Enterprise Application Integration 2009/2010
Assignment #4 – Application Integration with Message Queues

Objectives

- To gain familiarity with the use of Message Queues for Enterprise Application Integration.

Due Date

- **18th December 2009**

Scheduled Effort

- **14 hours**, according to the following plan:
  - 4 hours → Reading and learning technologies (e.g. doing tutorials)
  - 10 hours → Coding and testing

**PLEASE WRITE DOWN THE EFFECTIVE NUMBER OF HOURS SPENT IN EACH TASK
THE REPORTED EFFORT SHOULD BE PER STUDENT!**

Final Delivery

- All source code for the project.
- A small report (5 pages max) about the implementation of the project. The report must specify the number of hours spent per student while working on the assignment. Please use PDF. I will not open any word documents.

Software

For this project you will need to use Message Queuing middleware. You can either use Microsoft's Message Queues (MSMQ), which are directly embedded in Windows XP; or Java Messaging System (JMS), if you are using Java. Either is acceptable.

If you decide to use Microsoft Message Queues and .NET, please refer to the notes and examples given during classes and which are available on the course website ("*[06] Middleware*"). Note that you will have to make sure that MSMQ is installed. Go to "*Add/Remove Programs* ► *Add/Remove Windows Components* ► *Message Queuing*".

If you decide to use JMS, many messaging engines are available. We recommend that you use one of the following:

- **GlashFish ESB**, part of the OpenESB project (https://open-esb.dev.java.net/), which you used on the previous assignment and includes JMS.
- **JBoss Application Server** (http://labs.jboss.com/jbossas/downloads/), with the necessary JMS components, which also readily supports the Java Messaging System.
- **JORAM – Java Open Reliable Asynchronous Messaging** (http://joram.objectweb.org/). This is an open source JMS server that you can use in your projects.

Bibliography

The book “Enterprise Integration Patterns”, used in this course, describes the JMS API as well as MSMQ. In particular, chapters 4, 5, 6, 9 and 10 are related to this subject. (Although it may appear that 5 chapters is a lot, actually they are quite small! **Reading them is highly recommended.**)

If you are using JORAM, its webpage has several pointers to JMS tutorials. It also includes numerous examples.

If you are using Sun’s Application Server/GlassFish, a good starting point is the “JavaEE 5 Tutorial” (http://java.sun.com/javaee/5/docs/tutorial/doc/):

- Chapter 32: The Java Message Service API
- Chapter 33: Java EE Examples Using the JMS API

Note that even if you are using Sun’s Application Server, you are not required to develop Messaging Beans or deploy applications inside the application server. For this assignment, you may develop **simple stand-alone applications**.

Introduction

During the first assignment of this course you have created three applications that processed data from DPreview:

- **CameraSearchXML**, which using a query string representing a brand looked for corresponding cameras generating an XML file.
- **CameraSummaryXML**, which given the XML file previously generated, created another XML file with a summary of cameras available organized by brand.
- **CameraListBeautifier**, which given the XML file generated by *CameraSearchXML* created an HTML file (either stand-alone or directly rendered by a web browser).

The objective of this assignment is quite simple. You have to integrate the **three** applications using messaging.

The Assignment

The next image illustrates the Enterprise Service Bus (ESB) that you will be creating.

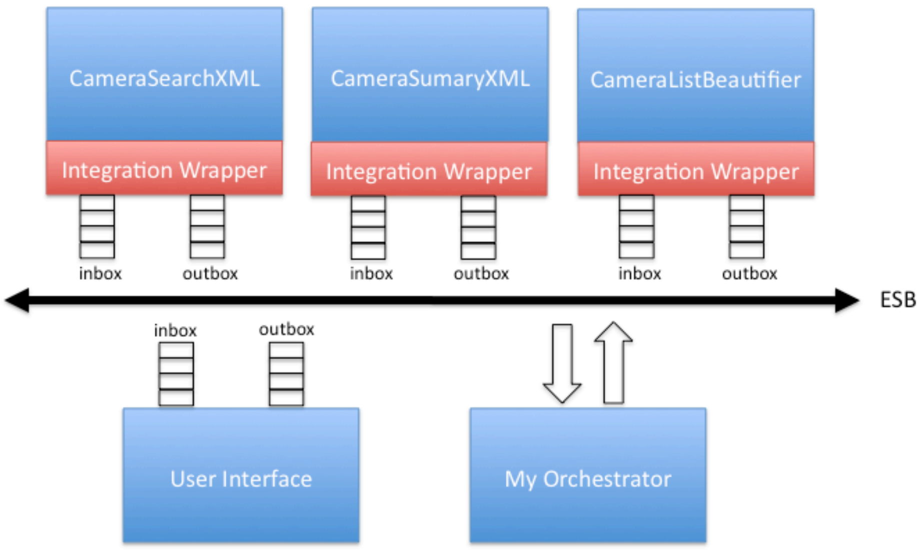

Figure 1 – Overall Architecture

All applications have an integration wrapper that either receives or sends messages though two privately owned messages queues (**inbox** and **outbox**). Messages queues are independent per application, being applications ignorant of all other. **MyOrchestrator** coordinates the overall operation, being the only application that knows about all others, and their messages queues.

Overall, it works like this:

- The **User Interface** waits for the user to introduce a query string using the keyboard. This corresponds to a camera brand. When the string is introduced, it puts a message on its *outbox*.
- The **MyOrchestrator** process reacts by reading the message from the User Interface outbox queue and passing it to the inbox of the **CameraSearchXML** application.
- The **CamaeraSearchXML** performs the query and creates an XML file. The XML file is put, as a message, on its outbox queue. (Note that the XML data must be transmitted through the network. You shouldn't just send a file name!)
- The **MyOrchestrator** application reacts by reading the message from the CameraSearchXML queue and passing it to the inbox of the **CameraSummaryXML** application and the **CameraListBeautifier** application. I.e., two messages are sent roughly at the same time.
- The **CameraSummaryXML** reacts by summarizing the information, producing a new XML file which is put on its outbox queue.
- The **CameraListBeautifier** application reacts by creating a "nice" HTML file containing the result of the search and putting it on its outbox queue. (Note: if in the first assignment you only rendered the XML on a browser you now need to generate a physical HTML file. Information on how to do this is available on the slides from classes: *[02] XSLT*.)
- The **MyOrchestrator** application reacts by reading the messages from the CameraSummaryXML outbox queue and the CameraListBeautifier outbox. When both messages are available, it creates a single message which is sent to the **User Interface** inbox queue. (Note: be careful not to assume a particular order in with the messages from CameraSummaryXML and CameraListBeautifier are produced.)
- The **User Interface** reacts by saving the file to disk using a unique name. The name of the file should be written to screen.

Note the following points:

- The User Interface will need at least two threads since it has to read data from the keyboard (query string) and from its inbox.
- Each new query string should correspond to a new request. Thus, each request should be uniquely identified and the identifier passed along with the messages.
- A new request can be entered even if the previous one has not completed, although the individual applications may only process a request at a time. Your system should not get "confused" with the concurrency!
- MyOrchestrator is an integration process. Thus, it should not be synchronously dependent of any particular application in the processing chain. For instance, if a request is being processed at the CameraSummaryXML, and a new message is put on the *outbox* of the User Interface, it should immediately read it and process it.
- The XML generated by *CameraSearchXML* and *CameraSummaryXML* should travel though the messages queues. (You shouldn't simply assume that all applications are running on the same machine!)
- You probably shouldn't hard-code the names of the message queues in the code, but put them on a configuration file. ☺

Valued feature (if you have time):

- Using transactions on the message queues, guarantying that messages are not lost and all requests are completely processed or aborted.