

A large, thick red chevron pointing to the right, positioned behind the text "High performance. Delivered."

High performance. Delivered.

API Design Best Practices

Red Hat JBoss Fuse (Camel)
Red Hat JBoss Data Virtualization, and
Red Hat JBoss Mobile Application Platform (FeedHenry)

Table of Contents

Table of Contents	3
1. APIs & the Enterprise	4
1.1. APIs vs. More Traditional SOA Approaches	4
1.2. Challenges in developing and consuming APIs	5
1.3. How JBoss tools make developing APIs simple	5
1.3.1. Creating an API with JBoss Developer Studio (JBDS)	5
1.3.2. Creating a Virtual Database with JDBS	8
2. Security – Authentication, Authorization	11
3. API publishing – Versioning, provisioning	19
4. Middleware connectors and ESB integration	21
5. API for Independent Web Services	24
6. Mediation – Message Validation, Message Transformation	28
7. Productization –Third party API integration	32
7.1. A-MQ Connector	32
7.2. Salesforce	33
7.3. Twitter	37
8. Certification and testing	39
9. Data services – Direct data access, Meta data services	40
10. Business intelligence	49
11. Red Hat Mobile Application Platform (FeedHenry) Integration:	50
12. Appendix	54
13. References	55

1. APIs & the Enterprise

What are APIs (Application Programming Interfaces)? What is new about them? Aren't they just a newer version of SOA, which has been around for a long time? The concept of an API is not new; Linux, Windows, C, and Java all had APIs since their inception that were typically compiled into the user's application code and are executed in the user's computer.

API Category	Example	Timeline
Operating System	API for MS Windows API for Apple Mac OS X (Cocoa)	1985- 2001-
Programming Languages	Java API	1995-
Application Services	API for SAP (BAPI)	1990s-
Infrastructure Services	Amazon Web Services API	2002-
Web Services	Twitter API	2006-

What is new is the rise and dramatic proliferation of lightweight public APIs, and the rapid expansion of these into enterprise applications. Today, enterprises are able to quickly build and maintain complex applications that orchestrate across public and private APIs to provide business users with the unified capabilities that they demand. Enterprise customers, suppliers, and partners are also demanding access to enterprise APIs to leverage within their own applications. So as an architect, it is important to understand and be able to resolve the unique challenges inherent both in building composite applications and in deploying APIs for public consumption.

While it is true, that there are many similarities when designing applications to use lightweight APIs and using more traditional approaches, there are some unique differences. This white paper is intended help developers and application architects understand some of these differences and to help them to take advantage of the benefits of these technologies. The key objectives of this white paper are to:

- Explain what makes lightweight APIs different from more traditional approaches such as SOAP (Simple Object Access Protocol)
- Identify and provide solutions to some of the common design and development challenges when implementing APIs
- Demonstrate the use of JBoss tools and technologies to quickly build a working API demo that showcases these development approaches
- Help you understand when and where to use which approach

1.1. APIs vs. More Traditional SOA Approaches

How are APIs different from the more traditional SOA approaches such as SOAP? First, it is important to understand that SOAP is really an API. However, the use of SOAP is based on a design contract (i.e., a WSDL) and provided using a highly structured XML document. Other applications (either directly or through the ESB) consume services according to the contract. Although the concept of the WSDL was intended to provide an open API that could be leveraged by any consumer, in practice SOAP became used primarily in point-to-point integrations.

What really distinguished APIs (as we use the term today) from the more traditional SOA approach was the introduction of lightweight web APIs. These lightweight APIs operate over a network through the HTTP protocol, using REST-style communication to access a remote resource or service. The responses are typically expressed in JSON or XML (RSS, Atom). The advent of lightweight APIs solved some of the key challenges of the more traditional SOA approaches such as SOAP. These small APIs could be

developed very quickly, had a much smaller network footprint than SOAP, and could be consumed easily by everything from a full-fledged enterprise application to a web browser running javascript or by mobile applications. These lightweight APIs have exploded into the marketplace, and are driving what is now called the “Internet of Things”, where in the next few years virtually all new devices will be connected in some way.

1.2. Challenges in developing and consuming APIs

APIs are lightweight, easy to build and consume, so end of story, right? Yes, APIs can be easy to build and consume, but to do this well, one must consider many of the same architectural issues that are found in any enterprise development program: is my API usable, does it scale, is it maintainable, is it secure, etc. Addressing all of these concerns is the subject of a book, not a white paper such as this one! What we would like to demonstrate is how to solve some of the most common business and technical problems when developing with APIs. For example, developers often face issues such as:

- How to secure an API
- How to integrate with Salesforce.com
- How to leverage social media (such as Twitter) from my application
- How to integrate and display data from multiple data sources

For this white paper, we have chosen various problems that cover a variety of different development challenges, and will be leveraging Red Hat JBoss Fuse, Camel and Red Hat JBoss Data Virtualization to address these challenges. The details and how to access the code samples can be found in the appendix. We hope that the examples in the next sections will be useful in jumpstarting your API program!

1.3. How JBoss tools make developing APIs simple

JBoss tools such as JBoss Developer Studio provides superior support for your entire development lifecycle. They include a broad set of tooling capabilities and support for multiple programming models and frameworks, including Java™ Enterprise Edition 6, RichFaces, JavaServer Faces (JSF), Enterprise JavaBeans (EJB), Java Persistence API (JPA), and Hibernate®, JAX-RS with RESTEasy. They support Context Dependency Injection (CDI), HTML5, and many other popular technologies, and provide the developer choices in supporting multiple JVMs, productivity with Maven, and in testing with Arquillian. They are also fully tested and certified to ensure that the plug-ins, runtime components, and their dependencies are compatible with each other.

1.3.1. Creating an API with JBoss Developer Studio (JBDS)

JBoss Developer Studio includes both certified visual tooling and the production-ready, fully supported Red Hat JBoss Enterprise Application Platform, which is supported by Red Hat for as part of a subscription. With a simple point and click interface, JBoss Developer Studio makes creating APIs simple. For example, perhaps you have created an API and would now like to add a component that fires a query to a database. With JBDS, this is a quick 3-step process:

1. add a logging component and configure it to log as desired
2. add a dynamic router
3. configure the router to pass the request to the database handler

The details are shown in the below figures. In this case, we are starting with a simple route. In Figure 1, we have started with a simple route, and have just added the “log” element on the right. In the subsequent figures we add and configure the dynamic router to route the generated log messages to the database.

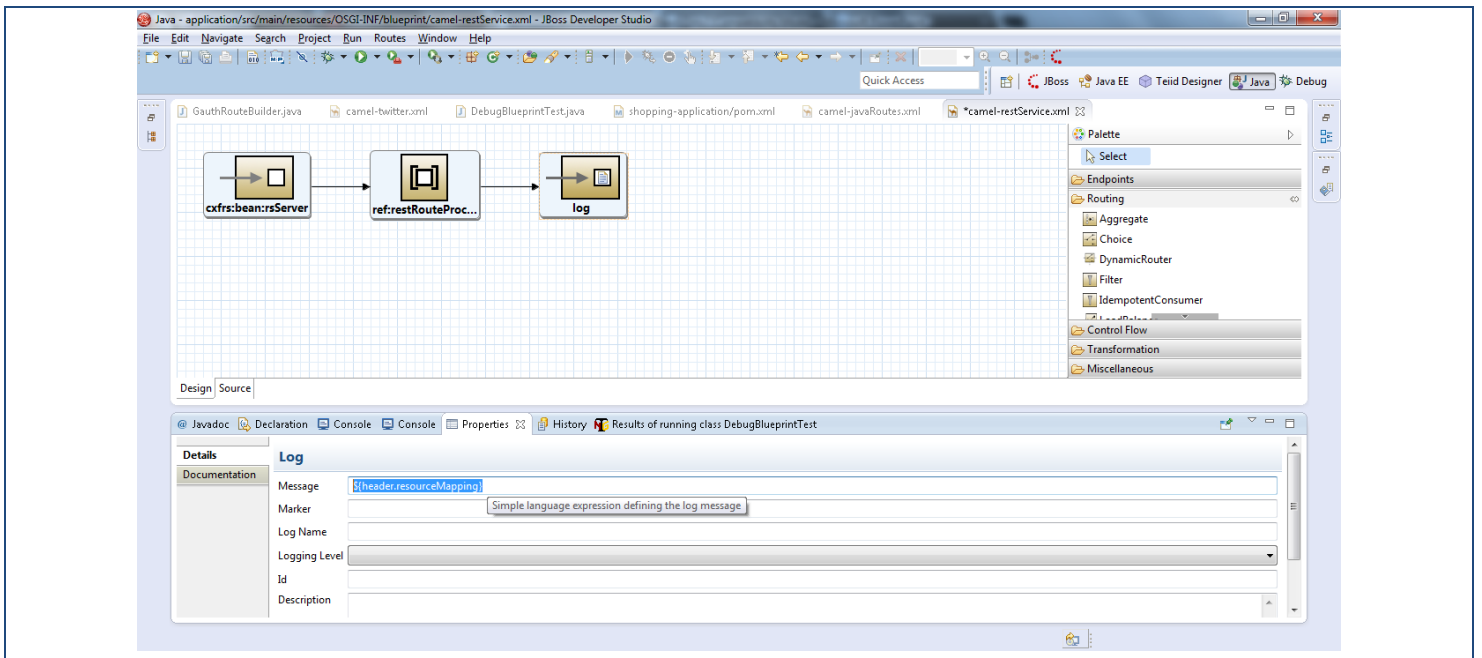


Figure 1. Add a logging component and configure it

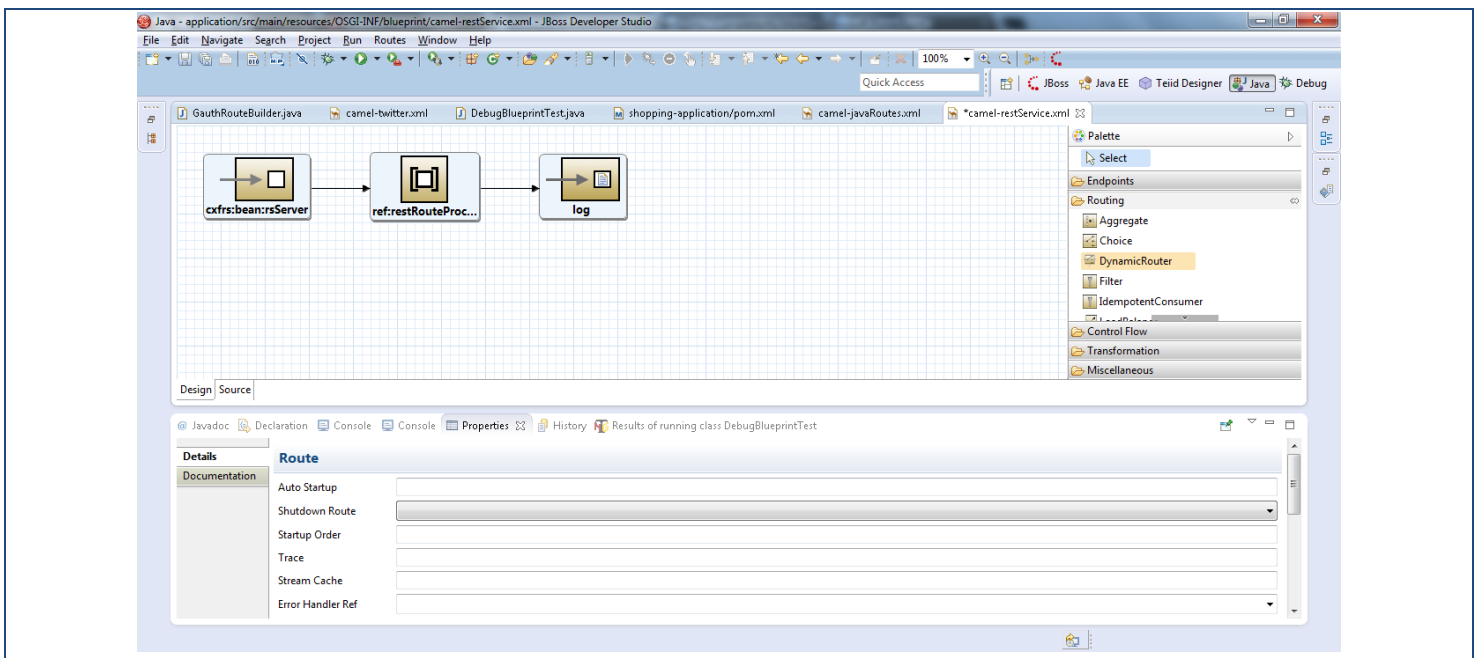


Figure 2. Add a dynamic router

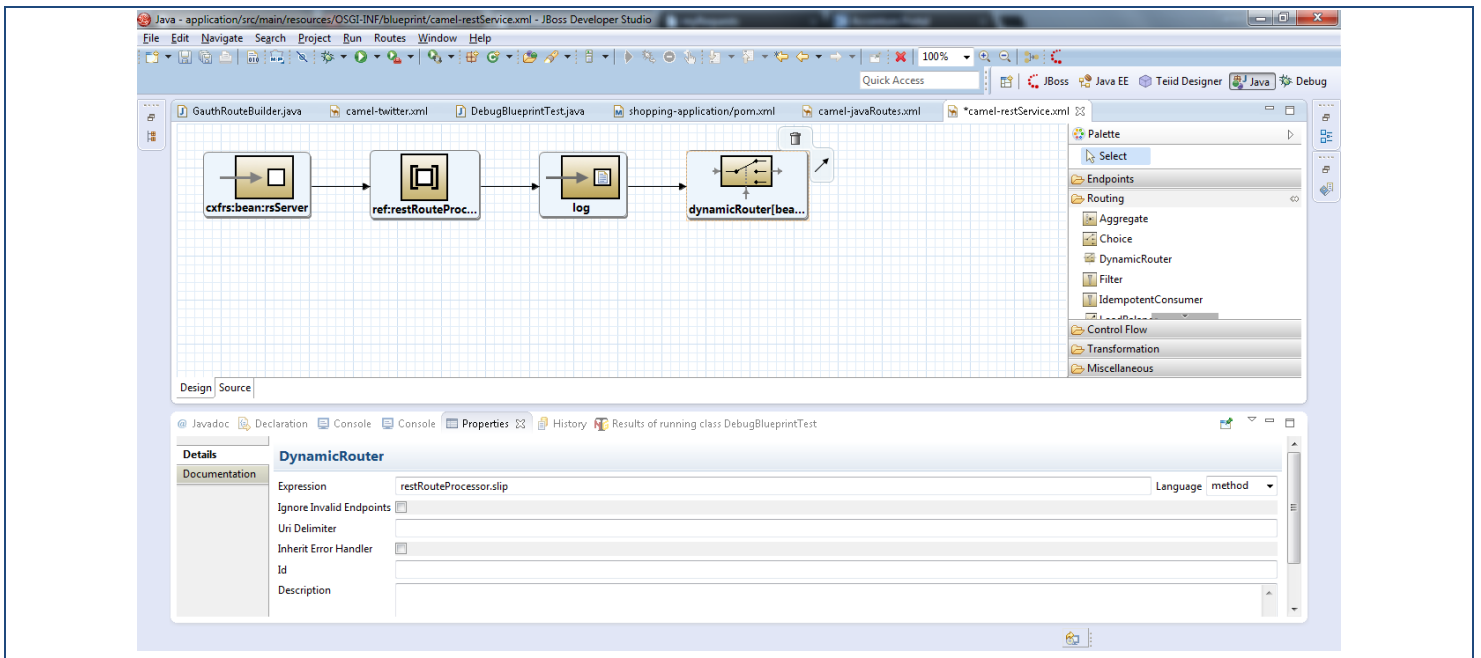


Figure 3. Configure the router

The configuration of the router maps the incoming request to the appropriate database handler service. This mapping is programmatically generated and is present in the restRouteProcessor.slip expression. To configure the database connection we need to add the JDV server as a data-source in our application. This configuration is added as a spring bean in the beans.xml

```
<!-- this is the JDBC data source Config for mysql database -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="${database.driverClassName}" />
    <property name="url" value="${database.url}" />
    <property name="username" value="${database.username}" />
    <property name="password" value="${database.password}" />
</bean>
```

Figure 4. Add the data source to Spring configuration

The default configuration of the data source properties are present under projects/shopping-demo-application/application-interface/src/main/resources/shoppingApplication.properties file.

```
# Database Connection Properties
database.driverClassName=org.teiid.jdbc.TeiidDriver
database.url=jdbc:teiid:shoppingApplicationVdb@mm://localhost:31000;version=1
database.username=user
database.password=user
```

Figure 5. Configure the database driver

To override these properties it is recommended to update the dv-support/application.properties file.

Once the database configuration we can now add a route to process the request forwarded by the Dynamic Router. Below is the route:

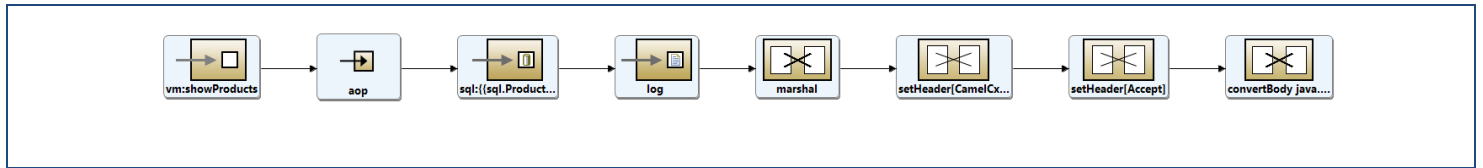


Figure 6. Add the new route

The sql query is fired by the sql component in the route. Camel has many built-in components to connect to different data sources. To use the sql component we just need to add the component in the application as a bean and link the component to the data source.

```
<!-- configure the Camel SQL component to use the JDBC data source -->
<bean id="sql" class="org.apache.camel.component.sql.SqlComponent">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Figure 7. Configure Camel to use the data source

1.3.2. Creating a Virtual Database with JDBS

Data virtualization allows an application to retrieve and manipulate data without requiring technical details for the underlying data sources. Creating a virtual database from multiple data sources is simple as well. Using the Teiid Designer within JDBS, this becomes a straightforward set of point and click steps. Teiid Designer is a visual tool that enables rapid, model-driven definition, integration, management and testing of data services without programming using the Teiid runtime framework. With Teiid Designer, not only do you create source data models and map your sources to target formats using a visual tool, but you can also:

- create a virtual database (or VDB) containing your models which you deploy to Teiid server and then access your data.
- resolve semantic differences
- create virtual data structures at a physical or logical level
- use declarative interfaces to integrate, aggregate, and transform the data on its way from source to a target format which is compatible and optimized for consumption by your applications

The set of figures below illustrate how to do this once the initial data sources are configured. For details on configuring the datasources and exposing this within your application, see **Section 9**.

With the underlying databases already configured, the first step in creating a virtual database is to go to the File Menu and create a new Teiid VDB.

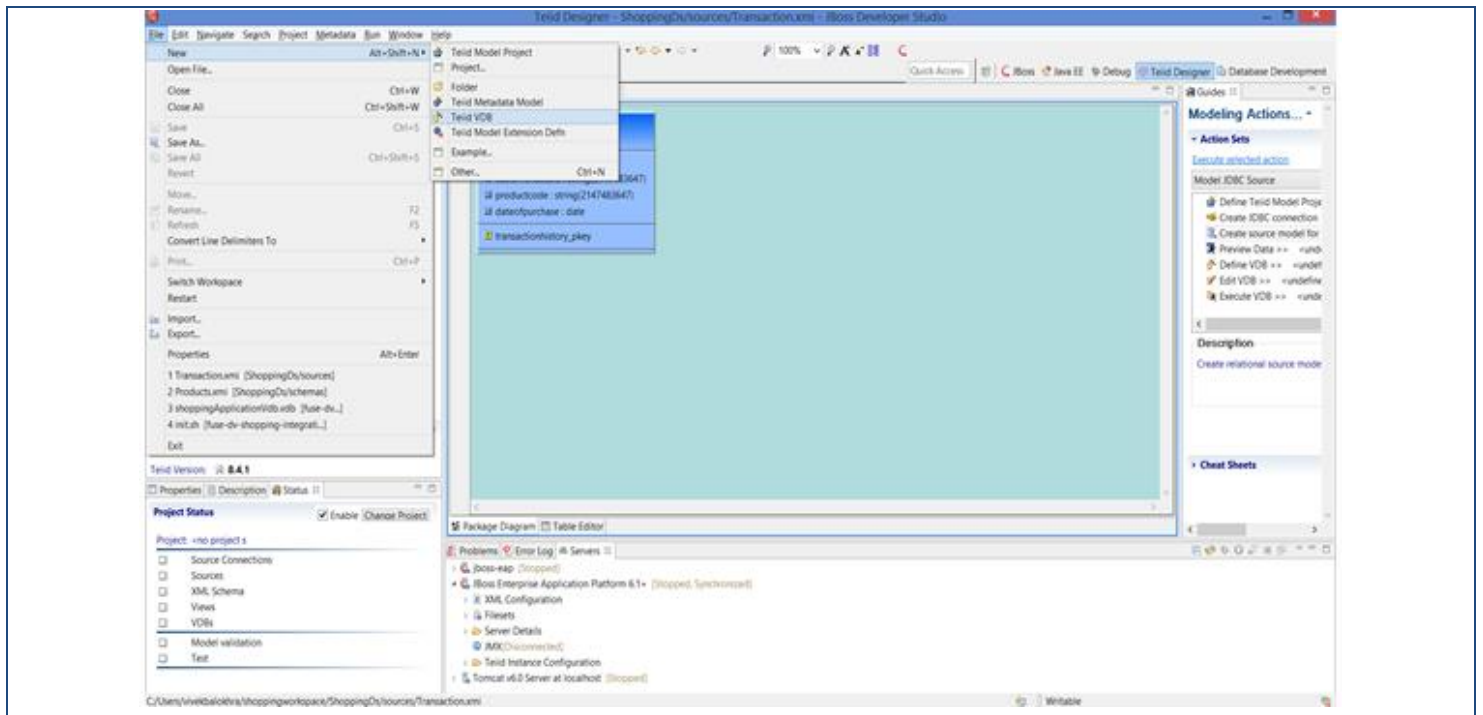


Figure 8. Creating a new Teiid VDB

The new VDB asks for models. We simply click on the Add button to add these to our VDB.

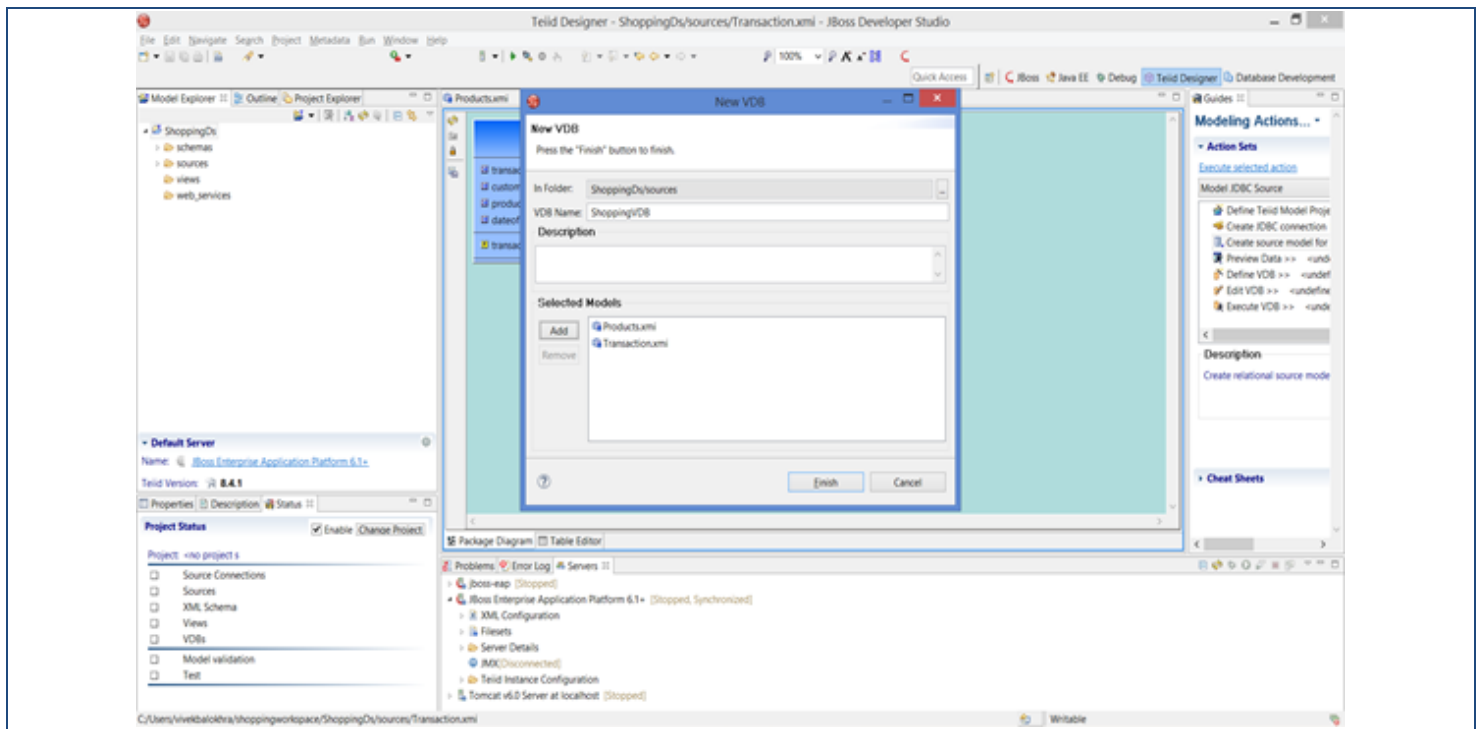


Figure 9. Adding the previously created models

The virtual database ShoppingVDB.vdb is now ready for use in your application. Further details are available in **Section 9**.

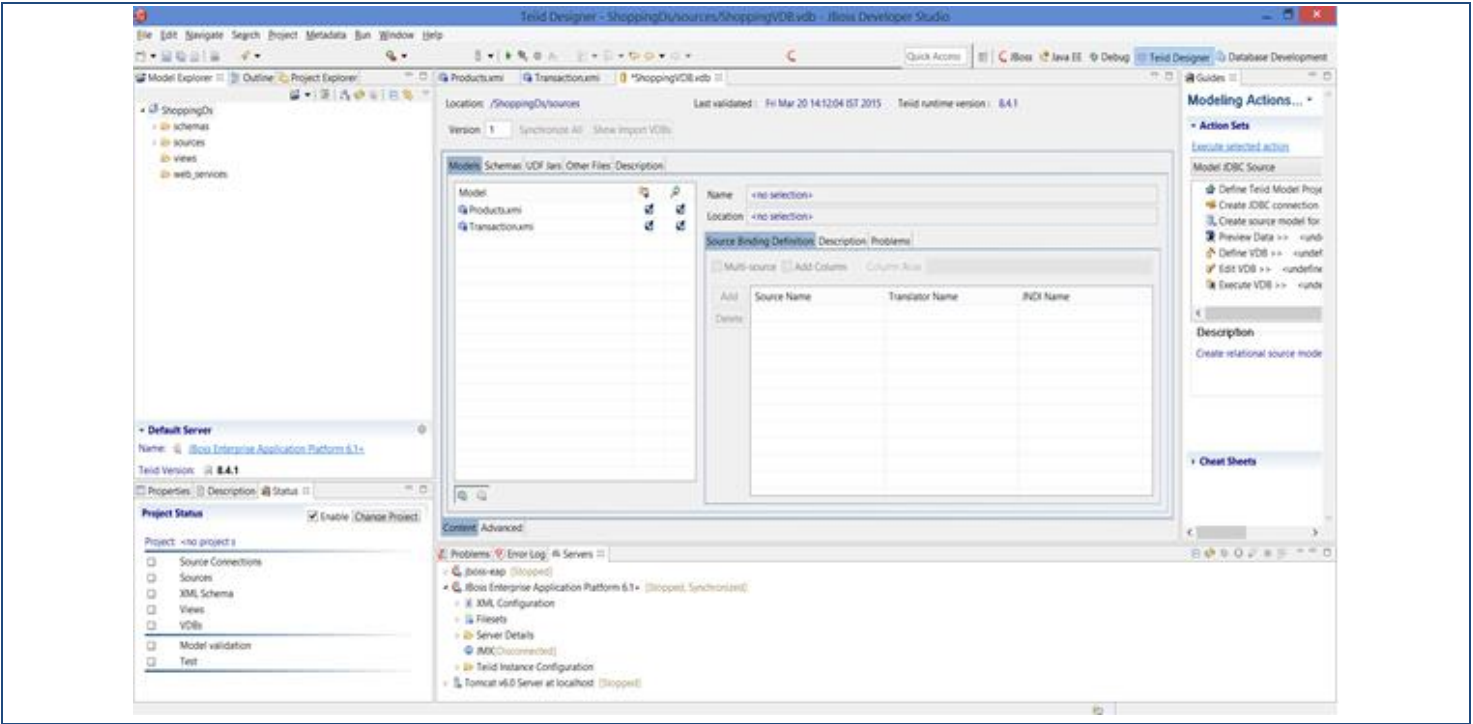


Figure 10. The newly created ShoppingVDB.vdb

2. Security – Authentication, Authorization

Authentication and Authorization are likely to be the most common use cases that an API developer will run across. Almost every application that is developed requires a user to prove who they are, that they should have access to the requested resource and what they are allowed to do.

Authentication is the process of ensuring that an entity in a system transaction (including users, servers, and clients) is who he, she, or it claims to be. Typically, authentication happens when a user logs into a system or application.

Authorization Services implement policies that restrict access to given resources (e.g.: only some users may be authorized to read the salary information of an employee). Authorization Services leverage Authentication Services to identify the user/application requiring the service. Generally, Authorization Services are "role-based", thus access rights to a service are assigned to roles (or groups) instead of users.

Typically, authentication and authorization are NOT provided by the API developer (or end-user applications for that matter). Authentication and authorization are provided by **endpoint security** components within the Camel framework. These endpoint security components secure the “endpoints” – i.e. specific URLs, as well as offer the ability to secure the API payloads.

Almost all enterprise applications leverage some form of central authentication or authorization. Increasingly, consumer facing web applications and APIs do the same, often leveraging common providers such as Facebook or Google to allow users to authenticate using a common set of credentials.

There are many camel components that support endpoint security. Some of the more commonly used include:

Camel Component	Use Case
JMS and ActiveMQ	SSL/TLS security and JAAS security for client-to-broker and broker-to-broker communication
Jetty	HTTP Basic Authentication and SSL/TLS security
CXF	SSL/TLS security and WS-Security
Crypto	creates and verifies digital signatures in order to guarantee message integrity
Netty	SSL/TLS security
MINA	SSL/TLS security
Cometd	SSL/TLS security
glogin and gauth	authentication in the context of Google applications

As you build and deploy enterprise APIs, you are likely to encounter many different standards for authentication and authorization. In reality – there is no “right” answer for which one to use, except to use the one that is standard within your enterprise.

Camel offers several forms & levels of security capabilities that can be utilized on camel routes. These various forms of security may be used in conjunction with each other or separately.

The broad categories offered are

1. **Route Security** - Authentication and Authorization services to proceed on a route or route segment
These capabilities are offered by the following components:
 - 1) [Shiro Security](#)
 - 2) [Spring Security](#)ⁱⁱ
2. **Payload Security** - Data Formats that offer encryption/decryption services at the payload level. These capabilities are offered by the following components:
 - 1) [XMLSecurity DataFormat](#)ⁱⁱⁱ (XML Encryption support)
 - 2) [XML Security component](#)^{iv} (XML Signature support)
 - 3) [Crypto DataFormat](#)^v (Encryption + PGP support)
 - 4) [Crypto component](#)^{vi} (Signature support)
3. **Endpoint Security** - Security offered by components that can be utilized by the endpoint URIs associated with the component. Some such components are:
 - 1) [Jetty](#)^{vii} - HTTP Basic Authentication support * SSL support
 - 2) [CXF](#)^{viii} - HTTP Basic Authentication & WS-Security support using the CXF Bus driven interceptor chain
 - 3) [Spring Web Services](#)^{ix} - HTTP Basic Authentication & WS-Security support
 - 4) [Netty](#)^x - SSL support
 - 5) [MINA](#)^{xi} - SSL support
 - 6) [Cometd](#)^{xii} - SSL support
 - 7) [JMS](#)^{xiii} - JAAS and SSL based security for client <--> broker communication
4. **Configuration Security** - Security offered by encrypting sensitive information from configuration files.
Camel offers the [Properties](#) component to externalize configuration values to properties files. Those values could contain sensitive information such as usernames and passwords. Those values can be encrypted and automatic decrypted by Camel

The role of fuse in the authentication process of the application is to integrate the authentication service providers (LDAP,database,o-auth) to our application. Validating the authenticated user can be done in the Web app layer using filters as well. However linking the authentication providers as a service in the integration layer is a better practice to add flexibility to the design.

Linking in such a way also makes the services readily available to front-end frameworks like Angular which can now directly link to the integration layer. However if the authentication process is coupled with the Web-app layer the requests have to necessarily pass through the Web-app layer.

In case a web app layer is mandatory in the application we can always expose the fuse security route as a service to the web-app layer.

For this white paper, demonstrating enterprise authentication would be difficult as there are a wide variety of authentication providers, and we can't really stand up all of them in a simple demo. So instead, we have focused on a more consumer-oriented solution – integrating with Google. We have chosen to use the gauth component in our shopping cart application. The user experience when using gauth is fairly straightforward as shown in the figures below.



Welcome to shopping application!

Choose an option

Show All Products

Buy Existing Products

About this application:

The shopping application is an application where a user can browse and buy products. The application also has features to add new products

 Follow @shoppingapp037

Figure 11. User browses to the application and clicks buy

No OAuth access token available. [Click here](#) to authorize this application to access your Google Contacts. This is necessary to buy products.

About this application:

The shopping application is an application where a user can browse and buy products. The application also has features to add new products

 Follow @shoppingapp037

Figure 12. Application displays a request to authenticate via Google

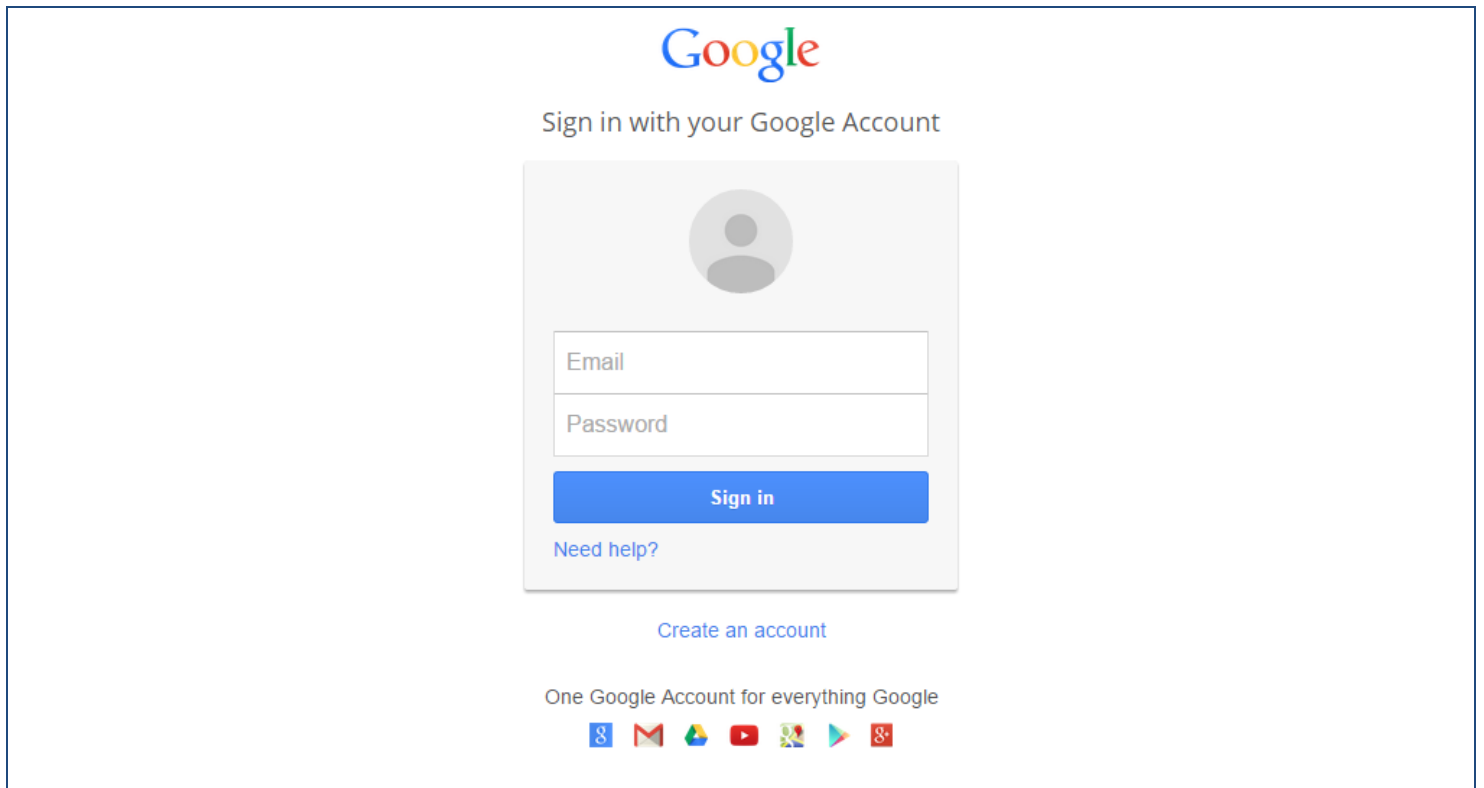


Figure 13. User is directed to the Google Authenticator to login

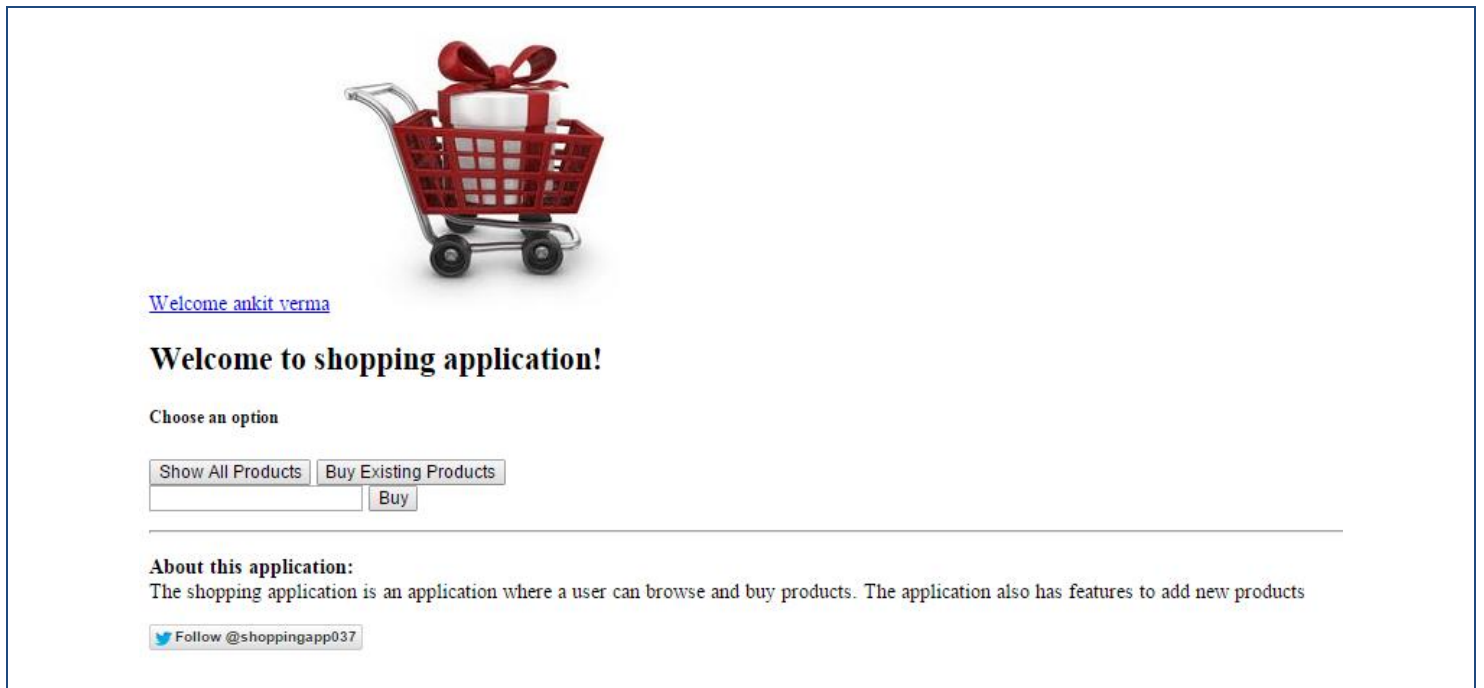


Figure 14. Once logged in, the user sees the full experience

Fuse makes simple to implement these steps. The approach we have taken is to create a Camel route named **GauthRouteBuilder** (see Figure 19. GauthRouteBuilder) In the GauthRouteBuilder:

1. On clicking the buy products a cookie is checked. If the cookie is not found the GET request to /shoppingApplication/application/authenticate will trigger the OAuth sequence of interactions.
2. The request is handled by a spring based controller which redirects the request to the CamelHttpTransportServlet. This servlet is responsible to forward the request to the GauthRouteBuilder.
3. The gauth: authorize endpoint obtains an unauthorized request token from Google and then redirects the user (browser) to a Google authorization page.
4. After the user logs in successfully to Google, the user is redirected back to /shoppingApplication/camel/handler (using the callback URL) along with an authorized request token.
5. The gauth: access endpoint exchanges the authorized request token against a long-lived access token.
6. The access token can be obtained from:


```
exchange.getOut().getHeader(GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN)
```
7. The access token secret can be obtained from


```
exchange.getOut().getHeader(GAuthUpgradeBinding.GAUTH_ACCESS_TOKEN_SECRET)
```
8. The token values are then saved as a cookie by TokenProcessor. It also forwards the request to /shoppingApplication/application/postAuthenticate
9. The post authentication request is handled by a spring controller which then stores the details of the user logged in as a session attribute. It uses the TokenLoginService class to fetch user details.

Understanding the gauth beans wiring:

Below are the beans we have created to use gauth functionality .

```
<bean id="gauthRouteBuilder" class="com.redhat.shopping.demo.application.services.GauthRouteBuilder"/>

<bean id="gauth" class="org.apache.camel.component.gae.auth.GAuthComponent">
  <property name="consumerKey" value="${consumer.key}"/>
  <property name="consumerSecret" value="${consumer.secret}"/>
</bean>
```

Figure 15. beans.xml

These beans are available at application-interface/./resources/camel/camel-AuthContext.xml. All the xmls within the resources/camel folder are loaded by the spring container. This configuration is provided in the application-interface/./resources/WEB-INF/web.xml file by adding the contextConfigLocation parameter to the application and spring context loader listener. Below is the configuration:

```
<web-app>

  <display-name>Application User Interface Module</display-name>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:camel/*.xml</param-value>
  </context-param>
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <servlet>
```

Figure 16. web.xml

The beans are set with values read from the resources/shoppingApplication.propertiesfile

```
consumer.key=anonymous
consumer.secret=anonymous
```

Figure 17. shoppingApplication.properties

Once the beans are created the gauth and camel-servlet components are loaded. Any camel route in our application can now use ghttp as an end point. Finally the gauthRouteBuider bean is added to a camel context where we are specifying it as a camel route. This configuration is done in camel-AuthContext.xml:

```
<camel:camelContext id="camelContext">
  <!-- JMX is not supported in GAE -->
  <camel:jmxAgent id="agent" disabled="true"/>
  <!-- refer to the route to use -->
  <camel:routeBuilder ref="gauthRouteBuilder"/>
</camel:camelContext>
```

Figure 18. camel-AuthContext.xml.xml (continued)

Understanding the Router:

```
public class GauthRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        getContext().setTracing(true);
        String encodedScope = URLEncoder.encode(
//            "https://accounts.google.com/o/oauth2/auth", "UTF-8");
            "https://www.googleapis.com/admin/directory/v1/users", "UTF-8");
        from("ghttp://authorize")
            .recipientList(simple("gauth:authorize?callback="+${header.callBackUrl}+"&scope="+encodedScope));
        from("ghttp://handler")
            .log("Inside Upgradation Process")
            .to("gauth:upgrade")
            .log("Upgraded tokens")
            .process(new TokenProcessor());
    }
}
```

Figure 19. GauthRouteBuilder

The router is invoked by the camel servlet and once the tokens are fetched they are saved as cookies by the TokenProcessor.


```

public class TokenProcessor implements Processor {

    private static final int ONE_HOUR = 3600;

    public void process(Exchange exchange) throws Exception {
        String accessToken = exchange.getIn().getHeader(GAUTH_ACCESS_TOKEN, String.class);
        String accessTokenSecret = exchange.getIn().getHeader(GAUTH_ACCESS_TOKEN_SECRET, String.class);

        if (accessToken != null) {
            HttpServletResponse servletResponse = exchange.getResponse().getHeader(
                Exchange.HTTP_SERVLET_RESPONSE, HttpServletResponse.class);

            Cookie accessTokenCookie = new Cookie("ACCESS-TOKEN", accessToken);
            Cookie accessTokenSecretCookie = new Cookie("ACCESS-TOKEN-SECRET", accessTokenSecret);

            accessTokenCookie.setPath("/shoppingApplication/");
            accessTokenCookie.setMaxAge(ONE_HOUR);

            accessTokenSecretCookie.setPath("/shoppingApplication/");
            accessTokenSecretCookie.setMaxAge(ONE_HOUR);

            servletResponse.addCookie(accessTokenCookie);
            servletResponse.addCookie(accessTokenSecretCookie);
        }

        exchange.getOut().setHeader(Exchange.HTTP_RESPONSE_CODE, 302);
        exchange.getOut().setHeader("Location", "/shoppingApplication/application/postAuthenticate");
    }
}

```

Figure 20. TokenProcessor

Once the cookies are created the request is redirected to /shoppingApplication/application/postAuthenticate. The ApplicationController handles the requests and uses the TokenLoginService to set user details in the current session

```

@RequestMapping(method = RequestMethod.GET, value={"postAuthenticate"})
public String handlePostAuthenticate(
    HttpServletRequest request,
    HttpServletResponse response,
    ModelMap model) throws Exception {
    User userDetails = null;
    try {
        userDetails = service.getContactNames(getAccessToken(request), getAccessTokenSecret(request));
    } catch (AuthenticationException e) {
        model.put("message", "OAuth access token invalid");
        return "/authorize.jsp";
    }
    request.getSession().setAttribute("userDetails", userDetails);
    return "/homePage.jsp";
}

```

Figure 21. ApplicationController

```

private Properties credentials;

/**
 * Sets properties that contains the application's consumer key and consumer secret.
 *
 * @param credentials consumer key and consumer secret.
 */
public void setCredentials(Properties credentials) {
    this.credentials = credentials;
}

public User getContactNames(String accessToken, String accessTokenSecret) throws Exception {
    ContactsService loginService = new ContactsService("shopping-application");
    OAuthParameters params = getOAuthParams(accessToken, accessTokenSecret);
    loginService.setOAuthCredentials(params, new OAuthHmacSha1Signer());
    URL feedUrl = new URL("https://www.google.com/m8/feeds/contacts/default/full");
    ContactFeed resultFeed = loginService.getFeed(feedUrl, ContactFeed.class);
    String userEmailId = resultFeed.getId();
    String userName = resultFeed.getAuthors().get(0).getName();
    List<String> contactsInfo = new ArrayList<String>();
    for (int i = 0; i < resultFeed.getEntries().size(); i++) {
        ContactEntry entry = resultFeed.getEntries().get(i);
        contactsInfo.add(entry.getTitle().getPlainText());
    }
    User user = new User();
    user.setContacts(contactsInfo);
    user.setEmailId(userEmailId);
    user.setUserName(userName);
    return user;
}

```

Figure 22. TokenLoginService

3. API publishing – Versioning, provisioning

Versioning of APIs is a critical concern that must be addressed when deploying your APIs. Once your API is deployed, you have minimal control over who consumes it. Changing the API contract can break all of the downstream applications that use it. Imagine if Twitter changed their API – how many web sites and mobile applications would break? And yet, applications must evolve over time. How do we meet the competing demands of updating and adapting to changing needs, while not breaking all of our application consumers? One way is through a consistent approach to version control of our APIs. Our recommendation is that all APIs should be versioned from the very start, providing clear direction to our consumers of how we plan to update our services.

“Name” versioning is the recommended approach to versioning of APIs. The version is represented in the URI itself. This represents the de facto standard approach. When providing versioning in the URI it should use the following format vX where X is an integer begin with 1. Some examples are:

`http://example.com/api/v2`
`http://api.example.com/v2/resource`
`http://apiv2.example.com/resource`

In our Shopping Application we have included versioning when we build, install and deploy it as OSGI bundle. The OSGI command which includes versioning too:

```
osgi:install -s war:mvn:com.redhat/application-interface/1.0.0-SNAPSHOT/war?Web-ContextPath=<version>/shoppingApplication
```

The version changes are respectively made by editing the run.sh file in our project folder.

We add the <warName> tag under <plugins> tag under the application's pom.xml. This <warName> tag specifies the pattern used for webContextPath. The webContextPath is called in run.bat .

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <version>2.6</version>
  <configuration>
    <webResources>
      <resource>
        <directory>${project.build.outputDirectory}</directory>
        <includes>
          <include>META-INF/LICENSE*</include>
          <include>META-INF/NOTICE*</include>
          <include>META-INF/DEPENDENCIES*</include>
        </includes>
      </resource>
    </webResources>
    <warName>${project.version}.shoppingApplication</warName>
    <archive>
      <manifest>
        <addClasspath>true</addClasspath>
      </manifest>
    </archive>
  </configuration>
</plugin>

```

Figure19. Configuring the WAR tag in the POM file

The new web context path is called using command : “osgi:install -s war:mvn:com.redhat/application-interface/1.0.0-SNAPSHOT/war?Web-ContextPath=% API_VERSION%.shoppingApplication”. In this way we are setting the version for the karaf build.

To set the build for eap we are adding the version details when we are building the application.

```

:eap
call mvn versions:set -DnewVersion=%API_VERSION%
call mvn clean install -DskipTests
goto eapDeployment

```

Figure20. EAP Versioning Configuration

Finally we specify the version value as a variable in run.bat file:

```
set API_VERSION=v1
```

Figure21. Run.bat Version Configuration

After running the application, we can see the versioning configured as soon we hit the URL The URL is now <http://localhost:9090/v1.shoppingApplication/>.

4. Middleware connectors and ESB integration

Why do enterprises deploy APIs? Typically, this is done to allow internal or external consumers simplified access to enterprise backend systems. Often the API provides a front-end that accesses multiple systems on the backend. A sample use case might be for a store that receives new products and inventory from their suppliers. The steps for this might be:

1. receive new product and inventory information
2. store the product information in the product master data repository and create a new SKU (Stock-Keeping Unit – a unique identifier for products in a catalog or store)
3. add the SKU and inventory to the ERP (Enterprise Resource Planning) system
4. add the SKU into the CRM (Customer Relationship Management) system

All of these interactions could occur with just a single call to an API. How is this possible? By leveraging an Enterprise Service Bus (ESB) to integrate and orchestrate across multiple systems. As well an ESB can also do complex data protocol transformations to ensure data is formatted properly for each of the systems noted above.

Red Hat JBoss Fuse provides an ESB - a lightweight standards based, loosely coupled platform. By relying on standards, it reduces the chances of vendor lock-in. By advocating loose coupling, the ESB platform reduces the complexity of integration. This section will demonstrate how easy it is to leverage Red Hat JBoss Fuse to build an orchestration across multiple back-end systems.

In our shopping application, we have a use case similar to the one described above: add product and inventory. This will be somewhat simplified, but still demonstrate the power of the ESB. Figure 23 shows the camel route that has been developed. In this example, we retrieve the incoming add product request from an activemq queue (unmarshalling it from XML with minimal code), we log the information, we load the product into a SQL database, and then we both add the product to our Salesforce CRM system, and also tweet the product addition on Twitter.

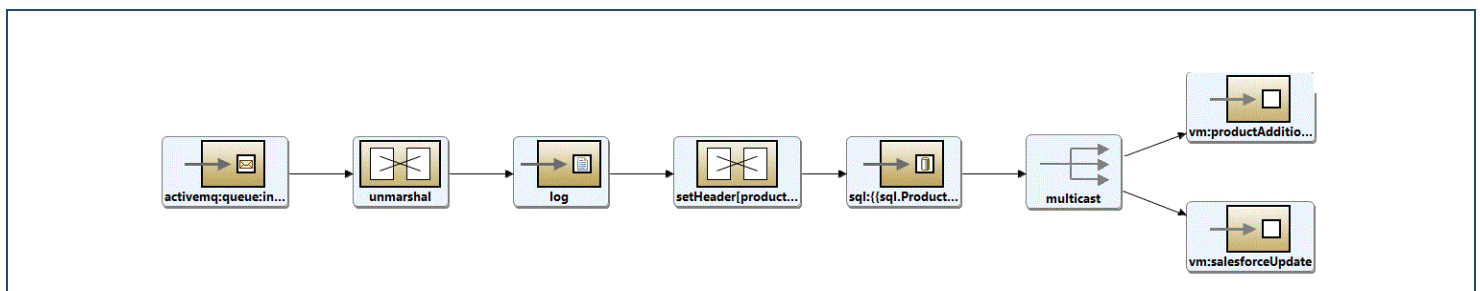


Figure 23. Camel route for adding products and inventory

Note: See the shopping cart application for a larger view of the route in camel-activemqRoutes.xml.

In subsequent sections, we will discuss in more detail how this route gets called via a web service, as well as how the interactions with Salesforce.com and Twitter occur. For now, we will focus mainly on the basic routing functionality.

Understanding the interaction with beans wiring:

This route is enabled through Spring configuration in resources/camel/beans.xml.

First we create an activemq bean using normal Spring configuration:

```

<!-- configuring activemq Start -->
<bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="brokerURL" value="${activemq.brokerUrl}" />
  <property name="userName" value="${activemq.user}" />
  <property name="password" value="${activemq.password}" />
</bean>
<!-- Configuring activemq End -->

```

Figure 24. beans.xml

The route diagram in Figure 23 is configured in camel-activemqRoutes.xml and is shown in XML below.

```

<camelContext trace="false" xmlns="http://camel.apache.org/schema/spring">
  <propertyPlaceholder location="classpath:sql.properties" id="properties"/>
  <route id="activeMqHandler">
    <from uri="activemq:insertProductsFromQueue"/>
    <log message="Inside Queue Addition Of Product"/>
    <unmarshal>
      <jaxb contextPath="com.redhat.shopping.demo.application.pojos.jpas"
        prettyPrint="true"
        partClass="com.redhat.shopping.demo.application.pojos.jpas.Products"/>
    </unmarshal>
    <log message="Adding product: ${body.productName}"/>
    <setHeader headerName="productCode">
      <simple>${body.productCode}</simple>
    </setHeader>
    <setHeader headerName="productName">
      <simple>${body.productName}</simple>
    </setHeader>
    <setHeader headerName="productLine">
      <simple>${body.productLine}</simple>
    </setHeader>
    <setHeader headerName="productScale">
      <simple>${body.productScale}</simple>
    </setHeader>
    <setHeader headerName="productVendor">
      <simple>${body.productVendor}</simple>
    </setHeader>
    <setHeader headerName="productDescription">
      <simple>${body.productDescription}</simple>
    </setHeader>
    <setHeader headerName="quantityInStock">
      <simple>${body.quantityInStock}</simple>
    </setHeader>
    <setHeader headerName="buyPrice">
      <simple>${body.buyPrice}</simple>
    </setHeader>
    <setHeader headerName="MSRP">
      <simple>214.3</simple>
    </setHeader>
    <log message="Executing {{sql.Products.addProduct}}"/>
    <to uri="sql:{{sql.Products.addProduct}}"/>
    <multicast parallelProcessing="true">
      <to uri="vm:productAdditionTweet"/>
      <to uri="vm:salesforceUpdate"/>
    </multicast>
    <log message="Adding product complete"/>
  </route>

```

Figure 25. camel-activemqRoutes.xml

This creates a route named activeMqHandler. When a product XML is retrieved from the queue, it is first unmarshalled. We log the addition, We then execute a SQL statement defined in the sql.properties file by routing it to a special sql uri. Then we multicast it to salesforce and twitter.

Middleware Connectors:

As shown in this section, we have used a number of different middleware connectors. The ones we have used in our Shopping Application are:

Connector	How it is used in our Shopping Application
File	See Section 4
JDBC	See Section 4
Twitter	See Section 7.3
Salesforce	See Section 7.2
Jetty	See Section 2
Gauth	See Section 2
Bean	See Section 2,4,6,7
SQL	See Section 4
CXF	See Section 4 and 5
ActiveMQ	See Section 7.1

So now that it is clear what happens when a message is retrieved from the queue, how does the message get on the queue in the first place? That is answered in the next section.

5. API for Independent Web Services

APIs can be simple RESTful services, or they can be the more traditional WSDL-based SOAP services. In this section, we will discuss how to expose a SOAP service.

In the previous section, we built a camel route for adding products. This route accepted data on an inbound ActiveMQ queue. Now, we need to make this available as a web service so that it can easily be consumed outside the enterprise. Using JBoss tools, this is a simple endeavor.

For this example, instead of developing the route in XML, we are building it in Java so that we can take advantage of some java APIs. This component, AddProductRoute.java, is shown below. Note that in the example below, we are loading products from a URI that needs to be accessible to the server running the web service. The examples include a file:// URI that loads the products from the local server to simplify the example. In an enterprise implementation we would use a different approach for loading products.

```
public class AddProductRoute extends RouteBuilder {

    private static final Logger LOGGER = Logger.getLogger(AddProductRoute.class);

    public void configure() {

        LOGGER.info("enter configure()");

        Tracer tracer = new Tracer();
        tracer.setLogLevel(LoggingLevel.INFO);
        getContext().addInterceptStrategy(tracer);

        errorHandler(new NoErrorHandlerBuilder());
        JaxbDataFormat dataFormat = new JaxbDataFormat("com.redhat.shopping.demo.application.pojos.jpas");
        dataFormat.setPartClass("com.redhat.shopping.demo.application.pojos.jpas.Products");
        dataFormat.setPrettyPrint(true);

        from("cxf:bean:productAddition")
        .process(new Processor() {

            public void process(Exchange exchange) throws Exception {
                LOGGER.info("Enter process()");
                String urlPath = (String)exchange.getIn().getBody(List.class).get(0);
                if(urlPath!=null){
                    exchange.getOut().setBody((new URI(urlPath).toURL().openStream()));
                }else{
                    exchange.getOut().setBody("File Path is Null");
                }

                LOGGER.info("Exit process()");
            }

        })
        .split().xpath("/products-list/products").parallelProcessing()
        .log("Adding Products")
        .to("activemq:insertProductsFromQueue")
        .process(new Processor() {
            @Override
            public void process(Exchange exchange) throws Exception {
                exchange.getIn().setBody(new String(IUtils.toByteArray(
                    new FileInputStream(new File("data/addProductsResponse.txt"))));
            }
        });

        LOGGER.info("Configured routes:\n" + this.toString());
        LOGGER.info("exit configure()");
    }
}
```

Figure 26. AddProductRoute

1. In the Add Product route we configure our starting endpoint as a cxfEndpoint. It listens on this cxf:bean:productAddition endpoint.
2. We move on to processor under the process () function. The String urlPath get its value when the web service is called, where the user adds the URL in argument which he wants to send to the web service. For shoppingApplication this URL is the reference URL of xml which contains the list of products to add.
- 3.
4. In the xml file we have parent tags as <products-list><products>..{product info tags}.. . Hence to traverse the product details tags we split through the xpath containing our parent tags.
5. After the split is done we log the product details retrieved from xml to the endpoint of an activemq queue. The products are sent individually to the queue.

The web service is created in camel-servicesContext.xml

```
<cxf:cxfEndpoint id="productAddition"
  address="/shoppingApplication/AddNewProductsService"
  serviceClass="com.redhat.shopping.demo.application.camel.ws.AddNewProducts"
  wsdlURL="classpath:wsdl/AddNewProductsService.wsdl" />
</beans>
```

Figure 27. camel-servicesContext.xml

When the server starts up, the beans are autowired together. The route created in AddProductRoute is wired using cxf:bean:productAddition to be the listener on the web service created in camel-servicesContext.xml.

To test all of this, we recommend using SoapUI^{xiv}, a simple graphical editor where one can create a WSDL project and add an existing WSDL to generate request/response structure. The WSDL for our application is AddNewProductsService.wsdl. Using this, one can create and execute sample requests for all the operations in the service. Our approach for this is shown in Figure 28 below. The shopping application has test XML files available for adding products as well.

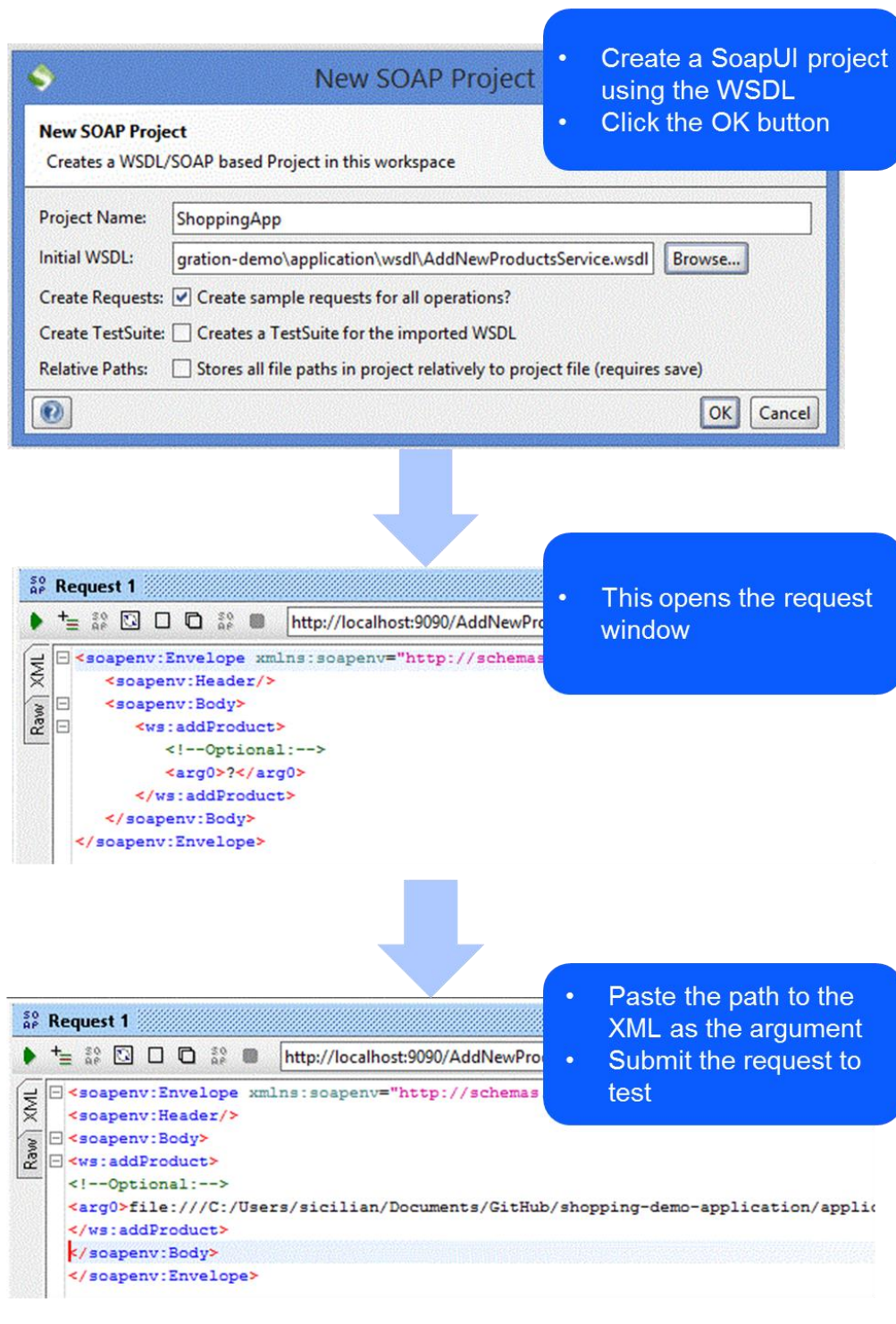


Figure 28. Testing the Web Service using Soap UI

A sample xml file containing the products is shown below:

```

<?xml version="1.0" encoding="UTF-8" ?>
<products-list>
  <products>
    <productCode>testid1testid1</productCode>
    <productName>1985 test Toyota Suprtestid1</productName>
    <productLine>Classic Cars</productLine>
    <productScale>1:10</productScale>
    <productVendor>Min Lin Diecast</productVendor>
    <productDescription>This replica features working kickstand, front suspensi
    <quantityInStock>7933</quantityInStock>
    <buyPrice>48.81</buyPrice>
    <MSRP>95.7</MSRP>
  </products>
  <products>
    <productCode>testid2testid1</productCode>
    <productName>1904 test Buictestid1k Runabout</productName>
    <productLine>Classic Cars</productLine>
    <productScale>1:10</productScale>
    <productVendor>Classic Metal Creations</productVendor>
    <productDescription>Turnable front wheels; steering function; detailed inte
    <quantityInStock>7305</quantityInStock>
    <buyPrice>98.58</buyPrice>
    <MSRP>214.3</MSRP>
  </products>
  <products>
    <productCode>testid3testid1</productCode>
    <productName>1948 test Porstestid1che Type 356 Roadster</productName>
    <productLine>Vintage Cars</productLine>
    <productScale>1:10</productScale>
    <productVendor>Highway 66 Mini Classics</productVendor>
    <productDescription>Official Moto Guzzi logos and insignias, saddle bags lo
    <quantityInStock>6625</quantityInStock>
    <buyPrice>68.99</buyPrice>
    <MSRP>118.94</MSRP>
  </products>
</products-list>

```

Figure 29. XML file containing the products details (data/addProducts.xml)

Note: See the shopping cart application for the actual file details

6. Mediation – Message Validation, Message Transformation

Message Validation is confirming the validity of messages. It's concerned with message level validation and not business level validation. It can validate messages to their schemas, but not executing any business rules on messages.

We have implemented a use case in our shopping application in which the “buy product” requests are validated and then executed. If the user credentials are not provided in the request then the validation fails and a validation exception is thrown which provides an appropriate message to the user, else the request is processed further.

```
public class BuyNewProduct extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        onException(ValidationException.class).handled(true)
        .process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                Response response = Response.serverError().status(Status.FORBIDDEN).entity(
                    "The user Details are missing").build();
                throw new WebApplicationException(response);
            }
        });

        from("vm:buyProductsByCode")
        .log("Buy Request Started")
        .log("${body}")
        .setHeader("productCode", simple("${body[0]}"))
        .validate().simple("${body[1]} != null")
        .setHeader("customerDetails", simple("${body[1]}"))
        .to("activemq:buyProductsByCode")
        .transform(constant("Thank you for buying the products"));
    }
}
```

Figure 30. BuyNewProduct.java Using Camel Validation Procedure

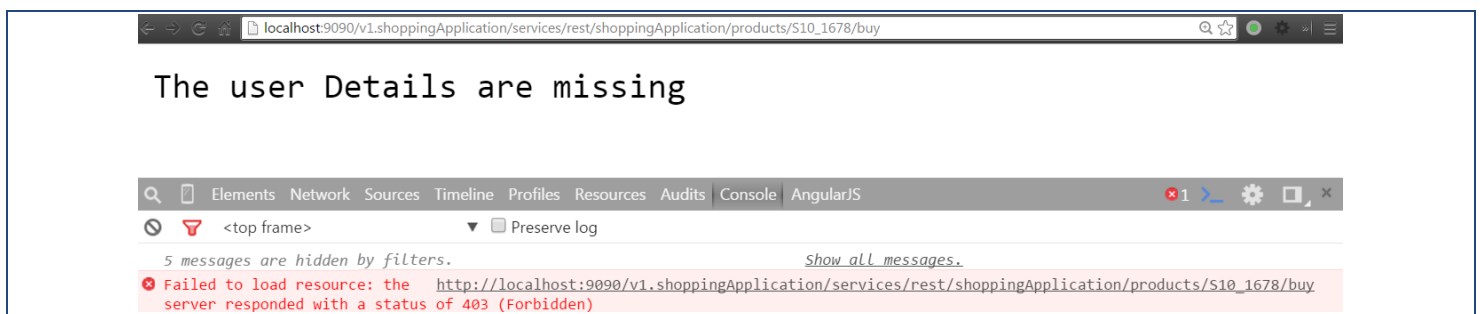


Figure 31. Validation Failure Message

The dynamic router in the rest server is another implementation of our shopping application which validates incoming request:

1. The endpoint `cxfrs:rsServer` is configured with the address of `http://localhost:9090/route`
2. Upon hitting this URL the dynamic router, which is surrounded by method reference of `restRouteProcessor`, calls it.
3. `RestRouteProcessor` implements the `Processor` class which sets the values of the exchange object headers as per the operationName. This Processor is being watched over by a Rest Service `ShoppingHomeService`. This service includes the annotations of `@Path`, `@Produces` and function name.
4. Whatever the path, a user hits after the `cxfrs:rsServer` URL, that path is being validated first in `ShoppingHomeService` class and the `RestRouteProcessor` sets the Operation Name according to the function called in rest service as per its path.
5. We will show the flow and snippets of `ShoppingHomeService` and `RestRouteProcessor` which explains our use case.

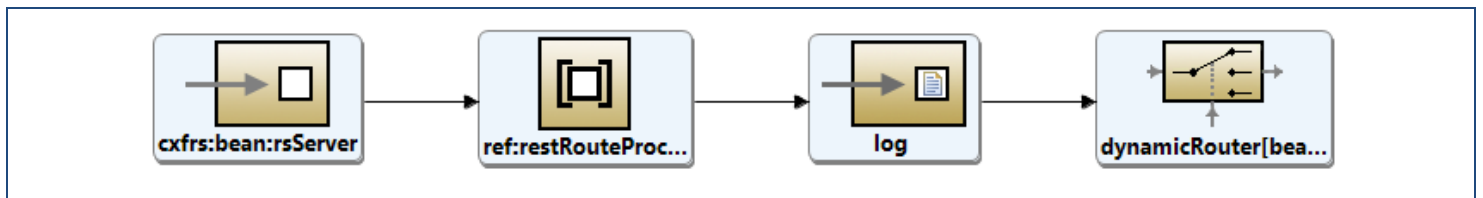


Figure 32. RestServiceFlow

```
@Path("/shoppingApplication")
public class ShoppingHomeService {

    @GET
    @Path("/products")
    @Produces(MediaType.APPLICATION_JSON)
    public String showProducts(){
        return null;
    }

    @GET
    @Path("/products/{productCode}/buy")
    @Produces(MediaType.APPLICATION_JSON)
    public String buyProductsByCode( @PathParam("productCode") String productCode,
                                     @QueryParam("customerDetails") String customerDetails){
        return null;
    }

    @GET
    @Path("/showTransactions")
    @Produces(MediaType.APPLICATION_JSON)
    public String showPreviousTransactions(@QueryParam("customerDetails") String customerDetails){
        return null;
    }
}
```

Figure 33. ShoppingHomeService to depict our rest service watching over the processor.

We can see the Shopping Application specifies when to return a JSON or a XML and by what path and its function name which is being validated.

Understanding the RestRouteProcessor:

RestRouteProcessor get its Operation Name(method) after the ShoppingHomeService responds.

```
public class RestRouteProcessor implements Processor {
    private String consumerType;
    public void process(Exchange exchange) throws Exception {
        Message inMessage = exchange.getIn();
        String operationName = inMessage.getHeader(
            CxfConstants.OPERATION_NAME, String.class);
        exchange.getIn().setHeader("resourceMapping",
            consumerType + ":" + operationName);
        exchange.getIn().setHeader("awaitingResponse", true);
    }

    public String slip(Exchange exchange) {
        if (exchange.getIn().getHeader("awaitingResponse", Boolean.class)) {
            exchange.getIn().setHeader("awaitingResponse", false);
            return exchange.getIn().getHeader("resourceMapping", String.class);
        }
        return null;
    }

    public String getConsumerType() {
        return consumerType;
    }

    public void setConsumerType(String consumerType) {
        this.consumerType = consumerType;
    }
}
```

Figure 34. RestRouteProcessor

Message Transformation:

Message transformation is modifying the format of the message from one type to another. (e.g.: from XML or text, json). It is used to modify the type of message required from one system to the type required by another system. It can also be used to transform the content (body) of a message the consumer sends to match the content of the provider. (e.g.: date format MM:DD:YY to DD:MM:YYYY)

Message parsing and formatting services provide the ability to manipulate messages into different formats, based on the message structure. This is a syntactical transformation of the data, based generally on technical considerations such that the message is more usable to receiving systems.

Message encryption services provide confidentiality of information, both while in transit, as well as when stored. Other technologies for confidentiality such as secure sockets layer (SSL)/transport layer security (TLS) or virtual private networks (VPNs) only provide confidentiality while the information is in transit, not while it is stored at a server.

We have applied the concept of message transformation in many routes. Below is an example for the same. We are displaying all the *Product* objects from SQL and transforming it to JSON as we need to produce a JSON response for the rest service. The result is displayed as an html table in the application page. See the figures below for details.

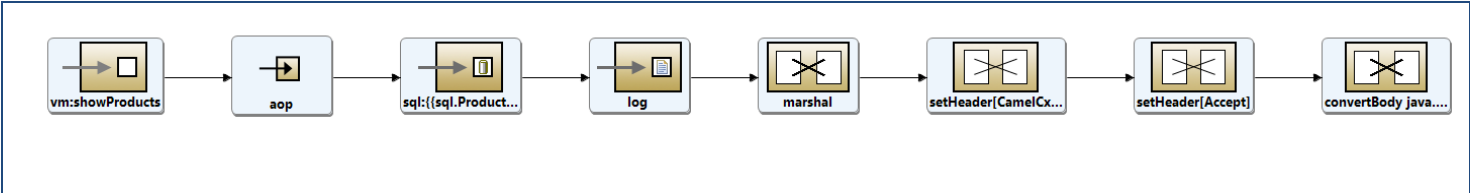



Figure 35. camel-showProducts.xml



Welcome to shopping application!

Choose an option

productCode	productName	productLine	productScale	productVendor	productDescription	quantityInStock	buyPrice	MSRP
S10_1678	1969 Harley Davidson Ultimate Chopper	Motorcycles	1:10	Min Lin Diecast	This replica features working kickstand, front suspension, gear-shift lever, footbrake lever, drive chain, wheels and steering. All parts are particularly delicate due to their precise scale and require special care and attention.	7927	48.81	95.7
S10_1949	1952 Alpine Renault 1300	Classic Cars	1:10	Classic Metal Creations	Turnable front wheels; steering function; detailed interior; detailed engine; opening hood; opening trunk; opening doors; and detailed chassis.	7305	98.58	214.3
S10_2016	1994 Moto Guzzi 1100i	Motorcycles	1:10	Highway 66 Mini Classics	Official Moto Guzzi logos and insignias, saddle bags located on side of motorcycle, detailed engine, working steering suspension, two leather seats, luggage rack, dual exhaust pipes, small saddle bag located on handle bars, two-tone paint with chrome accents, superior die-cast detail, rotating wheels, working kick stand, diecast metal with plastic parts and baked enamel finish.	6625	68.99	110.94
S10_4698	2003 Harley-Davidson Eagle Drag Bike	Motorcycles	1:10	Red Start Diecast	Model features, official Harley Davidson logos and insignias, detachable rear wheelie bar, heavy diecast metal with resin parts, authentic multi-color tampo-printed graphics, separate engine drive belts, free-turning front fork, rotating tires and rear racing slick, certificate of authenticity, detailed engine, display stand, precision diecast replica, baked enamel finish, 1:10 scale model, removable fender, seat and tank cover piece for displaying the superior detail of the v-twin engine.	5582	91.02	193.66
S10_4757	1972 Alfa Romeo GTA	Classic Cars	1:10	Motor City Art Classics	Features include: Turnable front wheels; steering function; detailed interior; detailed engine; opening hood; opening trunk; opening doors; and detailed chassis.	3252	85.68	136
S10_4962	1962 Lancia Delta 16V	Classic Cars	1:10	Second Gear Diecast	Features include: Turnable front wheels; steering function; detailed interior; detailed engine; opening hood; opening trunk; opening doors; and detailed chassis.	6791	103.42	147.74
S12_1099	1968 Ford Mustang	Classic Cars	1:12	Autoart Studio	Hood, doors and trunk all open to reveal highly detailed interior features. Steering wheel actually turns the front wheels.	60	95.34	194.57

Figure 36.Result of the SQL to JSON displayed in the application

7. Productization –Third party API integration

Use the API platform to orchestrate internal and third party APIs to create new mash up APIs that can power new business services and applications. Use the API platform to refactor backend API services and data sets to be more users centric.

7.1. A-MQ Connector

Fuse easily consumes and publishes messages from JMS queues or topics. The ActiveMQ component allows messages to be sent to or consumed from a JMS Queue or Topic.

Understanding the interaction of AMQ Connector with the Shopping Application

We are unmarshalling the *Products* which were inserted in queue. As per the camel route we can see that we set the headers as per the *Product* fields. After setting the values they are sent to the database via the *addProducts* query. After the query is executed the *Products* are sent to the twitter and salesforce endpoints.

For understanding the flow of the ActiveMQ camel route, we will again show the figure below to understand the concept of its flow:

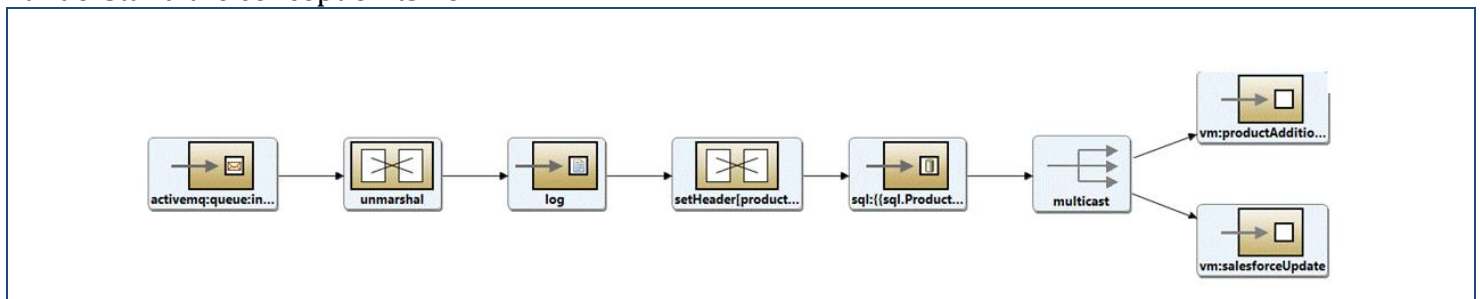


Figure 37. Add product camel route

Below we have mentioned the ActiveMQ camel route, which explains how the products have been given values and how they are sent from amq queue to sql update query.


```

<camelContext trace="false" xmlns="http://camel.apache.org/schema/spring">
<propertyPlaceholder location="classpath:sql.properties" id="properties"/>
<route id="activeMqHandler">
  <from uri="activemq:insertProductsFromQueue"/>
  <log message="Inside Queue Addition Of Product"/>
  <unmarshal>
    <jaxb contextPath="com.redhat.shopping.demo.application.pojos.jpa"
      prettyPrint="true"
      partClass="com.redhat.shopping.demo.application.pojos.jpa.Products"/>
  </unmarshal>
  <log message="Adding product: ${body.productName}"/>
  <setHeader headerName="productCode">
    <simple>${body.productcode}</simple>
  </setHeader>
  <setHeader headerName="productName">
    <simple>${body.productName}</simple>
  </setHeader>
  <setHeader headerName="productLine">
    <simple>${body.productLine}</simple>
  </setHeader>
  <setHeader headerName="productScale">
    <simple>${body.productScale}</simple>
  </setHeader>
  <setHeader headerName="productVendor">
    <simple>${body.productVendor}</simple>
  </setHeader>
  <setHeader headerName="productDescription">
    <simple>${body.productDescription}</simple>
  </setHeader>
  <setHeader headerName="quantityInStock">
    <simple>${body.quantityInStock}</simple>
  </setHeader>
  <setHeader headerName="buyPrice">
    <simple>${body.buyPrice}</simple>
  </setHeader>
  <setHeader headerName="MSRP">
    <simple>214.3</simple>
  </setHeader>
  <log message="Executing {{sql.Products.addProduct}}"/>
  <to uri="sql:{{sql.Products.addProduct}}"/>
  <multicast parallelProcessing="true">
    <to uri="vm:productAdditionTweet"/>
    <to uri="vm:salesforceUpdate"/>
  </multicast>
  <log message="Adding product complete"/>
</route>

```

Figure 38. camel-activemqRoutes.xml

7.2. Salesforce

Fuse easily interfaces with Salesforce API's for inbound as well as outbound. This component supports producer and consumer endpoints to communicate using Java DTOs. There is a companion maven plugin, Camel Salesforce Plugin that generates these DTOs. Two DTOs were generated namely ProductsObject_c and QueryRecordsProductsObject_c. ProductsObject_c shows only ProductName and ProductPrice.

Understanding the interaction of Salesforce Component with Fuse:

We are basically updating the Exchange object with new values as we need to show only *ProductName* and *ProductPrice*. Using camel-salesforce route we update the new values of Exchange Object to Salesforce site. Below we are showing the flow, which works in camel router:

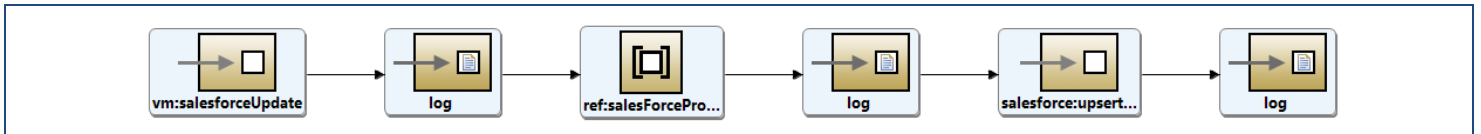


Figure 39. Camel route to insert product into Salesforce (MulticastInsertion:Salesforce)

Understanding the beans wiring in Salesforce:

Below are the beans we have created to use salesforce functionality. These beans are available at resources/beans.xml. Since all the files in resources folder are loaded in fuse the beans are created on deployment.

```
<bean id="salesForceProductProcessor"  
      class="com.redhat.shopping.demo.application.camel.beans.SalesForceProductProcessor" />
```

Figure 40. beans.xml

Below is the class of the ProductsObject:

```

/**
 * Salesforce DTO for SObject ProductsObject__c
 */
@XStreamAlias("ProductsObject__c")
public class ProductsObject__c extends AbstractSObjectBase {

    // ProductItemName__c
    private String ProductItemName__c;

    @JsonProperty("ProductItemName__c")
    public String getProductItemName__c() {
        return this.ProductItemName__c;
    }

    @JsonProperty("ProductItemName__c")
    public void setProductItemName__c(String ProductItemName__c) {
        this.ProductItemName__c = ProductItemName__c;
    }

    // ProductItemPrice__c
    private Double ProductItemPrice__c;

    @JsonProperty("ProductItemPrice__c")
    public Double getProductItemPrice__c() {
        return this.ProductItemPrice__c;
    }

    @JsonProperty("ProductItemPrice__c")
    public void setProductItemPrice__c(Double ProductItemPrice__c) {
        this.ProductItemPrice__c = ProductItemPrice__c;
    }

}

```

Figure 41. ProductsObject

Understanding the Router:

In the SalesForceProductProcessor we are replacing the original exchange object containing all the product's data to the exchange object containing only two fields i.e. name and price.

```

public class SalesforceProductProcessor implements Processor {

    public void process(Exchange exchange) throws Exception {
        ProductsObject__c object__c =new ProductsObject__c();
        Products product = exchange.getIn().getBody(Products.class);
        object__c.setProductItemName__c(product.getProductName());
        object__c.setProductItemPrice__c(product.getBuyPrice().doubleValue());
        object__c.setName(product.getProductName());
        exchange.getIn().setBody(object__c, ProductsObject__c.class);
    }

}

```

Figure 42. SalesforceProductProcessor

Finally in the camel-salesforce route we can see how the values are updated in salesforce.

```

<bean id="salesforce"
    class="org.apache.camel.component.salesforce.SalesforceComponent">
    <property name="loginConfig">
        <bean class="org.apache.camel.component.salesforce.SalesforceLoginConfig">
            <property name="clientId" value="${salesforce.clientId}" />
            <property name="clientSecret" value="${salesforce.clientSecret}" />
            <property name="userName" value="${salesforce.userName}" />
            <property name="password" value="${salesforce.password}" />
        </bean>
    </property>
</bean>

<camelContext id="salesforce-example-context" xmlns="http://camel.apache.org/schema/spring">
<route id="salesforce-route">
    <from uri="vm:salesforceUpdate"/>
    <log message="Entered Sales Force Update Route"/>
    <process ref="salesForceProductProcessor"/>
    <log message="Creating Products with name ${body.name}..."/>
    <to uri="salesforce:upsertSObject?sObjectName=ProductsObject__c&sObjectIdName=Name"/>
    <log message="Done creating Product with success=${body.success} and errors=${body.errors}"/>
</route>

```

Figure 43. camel-salesforce.xml

When this is executed, you can see the result in your salesforce instance:

ProductsObject__c ProductsObject__c@1:49 PM ProductsObject__c@1:49 PM

SELECT Id, OwnerId, IsDeleted, Name, CreatedDate, CreatedById, LastModifiedDate, LastModifiedById, SystemModstamp, ProductItemName__c, ProductItemPrice__c FROM ProductsObject__c

Query Results - Total Rows: 9

Id	OwnerId	IsDeleted	Name	CreatedDate	CreatedById	LastModifiedDate	LastModifiedById
a029000000QlinEAAT	00590000003BjtmAAC	false	1965 Aston Martin DB5	2015-02-18T09:27:24.000+0000	00590000003BjtmAAC	2015-02-18T09:27:24.000+0000	00590000003BjtmAAC
a029000000QlinOAAT	00590000003BjtmAAC	false	1993 Mazda RX-7	2015-02-18T09:27:26.000+0000	00590000003BjtmAAC	2015-02-18T09:27:26.000+0000	00590000003BjtmAAC
a029000000QlinJAAT	00590000003BjtmAAC	false	1936 Mercedes-Benz 500K Spe...	2015-02-18T09:27:25.000+0000	00590000003BjtmAAC	2015-02-18T09:27:25.000+0000	00590000003BjtmAAC
a029000000QlipGAAT	00590000003BjtmAAC	false	1985 Toyota Supra	2015-02-18T09:47:30.000+0000	00590000003BjtmAAC	2015-02-18T09:47:30.000+0000	00590000003BjtmAAC
a029000000QlipLAAT	00590000003BjtmAAC	false	1948 Porsche Type 356 Roadster	2015-02-18T09:47:32.000+0000	00590000003BjtmAAC	2015-02-18T09:47:32.000+0000	00590000003BjtmAAC
a029000000QlipQAAT	00590000003BjtmAAC	false	1904 Buick Runabout	2015-02-18T09:47:33.000+0000	00590000003BjtmAAC	2015-02-18T09:47:33.000+0000	00590000003BjtmAAC
a029000000QM8jZAAT	00590000003BjtmAAC	false	1904 test Buick Runabout	2015-02-25T02:45:10.000+0000	00590000003BjtmAAC	2015-02-25T02:45:10.000+0000	00590000003BjtmAAC
a029000000QM8jeAAD	00590000003BjtmAAC	false	1948 test Porsche Type 356 Ro...	2015-02-25T02:45:11.000+0000	00590000003BjtmAAC	2015-02-25T02:45:11.000+0000	00590000003BjtmAAC
a029000000QM8jUAAT	00590000003BjtmAAC	false	1985 test Toyota Supra	2015-02-25T02:45:08.000+0000	00590000003BjtmAAC	2015-02-25T02:45:08.000+0000	00590000003BjtmAAC

Figure 44. Results in SFDC

7.3. Twitter

Fuse allows direct, polling, or event-driven consumption of timelines, users, trends, and direct messages using the Twitter API.

The Twitter component enables the most useful features of the Twitter API by encapsulating Twitter4J. It allows direct, polling, or event-driven consumption of timelines, users, trends, and direct messages. Also, it supports producing messages as status updates or direct messages.

Twitter now requires the use of OAuth for all client application authentication. In order to use camel-twitter with your account, you'll need to create a new application within Twitter at <https://dev.twitter.com/apps/new> and grant the application access to your account. Finally, generate your access token and secret.

Understanding the interaction of Twitter Component with Fuse:

We are showing below the flow in which twitter component works: Here we are basically converting the products which were added during sql query (more explanation about product addition can be seen in Section 4) are being transformed to a String and then tweeted on the timeline of the user logged in.

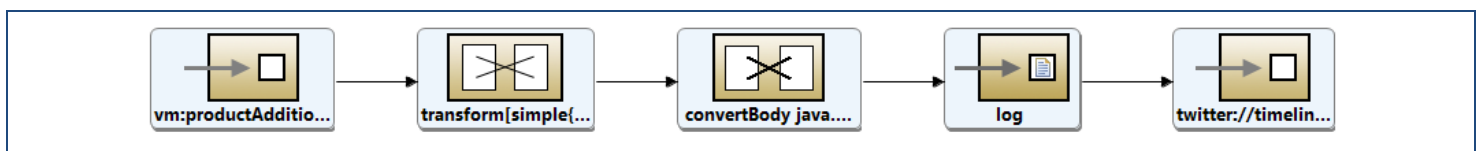


Figure 45. MulticastInsertion-twitter

This leverages the Camel TwitterComponent, which is configuration as shown below. This component allows for polling and tweeting.


```

<bean id="twitter"
  class="org.apache.camel.component.twitter.TwitterComponent">
    <property name="consumerKey" value="${twitter.consumerKey}" />
    <property name="consumerSecret" value="${twitter.consumerSecret}" />
    <property name="accessToken" value="${twitter.accessToken}" />
    <property name="accessTokenSecret" value="${twitter.accessTokenSecret}" />
  </bean>

```

Figure 46. Twitter config in camel-twitter.xml

When this is executed, the application automatically generates a tweet upon each product creation.



Figure 47. Twitter feed when products are inserted

8. Certification and testing

Team needs to be accountable for defining the testing methodology for the development factory/team. Typical scenario is to define unit test cases prior to even starting build activities. Those test cases are used to drive detailed requirements and build. That establishes processes & guidelines to ensure that unit / integration / consumer acceptance testing coverage is sufficient depending on API classification/type. Set requirements on regression testing and performance testing deployment gates that guides subsequent testing processes.

API hierarchy of needs –

1. Usability - Is it easy to set up?
2. Functionality - Does it work as expected / documented?
3. Reliability - Is it “reliable” over time?
4. Proficiency - Does it increase the developer's skills?
5. Creativity - Can it be used in new ways?

Putting more effort into API testing, can leads to a much healthier final product. Ensuring that all data access (read and write) goes only through the API significantly simplifies security and compliance testing and thereby certification, since there is only one interface.

Ensuring that all the required business rules are being enforced at the API tier allows time for much more complete user-experience tests once the UI is released, and not having to concentrate on testing every single business rule and path through the application near the end of the project. Ensuring that the API offers complete functionality allows for easy future expansion of the application as new business needs arise.

JBoss tools comes with camel plugins which make testing quite easy. We can create a test suite and configure test scripts to be executed depending upon the requirements. The test scripts can be executed periodically and also on every new commit in the git repository. JBoss tools can be easily integrated with Jenkins to automate the build and deployment cycle of the API services.

9. Data services – Direct data access, Meta data services

Red Hat JBoss Data Virtualization (JDV) allows for an application to implement a virtual layer so that multiple data endpoints act as a single data source within the application. Some key benefits include:

- Expose data through a single uniform interface. Data sources can be txt, csv or different database server
- Data from different sources can be handled by a Rest Service.
- JDV provides a platform to access data using the same query language irrespective of the database server. The data handling implementation is handled by JDV which decouples data handling code in the API level and the data configuration. The feature facilitates in API development a lot. Using JDV as the data management server the overhead of managing different drivers and protocols for accessing different data sources is not to be implemented in the API.
- Expose legacy data sources as data services. Since JDV exposes all its data sources by a REST service and by a query language. The feature supports easy implementation to handle data at API level.
- Provide a uniform means of exposing/accessing metadata
- Provide a searchable interface to data and metadata
- Expose data relationships and semantics
- Provide uniform access controls to information

For our shopping application, we chose to create a virtualization layer that allows the application to interact with two data sources (MySQL and Postgres) through a unified interface. The JDV server seamlessly forwards all the SQL calls to the proper data source.

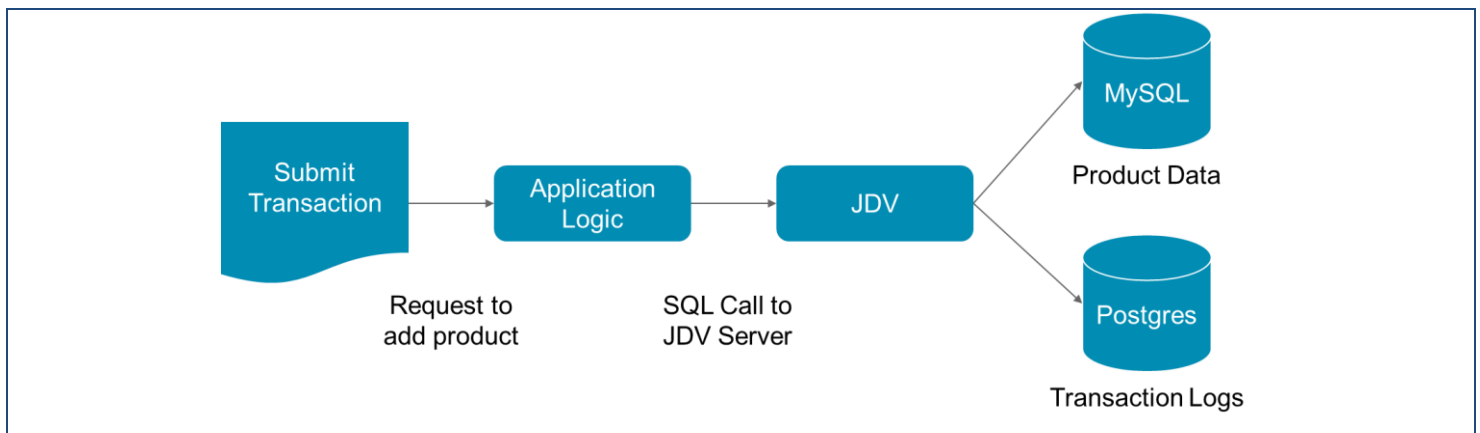


Figure 48. JDV Usage in the Shopping Cart Application

The steps to create this are very simple using JBDS. These steps are illustrated in the figures below.

1. Create a Teiid project
2. Create the MySQL data model
3. Create the PostgreSQL data model
4. Create a virtual database that exposes these two models as a single datasource to your application

First we will create the MySQL data model.

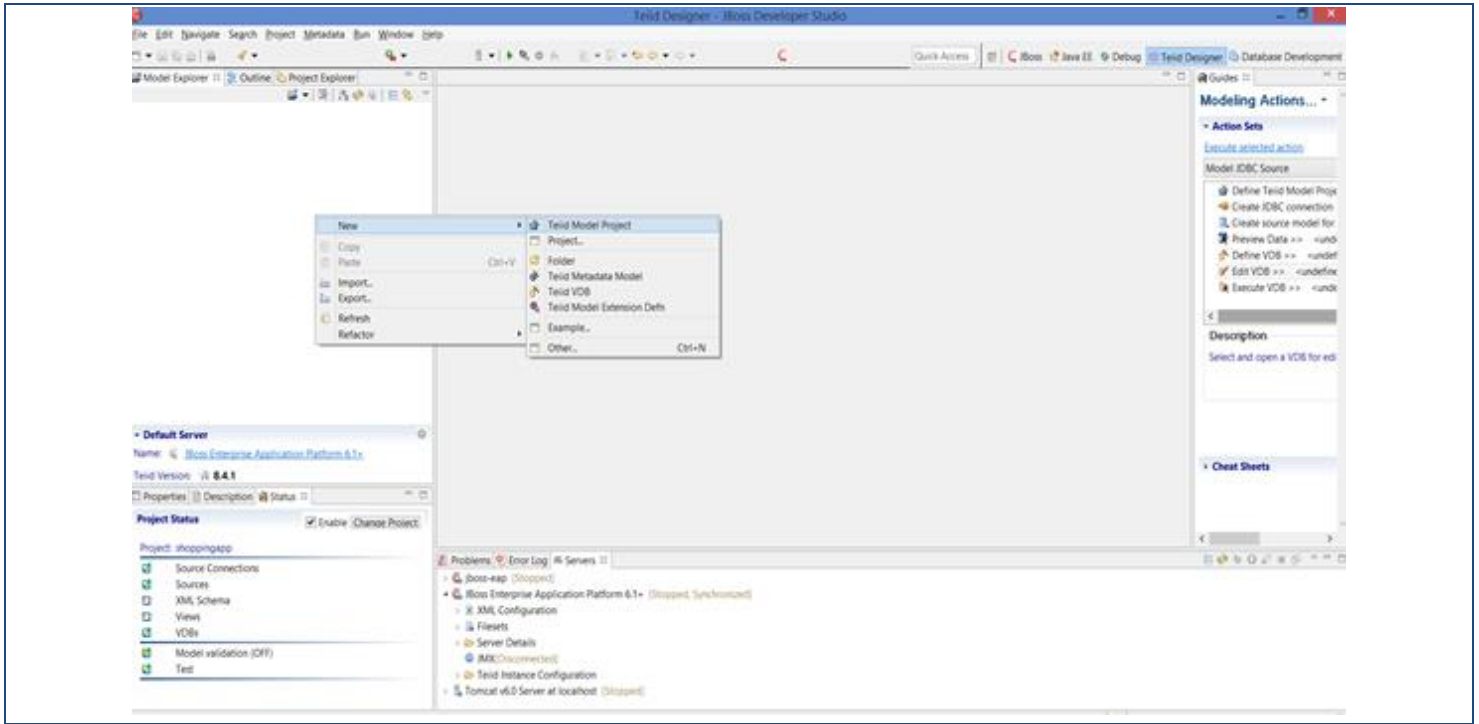


Figure 49. Creating a Teiid Model Project

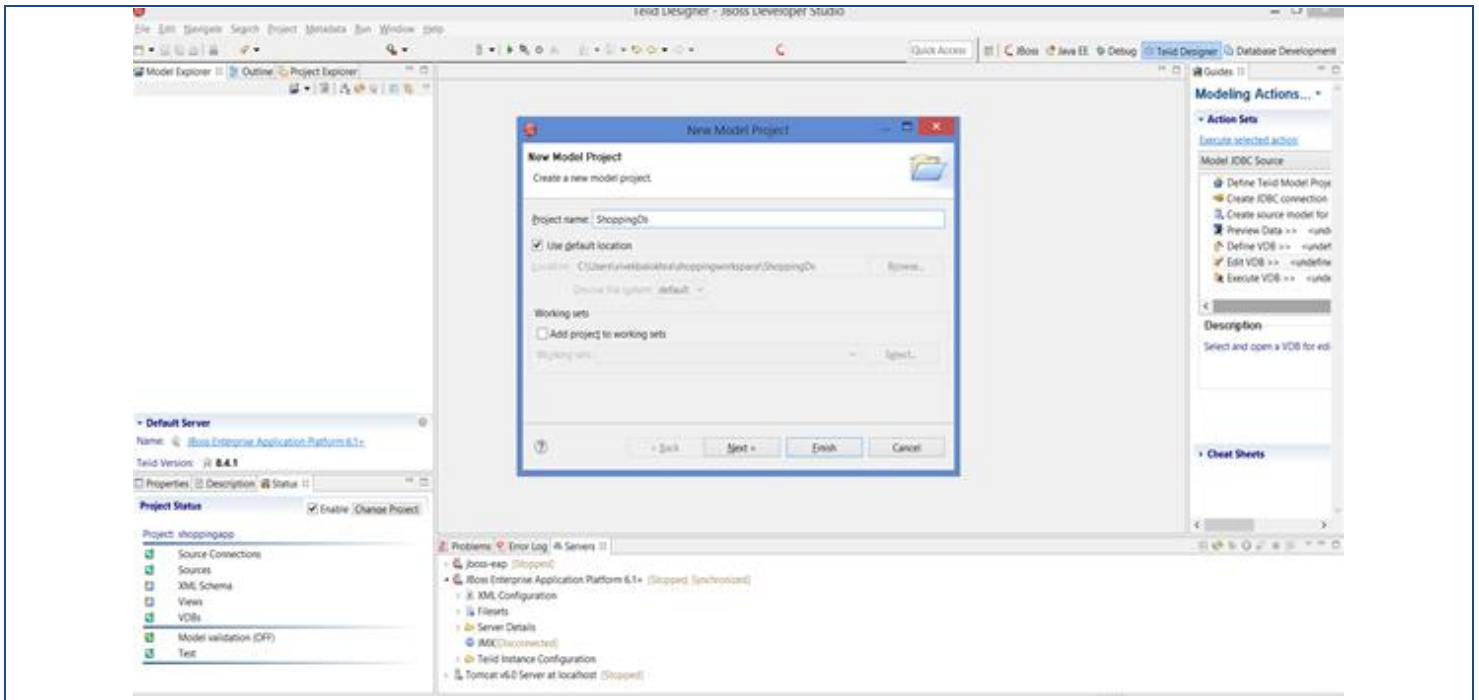


Figure 50. Creating the new Model Project

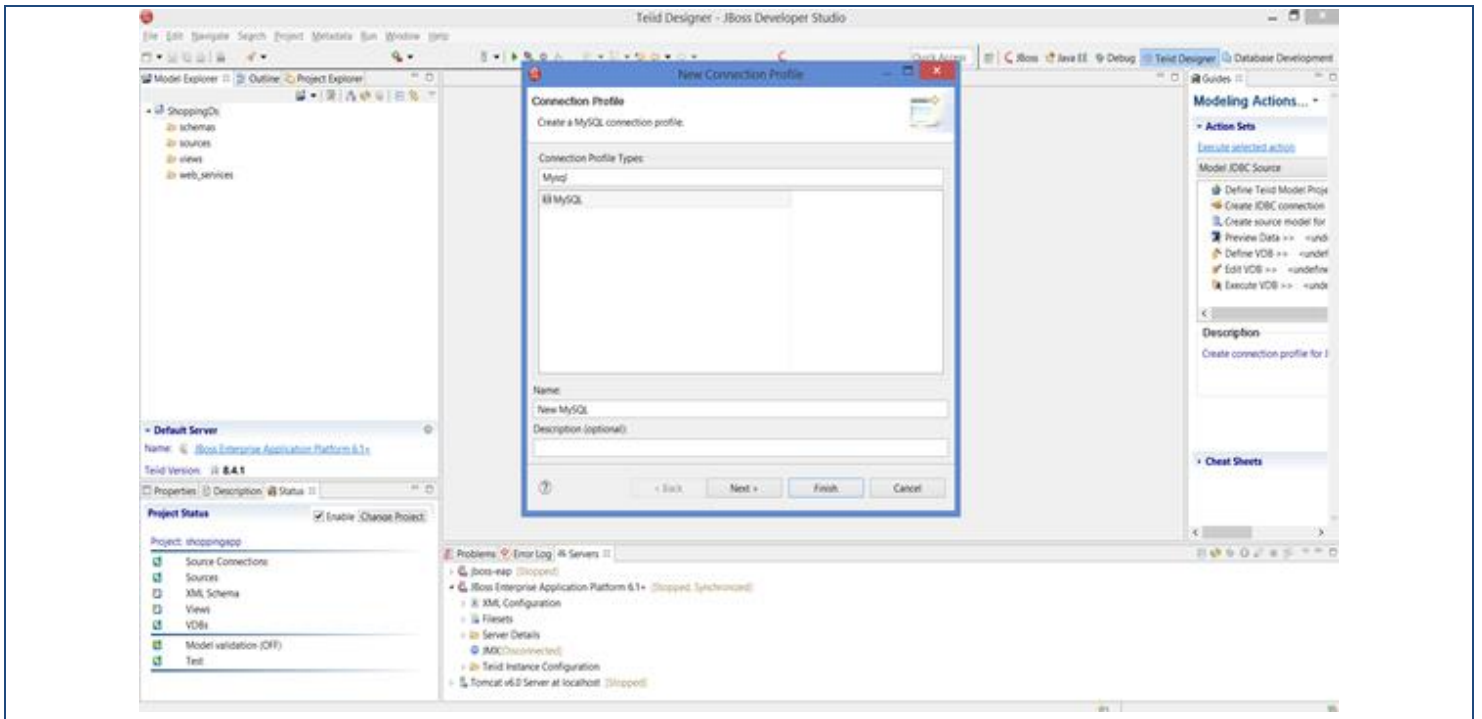


Figure 51. Creating a new JDBC connection and configuring MySQL

After configuring the connection details with its respective username and password we click on Finish.

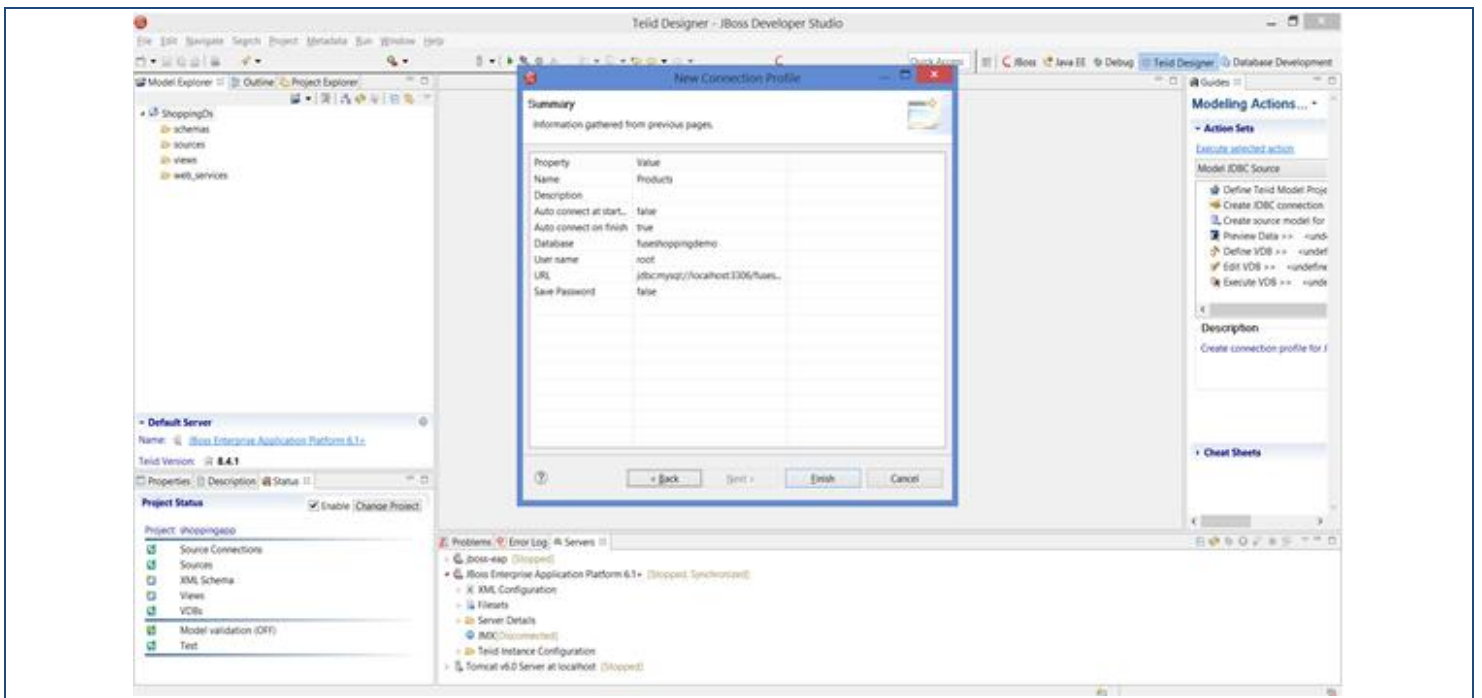


Figure 52. The completed MySQL datasource

We can now view the data model for the MySQL connection. The source model of the fuseShoppingDemo database now shows us the UML diagram of products with their fields.

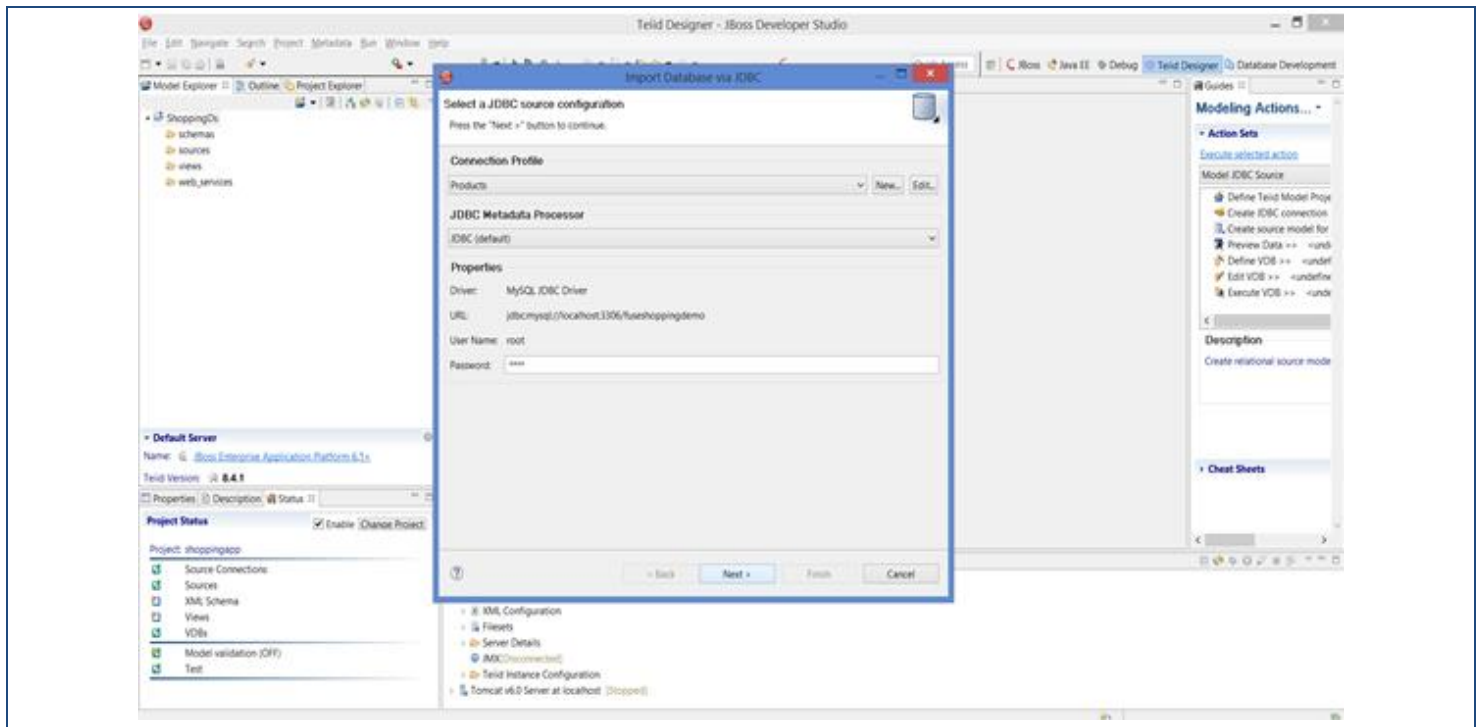


Figure 53. Select the MySQL data source

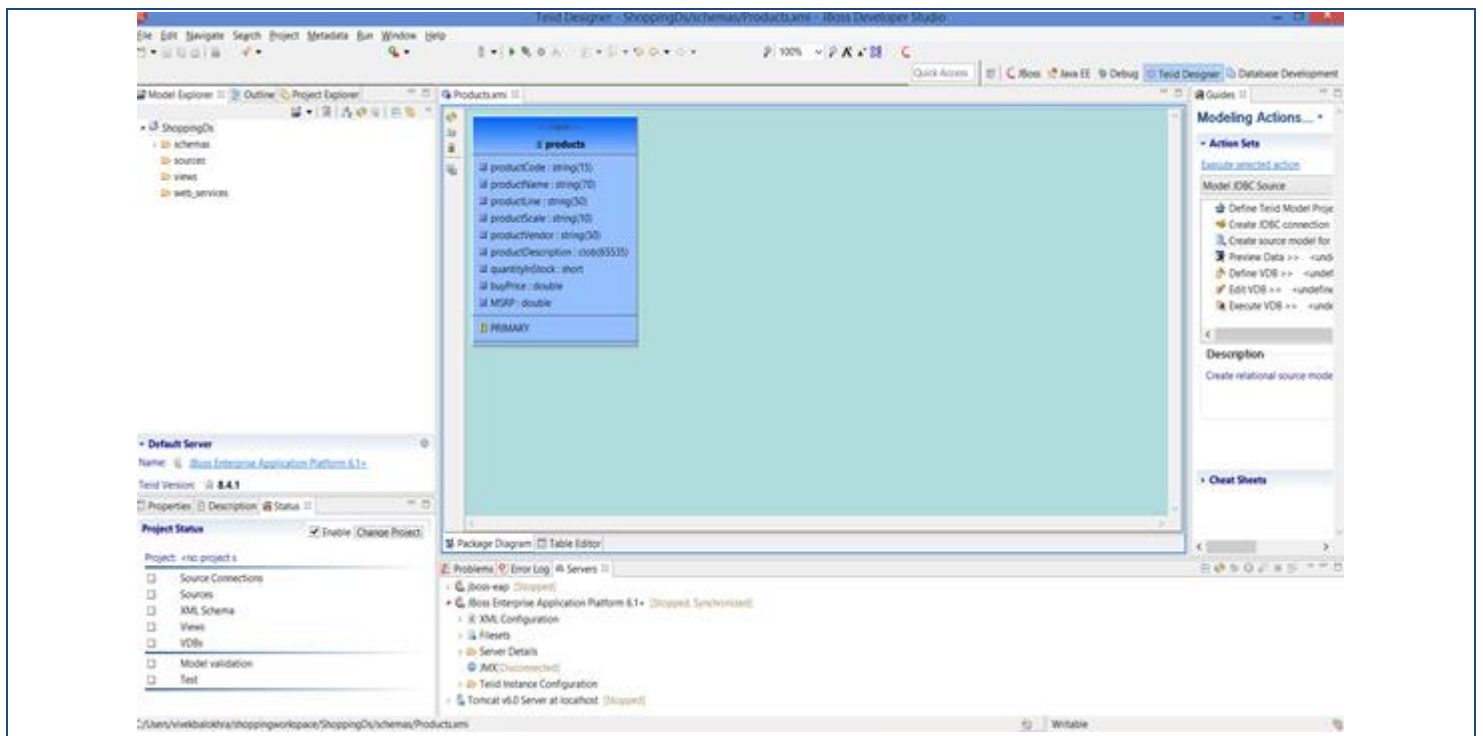


Figure 54. Viewing the source model of the fuseShoppingDemo database

Since we are creating a virtual database across two different data sources, we now need to follow the same steps for a PostgreSQL data source. We follow the same steps as for MySQL to create a new connection and view the source model. After connecting to the Transaction database in PostgreSQL, we can see the source model in the figure below.

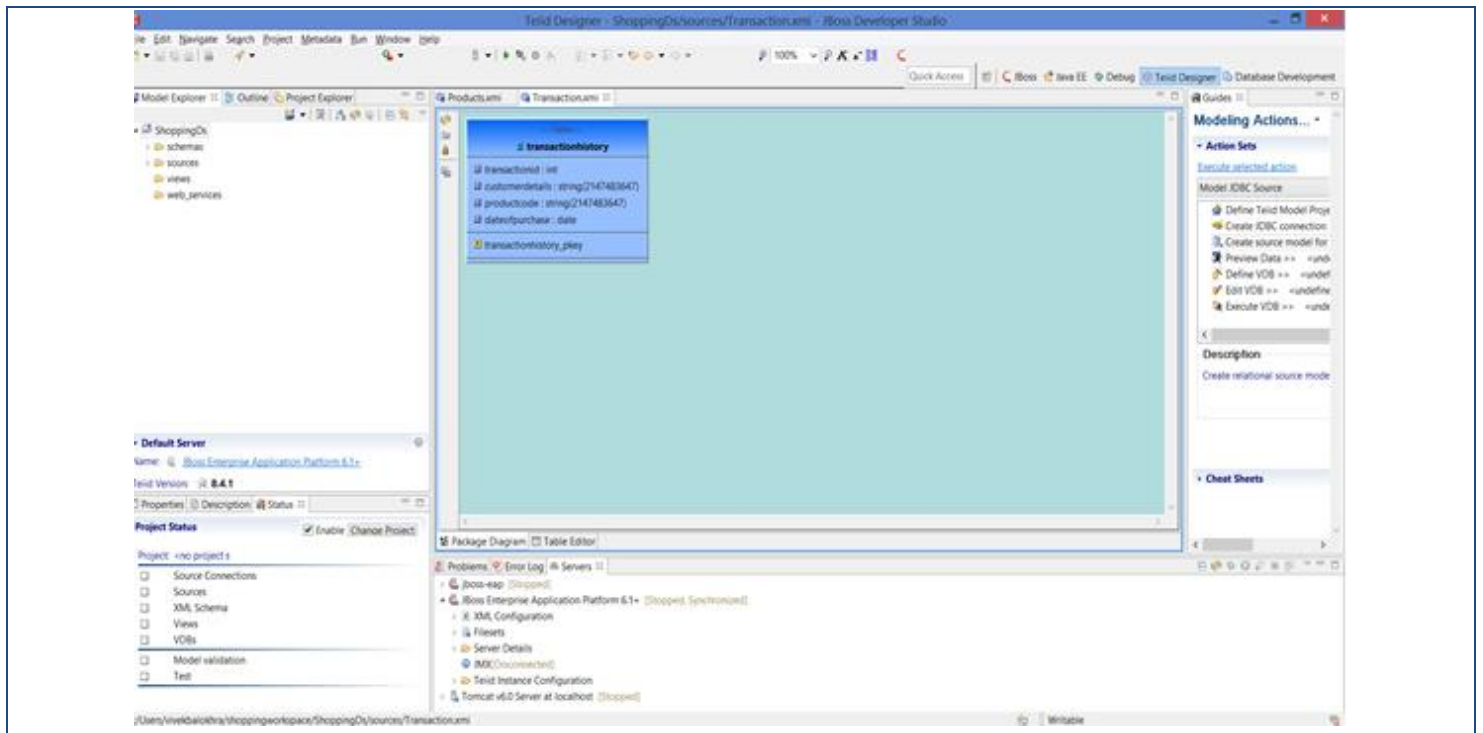


Figure 55. Viewing the source model of the Transaction database

Now that the underlying databases are configured, the next step in creating a virtual database is to go to the File Menu and create a new Teiid VDB.

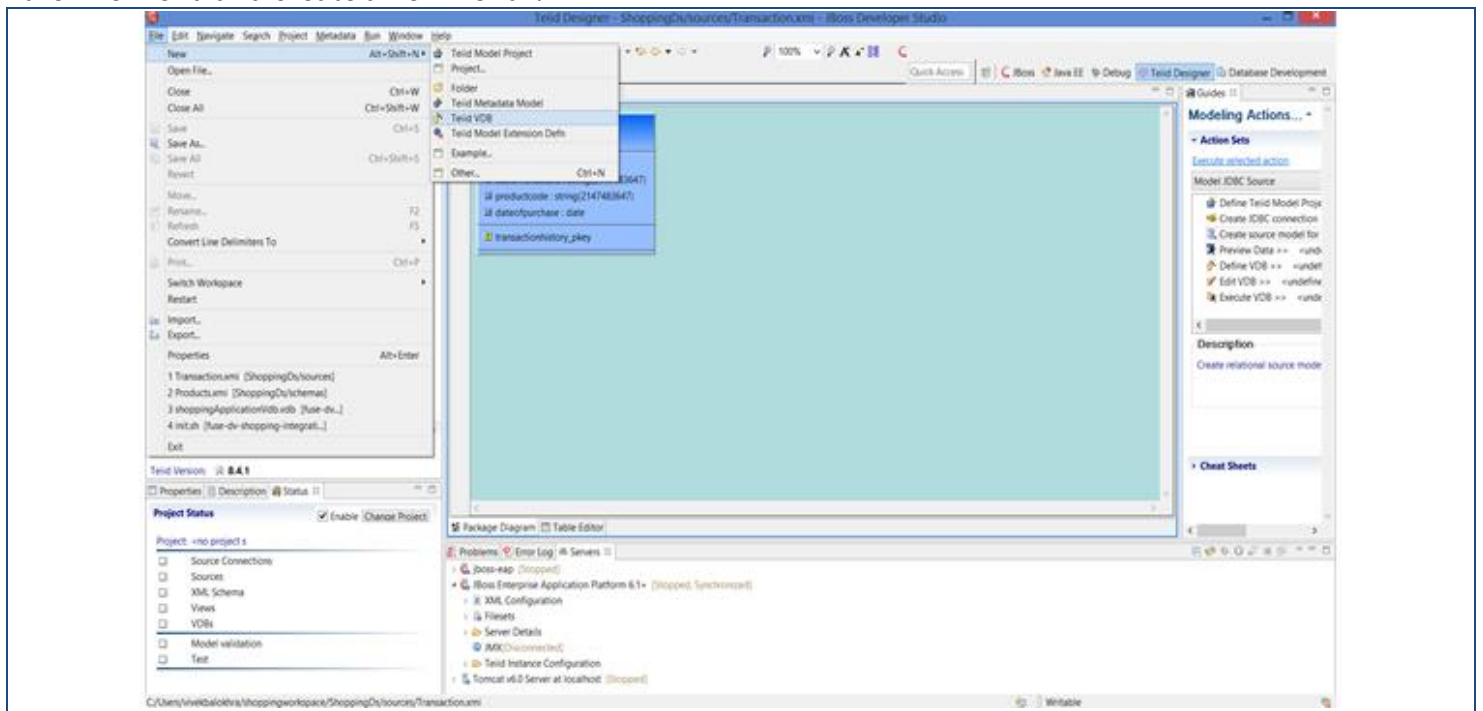


Figure 56. Creating a new Teiid VDB

The new VDB asks for models. We simply click on the Add button to add these to our VDB.

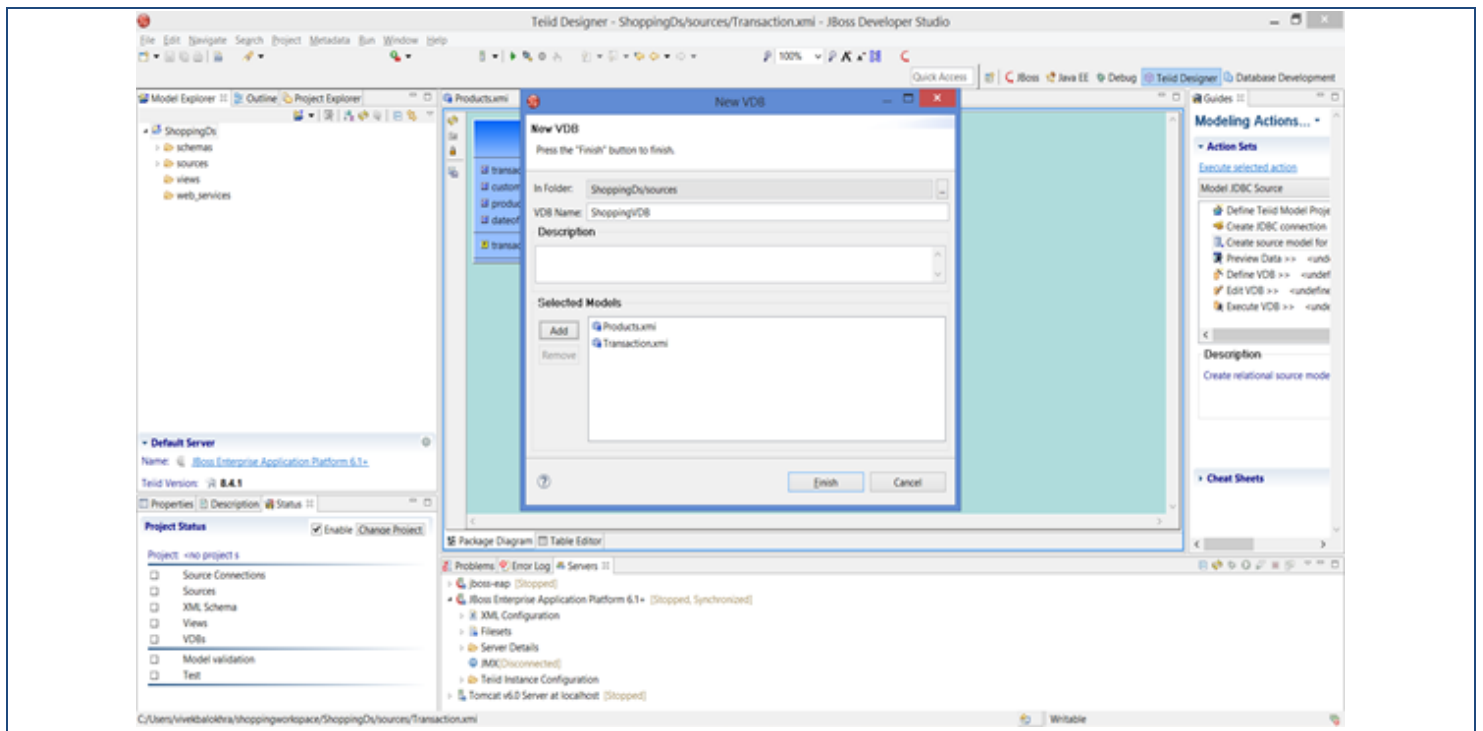


Figure 57. Adding the previously created models

The virtual database ShoppingVDB.vdb is now ready for use.

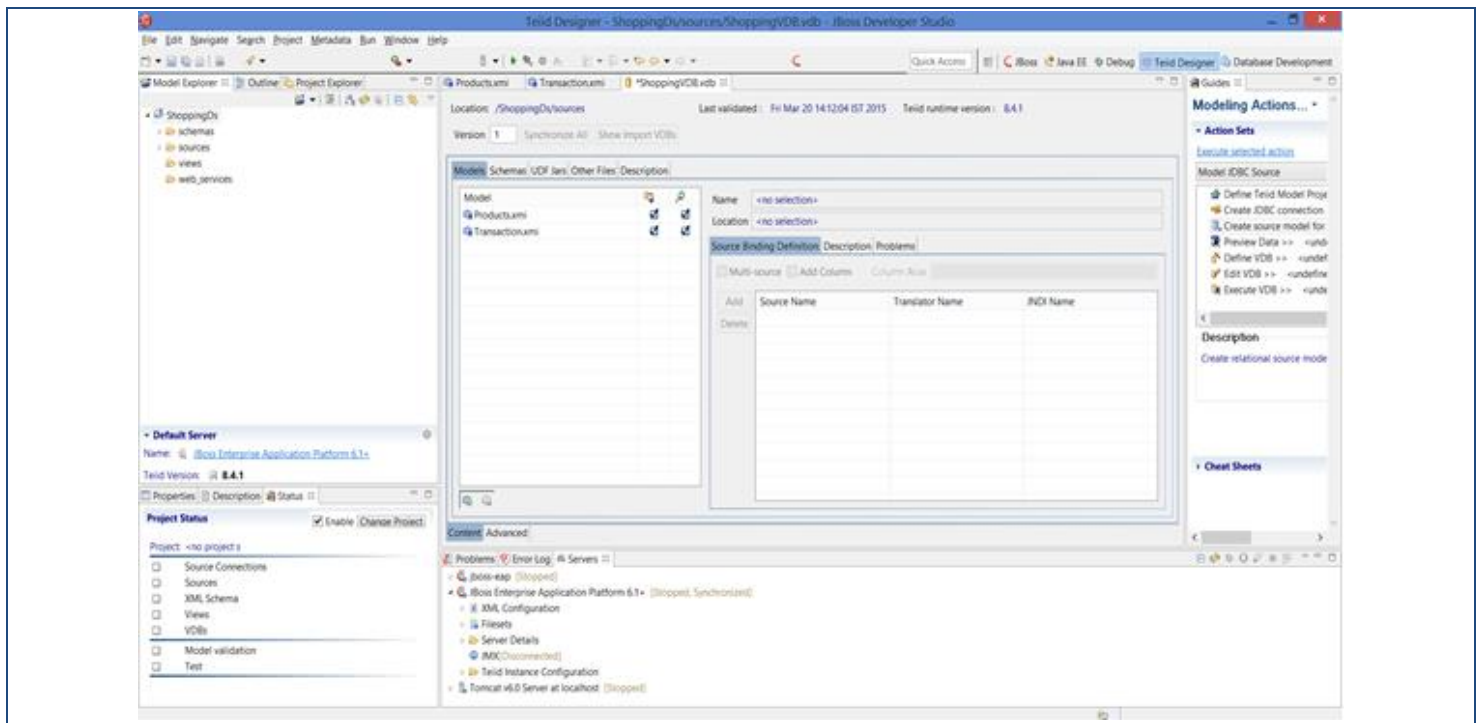


Figure 58. The newly created ShoppingVDB.vdb

To use this in the Shopping Cart Application, we need only configure the application to use the VDB as it would any other data source, using the Teiid JDBC driver:


```

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="${database.driverClassName}" />
    <property name="url" value="${database.url}" />
    <property name="username" value="${database.username}" />
    <property name="password" value="${database.password}" />
</bean>

```

```

com.redhat.application.cfg x
# Database Connection Properties
database.driverClassName=org.teiid.jdbc.TeiidDriver
database.url=jdbc:teiid:shoppingApplicationVdb@mm://localhost:31000;version=1
database.username=user
database.password=user

```

Figure 59. Data Source configuration in Shopping Cart Application

We also must configure JDV to point to both MySQL and Postgres:

```

<datasource jndi-name="java:/datasources/shoppingDS/mysql-ds" pool-name="mysqlDS" enabled="true" use-java-
context="true">
    <connection-url>jdbc:mysql://localhost:3306/fusesshoppingdemo</connection-url>
    <driver>mysql</driver>
    <security>
        <user-name>root</user-name>
        <password>root</password>
    </security>
</datasource>
<datasource jndi-name="java:/datasource/shoppingDS/postgres-ds" pool-name="PostgreDS">
    <connection-url>jdbc:postgresql://localhost:5432/postgres</connection-url>
    <driver>postgres</driver>
    <security>
        <user-name>postgres</user-name>
        <password>postgres</password>
    </security>
</datasource>

```

Figure 60. Data Source configuration in JDV

Now that this is configured, our application can expose the data as XML, ODATA, or JSON. These are shown in the screenshots below.

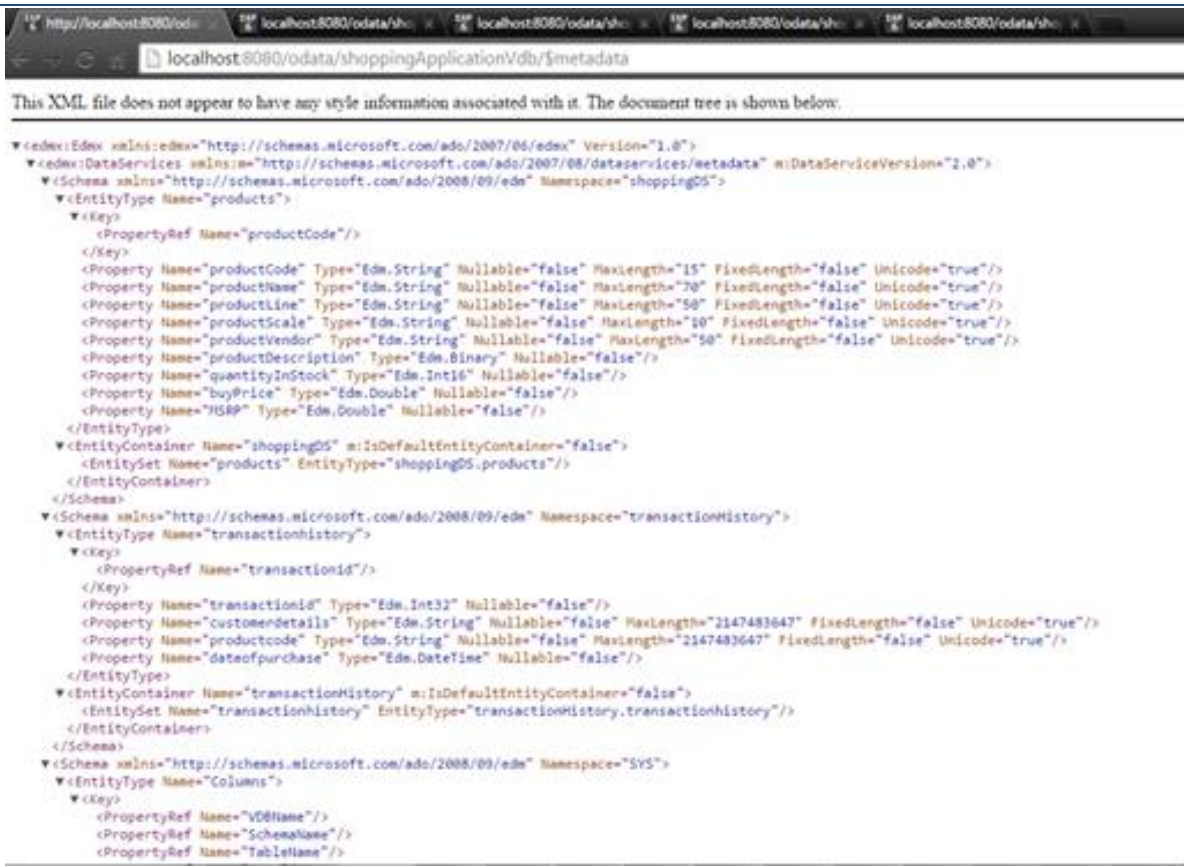


Figure 61. XML view of the Shopping Application metadata

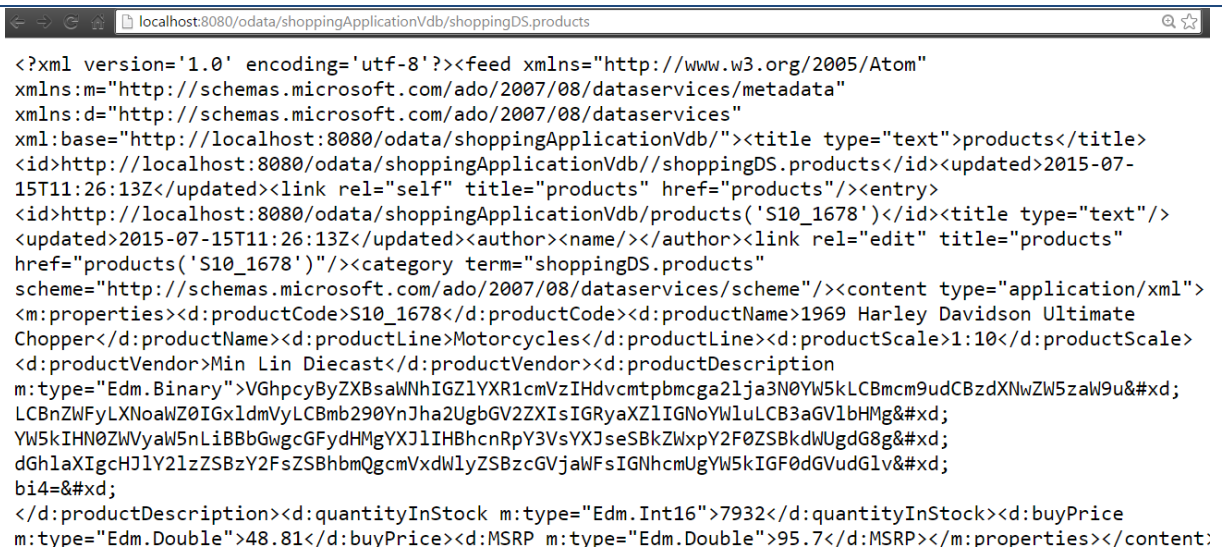
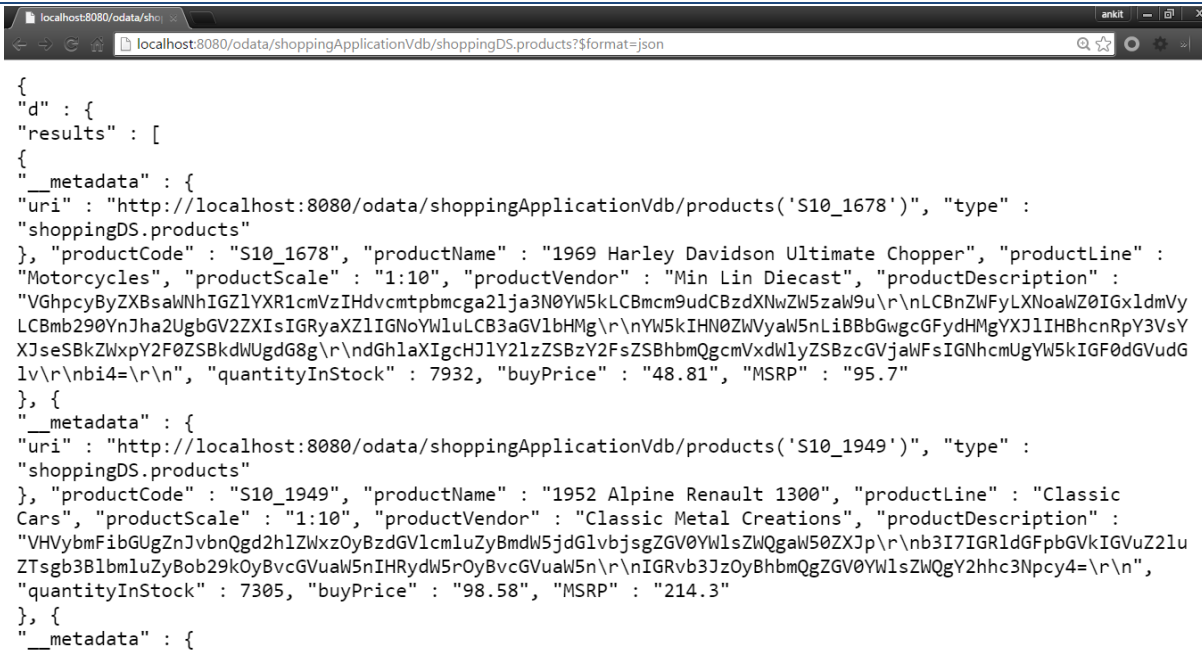


Figure 62. Products exposed as ODATA



```

{
  "d" : {
    "results" : [
      {
        "__metadata" : {
          "uri" : "http://localhost:8080/odata/shoppingApplicationVdb/products('S10_1678')", "type" :
            "shoppingDS.products"
        }, "productCode" : "S10_1678", "productName" : "1969 Harley Davidson Ultimate Chopper", "productLine" :
          "Motorcycles", "productScale" : "1:10", "productVendor" : "Min Lin Diecast", "productDescription" :
            "VGhpcyByZXBSaWZhIGZlYXR1cmVzIHdvcmtpbmcga2lja3N0YW5kLCBmcm9udCBzdXNwZW5zaW9u\r\nLCBnZWZyLXNoaWZ0IGxldmVy
            LCBmb290YnJha2UgbGV2ZXIsIGRyaXZlIGNoYWlucyB3aGVlbnHMg\r\nYXN0ZWN0ZWN0ZWN0LiBBbGwgcGFydHMgYXJlIHhcnRyY3VsY
            XJseSBkZWxpY2F0ZSBkdWUgdG8g\r\nndGhlaXJlcHJlY2l2ZSBzY2FsZSBhbmQgcmlvZGwlyZSBzcGVjaWFsIGNhcmUgYW5kIGF0dGVudG
            lv\r\nbi4=\r\n", "quantityInStock" : 7932, "buyPrice" : "48.81", "MSRP" : "95.7"
        }, {
          "__metadata" : {
            "uri" : "http://localhost:8080/odata/shoppingApplicationVdb/products('S10_1949')", "type" :
              "shoppingDS.products"
          }, "productCode" : "S10_1949", "productName" : "1952 Alpine Renault 1300", "productLine" : "Classic
            Cars", "productScale" : "1:10", "productVendor" : "Classic Metal Creations", "productDescription" :
              "VHVybmluZG9uZnJvbnQgd2h1ZWxzOyBzdGVlcmluZyBmdW5jdG1vbjsZGV0YWlsZWQgaw50ZXJp\r\nnb3I7IGRldGFpbGVkIGVuZ2lu
              ZTsgb3Blbm1uZyBob29kOyBvcGVuaW5nIHRydw5rOyBvcGVuaW5n\r\nnIGRvb3JzOyBhbmQgZGV0YWlsZWQgY2hhc3Npcy4=\r\n",
              "quantityInStock" : 7305, "buyPrice" : "98.58", "MSRP" : "214.3"
            }, {
              "__metadata" : {

```

Figure 63. Products exposed as JSON

10. Business intelligence

Leverage API platform to track and record every transaction for auditing and debugging. Enable real time monitoring to diagnose system performances and transaction issues.

Logging and Audit Tracing:

Logging and Audit Tracing Services permanently store application information in a data store that can be later accessed for operations management.

Typically, Logging and Audit Tracing Services are used to:

- Keep track of technical failures.
- Monitor the load of a server or the use of an application.
- Document the use of sensitive application or system privileges. (Creating new users, etc.)
- Store transaction and digital signature data for non-repudiation purposes.

In our Shopping Application we have used logging in various sections and have logged the events in our server log file located at jboss-fuse-6.1.0.redhat-379\data\log.

We have also created a class known as LogProductFetchTime which logs the time while showing products. This class is using the camel aop mechanism to fetch the time taken by the sql query. A screenshot of the class can be shown:

```
public class LogProductFetchTime {
    Logger log = Logger.getLogger(LogProductFetchTime.class);
    public void logQueryTimeAfter(){
        log.warn("After time =" + new Date().getTime());
    }
    public void logQueryTimeBefore(){
        log.warn("Before time =" + new Date().getTime());
    }
    public void logQueryTimeFinally(Exchange exchange){
        log.warn("Aop over product addition query ended");
    }
}
```

Figure 64. LogProductFetchTime

We have used log4j for logging and it can be configured by modifying the log4j.properties file. The log4j properties file is available in the resources folder of the project.

11.Red Hat Mobile Application Platform (FeedHenry) Integration:

The Red Hat Mobile Application Platform (FeedHenry) brings agility, visibility and efficiency to enterprise mobility. This mobile app platform embraces collaborative app development. The platform allows for easy integration of back-end APIs to develop and deploy native mobile applications. Building a mobile application on FeedHenry is out of the scope of an integration whitepaper. However, we felt it was important to demonstrate the ease of integration between Fuse and FeedHenry. For our purposes, we are using the FeedHenry environment to deploy an angular based application. This application is capable of consuming all the services that have been created in the shopping application. The screenshots below will demonstrate how FeedHenry is helping to create a frontend to display the products of shopping application on cloud.

```
myApp.service('ApiResources', function ($http) {  
    return {getProducts: function () {  
        return $http.get('http://170.248.0.194:8080/v1.shoppingApplica  
    }  
    };  
});
```

Figure 65.Angular Service Consuming API as a REST service

The front end application is created and we have created a service to retrieve all the products using the API. The service here is using the angular http module to make a get request on <http://localhost:8080/v1.shoppingApplication/services/rest/shoppingApplication/products> which returns all the products.

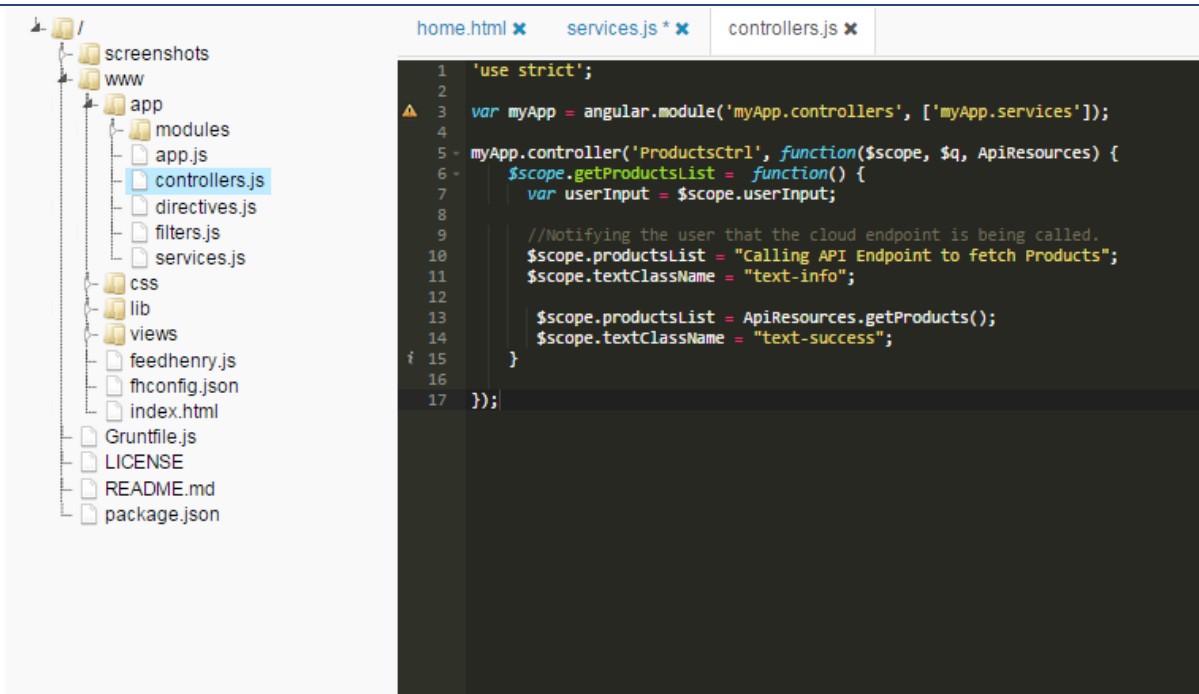


Figure 66. LogProductFetchTime

This service is used in a controller which maps the data retrieved to a model object.

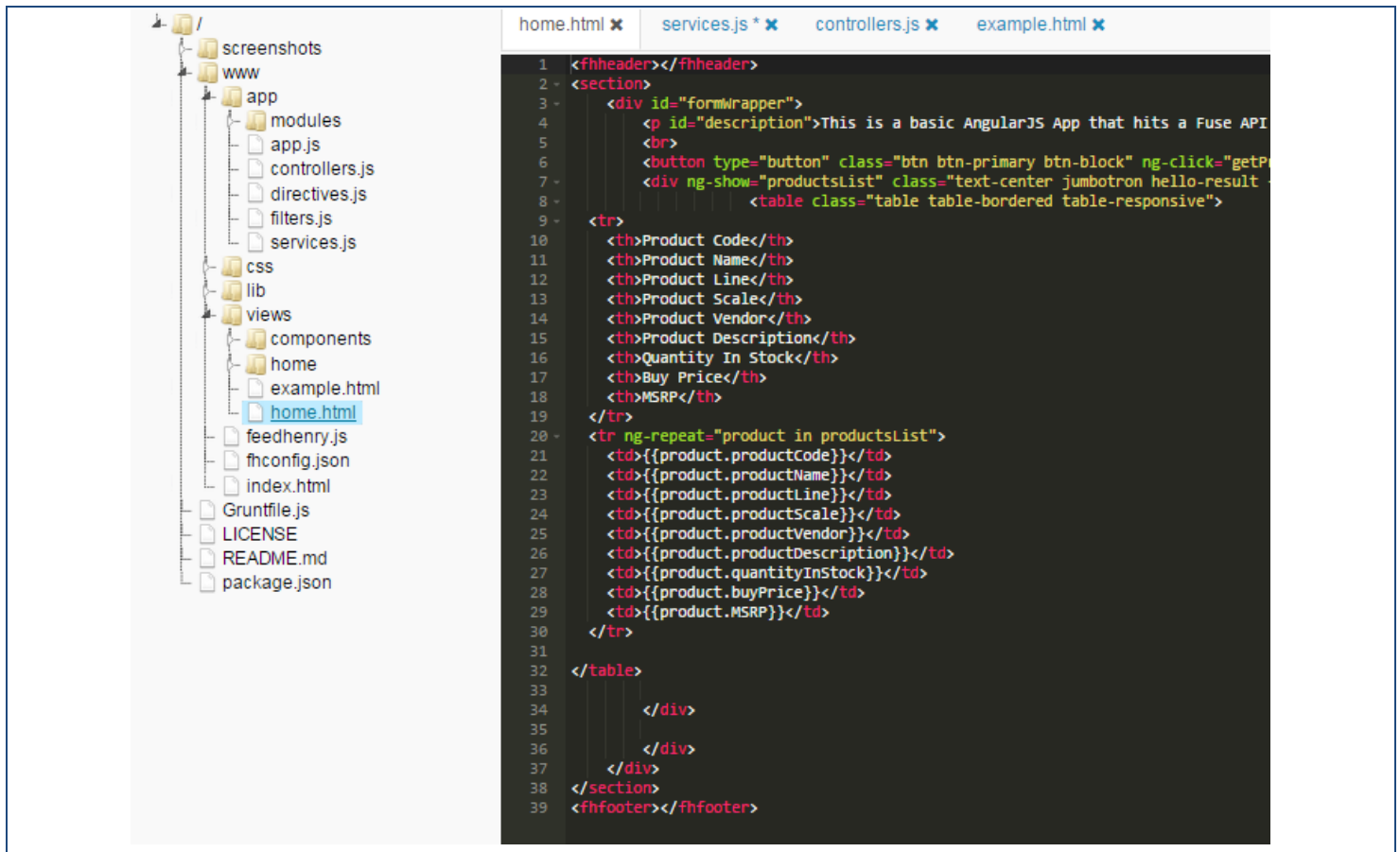


Figure 67. HTML file with angular directives to display products as a table

Finally the model is displayed on an html page.

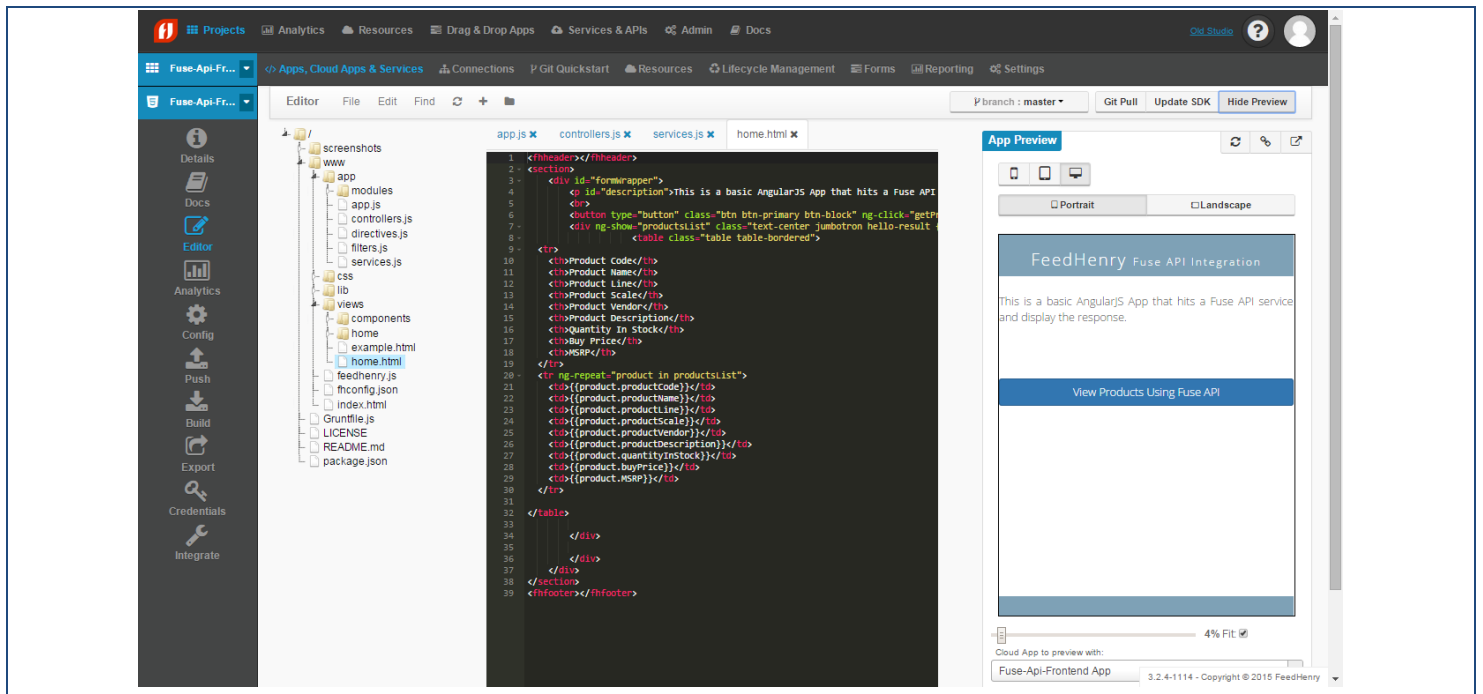


Figure 68. FeedHenry console with the quick view of the angular app

The results can be viewed using the feedhenry application preview panel. Above is the homepage which displays a button to view the products in the shopping application.

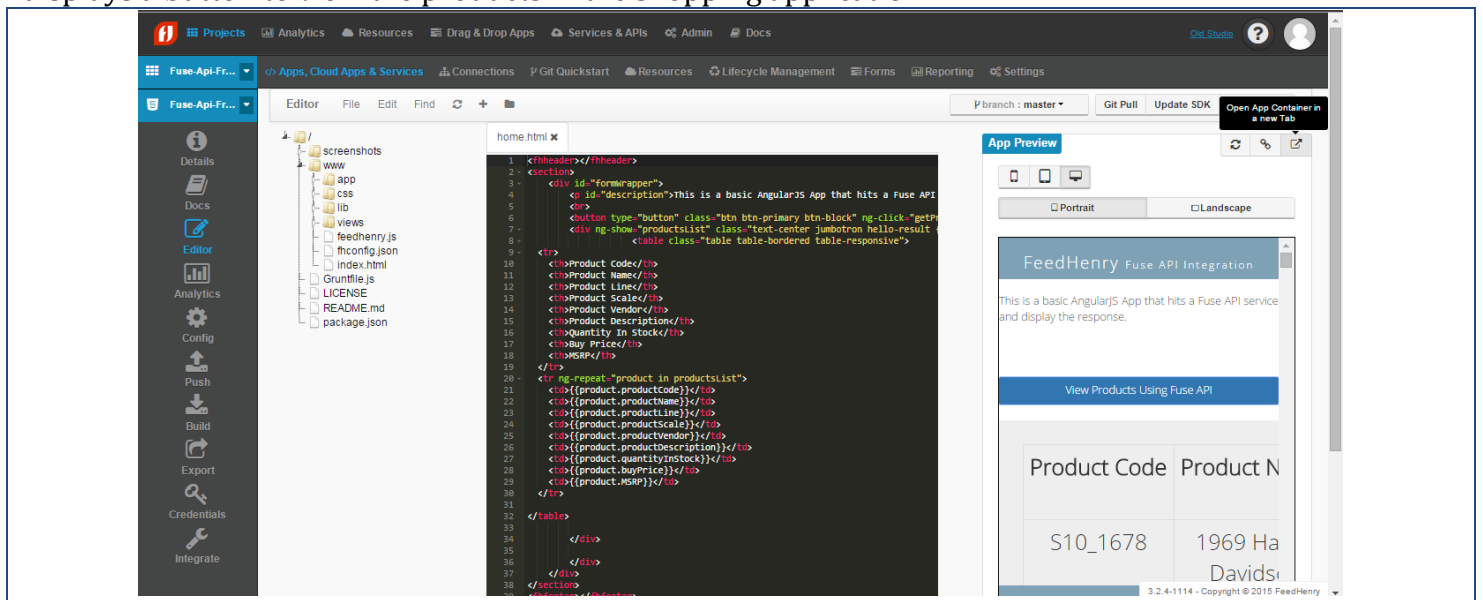


Figure 69. Products being displayed in the feedhenry console

Once the button is clicked the controller is invoked which uses the service mentioned above to populate data from the API.

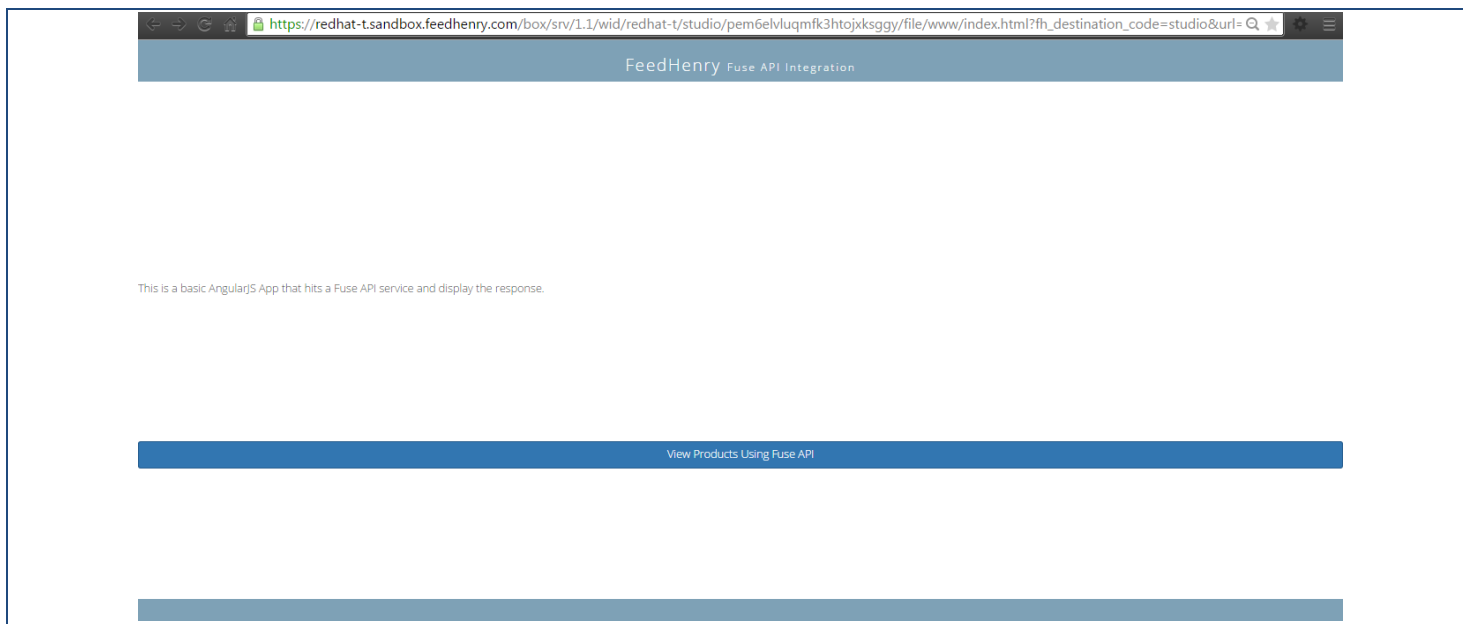


Figure 70. Full window view of the angular app

View Products Using Fuse API									
Product Code	Product Name	Product Line	Product Scale	Product Vendor	Product Description	Quantity In Stock	Buy Price	MSRP	
S10_1678	1969 Harley Davidson Ultimate Chopper	Motorcycles	1:10	Min Lin Diecast	This replica features working kickstand, front suspension, gear-shift lever, footbrake lever, drive chain, wheels and steering. All parts are particularly delicate due to their precise scale and require special care and attention.	7905	48.81	95.7	
S10_1949	1952 Alpine Renault 1300	Classic Cars	1:10	Classic Metal Creations	Turnable front wheels; steering function; detailed interior; detailed engine; opening hood; opening trunk; opening doors; and detailed chassis.	7305	98.58	214.3	
S10_2016	1996 Moto Guzzi 1100i	Motorcycles	1:10	Highway 66 Mini Classics	Official Moto Guzzi logos and insignias, saddle bags located on side of motorcycle, detailed engine, working steering, working suspension, two leather seats, luggage rack, dual exhaust pipes, small saddle bag located on handle bars, two-tone paint with chrome accents, superior die-cast detail, rotating wheels, working kick stand, diecast metal with plastic parts and baked enamel finish.	6625	68.99	118.94	
S10_4698	2003 Harley-Davidson Eagle Drag Bike	Motorcycles	1:10	Red Start Diecast	Model features, official Harley Davidson logos and insignias, detachable rear wheelie bar, heavy diecast metal with resin parts, authentic multi-color tampon-printed graphics, separate engine drive belts, free-turning front fork, rotating tires and rear racing slick, certificate of authenticity, detailed engine, display stand, precision diecast replica, baked enamel finish, 1:10 scale model, removable fender, seat and tank cover piece for displaying the superior detail of the v-twin engine	5582	91.02	193.66	
S10_4757	1972 Alfa Romeo GTA	Classic Cars	1:10	Motor City Art Classics	Features include: Turnable front wheels; steering function; detailed interior; detailed engine; opening hood; opening trunk; opening doors; and detailed chassis.	3252	85.68	136	
S10_4962	1962 LanciaA Delta 16V	Classic Cars	1:10	Second Gear Diecast	Features include: Turnable front wheels; steering function; detailed interior; detailed engine; opening hood; opening trunk; opening doors; and detailed chassis.	6791	103.42	147.74	
S12_1099	1968 Ford Mustang	Classic Cars	1:12	Autoart Studio Design	Hood, doors and trunk all open to reveal highly detailed interior features. Steering wheel actually turns the front wheels. Color dark green.	68	95.34	194.57	
S12_1108	2001 Ferrari Enzo	Classic Cars	1:12	Second Gear Diecast	Turnable front wheels; steering function; detailed interior; detailed engine; opening hood; opening trunk; opening doors; and detailed chassis.	3619	95.59	207.8	
S12_1666	1958 Setra Bus	Trucks and Buses	1:12	Welly Diecast	Model features 30 windows, skylights & glare resistant glass, working steering system, original logos	1579	77.9	136.67	

Figure 71. Products list displayed by angular app

The results can also be viewed in a separate window and not in the app preview panel. As you can see, pulling data from a REST API can be performed easily in FeedHenry, allowing for a simplified approach to mobile development.

12. Appendix

All of the steps required to deploy and run the application on Red Hat JBoss EAP and Red Hat JBoss Fuse are documented in Github. Browse to the repository located at <https://github.com/jbossdemocentral/fuse-dv-shopping-integration-demo>. The Readme.md file (<https://github.com/jbossdemocentral/fuse-dv-shopping-integration-demo/blob/master/README.md>) contains detailed instructions.

13. References

- ⁱ Shiro Security: <http://camel.apache.org/shiro-security.html>
- ⁱⁱ Spring Security: <http://camel.apache.org/spring-security.html>
- ⁱⁱⁱ XMLSecurity DataFormat: <http://camel.apache.org/xmlsecurity-dataformat.html>
- ^{iv} XML Security component: <http://camel.apache.org/xml-security-component.html>
- ^v Crypto DataFormat: <http://camel.apache.org/crypto.html>
- ^{vi} Crypto component: <http://camel.apache.org/crypto-digital-signatures.html>
- ^{vii} Jetty: <http://camel.apache.org/jetty.html>
- ^{viii} CXF: <http://camel.apache.org/cxf.html>
- ^{ix} Spring Web Services: <http://camel.apache.org/spring-web-services.html>
- ^x Netty: <http://camel.apache.org/netty.html>
- ^{xi} MINA: <http://camel.apache.org/mina.html>
- ^{xii} Cometd: <http://camel.apache.org/cometd.html>
- ^{xiii} JMS: <http://camel.apache.org/jms.html>
- ^{xiv} SoapUI: <http://www.soapui.org/>

Copyright © 2015 Accenture
All rights reserved.

Accenture, its logo, and
High Performance Delivered
are trademarks of Accenture.