SPRING BOOT

What is Spring boot?
Spring Boot is a brand new framework from the team at **Pivotal**, designed to simplify the bootstrapping and development of a new Spring application.
The framework takes an opinionated approach to configuration, freeing developers from the need to define boilerplate configuration
Spring Boot is a project built on the top of the Spring framework. It provides a simpler and faster way to set up, configure, and run both simple and web-based applications.

Without spring boot the problems are:

1. We need to hunt for all the compatible libraries for the specific spring version and add them.
2. Most of the times we have to configure **Datasource**, **JDBC Template**, **Transaction manager**, **DispatcherServlet**, **Handler Mapping**, **View Resolver**, etc beans in the same way.
3. We should always deployed in external server.
4. The problem with spring component-scanning and autowiring is that it's hard to see how all of the components in an application are wired together.

With Spring boot the advantages are:

1. **Starters** help easy dependency management.
2. **Auto configuration** for most of the commonly used built-in classes such **Datasource**, **JDBC Template**, **Transaction manager**, **DispatcherServlet**, **Handler Mapping**, **View Resolver**, using customisable properties. We need to enable auto configuration by adding either @EnableAutoConfiguration or @SpringBootApplication.
3. **Embedded Server**:
   The spring-But-started-web automatically pulls spring-boot-starter-tomcat which starts tomcat as an embedded server. So we don't have to deploy our application on any externally installed tomcat server.
4. **Actuators**
   The actuators let us look inside of our bean dependencies, auto config details, environment variables, configuration properties, memory usage, garbage collection, web requests, and data source usage.

Note:

a. Spring boot increases the speed of development because of starters and auto configuration.
b. One of the great out comes of spring boot is that it almost eliminates the need, to have traditional XML configurations.

Spring boot newly added in spring 4 and offers following main features:

Spring boot starters

Without spring boot, we need to configure all the dependencies required in our pom.xml.
Spring boot starter **aggregates** common groupings of dependencies into single **starter dependency** that can be added to a project Maven or Gradle build.

#pom.xml

```
<project>
   <dependencies>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-jdbc</artifactId>
         <version>1.9.1.RELEASE</version>
      </dependency>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-test</artifactId>
         <version>1.9.1.RELEASE</version>
      </dependency>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-web</artifactId>
         <version>1.9.1.RELEASE</version>
      </dependency>
   </dependencies>
</project>
```

Note: The list of starters found in <artifactId> spring-boot-starters</artifactId>

```
<modules>
        <module-name>spring-boot-starter</module-name>
        <module-name>spring-boot-activemq</module-name>
        <module-name>spring-boot-amqp</module-name>
        <module-name>spring-boot-aop</module-name>
```

```
        <module-name>spring-boot-jdbc</module-name>
        <module-name>spring-boot-test</module-name>
        <module-name>spring-boot-data-jpa</module-name>
        <module-name>spring-boot-starter-data-web</module-name>
        …
</modules>
```

The parent element is one of the interesting aspects in the pom.xml file.

```
<parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>1.9.1.RELEASE</version>
 </parent>
```

The advantage of using the spring-boot-starter-parent pom.xml file is that developers need not worry about finding the right compatible versions of different libraries such a Spring, Jersey, JUnit, Hibernate, Jackson, and so on. The jar versions are defined in <artifactId> **spring-boot-dependencies**</artifactId>
The **snapshot** of some of the properties as shown as follows:

```
<properties>
        <!—Dependency version -- >
        <activemq.version>5.14.5</ activemq.version>
        <antlr2.version>2.7.7</antlr2.version>
        <appengine-sdk.version>1.9.51</appengine-sdk.version>
        <tomcat.version>8.5.14</tomcat.version>
        <hikaricp.version>2.5.1</hikari.version>
        <commans-dbcp.version>1.4</commans.dbcp.version>
        <commans-dbcp2.version>2.1.1</commans-dbcp.version>

        <hibernate.version>5.0.12.FINAL</hibernate.version>
</properties>
        …
```

Below is the modified pom.xml file after adding parent starter:

```
<project>
   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>1.5.9.RELEASE</version>
   </parent>
```

```
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
  </dependencies>
</project>
```

Auto Configuration

Spring boot use this **convention over configuration** by scanning the dependent libraries available in the class path. For each ***spring-boot-starter-**** dependency in the POM file, spring boot executes a default ***AutoConfiguration*** classes**.**

Spring boot provides '**spring-boot-autoconfigure'** module (spring-boot-autoconfigure-<version>.jar) which contains many configuration classes to autoconfigure beans. The above jar file contains META-INF/spring.factories file which contains list of auto configure classes.

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration,\
org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\
jpaBaseConfiguration#transactionManager,\
jpaBaseConfiguration#jpaVendoreAdapter
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
…
```

Since spring boot provides many autoconfigure classes hence reduces the complexity of configuration. Spring boot auto-configuration is a runtime (more accurately,

application start up-time) process that considers several factors to decide what Spring configuration should and should not be applied. For example, the spring's jdbcTemplate available on the class path? If so and if there is a DataSource bean, then auto-configure a jdbcTemplate bean.

Example:
@Configuration
@ConditionalOnClass({**DataSource.class**,**JdbcTemplate.class**})
public class JdbcTemplateAutoConfiguration{ }

The **@EnableAutoConfiguration** enables the magic of auto configuration

Spring-Boot checks tomcat-jdbc (default), HikariCP, Commons DBCP and Common DBCP2 in this case sequence order i.e.,. Spring boot checks the availability of the following data source classes and uses the first one that is available in class path.

1. org.apache.tomcat.jdbc.pool.DataSource
2. com.zaxxer.hikari.HikariDataSource
3. org.apache.commons.dbcp.BasicDataSource
4. org.apache.commons.dbcp2.BasicDataSource

The starter-jdbc-automatically pulls tomcat-jdbc-{version}.jar

```
<dependency>
        <groupId>org.springframework.boot </groupId>
        <artifactId>spring-boot-starter-jdbc </artifactId>
</dependency>
```

We need to exclude tomcat-jdbc from classpath if we want to use other datasources.

```
<dependency>
  <groupId>org.springframework.boot </groupId>
  <artifactId>spring-boot-starter-jdbc </artifactId>
  <exclusions>
      <exclusion>
          <groupId>org.apache.tomcat</groupId>
          < artifactId>tomcat.jdbc</artifactId>
      </exclusion>
  </exclusions>
</dependency>
```

We ned to add following dependency to use Hikari datasource.

```
<dependency>
        <groupId> com.zaxxer </groupId>
        <artifactId>HikariCP </artifactId>
```

```
</dependency>
```

We need to add following dependency to use Commons-dbcp.

```
<dependency>
        <groupId> commons.-dbcp </groupId>
        <artifactId>commons-dbcp </artifactId>
</dependency>
```

We need to add following dependency to use Commons-dbcp2.

```
<dependency>
        <groupId> org.apache.commons </groupId>
        <artifactId>commons-dbcp2 </artifactId>
 </dependency>
```

Spring boot automatically configure above data sources based on corresponding jar file present in class path and connection properties should be configured in application.properties file. Various properties can be specified inside our **application.properties/application.yml** file. This section provides a list of common Spring Boot properties and references to the underlying classes that consumes them.

```
#src/main/resources/application.properties
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@172.16.238.128:1521:orcl
spring.datasource.username=checkinuser
spring.datasource.password=checkinuser

#tomcat-connection settings
#spring.datasource.tomcat.initialSize=20
#spring.datasource.tomcat.max-active=25

# Hikari settings
#spring.datasource.hikari.maximum-pool-size=20

# dbcp settings
#spring.datasource.dbcp.initial-size=7
#spring.datasource.dbcp.max-active=20

# dbcp2 settings
#spring.datasource.dbcp2.initial-size=7
#spring.datasource.dbcp2.max-total=20
```

```
#src/main/resources/application.yml
spring:
  datasource:
    driver-class-name: oracle.jdbc.driver.OracleDriver
    url:  jdbc:oracle:thin:@localhost:1521:xe
    username: system
    password: manager
    tomcat:
      initialSize:  20
      max-active:  25
```

If we want use other than above data sources then we need to configure explicitly.

```
<dependency>
                    <groupId> c3p0 </groupId>
            <artifactId>c3p0</artifactId>
            <version>0.9.1.2</version>
 </dependency>
```

At any point we can start to define our own configuration to replace specific part of the auto -configuration.

```
#connection.properties
jdbc.driverClassName =com.mysql.jdbc.Driver
jdbc.url = jdbc:mysql://localhost:3306/test
jdbc.username = root
jdbc.password = root1234
jdbc.initPoolSize=15
jdbc.maxPoolSize=25

@Configuration
@PropertySource("classpath:application.properties")
@EnableTransactionManagement
public class AppConfig {

   @Autowired
   private Environment env;
   @Bean
   public DataSource getDataSource() {
      ComboPooledDataSource dataSource = new ComboPooledDataSource ();
      dataSource.setDriverClassName(env.getProperty("jdbc.driverClassName"));
      dataSource.setUrl(env.getProperty("jdbc.url"));
      dataSource.setUsername(env.getProperty("jdbc.username"));
      dataSource.setPassword(env.getProperty("jdbc.password"));
```

```
        dataSource.setInitialPoolSize(env.getProperty("jdbc.initPoolSize",Integer.class));
        dataSource.setMaxPoolSize(env.getProperty("jdbc.maxPoolSize",Integer.class));
        return dataSource;
    }
}
```

By default Spring Boot features such as external properties, logging, etc are available in the ApplicationContext only if we use SpringApplication. So, Spring Boot provides @SpringBootTest annotation to configure the ApplicationContext for tests which uses SpringApplication behind the scenes.

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

Applicatin.class is poassed as a parameter to tell Spring Boot that this is the primary component.

Note: As an alternate to application.properties, we can use a.yaml file. YAML provides a JSON-like structured configuration compared to the flat properties file.

```
#application.yaml
Server:
        port:9080
```

The @SpringBootApplication enables Spring component-scanning and Spring Boot Auto-Configuration. In fact, @SpringBootApplication combines three other useful annotations:

1. **@SpringBootConfiguration**: This annotation hints that the contained class declares one or more @Bean definition. It can be used as an alternative to the spring's standard @Configuration annotation. The @Configuration is a specialisation of @Component hence candidate for component scanning i.e., needs to give configuration class package name in test class. But @SpringBootConfiguration can be found automatically (for example in tests) hence need not to give configuration class package name in test class.
2. **@ComponentScan:** Enables componet-scanning so that the web controller classes and other components we write will be automatically discovered and registered as beans in the Spring application context. This annotation is save as <context:component-scan/> element.

3. **@EnableAutoConfiguration**: This enables the magic Spring servlet container. Once the application is running along with the embedded container, we can issue real HTTP request again missed it and make against the results.Boot auto-configuration

Embedded Container

The spring-boot-starter-web pulls the spring-boot-starter-tomcat automatically which starts tomcat as a embedded container. So we don't have to deploy our application on any externally installed tomcat server. That's exactly what Spring Boot's **@WebIntegrationTest** annotation does. Any annotating a test class with **@WebIntegrationTest**, we declare that we want Spring Boot to not only create an application context for our test, but also to start an embedded servlet container. Once the application is running along with the embedded container, we can issue real HTTP requests against it and make assertion against the results.

```
<dependency>
        <groupId> org.springframework.boot </groupId>
    <artifactId>spring-boot-starter-web</artifactId>
 </dependency>
```

Spring Boot Actuators

The problem with Auto discovery and Auto configuration is that it's difficult to know which beans were configured and how these beans wired together. The Spring Boot Actuators provide us details such as which beans have been configured, bean dependencies, autoconfig report (which contains both positive and negative matches), environment variables, health, configuration properties, memory usage, garbage collections, web requests, and data source usage.
The following starter dependency should be added in pom.xml file:

```
<dependency>
    <groupId> org.springframework.boot </groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
 </dependency>
```

Spring Boot Actuators provides following details:

1. What beans have been configured in the Spring application context
2. What decisions were made by Spring Boot's auto-configuration
3. What environment variables, system properties, configuration properties, and command-line arguments are available to our application.
4. A trace of recent HTTP requests handled by our application.

5. Various metrics pertaining to memory usage, garbage collection, web requests, and data source usage

The Actuator provides following REST endpoints.

REST End Point	Description
/beans	Describe all beans in the application context and their relationship each other.
/autoconfig	provides an auto-configuration report describing what auto configuration conditions passed and failed.
/env	Retrieves all environment properties.
/health	Reports health metrics for the application, as provided by HealthIndicator implementation.
/metrics	
..	

Note: All Actuators REST end points display response in JSON format. Hence add JSON Viewer through Chrome extensions [In Chrome browser --> Customize and control Google chrome--> More Tools --> Extensions --> Get More Extensions --> In 'Chrome web store' --> search and store with **'json viewer**'--> ADD TO CHROME (JSON Viewer)]
Note: Otherwise we can also use https://jsoneditoronline.org to format JSON messages.

Spring Boot Test

Spring Boot provides a number of utilities and annotations to help when testing our application. Test support is provided by two modules; **spring-boot-test** contains core items, and **spring-boot-test-autoconfigure** supports auto-configuration for tests.
Most developers use the **spring-boot-starter-test** which imports both Spring Boot test modules (such as spring-boot-test and spring-boot-test-autoconfigure) as well as JUnit, TestNG, Mockito, AssesertJ, Hamcrest, etc.

```
<dependency>
        <groupId> org.springframework.boot </groupId>
        <artifactId>spring-boot-starter-test</artifactId>
 </dependency>
```

The **SpringApplication** creates an appropriate ApplicationContext (depending on classpath), loads external property files such as application.properties, enables logging and other features of spring boot. At startup, SpringApplication loads all the properties and adds them to the Spring Environment class.

@SpringBootTest annotation can be used as an alternative to the standard spring-test @ContextConfiguration annotation when we need Spring Boot features. This

annotation works by creating the ApplicationContext used in our tests via SpringApplication.
Note: Don't forget to add @RunWith(SpringRunner.class) to our test, otherwise the annotations will be ignored.

Example:
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes= {"SpringJdbcConfig.class"})
public class ApplicationTest{ … }

Spring Boot MVC

The spring-boot-starter-web pulls the spring-boot-starter-tomcat automatically which starts tomcat as a embedded container. So we don't have to deploy our application on any externally installed tomcat server.

Below dependency needed to get web starter:

```
<dependency>
        <groupId> org.springframework.boot </groupId>
        <artifactId>spring-boot-starter-web</artifactId>
 </dependency>
```

Below dependency needed to process jsp pages:

```
<dependency>
        <groupId> org.apache.tomcat </groupId>
        <artifactId>tomcat-embed-jasper</artifactId>
 </dependency>
```

Below dependency needed to process html pages:

```
<dependency>
        <groupId> org.springframework.boot </groupId>
        <artifactId> spring-boot-starter-thymeleaf</artifactId>
 </dependency>
```

The @EnableWebMvc should shouldn't be used if we want to get spring MVC Boot features. If we want to take complete control of spring MVC, we can add @EnableWebMvc. In traditional web applications, a war file is created and then deployed to a external servlet container, whereas Spring Boot packages all the dependencies, embedded servlet container as a fat JAR file.

http://localhost:9090/NewCustomer.jsp
http://localhost:9090/customer/registration/form

CommandLineRunner

If we need to execute some custom code just before Boot application starting up? We can make that happen with a runner i.e., Spring Boot provides CommandLineRunner interface to run specific pieces of code when an application is fully started. When we want to execute some piece of code exactly before the application start up completes, we can use it then.

```
@SpringBootApplication
public class Hello implements CommandLineRunner{
        public static void main(String[] args){
                SpringApplication.run(Hello.class,args);
        }
@Override
public void run(String… arg0)throws Exception{
…
    }
}
```

Fat Jar

The below plug-in should be added in pom.xml file to create Fat Jar:

```
<!—Built Configuration -- >
        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-plugin</artifactId>
                        </plugin>
                </plugins>
                <finalName>FormProcessing</finalName>
        </build>
```

2. Spring JPA

JPA-based applications use an implementation of **EntityManagerFactory** to get an instance of an **EntityManager**.
The JPA specification defines two kinds of entity managers:

1. **Application Managed**
   With application is managed entity managers, the application is responsible for opening or closing entity manager. This type of entity manager is most appropriate for use in standalone applications that don't run in a Java EE container.
2. **Container Managed**
   Entity managers are created and managed by a Java EE container. The application doesn't interact with the entity manager factory at all. Instead, entity managers are opting directly through injection or from JNDI. This type of entity Manager is most appropriate for use by a Java EE container.

Regardless of which variety of EntityManager we want to use, spring will take responsibility for managing EntityManagers for us.
If we want to use application managed entity manager, spring plays the role of an application. In we want to use the container managed entity manager, spring plays the role of the container.

Hence as a developer we are good enough to configure appropriate factory bean as given below:

- **LocalEntityManagerFactoryBean** produces an application managed EntityManagerFactory.
- **LocalContainerEntityManagerFactoryBean** produces a container managed EntityManagerFactory.

Conclusion: working with spring JPA, the intricate details of dealing with either form of EntityManagerFactorys are hidden.

The only real difference between application managed and container managed entity manager factories, as far as spring is concerned, is how each is configured in the Spring application context.
When we're
Application managed entity manager factories provide most of the configuration in persistence.xml.
#META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
        <persistence-unit name="SpringJPAPU" transaction-type="RESOURCE_LOCAL">
                <class><com.cognizant.product.entities.Customer></class>
                <properties>
                <property name="javax.persistence.jdbc.driver" value="Database Driver Name" />
                <property name="javax.persistence.jdbc.url" value="Database Url" />
                <property name="javax.persistence.jdbc.user" value="Database Username" />
                <property name="javax.persistence.jdbc.password" value="Database Password" />
                </properties>
        </persistence-unit>
</persistence>
```

Because so much of configuration goes into a persistence.xml file, hence little configuration is sufficient in spring.

```
@Configuration
public class SpringJpaConfig {
   @Bean
   public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
  LocalContainerEntityManagerFactoryBean emfb = new LocalContainerEntityManagerFactoryBean();
      emfb.setPersistenceUnitName (“SpringJPAPU”);
      return emfb;
   }
}
```

In case of container managed JPA ,an EntityManagerFactory can be produced using information provided by the container, which is spring container in this case.
Instead of configuring data-source details in persistence.xml, rather we can configure this information directly in the spring configuration file.

```
@Configuration
public class SpringJpaConfig {
   Bean
   public DataSource dataSource() {
      DriverManagerDataSource ds = new DriverManagerDataSource();
      ds.setDriverClassName("oracle.jdbc.driver.OracleDriver");
      ds.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
      ds.setUsername("system");
      ds.setPassword("manager");
      return ds;
   }

   @Bean
   public JpaVendorAdapter hibJpaVendorAdapter() {
      HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
      adapter.setDatabase(Database.ORACLE);
      adapter.setShowSql(true);
      adapter.setGenerateDdl(false);
      // adapter.setDatabasePlatform("org.hibernate.dialect.Oracle10gDialect");
      return adapter;
```

```
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource ds,
            JpaVendorAdapter jpaVendorAdapter) {
        LocalContainerEntityManagerFactoryBean emfb = new LocalContainerEntityManagerFactoryBean();
        emfb.setDataSource(ds);
        emfb.setJpaVendorAdapter(jpaVendorAdapter);
        emfb.setPackagesToScan("edu.aspire.entities");
        return emfb;
    }
}
```

We can use **jpaVendorAdapter** property to provide specifics about the particular JPA implementation (such as Hibernate, Toplink, etc) to use. Spring comes with a handful of JPA vendor adapters to choose from:

- Hibernate JpaVendorAdapter
- Open JpaVendorAdaptor
- Toplink JpaVendorAdaptor

In this case, we're using Hibernate as a JPA implementation, so we configure it with the Hibernate JpaVendorAdapter.

Just like all of Spring's other persistence integration options, Spring-JPA integration comes in template from jpaTemplate. Nevertheless, template-based JPA has been set aside in favour of a pure JPA approach.

If the property is annotated with @ PersistenceContext, then spring can inject the EntityManager into the repository.

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
@Transactional
public class CustomerDaoImpl implements ICustomerDao {
    @PersistenceContext
    private EntityManager em;
        …
}
```

Example:

```
/*
CREATE TABLE CUSTOMER(CNO NUMBER(5)PRIMARY KEY, CNAME VARCHAR2(20), ADDRESS
VARCHAR2(100), PHONE NUMBER(15));
CREATE SEQUENCE CUSTOMER_SEQ;
*/
package com.cognizant.customer.entities;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
```

```
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

@Entity
@Table(name = "CUSTOMER")
@NamedQueries({
   @NamedQuery(name = "cust.findAll", query = "select c from Customer c"),
   @NamedQuery(name = "cust.findByName", query = "select c from Customer c where c.cname=?") })
public class Customer implements Serializable {
   @Id
   @Column(name = "CNO")
   @SequenceGenerator(name="CUSTOMER_CNO_GENERATOR",
sequenceName="CUSTOMER_SEQ", allocationSize=1)
   @GeneratedValue(strategy=GenerationType.SEQUENCE,
generator="CUSTOMER_CNO_GENERATOR")
   private int cno;
   @Column(name = "CNAME")
   private String cname;

   @Column(name = "ADDRESS")
   private String address;

   @Column(name = "PHONE")
   private long phone;
   public Customer() {
   }
   public int getCno() {
      return cno;
   }
   public void setCno(int cno) {
      this.cno = cno;
   }
   public String getCname() {
      return cname;
   }
   public void setCname(String cname) {
      this.cname = cname;
   }
   public String getAddress() {
      return address;
   }
   public void setAddress(String address) {
      this.address = address;
   }
   public long getPhone() {
      return phone;
```

```
    }
    public void setPhone(long phone) {
        this.phone = phone;
    }
}
```

```
package com.cognizant.customer.dao;
import java.util.List;
import com.cognizant.customer.entities.Customer;
public interface ICustomerDao {
    public void create(Customer c);
    public Customer read(int cno);
    public void update(Customer c);
    public void delete(Customer c);

    //finder methods
    public List<Customer> findAll();
    public List<Customer> findByName(String cname);
}
```

```
package com.cognizant.customer.dao;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import com.cognizant.customer.entities.Customer;

@Repository("custdao")
@Transactional
public class CustomerDaoImpl implements ICustomerDao {
    @PersistenceContext
    private EntityManager em;

    @Override
    public void create(Customer c) {
        em.persist(c);
        System.out.println("Customer details successfully inserted");
    }

    @Override
    public Customer read(int cno) {
        return em.find(Customer.class, cno);
    }

    @Override
    public void update(Customer c) {
        em.merge(c);
        System.out.println("Customer details successfully modified");
```

```
    }

    @Override
    public void delete(Customer c) {
        em.remove(em.merge(c));
        System.out.println("Customer details successfully deleted");
    }
    @Override
    public List<Customer> findAll(){
        Query q = em.createNamedQuery("cust.findAll");
        return q.getResultList();
    }

    @Override
    public List<Customer> findByName(String cname){
        Query q = em.createNamedQuery("cust.findByName");
        q.setParameter(1, cname);
        return q.getResultList();
    }
}

package com.cognizant.customer.config;
import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import org.springframework.boot.SpringBootConfiguration;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.Database;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@SpringBootConfiguration
@ComponentScan(basePackages = {"com.cognizant.customer.dao"})
@EntityScan(basePackages  = {"com.cognizant.customer.entities"})
@EnableAutoConfiguration
@EnableTransactionManagement
public class SpringJpaConfig {
    //Not required because of DataSourceConfiguration.Tomcat matched:
    /*@Bean
    public DataSource dataSource() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("oracle.jdbc.driver.OracleDriver");
        ds.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
        ds.setUsername("system");
```

```
        ds.setPassword("manager");
        return ds;
    }*/

    //Not required because of JpaBaseConfiguration#jpaVendorAdapter matched
    /*@Bean
    public JpaVendorAdapter hibJpaVendorAdapter() {
        HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
        adapter.setDatabase(Database.ORACLE);
        adapter.setShowSql(true);
        adapter.setGenerateDdl(false);
        // adapter.setDatabasePlatform("org.hibernate.dialect.Oracle10gDialect");
        return adapter;
    }*/

    //Not required because of HibernateJpaAutoConfiguration matched
    /*@Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource ds,
            JpaVendorAdapter jpaVendorAdapter) {
        LocalContainerEntityManagerFactoryBean emfb = new
LocalContainerEntityManagerFactoryBean();
        emfb.setDataSource(ds);
        emfb.setJpaVendorAdapter(jpaVendorAdapter);
        emfb.setPackagesToScan("edu.aspire.entities");
        return emfb;
    }*/

    //Not required because of JpaBaseConfiguration#transactionManager matched
    /*@Bean
    public PlatformTransactionManager transactionManager(LocalContainerEntityManagerFactoryBean
entityManagerFactory) {
        EntityManagerFactory factory = entityManagerFactory.getObject();
        return new JpaTransactionManager(factory);
    }*/
}
```

#src/main/resources/application.properties

```
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=manager

#tomcat-connection settings
spring.datasource.tomcat.initialSize=20
spring.datasource.tomcat.max-active=25

# Hikari settings
#spring.datasource.hikari.maximum-pool-size=20

# dbcp settings
```

```
#spring.datasource.dbcp.initial-size=7
#spring.datasource.dbcp.max-active=20

# dbcp2 settings
#spring.datasource.dbcp2.initial-size=7
#spring.datasource.dbcp2.max-total=20

spring.jpa.show-sql=true
#spring.jpa.hibernate.ddl-auto=create
debug=true
package com.cognizant.customer.test;
import java.util.List;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import com.cognizant.customer.config.SpringJpaConfig;
import com.cognizant.customer.dao.ICustomerDao;
import com.cognizant.customer.entities.Customer;
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes={SpringJpaConfig.class})
public class SpringJpaTest {
   @Autowired
   ApplicationContext context;
   @Autowired
   ICustomerDao custDao;

   @Test
   public void testInsertJpa() {
      Customer cust = new Customer();
      cust.setCname("Praveen");
      cust.setAddress("Hyderabad");
      cust.setPhone(1212121212L);
      custDao.create(cust);
   }

   /*@Test
   public void testReadJpa() {
      Customer c = custDao.read(1);
      System.out.println(c.getCno() + "  " + c.getCname() +" " + c.getAddress() +"  " + c.getPhone());
   }

   @Test
   public void testUpdateJpa(){
      Customer c = custDao.read(1);
      c.setPhone(1212121212L);
      custDao.update(c);
   }
```

```
    @Test
    public void testDeleteJpa(){
        Customer c = custDao.read(1);
        custDao.delete(c);
    }

    @Test
    public void testFindAllJpa(){
        List<Customer> custs = custDao.findAll();
        System.out.println("***FindAll***:" + custs.size());
    }

    @Test
    public void testFindByNameJpa(){
        List<Customer> custs = custDao.findByName("Praveen");
        System.out.println("***FindByName***:" + custs.size());
    }*/
}
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>edu.aspire</groupId>
    <artifactId>SpringJPA</artifactId>
    <version>1</version>
    <packaging>jar</packaging>
    <name>Spring JPA Project</name>
    <url>http://www.java2aspire.com</url>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.3.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
        </dependency>

        <dependency>
            <groupId>oracle</groupId>
```

```
        <artifactId>oracle-jdbc</artifactId>
        <version>11</version>
      </dependency>
    </dependencies>

    <build>
      <plugins>
        <!-- using Java 8 -->
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <source>1.8</source>
            <target>1.8</target>
          </configuration>
        </plugin>
      </plugins>
    </build>
</project>
```

3. Spring DATA

This module is also called as spring data Jpa.
The create(), read(), update() and delete() methods are fairly common across all DAOs. The only difference is the domain types.
Spring Data internally provide implementation for all such common methods(as part of org.springframework.data.repository.**CrudRepository.**Hence developer need not to write implementation for these common methods.

Example:

```
package com.cognizant.customer.dao;
import org.springframework.data.jpa.repository.JpaRepository;
public interface ICustomerDao extends JpaRepository<Customer, Integer>{
}
```

The JpaRepository is parameterized such that it takes entity class and type of ID. It also inherits 18 methods for performing, common persistence operations such as saving, deleting, finding, etc from both CrudRepository and JpaRepository classes.

Spring DATA will automatically provide implementation for all these common methods if and only if we add
@EnableJpaRepositories (basePackages= "com.cognizant.customer.dao") in configuration class.

Example:

```
@SpringBootConfiguration
@EnableJpaRepositories (basePackages= "com.cognizant.customer.dao")
public class SpringDataConfig{
…
}
```

The @EnableJpaRepositories scans its base package for any interfaces that extends JpaRepository interface. When it finds any interface which extends JpaRepository, it automatically generates and implementation of that dao interface.

*Spring Data are not only provides implementation for commonly used methods but also provides a way to add **custom methods**. The method signature tells Spring Data everything it needs to know in order to create an implementation for the method. Spring Data defines a sort of **domain-specific language (DSL)** where persistence details are expressed in **method signature.***

Example:

```
public interface CutomerDao extends JpaRepository<Customer,Integer>{
//finder methods
public List<Customer> findCustomerByAddress(String addr);
}
```

Repository methods are composed of a **verb**, an optional **subject**, the word **by**, and a **predicate**. In case of findByCname(), the verb is **find** and the predicate **Cname** the subject isn't specified and is implied to be a customer.
Spring Data allows **four verbs** in the method name: **get, read, find, and count**. The get, read, and find verbs are synonymous; all three result in repository methods that query for data and return objects. The count verb, on the other hand, returns a count of matching objects, rather than the objects themselves.

The method name, **readCustomerByFirstNameOrLastName**(String first, String last), the verb is read and the predicate is **FirstNameOrLastName.**
The predicate is the most interesting part of the method name.
We can use any of the following **comparison operator** from property to parameter:
1) IsAfter, After, IsGreaterThan, GreaterThan
2) IsGreaterThanEqual, GreaterThanEqual
3) IsBefore, Before, IsLessThan, LessThan
4) IsLessThanEqual, LessThanEqual
5) IsBetween, Between

6) IsNull, Null
7) IsNotNull, NotNull
8) Isln, In
9) IsNotln, Notln
10) IsStartingWith, StartingWith, StartsWith
11) IsEndingWith, EndingWith, EndsWith
12) IsContaining, Containing, Contains
13) IsLike, Like
14) IsNotLike, Notlike
15) IsTrue, True
16) IsFalse, False
17) Is, Equals
18) IsNot, Not

The property value will be compared against the method parameter.
The full method signature looks like this:
public List<Customer>findByFirstnameOrLastname(String first, String last);
In above method signature, the comparison operator is left off, it's implied to be an equals operation.

In method signature public List<Customer>
readByFirstnamelgnoringCaseOrLastnamelgnoresCase(String first, String last), the conditions are IgnoringCase or IgnoresCase to ignore case on the firstname and lastname properties.
Note: The lgnoringCase and IgnoresCase are synonymous.
As an alternative to IgnoringCase/lgnoresCase, we may also use either AlllgnoringCase or AllignoresCase.
List<Customer> readByFirstnameOrLastname AllignoresCase(String first, String last);

We can sort the results by adding **OrderBy** at the end of the method name
To sort the results in ascending order by the lastname property, the method signature is:

public List<Customers readByFirstnameOrLastnameOrderByLastnameAsc(String first, String last);
To sort results in ascending order by the firstname property and decending order by the lastname property, the
method signature is:
List<Customers readByFirstnameorLastnameOrderByFirstnameAscLastname Desc(String first, String last);

Althouth Spring Data Jpa generates an implementation method to query for almost anything we can imagine
nevertheless, Spring Data's mini-DSL has its limits, and sometimes it isn't convenient or even possible to express the desired query in a method name. When that happens, Spring Data provides **@Query** annotation to write query explicitly.

Suppose we want to create a repository method to find all customers whose email address is a Gmail address
One way to do this is to define a findByEmailLike String mail method and passin %gmail.com to find Gmail
users. But it would be nice to define a more convenient findAllGmailCustomers() method that doesn't require

the partial email address to be passed in: List<Customer> findAllGmailCustomers(). Unfortunately, this method
t adhere to Spring Data's method-naming conventions (DSL). In situations where the desired data can't be adequately expressed in the method name, we can use the @Query annotation to provide Spring Data
with the query that should be performed, For the findAllGmailCustomers() method, we might use @Query like this:

@Query("select c from Customer c wherec.email like %gmail.com' ")
public List<Customer> fetchOnlyGmailCustomers();
We still don't need to write the implementation for fetchAllGmailCustomrs() method. We only give the query,
hinting to Spring Data about how it should implement the method.
Also, @Query can also be useful if we followed the naming convention, the method name would be incredibly
long. In such situation, we'd probably rather come up with a shorter method name and use @Query to specify
how the method should query the database
The @Query annotation is handy for adding custom query methods to a Spring Data JPA-enabled interface.

Sometimes we cannot describe functionality with Spring Data's method-naming conventions or even with a query given in the @Query annotation. Such specific scenarios can be implemented by using EntityManager
*(from Spring JPA) and remaining functionalities can be worked with Spring Data ie, we can **mix** EntitityManager (to work at lower level) in Spring JPA with Spring Data (grunt work for the stuff it knows how to do).*

The Spring Data Jpa generates the implementation for a repository interface whose name is same as interface's
name and **postfixed with impl.**

When Spring Data Jpa generates the implementation for a repository interface, it also looks for a class whose
name is the same as the interface's name **postfixed with Impl.** If the class exists, Spring Data JPA merges its
methods with those generated by Spring Data JPA. For the CustomerDao interface, the class it looks for is named CustomerDao**Impl**.

Example:
```
public interface IFindCustomerMobile {
public Customer getCustomer(long phone);
}
package com.cognizant.dao;
@Repository
public class CustomerDaoImpl implements IFindCustomerMobile {
@PersistenceContext
private EntityManager em;

public int getcustomer(long phone) {
…
```

```
  }
}
```

Notice that CustomerDaoImpl doesn't implement the CustomerDao interface. Spring Data JPA is still responsible for implementing that interface. Our CustomerDaoImpl implements IFindCustomerMobile interface.
We should also make sure that the getCustomer() method is declared in the CustomerDao interface. The easy
way to do that and avoid duplicating code is to change CustomerDao so that it extends IFindCustomerMobile.

```
import org.springframework.data.jpa.repository.JpaRepository;
public interface CustomerDao extends JpaRepository<Customer, Integer>, IFindCustomerMobile{
public List<Customer> findByPhone(String mobile);
public List<Customery findByCname(String name);
}
```

Example

```
/*
CREATE TABLE CUSTOMER(CNO NUMBER(5)PRIMARY KEY, CNAME VARCHAR2(20), ADDRESS
VARCHAR2(100), PHONE NUMBER(15));
CREATE SEQUENCE CUSTOMER_SEQ;
*/
package com.cognizant.customer.entities;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

@Entity
@Table(name = "CUSTOMER")
@NamedQueries({@NamedQuery(name = "cust.findByName", query = "select c from Customer c
where c.cname=?") })
public class Customer implements Serializable {
   @Id
   @Column(name = "CNO")
   @SequenceGenerator(name="CUSTOMER_DNO_GENERATOR",
sequenceName="CUSTOMER_SEQ", allocationSize=1)
   @GeneratedValue(strategy=GenerationType.SEQUENCE,
generator="CUSTOMER_DNO_GENERATOR")
   private int cno;

   @Column(name = "CNAME")
   private String cname;

   @Column(name = "ADDRESS")
   private String address;
```

```java
    @Column(name = "PHONE")
    private long phone;

    public Customer() {
    }

    public int getCno() {
        return cno;
    }

    public void setCno(int cno) {
        this.cno = cno;
    }

    public String getCname() {
        return cname;
    }

    public void setCname(String cname) {
        this.cname = cname;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public long getPhone() {
        return phone;
    }

    public void setPhone(long phone) {
        this.phone = phone;
    }
}

package com.cognizant.customer.dao;
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import com.cognizant.customer.entities.Customer;
//@Transactional
public interface ICustomerDao extends JpaRepository<Customer, Integer>{
    //finder methods
    public List<Customer> findAll();
    public List<Customer> findByCname(String cname);
}
```

```
package com.cognizant.customer.dao;
import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import com.cognizant.cystomer.entities.Customer;
//@Transactional
public interface ICustomerDao extends JpaRepository<Customer, Integer>{
   //finder methods
   public List<Customer> findAll();
   public List<Customer> findByCname(String cname);
}
```

#src/main/config/application.properties

```
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=manager

#tomcat-connection settings
spring.datasource.tomcat.initialSize=20
spring.datasource.tomcat.max-active=25

# Hikari settings
#spring.datasource.hikari.maximum-pool-size=20
# dbcp settings
#spring.datasource.dbcp.initial-size=7
#spring.datasource.dbcp.max-active=20

# dbcp2 settings
#spring.datasource.dbcp2.initial-size=7
#spring.datasource.dbcp2.max-total=20
spring.jpa.show-sql=true
#spring.jpa.hibernate.ddl-auto=create
debug=true
```

```
package com.cognizant.customer.config;
import org.springframework.boot.SpringBootConfiguration;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@SpringBootConfiguration
@EnableJpaRepositories(basePackages = "com.cognizant.customer.dao")
@EntityScan(basePackages  = {"com.cognizant.customer.entities"})
@EnableAutoConfiguration
@EnableTransactionManagement
public class SpringDataJpaConfig {
   //Not required because of DataSourceConfiguration.Tomcat matched:
   /*@Bean
   public DataSource dataSource() {
      DriverManagerDataSource ds = new DriverManagerDataSource();
      ds.setDriverClassName("oracle.jdbc.driver.OracleDriver");
```

```
        ds.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
        ds.setUsername("system");
        ds.setPassword("manager");
        return ds;
    }*/

    //Not required because of JpaBaseConfiguration#jpaVendorAdapter matched
    /*@Bean
    public JpaVendorAdapter hibJpaVendorAdapter() {
        HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
        adapter.setDatabase(Database.ORACLE);
        adapter.setShowSql(true);
        adapter.setGenerateDdl(false);
        // adapter.setDatabasePlatform("org.hibernate.dialect.Oracle10gDialect");
        return adapter;
    }*/

    //Not required because of HibernateJpaAutoConfiguration matched
    //Method name must be entitiyManagerFactory because Spring Data Jpa by default looks for an
EntityManagerFactory named 'entityManagerFactory'
    /*@Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource ds,
            JpaVendorAdapter jpaVendorAdapter) {
        LocalContainerEntityManagerFactoryBean emfb = new
LocalContainerEntityManagerFactoryBean();
        emfb.setDataSource(ds);
        emfb.setJpaVendorAdapter(jpaVendorAdapter);
        emfb.setPackagesToScan("edu.aspire.entities");
        return emfb;
    }*/

    //Not required because of JpaBaseConfiguration#transactionManager matched
    /*@Bean
    public PlatformTransactionManager transactionManager(LocalContainerEntityManagerFactoryBean
entityManagerFactory) {
        EntityManagerFactory factory = entityManagerFactory.getObject();
        return new JpaTransactionManager(factory);
    }*/
}

package com.cognizant.customer.test;
import java.util.List;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import com.cognizant.customer.config.SpringDataJpaConfig;
import com.cognizant.customer dao.ICustomerDao;
```

```
import com.cognizant.customer.entities.Customer;
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes={SpringDataJpaConfig.class})
public class SpringDataJpaTest {
    @Autowired
    ApplicationContext context;
    @Autowired
    ICustomerDao custDao;

    @Test
    public void testInsertJpa() {
        Customer c = new Customer();
        c.setCname("Praveen");
        c.setAddress("Hyderabad");
        c.setPhone(1212121212L);
        custDao.save(c);
    }

    /*@Test
    public void testReadJpa() {
        Customer c = custDao.findOne(1);
        System.out.println(c.getCno() + "  " + c.getCname() +" " + c.getAddress() +"  " + c.getPhone());
    }

    @Test
    public void testUpdateJpa(){
        Customer c = custDao.findOne(1);
        c.setPhone(7799208899L);
        custDao.save(c); //In Spring Data JPA the save() is either persist() or merge() based on primary key
present or not.
    }

    @Test
    public void testDeleteJpa(){
        Customer c = custDao.findOne(1);
        custDao.delete(c);
    }

    @Test
    public void testFindAllJpa(){
        List<Customer> custs = custDao.findAll();
        System.out.println("***FindAll***:" + custs.size());
    }

    @Test
    public void testFindByNameJpa(){
        List<Customer> custs = custDao.findByCname("ramesh");
        System.out.println("***FindByCname***:" + custs.size());
    }*/
}
```