**What is Design  Pattern ?**

A design pattern is a **well-proved solution** for solving the specific problem/task.

**Types of design patterns:**

**Creational:** Singleton, Factory, Abstract Factory, Builder, prototype

**Structural:** Adaptor, Composite, Proxy, Fly Weight, Façade, Bridge, Decorator

**Behavioral:** Template Method, Mediator, Chain of Responsibility, Observer, Strategy, Command, State, Visitor, Iterator, interpreter, memento

**Singleton:**

**class** Singleton {

**private** **static** Singleton single\_instance = **null**;

**public** String s;

**private** Singleton() {

        s = "Hello I am a string part of Singleton class";

    }

**public** **static** Singleton getInstance() {

**if** (single\_instance == **null**)

            single\_instance = **new** Singleton();

**return** single\_instance;

    }

}

**When do we use singleton design pattern?**

* Data Base Connection pool manager

**Why not to use Singleton design pattern**

* Singleton is anti-pattern
* Hard to handle with unit test cases
* Hard to mock and stub
* Concurrency problems mutability

**When can we use single ton is another places?**

* Spring singleton scope, dependency injection

**Factory**

**In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.**

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Inside Rectangle::draw() method.");

}

}

public interface Shape {

void draw();

}

public class ShapeFactory {

//use getShape method to get object of type shape

public Shape getShape(String shapeType){

if(shapeType == null){

return null;

}

if(shapeType.equalsIgnoreCase("CIRCLE")){

return new Circle();

} else if(shapeType.equalsIgnoreCase("RECTANGLE")){

return new Rectangle();

}

return null;

}

}

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Inside Circle::draw() method.");

}

}

public class FactoryPatternDemo {

public static void main(String[] args) {

ShapeFactory shapeFactory = new ShapeFactory();

Shape shape1 = shapeFactory.getShape("CIRCLE");

shape1.draw();

Shape shape2 = shapeFactory.getShape("RECTANGLE");

shape2.draw();

}

}

**Abstract Factory:**

public class ShapeFactory extends AbstractFactory {

@Override

public Shape getShape(String shapeType){

if(shapeType.equalsIgnoreCase("RECTANGLE")){

return new Rectangle();

}else if(shapeType.equalsIgnoreCase("SQUARE")){

return new Square();

}

return null;

}

}

**Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes.**

public class ShapeFactory extends AbstractFactory {

@Override

public Shape getShape(String shapeType){

if(shapeType.equalsIgnoreCase("RECTANGLE")){

return new Rectangle();

}else if(shapeType.equalsIgnoreCase("SQUARE")){

return new Square();

}

return null;

}

}

public class RoundedShapeFactory extends AbstractFactory {

@Override

public Shape getShape(String shapeType){

if(shapeType.equalsIgnoreCase("RECTANGLE")){

return new RoundedRectangle();

}else if(shapeType.equalsIgnoreCase("SQUARE")){

return new RoundedSquare();

}

return null;

}

}

public abstract class AbstractFactory {

abstract Shape getShape(String shapeType) ;

}

**Structural**

**Façade:  
Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system.**

public class FacadePatternDemo {

public static void main(String[] args) {

ShapeMaker shapeMaker = new ShapeMaker();

shapeMaker.drawCircle();

shapeMaker.drawRectangle();

shapeMaker.drawSquare();

}

}

public class ShapeMaker {

private Shape circle;

private Shape rectangle;

private Shape square;

public ShapeMaker() {

circle = new Circle();

rectangle = new Rectangle();

square = new Square();

}

public void drawCircle(){

circle.draw();

}

public void drawRectangle(){

rectangle.draw();

}

}

**Hystrix**

**stops cascading failure, if the failure happens Hystrix proved to close the particular circuit, its latency and fault tolerant library, improve your system’s overall resiliency.**

which is describing a strategy against failure cascading at different levels in an application.

**application.properties**

**client.url**=**http://localhost:8084/bankswitch/former/cases/hystrix**

@HystrixCommand(

  commandKey = **"ratingsByIdFromDB"**,

  fallbackMethod = **"findCachedRatingById"**,

  ignoreExceptions = { RatingNotFoundException.**class** })

**public** Rating findRatingById(Long ratingId) {

**return** Optional.ofNullable(ratingRepository.findOne(ratingId))

      .orElseThrow(() ->

**new** RatingNotFoundException(**"Rating not found. ID: "** + ratingId));

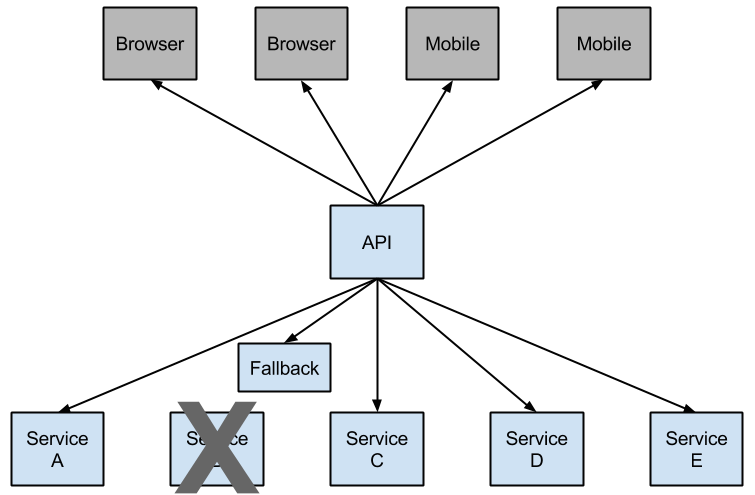
}

**public** Rating findCachedRatingById(Long ratingId) {

**return** cacheRepository.findCachedRatingById(ratingId);

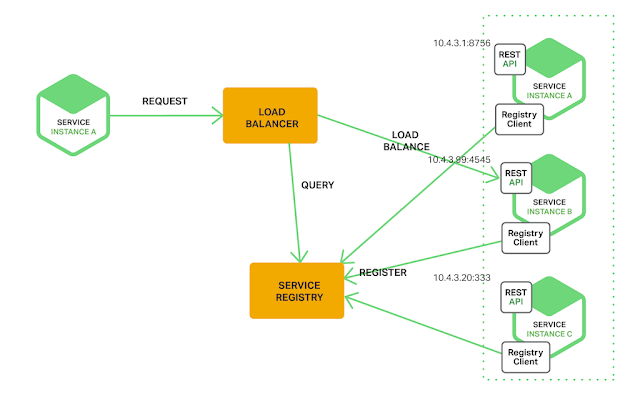
}

**import** org.springframework.boot.SpringApplication;  
**import** org.springframework.boot.autoconfigure.SpringBootApplication;  
**import** org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;  
@SpringBootApplication  
@EnableCircuitBreaker  
**public class** HystrixJava8Application {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(HystrixJava8Application.**class**, args);  
 }  
}

[](https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&ved=2ahUKEwii-ZyggrfjAhVFKuwKHUDnDesQjRx6BAgBEAU&url=https://cloud.spring.io/spring-cloud-netflix/multi/multi__circuit_breaker_hystrix_clients.html&psig=AOvVaw20ryhnZGcz86mYknIXjcRD&ust=1563283289937806)

**package** com.osp.hystrixjava8.presentation.resource;  
  
**import** com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;  
**import** org.springframework.beans.factory.annotation.Autowired;  
**import** org.springframework.beans.factory.annotation.Value;  
**import** org.springframework.web.bind.annotation.GetMapping;  
**import** org.springframework.web.bind.annotation.RestController;  
**import** org.springframework.web.client.RestTemplate;  
  
@RestController  
**public class** HelloController {  
 RestTemplate **restTemplate** = **new** RestTemplate();  
 @Value(**"${client.url}"**)  
 String **clientUrl**;  
 @GetMapping(**"/nohystrix"**)  
 **public** String getMappingExample(){  
 **return restTemplate**.getForObject(**clientUrl**,String.**class**);  
 }  
 **@GetMapping("/hystrix")  
 @HystrixCommand(groupKey = "fallback",commandKey="fallback", fallbackMethod = "hystrixFallBack")** **public** String getMapping(){  
 **return restTemplate**.getForObject(**clientUrl**,String.**class**);  
 }  
 **public** String hystrixFallBack(){  
 **return "Target Application is down"**;  
 }  
}

**Eureka**

[](http://1.bp.blogspot.com/-dXDcC3rnVeU/V7J79R5XxII/AAAAAAAADOQ/K53Ebgvj0uweGLXDeVKanQa3uFxxxRCvwCK4B/s1600/img.png)

**Functions:**

DROP FUNCTION IF EXISTS `TEST`.`get\_student\_name` $$

CREATE FUNCTION `get\_student\_name` (in\_id INTEGER)

RETURNS varchar(200)

BEGIN

DECLARE out\_name VARCHAR(200);

SELECT name

INTO out\_name

FROM Student where id = in\_id;

RETURN out\_name;

DELIMITER ;

private DataSource dataSource;

private JdbcTemplate jdbcTemplateObject;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

this.jdbcTemplateObject = new JdbcTemplate(dataSource);

}

public Student getStudent(Integer id) {

SimpleJdbcCall jdbcCall = new

SimpleJdbcCall(dataSource).withFunctionName("get\_student\_name");

SqlParameterSource in = new MapSqlParameterSource().addValue("in\_id", id);

String name = jdbcCall.executeFunction(String.class, in);

Student student = new Student();

student.setId(id);

student.setName(name);

return student;

}

**Procidure:**

public class StudentJDBCTemplate implements StudentDAO {

private DataSource dataSource;

private JdbcTemplate jdbcTemplateObject;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

this.jdbcTemplateObject = new JdbcTemplate(dataSource);

}

public Student getStudent(Integer id) {

SimpleJdbcCall jdbcCall = new

SimpleJdbcCall(dataSource).withProcedureName("getRecord");

SqlParameterSource in = new MapSqlParameterSource().addValue("in\_id", id);

Map<String, Object> out = jdbcCall.execute(in);

Student student = new Student();

student.setId(id);

student.setName((String) out.get("out\_name"));

student.setAge((Integer) out.get("out\_age"));

return student;

}

}

List<Customer> customers = Arrays.asList(c1, c2, c3, c4);

customerRepository.saveAll(customers);

**return** ResponseEntity.created(**"/customers"**);

DROP PROCEDURE IF EXISTS `TEST`.`getRecord` $$

CREATE PROCEDURE `TEST`.`getRecord` (

IN in\_id INTEGER,

OUT out\_name VARCHAR(20),

OUT out\_age INTEGER)

BEGIN

SELECT name, age

INTO out\_name, out\_age

FROM Student where id = in\_id;

END $$

DELIMITER ;

**How to insert multiple records in DB one go?**

Batch Processing

customerRepository.saveAll(customers);

**Fetching from one table based on falg n and then and insert into new tables active/inactive**

We can create procedure

First we get the all the records from the flag yes and we can insert using batch processing

**If any exception happens how to handle it , how to know where it failed and how the data is taken care?**

**3 JVM’s read and write on file how to do it? Approach??**

If it's two separate applications/processes altogether, the underlying filesystem should lock the files. When you get an I/O error from your output stream, you should be able to wrap a try/catch block around that, and then set your app up to retry later, or whatever the desired behavior is for your particular application.

**@Component @Repository @Controllers**

**Use of @Annotation**

**During boot startup main is not able to find some beans, beans creation failed how to resolve it ? where will you create the beans**

**Parallel steam it’s used in functioning v/s multithreading , which one is good**

**If queue capacity is 100 && 500 records are there how to do it ?**

**if appis down , messages are lying on queue, how to handle it ?how to make sure messages are not lost on queue?**

**what if we want to increase the listers, how to make sure only 1 listener writes that message**

**Array Vs array List**

**Abstract class v/s Interface**

**Deep & shallow cloning**

**Collections**

**Multithreading ,future, callback**

**Java8**

**Inner class**

**Spring boot, Spring mvc**

**Micro services**

**Integration tests &Unit Testing**