Dependency Graph Programming Guide

Fabric Engine Version 1.2.0-beta

Table of Contents

List of Examples

Chapter 1. Introduction

Fabric Engine is a platform for enabling high-performance computing inside of dynamic languages, running from both the command line and in the browser. A Fabric Engine application can enable its high-performance components by creating and manipulating dependency graphs and associated event graphs using the dynamic language interface that Fabric Engine provides. In many cases, such as through use of Fabric's JavaScript or Python scene graph, this work is done by a higher-level framework that provides specific functionality to applications; however, in some cases it may be necessary to work directly with the core of Fabric Engine to extend these frameworks, create new frameworks, or to do lower level computation.

This document explains the concepts that are central to Fabric Engine's dependency graph model of parallel computation, and explains in detail how to work directly with this model of Fabric Engine.

1.1. Playing with the Fabric Engine Core

The Fabric Engine *core* refers to the lowest-level access to Fabric Engine that is available to a dynamic language, as opposed to higher-level interfaces such as through the JavaScript and Python scene graphs. The Fabric Engine core is manipulated using a JavaScript or Python interface, and the easiest way to learn how to work directly with the dependency graph is through this interface. This can be done either using the browser's JavaScript console or using the Fabric Engine module for Node.js or Python.

1.1.1. Using the Browser's JavaScript Console

A simple Fabric shell wrapper is provided at `http://demos.fabric-engine.com/Core/shell.html` Go to this link (if prompted to install the Fabric extension then follow the instructions and reload the page), then open the JavaScript console. In Chrome, this is done by choosing Developer → JavaScript Console from the menu bar; in Firefox, you must install the Firebug extension to access a JavaScript console. Once you have the JavaScript console running, you can start executing JavaScript commands to drive Fabric.

1.1.2. Using the Fabric Engine Module for Node.js or Python

Follow the instructions at `http://documentation.fabric-engine.com/latest/FabricEngine-LanguageBindingsReference.pdf` to install the Fabric Engine Node.js and/or Python module on your system and use Fabric Engine from the command line.

1.2. Examples

This document includes lots of examples showing Fabric core commands and the resulting output. The commands are presented as if entered on the Node.js (for JavaScript) or Python command line:

Example 1.1. An example

JavaScript

```
> console.log('Hello, world!');
Hello, world!
undefined
>
```

Python

```
>>> print "Hello, world!"
Hello, world!
>>>
```

1.3. The FABRIC Object

The Fabric shell wrapper running in a web browser provides you with a global object called `FABRIC` through which you can manipulate the core. The same object can be obtained as follows in Node.js and Python:

Example 1.2. Obtaining the FABRIC object

JavaScript:

```
> FABRIC = require('Fabric').createClient()
[FABRIC] Fabric Engine version 1.0.22-release
[FABRIC] Searching extension directory '/Users/pzion/Library/Fabric/Exts'
[FABRIC] [ExceptSample] Extension registered
...
[FABRIC] Searching extension directory '/Library/Fabric/Exts'
[FABRIC] Warning: unable to open extension directory '/Library/Fabric/Exts'
{ build:
   { isExpired: [Function],
     getName: [Function],
     getPureVersion: [Function],
     getFullVersion: [Function],
...
  close: [Function],
  getMemoryUsage: [Function],
  swapFabricClient: [Function] }
>
```

Python:

```
>>> import fabric
[FABRIC] Fabric Engine version 1.0.22-release
>>> FABRIC = fabric.createClient()
[FABRIC] Fabric Engine version 1.0.22-release
[FABRIC] Searching extension directory '/Users/pzion/Library/Fabric/Exts'
[FABRIC] [ExceptSample] Extension registered
...
[FABRIC] Searching extension directory '/Library/Fabric/Exts'
[FABRIC] Warning: unable to open extension directory '/Library/Fabric/Exts'
>>>
```


Chapter 2. Concepts

In traditional software development, programmers create data structures and write functions to manipulate the data. In order for the program to perform operations in parallel, thereby taking advantage of multi-core CPUs, the programmer must call functions within the programming language itself to schedule the execution of the parallel pieces of code.

In a Fabric Engine application, no explicit scheduling of execution is needed; the parallelism is instead expressed through higher-level models. One of these models is the *dependency graph* model, the subject of this book. A Fabric Engine application builds a dependency graph that describes the computation it needs perform; the Fabric Engine core then automatically executes in parallel operations which are not interdependent. This model requires that the user express data dependencies rather than have them implicit in the program itself. By analyzing these interdependencies, the Fabric code creates an execution schedule that runs independent parts of the computation in parallel; in this way, Fabric achieves *task-based* parallelism.

In addition, Fabric supports the notion of *slicing* data. A *Node* in Fabric is a generic, typed data container that has one or more members that contain data; each Node also has a *slice count* N, and the Node acts as if it were N independent copies of the same data that are operated on in parallel. In this way, Fabric achieves *data-based* (or "SIMD") parallelism. For more information on Nodes, members, and slice counts, see Chapter 4, *Nodes*.

In addition to the dependency graph, Fabric provides a method of traversing the Nodes in the dependency graph through objects called Events and EventHandlers. The Fabric SceneGraph uses Events and EventHandlers to draw OpenGL viewports in its rendering system. For more information on Events and EventHandlers, see Chapter 5, *Event Graphs, Events and EventHandlers*.

The actual code that performs computation in Fabric is contained in objects called *Operators*. Operators can then be *bound* to Nodes and EventHandlers using glue objects called Bindings; Bindings tell Fabric what data should be passed to the functions defined in the code in an Operator. For more information on Operators, see Chapter 6, *Operators and Bindings*.

Chapter 3. Registered Types

Fabric supports a *registered types* system whereby user-defined compound types (ie. structures) are defined through JavaScript. Once a type has been registered, it can be used to define members of Nodes and EventHandlers (see below) as well as used in KL code.

To register a new type, call the `FABRIC.RT.registerType` function with the name of the type as the first parameter and the *specification* object as the second parameter. The specification object has the following members:

members
: An object containing the members to be contained in the type. The key names are the member names for the type and the key values are the names of already-registered types or built-in KL types (see the KL Programming Guide [http://documentation.fabric-engine.com/latest/FabricEngine-KLProgrammingGuide.pdf] for more information on atomic types).

It is possible to append brackets to obtain variable- or fixed-length arrays as members, eg. `Scalar[2][2]`, `Scalar[][]` and `Scalar[2][][4]`.

constructor
: A JavaScript or Python constructor that is used to provide the JavaScript or Python "prototype" for objects values returned from the Fabric core, as well as to provide a default value if none is given.

defaultValue
: (optional) The default value for the type.

klBindings
: (optional) KL code to include which provides operations involving the type, such as constructors, methods and operators

It is expected that KL source code to be bound to types will usually be loaded from another web resource, eg. through the `FABRIC.loadResourceURL` function. However, in the example below we use an explicit string to illustrate that a string is what the core needs.

To get an object containing information about all the currently-registered types, call `FABRIC.RT.getRegisteredTypes()`.

Example 3.1. Registered types

JavaScript:

```
// Registered types
> Vec3 = function(x, y, z) {
  if (typeof x === 'number' && typeof y === 'number' && typeof z === 'number') {
    this.x = x;
    this.y = y;
    this.z = z;
  }
  else if (x === undefined && y === undefined && z === undefined) {
    this.x = 0;
    this.y = 0;
    this.z = 0;
  }
  else throw'new Vec3: invalid arguments';
};

> vec3KLBindings = "\n\
// Construct a Vec3 from three Scalars\n\
function Vec3(Scalar x, Scalar y, Scalar z) {\n\
  this.x = x;\n\
  this.y = y;\n\
  this.z = z;\n\
}\n\
```

```
\n\
// Add two Vec3s\n\
function Vec3 + (Vec3 a, Vec3 b) {\n\
  return Vec3(a.x + b.x, a.y + b.y, a.z + b.z);\n\
}\n\
";

> FABRIC.RT.registerType('Vec3', {
  members: [{x: 'Scalar'}, {y: 'Scalar'}, {z: 'Scalar'}],
  constructor: Vec3,
  klBindings: {
    filename: "inline",
    sourceCode: vec3KLBindings
  }
});

> FABRIC.RT.getRegisteredTypes().Vec3;
{ name: 'Vec3',
  size: 12,
  defaultValue: { x: 0, y: 0, z: 0 },
  internalType: 'struct',
  members:
   [ { name: 'x', type: 'Scalar' },
     { name: 'y', type: 'Scalar' },
     { name: 'z', type: 'Scalar' } ] }
>
```

Python:

```
>>> # Registered types
class Vec3():
  def __init__( self, x = None, y = None, z = None ):
    if type( x ) is float and type( y ) is float and type( z ) is float:
      self.x = x
      self.y = y
      self.z = z
    elif x is None and y is None and z is None:
      self.x = 0
      self.y = 0
      self.z = 0
    else:
      raise Exception( 'Vec3: invalid arguments' )

>>> vec3KLBindings = """
// Construct a Vec3 from three Scalars
function Vec3(Scalar x, Scalar y, Scalar z) {
  this.x = x;
  this.y = y;
  this.z = z;
}
// Add two Vec3s
function Vec3 + (Vec3 a, Vec3 b) {
  return Vec3(a.x + b.x, a.y + b.y, a.z + b.z);
}
"""

>>> vec3TypeDesc = {
  'members': [{'x': 'Scalar'}, {'y': 'Scalar'}, {'z': 'Scalar'}],
  'constructor': Vec3,
  'klBindings': {
    'filename': "(inline)",
    'sourceCode': vec3KLBindings
   }
}

>>> FABRIC.RT.registerType('Vec3', vec3TypeDesc)

>>> FABRIC.RT.getRegisteredTypes()['Vec3']
```



```
   {u'defaultValue': {u'y': 0, u'x': 0, u'z': 0}, u'internalType': u'struct', u'name': u'Vec3',
u'members': [{u'type': u'Scalar', u'name': u'x'}, {u'type': u'Scalar', u'name': u'y'}, {u'type':
u'Scalar', u'name': u'z'}], u'size': 12}
   >>>
```

Chapter 4. Nodes

The fundamental unit of the Fabric dependency graph is called a *Node*. A Node contains data and has a list of Operators, written in the KL programming language, that manipulate the data.

4.1. Node Creation

Each node must have a unique name which is specified when it node is created. The name of the node must not conflict with the name of any Event, EventHandler or Operator (see below). To create a node, call `FABRIC.DG.createNode`.

Example 4.1. Node creation

JavaScript:

```
> node = FABRIC.DG.createNode( "vertices" );
{ getName: [Function],
  getErrors: [Function],
...
  evaluateAsync: [Function],
  bindings:
   { empty: [Function],
     getLength: [Function],
     getOperator: [Function],
     append: [Function],
     insert: [Function],
     remove: [Function] } }
>
```

Python:

```
>>> node = FABRIC.DG.createNode('vertices')
>>>
```

The name of a node can be retrieved through the `getName` method.

Example 4.2. Getting a node's name

JavaScript:

```
> node.getName();
'vertices'
>
```

Python:

```
>>> node.getName()
'vertices'
>>>
```

4.2. Node Members

A Node has zero or more *members*. Each member has a name (a non-empty string), a type (referred to as the name of a registered type), and, optionally, a default value. Members can be added to nodes with the `addMember` method and an object with details of all the members can be retrieved with the `getMembers` method.

Example 4.3. Adding and getting node members

JavaScript:

```
> node.addMember("position", "Vec3", new Vec3(0, 0, 0));
undefined
> node.getMembers().position;
{ name: 'position',
  type: 'Vec3',
  defaultValue: { x: 0, y: 0, z: 0 } }
>
```

Python:

```
>>> node.addMember("position", "Vec3", Vec3(0.0, 0.0, 0.0))
>>> node.getMembers()['position']
{'defaultValue': <__main__.Vec3 instance at 0x1109727e8>, 'type': 'Vec3', 'name': 'position'}
>>>
```

Each member has a value, or in the case of a node with a slice count greater than one (see below), one value per slice. The value of a member is retrieved using the `getData` method and set using the `setData` method. Both methods take the member name as the first argument and the slice index as the second argument.

Example 4.4. `getData` and `setData`

JavaScript:

```
> node.getData('position', 0);
{ x: 0, y: 0, z: 0 }
> node.setData('position', 0, new Vec3(1, 2, 3));
undefined
> node.getData('position', 0);
{ x: 1, y: 2, z: 3 }
>
```

Python:

```
>>> vars(node.getData('position', 0))
{'y': 0, 'x': 0, 'z': 0}
>>> node.setData('position', 0, Vec3(1.0, 2.0, 3.0))
>>> vars(node.getData('position', 0))
{'y': 2, 'x': 1, 'z': 3}
>>>
```

4.3. Node Slice Counts

Each Node has a *slice count*. Setting the slice count of a node to a number greater than one enables the core to compute over an array of data (“SIMD parallelization”); each member has one value for each slice of the node. The default slice count for a node is one.

Operators can run on nodes per-slice or on all slices at once, depending on how the operator is bound to the node. This will be explained below.

The slice count for a node is set with the `setSliceCount` (or `setSize`) method and retrieved with the `getSliceCount` (or `getSize`) method.

Example 4.5. Node slice counts

JavaScript:

```
   > node.getCount();
   1
   > node.getData('position', 1);
   Fabric core exception: DG.vertices.getData('{"memberName":"position","sliceI...'): index (1)
out of range (1)
   > node.setCount(2);
   undefined
   > node.getCount();
   2
   > node.getData('position', 1);
   { x: 0, y: 0, z: 0 }
   >
```

Python:

```
   >>> node.getCount()
   1
   >>> node.getData('position', 1)
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
     File "/home/andrew/src/python_modules/fabric/__init__.py", line 883, in getData
       self._dg._executeQueuedCommands()
     File "/home/andrew/src/python_modules/fabric/__init__.py", line 460, in
_executeQueuedCommands
       self.__client.executeQueuedCommands()
     File "/home/andrew/src/python_modules/fabric/__init__.py", line 315, in
executeQueuedCommands
       raise Exception( 'Fabric core exception: ' + result[ 'exception' ] )
   Exception: Fabric core exception: DG.verticies.getData('{"sliceIndex": 1,
"memberName": ...'): index (1) out of range (1)
   >>> node.setCount(2)
   >>> node.getCount()
   2
   >>> vars(node.getData('position', 1))
   {'y': 0, 'x': 0, 'z': 0}
```

4.4. Node Dependencies

Each node has zero or more named *dependencies*; the dependency is another node. If a node A has a dependency on another node B, then all of the operators of node A will run after all of those on node B have finished running.

- The name of a dependency must be a non-empty string.
- The name of the dependency is used to bind operators to the data in the dependency node
- Each dependency of a node must have a different name
- You cannot create a dependency loop between Nodes, ie. you cannot have Node A dependent on Node B at the same time as Node B is dependent on Node A.

Dependencies are added using the `setDependency` method, and dependencies of a node are retrieved using the `getDependencies` method.

Example 4.6. Node dependencies

JavaScript:

```
> anotherNode = FABRIC.DG.createNode( "originalVertices" );
{ getName: [Function],
  getErrors: [Function],
...
     insert: [Function],
     remove: [Function] } }
> node.setDependency( anotherNode, "original" );
undefined
> node.getDependencies();
{ original:
   { getName: [Function],
     getErrors: [Function],
...
        append: [Function],
        insert: [Function],
        remove: [Function] } } }
>
```

Python:

```
>>> anotherNode = FABRIC.DG.createNode("originalVertices")
>>> node.setDependency(anotherNode, "original")
>>> node.getDependencies()
{'original': <fabric._NODE object at 0x1050ffb10>}
>>>
```

4.5. Node Evaluation

Each node is either *clean* or *dirty*. A node become dirty if any of the following happen:

- The node's `setData` method is called
- Anything about the node changes (eg. added dependencies, added members)
- Any of the node's dependencies becomes dirty

Nodes can be *evaluated*. Evaluating a node does the following: if the node is clean, nothing happens. Otherwise,

- All the dependencies of the node are evaluated
- All of the operators bound to the node are executed
- The node is marked as clean

A node can be manually evaluated by calling the `evaluate` method. Nodes are automatically evaluated when:

- A node is a dependency of another node that is evaluated
- An EventHandler (see below) has an operator bound to the data in the node, and the EventHandler is executed

Example 4.7. Node Evaluation

JavaScript:

```
   > op = FABRIC.DG.createOperator( "offsetPosition" );
   { getName: [Function],
     getErrors: [Function],
   ...
     setEntryPoint: [Function],
     getDiagnostics: [Function] }
   > op.setEntryPoint( "offset" );
   undefined
   > op.setSourceCode( "require Vec3; operator offset( io Vec3 position, io Vec3 newPosition )
{ newPosition = position + Vec3(1.0,1.0,1.0); }" );
   undefined
   > binding = FABRIC.DG.createBinding();
   { getOperator: [Function],
     setOperator: [Function],
     getParameterLayout: [Function],
     setParameterLayout: [Function] }
   > binding.setOperator(op);
   undefined
   > binding.setParameterLayout(["self.position","self.newPosition"]);
   undefined
   > node.addMember( "newPosition", "Vec3" );
   undefined
   > node.bindings.append( binding );
   undefined
   > node.getData( "position", 0 );
   { x: 1, y: 2, z: 3 }
   > node.getData( "newPosition", 0 );
   { x: 0, y: 0, z: 0 }
   > node.evaluate();
   undefined
   > node.getData( "newPosition", 0 );
   { x: 2, y: 3, z: 4 }
   >
```

Python:

```
   >>> op = FABRIC.DG.createOperator("offsetPosition")
   >>> op.setEntryPoint("offset")
   >>> op.setSourceCode("require Vec3; operator offset(io Vec3 position, io Vec3 newPosition)
{ newPosition = position + Vec3(1.0,1.0,1.0); }")
   >>> binding = FABRIC.DG.createBinding()
   >>> binding.setOperator(op)
   >>> binding.setParameterLayout(["self.position", "self.newPosition"])
   >>> node.addMember("newPosition", "Vec3")
   >>> node.bindings.append(binding)
   >>> vars(node.getData("position", 0))
   {'y': 2, 'x': 1, 'z': 3}
   >>> vars(node.getData( "newPosition", 0 ))
   {'y': 0, 'x': 0, 'z': 0}
   >>> node.evaluate();
   >>> vars(node.getData( "newPosition", 0 ))
   {'y': 3, 'x': 2, 'z': 4}
```


Chapter 5. Event Graphs, Events and EventHandlers

Fabric provides a method of traversing the Nodes in the dependency graph through objects called Events and EventHandlers. An Event can be fired, which will then fire a list of EventHandlers; each EventHandler has a list of child EventHandlers which are visited in turn, and each EventHandler has a list of *pre-descend operators* that are executed before the child EventHandlers are visited and a list of *post-descend operators* that are executed after the child EventHandlers are visit. Each event handler can also bind to Nodes in the dependency graph to access from its Operators. A typical use of Events and EventHandlers is for OpenGL rendering. When an OpenGL viewport is created, a "redraw Event" is associated with it; this event is automatically fired whenever the viewport needs to be redrawn. The EventHandlers that "chain" off of the redraw Event can then issue OpenGL calls to draw the viewport contents.

5.1. Event Creation

To create an Event, call the `FABRIC.DG.createEvent` function. Like Nodes, Events must have a unique name that is not the same as that of a Node, Operator or EventHandler. To get an existing event's name, call its `getName` method.

Example 5.1. Event creation

JavaScript:

```
> event = FABRIC.DG.createEvent("anEvent");
{ getName: [Function],
  getErrors: [Function],
...
  setSelectType: [Function],
  select: [Function] }
> event.getName();
'anEvent'
>
```

Python:

```
>>> event = FABRIC.DG.createEvent("anEvent")
>>> event.getName()
'anEvent'
```

5.2. EventHandler Creation

Event event has a list of EventHandlers attached so it. When an Event is fired, each attached EventHandler is fired in sequence. To create an EventHandler, call `FABRIC.DG.createEventHandler`. To append the EventHandler to an Event, call the Event's `appendEventHandler` method.

Example 5.2. Creating EventHandlers and appending them to Events

JavaScript:

```
> eventHandler = FABRIC.DG.createEventHandler("trivialEventHandler");
{ getName: [Function],
  getErrors: [Function],
...
     insert: [Function],
     remove: [Function] } }
> event.appendEventHandler(eventHandler);
undefined
> event.getEventHandlers();
[ { getName: [Function],
    getErrors: [Function],
...
       insert: [Function],
       remove: [Function] } } ]
>
```

Python:

```
>>> eventHandler = FABRIC.DG.createEventHandler("trivialEventHandler")
>>> event.appendEventHandler(eventHandler)
>>> event.getEventHandlers()
[<fabric._EVENTHANDLER object at 0x2cebfd0>]
```

5.3. Operators and EventHandlers

Each EventHandler has two lists of Operators (or, rather, Bindings) called `preDescendBindings` and `postDescendBindings`, as well as a list of child EventHandlers. When an Event is fired, each of its EventHandlers is visited. For each EventHandler, Bindings in `preDescendBindings` are executed in sequence, then its child EventHandlers are visited in the same way, then Bindings in `postDescendBindings` are executed in sequence.

Example 5.3. Operators and EventHandlers

JavaScript:

```
> op = FABRIC.DG.createOperator("trivialOperator");
{ getName: [Function],
  getErrors: [Function],
...
  setEntryPoint: [Function],
  getDiagnostics: [Function] }
> op.setSourceCode("operator entry() { report('Ran trivialOperator'); }");
undefined
> op.setEntryPoint('entry');
undefined
> binding = FABRIC.DG.createBinding();
{ getOperator: [Function],
  setOperator: [Function],
  getParameterLayout: [Function],
  setParameterLayout: [Function] }
> binding.setOperator(op);
undefined
> binding.setParameterLayout([]);
undefined
> eventHandler.preDescendBindings.append(binding);
undefined
> event.fire();
```



```
[FABRIC] [MT] Ran trivialOperator
undefined
> anotherOp = FABRIC.DG.createOperator("trivialOperatorTwo");
{ getName: [Function],
  getErrors: [Function],
...
  setEntryPoint: [Function],
  getDiagnostics: [Function] }
> anotherOp.setSourceCode("operator entry() { report('Ran trivialOperatorTwo'); }");
undefined
> anotherOp.setEntryPoint('entry');
undefined
> binding = FABRIC.DG.createBinding();
{ getOperator: [Function],
  setOperator: [Function],
  getParameterLayout: [Function],
  setParameterLayout: [Function] }
> binding.setOperator(anotherOp);
undefined
> binding.setParameterLayout([]);
undefined
> eventHandler.postDescendBindings.append(binding);
undefined
> event.fire();
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] Ran trivialOperatorTwo
undefined
>
```

Python:

```
>>> op = FABRIC.DG.createOperator("trivialOperator")
>>> op.setSourceCode("operator entry() { report('Ran trivialOperator'); }")
>>> op.setEntryPoint('entry')
>>> binding = FABRIC.DG.createBinding()
>>> binding.setOperator(op)
>>> binding.setParameterLayout([])
>>> eventHandler.preDescendBindings.append(binding)
>>> event.fire()
[FABRIC] [MT] Ran trivialOperator
>>> anotherOp = FABRIC.DG.createOperator("trivialOperatorTwo")
>>> anotherOp.setSourceCode("operator entry() { report('Ran trivialOperatorTwo'); }")
>>> anotherOp.setEntryPoint('entry')
>>> binding = FABRIC.DG.createBinding()
>>> binding.setOperator(anotherOp)
>>> binding.setParameterLayout([])
>>> eventHandler.postDescendBindings.append(binding)
>>> event.fire()
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] Ran trivialOperatorTwo
>>>
```

EventHandlers in turn can have child EventHandlers. The child EventHandlers of a given EventHandler are fired, in order, after the pre-descend operators are executed and before the post-descend operators are executed. Child EventHandlers are added by calling the EventHandler's `appendChildEventHandler` method:

Example 5.4. Child EventHandlers

JavaScript:

```
> binding = FABRIC.DG.createBinding();
{ getOperator: [Function],
  setOperator: [Function],
  getParameterLayout: [Function],
  setParameterLayout: [Function] }
```

```
> binding.setOperator(op);
undefined
> binding.setParameterLayout([]);
undefined
> ceh = FABRIC.DG.createEventHandler("childEventHandler");
{ getName: [Function],
  getErrors: [Function],
...
     insert: [Function],
     remove: [Function] } }
> ceh.preDescendBindings.append(binding);
undefined
> eventHandler.appendChildEventHandler(ceh);
undefined
> event.fire();
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] Ran trivialOperatorTwo
undefined
>
```

Python:

```
>>> binding = FABRIC.DG.createBinding()
>>> binding.setOperator(op)
>>> binding.setParameterLayout([])
>>> ceh = FABRIC.DG.createEventHandler("childEventHandler")
>>> ceh.preDescendBindings.append(binding)
>>> eventHandler.appendChildEventHandler(ceh)
>>> event.fire()
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] Ran trivialOperatorTwo
```

EventHandlers can access data stored in Nodes by using their `setScope` method. Once a Node is bound to an EventHandler, that Node is guaranteed to be evaluated (if it is dirty) before any Event that could fire the EventHandler is fired. The name given in the `setScope` method is also available to child EventHandlers, their children, and so on, for binding. If a child EventHandler binds a scope with the same name, it overrides the parent's scope.

Example 5.5. The `setScope` method

JavaScript:

```
> node = FABRIC.DG.createNode("someNode");
{ getName: [Function],
  getErrors: [Function],
...
     insert: [Function],
     remove: [Function] } }
> node.addMember( "x", "Scalar" );
undefined
> node.addMember( "y", "Scalar" );
undefined
> squareOp = FABRIC.DG.createOperator("squareOp");
{ getName: [Function],
  getErrors: [Function],
...
  setEntryPoint: [Function],
  getDiagnostics: [Function] }
> squareOp.setSourceCode("operator entry( io Scalar x, io Scalar y ) { y = x * x; }");
undefined
> squareOp.setEntryPoint("entry");
undefined
> binding = FABRIC.DG.createBinding();
{ getOperator: [Function],
```



```
     setOperator: [Function],
     getParameterLayout: [Function],
     setParameterLayout: [Function] }
   > binding.setOperator(squareOp);
   undefined
   > binding.setParameterLayout(['self.x','self.y']);
   undefined
   > node.bindings.append(binding);
   undefined
   > displayOp = FABRIC.DG.createOperator("displayOp");
   { getName: [Function],
     getErrors: [Function],
   ...
     setEntryPoint: [Function],
     getDiagnostics: [Function] }
   > displayOp.setSourceCode( "operator entry( io Scalar x, io Scalar y ) { report(x + ' squared
is ' + y); }" );
   undefined
   > displayOp.setEntryPoint("entry");
   undefined
   > binding = FABRIC.DG.createBinding();
   { getOperator: [Function],
     setOperator: [Function],
     getParameterLayout: [Function],
     setParameterLayout: [Function] }
   > binding.setOperator(displayOp);
   undefined
   > binding.setParameterLayout(['mynode.x','mynode.y']);
   undefined
   > eventHandler.setScope("mynode",node);
   undefined
   > eventHandler.postDescendBindings.append(binding);
   undefined
   > event.fire();
   [FABRIC] [MT] Ran trivialOperator
   [FABRIC] [MT] Ran trivialOperator
   [FABRIC] [MT] Ran trivialOperatorTwo
   [FABRIC] [MT] 0 squared is 0
   undefined
   > node.setData('x',5.0);
   undefined
   > event.fire();
   [FABRIC] [MT] Ran trivialOperator
   [FABRIC] [MT] Ran trivialOperator
   [FABRIC] [MT] Ran trivialOperatorTwo
   [FABRIC] [MT] 5 squared is 25
   undefined
   >
```

Python:

```
   >>> node = FABRIC.DG.createNode("someNode")
   >>> node.addMember( "x", "Scalar" )
   >>> node.addMember( "y", "Scalar" )
   >>> squareOp = FABRIC.DG.createOperator("squareOp")
   >>> squareOp.setSourceCode("operator entry( io Scalar x, io Scalar y ) { y = x * x; }")
   >>> squareOp.setEntryPoint("entry")
   >>> binding = FABRIC.DG.createBinding()
   >>> binding.setOperator(squareOp)
   >>> binding.setParameterLayout(['self.x','self.y'])
   >>> node.bindings.append(binding)
   >>> displayOp = FABRIC.DG.createOperator("displayOp")
   >>> displayOp.setSourceCode( "operator entry( io Scalar x, io Scalar y ) { report(x + '
squared is ' + y); }" )
   >>> displayOp.setEntryPoint("entry")
   >>> binding = FABRIC.DG.createBinding()
   >>> binding.setOperator(displayOp)
   >>> binding.setParameterLayout(['mynode.x','mynode.y'])
```

```
>>> eventHandler.setScope("mynode",node)
>>> eventHandler.postDescendBindings.append(binding)
>>> event.fire()
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] Ran trivialOperatorTwo
[FABRIC] [MT] 0 squared is 0
>>> node.setData('x',5.0)
>>> event.fire()
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] Ran trivialOperatorTwo
[FABRIC] [MT] 5 squared is 25
>>>
```

5.4. EventHandler Data

EventHandlers themselves can also have data, and they set the name of their own scope, as seen by child EventHandlers, through `setScopeName`:

Example 5.6. EventHandler data

JavaScript:

```
> eventHandler.setScopeName("childEventHandler");
undefined
> eventHandler.addMember("x","Scalar");
undefined
> eventHandler.addMember("y","Scalar");
undefined
> binding = FABRIC.DG.createBinding();
{ getOperator: [Function],
  setOperator: [Function],
  getParameterLayout: [Function],
  setParameterLayout: [Function] }
> binding.setOperator(squareOp);
undefined
> binding.setParameterLayout(['childEventHandler.x','childEventHandler.y']);
undefined
> ceh.preDescendBindings.append(binding);
undefined
> binding = FABRIC.DG.createBinding();
{ getOperator: [Function],
  setOperator: [Function],
  getParameterLayout: [Function],
  setParameterLayout: [Function] }
> binding.setOperator(displayOp);
undefined
> binding.setParameterLayout(['childEventHandler.x','childEventHandler.y']);
undefined
> ceh.preDescendBindings.append(binding);
undefined
> event.fire();
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] 0 squared is 0
[FABRIC] [MT] Ran trivialOperatorTwo
[FABRIC] [MT] 5 squared is 25
undefined
> eventHandler.setData("x",7.31);
undefined
> event.fire();
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] 7.31 squared is 53.4361
```



```
[FABRIC] [MT] Ran trivialOperatorTwo
[FABRIC] [MT] 5 squared is 25
undefined
>
```

Python:

```
>>> eventHandler.setScopeName("childEventHandler")
>>> eventHandler.addMember("x","Scalar")
>>> eventHandler.addMember("y","Scalar")
>>> binding = FABRIC.DG.createBinding()
>>> binding.setOperator(squareOp)
>>> binding.setParameterLayout(['childEventHandler.x','childEventHandler.y'])
>>> ceh.preDescendBindings.append(binding)
>>> binding = FABRIC.DG.createBinding()
>>> binding.setOperator(displayOp)
>>> binding.setParameterLayout(['childEventHandler.x','childEventHandler.y'])
>>> ceh.preDescendBindings.append(binding)
>>> event.fire()
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] 0 squared is 0
[FABRIC] [MT] Ran trivialOperatorTwo
[FABRIC] [MT] 5 squared is 25
>>> eventHandler.setData("x",7.31)
>>> event.fire()
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] Ran trivialOperator
[FABRIC] [MT] 7.31 squared is 53.4361
[FABRIC] [MT] Ran trivialOperatorTwo
[FABRIC] [MT] 5 squared is 25
>>>
```


Chapter 6. Operators and Bindings

The KL code that runs within the dependency graph and event graph is contained in objects called Operators. The same operator can be bound to multiple Nodes and EventHandlers by using different Bindings (see below) that share the same Operator.

6.1. Operator Creation

An Operator is created by calling `FABRIC.RT.createOperator`, passing the name of the Operator as the first argument. Operator names must be unique and not shared with any Nodes, Events or EventHandlers.

Example 6.1. Operator creation

JavaScript:

```
> op = FABRIC.DG.createOperator( "doSomething" );
{ getName: [Function],
  getErrors: [Function],
...
  setEntryPoint: [Function],
  getDiagnostics: [Function] }
>
```

Python:

```
>>> op = FABRIC.DG.createOperator("doSomething")
>>>
```

6.2. Setting Operator Source Code

The source code contained in the Operator is set using the Operator's `setSourceCode` method. It takes a string containing the KL source code.

Source code is first usually loaded from an external resource, using eg. the `FABRIC.loadResourceURL` function, rather than being included as an inline string as is done in these examples.

After setting the source code, you can check if any warnings or errors were generated by the KL compiler by calling the `getDiagnostics` method, which returns an array of objects describing the warnings/errors, including the level (warning/error), line number, column number and message. You can later retrieve the source code by calling the `getSourceCode` method.

Example 6.2. Setting operator source code

JavaScript:

```
> op.setSourceCode("operator entry( io Scalar result, in Size index, in Container self )
{ result = 3.14 }");
undefined
> op.getDiagnostics();
[ { filename: '(unknown)',
    line: 1,
    column: 82,
```

```
        level: 'error',
        desc: 'syntax error, unexpected }, expecting ;' } ]
   > op.setSourceCode("operator entry( io Scalar result, in Size index, in Container self )
{ result = 3.14; }");
   undefined
   > op.getDiagnostics();
   []
   >
```

Python:

```
   >>> op.setSourceCode("operator entry( io Scalar result, in Size index, in Container self )
{ result = 3.14 }")
   >>> op.getDiagnostics()
   [{u'column': 82, u'line': 1, u'level': u'error', u'desc': u'syntax error, unexpected },
expecting ;', u'filename': u'(unknown)'}]
   >>> op.setSourceCode("operator entry( io Scalar result, in Size index, in Container self )
{ result = 3.14; }")
   >>> op.getDiagnostics()
   []
   >>>
```

6.3. Setting the Operator Entry Point

In addition to source code, an operator needs an entry point, which is the name of the KL operator (see the KL Programming Guide [http://documentation.fabric-engine.com/latest/FabricEngine-KLProgrammingGuide.pdf]) in the source code that should be called when the operator is invoked. Note that this *must* be an KL operator and not a KL function. The entry point is specified by calling the `setEntryPoint` method. By specifying the source code and entry point separately, it is possible to have multiple possible entry points into the same source code.

Example 6.3. Setting the operator entry point

JavaScript:

```
   > op.setEntryPoint('entry');
   undefined
   > op.getEntryPoint();
   'entry'
   >
```

Python:

```
   >>> op.setEntryPoint('entry')
   >>> op.getEntryPoint()
   'entry'
   >>>
```

6.4. Bindings

To make an operator run on a Node or EventHandler, you must create a Binding object which describes what data the KL operator arguments are bound to when the operator is run.

It is possible to have multiple bindings that all share a single operator.

A binding object is created by calling `FABRIC.DG.createBinding`, and you set the Operator called by the Binding by calling the Binding's `setOperator` method. This operator can later be retrieved by calling the Binding's `getOperator` method.

Example 6.4. Binding creation

JavaScript:

```
> binding = FABRIC.DG.createBinding();
{ getOperator: [Function],
  setOperator: [Function],
  getParameterLayout: [Function],
  setParameterLayout: [Function] }
> binding.setOperator(op);
undefined
> binding.getOperator();
{ getName: [Function],
  getErrors: [Function],
...
  setEntryPoint: [Function],
  getDiagnostics: [Function] }
>
```

Python:

```
>>> binding = FABRIC.DG.createBinding()
>>> binding.setOperator(op)
>>> binding.getOperator()
<fabric._OPERATOR object at 0x1958e50>
>>>
```

6.4.1. Binding Parameter Layouts

The way in which the KL operator arguments are bound is specified by calling the Binding's `setParameterLayout` method. `setParameterLayout` takes a single parameter that is an array of strings. The length of the array must be equal to the number of parameters taken by the KL operator in the Operator's source code, and each string describes what data that parameter should bind to.

Such string is of the format *`object.member`*, or *`object`* for a special usage which we will detail later. The *`object`* part refers to what Node, EventHandler or Event object contains the data to be bound, as follows:

- If *`object`* is `self`, the data is contained on the object where the binding is attached
- For Bindings that live on Nodes, *`object`* is the name of the direct dependency Node that contains the data
- For Bindings that live on EventHandlers, *`object`* is the name of an ancestor EventHandler in the call chain as specified by a call to its `setScopeName` method, or a Node that is connected to the EventHandler through a call to the EventHandlers's `setScope` method.

The *`member`* part refers to the data member on the object specified by *`object`*, with support for the following additional syntaxes:

- If *`member`* is simply the name of a member (eg. "position"), the parameter will be bound to that member once for each slice. The operator will be invoked once for each slice of the Node, potentially in parallel. The KL parameter in the operator must be an `io` parameter whose type is the type of the member.
- If *`member`* is the name of a member followed by `[]` (eg. `position[]`), the parameter will be bound to a variable-length array that contains the data for *all* the slices for that member. The length of the array will be equal to the

slice count of the Node. The KL parameter in the operator must be an `io` parameter whose type is a variable-length array of the type of the member.

- If *`member`* is `index`, the parameter will be index of the current slice for which the operator is being executed. The parameter must be an `in` parameter of type `Index` (or, equivalently, `Size`)
- If *`object`* is specified (instead of *`object.member`*), then the parameter must be of type `Container`, which allows you to get or set the Node slice count in KL. Calling `Container`'s `resize(Size)` method will immediately change the slice count of the Node, and the `size()` method will return its current slice count. The `resize(Size)` method requires that the parameter is specified as `io`.

Example 6.5. Binding parameters

JavaScript:

```
   > binding.setParameterLayout( ["self.result","self.index","self"] );
   undefined
   > node = FABRIC.DG.createNode("foo");
   { getName: [Function],
     getErrors: [Function],
   ...
        insert: [Function],
        remove: [Function] } }
   > node.bindings.append(binding);
   undefined
   > node.getErrors();
   [ 'binding 0: operator \'doSomething\': node \'self\': parameter 1: member \'result\':
\'result\': no such member' ]
   > node.addMember("result","Scalar");
   undefined
   > node.getErrors();
   []
   >
```

Python:

```
   >>> binding.setParameterLayout( ["self.result","self.index","self"] )
   >>> node = FABRIC.DG.createNode("foo")
   >>> node.bindings.append(binding)
   >>> node.getErrors()
   [u"binding 0: operator 'doSomething': node 'self': parameter 1: member 'result': 'result': no
such member"]
   >>> node.addMember("result","Scalar")
   >>> node.getErrors()
   []
   >>>
```

Even when a Binding binds a parameter to a member of a dependency of a Node, rather than a member of the Node itself, the parameter must still be declared as an `io` parameter in KL. This is a limitation of the system which will be removed in the future; in fact, it will become required that members of non-`self` objects be bound to `in` parameters.

