Instance variables
  + Constructors
  + Methods

**Whitespaces**

We all know that it is easier to read and understand short paragraphs compared to a large block of text. It is not very different when it comes to reading code as well. Well-placed and consistent whitespaces and blank lines can enhance the readability of the code.

The idea here is to introduce logical groupings in the code which can help organize thought processes while trying to read it through. There is no one single rule to adopt here but a general set of guidelines and an inherent intention to keep readability at the center of it:

* Two blank lines before starting static blocks, fields, constructors and inner classes
* One blank line after a method signature that is multiline
* A single space separating reserved keywords like if, for, catch from an open parentheses
* A single space separating reserved keywords like else, catch from a closing parentheses

### Indentation

Although quite trivial, almost any developer would vouch for the fact that **a well-indented code is much easier to read and understand**. There is no single convention for code indentation in Java. The key here is to either adopt a popular convention or define a private one and then follow it consistently across the organization.

* A typical best practice is to use four spaces, a unit of indentation. Please note that some guidelines suggest a tab instead of spaces. While there is no absolute best practice here, the key remains consistency!
* Normally, there should be a cap over the line length, but this can be set higher than traditional 80 owing to larger screens developers use today.
* Lastly, since many expressions will not fit into a single line, we must break them consistently:
  + Break method calls after a comma
  + Break expressions before an operator
  + Indent wrapped lines for better readability (we here at Baeldung prefer two spaces)

### Method Parameters

Parameters are essential for methods to work as per specification. But, **a long list of parameters can make it difficult for someone to read and understand the code**. So, where should we draw the line? Let's understand the best practices which may help us:

* Try to restrict the number of parameters a method accepts, three parameters can be one good choice
* Consider [**refactoring**](https://www.baeldung.com/cs/refactoring) the method if it needs more than recommended parameters, typically a long parameter list also indicate that the method may be doing multiple things
* We may consider bundling parameters into custom-types but must be careful not to dump unrelated parameters into a single type
* Finally, while we should use this suggestion to judge the readability of the code, we must not be pedantic about it

**Hardcoding**

Hardcoding values in code can often lead to multiple side effects. For instance, **it can lead to duplication, which makes change more difficult**. It can often lead to undesirable behavior if the values need to be dynamic. In most of the cases, hardcoded values can be refactored in one of the following ways:

* Consider replacing with constants or enums defined within Java
* Or else, replace with constants defined at the class level or in a separate class file
* If possible, replace with values which can be picked from configuration or environme

### Code Comments

[Code comments](https://www.baeldung.com/cs/clean-code-comments) can be **beneficial while reading code to understand the non-trivial aspects**. At the same time, care should be taken to **not include obvious things in the comments**. This can bloat comments making it difficult to read the relevant parts.

Java allows two types of comments: Implementation comments and documentation comments. They have different purposes and different formats, as well.

* Documentation/JavaDoc Comments
  + The audience here is the users of the codebase
  + The details here are typically implementation free, focusing more on the specification
  + Typically useful independent of the codebase
* Implementation/Block Comments
  + The audience here is the developers working on the codebase
  + The details here are implementation-specific
  + Typically useful together with the codebase

So, how should we optimally use them so that they are useful and contextual?

* Comments should only complement a code, if we are not able to understand the code without comments, perhaps we need to refactor it
* We should use block comments rarely, possibly to describe non-trivial design decisions
* We should use JavaDoc comments for most of our classes, interfaces, public and protected methods
* All comments should be well-formed with a proper indentation for readability

**Logging**

Anyone who has ever laid their hands onto production code for debugging has yearned for more logs at some point in time. The **importance of logs can not be over-emphasized in development in general and maintenance in particular**.

There are lots of libraries and frameworks in Java for logging, including SLF4J, Logback. While they make logging pretty trivial in a codebase, care must be given to logging best practices. An otherwise done logging can prove to be a maintenance nightmare instead of any help. Let's go through some of these best practices:

* Avoid excessive logging, think about what information might be of help in troubleshooting
* Choose log levels wisely, we may want to enable log levels selectively on production
* Be very clear and descriptive with contextual data in the log message
* Use external tools for tracing, aggregation, filtering of log messages for faster analytic

#### 1. Conventions

Using a naming convention is a great way to get started — it keeps things clear and lets you know exactly what you’re working with.

A naming convention basically means you decide you’ll call your variables by names that adhere to a certain set of rules. It can get hairy, and a lot of people don’t always agree on which is best. So to keep it simple.

#### 2. 360 no scope

The next thing, which follows nicely from using naming conventions, is using a convention to indicate variable scope. Again, there are no rules, and everyone has their own way of doing it — as long as it’s consistent throughout all of your code, it will always be clear what can be used from where.

#### 3. Say what you mean

This is pretty straightforward, but it’s probably the most common and maybe the easiest one to forget. Easily the most frustrating thing for another developer looking at your code is seeing a variable with a misleading name or, worse, named with a single letter.

#### 4. Whitespace is nice space

Using whitespace can be incredibly powerful and normally has absolutely no downside. Sometimes in languages like JavaScript where the file size of the source code itself is important, you might want your files to be small, and that whitespace can add a few extra kilobytes. When you can, keep all your whitespace during development so the code is readable. Then, use one of the many smart little programs that go through code and chop out all the whitespace just before you upload it.

#### 5. Commenting saves lives — or at least headaches

Adding comments to your code can be invaluable—they can quickly show what a complex function is doing, or maybe even explain why certain things need to happen in a certain order. They do have a downside, though, because too much commenting can have the opposite effect and actually make for messier code.

#### 6. Automate after three times

Writing slightly more technical code doesn’t mean it has to be less readable. Lines and lines of duplicate code are not only harder to read than a concise and elegant solution, but they also increase the chance for error. The great thing about programming is that you can express complex commands in tidy, reusable, and clever ways.

#### 7. The power of i

When you have a code block with multiple loops one after the other, you need different iterator variables. There is always debate about what to use, and the answer is slightly subjective, but when they’re one after another, it makes sense to declare your iterator outside of the loop and reuse it. It’s not only better to look at, as it’s always clear that “i” is your iterator variable, but it’s also slightly more efficient.

#### 8. Birds of a feather flock together (and they group similar variables)

When your projects start to get larger, your classes will likely have many variables. First, you should be keeping all of your variable declarations at the top of the page, or at the very least all together somewhere—this speeds up any kind of searching.

Second, even though they are all together, it often helps to arrange them in such a way that makes them even easier to comprehend. For example, grouping them all by what they are is a good way to go. It’s quite likely that you’ll have several types of the same object, so keep those all together in groups, and then maybe have a section for the miscellaneous ones underneath.

#### 9. Keep it functional

Mile-long function definitions are an easy way to clutter your code. Normally it’s best to take a look at what’s actually being done. If a function is doing more than its name suggests, then perhaps some of the excess functionality could be split out into its own function.

#### 10. Keep it classy

Similar to the functional problem, if there’s a large amount of functionality you’re keeping all in one place, it could be better to create a separate class to handle that functionality.

## Architecture Principles

Though SOA is one of the important architecture style helps in designing microservices.  There are few more architecture styles and design principles need to be considered while designing microservices.  They are:

### 3.2.1 Single Responsibility Principle ([Robert C Martin](https://en.wikipedia.org/wiki/Robert_Cecil_Martin))

Each microservice must be responsible for a specific feature or a functionality or aggregation of cohesive functionality.  The thump rule to apply this principle is: "Gather those things which change for the same reason, Separate those things which change for the different reason".

### 3.2.2 Domain Driven Design

Domain driven design is an  architectural principle in-line with object oriented approach. It recommends designing systems to reflect the real world domains.  It considers the business domain, elements and behaviors and interactions between business domains.  For example, in banking domain, individual microservices can be designed to handle various business functions such as retail banking, on-line banking, on-line trading etc. The retail banking microservice can offer services related to that eg. Open a bank account, cash withdraw, cash deposits, etc.

### 3.2.3 Service Oriented Architecture

The Service Oriented Architecture (SOA) is an architecture style, which enforces certain principles and philosophies.  Following are the principles of SOA to be adhered while designing microservices for cloud.

#### 3.2.3.1 Encapsulation

The services must encapsulate the internal implementation details, so that the external system utilizes the services need not worry about the internals. Encapsulation reduces the complexity and enhances the flexibility (adaptability to change) of the system.

#### 3.2.3.2 Loose Coupling

The changes in one microsystem should have zero or minimum impact on other services in the eco-system.   This principle also suggests having a loosely coupled communication methods between the microservices.  As per SOA, RESTful APIs are more suitable than Java RMI, where the later enforces a technology on other microservices.

#### 3.2.3.3 Separation of Concern

Develop the microservices based on distinct features with zero overlap with other functions. The main objective is to reduce the interaction between services so that they are highly cohesive and loosely coupled. If we separate the functionality across wrong boundaries will lead tight coupling and increased complexity between services.

The above core principles of SOA provided only a gist of SOA.  There are more principles and philosophies of SOA which nicely fits into design principles of microservices for cloud.

# Design Constraints

The design constraints (non-functional requirements) are the important decision makers while designing microservices.  The success of a system is completely depends on Availability, Scalability, Performance, Usability and Flexibility.

## 4.1 Availability

The golden rule for availability says, anticipate failures and design accordingly so that the systems will be available for 99.999% (Five Nines).  It means the system can go down only for a 5.5 minutes for an entire year.    The cluster model is used to support high availability, where it suggests having group of services run in Active-Active mode or Active-Standby model.

So while designing microservices, it must be designed for appropriate clustering and high-availability model.  The basic properties of microservices such as stateless, independent & full stack will help us to run multiple instances in parallel in active-active or active-standby mode.

## 4.2 Scalability

Microservices must be scale-able both horizontally and vertically.    Being horizontally scale-able, we can have multiple instances of the microservice to increase the performance of the system.  The design of the microservices must support horizontal scaling (scale-out).

Also microservices should be scale-able vertically (scale-in).  If a microservice is hosted in a system with medium configuration such AWS EC2 t2-small (1-core, 2-GB memory) is moved to M4 10x-large ( 40 core & 160GB memory) it should scale accordingly.  Similarly downsizing the system capacity must also be possible.

## 4.3 Performance

Performance is measured by throughput, response time (eg. 2500 TPS -transactions per second).  The performance requirements must be available in the beginning of the design phase itself. There are technologies and design choices will affect the performance.  They are:

* Synchronous or Asynchronous communication
* Blocking or Non-blocking APIs
* RESTful API or RPC
* XML or JSON , choice of
* SQL or NoSQL
* HornetQ or RabbitMQ
* MongoDB or Cassandra or CouchDB

So, appropriate technology and design decisions must be taken, to avoid re-work in the later stage.

## 4.4 Usability

Usability aspects of the design focuses on hiding the internal design, architecture, technology and other complexities to the end user or other system.  Most of the time, microservices expose APIs to the end user as well as to other microservices.  So, the APIs must be designed in a normalized way, so that it is easy to achieve the required services with minimal number of API calls.

## 4.5 Flexibility

Flexibility measures the adaptability to change.  In the microservices eco-system, where each microservice is owned by different teams and developed in agile methodology, change will happen faster than any other systems.  The microservices may not inter-operate if they don't adapt or accommodate the change in other systems.  So, there must be a proper mechanism in place to avoid such scenarios which could include publishing the APIs, documenting the functional changes, clear communication plans.

This briefly summarizes the important design constraints for microservices.

# 5. New Problem Spaces

Though there are many positives with microservices, it can create some new challenges.

## 5.1 Complete Functional Testing

The end to end functional testing will be a great challenge in microservices environment, because we might need to deploy many microservices to validate single business functionality. Each microservice might have its own way of installation and configuration.

## 5.2 Data Integrity across the eco-system

Microservice systems run independently and asynchronously, they communicate each other through proper protocols or APIs. This could result in data integrity issues momentarily or out-of-sync due to failures. So we might need additional services to monitor the data integrity issues.

## 5.3 Increased Complexity

The complexity increases many folds, when a single monolithic is split into ten to twenty microservices and introduction of load balance server, monitoring, logging and auditing servers in to the eco-systems increases the operational overhead.  Also the competency needed to manage and deploy the microservices becomes very critical, where the IT admins and DevOps engineers need to be aware of plethora of technologies used by independent agile development teams.

The articles  ["Microservices - Not a free lunch !"](http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html)  and [Service Disoriented Architecture](https://dzone.com/articles/service-disoriented) clearly warns us to be aware of issues with microservices, though they greatly support and favour this architecture style.

7 Microservices principles by Sam Newman

# Model around business concepts

# Adopt a culture of automation

# Hide internal implementation details

# Decentralize all the things

# Independently deployable

# Isolate failure

# Highly observable