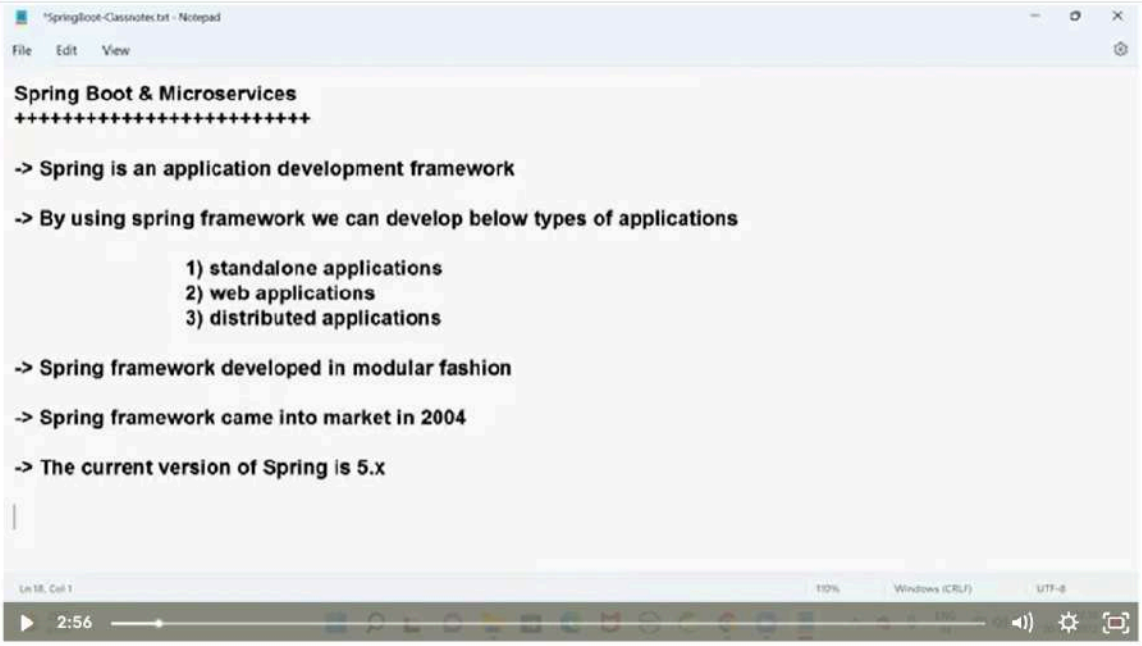


1) One To One (Ex: One Person will have one Passport)

2) One To Many (Ex : One Employee will have Multiple Addresses)

3) Many To One (Ex: Multiple Books belongs to one Author)

4) Many To Many (Ex: Multiple Users having Multiple Roles)

=> When DB tables are having Relation then we need to represent that relation in our Entity classes also.

=> The process of representing DB tables relation in Entity classes is called as Association Mapping.

Cascade Type : Default Type is NONE

Fetch Type : Default Type is LAZY

-> The current version of Spring is 5.x

-> Spring is non-invasive framework

-> Spring is versatile framework

Note: When we develop application by using spring, programmer is responsible to take care of configurations required for the application.

-> After few years Spring team realized that for every project configurations are required

-> Every Programmer dealing with configurations in the project

-> The configurations are common in the project

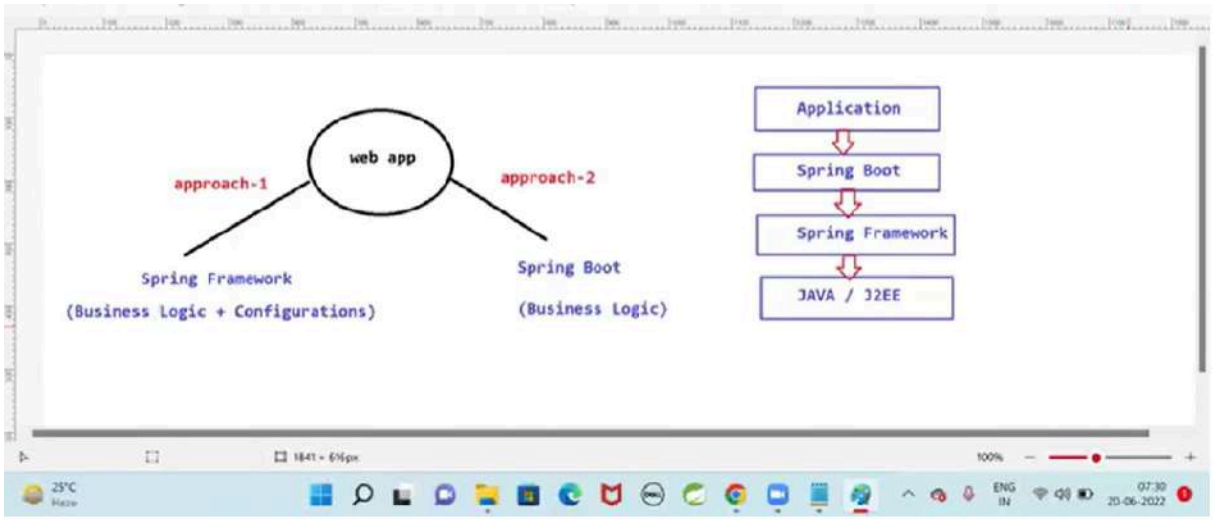
++++++++++++++++
Spring Boot
++++++++++++++++

=> It is another approach to develop spring based applications with less configurations

=> Spring Boot is an enhacemen of Spring framework

=> Spring Boot internally uses Spring framework

=> What type of applications we can develop using spring framework, same type of applications can be developed by using Spring boot also



Spring Boot = Spring Framework - XML Configurations

++++++++++++++++++++++++
Spring Boot Advantages
++++++++++++++++++++++++

1) Auto Configuration

2) POM starters

3) Embedded Servers

4) Rapid Development

5) Actuators

5) Actuators

-> Autoconfiguration means boot will identify the configurations required for our application and it will provide that configuration

- Starting IOC container
- Creating Connection Pool
- SMTP Connections
- Web Application Deployment
- Depedency Injections etc....

-> POM starters dependencies that we use to develop our application

-> Boot will provide web server to run our web applications. Those servers are called as Embedded Server

-> Rapid Development means fast development. We can focus only on business logic

-> Actuators are used to monitor and manage our application

File Edit View

Creating project using STS IDE
++++++++++++++++++++++++++

-> Go to STS IDE

-> New -> Spring Starter Project -> Fill the details and create the projet

Note: STS IDE will use start.spring.io internally to create the project

Note: TO create spring boot application internet connection is mandatory for our application.

Spring Boot Application Folder Structure
++++++++++++++++++++++++++++++++++

09-Spring-Boot-App ---------- Project Name (root folder)

- src/main/java
- src/main/resources
- src/test/java

++++++++++++++++++++++++++++++++
What is start class in Spring Boot?
++++++++++++++++++++++++++++++++

-> When we create boot application by default one java class will be created with a name Application.java i.e called as Start class of spring boot

-> Start class is the entry point for boot application execution

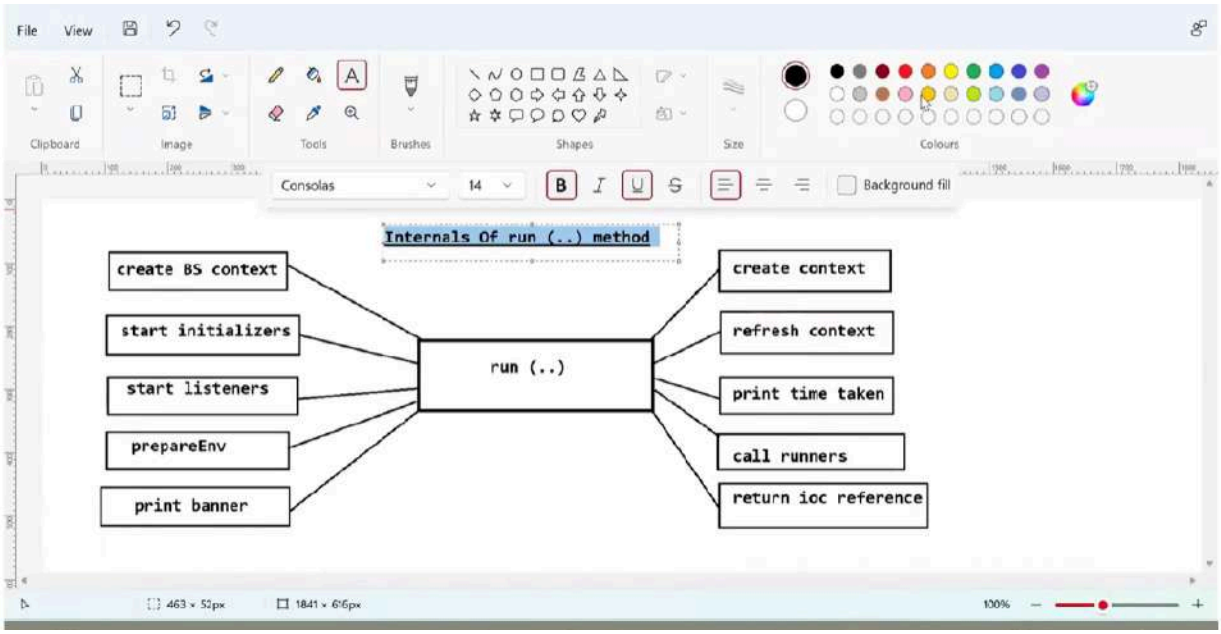
```
@SpringBootApplication
public class Application {

	public static void main(String[] args) {
		SpringApplication.run(Application.class, args);
	}
}
```

=> @SpringBootApplication annotation is equal to below 3 annotations

@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan

=> SpringApplication.run (..) method contains bootstrap logic. It is the entry point for boot application execution

- start stop watch
- start listners
- prepareEnv
- print Banner
- create IOC container
- refresh Container
- stop that stop watch
- print time taken to start application
- call runners
- return IOC reference



-> In Boot application console we can spring logo that is called as Banner

++
How ioc container will start in Spring Boot ?
++

=> For boot-starter, run method using "AnnotationConfigApplicationContext" class to start IOC container

=> For boot-starter-web, run ( ) method using "AnnotationConfigServletWebServerApplicationContext" to start IOC

=> For starter-webflux, run ( ) method using "AnnotationConfigReactiveWebServerApplicationContext" to start IOC

-> boot-starter is used to create standalone applications

-> boot-starter-web dependency is used to develop web applications

-> boot-starter-webflux is dependency is used to develop applications with Reactive Programmin

++++++++++++++++++++
Runners in Spring Boot
++++++++++++++++++++

=> Runners are getting called at the end of run ( ) method

=> If we want to execute any logic only once when our application got started then we can use Runners concept

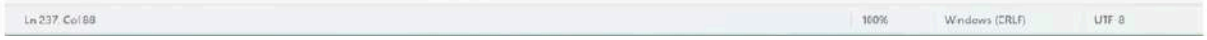
=> In Spring Boot we have 2 types of Runners

1) ApplicationRunner
2) CommandLineRunner

=> Both runners are functional interface. They have only one abstract method i.e run ( )

Use cases
---

1) send email to management once application started
2) read data from db tables and store into cache memory



++++++++++++++++++
Banner in Spring Boot
++++++++++++++++++

-> In Boot application console we can see spring logo that is called as Banner in Spring Boot
-> We can customize banner text by creating "banner.txt" file under src/main/resources
-> We can keep our company or project name as Banner text in Ascii format
-> Spring Boot banner having below 3 modes

1) console ---- > it is default mode
2) log
3) off

Note: If we set banner mode as off then banner will not printed

++++++++++++++++++++++++++++++++++++

```
@Component
public class CacheManager implements ApplicationRunner {

	@Override
	public void run(ApplicationArguments args) throws Exception {
		System.out.println("Logic executing to load data into cache....");
	}
}

@Component
public class SendAppStartMail implements CommandLineRunner {

	@Override
	public void run(String... args) throws Exception {
		System.out.println("logic executing to send email....");
	}
}

Note: @Component annotation is used to represent our java class as Spring Bean
```


```
++++++++++++++++++++++++++++
What is @SpringBootApplication ?
++++++++++++++++++++++++++++

-> This annotation used at start class of spring boot

-> This annotation is equal to below 3 annotations

	1) @SpringBootConfiguration
	2) @EnableAutoConfiguration
	3) @ComponentScan
```

NOTE -: @Component are used to represent as a bean class means if one class are bean then object creation are handle by IOC container now as we in spring framework to represent class as a bean we need to provide configuration in Xml file separately but in spring boot as it provide autoconfiguration so through this it will automatically configure.is that now object creation handle by IOC now if we will not use component then object will not create for that class.

```
What is ComponentScan ?
++++++++++++++++++++

-> The process of scanning packges in the application to identify spring bean classes is called as Component Scan

-> Component Scan is built-in concept

-> Component Scanning will start from base package

Note: The package which contains start class is called as base package.

-> After base packge scanning completed it will scan sub packages of base package.

Note: The package names which are starting with base package name are called as sub packages.

		in.ashokit    (base package)
		in.ashokit.dao
		in.ashokit.service
		in.ashokit.config
		in.ashokit.rest
		in.ashokit.util
```

```
                        in.ashokit.dao
                        in.ashokit.service
                        in.ashokit.config
                        in.ashokit.rest
                        in.ashokit.util
                        com.wallmart.security ----> This is not sub package so scanning will not happen
```

-> We can specify more than one base package also in our boot start class

```
@SpringBootApplication
@ComponentScan(basePackages = { "in.ashokit", "com.wallmart" })
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Note : as we are creating service class repository class component so who is creating object for those classes ,who is starting the ioc (in Spring its through Application Context) in spring boot .whenever we start our bootapplication this run method get exicuted and run method staring IOC container , IOC container doing this component scanning and identify the bean class object

Note: It is highly recommended to follow proper package naming conventions

Ex: companyDomainName.projectName.moduleName.layerName

com.tcs.passport (basePkgName)
com.tcs.passport.user.dao
com.tcs.passport.user.service

++++++++++++++++++++++++
What is @Bean annotation
++++++++++++++++++++++++

-> @Bean is a method level annotation

-> When we want to customize any object creation then we will use @Bean annotation

```
@Configuration
public class AppConfig {

    @Bean
    public AppSecurity createInstance() {
        AppSecurity as = new AppSecurity();
        // custom logic to secure our functionality
        return as;
    }
}
```

How to represent java class as a Spring Bean?
++

@Component
@Service
@Repository

@Controller
@RestController

@Configuration
@Bean

-> All the above annotations are class level annotations but @Bean is method level annotation

-> @Component is a general purpose annotation to represent java class as Spring Bean

-> @Service annotation is a specialization of @Component annotation. This is also used to represent java class as spring bean. It is allowing for implementation classes to be autodetected through classpath scanning.

Note: For business classes we will use @Service annotation

-> @ Repository annotation is a specialization of @Component annotaton. This is also used to represent java class as spring bean. It is having Data Access Exception translation

Note: For dao classes we will use @Repository

-> In Web applications to represent java class as controller we will use @Controller annotation. It is used for C2B communication.

-> In Distributed application to represent java class as distributed component we will use @RestController. It is used for B2B communication.

-> If we want to perform customized configurations then we will use @Configuration annotation along with @Bean methods. Based on requirement we can have multiple @Configuration classes in the project.

Ex: Swagger Config, DB Config, RestTemplate Config, Kafka Config, Redis Config, Security Config etc...

Note: @Bean annotated method we can keep in any spring bean class but it is highly recommended to keep them in @Configuration classes.

Autowiring
+++++++++

-> Autowiring is used to perform dependency injection

-> The process of injecting one class object into another class object is called as dependency injection.

-> In Spring framework IOC container will perform dependency injection

-> We will provide instructions to IOC to perform DI in 2 ways

1) Manual Wiring (using ref attribute in beans config xml file)

2) Autowiring

-> Autowiring means IOC will identify dependent object and it will inject into target object

-> Autowiring will use below modes to perform Dependency Injection

1) byName
2) byType
3) constructor (internally it will use byType only)

-> To perform Autowiring we will use @Autowired annotation

-> @Autowired annotation we can use at 3 places in the program

1) variable level (Field injection - FI)
2) setter method level (Setter Injection - SI)
3) constructor (Constructor Injection - CI)

Spring Data JPA
++++++++++++++

-> It is used to develop persistence logic in our application

-> The logic which is responsible to communicate with database is called as Persistence Logic

-> Already we have JDBC, Spring JDBC, Hibernate, Spring ORM to develop persistence logic

-> If we use JDBC,or Spring JDBC or Hibernate or Spring ORM then we should common logics in all DAO classes to perform CURD Operations

-> Data JPA simplified persistence logic development by providing pre-defined interfaces with methods

-> Data JPA provided Repository methods to perform CURD operations

Note: If we use Data JPA then we don't need to write logic to perform curd operations bcz Data JPA will take care of that

Note: If we use Data JPA then we don't need to write logic to perform curd operations bcz Data JPA will take care of that

-> Data JPA provideed Repository interfaces to perform CURD operations

1) CrudRepository (Methods to perform Crud Operations)

2) JpaRepository (Methods to perform Crud Operations + Pagination + Sorting + QBE)

Hibernate Vs Data JPA
++++++++++++++++++

-> In Hibernate we should implement all methods to perform CRUD operations
-> Data JPA providing predefined methods to perform CRUD Operations

-> In Hibernate we should boiler plate code (same code in multiple classes)
-> In Data JPA we don't need to write any method because JPA Repositories providing methods for us

Environment Setup
++++++++++++++++++

1) MySQL Database (Server s/w)
2) MySQL Workbench (Client s/w)

Entity Class
++++++++++

-> The java class which is mapped with DB table is called as Entity class

-> To map java class with DB table we will use below annotations

@Entity : It represents our class as Entity class. (It is mandatory annotation)

@Table : It is used to map our class with DB Table name

Note : @Table is optional if our class name and table is same. If we don't give @Table then it will consider class name as table name.

@Id : It represents variable mapping with primary column in table (It is mandatory annotation)

@Column : It is used to map our class variables with DB table column names

Note: @Column is optional if class variable name and DB table column names are same. If we don't give @Column then it will consider variable name as column name.

++++++++++++++++++++
Repository Interfaces
++++++++++++++++++++

-> JPA provided repository interfaces to perform Curd Operations

-> For Every DB table we will create one Repository interface by extending Jpa Repository

Syntax:

```
public interface PlayerRepository extends CrudRepository<Entity, ID>{
}
```

Example

```
public interface PlayerRepository extends Repository<Entity, ID>{
}
```

Note: When our interface extending properties from JPA Repository interfaces then JPA will provide implementation for our interface in Runtime using Proxy Design Pattern.

++++++++++++++++++++
Datasource properties
++++++++++++++++++++

-> Datasource properties represents with which database we want connect

- DB URL
- DB Uname
- DB Pwd
- DB Driver Class

-> We will configure datasource properties in application.properties file or application.yml file

```
spring.datasource.url=jdbc:mysql://localhost:3306/
spring.datasource.username=ashokit
spring.datasource.password=AshokIT@123
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

++++++++++++++
ORM Properties
++++++++++++++

-> Hibernate provided some additional benefits while developing persistence logic

-> Tables can be created dynamically using "auto-ddl" property

-> We are calling JPA methods to perform DB operations. Those methods will generate queries to execute. To print those queries on console we can use 'show-sql' property

++++++++++++++++++++++++++
Build First App using Data JPA
++++++++++++++++++++++++++

1) Create spring starter application with below dependencies

    a) starter-data-jpa
    b) mysql-connector

2) Create Entity class using Annotations

3) Create Repository interface by extending CrudRepository

4) Call Repository methods in start class to perform DB operations

CrudRepository methods
++++++++++++++++++++

1) save (.) :: upsert method (insert and update) - for one record

2) saveAll (..) :: upsert method (insert and update) - for multiple records

3) findById (.) :: To retrieve record using primary key

4) findAllById (..) :: To retrieve multiple records using primary keys

5) findAll ( ) :: To retrieve all records from table

6) count ( ) :: To get total records count

7) existsById (.) : To check record presence in table using Primary key

8) deleteById (.) : To delete single record using primary key

9) deleteAllById (..) : To delete multiple records using primary keys

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {

	@Id
	@Column(name = "USER_ID")
	private Integer userid;

	@Column(name = "USER_NAME")
	private String username;

	@Column(name = "USER_GENDER")
	private String gender;

	@Column(name = "USER_AGE")
	private Integer age;

	@Column(name = "USER_COUNTRY")
	private String country;

}
```

```java
public interface UserRepository extends CrudRepository<User, Integer> {

}
```

```
        }*/

        /*Iterable<User> allById = repository.findAllById(Arrays.asList(101,102,103));
        allById.forEach(user -> {
            System.out.println(user);
        });*/

        /*Iterable<User> findAll = repository.findAll();
        findAll.forEach(user -> {
            System.out.println(user);
        });*/

        /*long count = repository.count();
        System.out.println("Total Records in table :: "+ count);*/

        /*boolean existsById = repository.existsById(101);
        System.out.println("Record Presence with id - 101 :: " + existsById);*/

        //repository.deleteById(104);

        repository.deleteAllById(Arrays.asList(102,103));
    }
```

```
spring.datasource.url=jdbc:mysql://localhost:3306/sbms
spring.datasource.username=ashokit
spring.datasource.password=AshokIT@123
spring.datasource.driver-class-name=com.mysql.jdbc.Driver


spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true
```

-> In CrudRepository interface we have methods to retrieve records

findById (ID) ----> to retrieve based on primary key

findAllById(Iterable<ID> ids) ----> to retrieve based on multiple primary keys

findAll( ) -----> to retrieve all records

Requirement : Retrieve records based on USER_COUNTRY column data

=> Retrieve user records who are belong to INDIA

Requirement -1 : Retrieve users records who are belongs to INDIA
Requirement -2 : Retrieve users whose age is below 30 years

Note: user_age and user_country are non-primary columns in the table

-> To retrieve data based on Non-Primary key columns we don't have pre-defined methods. To implement these kind of requirements we have below 2 options

1) findby methods

2) custom queries

Find By Methods

++++++++++++++++

-> Find By Methods are used to construct queries based on method name

-> Method Name is very important to prepare query dynamically

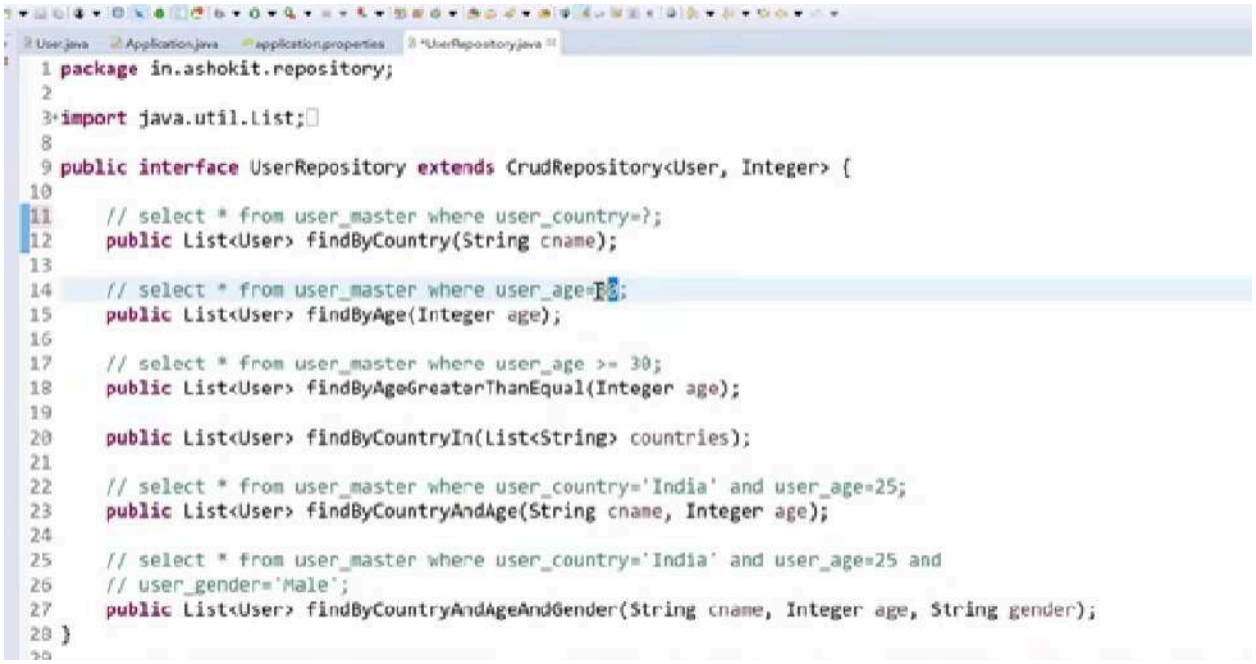
-> Based on our method name, JPA will prepare the query in Runtime and it will execute that

Ex: Retrieve user data based on country_name

```
findbycountry(String countryname);
```

Ex : Retrieve user data based on user_age

```
findbyage(Integer age);
```



```java
package in.ashokit.repository;

import java.util.List;

public interface UserRepository extends CrudRepository<User, Integer> {

    // select * from user_master where user_country=?;
    public List<User> findByCountry(String cname);

    // select * from user_master where user_age=?;
    public List<User> findByAge(Integer age);

    // select * from user_master where user_age >= 30;
    public List<User> findByAgeGreaterThanEqual(Integer age);

    public List<User> findByCountryIn(List<String> countries);

    // select * from user_master where user_country='India' and user_age=25;
    public List<User> findByCountryAndAge(String cname, Integer age);

    // select * from user_master where user_country='India' and user_age=25 and
    // user_gender='Male';
    public List<User> findByCountryAndAgeAndGender(String cname, Integer age, String gender);
}
```

```java
UserRepository repository = context.getBean(UserRepository.class);

/*List<User> findbycountry = repository.findByCountry("INDIA");

findbycountry.forEach(user -> {
    System.out.println(user);
});*/

/*List<User> findByAge = repository.findByAge(30);
findByAge.forEach(user -> {
    System.out.println(user);
});*/

//List<User> list = repository.findByAgeGreaterThanEqual(30);

//List<User> list = repository.findByCountryIn(Arrays.asList("INDIA", "USA"));

//List<User> list = repository.findByCountryAndAge("India", 25);

List<User> list = repository.findByCountryAndAgeAndGender("India", 25, "Male");

list.forEach(user -> {
```

In repository class we will create query method and in main class we can call all these custom queries.

++++++++++++++++
Custom Queries
++++++++++++++++

-> We can write our own query and we can execute that query using JPA that is called as Custom Query

-> Custom Queries we can write in 2 ways

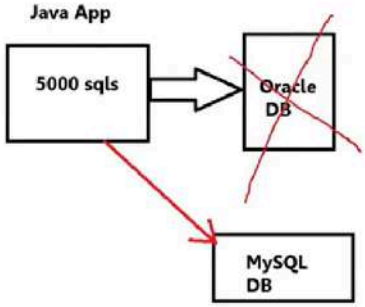
1) HQL Queries

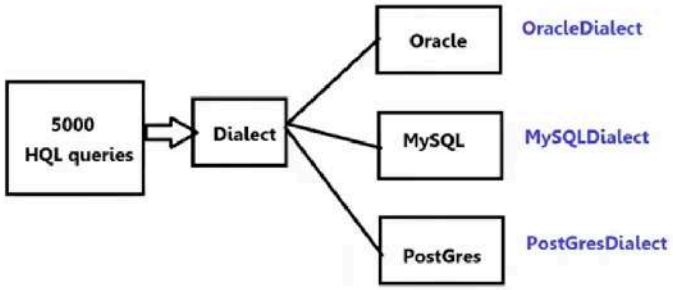
2) Native SQL Queries

Native SQL Vs HQL
++++++++++++++++

-> Native SQL queries are DB dependent
-> HQL Queries are DB in-dependent

-> Native Queries will use table name and column names in the query directley
-> HQL queris will use Entity Class Name and Entity Class Variable Names in the query





Native SQL Vs HQL
++++++++++++++++

-> Native SQL queries are DB dependent
-> HQL Queries are DB in-dependent

-> Native Queries will use table name and column names in the query directley
-> HQL queris will use 'Entity Class Name' and 'Entity Class Variable Names' in the query

-> Native SQL queries will execute in Database Directley
-> HQL queries can't execute in DB directley (HQL queries will be converted to SQL query using Dialect class before execution)

-> As SQL queries are executing directley, performance wise they are good. HQL queries should be converted before execution hence HQL queries will take more time than SQL queries.

-> Performance wise SQL queries are good
-> Maintenence wise HQL queies are good

Retrieve all records from table

SQL : SELECT * FROM USER_MASTER

HQL : From User

Retrieve all records of users who are belongs to country 'India'

SQL : SELECT * FROM USER_MASTER WHERE USER_COUNTRY = 'INDIA'

HQL : From User where country='India'

Retrieve users who are belongs to 'India' and age is 25

SQL : SELECT * FROM USER_MASTER where USER_COUNTRY='INDIA' AND USER_AGE = 25

HQL : From User where country='India' and age = 25

++++++++++++++++++
JpaRepository
++++++++++++++++++

-> It is a predefined interface available in data jpa

-> Using JpaRepository interface also we can perform CRUD operations with DB tables

-> JpaRepository also having several methods to perform CRUD operations

-> JpaRepository having support for Pagination + Sorting + QBE (Query By Example)

Note: CrudRepository interface doesn't support Pagination + Sorting + QBE

Sorting
++++++++
-> Sorting is used to sort the records either in ascending or in descending order

-> We can pass Sort object as parameter for findAll ( ) method like below

```
List<User> users = repository.findAll();
List<User> users = repository.findAll(Sort.by("age").ascending());
List<User> users = repository.findAll(Sort.by("username","age").descending());

users.forEach(user -> {
        System.out.println(user);
});
```

++++++++++
Pagination
++++++++++

-> The process of dividing all the records into multiple pages is called as Pagination

-> If we retrieve all the records at once then performance issues we will get in the

-> When we have lot of data in table then we will divide those records into multiple pages and we will display

-> Data will be displayed based on below 2 conditions

=> PAGE NUMBER (User landed on which page)

=> PAGE SIZE (How many records should be displayed in single page)

-> Data will be displayed based on below 2 conditions

=> PAGE NUMBER (User landed on which page)

=> PAGE SIZE (How many records should be displayed in single page)

```
int pageSize = 3;
int pageNo  = 0;   (it will come from UI)

PageRequest pageRequest = PageRequest.of(pageNo, pageSize);

Page<User> pageData = repository.findAll (pageRequest) ;

int totalPages = pageData.getTotalPages();
System.out.println("Total Pages :: "+ totalPages);

List<User> users = pageData.getContent();
users.forEach(user -> {
      System.out.println(user);
});
```

++++++++++++++++++++++++
Query By Example (QBE)
++++++++++++++++++++++

-> It is used to prepare the query dynamically

-> To implement Dynamic Search Option we can use QBE concept

```
User entity  = new User();

entity.setCountry("India");
entity.setAge(25);

Example<User> example = Example.of(entity);

List<User> users = repository.findAll(example);

users.forEach(user -> {
      System.out.println(user);
});
```

++++++++++++++++++++++++
Timestamping in Data JPA
++++++++++++++++++++++++

-> For every table we need to maintain below 4 columns to analyze data

CREATED_DATE
CREATED_BY

UPDATED_DATE
UPDATED_BY

Note: In realtime we will maintain these 4 columns for every table

-> CREATED_BY & UPDATED_BY represents which user creating and updating records in table. In Web applications we will use logged in user data and we will set for these 2 columns.

```
public class Product {

	@Id
	@Column(name = "PRODUCT_ID")
	private Integer pid;

	@Column(name = "PRODUCT_NAME")
	private String pname;

	@Column(name = "PRODUCT_PRICE")
	private Double price;

	@CreationTimestamp
	@Column(name = "CREATED_DATE", updatable = false)
	private LocalDateTime createdDate;

	@UpdateTimestamp
	@Column(name = "UPDATED_DATE", insertable = false)
	private LocalDateTime updatedDate;

}
```

-> updatable = false means that column value should not updated when we do update operation on the table.

-> insertable = false means that column value should not inserted when we do insert operation on the table.

++

Q) Why to use Wrapper classes instead of Primitive datatypes in Entity class ?

-> Primitive data types will take default values when we dont set the value for variable

Ex : if we take 'price variable with double' data type then it will insert '0' if we don't set price for the record. It is not recommended.

-> If we take wrapper classes it consider 'null' as default value when we don't set value for a variable.

++

++++++++++++
Primary Keys
++++++++++++

-> Primary Key is a constraint (rule)

-> Primary constraint is the combination of below 2 constraints

1) UNIQUE

2) NOT NULL

-> When we use PRIMARY KEY constraint for a column then that column value shouldn't be null and it should be unique.

-> It is not recommended to set values for Primary Key columns manually

-> We will use Generators to generate the value for primary key column

```
@Id
@Column(name = "PRODUCT_ID")
@GeneratedValue(strategy=GenerationType.IDENTITY)
private Integer pid;
```

-> In Oracle database we will use sequence concept to generate value for primary key column. For every primary key one sequence will be created in database

Ex:

```
create sequence product_id_seq
start with 101
increment by 1;
```

-> If we want to generate primary key column value like below then we should go for Custom Generator (Our own Generator we have to develop)

TCS01
TCS02
TCS03
TCS04

++++++++++++++++++++
Composite Primary Keys
++++++++++++++++++++

-> Table can have more than one primary key column

-> If table contains more than one primary key column then those primary keys are called as Composite Primary Keys

```
CREATE TABLE ACCOUNTS
(
  ACC_ID              NUMBER,
  ACC_NUMBER          NUMBER,
  HOLDER_NAME         VARCHAR2(50)
  ACC_TYPE            VARCHAR2(10),
  BRANCH_NAME         VARCHAR2(10)

  PRIMARY KEY (ACC_ID, ACC_NUMBER, BRANCH_NAME)
```

```
EmpRepository empRepository = context.getBean(EmpRepository.class);
AddressRepository addrReposiotry = context.getBean(AddressRepository.class);

Employee e = new Employee();
e.setEmpName("Raja");
e.setEmpSalary(4000.00);

Address a1 = new Address();
a1.setCity("Hyd");
a1.setState("TG");
a1.setCountry("India");
a1.setEmp(e);

Address a2 = new Address();
a2.setCity("GNT");
a2.setState("AP");
a2.setCountry("India");
a2.setEmp(e);

// setting addresses to emp
List<Address> addrList = Arrays.asList(a1, a2);
```

```
@Entity
@Data
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer personId;

    private String personName;

    private String personGender;

    @OneToOne(mappedBy = "person", cascade = CascadeType.ALL)
    private Passport passport;

}
```

```
public class Passport {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer passportId;

    private String passportNum;

    private LocalDate issuedDate;

    private LocalDate expiryDate;

    @OneToOne
    @JoinColumn(name = "person_id")
    private Person person;

}

@SpringBootApplication
public class Application {
```


```
public class Application {
```

```
}

@SpringBootApplication
public class Application {

	public static void main(String[] args) {
		ConfigurableApplicationContext context = SpringApplication.run(Application.class, args);

		PersonRepository personRepo = context.getBean(PersonRepository.class);

		PassportRepository passportRepo = context.getBean(PassportRepository.class);

		/*Person person = new Person();
		person.setPersonName("Ashok");
		person.setPersonGender("Male");
		person.setPersonName("Ashok");
		person.setPersonGender("Male");
```

```
/*Person person = new Person();
person.setPersonName("Ashok");
person.setPersonGender("Male");

Passport passport = new Passport();
passport.setPassportNum("KV7979HKI");
passport.setIssuedDate(LocalDate.now());
passport.setExpiryDate(LocalDate.now().plusYears(10));

person.setPassport(passport);
passport.setPerson(person);

personRepo.save(person);*/

//personRepo.findById(1);

//personRepo.findById(1);
```

```
        /*Person person = new Person();
        person.setPersonName("Ashok");
        person.setPersonGender("Male");

        Passport passport = new Passport();
        passport.setPassportNum("KV7979HKI");
        passport.setIssuedDate(LocalDate.now());
        passport.setExpiryDate(LocalDate.now().plusYears(10));

        person.setPassport(passport);
        passport.setPerson(person);

        personRepo.save(person);*/

        //personRepo.findById(1);

        personRepo.deleteById(1);

    }

}
```

Done spring boot now next will start rest Apis and microservices.