**Restful web service creation**

1. **Jackson Processor** is used to convert a JAVA object into JSON and vice versa. It is available as **an external JAR** that can be downloaded from maven repository by adding **Jackson-core-asl and Jackson-mapper-asl** as maven dependencies It can be **applied to a JAVA object though an annotation** @**JSONAutoDetect.**
2. In Spring MVC a URL is mapped to a JAVA class using annotations @Controller & @RequestMapping. These are available in **spring-mvc artefact**, and can be downloaded as maven dependency. **Appropriate version of spring-mvc** is required in order to use @RequestMapping with “headers” and “produces” parameters. Used 4.0.9..RELEASE
3. A spring MVC project has basic dependency requirements for **spring-core, spring-web & spring-webmvc artifacts** that should be included in maven dependencies. Versions for these artefacts should be same i.e. Tested for 4.0.9.RELEASE
4. **Context is provided for Dispatcher servlet through an XML file containing MVC specific declarations such as component scan base package and view resolver path. On a http request, all classes annotated as @controller are searched for request handlers by framework**
5. **ContextLoaderListener** with **contextConfigLocation** parameter is **used to integrate Spring with JAVA web framework**. **contextConfigLocation** is path to an XML file referencing spring namespaces.
6. **View Resolver by default loads index.jsp or a default welcome file, if present in webapp folder for root path. Otherwise controller method is invoked. These methods return a String or a ModelView object that is mapped by view resolver to a JSP file in specified path.**
7. **A Restful web service can be quickly created as a Spring boot project through STS, Spring Starter Project. A template project gets created, can be run as a JAVA application launching application start-up page in browser. Starter project is auto deployed on embedded tomcat container. All maven dependencies needed for REST are auto generated. Including Actuator support within project enables monitoring service metrics as URL paths – env, metrics, dump, trace, health, beans, info. Metrics display how many times service has been executed. Trace gives stack trace. Dump displays memory dump. Service can be shut down through shutdown URL**
8. **JAX-RS is set of APIs for developing Restful Web services and has been a part of Java specifications since JavaEE6. Jersey is Open Source framework based on JAX-RS for building and consuming Restful Web services. It is mainly used to consume Restful APIs from different providers and developed on different platforms. Spring has created projects RestEasy & RestTemplate to consume Rest Services. These are part of Spring MVC module which needs to be included.**
9. **Basic REST operations can be tested through Postman.**
10. **HATEOAS is a way to provide links to resources in HTTP response to client**
11. **While creating a REST service using Jersey, additional JARs need to be made available in class-path. Maven provides jersey-webapp project template through which all Jersey dependent JARs are downloaded to local repository. Jersey libraries include Rest specific annotations i.e. @PATH, @Get/@POST, @Produces etc. All URL requests are directed to respective classes with these annotations.**
12. **Rest API supports several output types like Plaintext, XML, JSON based on arguments provided to @Produce/@Consume annotations. Additional annotations like @XmlRootElement should be applied to convert objects or collections to desired returned format. URL query parameters are accessible through @QueryParam annotation on method parameters**
13. **In addition to @PathParam and @QueryParam annotations, there is support for @MatrixParam, @HeaderParam & @CookieParam annotations. @HeaderParam is used to access custom values provided in request header i.e. Authentication info. @CookieParam allows access to cookies**
14. **JAX-RS has its own set of exceptions for which it provides in built exception mappers. For other exceptions it supports writing Custom exception mappers, which then intercepts exception handling method of application server and sets its own status code along with content. All exception mappers are annotated with @Provider annotation**
15. **Client on its end uses HTTP headers “Accept” and “Content-Type” to inform server of the format of request. If client needs response in form of JSON or XML it specifies in “Accept” header. For PUT/POST to server it uses “Content-Type” header**
16. **While Spring MVC framework creates a complete web application including UI forms and controller methods for retrieving data, A Restful web service exposes selected resources to clients. Clients can fetch data by consuming service methods and display data in its own frontend.**
17. **Security in Restful services: In JAX-RS a "filter" can modify request and response. For basic authentication encoded user ID/password are passed in header. Jersey supports a HttpBasicAuthFilter to do this, which places user ID/password in authorization header of http request header. Client has a class method AddFilter through which filter can be added**

***client.addFilter(new HttpBasicAuthFilter("userId", "password");***

1. **Security in Restful services over SSL requires use of certificates. A Self signed certificate can be created for testing purpose for web application, however browser will get a warning that certificate is not from a verified source**

**Securing REST service through BASIC Authentication**

1. **To incorporate Spring security to an application, security xml configuration is added to an XML file (security-context.xml) and that should be included into web.xml as a parameter in <context-param> section.**
2. **Additional JARs required as dependencies for spring security i.e. spring-security-core, spring-security-web, spring-security-acl, spring-security-core should be added as Maven dependencies. Tested with version 3.2.10. Schema version of spring security in XML configuration should match with version of JARs.**
3. **Spring Security maintains a filter chain internally as springSecurityFilterChain where each filter has a responsibility and filters can be added or removed from the configuration depending services required.**
4. **To enable filter chain add springSecurityFilterChain of type DelegatingFilterProxy in security configuration after <Context-Param> followed by listener classes ContextLoaderListener & RequestContextListener. This is followed by DispatcherServlet and other servlets.**
5. **A custom spring security login form gets created through tags using inbuilt variables j\_username & j\_password. Servlet j\_** **spring\_security\_check is responsible for performing authentication. Session log out is controlled through j\_spring\_security\_logout tag**

<a href="j\_spring\_security\_logout">logout </a>

<input type="text" name="j\_username" value="${USERNAME}">

1. **In Spring Security 3.X authorized user credentials can be specified within user-service tag in XML file. Password can be optionally encoded with <password-encoder/> tag**

<user-service>

<user name="concretepage" password="con1234" authorities="ROLE\_USER" />

</user-service>

1. **To enable spring security with database <jdbc-user-service> user tag is used, along with SQL query fetching credentials and authorities from DB**

<authentication-provider>

<password-encoder hash="sha"/>

<**jdbc-user-service** data-source-ref="dataSource" authorities-by-username-query="SELECT username, authority FROM authorities WHERE username = ?"

users-by-username-query="SELECT username, password, enabled FROM users WHERE username = ?"/>

</authentication-provider>

1. **Channel security feature enables https access to specified URL pattern with a change in configuration**

<intercept-url pattern="/login" access="ROLE\_USER" requires-channel="https" />

1. **User role is accessed in controller through GrantedAuthority class. A GrantedAuthority can be seen as permission or a right. A role is a type of Granted Authority. A user can be assigned different roles by putting them in security context**

private boolean hasRole(String role) {

Collection<GrantedAuthority> authorities = (Collection<GrantedAuthority>)

SecurityContextHolder.getContext().getAuthentication().getAuthorities();

boolean hasRole = false;

for (GrantedAuthority authority : authorities) {

hasRole = authority.getAuthority().equals(role);

if (hasRole) {

break;

}

}

return hasRole;

}

1. **UserDetails interface provide access to core user information. Its implementations, such as LdapUserDetailsImpl adds additional information used by LDAP services. UserDetails instance can be retrieved through** SecurityContextHolder

UserDetails userDetails = (UserDetails)SecurityContextHolder.getContext().

getAuthentication().getPrincipal();

1. **Point cut expressions are also supported with Spring security, enabling application of secured access on targeted methods**

<global-method-security >

<protect-pointcut expression="execution(\* com.concretepage.service.\*Service.\*(..))"

access="ROLE\_USER"/>

</global-method-security>

1. **Spring security configuration can be overridden by extending from WebSecurityConfigurerAdapter**. Extending WebSecurityConfigurerAdapter enables user authentication prior to a URL access

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {}

1. **A single namespace element can add multiple beans and processing steps to application context, such as starting embedded LDAP server**

<security:ldap-server />

1. **Security namespace can be included in application context by putting spring-security-config JAR in classpath** and adding schema declarations for spring security to XML file

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/security

http://www.springframework.org/schema/security/spring-security-3.0.3.xsd">

1. Spring security provides **support for Around advise** for method invocations through Spring AOP and web requests through Filter
2. **Spring security exceptions** are detected by ExceptionTranslationFilter, that either returns error code 403 or launches AuthenticationEntryPoint for user credential request
3. Each **secure object has an interceptor class** that is sub-classed from AbstractSecurtiyInterceptor, which provides a workflow for handling requests. This involves looking up configuration attributes for request
4. Spring security **delegates authentication request** to **a list of configured authentication providers**, each queried to see if it if it can perform the authentication. Providers may depend on a specific authentication mechanism i.e. JASIG CAS that uses a service ticket instead of conventional user name /password and would be authenticated by CasAuthenticationProvider. ProviderNotFoundException exception is thrown if suitable provider is not found on attempt to authenticate

<bean id="authenticationManager"

class="org.springframework.security.authentication.ProviderManager">

<property name="providers">

<list>

<ref local="daoAuthenticationProvider"/>

<ref local="anonymousAuthenticationProvider"/>

<ref local="ldapAuthenticationProvider"/>

</list>

</property>

</bean>

1. **DaoAuthenticationProvider uses UserDetailsService (as a DAO) in order to lookup the username, password and GrantedAuthoritys and authenticates by comparing password submitted in a UsernamePasswordAuthenticationToken with one loaded by the UserDetailsService. It can optionally use PasswordEncoder & SaltSource**

<bean id="daoAuthenticationProvider"

class="org.springframework.security.authentication.dao.DaoAuthenticationProvider">

<property name="userDetailsService" ref="inMemoryDaoImpl"/>

<property name="saltSource" ref="saltSource"/>

<property name="passwordEncoder" ref="passwordEncoder"/>

</bean>

1. **Most authentication providers use a UserDetailsService, even if the username and password are not part of the authentication decision** **They use returned UserDetails object to get GrantedAuthority information while actual authentication is done by a third party system i.e. LDAP, X509, CAS etc**
2. **Spring security includes JdbcDaoImpl class that implements UserDetailsService to obtain user authentication information from a JDBC data source, avoiding full featured ORM to store user details**

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="org.hsqldb.jdbcDriver"/>

<property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>

<property name="username" value="sa"/>

<property name="password" value=""/>

</bean>

<bean id="userDetailsService" class**="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl"**>

<property name="dataSource" ref="dataSource"/>

</bean>

1. **Spring security maintains a chain of filters internally where each filter has a particular responsibility and filters can be added or removed from configuration depending on services required. Spring's DelegatingFilterProxy delegates Filter's methods to a filter bean which is obtained from the Spring application context. The bean must implement javax.servlet.Filter**

<filter>

<filter-name>myFilter</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

</filter>

<filter-mapping>

<filter-name>myFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

1. **Spring Security's web infrastructure provides FilterChainProxy** that **deals with application context file for managing our web security beans. Filter chain is declared in the application context with bean name FilterChainProxy.**

<bean id="filterChainProxy" class="org.springframework.security.web.FilterChainProxy">

<sec:filter-chain-map path-type="ant">

<sec:filter-chain pattern="/webServices/\*\*" filters="

securityContextPersistenceFilterWithASCFalse,

basicAuthenticationFilter,

exceptionTranslationFilter,

filterSecurityInterceptor" />

<sec:filter-chain pattern="/\*\*" filters="

securityContextPersistenceFilterWithASCTrue,

formLoginFilter,

exceptionTranslationFilter,

filterSecurityInterceptor" />

</sec:filter-chain-map>

</bean>

**Micro Services Patterns**

1. **Micro service involves developing a small functional feature focussed on a business requirement as a single service, each service having its own light weight communication mechanism deployed on either single or multiple servers. Services are independent in terms of development & deployment. They allow flexible way to integrate with CI tools i.e. Jenkins, Hudson, Bamboo etc.**
2. **Distributed system in microservice architecture requires DevOps skills and is complex to manage especially when no. of services increases. Micro services are preferred choice for applications requiring support on multiple devices. Large sites like Amazon, Twitter, Ebay, Netflix have evolved to use Micro service architecture**
3. **Micro service Architecture goal is to accelerate software development through enabling continuous delivery/deployment. The application should be decomposed in a way so that most new and changed requirements only affect a single service. It uses Common Closure Principle (CCP), which states that classes changing for the same reason should be in the same package**
4. **Decomposition on business capabilities is based on things that business does to generate value i.e. Order Management services, Customer** **Management services etc. Development teams are organized around delivering business value rather than technical features. However Identifying business capabilities and hence services requires understanding of the business**
5. **Domain-Driven Design (DDD) refers to the application’s problem space as the domain, with each domain consisting of multiple subdomains corresponding to a different part of business. Sub domain could include Product catalog, Inventory management, Order management, Delivery management etc.**
6. **Single Responsibility Principle states each class handles only one responsibility. If there are 2 reasons to change for a class, its functionality should be split the in two classes**
7. **Services must use event driven architecture to maintain database consistency. Each service publishes an event whenever it updates its data.** **Other service subscribing to events update their data. As an example Order Service creates an Order in a pending state and publishes an OrderCreated event.** **Customer Service receives the event and attempts to reserve credit for that Order publishing either a Credit Reserved or a CreditLimitExceeded event.** **Order Service receives the event from the Customer Service and changes the state of the order to approved or cancelled**
8. **A Database per service architecture keeps persistent data private to that service and accessible only via its API. Options to keep service’s persistent data private in RDBMS could be using Private-tables-per-service, Schema-per-service or Database-server-per-service**
9. **Command Query Responsibility Segregation is used to implement queries that join data from multiple services. Application is split into command-side handling create, update, delete requests and emitting events when data changes. Query-side handles queries by executing them against one or more materialized views that are kept up to date by subscribing to events emitted when data changes**
10. **Event Sourcing: Applications persists events in an event store that behaves like a message broker enabling services to subscribe to events, and events are delivered to interested subscribers. Entities current state is reconstructed by replaying the events**
11. **MS Requires registration to locate or be available for other services. Registration options include self-registration that requires testing consideration & registry through 3rd party that requires exposing health check endpoint. Available service registry solutions are Consul, Eureka, Amalgam8**
12. **Actual MS invocation is either done from server proxy available as part of service registry or through client directly. While client side invocation is faster, it may bring in infrastructure logic within MS, making it harder to test. A sidecar on client side can alternatively handle load balancing & perform outbound MS calls similar to service proxy. Commonly used sidecar solutions are Netflix Prana, Amalgam8 & Kubernetes**
13. **An API Gateway is similar to server proxy, but simplifies MS access for external client**

**Basic Spring MVC Project in Eclipse**

1. **A Spring MVC project requires framework JARs that can be downloaded from spring repository** <http://repo.spring.io/release/org/springframework/spring/4.0.4.RELEASE/> **and then placed in class path (folder WEB-INF/lib for a dynamic web project) . Alternatively a Maven project automatically downloads MVC dependencies from maven repository.**
2. **Spring MVC framework uses both XML based controller mapping and annotation based mapping. Annotation based mapping is commonly used in Spring MVC web applications**
3. **Through @PathVariable annotation Spring MVC binds incoming value in HTTP request with a parameter variable of method to be invoked. @RequestParam annotation is used to bind query parameters in HTTP request to parameter variables of method to be invoked. When a POST request (Submit) is mapped from a web form to a controller class method, @RequestParam is used to bind values sent in HTTP request to parameters of method.**
4. **Use of @ModelAttribute annotation instead of @RequestParam reduces the code by eliminating need to explicitly bind each property to ModelAndView object inside controller method. This is suitable approach in case of User registration where many attributes of a user are required to be entered in an HTML form and then processed. @ModelAttribute annotation can also be applied to a method in which case all HTTP responses will automatically add code corresponding to this method**

@ModelAttribute

**public** **void** addCommonObjects(Model model) {

model.addAttribute("hdrMsg", "DCE, India");

}

1. **Data Binding related errors are caught by adding a variable of type BindingResult in method taking @ModelAttribute as parameter. A <form:errors> tag is added to JSP form, such as when binding error occurs, complete description is displayed on form**
2. **Spring MVC allows customization of data binding i.e. excluding a field on form. This can be done in a controller method annotated @InitBinder. Spring MVC internally use property editors to perform data binding tasks. These include classes for defining formatting for accepted Inputs**

***@InitBinder***

***public void initBinder(WebDataBinder binder)***

***{***

***SimpleDateFormat dateFormat = new SimpleDateFormat("dd/mm/yyyy");***

***binder.registerCustomEditor(Date.class, "DOB", new CustomDateEditor(dateFormat, false));***

***binder.registerCustomEditor(String.class, "studentName", new StudentNameEditor());***

***}***

1. **Spring MVC allows definition of custom property editors to handle custom requirements for a field i.e. having a landline number prepended with area code. A custom property editor class should extend PropertyEditorSupport class. Custom property editor should be registered inside @InitBinder annotated method.**
2. **Form validation can be achieved by putting annotations on class fields linked to form i.e. @Size(min=2, max=20) etc. A @Valid annotation should be placed alongside @ModelAttribute to enable form validation annotations. This also requires adding third party JAR files i.e. hibernate to classpath (WEB-INF/lib folder). Error messages can be customized by configuring spring MessageSource allowing messages to be read from a properties file**
3. **Spring MVC provides an option of adding Interceptors such that any HTTP request first goes though the interceptor class and then to other controllers. Interceptor class can be used for displaying “Site under maintenance” message while some trouble shooting work is being carried on. Interceptor class extends HandleInterceptorAdapter class of framework and is enabled by configuring in spring MVC XML file. Interceptor class can also be configured for a specified URL rather than globally across URLs. Interceptors are also used in Internationalization. This is done through registering localeChangeInterceptor along with localeResolver bean of type cookieLocaleResolver. ThemeChangeInterceptor enables change of theme of the page i.e. color, font etc. along with cookieThemeResolver.**