Software’s required

1. Java 17
2. PostgreSQL – zip
3. STS/Eclipse IDE
4. Node.js 16 or later
5. VS Code

OOPS concepts

1. Encapsulation – Java Beans (private data & public methods)
2. Inheritance - extends
3. Abstraction – abstract class & interface
4. Polymorphism – overloading & overriding

interface A {   
 void m1();  
}

class B implements A {   
 public void m1() { … }  
}  
class C implements A {   
 public void m1() { … }  
}  
  
Factory pattern: it abstracts object creation  
class Z {   
 public static A getInstance() { return new B(); //or return new C(); }  
}

//1st way

Client program

A a1 = new B();  
a1.m1();

2nd way

Client program

A a1 = Z.getInstance(); //   
a1.m1();

Collection Framework

It provides APIs to manage the data in various forms

* List: Index based – ArrayList, LinkedList
* Set: Stores only unique data – HashSet, TreeSet, LinkedHashSet
* Queue: Removes the data in FIFO / Sorted order – PriorityQueue, ArrayDeque

Map: It maintains the data in key value pairs

* HashMap
* LinkedHashMap
* TreeMap

Collection Framework – add(), remove(), size(), clear(), iterator(), forEach(), stream()

Collection: It is an interface at the top level, which provides all the methods

List, Queue & Set extends Collection – these are also interfaces

Comparable: It is used to provide default sorting technique to the complex objects, it has a method

* public int compareTo(T t);

compareTo returns an int value which could be 0 or +ve or -ve when two values are compared

Integer.compare(x, y): It compares two int values and returns 0 or +ve or -ve

implementing comparable

class User implements Comparable<User> {  
 userId, name, dob;  
   
 public int compareTo(User other) {   
 return Integer.compare(userId, other.userId);  
 }  
}

Comparator<T>: It is used to provide multiple sorting logic when you want to override the default sorting logic i.e., Comparable<T>, it is not implemented in the same class, instead implemented independently and its objects is supplied to the TreeSet, it has a method called

* public int compare(x, y)

Implementing Comparator

class SortById implements Comparator<User> {

public int compare(User u1, User u2) { ... }

}

class SortByName implements Comparator<User> {

public int compare(User u1, User u2) { ... }

}

Set s1 = new TreeSet(new SortByName());

Set s2= new TreeSet(new SortById());

// local anonymous classes

Comparator<User> c1 = new Comparator<User>() {

public int compare(User u1, User u2) { ... }

}

Comparator<User> c2 = new Comparator<User>() {

public int compare(User u1, User u2) { ... }

}

Set s1 = new TreeSet(c1);

Set s2= new TreeSet(c2);

// lambda expression : interface having only one abstract method

Comparator<User> c3 = ( u1, u2 ) -> { return Integer.compare(u1.id, u2.id) }

Comparator<User> c4 = ( u1, u2 ) -> Integer.compare(u1.id, u2.id) //

Comparator c5 = new Comparator<User> {

public int compare(User u1, User u2) {

return Integer.compare(u1.id, u2.id);

}

}

Set<User> set = new TreeSet(c4);

Set<User> set = new TreeSet( (u1, u2) -> Integer.compare(u1.id, u2.id) );

Streams & Lambda expressions

Streams: These take care of working with collections in a way so that you can process and manipulate it in an easier way

Suppose you have a collection & you want to apply some conditions

Functional Interface: Interfaces which have only one abstract methods, you can pass function itself as a parameter instead of passing an object

public void test(X x) {   
  
}

test( lambda expression );

List of functional interfaces

|  |  |  |
| --- | --- | --- |
| Functional Interface | Method name | Lambda expression |
| Comparator<T> | int compare(x, y) | (x, y) -> intValue; |
| Predicate<T> | boolean test(T t) | t -> booleanValue |
| Consumer<T> | void accept(T t) | t -> statement |
| Function<T> | T apply(T t) | t -> someValue |

Stream methods

1. filter(Predicate) : filter(t -> t == 2)
2. forEach(Consumer): forEach(t -> System.out.print(t) )
3. sorted(Comparator): sort( (x, y) -> intValue )
4. map(Function): map(t -> someValue)

Activity:

1. Create a class Item with id, name, price, ratings – generate 2 constructors, setters, getters, toString
2. Create some Item objects atleast 10 to 15 in the main method and add them in a List<Item>

ex: name: Mobile, price:20000, ratings: 4.5

Stream has two types of methods

1. Intermediate methods: This generates another stream where you can chain the methods, ex: filter, sort, map,
2. Terminal methods: This is the end of stream, which doesn’t generate another stream, here you can’t chain the methods, ex: forEach, sum, count, collect

Chaining the methods

filter(..).filter(..).filter(…)

filter(…).map(…).sort(..)

PostgreSQL

It is an RDBMS software that maintains the data, SQL is the language RDBMS understand.

Sub languages of SQL

1. DDL: create, alter, drop & truncate
2. DML: insert, update & delete
3. DQL: select
4. DCL: grant, revoke
5. TCL: commit, rollback

Create a folder named pgsql\_data in some drive like C or D

How to set username & password

initdb.exe -D pathOfpgsql\_data -U username -W -E UTF8 -A scram-sha-256

How to start the database

**pg\_ctl -D D:\Labs\pgsql\_data -l logfile start**

How to login to the database

psql.exe -U postgres

Datatypes

* int
* varchar
* bigint
* float
* date
* json

Constraint: primary key, not null, foreign key, unique, check

Functions: upper(), lower(), to\_date(), sum(), count(), avg()

Views: It is a virtual table that refers to the select query

create view view\_name as (sql query)

Index: It is used to increase search performance

Full Text Search

like operator is used to perform the text search, but it is not effective during the complex searches like searching from the dictionary words or the words which are derived from the base word

where text like ‘%a%’; # this will search the whatever the letters comes before or after a

where text like ‘satisf%’; # this will search satisfy, satisfied, satisfies

like operator doesn’t ignore stop words like from, to, where, and, a, and so on.

Full Text search can ignore lot of stop words and searches the text derived from the base words

ex: satisfy : it searches satisfies, satisfied, satisfy

ex: inspire: it searches inspires, inspired, inspiration

tsvector: it is a datatype, it would parse the text (document) & break the text into tokens and looks each token in the dictionary to find the lexemes(base words and their derived words), it also ignores the stop words like a, and, or, between, then, from, and so on.

tsquery: it is used to mention the word that you want to search

Example: using tsvector & tsquery

select \* from test where to\_tsvector(description) @@ to\_tsquery(‘friendly’);

the above query finds all the derived words from the base word friend like friends, friend, friendly

Searching the lexemes

select name, description from test;

select name, description from test where to\_tsvector(description) @@ to\_tsquery('satisfy');

select name, description from test where to\_tsvector(description) @@ to\_tsquery('friend');

Searching from the stop words

select name, description from test where to\_tsvector('simple',description) @@ to\_tsquery('simple','to');

Concatenating the documents to find the words in multiple columns

select name, description from test where to\_tsvector(name ||’ ‘||description) @@ to\_tsquery(‘word’);

Transaction isolation

When concurrently query is executed user might sometimes see the other user updates and sometimes they might not see other user updates

1. Read committed
2. Repeatable Read

Read Committed:

create table t1(id int, name varchar(10));

open second command prompt;

CMD1 : insert 2 records

CMD2 : select query – shows those two records, because by default its auto-commit & transaction isolation level is read commit

Repeatable Read: In this isolation level, a user under transaction can’t see another user changes even if the another user has done commit in the current transaction, they can see the changes in the next transaction

T1 -> begin transaction isolation level repeatable read;

T2 -> begin; -> update some data -> commit;

T1 -> select -> you don’t see the T2 changes

T1 -> commit/rollback -> you see the T2 changes

Activity:

T1 -> begin transaction isolation level repeatable read;

T2 -> begin; -> update some data

T1 -> update same data of T2 -> you will see T1 will be locked

T2 -> commit/rollback -> you will see the T1 will be unlocked

Performance Tips

1. You must not use \* while using select, instead you must use column names which you want to retrieve
2. Avoid using distinct because it selects common value & groups them as a single value
3. Use limit when you want to preview the result
4. Use index to increase search speed
5. Use like in the beginning of the phrase i.e., like ‘A%’ instead of ‘%A%’
6. Use full text search and index the vector(tsvector) to increase the search speed
7. Use prepared statement in the application that executes the query for different values without reparsing and recompiling the query each time

Note: In Postgresql you can use explain analyze command to find out how much time a query takes to execute

ex: explain analyze select name, description from test;

Query to create tsvector in the table

alter table test add document tsvector;

update test set document=to\_tsvector (name||' '||description);

Query without indexing

explain analyze select name, description from test where to\_tsvector (name||' '||description) @@ to\_tsquery('friend');

Output: Execution time will be shown

Query with indexing

explain analyze select name, description from test where document @@ to\_tsquery('friend');

Output: Execution time will be shown which will lesser to the previous query

JDBC: Java Database Connectivity that provides set of API’s to interact with any database

JDBC API’s >> Database Vendors provide the implementations for their database

i.e.,

MySQL has implemented JDBC API to interact with MySQL database

Postgres has implemented JDBC API to interact with Postgres database

These vendors provide the implementations in a jar file which you must use in your project.

How to interact with the database using JDBC

1. Loading the JDBC drivers (implementations) – optional in JDBC

Class.forName(driverClassName);

1. Connecting to the database

Connection connection = DriverManager.getConnection(url, un, pwd);

1. Creating the statements

PreparedStatement statement = connection.prepareStatement( sqlQuery );

sqlQuery = “insert into test values(?, ?, ?)”;

sqlQuery = “select name, phone from test where id = ?”;

// setting values to the ? positions  
statement.setInt(1, value);

statement.setString(2, value);  
statement.setDouble(3, value);  
statement.setString(4, value);

1. Execute statements

int count = statement.executeUpdate(); // for DML queries

ResultSet result = statement.executeQuery(); // for DQL queries

1. Closing resources

result.close();  
statement.close();  
connection.close();

Activity:

1. Create a Student class with 3 properties rollNo, name, dob, retrieve all the student records from the database and store each record’s value to the corresponding Student object properties and then add that student object to the List<Student> and return the List<Student>
2. Perform a store operation by storing rollNo, name & dob in the database by getting the values from the student object

ex: Student student = new Student(5, ‘Frank’, LocalDate.parse(‘2000-09-20’));

student object must be stored in student table where roll\_no = 5, name = Frank and dob = 2000-09-20

Note: Create a separate class like StudentDaoJdbcImpl for database operations & call them from the main method, the class must have two methods as below

1. public int save(Student student) { … } // stores student using insert query
2. public List<Student> findAll() { … } // retrieves all the students
3. public Student find(int rollNo) { … } // this returns student based on rollNo

ORM framework

Object Relational Mapping: Which maps Java objects directly to the database tables, it provides lot of benefits over JDBC(low-level implementation)

ORM simplifies data access by providing many inbuilt methods & features which gives lot of benefits over JDBC

1. Handling exceptions automatically
2. Establishing connection & closing resources are not required
3. Type conversion from Java types to SQL types and vice versa are not required
4. Separates the datasource configurations from the application

ORM uses entities which are java bean classes with table & column details

@Table(name = “student”)  
@Entity  
class Student {   
 @Id // to recognize primary key column  
 @Column(name = “roll\_no”)  
 private int rollNo;  
  
 @Column(name = “name”)  
 private String name;  
}

ORM framework is specified by JPA (Java Persistence API) which is implemented by many ORM frameworks like Hibernate, Spring Data Jpa

Note: In ORM you get inbuilt methods to perform CRUD operations which means you don’t have to write SQL queries, the inbuilt methods are  
save(object): generates insert query  
get(value, entityClassName): generates select query based on primary key  
delete(object): delete query will be generated

Hibernate uses a configuration file “hibernate.cfg.xml” which will have datasource information’s like url, username, password

hibernate.cfg.xml

url = db\_url  
username = db\_username  
password = db\_password

Hibernate provides some APIs to interact with the database

Configuration: loads the hibernate.cfg.xml file to identify datasource details  
SessionFactory: creates a connection pool to reuse the connections  
Session: used to perform save(), update(), delete(), get(), loadAll() operations  
Transaction: used while perform DML operations

2 libraries we have to use

1. postgresql: provides JDBC driver implementation
2. hibernate-core : provides all the Hibernate & JPA APIs

src/main/resources/hibernate.cfg.xml: keep this hierarchy if you get unable to locate the hibernate.cfg.xml

In ORM we get an independent query language called JPQL/HQL

JPQL: Java Persistence Query Language

HQL: Hibernate Query Language

Both are used to write queries for an entity

Note: You don’t use table names or column names instead you use entity class names & properties, ORM will generate corresponding SQL for the table & column

Some JPQL/HQL

|  |  |
| --- | --- |
| JPQL/HQL | SQL |
| Select s from Student s; | Select \* from student; |
| Select s from Student s where s.rollNo=?1 | Select \* from student where roll\_no = ? |

To get all the entities we must use select s from Student s;

Query<Student> query = session.createQuery(“select s from Student s”, Student.class);

List<Student> students = query.getResultList();

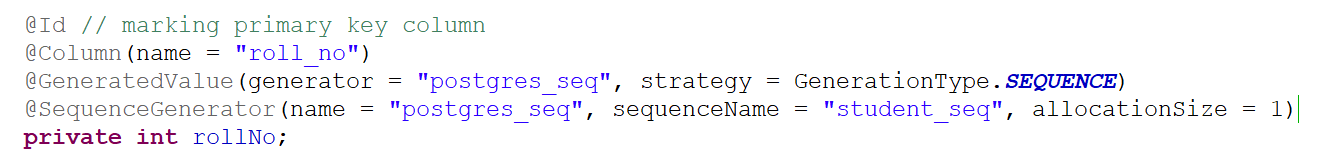
interface StudentDAO {   
 int save(Student student);  
 List<Student> findAll();  
 Student find(int rollNo);  
}

StudentDAOJdbcImpl & StudentDaoOrmIml both implements StudentDAO

Generating the primary key

@GeneratedValue(strategy = GeneratorType.IDENTITY) : use it when the table column has auto increment type like serial in postgres, auto\_increment in mysql

@GeneratedValue(generator = “someName”, strategy = GeneratorType.SEQUENCE): use when you have a sequence, in this you must also use another annotation i.e., @SequenceGenerator(name = “someName”, sequenceName = “sequence\_name”)



Layered architecture

Application must be divided into multiple layers, so that each layer takes the responsibility of writing some logics

View: Will have presentation logics

Controller: Takes the request, access the service and generate the response

Service: It will have business logics, access the DAO and returns the data to the Controller

DAO: It will have database logics and returns the data to the Service

Spring Framework:

Framework provides common logics every application needs so that developers can make use of that to speed up the application development

Spring Framework provides common features every application needs like

* Design patterns : factory, singleton, prototype, proxy and many more
* Connection pools: you don’t have create database connections & close their resources
* Type conversion: like strings to numbers, strings to date, java objects to sql types, sql types to java objects
* Exception Handling: it handles all the checked exceptions like SQLException, ClassNotFoundException, you can only use custom exceptions in your application

Spring framework can be used to develop all types of applications like web, rest based, cloud based applications, desktop applications

Spring does all these things by providing spring modules which are

1. Spring Core/Spring IOC(Inversion of Control): It provides a platform where all the design patterns are implemented, it also creates reusable objects which are required for the application, also performs dependency injection

Dependency Injection: It is a process where an object is supplied to another object

1. Spring Web: It provides a platform where you can create web & rest based applications
2. Spring Boot: it provides a platform where it provides auto-configuration features to easily develop spring based applications
3. Spring Cloud: It provides a platform to develop applications that can be deployed on any cloud environment, ex: microservices

Spring Core:

1. spring-context: Library used for spring core

Application Structure

1. src: DAO, Service, Controller, Utility
2. xml file: Configure all the classes which spring needs to maintain

src/main/resources/xml-files

src/main/java/java-files

interface TestDao { …. }  
class DaoOneImpl implements TestDao { }   
class DaoTwoImpl implements TestDao { }

With Factory pattern

TestDao dao = ObjectFactory.getInstance(); // returns the object that implements TestDao

In the above code you will create the factory pattern

In Spring: You don’t have to create factory pattern, you must declare that class in the xml file

beans.xml

<bean id = “a” class = “com.npci.DaoTwoImpl” />

Spring container creates a singleton object(default scope of the bean) of DaoOneImpl and maintains it in the container

Client code

ApplicationContext context; // this refers to the container

context = new ClassPathXmlApplicationContext(“beans.xml”);

TestDao dao = (TestDao)context.getBean(“a”);

// spring container gives the object whose bean:a

Dependency Injection: It is a process of initializing the object by spring container where an object is supplied to another object using setter or constructor.

Note: As a developer we initialize the object by calling setters/constructors, but in spring it is done automatically by the spring container using XML file

Type of DI

1. Setter injection: initializes the object using setter method with <property> tag
2. Constructor injection: initializes the object using constructor with <constructor-arg> tag

class Datasource {   
 String username;  
 String password;  
 String url;  
}

class ProfileDaoImpl implements ProfileDao {   
 private Datasource data; // setter method – setData(…)  
}

class ProfileServiceImpl {   
 private ProfileDao profileDao; // setter method – setProfileDao(…)  
}

In XML file for setter injection

<bean id = “ds” class = “com.npci.dao.Datasource”>  
 <property name = “username” value = “postgres” /> // calls setUsername  
 <property name = “password” value = “Welcome@1234” /> // calls setPassword  
 … some more property initialization ….  
</bean>  
<bean id = “dao” class = “com.npci.dao.ProfileDaoImpl”>  
 <property name = “data” ref = “ds” />   
</bean>

<bean id = “service” class = “com.npci.service.ProfileServiceImpl”>  
 <property name = “profileDao” ref = “dao” />   
</bean>  
In XML file for constructor injection

<bean id = “ds” class = “com.npci.dao.Datasource”>  
 <constructor-arg index = “0” value = “postgres” /> // value to the 1st parameter  
 <constructor-arg index = “1” value = “Welcome@1234” /> // value to the 2nd parameter  
 … some more property initialization ….  
</bean>

class Datasource {

private String username;

private String password;

}

interface ProfileDao { crud methods }

class ProfileDaoImplTwo implements ProfileDao {

private Datasource data; // setter method

// crud methods

}

class ProfileServiceImpl {

private ProfileDao profileDao; // setter method

}

<bean id = "service" class = "com.service.ProfileServiceImpl>

<property name = "profileDao" ref = "dao" />

</bean> [ profileDao = null ]

<bean id = "ds" class = "com.dao.Datasource">

<property name = "username" value = "postgres" />

<property name = "password" value = "Welcome@1234" />

</bean> [username = postgres, password = Welcome1234]

<bean id = "dao" class = "com.dao.ProfileDaoImplTwo">

<property name = "data" ref = "ds" />

</bean> [ data = ds]

Spring Boot:

It helps you to quickly create spring applications that is ready to deploy in the production

* It auto-configures the beans in the application without any XML file, spring boot uses some starter libraries and annotations to auto-configure
* It internally uses embedded server(tomcat is the default embedded server) – so that you don’t need any external server for web applications
* It make use all the spring features but most of the things are automated

Without Spring Boot

class Datasource { }   
class Utility { }

xml file: bean : Datasource  
 bean: Utility

With Spring Boot

@Component // default id will be className, begins with lower-case  
class Datasource { } // spring creates object and maintains it in the container  
  
@Component(“hello”) // id is hello  
class Utility { } // spring creates object & maintains it in the container

How does spring boot helps in supplying a dependency: It uses @Autowired annotation

interface X { }   
@Component  
class A implements X { }   
@Component   
class B {   
 @Autworired // supplies the object of type X  
 X x;  
}

Spring has provided many annotations which spring boot can use

There are annotations that helps spring to create objects & maintain in the container

1. @Component: Use in some utility classes
2. @Service: Use in service layer
3. @Repository: Use in DAO layer
4. @RestController: Use in Webservice Controller layer
5. @Controller: Use in Web controller layer
6. @Configuration: Use in Manual Bean Configuration class

Note: All these annotations you must use only on top of classes not on any interface or variables

Note: @Component is the base annotation, all other annotations like @Service, @Repository are derived from the @Component.

application.properties

This is the configuration file spring boot uses where you can write datasource details, server details and any other application details

ex:   
server.port = 9090  
spring.datasource.username = postgres  
spring.datasource.password = Welcome@1234

@SpringBootApplication: It is the main annotation that does auto-configurations in the spring boot application, things this annotation does

1. Component scanning – scans all the classes having @Component, @Service, @Repository, after scanning it creates their object in the spring container
2. Automatically configures the application based on the starter libraries added in the classpath
   1. Spring Data Jpa: If you add this library spring boot automatically connects to the database, it also implements DAO layer for you with all the CRUD operations
   2. Spring Web: This library provides default embedded tomcat server & other web related features are auto-configured(Front Controller & Request mapping)

Note: All the component classes must be in the base package or its sub-package from where the @SpringBootApplication begins

Ex: If com.org has @SpringBootApplication, then all your classes must be part of com.org or subpackage of com.org

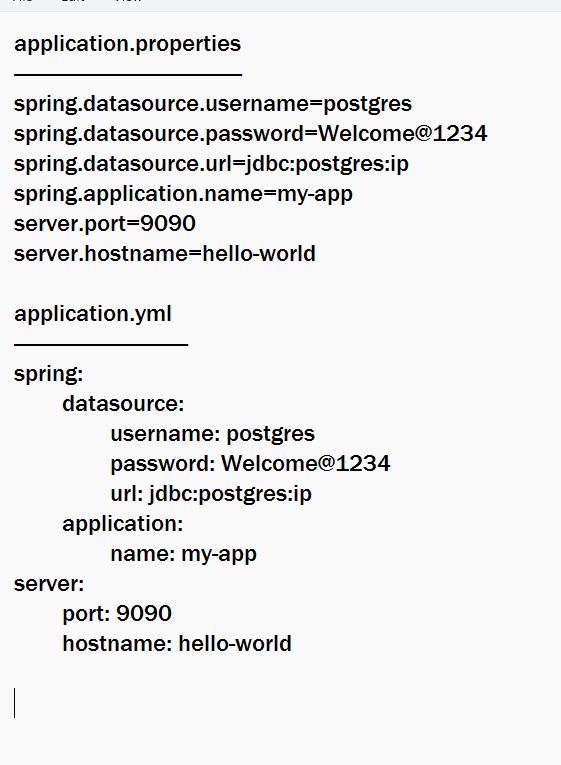
Spring Boot project

src >> java classes

resources >> application.properties or any other properties file

Configuration files

1. property
2. YAML



Running the jar

java -jar spring-boot-demo-0.0.1-SNAPSHOT.jar --server.port=9091

Spring Web:

1. Web applications: where you will have views in the same project
2. Web services: where you will create online APIs in the application but you will not have views in the same project

Spring Web provides all the necessary environment for web

* Front Controller
* View Resolver
* Request Mapping

Webservices allow multiple applications to exchange the data in a common format called JSON/XML/CSV

Example:

Payment payment = new Payment(777888, 5000); is represented in XML as

<payment>  
 <accountNumber>777888</accountNumber>  
 <amount>5000</amount>  
</payment>

JSON format for the above data

{ “accountNumber” : 777888, “amount”:5000 }

CSV format

777888;5000

In Spring we have annotations to create web applications & web services

1. @Controller: It is for web application controller
2. @RestController: It is for web service controller

Principles while using web service(REST – Representational State Transfer)

1. HTTP methods: GET, POST, PUT, DELETE
2. URL: location of the webservice

GET: For retrieving   
POST: For creating new resource  
PUT: For updating the resource  
DELETE: For deleting the resource

Client Server: C#  
Javascript: @POST: /someUrl  
Firstname: text POST: /url register() { }   
Lastname: text  
Phone: numbers  
DOB: date  
<<Submit>>

Spring web provides annotations for HTTP methods

1. @GetMapping
2. @PostMapping
3. @PutMapping
4. @DeleteMapping

FrontController:

1. It takes care of accepting all the incoming request
2. It takes care of using ApplicationContext so that you don’t have to getBean() anywhere, i.e., it is the one that initialize the spring container which is your application context – from this component-scanning occurs
3. Maps the request to the appropriate controller method

i.e., /login mapping to login() method

Path Parameters: path/{param}/{param} used to pass values in the URL, which can be extracted using @PathVaiable

Request Body: It can have complex structure in the request that can be JSON/XML which will be converted to Java object of a specific type by spring framework

@Autowired:

It supplies the object to another object, it is a replacement for <property ref = “beanId” .. />

public class ServiceImpl {   
 @Autowired  
 private ProfileDao dao;  
}

Spring Data JPA: It is a starter library that helps spring boot to automatically connect to the database by using datasource details in application.properties/yml

* It takes care of implementing the DAO layer using the repository type interface
* It creates object of the DAO implementation in the spring container so that it can be inject to any other objects

Spring Data JPA gives interfaces like

1. CrudRepository<T, ID>: T is an entity class, ID is the primary key type, this provides crud methods like save(T), delete(T), findById(ID), findAll() methods
2. JpaRepository<T, ID>: This extends CrudRepository, this provides some extra methods like sort(), and pagination methods

Developers has to extend any one of these interfaces to get the implementation of the DAO layer

interface ProfileRepository extends JpaRepository<Profile, Integer> { }

Spring Data JPA implements ProfileRepository and all the inherited methods of JpaRepository that interacts with the table Profile is mapped

ex:

save(T): is implemented as save(Profile)

findById(ID): is implemented as findById(Integer)

You can just autowired the ProfileDao to perform the operations

@Autowired  
ProfileRepository dao;

To save: dao.save(profileObject)

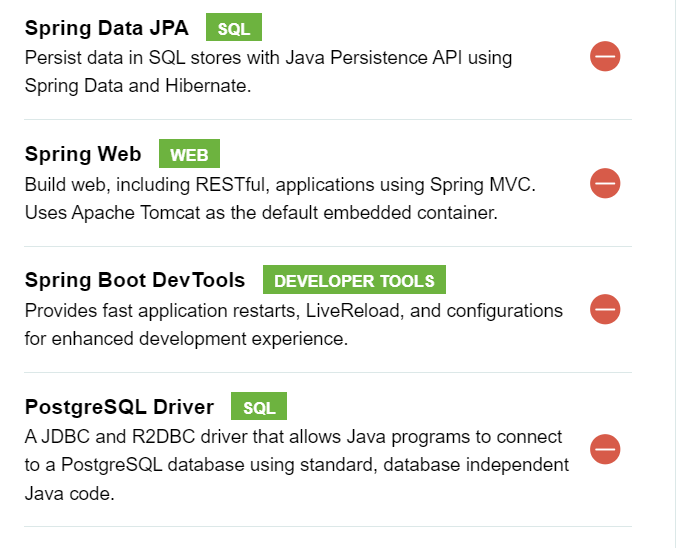
To find by id: dao.findById(id);

Library: spring data jpa

application.yml

database credentials like username, password, url, driverClassName

Libraries



Things to create

1. Profile table: id (primary key – serial), name, password, dob, phone – create from spring boot
2. Profile – entity class
3. ProfileRepository extends JpaRepository<Profile, Integer>
4. ProfileNotFoundException: if id is not found throw this exception
5. ProfileServiceImpl – calls ProfileRepository methods
6. ProfileController – calls ProfileService methods
7. application.properties – datasource details

Service layer depends on DAO layer

@Autowired  
private ProfileRepository dao; // spring boot injects the proxy implementation of the Repository, it is implemented to access profile table

dao.findAll(); returns List<Profile>  
dao.save(profile): saves the profile entity & returns the same entity  
dao.deleteById(id): deletes the profile entity matching to the id.

@RestController class : Will have @Autowired on ProfileService

save(T): It saves if id is not present else updates the entity if id is already present

Activity:

1. Create a webservice that accepts phone number to find an entity, it must throw exception if profile is not present– create appropriate methods in the service layer & call them from the controller
2. Create a webservice that accepts the id to delete the entity, it must throw exception if profile is not present – create appropriate methods in the service layer & call them – JpaRepository has an inbuilt method to delete, you can use that method from the service layer
3. Create a webservice that accepts the id and other profile details in JSON format to update the entity – it must throw exception as well, create appropriate methods in the service layer & call them – use save(T) method of DAO to update the entity

Parent & Child mappings

When a table is referenced by another table we can say it has a parent & child relationship, In ORM we can join these tables using mapping annotations which will simplify joining the tables without using any complex join queries.

1. @OneToOne
2. @OneToMany
3. @ManyToOne
4. @ManyToMany

These annotations takes care of joining tables & also obtaining entity object from the table

@Entity  
class Employee {   
 @OneToOne  
 Address address;   
 // here when you get employee it automatically gets address belonging to the employee  
 // when you store employee & address, automatically the employe id is stored in address table  
}  
@Entity  
class Address {   
  
}

We can use Profile & Contact to have a one to many mappings

class Profile {   
 int profileId;

@OneToMany  
 private List<Contact> contacts; // gets all the contacts of a Profile  
}

class Contact {   
 int contactId;  
 int profileIdRef;  
}

When we add contact for a particular profile we must mention the profile-id so that while storing contact profile-id will be stored in the contact table

Things to create

1. Contact entity: contactId, name, phone, profileIdRef
2. Profile entity: List<Contact> must have @OneToMany
3. ContactRepository: To perform CRUD operation on Contact entity
4. addContact(Contact): We must create in the Service, which should store the Contact for a particular profile-id
5. Autowired ContactRepository in the ProfileServiceImpl
6. Webservice to add & list the contacts for a particular profile

WAR File: These are created when you want to deploy in an external server, if you use jar of your application it runs in embedded server, but you can use war files to run in an external server

Changing the embedded server

By default spring boot uses tomcat but you can change the embedded server, these are the list of embedded servers spring boot supports

1. Eclipse – Jetty Server
2. Jboss – Undertow Server

exclusion: you must exclude the tomcat-server because spring web by default will have tomcat server

Microservices

They are small independent services/webservices that you can develop, test, build & deploy independently without affecting other services

Benefits

1. You don’t have to test other services when a service is modified
2. If a particular service go down other services will be still working
3. You can scale a particular service that has high traffic without scaling other services
4. You can use multiple technologies for multiple services
5. Releasing the new feature will not take more time as it doesn’t need to be integrated with other services

Microservice use some design patterns

1. Service Discovery: This registers the microservices
2. Client Side Load Balancer: It takes care of distributing the load across multiple instances of microservices
3. Distributed Configuration: Multiple microservices can share common configurations (Git, AWS can keep those configurations) in some remote environment

Spring Cloud: It is a library that implements all the microservice design pattern

1. Service Discovery: Is implemented by Eureka Server
2. Microservice client: Is implemented by Eureka Client
3. Client Side Load Balancer: Is implemented by Eureka Client library
4. Distributed Configuration: Is implemented by Configuration Server

Spring cloud libraries are interdependent to each other

Some default behaviors of Eureka Client

Eureka Client always look Eureka Server in 8761 port when it has to be registered in the service discovery by default.

Eureka Client automatically registers with Eureka Server by default

Eureka Client sends heart beats to Eureka Server every 30s to acknowledge its active status

Some of the features Eureka Server gives

Eureka Server gives a dashboard so that admins can see all the microservices that are registered in the eureka server.

Projects

1. Service Discovery – Spring Cloud – Eureka Server - @EnableEurekaServer
2. Microservice – Spring Cloud – Eureka Client – no need of any annotation – just use the library spring boot automatically configures it

Libraries for Service Discovery project

1. Eureka Server
2. Dev tools

Libraries for Microservice project

1. Web
2. Eureka Client
3. Dev tools
4. Actuators (monitor the application) – optional
5. JPA - optional

Communicating from one microservice to another microservice

RestTemplate: It is an HTTP client that can send HTTP requests to the remote service, it provides methods to represent the HTTP methods (GET, POST, PUT, DELETE)

Client microservice

@Bean  
@LoadBalanced  
public RestTemplate template() {

RestTemplate rest = new RestTemplate(); // should be registered in the spring container  
return rest;  
}

@Bean: Calls the methods and registers the object it returns in the spring container, so that you can autowire/inject that object in another object

@Autowired  
RestTemplate temp; // spring injects the RestTemplate

// calling a remote service using GET

temp.getForObject(URL)

// calling a remote service using POST

temp.postForObject(URL, data);

@LoadBalanced: It adds client side load balancer and resolves the Physical address using the instance-id & if there are multiple instances with same instance id, it distributes the request

Note: @LoadBalanced is written on a method that returns RestTemplate, hence any request you make from RestTemplate will be queried in the service discovery by the load balancer to identify the physical address

Ribbon: It is the load balancer used by the spring cloud which uses round robin algorithm

Second Microservice

A wallet whose fund must be updated by calling the First Microservice

First Microservice: Instance ID is: ACCOUNT-MS

/api/first/{account}: This is a microservice that returns the account number, balance & port

Second Microservice

/api/second/wallet/{account}: This must call ACCOUNT-MS/api/first/{account}: We can use the balance amount and add to the wallet amount

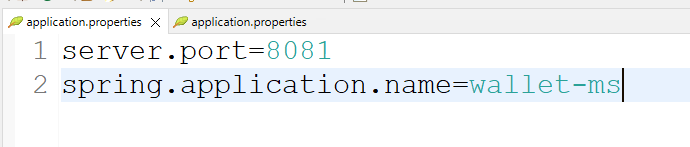
if wallet amount = 500

then balance amount = 5000, then total wallet amount = 5500

Dependency in Second Microservice

1. Web, Devtools, Eureka Client

application.properties in wallet microservice (second microservice)



Account Microservice (First Microservice) returns   
{  
 “account” : 12345, “balance”:7000, “port”: 9091   
}

Wallet Microservice must consume this data and process it as   
{  
 “account”:12345, balance: 7000, port: 9091, “walletId”:10,   
“walletAmount”:500, “total”:7500  
}

Things to create

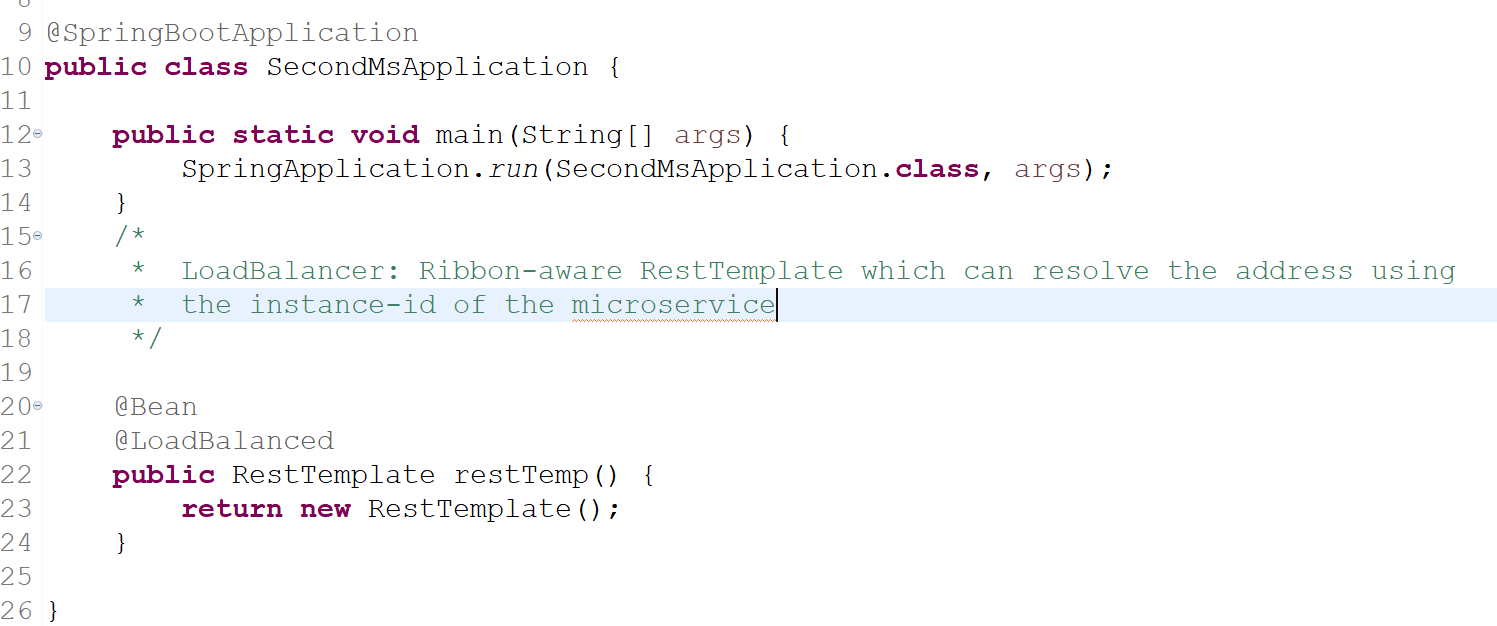
Account: representing the json data : account, balance, port

Wallet : representing walletId, walletAmount, totalAmount, Account

Creating RestTemplate with @LoadBalanced & @Bean annotation

@Bean  
@LoadBalanced  
public RestTemplate restTemp() {  
 return new RestTemplate();   
}

The above method we can write in main class

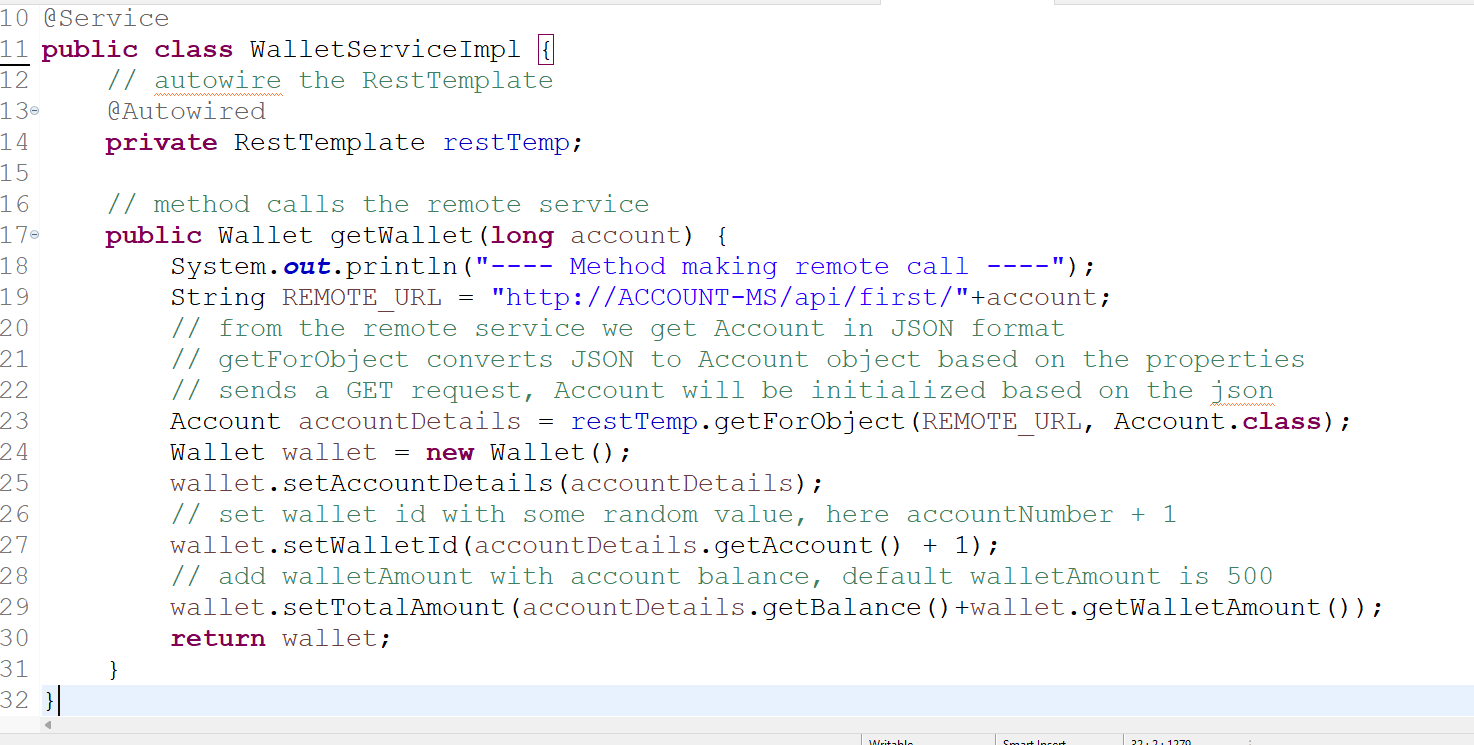


Service layer using RestTemplate to call the ACCOUNT-MS

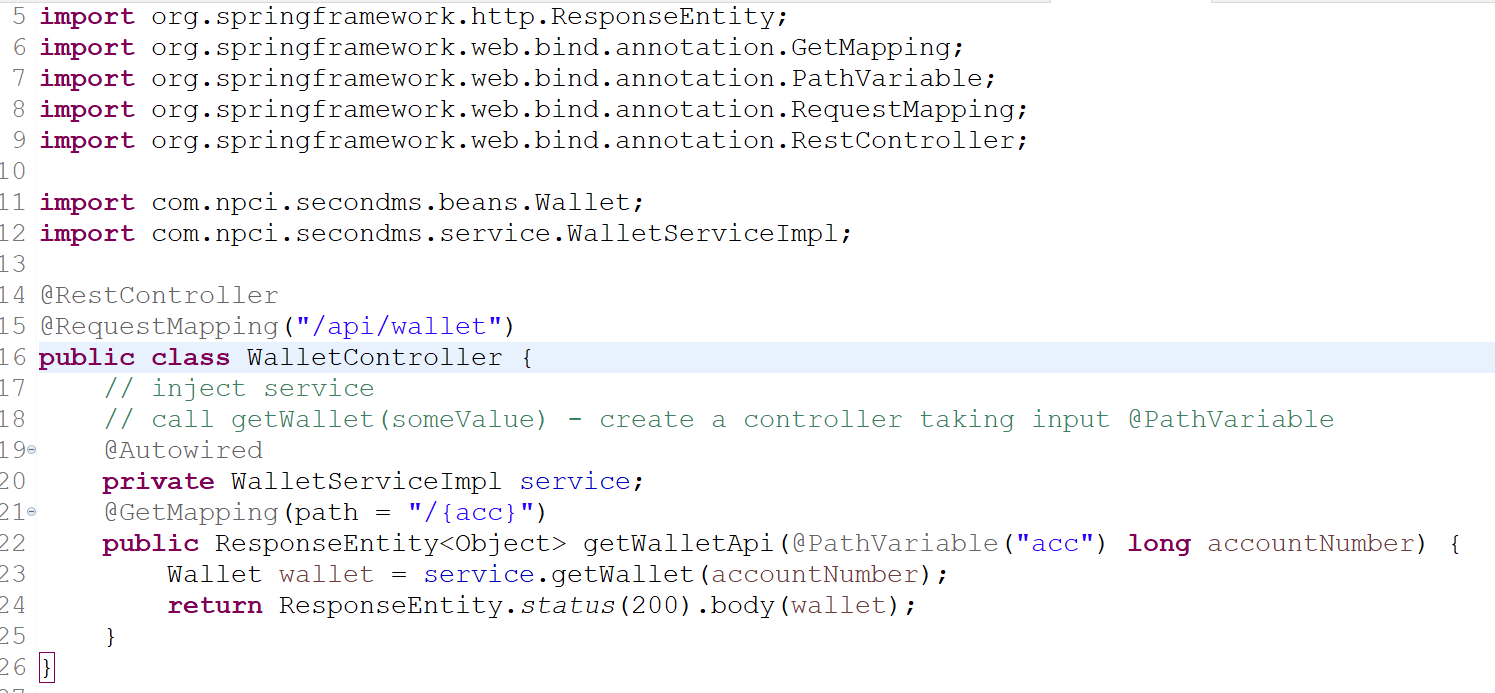
Client should not use the ACCOUNT-MS port number or ip address, client uses only the instance id that is registered in the service-discovery

i.e., <http://ACCOUNT-MS/api/first/accountNumberValue>

WalletServiceImpl.java



Controller will call the getWallet(someValue)



Things to check

1. Checking how a load balancer can route the request to multiple instances
2. Checking how a second microservice is consuming first microservice data

Feign Client:

It is used to communicate with the microservices without using RestTemplate & LoadBalancer, it internally uses Load Balancer.

* it gives a reusable interface that will have all the remote calls with HTTP mappings

@FeingClient(“http://ACCOUN-MS/api/first”)   
interface Client {   
 @GetMapping(“/{acc}”)  
 public Account getAccountDetails(@PathVariable(“acc”) long acc)  
}

@Autowired  
Client client;

To make remote calls

Account account = client.getAccountDetails(2333); // Feign Client sends an HTTP GET request to “http://ACCOUNT-MS/api/first/2333”

Library:

Open Feign: it provides @FeignClient annotation

Note: You must use @EnableFeignClients to scan all the @FeignClient annotations which makes spring boot to create a proxy implementation for the interface

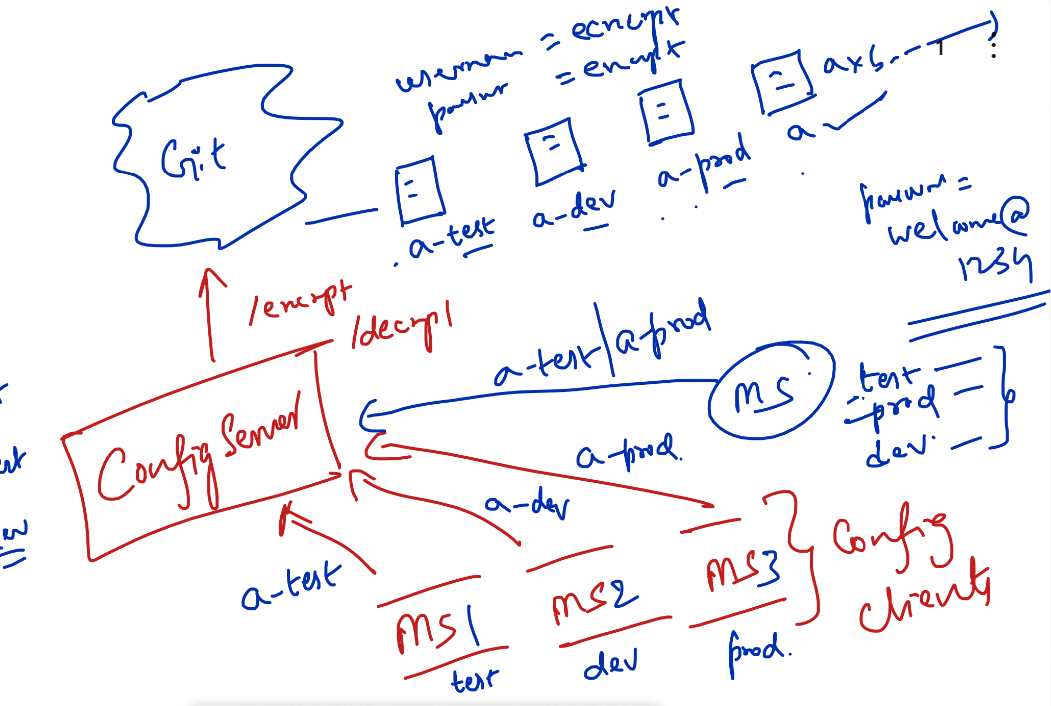
Centralized Cloud Configuration / Distributed Cloud Configuration

* Whenever multiple services wants to use common configurations you can use this and also
* When different profiles wants to load different configurations to the application then also it will be useful, like dev, test, prod are the profiles who will have different configurations based on their environment
* You can also encrypt/decrypt the sensitive informations, because they are centralized they might be accessible to other’s

Configuration Server: It is a program which can provide you the centralized cloud configuration, it takes care of

1. connecting to the centralized server
2. fetching the properties for the microservices
3. encrypting and decrypting the properties for the microservices

Configuration Client: It is a program which connects to the configuration server to pull the configurations, microservice must use this



Things to create

1. Create configuration files in GIT
   1. hello.properties
   2. hello-test.properties
   3. hello-dev.properties
   4. hello-prod.properties
   5. hello-sensitive.properties
2. Create a configuration server that connects to the GIT
   1. Dependency: ConfigServer : @EnableConfigServer
   2. application.properties: URL of the GIT, label name of the branch
3. Create a microservice/to the existing microservice configure the URL of the configuration server
   1. Dependency: Config Client
   2. application.properties: URL of the configuration server

Steps in Configuration Server

1. Creating configuration server program

Dependency: Config Server

1. Configuring application.properties to connect to the GIT

spring.cloud.config.git.uri=git-uri

spring.cloud.config.default-label=master

1. Using @EnableConfigServer: This enables your program to behave like configuration server to do all of its job like connecting to git, pulling the configurations, encrypting, decrypting

Steps in Configuration Client

1. Configuring application.properties

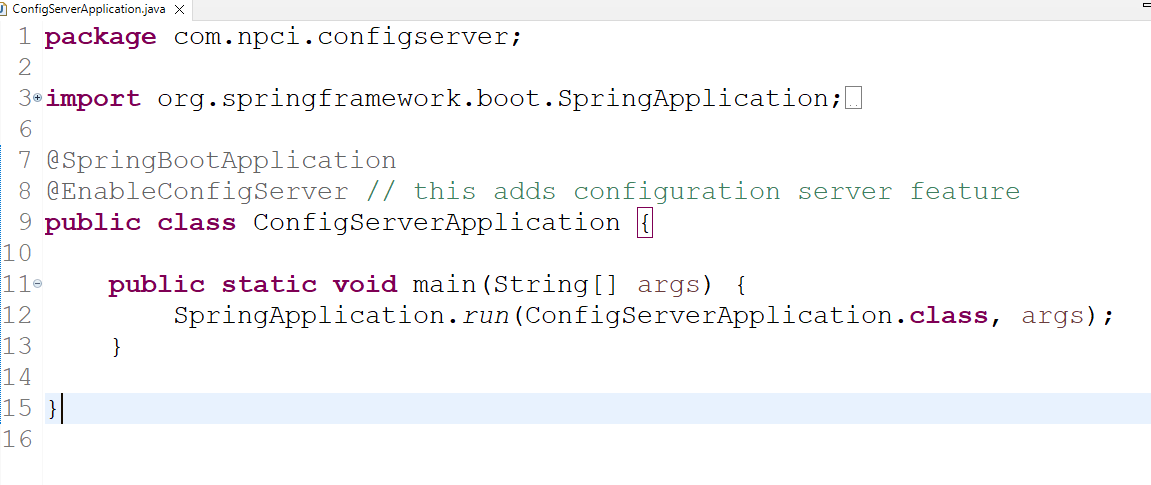
spring.application.name=hello

spring.config.import=optional:configserver:url-of-configserver

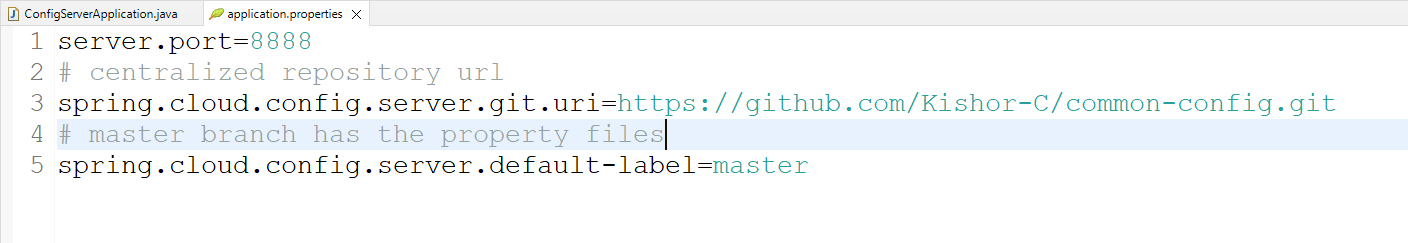
spring.profiles.active=prod

Configuration Server

Add @EnableConfigServer in main class



application.properties



Configuration client mentions the property files that to be fetched by the configuration server,

Note: a property file can have any number of properties

Since message is the property we can inject this value to a variable using @Value(“${message}”)

however you can keep datasource information’s also.

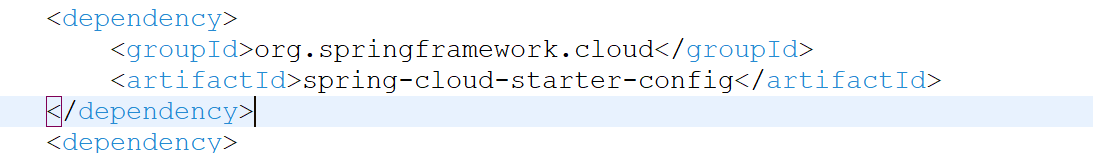
spring.datasource.username=postgres

spring.datasource.password=Welcome@123 or encrypted data

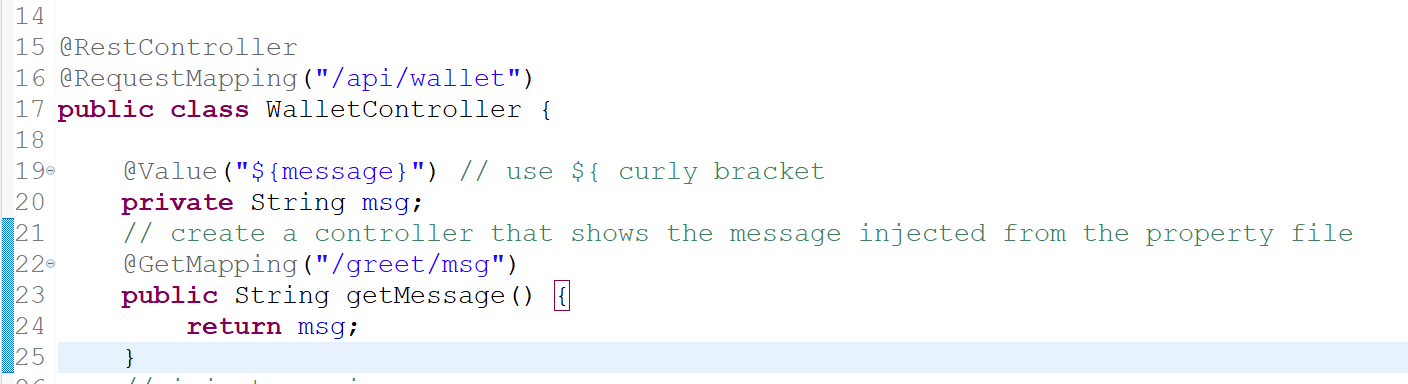
Your config client must have Data JPA to read these properties.

Library: Config Client

pom.xml



In Controller inject ${message}



application.properties



Programs to run in the order

1. Service Discovery
2. Configuration Server
3. First Microservice
4. Second Microservice

Output:

