Spring Tool Suite – STS

Maven- Apache build tool

Spring Bean

* element/tag defined in the Spring Bean Configuration File
* objects that are managed by the spring application container (efficiently instantiate or destroy)
* used to create instances of classes through our spring app
* mapping of application classes with unique identifiers that spring can use
* contains
  + unique id
  + reference to a class
  + class should have default no argument constructor
* can wire up beans in our configuration file
* Scopes
  + Singleton (default) – single instance per IoC container –good for stateless
  + Prototype- multiple instances per IoC container- any instances of that bean that are created are no longer managed by Spring- good for stateful beans
  + Request – one instance per HTTP request
  + Session- one instance per HTTP session
  + Global Session – one instance per global HTTP Session (for Portlest-based web apps)

Annotation Configuration

* another method of configuring Spring Applications
* alternative to strictly XML configuration
* adding specific annotations to our source files
* context namespace- <context:annotation-config />
* still use XML for ‘base’ bean definitions/ annotations for DI (dependency injection)
* annotations for bean definitions and DI <context:component-scanner />

JSR Java Specific Requests

Spring Expression Language

* useful for manipulating and querying object graph at runtime
* built to work with Spring framework
* can be used independently of Spring Framework
* Functionality
  + literal expressions
  + regular expressions
  + mathematical operators
  + Boolean operators
  + relational operators- lt instead of < and gt
  + invoke methods
  + access and set properties
  + work with bean references
* works with annotation and XML

Safe Navigation Operator (?)- apple.getValue()?.toUpperCase()

Modules

* core container
* data access
* web
* testing

Features

* leverages spring
* opinionated
* embeds web server
* maven/gradle support
* little or no code generation
* little or no XML configuration
* favors convention
* written in Java (or Groovy)
* supports containerized apps
* built-in web server
* expresses REST services
* maintenance is easy
* automatic configuration
* use annotations
* starters available
* app deployed via JAR file

Commands (mostly conditional)

* @EnableAutoConfiguration //can be overridden
* custom configuration classes
* @ConditionalOnProperty
* @PropertySource(“classpath:mysql.properties”)
* @ConditionalOnProperty(
  + name=”ssmysql”,
  + havingValue = “ssenv” )
* @ConditionalOnResource
* @ConditionalOnWebApplication
* @ConditionalOnExpression

Configuring DB Access in Spring

* Database Driver dependency on db such as
  + mysql-connector-java –com.mysql.jbdc.Driver //for SQL dbs
* Data Source bean – username, password, db driver(^), db url
  + commons-dbcp –org.apache.commons.dbcp.BasicDataSource
  + spring-jdbc –org.springframework.jbdc.datasource.DriverManagerDataSource
* class MyDataAccessObject
* private myJdbcTemplate (instance)
  + spring-jdbc –org.springframework.jdbc.core.JdbcTemplate
* ConfiguringADataSource/pom.xml can add dependenciy via spring
  + dependencies – add
    - mysql mysql-connector-java
    - DriverManagerDataSource
* dbinfo.properties
  + dbUsername=admin
  + dbPassword=admin
  + dbDriver=com.mysql.jdbc.Driver
  + dbUrl=jdbc:mysql://localhost:3306/springdemodb
* bean id=”myDriverManagerDataSource”
* <context:property-placeholder location=dbinfo.properties />
* JbdcTemplate class
  + commonly used for executing queries against a db
  + part of spring-jdbc jar
  + create an instance by passing in a DataSource
  + new JdbcTemplate(myDataSource)
* DAO class with @Component
* return myJbdcTemplate.query(“select \* from foodgroups”, new RowMapper<DAO>)

Database Exception Handling in Spring

* Spring translates exceptions into classes in its own exception handling hierarchy
* no need for devs to handle database related exceptions
* DataAccessException the root of hierarchy
  + ex DeadlockLoserDataAccessException
* exceptions are runtime errors (no need for try catch blocks)
* @Repository annotation registers our Data Access Object components with Spring and ensures that our objects are going to provide the exception translations

Spring Basic Database Operations

* JdbcTemplate class
  + query(), queryForObject()
  + update()
  + execute() for arbitrary code
* NamedParameterJdbcTemplate class
  + (Select \* from food where id = :12)
* SimpleJdbcInsert class
* SimpleJdbcCall, Integer.class

Aspect Oriented Programming

* isolating cross cutting concerns into aspects
  + cross-cutting concerns are aspects of a program that affect other concerns
  + ccc are parts of a program that rely on or must affect many other parts of the system
  + ex. logging metrics will be used through MSs but they do not relate directly to the classes that implement them
* tracing, security, and transaction management are cross-cutting concerns
* implementing AOP in Spring Apps
  + Spring AOP
    - XML based
    - Annotation based (AspectJ)
    - integrates nicely with Spring IoC container and easy to set up
    - can only work with method executions as joinpoints
  + Aspect J Project
    - more available joinpoints
    - not as easy to set up and use in a Spring app
* Aspect
  + module that is responsible for providing cross-cutting requirements
  + a class
* Advice
  + actual segment of code that implements a cross-cutting requirement
  + a method in the aspect class
* Join Point
  + potential point in an applications execution where we can plug in an aspect
  + in method executions or exception handling
* Pointcut
  + one or more join points where advice should be executed
  + configuration of one or more join points where our advice should be executed
* Introduction
  + we can declare new methods or attributes to existing classes
* Target Object
  + the object that is being advised
* Weaving
  + process of linking aspects with rest of the application types to create advised objects
* Implementation-
  + <aop:config>
  + <aop:aspect id=”loggerAspect” ref=”log”>
  + <aop:before method=”asdf” pointcut=”execution(void com.demo.Car.drive())” />
    - <aop:pointcut expression=”execution(void com.demo.Car.drive())” id=”qwer” />
    - <aop:before method=”asdf” pointcut-ref=”qwer” />
* Annotation implementation-
  + @Component
  + @Aspect
  + public class BusinessLogic {
    - @Before(”execution(void com.demo.Car.drive())”)
    - public void asdf(){
  + @Configuration
  + @ComponentScan(“com.demo.aop”)
  + @EnableAspectJAutoProxy
  + public class AspectConfig{
  + main method {
  + ApplicationContext d = new AnnotationConfigApplicationContext(AspectConfig.class)

Advice Types

* @Before(“execution(void com.demo.Car.drive())”)
  + parenthesis states where in the business logic the advice will be connected to, is used immediately before the advice aka - log(“Vehicle about to move”)
  + can throw exception to prevent the business logic code being run
* @AfterReturning(pointcut=”execution(void com.demo.drive())”, returning=”ret”)
* public void vehicleAboutToMove(String ret){
  + runs after unless exception is thrown
  + allows business logic to access advise’s return
* @AfterThrowing(pointcut=”execution(void com.demo.drive())”, throwing=”ex”)
* public void vehicleAboutToMove(Exception ex){
  + only if exception is thrown,
  + this advise has access to exception ex
* @After(”execution(void com.demo.drive())”)
  + called After finally
  + executes no matter what
* @Around(”execution(void com.demo.drive(..))”)
* public void vehicleAboutToMove(ProceedingJoinPoint pjp) throws Throwable{
  + logs “before”;
  + try {
    - pjp.proceed();
  + } catch(Exception ex){
    - logs “exception” + ex;
  + }
  + logs “after”;
* }
  + runs before and after
  + most powerful
  + can manipulate parameters in business logic from within Around advice
  + can control what is being returned to the caller
  + Object[] callerArgs = pjp.getArgs()
  + int speed = ((Integer)callerArgs[0]).intValue();
  + Object[] newArgs = new Object[1]
  + try {
    - pjp.proceed(newArgs)

Pointcuts

* used to identify one or more joint points where advice should be executed
* Declaring a Pointcut
  + Signature – name and parameters
  + Expression – determines which JoinPoints will be matched
  + @Pointcut(“execution(void drive())”)
  + public void demo(){}
* Pointcut Expressions
  + identify specific join points explicitly
    - “execution(String com.demo.Car.drive())”
  + use patterns to match one or more join points
    - “execution(String com.demo.car.\*())”
* Pointcut Designators
  + refine which join points our advice matches to
  + common designators in Spring Aop
    - *execution, within, args, …*
* @Pointcut(“execution(\* com.demo.Car.dr\*(..))”)
  + first \* means any return type
  + second \* means matches to any method that starts with “dr”
  + .. in the () means it accepts any number of parameters
* Spring AOP Pointcut Designators
  + Spring AOP supports subset of designators from AspectJ language
  + execution – matching method execution join points
  + within – matching join points within certain types that we define
  + this – matching join points where the bean reference is an instance of a given type
  + target – matching join points where the target object is an instance of a given type
  + args – matching join points where method has number and type of arguments we define
  + @target- mjp where class of executing object is annotated with specific annotations
  + @args - mjp where the runtime type of actual arguments passed have types we specify
  + @within – mjp within types that have a specified annotation
  + @annotation – where the method being executed has a specified annotation
  + bean (specific to Spring AOP) – for a bean with a given name

Spring Security

* standard for securing Spring-based apps
  + stand alone and web based
* powerful and highly customizable authentication and access-control framework
* Authentication
  + In Memory
    - define the users directly within our security configuration
  + Databases
  + LDAP
  + OpenID
* Authorization
  + HTTPRequest level
  + Method level
  + Permissions level
* Security namespace
  + <security:authentication-manager>
    - <security:authentication-provider />
  + <security:authentication-manager/>
  + <security:http>
  + </security:http>
* add dependency-
  + groupID: org.springframework.security
  + artifact: spring-security-web
* filter
  + <filter>
    - <filter-name>
    - <filter-class>
  + <filter-mapping>
    - <filter-name>
    - <url-pattern>/\*
  + user-service-ref=

Authenticating Spring Security with DB

* configure a data source
* create a data access object
  + org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl
* set up authentication provider with data access object
* create a user schema with a
  + - users table includes- username, password, enabled
    - authorities table- username, authority

Spring Security and Database Encrypted Passwords

* Spring Security Classes (encoding/encrypting)
  + MD5
  + SHA
  + BCrypt (recommended)
* Spring Security JSP Tags
  + Dependencies – org.springframework.security spring-security-taglibs
  + JSP declaration: <%@ taglib prefix=”sec” uri=”springframework.org/security/tags” %>
  + authorize tag – to determine if the contents of the tag itself should be evaluated
    - <sec:authorize ifAnyGranted
  + authentication tag
    - <sec:authentication property=”name />
  + accesscontrollist tag – to wrap content and then evaluate whether the content should be shown
  + crsfInput – include a hidden form field that contains name and value of protection token
  + csrfMetaTags – inserts metatags that contain CSRF information

Customizing Spring Security

* customizations initiated by configuring the http element within the application context
* custom login page using form-login tag
* same for logout page
* separate authentication-failure-url for authentication errors
* authorization errors (403) <security:access-defined-handler error-page=””>

Access Control Using Expressions

* expressions can secure- URLs or Methods
* set user-expressions=”true”
* expressions are Boolean
* common expressions-
  + hasRole (hasAuthority)
  + hasAnyRole (hasAnyAuthority)
  + permitAll
  + denyAll
  + hasPermission
* hasRole(‘ROLE\_USER’)
  + security:http auto-config=”true
  + security:intercept-url access=
* @PreAuthorize

Spring Configurations

* @EnableAutoConfiguration – does most heavy lifting but can be overridden
* custom config classes
* @ConditionalOnClass(DataSource.class)
* @ConditionalOnBean - @ConditionalOnMissing
* @ConditionalOnProperty
* @PropertySource(“classpath☺ part of property
* @ConditionalOnResource
* @ConditionalOnWebApplication
* @ConditionalOnExpression

Spring Security- adding OAuth

maven dependencies-

<dependency>

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

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<dependency>

<groupId>com.okta.spring</groupId>

<artifactId>okta-spring-boot-starter</artifactId>

<version>1.0.0</version>

</dependency>

Application Class-

@SpringBootApplicatoin

@EnableTransactionManagement

@EnableResourceServer

@EnableGlobalMethodSecurity(prePostEnabled=true)

public class Application {

public static void main(String[] args){

SpringApplication.run(Application.class, args); }

@Configuration

static class OktaOauth2WebSecurityConfigurerAdapter extends WebSecurityConfigurerAdapter{

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests().anyRequest().authenticated()

.and()

.oauth2ResourceServer().jwt();

Register free Okta Account-

<https://developer.okta.com/>

select Service

copy Client ID and enter it in the application.properties file aka ${clientID}

Acquire a token from -

<https://oidcdebugger.com/>

select Web

set Redirect URI to <https://oidcdebugger.com/debug>

export TOKEN=${YOUR\_TOKEN}

http http://localhost:8080 "Authorization: Bearer $TOKEN"

Spring Boot Security

1st step- add spring-boot-starter-security to pom.xml file

user and given password in application.yml

* spring:
  + security:
  + user:
    - name: asdf
      * password: asdf

Security Flow

principle 🡪 authentication 🡪 securityContext 🡪 HTTP Session

Filters

* SecurityContext Persistence Filter
  + security context holder – creates an empty security context for every request (unless you already have one)
* Logout Filter
* Authentication Filter - delegates to
  + authentication manager bean (interface). authentication manager will take authentication request object and returns an authentication object. it delegates to
    - authentication provider to do actual authentication and it needs original user details and delegates to
      * user details
    - **authentication request**
      * authenticated: false
      * principle: username
      * credentials: password
    - **authentication principle**
      * authenticated: true
      * principle: UserDetails
      * Authorities: eg ADMIN
* Remember me Filter – checks for cookie
* Exception Translation Filter – checks for exception, redirects you to login if necessary
* FilterSecurityInterceptor – is security configured on homepage? aka is it a secured resource?

**SPRING SECURITY**

SecurityConfiguration extends WebSecurityConfigurerAdapter

@Override

Configure(Authentication Manager Builder)

@Override

Configure(HttpSecurity)

Maven dependency- ***spring-boot-starter-security*** and ***spring-security-test***

Or – Add Spring Security during init

Application.properties

Server.port=6666

Spring.application.anme=Main (or other name with Main method)

@SpringBootApplication

Public class Main {

@Value(“${server.port”})

Private int portNum;

@Value(“${spring.application.name}”)

Private String appName;

@Bean CommandLineRunner cliRunner(){

Return args -> System.out.println(appName + “ is running”)

Main method

Steps

1. Extends WebSecurityConfigurerAdapter
2. @Configuration annotation to class
3. @EnableWebSecurity to class
4. Set up Authentication via @Override configure(AuthenticationManagerBuilder auth)
5. Setup Authorization via @Override protected void configure(HttpSecurity http)
6. Set up Password Encoder

Ex 6

@Bean

Public PasswordEncoder passEncoder(){

Return new BCryptPasswordEncoder();

Ex 4

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

auth.inMemoryAuthentication().

.withUser(“user1”).password(passEncoder().encode(“pass”)).roles(“USERS”);

auth.jdbcAuthenticatoin(); // or 2

auth.ldapAuthentication(); // or 3

}

Ex 5

@Override

Protected void configure(HttpSecurity http){

http.authorizeRequests()

.antMatchers(“/\*\*”).hasRole(“USER”)

Or .antMatchers(“/\*\*”).hasAnyRole(“USER”, “ADMIN”);

Apache Ant - ? = one char, \* = 0 or more chars, \*\* = 0 or more folders in a path