Spring Boot Starter Parent

The *spring-boot-starter-parent* project is a special starter project that provides default configurations for our application and a complete dependency tree to quickly build our Spring Boot project. It also provides default configurations for Maven plugins, such as maven-failsafe-plugin, maven-jar-plugin, maven-surefire-plugin, and maven-war-plugin.

Beyond that, it also inherits dependency management from spring-boot-dependencies, which is the parent to the spring-boot-starter-parent.

We can start using it in our project by adding it as a parent in our project’s pom.xml:

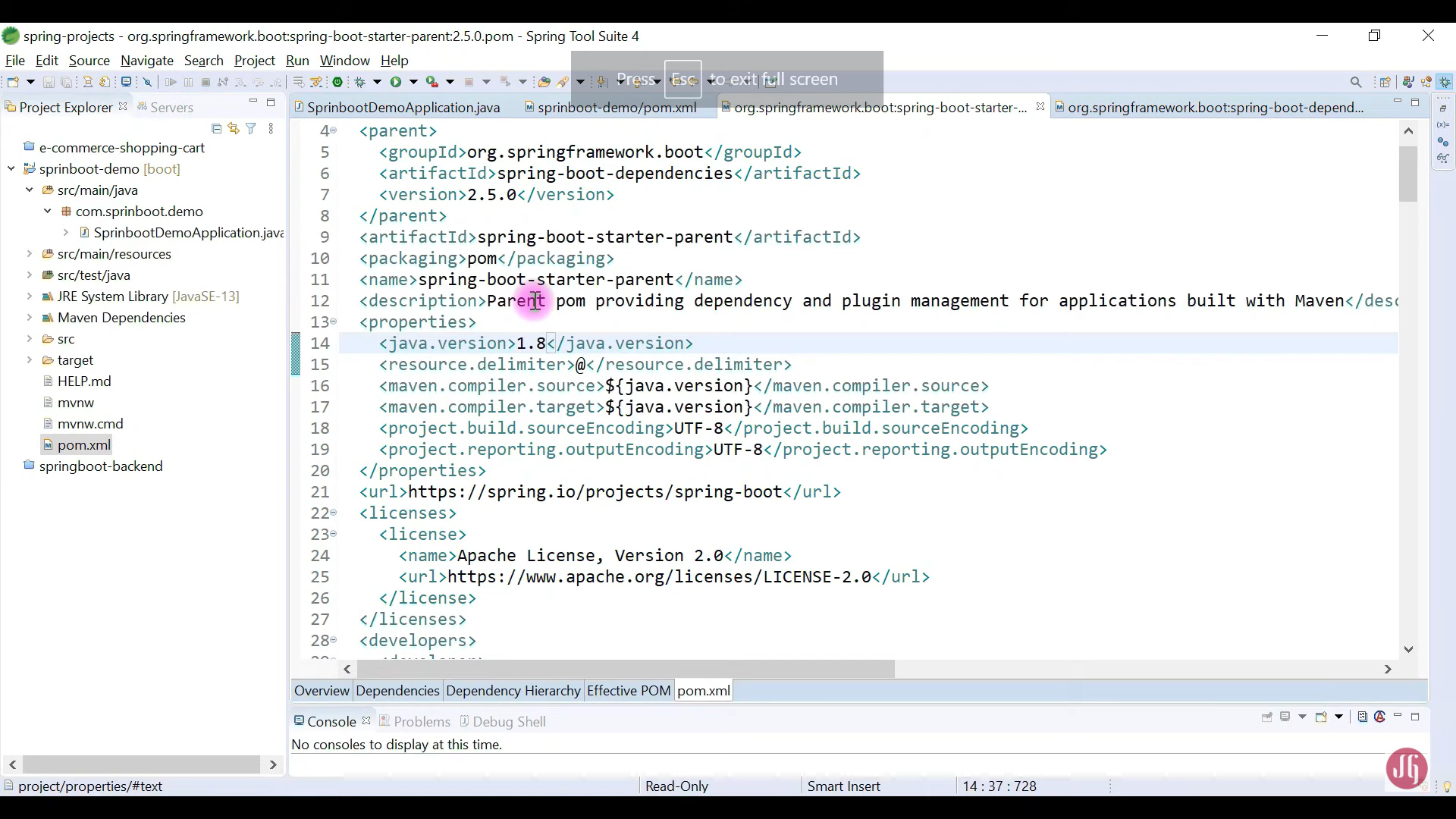
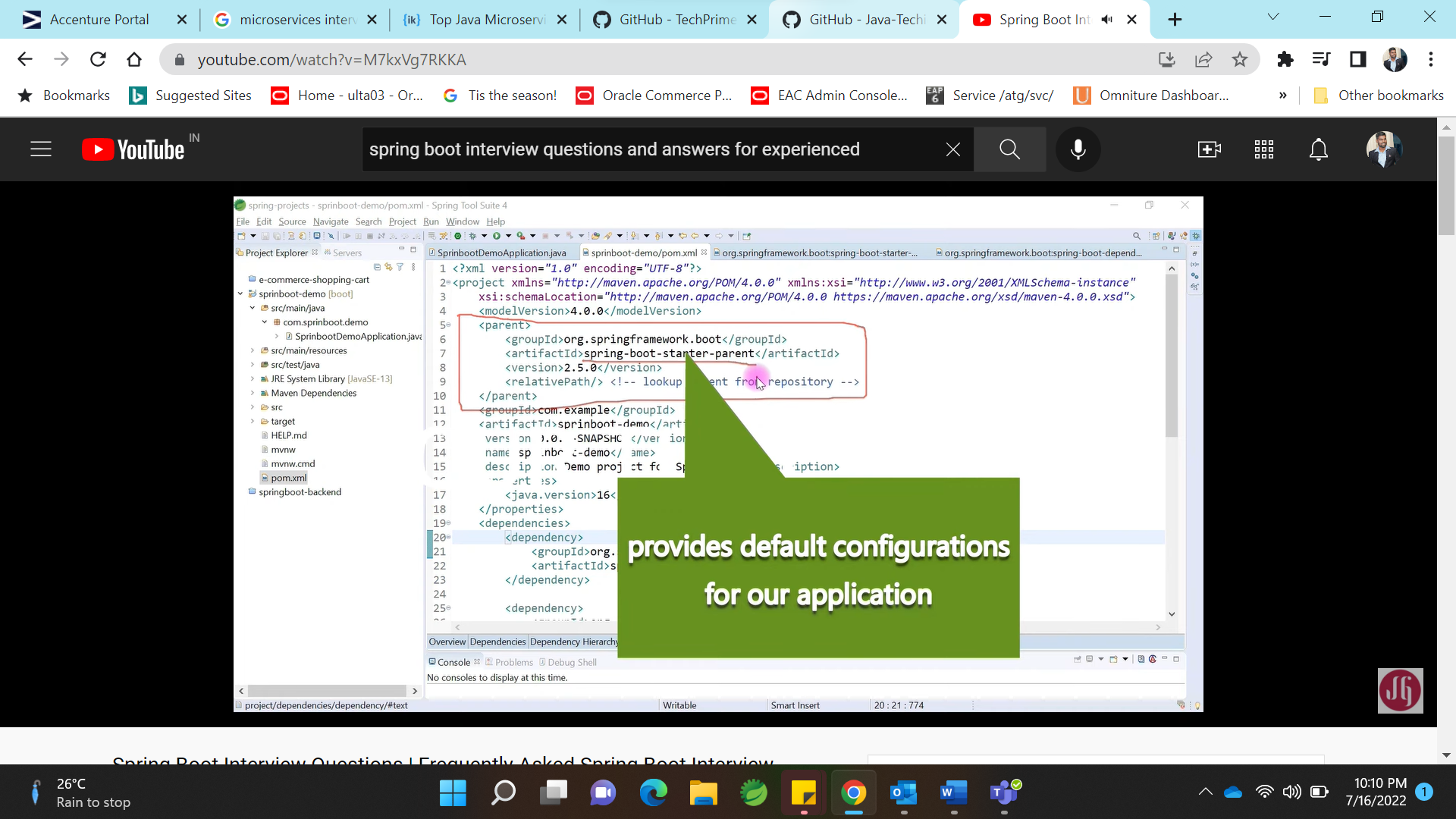
<**parent**>

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

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

<**version**>3.1.5</**version**>

</**parent**>



Parent dependency :

Maintain the version of dependency

Import maven plugins and default configs.



### Q1. What is Singleton Pattern and how do you implement it?

Singleton design pattern ensures that a class has only one instance and provides a global point of access to it.

This is one of the most asked question in interviews. Then interviewer asks to implement a [**Singleton Design Pattern Using Java**](https://codingnconcepts.com/java/singleton-design-pattern-using-java/).

Follow up questions in singleton pattern are as follows:

* How to make singleton class thread safe?  
  **using double checked locking**
* How to prevent deserialization to create new object of singleton class?  
  **using readResolve method to return same instance**
* How to prevent cloning to create a new object of singleton class?  
  **override clone method to return same instance**
* How to prevent reflexion to create a new object of singleton class?  
  **throw exception from private constructor if instance already exist**

###### Singleton Class

public class Singleton implements Serializable, Cloneable{

private static final long serialVersionUID = 1L;

private static Singleton instance = null;

private static Object DUMMY\_OBJECT = new Object();

private Singleton(){

/\*To prevent object creation using reflection\*/

if(instance!=null){

throw new InstantiationError( "Singleton Object is already created." );

}

}

public static Singleton getInstance(){

/\*Double checked locking\*/

if(instance == null){

synchronized (DUMMY\_OBJECT) {

if(instance == null){

instance = new Singleton();

}

}

}

return instance;

}

public static void print(){

System.out.println("I am a singleton class.");

}

/\*To prevent object creation using deserialization\*/

private Object readResolve() throws ObjectStreamException{

return instance;

}

/\*To prevent object creation using cloning\*/

@Override

protected Object clone() throws CloneNotSupportedException {

return instance;

}

}

### Q2. What is Immutable Object? How do you write an Immutable Class?

Immutable Class means that once an object is initialized from this Class, we cannot change the state of that object.

In other words, An immutable object can’t be modified after it has been created. When a new value is needed, the accepted practice is to make a copy of the object that has the new value.

In order to create an [**Immutable Class in Java**](https://codingnconcepts.com/java/immutable-class-using-java/), you should keep following points in mind:-

1. Declare the **class as final** so that it cannot be extended and subclasses will not be able to override methods.
2. Make all the **fields as private** so direct access in not allowed
3. Make all the **fields as final** so that value cannot be modified once initialized
4. Provide **no setter methods** — setter methods are those methods which modify fields or objects referred to by fields.
5. Initialize all the final fields through a **constructor** and perform a deep copy for mutable objects.
6. If the class holds a mutable object:
   * Don’t provide any methods that modify the mutable objects.
   * Always return a copy of mutable object from **getter method** and never return the actual object reference.

Let’s apply all the above points and create our immutable class ImmutablePerson

###### ImmutablePerson.java

/\*\*

\* Immutable class should mark as final so it can not be extended.

\* Fields should mark as private so direct access is not allowed.

\* Fields should mark as final so value can not be modified once initialized.

\*\*/

public final class ImmutablePerson {

// String - immutable

private final String name;

// Integer - immutable

private final Integer weight;

// Date - mutable

private final Date dateOfBirth;

/\*\*

\* All the final fields are initialized through constructor

\* Perform a deep copy of immutable objects

\*/

public ImmutablePerson(String name, Integer weight, Date dateOfBirth){

this.name = name;

this.weight = weight;

this.dateOfBirth = new Date(dateOfBirth.getTime());

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*PROVIDE NO SETTER METHODS \*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* String class is immutable so we can return the instance variable as it is

\*\*/

public String getName() {

return name;

}

/\*\*

\* Integer class is immutable so we can return the instance variable as it is

\*\*/

public Integer getWeight() {

return weight;

}

/\*\*

\* Date class is mutable so we need a little care here.

\* We should not return the reference of original instance variable.

\* Instead a new Date object, with content copied to it, should be returned.

\*\*/

public Date getDateOfBirth() {

return new Date(dateOfBirth.getTime());

}

@Override

public String toString() {

return "Person { name: " + name + ", weight: " + weight + ", dateOfBirth: " + new SimpleDateFormat("dd-MM-yyyy").format(dateOfBirth) + "}";

}

}

### Q1. S.O.L.I.D. (5 class design principle) in Java?

These 5 design principle are know as SOLID principles:-

* **Single Responsibility Principle** - One class should have one and only one responsibility. We should not write thousands lines of code in a single class (a.k.a. God Class), instead we should refactor and layer our classes such as Controllers (API), Services (Business Logic), Repositories (Database Connection) for application development.
* **Open Close Principle** - Software entities (Classes and methods) should be open for extension, but closed for modification. We should not change the code of existing class or method to add new functionality, instead we should inherit the Class or overload/override the Method.
* **Liskov Substitution** - Child class should be able to substitute Parent class during runtime polymorphism.
* **Interface Segregation** - Clients should not be forced to implement unnecessary methods which they will not use
* **Dependency Inversion** - Depend on abstractions (interfaces and abstract classes), instead of concrete implementations (classes).

## What is volatile keyword?

Each thread runs in one core of multi-core processor. Each Core has their own local cache. All the cores share one shared cache.

thread-1 thread-2

┌―――――――――――――┐―――――――――――――┐

| Core 1 | Core 2 |

|―――――――――――――|―――――――――――――|

| Local Cache | Local Cache |

|―――――――――――――┘―――――――――――――|

| Shared Cache |

└―――――――――――――――――――――――――――┘

When you apply volatile keyword to a property. Any updates in that property done by thread-1 in local-cache is pushed down to shared-cache to make sure that the update is visible to thread-2

// atomic operation

volatile boolean flag = true;

volatile keyword doesn’t work when you do compound operations such as count++ which is read, increment and write back. In such case we can use AtomicInteger or AtomicLong

| **Type** | **Use Case** |
| --- | --- |
| volatile | Flags |
| AtomicInteger, AtomicLong | Counters |

**Java Code Review Checklist**

* Make sure it handles constants efficiently. ...
* Check for proper clean Up. ...
* Handle strings appropriately. ...
* Optimize to use switch-case over multiple If-Else statements. ...
* Ensure the code follows appropriate error handling procedures. ...