

The SOA Magazine

Feature Article



SOA with Spring (Part 2)

by Rizwan Ahmed

Published: March 9, 2010 (SOA Magazine Issue XXXVII: March 2010)

[Download PDF](#)

[Digg This](#) • [De.licio.us](#) • [Slashdot](#) • [Technorati](#) • [StumbleUpon](#) • [Google Bookmark](#)

Abstract: In Part 1 of this article series [REF-1], I had used the popular Spring framework [REF-2], a lightweight container that provides automated configuration and wiring of application objects, to create a contract-first Web service. Demonstrating Spring's pluggable architecture, we had seen how to configure the endpoint with Spring's support for OXM (Object-XML Mapping) and handle service-level and runtime exceptions. In this article we'll go over the security configuration using Spring's support for WS-Security providing message-level authentication, and optionally message confidentiality and message integrity services, ORM (Object-Relational Mapping) to deal with persistence at the object level, and DAO for data access to a relational database storing user-credential information. Next, we'll look at how Spring automatically generates the WSDL document using the data contract created earlier and, lastly, the client configuration required to invoke and consume the Web service.

Steps for Creating a Spring-WS Contract-First Service (continued from Part 1)

Referring back to Part 1 of the article series [REF-1], I had mentioned that the application context XML file is a composition of application beans (wired and managed by the Spring container) relevant to the particular Spring module being used. Example 1 is a snippet of the application's Web service (WS) context file described earlier containing the endpoint mappings which was configured to define an annotated Payload endpoint (PayloadRootAnnotationMethodEndpointMapping) that autodetects endpoint classes and methods using annotations (@Endpoint and @PayloadRoot) which it then uses to appropriately route the SOAP message to. Since these bean wirings are relevant only to the Web service's endpoint configuration, message routing and exception handling (all within the scope of the Spring-WS module [REF-3]), they are stored within a configuration file aptly called applicationContext-ws.xml. In this article we secure the service by authenticating the incoming Web service request containing a username token to user-credential information stored within a database and authorize access to the secured endpoint only to a particular role that the user must belong to. Since you would typically have a separate application context file for each Spring module that you would need to adapt, we would need to create separate, appropriately named, XML context files containing wirings of application objects for the Security, ORM and DAO module. The individual file classpaths are listed within the application's web.xml bootstrapped to the org.springframework.web.context.ContextLoaderListener. At runtime, Spring assembles a "virtual" container which essentially contains and manages the lifecycle and configuration of all its application context beans.

```
<!-- ===== ENDPOINT MAPPINGS ===== -->

<bean id="annotationMapping"
class="org.springframework.ws.server.endpoint.
mapping.PayloadRootAnnotationMethodEndpointMapping">
<description>
  Detects @PayloadRoot annotations on @Endpoint
  bean methods. The FormClientDemographicsEndpoint
  has such annotations. It uses two interceptors:
```

```

    one that logs the message payload, and the other validates
    it according to the messages.xsd' schema file.
</description>
<property name="interceptors">
  <list>
    <bean class="org.springframework.ws.server.
    endpoint.interceptor.PayloadLoggingInterceptor"/>
    <bean
      class="org.springframework.ws.soap.server.endpoint.
      interceptor.SoapEnvelopeLoggingInterceptor"/>
    <bean class="org.
    springframework.ws.soap.server.endpoint.interceptor.
    PayloadValidatingInterceptor">
      <property name="xsdSchemaCollection"
      ref="schemaCollection"/>
      <property name="validateRequest" value="true"/>
      <property name="validateResponse" value="true"/>
    </bean>
    <ref bean="wsSecurityInterceptor"/>
  </list>
</property>
</bean>

```

Example 1: Snippet of the application's WS context configuration file (applicationContext-ws.xml)

Enforcing Security

We see in Example 1 that the endpoint mapping bean "annotationMapping" is wrapping several interceptor implementations, which, as the name suggests, intercepts the request and performs some action prior to invoking the endpoint. The "wsSecurityInterceptor" is one such interceptor implementing the Spring-WS EndpointInterceptor interface, and provides message-level security on the SOAP message en-route to the endpoint. The specific security considerations being: is the SOAP message from a valid user and is the user authorized to access the Web service endpoint. Optionally, we could also have decided to digitally sign the message and/or encrypt the entire XML message payload (security is best addressed taking a holistic perspective that encompasses the entire development process [REF-4], however, standalone message-level security will suffice for the scope of what I'm trying to address here). The Spring community has a side project, Spring Security [REF-5], that provides a flexible and easily configurable framework (akin to Spring) that we can adapt for our authentication and authorization requirements.

However, in this paper we will look at a Spring-WS provided implementation of WS-Security. WS-Security is an OASIS specification [REF-7] that describes an abstract message security model to protect the message content from being disclosed (confidentiality) or modified without detection (integrity) and to enable authentication of a user name token embedded in the SOAP message. Message confidentiality and integrity, beyond the scope of what I wish to address in this paper, is typically ensured in WS-Security through mechanisms such as XML Encryption and XML Digital Signature, respectively, in conjunction with security keys and the appropriate algorithm (such as HMAC-SHA1).

Example 2 shows the application's security context configuration file. The bean id "wsSecurityInterceptor" is mapped to the Spring-WS class XwsSecurityInterceptor which is a WS-Security endpoint interceptor based on the XML and Web Services Security package (XWS-Security) [REF-8]. XWS-Security is an implementation of the aforementioned OASIS WS-Security specification and provides the mechanisms to ensure quality levels of Web service endpoint protection through message integrity, message confidentiality and message authentication. A critical component of setting up the application for XWS-Security is to setup the appropriate database infrastructure for the type of security (XML Digital Signature, XML Encryption and/or Username Token verification) to be used in conjunction with a combination of keystore files, truststore files and a username-password table. In order to keep things simple I'll use only Username token verification, which essentially specifies a process for sending a username token embedded (and optionally encrypted) within the message, which the configured infrastructure, as in our case, will then verify against user credentials stored in a HSQL datastore's username-password table to pass authentication.

The security interceptor is configured with the "policyConfiguration" and "callbackHandler" properties (refer to Example 2). The "policyConfiguration" is used to validate incoming messages according to the policy defined in "securityPolicy.xml" (refer to Example 3). The "callbackHandler" is used to validate user credentials

(supplied as a password digest specifically requested for in the security policy) and optionally retrieve certificates and private keys. Security policy configuration files are written in XML with elements within the file which specify the security mechanism(s) to use for an application and are enclosed within `<xwss:SecurityConfiguration>` tags. Within these declaration elements are elements that specify which type of security mechanism is to be applied to the SOAP message. For example, to apply XML Digital Signature, the security configuration file would include an element, along with a keystore alias that identifies the private key/certificate associated with the sender's signature. Since we are only going to use a simple username-password authentication security mechanism, therefore, our security policy as shown in Example 3, defines that all incoming SOAP messages must have a username token with a password digest in it.

```
<description>
  This application context contains the WS-Security and Spring Security
  beans.
</description>
<security:global-method-security secured-annotations="enabled"/>
<bean id="securityService"
  class="scrs.forms.service.springws.security.FormsSecurityService">
  <description>
    A security service used to obtain Forms User information.
  </description>
  <constructor-arg ref="formsSecurityDao"/>
</bean>
<bean id="wsSecurityInterceptor" class="org.springframework.ws.
soap.security.xwss.XwsSecurityInterceptor">
  <description>
    This interceptor validates incoming messages according
    to the policy defined in 'securityPolicy.xml'.
    The policy defines that all incoming requests must
    have a UsernameToken with a password digest in it.
    The actual authentication is performed by
    the Spring Security callback handler.
  </description>
  <property name="secureResponse" value="false"/>
  <property name="policyConfiguration"
    value="classpath:scrs/forms/service/springws/
    security/securityPolicy.xml"/>
  <property name="callbackHandler">
    <bean class="org.springframework.ws.soap.security.xwss.
    callback.SpringDigestPasswordValidationCallbackHandler">
      <property name="userDetailsService" ref="securityService"/>
    </bean>
  </property>
</bean>
```

Example 2: Snippet of the application's security context configuration file (applicationContext-security.xml)

```
<xwss:SecurityConfiguration dumpMessages="false" xmlns:xwss=
"http://java.sun.com/xml/ns/xwss/config">
  <xwss:RequireUsernameToken
    <passwordDigestRequired="true" nonceRequired="true"/>
</xwss:SecurityConfiguration>
```

Example 3: The XWS-Security configuration file

The actual authentication is performed by the Spring-WS callback handler `SpringDigestPasswordValidationCallbackHandler` (Example 2) that validates the password digest using the Spring Security provided `UserDetailsService` interface. This callback handler is wired with a reference to the "securityService" bean which in turn is constructor injected with the DAO class that obtains user credentials information stored in the database. The "securityService" bean is mapped to a `FormsSecurityService` class (see Example 4) that takes in the DAO (`FormsSecurityDao`) constructor and is used to load an `ScrsFormsUserDetails` object containing role(s) and authorization information for the user (refer to Example 5). The `SpringDigestPasswordValidationCallbackHandler` then compares the digest of the password contained in this details object (Example 5) with the digest in the message to pass authentication. Spring Security provides a `@Secured` annotation that can then be used to authorize access to the service implementation's business method only to a pre-defined role that the currently authenticated user must belong to (refer to Example 5 in [REF-1]).

Spring Security comes with authentication providers for every occasion depending on your preference and infrastructure needs: authentication using container adapters (eg. JBOSS, Tomcat, Jetty etc.), authentication against JA-SIG Central Authentication Service (CAS) for single sign-on, authentication against LDAP repository, authentication using JAAS and authentication against database using built-in DAO implementations such as in-memory and JDBC should you choose not to implement your own DAO for persisting security information.

```
public class FormsSecurityService implements UserDetailsService {
    private FormsSecurityDao formsSecurityDao;
    public FormsSecurityService(FormsSecurityDao formsSecurityDao) {
        this.formsSecurityDao = formsSecurityDao;
    }
    @Transactional
    public ScrsFormsUserDetails loadUserByUsername
    (String username) throws UsernameNotFoundException,
    DataAccessException {
        ScrsFormsUser scrsFormsUser = formsSecurityDao.get(username);
        if (scrsFormsUser != null) {
            return new ScrsFormsUserDetails(scrsFormsUser);
        }
        else {
            throw new UsernameNotFoundException
            ("Forms User '" + username + "' not found");
        }
    }
}
```

Example 4: The "securityService" bean that is constructor injected with a reference to the DAO performing actual user-credentials lookup

```
public class ScrsFormsUserDetails implements UserDetails {
    private ScrsFormsUser scrsFormsUser;
    public static final GrantedAuthority[] GRANTED_AUTHORITIES =
        new GrantedAuthority[]{new GrantedAuthorityImpl("ROLE_SCRS_FORMS")};
    public ScrsFormsUserDetails(ScrsFormsUser scrsFormsUser) {
        this.scrsFormsUser = scrsFormsUser;
    }
    public GrantedAuthority[] getAuthorities() {
        return GRANTED_AUTHORITIES;
    }
    // Getters and setters
}
```

Example 5: The UserDetails information includes the role and authorization information for the user

The last part in the equation for obtaining user-credential information is the persistence model for the security related data. Spring integrates with a variety of data access technologies: whether it's direct JDBC, iBATIS or an Object Relational Mapping (ORM) framework such as Hibernate [REF-9] or the Java Persistence API (JPA) [REF-10]. I'll follow the architecturally best practice of accessing the persistence context via DAO (Data Access Object) which exists to provide a means to read (and write) data to the database. Implementing the DAO using direct JDBC (wired with an instance of Spring's JdbcTemplate) would very well have served our purpose. I'll, however, throw in an ORM abstraction layer upon the DAO built around the Java Persistence API (JPA). Through ORM, the intricate details of data access, such as, SQL statements, database connections and result sets are obfuscated and we can deal with data persistence at the object (entity) level.

JPA [REF-10] is a POJO based persistence mechanism, which replaces the earlier unwieldy entity EJBs, drawing ideas from both Hibernate and Java Data Objects (JDOs) with a mix of Java 5 annotations thrown in for good measure. Spring JPA based applications deployed within the context of a JEE container (such as Websphere or JBoss) have their entity objects managed by entity managers typically created from a Spring pre-configured "entityManagerFactory" bean (mapped in our case to an instance of LocalContainerEntityManagerFactoryBean as shown in Example 6). The LocalContainerEntityManagerFactoryBean is a factory object, intended for use in a JEE application server environment, that creates a shared JPA EntityManagerFactory in the Spring application context from which a transactional EntityManager is then derived to be injected into the DAO that performs the database access.

Example 7: The persistence.xml enumerates one or more persistent (or domain) classes

```

@Entity
@Table(name = "SCRS_FORMS_USER")
public class ScrsFormsUser {
    @Column(name = "USERNAME")
    private String username;
    @Column(name = "PASSWORD")
    private String password;
    public ScrsFormsUser(String username, String password) {
        this.username = username;
        this.password = password;
    }
    // getters and setters }

```

Example 8: The domain object representing an ScrsFormsUser

The EntityManagerFactory (wired in the Spring application context as shown earlier) can then be passed to JPA based DAOs via dependency injection. One way of doing this is by using annotations to inject a transactional EntityManager into the DAO class. Therefore, the application DAOs do not interact with the entity manager factory at all and instead obtain the entity managers directly through runtime injection via the @PersistenceContext annotation. It's important to note that Spring can perform the injection only if a PersistenceAnnotationBeanPostProcessor is enabled (refer to Example 6). The JpaFormsSecurityDao class, as shown in Example 9, is decorated with the @PersistenceContext annotation that expresses a runtime dependency on the entity manager which is configured within the "entityManagerFactory" bean wired along with a JNDI obtained "dataSource" bean.

```

@Repository
public class JpaFormsSecurityDao implements FormsSecurityDao {
    @PersistenceContext
    private EntityManager entityManager;
    public ScrsFormsUser get(String username) throws DataAccessException {
        Query query = entityManager.createQuery
        ("SELECT f FROM ScrsFormsUser f WHERE f.username = :username");
        query.setParameter("username", username);
        try {
            return (ScrsFormsUser) query.getSingleResult();
        }
        catch (NoResultException e) {
            return null;
        }
    }
}

```

Example 9: The JPA Data Access Object

A high level diagram detailing the sequence of application level events and processes for the security infrastructure is shown in Figure 1.

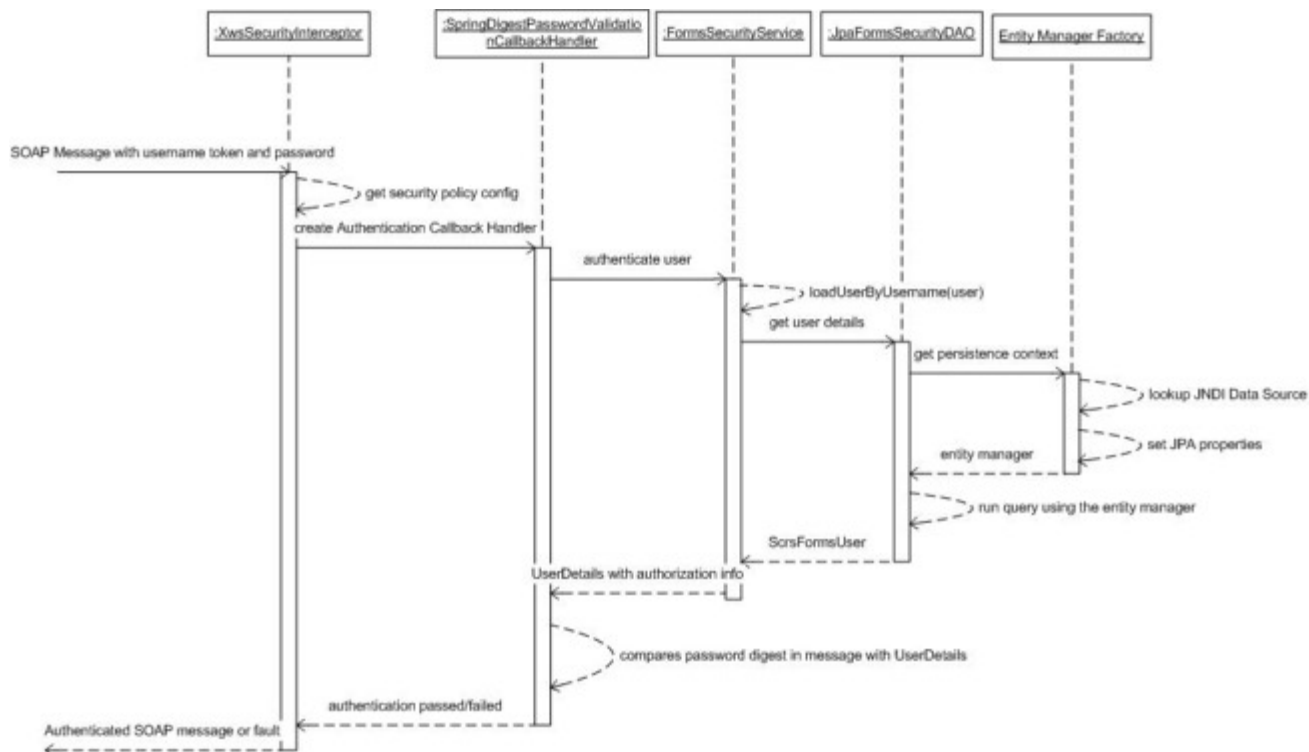


Figure 1: Sequence diagram of the security infrastructure. (Click [here](#) to view a larger version of this image.)

Creating the WSDL Contract

Throughout the article we have followed a contract-first methodology for creating services, supported by Spring-WS. We had created the data service contract in an XML schema file prior to configuring the endpoint and providing an implementation. As you might recall, I had mentioned that only the data contract has to be created manually, the operational contract and the WSDL document is generated automatically by the configured Spring-WS runtime infrastructure. We had seen earlier how the `org.springframework.ws.transport.http.MessageDispatcherServlet` serves as the router for SOAP messages to its endpoint based on the URL mapping defined in the application's `web.xml`. We create an additional servlet mapping for the URL pattern `*.wsdl` (refer to Example 10 snippet of the application's `web.xml`), which essentially results in the `MessageDispatcherServlet` receiving requests to serve up the WSDL file.

```
<servlet>
  <servlet-name>ws</servlet-name>
  <servlet-class>org.springframework.ws.
    transport.http.MessageDispatcherServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>ws</servlet-name>
  <url-pattern>/services</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>ws</servlet-name>
  <url-pattern>*.wsdl</url-pattern>
</servlet-mapping>
```

Example 10: The URL pattern `*.wsdl` is mapped to the `MessageDispatcherServlet`

We next configure a "scrsforms" bean mapped to Spring-WS's `org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition` class (refer to Example 11). The `DefaultWsdl11Definition` is a special bean that the `MessageDispatcherServlet` works with to generate WSDL documents from the XML schema defined as the data portion of the contract. It is wired with property elements that specify values as to the XML schema location, the Web service's preferred port type name, location URL and target namespace (again, refer to Example 11).

```

<bean id="scrsforms" class="org.springframework.
ws.wsdl.wsdl11.DefaultWsdl11Definition">
  <description>
    Builds a WSDL from the message.xsd.This bean
    definition represents the scrsforms.wsdl file found
    in the root of the web application.
  </description>
  <property name="schema" value="/messages.xsd"/>
  <property name="portTypeName" value="ScrsFormsSpringWS"/>
  <property name="locationUri"
value="http://<machine>:<port>/forms/services"/>
  <property name="targetNamespace"
value="http://scrs.forms.springws.service"/>
</bean>

```

Example 11: A DefaultWsdl11Definition bean configured in the WS application context (applicationContext-ws.xml)

When the MessageDispatcherServlet receives a request for /scrsforms.wsdl, it will look in the Spring context for a WSDL definition bean named "scrsforms". The DefaultWsdl11Definition class works by reading the XML schema definition wired within one of its property elements. It looks through the schema file for element definitions whose names end with Request and Response and assumes that those suffixes indicate a message that is to be sent to and received from a Web service operation and creates a corresponding <wsdl:operation> element in the WSDL it produces. In other words, when DefaultWsdl11Definition processes the messages.xsd file it assumes that the <GetClientDemographicsRequest> and <GetClientDemographicsResponse> elements are input and output messages for an operation called GetClientDemographics and creates the following definition in the generated WSDL file (refer to Example 12). Notice that the was wrapped by the named using the value wired into its "portTypeName" property. The location URI, which refers to the actual endpoint location of the service is defined picked up from the "locationUri" property and embedded within a element in the generated WSDL file.

```

<wsdl:portType name="ScrsFormsSpringWS">
  <wsdl:operation name="GetClientDemographics">
    <wsdl:input message="tns:GetClientDemographicsRequest"
name="GetClientDemographicsRequest">
      </wsdl:input>
    <wsdl:output message="tns:GetClientDemographicsResponse"
name="GetClientDemographicsResponse">
      </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>

```

Example 12: Snippet of the generated WSDL using properties wired into DefaultWsdl11Definition

Consuming Spring-Driven Web Services

Spring-WS's org.springframework.ws.client.core.WebServiceTemplate class [REF-12] provides a message-driven approach to sending and receiving XML messages from Web services and is the centerpiece of its client API. Sending messages to a Web service involves producing a SOAP envelope, body and communications boilerplate code that is pretty much the same for every Web service client. Whereas, configuring WebServiceTemplate in Spring is rather straightforward, Spring-WS makes things even easier with a convenient support class, the WebServiceGatewaySupport that automatically provides a WebServiceTemplate to client classes that subclass it. I've implemented a FormClientDemographicsServiceGateway client class that subclasses WebServiceGatewaySupport (refer to Example 13) and is constructor wired, via an abstract bean definition ("abstractClient") in the client application context (see Example 14), to the SaajSoapMessageFactory that essentially is tasked to construct the message according to the SOAP WS-I Basic Profile specification (we could very well have written an Axis [REF-13] or JAX-WS client to do the same, but since this article is about Spring I wanted to show the Spring way of doing things). At the same time, a "destinationProvider" property is configured with the destination URI for the service endpoint's WSDL file. WebServiceTemplate (inherited by the extending class FormClientDemographicsServiceGateway) provides a marshalSendAndReceive() method for sending and receiving XML messages that are marshaled to and from Java objects. For that to happen though, we need to wire up both the marshaller and unmarshaller properties in the client application context (configured with


```

</bean>
<bean id="securityInterceptor" class="org.springframework.ws.
soap.security.wss4j.Wss4jSecurityInterceptor">
  <property name="securementActions" value="UsernameToken" />
  <property name="securementUsername" value="JohnDoe" />
  <property name="securementPassword" value="changeit" />
</bean>

```

Example 14: The client application context

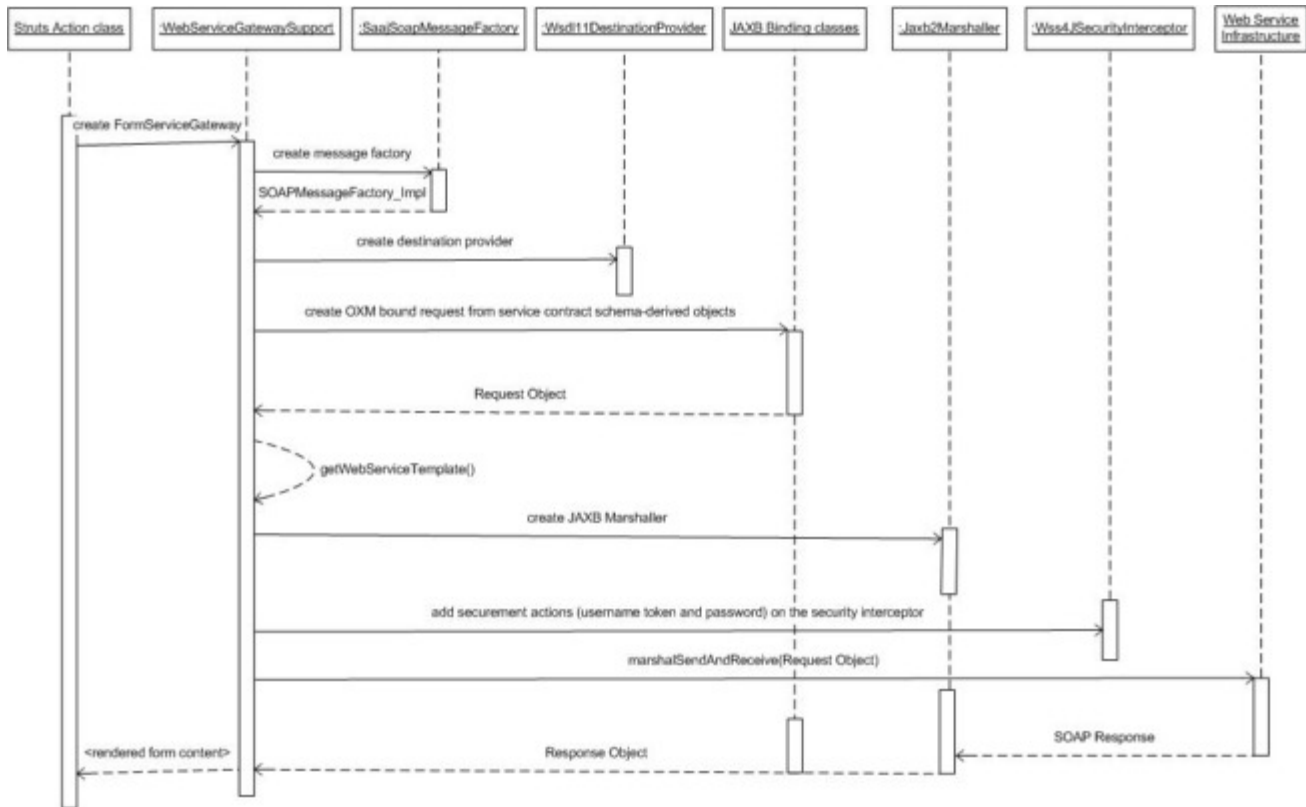


Figure 2: Sequence diagram of the client infrastructure. (Click [here](#) to view a larger version of this image.)

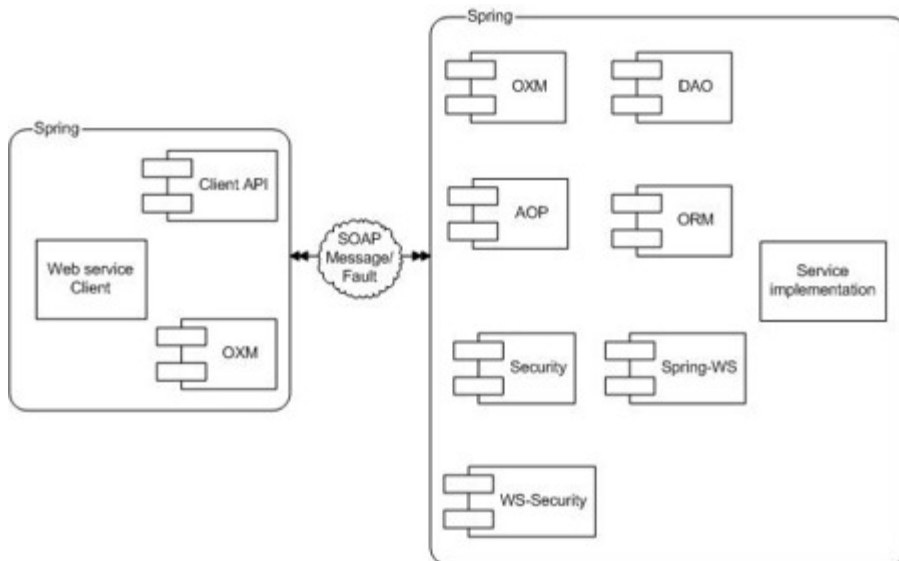


Figure 3: Component diagram showing the various Spring-managed modules for the client and service infrastructure

Conclusion

Spring is a leading open source, POJO framework providing a one stop shop for all your enterprise

application development needs. I've demonstrated how the Spring framework, via its modular and pluggable architecture (refer to Figure 3) that uses concepts such as IoC and DI, allows one to define, create, secure and consume contract-first Web services leading to an end-to-end service-oriented solution. While there is no governing standard behind Spring, as opposed to EJB3 which is created by experts from application server vendors with direct input over its governance, the Spring technology stack is pretty mature, easily configured and widely adopted. The flexibility of configuring the service assembly and its ease of use in test-driven development is, perhaps, one of its biggest selling features.

References

[REF-1] "SOA with Spring (Part 1)" by Rizwan Ahmed, SOA Magazine February 2010,

<http://www.soamag.com/I36/0210-2.php>

[REF-2] "About Spring", Spring Source, <http://www.springsource.org/about>

[REF-3] "Spring Web Services", Spring Source, <http://static.springsource.org/spring-ws/sites/1.5/>

[REF-4] "Case Study in Secure Software Development" by Rizwan Ahmed, Java Developer's Journal, December 2009, <http://java.sys-con.com/node/1227919>

[REF-5] "Spring Security", Spring Source, <http://static.springsource.org/spring-security/site/index.html>

[REF-6] "Web Services Security", IBM Developerworks SOA and Web services,

<http://www.ibm.com/developerworks/library/specification/ws-secure/>

[REF-7] "OASIS Web Services Security: SOAP Message Security 1.1", OASIS Standard Specification,

<http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>

[REF-8] "Introduction to XML and Web Services Security", The Java™ Web Services Tutorial,

<http://java.sun.com/webservices/docs/1.6/tutorial/doc/index.html>

[REF-9] "Hibernate: Relational Persistence for Java and .NET", Hibernate.org, <https://www.hibernate.org/>

[REF-10] "Java Persistence API", Oracle/Sun Developer Network,

<http://java.sun.com/javase/technologies/persistence.jsp>

[REF-11] "Object Relational Mapping (ORM) data access", Spring Source,

<http://static.springsource.org/spring/docs/2.0.x/reference/orm.html>

[REF-12] "Spring in Action" by Craig Walls, Manning Publications Co., <http://www.manning.com/walls3/>

[REF-13] "Web Services - Axis", Apache Web Services Project, <http://ws.apache.org/axis/>

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



[Home](#) [Past Issues](#) [Contributors](#) [What is SOA?](#) [SOA Glossary](#) [SOA School](#) [SOA Books](#) [About/RSS](#)

Copyright © 2006-2010
SOA Systems Inc.

[Legal](#)