UNIT II:
Spring Boot: Creating a Spring Boot Application, Spring Boot Application Annotation, What is Auto wiring , Scope of a bean, Logger, Introduction to Spring AOP, Implementing AOP advices, Best Practices : Spring Boot Application

Spring Boot

So far you have learned that Spring is a lightweight framework for developing enterprise Java applications. But using Spring for application development is challenging for developer because of the following reason which reduces productivity and increases the development time:

1. Configuration

You have seen that the Spring application requires a lot of configuration. This configuration also needs to be overridden for different environments like production, development, testing, etc. For example, the database used by the testing team may be different from the one used by the development team. So we have to spend a lot of time writing configuration instead of writing application logic for solving business problems.

2. Project Dependency Management

When you develop a Spring application you have to search for all compatible dependencies for the Spring version that you are using and then manually configure them. If the wrong version of dependencies is selected then it will be an uphill task to solve this problem. Also for every new feature added to the application, the appropriate dependency needs to be identified and added. All this reduces productivity.

So to handle all these kinds of challenges Spring Boot came into the market.

What is Spring Boot?

Spring Boot is a framework built on top of the Spring framework that helps the developers to build Spring-based applications very quickly and easily. The main goal of Spring Boot is to create Spring-based applications quickly without demanding developers to write the boilerplate configuration.

But how does it work? It works because of the following reasons,

1. Spring Boot is an opinionated framework

Spring Boot forms opinions. It means that Spring Boot has some sensible defaults which you can use to quickly build your application. For example, Spring Boot uses embedded Tomcat as the default web container.

2. Spring Boot is customizable

Though Spring Boot has its defaults, you can easily customize it at any time during your development based on your needs. For example, if you prefer log4j for logging over Spring Boot built-in logging support then you can easily make a dependency change in your pom.xml file to replace the default logger with log4j dependencies.

The main Spring Boot features are as follows:

- Starter Dependencies
- Automatic Configuration
- Spring Boot Actuator
- Easy-to-use Embedded Servlet Container Support

Creating a Spring Boot Application

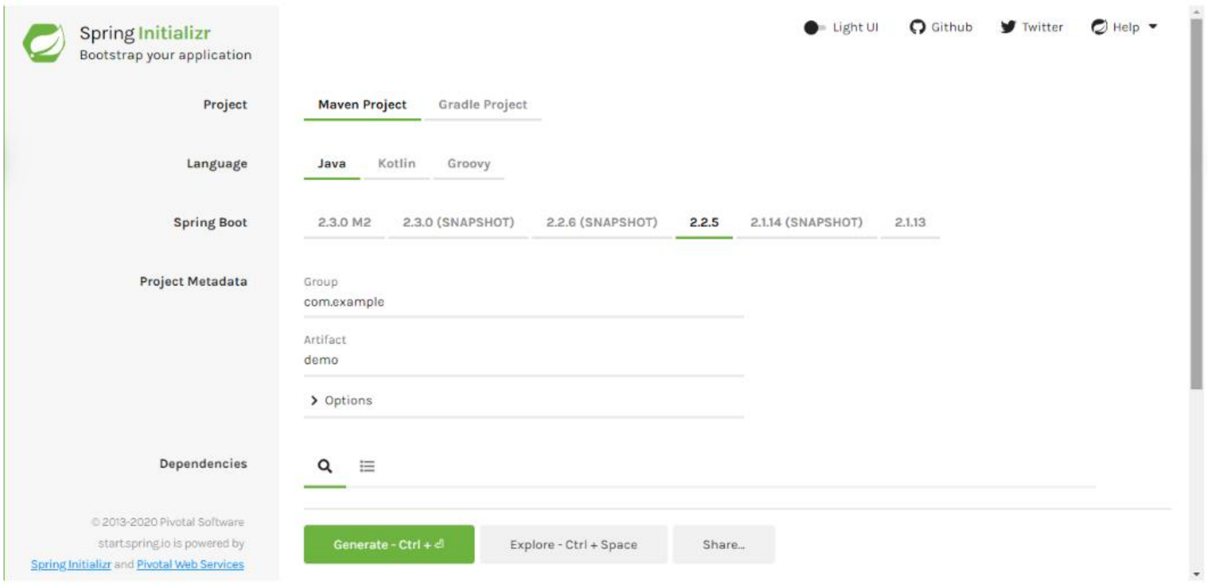
There are multiple approaches to create a Spring Boot application. You can use any of the following approaches to create the application:

- **Using Spring Boot CLI**
- **Using Spring Initializr**
- **Using the Spring Tool Suite (STS)**

In this course, you will learn how to use Spring Initializr for creating Spring Boot applications.

It is an online tool provided by Spring for generating Spring Boot applications which is accessible at http://start.spring.io/. You can use it for creating a Spring Boot project using the following steps:

Step 1: Create your Spring Boot application launch Spring Initializr. You will get the following screen:


Spring Initializr
Bootstrap your application
Light UI
Github
Twitter
Help
Generate - Ctrl + ⏎
Explore - Ctrl + Space
Share...
© 2013-2020 Pivotal Software
start.spring.io is powered by
Spring Initializr and Pivotal Web Services

Project: Maven Project | Gradle Project
Language: Java | Kotlin | Groovy
Spring Boot: 2.3.0 M2 | 2.3.0 (SNAPSHOT) | 2.2.6 (SNAPSHOT) | 2.2.5 | 2.1.14 (SNAPSHOT) | 2.1.13
Project Metadata
Group: com.example
Artifact: demo
> Options
Dependencies

Note: This screen keeps changing depending on updates from Pivotal and changes in the Spring Boot version.

Step 2: Select Project as Maven, Language as Java, and Spring Boot as 2.1.13 and enter the project details as follows:

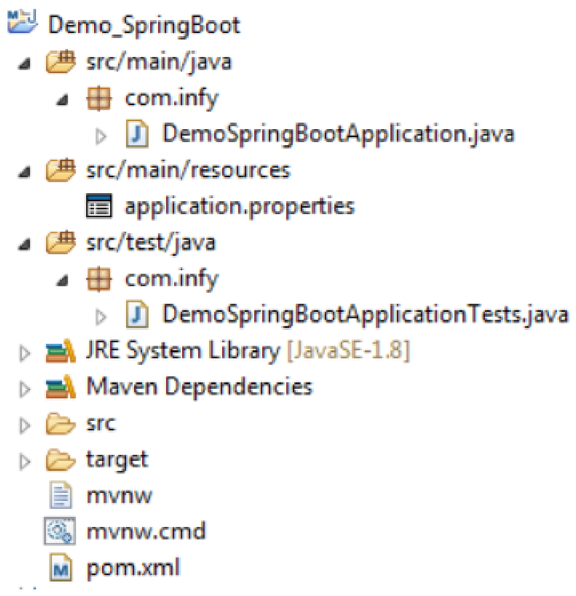
- Choose com.infy as Group
- Choose Demo_SpringBoot as Artifact

Click on More options and choose com.infy as Package Name

Step 3: Click on Generate Project. This would download a zip file to your local machine.

Step 4: Unzip the zip file and extract it to a folder.

Step 5: In Eclipse, Click File → Import → Existing Maven Project. Navigate or type in the path of the folder where you extracted the zip file to the next screen. After finishing, our Spring Boot project should look like as follows:



```
Demo_SpringBoot
  src/main/java
    com.infy
      DemoSpringBootApplication.java
  src/main/resources
    application.properties
  src/test/java
    com.infy
      DemoSpringBootApplicationTests.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

You have created a Spring Boot Maven-based project. Now let us explore what is contained in the generated project

Understanding Spring Boot project structure

The generated project contains the following files:

1. pom.xml

This file contains information about the project and configuration details used by Maven to build the project.

```
<?xml version="1.0" encoding="UTF-8"?>

<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>

        <parent>

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

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

                <version>2.1.13.RELEASE</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.infy</groupId>
        <artifactId>Demo_SpringBoot</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>Demo_SpringBoot</name>
        <description>Demo project for Spring Boot</description>
        <properties>
                <java.version>1.8</java.version>
        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter</artifactId>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>
        </dependencies>
        <build>
                <plugins>
```

```
                <plugin>

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

                    <artifactId>spring-boot-maven-plugin</artifactId>

                </plugin>

            </plugins>

        </build>

</project>
```

2. application.properties

This file contains application-wide properties. To configure your application Spring reads the properties defined in this file. In this file, you can define a server's default port, the server's context path, database URLs, etc.

3. DemoSpringBootApplication.java

```
@SpringBootApplication

public class DemoSpringBootApplication {

    public static void main(String[] args) {

        SpringApplication.run(DemoSpringBootApplication.class, args);

    }

}
```

It is annotated with @SpringBootApplication annotation which triggers auto-configuration and component scanning and can be used to declare one or more @Bean methods also. It contains the main method which bootstraps the application by calling the run() method on the SpringApplication class. The run method accepts DemoSpringBootApplication.class as a parameter to tell Spring Boot that this is the primary component.

4. DemoSpringBootApplicationTest.java

In this file test cases are written. This class is by default generated by Spring Boot to bootstrap Spring application.

Spring Boot Starters

Spring Boot starters are pre-configured dependency descriptors with the most commonly used libraries that you can add to your application. So you don't need to search for compatible libraries and configure them manually. Spring Boot will ensure that the necessary libraries are added to the build. To use these starters, you have to add them to the pom.xml file. For example, to use spring-boot-starter following dependency needs to be added in pom.xml:

```
<dependency>

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

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

</dependency>
```

Spring Boot comes with many starters. Some popular starters which we are going to use in this course are as follows:

- spring-boot-starter - This is the core starter that includes support for auto-configuration, logging, and YAML.
- spring-boot-starter-aop - This starter is used for aspect-oriented programming with Spring AOP and AspectJ.
- spring-boot-starter-data-jdbc - This starter is used for Spring Data JDBC.
- spring-boot-starter-data-jpa - This starter is used for Spring Data JPA with Hibernate.
- spring-boot-starter-web - This starter is used for building a web application using Spring MVC and Spring REST. It also provides Tomcat as the default embedded container.
- spring-boot-starter-test - This starter provides support for testing Spring Boot applications using libraries such as JUnit, Hamcrest, and Mockito.

Spring Boot Starter Parent

The Spring Boot Starter Parent defines key versions of dependencies and default plugins for quickly building Spring Boot applications. It is present in the pom.xml file of the application as a parent as follows:

```
<parent>

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

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

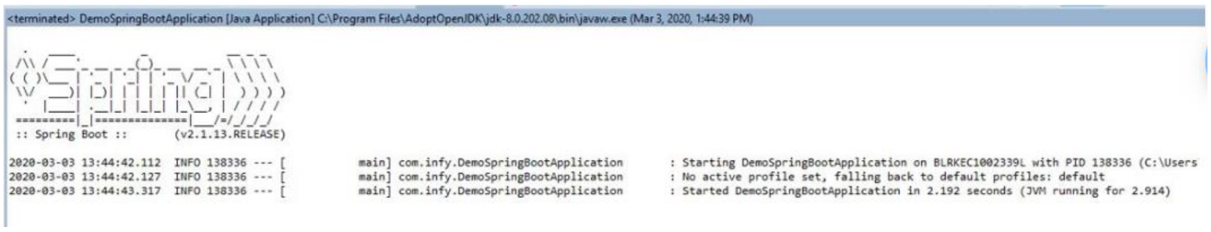
```
        <version>2.1.13.RELEASE</version>

        <relativePath/>

</parent>
```

It allows you to manage the following things for multiple child projects and modules:

- **Configuration** – The Java version and other properties.
- **Dependencies** – The version of dependencies.
- **Default Plugins Configuration** – This includes default configuration for Maven plugins such as maven-failsafe-plugin, maven-jar-plugin, maven-surefire-plugin, maven-war-plugin.

Executing the Spring Boot application

To execute the Spring Boot application run the DemoSpringBootApplication as a standalone Java class which contains the main method. On successful execution you will get the following output in the console:



```
<terminated> DemoSpringBootApplication [Java Application] C:\Program Files\AdoptOpenJDK\jdk-8.0.202.08\bin\javaw.exe (Mar 3, 2020, 1:44:39 PM)

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::       (v2.1.13.RELEASE)

2020-03-03 13:44:42.112  INFO 138336 --- [           main] com.infy.DemoSpringBootApplication       : Starting DemoSpringBootApplication on BLRKEC1002339L with PID 138336 (C:\Users
2020-03-03 13:44:42.127  INFO 138336 --- [           main] com.infy.DemoSpringBootApplication       : No active profile set, falling back to default profiles: default
2020-03-03 13:44:43.317  INFO 138336 --- [           main] com.infy.DemoSpringBootApplication       : Started DemoSpringBootApplication in 2.192 seconds (JVM running for 2.914)
```

Spring Boot Runners

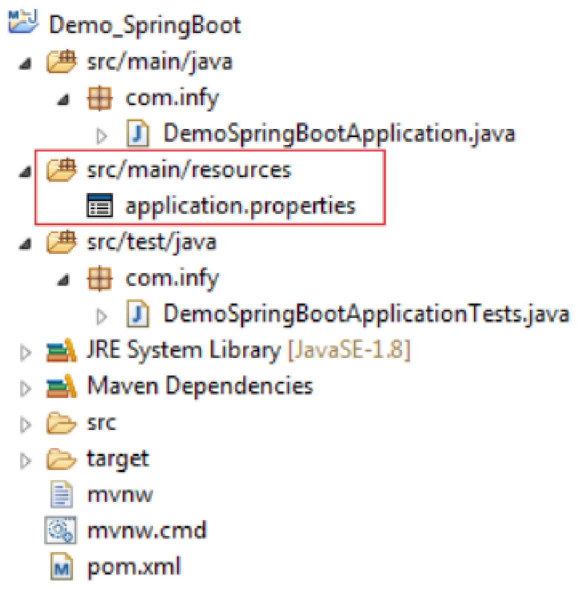
So far you have learned how to create and start the Spring Boot application. Now suppose you want to perform some action immediately after the application has started then for this Spring Boot provides the following two interfaces:

- CommandLineRunner
- ApplicationRunner

CommandLineRunner is the Spring Boot interface with a run() method. Spring Boot automatically calls this method of all beans implementing this interface after the application context has been loaded. To use this interface, you can modify the DemoSpringBootApplication class as follows:

```
@SpringBootApplication

public class DemoSpringBootApplication implements CommandLineRunner {
        public static void main(String[] args) {
                SpringApplication.run(DemoSpringBootApplication.class,
args);
        }
        @Override
        public void run(String... args) throws Exception {
                System.out.println("Welcome to CommandLineRunner");
        }
}
```

Spring Boot application is configured using a file named application.properties. It is auto detected without any Spring based configurations and is placed inside the "src/main/resources" directory as shown below:



```
Demo_SpringBoot
  src/main/java
    com.infy
      DemoSpringBootApplication.java
  src/main/resources
    application.properties
  src/test/java
    com.infy
      DemoSpringBootApplicationTests.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

In this file, various default properties are specified to support logging, AOP, JPA, etc. All the default properties need not be specified in all cases. We can specify them only on-demand. At startup, the Spring application loads all the properties and adds them to the Spring Environment class.

To use a custom property add it to the application.properties file.

application.properties

```
        message= Welcome Spring
```

Then autowire the Environment class into a class where the property is required.

```
        @Autowired

                Environment env;
```

You can read the property from Environment using the getProperty() method.

```
        env.getProperty("message")
```

You can use other files to keep the properties. For example, InfyTelmessage.properties.

InfyTelmessage.properties

```
        message=Welcome To InfyTel
```

But by default Spring Boot will load only the application.properties file. So how you will load the **InfyTelmessage.properties** file?

In Spring @PropertySource annotation is used to read from properties file using Spring's Environment interface. The location of the properties file is mentioned in the Spring configuration file using @PropertySource annotation.

So InfyTelmessage.properties which are present in classpath can be loaded using @PropertySource as follows:

```
import org.springframework.context.annotation.PropertySource;

@SpringBootApplication

@PropertySource("classpath:InfyTelmessage.properties")

public class DemoSpringBootApplication {

        public static void main(String[] args) throws Exception {

                //code

        }

}
```

To read the properties you need to autowire the Environment class into a class where the property is required

```
@Autowired

Environment env;
```

You can read the property from Environment using the getProperty() method.

```
env.getProperty("message")
```

Understanding @SpringBootApplication Annotation

We have already learned that the class which is used to bootstrap the Spring Boot application is annotated with @SpringBootApplication annotation as follows:

```
@SpringBootApplication

public class DemoSpringBootApplication {

   public static void main(String[] args) {

      SpringApplication.run(DemoSpringBootApplication.class, args);

   }

}
```

Now let us understand this annotation in detail.

The **@SpringBootApplication** annotation indicates that it is a configuration class and also triggers auto-configuration and component scanning. It is a combination of the following annotations with their default attributes:

@EnableAutoConfiguration – This annotation enables auto-configuration for the Spring boot application which automatically configures our application based on the dependencies that you have added.

@ComponentScan – This enables the Spring bean dependency injection feature by using @Autowired annotation. All application components which are annotated with @Component, @Service, @Repository, or @Controller are automatically registered as Spring Beans. These beans can be injected by using @Autowired annotation.

@Configuration – This enables Java based configurations for Spring boot application.

The class that is annotated with @SpringBootApplication will be considered as the main class, is also a bootstrap class. It kicks starts the application by invoking the SpringApplication.run() method. You need to pass the .class file name of your main class to the run() method.

SpringBootApplication- scanBasePackages

By default, SpringApplication scans the configuration class package and all it's sub-packages. So if our SpringBootApplication class is in "com.eta" package, then it won't scan com.infy.service or com.infy.repository package. We can fix this situation using the SpringBootApplication scanBasePackages property.

```
package com.eta;

@SpringBootApplication(scanBasePackages={"com.infy.service","com.infy.repository"})

public class DemoSpringBootApplication {

    public static void main(String[] args) {

        SpringApplication.run(DemoSpringBootApplication.class, args);

    }

}
```

Demo on Spring Boot

Objective: To understand the Spring IOC feature using SpringBoot

Demo Steps:

Demo5Application .java

```
package com.infy;

import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
import
org.springframework.context.support.AbstractApplicationContext;

import com.infy.service.CustomerServiceImpl;

@SpringBootApplication

public class Demo5Application{

        public static void main(String[] args) {



                        CustomerServiceImpl service = null;

                        AbstractApplicationContext context =
(AbstractApplicationContext) SpringApplication

                                        .run(Demo5Application.class, args);

                        service = (CustomerServiceImpl)
context.getBean("customerService");

                        System.out.println(service.fetchCustomer());

               context.close();

        }

}
```

CustomerService.java

```
        package com.infy.service;

        public interface CustomerService {
```

```
        public String fetchCustomer();

        public String createCustomer();

    }
```

CustomerServiceImpl.java

```
package com.infy.service;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Service;

@Service("customerService")

public class CustomerServiceImpl implements CustomerService {

    @Value("10")

    private int count;

    public String fetchCustomer() {

        return " The no of customers fetched are : " + count;

    }

    public String createCustomer() {

        return "Customer is successfully created";

    }

}
```

OUTPUT:

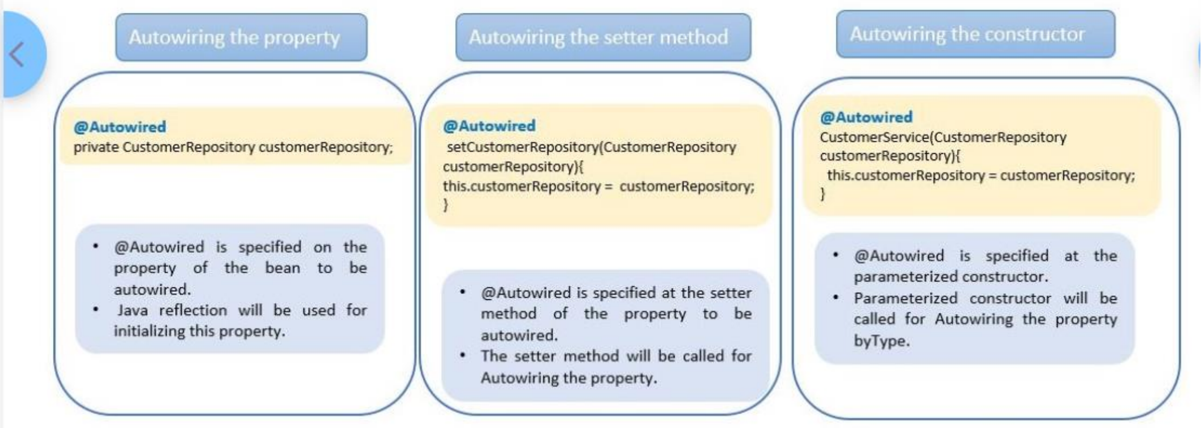
```

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::       (v2.1.13.RELEASE)

2020-04-07 17:21:31.908  INFO 119940 --- [           main] com.infy.Demo5Application
: Starting Demo5Application on
2020-04-07 17:21:31.912  INFO 119940 --- [           main] com.infy.Demo5Application
: No active profile set, falling back to default profiles:
2020-04-07 17:21:34.211  INFO 119940 --- [           main] com.infy.Demo5Application
: Started Demo5Application in 3.998 seconds
The no of customers fetched are : 10

```

What is Autowiring ?

In Spring if one bean class is dependent on another bean class then the bean dependencies need to be explicitly defined in your configuration class. But you can let the Spring IoC container to inject the dependencies into dependent bean classes without been defined in your configuration class. This is called as autowiring.

To do autowiring, you can use @Autowired annotation. This annotation allows the Spring IoC container to resolve and inject dependencies into your bean.@Autowired annotation performs **byType Autowiring** i.e. dependency is injected based on bean type. It can be applied to attributes, constructors, setter methods of a bean class.

Autowiring is done only for dependencies to other beans. It doesn't work for properties such as primitive data types, String, Enum, etc. For such properties, you can use the **@Value** annotation.



```
@Autowired
private CustomerRepository customerRepository;
```

- @Autowired is specified on the property of the bean to be autowired.
- Java reflection will be used for initializing this property.

```
@Autowired
setCustomerRepository(CustomerRepository customerRepository){
this.customerRepository = customerRepository;
}
```

- @Autowired is specified at the setter method of the property to be autowired.
- The setter method will be called for Autowiring the property.

```
@Autowired
CustomerService(CustomerRepository customerRepository){
 this.customerRepository = customerRepository;
}
```

- @Autowired is specified at the parameterized constructor.
- Parameterized constructor will be called for Autowiring the property byType.

Now let us see how to use @Autowired annotation.

@Autowired on Setter methods

The @Autowired annotation can be used on setter methods. This is called a Setter Injection.

```
package com.infy.service;

public class CustomerServiceImpl implements CustomerService {



        private CustomerRepository customerRepository;

        @Autowired

        public void setCustomerRepository(CustomerRepository
customerRepository) {

                this.customerRepository = customerRepository;

        }

      --------

}
```

In the above code snippet, the Spring IoC container will call the setter method for injecting the dependency of CustomerRepository

@Autowired on Constructor

The @Autowired annotation can also be used on the constructor. This is called a Constructor Injection.

```
package com.infy.service;

public class CustomerServiceImpl implements CustomerService {
        private CustomerRepository customerRepository;

        @Autowired

        public CustomerServiceImpl(CustomerRepository customerRepository) {

                this.customerRepository = customerRepository;

        }

     --------------

}
```

@Autowired on Properties

Let us now understand the usage of @Autowired on a property in Spring.

We will use @Autowired in the below code to wire the dependency of CustomerService class for CustomerRepository bean dependency.

```
package com.infy.service;

public class CustomerServiceImpl {

// CustomerService needs to contact CustomerRepository, hence injecting the

customerRepository dependency

        @Autowired

        private CustomerRepository customerRepository;

------------

}
```

@Autowired is by default wire the dependency based on the type of bean.

In the above code, the Spring container will perform dependency injection using the Java Reflection API. It will search for the class which implements CustomerRepository and injects its object. The dependencies which are injected using @Autowired should be available to the Spring container when the dependent bean object is created. If the container does not find a bean for autowiring, it will throw the NoSuchBeanDefinitionException exception.

If more than one beans of the same type are available in the container, then the framework throws an exception indicating that more than one bean is available for autowiring. To handle this **@Qualifier** annotation is used as follows:

```
package com.infy.service;

public class CustomerServiceImpl {

        @Autowired

      @Qualifier("custRepo")

        private CustomerRepository customerRepository;

      -----------

}
```

In Spring @Value annotation is used to insert values into variables and method arguments. Using @Value we can either read spring environment variables or system variables.

We can assign a default value to a class property with @Value annotation:

```
public class CustomerDTO {

        @Value("1234567891")

        long phoneNo;

        @Value("Jack")

        String name;
        @Value("Jack@xyz.com")

        String email;

        @Value("ANZ")

        String address;

}
```

Note that it accepts only a String argument but the passed-in value gets converted to an appropriate type during value-injection.

To read a Spring environment variable we can use @Value annotation:

```
public class CustomerDTO {

        @Value("${value.phone}")

        long phoneNo;

        @Value("${value.name}")

        String name;

        @Value("${value.email}")

        String email;

        @Value("${value.address}")

        String address;

}
```

Demo On @Autowiring Spring Boot

Highlights:

Objective: To understand the Autowiring in Spring

Demo Steps:

Demo6Application .java

```
package com.infy;

import java.util.List;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.support.AbstractApplicationContext;

import com.infy.dto.CustomerDTO;

import com.infy.service.CustomerServiceImpl;

@SpringBootApplication

public class Demo6Application {

        public static void main(String[] args) {

                CustomerServiceImpl service = null;

                AbstractApplicationContext   context   =   (AbstractApplicationContext)
SpringApplication.run(Demo6Application.class,

                                args);

                service = (CustomerServiceImpl) context.getBean("customerService");

                List<CustomerDTO> listcust = service.fetchCustomer();

                System.out.println("PhoneNumer" + "   " + "Name" + "   " + "Email" + "      "
+ "Address");

                for (CustomerDTO customerDTO2 : listcust) {

                        System.out.format("%5d%10s%20s%10s",
customerDTO2.getPhoneNo(), customerDTO2.getName(),

                                        customerDTO2.getEmail(),
customerDTO2.getAddress());
```

```
                    System.out.println();
                }
            }
}
```

CustomerService.java

```
package com.infy.service;
import java.util.List;
import com.infy.dto.CustomerDTO;
public interface CustomerService {
        public String createCustomer(CustomerDTO customerDTO);
        public List<CustomerDTO> fetchCustomer();
}
```

CustomerServiceImpl.java

```
package com.infy.service;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.infy.dto.CustomerDTO;
import com.infy.repository.CustomerRepository;
@Service("customerService")
public class CustomerServiceImpl implements CustomerService{

        @Autowired
        private CustomerRepository customerRepository;
```

```
	public String createCustomer(CustomerDTO customerDTO) {
		customerRepository.createCustomer(customerDTO);
		return "Customer with " + customerDTO.getPhoneNo() + " added successfully";
	}

	public List<CustomerDTO> fetchCustomer() {
		return customerRepository.fetchCustomer();
	}
}
```

CustomerRepository.java

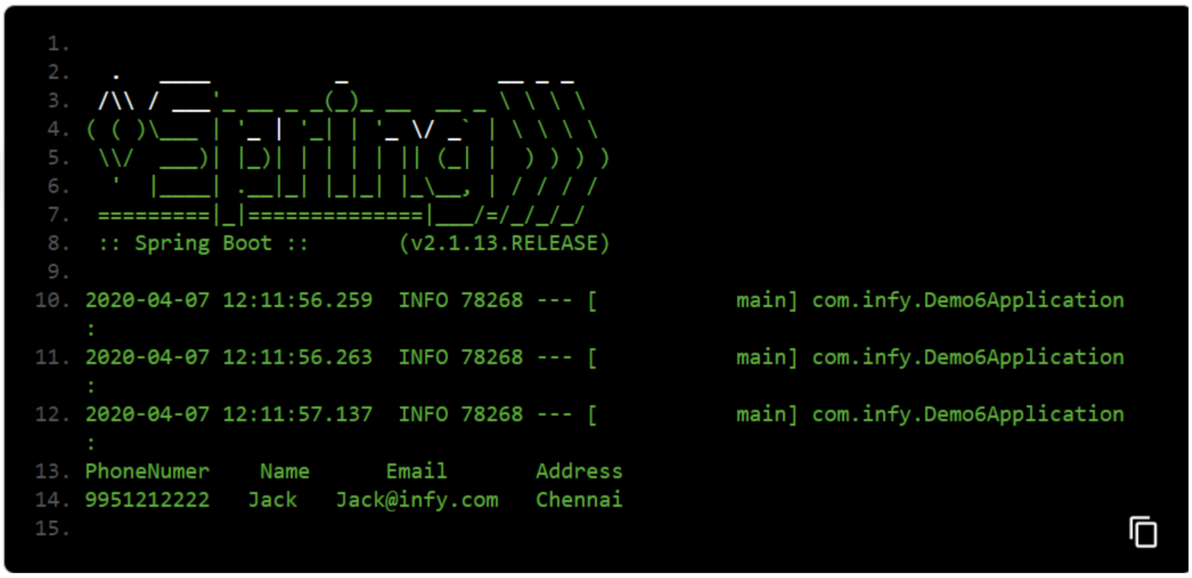
```
package com.infy.repository;
import java.util.ArrayList;
import java.util.List;
import javax.annotation.PostConstruct;
import org.springframework.stereotype.Repository;
import com.infy.dto.CustomerDTO;
@Repository
public class CustomerRepository {
	List<CustomerDTO> customers = null;
//Equivalent/similar to constructor. Here, populates the DTOs in a hard-coded way
	@PostConstruct
	public void initializer()
	{
		CustomerDTO customerDTO = new CustomerDTO();
		customerDTO.setAddress("Chennai");
		customerDTO.setName("Jack");
		customerDTO.setEmail("Jack@infy.com");
		customerDTO.setPhoneNo(9951212222l);
		customers = new ArrayList<>();
		customers.add(customerDTO);
	}

	//adds the received customer object to customers list
	public void createCustomer(CustomerDTO customerDTO)
	{
		customers.add(customerDTO);
	}

	//returns a list of customers
	public List<CustomerDTO> fetchCustomer()
	{
		return customers;
	}
}
```

CustomerDTO.java

```
package com.infy.dto;
public class CustomerDTO {
	long phoneNo;
	String name;
```

```
}

String email;

String address;

public long getPhoneNo() {

        return phoneNo;

}

public void setPhoneNo(long phoneNo) {

        this.phoneNo = phoneNo;

}

public String getName() {

        return name;

}

public void setName(String name) {

        this.name = name;

}

public String getEmail() {

        return email;

}

public void setEmail(String email) {

        this.email = email;

}

public String getAddress() {

        return address;

}

public void setAddress(String address) {

        this.address = address;
        }

        public CustomerDTO(long phoneNo, String name, String email, String address)
{

                this.phoneNo = phoneNo;

                this.name = name;

                this.email = email;

                this.address = address;

        }

        public CustomerDTO() {

        }

}
```

OUTPUT:



```

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::       (v2.1.13.RELEASE)

2020-04-07 12:11:56.259  INFO 78268 --- [           main] com.infy.Demo6Application
:
2020-04-07 12:11:56.263  INFO 78268 --- [           main] com.infy.Demo6Application
:
2020-04-07 12:11:57.137  INFO 78268 --- [           main] com.infy.Demo6Application
:
PhoneNumer    Name      Email        Address
9951212222    Jack    Jack@infy.com    Chennai

```

Scope of a bean

The lifetime of a bean depends on its scope. Bean's scope can be defined while declaring it in the configuration metadata file.

A bean can be in singleton or prototype scope. A bean with the "**singleton**" scope is initialized during the container starts up and the same bean instance is provided for every bean request from the application. However, in the case of the **"prototype"** scope, a new bean instance is created for every bean request from the application.

Let us now understand about bean scope in detail.

There will be a single instance of "singleton" scope bean in the container and the same bean is given for every request from the application.

The bean scope can be defined for a bean using @Scope annotation in Java class. By default, the scope of a bean is the singleton

```
package com.infy.service;

@Service("customerService")

@Scope("singleton")

public class CustomerServiceImpl implements CustomerService {

        @Value("10")

        private int count;

        public int getCount() {

                return count;

        }

        public void setCount(int count) {

                this.count = count;

        }

        public String fetchCustomer() {

                return " The number of customers fetched are : " + count;
        }

}
```

For the "prototype" bean, there will be a new bean created for every request from the application.

In the below example, customerService bean is defined with prototype scope. There will be a new customerService bean created for every bean request from the application.

```
package com.infy.service;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Service;

@Service("customerService")

@Scope("prototype")

public class CustomerServiceImpl implements CustomerService {

        @Value("10")

        private int count;

        public int getCount() {

                return count;

        }

        public void setCount(int count) {

                this.count = count;
```

```
    }

    public String fetchCustomer() {

        return " The number of customers fetched are : " + count;

    }

}
```

Demo: Scope of a bean Spring Boot

Highlights:

Objective: To understand the Singleton scope of a bean

Demo Steps:

Customerservice.java

```
package com.infy.service;

public interface CustomerService {

    public String fetchCustomer();

}
```

CustomerserviceImpl.java

```
package com.infy.service;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Service;
```

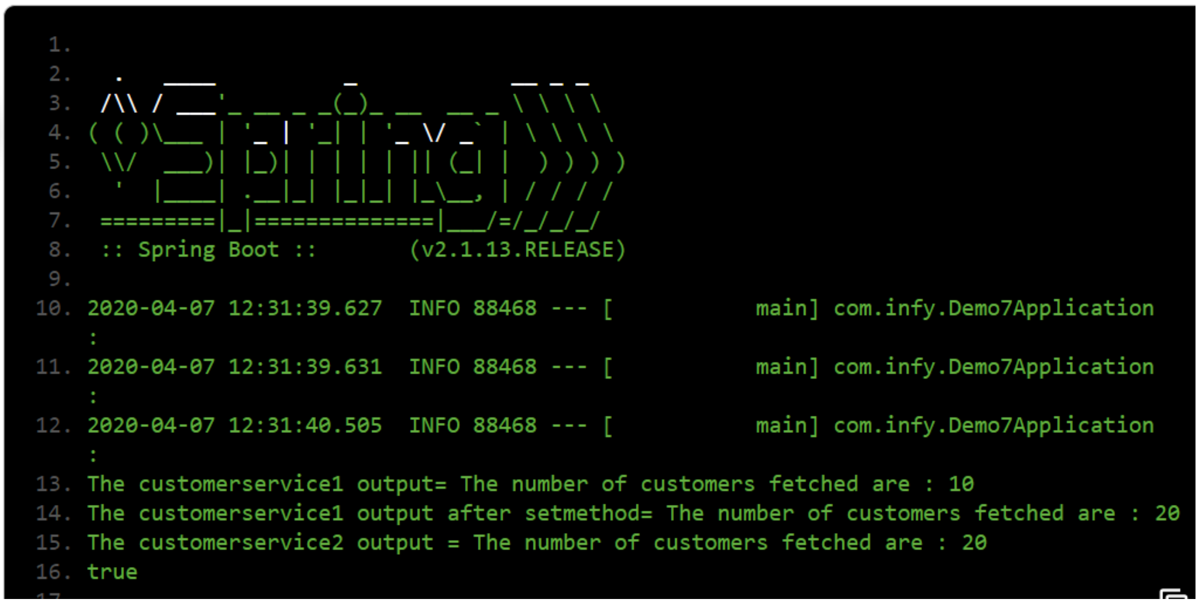
```
@Service("customerService")

@Scope("singleton")

public class CustomerServiceImpl implements CustomerService {

	@Value("10")

	private int count;

	public int getCount() {

		return count;

	}

	public void setCount(int count) {

		this.count = count;

	}

	public String fetchCustomer() {

		return " The number of customers fetched are : " + count;

	}

}
```

Demo7Application .java

```
package com.infy;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import
org.springframework.context.support.AbstractApplicationContext;

import com.infy.service.CustomerServiceImpl;

@SpringBootApplication

public class Demo7Application {

	public static void main(String[] args) {

		AbstractApplicationContext context =
(AbstractApplicationContext)
SpringApplication.run(Demo7Application.class,

				args);

		CustomerServiceImpl service1 = (CustomerServiceImpl)
context.getBean("customerService");

		System.out.println("The customerservice1 output=" +
service1.fetchCustomer());

		service1.setCount(20);

		System.out.println("The customerservice1 output after
setmethod=" + service1.fetchCustomer());

		CustomerServiceImpl service2 = (CustomerServiceImpl)
context.getBean("customerService");
```

```
                System.out.println("The customerservice2 output =" +
service2.fetchCustomer());

                System.out.println(service1==service2);

                context.close();

        }

}
```

Output



```

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::       (v2.1.13.RELEASE)

2020-04-07 12:31:39.627  INFO 88468 --- [           main] com.infy.Demo7Application
:
2020-04-07 12:31:39.631  INFO 88468 --- [           main] com.infy.Demo7Application
:
2020-04-07 12:31:40.505  INFO 88468 --- [           main] com.infy.Demo7Application
:
The customerservice1 output= The number of customers fetched are : 10
The customerservice1 output after setmethod= The number of customers fetched are : 20
The customerservice2 output = The number of customers fetched are : 20
true
```

Logger

What is logging?

Logging is the process of writing log messages to a central location during the execution of the program. That means Logging is the process of tracking the execution of a program, where

- Any event can be logged based on the interest to the
- When exception and error occurs we can record those relevant messages and those logs can be analyzed by the programmer later

There are multiple reasons why we may need to capture the application activity.

- Recording unusual circumstances or errors that may be happening in the program
- Getting the info about what's going in the application

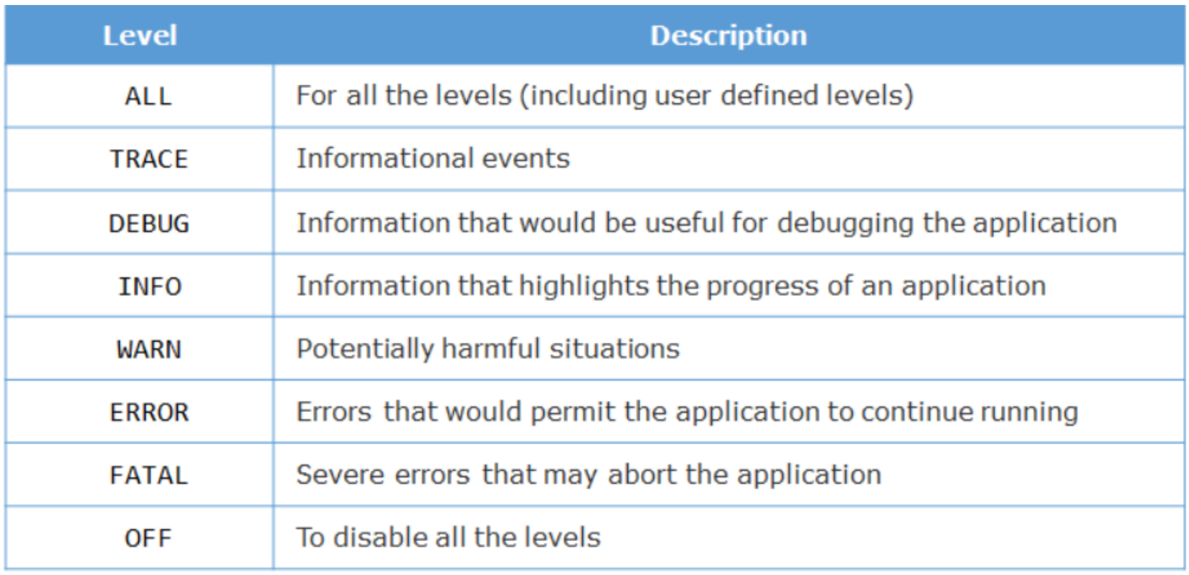
There are several logging APIs to make logging easier. Some of the popular ones are:

- JDK Logging API
- Apache Log4j
- Commons Logging API

The Logger is the object which performs the logging in applications.

Levels in the logger specify the severity of an event to be logged. The logging level is decided based on necessity. For example, TRACE can be used during development and ERROR during deployment.

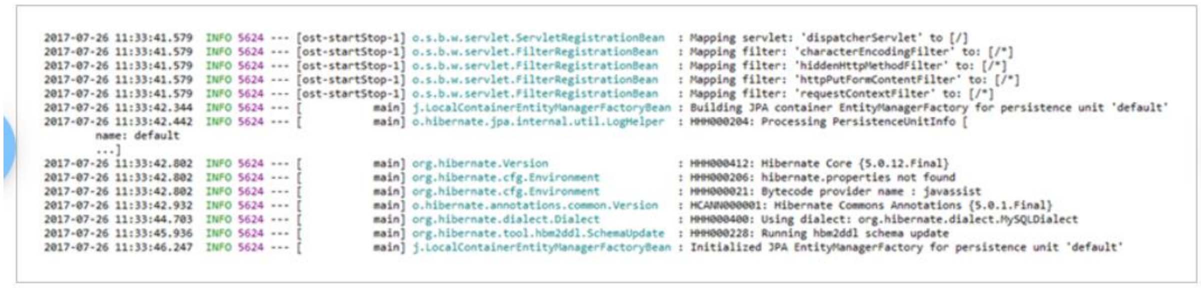
The following table shows the different levels of logging.



Level	Description
ALL	For all the levels (including user defined levels)
TRACE	Informational events
DEBUG	Information that would be useful for debugging the application
INFO	Information that highlights the progress of an application
WARN	Potentially harmful situations
ERROR	Errors that would permit the application to continue running
FATAL	Severe errors that may abort the application
OFF	To disable all the levels

You know that logging is one of the important activities in any application. It helps in quick problem diagnosis, debugging, and maintenance. Let us learn the logging configuration in Spring Boot.

While executing your Spring Boot application, have you seen things like the below getting printed on your console?



```
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean  : Mapping servlet: 'dispatcherServlet' to [/]
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingFilter' to: [/*]
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'httpPutFormContentFilter' to: [/*]
2017-07-26 11:33:41.579  INFO 5624 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter' to: [/*]
2017-07-26 11:33:42.344  INFO 5624 --- [           main] j.LocalContainerEntityManagerFactoryBean : Building JPA container EntityManagerFactory for persistence unit 'default'
2017-07-26 11:33:42.442  INFO 5624 --- [           main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [
        name: default
        ...]
2017-07-26 11:33:42.802  INFO 5624 --- [           main] org.hibernate.Version                    : HHH000412: Hibernate Core {5.0.12.Final}
2017-07-26 11:33:42.802  INFO 5624 --- [           main] org.hibernate.cfg.Environment            : HHH000206: hibernate.properties not found
2017-07-26 11:33:42.802  INFO 5624 --- [           main] org.hibernate.cfg.Environment            : HHH000021: Bytecode provider name : javassist
2017-07-26 11:33:42.932  INFO 5624 --- [           main] o.hibernate.annotations.common.Version   : HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
2017-07-26 11:33:44.703  INFO 5624 --- [           main] org.hibernate.dialect.Dialect            : HHH000400: Using dialect: org.hibernate.dialect.MySQLDialect
2017-07-26 11:33:45.936  INFO 5624 --- [           main] org.hibernate.tool.hbm2ddl.SchemaUpdate  : HHH000228: Running hbm2ddl schema update
2017-07-26 11:33:46.247  INFO 5624 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
```

Do you have any guess what are these?

Yes, you are right. These are logging messaged logged on the **INFO level.** However, you haven't written any code for logging in to your application. Then who does this?

By default, Spring Boot configures logging via **Logback** to log the activities of libraries that your application uses.

As a developer, you may want to log the information that helps in quick problem diagnosis, debugging, and maintenance. So, let us see how to customize the default logging configuration of Spring Boot so that your application can log the information that you are interested in and in your own format.

Have you realized that you have not done any of the below activities for logging which you typically do in any Spring application?

- Adding dependent jars for logging
- Configuring logging through Java configuration or XML configuration

Still, you are able to log your messages. The reason is Spring Boot's default support for logging. The spring-boot-starter dependency includes spring-boot-starter-logging dependency, which configures logging via Logback to log to the console at the INFO level.

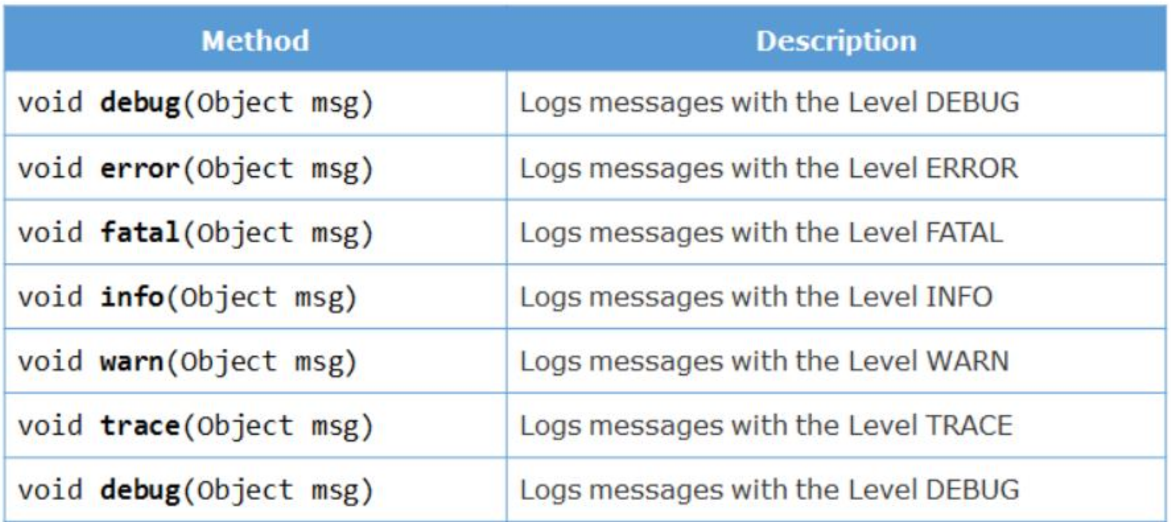
Spring Boot uses Commons Logging API with default configurations for Java Util Logging, Log4j 2, and Logback implementation. Among these implementations, Logback configuration will be enabled by default.

You, as a developer, have just created an object for Logger and raise a request to log with your own message in LoggingAspect.java as shown below.

```
public class CustomerServiceImpl implements CustomerService

{

private static Logger logger = LoggerFactory.getLogger(CustomerServiceImpl.class);

public void deleteCustomer(long phoneNumber) {
```

```
                public void deleteCustomer(long phoneNumber) {

                try {

                        customerRepository.deleteCustomer(phoneNumber);

                } catch (Exception e) {

                        logger.info("In log Exception ");

                        logger.error(e.getMessage(),e);

                }

        }

}
```

Apart from info(), the Logger class provides other methods for logging information:

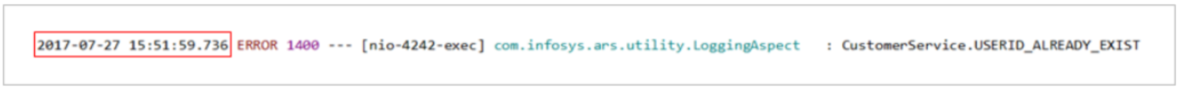


Method	Description
`void debug(Object msg)`	Logs messages with the Level DEBUG
`void error(Object msg)`	Logs messages with the Level ERROR
`void fatal(Object msg)`	Logs messages with the Level FATAL
`void info(Object msg)`	Logs messages with the Level INFO
`void warn(Object msg)`	Logs messages with the Level WARN
`void trace(Object msg)`	Logs messages with the Level TRACE
`void debug(Object msg)`	Logs messages with the Level DEBUG

The default log output contains the following information.

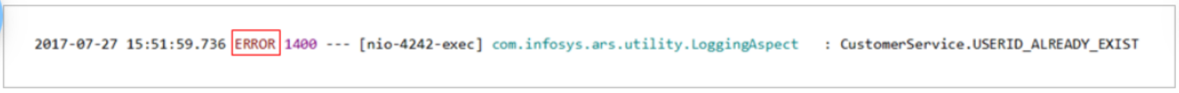
Date and Time

Date and Time



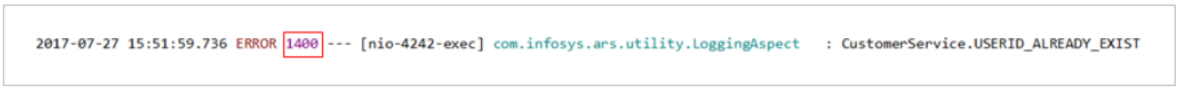
```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

Log level



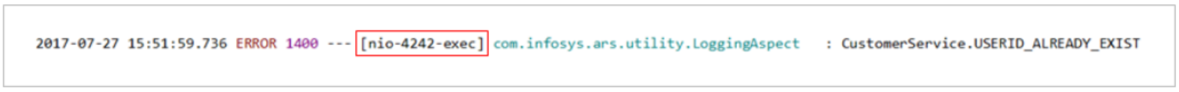
```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

Process id



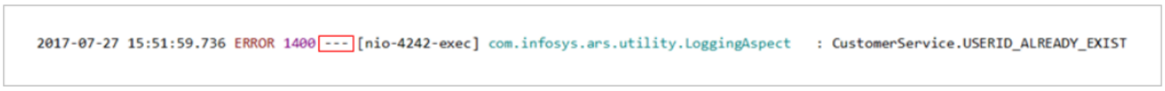
```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

Thread name



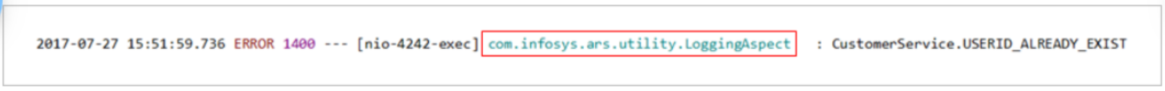
```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

Separator



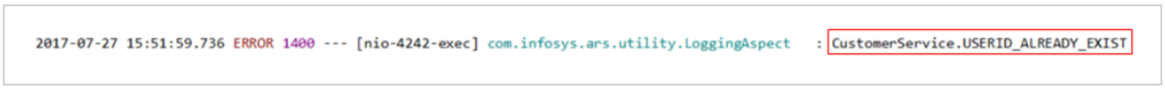
```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

Logger name



```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

Log message



```
2017-07-27 15:51:59.736 ERROR 1400 --- [nio-4242-exec] com.infosys.ars.utility.LoggingAspect    : CustomerService.USERID_ALREADY_EXIST
```

But, how to change this default configuration if you want to,

- log the message in a file rather than console
- log the message in your own pattern
- log the messages of a specific level
- use Log4j instead of Logback

Log into file

By default Spring Boot logs the message on the console. To log into a file, you have to include either logging.file or logging. path property in your application.properties file.

Note: Please note that from Spring boot 2.3.X version onwards logging.file and logging.path has been deprecated we should use "logging.file.name" and " logging.file.path" for the same.

Custom log pattern

Include logging.pattern.* property in application.properties file to write the log message in your own format.

Logging property	Sample value	Description
logging.pattern.console	%d{yyyy-MM-dd HH:mm:ss,SSS}	Specifies the log pattern to use on the console
logging.pattern.file	%5p [%t] %c [%M] - %m%n	Specifies the log pattern to use in a file

Demo: Logger

Highlights:

Objective: To implement Logging

Demo Steps:

CustomerService.java

```
package com.infy.service;

import com.infy.dto.CustomerDTO;

public interface CustomerService {

        public String createCustomer(CustomerDTO dto);

        public String fetchCustomer();

        public void deleteCustomer(long phoneNumber) throws Exception;

}
```

CustomerServiceImpl.java

```
package com.infy.service;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.infy.dto.CustomerDTO;

import com.infy.repository.CustomerRepository;

@Service("customerService")

public class CustomerServiceImpl implements CustomerService {

	private static Logger logger =
LoggerFactory.getLogger(CustomerServiceImpl.class);

	@Autowired

	private CustomerRepository customerRepository;


	@Override

	public String createCustomer(CustomerDTO dto) {

		return customerRepository.createCustomer(dto);

	}

	@Override

	public String fetchCustomer() {

		return customerRepository.fetchCustomer();

	}

   @Override

	public void deleteCustomer(long phoneNumber) {

		try {

			customerRepository.deleteCustomer(phoneNumber);

		} catch (Exception e) {

			logger.info("In log Exception ");

			logger.error(e.getMessage(),e);
		}
	}
}
```

CustomerRepository.java

```
package com.infy.repository;

import java.util.ArrayList;

import java.util.List;

import javax.annotation.PostConstruct;

import org.springframework.stereotype.Repository;

import com.infy.dto.CustomerDTO;

@Repository

public class CustomerRepository {

	@PostConstruct

	public void initializer()

	{

		CustomerDTO customerDTO = new CustomerDTO();

		customerDTO.setAddress("Chennai");

		customerDTO.setName("Jack");

		customerDTO.setEmail("Jack@infy.com");

		customerDTO.setPhoneNo(9951212222l);

		customers = new ArrayList<>();

		customers.add(customerDTO);

	}

	List <CustomerDTO> customers=null;

	public String createCustomer(CustomerDTO dto) {
```

```
            customers = new ArrayList<>();

            customers.add(dto);

            return "Customer added successfully"+customers.indexOf(dto);

        }

        public String fetchCustomer()  {

            return " The customer fetched  "+customers;

        }

        public void deleteCustomer(long phoneNumber) throws Exception

        {


            for(CustomerDTO customer : customers)

            {

                if(customer.getPhoneNo() == phoneNumber)

                {

                    customers.remove(customer);

                    System.out.println(customer.getName()+"of
phoneNumber"+customer.getPhoneNo()+"\t got deleted successfully");

                    break;

                }

                else

                    throw new Exception("Customer does not exist");

            }


        }

    }
```

CustomerDTO.java

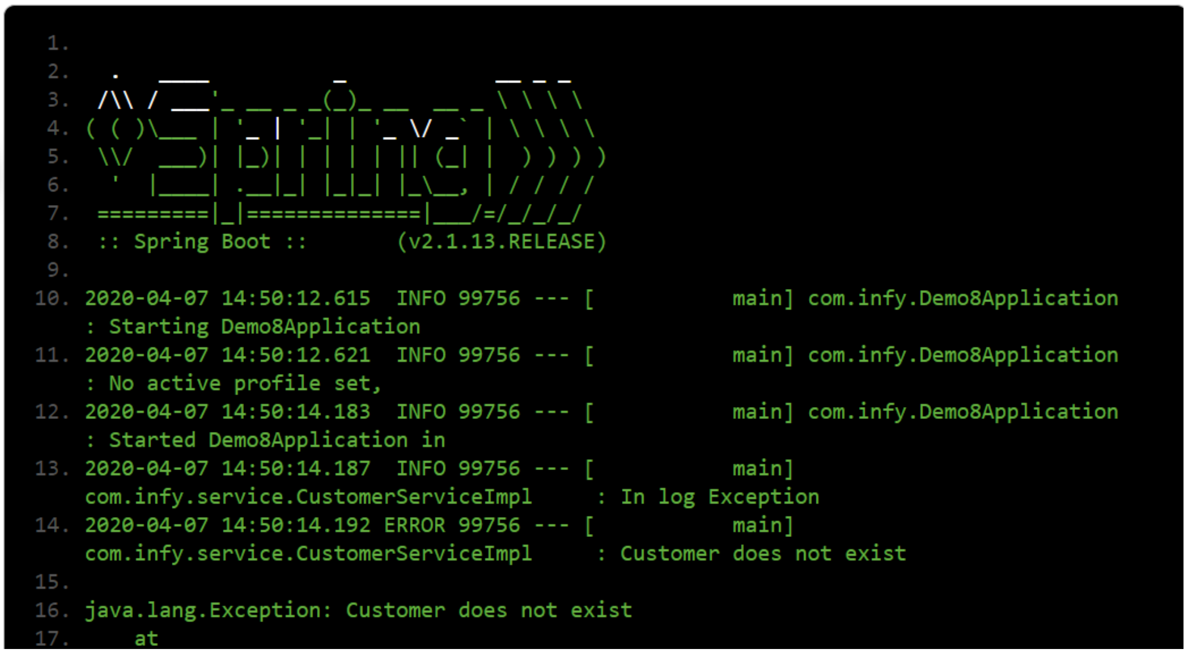
```java
package com.infy.dto;

public class CustomerDTO {

	long phoneNo;

	String name;

	String email;

	String address;

	public long getPhoneNo() {

		return phoneNo;

	}

	public void setPhoneNo(long phoneNo) {

		this.phoneNo = phoneNo;

	}

	public String getName() {

		return name;

	}

	public void setName(String name) {

		this.name = name;

	}

	public String getEmail() {

		return email;

	}

	public void setEmail(String email) {

		this.email = email;

	}
```

```
        public String getAddress() {
                return address;
        }
        public void setAddress(String address) {
                this.address = address;
        }
        public CustomerDTO(long phoneNo, String name, String email, String address)
{
                this.phoneNo = phoneNo;
                this.name = name;
                this.email = email;
                this.address = address;
        }
        public CustomerDTO() {
        }
}
```

Demo8Application.java

```
package com.infy;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.support.AbstractApplicationContext;
import com.infy.service.CustomerServiceImpl;
@SpringBootApplication
public class Demo8Application {
        public static void main(String[] args) {
                CustomerServiceImpl service = null;

                AbstractApplicationContext context = (AbstractApplicationContext)
SpringApplication.run(Demo8Application.class,

                                args);



                service = (CustomerServiceImpl) context.getBean("customerService");

                service.deleteCustomer(1151212222l);

                // service.deleteCustomer(9951212222l);

        }

}
```

Output:



```

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::       (v2.1.13.RELEASE)

2020-04-07 14:50:12.615  INFO 99756 --- [           main] com.infy.Demo8Application
: Starting Demo8Application
2020-04-07 14:50:12.621  INFO 99756 --- [           main] com.infy.Demo8Application
: No active profile set,
2020-04-07 14:50:14.183  INFO 99756 --- [           main] com.infy.Demo8Application
: Started Demo8Application in
2020-04-07 14:50:14.187  INFO 99756 --- [           main]
com.infy.service.CustomerServiceImpl     : In log Exception
2020-04-07 14:50:14.192 ERROR 99756 --- [           main]
com.infy.service.CustomerServiceImpl     : Customer does not exist

java.lang.Exception: Customer does not exist
    at
```