Barium (a Thorium extension), an Explanation

# Background

The Barium Framework is a test framework based on the Thorium platform. Barium and Thorium are both developed for the specific needs of the Billing project. They are based on JUnit and are therefore highly adaptable and work in concert with several different IDE’s.

Thorium creates the foundation and implements the basic operations and data creation needs for the billing project. (Thorium is described in its own manual)

Barium targets the needs of the integrations between PAN, CRM and Dynamics AX. Its primary purpose is to ensure a stable and fully functional integration motor and to aid in the database scripting in the PAN database.

# The Janus aspect of Barium

Barium has two modes, or two different ways of using it. The simplest one is to mimic the behavior of the integration motor; we are talking about Mule here. This emulation is only for the testing of PAN functionality and enables us to test the whole invoicing solution within PAN. This part of Barium needs no other resources outside of PAN.

The other mode is more complex. It gives us the possibility to replace the different services that the integration motor uses. That is, we can emulate PAN, CMR and/or Dynamics AX. This gives us the possibility to test the integrations and the integration platform to ensure its stability and functionality. This mode is highly dependent of code within the integration motor itself.

# Emulating Mule

This is accomplished by one simple operation, emulateMule. The operation polls the PAN database stored procedures that takes part in the invoicing process and emulates the behavior of the MULE integration platform.

This operation cannot be used simultaneous with a running MULE instance!

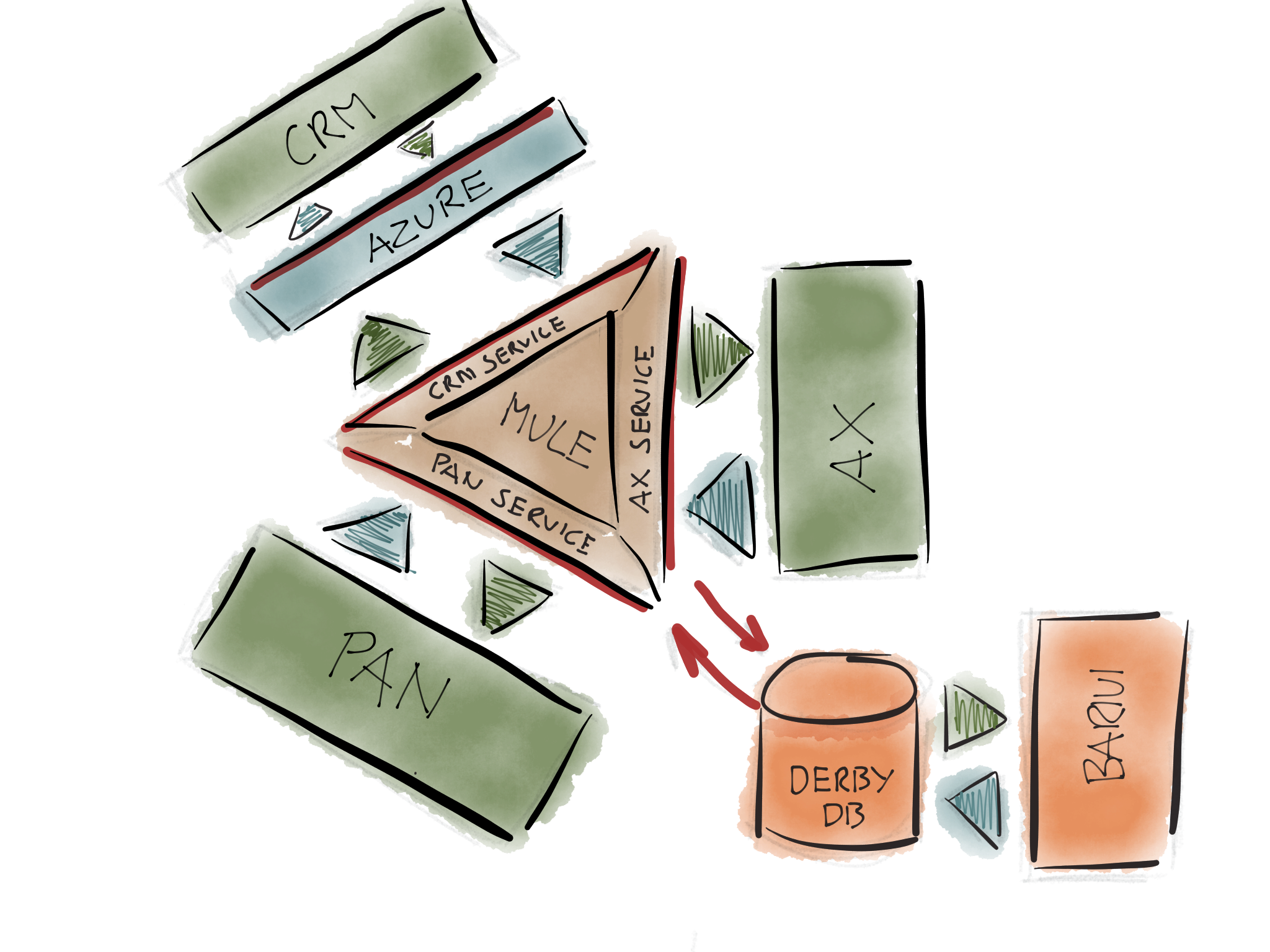
# Emulating the services (PAN, CRM, AX)

This is a bit more complicated. The integration platform has special instances of the service interface implementation that can replace the real ones. This special implementation mimics the behavior of the real one and is controlled by records in an Apache Derby database provided by the Barium framework.

To utilize any of these special test interfaces there has to be a special build of the MULE instance. This is controlled by a Maven build and the version of the interfaces is controlled by Spring configuration files.

The Barium framework has agents that writes commands and reads result from the Derby database. These agents are started, stopped and controlled by the XML scripts. The agents are either producers or consumers, aka the either create data that mimics the output from other systems or they receive data from “other” systems and delivers an appropriate response.

A simple sketch about the different parts I have talked about. The special test interfaces are red and the barium code is orange. There is one thing I have to point out, as soon as you replace one of the service interfaces it means that the system behind it doesn’t have to exist. This means that you can run the MULE integration motor by itself…



# The Producer Agents

\* Adds an agent that creates InvoiceRules according to the parameter.

public void addCrmCreateInvoiceRuleAgent( ProducerParameter parameter)

\* Adds an agent that updates the invoiceRule part of an already created InvoiceRule

public void addCrmUpdateInvoiceRuleAgent( ProducerParameter parameter)

\* Adds an agent that updates the client part of an already created InvoiceRule

public void addCrmUpdateClientAgent( ProducerParameter parameter)

\* Adds an agent that updates the agreement part of an already created InvoiceRule

public void addCrmUpdateAgreementAgent( ProducerParameter parameter)

\* Adds an agent that updates or creates a market message

public void addCrmUpdateMarketMessageAgent( ProducerParameter parameter)

\* Adds an agent that creates InvoiceRules according to the parameter

\* The resulting stored in the correct Azure queue instead of in the Derby DB

public void addAzureCreateInvoiceRuleAgent( ProducerParameter parameter)

\* Adds an agent that updates the InvoiceRule part of the created InvoiceRule according to the parameter

\* The resulting stored in the correct Azure queue instead of in the Derby DB

public void addAzureUpdateInvoiceRuleAgent( ProducerParameter parameter)

\* Adds an agent that updates the client part of the created InvoiceRule according to the parameter

\* The resulting stored in the correct Azure queue instead of in the Derby DB

public void addAzureUpdateClientAgent( ProducerParameter parameter)

\* Adds an agent that updates the agreement part of the created InvoiceRule according to the parameter

\* The resulting stored in the correct Azure queue instead of in the Derby DB

public void addAzureUpdateAgreementAgent( ProducerParameter parameter)

\* Adds an agent that updates or creates a Market Message according to the parameter

\* The resulting stored in the correct Azure queue instead of in the Derby DB

public void addAzureUpdateMarketMessageAgent( ProducerParameter parameter)

\* Adds an agent that creates order lines that the invoicing in AX uses, all according to the parameter

public void addPanOrderLineAgent( ProducerParameter parameter)

\* Adds an agent that creates order lines that the invoicing in AX creates. The result from invoicing...

public void addAxInvoiceOrderLineAgent( ProducerParameter parameter)

# The Consumer Agents

\* Adds an agent that creates the messages that controls the invoicing in AX according to the parameter

public void addPanInvoiceControlAgent( ControlParameter parameter)

\* Adds an agent that controls the state of the Mule AxService

public void addAxServiceFailAgent( ConsumerParameter parameter)

\* Adds an agent that controls the state of the Mule CrmService

public void addCrmServiceFailAgent( ConsumerParameter parameter)

\* Adds an agent that controls the state of the Mule PanService

public void addPanServiceFailAgent( ConsumerParameter parameter)