White Paper: Agile Software Development of Service Oriented Architectures, Business Process Models and Enterprise Service Buses

By Dr Ant Kutschera and Rowan Mountford
September 2008

A modern IT environment in the enterprise makes use of many systems, such as an Enterprise Service Bus (ESB), Business Process Modelling (BPM) Engine and a Java Enterprise Edition (Java EE) Application Server (AS). These systems would typically connect with anything like an Email Server, FTP Server, File Server, LDAP Server, Databases or other systems such as SAP or legacy systems. Inside the systems, technologies like Service Oriented Architecture (SOA), Enterprise Java Beans (EJB), Web Services (WS), Object Relational Mapping (ORM), Model View Controller (MVC), Inversion of Control (IoC) and Dependency Injection (DI), and many others are common place.

Agile software development is the process by which a system is produced quickly and efficiently by means of frequent and complete releases allowing the stakeholders in the project to get their hands on the system in order to test and review it. A prototype or proof of concept can be developed into a useful productive system by means of an iterative and incremental process whereby feedback is given by the stakeholders, based upon the rapid successive releases of the software.

However, agile development of applications in an enterprise environment as described above can prove difficult, because of the complex nature of the environment. For example, just to get the environment in place before software coding actually begins can be technically and politically hard as well as time consuming. Such time consuming (and hence budget consuming) tasks can kill an agile development project before it has even taken off.

This paper is split into three sections. The first puts forward an agile development platform and gives an example tying all the systems in the proposed platform together. The second section discusses what the authors discovered during the implementation of the example as well as problems and solutions in agile development in such environments. Finally recommendations are made in the third section.

Section I: The Platform and Example

The Platform

This section describes a software stack conducive to agile development in the enterprise.

Open source software offers two advantages over traditionally licensed software which are useful to agile development: simple and free licences making it quick to start using the software (no need to go through the purchasing department of your company); availability of source code which can help in debugging problems encountered during software development of dependent applications. One problem with open source is that it may not be supported for longer periods into the future, or that it is not even supported at the time of use. By looking for software that is backed by companies providing consulting and support services, this risk is minimised. It is precisely this which has made Linux (the open source operating system) so successful in the server marketplace.

For these reasons the platform is based on open source software that is backed by consulting/support service providers. The selection of the software is also based on the authors own experiences of open source systems.

Modern enterprise architectures might include any of the following, amongst others:

- Web Server
- Application Server (Java EE)
- Messaging Engine (JMS)
- IoC / DI Framework (Spring, EJB 3)
- ORM (Hibernate)
- Web Services
- MVC Framework for GUIs (Struts)
- BPM Server / Management
- Databases
- Directory Server (LDAP)
- Mail Server (Remote access through a firewall)
- File Server
- FTP Server (Remote access through a firewall or VPN)
- SMS Mobile Network Notification
- Monitoring Systems
- Reporting and Statistics

Additionally such architectures might have the requirement for:

- Remote access via desktops or laptops
- Access to business partners
- Mobile access
- Local web access
- Local access via rich applications

Taking all these requirements into account and selecting software based on supported open source systems, the proposed agile development platform looks as follows.

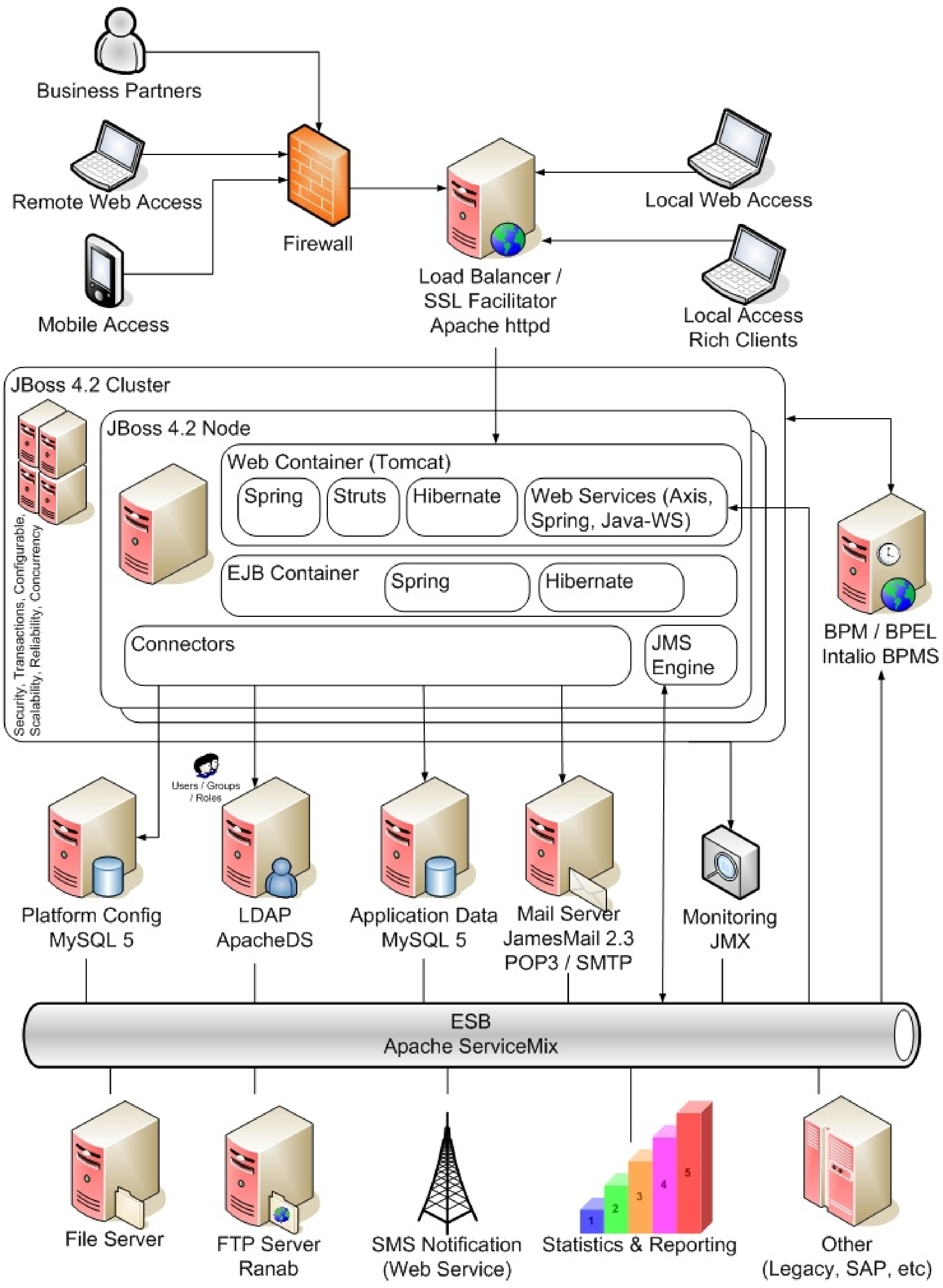


Figure 1: The Proposed Platform

Note that the above platform is considerably extensive – it includes many systems and sub-systems. However, not all need to be used whilst developing an application. Only those relevant to the architecture of the desired application need be selected.

There are also some caveats that come with the diagram:

- The BPM Server from Intalio could be integrated into the JBoss Application Server, but due to licence restrictions of the Community Edition (free), it may not be. The Community Edition actually runs inside the Apache Geronimo Java EE Application Server, however this server was not the Application Server of choice due to the better support / consulting services provided by JBoss and the fact that it is easier to find a JBoss expert than it is to find a Geronimo expert – part of the aim of this architecture is to allow a prototype to be grown into a full production system.
- The FTP Server may need to be accessed by Business Partners – this can be made available to them over a VPN or, less securely, through a firewall.
- The Mail Server will probably need access to the outside world. As such it would be granted access through the firewall.
- SMS Notification can be provided by subscription to an external Web Service provided by partner companies (search Google for such companies).

Additionally, if required, the Apache httpd Web Server can be used to provide secure connections over SSL to multiple domains. It then connects to the Tomcat Web Container in JBoss via an AJP connector. Apache httpd can also be used for load balancing over several instances of the web container in a cluster.

Architectural Layers

The platform encourages the use of architectural layers according to the following diagram.

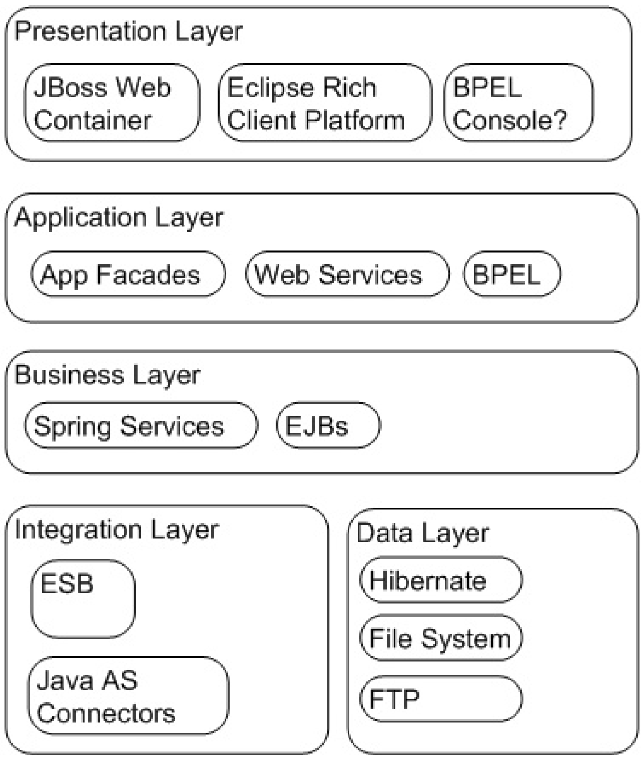


Figure 2: Suggested Layers in the platform

The presentation layer provides the end users with views into the system. In the example provided below, this was in the form of a web application. Alternatives are rich clients which communicate with the platform using remote services. The BPEL Console is listed in this layer, because it can provide (trained) business users with information on business process instances which they have interacted with, although this is unlikely to be done in reality, because of the complexity of the console. The console is really intended as a tool for use by support teams.

The application layer provides a place for each application to have a façade (or façades) into the business layer. These façades can provide the transactional demarcation required by the application as well as bring together the services from the business layer which they rely on. This layer would also include mappings which each application must do in order to use more generic business services. As such, web services called by the BPEL Engine would live here, as would the BPEL itself.

The business layer provides a place for hard core business services to exist. These could be implemented using technologies like Spring or EJB.

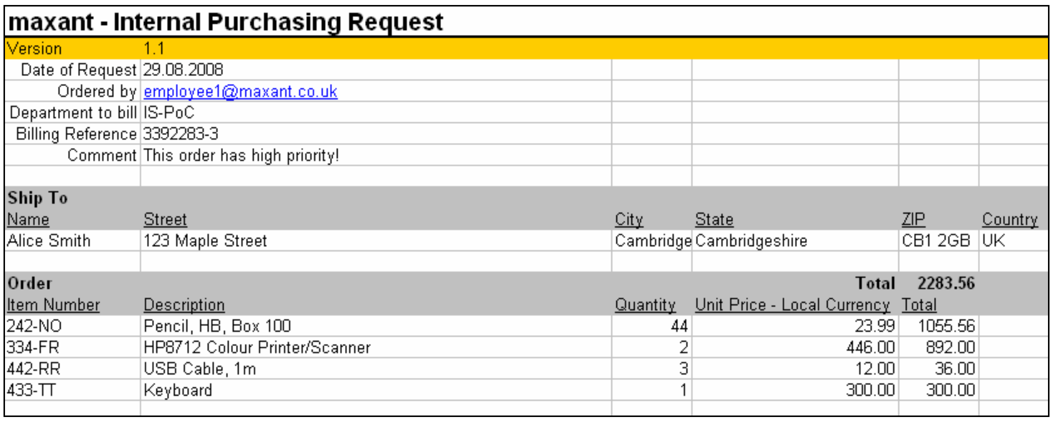
The integration layer defines services and components which integrate the platform with external or low level systems, like those from business partners or application databases. Typically anything which the ESB connects to is in this layer.

Finally, the data layer is the place where object relational mapping (ORM) components live. Additionally, connections to the file system or FTP could be placed here, although they could equally be a part of the integration layer.

The Example

To test the platform, a realistic business case has been thought up and implemented using the platform. The business case is as follows:

> *The maxant corporation has just acquired a global software supplier and would like to roll out its internal purchasing process to all offices. The internal purchasing process is one whereby employees can create orders to purchase equipment and hardware that they need for their daily jobs. Each staff member is given a credit rating, and can purchase up to that credit rating. Purchase orders are submitted as a spreadsheet to a central file server where they are processed manually by the purchasing department who performs a credit check and approves large orders. Very large orders are approved only by the VP of purchasing. Due to the acquisition, these purchase orders may now be submitted in any currency (but the purchasing department works in Swiss Francs). Furthermore, due to the acquisition, there now exists the need (and hence budget) to partly automate this process. Employees can use FTP over the VPN to submit orders in spreadsheet format to the central server. Email should be used as the notification mechanism. Web access should be used for approving or declining purchase orders. The project should be completed as soon as possible and rolled out globally. After a year, its success will be measured and if proven, a second budget may be made available to extend it, such that orders can be entered using a web form, as opposed to the existing spreadsheet form. Since this is the first IT project being run after the acquisition, maxant would like to prove the technologies and business process by using an agile development methodology. The aim is to work closely with the stake holders in order to give them constant feedback and confidence that the project will deliver. As such, this project is highly visible to senior management within the maxant corporation.*



maxant - Internal Purchasing Request

Version	1.1				
Date of Request	29.08.2008				
Ordered by	employee1@maxant.co.uk				
Department to bill	IS-PoC				
Billing Reference	3392283-3				
Comment	This order has high priority!				
Ship To					
Name	Street	City	State	ZIP	Country
Alice Smith	123 Maple Street	Cambridge	Cambridgeshire	CB1 2GB	UK
Order			**Total**	**2283.56**	
Item Number	Description	Quantity	Unit Price - Local Currency	Total	
242-NO	Pencil, HB, Box 100	44	23.99	1055.56	
334-FR	HP8712 Colour Printer/Scanner	2	446.00	892.00	
442-RR	USB Cable, 1m	3	12.00	36.00	
433-TT	Keyboard	1	300.00	300.00	

Figure 3: An example of the Purchase Order Spreadsheet

Taking these requirements, business analysts and architects would be able to formulate the following technical requirements (in visual form):

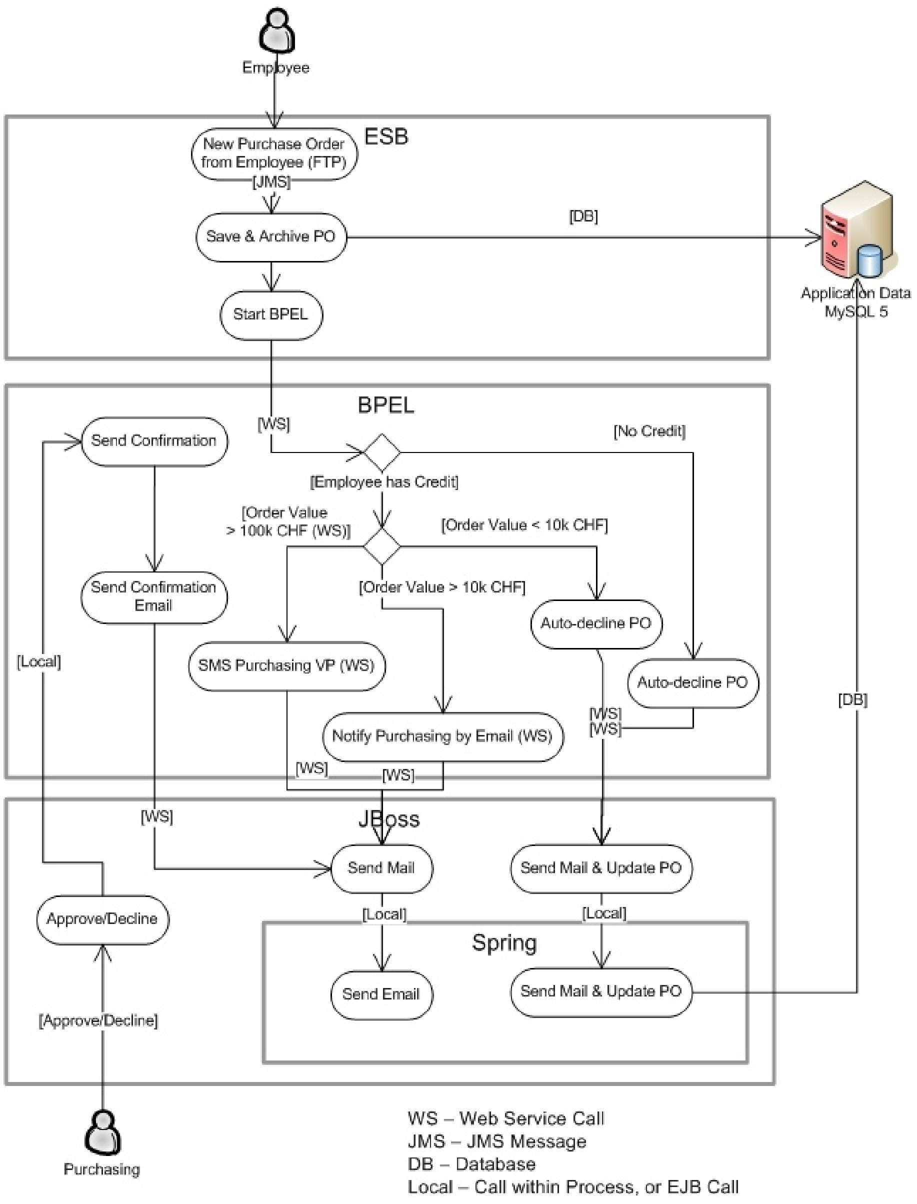


Figure 4: Pseudo Activity Diagram

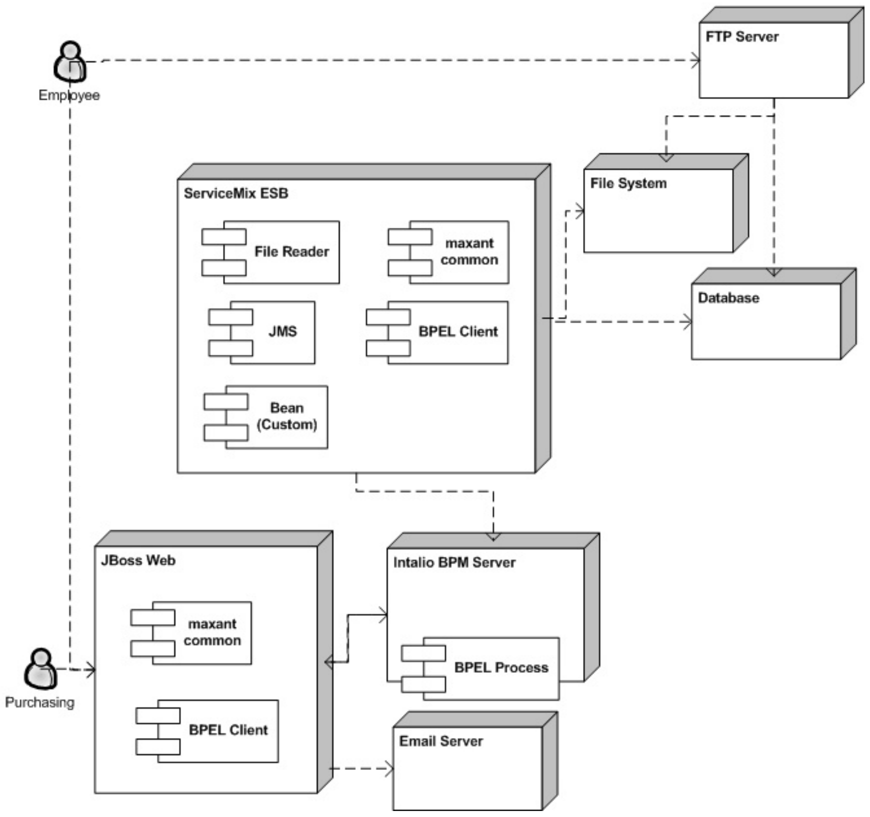


Figure 5: Deployment Diagram, showing dependencies

The activity diagram shows an ESB, BPEL Engine and a JBoss Application Server using Spring. The reasons that these Technologies were chosen were as follows.

- Endpoints such as polling for file inputs are typical for an ESB. Furthermore, when reading for files it is typical to process them in two stages. The first is to read and archive the file, ensuring its raw contents has been captured in the system. The second is to then validate and send the data for further processing. An ESB with a JMS core is ideal in this situation, because the JMS engine is an architecturally nice place for the raw data to await further processing – it simply waits there temporarily until a subscriber comes along and is read to use it. If required, the processing component can be scaled up by adding more instances, each of which reads from the JMS queue, which automatically load balances the incoming files. In order to parse the spreadsheets, the Apache POI library was used.
- BPEL was selected to model the high level business process since that process was very clear. Furthermore, maxant corporation has a software selection policy in place which dictates that where possible, business processes should be implemented in BPEL and designed using BPM Notation (BPMN) so that they are uniformly documented and maintainable. Additionally, the order size which dictates whether it is large or very large needs to be able to change fairly quickly. Ideally perhaps a rules engine might be used, however, these are fairly complex for the agile development environment in question. A competent Business Analyst could reset these thresholds for order size by editing the BPMN and getting it redeployed, without the need for a developer to be involved – exactly the same reason for using a rules engine.
- Spring was selected for building services so that they could be built in a common library and deployed to both the ESB as well as JBoss. As such, there was then no need to use EJB for the services (see later). Each client of these common services then configures Spring depending upon its relevant transaction manager and environment. Additionally, the ServiceMix ESB is based upon Spring and Spring-

like configurations, so adding our own Spring Beans into ServiceMix components was very easy.

Note that other architectures and designs could equally have been chosen. See later in this paper for a discussion over the use of an ESB and BPEL in this example.

The BPEL was implemented using the Intalio Designer. The resulting BPMN is shown on the following page.

While the BPMN may look complicated to untrained eyes, the process is relatively simple. The upper "pool" (the layer shown at the top) describes the entry points into the business process. The one on the left is the starting point for the process, called once the spreadsheet is captured by the ESB. The interface on the right is the entry point after the process is put on hold awaiting approval from purchasing (when the instance is re-hydrated).

The middle pool shows the business process as it is executed in the BPEL engine. This example is made up of tasks (all of which make calls to existing web services in JBoss) and decision points (has the employee got credit, is the order large, is the order very large). If the process is put on hold awaiting approval, it is dehydrated by the BPEL engine until someone from purchasing re-hydrates the process instance by approving / declining it using the web application.

The lower pool shows the external endpoints which are called. In this example, all of them happen to be implemented in JBoss. In other examples, some may be provided by business partners.

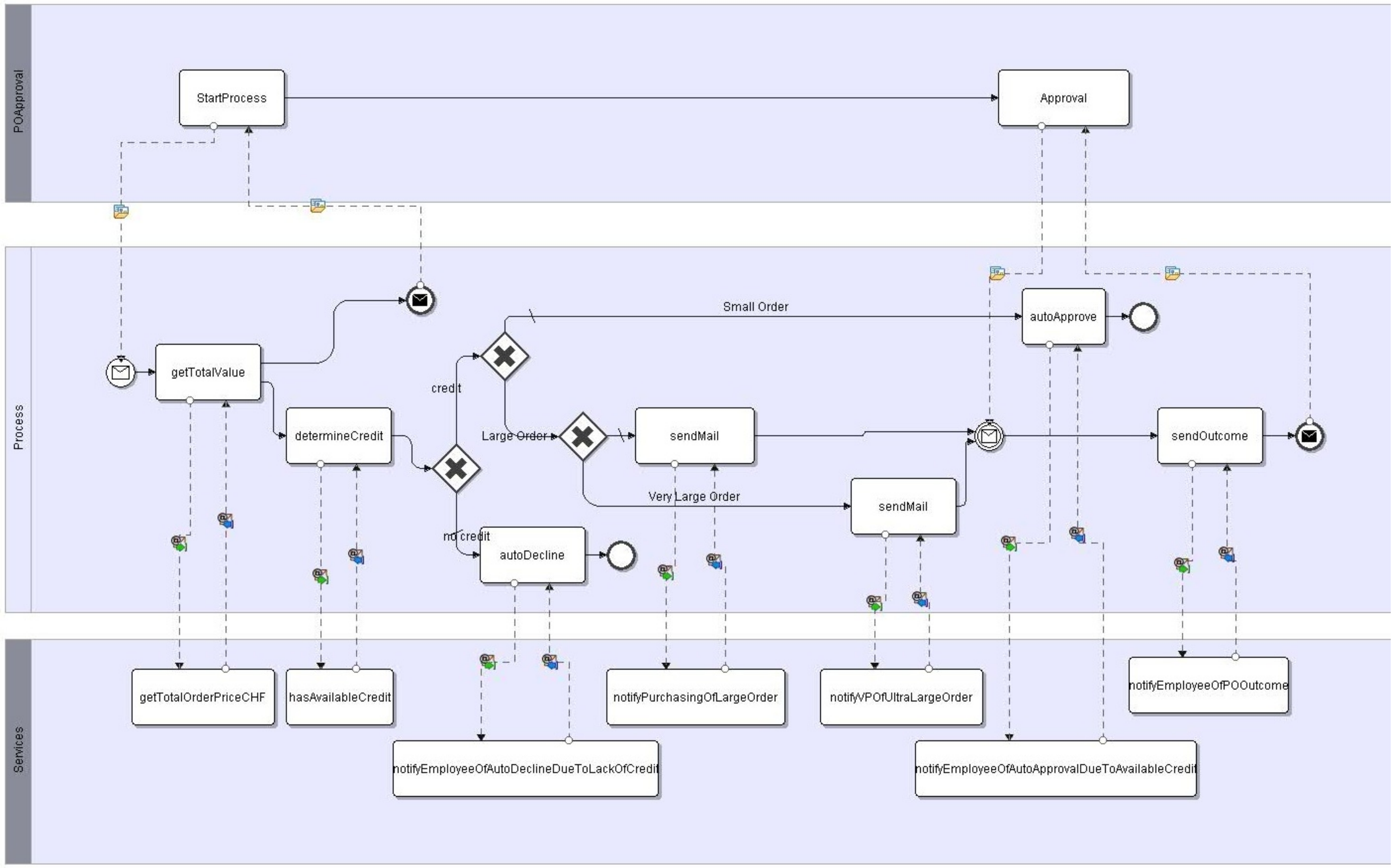


Figure 6: BPMN for the Example

Section II: Discussion

The example provided above was implemented using the platform described at the start of this paper. There now follows a discussion about the authors' discoveries during the implementation as well as their recommendations regarding agile development in such environments.

Service Architecture

As the platform being assessed is geared towards a SOA, services were created to provide the detailed functionality required by this example. These services were as follows.

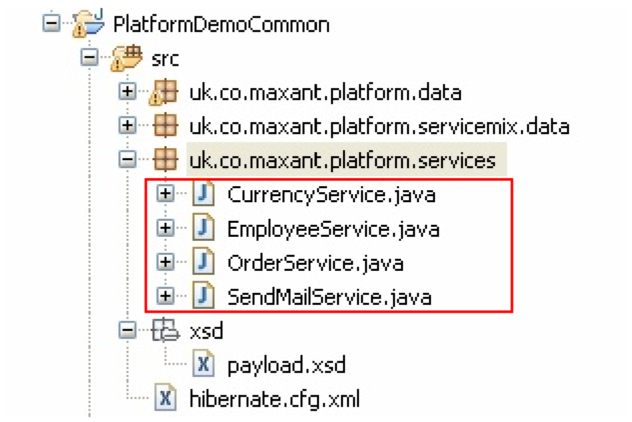


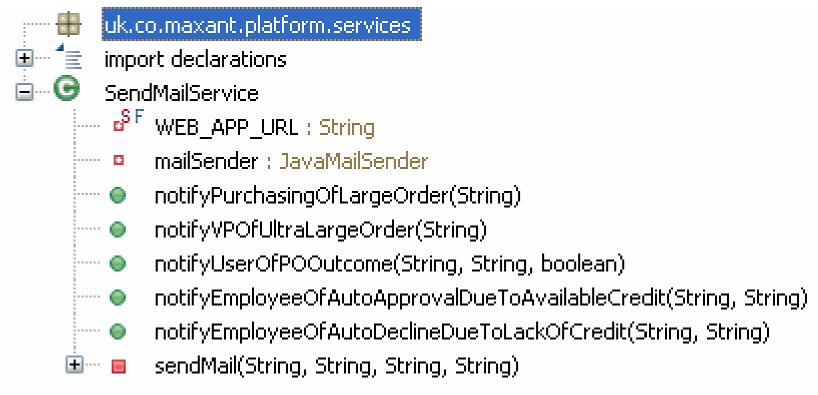
Figure 7: Package Structure showing Services

All of the services listed above were implemented as Spring services. They were implemented in a common library that was made available to both ServiceMix ESB and JBoss AS. The Spring *configuration* was then provided by the library client (the server using that library), for example the JBoss Web Application used the Spring Servlet Configuration in order to specify that the services were to use the JBoss Transaction Manager and take the MySQL JDBC Datasource from JNDI.

Additionally, the common library provided the centralised "payload" structure used within the ESB. This structure was based on an XML schema (the ServiceMix ESB, like many others today, passes XML between its components). Java XML Binding (JAXB) was used to generate Java Classes (a form of Model Driven Development – MDD) to represent this data structure as well as marshal and unmarshal the XML data out of and in to Java objects at runtime. Binding was used as the mechanism to be able to programmatically manipulate the XML data in Java objects, rather than having to parse raw XML, potentially invalidating it (Java objects ensured correctly typed data for example).

The common library also provided Web Service façades which had been generated based on Java Interface definitions. These web services were used to call the Spring services from the BPEL engine. As this was an example, some of the methods provided stub implementations (for example the Currency Service). In some cases, services delegated work to other services.

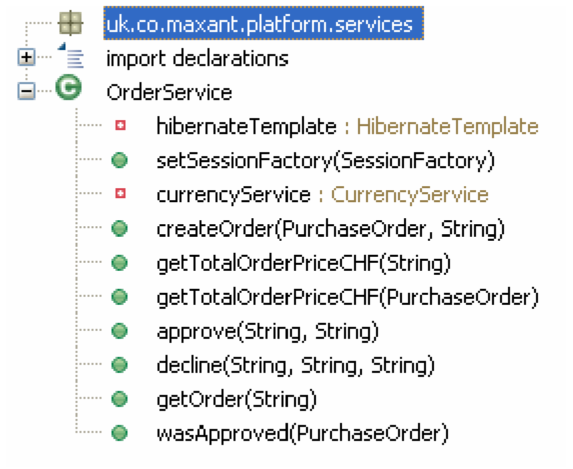
These common services are described in more detail below.



```
uk.co.maxant.platform.services
import declarations
SendMailService
    WEB_APP_URL : String
    mailSender : JavaMailSender
    notifyPurchasingOfLargeOrder(String)
    notifyVPOfUltraLargeOrder(String)
    notifyUserOfPOOutcome(String, String, boolean)
    notifyEmployeeOfAutoApprovalDueToAvailableCredit(String, String)
    notifyEmployeeOfAutoDeclineDueToLackOfCredit(String, String)
    sendMail(String, String, String, String)
```

Figure 8: Outline of the Send Mail Service

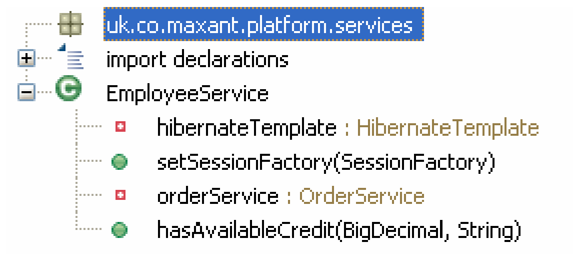
The Send Mail Service was used to send HTML email to employees, purchasing or VPs. The Spring `JavaMailSender` was injected by Spring and provided an encapsulation for sending email – it took its Mail Session from the JNDI tree. The private `sendMail()` method did the majority of the work, while the public methods called it, passing in the text to send in emails as well as the addresses to send the emails to.



```
uk.co.maxant.platform.services
import declarations
OrderService
    hibernateTemplate : HibernateTemplate
    setSessionFactory(SessionFactory)
    currencyService : CurrencyService
    createOrder(PurchaseOrder, String)
    getTotalOrderPriceCHF(String)
    getTotalOrderPriceCHF(PurchaseOrder)
    approve(String, String)
    decline(String, String, String)
    getOrder(String)
    wasApproved(PurchaseOrder)
```

Figure 9: Outline of the Order Service

The Order Service used Hibernate for persistence of the purchase orders. The `createOrder(PurchaseOrder, String)` method was called by the ESB. Some methods such as `getTotalOrderPriceCHF(String)` were intended to be called from the BPEL engine (using the `OrderWebService` web service, which contained a light weight method delegating the work to this Spring service). Other methods such as `approve(String, String)` were called from the web application deployed in JBoss.



```
uk.co.maxant.platform.services
import declarations
EmployeeService
    hibernateTemplate : HibernateTemplate
    setSessionFactory(SessionFactory)
    orderService : OrderService
    hasAvailableCredit(BigDecimal, String)
```

Figure 10: Outline of the Employee Service

The Employee Service used Hibernate to provide a read only view of the database in order to determine if an employee had credit available in order to place a purchase order.

This service was called from the `EmployeeWebService` which delegated its call to this service. The web service was called from the BPEL engine.

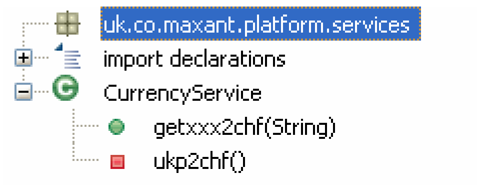


Figure 11: Outline of the Currency Service

The Currency Service provided a stub implementation for converting from British pounds (the currency used by the employees department) to Swiss Francs (the currency used at maxant corp. headquarters). In the ideal world this would have looked up the currencies according to some maxant policy, for example online with a partner or in a database that is updated via batch as and when required.

Correlation

In order to put a business process instance on hold (dehydrate) and to start it up later on (re-hydrate), as was required by this example due to the approval process, a BPEL engine needs a way to correlate calls to the BPEL which belong together. In this case the call to continue a waiting instance needed to include a correlation token so that the BPEL engine could find the correct instance to continue working with. This correlation token was the order ID. The ID was simply a Java UUID generated at the point when the ESB was persisted the order to the database. The ID could equally have been generated from the database or indeed in some other fashion. What is important is that the ID is unique among all business process instances and that it is available upon starting the BPEL in the first place.

BPEL at maxant corp.

The authors have come across several uses of BPEL. It should be made clear that in our opinion, BPEL has limited use and is not a holy grail signalling the end of software languages such as Java. BPEL is not useful for "programming in the small", neither was it ever intended for such. BPEL is however very good for two things. Firstly, one reason it was created, was for the orchestration of web services. By this, it is meant that existing web services which provide distinct business functionality, but which alone are not so useful, can be brought together to provide fully functional applications, or even composite application networks (several applications working in harmony to provide a super application at the enterprise level, integrating multiple business units).

Secondly, BPEL is useful for long lived business processes. In the example, some business process instances need approval from the purchasing department (if they were large or very large orders). Once the process sent an email to the purchasing department, it might be days before they respond to the email and approve or decline the instance. The state of the business process instance needs to be saved somewhere, and a BPM Engine is the ideal place for it. Business analysts and/or support teams can use tools such as the Intalio BPM Console to view incomplete instances and understand what they are waiting for or where they have failed.

For low level details of business processes (saving data to a database, processing data, iterating over lists of data, anything else low level in relation to a business process), the business process should delegate to an external (web) service. There are attempts in the software world to bring lower level languages such as Java into BPM engines, in order to include say database calls directly in the BPEL. Doing so however goes against the paradigm described above and means that processes have the potential to become exceedingly complex. The point of BPEL is to model business processes, at a high level.

An interesting area where BPEL can be useful is where partner systems which it orchestrates are not reliable. If a call to a partner system fails, the business process instance is put on hold, but can be resubmitted at a later time. This initially sounds very useful. However, the practicalities of it need to be examined. If your system is processing millions of messages a day and just 1% are failing then already tens of thousands of messages will need to be resubmitted. Will this occur automatically? Only if the BPEL Engine provides notification of failed calls and provides some kind of API so that they can be resubmitted programmatically. If this mechanism does not exist, then the effort required to analyse the failed calls will bring the support team's productivity to a halt. Not only that, a batch process which can resubmit the process instances needs to be written, deployed and maintained. Yet there is another problem here, namely what happens to failed instances when a new version of the BPEL is rolled out? They still need to be completed, but the old version is no longer available, Some BPEL engines may allow such instances to be completed using the old version of the business process, but what happens if for example a database is upgraded and the failed business process instances cannot be completed because they will no longer be able to be persisted at a point after the failed call to a partner? Problems like these need to be carefully analysed.

Overall, a company which is deciding whether or not to use BPEL should question whether it wants to orchestrate its SOA in BPEL or perhaps by more simply writing façades in its preferred technology (Java, C#, or whatever). BPMN itself is difficult to study in a single glance because many of the details are hidden: the overall BPMN is easy to view, but the conditions within decision points or details of mappings or correlations are buried within other screens (or within BPEL code). Indeed, a branch coming out of a condition can be labelled wrongly and mislead the person reading the BPMN. While reading Java or C# is equally tough to the untrained eye, and while it too can contain misleading information (badly commented code), the authors feel that whether you are going to study BPEL or Java / C#, the effort and cost will be similar. So one needs to consider the other advantages of each solution before deciding upon one or the other. One such advantage of BPEL could be its ability to handle long lived process instances. Another could be its ability to orchestrate unreliable systems, although the pitfalls of this have already been discussed.

So to summarise, BPEL as the authors like to see it used, is for orchestrating web services which implement the low level details of processes, and for saving state of long lived processes. That is all, nothing more, nothing less.

Development Environment

The development environment used throughout was Eclipse. Version 3.3 (Ganymede) was selected for the platform since it was the latest stable release at the time of writing this paper. The only additional plugin which was added was the Quantum DB plugin which provides a database development environment. Even then, the standard MySQL console was used most of the time for executing scripts or viewing database structure or data.

The Community Edition of Intalio BPM Designer is also based on Eclipse. However when the Community Edition is downloaded in includes its Eclipse, as opposed to being able to install the designer as a plugin it into an existing installation of Eclipse (to do that you need to buy a licence from Intalio). As such, the Intalio Designer was run as a separate development environment.

The JBoss Server plugin in Eclipse was used for deploying to and running JBoss. This plugin is now a standard part of the Web Tools Platform (WTP) also included per default in Eclipse 3.3.

Reasons for choosing Eclipse were:

- Wide use within the industry, making it easy for companies using the platform to find staff to program in it,
- Wide range of plugins available for it, making it extensible according to customers demands
- Integration with the Ant build tool, allowing for custom build scripts to be implemented, for example allowing the generation of the data layer (Hibernate objects and mappings), or build and deployment of libraries.
- Integration with version control systems like CVS and Subversion, allowing version control to be included in the agile development foreseen for this platform (useful for refactoring and code reviewing)
- Excellent Web Service and Web Service Client generation wizards based on Java Classes or WSDL definitions
- Authors extensive experience with it.

Hibernate was used for the data layer. It provides a generation tool in order to generate the Hibernate Mappings as well as the POJO Java classes representing the tables. In order to enable agile development, it is recommended that the database be created as a model, and that all mappings and POJOs be generated from it (a form of model driven development – MDD). This should provide the fastest way to develop and work with a database. Additionally, it is recommended that each table have a column / primary key which is generated from the database (as sequence or autoincrement in MySQL terminology). The `equals()` and `hashCode()` methods can be implemented on the generated objects in order to include these details. POJO Constructors should also define an ID (always negative to differentiate between an unwritten object and one which emanated from the database). See the Hibernate reference for more information.

Eclipse 3.3 includes a very good web service generation wizard which was used to generate all web service clients for calling the BPEL from either the ESB (to start the process) or from the Web Container (in order to continue a process instance awaiting approval). Several Spring services needed to be called from BPEL and these were turned into web services by generating web services based upon the interface of the Spring service. These web services then simply called through to the Spring Service. Since the web services lived in the web container, they could access the Spring services using the `WebApplicationContext` made available through the Spring `ContextLoaderListener` which was loaded when the web application started.

Finally, the Intalio BPM Console was used for testing the BPEL part of the system. It is a web application where business processes can be explored and business process instances can be examined. If an instance is held up due to an error, the console can show exactly where the problem is and allows the instance to be resubmitted at the point where it previously failed. These capabilities were extremely useful in debugging the BPEL and web service orchestration and helped to speed the development up significantly.

EJB related to the Example

In the example that has been implemented to test the platform, EJB was not used. The primary reason for not using it was that a common library based on Spring was chosen because the ESB already uses Spring and the Web Container could be easily extended to use the same. To call services from the web container hence did not require calls to EJB. In fact, to make those services available in EJB would have required additional work in writing delegates which provide the same interface and simply call through to the Spring services in the EJB container. In an agile development environment, one aim is to reduce the need for additional work!

One good reason to put the Spring services inside EJB wrappers would have been to check the security context under which the services were running. However, all calls to the services made from the web Container already had a security check within the web container (Java EE declarative security). In the ESB this was a different story – it simply called the Spring service to persist the purchase order. Here, no security check was made. By putting the services in the EJBs layer, security could have been checked (if declarative security were put on EJB methods). But this would have required the ESB to make the service call in an environment able to communicate with EJBs securely. An alternative option would be to enable security within Spring, using an extension such as ACEGI Security available in Sourceforge. Another alternative would have been to use Spring remoting to make the service available remotely (in the web container). The ESB would then call the remote service using HTTP Basic Authentication to connect and authenticate (see http://blog.maxant.co.uk/pebble/2008/08/07/1218135480000.html).

Façades and Transactions

A façade is a standard software design pattern which encapsulates calls to several lower level services or beans. It can be used to set the correct transaction or security context before calls to lower levels are made. It can be used to encapsulate a part of a business process in order to make that part reusable by several parts of the system.

An example is sending an email after making a database update. Here, one would use the Order Service to update the database and use the Email Service to send the email. But ideally, a client would be able to call a single method to do both of those things in a single transaction, so that if the email could not be sent, the database would also not be updated. Alternatively, if BPEL were orchestrating this, the BPEL engine might fail to send the email, but life would be rosy again when the instance were resubmitted by someone in the support team or a batch retry process.

In the example, this functionality was implemented within the Order Service which provided a façade method to encapsulate both the database update and the sending of the email. This method was then made available through the Order Web Service, so that it could be called from the BPEL engine.

One option might have been to make a web service call both the Order Service and then the Email Service. However, a web service has no transactional context and can only start a transaction by making a call to a service (under container managed transactions – CMT – in any case). So if the database call made from a web service were successful but the email call failed, there would be no rollback of the database meaning that we didn't get what was required, namely the two steps within a single transaction. If the database call were idempotent, this would not be so critical, as the failing email would cause a fault in the call from BPEL to the web service. The BPEL process instance would fail and await some maintenance where it would be resubmitted to the web service. During this second call to the web service, the database update would be called again (remember it had not been rolled back the last time because email sending was in a different transaction). If the database call were not idempotent, this would cause problems (e.g. duplicate data). So the strategy chosen does depend upon knowing whether successive calls to say a database would be harmful or not.

The reason that these two steps were not called independently from the BPEL Engine was because it was deemed too complex for the BPEL engine to worry about. Remember, in the authors opinion, BPEL should be kept simple and to the point, its aim being to model the business process from a high level perspective.

ESB in the Example

An ESB is typically used when integrating file systems into an enterprise application. Indeed the ServiceMix ESB comes with file and FTP adapters for reading and writing to the file system.

Reading a file as a starting input to a process is actually a two step process – reading the file successfully, and then starting the business process. The first step has some complexities that are worth discussing. First of all the file system can lock files. Secondly the file system might be having a file streamed to it at the point where the ESB tries to start reading it. Depending upon whether the file adapter considers these points you can run into trouble while reading the file. The same is true of writing or moving (archiving) files.

The second step of starting the business process also has dangers – the BPEL Engine might not be running at the time which the file is read. If this is the case, what should happen? Should the file be left on the file system to be attempted later? Should the file be stored inside the ESB until later? In the authors experience the best approach with ESBs is to use a design pattern like this:

- Reads the input without validating it, and store the data in a JMS queue
- Transform and Process the data from the queue and store to another queue
- (Write the data from the second queue to the target system)

The third point is in brackets as it can be included into the second point, depending upon whether the transformation and processing is complex or not. In the example, the second and third points were indeed combined, because the transformation was negligible. So the ESB contained two components (the task blocks below):

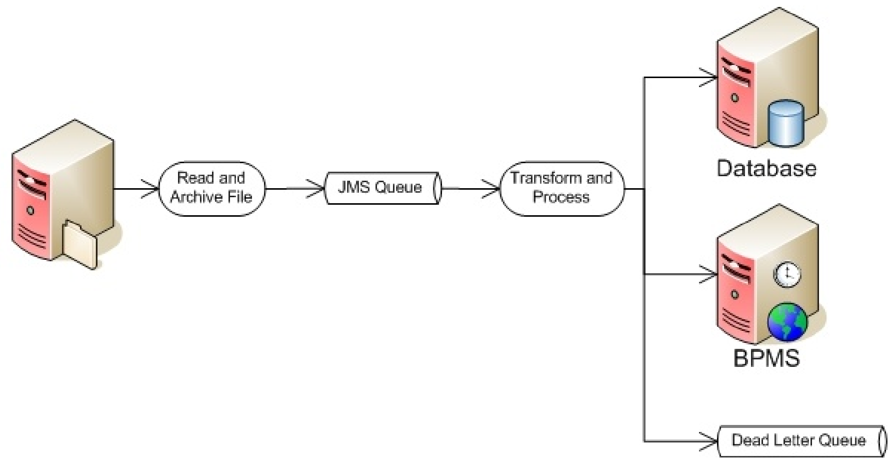


Figure 12: ESB Components

The first component (read and archive) read the file as a raw binary stream. Since the second component (transform and process) is the one which validates the data, this raw stream needed to be sent over JMS to the second component. The ESB however requires that XML data be sent between components. As such the central XML structure used to house the raw data contained a string element which was filled with Base64 encoded data built from the binary stream. The data was encoded and sent by the first component. To marshal and unmarshal the XML data, JAX-Binding was used.

The second component decoded the data and used the Apache POI Library in order to read and process the data as a spreadsheet. The data was stored directly to the database by calling a method in a common Spring service. To start the BPEL process, its Web Service end point was called, using a generated client within the common library.

To cope with the fact that the BPEL engine might not be running at the time when the call to it was made, the second ESB component cleverly caught such faults, rolled back the database transaction where the purchase order had been persisted, and resubmitted the data to the end of the JMS queue. The XML sent back to JMS also contained a counter incremented which was incremented, and which tracked how often the ESB had attempted to submit the data. After a third failure, the ESB sent the data to a dead letter queue (DLQ). Whilst not implemented in the example, an application could have been written to subscribe to the DLQ and react appropriately, for example sending an email and archiving the problematic file to the file system for later resubmission.

Thoughts on Agile Development using this Platform

In order to be able to develop in an agile manner, using the platform described in this paper, there are several things to consider.

Reducing the amount of code written, unless it is almost certain that it will be required in the future will speed up development. For example, don't use BPEL just because it looks nice and might help document the system unless there is a valid business case for using it in the first place. Another example might be to not add functionality in a service because it may one day be useful – wait until it becomes a definite requirement (and refactor if necessary). Similarly, there is no need to worry about performance until it becomes an issue – writing code to be more efficient will cost more than making it efficient at a later date when the bottle necks are known.

The aim is to reduce the time to demonstration and delivery, which when done iteratively relies on the ability to refactor. In order to allow refactoring, a set of good unit tests must be created. Getting the coverage of unit tests right is hard, because aiming for 100% coverage will required the same amount of effort (if not more) as required to develop the services themselves. Aiming for only 25% coverage on the other hand will mean that there are likely to be bugs in the system.

Generate as much code as possible. The paradigm used in this platform was to generate the data layer from the database – the SQL is the model or master copy. Web Service Clients as well as Web Services were generated where possible.

Services were developed to provide business functionality, not CRUD functionality (simple create / read / update / delete). As such, services might embrace several tables, as required. Services were refactored only when duplicated code was found, in which case the duplicated code was refactored into its own service method and "super" services were created to call lower level services. The result was a service hierarchy, where the top level services encapsulate lower level services in a façade. As such, these façades could end up orchestrating your services. This is a real alternative to building a BPEL layer, as already discussed above.

Release frequently! To do this, each piece of functionality that is built could be done on its own branch under source control. Before a branch is merged into the head or trunk or main branch, it gets the head merged into itself. All unit tests are carried out to ensure the system still works (including new tests to test the new functionality). The branch is then merged into the head (during this merge, check the two are now identical). An "integration token" can/should be used to reduce the chances of several people merging into the head at the same time as this can get messy. Anyone breaking the build must pay a fine to the rest of the group (a slice of cake each for example).

The customer (or their representative) should test and play with releases as they become available. He doesn't need to take every single one, but should update what he uses frequently. For this to be useful to him, build the application as a thin but end to end capable application. As time passes, add more and more functionality to fatten it out.

Avoid building a system which is functionally rich (fat) at one end (does everything with the input during the first step but then does nothing afterwards), since the user will not be able to use it. It is better to build something which takes un-validated input and uses it to complete a process as far as it can. You can build stub services to help you do this which will also define the interfaces which these services need. Afterwards, individual developers can go off and implement these services when they have the time, but in the mean time, the user can use his system from end to end. You can define assumptions like "the system assumes the data being input is valid", in order to communicate the lacking functionality to the user at each stage. Building applications in this manner also allows you to get a feeling for where the most effort will be required, as opposed to simply implementing a sequence of steps in a business process from start to end and suddenly realising that the last step in the process will take the most effort and there is no budget left! If you run out of budget on a system where you have started thin and fattened as you go, it may not matter so much to the client that the last bits of functionality are not implemented, because at least they can use the system if they follow the list of assumptions you provide them with. Of course, if you plan your development right from the start, it will help, but a detailed plan and analysis goes against the point of agile development, where the aim is to deliver working systems (although perhaps not fully functional) as soon as possible.

Databases

In order to maintain some form of structure and dominance over the data in the entire system, several database schemas were used. Each application had its own schema. Additionally, a central "Application Schema" was created for application relevant data, rather than configuration data. The database schemas were as follows.

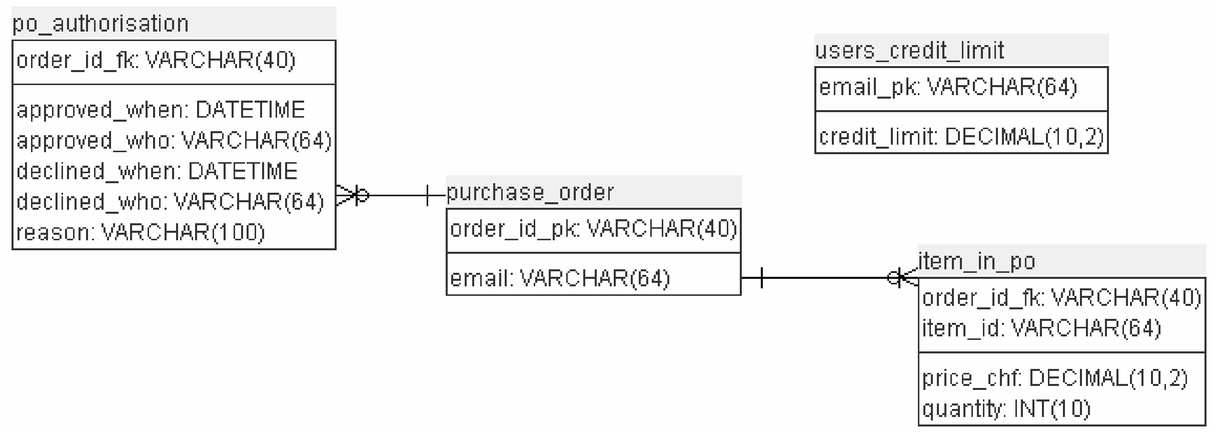


Figure 13: Application Schema

The Application schema held any tables which are used by the "application", whether it is in the ESB, BPEL engine or Application Server.

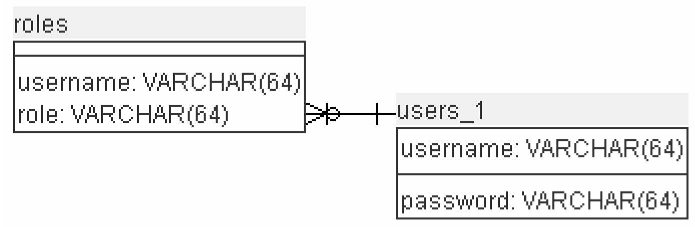
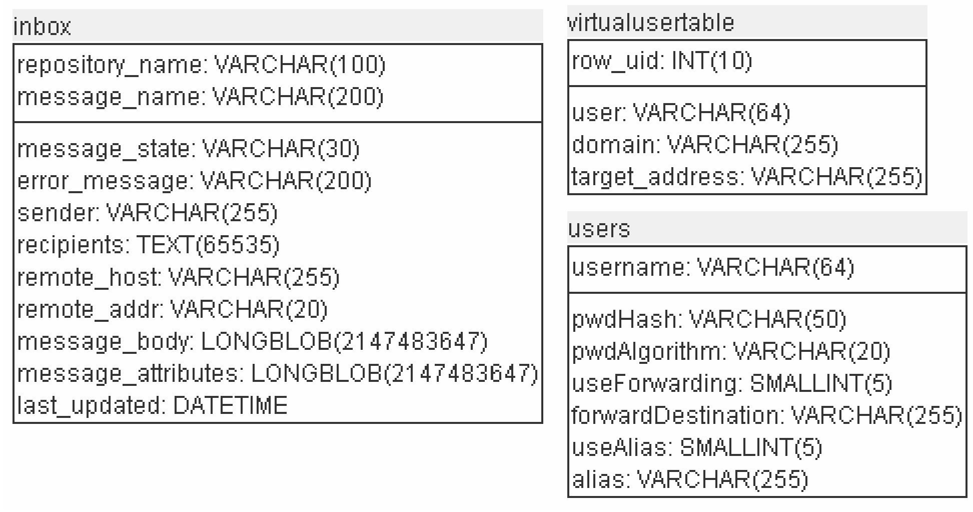


Figure 14: JBoss Schema – Security Realm

The JBoss schema held a security realm, based on credentials and a list of roles per user.



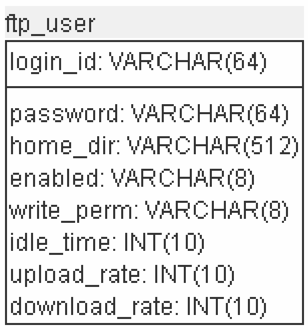
inbox
repository_name: VARCHAR(100)
message_name: VARCHAR(200)
message_state: VARCHAR(30)
error_message: VARCHAR(200)
sender: VARCHAR(255)
recipients: TEXT(65535)
remote_host: VARCHAR(255)
remote_addr: VARCHAR(20)
message_body: LONGBLOB(2147483647)
message_attributes: LONGBLOB(2147483647)
last_updated: DATETIME

virtualusertable
row_uid: INT(10)
user: VARCHAR(64)
domain: VARCHAR(255)
target_address: VARCHAR(255)

users
username: VARCHAR(64)
pwdHash: VARCHAR(50)
pwdAlgorithm: VARCHAR(20)
useForwarding: SMALLINT(5)
forwardDestination: VARCHAR(255)
useAlias: SMALLINT(5)
alias: VARCHAR(255)

Figure 15: James Mail Schema

The James Mail schema contained the "inbox" table which held mail messages in the inbox, a "virtualusertable" for holding relationships between email addresses and users inboxes and the "users" table which held the users and their credentials.



ftp_user
login_id: VARCHAR(64)
password: VARCHAR(64)
home_dir: VARCHAR(512)
enabled: VARCHAR(8)
write_perm: VARCHAR(8)
idle_time: INT(10)
upload_rate: INT(10)
download_rate: INT(10)

Figure 16: FTP Schema

The "ftp_user" table held user / credential information for the FTP server.

Changes required to Selected Software

In order to get all of the selected software to run at the same time and with a common library, several changes were required. These are listed below:

- ServiceMix ESB was upgraded to Spring 2.5, by removing the existing Spring libraries and dropping the latest Spring library in, in their place,
- JBoss had the mail session, JBoss data source and application data source added to its JNDI tree,
- JBoss was configured to use a database security realm based on the JBoss schema shown above,
- The Axis library was added to the JBoss lib path, as this was the only way in which Axis could be successfully used – otherwise class loader issues were a problem,
- Intalio BPMS, ServiceMix ESB and JBoss were reconfigured to use independent ports, so that they could all run at the same time on the same host.

Maturity of Selected Software

First of all, JBoss, MySQL, Spring, Hibernate, Apache httpd (Web Server), Struts, Apache Axis and James Mail server are all relatively mature products which have been around in several versions for some time as well as being tested in production environments by many users. As such, there were no problems with these products during the example.

It should also be mentioned that all applications ran using Java version 6 (the latest stable release).

Apache ServiceMix was deemed to be relatively immature. The documentation for this product is very poor. Functionality of the adapters is also not too mature, for example the file adapter does not allow for renaming during archiving (in case the archive already exists), nor does it roll back the file reading transaction if archiving fails because the file already exists (at least not when it is used out of the box). This resulted in the same input file being submitted to the BPEL over and over again! As such a custom component needs to be written, making it expensive. Additionally, although it is based on the Sun JBI Specification, there is no guarantee that a deployment deliverable will run on another ESB, although this is really a failing in the JBI specification. Compared to the EJB specification, the JBI specification does not go into enough detail to define what a component must look like. Additionally, configuration of a component is contained within a zipped archive (for example the URL of a web service to call). This is poor, since it means that a client of a component may need to take the archive, extract it, update the deployment descriptor and re-archive it before it can be deployed. The Java EE specification on the other hand allows a level of indirection to be introduced which means the deployment descriptor in an archive must only refer to a name, which can then be configured within the application server, using JNDI. At runtime, the Java EE component finds the required reference in the JNDI tree. ServiceMix does not appear to offer such a level of indirection, making deployment an ugly task.

The Intalio BPEL engine (based on Apache ODE) was relatively stable and reliable. The documentation for the engine and the Intalio Designer were fairly good (the examples which existed contained step for step implementation guilds). However, the Intalio Designer has a few things which took getting used to. For example it is often required that a view in the designer be closed and reopened, in order for it to tell you the truth. Or if artefacts like XSDs were deleted, references to them might linger and sporadically reappear, causing compile errors which would simply disappear with a second compile. Another example was that the authors found a bug whereby the GUI fails to validate BPEL even though it is valid BPEL, not allowing the model to be saved (the bug was confirmed by Intalio). Another potential bug was found with correlation whereby the engine would correlate messages only if XML elements were used as the correlation token, rather than XML attributes. A somewhat annoying feature of the Intalio Designer is that if for example a BPEL property name is changed, that change is not propagated to the parts of the BPEL where the property is used / referenced. So the change needs to be made manually in all places. A modern IDE in 2008 is expected to be able to help with such refactoring by allowing the change to be made in one place and letting the IDE propagate the change throughout the BPEL. All of these incidents lead the authors to classify the Intalio Designer as somewhat unready for the masses. However it should be noted that none of these problems meant that the BPEL model could not be implemented and deployed – all had workarounds. A nice feature of the designer is the ability to deploy with a single click. Of course this is less useful where an automated build process based on a continuous integration build server is required.

For the above reasons, it is recommended that development with the Intalio Designer be done incrementally in small steps: implement, test, debug, over and over again. This might slow down the implementation however it reduces bug fixing time and also means there is always a running model, which is useful for the agile development requirement.

The one useful feature of the Intalio BPM Console (or perhaps the Designer) which was notably missing was a "debug" or "simulation" feature whereby one could step through a business process instance seeing its state after each step. The data for each step was available within the console, so it should not be too long before the tool is extended to include such functionality, which is after all available in most commercial process modelling products on the market.

The authors have little experience with the Ranab FTP Server, however for proof of concepts it is adequate. Whether it can perform under load has yet to be tested by the authors, and as such no recommendation can be made. What is useful is that it is configured in a database. Since FTP, Email, JBoss Security and application data are all in the database, they could be tied together to ensure referential integrity of the entire system. Indeed an administrative GUI could be provided for managing such data centrally, allowing the configuration of the environment to be quick and consistent, in turn allowing the development of an application to get under way that much quicker.

Apache DS was not actively used in the example.

The entire software stack was tested on Windows XP, however, since it all runs on Java, it all has the ability to run under any operating system supported by Java.

Development Efficiency

Whilst developing the example, the authors noticed that it took a while longer than expected. The expectation was borne out of the experience of building applications like the example within a single server environment, where everything was build in an ESB or everything was build in an application server. The architectural decisions to use a BPEL engine, ESB *and* an application server meant that considerable time was lost in development.

The creation of the BPEL (including debug and testing time) was the slowest. Partly, this is because BPMN is not as easy to study as a Java class which performs the same functionality – conditions, properties, variables, correlation, etc. are hidden. One alternative is to read the BPEL which is generated, since this contains most artefacts (except WSDL definitions of external services and XML schema definitions of data structures), however the BPEL cannot be simply edited, because the BPEL IDE did not generate in two directions – it could only generate BPEL from BPMN. This led the authors to feel that BPEL implementation efficiency alone is certainly not a good reason to select the technology.

On the other hand, it should be noted that a BPEL engine does bring some added value into play, in the production environment. Most of all, the BPM Console (used for monitoring process instances) brings very useful debugging and monitoring functionality, which allows a support team to quickly analyse the problems with a process instance. Based purely on an application server, such analysis can be painful because the state between each step is not normally stored, meaning that it can be impossible to re-run a bug exactly how it happened in the productive environment.

Statistics and Reporting

While not included in the example, the authors experience has shown that of collection of statistics about how an application is used is a worthwhile exercise. Traditionally, statistics might be gathered using some data warehousing techniques, trying to gather the data together from several existing data sources. In our experience, it is very simple to add a little library that reports statistics data to a table or two. This has the advantage over traditional data warehousing, that exactly what is required get logged and not what happens to be available elsewhere.

If for example the unique key of a row is needed, then it should be logged, as opposed to the friendly yet non-unique name that might be logged elsewhere. Join statements can be used when generating reports to make them user friendly. Alternatively, both the unique key and friendly name can be logged - making the database human readable, and providing accuracy in reports that need it. In one project we logged over 60 columns, allowing us to create around 30 reports (letting users drill down in detail where they needed it). Of course, if you log that much information, you need to estimate your data growth rate (about 20 Mb a day in that particular case) and define a suitable strategy for deleting or archiving old data, before you run out of disk space, or worse, start paying the data centre for additional space when it isn't really needed.

Section III: Recommendations

The platform proposed in this paper is suitable for agile development of enterprise systems, so long as the advice given above is heeded.

While the ESB is somewhat immature in the authors' opinion, it suffices for integration of file and FTP systems. For other integration, it may well be as simple to write a JCA adapter and implement the integration inside the EJB container of the Application Server.

The decision to use BPEL should be weighed against orchestrating services in the Application Server. BPMN is not simple to read and can be misleading if for example condition branches are wrongly labelled (since the actual condition logic is hidden inside the BPEL, not visible in the BPMN on the branch). However, the requirement to maintain long lived business process instances would be a good criterion for selecting to use the BPEL engine. Under no circumstance should any complex logic be housed in the processes deployed to the BPEL engine.

Jboss provides a very stable and reliable application server upon which to base agile development projects. Eclipse provides tooling to make development of Java EE components, their deployment to the application server and the running of the application server efficient.

The use of Spring throughout an agile application will significantly increase the productivity of the development team. While EJB 3 might offer similar mechanisms to build an application based on configuration, it does not include the libraries which Spring does, for example JDBC and email templating.

In order to be able to release a system frequently and reliably, unit tests must be written to cover a large proportion of the project. This allows it to be refactored as and when necessary, as new functionality is added to the application.

Any application developed in an agile manner should be built "thin" – providing an end to end scenario from the very start. This end to end scenario can then be fattened up with additional functionality as time goes by. This ensures that the customer can always use the system and that if a point in time comes where budget is all used up and the application is not complete, that it will run to some extend from end to end, rather than being fully functional for part of the process and completely non-functional for the remainder of it.

For any questions regarding the example, the platform or indeed anything mentioned in this paper, both authors and their companies are available to provide consulting services. Our contact details are below.

References

Apache DS	http://directory.apache.org
Apache httpd	http://httpd.apache.org
BPMN	http://www.bpmn.org
Eclipse	http://www.eclipse.org
Hibernate	http://www.hibernate.org
Intalio	http://www.intalio.com http://bpms.intalio.com
JBoss	http://www.jboss.org
maxant	http://www.maxant.co.uk
maxant Blog	http://blog.maxant.co.uk
MySQL	http://www.mysql.com
Ranab FTP	http://javaboutique.internet.com/FTPServer
ServiceMix	http://servicemix.apache.org
Spring	http://www.springframework.org

About the Authors

Rowan Mountford works as an enterprise architect for Logica UK and has interest in SOA, BPM and ESBs from his day to day work.

Dr Ant Kutschera has nearly 10 years experience of implementing software in the enterprise, in all aspects of the software life cycle, including but not limited to requirements gathering, architecture, design, programming, testing, delivery, support and maintenance. He currently works for various clients as an independent consultant, specialising in software architecture, Java EE, Rich Clients and SOA. His interests also lie in ESB / EAI. He can be contacted through whitepapers@maxant.co.uk.