https://drive.google.com/drive/folders/1gxGGsCLKSuBCtxtNr1dlAMYRFaMYWPvP

Architectural Patterns: MVC, SOA, REST, Microservices

EIP (Apache Camel / Spring Integration) : splitter/aggregator, content-based router, scatter/gather etc.,

Container:

It a pre-written progrim, provides the below features:

1. Lifecycle management

2. Declarative Services (security, transaction, exception handling, profyling, logging etc.)

\* Spring is a pojo container

Data Tier:

-> jdbc

-> hibernate -> jdbc

-> jpa -> hibernate/toplink -> jdbc

-> spring data repositories -> jpa -> hibernate/toplink -> jdbc

@PostMapping

public ResponseEntity<String> addCustomer(@RequestBody Customer customer, UriComponentsBuilder ucbuilder)

{

customerService.addCustomer(customer);

// return new ResponseEntity<String>("inserted a new resource", HttpStatus.CREATED);

HttpHeaders headers = new HttpHeaders();

headers.add("info", "new customer registration");

headers.setLocation(ucbuilder.path("/customer/{id}").buildAndExpand(customer.getCid()).toUri());

return new ResponseEntity<String>(headers, HttpStatus.CREATED);

}

@DeleteMapping("/{id}")

public ResponseEntity<String> deleteCustomer(@PathVariable("id") long customerId)

{

customerService.deleteCustomer(customerId);

return new ResponseEntity<String>(HttpStatus.NO\_CONTENT);

}

Spring Core [IOC, DI & AOP]

IOC - Inversion of control

DI - Dependency Injection

AOP - Aspect-Oriented Programming [Provides declarative services, using Proxy pattern]

Spring Bean Scope : singleton(default), prototype, request, session, custom scope

Spring Configuration Beans are annotated with @Bean (Datasource,RestTemplate, JDBCTemplate, ResourceBundle etc.,)

Spring Stero-type annotations:

Presentation Tier : @Controller/@RestController

Business Tier : @Service

Data Tier : @Repository

Common Component : @Component (ex: AOP, BeanPostProcessor )

BeanPostProcessor

|\_These are special beans, registered to the applicationcontext for global initialization

|\_ There are in-built bean post processor and custom bean post processor

In-built bean post processors : commonannotationbeanpostprocessor, AutowiredAnnotationBeanPostProcessor, @EnableAspectJAutoProxy(DefaultAdvisorAutoProxyCreator)

AOP

|\_ Aspect : It is a class, contains one or more advices

|\_ Advice : It contains cross-cutting code (security, transaction etc.,)

|\_ BeforeAdvice -> Ex: Security

|\_ AfterAdvice / AfterReturn

|\_ AroundAdvice (Before and AfterReturn) -> Ex: Transaction, Logging, Profyling

|\_ ThrowsAdvice (Exceptin handling)

|\_ Target : it is the real object (AccountBean, PaymentDao etc.,)

|\_ Pointcut : AOP query language, used to apply advices to target joinpoint (Spring supports method-level join point)

joinpoint :

|\_ Spring-AOP supports only method-level joinpoint

|\_ Aspectj supports different joinpoints (attribute, method, constructor)

To create dynamic pointcut, the class must implement the below interface:

DynamicMethodMatcherPointcut

\*BeanPostProcessor is part of the bean life cycle, for global initialization

\* Advices are uses at runtime, by calling a method on proxy object

To provide declarative services: (security, logging, transaction etc.,)

1. Servlet Container : Filters

2. EJB Containers : Interceptors

#. Pojo Container (Spring) : AOP

Transaction : Unit of work, either commit or rollback. Follows ACID properties

jdbc transaction:

connection.setAutoCommit(false);

....insert

....update

...insert

commit/rollback

Spring Dao Layer :

==============

All technology-specific(jdbc,hibernate,jpa etc.,) exceptions are converted into DataAccessException

|\_ Root of the hierarchy of data access exceptions

JPA/Hibernate

|\_ Bean-managed persisitence and transaction

|\_ Container-managed persistence and transaction

1. Bean-managed persisitence

EntityManagerFactory emf=....

EntityManager entitymanager = emf.getEntityManager();

.....

2. Container-managed persistence

@Autowired @PersistenceContext

EntityManager entitymanager;

3. Bean-managed Transaction

EntityTransaction tx = entityManager.getTransaction();

tx.begin();

......

tx.commit(); / tx.rollback();

4. Container-managed Transaction

@Transactional

There are 7 transaction attributes:

Propagation.REQUIRED (default)

Propagation.REQUIRES\_NEW

Propagation.MANDATORY

Propagation.SUPPORTS

Propagation.NOT\_SUPPORTED

Propagation.NEVER

Propagation.NESTED

Servlet : java specification to develop web component

Servlet Containers : Tomcat/jetty/undertow

SpringMVC

|\_ FrontController

|\_ DispatcherServlet (extends HttpServlet)

Web application Development:

1. Servlet + JSP + POJO

2. Spring MVC + POJO

\* Web application must be deployed into Servlet Container (Tomcat)

Tomcat:

\*context-path is used to uniquely differentiate the application

\*url-pattern is used to uniquely differentiate the resource inside the application

Spring Application context

|\_ Creates HierarchicalBeanFactory for a webapplicaton declaratively

|\_Parent-factory : Business Tier and Data Tier Components

|\_Child-factory : Spring mvc components

@SpringBootApplication

===================

It uses the below annotations:

@Configuration

|\_ Scan for configuration beans annotated with @Bean (JdbcTemplate, DataSource, SessionFactory, RestTemplate)

@ComponentScan

|\_Scan for Strerio-type annotations:

|\_ Web component : @Controller / @RestController

|\_ Business component : @Service

|\_ Data component : @Repository

|\_ Generic component : @Component (AOP)

@EnableAutoConfiguration

|\_ search for spring-boot-starter classes and provide required configuration

|\_ spring-boot-starter-web -> @EnableWebMvc

|\_ spring-boot-starter-security -> @EnableWebSecurity

|\_ spring-boot-jms -> @EnableJms

spring.jpa.hibernate.ddl-auto=update

spring.datasource.url=jdbc:mysql://localhost:3306/demo

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

12 factors app

IX. Disposability

Maximize robustness with fast startup and graceful shutdown

@RestController

|\_ @Controller, @ResponseBody

@GetMapping("hello")

|\_ @RequestMapping(path = "/hello", method = RequestMethod.GET)

url : http://localhost:8080/myapp/hello

protocol : http / https

host: localhost

port : http(80), https(443)

Context-path/root : /

Url-pattern : welcome-file-list (index.html, home.html)

jetty server (instead of tomcat)

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jetty</artifactId>

</dependency>

Spring Webservices

|\_ JAX-RS (Java Api for Restful services)

|\_ JAX-WS (Java Api for Soap web services)

JavaEE

|\_ JAAS - Java Authentication and Authorization Service

Spring Security

|\_ It support JAAS and provides additional security features

Messaging Destinations:

|\_ Queue (Point-to-Poing) - customer account alerts

|\_ Topic (Publish/Subscribe) - bank offers

Brokers : 3

Topic : Order

Partitions : 3

replication-factor : 3

start zookeeper: (port: 2181)

zookeeper-server-start.bat ..\..\config\zookeeper.properties

start kafka-server (port:9092)

kafka-server-start.bat ..\..\config\server.properties

Create a new topic:

kafka-topics.bat --bootstrap-server localhost:9092 --create --topic simpleTopic

Kafka API

|\_ Producer API, Consumer API, Stream API, Connector API

Spring integration with Kafka

|\_ Producer (KafkaTemplate)

|\_ Consumer (@KafkaListener)

Kafka console producer:

kafka-console-producer.bat --bootstrap-server localhost:9092 --topic testTopic

kafka console consumer:

kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic testTopic --from-beginning

https://drive.google.com/drive/folders/1gxGGsCLKSuBCtxtNr1dlAMYRFaMYWPvP?usp=share\_link

https://github.com/jbossramana/springboot

bit.ly/Kafka-Advance

Architectural Patterns: MVC, SOA, REST, Microservices

EIP (Apache Camel / Spring Integration) : splitter/aggregator, content-based router, scatter/gather etc.,

Container:

It a pre-written progrim, provides the below features:

1. Lifecycle management

2. Declarative Services (security, transaction, exception handling, profyling, logging etc.)

\* Spring is a pojo container

Data Tier:

-> jdbc

-> hibernate -> jdbc

-> jpa -> hibernate/toplink -> jdbc

-> spring data repositories -> jpa -> hibernate/toplink -> jdbc

Spring Core [IOC, DI & AOP]

IOC - Inversion of control

DI - Dependency Injection

AOP - Aspect-Oriented Programming [Provides declarative services, using Proxy pattern]

Spring Bean Scope : singleton(default), prototype, request, session, custom scope

Spring Configuration Beans are annotated with @Bean (Datasource,RestTemplate, JDBCTemplate, ResourceBundle etc.,)

Spring Stero-type annotations:

Presentation Tier : @Controller/@RestController

Business Tier : @Service

Data Tier : @Repository

Common Component : @Component (ex: AOP, BeanPostProcessor )

BeanPostProcessor

|\_These are special beans, registered to the applicationcontext for global initialization

|\_ There are in-built bean post processor and custom bean post processor

In-built bean post processors : commonannotationbeanpostprocessor, AutowiredAnnotationBeanPostProcessor, @EnableAspectJAutoProxy(DefaultAdvisorAutoProxyCreator)

AOP

|\_ Aspect : It is a class, contains one or more advices

|\_ Advice : It contains cross-cutting code (security, transaction etc.,)

|\_ BeforeAdvice -> Ex: Security

|\_ AfterAdvice / AfterReturn

|\_ AroundAdvice (Before and AfterReturn) -> Ex: Transaction, Logging, Profyling

|\_ ThrowsAdvice (Exceptin handling)

|\_ Target : it is the real object (AccountBean, PaymentDao etc.,)

|\_ Pointcut : AOP query language, used to apply advices to target joinpoint (Spring supports method-level join point)

joinpoint :

|\_ Spring-AOP supports only method-level joinpoint

|\_ Aspectj supports different joinpoints (attribute, method, constructor)

To create dynamic pointcut, the class must implement the below interface:

DynamicMethodMatcherPointcut

\*BeanPostProcessor is part of the bean life cycle, for global initialization

\* Advices are uses at runtime, by calling a method on proxy object

To provide declarative services: (security, logging, transaction etc.,)

1. Servlet Container : Filters

2. EJB Containers : Interceptors

#. Pojo Container (Spring) : AOP

Transaction : Unit of work, either commit or rollback. Follows ACID properties

jdbc transaction:

connection.setAutoCommit(false);

....insert

....update

...insert

commit/rollback

Spring Dao Layer :

==============

All technology-specific(jdbc,hibernate,jpa etc.,) exceptions are converted into DataAccessException

|\_ Root of the hierarchy of data access exceptions

JPA/Hibernate

|\_ Bean-managed persisitence and transaction

|\_ Container-managed persistence and transaction

1. Bean-managed persisitence

EntityManagerFactory emf=....

EntityManager entitymanager = emf.getEntityManager();

.....

2. Container-managed persistence

@Autowired @PersistenceContext

EntityManager entitymanager;

3. Bean-managed Transaction

EntityTransaction tx = entityManager.getTransaction();

tx.begin();

......

tx.commit(); / tx.rollback();

4. Container-managed Transaction

@Transactional

There are 7 transaction attributes:

Propagation.REQUIRED (default)

Propagation.REQUIRES\_NEW

Propagation.MANDATORY

Propagation.SUPPORTS

Propagation.NOT\_SUPPORTED

Propagation.NEVER

Propagation.NESTED

Servlet : java specification to develop web component

Servlet Containers : Tomcat/jetty/undertow

SpringMVC

|\_ FrontController

|\_ DispatcherServlet (extends HttpServlet)

Web application Development:

1. Servlet + JSP + POJO

2. Spring MVC + POJO

\* Web application must be deployed into Servlet Container (Tomcat)

Tomcat:

\*context-path is used to uniquely differentiate the application

\*url-pattern is used to uniquely differentiate the resource inside the application

Spring Application context

|\_ Creates HierarchicalBeanFactory for a webapplicaton declaratively

|\_Parent-factory : Business Tier and Data Tier Components

|\_Child-factory : Spring mvc components

@SpringBootApplication

===================

It uses the below annotations:

@Configuration

|\_ Scan for configuration beans annotated with @Bean (JdbcTemplate, DataSource, SessionFactory, RestTemplate)

@ComponentScan

|\_Scan for Strerio-type annotations:

|\_ Web component : @Controller / @RestController

|\_ Business component : @Service

|\_ Data component : @Repository

|\_ Generic component : @Component (AOP)

@EnableAutoConfiguration

|\_ search for spring-boot-starter classes and provide required configuration

|\_ spring-boot-starter-web -> @EnableWebMvc

|\_ spring-boot-starter-security -> @EnableWebSecurity

|\_ spring-boot-jms -> @EnableJms

spring.jpa.hibernate.ddl-auto=update

spring.datasource.url=jdbc:mysql://localhost:3306/demo

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

12 factors app

IX. Disposability

Maximize robustness with fast startup and graceful shutdown

@RestController

|\_ @Controller, @ResponseBody

@GetMapping("hello")

|\_ @RequestMapping(path = "/hello", method = RequestMethod.GET)

url : http://localhost:8080/myapp/hello

protocol : http / https

host: localhost

port : http(80), https(443)

Context-path/root : /

Url-pattern : welcome-file-list (index.html, home.html)

jetty server (instead of tomcat)

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jetty</artifactId>

</dependency>

Spring Webservices

|\_ JAX-RS (Java Api for Restful services)

|\_ JAX-WS (Java Api for Soap web services)

JavaEE

|\_ JAAS - Java Authentication and Authorization Service

Spring Security

|\_ It support JAAS and provides additional security features

Messaging Destinations:

|\_ Queue (Point-to-Poing) - customer account alerts

|\_ Topic (Publish/Subscribe) - bank offers

Brokers : 3

Topic : Order

Partitions : 3

replication-factor : 3

start zookeeper: (port: 2181)

zookeeper-server-start.bat ..\..\config\zookeeper.properties

start kafka-server (port:9092)

kafka-server-start.bat ..\..\config\server.properties

Create a new topic:

kafka-topics.bat --bootstrap-server localhost:9092 --create --topic simpleTopic

Kafka API

|\_ Producer API, Consumer API, Stream API, Connector API

Spring integration with Kafka

|\_ Producer (KafkaTemplate)

|\_ Consumer (@KafkaListener)

Kafka console producer:

kafka-console-producer.bat --bootstrap-server localhost:9092 --topic testTopic

kafka console consumer:

kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic testTopic --from-beginning

Webservice / WebAPI (CRUD)

SOAP (Message-Exchange Protocol)

--------

HTTP (Application Protocol)

--------

TCP/IP (Transport Protocol)

HTTP

|\_Methods - GET,POST,PUT,DELETE

|\_Media-Type (MIME) - Multipurpose Internet Mail Extensions (application/xml, application/json)

\* SOAP uses only HTTP-POST method of all the CRUD operations

SOAP uses XML schemas to create request.xmd, response.xsd, fault.xsd

REST - Not a protocol (An Architectural Design Pattern)

--------

HTTP (Application Protocol)

--------

TCP/IP (Transport Protocol)

REST uses all the HTTP capabilites to develop webservices

HTTP

|\_Methods - GET,POST,PUT,DELETE

|\_Media-Type (MIME) - Multipurpose Internet Mail Extensions (application/xml, application/json)

\* REST - Represenatation(Server) State Transfer(Client)

Client -> Server

(State Transfer) (Representation)

To develop REST API, using java:

|\_ JAX-RS (JSR 370: JavaTM API for RESTful Web Services (JAX-RS 2.1) Specification) -> @Get, @Put

|\_ Eclipse jersey, Apache cxf + Spring

|\_ Spring MVC (@GetMapping...)

Client

|\_ GET /order

REST API

--------------

User Registration Service - REST API (CRUD)

Resource Name : user

C -> POST

R -> GET

U -> PUT

D -> DELETE

Endpoints: /user, /user/<user-id>

/user GET, POST, DELETE

Retrieve all users

/user GET

Insert a new user

/user POST

delete all users

/user DELETE

Marshalling is the process of transforming Java objects into XML documents. Unmarshalling is the process of reading XML documents into Java objects.

/user/<user-id> GET,PUT,DELETE

To Retrieve user details , id = 101

/user?id=101 GET X

/user/101 GET (Correct)

To update user details, id=101

/user/101 PUT

To delete user details, id=101

/user/101 DELETE

Custom Queries: (Request/Query Parameter - these are application-specific)

Retrieve all users, registerd in year 2021

/user/2021 GET X

/user?year=2021 (Correct)

Retreive first 10 users, registered in the year 2021

/user?year=2021&start=1&size=10

\* Matrix parameters uses ; /user;year=2021;start=1;size=10

Architectural Patterns : MVC, REST, SOA, Microservices

REST patterns : Stateless, Pagination, Content-Negotiation

<dependency>

<groupId>com.fasterxml.jackson.dataformat</groupId>

<artifactId>jackson-dataformat-xml</artifactId>

</dependency>

Order Service class -------------

package demo.boot.service;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.stream.Collectors;

import org.springframework.stereotype.Service;

import demo.boot.model.Order;

@Service

public class OrderService implements IOrderService{

HashMap<Long,Order> orderData = new HashMap<>();

@Override

public List<Order> getAllOrders() {

return new ArrayList<Order>(orderData.values());

}

@Override

public Order getOrderById(long id) {

// TODO Auto-generated method stub

return orderData.get(id);

}

@Override

public void insertOrder(Order order) {

order.setId(orderData.size()+1);

orderData.put(order.getId(), order);

}

@Override

public void updateOrder(Order order) {

orderData.put(order.getId(),order);

}

@Override

public void deleteOrderById(long id) {

orderData.remove(id);

}

@Override

public List<Order> getOrdersByAmt(long amt) {

List<Order> orderList = new ArrayList<Order>(orderData.values()).stream()

.filter(order -> order.getAmount() > amt)

.collect(Collectors.toList());

return orderList;

}

}

Order Controller class - ------

package demo.boot.controller;

import java.util.ArrayList;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.DeleteMapping;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.PutMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.RestController;

import demo.boot.model.Order;

import demo.boot.service.IOrderService;

@RestController

@RequestMapping("/order")

public class OrderController {

@Autowired

IOrderService orderService;

@GetMapping

public List<Order> getAllOrders() {

return orderService.getAllOrders();

}

@GetMapping("/{id}")

public Order getOrderById(@PathVariable long id) {

return orderService.getOrderById(id);

}

@PostMapping

public void insertOrder(@RequestBody Order order) {

orderService.insertOrder(order);

}

@PutMapping("/{id}")

public void updateOrder(@RequestBody Order order) {

orderService.updateOrder(order);

}

@DeleteMapping("/{id}")

public void deleteOrderById(@PathVariable long id) {

orderService.deleteOrderById(id);

}

@GetMapping("/search")

public List<Order> getOrdersByAmt(@RequestParam long amt) {

return orderService.getOrdersByAmt(amt);

}

}

CustomerDao.java ----------------

@Repository

public class CustomerDao implements ICustomerDao {

@PersistenceContext

EntityManager em;

@Override

public Customer addCustomer(Customer customer) {

// TODO Auto-generated method stub

em.persist(customer);

return customer;

}

@Override

public Customer updateCustomer(Customer customer) {

em.merge(customer);

return customer;

}

@Override

public List<?> getCustomers() {

// TODO Auto-generated method stub

return em.createQuery("from Customer").getResultList();

}

@Override

public List<?> getCustomerByCountry(String countryName) {

// TODO Auto-generated method stub

return em.createQuery("from Customer where country like :country")

.setParameter("country", countryName).getResultList();

}

@Override

public Customer getCustomerById(long customerId) {

// TODO Auto-generated method stub

return em.find(Customer.class, customerId);

}

@Override

public void deleteCustomer(long customerId) {

Customer customer = em.find(Customer.class, customerId);

em.remove(customer);

}

}

CustomerService.java -----------------

@Service

@Transactional

public class CustomerService implements ICustomerService {

@Autowired

ICustomerDao customerDao;

@Override

public Customer addCustomer(Customer customer) {

// TODO Auto-generated method stub

return customerDao.addCustomer(customer);

}

@Override

public Customer updateCustomer(Customer customer) {

// TODO Auto-generated method stub

return customerDao.updateCustomer(customer);

}

@Override

public List<?> getCustomers() {

// TODO Auto-generated method stub

return customerDao.getCustomers();

}

@Override

public List<?> getCustomerByCountry(String countryName) {

// TODO Auto-generated method stub

return customerDao.getCustomerByCountry(countryName);

}

@Override

public Customer getCustomerById(long customerId) {

// TODO Auto-generated method stub

return customerDao.getCustomerById(customerId);

}

@Override

public void deleteCustomer(long customerId) {

customerDao.deleteCustomer(customerId);

}

}

CustomerController.java --------------------------

@RestController

@RequestMapping("/customer")

public class CustomerController {

@Autowired

ICustomerService customerService;

@PostMapping

public Customer addCustomer(@RequestBody Customer customer)

{

return customerService.addCustomer(customer);

}

@PutMapping("/{id}")

public Customer updateCustomer(@RequestBody Customer customer)

{

return customerService.updateCustomer(customer);

}

@GetMapping

public List<?> getCustomers()

{

return customerService.getCustomers();

}

@GetMapping("/search")

public List<?> getCustomerByCountry(@RequestParam("country") String countryName)

{

return customerService.getCustomerByCountry(countryName);

}

@GetMapping("/{id}")

public Customer getCustomerById(@PathVariable("id") long customerId)

{

return customerService.getCustomerById(customerId);

}

@DeleteMapping("/{id}")

public void deleteCustomer(@PathVariable("id") long customerId)

{

customerService.deleteCustomer(customerId);

}

}

@PostMapping

public ResponseEntity<String> addCustomer(@RequestBody Customer customer, UriComponentsBuilder ucbuilder)

{

customerService.addCustomer(customer);

// return new ResponseEntity<String>("inserted a new resource", HttpStatus.CREATED);

HttpHeaders headers = new HttpHeaders();

headers.add("info", "new customer registration");

headers.setLocation(ucbuilder.path("/customer/{id}").buildAndExpand(customer.getCid()).toUri());

return new ResponseEntity<String>(headers, HttpStatus.CREATED);

}

@DeleteMapping("/{id}")

public ResponseEntity<String> deleteCustomer(@PathVariable("id") long customerId)

{

customerService.deleteCustomer(customerId);

return new ResponseEntity<String>(HttpStatus.NO\_CONTENT);

}

Level-0

SOAP-Based Webservices

|\_ HTTP POST method + SOAP Action (CRUD)

|\_ /customer

REST-Based

Level-1

/customer GET

/customer/<id> GET

Level-2

/customer GET,POST,DELETE

/customer/<id> GET,PUT,DELETE

Level-3

Level-2 + HATEOAS (hyperlink as part of response)

Swagger helps users build, document, test and consume RESTful web services.

It can be used with both a top-down and bottom-up API development approach.

In the top-down, or design-first, method, Swagger can be used to design an API before any code is written.

springdoc-openapi java library helps to automate the generation of API documentation using spring boot projects. springdoc-openapi works by examining

an application at runtime to infer API semantics based on spring configurations, class structure and various annotations.

Automatically generates documentation in JSON/YAML and HTML format APIs. This documentation can be completed by comments using swagger-api annotations.

This library supports:

OpenAPI 3

Spring-boot (v1, v2 and v3)

JSR-303, specifically for @NotNull, @Min, @Max, and @Size.

Swagger-ui

OAuth 2

GraalVM native images

SOLID is an acronym that stands for five key design principles: single responsibility principle, open-closed principle, Liskov substitution principle, interface segregation principle, and dependency inversion principle. All five are commonly used by software engineers and provide some important benefits for developers

public class Order {

float amt;

public float getAmt() {

return amt;

}

public void setAmt(float amt) {

this.amt = amt;

}

}

PaymentService.java -------

@PostMapping("v3")

String getMsgV3(@RequestBody Order order)

{

if(order.getAmt() > 50000)

return "UPI Payment";

else

return "Cash Payment";

}

OrderService.java -----------

@PostMapping("v3")

String getPaymentV3(@RequestBody Order order)

{

String msg = restTemplate.postForObject

("http://PaymentService/v3",order,String.class);

return msg;

}

OrderService.java -> using HttpEntity

@PostMapping("v4")

String getPaymentV4(@RequestBody Order order)

{

HttpHeaders httpHeaders = new HttpHeaders();

httpHeaders.setContentType(MediaType.APPLICATION\_JSON);

httpHeaders.setAccept(Arrays.asList(MediaType.APPLICATION\_JSON));

HttpEntity<Order> entity = new HttpEntity<Order>(order,httpHeaders);

return restTemplate.exchange("http://PaymentService/v3", HttpMethod.POST, entity, String.class).getBody();

}

resilience4j.circuitbreaker:

configs:

default:

slidingWindowType: COUNT\_BASED

slidingWindowSize: 100

permittedNumberOfCallsInHalfOpenState: 10

waitDurationInOpenState: 10

failureRateThreshold: 60

registerHealthIndicator: true

String msg = restTemplate.postForObject("http://PaymentService/v3",order,String.class);

return msg;

server:

port: 8081

spring:

application:

name: OrderService

profiles:

active:

- development

cloud:

config:

uri:

- http://localhost:8888

eureka:

client:

register-with-eureka: true

fetch-registry: true

service-url:

defaultZone: http://localhost:8761/eureka

resilience4j.circuitbreaker:

configs:

default:

slidingWindowType: COUNT\_BASED

slidingWindowSize: 100

permittedNumberOfCallsInHalfOpenState: 10

waitDurationInOpenState: 10

failureRateThreshold: 60

registerHealthIndicator: true

management:

endpoints:

web:

exposure:

include:

- "\*"

endpoint:

shutdown:

enabled: true

https://drive.google.com/drive/folders/1gxGGsCLKSuBCtxtNr1dlAMYRFaMYWPvP?usp=share\_link

FROM openjdk:8-jdk-alpine

VOLUME /tmp

EXPOSE 8080

ADD target/dockerdemo-0.0.1-SNAPSHOT.jar dockerdemo.jar

ENTRYPOINT [ "sh", "-c", "java -jar /dockerdemo.jar" ]