#mvn -version

#mvn archetype:generate

'groupId': com.epam.brest.cource

'artifactId': sampleuserapp

**Fully qualified name (FQN):**

packagename.packagename2.packagename3.ClassName

JDK/JRE/ JAVAC(синтакис, лексика, семантика, оптимизация) -> ByteCode -> JVM

1. #java App - wrong. FQN is needed
2. #java com.epam.brest.cource.App - wrong, there is no `com.epam.brest.cource` folder in the current working directory
3. #java -classpath . com.epam.brest.cource.App - wrong
4. #java -cp ../../../../ com.epam.brest.cource.App
5. #java -cp absPath com.epam.brest.cource.App
6. echo Main-Class: com.epam.brest.cource.App > manifest.txt
7. jar cvfm App.jar manifest.txt com/epam/brest/cource/\*.class
8. javac com/epam/brest/cource/App.java
9. java -jar App.jar

Ant / Ivy / Maven / Gradle (Groovy)

**ANT (Example)**

<project name="MyProject" basedir=".">  
  
 <property name="version">1.2.3</property>  
 <property name="dir.src">src/main/java</property>  
 <property name="dir.build">build</property>  
 <property name="dir.build.classes">${dir.build}/classes</property>  
 <property name="dir.build.javadoc">${dir.build}/javadoc</property>  
 <property name="file.jar">${dir.build}/MyProject-${version}.jar</property>  
  
 <path id="projectClasspath">  
 <fileset dir="lib">  
 <include name="\*\*.jar"/>  
 </fileset>  
 </path>  
  
 <target name="clean">  
 <delete dir="${dir.build}"/>  
 </target>  
  
 <target name="init">  
 <mkdir dir="${dir.build}"/>  
 <mkdir dir="${dir.build.classes}"/>  
 </target>  
  
 <target name="compile" depends="init">  
 <echo>Compiling Java source</echo>  
 <javac classpathref="projectClasspath" srcdir="${dir.src}" destdir="${dir.build.classes}" />  
 </target>  
  
 <target name="jar" depends="compile">  
 <echo>Making JAR file</echo>  
 <jar basedir="${dir.build.classes}" file="${file.jar}" />  
 </target>  
  
 <target name="javadoc">  
 <echo>Making JavaDoc from source</echo>  
 <javadoc sourcepath="${dir.src}" destdir="${dir.build.javadoc}"></javadoc>  
 </target>  
  
</project>

**Gradle (Example)**

**apply plugin: "idea"  
apply plugin: "eclipse"  
  
subprojects {  
 apply plugin: "java"  
  
 tasks.withType(JavaCompile) {  
 sourceCompatibility = JavaVersion.VERSION\_1\_7  
 targetCompatibility = JavaVersion.VERSION\_1\_7  
 }  
  
 repositories {  
 mavenCentral()  
 }  
}  
  
task wrapper(type: Wrapper) {  
 gradleVersion = "1.12"  
}**

**MAVEN**

**POM** - Project Object Model

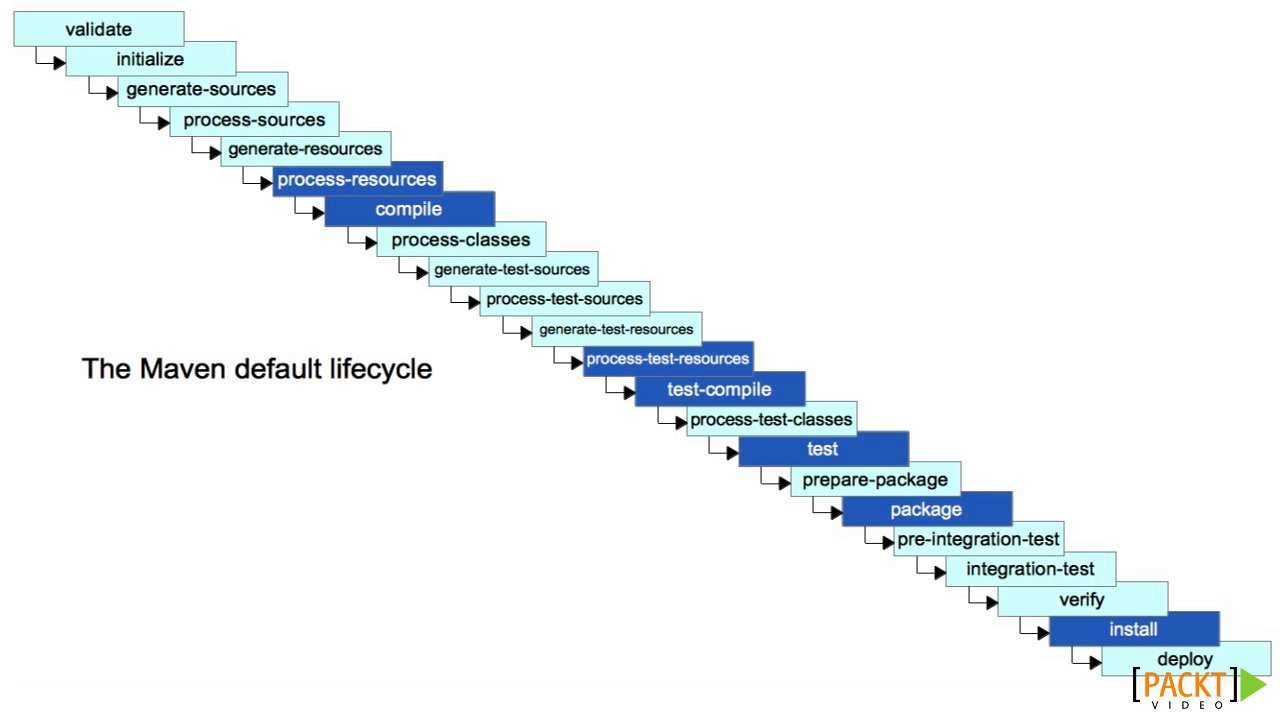
Концепции:

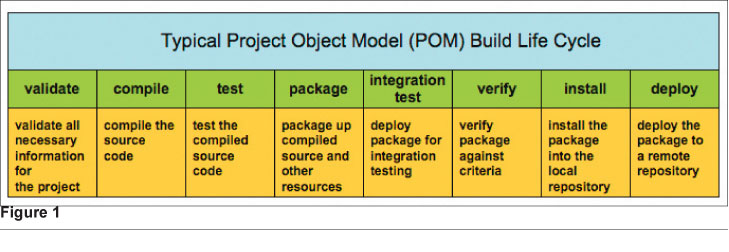
1. Inheritance (наследование):

* super pom
* org parent pom
* project root pom
* module pom

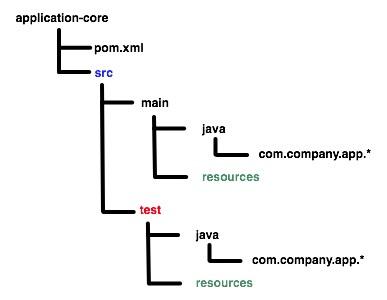
1. Plugins
2. Repositories (artefacts, Artefactory, Nexus, Maven Cental, hosting: jetty)
3. Dependencies
4. IDE Independence

**MAVEN Lifecycles:**

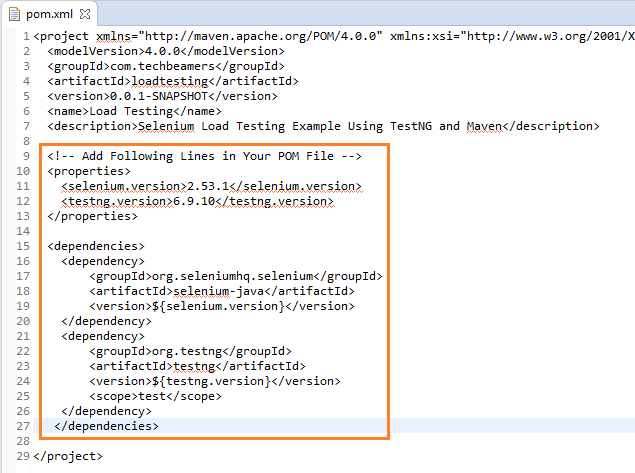




**MAVEN Project Structure:**



**MAVEN Versioning:**



**MAVEN Phases:**

#mvn clean

#mvn test

#mvn clean package

#mvn -help

#mvn -X -e clean package

#mvn dependency:tree

#mvn dependency:tree -Dverbose -Dincludes=commons-collections

<plugin>

<groupId>org.mortbay.jetty</groupId>

<artifactId>maven-jetty-plugin</artifactId>

<version>6.1.26</version>

<configuration>

<scanIntervalSeconds>10</scanIntervalSeconds>

<stopKey>stop</stopKey>

<stopPort>9999</stopPort>

</configuration>

<executions>

<execution>

<id>start-jetty</id>

<phase>pre-integration-test</phase>

<goals>

<goal>run</goal>

</goals>

<configuration>

<scanIntervalSeconds>0</scanIntervalSeconds>

<daemon>true</daemon>

</configuration>

</execution>

<execution>

<id>stop-jetty</id>

<phase>post-integration-test</phase>

<goals>

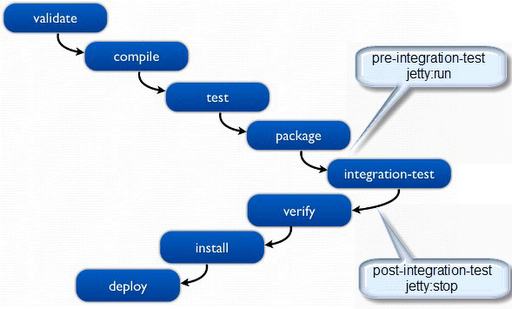
<goal>stop</goal>

</goals>

</execution>

</executions>

</plugin>



**MAVEN Profile:**



**MAVEN Plugin:**

<build>

<plugins>

<plugin> <!-- Build an executable JAR -->

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-jar-plugin</artifactId>

<version>3.0.2</version>

<configuration>

<archive>

<manifest>

<addClasspath>true</addClasspath>

<classpathPrefix>lib/</classpathPrefix>

<mainClass>com.mypackage.MyClass</mainClass>

</manifest>

</archive>

</configuration>

</plugin>

</plugins>

</build>

**GIT:**

#git clone …

mv -> git\_folder

cd git\_folder

#git status

#git add --all

#git status

#git commit -am "<commit message>"

#git push

Sping путать с Java Enterptise Edition. Ща близки. Spring как альтернативная основа JEE.

Java – программирование в xml

Наиболее распространненое и понятное

SpringCore

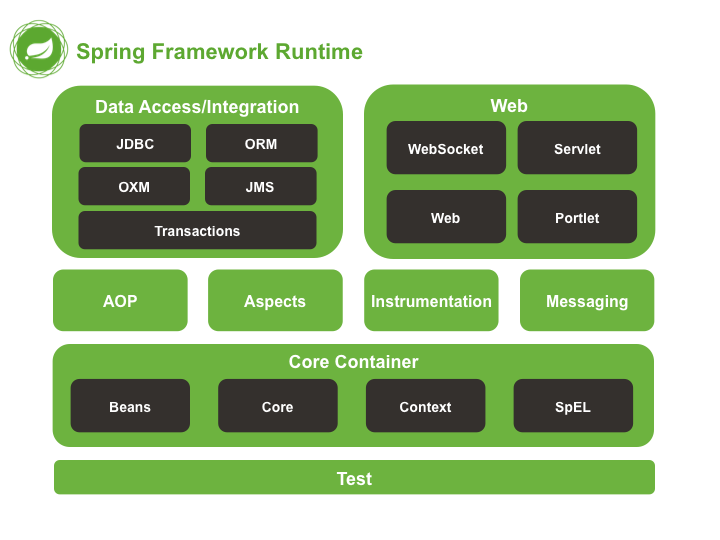
SpringJDBC

SpringData

MVC – низкоуровневая обертка для Servlet

SpringTesting

IOC это контейнер. Это основная часть в которой конифигурируются компоненты приложения. Core Container



более 10 лет

Java -> Annotations -> Annotations in Spring

когда один объект требует уже построенный объект

сильносвязанный код

необходимо для тестирования

концепция тестим вперед

тяжело тестировать монолитное приложение

победили микросервисы

IOC

модули верхнего уровня не должны зависеть от уровней нижнего уровня

они должны зависеть от абстракций,

абстракции должны зависеть от деталей

класс X зависит от класса Y когда:

- Y встроен в X

- X и есть Y так как он его extends

- принцип транзитивности X зависит от Z, а Z – зависит от Y

- если X зависит от Y, то это не значит, что Y зависит от X

получаются деревья зависимостей

сильносвязанный код

выделение общих частей кода -> вынесение в один общий модуль (кто кого и как использует)

процедурные языки

ООП

инкапсуляция – помогла разорвать внутренности класса с его публичной частью

появились объекты

методы, что используют друг друга предложили скрыть (private, protected)

возможность оторвать кусок кода и его отдельно тестировать (тестирование private через reflection)

однако черерез конструктор связывание идет клиентского кода со всеми классами, как через dectruct-ор

если еще россыпь setter-ов, билдер-ов

Interfaces + implementation

все зависит от interfaces, а не от имплементации

ClientCode -> Interfaces <- Implementation

Конструктор – new ArrayList

Coupled (MVC не было. UI - Бизнес логика – DataAccess все в одном месте) –

Decoupled (схема, интерфейсы)

JSP проектировались чтоб ходить в JDBC

зависимость от реализации одной строчкой

(пример)

Dependency Injection (код ничего не должен знать про фабрику) **инъекция зависимостей**

некто должен прочитать конфиги и сам собрал “товарищей” и внедрил их в код как-то незаметно

DI позволяет получать свои зависимости во время своего создания, во время подковерного вызова setter-ов от некой третьей стороны, которая, на самом деле коорденирует все объекты в системе

по сути это паттерн для ослабления зависимостей между объектами в системе

они ничего не знают друг о друге и могут тестироваться независимо, подменяться на другие реализации

это основная идея минимально связанных модулей

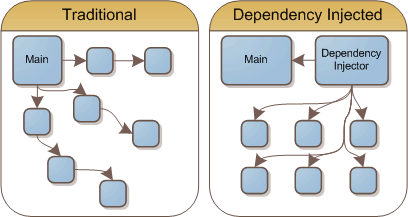
компоненты приложения начинают жить в неком контейнере (Spring, Jboss)

все компоненты (надо стремиться) создаются контейнером и им же освобождаются

чтоб GC это потом мог незамедлительно убрать

обычно говорят что не звоните вам мы вам сами позвоним

много людей собираются пробоваться на определенную роль

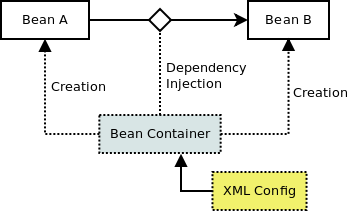


до: Human(). Конструктор. В нем new Head()

после: public setHead(Head head)

мы не будем вызывать setter и не будем вызывать конструктор

когда использовать: тестирование (инжекция mock, эмуляция вызовов)



1. создает объекты

2. конфигурирует

3. связывает объекты вместе

4. управляет их жизненным циклом (можно подписываться на всякие события связанные с жизненным циклом объекта)

xml, java, annotations, groovy конфиги -> классы совмещаются с метаданными конфигов -> инициализация, создание ApplicationContext (точка входа в IOC контейнер). Их может быть несколько. Можно выстроить некую иерархию.

в метаданных описываются правила как создавать объекты, как их инициализировать

ApplicationContext. Read-Only in Runtime. Его можно Перезагрузить. Метод refresh.

Можно объявить новый. Закрыть. Можно использовать фабричные методы для получения бинов.

Можно использовать различные файловые ресурсы. Конфиг может лежать в разном типе файлов.

Можно работь с жизненным циклом бинов.

Application Context,

Bean,

class

name/id/alias

scope

Singleton - Возвращает один и тот же экземпляр бина на каждый запрос контейнера Spring IoC (по умолчанию).

Prototype - Создает и возвращает новый экземпляр бина на каждый запрос.

Request - Создает и возвращает экземпляр бина на каждый HTTP запрос\*.

Session - Создает и возвращает экземпляр бина для каждой HTTP сессии\*.

Global-session - Создает и возвращает экземпляр бина для глобальной HTTP сессии\*.

constructor-arg

lazy-initialization mode

initialization/destruction model

Resource res = new FileSystemResource("beans.xml");

BeanFactory factory = new XmlBeanFactory(res); (урезанный контейнер, без событий)

IService service = (IService) factory.getBean(“service”);

ClassPathResource res = new ClassPathResource("beans.xml");

BeanFactory factory = new XmlBeanFactory(res);

ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(

new String[] {"applicationContext.xml", "applicationContext-part2.xml"});

BeanFactory factory = (BeanFactory) appContext;

<https://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/context/ApplicationContext.html>

FileSystemXmlApplicationContext: Загружает определения бинов из файла контекста который расположен в файловой системе.

ClassPathXmlApplicationContext: Ищет файл контекста в CLASSPATH.

WebXmlApplicationContext: Ищет файл контекста в веб приложении.

SpringBoot - !!!

Bean Definition (описание рецепта)

xml – мы не размазываем конфиги по всему приложению

2 способа инжектирования

- setter:

- constructor

public class Human {

public Human () {}

}

<bean id=“firstHuman” class=“app.Human”></bean>

public class Human {

public Human (String name, Integer age) {

...

}

}

<bean id=“firstHuman" class=“app.Human">

<constructor-arg type=“String” value=“Alex”/>

<constructor-arg type="int” value=“40”/>

</bean>

public class Human {

public Human (Hand hand, Head head) {

...

}

}

<bean id=“firstHuman" class=“app.Human">

<constructor-arg ref=“hisHand”/>

<constructor-arg ref=“hisHead”/>

</bean>

<bean id=“hisHand" class=“app.Hand”/>

<bean id=“hisHead" class=“app.Head”/>

public class Human {

public Human (Hand hand, int age) {

...

}

}

<bean id=“firstHuman" class=“app.Human">

<constructor-arg index=“0” ref=“hisHand”/>

<constructor-arg index=“1” value=“75”/>

</bean>

<bean id=“hisHand" class=“app.Hand”/>

public class Human {

public void setHead(Head head) ...

public void setName(String name) ...

}

<bean id=“human” class=“app.Human”>

<property name= “name” value=“Roman”/>

<property name= “head” ref=“hisHead”/>

</bean>

<bean id=“hisHead” class=“app.parts.Head”/>

Может не быть конструктора на 10 параметров, сеттеры есть всегда, имя преобразует в set + Имя и ищет

public class Human {

public void setHead(Head head) ...

public void setName(String name) ...

}

<bean id=“human” class=“app.Human”>

<property name= “name” value=“Roman”/>

<property name= “head”>

<bean class=“app.parts.Head”/>

</property>

</bean>

public class Human {

public void setHead(Head head) ...

public void setName(String name) ...

}

<bean id=“human” class=“app.Human”>

<property name= “name” value=“”/>

<property name= “head”>

<null/>

</property>

</bean>