CHAPTER 5 – WORKING WITH CONFIGURATION PROPERTIES

* 5.1 FINE TUNING AUTOCONFIGURATION:
  + Two different kinds of configuration in Spring:
    - Bean wiring: configuration that declares application components to be created as beans in the Spring application context and how they should be injected into each other
    - Property injection: Configuration that sets value on beans in the Spring application context.
  + These two types of configurations are often declared explicitly in the same place in Spring’s XML and Java-based configuration.
    - In java based configuration @bean is most likely to both instantiate a bean and then set values to its properties.
    - Example:

@Bean

Public Datasource dataSource(){

Return new EmbeddedDataSourceBuilder()

.setType(H2)

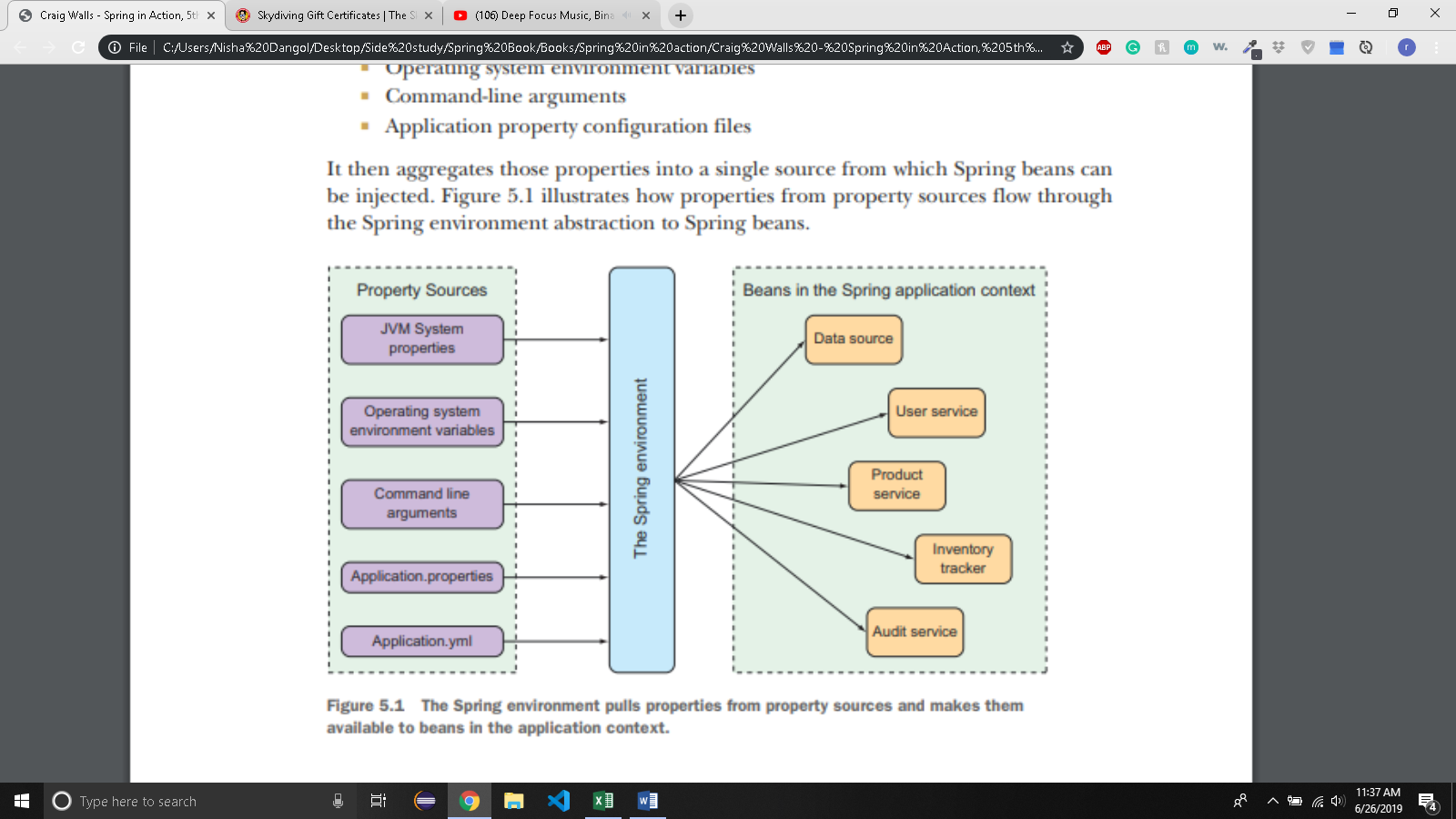
.addScript(\*taco\_schema.sql”)

.addScripts(“user\_data.sql”,”ingredient\_data.sql”)

.build();

}

* + - Here, the addScript() and addScripts()methods set some String properties with the name of SQL scripts that should be appliced to the database once the dataSource is ready.
    - Spring boot’s autoconfiguration makes this method completely unnecessary.
      * If the H2 dependency is available in the run-time classpath , then Spring boot automatically creates an appropriate Datasource bean in the Spring application context.
      * The bean applies the SQL scripts schema.sql and data.sql
    - However, if you want to add more sql scripts or change the sql name you need configuration properties.
* 5.1.1 UNDERSTANDING SPRING’S ENVIRONMENT ABSTRACTION:
  + Spring environment abstraction is a one stop shop for any configurable property.
  + It abstracts the origins of properties so that beans needing those properties can consume them from Spring itself.
  + The Spring environment pulls from several property sources:
    - JVM system properties
    - Operating system environment variables
    - Command line arguments
    - Application property configuration files
  + It then aggregates those properties into a single source from which Spring beans can be injected.



* + The beans that are automatically configured by Spring Boot are all configurable by properties drawn from the Spring environment.
  + Example: Suppose you want the application’s servlet container to listen for requests on 9090 instead of default port 8080.
    - In src/main/resources/application.properties:
      * server.port = 9090
    - Or, in src/main/resources/application.yml
      * server:

port: 9090

* + - Or, using command line arguments
      * $java -jar tacocloud-0.0.5-SNAPSHOT.jar - - server.port = 9090
    - Or, if you want the application to always start on specific port, you can set it as an operating system environment variable:
      * $ export SERVER\_PORT = 9090
* 5.1.2 CONFIGURING A DATA SOURCE
  + Later, you might want to use more permanent database.
  + In that case, you can configure the URL and credentials for your database via configuration properites.
  + Example: for MySql database, you might add following configuration properties in application.yml.

spring:

datasource:

url: jdbc:mysql://localhost/tacocloud

username: tacodb

password: tacopassword

* + Although you’ll need to add JDBC driver to the build, you don’t need to specify the JDBC driver class. Spring boot can figure it out from the structure of the database URL.
  + But if there’s a problem you can specify the class by adding at the bottom of the above code:
    - driver – class – name: com.mysql.jdbc.Driver
  + Spring bean uses this connection data when autoconfiguring the DataSource bean.
  + The DataSource bean will be pulled using the Tomcat’s JDBC connection pool if it’s available on the classpath.
  + If not, Spring boot looks for uses one of these connection pool implementation on the classpath:
    - HikariCP
    - Commons DBCP 2
  + There is a way to specify the database initialization scripts to run when the application starts. spring.datasource.schema and spring.datasource.data.

spring:

dataSource:

schema:

-order-schema.sql

-ingredient-schema.sql

-taco-schema.sql

-user-schema.sql

data:

-ingredients.sql

* + You can also configure the data source in JNDI and have Spring look it up from there. In that case, spring.datasource.jndi-name:

spring:

datasource:

jndi-name: java:/comp/env/jdbc/tacoCloudDS

* + - If you set the spring.datasource.jndi-name property, the other data source connection properties (if set) are ignored.
* 5.1.3 CONFIGURING THE EMBEDDED SERVER
  + If you set server.port = 0, the server won’t start on port 0 but on random available port.
  + This is useful when running automated integration test to ensure that any concurrently running tests don’t clash on a hard-coded port number.
  + It is also useful when you don’t care what port your application starts on because it’s a microservice that will be looked up from a service registry.
  + One of the most common things you need to do with underlying container is to set it up to handle HTTPs requests. For that, the first you need to do is create a keystore using JDK’s keytool command-line utility.

$keytool -keystore mykeys.jks -genkey -alias tomcat -keyalg RSA

* + You will be asked several questions about name and organization which is irrelevant.
  + But remember the password. For the sake of the example, I chose letmein as the password
  + It would be inconvenient to write all configuration in commandline. So, you can write them in application.properties or application.yml file.
  + In the yml file:

server:

port: 8443

ssl:

key-store: <file:///path/to/myKeys.jks>

key-store-password: letmein

key-password: letmein

* + - Here the server port is set to 8443 which is a common choice for development HTTPs servers.
    - The server.ssl.key-store property should be set to the path where the keystore file is created.
    - The key-store-password and key-password are set to the password that was given when creating the keystore.
  + With these properties in place,your application should be listening for HTTPS request on port 8443.
  + You might get waring about the server not being able to verify its identity, depending on your choice of browser. But don’t worry about it when serving from localhost during development.
* 5.1.4 CONFIGURING LOGGING
  + Even if your application doesn’t log directly, the libraries that your application uses will certainly log their activity.
  + By default, Spring boot configures logging via Logback to write to console at an INFO level.
  + For full control over the logging configuration, you can create a logback .xml file at the root of the classpath(in src/main/resources).
  + You can change the logging levels and specify a file where the logs should be written without creating the logback.xml file.
  + In application.yml:

Logging:

level:

root: WARN

org.springframework.security: DEBUG

* + - Sets the root logging level to WARN, but log Spring Security logs at DEBUG level.
  + Suppose you want to create the log entries to the file TacoCloud.log at /var/logs/. The logging.path and logging.file properties can help achieve that:

loggin:

path: /var/logs/

file: TacoCloud.log

level:

root: WARN

org.springframework.security: DEBUG

* + - Assuming that the application has write permissions to /var/logs/, the log entries will be written to /var/logs/TacoCloud.log. By default, the log files rotate once they reach 10 MB in size.
* 5.1.5 USING SPECIAL PROPERY VALUES:
  + When setting properties, you can derive their values from other configuration properties.
  + For example, suppose you want to set a property named greeting.welcome to echo the value of another property names spring.application.name. To achieve this, you could use ${} placholder markers when setting greeting.welcome

greeting:

welcome:${spring.application.name}

* + You can even embed that placeholder amidst other text:

greeting:

welcome: You are using ${spring.application.name}.

* 5.2 CREATING YOUR OWN CONFIGURATION PROPERTIES:
  + Configuration properties aren’t exclusive to the beans that Spring creates.
  + You can take advantage of configuration properties in your own beans.
  + As mentioned earlier, configuration properties are nothing more than properties of beans that have been designated to accept configurations from Spring’s environment abstraction.
  + How do those beans consume those configurations?
  + Spring boot provides @ConfigurationProperties annotation to inject configuration properties.
    - When placed on any Spring bean, it specifies that the properties of that bean can be injected from properties in the Spring Environment.
    - Suppose you have a method in OrderController to list the authenticated user’s past orders:

@GetMapping

Public String ordersForUser(@AuthenticationPrincipal User user, Model model){

model.addAttribute(“orders”, orderRepo.findByUserOrderByPlacedAtDesc(user));

return “orderList”;

}

* + - This will display all the orders placed by users from recent to the old. However, if you want to limit the number of orders shown in the page, you can do the following:
* @GetMapping

Public String ordersForUser(@AuthenticationPrincipal User user, Model model){

Pageable pageable = PageRequest.of(0,20);

model.addAttribute(“orders”, orderRepo.findByUserOrderByPlacedAtDesc(user,pageable));

return “orderList”;

}

* + - In the repository interface, you can make following changes:

List<Order> findByUserOrderByPlacedAtDesc(User user, Pageable pageable);

* + - Pageable is Spring Data’s way of selecting some subset of results by a page number and page size. In the above code, you constructed a PageRequest object that implemented Pageable to request the first page(page 0) with a page size of 20 to get up to the 20 of the most recently placed orders for the user.
    - What if you want to change the page size later? Rather than hardcode the page size you can set it with custom configuration property.

@Controller

@RequestMapping(“/order”)

@SessionAttributes(“order”)

@ConfigurationProperties(prefix = “taco.orders”)

Public class OrderController{

Private int pageSize = 20;

Public void setPageSize(int pageSize){

this.pageSize = pageSize;

}

……..

@GetMapping

Public String ordersForUser(@AuthenticationPrincipal User user, Model model){

Pageable pageable = PageRequest.of(0,pageSize);

Model.addAttribute(“orders”,orderRepo.findByUserOrderByPlacedAtDesc(user,pageable));

Return “orderList”;

}

}

* + - @ConfigurationProperties’ prefix attribute is set to tacos.orders which means that when setting the pageSize property, you need to use a configuration property named taco.orders.pageSize.
    - The new pageSize property defaults to 20. But you can easily change it to any value you want by setting a taco.orders.pageSize property.
    - In application.yml

taco:

orders:

pageSize: 10

* + - Or, if you want to make a quick change while in production, you can do that so without having to rebuild and redeploy the application by setting the taco.orders.pageSize property as an environment variable:

$export TACO\_ORDERS\_PAGESIZE = 10

* 5.2.1 DEFINING CONFIGURATION PROPERTIES HOLDERS
  + How to set configuration data in property holders?
  + @ConfigurationProperties are often placed on those beans whose sole purpose in the application is to be holders of configuration data.
  + This keeps configuration-specific details out of the controllers and other application classes.
  + It also makes it easier to share common configuration properties among several beans that may make use of that information.

package tacos.web;

import org.springframework.boot.context.properties. ConfigurationProperties; import org.springframework.stereotype.Component;

import lombok.Data;

@Component

@ConfigurationProperties(prefix="taco.orders")

@Data

public class OrderProps

{

private int pageSize = 20;

}

* + - Here, the pageSize property defaults to 20.
    - OrderProps is annotated with @ConfigurationProperties with a prefix of taco.orders.
    - @Component will make this class as a bean during runtime
    - Configuration property holders are just beans that have their properties injected from the Spring environment.
    - They can be injected into any other bean that needs those properties.
    - For OrderController, this means removing the pageSize property from OrderController and instead injecting and using the OrderProps beans:

@Controller

@RequestMapping("/orders")

@SessionAttributes("order")

public class OrderController {

private OrderRepository orderRepo;

private OrderProps props;

public OrderController(OrderRepository orderRepo, OrderProps props)

{

this.orderRepo = orderRepo;

this.props = props;

}

...

@GetMapping

public String ordersForUser( @AuthenticationPrincipal User user, Model model)

{

Pageable pageable = PageRequest.of(0, props.getPageSize()); model.addAttribute("orders", orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

return "orderList";

}

...

}

* + - * Now OrderController is no longer responsible for handling its own configuration properties.
      * This allows to reuse the properties in OrderProps in any other bean that may need them.
    - Now, let’s say you are using pageSize property in several other beans when you decide to limit its value to no less than 5 and no more than 25. If you didn’t have the holder bean(OrderProps), you would have to change the code in all the beans that used pageSize. But if you have OrderProps, you only need to make change to that:

package tacos.web;

import javax.validation.constraints.Max;

import javax.validation.constraints.Min;

import org.springframework.boot.context.properties.ConfigurationProperties; import org.springframework.stereotype.Component;

import org.springframework.validation.annotation.Validated;

import lombok.Data;

@Component

@ConfigurationProperties(prefix="taco.orders")

@Data

@Validated

public class OrderProps

{

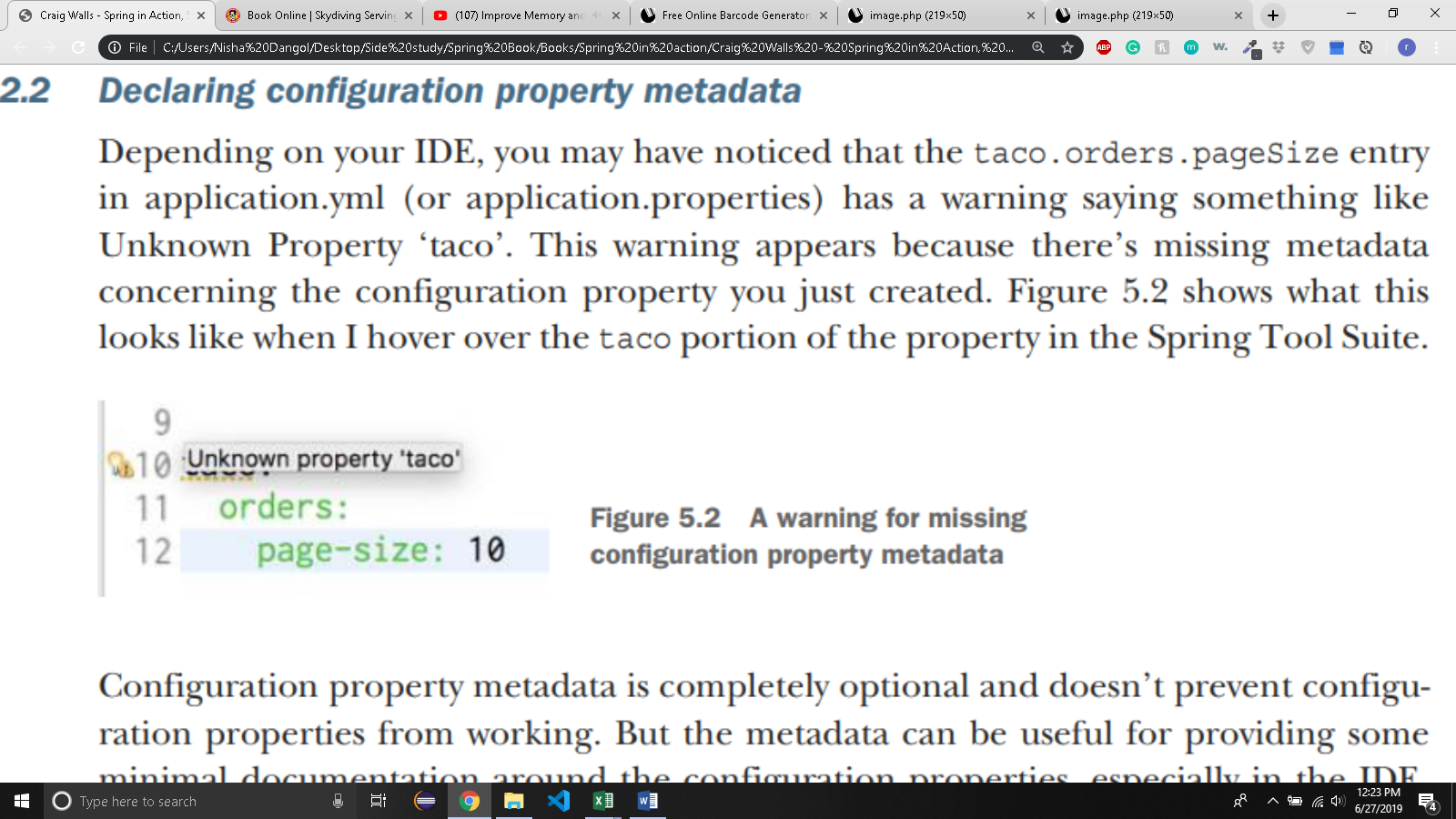
@Min(value=5, message="must be between 5 and 25")

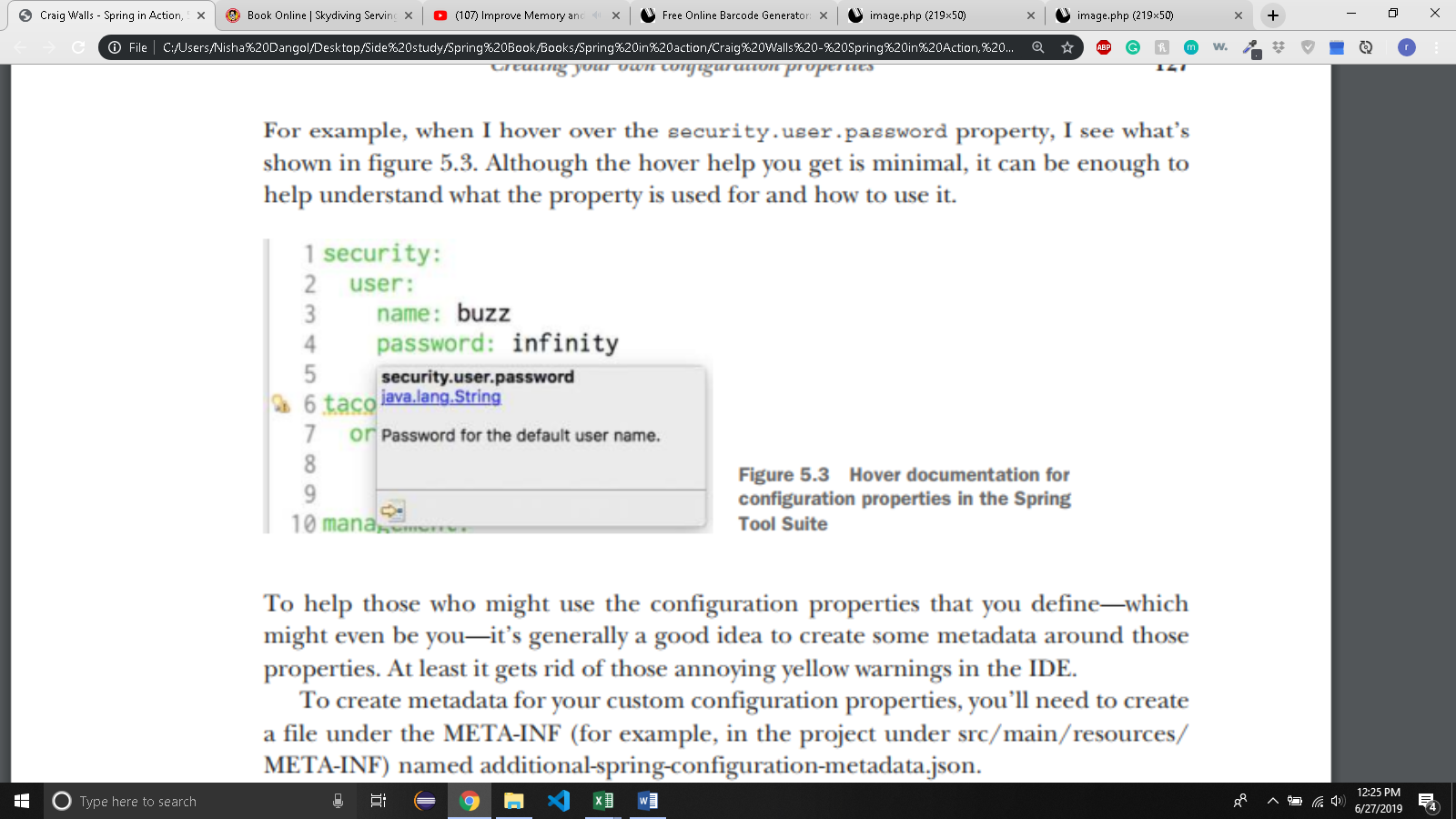
@Max(value=25, message="must be between 5 and 25")

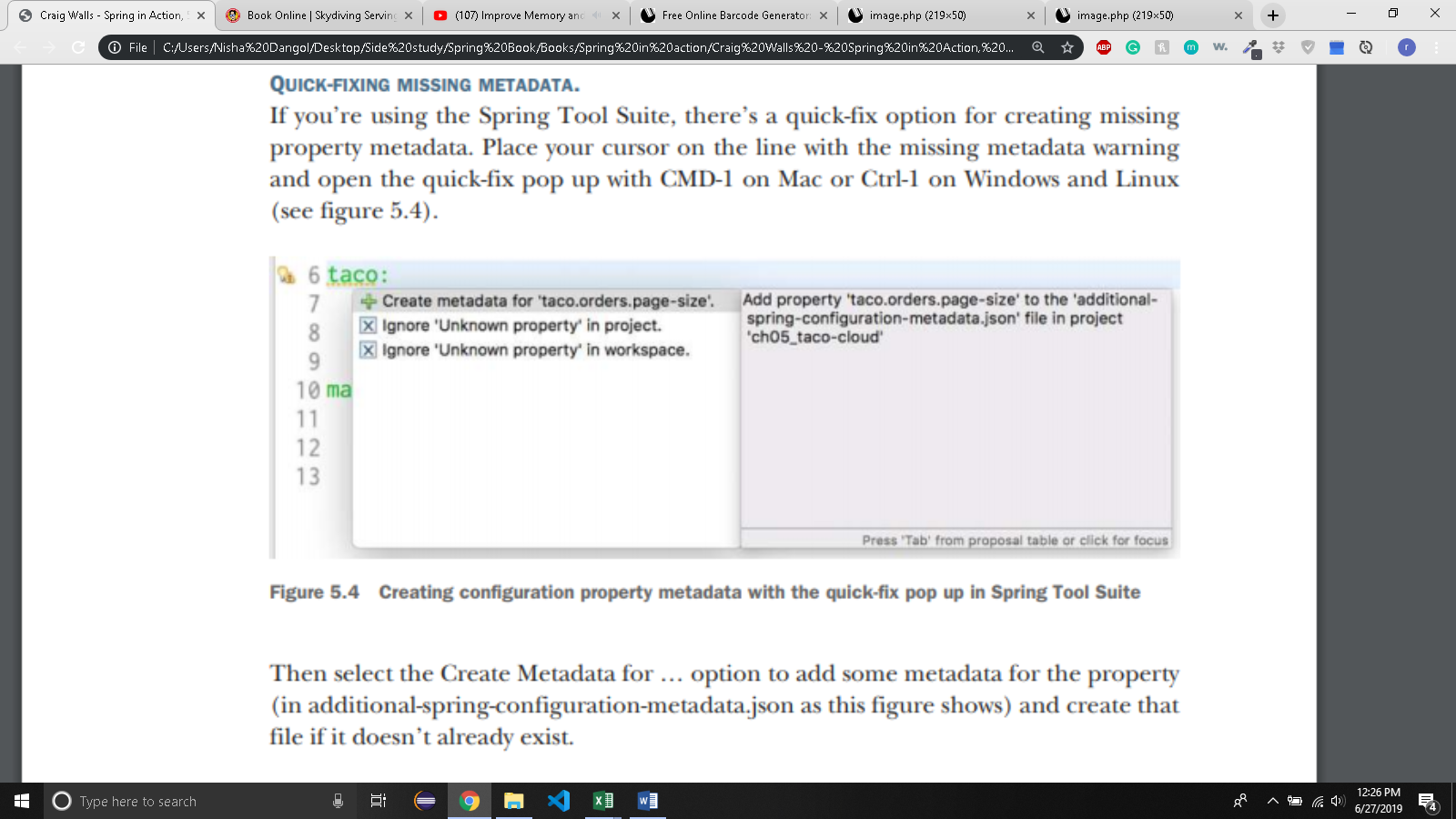
private int pageSize = 20;

}

//end::validated[]

* 5.2.2 DECLARING CONFIGURATION PROPERTY METADATA
  + Depending on your IDE, you may have noticed a warning in taco.orders.pageSize entry in application.yml saying ‘Unknown property taco’.
  + This warning appears because there is missing metadata concerning the configuration property you just created.
* 
  + Configuration property metadata is optional and doesn’t prevent configuration properties from working.



* + 
  + For taco.order.pageSize, you can set up the metadata with following JSON:

{

“properties” : [

{

“name”: “taco.order.page-size”,

“type”: “java.lang.String”,

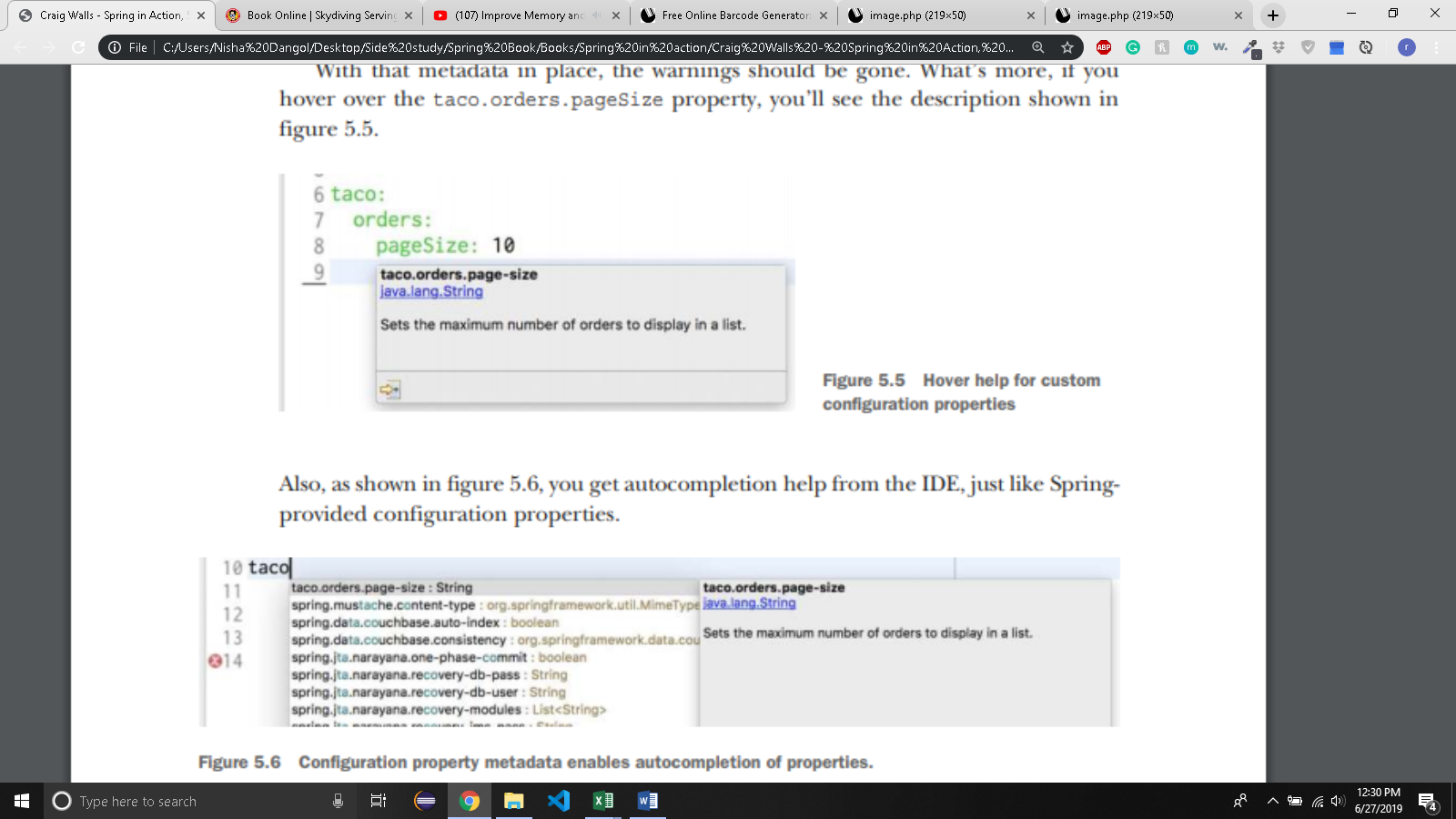
“description”: “sets the maximum number of orders to display in a list”

}

]

}

* + - taco.orders.page-size is equivalent to taco.orders.pageSize



* + What if you need to configure different properties for different deployment environments. Let’s see how to use Spring profiles to set up environment specific configuration.
* 5.3 CONFIGURINIG WITH PROFILES
  + When applications are deployed to different run time environments, usually some configuration details differ.
  + Profiles are a type of conditional configuration where different beans, configuration classes, and configuration properties are applied or ignored based on what profiles are active at runtime.
  + Suppose you want to use H2 database and set logging level to DEBUG during development. In application.yml

logging:

level:

tacos.DEBUG

* + But in production, you want to use MySQL and logging level to WARN. If you want to deploy this application in a production setting with no further changes to application.yml, you’d still have DEBUG logging for tacos package and an embedded H2 database. You need to define a profile with properties suited for production.
* 5.3.1 DEFINING PROFILE SPECIFIC PROPERTIES
  + Create yet another YAML or properties file containing only the properties for production.
  + The naming convention: application-{profile name}.yml or application-{profile name}.properties.
  + Then specify the configuration properties appropriate to that profile.
  + For example, create new file application-prod.yml:

spring:

datasource:

url: jdbc:mysql://localhost/tacocloud

username: tacouser

password: tacopassword

logging:

level:

tacos: WARN

* + - You can define properties for as many profiles as you need by creating addional YAML or properties file.
* 5.3.2 ACTIVATING PROFILES
  + Best way to set the active profiles with the environment variables
  + On the production environment, you would set the SPRING\_PROFILES\_ACTIVE like this:

% export SPRING\_PROFILES\_ACTIVE=prod

* + From then on, any applications deployed on that machine will have the prod profile active and the corresponding configuration properties will take precedence over the properties in the default profile.
  + If you’re running the application as the executable JAR, set the active profile with a command-line argument like this:

% java -jar taco-cloud.jar --spring.profiles.active=prod

* + You can specify more than one active profile:

% export SPRING\_PROFILES\_ACTIVE=prod,audit,ha

* 5.3.3 CONITIONALLY CREATING BEANS WITH PROFILES
  + Suppose there are beans that you only need when certain profiles are active
  + For instance the CommandLineRunner bean declared to load the embedded database with ingredient data when the application starts. To prevent the ingredient data from being loaded every time the application starts in a production deployment, you could use @Profile

@Bean

@Profile("dev")

public CommandLineRunner dataLoader(IngredientRepository repo, UserRepository userRepo, PasswordEncoder encoder)

{

...

}

* + Suppose you need Commanline runner only when dev or qa profile are active:

@Bean

@Profile("dev",”qa”)

public CommandLineRunner dataLoader(IngredientRepository repo, UserRepository userRepo, PasswordEncoder encoder)

{

...

}

* + Or you want commandline runner everytime except the prod profile:

@Bean

@Profile(“!prod”)

public CommandLineRunner dataLoader(IngredientRepository repo, UserRepository userRepo, PasswordEncoder encoder)

{

...

}

* + It’s also possible to use @profile on an entire @Configuration-annotated class.

@Profile({"!prod", "!qa"})

@Configuration

public class DevelopmentConfig

{

@Bean

public CommandLineRunner dataLoader(IngredientRepository repo, UserRepository userRepo, PasswordEncoder encoder)

{

...

}

}