# Spring

Open Source, Light Weight, Loosely Coupled, AOP and Dependent management-based application framework to develop various types of applications.

Evolution with time

Spring is evolving according to trends… (Boot -> Microservices -> Cloud)

## Spring Versions

|  |  |  |
| --- | --- | --- |
| Version | Date | Notes |
| 0.9 | 2003 |  |
| 1.0 | March 24, 2004 | First production release. |
| 2.0 | 2006 | Second Production release |
| 3.0 | 2009 | Third Production release |
| 4.0 | 2013 | Fourth Production release |
| 5.0 | 2017 | Fifth Production release |
| 6.0 | November 16, 2022 | Sixth Production release |

Spring framework latest version is **6** released in 16 Nov,2022

Spring Boot latest version is **3** released in 16 Nov,2022

Spring 6 and Boot 3 versions have support of Java 17 and Jakarta EE 9 too.

Jakarta package instead of javax package

It is certain that Spring Boot 3 is based on Spring Framework 6 and that Spring Boot 3 will completely open the way to cloud-native and removed all deprecated features.

Minimum Java 8 version to be installed to work with Spring 5 or higher versions

Spring 5 is base line version on Java 8. It uses many features of Java 8 and 9 as well

## Open source

Source code available and Free software

Community develops software to make software community better

## How Open-Source software’s earn money?

Training

Certification

Certification exam trainings

## Light weight

Software light weight

Containers are light weight

Can write programs on simple java beans

## Loosely Coupled

Degree of dependency among modules is very less

No dependency on other modules

Interface model-based development

Implement the logic in implemented classes

## AOP

It is all about developing secondary logics/middleware service

Enabling/Disabling secondary logics to main logic

Primary and secondary logics are combined at runtime

## Dependency Lookup

Main component becomes dependent on another component

Vehicle needs Engine

It spends time to search and gather the other component

Ex: Getting Data source from JNDI context

Advantage:

It searches and gather only required component

Disadvantage:

Main component must have additional logic to search and gather additional logic

## Dependency Injection

Dependency Injection is the concrete example of IOC.

It is the Design pattern implemented IOC

Containers manages the Beans initialization and injection

Underlying server/container/framework creates all the beans (Main and dependent components), inject them at runtime.

Inversion Of Control is also called Dependency Management

### Advantage:

Main component no need to spend time for searching and gathering the dependent components.

Rather focusing on objects and its dependencies, focus on actual business logic

# Spring Bean –

An object that is managed by Spring Container/Framework

## Spring Bean scopes:

Singleton – default – single & same instance for all requests

Prototype - new instance for every request

Request - HTTP request

Session – HTTP Session

Global session – Global HTTP Session

Scope declaration in springconfig.xml

<!-- A bean definition with singleton scope -->

<bean id = "..." class = "..." scope = "singleton">

<!-- collaborators and configuration for this bean go here -->

</bean>

Scope declaration with Annotation approach

### Method level

@Bean

@Scope("singleton")

**public** Person **personSingleton**() { **return** **new** **Person**(); }

### Class level

@Component

//@Scope("singleton")

//@Scope(scopeName = "prototype")

@Scope(value = ConfigurableBeanFactory.***SCOPE\_PROTOTYPE***)

**public** @Data **class** Engine { }

# @Bean vs @Component

|  |  |
| --- | --- |
| @Bean | @Component |
| Method level | Class level |
| Instance to be explicitly created and returned | Spring container instantiates instant and make it available in class path |
| It works only when class is declared with @Configuration | It works without @Configuration annotation |
| Should implement if we want specific implementation based on dynamic condition | Can’t write specific implementation based on dynamic condition |
| Bean can be created even if the class is outside of the Spring Container | Bean can’t be created even if the class is outside of the Spring |

# IOC Container:

A software that manages life cycle of spring beans and its dependent beans/objects.

Two types of Containers in Spring:

|  |  |
| --- | --- |
| BeanFactory | ApplicationContext |
| Interface is the initial container | Built on top of BeanFactory, Having additional features like ability to support of AOP Integration, resolve textual message from properties file, publish application events, etc.. |
| Deprecated in Spring 3.0 | Recommended and being used. |

@ComponentScan – refers the package to lookup

@Component – Marks a class to be managed by Spring Container

@Primary - Indicates that a bean should be given preference when multiple candidates are qualified to autowire a single-valued dependency.

# Spring Modules

## Core Module:

Fundamentals of Core

Provides two containers for Spring Beans Management

Dependency Management

## JDBC

Abstraction layer on JDBC

Avoids Boiler plate code by providing library

## ORM

Implementation of ORM concept

ORM framework

SQL independent - HQL makes SQL Independent

Simplifies persistence logic

## Web

Provides plugins to make spring apps integrate with other MVC frameworks (Struts, etc)

Provides MVC based apps development

## JEE

Provides abstraction layer on multiple JSE, JJEE technologies and simplifies Application Development process

Provides abstraction layer on JDBC, JMS, JTA, RMI, EJB, JNDI and etc..

## AOP

Not replacement for OOP

Compliments to OOP

It is all about developing secondary logics/middleware service

Enabling/Disabling secondary logics.

Primary and secondary logics are combined at runtime

Ex: Security, Logging, Transaction Management etc.

AOP technologies:

AspectJ/Spring AOP

## SECURITY

Simplifies authentication and authorization logic

Secure web applications /rest/microservices

## BOOT

Makes Spring app development and Microservices easier

Provides required configuration and runtime environment

## Cloud

Provides features to develop cloud ready applications in distributed system

To build cloud native applications

Load balancing, Configuration management, Health check, Logs Managements, Fault and tolerance and Resilience management.

# Spring Examples

Vehicle – Interface

Car – Subclass of Vehicle

Bike – Subclass of Vehicle

### Vehicle

**public** **interface** Vehicle {

**void** drive();

}

### Car

**public** **class** Car **implements** Vehicle {

@Override

**public** **void** drive() {

System.***out***.println("Car is running...");

}

}

### Bike

**public** **class** Bike **implements** Vehicle {

@Override

**public** **void** drive() {

System.***out***.println("Bike is running..");

}

}

### Spring.xml – Configuration File

Put this file under src/main/java

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<bean id="vehicle" class="com.manga.spring.ex1.Car" />

<!--

<bean id="vehicle" class="com.manga.spring.ex1.Bike" />

-->

</beans>

### Test

ApplicationContext appContext = **new** ClassPathXmlApplicationContext("spring.xml");

Vehicle vehicle = (Vehicle) appContext.getBean("vehicle");

vehicle.drive();

**@Component** annotation class marks class is available in class path and to be managed by spring container

* Remove bean entries in spring.xml file (configuration file)
* Add @Component to the Car class

### Car

@Component

**public** **class** Car **implements** Vehicle {

@Override

**public** **void** drive() {

System.***out***.println("Car is running...");

}

}

Note: context:component-scan to be declared in xml in case a class is defined outside of main package or its sub packages.

### Spring.xml

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:component-scan base-package="com.manga.spring.ex1" />

</beans>

### Test

Run the Test class

ApplicationContext appContext = **new** ClassPathXmlApplicationContext("spring.xml");

Vehicle vehicle = (Vehicle) appContext.getBean("car");

vehicle.drive();

### Bike

@Component(value = "myBike")

**public** **class** Bike **implements** Vehicle {

@Override

**public** **void** drive() {

System.***out***.println("Bike is running..");

}

}

### Test

ApplicationContext appContext = **new** ClassPathXmlApplicationContext("spring.xml");

//Vehicle vehicle = (Vehicle) appContext.getBean("bike"); //bike bean unavailable

Vehicle vehicle = (Vehicle) appContext.getBean("myBike");

vehicle.drive();

## Setter & Constructor Injection of Property through Bean configuration in XML

### Tyre Class:

**public** **class** Engine {

**private** String brand;

**public** Engine() {

}

**public** Engine(String brand) {

**this**.brand = brand;

}

**public** String getBrand() {

**return** brand;

}

**public** **void** setBrand(String brand) {

**this**.brand = brand;

}

@Override

**public** String toString() {

**return** "Engine [brand=" + brand + "]";

}

}

### spring.xml

<bean id="engine" class="com.manga.spring.ex1.Engine">

<constructor-arg value="BENZ"></constructor-arg>

<property name="brand" value="HERO"></property>

</bean>

### Test

ApplicationContext appContext = **new** ClassPathXmlApplicationContext("spring.xml");

Engine engine = (Engine) appContext.getBean("engine");

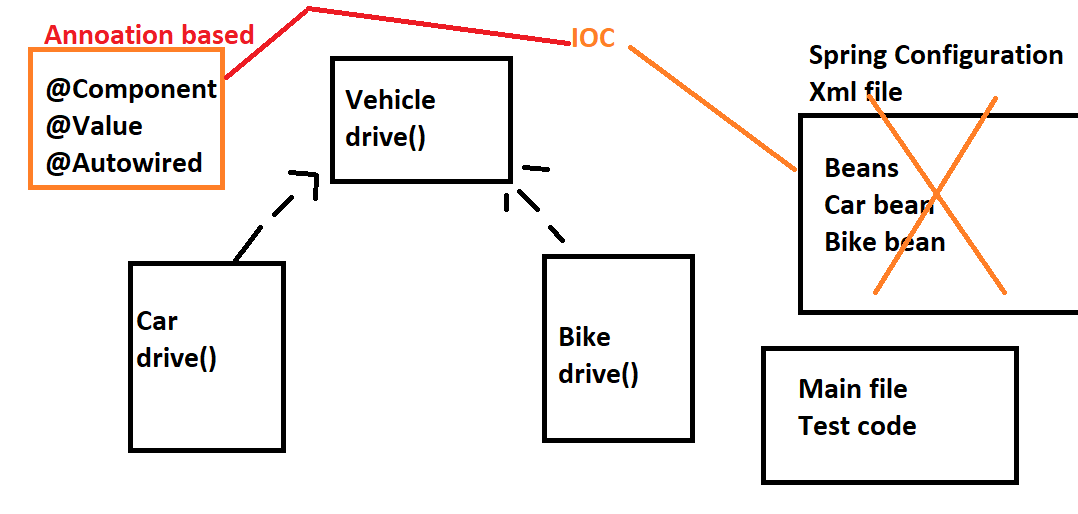
System.***out***.println(engine);

Note: When Both Constructor and Setter Injections are in place, Setter Injection values are reflected.

## Java Based configuration – Annotation approach Auto Component Scanning

### Configuration Bean/Class

Everything is annotation Based. No XML Based



### Cat class

**public** **class** Cat {

**public** **void** sound() {

System.***out***.println("Cat sounds like meow meow.. ");

}

}

### Configuration File

### @Configuration

### Indicates that a class declares one or more [@Bean](eclipse-javadoc:%E2%98%82=SpringEx1/C:%5C/Users%5C/marepalli%5C/.m2%5C/repository%5C/org%5C/springframework%5C/spring-context%5C/5.3.23%5C/spring-context-5.3.23.jar=/maven.pomderived=/true=/=/maven.groupId=/org.springframework=/=/maven.artifactId=/spring-context=/=/maven.version=/5.3.23=/=/maven.scope=/compile=/=/maven.pomderived=/true=/%3Corg.springframework.context.annotation(Configuration.class%E2%98%83Configuration%E2%98%82Bean) methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime

### @Bean

Indicates that a method produces a bean to be managed by the Spring container.

### AppConfig

@Configuration

**public** **class** AppConfig {

@Bean

**public** Cat getCat() {

**return** **new** Cat();

}

}

### Test

ApplicationContext appContext = **new** AnnotationConfigApplicationContext(AppConfig.**class**);

Cat cat = (Cat) appContext.getBean(Cat.**class**);

cat.sound();

## Annotation Component & Component Scan- No bean Configuration

Add @Component to Cat class

### Cat

@Component

**public** **class** Cat {

**public** **void** sound() {

System.***out***.println("Cat sounds like meow meow.. ");

}

}

AppConfig

No bean declaration, add package of the class to ComponentScan annotation under AppConfig

@Configuration

@ComponentScan("com.manga.spring.ex1")

**public** **class** AppConfig {

/\*

\* @Bean public Cat getCat() { return new Cat(); }

\*/

}

### Test

ApplicationContext appContext = **new** AnnotationConfigApplicationContext(AppConfig.**class**);

Cat cat = (Cat) appContext.getBean(Cat.**class**);

cat.sound();

## @ComponentScan Example

What to do when the component class is defined under a outside of base package.

Move GameRunner class to different package and run the class

Output: : No qualifying bean of type 'com.deloitte.manga.GameRunner' available

Package path should be mentioned

@ComponentScan({"com.deloitte.app.spring","com.deloitte.manga"})

Note: include all the packages of component class including base package

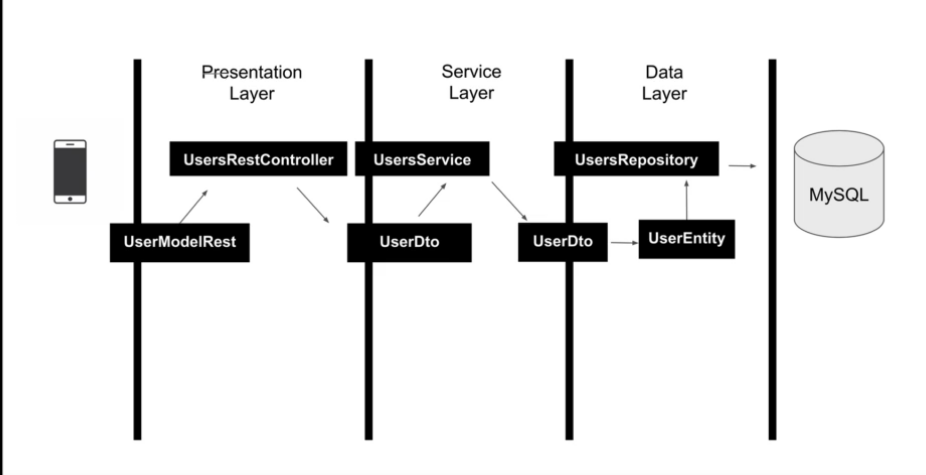
# Enterprise Application Flow Example with Dependency Injection

UI <-> Controller <-> Service <-> DAO <-> DB

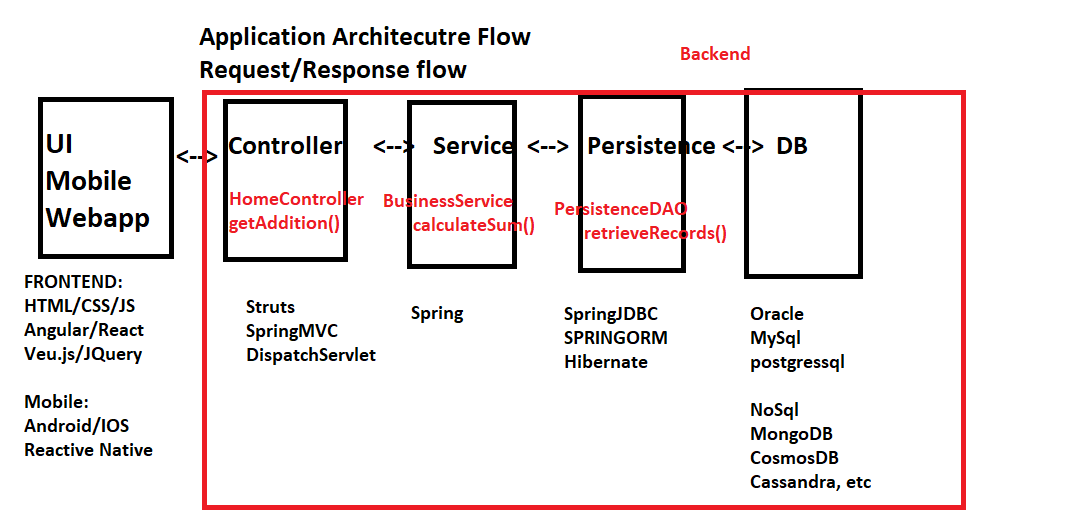
Presentation Layer – > Controller Layer -> Service Layer – Data (Access) Layer

ModelRest – Between UI and Controller

DTO - DataTransferObject – Between Controller and Service

Entity – Between Data Layer and DB

## Application Architecture Flow and Technologies by Layer



Here Ignore UI and DB

Add Dependencies: Spring WEB and DEV tools

Classes:

HomeController

BusinessService

PersistenceDAO

@RestController

**public** **class** HomeController {

@Autowired

**private** BusinessService businessService;

@GetMapping("/sum")

**public** **long** getAddition() {

**return** businessService.calculateSum();

}

}

@Component

//@Service

**class** BusinessService {

@Autowired

**private** PersistenceDAO persistenceDAO;

**public** **long** calculateSum() {

List<Integer> records = persistenceDAO.retrieveRecords();

**return** records.stream().reduce((x, y) -> x + y).get();

// return records.stream().reduce(Integer::sum).get();

}

}

@Component

//@Repository

**class** PersistenceDAO {

**public** List<Integer> retrieveRecords() {

**return** Arrays.*asList*(10, 20, 300);

}

}

## Test:

localhost:<portno.>/sum

# Project Lombok

## Why Project Lombok

* Reduce boilerplate code like getters, setters, to String methods, and constructors, etc.,
* . Languages like Kotlin & Scala which also JVM base but don’t have this boiler plate code.

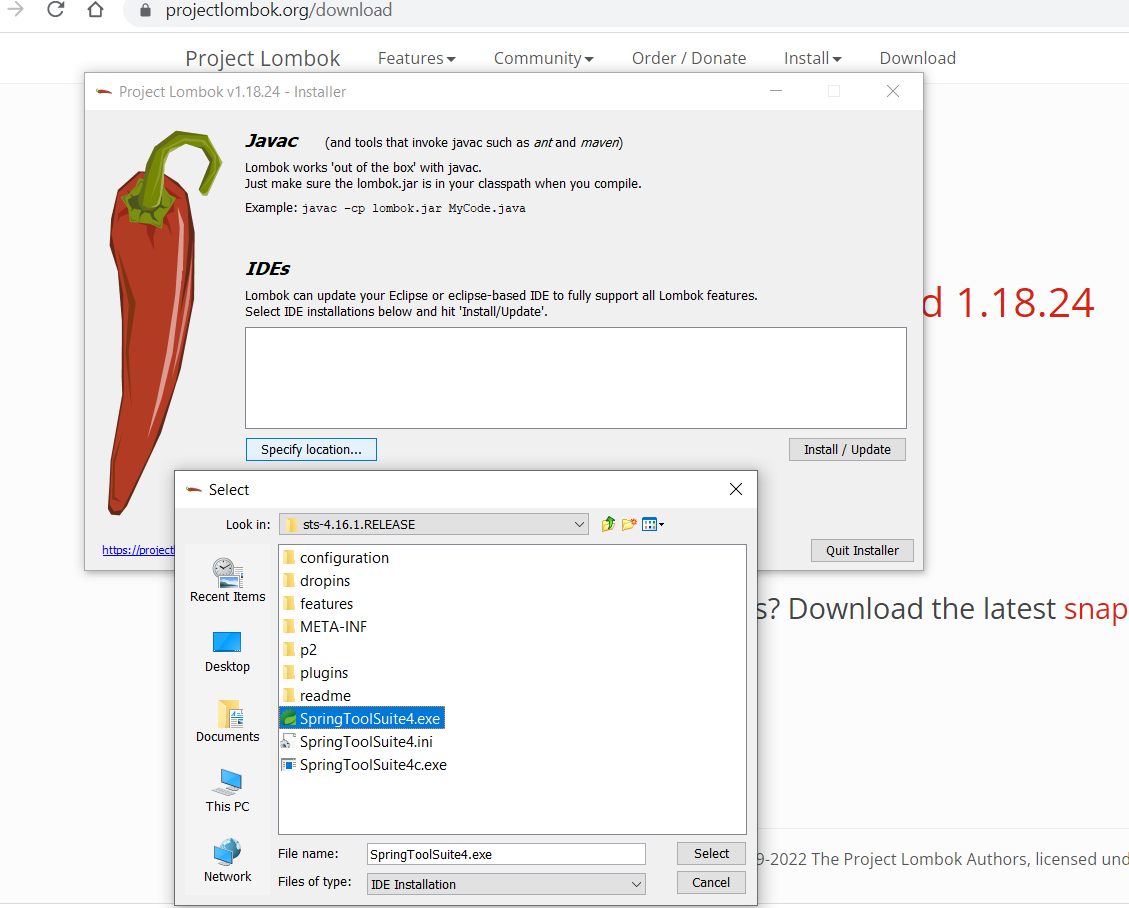
## What is Project Lombok?

* Java Library tool to minimize/remove the boilerplate code and improve productivity.
* Achieve by adding annotation
* It also increases the readability and saves space. boiler plate code is unavailable in source code.

## Lombok Installation in Eclipse (STS)

* Download Lombok.jar file from <https://projectlombok.org/download> or from Maven repository
* Run the executable jar file
* then give IDE exe file path as below
* click on Install/Update
* Restart the IDE

Note: Observe lombak.jar to be placed in IDE exe file location and configuration file (Ex: SpringToolSuite4.inf) to be updated with Javaagent as Lombak.



## Dependency

<dependencies>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.24</version>

<scope>provided</scospe>

</dependency>

</dependencies>

Or

Add dependency from spring starter project.

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

## Annotations

Refer Lombok features - <https://projectlombok.org/features/>

@Setter(AccessLevel.***PRIVATE***)

@Getter(AccessLevel.***PROTECTED***)

@ToString(includeFieldNames = **false**)

@AllArgsConstructor

@NoArgsConstructor

@EqualsAndHashCode

@Log4j2

@RequiredArgsConstructor - Forms constructor with variables that are marked as not null

@Data- Equivalent to @Getter @Setter @RequiredArgsConstructor @ToString @EqualsAndHashCode.

@Cleanup

***val*** – it acts as var in Javascript. Hassel free final local variable.

## Lombok Examples

### Ex1 - Employee class

@Getter

@Setter

@ToString

@EqualsAndHashCode

@AllArgsConstructor

@RequiredArgsConstructor

//@Data

**public** **class** Employee {

**private** int id;

**private** String firstName;

**private** String lastName;

}

Spring Main class

@SpringBootApplication

**public** **class** LombokExApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(LombokExApplication.**class**, args);

Employee e = **new** Employee(101, "manga", "arepalli");

System.***out***.println(e);

}

}

### Ex2 - Employee class

@Data

@AllArgsConstructor

//@RequiredArgsConstructor - Forms constructor with variables that are marked as not null

@NoArgsConstructor

@Setter(AccessLevel.***PRIVATE***)

@Getter(AccessLevel.***PROTECTED***)

@EqualsAndHashCode

@ToString(includeFieldNames = **false**)

@Log4j2

**public** **class** Employee {

@ToString.Exclude

@Setter(value = AccessLevel.***NONE***)

@Nonnull

**private** **int** id;

@Getter

@ToString.Include

**private** String name;

**private** **int** age;

**public** **void** drive() {

***log***.debug("Debug statement");

}

**public** **void** main(String[] args) **throws** IOException {

@Cleanup

BufferedReader br=**new** BufferedReader(**null**);

drive();

**val** s=List.*of*(10,20);

**for** (**val** i : s) {

System.***out***.println(i);

}

}

}

# Logging support in Lombok



# Spring Data JDBC

One of the Spring modules

## Spring JdbcTemplate

is a powerful mechanism to connect to the database and execute SQL queries.

It internally uses JDBC API, but eliminates a lot of problems of JDBC API.

## Problems with JDBC

* Need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
* Need to perform exception handling on database logic
* Need to handle transactions
* Repetition of all this code is a time consuming task

## Advantages

Spring JdbcTemplate eliminates boiler plate code of JDBC API.

Provides methods to write the queries directly, so it saves a lot of work and time

## JDBCTemplate class

It is central class in JDBC support classes.

Takes care of creation and release of resources such as connection, resultset, etc.

Handles the exception and provides users friendly message by the help of exception classes defined in org.springframework.dao

## Methods of Spring JDBCTemplate class

|  |  |
| --- | --- |
| Method | Description |
| public int update(String query) | To insert, update and delete records. |
| public int update(String query,Object... args) | To insert, update and delete records using PreparedStatement using given arguments. |
| public void execute(String query) | to execute DDL query. |
| public T execute(String sql, PreparedStatementCallback action) | executes the query by using PreparedStatement callback. |
| public T query(String sql, ResultSetExtractor rse) | is used to fetch records using ResultSetExtractor. |
| public List query(String sql, RowMapper rse) | is used to fetch records using RowMapper. |

## Hikari

Is a JDBC data source implementation that provides a default connection pooling mechanism.

Compared to others, it is light weight and performing better

This dependency is automatically included in **spring-boot-starter-data-jpa** and **spring-boot-starter-data-jdbc.**

### Hikari configuration parameters in application.properties

spring.datasource.hikari.connectionTimeout=30000

spring.datasource.hikari.idleTimeout=600000

spring.datasource.hikari.maxLifetime=1800000

# Spring Boot + SPRING JDBC + MySQL

## Example1 – JDBCTemplate

### Maven Dependencies

<dependency>

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

<artifactId>spring-boot-starter-data-jdbc</artifactId>

</dependency>

<dependency>

<groupId>com.mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

<scope>runtime</scope>

</dependency>

### Application.properties

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

spring.datasource.username=root

spring.datasource.password=root

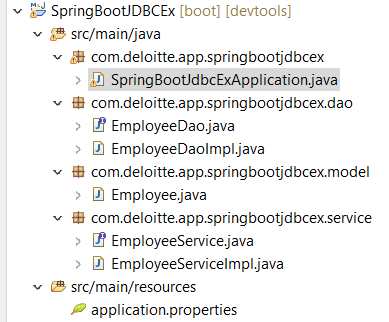
### DB script

create table emp (id int, name varchar(50), age int);

insert into emp values (101, 'Manga', 20);

select \* from emp;

### Directory structure



### Backend Code

#### Model:

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** **int** age;

//setters, getters and constructor

}

#### DAO Layer

**public** **interface** EmployeeDao {

**public** **abstract** **int** insertEmployee(Employee emp);

**public** **abstract** List<Employee> getEmployees();

**public** **abstract** Employee getEmployeeById(**int** empId);

**public** **abstract** **int** updateEmployee(Employee e);

**public** **abstract** **int** deleteEmployee(**int** empId);}

@Repository

**public** **class** EmployeeDaoImpl **implements** EmployeeDao {

@Autowired

**public** JdbcTemplate jdbcTemplate;

@Override

**public** **int** insertEmployee(Employee emp) {

String query = "insert into employee values(?,?,?)";

**return** jdbcTemplate.update(query, **new** Object[] { emp.getId(), emp.getName(), emp.getAge() });

}

@Override

**public** List<Employee> getEmployees() {

**return** jdbcTemplate.query("select \* from employee",

BeanPropertyRowMapper.*newInstance*(Employee.**class**));

}

@Override

**public** Employee getEmployeeById(**int** empId) {

**return** jdbcTemplate.queryForObject("select \* from employee where id=?",

BeanPropertyRowMapper.*newInstance*(Employee.**class**),empId );

}

@Override

**public** **int** updateEmployee(Employee emp) {

**return** jdbcTemplate.update("update employee set name=?, age=? where id=?",**new** Object[] { emp.getName(), emp.getAge(),emp.getId() });

}

@Override

**public** **int** deleteEmployee(**int** empId) {

**return** jdbcTemplate.update("delete from employee where id =?", empId);

}

}

#### Service Layer

**public** **interface** EmployeeService {

**public** **abstract** **int** insertEmployee(Employee emp);

**public** **abstract** List<Employee> getEmployees();

**public** **abstract** Employee getEmployeeById(**int** empId);

**public** **abstract** **int** updateEmployee(Employee e);

**public** **abstract** **int** deleteEmployee(**int** empId);

}

@Service

**public** **class** EmployessServiceImpl **implements** EmployeeService {

@Autowired

**private** EmployeeDao employeeDao;

@Override

**public** **int** insertEmployee(Employee emp) {

**return** employeeDao.insertEmployee(emp);

}

@Override

**public** List<Employee> getEmployees() {

**return** employeeDao.getEmployees();

}

@Override

**public** Employee getEmployeeById(**int** empId) {

**return** employeeDao.getEmployeeById(empId);

}

@Override

**public** **int** updateEmployee(Employee e) {

**return** employeeDao.updateEmployee(e);

}

@Override

**public** **int** deleteEmployee(**int** empId) {

// **TODO** Auto-generated method stub

**return** employeeDao.deleteEmployee(empId);

}

}

### Test Class

ApplicationContext appCtx = SpringApplication.*run*(SpringBootJdbcExApplication.**class**, args);

EmployeeService empService = appCtx.getBean(EmployeeService.**class**);

//empService.insertEmployee(new Employee(106,"aarohi",2));

//List<Employee> allEmployees = empService.getAllEmployees();

//allEmployees.forEach(System.out::println);

Employee emp = empService.getEmployeeById(101);

System.***out***.println(emp);

## Example 2 – NamedParameterJDBCTemplate

### NamedParameterJDBCTemplate

is wrapper of JDBCTemplate class

Spring provides another way to insert data by named parameter.

In such way that named parameters are used instead of? as place holder.

Improves readability of the code

Easy to remember the data for the column.

### Change is only DAO Implementation

@Autowired

**public** NamedParameterJdbcTemplate namedJdbcTemplate;

@Override

**public** **int** insertEmployee(Employee emp) {

String query = "insert into employee values(:id,:name,:age)";

//1. using Map

Map<String, Object> params = **new** HashMap<>();

params.put("id", emp.getId());

params.put("name", emp.getName());

params.put("age",emp.getAge());

//2. MapSqlParameterSource class to hold the parameter names and values in more convenient wa

MapSqlParameterSource mapSqlparams = **new** MapSqlParameterSource();

mapSqlparams.addValue("id", emp.getId());

mapSqlparams.addValue("name", emp.getName());

mapSqlparams.addValue("age",emp.getAge());

//3. have domain model class, you can use the BeanPropertySqlParameterSource class that can inspect the properties of a given domain model class to generate parameter names and fill values accordingly

BeanPropertySqlParameterSource paramSource = **new** BeanPropertySqlParameterSource(emp);

**return** namedJdbcTemplate.update(query, params);

//return namedJdbcTemplate.update(query, mapSqlparams);

//return namedJdbcTemplate.update(query, paramSource);

}

@Override

**public** List<Employee> getEmployees() {

**return** namedJdbcTemplate.query("select \* from employee",

BeanPropertyRowMapper.*newInstance*(Employee.**class**));

}

@Override

**public** Employee getEmployeeById(**int** empId) {

String sql="select \* from employee where id=:id";

SqlParameterSource param = **new** MapSqlParameterSource("id", empId);

**return** namedJdbcTemplate.queryForObject(sql,param, BeanPropertyRowMapper.*newInstance*(Employee.**class**) );

}

@Override

**public** **int** updateEmployee(Employee emp) {

String sql = "update employee set name=:name, age=:age where id=:id";

MapSqlParameterSource param = **new** MapSqlParameterSource();

param.addValue("name", emp.getName());

param.addValue("id", emp.getId());

param.addValue("age", emp.getAge());

**return** namedJdbcTemplate.update(sql,param);

}

@Override

**public** **int** deleteEmployee(**int** empId) {

String query="delete from employee where id=:id";

SqlParameterSource param = **new** MapSqlParameterSource("id", empId);

**return** namedJdbcTemplate.update(query, param);

}

# H2 Database

Temporary Database and In Memory DB

Java SQL Light weight Database

When server restarts, H2 Database becomes empty

Testing purpose, it can be used.

Ref: <https://github.com/h2database/h2database>

By default, H2 Spring Boot configures the application to connect to an in-memory store with the username sa and an empty password.

the in-memory database is volatile, and results in data loss after application restart.

## Advantages

Zero configuration

It is easy to use.

It is lightweight and fast.

It provides simple Configuration to switch between a real database and in-memory database.

It supports standard SQL and JDBC API.

It provides a web console to maintain in the database

## File-based storage for non-volatile database configuration

spring.datasource.url=jdbc:h2:file:/data/demo

## Data Source Initialization

Spring Boot will automatically pick up the sql script from data.sql or schema.sql files and run them against an embedded in-memory database

## Enable H2 DB

Add maven dependency

spring.h2.console.enabled=true in application.properties

**Application.properties**

spring.h2.console.enabled=true

spring.datasource.username=sa

spring.datasource.password=sa

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

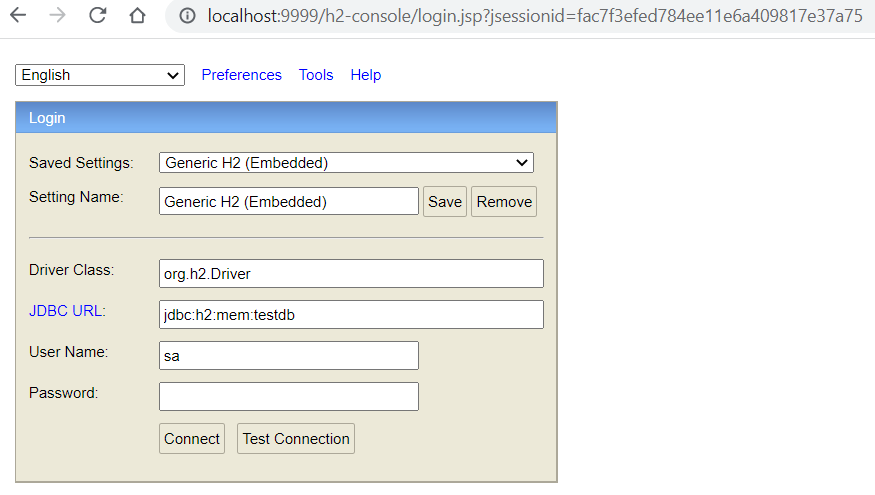
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

#set console path to access it from url

#spring.h2.console.path=/h2-ui

## Accessing the H2 database

localhost:portno/h2-console



# Spring Boot + JDBC + H2 Database

### Maven Dependencies

<dependency>

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

<artifactId>spring-boot-starter-data-jdbc</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

Let’s use same code for H2 Database. Only change is in application.properties

### Application.properties

spring.h2.console.enabled=true

spring.datasource.url=jdbc:h2:mem:testdb

#spring.datasource.url=jdbc:h2:file:C:/h2db/testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=sa

Note: data.sql/schema.sql file is not considered with file storage option

### data.sql

Place this file under resources folder which is same location of application.properties

create table emp (id int, name varchar(50), age int);

insert into emp values (101, 'Manga', 20);

select \* from emp;

commit;

### Test Class

Run the same code and see the output

# Simple JDBC Template

Java-5-based convenience wrapper for the classic Spring JdbcTemplate, taking advantage of **varargs** and autoboxing, and exposing only the most required operations in order to simplify JdbcTemplate usage.

As of Spring 3.1 SimpleJdbcTemplate has been deprecated, and most of the features of SimpleJdbcTemplate have been integrated into JdbcTemplate, except named parameter which is provided by NamedParameterJdbcTemplate.

The JdbcTemplate and NamedParameterJdbcTemplate now provide all the functionality of the SimpleJdbcTemplate.

# JPA

The java persistence API provides a SUN specification for persisting, reading, and managing data from your java object to your relational tables in the database.

JPA specifies the set of rules and guidelines for developing interfaces that follow standards.

Straight to the point: JPA is just guidelines to implement ORM and there is no underlying code for the implementation.

JPA specification implementation is done by:

* Hibernate
* iBatis
* Toplink
* OpenJPA etc.,

# ORM

Object Relation Mapping is simply the process of persisting any java object directly into a database table.

Usually, the name of the object being persisted becomes the name of the table

Each field within that object becomes a column

Hibernate is one examples of ORM.

In short, JPA is the interface while hibernate is the implementation.

# Spring Data JPA

Part of Spring

The goal of spring data repository abstraction is to significantly reduce the amount of boilerplate code required to implement a data access layer for various persistence stores.

Ability to create repository implementations automatically, at runtime, from a repository interface.

is not a JPA provider, it is a library/framework that adds an extra layer of abstraction on the top of our JPA provider line Hibernate

## Advantages of Spring Data JPA template

Avoids writing boiler plate code

It dynamically generates queries from queries methods name.

## @Repository

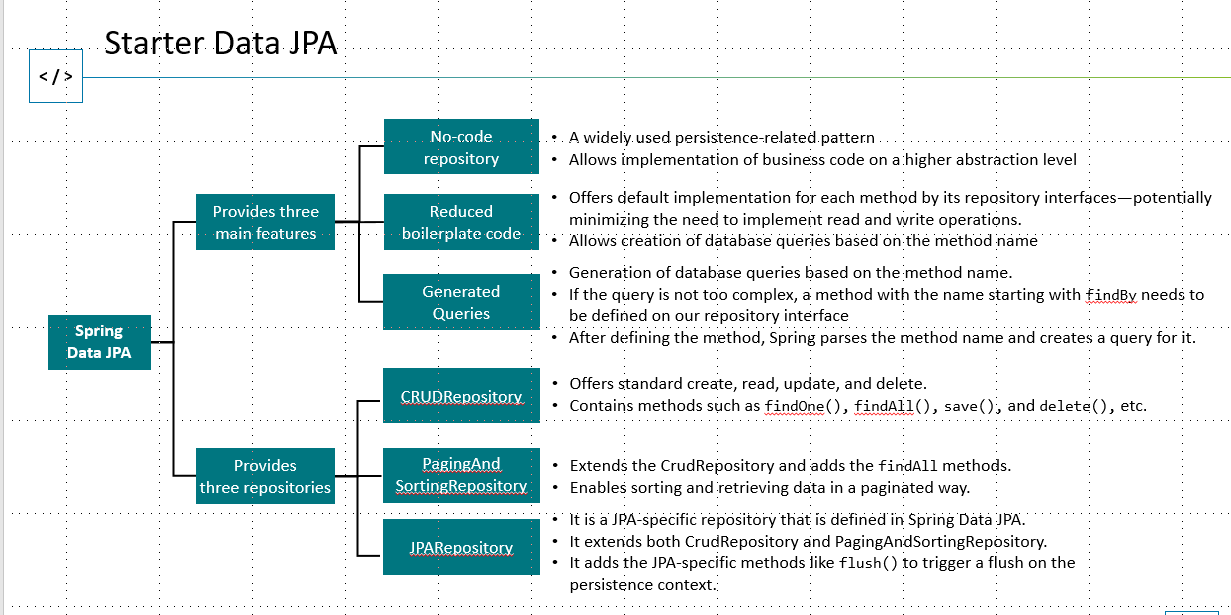
The @Repository annotation is a marker for any class that fulfills the role or stereotype of a repository (also known as Data Access Object or DAO).

## @JpaRepository

JpaRepository<Employee, Long> is a JPA-specific extension of the Repository. It contains the full API of CrudRepository and PagingAndSortingRepository.

It contains API for basic CRUD operations and API for pagination and sorting.

# Spring Boot + Spring JPA + Hibernate + MySQL



## Maven Dependencies

spring-boot-starter-data-jpa

mysql-connector-java

## Application.properties

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

spring.datasource.username=root

spring.datasource.password=root

spring.jpa.database-platform = org.hibernate.dialect.MySQL5Dialect

spring.jpa.generate-ddl=true

spring.jpa.hibernate.ddl-auto = update

spring.jpa.show-sql=true

## Entity or Modal

Apparently, Java persistence api is renamed to Jakarta persistence. So you will have to import Jakarta persistence instead of javax.persistence

@Entity

@Table(name = "employee")

**public** **class** Employee {

@GeneratedValue(strategy = GenerationType.***AUTO*** )

@Id

@Column(name="id")

**private** **int** id;

@Column(name="name")

**private** String name;

@Column(name="age")

**private** **int** age;

**//Default constructor**

**public** Employee(String name, **int** age) {

**super**();

**this**.name = name;

**this**.age = age;

}

**public** Employee(**int** id, String name, **int** age) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.age = age;

}

//setters & getters

@Override

**public** String toString() {

// **TODO** Auto-generated method stub

**return** **this**.id + " " + **this**.name + " " + **this**.age;

}

}

## Repository

@Repository

**public** **interface** EmployeeDao **extends** JpaRepository<Employee, Integer> {

//Custom methods

List<Employee> findByAge(**int** age);

List<Employee> findByName(String name);

@Query(value = "select \* from employee where age<=:age", nativeQuery = **true**)

List<Employee> getRecordsByAge(**int** age);

@Query("select e from Employee e where e.age<=:age") //HQL //Database independent

List<Employee> getRecordsByAge(**int** age, Sort sort);}

## Test class

ApplicationContext appCtx = SpringApplication.*run*(SpringbootjpaApplication.**class**, args);

EmployeeDao empDao = appCtx.getBean(EmployeeDao.**class**);

// inserting a record

Employee emp = empDao.save(**new** Employee("Manga", 30));

System.***out***.println("Inserted record");

System.***out***.println(emp);

// deleting the record by Id

empDao.deleteById(emp.getId());

System.***out***.println("record with "+emp.getId()+" id deleted");

// deletes all records

// empDao.deleteAll();

// fetching all records

System.***out***.println("All records");

empDao.findAll().forEach(System.***out***::println);

//fetching record by name - custom method

System.***out***.println("Records with find by name");

empDao.findByName("Manga").forEach(System.***out***::println);

empDao.findByAge(20).forEach(System.***out***::println);

empDao.getRecordsByAge(50, Sort.*by*(Sort.Direction.***ASC***, "age")).forEach(System.***out***::println);

Note: save() can be used for both for insert and update methods

# Spring Boot

## Definition

Spring Boot is a project that is built on the top of the Spring Framework.

It provides an easier and faster way to set up, configure, and run both simple and web-based applications.

It makes spring development easier

It is a Spring module that provides the RAD (Rapid Application Development) feature to the Spring Framework

Use Spring STS IDE or Spring Initializer to develop Spring Boot Java applications.

## Dependency Management

* Spring Boot manages dependencies, its versions and configuration automatically.
* Each release of Spring Boot provides a list of dependencies that it supports.
* Spring Boot upgrades all dependencies automatically in a consistent way when we update the Spring Boot version

## Spring Boot Starters

Spring Boot offers numerous starters that allow us to add JARs in the classpath.

All the starters follow a naming pattern: spring-boot-starter-\* where \* denotes a particular type of application. There are many starters available. Here are just 5 of them to get a glimpse.

J

|  |  |  |
| --- | --- | --- |
| Starter | Transitive dependency | Dependency |
| spring-boot-starter | spring-boot, spring-boot-autoconfigure,  spring-boot-starter-logging, spring-core, snakeyaml | It is used for Core starter, autoconfiguration support, logging, and YAML |
| spring-boot-starter-data-jpa | Spring-boot-starter, spring-boot-starter-aop, spring-boot-starter-jdbc, hibernate-entitymanager, javax.transaction-api, spring-data-jpa, spring-aspects | It is a starter for using spring data JPA with Hibernate |
| spring-boot-starter-test | Junit, mockito-core, hamcrest-core, hamcrest-library, spring-core, spring-test | It is a starter for testing using libraries Junit, Hamcrest, and Mockito |
| spring-boot-starter-web | Spring-boot-starter, spring-boot-starter-tomcat, spring-boot-starter-validation, jackson-databind, spring-web, spring-webmvc | It is a Starter for building Web application using Spring MVC, REST, and Tomcat as a default embedded container |
| spring-boot-starter-data-mongodb | Spring-boot-starter, mongo-java-driver,  spring-data-mongodb | It is a starter for using MongoDB and Spring Data MongoDB |

## STS installation

https://spring.io/tools or spring.io -> project -> spring tools

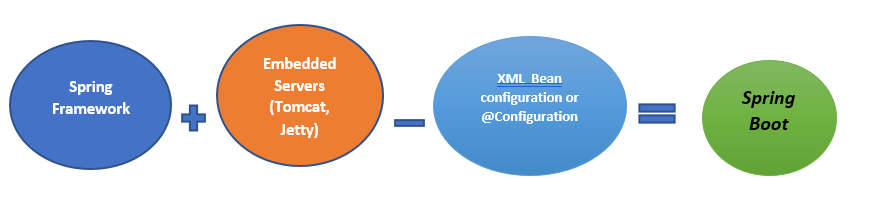
download spring tools for eclipse

Jar is downloaded -> Double click on Jar -> STS is excluded

Spring Boot application Creation

* Through STS -> File -> STS project -> fill details -> save
* STS initializer -> Fill details -> download zip, extract it and open it in editor

Refer [GitHub - spring-projects/spring-boot: Spring Boot](https://github.com/spring-projects/spring-boot) for more details



## Limitations of Spring Boot

Spring Boot can use dependencies that are not going to be used in the application which increase the size of the application.

## Advantage to the Developer

Reduces **Development time, Developer Efforts,** and **increases productivity**.

## Prerequisite of Spring Boot

Java 8

Spring Framework

Maven

IDE - STS/Eclipse IDE

## @SpringBootApplication

Enables spring boot auto configuration and component scanning

This is a convenience annotation that is equivalent to declaring

@Configuration, @EnableAutoConfiguration and @ComponentScan.

## Application Runner and Command Line Runner

Both are interfaces used to run the code after the Spring Boot application started.

We can use this to perform any action immediately after the application has been started.

### Application Runner Example

@SpringBootApplication

**public** **class** SpringBootApp **implements** ApplicationRunner{

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootApp.**class**, args);

System.***out***.println("Spring boot application has started");

}

@Override

**public** **void** run(ApplicationArguments args) **throws** Exception {

System.***out***.println("Hello from Application runner");

}

}

### Command Line Runner Example

@SpringBootApplication

**public** **class** SpringBootApp **implements** CommandLineRunner{

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootApp.**class**, args);

System.***out***.println("Spring boot application has started");

}

@Override

**public** **void** run(String... args) **throws** Exception {

System.***out***.println("Hello from Command Line runner");

}

}

## Application Properties

Spring Boot configuration is configured in application properties or yaml file.

Application properties support us to work in different environments like DEV, TEST, PROD

n no. of properties can be mentioned in a single file.

Configuration files to be placed in class path under java/main/resources folder

### Properties File

Key-value based configuration file

#### application.properties

server.port=7777

spring.application.name=springbootex

Refer Common Application Properties on click of [https://docs.spring.io/spring-boot/docs/1.4.x/reference/html/commo n-application-properties.html](https://docs.spring.io/spring-boot/docs/1.4.x/reference/html/commo%20%20n-application-properties.html)

### YAML File

Hierarchical configuration file

It doesn’t contain repeated prefixes.

Spring boot supports it

It is super set of JSON file

It is alternative of properties file

#### application.yml

spring:

application:

name: springboot

datasource:

password: password

url: dburl

username: sa

## @Value annotation usage

The @Value annotation is used to read the environment or application property value in Java code.

Syntax: @Value("${property\_key\_name}")

### Usages:

1. Variable initialization
2. Get value from configuration file (application.properties)
3. Default value assignment in case no value against declared key
4. List value assignment from comma, separated values from config file

@SpringBootApplication

@RestController

**public** **class** SpringbootexApplication {

@Value("${spring.application.name}") //same convention works for yaml file too

**private** String name;

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringbootexApplication.**class**, args);

}

@RequestMapping("/")

**public** String name() {

**return** name;

}

}

**Note** – In case, any property is not found while running the application, Spring Boot throws the Bean creation exception. To avoid this set the default value for the property in Java code

Syntax:

@Value("${property\_key\_name:default\_value}")

Ex: 2 with default value

@Value("${spring.application.name1: welcome to spring boot}")

Ex:3 Initializing list with comma separated values from properties file

@Value("${user.skills}")

//comma separated value is assigned as list

**private** List<String> skills;

@GetMapping("/greet")

**private** String greeting() {

**return** "skills: "+ skills;

}

Application.properties

user.skills=Java,HTML, CSS, Javascript, Spring, SpringBoot, Microservices, Spring Cloud

Test Endpoint:

http://localhost:portno./greet

## Spring Boot Active Profiles

Spring Boot supports different properties based on the Spring active profile.

we can keep separate files for development, QA and production to run the Spring Boot application.

### application.properties

server.port=7777

spring.application.name=springbootex

### application-dev.properties

server.port=7777

spring.application.name=springbootex from dev

### application-qa.properties

server.port=7777

spring.application.name=springbootex from qa

### application-prod.properties

server.port=7777

spring.application.name=springbootex from prod

### Property to set active profile in main configuration file

spring.profiles.active=dev

# Microservices

It is architectural development style which is evolution of SOA. An application is made up smaller services that handle smaller portion functionality and data by communicating each other with light weight protocol like HTTP

## Microservice:

is a web service, performs specific functionality individually.

## Principles of Microservice:

* Single responsibility

Each micro service has a single responsibility to perform single functionality.

* Build Around Business Capability

A microservice shall never restrict itself from adopting an appropriate technology stack or backend database storage that is most suitable for solving the business purpose

* Design for failure

Microservices must be designed with failure cases in mind. Microservices must exploit the advantage of this architecture and **going down one microservice should not affect the whole system**, other functionalities must remain accessible to the user

|  |  |  |
| --- | --- | --- |
| Monolithic | SOA | Microservice |
| All the functionality of project exists in single codebase | Evolved to deal with problems with Monolithic application.  **Large application split into no. of smaller services, but these services can’t communicate each other directly.** | **An application is made up smaller services that handle smaller portion functionality and data by communicating each other with light weight protocol like HTTP** |
| Unreliable, a single bug in any module brings whole application down | There used to be an enterprise server bus or middleware server for the intercommunication. | Microservices are the smaller services that communicate each other and work together |
| Need to deploy whole application even for smaller change | **No guide to have independent database** | Each microservice has its own database |
| As Size of the application increases, deployment & startup time increases |  | Microservices are self-contained and, hence, deployed independently. |
| Very difficult to adopt new technology |  |  |

## Evolution till 2022

|  |  |
| --- | --- |
| 2005 | 2022 |
| WAR files | JAR files |
| App Servers | Microservices |
| Hot Deploy | Continuous Deployed |
| Servers for deployment | Heroku/AWS/Azure/GCP/for Deployment |

## 12 Factors App for Microservices.

Website: <https://12factor.net/>

12 Factors app is a methodology to for building Web Apps or Software as A Service (SAS)

The 12-factory methodology can be applied to any apps written in any programming language and which use any combination of backing services like Database, Queue, Memory, Cache etc.,

## Who must Apply 12 Factor Methodology?

A developer who builds apps which run as a service or Ops Engineer who deploys or manages applications.



|  |  |  |  |
| --- | --- | --- | --- |
| Sr.No | Factor Name | Description | Comment |
| 1 | Codebase | One codebase tracked in revision control; many deploys | Use Code version tool like Git, SVN, etc., |
| 2 | Dependencies | Explicitly declare and isolate dependencies | Dependencies should be explicitly managing  Use Maven or Gradle for dependency management |
| 3 | Config | Store config in the environment | Configuration strictly separated from code. Don’t check-in passwords. Configuration belongs to environment not to app.  Spring Cloud Config that provides configuration as a service.  Ex: application.yml |
| 4 | Backing Services | Treat backing services as attached resources | Backing service should be attachable/dettachable  Like Database configuration in configuration  Spring.datasource.url=mysql/oracle |
| 5 | Build, release, run | Strictly separate build and run stages | Build: mvn clean install  Release: mvn heroku:deploy or  Docker push…  Run: java -jar executablejar.jar/.war  Spring boot app is the best example |
| 6 | Process | Execute the app as one or more stateless processes | Process should be stateless. when something is terminated, it shouldn’t hold any critical data |
| 7 | Port Binding | Export services via port binding | Port binding should be run by application not by external infrastructure  Modern application self-contained, follows its own binding |
| 8 | Concurrency | Scale out via the process model | Able to scale up and scale out by diversifying workloads.  Java.util.Concurrent, Streams |
| 9 | Disposability | Maximize robustness with fast startup and graceful shutdown | Quick startup – Ideal start time is 1 min  Resilience Tolerance  Graceful shutdown – release all resource before shutdown.  Application servers are not disposable where Microservices are disposable.  Microservices are easy to replace, easy to modify, easy to decouple from external Infrastructure |
| 10 | Dev/Prod parity | Keep development, staging, and production as similar as possible | |  |  |  | | --- | --- | --- | | Dev | Stage | Prod | | Tomcat | Tomcat | Jboss (wrong) | | Tomcat | Tomcat | Tomcat | | Jetty | Jetty | Jetty |   Parity means reproducibility |
| 11 | Logs | Treat Logs as streams | Logs are for monitoring and troubleshooting the issue  Use Log4j/Slfj, etc.. |
| 12 | Admin processes | Run admin/management tasks as one-off processes | Admin tasks should be run as isolated processes. Admin task should have separate process/container |

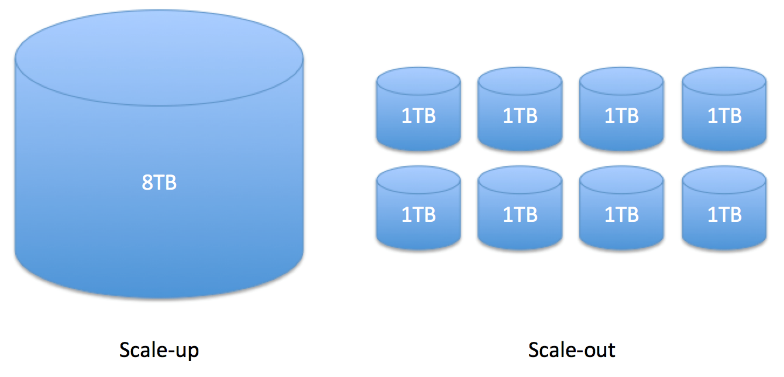
# Scale Up vs Scale Out

## Scale Up

* Vertical scaling is called Scale Up.
* Adding further resources to the existing system.
* Ex: Increase the RAM size from 1 core to 4 cores

## Scale Out

* Horizontal scaling is called Scale out.
* Adding discrete units to the system to add capacity.
* Ex: Having a Load Balancer where your app is hosted on multiple instances.



# Building Restful webservices/Microservices

Spring Boot provides a very good support to building RESTful Web Services for enterprise applications.

Spring Boot Starter Web dependency to be added in pom.xml to build restful webservices.

## Restful Webservices

REST stands for Representational State Transfer. REST is an architectural approach, not a protocol.

We can build REST services with both XML and JSON.

JSON is the more popular format with REST.

### Major HTTP Methods:

GET: It reads a resource.

PUT: It updates an existing resource.

POST: It creates a new resource.

DELETE: It deletes the resource.

## @Restcontrol

* To define rest service
* It marks class as rest service
* Annotation is used at class level
* It serves TEXT/JSON/XML/custom request and response
* no need to use view-resolvers or send HTML in response as Spring MVC

@Restcontrol = @Control + Response Body

## @RequestMapping

* To define request URI to access the REST endpoint
* The default request method is GET
* Used at method level

## @RequestBody

* To define request body **content type**
* Used at parameter level in method

public ResponseEntity<Object> createProduct(@RequestBody Product product) {}

## @PathVariable

* To define the custom or dynamic request URI (Universal Resource Identifier)
* Path variable in request URI is defined in curly braces {}

public ResponseEntity<Object> updateProduct(@PathVariable("id") String id) {

}

## @RequestParam

* To read request parameter from Request URI
* By default, it is required parameter
* Default value can be set

public ResponseEntity<Object> getProduct(

@RequestParam(value = "name", required = false, defaultValue = "honey") String name) {

}

## GET API

* To be used to retrieve the data
* Default HTTP Method request
* No request body required
* Request Param or path variable can be used to define the custom/dynamic URI

## POST API

* To be used to create the data server end
* Request body required
* Request Param or path variable can be used to define the custom/dynamic URI

## PUT API

* To be used to update the existing resource data server end
* Request body required
* Request Param or path variable can be used to define the custom/dynamic URI

## DELETE API

* To be used to delete the data from existing resource server end
* No request body required
* Request Param or path variable can be used to define the custom/dynamic URI

Like <https://jsonplaceholder.typicode.com/posts/1>

## Rest Controller creation using GET, POST, PUT, DELETE

To hit output

**Curl** stands for Client URL

curl <http://localhost:8181/>

### Rest Controller

**package** com.deloitte.app.ws.ui.controller;

@RestController

@RequestMapping("/users")

**public** **class** UserController {

@GetMapping

**public** String getUser() {

**return** "Get User endpoint is called";

}

@PostMapping

**public** String createUser() {

**return** "Create user endpoint is called";

}

@PutMapping

**public** String updateUser() {

**return** "Updated endpoint is called";

}

@DeleteMapping

**public** String deleteUsers() {

**return** "Delete user endpoint is called";

}

}

## Endpoints Testing:

See these endpoints output in POSTMAN by selecting HTTP methods.

<http://localhost:8181/users/>

# POSTMAN

Webservices Testing tool

Download Postman client or use POSTMAN web version by logging into it using Google account but local service can’t be tested from here

URL: go.postman.co/home

## Usage

Create workspace

Create collection

Create request with an endpoint & save

Endpoint: https://jsonplaceholder.typicode.com/posts/1

## Code option

Right side, find code option and select Curl/Javascript/java/C# code to call service from respective language.

## CURL

Stands for Client URL

Get the curl from Code option

### How to import curl in Postman

File -> Import -> Select Raw Text -> paste curl -> Continue

## Headers

Contenty-type - application/json or application/xml or etc.. – Content type client to server

Accept – application/json or any – Content type from server to client

## @PathVariable

@GetMapping(path = "/{userId}")

**public** String getUser(@PathVariable String userId) {

**return** "Get User endpoint is called with Path variable - "+userId;

}

Request URI - <http://localhost:8182/users/101>

## @RequestParam

Request Parameters Optional or Required

Query String:

<http://localhost:8181/users?page=1&&limit=10&&sort=asc>

Endpoint:

@GetMapping

**public** String getUsers(@RequestParam(value="page", defaultValue = "1") **int** page,

@RequestParam(value="limit", required = **true**) **int** limit,

@RequestParam(value="sort", defaultValue = "asc", required = **false**) String sort) {

**return** "Get Users endpoint is called with page = "+page+" limit "+limit+" sort "+sort;

}

## Returning Java Object as json

By default, JSON format is returned

http://localhost:8181/users/

@GetMapping

**public** User getUsers() {

User userResp = **new** User();

userResp.setUserId(101);

userResp.setFirstName("Manga");

userResp.setLastName("Arepalli");

userResp.setEmail("amr@gmail.com");

**return** userResp;

}

## Return List of Objects

@RequestMapping(method = RequestMethod.***GET***)

**public** List<User> getUser() {

User user1 = **new** User(101, "Manga", "Arepalli", "marepalli@dleoitte.com");

User user2 = **new** User(102, "Mihiran", "Arepalli", "mihi@dleoitte.com");

**return** List.*of*(user1, user2);

}

## Returning Java Object as xml

@GetMapping(produces = {MediaType.***APPLICATION\_XML\_VALUE***,

MediaType.***APPLICATION\_JSON\_VALUE***})

**public** User getUsers() {

User userResp = **new** User();

userResp.setUserId(101);

userResp.setFirstName("Manga");

userResp.setLastName("Arepalli");

userResp.setEmail("amr@gmail.com");

**return** userResp;

}

Set Accept: application/xml in Postman and try hitting end point,

In case it gives 500 error. Add Jackson XML library in pom.xml and try

<dependency>

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

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

<version>2.13.0</version>

</dependency>

## Setting Response Status code in Response Entity

@GetMapping(produces = {MediaType.***APPLICATION\_XML\_VALUE***,

MediaType.***APPLICATION\_JSON\_VALUE***})

**public** ResponseEntity<User> getUsers() {

User userResp = **new** User();

userResp.setUserId(101);

userResp.setFirstName("Manga");

userResp.setLastName("Arepalli");

userResp.setEmail("amr@gmail.com");

**return** **new** ResponseEntity<User>(userResp, HttpStatus.Ok);

// return new ResponseEntity<User>( HttpStatus.BAD\_REQUEST);

}

@RequestMapping(method = RequestMethod.***GET***, produces = {MediaType.***APPLICATION\_XML\_VALUE***,

## Returning User array in Response Entity:

MediaType.***APPLICATION\_JSON\_VALUE***})

**public** ResponseEntity<User[]> getUser() {

User user1 = **new** User(101, "Manga", "Arepalli", "marepalli@dleoitte.com");

User user2 = **new** User(102, "Mihiran", "Arepalli", "mihi@dleoitte.com");

**return** **new** ResponseEntity<User[]>(**new** User[] {user1,user2},HttpStatus.***BAD\_GATEWAY***);

}

## Reading HTTP POST Request Body @RequestBody annotation with Bean validations

### Dependency

<**dependency**>

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

<**artifactId**>spring-boot-starter-validation</**artifactId**>

</**dependency**

### Spring validation Annotations:

**@Notnull, @Max, @Min, @Size, @pattern**

@Valid annotation and Binding Result class through which we can get the errors raised by Validator implementation

Application**.**properties**:**

server.error.include-message=always

server.error.include-binding-errors=always

Model Request

**package** com.deloitte.app.ws.ui.model.request;

**import** javax.validation.constraints.Email;

**import** javax.validation.constraints.NotNull;

**import** javax.validation.constraints.Size;

**public** **class** UserDetailsRequestModel {

**private** **int** userId;

@NotNull(message = "First Name can't be blank")

**private** String firstName;

@Size(min = 6, max=10, message = "lastName should be >=6 & <=10")

**private** String lastName;

@Email

**private** String email;

//setters & getters

}

**Controller**

Import @Valid annotation in method parameter from javax.valdation

@PostMapping(consumes = {MediaType.***APPLICATION\_XML\_VALUE***, MediaType.***APPLICATION\_JSON\_VALUE***},

produces = {MediaType.***APPLICATION\_XML\_VALUE***, MediaType.***APPLICATION\_JSON\_VALUE***})

**public** ResponseEntity<User> createUser(@Valid @RequestBody UserDetailsRequestModel userReqModel) {

User user = **new** User();

user.setUserId(userReqModel.getUserId());

user.setFirstName(userReqModel.getFirstName());

user.setLastName(userReqModel.getLastName());

user.setEmail(userReqModel.getEmail());

**return** **new** ResponseEntity<User>(user, HttpStatus.***OK***);

}

## Rest Controller creation to store Users Temporally, handle HTTP GET, POST, UPDATE and DELETE

Map<Integer, User> usersMap;

@GetMapping(path = "/{userId}")

**public** ResponseEntity<User> getUser(@PathVariable Integer userId) {

**if**(usersMap!=**null** && usersMap.containsKey(userId)) {

**return** **new** ResponseEntity<User>(usersMap.get(userId), HttpStatus.***OK***);

}

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

}

@PostMapping(consumes = {MediaType.***APPLICATION\_XML\_VALUE***, MediaType.***APPLICATION\_JSON\_VALUE***},

produces = {MediaType.***APPLICATION\_XML\_VALUE***, MediaType.***APPLICATION\_JSON\_VALUE***})

**public** ResponseEntity<User> createUser(@Valid @RequestBody UserDetailsRequestModel userReqModel) {

**if**(usersMap==**null**) {

usersMap = **new** HashMap<Integer, User>();

}

User user = **new** User();

user.setUserId(userReqModel.getUserId());

user.setFirstName(userReqModel.getFirstName());

user.setLastName(userReqModel.getLastName());

user.setEmail(userReqModel.getEmail());

usersMap.put(userReqModel.getUserId(), user);

**return** **new** ResponseEntity<User>(user, HttpStatus.***OK***);

}

@PutMapping(path="/{userId}", consumes = {MediaType.***APPLICATION\_XML\_VALUE***, MediaType.***APPLICATION\_JSON\_VALUE***},

produces = {MediaType.***APPLICATION\_XML\_VALUE***, MediaType.***APPLICATION\_JSON\_VALUE***})

**public** ResponseEntity<User> updateUser(@Valid @RequestBody UserDetailsRequestModel userReqModel, @PathVariable Integer userId) {

**if**(usersMap!=**null** && usersMap.containsKey(userId)) {

User user = usersMap.get(userId);

user.setFirstName(userReqModel.getFirstName());

user.setLastName(userReqModel.getLastName());

user.setEmail(userReqModel.getEmail());

usersMap.put(user.getUserId(), user);

**return** **new** ResponseEntity<User>(user, HttpStatus.***OK***);

}

**return** ResponseEntity.*noContent*().build();

}

@DeleteMapping(path = "/{userId}")

**public** ResponseEntity<Void> deleteUsers(@PathVariable Integer userId) {

usersMap.remove(userId);

**return** ResponseEntity.*noContent*().build();

}

## Run webservice as a standalone application

Prerequisites:

Maven and Java must be installed in local computer and its path to be set

Follow below steps or From STS -> Run as Maven -> Maven Install

* Go to project root folder
* mvn install
* refer target folder
* Executable jar is generated
* Run it using
* Java -jar <jarfilename>

Server is up and running..

Check services in postman

Ctrl+c or close console to stop the server

# Spring Boot Actuator

Spring Boot Actuator is a sub project of Spring Boot.

It is available since April 2014, together with the first Spring Boot release.

Spring Boot’s ‘Actuator’ dependency is used to monitor and manage the Spring web application with HTTP Endpoints.

Actuator is used to simply monitor (check status, health, metrics, traffic) and manage the web service/application.

Starter: spring-boot-starter-actuator

## Advantages of Monitoring/Managing the Application

* Increases customer satisfaction.
* It reduces downtime.
* It boosts productivity.
* It improves Cybersecurity Management.

## Features:

/health

/beans- displays complete beans list in your microservice

/httptrace – displays http trace information

/threaddump – displays threads dumps in spring boot

## Actuator end points - Production ready features

<https://docs.spring.io/spring-boot/docs/2.1.7.RELEASE/reference/html/production-ready-endpoints.html>

## Maven Dependency

Add Spring Actuator to micro service project like Spring API Gateway

<dependency>

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

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

</dependency>

## Application.properties

server.port=8014

spring.application.name=config-client

spring.config.import=optional:configserver:http://localhost:8013

#actuator separate url

management.server.port=9999

#Exposing all endpoints

management.endpoints.web.exposure.include=\*

#management.endpoints.web.exposure.include=beans, loggers

#Enabling Specific endpoints

#management.endpoints.enabled-by-default=false

# let's enable and expose only the /health endpoint:

#management.endpoint.health.enabled=true

#management.endpoints.web.exposure.include=health

## End Points:

Incorrect:

Localhost:9999/health

Correct:

<http://localhost:9999/actuator>

<http://localhost:9999/actuator/heatlth>

<http://localhost:9999/actuator/metrics>

<http://localhost:9999/actuator/beans>

# REST API Documentation / Microservices Documentation

1. OpenAPI 3 (Swagger)
2. REST DOCs

## Open API3 – Documenting REST APIs

springdoc-openapi helps to automate the generation of API documentation using spring boot projects

Swagger was renamed to Open API.

### Dependency

To have [springdoc-openapi](https://github.com/springdoc/springdoc-openapi) automatically generate the OpenAPI 3 specification docs for our API, Add the [springdoc-openapi-ui](https://search.maven.org/search?q=g:org.springdoc%20AND%20a:springdoc-openapi-ui&core=gav) dependency to our *pom.xml*:

Refer: <https://springdoc.org/>

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-ui</artifactId>

<version>1.6.12</version>

</dependency>

Note: It may not work with springboot 3.0

Downgrade SpringBoot starter parent version to 2.7.6 in pom.xml

<parent>

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

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

<version>2.7.6</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

Start the server and access

Swagger UI page URL

<http://server:port/swagger-ui/index.html>

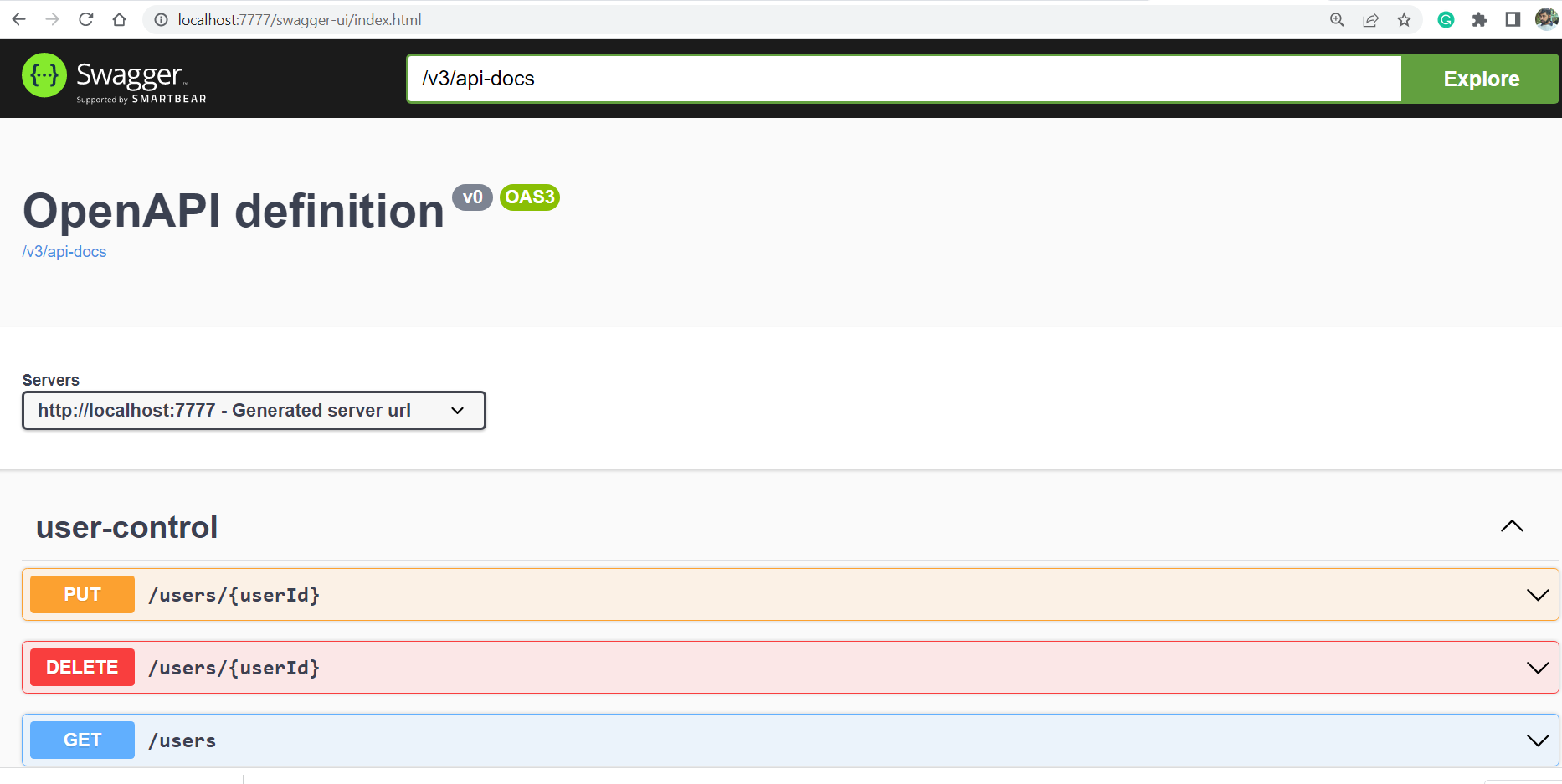
### API Docs

<http://server:port/v3/api-docs>

### YAML file URL

<http://server:port/v3/api-docs.yaml> - File is automatically downloaded.

Try it out by hitting web point from the ui



### SWAGGER UI Custom path

Application.properties

# swagger-ui custom path

springdoc.swagger-ui.path=/swagger-ui-custompath.html

### Generate documentation using Open API specification Annotations @Operation and @ApiResponses

Add following snippet to Get API by id and see the documentation in UI against the same endpoint.

@Operation(summary = "It's the Get API to get an User by its id")

@ApiResponses(value = {

@ApiResponse(responseCode = "200", description = "Found the user",

content = { @Content(mediaType = "application/json",

schema = @Schema(implementation = User.**class**)) }),

@ApiResponse(responseCode = "400", description = "Invalid id supplied",

content = @Content),

@ApiResponse(responseCode = "404", description = "User not found",

content = @Content) })

Drawback:

when Model class has circular dependency and swagger is integrated to spring boot, the application will crash

# Deploy Microservice to Heroku Cloud

Heroku is a Cloud platform provides PAAS ( Platform As A service) to host/deploy applications developed using various technologies.

Steps to Deploy microservice/app:

* Go to Heroku.com
* Signup
* Create a new app
* Mention app name
* Give the Github repository where your app/microservice has been uploaded
* Find the URL to access the app/microservices

# IAAS vs PAAS vs SAAS2SaaS vs PaaS vs IaaS: What's The Difference & How To Choose – BMC Software | Blogs

# Spring Cloud

Spring Cloud term may mislead you think like it provides services like IAAS, PAAS, and SAAS like other Cloud platforms GCP, Amazon Web services, Azure, Oracle Cloud, IBM cloud, etc., but it’s not a Cloud provider.

**Provides features to develop cloud native/ready applications in distributed system.**

The major use-case for Spring Cloud is the ready-to-use solution that it provides to common problems observed in distributed environments like load balancing, service discovery, circuit breaking, Distributed tracing, Microservices communication, etc., which can easily be integrated in an existing Spring project.

## Spring Cloud Features

* Service Registry and Discovery
* Microservices communication
* Load Balancing
* Central Configuration Management
* Distributed Messaging
* Distributed Tracing
* Fault Tolerance & Resilience

## Spring Cloud Starters examples under springframework.cloud

Spring-cloud-starter-config

Spring-cloud-starter-netflix-eureka-client

Spring-cloud-starter-netflix-eureka-server

## Eureka

Eureka is implemented by Netflix initially for their own purpose, later it became famous and is being used in spring and other.

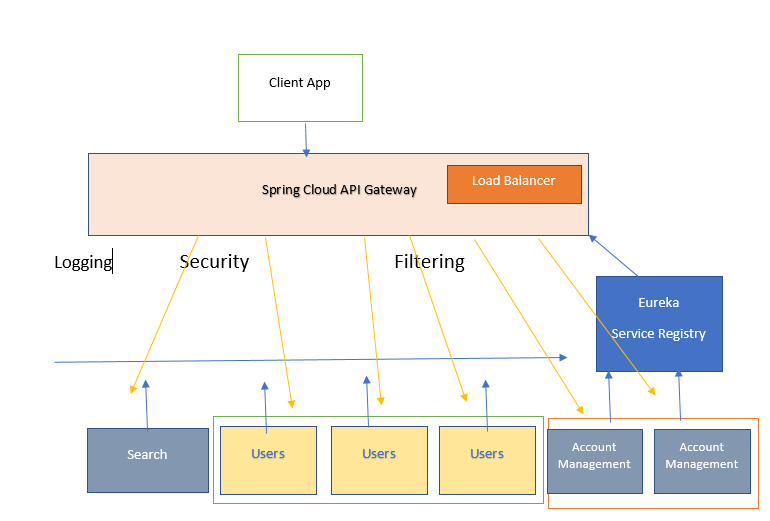
Eureka helps microservices finds each other.

It is an application holds client-server applications.

## API gateway

provides flexible way to routing requests based on a criterion.

# Eureka - A Discovery service – Service Registry Example



## Netflix OSS (Open-Source Software)

1. Eureka Server/Client
2. Zuul – Deprecated – Alternative: Spring Cloud API Gateway)
3. Ribbon – Deprecated - Client Side Load Balancer – Alternative: Spring Cloud Load Balancer
4. Hystrix – for Fault Tolerance & Resilience

**Imp points:**

Eureka, Zuul, Spring API Gateway, and other all are micro services.

Eureka server micro service to be started first.

Other services next, at last Zuul or Spring Cloud API Gateway

A micro service can be started multiple times with different port numbers which results multiple instance.

Multiple instances share multiple client requests.

Webflux is for Reactive apps development asynchronously

**Services to be created:**

1. Eureka Service
2. User Service
3. Account Management Service
4. Spring API Gateway Service

# Eureka Service creation

PhotoAppDiscoveryService

Dependencies: **Eureka Service**

**@EnableEurekaServer** to Main class

@SpringBootApplication

@EnableEurekaServer

**public** **class** PhotoAppDiscoveryServiceApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(PhotoAppDiscoveryServiceApplication.**class**, args);

}

}

Note: Mention 8761 port number for the Eureka server project because by default it uses known port no. i.e., 8761

## Application.properteis

server.port=8761

spring.application.name=discoveryservice

#Eureka server itself can be a client and can be registered for other server. So, prevent it as below

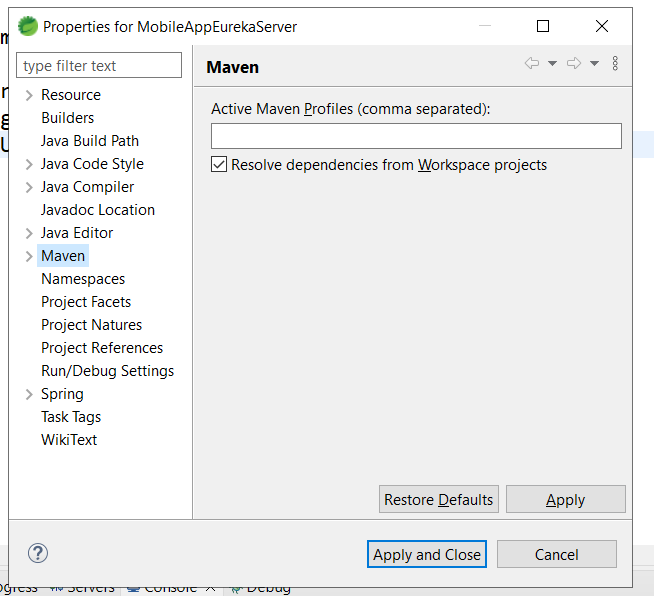
eureka.client.registerWithEureka=false

eureka.client.fetchRegistry=false

## Troubleshooting

### The requested profile "pom.xml" could not be activated because it does not exist.

* Right click on your project in Eclipse/STS
* Click Properties
* Select Maven in the left-hand side list.
* You will notice "pom.xml" in the Active Maven Profiles text box on the right-hand side.
* Clear it and click Apply.
* Right Click on project -> Run -> Maven Install
* Restart server



### Eureka Does Not Start

If Eureka does not start because of the following error: **Java.lang.NoClassDefFoundError: com/sun/jersey/client/apache4/ApacheHttpClient4**

You can either add the following dependency

1. <dependency>
2. <groupId>com.sun.jersey.contribs</groupId>
3. <artifactId>jersey-apache-client4</artifactId>
4. <version>1.19.4</version>
5. </dependency>

or remove eureka-client dependency

1. <dependency>
2. <groupId>org.springframework.cloud</groupId>
3. <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
4. </dependency>

Jaxb deprecated from Java9 replaced by Java XML. In case of any issue related to JAXB, add its dependencies.

# @EnableDiscoveryClient vs @EnableEurekaClient

**import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;

**import** org.springframework.cloud.netflix.eureka.EnableEurekaClient;

**@EnableDiscoveryClient** lives in spring-cloud-commons and picks the implementation on the class path.

**@EnableEurekaClient** lives in spring-cloud-netflix and only works for eureka.

If eureka is on your class path, they are effectively the same

# User Service creation

PhotoAppApiUser

Dependencies: Web, Eurekaclient, SpringBoot DevTools

@EnableEurekaClient

**Main class: Registered as client**

@SpringBootApplication

@EnableEurekaClient

**public** **class** PhotoAppApiUserApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(PhotoAppApiUserApplication.**class**, args);

}

}

**Application.properteis:**

spring.application.name=users-ws

server.port=${port:0}

#eureka.instance.instance-id=${spring.application.name}:${spring.application.instance\_id:${random.value}}

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

spring.devtools.restart.enabled=true

UserController:

@RestController

@RequestMapping("/users")

**public** **class** UserController {

@GetMapping(path = "/status/check")

**public** String status() {

**return** "Working from PhotoAppApi Service";

}

}

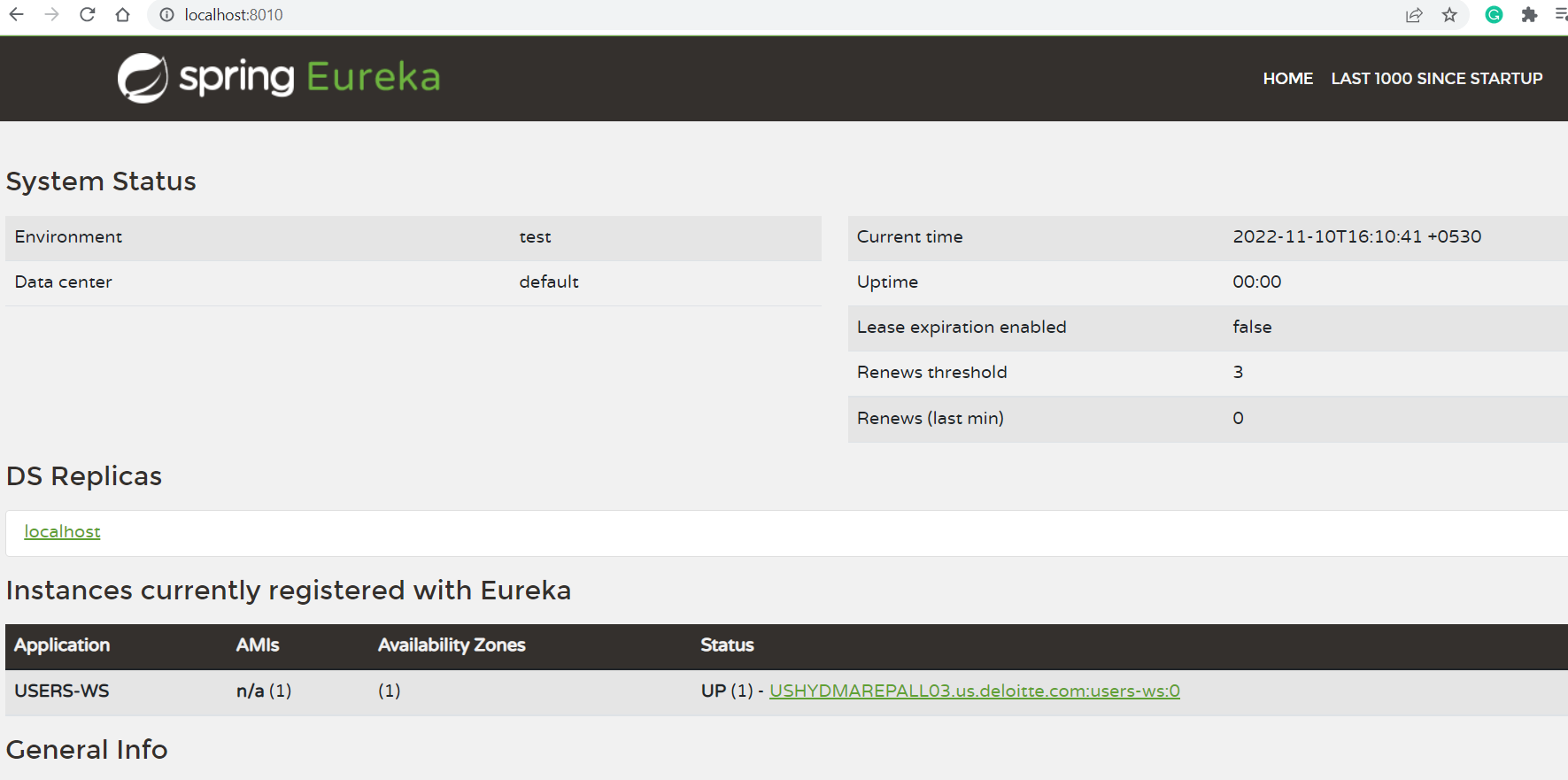
**Access User Service via Eureka Server**

1. Run Eureka Service
2. Run User Service

Open Eureka dashboard and See… User Service must have registered under Client

Click on URL under status and just append path of the API

<http://ushydmarepall03.us.deloitte.com:52460/users/status/check>



## Assign Dynamic Port no. from Command prompt

java -Dserver.port=8182 -jar <jar filename>

# Account Management Service creation

**Same as User Service creation**

**Main class:**

@SpringBootApplication

@EnableEurekaClient

**public** **class** PhotoAppApiAccMgmtApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(PhotoAppApiAccMgmtApplication.**class**, args);

}

}

**Controller:**

@RestController

@RequestMapping("/users/")

**public** **class** AccMgmtController {

@GetMapping(path = "/status/check")

**public** String status() {

**return** "Working from PhotoAppAccount Management Service, Congratulations!";

}

}

**Application.properites:**

spring.application.name=accms-ws

server.port=${port:0}

#eureka.instance.instance-id=${spring.application.name}:${spring.application.instance\_id:${random.value}}

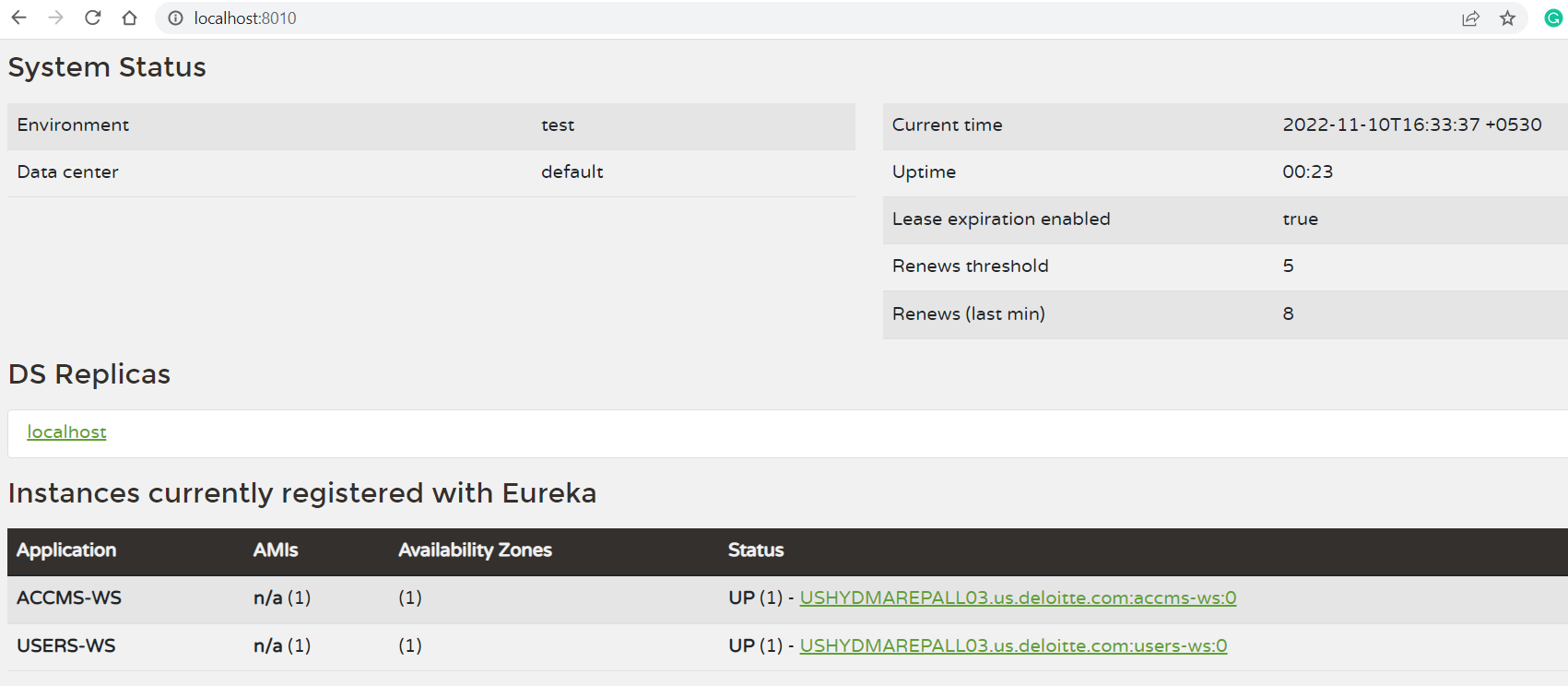
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

spring.devtools.restart.enabled=true

**Accessing Account Management service via Eureka service:**

Run the Account Management service.

Open Eureka server and click link against ACCMS service.



# Zuul API Gateway (Deprecated)

Note: Zuul API Gateway is in maintenance mode. New Spring Boot and spring versions don’t support Zuul.

Zuul and Ribbon were removed from the s­pringboot main channel in 2.5.x.

**Spring Cloud API Gateway is the latest and recommended.**

However, you can explicitly add the Zuul dependency.

Dependencies:

Web, Eureka Discovery, Dev Tools and Zuul

Maven Dependency:

<!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-netflix-zuul -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-zuul</artifactId>

<version>2.2.10.RELEASE</version>

</dependency>

Create Zuul Service:

@SpringBootApplication

@EnableEurekaClient

@EnableZuulProxy

**public** **class** ZuulApiAppApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ZuulApiAppApplication.**class**, args);

}

}

Application.properties:

server.port=8011

spring.application.name=zuul

eureka.client.service-url.defaultZone=http://localhost:8010/eureka

spring.devtools.restart.enabled=true

Open Eureka Dashboard and see that zuul is registered

It can’t be tested with latest Spring version. Hence, Ignore it, start working with Spring CLOUD API Gateway.

# Spring Cloud API Gateway

API Gateway provided by Spring to route requests based on some criteria.

## Dynamic Routing with Gateway

The Spring Cloud Gateway has three important parts to it. Those are −

**Route** − These are the building blocks of the gateway which contain URL to which request is to be forwarded to and the predicates and filters that are applied on the incoming requests.

**Predicate** − These are the set of criteria which should match for the incoming requests to be forwarded to internal microservices.

For example, a path predicate will forward the request only if the incoming URL contains that path.

**Filters** − These act as the place where you can modify the incoming requests before sending the requests to the internal microservices or before responding back to the client.

# Spring Cloud API Gateway service creation

eureka.instance.hostname=localhost

this entry to be added in every microservice to be called…

For any change.. Gateway service to be restarted..

**Dependencies:**

* Eureka discovery client
* Gateway
* Spring web

@SpringBootApplication

@EnableEurekaClient

**public** **class** ApiGatewayAppApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ApiGatewayAppApplication.**class**, args);

}

}

**Application.properties:**

server.port=8012

spring.application.name=gateway

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

## **Automatic Mapping of Gateway Routes:**

#discovery.locator to ensure that the gateway can read from the Eureka server.

spring.cloud.gateway.discovery.locator.enabled = true

#Works with lowercase of eureka client service

spring.cloud.gateway.discovery.locator.lower-case-service-id=true

**Access Microservices via Spring Cloud API Gateway - Automatic**:

<http://localhost:8012/user-ws/users/status/check>

<http://localhost:8012/accms-ws/users/status/check>

## **#Manual Configuring API Gateway Routes**

spring.cloud.gateway.routes[0].id=users-status-check

spring.cloud.gateway.routes[0].uri=lb://users-ws

spring.cloud.gateway.routes[0].predicates[0]=Path=/users/status/check

spring.cloud.gateway.routes[0].predicates[1]=Method=GET

Note: GET should be capital

**Multiple configurations:**

#Manual Configuring API Gateway Routes

spring.cloud.gateway.routes[0].id=admin-service-id

spring.cloud.gateway.routes[0].uri=lb://admin-service

spring.cloud.gateway.routes[0].predicates[0]=Path=/status

spring.cloud.gateway.routes[0].predicates[1]=Method=GET

#Manual Configuring API Gateway Routes

spring.cloud.gateway.routes[1].id=user-service-id

spring.cloud.gateway.routes[1].uri=lb://user-service

spring.cloud.gateway.routes[1].predicates[0]=Path=/users

spring.cloud.gateway.routes[1].predicates[1]=Method=GET

**Access Microservices via Spring Cloud API Gateway – Manual configuration**:

<http://localhost:8012/users/status/check>

## **Starting up Multiple Instances/More Services**

**Make this change in Eureka client services**

Application.properties

server.port=${PORT:0}

eureka.instance.instance-id=${spring.application.name}:${spring.application.instance\_id:${random.value}}

(Or)

**Run multiple micro service instances from Command prompt with diff port number**

java -Dserver.port=8182 -jar <jar filename>

# Load Balance

Run the same service twice or more than that and hit the endpoint and see the output.

The output is seen with two or more ports.

**Controller change in UserAPI:**

@RestController

@RequestMapping("/users")

**public** **class** UserController {

@Autowired

Environment env;

@GetMapping(path = "/status/check")

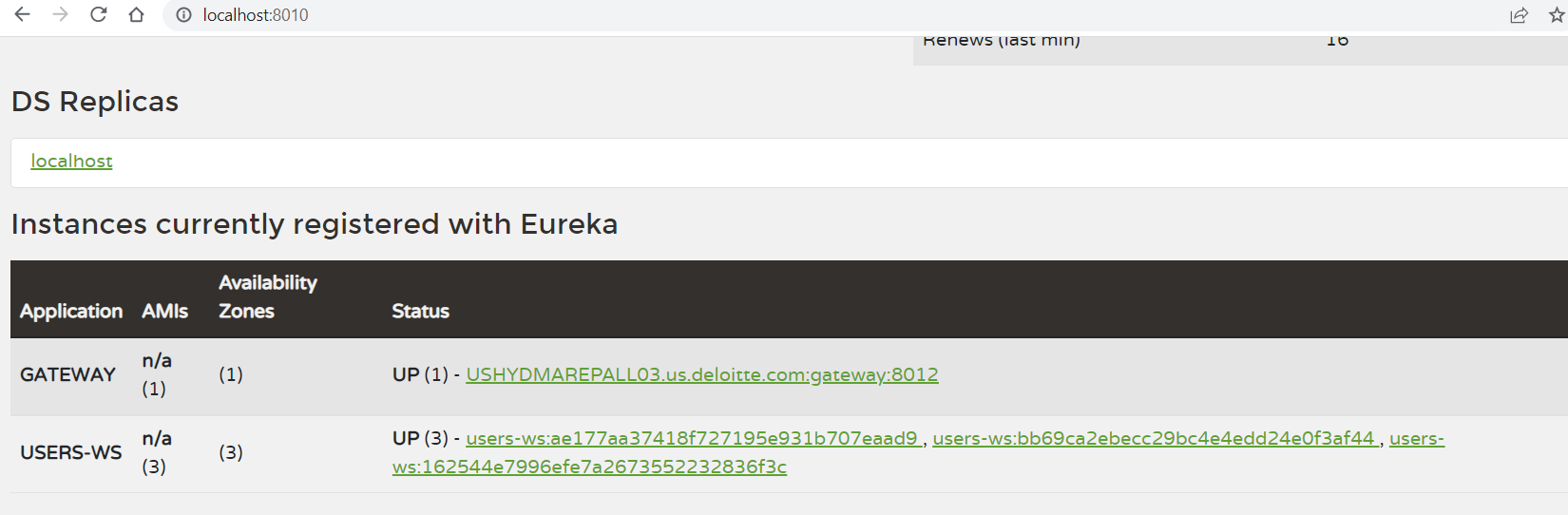
**public** String status() {

**return** "Working from PhotoAppApi Service, Congratulations!" +env.getProperty("local.server.port");

}

}

Load is balanced to all available instances



Test:

<http://localhost:8012/users/status/check>

Working from PhotoAppApi Service, Congratulations!55582

Working from PhotoAppApi Service, Congratulations!55585

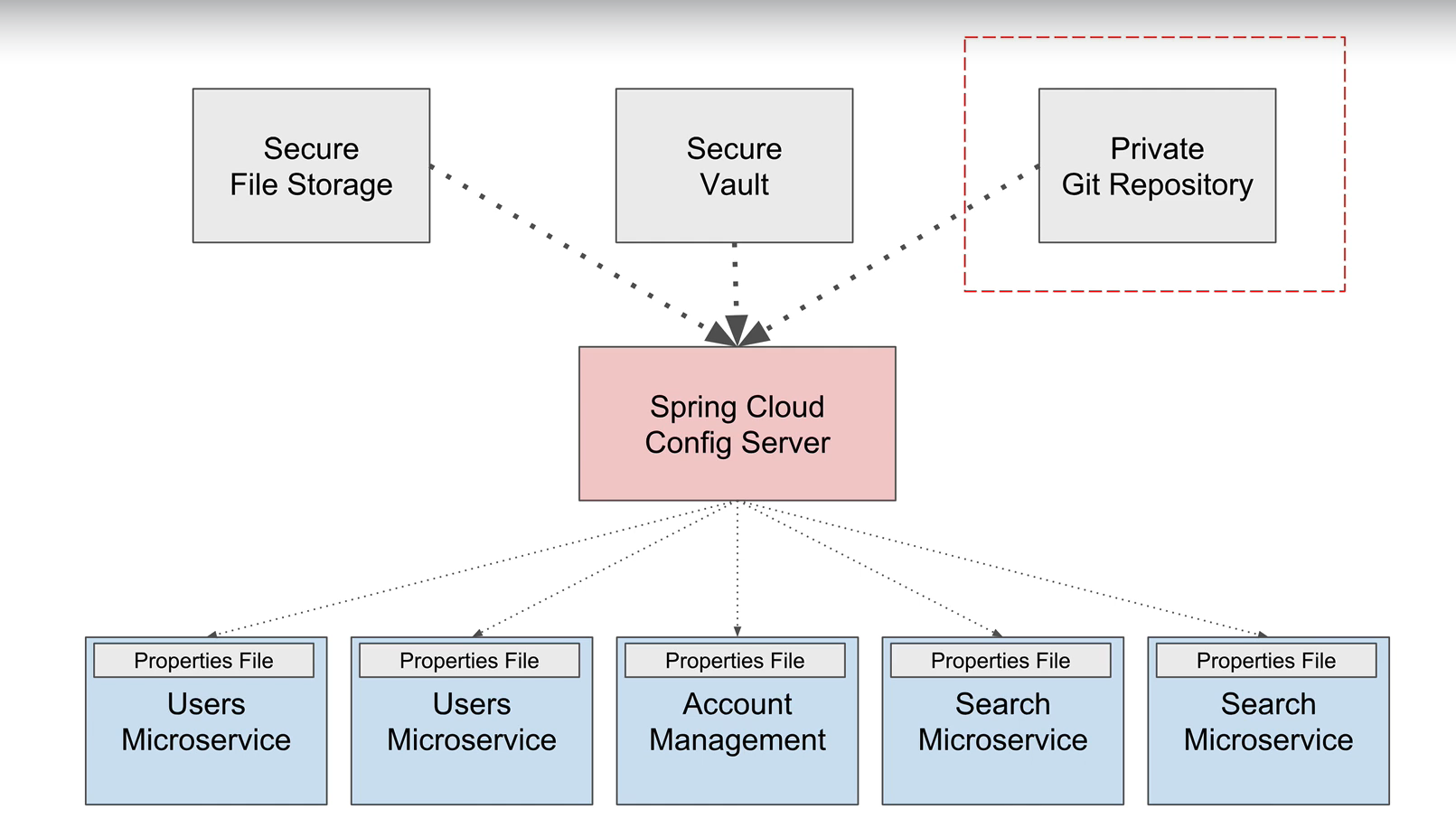
Working from PhotoAppApi Service, Congratulations!55586

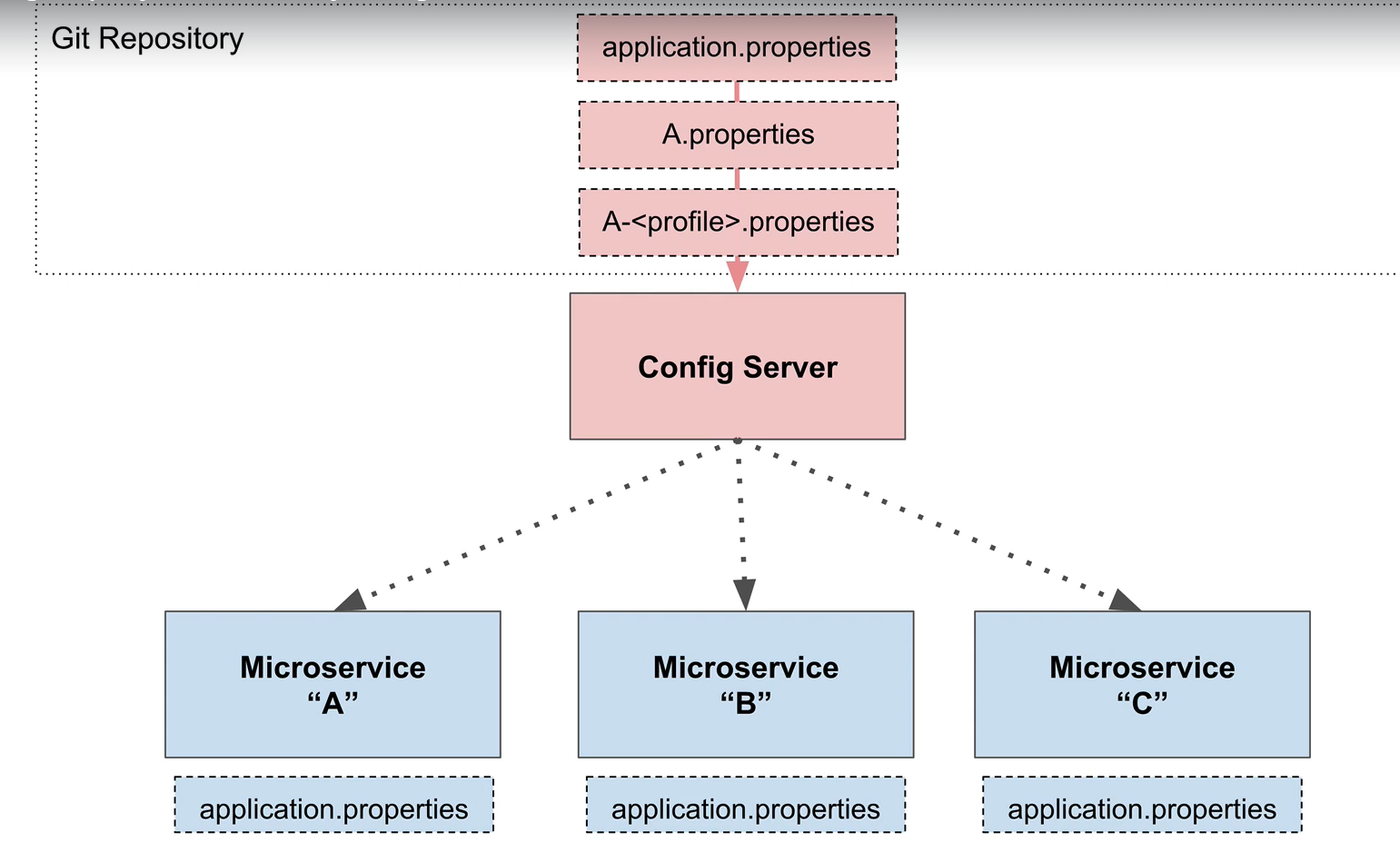
# Spring Cloud Config

Spring Cloud Config provides server-side and client-side support for externalized configuration in a distributed system.

# Spring Cloud Config Server

It is for centralized configuration. With the Config Server, you have a central place to manage external properties for applications across all environments





## Private GitHub Repository creation

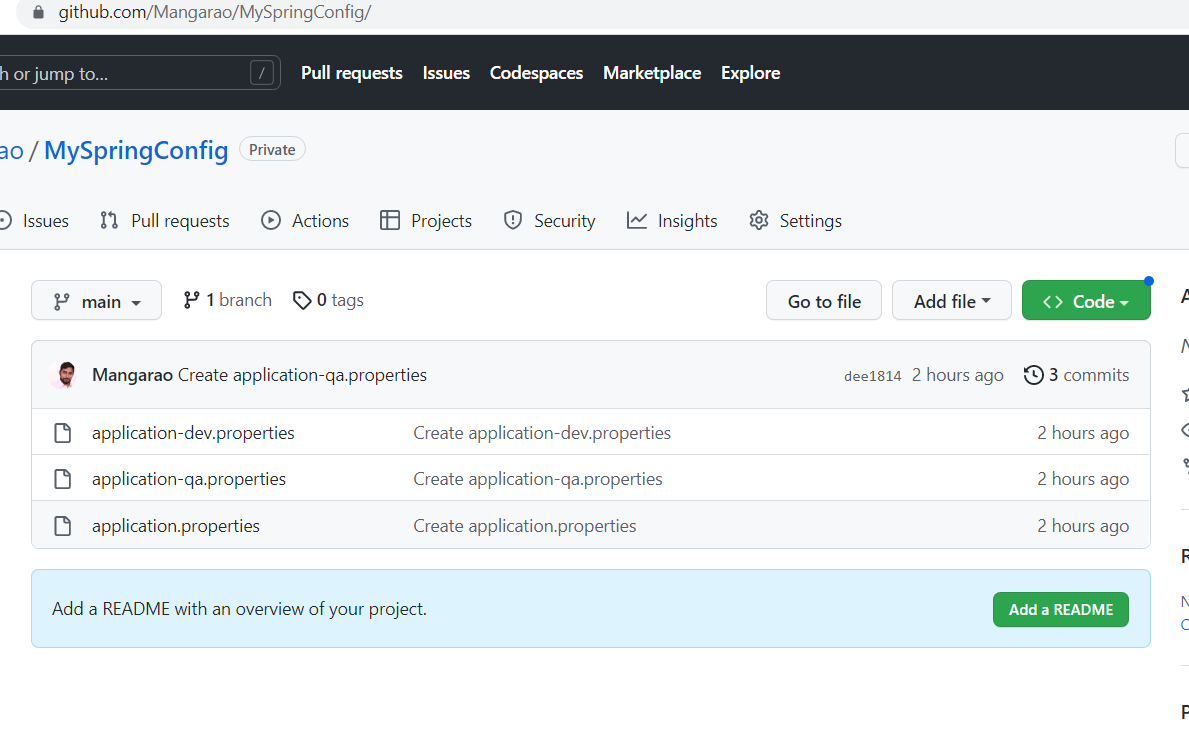
Login to Github

Create private repository

Directly create new files from the repository

or

Clone it to local and push the properties file to Git repository



## Add application.properties file to the Git

application.properties

message=Hello, from Git repo file

secret.token=sdsdggssdsd12dssgsdgs

application-dev.properties

message=Hello, from Git repo file from git DEV

secret.token=sdsdggssdsd12dssgsdgs from git DEV

application-qa.properties

message=Hello, from Git repo file from git QA

secret.token=sdsdggssdsd12dssgsdgs from git QA

# Config Server – Git Backend

Create microservice for it: PhotoAppApiConfigServer

**Dependency:** Config server

@EnableConfigServer to main class

@SpringBootApplication

@EnableConfigServer

**public** **class** SpringApiConfigServerApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringApiConfigServerApplication.**class**, args);

}

}

Note: when public repository is created, no need to mention username and password in the config server properties file

Application.properties:

spring.application.name=PhotoAppApiConfigServer

server.port=8013

spring.profiles.active=git

spring.cloud.config.server.git.uri=https://github.com/Mangarao/SpringApiConfigGitRepo

spring.cloud.config.server.git.username=mangarao.arepalli@gmail.com

#secret key as password

spring.cloud.config.server.git.password=ghp\_NJSzoKWfwTgWDQCRs1ReAjW3fEgD5h1oobZr

#verify any repository issue in server

spring.cloud.config.server.git.clone-on-start=true

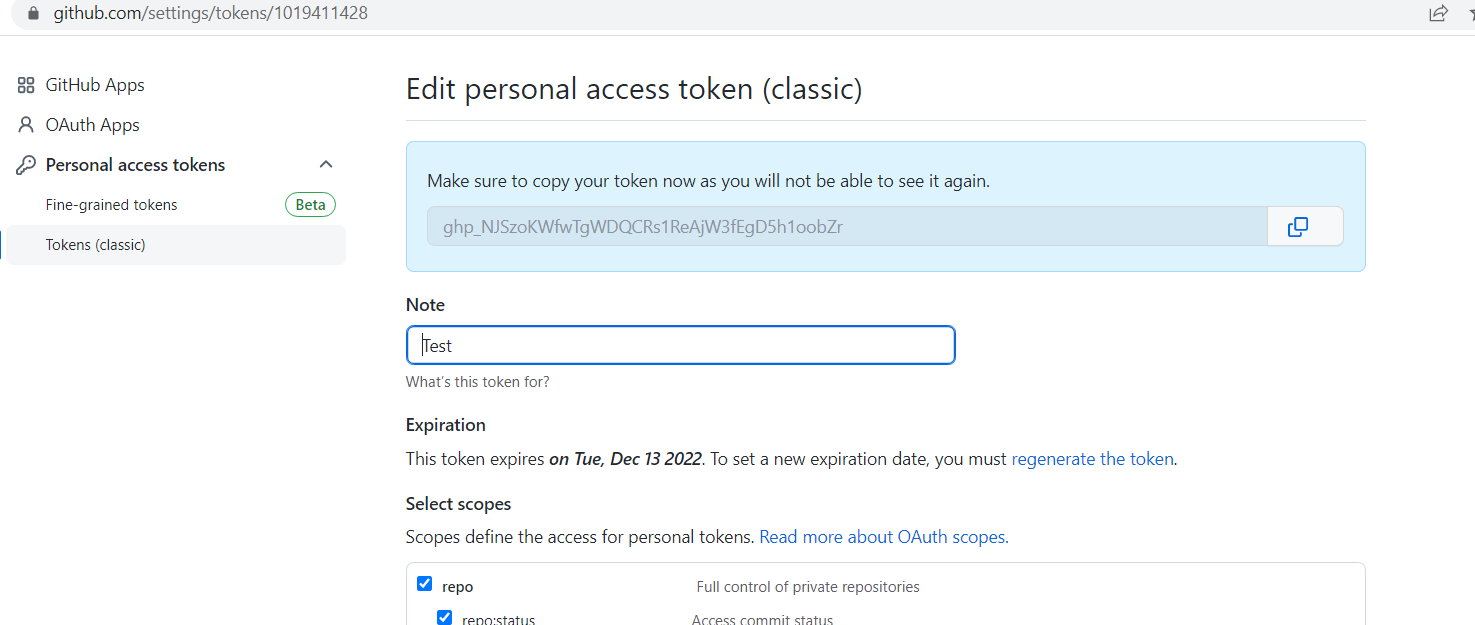
#In newer vesion add following property

## #spring.cloud.config.server.git.default-label=main

# How to fix 'Can't connect to any URI: https://github.com/\*/Github.git - not authorized' error

Mention secret key in place of password

Open Github account -> Settings Developer settings -> Personal access token-> Create a new token by selecting repo option only -> copy the token and use it as Github password in spring configuration file or STS git to communicate with Git



## Querying the Configuration

Convention:

Config server <ip address>:port no./AppName/<profile-name>

In case no profile, mention default

<http://localhost:8013/PhotoAppApiConfigServer/defult>

<http://localhost:8013/PhotoAppApiConfigServer/qa>

<http://localhost:8013/PhotoAppApiConfigServer/dev>

/{application}-{profile}.properties –

http://localhost:8013/ PhotoAppApiConfigServer -application.properties

http://localhost:8013/ PhotoAppApiConfigServer -dev.properties

/{application}-{profile}.yml – In case YAML file configuration

http://localhost:8013/ PhotoAppApiConfigServer -application.yml

# Config Client - Configure a service to client of Config Server

Create a microservice for client of Config

Dependencies: Spring Web, **Config Client**

Config client dependency gets added…

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

In Newer versions, please add following dependency

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-bootstrap</artifactId>

</dependency>

## Application.properties

server.port=8014

spring.application.name=config-client

message = Hello from Microservice A client

secret.token = token from Microserivce A client

spring.config.import=optional:configserver:http://localhost:8013

# localhost:8013 is a domain and port on which config server is running

spring.profiles.active=qa

In newer versions, you can skip creating bootstrap.properites

Add bootstrap.properties fle with following properties under Resource: - only for Old versions

spring.cloud.config.uri=http://localhost:8013

spring.cloud.config.name=PhotoAppApiConfigServer

Controller class for Testing

@RestController

@RequestMapping("/users/")

**public** **class** ConfigClientController {

@Autowired

Environment env;

@GetMapping("/gitProperty")

**public** String getPropertyValue() {

**return** "Secret token "+env.getProperty("secret.token")+" message "+ env.getProperty("message");

}

}

Test:

<http://localhost:8014/users/gitProperty>

Note: For every change in External configuration file, Restart the client config service to reflect the changes

# Create own config server – File Backend

Create configuration files in local folder and set the path in application.properties

Change only in application.properties of Config server

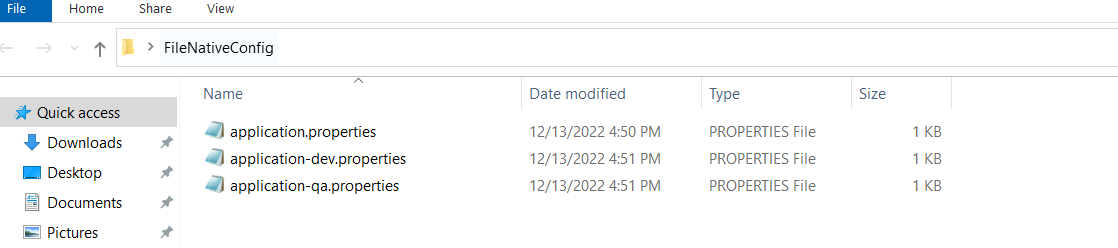
## Application.propertis

spring.profiles.active=native

spring.cloud.config.server.native.search-locations=file:///C:\\Users\\marepalli\\Desktop\\springioproject\\filenative

## How to set profile from command prompt

java -jar <executable jar> --spring.profiles.active=qa



**Note: In case property exists in local and config server properties files, priority goes to config server**

**And File name with micro service of client name is on first priority always.**

# Create a property file name with client micro service and place it in Config server.

**Client service name in application.properties file is** spring.application.name=config-client

### config-client.properties

message=Hello, this message is from File base approach with config client service name file

secret.token=sdgsdgsd this key is from File base approach with config client service name file

Note:

in client-config application.properties file, don’t set any active profile to test the output of config-client properties files entries.

#spring.profiles.active=qa

## Create a property file name with client microservice name and profile like qa

### config-client-qa.properties

message=Hello, this message is from File base approach with config client service name file pointed to qa

secret.token=sdgsdgsd this key is from File base approach with config client service name file pointed to qa

set spring.profiles.active=qa in client config application.properties file and test it

spring.profiles.active=qa

**Test:**

**Config Server endpoint:**

<http://localhost:8013/client-service/default>

**Config client endpoint:**

<http://localhost:8014/users/gitProperty>

# Selecting Beans by Profile:

@Repository

@Profile("production")

**public** **class** DataSourceBean {

}

@Repository

@Profile("dev")

**public** **class** LocalDataSourceBean {

}

Test:

<http://localhost:8013/application/native>

<http://localhost:8013/application/default>

<http://localhost:8013/application/qa>

<http://localhost:8013/application/dev>

## Test config client

<http://localhost:8014/users/gitProperty>

# Microservices communication

Communication among two or more microservices

Synchronous HTTP Communication is possible in two ways with:

1. REST Template
2. Feign Client

RestTemplate (Deprecated)

is a synchronous client to perform HTTP requests used for Microservices communication

REST Template is the easiest way to establish synchronous communication from m1 to m2.

Asynchronous communication over AMQP using Web Client

WebClient

is a reactive, asynchronous client to perform HTTP requests a part of Spring Webflux framework used for Microservices communication

# RestTemplate Example

Calling Another Service/API from one Service/API

## Target Service – Microservice 2

@RestController

@RequestMapping("/calc")

**public** **class** CalcController {

@GetMapping("/square/{num}")

**public** **int** square(@PathVariable **int** num) {

**return** num \* num;

}

@GetMapping("/bye")

**public** String sayBye() {

**return** "Bye from Calc service";

}

}

Note: Define RestTemplate bean in Main or separate Configuration class.

@SpringBootApplication internally contains @Configuration annotation

@LoadBalanced – it is for client-side load balancing

## Source Service - Microservice 1

**Main Class**

@Bean

@LoadBalanced

**public** RestTemplate getRestTemplate() {

**return** **new** RestTemplate();

}

**Controller**

@Autowired

RestTemplate restTemplate;

@Autowired

WebClient.Builder webClient;

1. **Using getForObject()**

@GetMapping("/callA/{num}")

**public** String callAnotherService(@PathVariable **int** num) {

String url="http://localhost:8089/calc/square/"+num;

**return** restTemplate.getForObject(url, String.**class**);

}

1. **Using getForEntity()**

@GetMapping("/callB/{num}")

**public** ResponseEntity<String> callAnotherService1(@PathVariable **int** num) {

String url="http://localhost:8089/calc/square/"+num;

**return** restTemplate.getForEntity(url, String.**class**);

}

getForObject() will perform a GET and return an object

getForEntity() : executes a GET request and returns an object of ResponseEntity class that contains both the status code and the resource as an object.

# WebClient Example

Dependency:

Spring Reactive web (spring-boot-starter-webflux)

**Main Class:**

@Bean

**public** WebClient.Builder getWebClientBuilder(){

**return** WebClient.*builder*();

}

**Controller:**

@Autowired

WebClient.Builder webClient;

@GetMapping("/callA/{num}")

**public** String callAnotherService(@PathVariable **int** num) {

String url="http://localhost:8089/calc/square/"+num;

**return** webClient.build().get().uri(url).retrieve().bodyToMono(String.**class**).block();

// bodyToMono () -> Mono like asynchronous call, control comes out of the

// block() stops asynchronous call

}

# FeignClient

it is convenient tool to use, internally uses RestTemplate, Recommended for Synchronous calls.

Is an HTTP Client, Declarative with @FeignClient, Load Balanced

Feign makes writing web service clients easier with pluggable annotation support.

One great thing about using Feign is that we don't have to write any code for calling the service, other than an interface definition.

## Dependencies:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

## Feign Client:

@EnableFeignClients annotation to Main class

@SpringBootApplication

@EnableFeignClients

**public** **class** FeignClientDemoApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(FeignClientDemoApplication.**class**, args);

}

}

## Feign Client Declaration:

Declare a Feign client using the @FeignClient annotation.

The *value* argument passed in the @FeignClient annotation is a mandatory, arbitrary client name, while with the url argument, we specify the API base URL.

Feign client class is composed of following:

Decoder – ResponseEntityDecoder, which wraps SpringDecoder, used to decode the Response

Encoder – SpringEncoder is used to encode the RequestBody.

Logger – Slf4jLogger is the default logger used by Feign.

Contract – SpringMvcContract, which provides annotation processing

Feign-Builder – HystrixFeign.Builder is used to construct the components.

Client – LoadBalancerFeignClient or default Feign client

## Feign Interface:

Works as a layer to call External service

Note: URL to be used not path inside @FeignClient annotation declaration

@FeignClient(name = "calculator", url = "http://localhost:8089/calc")

**public** **interface** FeignServiceUtil {

@RequestMapping(method = RequestMethod.***GET***, value = "/bye")

String getByeMessage();

@RequestMapping(method = RequestMethod.***GET***, value = "/square/{num}")

String getCalcSquare(@PathVariable **int** num);

}

## Controller

@Autowired

FeignServiceUtil feignServiceUtil;

@RequestMapping({ "/{num}", "/hello/{num}" })

**public** String sayHello(@PathVariable **int** num) {

**return** feignServiceUtil.getCalcSquare(10);

}

@GetMapping("/bye")

**public** String sayBye() {

**return** feignServiceUtil.getByeMessage();

}

## Test

1. Up the External service
2. Up the Feign service app and hit the URL

Calling External dummy microservice.

@FeignClient(name = "microservice2", url = "https://jsonplaceholder.typicode.com/users")

**public** **interface** FeignServiceUtil {

@RequestMapping(value = "/9")

**public** String callMicroservice2();

}

Controller:

@Autowired

FeignServiceUtil feignServiceUtil;

@RequestMapping("/ms1")

**public** String wish() {

**return** feignServiceUtil.callMicroservice2();

}