project:Einstein

The Language of the Enterprise™

Language Primer

Language Version: 0.1

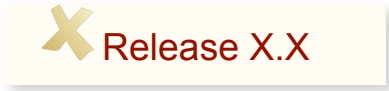
DRAFT - THIS IS ROUGH AND READY

Introduction

The purpose of this document is to to introduce Einstein a 4GL designed for enterprise development.

Einstein was borne out of experience from contemporary ESBs, experiences both good and bad, that highlighted the need for a new high level programming language for heavily distributed complex enterprise applications. A language that supports meta-system programming.

Symbols:



Denotes a feature not in the current release but scheduled to be in the release specified.

Denotes that the feature has newly been added.

Denotes a new feature which may still be experimental - i.e. it may be removed or may not work properly.

Denotes a highly experimental feature.

Denotes a volatile feature likely to change in future release, let's be honest at this stage everything is pretty volatile - however these are the most volatile.

Background

What it is

Distributed

Einstein was designed so that (potentially) each instruction could be executed on a separate machine to each other. It is then up to the user to decide how much distribution, and concurrency, they wish to actually use in their code. Einstein is still in it's infancy so don't expect to see a fully functioning Distributed Execution Model, but in the coming months it will be added.

Event Driven

Einstein is a flow based programming language where instructions communicate through messages.

Concurrency Friendly

Einstein has been written to be concurrency friendly, it uses a combination of well tried shared state avoidance techniques (message passing, read only tuples etc.) with runtime awareness of thread safety. Einstein does not completely eschew the idea of shared state, but instead adapts to the users choice - i.e. using non-thread safe resources causes automatic limits on thread usage.

Complexity friendly

Complexity is a fact of life in distributed environments, Einstein has adopted a few simple techniques to reduce the complexity of implementing distributed systems including Resources and Models.

A Replacement for a Hodge-podge of Technologies

Although Einstein is far away from this goal right now, Einstein aims to replace the plethora of technologies used to integrate multiple systems. Currently we rely on a mixture of ESBs, grid technologies, distributed state systems, co-ordination technologies (like BPEL), workflow systems amongst many others when we need to write a cross-resource system. Einstein is designed to think in terms of distributed requirements so that you need to learn one tool to join many systems together. It's early days; but this remains our aim.

What it isn't

A Java Replacement

Einstein is not a replacement for Java, Einstein is currently implemented in Java and could easily be implemented on .NET just the same. It is a 4GL language which assumes access to a 3GL for low-level work. It makes full use of the vast array of APIs available on the host platform using Resources.

Erlang, Scala etc.

Einstein is designed for distributed execution in complex messy enterprise environments. Every language has it's own aims that colors it's abilities - Einstein does not aim for lightweight, high speed, massive concurrency; although there's no reason why we won't achieve that one day. Instead we aim for qualities such as simplicity, quick adoption, high integration and enterprise readiness. Erlang, Scala and many other languages solve the problems they attack well; it's best to understand each language well and understand which problems they are trying to solve.

Limitations of Existing Technologies

Problems faced by ESBs

Conflicting Requirements

ESBs are expected to support both high-level audited and high-speed lightweight routing. ESBs are expected to talk XML natively, yet support simple object messages. ESBs are expected to support fully distributed paradigms and high speed local execution.

There are many conflicting requirements placed on ESBs - which ultimately allows multiple ESBs to operate in the same business area without necessarily competing; each one being strong at one or more of the requirements. However finding something that supports all the different use-cases is not trivial.

Einstein was designed with these conflicts in mind, it does not impose a particular model upon the user, rather the user can switch into and out of various models at will. So one minute you can be doing native XML processing and the next lightweight object manipulations; you can switch in and out of distributed, thread-based and local execution models at will. The instructions, systems, variables and operators of the language remain the same it's just the means of execution and access that are adjusted to suit a a given model.

Not Quite a Language

This is the feeling from using a contemporary ESBs that you are working with something similar to a language but falls short of providing an actual means to increase the level of abstraction. So far, all the ESBs fall short of a full language, and that is because they **are** ESBs; that is they are built around a single language construct: routing. The difference you'll find with Einstein is that routing is an operation, one of many. Comparing an ESB to Einstein is like comparing Spring IOC to Java. The limitations placed on an ESB is that it is primarily trying to solve one problem in the distributed application domain, routing.

However, let's not be disparaging, it's the ideas that go into ESBs that formed the basis of Einstein; Einstein is built out of the innovative ideas found in Mule, Apache Synapse and others. And in fact Einstein itself is designed to make use of the functionality in these ESBs through the provider framework.

XML

Well it lacks clarity, try converting this to XML:

```
@xmlProfile;
   while "bool:true" {
      read "camel:http://rss.news.yahoo.com/rss/topstories";
      split "xpath://*[local-name()='content']/@url" {
          @pojoProfile;
          write "java:org.cauldron.einstein.ri.examples.ImageMaker";

       } > "java:org.cauldron.einstein.ri.examples.MontageMaker";
  };
```

You'll soon feel that you've entered XML hell. The reason ESBs XML configuration can seem acceptable is that **there is very little functionality described** in the average ESB configuration. Once the system being developed exceeds simple configuration, XML doesn't cut the mustard. Consider for example how this construct (which selectively routes according to the value of the payload) looks in your favourite ESB:

```
( < "text:red") => "java:org.cauldron.einstein.ri.parser.test.ByPayloadRouter"
                  : [
                        red : write payload to "console:Roses are ",
                        blue : write payload to "console: Violets are "
                  ];
```

now consider this is a couple of lines out of, say, a 200 line Einstein program - how would the XML look now. The truth is XML doesn't scale with complexity ; especially when programming. Still don't believe me?

http://radio.weblogs.com/0112098/2004/03/26.html

That's James Strachan apologizing for Jelly - you just can't program in XML!

Problems with using orchestration frameworks

The problem with JBI, BPEL and many other orchestration frameworks/languages is they are often very limited DSLs confined to one area of system integration. They provide a limited solution to a partial perception of the requirements of integration. BPEL is a classic example, based around the need of web-services only. Einstein's aim is to provide a general purpose language for distributed message based programming.

Problems with SCA

Clarity

SCA is not the clearest mechanism for building composite systems, the verbosity of the configuration can easily obfuscate the processes taking place.

More Composition than Co-ordination

SCA aims more towards being a composition DSL than an actual programming language, any high level construct needs to be written in another language and then referred to in SCA. SCA has many advantages when it comes to service composition but is not appropriate as a language for building meta-systems.

Why a new language?

Indeed so why the hell have we written another programming language? Well the answers might still not be obvious, but in order of importance they include:

1. It's because Einstein effectively is it's own virtual machine. Although, (right now) we don't generate bytecode, we do have related operations that need to be executed in a distributed manner which operate on high-level primitives, i.e. Messages and URLs - this is incompatible with Java's single JVM design. The equivalent Java code is just too messy - there's a reason we don't write applications in Assembler anymore - the abstraction should suit the task at hand.

2. Implementing a high-level language like Einstein in Java would be verbose and hard to follow, the expressivity of the operators used and the syntax provided would be both complex and verbose if converted to a string of method calls. Einstein isn't really a DSL, we use that term to ease folk into the idea of using a different language, it's actually intended as a complete Fourth Generation Language (4GL).

3. Well we shouldn't do this in XML!

4. Einstein's parser uses the Deesel project to allow intermixing of the 4GL, Java's 3GL and other related DSLs, While potentially possible with a Java method call based DSL this would be hopeless in XML.

5. Because somethings just require their own grammar and syntax to express themselves, sure SQL queries can be written in Java, but they're so much more natural and elegant in SQL's own DSL.

So how does it work?

The language design is still ongoing, parts of what is described here are current, some are intended. Please visit the website or join or email list to keep up with the latest status on the project.

The Big Picture

Using Einstein it's possible to write applications that interact with a diverse set of services, protocols and network resources in a simple and intuitive manner. Distributed execution, state and communication becomes as simple to program as lines of code and variables. Einstein is a language designed for the complex environment of distributed SOA.

Resources

Resources can **best** be compared to variables in Java. URIs can **best** be compared to types in Java they are (mostly) static, refer to actual resources and can provide functionality. I emphasize best, because Einstein is not Java, similarities are often superficial, Einstein is a 4GL and therefore has abstractions that don't map easily onto Java's language constructs.

URIs

URIs in Einstein are almost the same as standard URIs (with URLs as a subset), except they support two extra features:

Metadata - that is information which relates to a given provider (or protocol in URL/URI parlance).
Nesting - a URI can have an indefinite amount of nesting.

So a URI in Einstein could look like:

```
"cache(timeout=2000):dynamic(lang=groovy):jms(impl=RabbitMQ):$payload.destinationAddress"
```

A DSL for URLs is currently under design which will allow constructs such as:


```
uri::{
  camel (thing=thang) {
     http://neil : password@thing.server.com : 8000/Place/ "another%20Place" ? (a=1, b=2)
   }
}

uri::{
    camel (thing=thang) {
    "http://neil : password@thing.server.com : 8000/Place/ another%20Place?a=1&b=2"
     }
}

uri::{ "http://www.dipa.dhamma.org"};

uri :: {http { www.dipa.dhamma.org : 80 / index.html }};
```

When complete, even the most complex URIs will be specified in a clean manner.

URIs, through the Provider framework, allow a uniform means for accessing both processing and data resources. Each Provider can implement multiple facades allowing a wide variety of actions to be performed against URIs - such as querying, execution, stack operations, send/receive. So when we split a **DataObject** into multiple **DataObjects** we can use any Provider (and therefore URI) that supports the **QueryFacade** interface.

Instantiating Resources

Resources are instantiated through the resource declaration statement:

```
resource "java:org.cauldron.einstein.ri.parser.test.AlternatingRouter" myRouter;
```

These can only be placed in sequential groups. In this example we create an instance of a Java class (AlternatingRouter) and assign it to the variable myRouter, this variable is scoped - currently sequential groups form scopes.

Once we have an instance of our resource we can now apply our instructions to it:

```
< "text: row";
route myRouter [even : > "console:even ", odd : > "console:odd "];
route myRouter [even : > "console:even ", odd : > "console:odd "];
route myRouter [even : > "console:even ", odd : > "console:odd "];
route myRouter [even : > "console:even ", odd : > "console:odd "];
```

In this rather trivial example we route the same instruction to the same router four times, however each time the router alternates between first and second choice (odd/even). This is to illustrate that resources are stateful if declared.

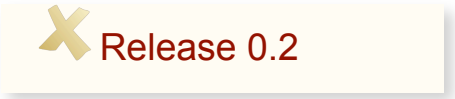
Static Resources

If we declare a resource statically (i.e. we use the URL directly) then the underlying resource should a) be stateless b) should definitely be treated as stateless.

```
< "text: row";
```

So in the above example we use the resource *text:row* directly and the text resource used will be shared across multiple threads.

Resource Characteristics



Resource characteristics are used by Execution Models to determine how much concurrency and distribution to apply. A resource characteristic applies to a resource but Einstein may mark all groups between it's declaration and usages with that characteristic. So if a resource is not thread safe Einstein in conjunction with the Execution Model will ensure that only the current thread can access the resource.

Volatile

This characteristic describes the resource as not being safe for multithread access. Execution models will not allow multi-threaded access to these resource. Beware this also means no listeners or asynchronous write responses will be allowed.

Local

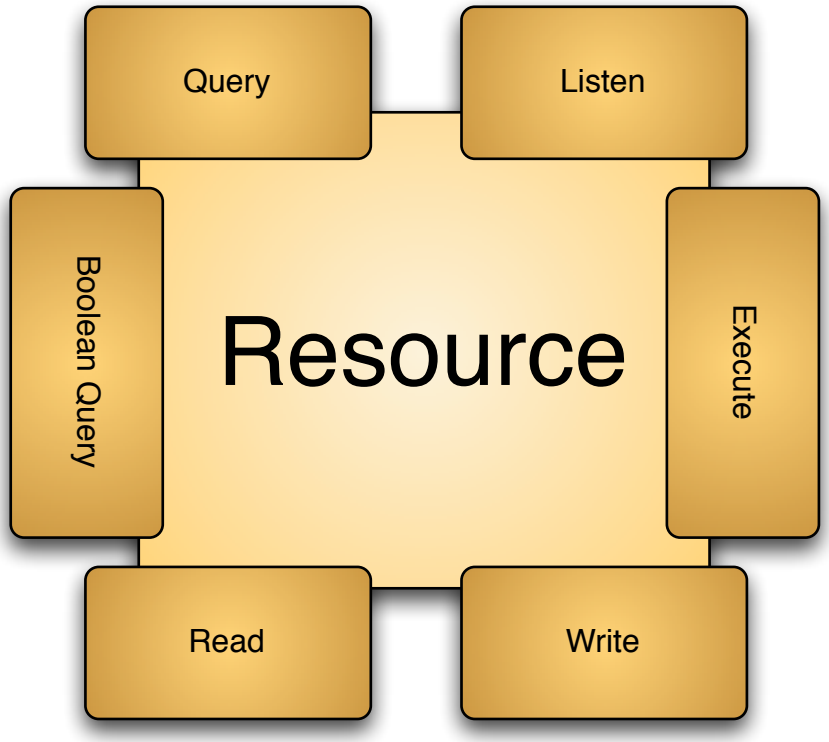
This characteristic describes the resource as not being suitable for multi-node access, specifically the resource should not be serialized.

Idempotent

These means a resource can receive a message multiple times and achieve the same affect as receiving it once. The Idempotent characteristic will be used to determine error behaviors, i.e. can a block be re-executed without side effects.

Providers

How they Work



A Provider gives a consistent interface for any resource capable of receiving, transmitting, executing, querying, routing or processing messages in some manner.

A Provider can be split into three components: ResourceProvider, Resource, Facades.

The Resource Provider

This implements the **ResourceProvider** interface and acts as a factory for resources, it is also the class that is registered with Einstein and contains the metadata annotations the describe how it works.

Resources

The Resource is the class implementing **Resource** that is actually instantiated whenever we use URIs, it contains the state (if any) of the resource and a means for accessing Facades.

Facades

Facades are actually views on a Resource, they provide a simple coarse grained interface to the functionality provided by the resource. A single resource may have many facades.

Facade Safety

Because a Provider implementation provides metadata on which facades are potentially available the compiler can spot incorrect usages for example if I write

```
listen “text:Hello”;
```

the compiler will generate an error informing me that text: does not support the ListenFacade required by the listen instruction. Because of this providers must be present at compile time as well as runtime.

Registry

Providers are automatically loaded into the Einstein registry by the use of the @RegisterProvider annotation, which also provides all metadata relating to the provider.

```
@RegisterProvider(name = "esper",
   metadata = @ProviderMetaData(
     core = @CoreMetaData(
       name = "esper",
       shortDescription = "Provides CEP functionality using the Esper libraries.",
       description = "Supports the Esper CEP library, the standard pattern is to write to
a resource and then add a listener to the resource with a text based query.",
       syntax = "'esper:'",
       example = "     resource \"esper:\" esper;\n\n    listen esper \"text:select
avg(value) from org.cauldron.einstein.ri.examples.esper.TessaractWidget.win:time(30 sec)
\" {\n         ( -> \"xpath:/underlying\" ) >> \"console:Tesseract average value is: \";
\n\n    };execute \"java:org.cauldron.einstein.ri.examples.esper.EventMaker\" >> esper;"
       ),
       alwaysSupported = {
          ReadFacade.class, WriteFacade.class,
          DispatchFacade.class, ListenFacade.class}
        )
)
```

Supplied Providers

Text

The text provider simply generates a message with a textual payload defned by the URI, such as:

```
read from "text:Hello World!" >> “console:”;
```

Console I/O

We can request text from the console or write a payload to the console by using the Console Provider. Text is read in using the JLine library:

http://jline.sourceforge.net/

For example:

```
read from "console:Your name:" >> name;
read from "console(mask=X):Enter your password: " >> password;
```

Will ask for your name and your password, with the password masked by the letter X.

Writing to console out takes one of two forms, either you can simply prefix the message with a textual value like:

```
write to "console:High value widget value= "
```

or we can use printf style format, which is especially useful with tuples:

```
>> "console(format=printf):%s said '%s'%n"
```

Counter

Counters are used primarily for tracking fixed length loops but can be used in a variety of ways. Each time a counter is read from it changes it’s value either incrementing or decrementing depending on the counter’s configuration.

```
resource "counter:1:10:3" count;
while count { read count >> "console:Counter “ };
```

This will count from 1 to 10 in steps of 3 displaying the values to the console.

Regular expressions

Initial regular expression support has been added to Einstein, there's still a long way to go but you immediately use them for conditionals (while, if), split and extract.

```
// -> is the same as the extract instruction.
```

```
(<< "text: The quick brown fox" ) ->  "regex:b[a-z]+";
```

Will give you a new message with a textual content of "brown".

```
// % is the same as the split instruction.
(<< "text: The quick brown fox" )  % "regex:(\\w+)";
```

Will create a tuple [The, quick, brown, fox].

```
<< "text:1 2 3";
if "regex:A.*" {
   << "text:Matched";
} else {
   << "text:Not Matched";
};
```

XPath

The XPath provider allows the querying of payloads using the JXPath and XPath APIs, for more information on JXPath:

http://commons.apache.org/jxpath/

```
read from "camel:http://rss.news.yahoo.com/rss/topstories";
split with "xpath://item/title/text()" { write payload "console:Headline: "};
```

In this example we read the headlines from Yahoo news by splitting the XML using the xpath expression which extracts each headline.

Camel Provider

The camel provider supports the use of Apache Camel's transports:

http://activemq.apache.org/camel/components.html

For example we can request the news from Yahoo:

```
read from "camel:http://rss.news.yahoo.com/rss/topstories"
```

The Camel provider is **highly** experimental.

Esper

```
    resource "esper:" esper;
```

```
    listen esper "text:select avg(value) from
org.cauldron.einstein.ri.examples.esper.TessaractWidget.win:time(5 sec)" {
         ( -> "xpath:/underlying" ) > "console:Tesseract average value is: ";
    };


    while "bool:true" {
         ** "java:org.cauldron.einstein.ri.examples.esper.EventMaker";
         > esper;
    };
```

In our example we're going to create a stream of widgets with values from 0 - 100 and we're going to display a running average of the values over a 5 second window, Esper will receive the events and calculate the average for us.

```
    resource "esper:" esper;
```

So let's break down the example. The first thing we do is to declare a resource called esper, which is resourced from the provider URI “esper:". The provider URI has no configuration in this example.

```
    listen esper "text:select avg(value) from
    org.cauldron.einstein.ri.examples.esper.TessaractWidget.win:time(5 sec)" {...}
```

Next we create a listener, the listener listens to the resource esper and is configured by a text query resource which contains an Esper query.

The query selects the average amount field from all Tesseracts in the last 5 seconds. Each time the value of the average changes the supplied group will be executed.

```
    ( -> "xpath:/underlying" ) > "console:Tesseract average value is: ";
```

The enclosed expression extracts the underlying value from the incoming event (the event has a method **getUnderlying()**) and we pass this to standard output with a prefix of "console:Tesseract average value is: ".

```
    while "bool:true" {
         ** "java:org.cauldron.einstein.ri.examples.esper.EventMaker";
         > esper;
    };
```

Now the listener is set up, we loop forever;, in the loop we execute a stateless class **org.cauldron.einstein.ri.examples.esper.EventMaker** which produces various widgets, amongst which is the TesseractsWidget and send the result to our esper resource.

And that's it, the result of this is:

```
Tesseract average value is: {avg(value)=50.19778415838906}
Tesseract average value is: {avg(value)=50.1952822345592}
Tesseract average value is: {avg(value)=50.195893380529}
```

```
Tesseract average value is: {avg(value)=50.19896761863593}
Tesseract average value is: {avg(value)=50.19766883112768}
Tesseract average value is: {avg(value)=50.20036424690377}
Tesseract average value is: {avg(value)=50.200450640749985}
Tesseract average value is: {avg(value)=50.19990808694293}
Tesseract average value is: {avg(value)=50.203123144840994}
Tesseract average value is: {avg(value)=50.20592852976951}
```

JGroups

Basic jGroups support is currently in Einstein, it's more a demonstration than realistic support for what is a very powerful API.

```
resource "jgroups:chat" multicast;
resource "stack:name" name;

read "console:Your name:" >> name;

listen multicast { >> "std(format=printf):%s said '%s'%n" };
poll "console:>" { [browse name, current] >> multicast };
```

In this example we can see how to produce a simple chat client using JGroups, it firstly asks for your name and then starts a listener on a jGroups resource - each message received is printed to standard out with 'printf' style formatting. We then poll from the console and place the resuts in a tuple with the user name and send it to JGroups. Simple!

Time Provider

THIS IS MARKED AS VOLATILE AS THE TIME ASPECT MAY BE SEPERATED FROM THE SCHEDULE ASPECT, AS THIS FEELS LIKE OVERLOADING OF THE RESOURCE.

This provider currently supports retrieval of the time and listening for a periodic time schedule.

```
write payload "console:The time is " << "time:yyyy-MM-dd'T'HH:mm:ss.SSSZ" ;
```

In this example we retrieve the current time, internally the time provider uses Joda Time and the format for the time can be found here:

http://joda-time.sourceforge.net/apidocs/org/joda/time/format/DateTimeFormat.html

If we would like to schedule a regular action we can listen to a time resource

```
resource "time(schedule=cron):0/1 * * * * ?" everySecond;
resource "stack:TimeStack" stack;
listen everySecond  {write to stack};
```

Firstly we declare a Cron based time resource, where the Cron expression resolves to every second. Next we create a stack to put messages on. Finally we listen to the timed resource, which means that **write to stack** will be called each second with it's input being the current time.

Messages

All data in Einstein takes the form of a message. A message contains a payload and associated metadata which relates to such concerns as routing, security and so on. The payload takes the form of a DataObject, DataObject's are a means to interact with payloads without knowing the nature of the data in the payload. So one may query, join, split, etc a payload without knowing whether it is XML or an object graph. This abstraction is essential to allow the instructions and operators of the language to perform the same operations no matter what data it is dealing with. Once you have written a splitting router for example (by combining the **split** and **route** instructions) it will do that operation to any data it receives, improving re-use.

Message payloads are immutable, messages are thread-safe.

The payload of a message can not be modified, the Message Model is responsible for ensuring that the messages passed to resources cannot be modified, either by cloning or guarding. Resources should never try to modify the payload of the inbound message.

Messages themselves do not directly expose their state, an action must be passed to a message, and depending on the action, the appropriate state is passed back to the action. This inversion of control helps to make sure that, as much as possible, the interactions with a message can be audited. This allows for easy auditing and debugging of a message's lifecycle.

Data Models

The payload of a message contains a DataObject rather than a Java Object. A DataObject allows multiple data formats to interact with Einstein, so we can interact with XML in the same way as we interact with an Object Graph or CSV files etc.

Each DataObject has a DataModel which allows us to create new DataObjects.

We can change the DataModel currently in use with the **using** keyword.

```
using (DataModel:XMLDOM)  {

    resource "time(schedule=cron):0/1 * * * * ?" everySecond;

    listen everySecond "bool:true" {

         read "camel:http://rss.news.yahoo.com/rss/topstories";
         split "xpath://*[local-name()='content']/@url" {

            using (DataModel:POJO) {
                execute "java:org.cauldron.einstein.ri.examples.ImageMaker";
            };

         };
         write each "java:org.cauldron.einstein.ri.examples.MontageMaker";

    };
};
```

In the above example we start using the XMLDOM DataModel, which means when we split later on we apply an XPath query directly to the DOM and get some DOM fragments back. When later we switch to POJO we actually make no real change to the object but the Object Graph Data Model, however the result of execution will be kept in object format, not converted to XML. Whenever we switch data models, incoming messages are converted to that format automatically.

Rosetta Stone

Conversion between Data Models occurs using the RosettaStone (which we obtain from the current runtime), the RosettaStone is built up from a set of ConversionStrategy implementations. A ConversionStrategy is registered with the Einstein Registry by having the RegisterConverter annotation which describes what model it can convert from and to.

Data Models allow Einstein to interact with a wide variety of rich data types without understanding the semantics of the underlying data set. So for example we can split an XML fragment in the same way that we'd split a **java.util.List**. By making an abstraction for instructions to interact with various data models we allow easy inter-conversion of data types and very high reusability of code.

Execution Correlation

Distributed Execution Threads

Einstein programs may execute across multiple threads and multiple physical machines. This presents a few challenges, not least conceptually. The equivalent concept to a thread of execution in Einstein is a correlated message flow. Each message has an execution correlation which is used to determine the thread of execution it participates in. We can say that each message in a single Flow Execution Group has the same execution correlation, however if we create a Tuple Execution Group each message will have the same execution correlation as an ancestor.

Scoping

Scopes therefore relate to execution correlations, if a resource is created by a message flow with a given execution correlation then any messages which have that correlation as an ancestor are in that scope. This (potentially) allows multiple threads of execution on different nodes to refer to a common shared resource.

Execution Groups

Einstein has three different types of execution groups: flow, tuple and competing.

Why?

The reason for the three groups is to support concurrency and tuple manipulation.

Flow Execution Group

This is the usually way that instructions are executed in Einstein, a flow group looks like this:

```
        {
            read "my:firsturl";
            execute "java:org.me.MyService";
            write "my:secondurl";
        }
```

The braces denote a sequential block, in a sequential block the result of the previous instruction is the input to the next instruction.

Tuple Execution Group

List

A tuple based execution group allows instructions to be invoked independently of each other with the combined results being available after execution. For example:

```
        [read "text:1", read "text:2", read "text:3"]
```

Will produce a tuple of **["1","2","3"]**; the order of execution is not guaranteed and may or may not be parallel, the order of the results is guaranteed to match the order of the instructions. The input for each instruction is the result of the previous sequential operation.

Map

Map based execution is the most flexible, it can work in one of two different ways. A map based execution group can have it's members referred to by the routing instruction, i.e.

```
      route "java:org.me.MyRouter" [
            red : write "jms://redQueue",
            blue : write payload to "http://blueserver.com/Service"
      ]
```

But also a Map based group can be executed directly:

```
      [ red : get "text:roses", blue : read "text:violets" ]
```

In the above example the result would be a map of **["red":"roses", "blue" : "violets"]**. Maps are essentially named tuples.

Competing Execution Group

A competing group has all instructions executed at the potentially the same time; however, unlike tuples, the value of the group is the first successful result returned, not the combined results. This is useful when accessing multiple resources for the same data (e.g. price feeds).

```
( read “http://myfirstserver.com/PriceFeed”, read “http://mysecondserver.com/PriceFeed” )
```

As long as one instruction executes successfully no exception will be thrown if others fail, if all fail an exception will be thrown. This provides built in fault tolerance to the system.

Tuples

The result of any type of execution is a Message Tuple - a Message Tuple is one or more messages that internally may be single, aggregated as a list or aggregated as a map.

Positional Tuple References

Einstein supports tuple positional references such as:

```
[
  read from "text:Hello", read from "text:World",
  read from "text:1", read from "text:2"
];
0 to 1;
```

This will produce a tuple with two entries **[Hello, World]**.

```
[
  read from "text:Hello", read from "text:World",
  read from "text:1", read from "text:2"
];
2 to end;
```

This will produce a tuple with two entries **[1, 2]**.

```
[
  read from "text:Hello", read from "text:World",
  read from "text:1", read from "text:2"
];
[3,2,0,1];
```

This will produce a tuple **[2, 1, Hello, World]**, sometimes the short hand make look a little obscure so you can use the element instruction in full:

```
[
  read from "text:Hello", read from "text:World",
  read from "text:1", read from "text:2"
];
element 1 to 3;
```

Merging Tuples

To convert a group of messages into a single message we use the **merge** instruction. Merging produces a single message - however the Data Model now understands that it holds a collection and can be re-split later on with the **split** operator. For most data models the following should have no effect on the payloads:

```
    [red : read "text:roses", blue : read "text:violets"];
    merge;
```

```
    split;
```

So there we have it, Positional Tuple References. A feature on the table for 0.2 is a special tuple position '.' which refers to the element matching the current position in a new tuple (like Erlang's '_') but that's still under consideration.

Operators

Syntax

Operators which have a left hand side (LHS) argument (i.e. binary. ternary) should either have an Execution Group as the LHS argument or a keyword based instruction so the following is valid

```
read “text:Hello World!” >> “console:”;
(<< “text:Hello World!”) >> “console:”;
```

but

```
<< “text:Hello World!” >> “console:”;
```

is not, this rule exists so that we don’t create artificial operator precedence rules. If in doubt enclose it in parenthesis. You’ll soon get the hang of when to use the operator or instruction forms and how to combine then.

Types

Each operator may have more than one form unary, binary, ternary etc. at some point all variations will be fully documented and available through the compiler help, this is not the case at present.

N-ary

N-ary operators work by first creating a tuple out of their arguments and then operating on the tuple. This means that a seemingly sequential operation can be concurrent. For example merging using the merge operator ~

```
read “text: hello” ~ read “text: world” ~ read “text:!”;
```

is actually the same as

```
~[read “text: hello” , read “text: world” , read “text:!”];
```

Unary

Unary operators are always postfix in Einstein, that is the operator symbol comes before the argument. Unary operators have a variation called “unary reversed” this means that the argument is executed before the operator. Merging a tuple (~) is unary reversed whereas extract (->) is plain unary

```
//unary reversed, the tuple is created before the merge is performed.
~[read “text: hello” , read “text: world” , read “text:!”];
read “text: hello world”;
//unary
-> “regex:(h\\w+)”;
```

Binary

Binary operators often have a unary variation since the result of the previous command in a Flow Execution Group is the input to the next command. For example:

```
read “text: hello world” -> “regex:(h\\w+)”;
```

is the same as

```
read “text: hello world”;
-> “regex:(h\\w+)”;
```

Binary reversed operators execute the operator before the LHS. So a binary examples is:

```
read “text: hello world” -> “regex:(h\\w+)”;
```

And a binary reversed example is :

```
write “console:You said:” << “console:Enter text”;
```

so in this example the right hand side is executed:reading from the console; before the left hand side: writing to standard output.

Ternary

Ternary operators are a lot more common in Einstein than Java, but just like in Java they need to be treated like salt - a little is great, a lot will make you sick! Ternary operators have the syntax:

```
<argument> := <execution-group> | <reference> | <literal>
[ <execution-group> | <instruction>  ] <operator-symbol> <argument> “:” <argument>
```

A common ternary operator is a variation on read, read with timeout

```
<<  “jms:queue” : 3000;
```

Operator Only

```
“(“ <operator-symbol> “)”
```

Even more than ternary operators, use these wisely!! The operator only form exists for when you need to express an action concisely, consider:

```
listen “jgroups:chat” { (~) >> “console:”};
```

This listens to a JGroups cluster, merges any tuples it receives and sends the result to standard output. The form is expressive but if used too often will lead to confusing code.

Systems

A system in Einstein is the equivalent of a Class in Java, it combines functions, events, state, agents and communication.

Events

A system can listen for events and act upon receipt of those events. A listener can listen to events on either a member **variable** (space style), **transient** (in memory) variable or a **port** (socket style). E.g.

```
system MySystem {
    using (Profile:Local);

    resource transient myVar;

    port myEndpoint;

    listen myVar {
          write "console:Received:"
    }

    listen myEndpoint {
          write myVar;
    }

    listen “http://my.server.com/myService” {
          write myEndpoint;
    }

}
```

Agents

Agents are like listeners but they can be run once only, at regular intervals and so on, they respond to system or timed events

```
agent [<scope>] [timed <timing>|start|stop|both] [loop] <group>
```

The default scope is 'system'.

```
system MySystem {
    using (Profile:Distributed);

    agent node start {
    }

    agent start loop{
```

```
            //This agent should have it's instructions executed locally.
            @execution=local;
            (read "jms:in") > "jms:out";
    }

    agent domain timed "..." {
    }
}
```

State

Systems hold state in much the same way as a Java Object would, except of course that systems can be distributed entities - therefore the state may also be distributed. Variables may be either persistent or transient. The state of a systems is **never** available to other systems, ports should be used for inter-system communication. The reason for this is that shared memory is evil and Einstein is trying to wean

Communication

Ports are what are commonly known in messaging as endpoints; they are also similar to the TCP ports associated with a network interface. Systems are communicated with via ports, first they are declared in a system, the system is instantiated and then the port can then be communicated to on the created instance:

```
system MySystem {
    port myPort;

    listen myPort {
           write "jms://out.queue";
    }

}
```

```
//We create a system based on MySstem. it will be known as 'my.system'.
create MySystem() as my.system;

//We can now reference the created system resource through the system provider
read "text: Hello world" >> "system://my.system/myPort";
```

Functionality

As you would expect, a system can also have methods just as in OO languages, the key difference is that there is always an implicit variable which is the input. The parameters specified are the names of the expected members of the **ExecutionGroup** passed to it, i.e. this only applies if a Map style **ExecutionGroup** is passed to the method.

```
system MySystem {
    port myPort;

    method myMethod ( y, z) {
```

```
            ( x >> y ) >> z;
    }

}
```

Profiles and Models

The following section will briefly explain models, model information can be declared within a system and will apply to the system itself and any methods, variables etc. however when a system instance is created these can be overriden.

```
//We create a system based on MySstem but make the execution model distributed and
//the default data model XML, it will be known as 'distributed.my.system'.
create MySystem() (@execution=distributed, @data=XML) as distributed.my.system;

//We can now reference the created system resource through the system provider
read "text: Hello world" >> "system://distributed.my.system/myPort";
```

Type Safety

Release 0.2 - 0.3

Guarantees and Requirements

A guarantee is a URI that is guaranteed that, when queried against the current message payload, will return true. When made, a guarantee holds true until a message modification (technically messages are read only so this means the actual lifetime of a message object). So:

```
read "text:red";
guarantee "regex:[a-zA-Z]";

.... some other code here that does not modify the message state ....

require "regex:[a-zA-Z]";
write "console:Roses are ";
```

So when we make a guarantee (say at the end of a method) it can be checked by a requirement later. This works by attaching the URI directly to the message and scrubbing all attached guarantee URIs on message modification. When the require instruction is executed it first checks for a guarantee; if the textually matching guarantee is found then no action is taken; if however it is not found then the requirement URIis executed against the message (unless of course some flag says to fail).

So why do this? Guarantee checks are almost instantaneous, a pure string comparison, and so can be used as a lightweight Design By Contract (DBC) construct. If no Guarantee has been made then the check will still occur, but will now incur a processing cost.

By changing execution mode or a system variable it should be possible to enforce the actual guarantees and the requirements .. i.e. we can check to make sure that the guarantees made are in fact valid.

Finally because they can be treated as static strings it's possible at a later stage to check for invalid combinations at compile time not just runtime.

Types

Types in Einstein are a collection of Guarantees, Guarantees make assertions about the message in transit which can be either verified or trusted.

```
"type" <identifier> "(" <guarantee-uri> ( "," <guarantee-uri> ) * ")"
"type" <identifier> "constrains" <identifier> "with" "(" <guarantee-uri> ( ","
<guarantee-uri> ) * ")"
```

e.g.

```
type SmallWidgetType ("groovy: payload instanceof Widget", "xpath://widgetSize < 100");
type SmallBlueWidgetType constrains SmallWidgetType with ("xpath://widgetColor =
'blue'");
```

Now we have a type we can use typed resources

```
resource SmallBlueWidgetType "jms://in";
```

What does this do?

Firstly it tells the compiler that the type of messages a resource can handle, so compile time type errors can be spotted. Secondly, a given provider can determine which type of resources it deals with and therefore the compiler will complain if you make an incorrect type assertion. Thirdly at runtime the types can be checked for validity and exceptions can be thrown. Fourthly, the degree of type checking can be altered at runtime.

Runtime Type Safety Check Levels

Permissive - None.
Proof - The guarantees specified by a type are checked on the declaration and on use.
Trust - Messages are checked for their runtime type, but the guarantees associated with the type are not checked.
Secure - Guarantees are not checked in declarations, but the type guarantees are checked on use.

Other Models

Execution Models

Execution Models are another very important aspect of Einstein, they provide an abstraction of how instructions will be executed. Execution models can be entirely user defined - allowing the potential for Einstein to execute on a variety of different platforms - from simple, direct execution through SEDA to grid/fabric models. Execution Models themselves provide the model for distributed variables, stacks, execution and communication - all the features of a system are affected by the Execution Model.

The Execution Model specifically provides means for performing these actions:

- Executing one or more instructions.
- Iterating over a data set with one or more instructions.
- Manipulating a stack which relates to the means of execution.
- Manipulating variables in a way which relates to the means of execution.

Why would we require multiple data models? Well unlike a simple low level programming language Einstein is designed to work across multiple processors and physical nodes. The reason for that is, simply, that applications designed to scale need to spread across all processing resources available to them - but not all applications and not all parts of applications need to scale in the same way. If I need to do a fast local iteration I don't want the instructions to be executed in the same way as the parts of the application that are distributed across a grid of processing nodes.

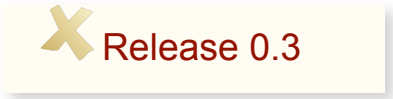
The most common execution models are : immediate/direct, multi-threaded and distributed.

Resources
A resource in a direct execution model is trivial to implement, but what about a distributed system. Well we still need to make use of the concept of resource, but now we're more likely to use a distributed state management system to implement them, like Coherence , ehcache or Terracotta or pass them along with the messages.

Message Model

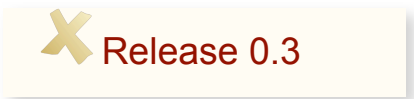
.....

Transaction Model



The Transaction Model determines how transactions should be started and managed.

Exception Model



The Exception Model determines what should be done when an error occurs during an instructions execution, this may include retrying, rolling back transactions and so forth.

Other Language Features

Three ways to say Nothing!

In Einstein we have three ways to say nothing: Null, Empty and Void. A Null message has no payload, i.e. it has no value and is the same as Java's null; an Empty message is a tuple of zero length, similar to an empty list; a Void message however doesn't exist! Okay, that needs some justification. If I create a tuple from two valid messages and a void message then I will have a tuple of size 2. I.e. it's as if it doesn't exist. Void messages are of greatest use when iterating across a set of messages:

```
[ read from "text:Hello", read from "text:World",  read from "text:1", read from "text:
2" ];
each { if  "regex:(\\w)" };
```

This will create a tuple containing two messages (hello and world). How? The if instruction in Einstein without arguments means pass through the current message if the condition is true otherwise return a void message. So we use a regular expression condition that says match a word. The void message which has no value is then not added to the new tuple created by each. Void messages still form part of an execution flow they just don't have a value. If you try to write a void message nothing will happen, it doesn't exist.

Exception Handling

Release 0.2 - 0.3

Idempotent Code Blocks

```
idempotent { ... }
```

An idempotent code block will be used to identify part of the code as idempotent, i.e. it is possible to run the code multiple times without obtaining a different result. Exception Models will be required to provide a strategy for dealing with exceptions within retriable blocks; usually this strategy will be to look at the type of exception raised and from the exception type determine whether the block should be retried (i.e. is it a communication failure) or bubbled up to a catch Instruction. There is no catch clause because an idempotent block is not an error handling block but a hint to the compiler that it is possible to retry execution if an error occurs.

Retry Blocks

```
retry {...}
```

A retry block will allow re-execution upon any exception message that matches - usually this check will be for IO exceptions), the compiler should check that everythin the retry block is idempotent i.e. it's an explicit idempotent block or all the instructions are determined to be idempotent (e.g. < "text:hello" etc..). If a retry block is placed around code that cannot be determined to be idempotent a compiler error should occur. The user then needs to wrap explicit idempotent blocks around code the compiler cannot determine as idempotent code. Providers will be required to provide a means for identifying which resources can be identified statically as idempotent.

Catchable Code Blocks

```
try { ... } [catch { .... }] [finally { ....}]
```

If an error occurs in the try part, the current Exception Model may, depending on other exception handlers within the block, be passed the query-resource and the catch block to determine what further action to take; the finally block will always be executed. If the query-resource matches the Exception Message that was generated by the Exception Model then the Exception Message will be passed into the Catch Block, an Exception Message is similar to a normal Message (i.e. it implements Message and MessageGroup).

The finally block will always be called it's input is a null Message to avoid confusion.

Transactional Blocks

```
transact { ... } [catch { ....}] [oncommit {...}]
```

Transactional blocks exist to provide ACID behaviors in Einstein, they also have behavior that relates to error handling. If an exception occurs during the execution of the main block the transaction will be rolled back and the Exception Message passed to the catch block in the same way as try/catch, if the transaction is committed succesfully the result of the block is passed to the oncommit block.

Appendix A: Using the Compiler

Getting help from the compiler

One of the most important things you need when learning a new language or new application is access to quick simple explanations of how each part of it works.

Einstein enforces the supplying of annotation based metadata on instructions and providers, that includes third party providers. That annotation information is then available to the compiler and runtime. This means, when you get a parser error you can be given the low down of how you should use the instruction or, as we will see shortly, you can use the compiler as a 'Man page' style helps system.

In the bin directory type **./ec,** this will list compiler options which look like:

```
usage: ec
 -C,--noclass                    Do not compile to class files
 -cp,--classpath              Specify where to find user class files
 -d,--directory               Specify where to place generated class
                                 files
 -ih,--instruction-help       Provides information on how to use a given
                                 provider e.g. -ph text .
 -li,--list-instructions         Lists all available providers.
 -lp,--list-providers            Lists all available providers.
 -ph,--provider-help          Provides information on how to use a given
                                 provider e.g. -ph text .
 -sp,--sourcepath             Specify where to find input source files
 -t,--temppath                Specify where to place temporary
                                 intermediate files
 -T,--notemp                     Do not create temporary intermediate files
 -X,--stacktrace              Stacktraces are displayed when errors
                                 occur.
```

The options we're interested are the help options, so let's look and see what instructions are available right now:

```
> ec -li


flatten          - Flatten a tuple.
choose           - Choose from instructions in a named tuple.
extract          - Extract data from a payload using the supplied query.
append           - Appends the results of an execution to the current tuple.
execute          - Execute a resource.
poll             - Polls for messages on a resource, that match the (optional) query.
write            - Blocking write to a resource.
listen           - Listens for messages on a resource, that match the (optional) query.
check            - Destructive non-blocking read from a resource.
```

```
read              - Destructive blocking read from a resource.
poll              - Swaps the current message with a resource.
if                - Conditional execution.
add               - Adds the results of multiple exeuctions into a tuple.
clear             - Produces a new empty tuple.
each              - Iterates over a tuple.
using             - Used to apply meta instructions to an execution group.
dispatch          - Asynchronous non-blocking write to a resource.
while             - Conditional looped execution.
split             - Splits a message and (optionally) iterates over the result.
browse            - Non-destructive read from a resource.
merge             - Merge multiple element tuples into a single one.
resource          - Declare a resource based on a URI.
route             - Routes to instructions in a named tuple.
```

So now we know the instructions, less ask for some help on using while:

```
> ec --instruction-help while


Instruction 'while'


Description
-----------


  If the boolean query supplied evalautes to true then the first supplied group is
executed, the result of the group execution is the input for the condition and so on.


Syntax
------


  'while'


Example
-------


  while "xpath://thing[size > 5]" {write to bigThings; read from thingResource};


Result
-------


The result of the last loops execution, or the inbound message if it is never executed.
```

Great, but what about the providers? Let's list them too:

```
> ec --list-providers


camel             - Supports the providers supplied by the Camel ESB.
stack             - This is a named local in memory stack.
xpath             - Allows messages to be queried using XPath/JXPath.
```

```
java             - A Provider for java classes.
bool             - Support for boolean true and false.
esper            - Provies CEP functionality using the Esper libraries.
text             - Used to create textual messages.
time             - Time related functions, including listening and receiving.
```

Well there will be more soon :-), okay but at least we know which are available, so let's dig further and find out more about the java provider:

```
> ec --provider-help java

Description
-----------

  A Provider for java classes.

Syntax
------

  'java:'  [ '?' ( '=' ) ]

Example
-------

  execute "java:org.me.MyService?initialValue=10";

Facades
-------

These Facades will always be honored by the provider:

org.cauldron.einstein.api.provider.facade.ReadFacade
org.cauldron.einstein.api.provider.facade.WriteFacade
org.cauldron.einstein.api.provider.facade.ExecuteFacade

This provider can support all Facades, but it depends on the URI.
```

Appendix B: The Instruction Set

The instruction set in Einstein is slowly increasing as design and implementation work moves forwards on the project. This section is not guaranteed to be up to date, the compiler itself provides help on instructions and their usage.

Transmission/Storage Instructions

When an **all** qualifier is applicable, this means read all messages not just the latest one.
The **payload** qualifier, means deal with just the payload not the whole message.

Read (Blocking) <<

Syntax

```
"read" [ "all ] [ "payload" ] ["from"] <uri>
[<expression-group>] "<<" <uri>
```

Description

Requests a message from a resource and blocks until a result is obtained.

Result

A message from the specified resource.

Read (Blocking With Timeout) << :

Syntax

```
"read" [ "all ] [ "payload" ] ["from"] <uri> [ <timeout> ]
[<execution-group>] "<<" <uri> [ ":" <timeout> ]
```

Example

```
read "http://www.myserver.com/Service"  2000 ;
( write  "console:" ) <<  "http://www.myserver.com/Service"  : 2000 ;
```

Description

Requests a message from a resource and blocks until a result is obtained or the timeout is reached.

Result

Returns a null message if timed-out.

Check (Non-blocking Read) <&

Syntax

```
"check" [ "all ] [ "payload" ] <uri>
[<execution-group>] "<&" <uri>
```

Example

```
check "jms://in";
```

Description

Requests a message from a resource returns an empty message if nothing available yet,

Result

A message from the specified resource or a null message if non available.

Browse (Non-destructive Non Blocking Read) <?

Syntax

```
"browse" [ "all" ] [ "payload" ] <uri>
[<execution-group>] "<??" <uri>
```

Example

```
browse "jms://in";
```

Description

Requests all messages from a resource returns a null message if nothing available yet, read is non-destructive.

Result

A message tuple from the specified resource or a null message if non available.

Listen <!

Syntax

```
"listen" [ "payload" ] <uri> [ <query-uri> ] <expression-group>
[ <execution-group> ] "<!" <uri> ":" <execution-group>
```

Example

```
 listen esper "text:select avg(value) from
org.cauldron.einstein.ri.examples.esper.TessaractWidget.win:time(30 sec)" {
         -> "xpath:/underlying" ;
         write payload to "console:Tesseract average value is: ";

    };
```

Description

Listens for messages on a URI, that match the query supplied - if any, and executes the supplied execution group with the message when it replies. Once the listener is registered execution continues immediately with the next instruction. Once registered messages will be passed to the execution group until the resource is stopped.

Result

The current message.

Write (Blocking with Timeout) >>

Syntax

```
"write" [ "all ] [ "payload" ] <uri> [ <timeout> ]
[<execution-group>] ">>" <uri> [ ":" <timeout> ]
```

Description

Sends a message to a resource, blocks awaiting a response or timeout.

Result

The response to the operation if any, if the resource does not produce responses it may choose to return the inbound message. If the timeout is reached an exception should be thrown by the resource.

Send Async (Dispatch) &>

Syntax

```
“dispatch” [ “all ] [ “payload” ] <uri> [ <execution-group> ]
[<execution-group>] “&>“ <uri> [ “:” <execution-group> ]
```

Example

```
( << “text:Hello”) &> “http://www.myserver.com/Service” : {
       write payload to “console:Result is “;
}
```

Description

Sends a message to a resource but does not await a response, an optional block can be specified which will be executed on message receipt.

Result

The inbound message.

Swap (Combined Send/Receive) <->

Syntax

```
“swap” <uri>
[<execution-group>] “<->“ <uri>
```

Example

```
(read “console:”) <-> “jms:queue”
```

Description

Takes a message from a resource using a blocking read, writes the current message to the resource, ignores any response and makes the taken message the current message.

Result

A message from the specified resource.

Tuple Instructions

Join &

Syntax

```
"add" <execution-group>
<execution-group> ("&" <execution-group>)
```

Example

```
(read "console:") & "jms:queue"
```

Description

Creates a new tuple from the contents of the current tuple and the contents of the result of executing the execution group supplied. This can be used to create a new tuple by using the n-ary form ... & ... & ... etc or to simply join two tuples together.

Result

A tuple.

Split %

Syntax

```
"split"  <query-uri> [ ":" <execution-group> ]
<execution-group> "%" <query-uri> [":" <execution-group>]
```

Example

```
( read "jms:in" ) % "xpath://item" : { execute "java:org.me.MyService" };
```

Description

Split, first converts the inbound tuple to a single message, then splits according to an XPath expression queried against the payload. If an optional execution group is supplied then it is executed against each generated message - execution order is not guaranteed. The resulting messages are then combined to form a tuple.

Result

A tuple contain the split (and optionally post-processed) messages.

Merge ~

Syntax

```
"merge"
"~"  <execution-group>
"(~)"
```

Example

```
~[<< “jms:in1”, << “jms:in2”];
```

Description

Merges a tuple into a single message, the format of the payload in the merged message is dependent on the current data model, in an object graph data model the result would be a java.util.List for example.

Result

A single message containing the result of merging a tuple.

Each ...

Syntax

```
“each” <execution-group>
<execution-group> “...” <execution-group>
```

Example

```
[ read “text:Hello” ,  read “text:Goodbye” ] ... {
       write “console: I say “;
}
```

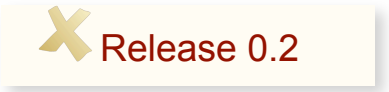
Description

Iterate over a tuple, executing the supplied execution group with the tuple as input.

Result

The result of executing the execution group.

Give >>>



Syntax

```
“give” <map-or-tuple-execution-group>
<execution-group> “>>>” <map-or-tuple-execution-group>
```

Example

```
[ read “text:World”, read “text:Cruel World” ] >>> [ write “console:Hello ”,
write “console:Goodbye”]
```

Description

Each element of the current tuple is passed to it’s equivalent element in the tuple or map execution group, since a Map is just a named tuple it has the same effect. The result is the result of the tuple/map execution group.

Result

A tuple.

Flatten I-

Syntax

```
    “flatten”
    “|-” <execution-group>
```

Example

```
    |- [ read “text: Hello” , [ read “text:1”, read “text:2”]]
```

Description

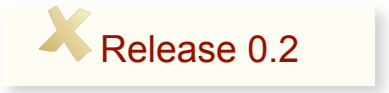
Flattens a tuple such that no element in the tuple is a tuple.

Result

A tuple which does not contain tuples.

Message Payload Operators

Insert <-



Syntax

```
"insert" [ <query-uri> ] <execution-group>
<execution-group>  "<-" [ <query-uri> ":" ] <execution-group>
```

Example

```
( read "jms:in" ) <- "xpath://price" : { < "http://mypricefeed.org/
latestPrice" } ;
```

Description

Inserts a new value into the current payload at the position described by the supplied query. If the query is not present then the payload is treated as a list, the value is then added to the list.

Result

A duplicated version of the current message, with a new payload.

Extract ->

Syntax

```
"extract" <query-uri>
<execution-group>  "->" <query-uri> ":"
```

Example

```
( read "jms:in" ) -> "xpath://size" ;
```

Description

Extracts the part of the payload specified by the query.

Result

A duplicated version of the current message, with a new payload.

Message Operators

Release 0.2

Part .

Syntax

```
<part-identifier> := “properties” | “correlation” | “security” ...
“part” <part-identifier> [ <execution-group> ]
<execution-group> “.” <part-identifier> [ “:” <execution-group> ]
```

Example

```
( read “jms:in” ) . properties : {
        > “console:These are the current messages properties:”;
}
>> “jms:out”;
```

Description

Creates a new message from the non-payload part of an existing message. If an optional execution group is supplied then that group acts on the message part but the next instruction will continue to work on the original message. There is no write equivalent of the part operator as only Einstein may change these values; to add or remove properties the following operators should be used.

Result

The original message if an execution group is supplied or a new message with the payload being the message part specified.

Get Property @@

Syntax

```
“get” <property-identifier> [ <execution-group> ]
<execution-group> “@@” <property-identifier> [ “:” <execution-group> ]
```

Example

```
( read “jms:in” ) @@ emailAddress : {
        write “console:The email address associated with this message is:”;
}
>> “jms:out”;
```

Description

Creates a new message from the string valued property of an existing message. If an optional execution group is supplied then that group acts on the property but the next instruction will continue to work on the original message.

Result

The original message if an execution group is supplied or a new message with the payload being the message property specified.

Set Property @=

Syntax

```
"set" <property-identifier> [ <execution-group> ]
<execution-group> "@=" <property-identifier> [ ":" <execution-group> ]
```

Example

```
( read "jms:in" ) @= emailAddress : ( read "text:neil.ellis@mangala.co.uk" );
write "jms:out";
```

Description

Changes the value of a property to be the string value of the payload of the message produced by executing the execution group, if no execution group is supplied then the property is removed instead.

Result

A copy of the old message with new properties but the same payload. (NB: Guarantees are not affected by this write as it does not change the payload itself.)

Control Flow

Execute **

Syntax

```
execute <uri>
** <uri>
```

Example

```
** "java:org.cauldron.einstein.ri.examples.esper.EventMaker";
```

Description

Executes the supplied resource, using the current invocation and provisioning models.

Result

The result of executing the supplied service.

If ?

Syntax

```
if [‘not’] <query-url> <execution-group> [ “else” <execution group> ]
<execution-group> “?” <query-url> “:” <execution-group>
```

Example

```
if “xpath:.[@thing=1]” {
     write “jms:out”;
}
```

Description

If the query supplied is true then execute the execution group, else execute the next execution group.

TERNARY USE BEWARE: This is different to how Java’s ternary operator works. The first argument after the “?” is the URI to evaluate, if it is true then the execution group is executed.

Result

The result of executing either of the supplied execution groups, if none were executed then the current message.

While

Syntax

```
while [‘not’] <query-url> <execution-group>
```

Example

```
while “range:1-10” {
   write “ console:Hello World!”;
}
```

Description

Repeatedly executes an instruction group until the result matches the query.

Result

The result of the last invocation of the execution group.

Route

Syntax

```
route <router-uri> <map-execution-group>
<execution-group> “=>” <router-uri> “:” <map-execution-group>
```

Example

```
( < "text:red") => "java:org.cauldron.einstein.ri.test.ByPayloadRouter" :
[  red : >> "console:Roses are ", blue : >> "console: Violets are "];
```

Description

Passes execution control to the specified resource, the resource is handed a map of executables it can choose to execute in any way it sees fit.

NB: This may be removed from the language by version 1.0, other language constructs should really provide all possible functionality that could be placed in a **route** instruction.

Result

Dependent on the router implementation.

Tuple Conditionals

Release 0.2

Any +?

Syntax

```
"any"  <query-group> <execution-group>
<execution-group> "+?" <query-group> ":" <execution-group>
```

Example

```
+? ("xpath:/.[@color='red']", "xpath:/.[@color='blue']")  : {
        > "console: My favorite color is:";
};
```

Description

If any of the query group is true then execute the execution group with the current message

Result

The result of executing the execution group.

None !?

Syntax

```
"none"  <query-group> <execution-group>
<execution-group> "!?" <query-group> ":" <execution-group>
```

Example

```
!? ("xpath:/.[@color='red']", "xpath:/.[@color='blue']")  : {
        > "console: My favorite color is neither red or blue it is:";
};
```

Description

If none of the query group is true then execute the execution group with the current message

Result

The result of executing the execution group.

One ??

Syntax

```
"one"  <query-group> <execution-group>
<execution-group> "??" <query-group> ":" <execution-group>
```

Example

```
    ?? (“xpath:/.[@color=’red’]”, “xpath:/.[@color=’blue’]”)  : {
           > “console: My favorite color is:”;
    };
```

Description

If one and only one of the query group is true then execute the execution group with the current message

Result

The result of executing the execution group.

Some *?

Syntax

```
    “some”  <query-group> <execution-group>
    <execution-group> “*?” <query-group> “:” <execution-group>
```

Example

```
    *? (“xpath:/.[@color=’red’]”, “xpath:/.[@color=’blue’]”)  : {
           > “console:Impossible, how can it be more than one in this example!!”;
    };
```

Description

If more than one of the query group is true then execute the execution group with the current message

Result

The result of executing the execution group.

All &?

Syntax

```
    “all”  <query-group> <execution-group>
    <execution-group> “&?” <query-group> “:” <execution-group>
```

Example

```
    &? (“xpath:/.[@height < 140]”, “xpath:/.[@eyeColor=’blue’]”,“xpath:/.
    [@hair=’blonde’]” )  : {
           > “console: You are a small, blue-eyed blonde:”;
    };
```

Description

If all of the query group is true then execute the execution group with the current message

Result

The result of executing the execution group.