Enterprise Service Bus

Daniel Hagimont

IRIT/ENSEEIHT
2 rue Charles Camichel - BP 7122
31071 TOULOUSE CEDEX 7

Daniel.Hagimont@enseeiht.fr
http://hagimont.perso.enseeiht.fr

1

This lecture is about Enterprise Service Bus.

Integration - requirements

- Software bricks (applications)
  - Coarse-grain
  - Distributed
  - Different technologies (protocols, systems, API ...)
  - After development
- Collaboration/Integration
  - Distributed communication
  - Adaptation of interfaces and data
  - Complex collaboration schemes (not only client-server)

The objective behind ESB is to provide support for the integration of different applications in a computing infrastructure.

Imagine the different types of software used in an organization. Let's consider the case of administration at N7. There are many software specialized for managing staff, managing students, managing accounting (budgets) … All these software are generally heterogeneous, i.e. there isn't a unique software covering every aspect, provided by a unique company.

The problem is to integrate these software into a consistent information system.

So, the requirement is to manage applications which are coarse-grain bricks, running on different machines (distributed). All these bricks are using different technologies. And the integration happens after development, generally at installation time.

ESB addresses the need for a way to allow collaboration between these software bricks, but it is unanticipated, so it has to allow adaptation of interfaces and data, and also different interaction schemes.

Problem statement

- For so many years, CIO are confronted with the problem of
  - Integrating heterogeneous applications
  - Building complex software architectures
  - Maintaining them
- <u>With applications which were not anticipated to work together</u>

Such an integration (after development, of existing applications) has been a difficult (hardly addressed) challenge for many years.

Integration vs interoperability

- Definition : interoperability is the capacity for a system to exchange information and services in a heterogeneous technological and organisational environment (IEEE, 1990)
- L'interoperability can be ensured by
  - The developer (CORBA, RPC, RMI)
  - The integrator (applications already exist)

We need to depict more precisely the difference between integration and interoperability.

A definition (IEEE) of interoperability is given here. We see that the interoperability problem can be both addressed at development time or later at integration time.

We can observe that interoperability is a general property. It can be obtain at development time by sharing common tools between the developed applications.

Integration targets interoperability between applications which were developed independently.

Point to point integration

- Adhoc technologies
- *The accidental architecture*
- Spaghetti effect
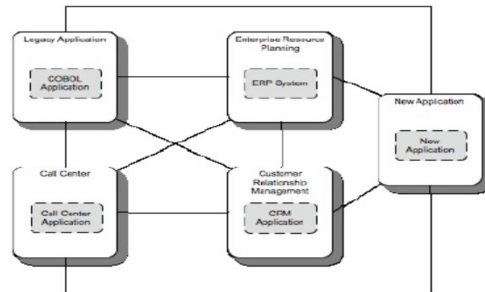
5

A first solution is point to point integration.

Each pair of applications which have to collaborate is interconnected with adhoc technologies, i.e. implementing a specific connector, with a programming language or scripting language. An interconnection is a way to exchange events and data between applications.

This solution leads to what we call the accidental architecture (unanticipated) or also the spaghetti effect.

The obtained architecture is complex and almost impossible to debug.

ETL (Extract, Transform, Load)

- The most popular solution
- Export of data, adaptation and injection in other applications
- In batch mode (generally at night)
- Problem of update latency

A widely used solution is called ETL for Extract Transform Load.

The principle is that each application exports its data (useful for others). Then, these data can be adapted and imported by other applications.

This is generally done in batch, periodically (generally at night).

The main problem with this approach is the latency of updates.

This latency can be a real problem, for instance in the management of a stock, we can sell products that are not available or not be able to sell an in-stock product.

EAI (Enterprise Application Integration)

- As a multi-socket
  - One connector per application
 - The EAI route messages between applications
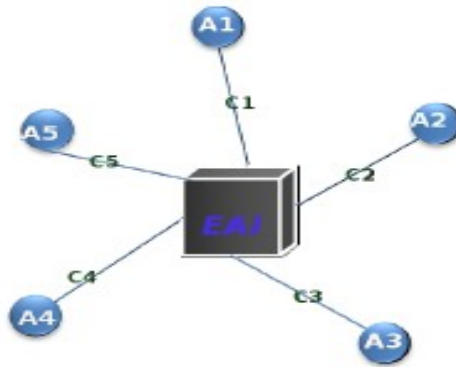
The analysis of the previous proposals led to the design of a type of middleware called EAI (Enterprise Application Integration). This is a kind of hub (like an electrical multi-socket) which allows connecting applications. Each application is connected to the EAI with a connector which may be developed for that specific application. The connector allows sending events and data to the EAI which routes them between applications, following a defined policy.

Therefore, compared to the spaghetti architecture, it provides an organized architecture and a systematic way of integrating applications.

Overall, the EAI provides a means to develop connectors and a server where the interconnection pattern is defined.

ESB (Enterprise Service Bus)

- A decentralized EAI
- Rely on standards (XML, WS, JMS ...)
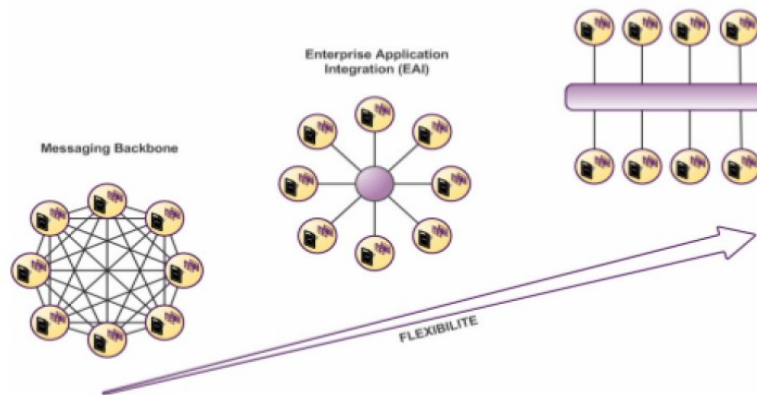
EAI are generally centralized.

ESB is an evolution where the middleware is decentralized and where standards are used for data representation (e.g. XML) and communication (e.g. web services and JMS).

What makes an ESB

- A bus (MOM)
- Data (often XML)
- Adaptators/connectors (WS, …)
- A control flow (routing)

- Objective : foster interconnection

9

So, what makes an ESB is :

- a message oriented middleware (MOM) generally implementing JMS for communication

- data representation with standards (often XML)

- connectors/adaptators for interfacing with existing applications

- means for controlling routing of data

The overall objective (as for EAI) is to provide a structured way to implement application interconnection.

ESB : products

- Proprietary
  - BEA Aqualogic (bought by Oracle)
  - IBM WebSphere Enterprise Service Bus
  - Sonic ESB from Progress Software
  - Cape Clear (spinoff from IONA)
  - Mule
- OpenSource
  - Apache ServiceMix
  - Jboss ESB
  - OW2 Petals (Toulouse!)

There are many providers of such technologies.

Notice that the Apache foundation has it own implementation.

In the labwork associated with this lecture, we will use Mule (for its simplicity).

Mule

- Mule is a Java-based enterprise service bus (ESB) and integration platform that allows developers to quickly and easily connect applications to exchange data following the service-oriented architecture (SOA) methodology. Mule enables easy integration of existing systems, regardless of the different technologies that the applications use, including JMS, Web Services, JDBC, HTTP, and more.

11

Here is the definition of mule by MuleSoft.

What Mule ESB does

- Decouples business logic from integration
- Location transparency
- Transport protocol conversion
- Message transformation
- Message routing
- Message enhancement
- Reliability (transactions)
- Security
- Scalability

12

Mule has the following characteristics:

- decouples business logic : don't need to code integration behavior within applications

- location transparency : an application does not need to know the producer of data it may receive

- transport protocol conversion : applications using different protocols can be interconnected

- message transformation : messages exchanged between applications using different data format can be adapted

- message routing : an update in one application can be routed to other applications

- message enhancement : as for message transformation

Mule also addresses reliability, security and scalability.

Overall view

Generally File, WS, mail, JDBC ...

Generally MOM (JMS+ XML)
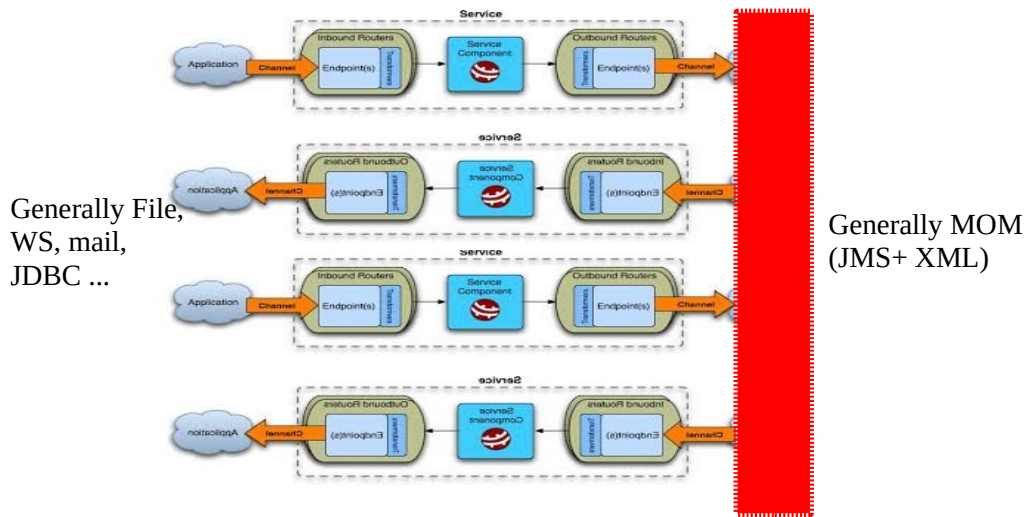
Basically, Mule allows building connectors (horizontal lines) by assembling (reusing) components.

Mule could be used to build a connector between each pair of application, but it would lead to a spaghetti architecture.

Instead, the philosophy of ESB is that each application should be connected with a connector to a communication bus, generally a MOM (relying on JMS and XML standards).

On the left, each connector is connected with an application with a protocol that the application uses (e.g. File, Web service, mail, JDBC …).

This is a means to have a clear and adaptable architecture.

Mule concepts

- Endpoints
  - ➢ Channel for sending or receiving data
- Scopes
  - ➢ Processing blocks : polling, synchronizing, grouping ...
- Components
  - ➢ Custom logic
- Transformers
  - ➢ Data conversion
- Filters
  - ➢ Filtering messages in flows
- Flow controls
  - ➢ Routing messages in different branches of the flow
- Error handlers

14

As said before, Mule relies on a set of components, allowing to build connectors.

Here are the types of components that Mule provides:

Endpoints. They are the contact points (or interfaces) with applications. They implement a protocol which is used by applications to export/import data.

Scopes. They implement a processing (in the sense of scheduling) behavior in a connector. They can be used for polling periodically a state change, waiting for an event (synchronizing), etc.

Components. They are used to implement a custom component.

Transformers. They are components used to implement data conversion.

Filters. They are used to filter messages propagated in the connector.

Flow controls. They are used to create branches in the flow of messages within a connector, for creating different routes for messages or replicating messages.

Error handlers. They are used for defining error handlers.

In the following, not all these concepts are presented, the goal being to introduce the main concepts.

Mule Studio
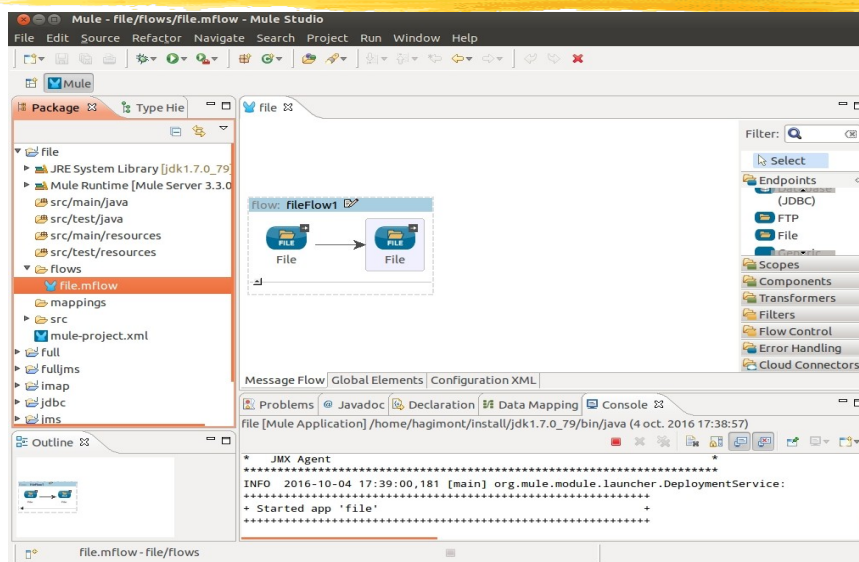
Mule comes with a graphical development environment called Mule studio. It allows to draw a component architecture which implements a connector. Components can be selected from a palette (on the right).
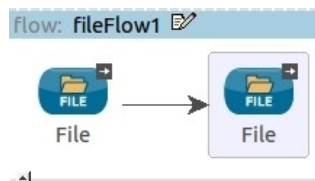
Here, the "Message Flow" tab corresponds to the graphical panel. We can select the "Configuration XML" tab which shows the XML description of the architecture. In the documentation, Mule says that the description of the architecture should be made in XML and that the graphical tools is provided to help the design.

First example

```
<?xml version="1.0" encoding="UTF-8"?>

<mule xmlns="http://www.mulesoft.org/schema/mule/core"
xmlns:file="http://www.mulesoft.org/schema/mule/file"
xmlns:doc="http://www.mulesoft.org/schema/mule/documentation"
xmlns:spring="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="CE-3.3.0" xsi:schemaLocation="
    http://www.mulesoft.org/schema/mule/file
    http://www.mulesoft.org/schema/mule/file/current/mule-file.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-current.xsd
    http://www.mulesoft.org/schema/mule/core
    http://www.mulesoft.org/schema/mule/core/current/mule.xsd ">

    <flow name="fileFlow1" doc:name="fileFlow1">
        <file:inbound-endpoint path="/tmp/in" responseTimeout="10000" doc:name="File"/>
        <file:outbound-endpoint path="/tmp/out" responseTimeout="10000" doc:name="File"/>
    </flow>
</mule>
```

flow: fileFlow1

File → File

16

Here is the XML description of a simple example.

It defines a connector which links 2 File endpoints.

A File endpoint detects the creation (or modification) of a file. There are several possible parameters, e.g. whether the file should be deleted in the source.

In this example, the creation of a source file is detected in the source, the content of the file is transmitted as a message and the file is stored in the target.

In the XML description, each time a component is used, an XMLschema is added (like an include) in the header.

Notice that this is a simple connector, but a connector used in a ESB architecture should have at least one endpoint which is JMS (like in the overall view of ESB).

Endpoints

- **Examples**
  - `<file:inbound-endpoint path="/tmp/in" responseTimeout="10000" doc:name="File"/>`
  - `<jms:outbound-endpoint queue="MyQueue" connector-ref="Active_MQ" doc:name="JMS"/>`
- **Endpoints**
  - Inbound and outbound
    - Ajax, JDBC, FTP, File, HTTP, JMS, RMI, SSL, TCP, UDP, VM …
  - Inbound only
    - IMAP, POP3, Servlet, Twitter …
  - Outbound only
    - SMTP …
- **New endpoints can be developed**

17

There are many endpoints available. Some can be used for free and some have to be purchased.

We have seen the File endpoint on the previous slide. Another example is JMS which connects with a JMS MOM.

Endpoints can be inbound or outbound. And some endpoints can be both.

New endpoints can be developed in order to connect with application specific protocols.

Transformers

- Default transformers (associated with endpoint)
  - jmsmessage-to-object-transformer
- Custom transformers
  - Override default transformers
  - object-to-xml, xml-to-object, json-to-object …
- New transformers can be developed

```
public class MyTransformer extends AbstractTransformer {
        public Object doTransform(Object src, String encoding) throws TransformerException {
        }
}
```

Transformers are components associated with endpoints, which adapt the data format of the content of messages.

Some endpoints have default transformers, but their transformers can be redefined. For instance, a JMS inbound endpoint (<jms:inbound-endpoint>) has a default transformer jmsmessage-to-object-transformer, so that a JMS message is implicitly received as a Java object.

Other available transformers are called custom transformers and can be chained after/before endpoints.

And finally, users may program their own transformers, but generally they don't have to.

Components

- Customize the message flows

```
public class Filter implements Callable {
        public Object onCall(MuleEventContext eventContext) throws Exception {
                person p = (person)eventContext.getMessage().getPayload();
                return null;
        }
}
```

Components in Mule are simply components programmed in Java. They can adapt the messages which flow in a connector.

In this example, we know that the payload of the message is a Person Java object and we can adapt it or even remove the message (return null).

Transformer-component example

```
<flow name="xmlFlow1" doc:name="xmlFlow1">
    <file:inbound-endpoint path="/tmp/in" responseTimeout="10000" doc:name="File"/>
    <mulexml:xml-to-object-transformer doc:name="XML to Object"/>
    <component class="Filter" doc:name="Java"/>
    <mulexml:object-to-xml-transformer doc:name="Object to XML"/>
    <file:outbound-endpoint path="/tmp/out" responseTimeout="10000" doc:name="File"/>
</flow>
```

Custom component

Use xstream (xml to Java bean conversion)

flow: xmlFlow1

File → XML to Object → Java → Object to XML → File

20

Here is an example where we use transformers and a component.

A File endpoint allows detecting the creation of a file in the /tmp/in directory of the local machine. The content of the file is transformed by a XML-to-Object transformer. This transformer uses Xstream which transforms the XML textual representation (the content of the file) into a Java bean object (by default each field in the XML corresponds to a field in the Java bean). Then a Java component is used to adapt this Java bean. The adapted Java bean is then passed to an Object-to-XML transformer which converts the Java bean into an XML textual document. This XML document is then stored in a file in the /tmp/out directory.
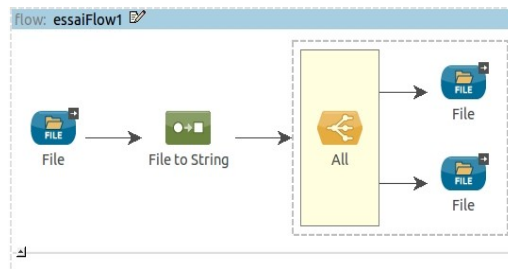
Flow controls

- All
  - Sends messages to all routes
- Choice
  - Routes messages based on expressions
- First successful
  - Sends a message to a list of routes until one is processed successfully
- Round robin
  - Send a message to the next route in the circular list of route
- …

Flow control components allow routing messages following different paths. Here are some examples of such components. In all examples, it's a one-to-N hub, with different routing policies.

Flow control example

```
<flow name="essaiFlow1" doc:name="essaiFlow1">
    <file:inbound-endpoint path="/tmp/in" responseTimeout="10000" doc:name="File"/>
    <file:file-to-string-transformer doc:name="File to String"/>
    <all doc:name="All">
        <processor-chain>
            <file:outbound-endpoint path="/tmp/out1" responseTimeout="10000" doc:name="File"/>
        </processor-chain>
        <processor-chain>
            <file:outbound-endpoint path="/tmp/out2" responseTimeout="10000" doc:name="File"/>
        </processor-chain>
    </all>
</flow>
```

Here is an example with a All flow control component.

A File endpoint allows detecting the creation of a file in the /tmp/in directory of the local machine. The File-to-String transformer produces a String (this is necessary to have at least one processing component). Thanks to the All flow control component, the message is replicated and routed towards two destination File endpoints.

Global elements

- Global elements have to be declared to configure some mule elements
  - JMS connector

    ```
    <jms:activemq-connector name="Active_MQ" specification="1.1" username="admin"
    password="admin" brokerURL="tcp://localhost:61616" validateConnections="true"
    doc:name="Active MQ"/>
    ```

  - IMAP connector

    ```
    <imaps:connector name="IMAP" validateConnections="true" checkFrequency="1000"
    doc:name="IMAP">
        <imaps:tls-client path="*" storePassword="*"/>
        <imaps:tls-trust-store path="*" storePassword="*"/>
    </imaps:connector>
    ```

23

Some definitions are declared in Global Elements (another tab in Mule Studio). They mainly define connectors with external servers, such as a JMS server, a IMAP server or a database server.

Here are two examples :

- a JMS connector describing the connection with an Apache ActiveMQ server

- an IMAP connector describing the connection with an IMAP server

Global elements

- Global elements have to be declared to configure some mule elements
  - Data source (with a bean)

    ```
    <spring:beans>
        <spring:bean id="dataSource" name="dataSource"
            class="org.enhydra.jdbc.standard.StandardDataSource"
            destroy-method="shutdown">
            <spring:property name="driverName" value="org.hsqldb.jdbcDriver"/>
            <spring:property name="url" value="jdbc:hsqldb:hsql://localhost/xdb"/>
            <spring:property name="user" value="sa"/>
        </spring:bean>
    </spring:beans>
    ```

  - Database connection

    ```
    <jdbc:connector name="Database__JDBC_" dataSource-ref="dataSource"
    validateConnections="true" queryTimeout="-1" pollingFrequency="0"
    doc:name="Database (JDBC)"/>
    ```
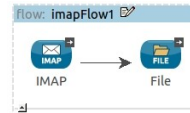
Here are some other global elements :

- a Data source, i.e. the address of a database accessible with JDBC

- the database connector which uses this data source

IMAP / SMTP examples

- **IMAP**

```
<flow name="imapFlow1" doc:name="imapFlow1">
    <imaps:inbound-endpoint host="imap.gmail.com" port="993" user="tpdhlogin"
    password="tpdhpasswd" responseTimeout="10000" connector-ref="IMAP" doc:name="IMAP"/>
        <file:outbound-endpoint path="/tmp/out" responseTimeout="10000" doc:name="File"/>
</flow>
```

- **SMTP**

```
<flow name="smtpFlow1" doc:name="smtpFlow1">
    <file:inbound-endpoint path="/tmp/in" responseTimeout="10000" doc:name="File"/>
    <file:file-to-string-transformer doc:name="File to String"/>
    <smtp:outbound-endpoint host="mail.enseeiht.fr" port="587" user="hagimont"
        to="hagimont@enseeiht.fr" from="hagimont@enseeiht.fr" subject="email from Mule"
        replyTo="tpdhlogin@gmail.com" responseTimeout="10000" connector-ref="SMTP"
        doc:name="SMTP"/>
</flow>
```
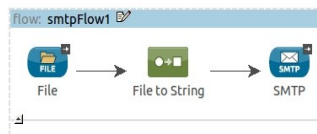
25

The first example receives emails from an IMAP server (gmail.com) using the account tpdhlogin/tpdhpasswd. Each email is then stored in a file in the /tmp/out directory.

The second example detects the creation of a file in the /tmp/in directory, transforms the file content into a String which is sent in an email (to me) using the SMTP server from N7.

JDBC example

```
<flow name="jdbcFlow1" doc:name="jdbcFlow1">
    <file:inbound-endpoint path="/tmp/in" responseTimeout="10000" doc:name="File"/>
    <mulexml:xml-to-object-transformer doc:name="XML to Object"/>
    <jdbc:outbound-endpoint exchange-pattern="one-way" queryKey="insertion"
        queryTimeout="-1" connector-ref="Database__JDBC_" doc:name="Database (JDBC)">
        <jdbc:query key="insertion" value="insert into accounts values
                        (#[payload.nom], #[payload.prenom], #[payload.email]);"/>
    </jdbc:outbound-endpoint>
</flow>
```

This last example detects the creation of a file in the /tmp/in directory, transforms the content of the file (supposedly an XML document) into a Java bean object which is passed to a Database component. This database component :

- references a database connector (Database__JDBC_) which includes the address of the database

- includes the definition of a query to execute. The query inserts a new account in the database. Notice that the fields of the received Java bean object can be accessed in the request with #[payload.field]

Bibliography

- **General**
  - ➤ Enterprise Service Bus:
    Theory in Practice (O'Reilly)

- **Technical**
  - ➤ Open-Source ESBs in Action
   (Manning)
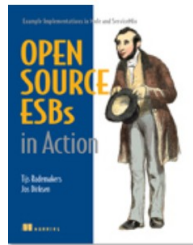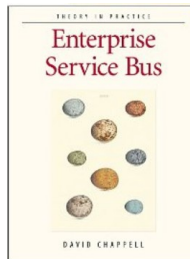
If you are interested by ESB

- the first book is more about the philosophy of ESB

- the second is more tutorial oriented with both references to Mule and Apache ServiceMix.

Connector market place

ESB is very popular in the industry as it responds to a very practical problem.

Mule is provided for free with a set of basic components. More sophisticated components can be purchased from a marketplace.
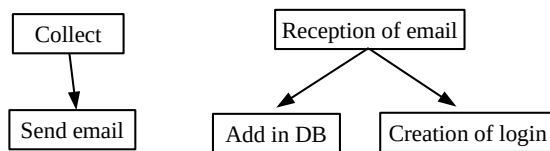
The end
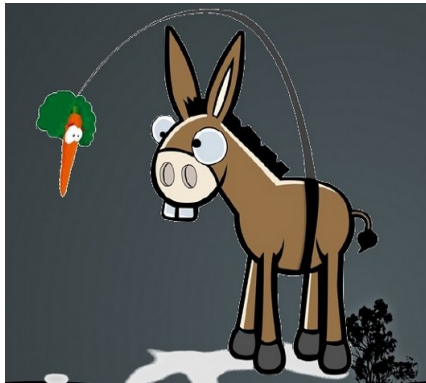
Labwork - ESB - Mule

- Scenario : registration of students
  - Data collection (Collect application)
    - Allows to export data to XML format
  - Reception by email and validation by a secretary
    - Validation by replying to the email
  - Integration in a list managed in the web server
    - Addition in a database
  - Creation of a login for the student
    - Using a web service

```
┌─────────┐              ┌─────────────────┐
│ Collect │              │ Reception of email │
└─────────┘              └─────────────────┘
     │                      ↙            ↘
     ↓                ┌──────────┐  ┌──────────────────┐
┌────────────┐        │ Add in DB │  │ Creation of login │
│ Send email │        └──────────┘  └──────────────────┘
└────────────┘
```

30

For the labwork, we will use Mule to implement a scenario. The registration of students in a school involves different software:

- a data collection application which allows to enter in a graphical interface the name, email, login, password of a student. The data are exported to an XML file.

- Email application. A registration is sent by email to a secretary who validates the registration. The secretary validates the registration by replying to the email (@validation).

- On reception of the validation (@validation) :

        - the registration is added in a database

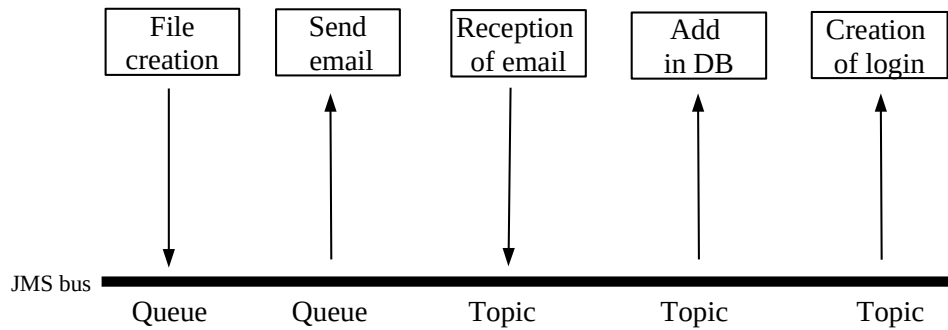   - a login is created for the new student (with a web service)

First integration (spaghetti)

File creation → Send email

Reception of email → Add in DB

Reception of email → Creation of login
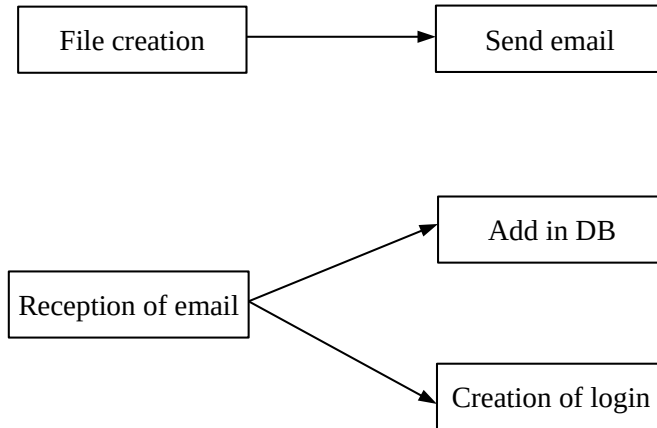
31

This is a first possible integration scheme. Since applications are directly interconnected with connectors, this is typically a spaghetti architecture.

Deuxième intégration (ESB)

File creation	Send email	Reception of email	Add in DB	Creation of login

JMS bus

| Queue | Queue | Topic | Topic | Topic |

An ESB integration relies on a JMS bus. Each application is connected to that bus.

- The detection of a new file exported by the Collect application sends the file content on a JMS queue.

- A message received on that queue triggers the emission of a email to the secretary. The secretary validates the registration by replying to the email.

- The reception of the validation email generates a message (with the registration characteristics) on a topic. Two connectors subscribe to that topic:

    - A message reception on the topic adds the registration into the database

    - A message reception on the topic creates the login for the student.

Different examples
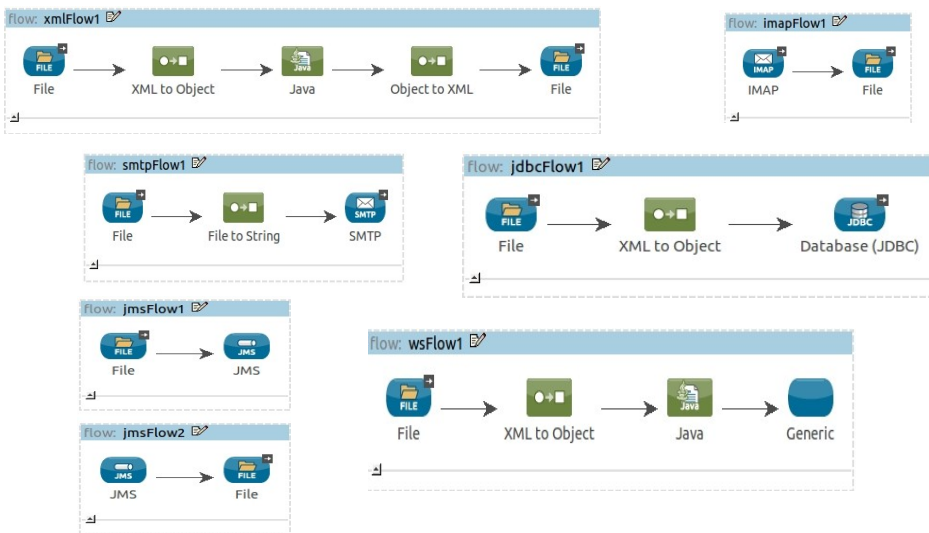
This is a list of the little examples that are given to you for the labwork. From these little examples, you can implement the registration scenario described above.