|  |  |
| --- | --- |
| **Single Responsibility Principle :** A class should have only one responsibility | we have two classes Person and Account. Both have single responsibility to store their specific information. If we want to change the state of Person then we do not need to modify the class Account and vice-versa |
| **Open / Closed Principal :** Software components should be open for extension, but closed for modification | I want to authenticate username and password; I will extend the **UsernamePasswordAuthenticationFilter** and implement the **attemptAuthentication** |
| **Liscov Substitution Principle (LSP) :**  Derived types must be completely substitutable for their base types  This requires the objects of the subclasses to behave in the same way as the objects of the superclass |  |
| **Interface Segregation Principle (ISP) :**  Clients should not be forced to implement unnecessary methods which they will not use | Created an interface **Reportable** and add menthods genrateExcel() and generatePdf(), if a classs wants only implementation of  They should be like PdfReportable and ExcelReportable |
| **Dependency Inversion Principle (DIP) :**  Hight level module should not depend on the lowlevel modules, and vice versa, both should depend on abstraction |  |
| **DRY (Don’t Repeat Yourself) :**  This principle states that each small pieces of knowledge (code) may only occur exactly once in the entire system. This helps us to write scalable, maintainable and reusable code. |  |
| **KISS (Keep it simple, Stupid!):**  This principle states that try to keep each small piece of software simple and unnecessary complexity should be avoided. This helps us to write easy maintainable code. | **YAGNI (You ain't gonna need it):**  This principle states that always implement things when you actually need them never implements things before you need them. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Creational**  **Factory Method**    Define an interface for creation of object, let the sub class decide on which class should be instantiate | **Abstract Factory**    With Abstract Factory design pattern we create a concrete implementation of a Factory.    E.g. DeviceFactory can be Abstract and it can give us GoogleDeviceFactory, AppleDeviceFactory etc. With AppleDeviceFactory we will get products like- iPhone, iPad, Mac etc. With GoogleDeviceFactory we will get products like- Nexus phone, Google Nexus tablet, Google ChromeBook etc. | **Builder**    We can use Builder pattern to create complex objects with multiple options.    E.g. when we have to create a Meal in a restaurant we can use Builder pattern. We can keep adding options like- Starter, Drink, Main Course, and Dessert etc.    Main feature of Builder pattern is step-by-step building of a complex object with multiple options. | **Prototype**    A fully loaded class should be copied or cloned. |
| **Singletone**    Ensure a class has one instance and provide a global point of access to it. | **Structural**  **Adapter**    Converts the interface of a class into another interface client expects | **Bridge**  Decouples an abstraction from its implementation so that the two can vary independently | **Composite**  Composes objects into tree structures to represent part-whole hierarchies |
| **Decorator**  Attaches the additional responsibility at run time | **Façade**  Provide a unified interface to a set of interfaces in a subsystem | **Flyweight**  Uses sharing to support large number of fine-grained objects efficiently | **Proxy**  Provides a surrogate or placeholder for another object to control access to it |
| **Behavioral Patterns**  **Chain of Responsibility**  Avoid coupling the senders of request to its receiver by giving more than one object a change to handle the request | **Observer**  Defines a one-to-many dependency between object so that when one object changes state all its dependents are notified and updated automatically | **Command**  Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests and support undoable operations | **State**  Allows an object to alter its behavior when its internal state changes |
| **Interpreter**  Given a language, defines a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language | **Strategy**  Defines a family of algorithms, encapsulate each one, and make them interchangeable | **Iterator**  Given a language, defines a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language | **TemplateMethod**  Define the Skelton of an algorithm in an operation, deferring some steps to subclasses |
| **Mediator**  Defines an objects that encapsulates how a set of objects interact | **Memento**  Without violationg encapsulation, capture and extermalize an object’s internal state so that the object can be restored to this sate later |  |  |

**Microservices Design patterns**

|  |  |  |  |
| --- | --- | --- | --- |
| 1. Decomposition Pattern    1. Decompose by Business capability    2. Decompose by Subdomain    3. Strangler Pattern | 1. Integration Pattern    1. API Gateway Pattern    2. Aggregator Pattern    3. Client side UI Composition pattern | 1. Database Pattern    1. Database per service    2. Shared database per service    3. Common Query responsibility Segregation    4. Saga Pattern | 1. Observability Pattern    1. Log Aggregation    2. Performance Matrix    3. Distributed Tracing    4. Health Check |
| 1. Cross cutting concern pattern    1. External Configuration    2. Service Discovery Pattern    3. Circuit Breaker Pattern 🡪 Resilance4j 🡪    4. Blue Green Deployment Pattern 🡪 it reduces the down time or no downtime at all | 1. Sidecar, Chained Microservice, Branch Microservice, Event Sourcing Pattern, Continuous Delivery Patterns, |  |  |

Software Coding best practices

|  |  |  |  |
| --- | --- | --- | --- |
| **Use consistent indentation** There is no right or wrong indentation that everyone should follow. The best style, is a consistent style. Once you start competing in large projects you’ll immediately understand the importance of consistent code styling. | **Avoid Deep Nesting** Too many levels of nesting can make code harder to read and follow. | **Limit line length** Long lines are hard to read. It is a good practice to avoid writing horizontally long lines of code. | **File and folder structure**  You should avoid writing all of your code in one of 1-2 files. That won’t break your app but it would be a nightmare to read, debug and maintain your application later. Keeping a clean folder structure will make the code a lot more readable and maintainable. |
| **Naming conventions** Use of proper naming conventions is a well known best practice. Is a very common issue where developers use variables like X1, Y1 and forget to replace them with meaningful ones, causing confusion and making the code less readable. | **Keep the code simple** The code should always be simple. Complicated logic for achieving simple tasks is something you want to avoid as the logic one programmer implemented a requirement may not make perfect sense to another. So, always keep the code as simple as possible.  ```  if (a < 0 && b > 0 && c == 0) {     return true;  } else {     return false;  ```  return a < 0 && b > 0 && c == 0; | **Unit Testing**  **Code maintainability**  **Code Reusability**  **Dry 🡪 Don’t repeat your self** |  |

**Software Architecture and Best Practices**

[**https://www.lucidchart.com/blog/how-to-design-software-architecture**](https://www.lucidchart.com/blog/how-to-design-software-architecture)

|  |  |  |  |
| --- | --- | --- | --- |
| **Have a clear understanding of your requirements**  1. Start with a high-level view 2. Map your Functional Requirements 3. Consider non-functional requirements | **Start thinking about each component**   1. How my design should look like 2. Consider and document the implications of the requirements 3. design the final architecture later | **Divide your architecture into slices** | **Prototype**  Keep careful revision history  Have a single source of truth  Diagram the prototype |
| **Identify the non functional requirements**  **Performance:** How well your entire system performs as well as individual slices or layers  **Scalability:** Current and future potential to scale your system along with your needs  **Portability:** Your data portability as well as the potential portability of components of your system, if applicable or necessary  **Extensibility:** Accounting for your system’s and company’s future growth, how well your system can adapt and the effort involved with adaptation  **Compliance:** Another essential factor—and one with a sizable impact on your overall project design | **Best practices for Sofware architecture design**  Visualize your design  Don’t choose patterns  Rememeber that the first design is only the first iteration  Be caution son scop creep  Keep boundarsies and interfaces in mind |  |  |

## AWS

|  |  |  |  |
| --- | --- | --- | --- |
| **AWS Elastic Beanstalk** is an orchestration service offered by Amazon Web Services for deploying applications which orchestrates various AWS services, including EC2, S3, Simple Notification Service, CloudWatch, autoscaling, and Elastic Load Balancers | CloudWatch **collects monitoring and operational data in the form of logs, metrics, and events**, and visualizes it using automated dashboards so you can get a unified view of your AWS resources, applications, and services that run in AWS and on-premises. | AWS Lambda is an event-driven, serverless computing platform provided by Amazon as a part of Amazon Web Services. It is a computing service that runs code in response to events and automatically manages the computing resources required by that code |  |
|  |  |  |  |

|  |  |
| --- | --- |
| Choreography SAGA  In choreography-based SAGA, the process is managed by local transactions itself.  The benefits of this method are as follows:   1. Easy to implement. 2. Appropriate if the number of local transactions is small. 3. Loosely coupled endpoints. 4. Suitable for agile organizations.   There are some drawbacks of the method:   1. It gets very complicated when the number of local transactions increases. 2. Cyclic dependency may occur between services. | Orchestration SAGA:  is a separate service that performs orchestration  The benefits of this method are:   1. Easy to implement. 2. Easy to understand and maintain. 3. Complexity is not an issue when local transactions’ count dramatically increases.   The drawbacks are:   1. This method creates a situation that the communication channel becomes intelligent and the endpoints become a dummy. 2. All the business logic is implemented inside the orchestrator. So the orchestrator becomes the monolith. |

1. We can achieve distributed transactions in Microservices based on
   1. 2 Phase commit
      1. The Coordinator first creates a global transaction with all the context information
      2. Prepare for the Microservice (Microservice name) à once the microservice is ready to perform the operations, then object is locked for further changes and tell the co-ordinator
      3. Prepare the second microservice is ready
      4. Once the coordinator confirm the all the microservices are ready, then prepare for commit, at this point all the services are unlocated.

Advantages à Strong consistency

Disadvantages à Locking the microservices that leads to latency

1. 3 Phase commit have additional step as Pre-Commit

AWS

|  |  |  |  |
| --- | --- | --- | --- |
| AWS Lambda is an event-driven, serverless computing platform provided by Amazon as a part of Amazon Web Services. It is a computing service that runs code in response to events and automatically manages the computing resources required by that code |  |  |  |
|  |  |  |  |

**pandas** is a fast, open source data analysis and manipulation tool,  
built on top of the [Python](https://www.python.org/) programming language.

**NumPy** adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

**12 Factor App**

[**https://dzone.com/articles/12-factor-app-principles-and-cloud-native-microser**](https://dzone.com/articles/12-factor-app-principles-and-cloud-native-microser)

**12 Factor App**

|  |  |  |  |
| --- | --- | --- | --- |
| 1. Code base 🡪 one codebase tracked in revision control, many deploys | 1. Dependencies 🡪 Explicitly declare and isolate dependencies | 1. Config 🡪 Store Config in the environment | 1. Baking Service 🡪 Treat baking services as attached resources |
| 1. Build, release, run 🡪 Strictly separate build and run stages | 1. Process 🡪 Execute app as one or more stateless process | 1. Port binding 🡪 Export services via port binding | 1. Concurrency 🡪 Scale out via the process model |
| 1. Disposability 🡪 maximize robustness with fast start-up and gracefull shutdown | 1. Dev/Prod parity 🡪Keep development, stating and production as similar as possible | 1. Logs 🡪 Treat logs as event stream | 1. Admin 🡪 Run admin/management tasks as one-off process. |