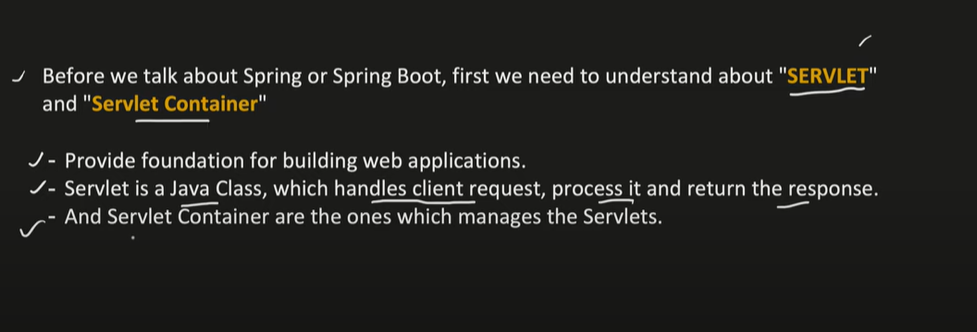
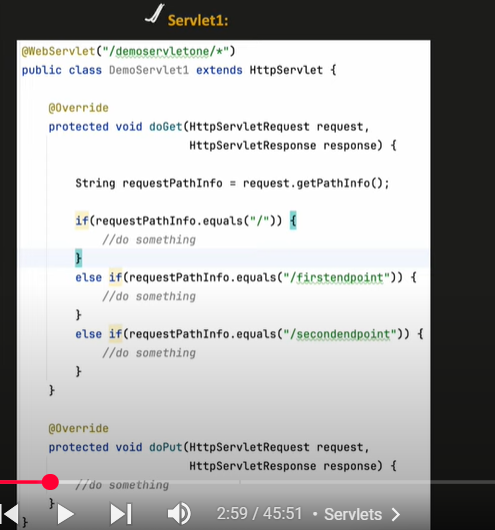
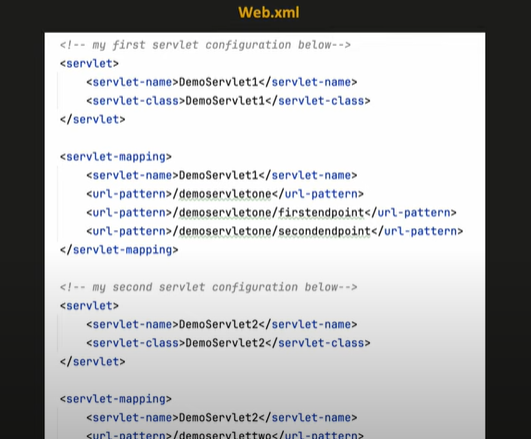
**Why spring boot?**

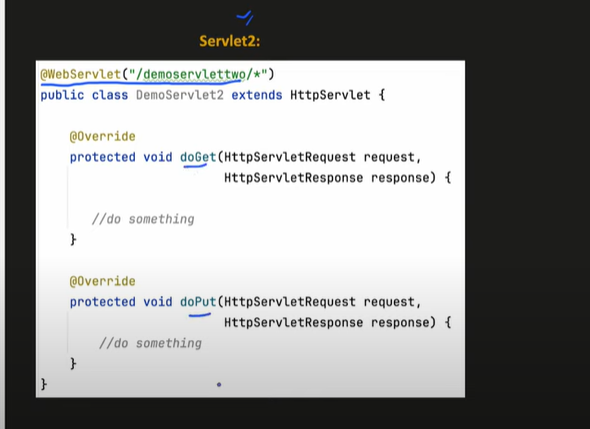
In 2015 and 2016, servlets were important and recognized.

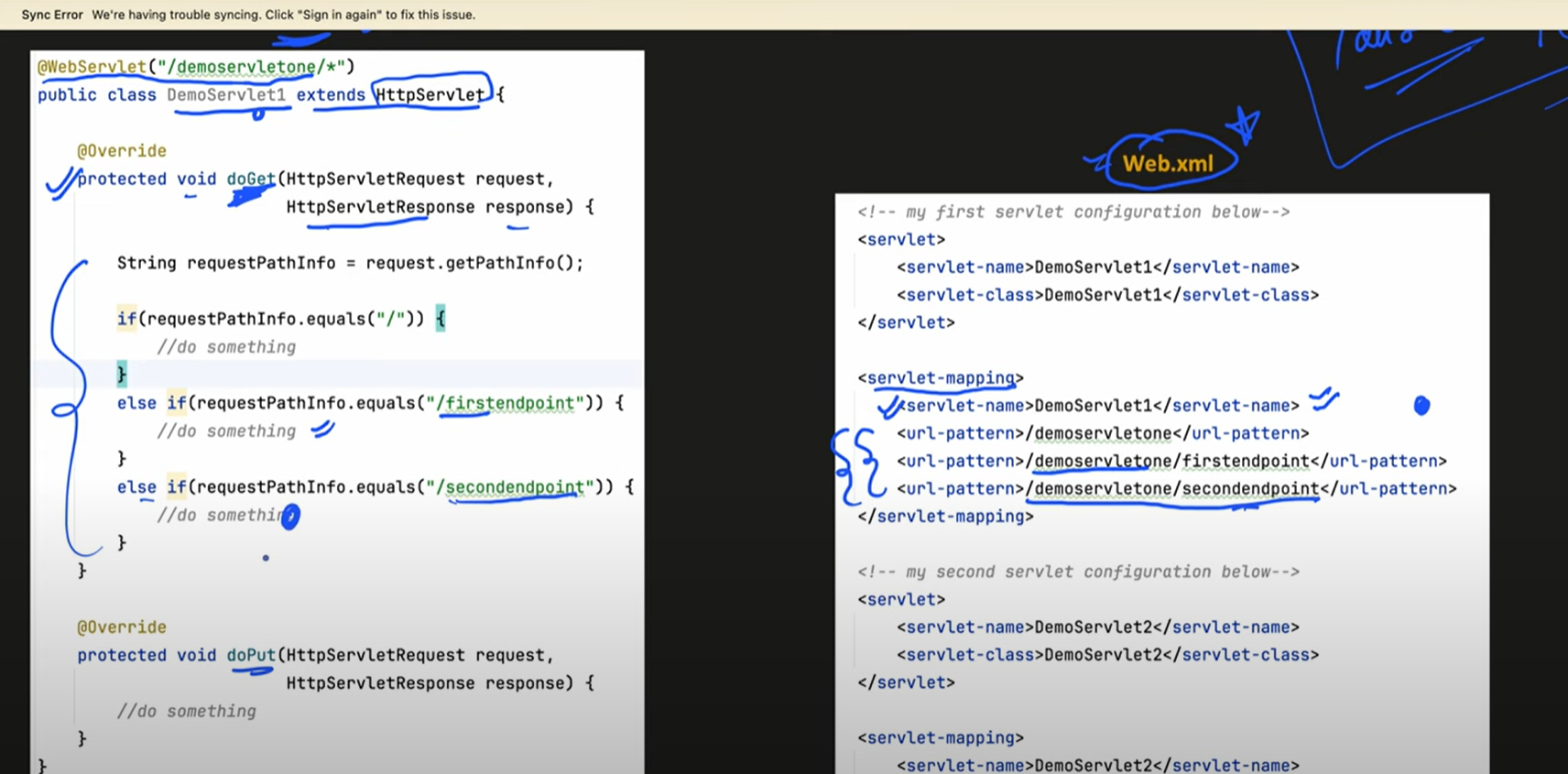
**Problems in servlets:**



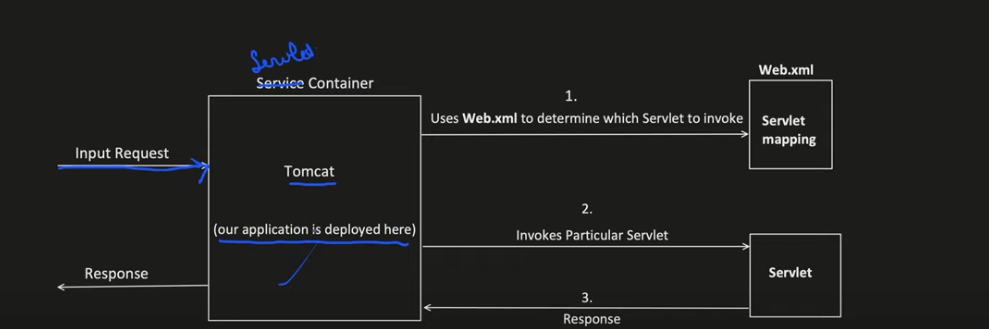


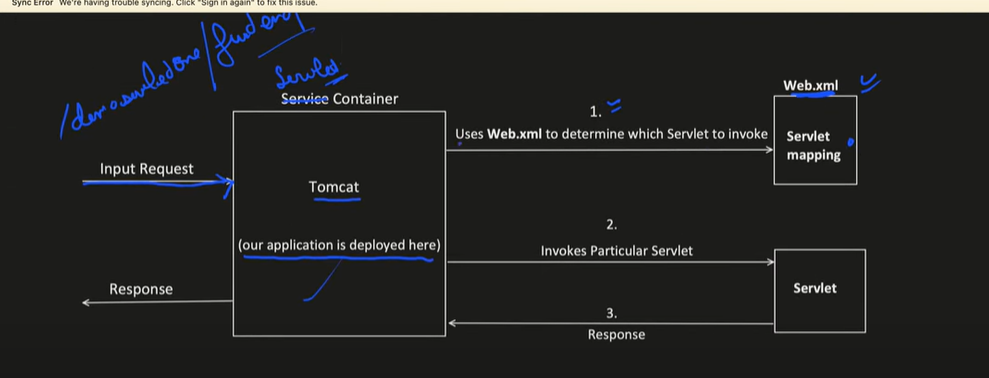






Servelts converted into war and deployed in tom cat server





**Spring Framework** came then that is family and has a lot of features including spring boot

**Before Spring boot -> Spring MVC**

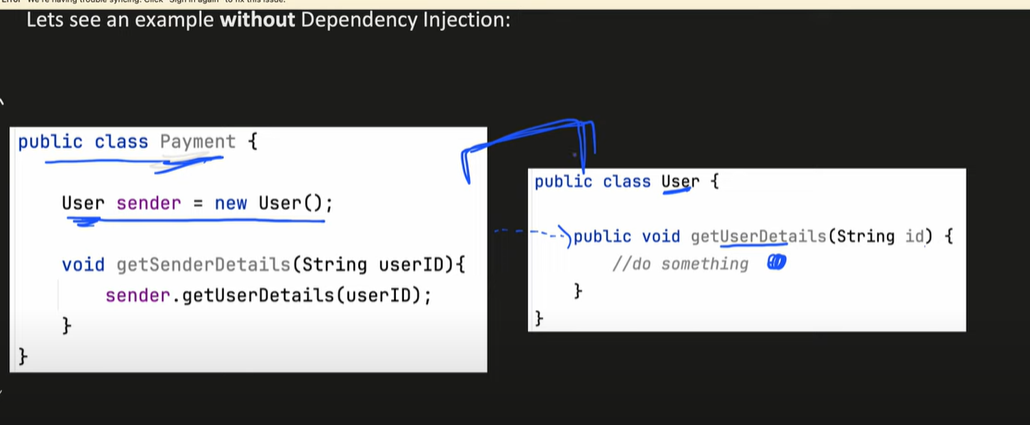


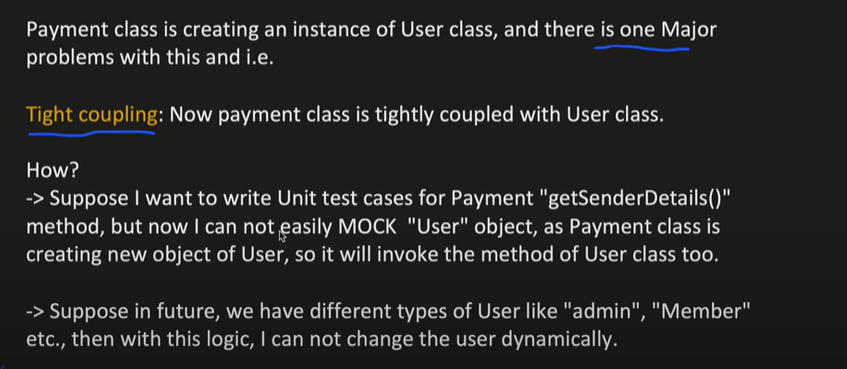
Web.xml is working as filter, every request to it.

So, now annotations came.

It introduced Inversion of Control to manage object dependencies and life cycle of objects (through dependency injection)

Dependency injection is implementation of IOC





Great question! The key difference between the two cases lies in **how dependencies are managed** and **how much control you have over them**. Let me break it down and explain why the second approach (using dependency injection) works better with Mockito, while the first approach (tight coupling) does not.

**Case 1: Tight Coupling (Direct Object Creation)**

In this approach, the Payment class directly creates and depends on the User object:

java

Copy

class Payment {

private User user = new User(); // Tight coupling

void getUserDetails(String id) {

user.getUserDetails(id);

}

}

**Why Mockito Doesn't Work Here:**

1. **No Control Over Dependency**:
   * The User object is created inside the Payment class.
   * You cannot replace User with a mock object because it's hardcoded.
2. **Mockito Cannot Intervene**:
   * Mockito works by injecting mock objects into the class under test.
   * Since User is instantiated directly, Mockito has no way to replace it with a mock.
3. **Testing Becomes Difficult**:
   * To test Payment, you are forced to use the real User object, which might have its own dependencies (e.g., database, API).
   * This makes it hard to isolate and test Payment in isolation.

**Case 2: Dependency Injection (Constructor Injection)**

In this approach, the Payment class depends on an abstraction (UserService) and receives its dependency via the constructor:

java

Copy

class Payment {

private UserService userService;

public Payment(UserService userService) { // Dependency injection

this.userService = userService;

}

void getUserDetails(String id) {

userService.getUserDetails(id);

}

}

**Why Mockito Works Here:**

1. **Control Over Dependency**:
   * The Payment class does not create the UserService object itself. Instead, it receives it from the outside (via the constructor).
   * This allows you to inject a mock UserService during testing.
2. **Mockito Can Intervene**:
   * You can use Mockito to create a mock UserService and inject it into Payment.
   * For example:

java

Copy

UserService mockUserService = mock(UserService.class);

Payment payment = new Payment(mockUserService); // Inject mock

1. **Isolation for Testing**:
   * Since Payment depends on an abstraction (UserService), you can easily replace the real implementation with a mock.
   * This allows you to test Payment in isolation, without worrying about the behavior of UserService.

**Key Differences**

| **Aspect** | **Tight Coupling (Case 1)** | **Dependency Injection (Case 2)** |
| --- | --- | --- |
| **Dependency Creation** | User is created inside Payment. | UserService is provided from outside. |
| **Control** | No control over User object. | Full control over UserService object. |
| **Mocking** | Cannot mock User. | Can mock UserService easily. |
| **Testing** | Hard to test Payment in isolation. | Easy to test Payment in isolation. |
| **Flexibility** | Rigid design; changes to User affect Payment. | Flexible design; changes to UserService don't affect Payment. |

**Example: Mockito in Action (Case 2)**

Here’s how you can use Mockito to test the Payment class in the dependency injection approach:

java

Copy

import static org.mockito.Mockito.\*;

import static org.junit.jupiter.api.Assertions.\*;

class PaymentTest {

@Test

void testGetUserDetails() {

// Create a mock UserService

UserService mockUserService = mock(UserService.class);

// Define the behavior of the mock

User mockUser = new User("123", "John Doe", true);

when(mockUserService.getUserDetails("123")).thenReturn(mockUser);

// Inject the mock into Payment

Payment payment = new Payment(mockUserService);

// Test the method

User result = payment.getUserDetails("123");

// Verify the result

assertEquals("John Doe", result.getName());

}

}

**What Happens Here:**

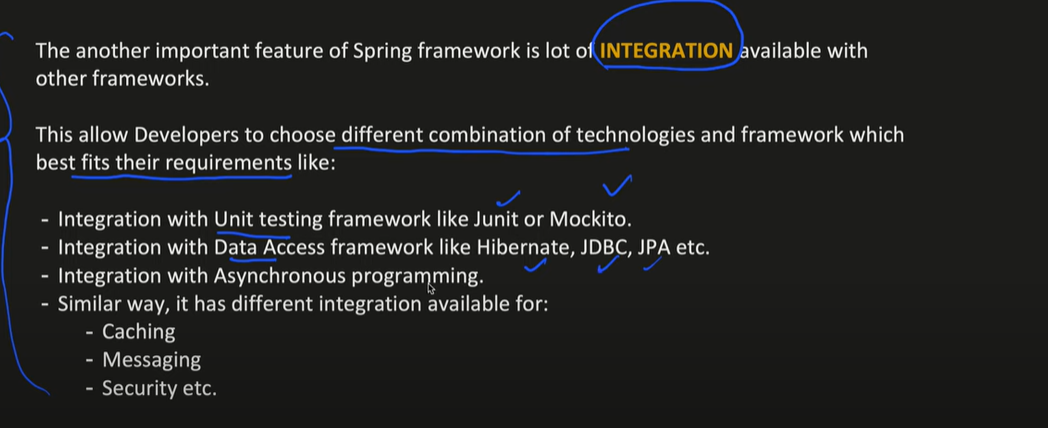
1. **Mock Creation**:
   * UserService mockUserService = mock(UserService.class); creates a mock object for UserService.
2. **Behavior Definition**:
   * when(mockUserService.getUserDetails("123")).thenReturn(mockUser); defines what the mock should return when getUserDetails is called with "123".
3. **Dependency Injection**:
   * Payment payment = new Payment(mockUserService); injects the mock into Payment.
4. **Test Execution**:
   * payment.getUserDetails("123"); calls the method under test.
   * The mock UserService returns the predefined mockUser.
5. **Verification**:
   * assertEquals("John Doe", result.getName()); checks that the result is as expected.

**Why This Works**

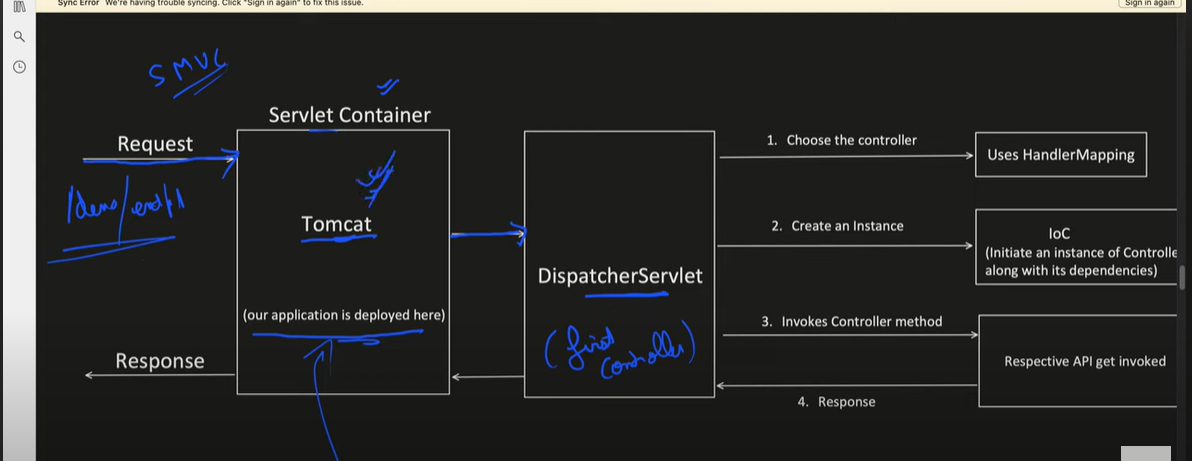
* **Decoupling**: Payment depends on an abstraction (UserService), not a concrete implementation.
* **Testability**: You can replace UserService with a mock during testing.
* **Flexibility**: You can change the implementation of UserService without affecting Payment.

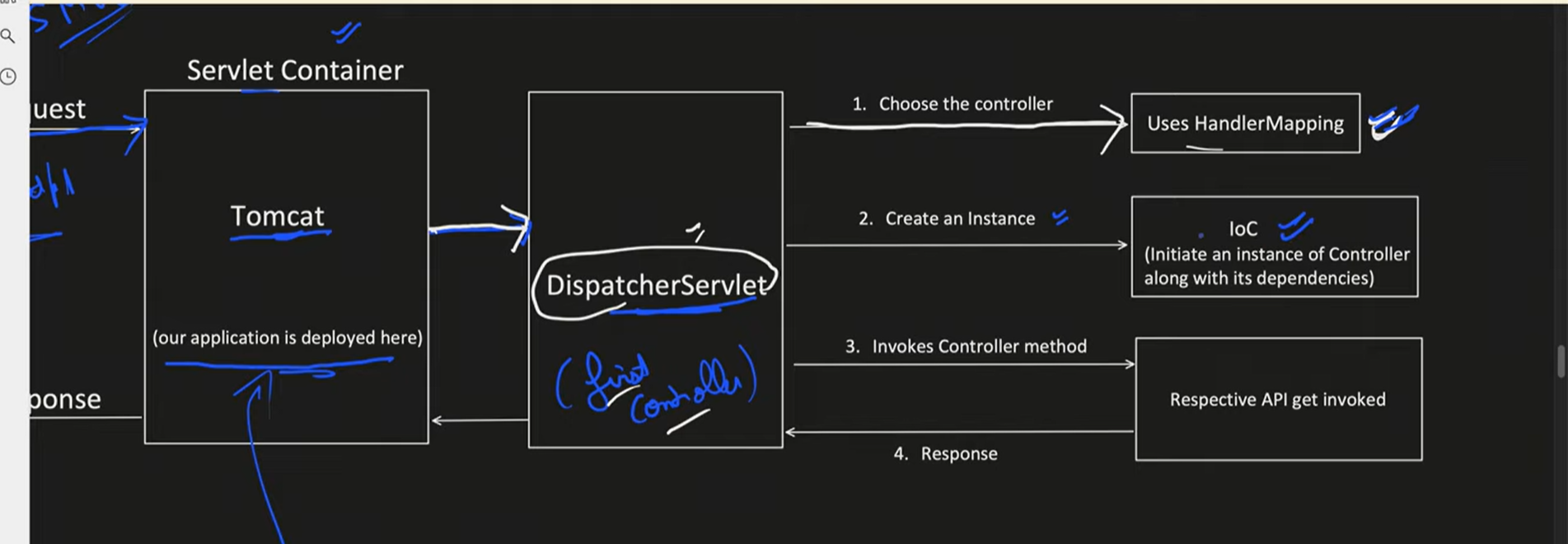
**Summary**

* **Tight Coupling**: Makes testing difficult because you cannot replace dependencies with mocks.
* **Dependency Injection**: Enables mocking by allowing you to inject dependencies from outside.
* **Mockito**: Works well with dependency injection because it can replace real dependencies with mocks.

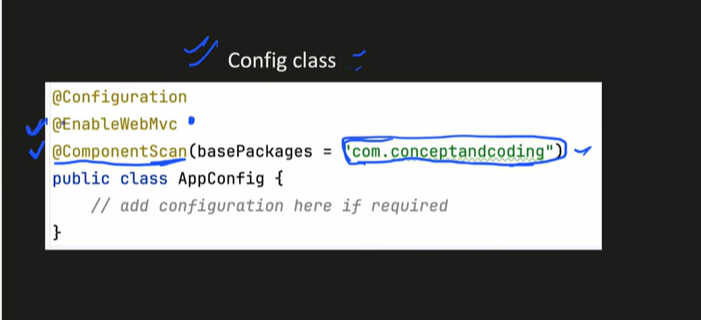


We have to deploy our application in servlets and Spring MVC



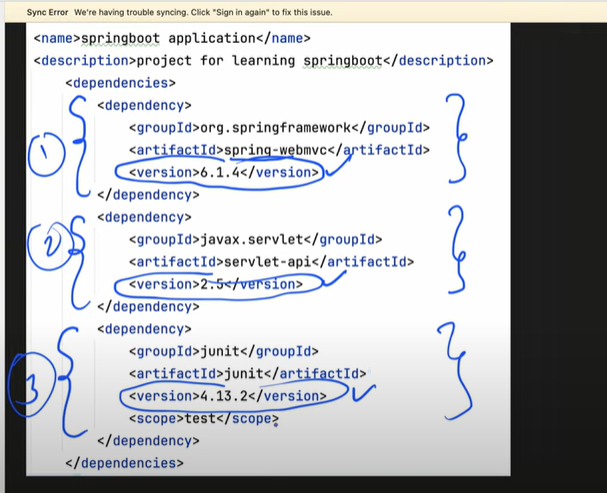


Using Handler mapping, dispatcher servlet will decide which controller and method to call (get, post, put)





**Advantages of spring boot:**

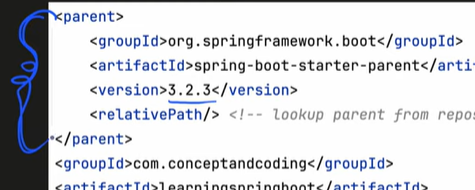


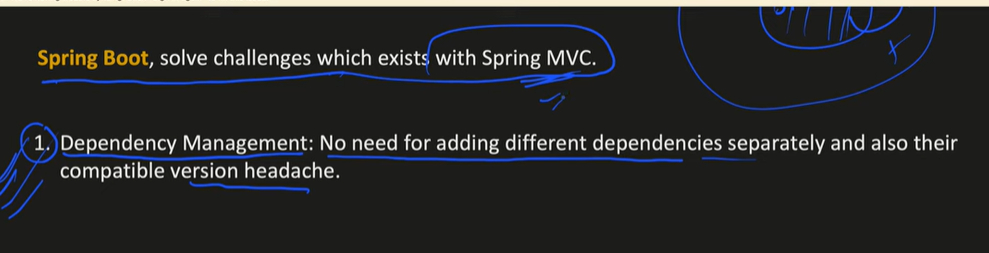
Dependency management we write dependencies in pom.xml

Auto loads dependency required for web and test using starter-web and starter-test, in spring MVC we need to write about version and dependencies.

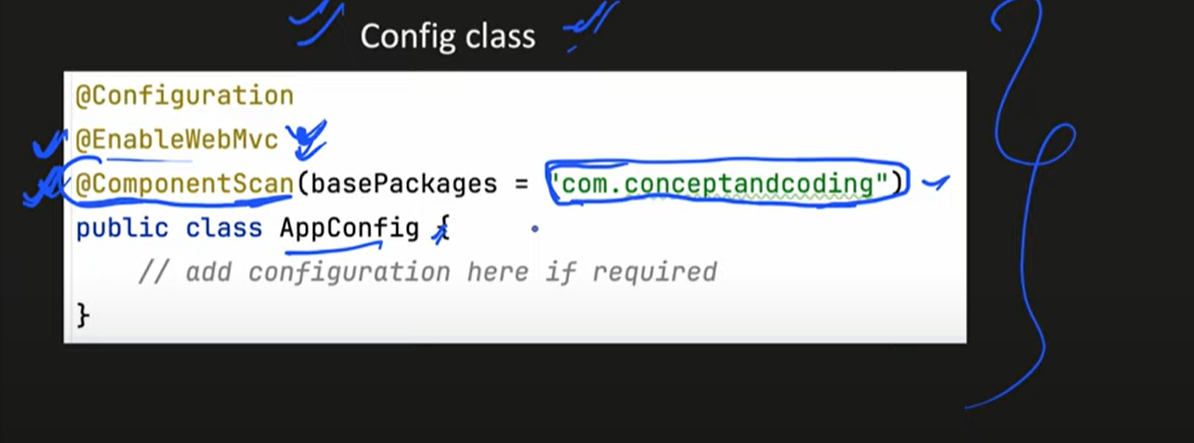
Auto-Compatible

Only need to define spring version in project



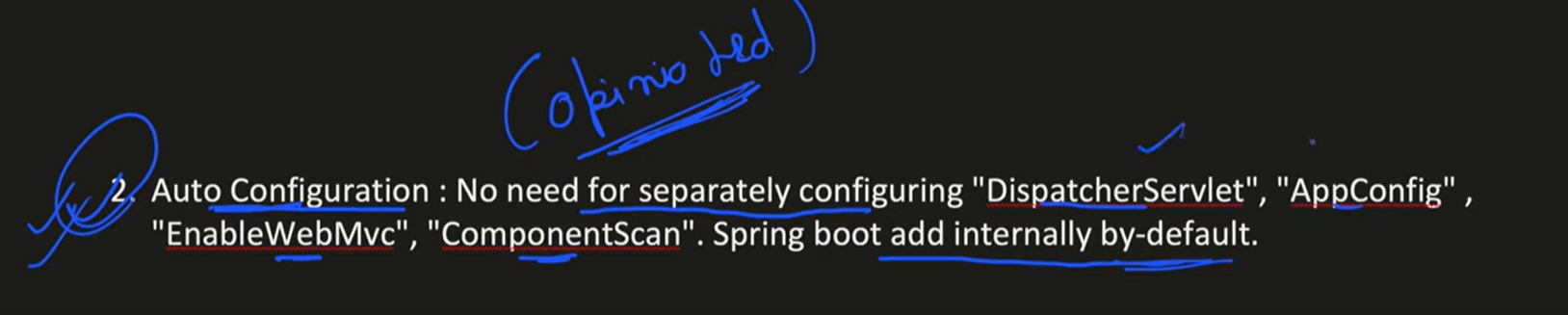


Automatically add configuration and component scan

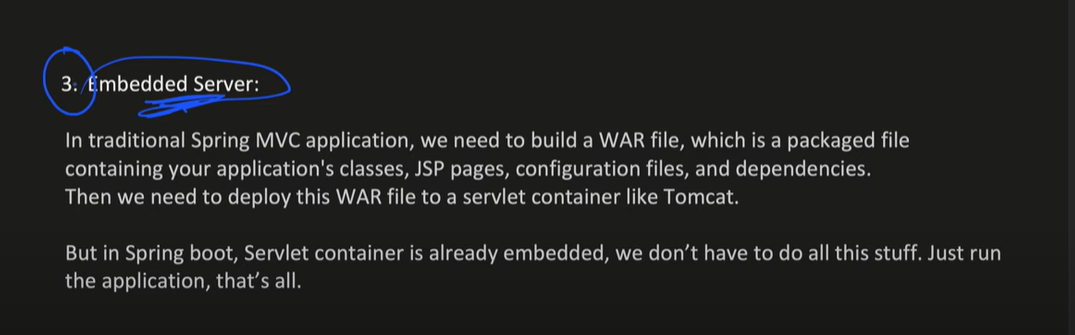


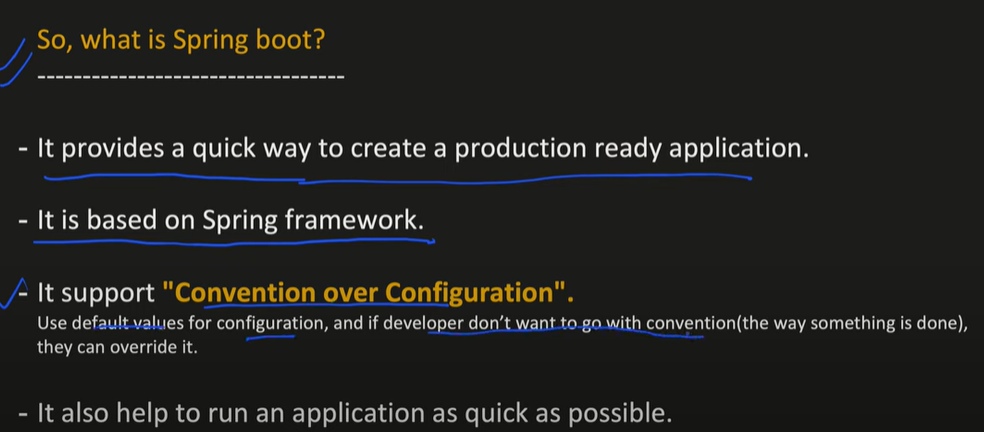
By default component scan on package where our spring boot application is placed.

@AutoConfiguration



So it is Opinated, if you agree with it.





Stand-Alone java web applications -> Java JAR

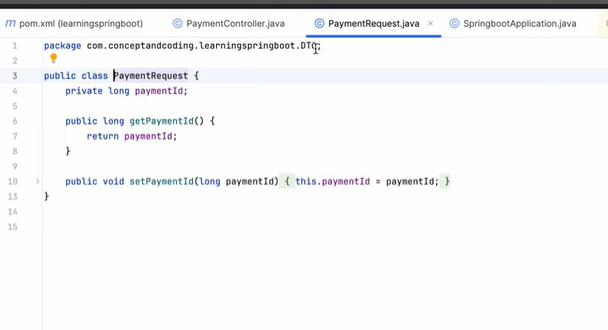
You send request to server and get response that’s it JAR

In world of microservices, each application is standalone application

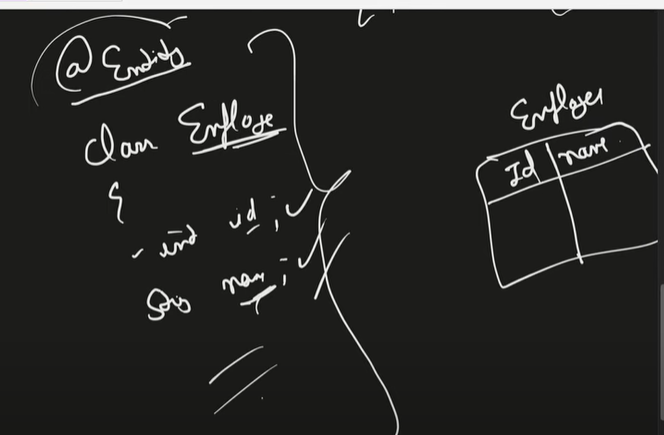
WAR -> HTML, CSS, JSP pages, everything bundled out in one format



* We need data transfer objects because it will help us to reduce our code such like if I’ve a login request dto and it has useremail and in future I change this to simple email so I only change this and not other things so it keeps things simple and keep only those needed.



* Utility Helper methods used in multiple class, can say common method across application or certain classes
* Entity is actually direct representation of a table in database, so simple called as POJOs



Our framework like JPA, Hibernate and JDBC do mapping for us

Exact same name for fields and datatypes