|  |
| --- |
|  |
| SOEN 487 Project Report |
| Supply Chain Management Application |
|  |
| **Team #4**  **Yanan Chang**  **Sebastien Charbonneau**  **Yousra Saidani**  **Wei Wang** |
|  |

|  |
| --- |
|  |

**April 2018**

**Concordia University**

Table of Contents

[1. Introduction 2](#_Toc510295077)

[2. Project Directory Layout 2](#_Toc510295078)

[3. Deployment Architecture 4](#_Toc510295079)

[3.1. Global Architecture of the Project 4](#_Toc510295080)

[3.2. Manufacturer Service 4](#_Toc510295081)

[3.3. Warehouse Service 4](#_Toc510295082)

[3.4. Retailer Service 5](#_Toc510295083)

[3.5. Web Application Service 6](#_Toc510295084)

[4. Non-Functional Requirements 6](#_Toc510295085)

[4.1. Libraries 6](#_Toc510295086)

[4.2. Tools 6](#_Toc510295087)

[5. User Manual and Deployment 6](#_Toc510295088)

[5.1. Prior to Deployment 6](#_Toc510295089)

[5.2. Deployment Instructions 7](#_Toc510295090)

[6. Composite Application / BPEL 8](#_Toc510295091)

[7. Petri Net Specification 8](#_Toc510295092)

7.1 Analysis of Correctness ……………………………………………………………………………………………………………….. 9

7.2 Mapping ……………………………………………………………………………………………………………………………………… 9

# **Introduction**

This document details the architecture and deployment procedure for a Supply Chain Management application. It was developed as the course project for SOEN 487 – Web Services and Applications.

The application is composed of several SOAP and RESTful web services, which interact with each other and coordinate to process orders and manage an inventory for three types of consumer electronics (DVD players, video cameras and TVs) produced by four manufacturers (Brand1, Brand2, etc).

The web services were all developed in Java. The web service composition was done in BPEL using OpenESB.

# **Project Directory Layout**

The web services that compose the application are all separate Java projects, which is needed so that they can be deployed separately and interact with each other only via HTTP. However, in order to include them in the single git repository for the project, the projects were all placed in sub-folders of the original Java web application project for PM1, soen487-w18-team04.

The main folder, soen487-w18-team, contains the web service for PM1.1 and PM1.2, XMLReader. The following table lists the project sub-folders of soen487-w18-team and their contents.

Note: Naturally, sub-folders not specific to this project, such as .git and build were omitted from the list.

|  |  |
| --- | --- |
| Sub-folder | Contents |
| CustomerDB | REST web service for PM 2.1. |
| CustomerDB\_Client | Client application for CustomerDB. |
| doc | Project report (this document). |
| Manufacturer | SOAP web service for the manufacturer of Brand1 products. |
| Manufacturer2 | SOAP web service for the manufacturer of Brand1 products. |
| Manufacturer3 | SOAP web service for the manufacturer of Brand1 products. |
| Manufacturer4 | SOAP web service for the manufacturer of Brand1 products. |
| ManufacturerTestClient | A simple console-based text client designed to demonstrate the Manufacturer web service for PM1.3. |
| PM3 | HelloWorld BPEL application implemented for PM3.1. Sub-folder HelloWorld contains the BPEL module. Sub-folder HelloWorldCA contains the composite application. |
| PM3-2 | BPEL module for the first part of PM3.2, which invokes the XMLReader web service from PM1. |
| PM3-2\_CA | Composite application for the first part of PM3.2. Its BPEL module is in folder PM3-2. |
| PM3-2REST | BPEL module for the second part of PM3.2, which invokes a public REST service. |
| PM3-2REST\_CA | Composite application for the second part of PM3.2. Its BPEL module is in folder PM3-2REST. |
| RetailerSystem | Parent folder for the Retailer System. |
| RetailerSystem/RetailerCompositeApp | Composite application for the retailer system. |
| RetailerSystem/RetailerService | Web service for the Retailer System |
| RetailerSystem/WarehouseService1 | Web service for Warehouse 1. |
| RetailerSystem/WarehouseService2 | Web service for Warehouse 2. |
| RetailerSystem/WarehouseService3 | Web service for Warehouse 3. |
| RetailerSystem/WarehouseService4 | Web service for Warehouse 4. |
| RetailerSystem/WarehousesInvocationModule | BPEL module for the warehouses. |
| SupplyChainWebApp | Customer-facing client application for the Supply Chain Management Application. |
| SupplyChainWebService | A SOAP based web service for the web application. |
| TemperatureService | REST service implemented for PM2.2. |
| XMLReaderClient | A simple console-based text client designed to demonstrate the XMLReader web service for PM1.1 and 1.2. |

# **Deployment Architecture**

## 3.1. Global Architecture of the Project

The Supply Chain System follows a microservice architecture. In total, there are 4 microservices:

* Manufacturer service
* Warehouse service
* Retailer service
* Web application service

See figure 1 for an illustration of the application’s overall architecture.

## 3.2. Manufacturer Service

The Manufacturer Service can be found as a project **Manufacturer#** in **soen487-w18-team04** folder (where # is the number of the Manufacturer).

There are 4 manufacturers in total: Brand1, Brand2, Brand3 and Brand4. Each one has its own SOAP web service, which is used by the warehouse services to place orders for products.

The Manufacturer services are organized into 2 main parts:

* **soen487.g4.sc.manufacturer.business**, which contains the domain model classes for the Manufacturer: Product and PurchaseOrder.
* **soen487.g4.sc.manufacturer.service**, which contains the SOAP web service. It also has an InitializeInventory class which is used to randomly initialize the manufacturer's product unit prices and stock levels.

The data source for a Manufacturer web service is 2 XML files: product.xml and purchase-order.xml. product.xml stores information on the list of products produced by the manufacturer and purchase-order.xml keeps a history of the purchase orders that have been successfully processed.

## 3.3. Warehouse Service

The Warehouse Service can be found as a project **WarehouseService#** in **soen487-w18-team04/RetailerSystem** folder (where # is the number of the Warehouse).

There are 4 warehouses in total. Each warehouse offers a SOAP web service and a RESTful web service.

These services are used by a retailer to place an order (using SOAP web service) and to get information on the products available in the inventory of the warehouse (RESTful web service).

WarehouseService is organized into 3 main parts:

* **soen487.g4.sc.warehouse.service**, which contains both the SOAP web service and RESTful API.
* **soen487.g4.sc.warehouse.business**, which contains the domain model classes for Warehouse, like Customer, Item.
* **soen487.g4.sc.warehouse.util**, which contains helper classes that will be called by the service layer.

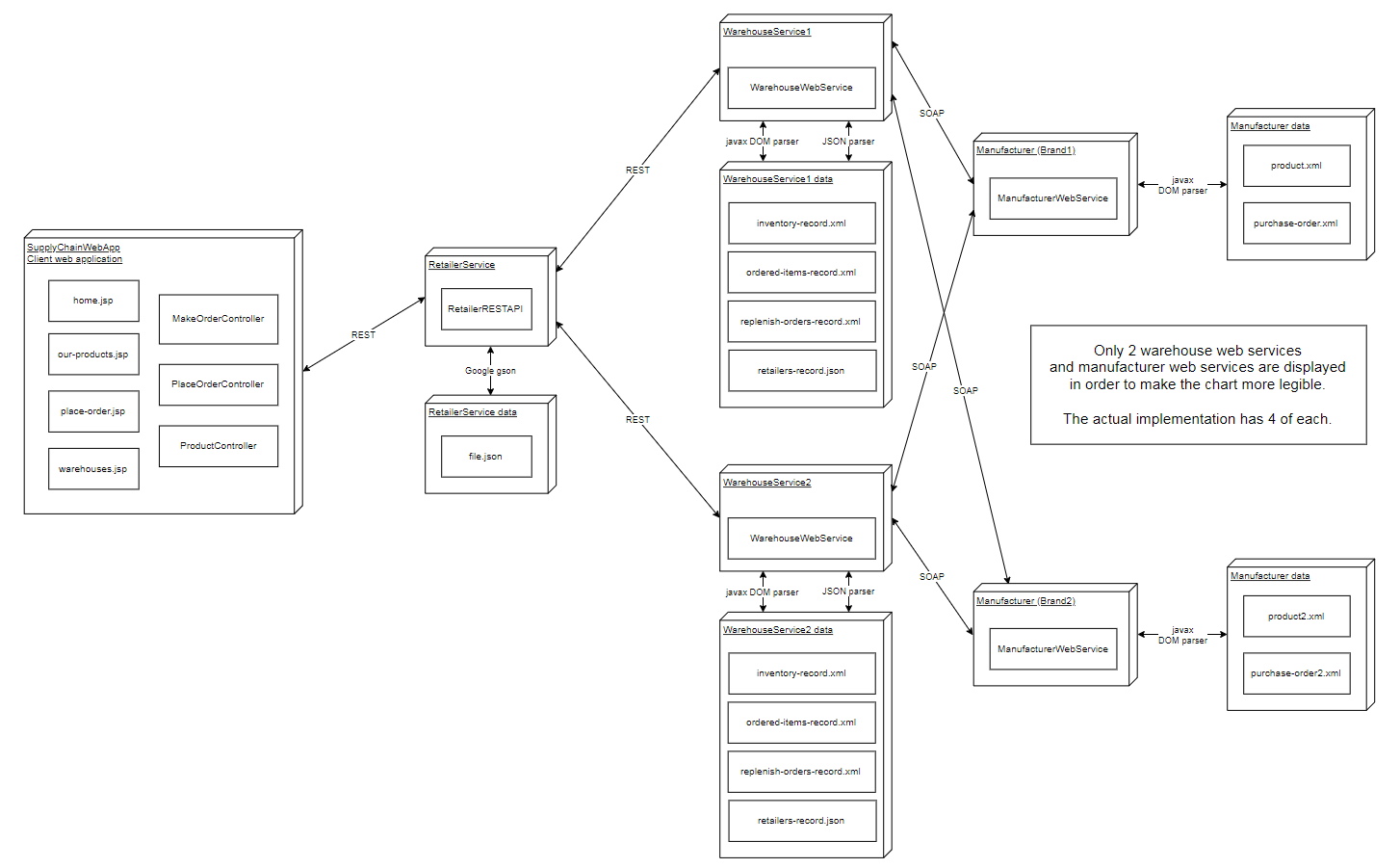


Figure 1: Application architecture (see Architecture.png for a larger version)

The database for Warehouse is XML and JSON documents. There are 4 database files: inventory-record.xml, ordered-items-record.xml, replenish-orders-record.xml and retailers.record.json.

## 3.4. Retailer Service

The Retailer Service can be found as a project **RetailerService** in **soen487-w18-team04/RetailerSystem** folder.

The Retailer communicates with multiple warehouses (4 warehouses) by calling their RESTful API through GET HTTP method. Also, it offers its own RESTful web service that will be used by the web client application.

When the retailer is deployed, it establishes a communication to all 4 warehouses to get their inventory of products (by calling their RESTful API through GET warehouses/#/products). Then, it combines all the products of all 4 warehouses into a catalogue of items based on predefined criteria (currently manufacturer name and product type) and saves this catalogue into its internal database (found in service.file.json).

## 3.5. Web Application Service

The Web Application Service can be found as a project **SupplyChainWebSerice** in **soen487-w18-team04** folder. It is used by the web application order to invoke the Retailer Composite Application. This service is a simple SOAP service that has one web service method placeOrder.

# **Non-Functional Requirements**

## 4.1. Libraries

The Warehouse service uses JSON.simple, an open source Java toolkit for encoding and decoding JSON text. json-simple-1.1.1.jar, which is included in the project’s git repository, must be added to the project’s libraries prior to deploying these services. In NetBeans, go to project Properties -> Libraries -> Compile time -> Add JAR/Folder, then select the jar file and click OK.

The Retailer services uses Google Gson, an open source Java library for serializing and deserializing Java objects to/from JSON. You’ll need to add gson-2.6.2.jar, which is included in the project’s git repository, to the project’s libraries as detailed above.

## 4.2. Tools

We used the OpenESB Studio 3.1.0 IDE for development, so the web services that compose the application are all Netbeans projects.

# **User Manual and Deployment**

## 5.1. Prior to Deployment

Since the web services use XML and JSON files as a data source, the path to those files needs to be modified in the concerned web service classes prior to deployment.

* For each Manufacturer# web service, the files product.xml and purchase-orders.xml must be copied onto the server, but outside the project folder (so they aren’t compiled during deployment). Then, adjust the ManufacturerWebService class’ PO\_FILE\_PATH and PRODUCT\_FILE\_PATH accordingly.
* For each WarehouseService#, the files inventory-record.xml, ordered-items-record.xml, replenish-orders-record.xml and retailers-record.json must be copied onto the server, but outside the project folder. Then adjust the WarehouseWebService class’ XMLPATH, PLACED\_ORDERS\_FILE, ORDERED\_ITEMS\_FILE and RETAILERS\_RECORDS\_FILE with the new path of the files.
* For the RetailerService, the file file.json must be copied onto the server, but outside the project folder. Then, adjust the path for the RetailerStarterRestApi class’ RETAILER\_CATALOG accordingly.
* For the SupplyChainWebApp, both gson and jaxrs-ri-2.26 libraries have to be added in order to access the Restful service of the retailer. Also, for the front-end, jstl-1.2.jar library has to be added. These dependencies can be found in soen487-w18-team04 fodler.

## 5.2. Deployment Instructions

First, start the Glassfish and OpenESB Standalone servers. Then, deploy the following services in order:

1. Deploy all 4 Manufacturers services .
2. Deploy all 4 Warehouses services.
3. Deploy the Retailer service.
4. Deploy the web application service (SupplyChainWebAppWebService).
5. Go to another IDE other than OpenESB (such as Eclipse) and start a Tomcat server on a port **other than 8080** (3000 for example). Then, open the project SupplyChainWebApp and start the web app Tomcat. Go to <http://localhost:3000/SupplyChainWebApp/home.jsp> to visit the homepage of the application. From there, choose which option you want to access (either browse the available products or place an order).
6. Build WarehousesInvocationModuleService (found as a project inside of RetailerSystem).
7. Build RetailerServiceCompositeApp (found as a project inside of RetailerSystem).

## 5.3. User Manual

# **Composite Application / BPEL**

The Retailer Service Composite Application (found as a project inside of RetailerSystem folder) invokes a BPEL process upon receiving a request form the web application[[1]](#footnote-1).

The BPEL process invokes in a sequential manner the four warehouses if necessary. Upon the reception of the placed order, the order will be handled first by Warehouse1, if the order could be fulfilled completely at Warehouse1, an item shipping list is created by Warehouse1 and returned as a reply to web service PlaceOrder. If the order could be partially completed, the uncomplete part will be transferred to the next warehouse for further handling, and the process logic repeats as in warehouse 1. To consult the BPEL process, open the file WarehousesInvocationModule\_2.bpel (found as file inside of RetailerSystem => WarehousesInvocationModuleService).

# **Petri Net Specification**

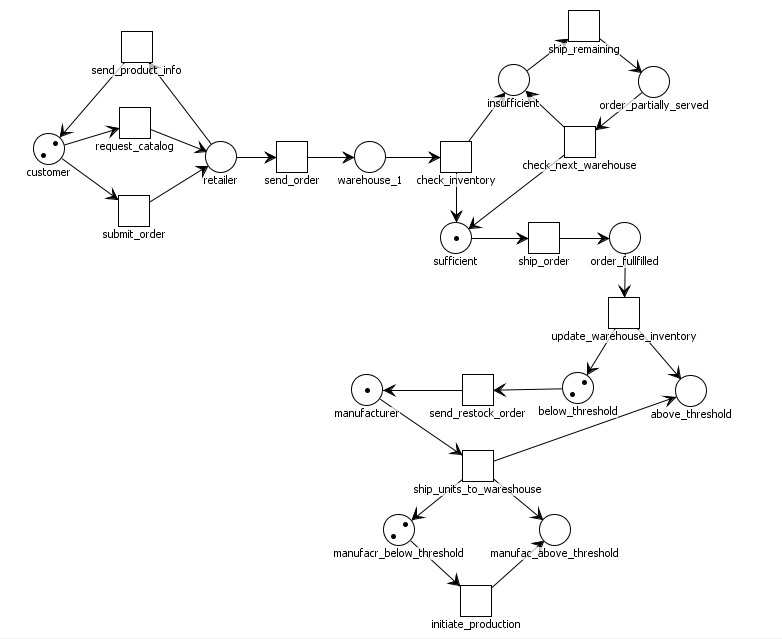


Figure 2: Business process modeling with Petri net

## Analysis of correctness

* Boundedness: It is an open net with an interface and a set Ω of final markings. In the interface, there is no input places; output places are: above\_threshold, manufac\_above\_threshold.

Final markings: [{order\_fullfilled}]

* Reachability: for every marking m, it is reachable since there is an occurrence sequence leading from m0 to m.
* Terminating: there is no infinite occurrence sequence, it terminates at [order\_fullfilled].
* Deadlock: for every reachable marking m, except final marking, it enables a transition. Therefore this Petri net is deadlock-free.

## Mapping

The business process is modeled by Petri net based on the order of operations between the components. The deployment architecture includes the following components:

* Objects: customer, retailer service, warehouse service and manufacturer service, which modeled by the corresponding places in the net: customer, retailer, warehouse\_1 (the first warehouse chosen to supply the order) and manufacturer.
* Retailer service: receives orders from customers and sends the order to warehouses, which modeled by transitions in the net: submit\_order, send\_order.
* Warehouse service: checks inventory and supplies partial of or fulfills the customer order, ship goods, and restocks inventory, which modeled by transitions in the net: check\_inventory, ship\_remaining, check\_next\_inventory, ship\_order, update\_warehouse\_inventory, send\_restock\_order.
* Manufacturer service: receives orders from warehouses and launches production, which modeled by transitions in the net: ship\_units\_to\_warehouse, initiate\_production.

1. When we linked the web application, we encountered a “8080 binding” problem when deploying the CASA app. It is when we deployed the web application on another port (3000 as described in step 5 in Deployment Instructions) that it deployed successfully. That’s why step 5 in Deployment Instructions has to be performed. Otherwise, the CASA will not deploy. [↑](#footnote-ref-1)