**SPRING BOOT FIRST PROJECT**

* **Spring boot helps building quality java code easy and fast.**







Spring boot takes care of a lot of complexity comes on way of creating Enterprise java projects. It helps in getting everything wired up and running in few minutes.



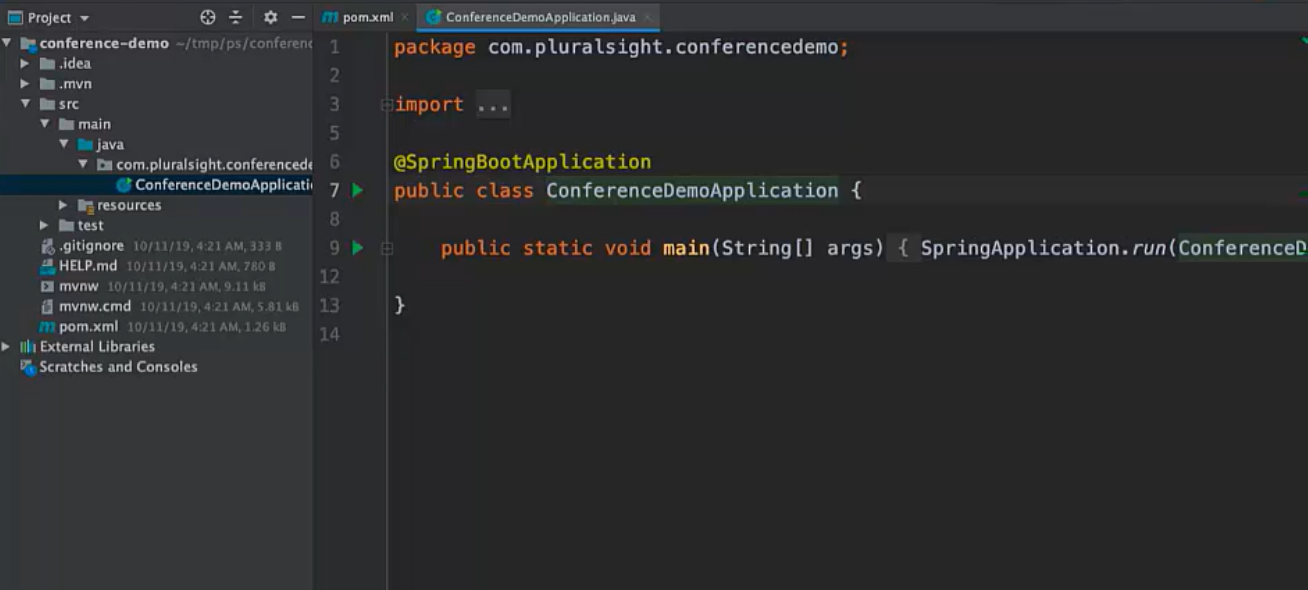
**WHY SPRING BOOT**

**Before spring boot it would have taken days for developers to wire up spring MVC and database setup before actual business logic is written.**

**With spring boot these configurations are wired up and running within minutes.**



**Once we import the spring project into intellij.**

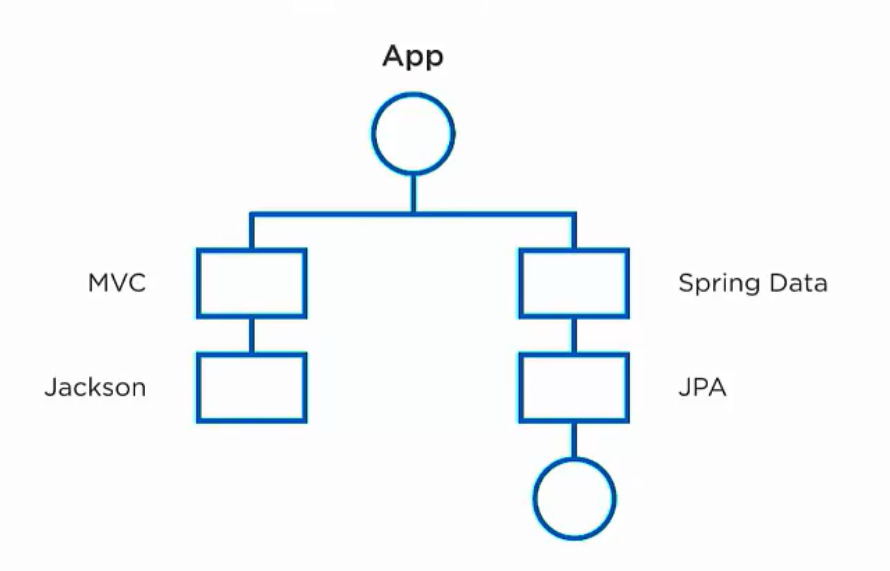


**We have a starter java class annotated with SpringBootApplication.**

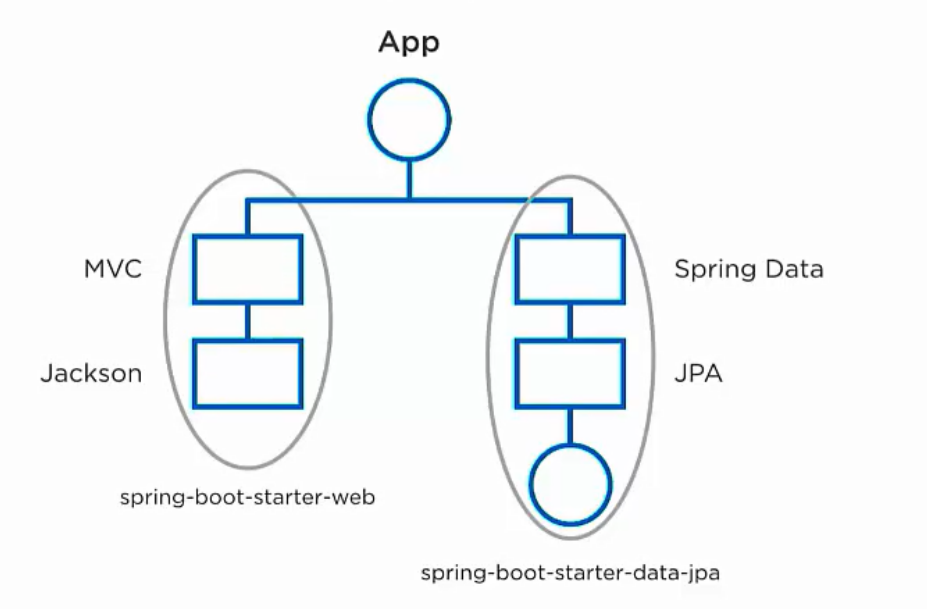
**Packages Creation**

* First create **controller, model, service,repository** packages.





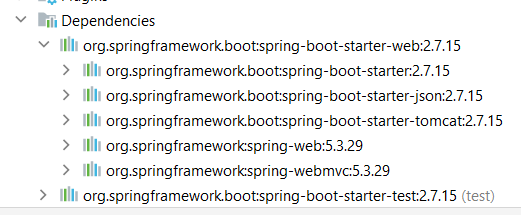
Starters are way to integrate dependency in your project.



To avoid version mismatch spring uses starter dependency.

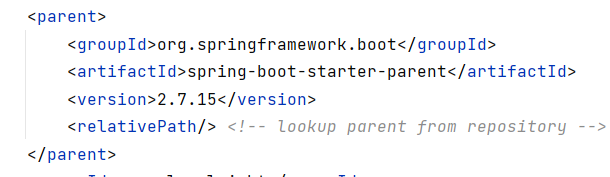
**2 MAJOR dependency:**

1. **Starter-web –** which contains other transitive dependencies.



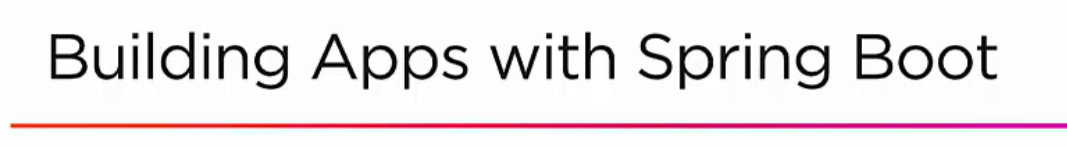
1. **Starter-test -- this contains other mokito and test dependencies.**

* Spring boot starter doesn’t have any version of dependency. All the dependency is marked by the parent declaration of version.

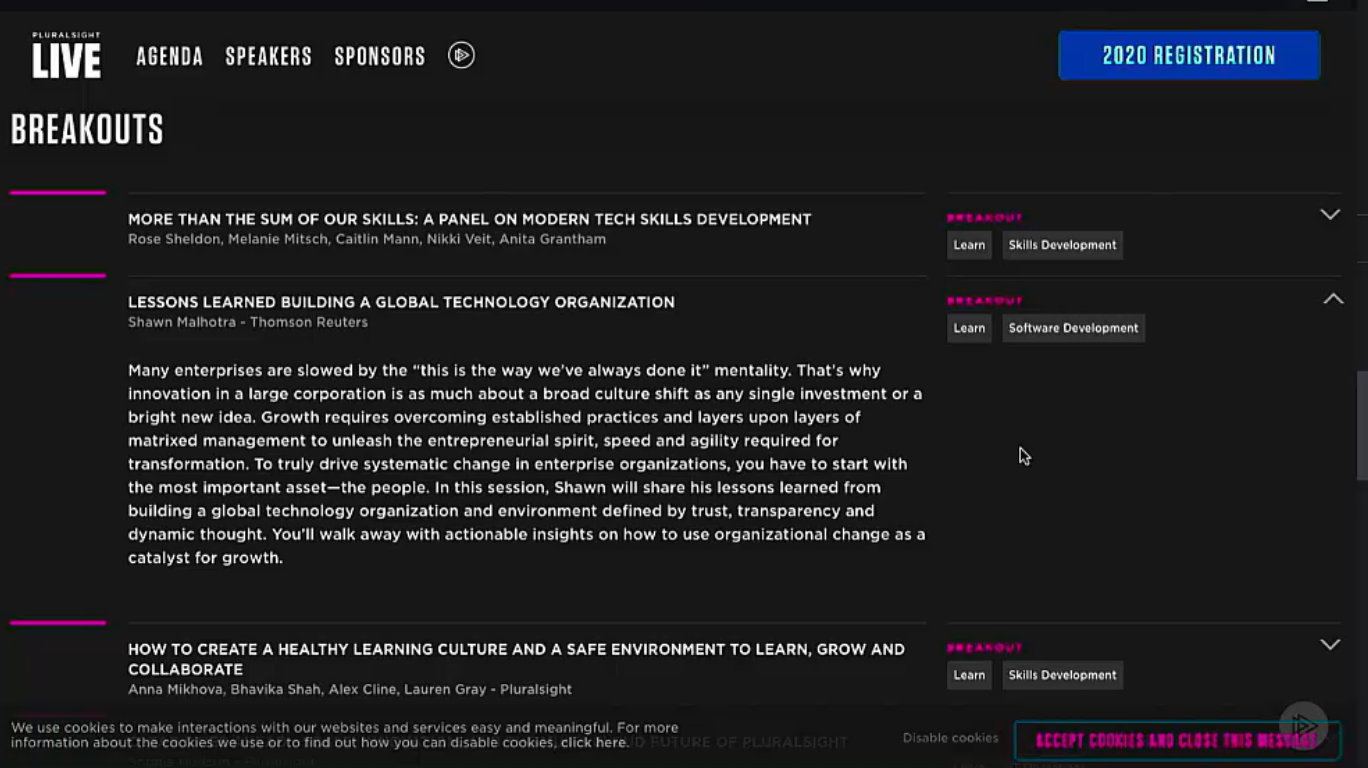


**SPRING BILL OF MATERIAL**. Available in the effective pom declaration





We are building a conference app which has speakers topics and description about the session etc.



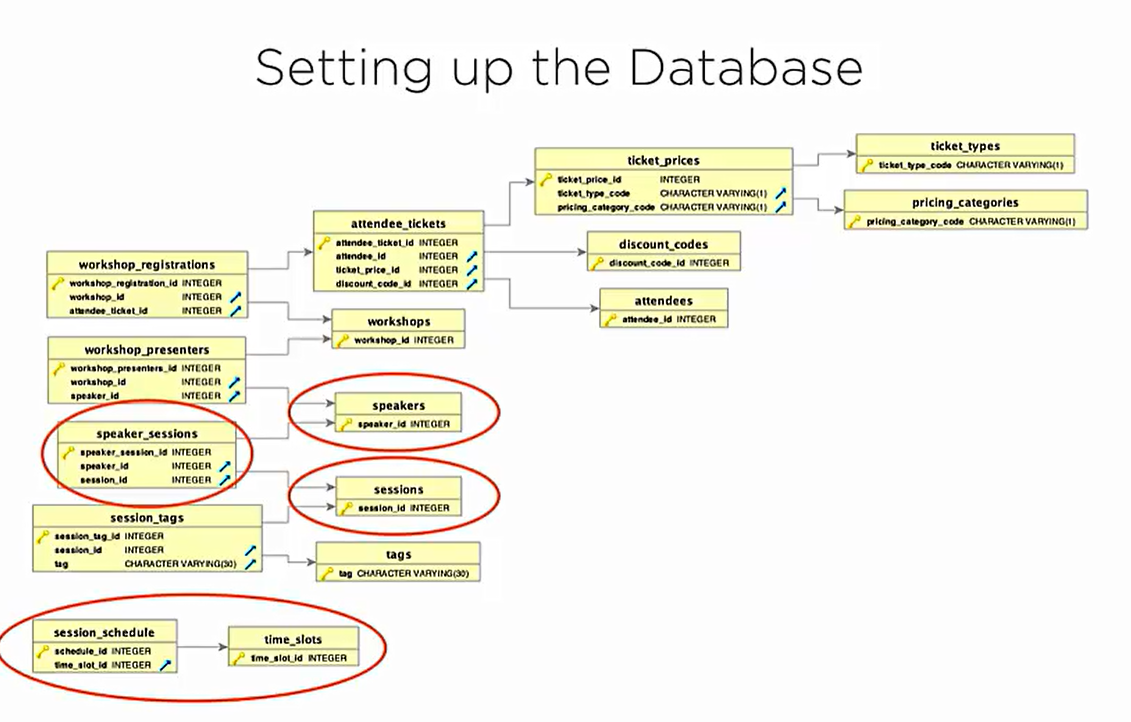
**STEP 1 DATABASE SETUP**

1. Spring JPA setup
2. Add dependency for JPA and postgres. – **JPA** its is an API which act as a bridge between java application and relational databases. Provide ORM tool facility.

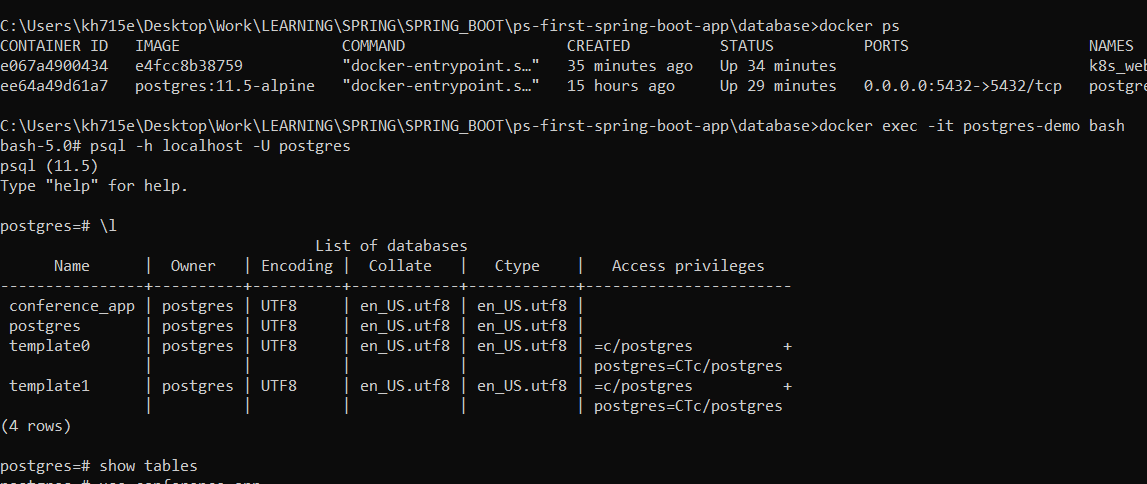


1. Put the connection user name etc details in resource app properties file.

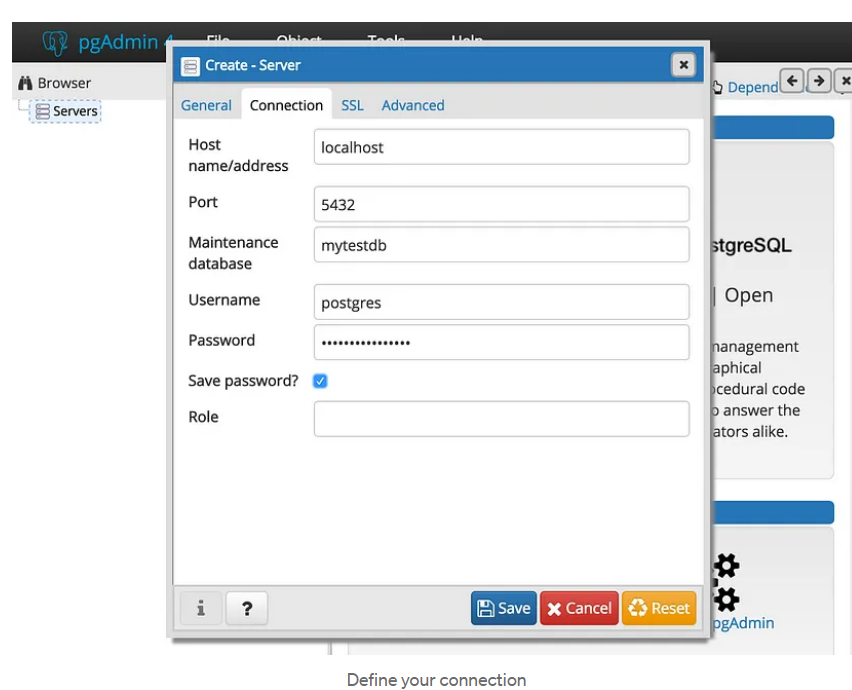
**BASIC SCHEMA** for speakers and sessions



Connect to postgres using docker



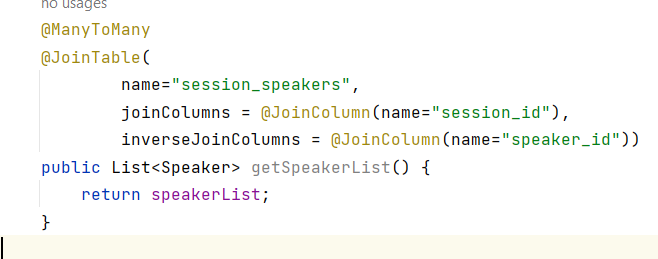
Connect PGADMIN to database



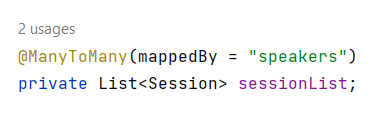
Now we created 2 models

1. Speaker 2) Session

**These models have many-many relationship**

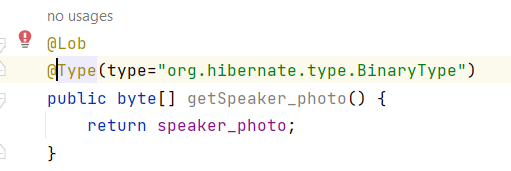
* Add a List<Speakers> in the session mod
* On Session class
* 

On Speakers class



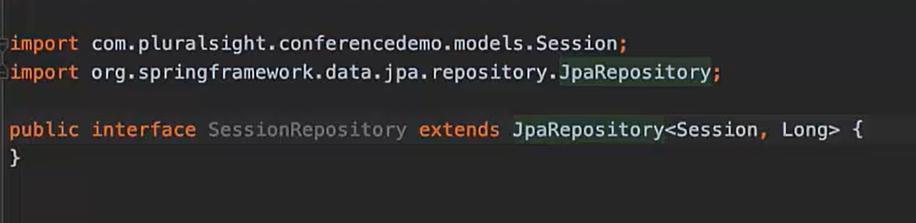
**ADDING PHOTOS TO SPEAKERS**

Binary lob data type



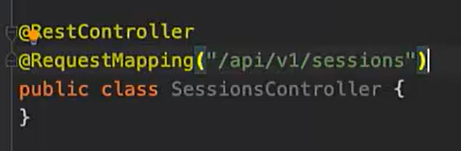
**CREATING JPA REPOSITORY**

* **Data access layer are JPA repository which is mostly interfaces.**



* **Just by creating repository like this we have all CRUD operation handled.**

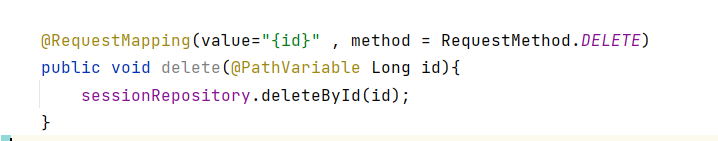
**CREATING THE API REST CONTROLLERS**



**@RestController – It tells this controller will respond to payload incoming and outgoing as JSON REST endpoints.**

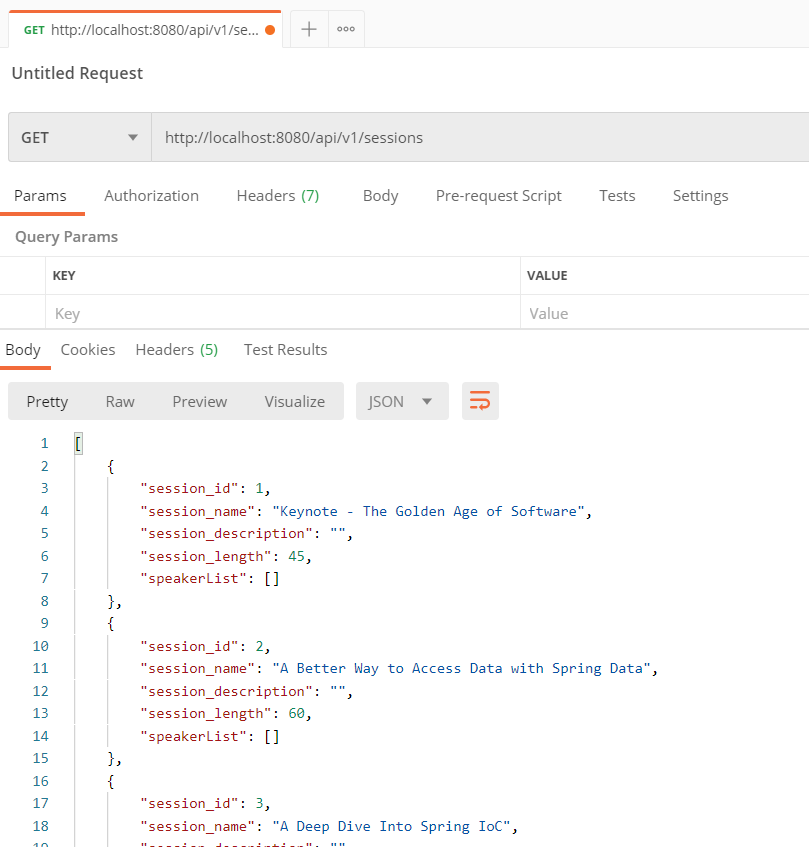
**@RequestMapping – It tells the router what the mapping will look like.**

**We use delete verb like this as rest done’t have @DeleteMapping annotation**

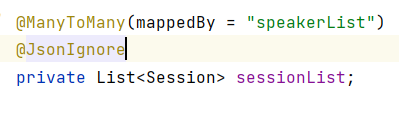


**ONCE DONE START THE CONFERENCEDEMOAPPLICATION MAIN CLASS**

1. **Hitting get API to get all sessions**



We can get **serialization problem** as one session has multiple speakers and again 1 speaker has multiple session it goes on and on.



Add @JsonIgnore add speakers to avoid this issue.

1. GETbyId is creating some serialization issue to fix it use annotation below on entity.
2. 

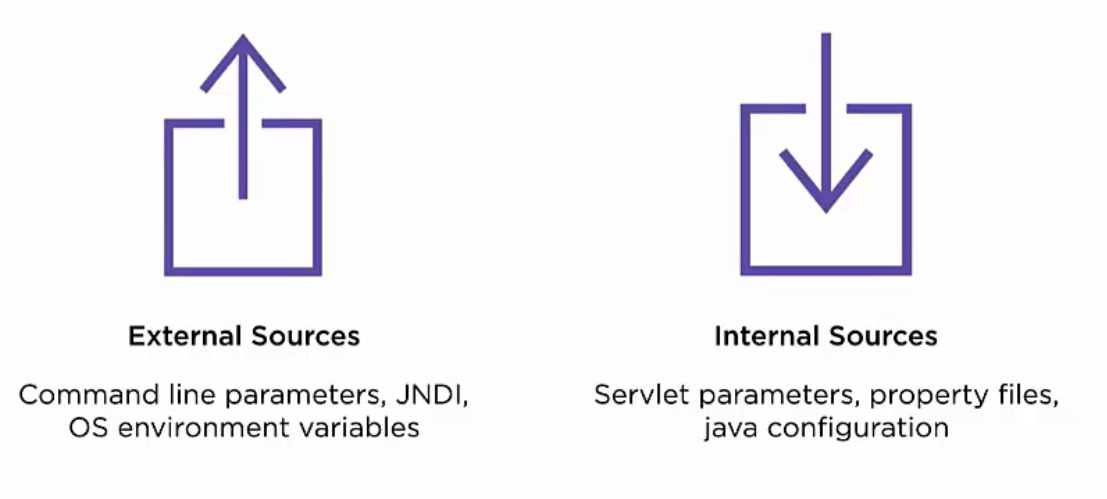


**SPRING BOOT CONFIGURATION AND ENV NEEDS**

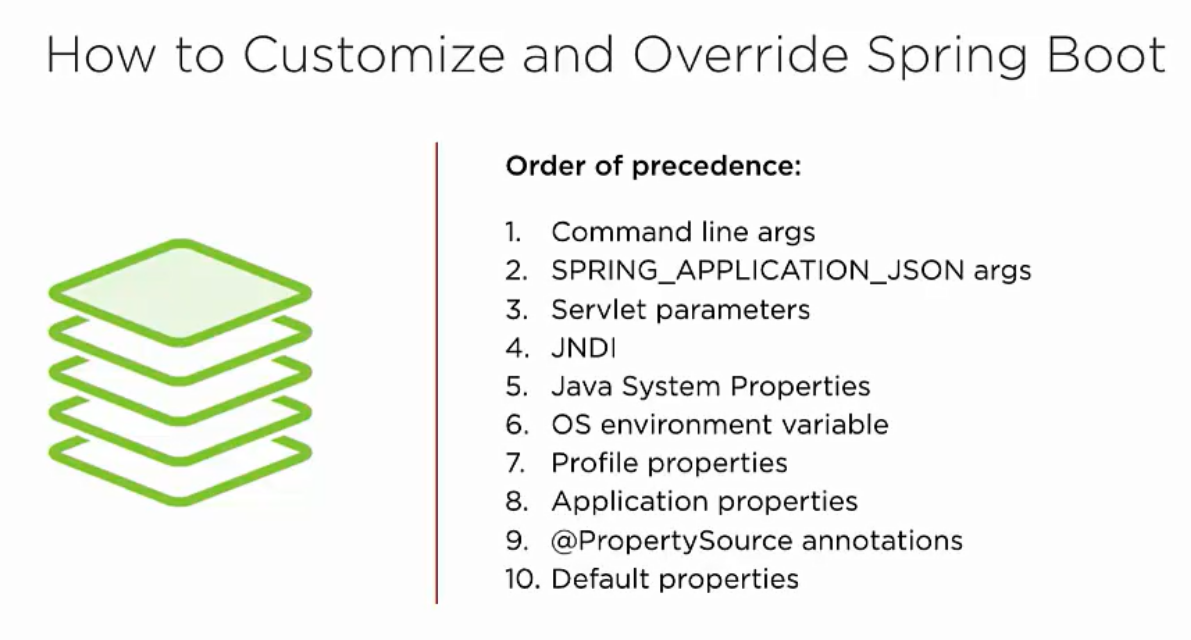
Q1) How to setup application configuration for diff env.

Q2) In dev we want all logs but in prod we only want error logs.





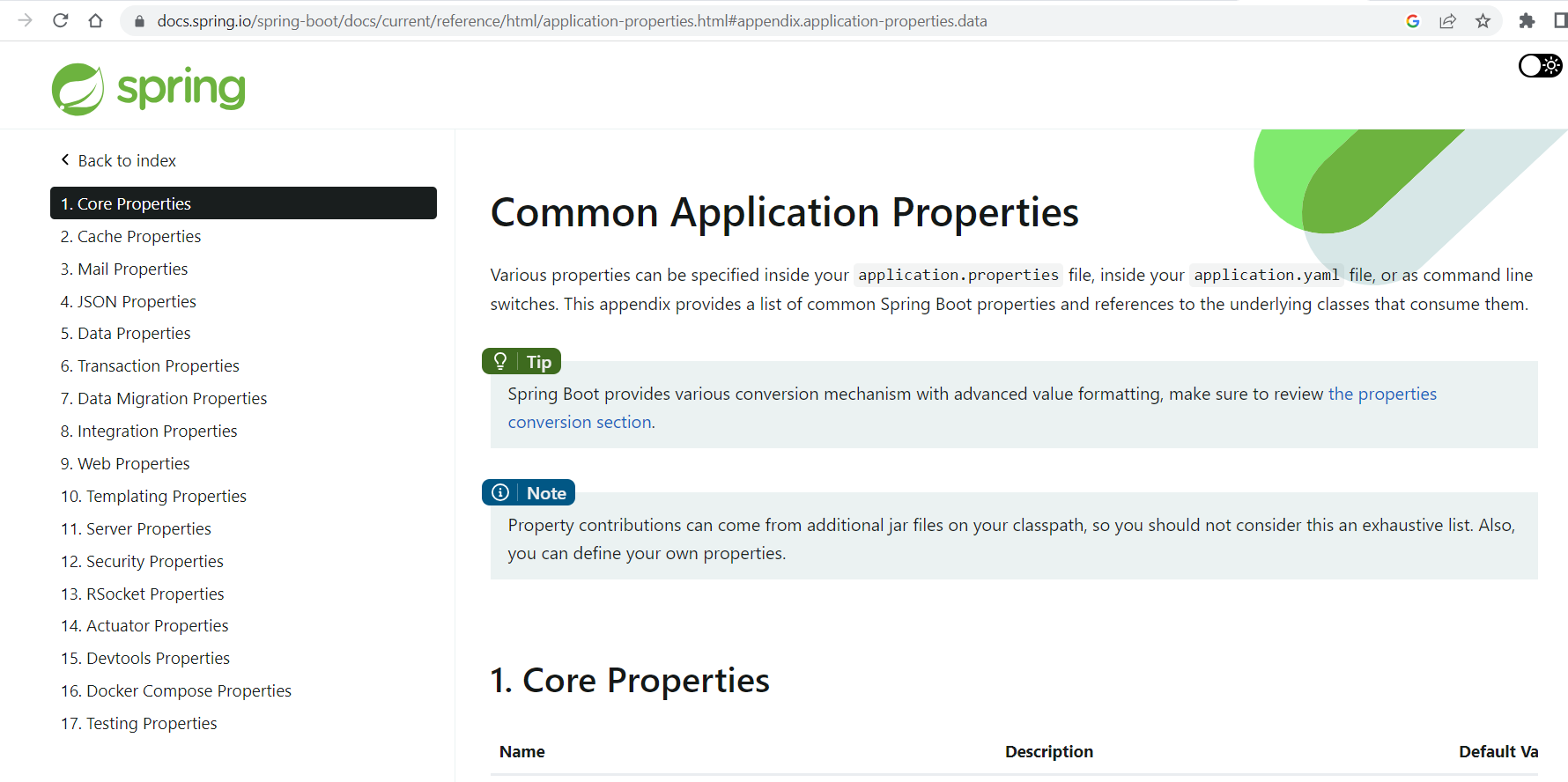
**External sources takes precedence over internal.**



We should have 1 property file as internal source to setup property defaults.

And should have 1 external source to override it.

**Common spring application properties**

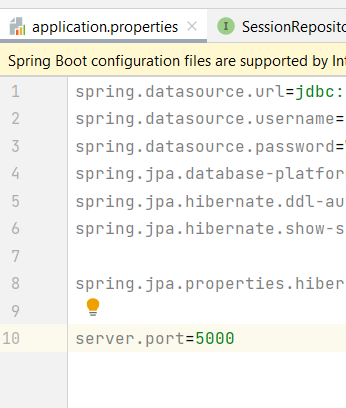


[https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties.data](https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html%23appendix.application-properties.data)

If we want to change the server-port on which the application is running we can simply change the default 8080 to any port.



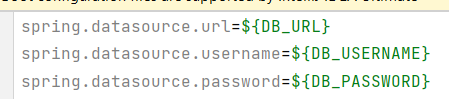
Change in application.properties

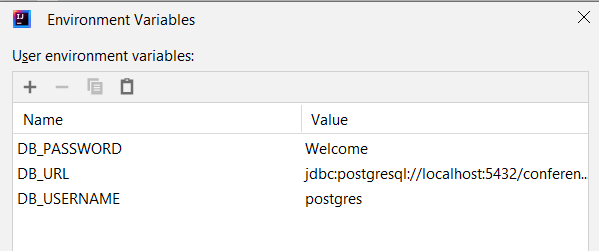


Writing and hardcoding the database connection URL username and password in application properties is not a good idea. In this way everyone having the app code can change it.

It is always advisable to keep database properties in some external environment variable.

Remove the hardcoded url ,username and password and put in environment variable at run configuration.

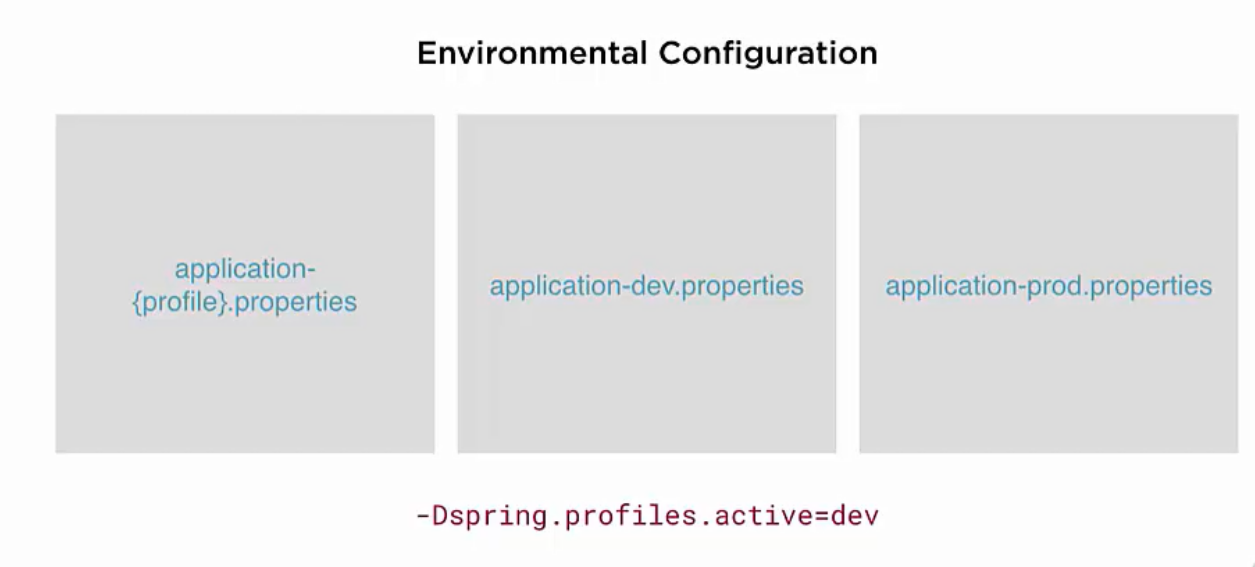






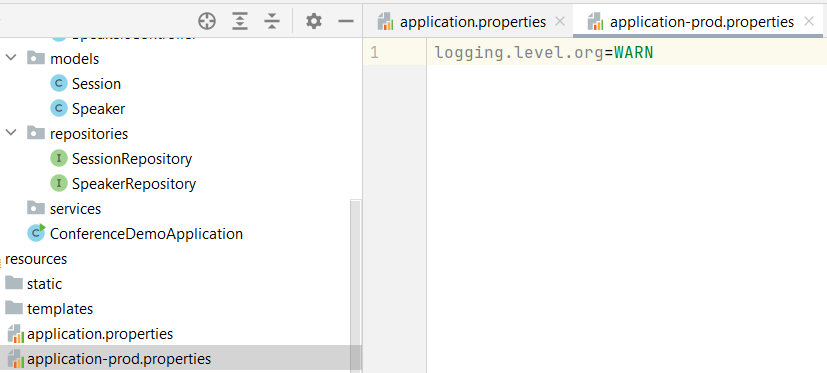
As we have different env and different env specific property so how does spring handle it.

It handles using **combination of profile and profile specific application property file.**

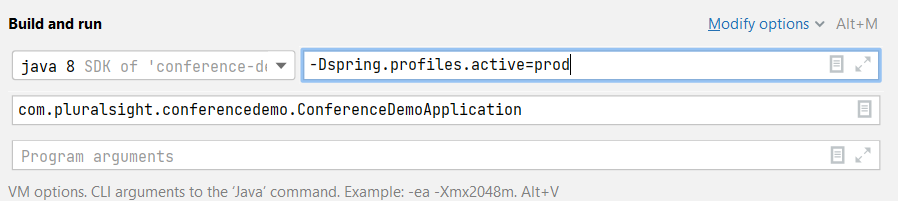


In this case **the dev profile** is active so spring boot will look for application-dev.properties file on the classpath which is source/main/resources of our project.

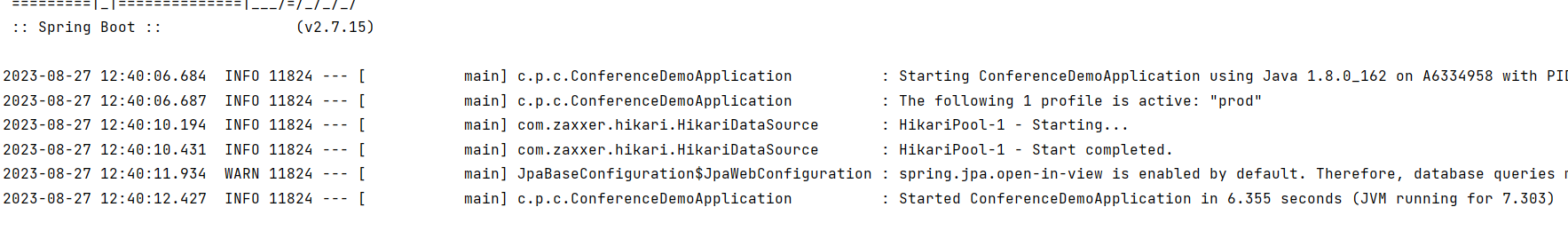
**Example:** Let suppose we want only warn or error log in production hence we create a new **application-prod.properties** in the resource folder.



In VM options of run configuration specify **-Dspring.profiles.active=prod**



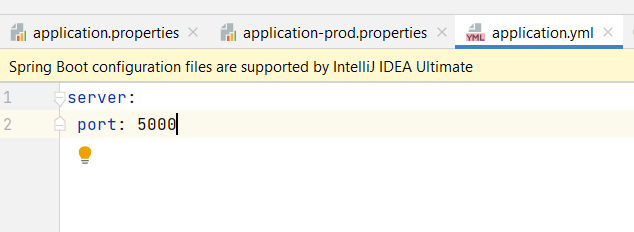
Our logs of info under org is gone now



**USING APPLICATION.YML FILE**

We can mix properties and yml file to work with but it is advisable to stick with one.

Example: we can take the server port and put in yml file and still things will work.



We can also use profiles like **application-prod.yml**

**HOW TO USE CUSTOME PROPERTY TO USE IN APPLICATION**

**Example:** Let suppose we create a new custom property called **app.version=1.0.0**

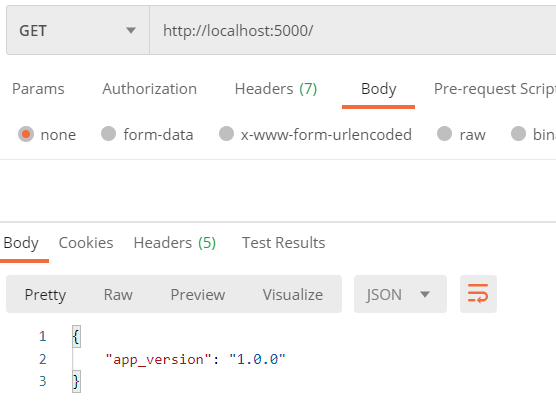
**We want to specify a app version when we call /home controller API.**

Since **JACKSON is our Marsheller** for Object to json it takes the map and its key value pair and send that as a JSON payload.

@Value("${app.version}")  
private String appVersion;

This tells the spring to find the value of app.version from properties file and inject it in the variable.





**CONFIGURATION OUTSIDE PROPERTY FILE USING 100% JAVA CONFIGURATION**

