Spring Dec22

Chapter 1: Introduction

Link: https://spring.io/projects

Download:

1. Download Eclipse
2. Download Maven
3. Set the path of Java and Maven

Eclipse Download: https://www.eclipse.org/downloads/

Maven Download: https://maven.apache.org/download.cgi

Files

Maven is distributed in several formats for your convenience. Simply pick a ready-made binary distribution archive and source archive if you intend to build Maven yourself.

In order to guard against corrupted downloads/installations, it is highly recommended to verify the signature of the relea by the Apache Maven developers.

	Link	Checksums
Binary tar.gz archive	apache-maven-3.8.6-bin.tar.gz	apache-maven-3.8.6-bin.tar.gz.sha512
Binary zip archive	apache-maven-3.8.6-bin.zip	apache-maven-3.8.6-bin.zip.sha512
Source tar.gz archive	apache-maven-3.8.6-src.tar.gz	apache-maven-3.8.6-src.tar.gz.sha512
Source zip archive	apache-maven-3.8.6-src.zip	apache-maven-3.8.6-src.zip.sha512

Path Setup:
JAVA_HOME=<JDK Home directory>
M2_HOME =<Maven Home Directory>
//M2=%M2_HOME%\bin
PATH=%M2%

Test Setup:
Java:
Go to Command prompt: java -version

Maven: mvn --version

Spring maven dependencies:

```
<dependencies>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-core</artifactId>
                        <version>${spring.version}</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-context</artifactId>
                        <version>${spring.version}</version>
                </dependency>
        </dependencies>

        <properties>
                <spring.version>5.1.1.RELEASE</spring.version>
        </properties>
```

spring configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">


</beans>
```

Spring configuration file:

1. **XML file**
2. **Multiple XML file**

   XML file 1:

spring.xml

```
   <bean name="address" class="com.easylearning.entity.Address" lazy-init="true">
                <property name="city" value="pune"></property>
                <property name="pinCode" value ="411021"/>
   </bean>
   <bean name="employee" class="com.easylearning.entity.Employee" scope="prototype">
                <property name="id" value="1"/>
                <property name="name" value="Jack"/>
                <property name="salary" value="5000"></property>
```

```
    </bean>
```

XML file 2:

spring1.xml

```
    <bean name="address" class="com.easylearning.entity.Address" lazy-init="true">
                <property name="city" value="pune"></property>
                <property name="pinCode" value ="411021"/>
    </bean>
```

spring.xml

```
    <import resource="spring1.xml"/>
    <bean name="employee" class="com.easylearning.entity.Employee" scope="prototype">
                <property name="id" value="1"/>
                <property name="name" value="Jack"/>
                <property name="salary" value="5000"></property>
    </bean>
```

3. **Java Config file**

```
        @Bean
        public Address address() {
                Address address = new Address("Pune", "411021");
                return address;
        }
        @Bean
        public Employee employee() {
                Employee emp  =  new Employee(1, "jack", 1000);
                emp.setAddress(address());
                return emp;
        }
```

4. **Multiple config file**

Config1

```
public class Config1 {
        @Bean
        public Address address() {
                Address address = new Address("Pune", "411021");
                return address;
        }
}
```

Config2

```
@Import(value = Config1.class)
public class Config2 {
        @Bean
        public Employee employee(Address address) {
                Employee emp  =  new Employee(1, "jack", 1000);
                emp.setAddress(address);
                return emp;
        }
}
```

5. **Combination of Java config file and xml file**

XML file.

```
 <bean name="address" class="com.easylearning.entity.Address" >
      <property name="city" value="pune"></property>
     <property name="pinCode" value ="411021"/>
  </bean>
```

Java Config file:

```
@ImportResource(value = "classpath:spring.xml")
public class Config2 {
        @Bean
        public Employee employee(Address address) {
                Employee emp  =  new Employee(1, "jack", 1000);
                emp.setAddress(address);
                return emp;
        }
}
```

Create ApplicationContext

```
1. ApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");
2. ApplicationContext context =new AnnotationConfigApplicationContext(Config.class);
3. ApplicationContext context = new FileSystemXmlApplicationContext(absolutepath);
```

Create BeanFactory

```
1. ClassPathResource resource = new ClassPathResource("spring.xml");
   BeanFactory factory = new XmlBeanFactory(resource);
```

```
2. Resource resource = new FileSystemResource(path);
   BeanFactory factory = new XmlBeanFactory(resource);
```

Type of Injection

1. **Setter Injection**

```
   <bean name="address" class="com.easylearning.entity.Address" >
        <property name="city" value="pune"></property>
        <property name="pinCode" value ="411021"/>
   </bean>
   <bean name="employee" class="com.easylearning.entity.Employee">
        <property name="id" value="1"/>
        <property name="name" value="Jack"/>
        <property name="salary" value="5000"/>
        <property name="address" ref="address" /> <!--setter injection-->
   </bean>
```

2. **Constructor Injection**

```
  <bean name="address" class="com.easylearning.entity.Address" >
        <property name="city" value="pune"></property>
        <property name="pinCode" value ="411021"/>
   </bean>
   <bean name="employee" class="com.easylearning.entity.Employee">
        <constructor-arg name="address" ref="address" /> <!--constructor injection-->
        <property name="id" value="1"/>
        <property name="name" value="Jack"/>
        <property name="salary" value="5000"/>
   </bean>
```

3. **Method Injection**

If we want to create the different object of the bean but contained object must be different object then we should do method injection.

```
public class TicketVendorMachine{
        private Ticket ticket;
        public TicketVendorMachine() {}
        public void setTicket(Ticket ticket) {
                this.ticket = ticket;
        }
        public Ticket getTicket() {
                return this.ticket;
        }
}

public class Ticket {
        private static int ticketNoGenerator;
        private int ticketNo;
```

```
        public Ticket() {
                ticketNo = ++ticketNoGenerator;
        }
        public int getTicketNo() {
                return ticketNo;
        }
        public void setTicketNo(int ticketNo) {
                this.ticketNo = ticketNo;
        }
}
/*
In above class if we set the scope of TicketVendorMachine to singleton and Ticket to
prototype, TikcetVendorMachine will give the same ticket. In order to avoid this case,
we need do to method injection.
*/
/*spring.xml*/
---------------
         <bean name="ticket" class="com.easylearning.entity.Ticket" scope="prototype">
         </bean>
         <bean name="machine" class="com.easylearning.entity.TicketVendorMachine"
scope="singleton">
                <property name="ticket"  ref="ticket"/>
         </bean>

 Config.clas
------------
        @Scope("prototype")
        @Bean
        public Ticket ticket() {
                return  new Ticket();
        }

        @Scope("singleton")
        @Bean
        public TicketVendorMachine machine() {
                return new TicketVendorMachine(ticket());
        }

Main.java
------------
TicketVendorMachine machine1 = context.getBean("machine", TicketVendorMachine.class);
                TicketVendorMachine machine2 = context.getBean("machine",
TicketVendorMachine.class);
                System.out.println(machine1 == machine2); //true
                System.out.println(machine1.getTicket() == machine2.getTicket());
//true but we want different object of ticket
```

```
public abstract class TicketVendorMachine{
```

```
        private Ticket ticket;
        public TicketVendorMachine() {  }
        public void setTicket(Ticket ticket) {
                this.ticket = ticket;
        }
        public abstract Ticket getTicket();
}

spring.xml
-----------
         <bean name="ticket" class="com.easylearning.entity.Ticket" scope="prototype">
         </bean>
         <bean name="machine" class="com.easylearning.entity.TicketVendorMachine"
scope="singleton">
                <lookup-method name="getTicket"  bean="ticket"/>
         </bean>

Config.java
------------
@Configuration
public class ConfigNew {
        @Scope("prototype")
        @Bean
        public Ticket ticket() {
                return  new Ticket();
        }
        @Bean
        public TicketVendorMachine machine() {
                return new TicketVendorMachine() {
                        @Override
                        public Ticket getTicket() {
                                return ticket();
                        }
                };
        }
}

Main.java
------------
TicketVendorMachine machine1 = context.getBean("machine", TicketVendorMachine.class);
                TicketVendorMachine machine2 = context.getBean("machine",
TicketVendorMachine.class);
                System.out.println(machine1 == machine2); //true
                System.out.println(machine1.getTicket() == machine2.getTicket());
//false
```

4. **Factory Injection**

```
public class Server {
        private boolean authenticated;
```

```
        public Server() {
        }
        public Server(boolean authenticated) {
                this.authenticated = authenticated;
        }
        public static Server getServer(int appId) { //get server is the factory method
                return new Server(validAppId(appId));
        }
        public static boolean validAppId(int appId) {
                // business logic to verify
                if (appId <= 100) {
                        return true;
                }
                return false;
        }
        public boolean isAuthenticated() {
                return authenticated;
        }
        public void setAuthenticated(boolean authenticated) {
                this.authenticated = authenticated;
        }
}
```

Scope of Bean

1. **singleton -** One Instance per spring context (default)
2. **prototype-** New bean when requested
3. **request -** one bean per Http request
4. **session -** one bean per Http session

Spring-Core

- **IOC Container**
- **Application Context**
- **Bean Factory**

Application Context:

- All the functionality of BeanFactory
- Spring AOP feature
- I18n
- WebApplicationContext for web applications

Lifecycle of Spring:

- instantiate
- set state(inject dependencies)
- Aware interface callback Methods

```
@Scope(scopeName = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
@Component
public class Address implements BeanNameAware, ApplicationContextAware {
       //properties declaration and setter,getter method

        System.out.println("setBeanName: "+name);
        }
        public void setApplicationContext(ApplicationContext applicationContext) throws
BeansException {
                System.out.println("setApplicationContext called...");
                System.out.println(new Date(applicationContext.getStartupDate()));
        }
}

//Example of ResourceLoaderAware(when want to laod the properties file)
public class SampleBean implements ResourceLoaderAware {
        private ResourceLoader resourceLoader;
        public void setResourceLoader(ResourceLoader resourceLoader) {
                this.resourceLoader = resourceLoader;
        }
        public void readResource() throws IOException {
                Resource resource =
resourceLoader.getResource("classpath:test.properties");
                //Resource resource =
resourceLoader.getResource("file:e:/abc/test.properties");
                InputStream inputStream = resource.getInputStream();
                Properties prop =  new Properties();
                prop.load(inputStream);
                //prop object loaded the data in properties file
        }
}
```

- Initializing Methods

```

```

- bean is ready to used
- destroy methods
- destroy

```
((ConfigurableApplicationContext)context).close();
```

AOP: A solution for Implementing the cross cutting concern.

- Logging
- Security
- Transaction

Basic Terms used in AOP

- **Advice:** (what & when) the job of an aspect
  - **@Before**
  - **@After**
  - **@AfterReturning**
  - **@AfterThrowing**
  - **@Around**
- **Jointpoint :** (where) a point in executio of the application where an aspect can be plugged-in
- **Pointcut:** One or more joinpoints at which advice should be woven
- **Aspect:** is a merger of advice & pointcut. What it does and when and where it does.
- **Introduction:** allows to add new methods or attributes to existing classes.
- **Proxy:** the object created after applying advice to the target object
- **Weaving:** the process of applying aspects to create a new proxied object

Spring AOP:

- Weaving process happens during run time. can cost application performance.
- uses proxy based AOP. Hence, aspect work works only on the method execution.

```
//execution(* package.*.*(..))
1st * = return type
2nd * = class name
3rd * = method name
..    = any number of parameter
```

```
package com.easylearning.service;
public class LoanService {
 public void issueLoan(String customerId) {
  System.out.println("Loan Issued to customer: " + customerId);
 }
 public void payDue(String customerId) {
  System.out.println("Pending due cleared: " + customerId);
 }
}

//Aspect class
public class LogBeforeAndAfter implements MethodBeforeAdvice, AfterReturningAdvice {
 @Override
 public void afterReturning(Object returnValue, Method method, Object[] args, Object
target) throws Throwable {
   System.out.println("Exit: " + target.getClass().getName()+ " : "+method.getName()+"
at " + LocalDateTime.now());
```

```
 }

 @Override
 public void before(Method method, Object[] args, Object target) throws Throwable {
  System.out.println("Entered: " + target.getClass().getName()+ " :
"+method.getName()+" at " + LocalDateTime.now());
 }
}

--spring.xml
<bean id="loanService" class="com.easylearning.service.LoanService" />
<bean id="logBeforeAndAfter" class="com.easylearning.service.LogBeforeAndAfter" />

<bean id="loanServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
 <property name="target" ref="loanService" />
 <property name="interceptorNames">
   <list>
    <value>logBeforeAndAfter</value>
   </list>
 </property>
</bean>
```

AspectJ

AspectJ is an AOP extension created by PARC(company) that promotes modularization

of crosscutting concerns such as logging, transaction, exception handling, security.

```
<dependency>
    <groupId>aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.5.4</version>
</dependency>
<dependency>
    <groupId>aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.5.4</version>
</dependency>
```

- Weaving process happens more during cokmpile time and less during runtime. Hence less impact on application performance
- gives more fine grained control on JOint Points
- Annotation driven

Spring JDBC:

Problem with Traditional JDBC

- It's redundant
- not follow DRY concept
- Bigger code.
- you find yourself doing the same things over and over again

- Maintenance is high
- closing the resources

How spring Handle above problems

- Implicit access to resources
- Many operations become one-liners
- No try/catch blocks
- Pre-built Integration classes
  - JDBC: JdbcTemplate
  - JDO: JdoTemplate
  - Hibernate: HibernateTemplate
  - MyIbatis: SqlMapClientTemplate
  - many more

Data Source by JDBC driver:

- DriverManagerDataSource : Return a new connection every time that a connection is requested.
- SingleConnectionDataSource : Returns the same connection every time that a connection is requested.

Maven dependencies

```
<dependency>
 <groupId>com.oracle.database.jdbc</groupId>
 <artifactId>ojdbc8</artifactId>
 <version>21.1.0.0</version>
</dependency>
<dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-jdbc</artifactId>
 <version>${spring.version}</version>
</dependency>
```

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.27</version>
</dependency>
```

Spring.xml

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="" />
  <property name="url" value="" />
  <property name="username" value="" />
  <property name="password" value="" />
</bean>
```

Transaction:

- Support Programmatic and declarative transaction
- Programmatic transaction management is achieved via
  - PlatformTransactionManager
  - TransactionTemplate
- Declarative transaction management achieved via
  - Spring's AOP
  - Annotations
- Supports many transaction properties
  - propagation
  - isolation level
  - Rollback Condition
  - Readonly
  - Timeout

Declarative Transaction Management API

- XML way
  - <aop:config>, <aop:pointcut>, <aop:advisor>
  - <tx:advice>
- Annotation way
  - @Transactional

Transaction Attributes

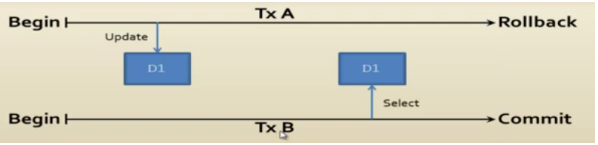
- Propagation
  - REQUIRED : if there's an existing transaction in progress, the current method should run within this transaction. Otherwise, it should start a new transaction and run within its own transaction.
  - REQUIRES_NEW: The current method must start a new transaction and run within its own transaction. if there's an existing transaction in progress, it should be suspended.
  - SUPPORTS : If there's an existing transaction in progress, the current method can run within this transaction. Otherwise, it is not necessary to run within a transaction.
  - NOT_SUPPORTED : The current method should not run within a transaction. If there's an existing transaction in progress, it should be suspended.

    - MANDATORY: the current method should not run within a transaction, it there's an existing transaction in progress, an exception will be thrown
    - NEVER
    - NESTED: if there's an existing transaction in progress, the current method should run within the nested transaction (supported by jdbc 3.0 savepoint feature) of the transaction. Otherwise it should start a new transaction and run within its own transaction
- Isolation : how much transaction is impacted with the activities of another concurrent transaction.
- Read-Only
- Timeout
- Rollback rules

Transaction Isolation Issues

- Lost Update
- Dirty read
- unrepeatable read
- Second lost update problem
- phantom reads

Dirty Read



Transaction-A updates a row but not commit

Transaction-B select the same row and reads the not yet committed updates

Transaction-A rollback undergoing its changes

Transaction-Ab is about to commit invalid data

Transaction problem can be removed by setting isolation level.

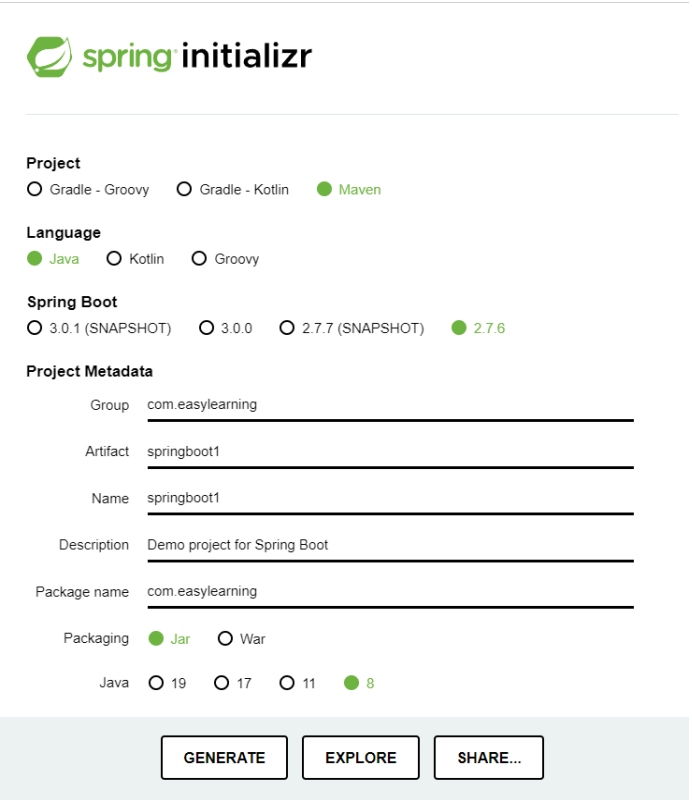
- **Read Uncommitted :** permits dirty read but not lost updates. One transaction may not write to a row if another uncommitted transaction has already written to it. Any transaction may read row. However, this isolation level may be implemented using exclusive write locks.
- **Read committed :** Permits unrepeatable reads but not dirty reads. This may be achieved using momentary shared read locks and exclusive write locks. Reading transactions don't block other transactions from accessing a row. However, an uncommitted writing transaction blocks all others transaction from accessing the row.
- **Repeatable read**: Permits neither unrepeatable read nor dirty reads. Phantom reads may occur. This may be achieved using shared read locks and exclusive write locks. Reading transactions block writing transactions (but not other reading transactions) and writing transaction block all others transactions.

- **Serializable**: Provides the strict transaction isolation. It emulates serial transaction execution, as if transactions had been executed one after another, serially only row-level locks. There must be another mechanism that prevents a newly inserted row from becoming visible to a transaction that has already executed a query that would return the row.

Spring Boot:

Link ref:https://start.spring.io/

Step 1:



Spring boot

1. Auto Configuration
   - Auto Configuration a data source for **Hibernate** jar on the classpath
   - Auto Configuration of **Dispatcher Servlet** in Spring MVC jar on the classpath
2. Built around well know patterns
3. spring-boot-starter-actuator : Use for monitoring & tracing the application

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

 <dependency>
      <groupId>org.springframework.data</groupId>
```

```
      <artifactId>spring-data-rest-hal-explorer</artifactId>
  </dependency>
```

URL: http://localhost:8081/actuator

http://localhost:8081/application

application.properties

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

4. spring-boot-starter-tomcat(jetty/undertow):
5. spring-boot-starter-logging- for logging using logback
6. spring-boot-starter-log4j2- Logging using Log4j2
7. Default Error Handling
8. Dev tools

```
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
      <optional>true</optional>
  </dependency>
```

Spring Boot Starter Project:
url: *http://start.spring.io/*
Type of Application

- [ ] spring-boot-starter-web — Web & RESTful application
- [ ] spring-boot-starter-web-services — SOAP web services
- [ ] spring-boot-starter-test — Unit testing and Integration testing
- [ ] spring-boot-starter-jdbc — Traditional JDBC
- [ ] spring-boot-starter-hateoas — HATEOAS feature
- [ ] spring-boot-starter-security — Authentication ans Authorization
- [ ] spring-boot-starter-data-jpa — Spring Data JPA with Hibernate
- [ ] spring-boot-starter-data-rest — Expose Simple test services using spring data Rest
- [ ] spring-boot-starter-cache — caching feature

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
</dependency>
```

```
<!-- https://mvnrepository.com/artifact/ch.qos.logback/logback-core -->
<dependency>
    <groupId>ch.qos.logback</groupId>
     <artifactId>logback-classic</artifactId>
</dependency>
```

application.properties

server.port=8081

logging.level.org.springframework=debug

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-aop</artifactId>
<version>2.2.2.RELEASE</version>
</dependency>

//execution(* package.*.*(..))
1st * = return type
2nd * = class name
3rd * = method name
..    = any number of parameter
```