Server Programming Guide

Fabric Engine Version 1.2.0-beta

Table of Contents

List of Examples

Chapter 1. Introduction

This document outlines the installation and use of Fabric from the command line, as is typically used in a server environment. Currently, bindings exist for JavaScript using Node.js on Mac OS X and Linux, and for Python on Windows, Mac OS X, and Linux. Both Node.js and Python bindings are implemented as modules.

The rest of this document outlines instructions on how to install each module, a tutorial for each language to get you up-and-running, and links to reference documentation. The last section also outlines how to install and use the Fabric Extensions with the bindings for your chosen language.

This document focuses only on installing and configuring the specific language bindings, for a detailed reference on Fabric itself and the KL language please see `the other relevant documentation` [http://documentation.fabricengine.com/latest].

Chapter 2. Node.js Module

2.1. Supported Platforms

The Fabric Engine module for Node.js currently runs on:

- Mac OS X running on Intel processors, version 10.6 ("Snow Leopard") or later
- 32-bit and 64-bit Linux. We do our development on Ubuntu 10.04 and above but it should work with any modern distribution

2.2. Installation

2.2.1. Step Zero: Get Node.js

The first thing you'll need to do is install Node.js if you haven't already. We currently support the 0.6.x series of Node, although the module may work with later versions as well. Node.js may be available through your package manager but you'll need to make sure that it's at least version 0.6.x. Otherwise it can be installed from here: `Node.js` [http://nodejs.org/]

2.2.2. Step One: Download

From the `Fabric distributions page` [http://dist.fabricengine.com/latest/], download the version of the Node.js module that applies for your operating system.

For Linux, choose the file based on whether you are running a 32-bit or 64-bit operating system. If you're not sure which version to download, run `uname -m` which will report either `i686` or `x86_64`.

- `FabricEngine-NodeModule-Linux-i686-VERSION.tar.bz2`
- `FabricEngine-NodeModule-Linux-x86_64-VERSION.tar.bz2`

For Mac OS X choose:

- `FabricEngine-NodeModule-Darwin-universal-VERSION.tar.bz2`

2.2.3. Step Two: Install

Unpack the module by running `tar jxf FabricEngine-NodeModule-XXX.tar.bz2` from a command line.

Create the directory `~/node_modules`. Next, inside the module that you just unpacked there should be a subfolder called `node_modules` with a subfolder of its own called `Fabric`. Move this `Fabric` folder to `~/node_modules`. You can also create a symlink if it makes more sense for your setup.

2.2.4. Step Three: Test

Now we will test that the installation works. Run `node` from a command line and execute the command `require('Fabric').createClient().build.getName();`. You should see something like this:

```
user@host~$ node
> require('Fabric').createClient().build.getName();
[FABRIC] Fabric Engine version 1.0.22-release
'Fabric Engine'
>
```

This confirms that the Fabric Engine Node.js module is installed and working. You can press `Ctrl-C` or run `process.exit();` to exit.

2.2.5. (Optional) Step Four: Install Extensions

To install the optional Fabric extensions, see Chapter 4, *Fabric Extensions*.

2.3. Getting Started with Fabric Engine on Node.js

This is a tutorial intended to help get you started with using the Fabric Engine Node.js module. It assumes that you've already installed the Fabric Engine Node.js module as specified above.

All of the examples in this tutorial are provided in the `Fabric Engine PublicDev github repository` [https://github.com/fabric-engine/PublicDev/] under `Examples/Node`. The filenames given in this tutorial refer to the filenames within a checkout of this repository.

2.3.1. Conventions

We will be showing some examples of direct command-line usage, e.g.:

```
host~ user$ node
> FC = require('Fabric').createClient();
[FABRIC] Fabric Engine version 1.0.22-release
{ build:
 { isExpired: [Function],
...
close: [Function],
getMemoryUsage: [Function] }
>
```

In general, we may omit the Fabric Engine startup messages (prefixed by `[FABRIC]`) from example output. In cases when it's obvious, we may also omit the startup of Node.js and the creation of the Fabric Engine client:

```
> FC.build.getFullVersion();
'1.0.22-release'
>
```

In other examples, we will be working with JavaScript sources files that will be run directly:

Example 2.1. Examples/Node/Tutorial/trivialHTTPServer.js

```
var fabric = require('Fabric');
var http = require('http');

http.createServer(function (req, res) {
(function (fabricClient) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end(fabricClient.build.getName() + " version " + fabricClient.build.getFullVersion()
 + "\n");
  fabricClient.close();
})(fabric.createClient());
}).listen(1337, "127.0.0.1");

console.log('Server running at http://127.0.0.1:1337/');
```

We will then generally show the output of running the command directly:

```
host:Tutorial user$ node trivialHTTPServer.js
[FABRIC] Fabric Engine version 1.0.22-release
Server running at http://127.0.0.1:1337/
```

In cases of using a server (as in the example above), we will also show examples of accessing the server, generally using `curl` or some other command line tool:

```
host~ user$ curl http://127.0.0.1:1337/
Fabric Engine version 1.0.22-release
host~ user$
```

2.3.2. A Standalone Node.js Application

Generally, Node.js is used to create server-type applications that accept incoming requests and produce responses. Fabric integrates well with this type of application, but we will start by creating a standalone Node.js application that instead simply computes and prints a result; think of it as a kind of "Hello, world!" for using Fabric Engine with Node.js.

Since Fabric Engine augments an existing platform (in this case Node.js) to perform heavy-duty computation, the simplest meaningful Fabric Engine examples are still fairly complex. The examples here should be fairly obvious if you read through the code and the comments, be we encourage you to refer to the “Fabric Engine KL Programming Guide” [http://documentation.fabricengine.com/latest/FabricEngine-KLProgrammingGuide.pdf], the “Fabric Engine Dependency Graph Programming Guide” [http://documentation.fabricengine.com/latest/FabricEngine-DependencyGraphProgrammingGuide.pdf] and the “Fabric Engine Map Reduce Programming Guide” [http://documentation.fabricengine.com/latest/FabricEngine-MapReduceProgrammingGuide.pdf] for more detail as to what's possible with the Fabric Engine core and the KL programming language.

Our first example will compute the sum of the first N terms of the series `1/1 + 1/4 + 1/9+....` by computing all of the terms in parallel and then summing the results. This provides a simple example of using Fabric Engine for a parallel computation.

First, the source code for the Node.js source file:

Example 2.2. Examples/Node/Tutorial/Series/Series-Display.js

```
var fabricClient = require('Fabric').createClient();
var fs = require('fs');

// Take the number of terms to compute on the command
// line; default to 10

var numTerms = parseInt(process.argv[2]) || 10;

// Create the operator that computes the value for
// each term of the series

var computeTermOp = fabricClient.DG.createOperator("computeTermOp");
computeTermOp.setSourceCode('computeTerm.kl', fs.readFileSync('computeTerm.kl', 'utf8'));
computeTermOp.setEntryPoint('computeTerm');

// Create the binding that binds the computeTermOp to the
// terms node.  A binding binds the members of the node
// to the arguments to the operator

var computeTermBinding = fabricClient.DG.createBinding();
computeTermBinding.setOperator(computeTermOp);
computeTermBinding.setParameterLayout([
  "self.index",   // self.index is special: the index of the
                  // slice being operated on
  "self.result"
]);

// Create the node that holds the terms in the series.
// The number of terms is the "count" of the node,
// ie. the SIMD multiplicity

var termsNode = fabricClient.DG.createNode("termsNode");
termsNode.setCount(numTerms);
termsNode.addMember("result", "Scalar");
termsNode.bindings.append(computeTermBinding);

// Create the operator that sums the terms of the series

var sumTermsOp = fabricClient.DG.createOperator("sumTermsOp");
sumTermsOp.setSourceCode('sumTerms.kl', fs.readFileSync('sumTerms.kl', 'utf8'));
sumTermsOp.setEntryPoint('sumTerms');

// Create the binding that binds sumTermsOp to the members of
// sumNode

var sumTermsBinding = fabricClient.DG.createBinding();
sumTermsBinding.setOperator(sumTermsOp);
sumTermsBinding.setParameterLayout([
  "terms",            // terms is special: it is a Container
                      // object that allows you to get and set
                      // the slice count of the node
  "terms.result<>",   // the <> syntax specifies that we want to bind
                      // to all the slices at once
  "self.result"
]);

// Create the node to hold the result, add termsNode as a
// dependency and append the binding for sumTermsOp

var sumNode = fabricClient.DG.createNode("sumNode");
sumNode.addMember("result", "Scalar");
sumNode.setDependency(termsNode, "terms");
sumNode.bindings.append(sumTermsBinding);

// Evaluate the sumNode (which evalutes its dependecy, the
// termsNode, first) and then print the result.

sumNode.evaluate();
console.log(sumNode.getData('result', 0));

// Close the Fabric Engine client.  If the client isn't closed
// then Node.js will keep this script alive!

fabricClient.close();
```

Before we can do anything with the Fabric Core, we need to call `require('Fabric')`. The result of the require statement is an object that allows you to create Fabric Core client connections by calling its `createClient()` method.

Important note: As of this writing, it is not safe to perform multiple computations on the same client connection at the same time. As such, it is highly recommended for request server architectures (which we will demonstrate below) that one client connection is created per server request!

Once we've created the client connection, we use it to build a dependency graph. We load the operator source code from external files (whose contents are given below), create the operators and bindings, set the number of terms, perform the evaluation and print the result. Once we are done we close the client connection to tell the Fabric Engine core that we are done using it.

Important note: If you do not close a client connection the Node.js script will never exit. This is the same as for other connections in Node.js such as request connections accepted by the HTTP server.

The source code for the two operators follow:

Example 2.3. `Examples/Node/Tutorial/Series/computeTerm.kl`

```
operator computeTerm(
  Size index,
  io Scalar result
  )
{
  result = 1.0 / (Scalar(index+1) * Scalar(index+1));
}
```

Example 2.4. `Examples/Node/Tutorial/Series/sumTerms.kl`

```
operator sumTerms(
  Container self,
  io Scalar terms<>,
  io Scalar result
  )
{
  result = 0.0;
  for (Size i=0; i<self.size; ++i)
    result += terms[i];
}
```

And the output from running the program to request the first 1000 terms of the series:

```
host:Series user$ node Series-Display.js 1000
[FABRIC] Fabric Engine version 1.0.22-release
1.643934845924377
host:Series user$
```

2.3.3. An HTTP Server Using Fabric Engine

We now modify our series computation example to show how it is easily transformed into an HTTP server for the same result.

The key modification that is made to the program is to perform the computation asynchronously. To do so, we replace the call to `sumNode.evalute()` with a call to `sumNode.evaluateAync(...)`, passing a callback function that gets executed when the computation is finished. While the computation is being performed by Fabric Engine, control returns to Node.js which will continue to service other incoming HTTP requests. Fabric Engine provides generic, high-performance, parallel computation as an asynchronous service to Node.js, similar to an asynchronous web service or database query.

The other modifications that are made are:

- We will use a standard HTTP server and pass the number of terms to compute as the GET variable `n`
- In order to server multiple requests at once, we will create one Fabric Engine client connection per incoming HTTP request

The actual "meat" of the computation is exactly as before. The source code is:

Example 2.5. `Examples/Node/Tutorial/Series/Series-HTTP.js`

```
var fabric = require('Fabric');
var fs = require('fs');
var http = require('http');
var url = require('url');

// Create the HTTP server
http.createServer(function (req, res) {

  // Take the number of terms to compute from the 'n' GET
  // parameter; default to 10
  var numTerms = parseInt(url.parse(req.url, true).query['n']) || 10;

  // Ensure that each request has its own client
  // connection and local scope using the usual JavaScript trick
  (function (fabricClient) {

    // Create the operator that computes the value for
    // each term of the series

    var computeTermOp = fabricClient.DG.createOperator("computeTermOp");
    computeTermOp.setSourceCode('computeTerm.kl', fs.readFileSync('computeTerm.kl', 'utf8'));
    computeTermOp.setEntryPoint('computeTerm');

    // Create the binding that binds the computeTermOp to the
    // terms node.  A binding binds the members of the node
    // to the arguments to the operator

    var computeTermBinding = fabricClient.DG.createBinding();
    computeTermBinding.setOperator(computeTermOp);
    computeTermBinding.setParameterLayout([
      "self.index",   // self.index is special: the index of the
```

```
                        // slice being operated on
      "self.result"
    ]);

    // Create the node that holds the terms in the series.
    // The number of terms is the "count" of the node,
    // ie. the SIMD multiplicity

    var termsNode = fabricClient.DG.createNode("termsNode");
    termsNode.setCount(numTerms);
    termsNode.addMember("result", "Scalar");
    termsNode.bindings.append(computeTermBinding);

    // Create the operator that sums the terms of the series

    var sumTermsOp = fabricClient.DG.createOperator("sumTermsOp");
    sumTermsOp.setSourceCode('sumTerms.kl', fs.readFileSync('sumTerms.kl', 'utf8'));
    sumTermsOp.setEntryPoint('sumTerms');

    // Create the binding that binds sumTermsOp to the members of
    // sumNode

    var sumTermsBinding = fabricClient.DG.createBinding();
    sumTermsBinding.setOperator(sumTermsOp);
    sumTermsBinding.setParameterLayout([
      "terms",            // terms is special: it is a Container object
                          // that allows you to get and set the slice
                          // count of the node
      "terms.result<>",   // the <> syntax specifies that we want to bind
                          // to all the slices at once
      "self.result"
    ]);

    // Create the node to hold the result, add termsNode as a
    // dependency and append the binding for sumTermsOp

    var sumNode = fabricClient.DG.createNode("sumNode");
    sumNode.addMember("result", "Scalar");
    sumNode.setDependency(termsNode, "terms");
    sumNode.bindings.append(sumTermsBinding);

    // Evaluate the sumNode asynchronously.  The given
    // callback is issue when the computation is done.

    sumNode.evaluateAsync(function () {

      // Return the result as the HTTP content

      res.writeHead(200, {'Content-Type': 'text/plain'});
      res.end(sumNode.getData('result', 0) + "\n");

      // Close the Fabric Engine client.  If the client isn't closed
      // then Node.js will keep this script alive!

      fabricClient.close();
    });

  })(fabric.createClient());

}).listen(1337, "127.0.0.1");

console.log('Server running at http://127.0.0.1:1337/');
```

The code for the operators is exactly as before. When run, the HTTP server starts:

```
host:Series user$ node Series-HTTP.js
```

```
[FABRIC] Fabric Engine version 1.0.22-release
Server running at http://127.0.0.1:1337/
```

From another command line, we can request computation, passing the number of terms as the `n` query parameter:

```
host:~ user$ curl http://127.0.0.1:1337/?n=1000
1.643934845924377
host:~ user$
```

2.3.4. Supported and Unsupported Features of Fabric Engine Node.js Module

Fabric Engine was originally developed as a browser plug-in that could be used to render high-end 3D effects inside of normal web pages. Many of the features that are available in the context of high-end browser 3D are not available or available differently when Fabric Engine is being used as a Node.js module.

Features that are not supported or that are partially supported:

- OpenGL viewports are not supported. It is expected that some sort of off-line rendering support will be made available in a future version of the Node.js module.
- ResourceLoadNodes are available but can only load URLs that use the `file:` method for local file access. A future version of the Node.js module will add support for access to remote URLs using the `http:` method.

2.3.5. Where to Go from Here

You have learned the basics of how to use Fabric Engine with Node.js and how to use it for asynchronous computation as a Node.js service. As mentioned above, much more information can be obtained by reading the "Fabric Engine KL Programming Guide" [http://documentation.fabricengine.com/latest/FabricEngine-KLProgrammingGuide.pdf], the "Fabric Engine Dependency Graph Programming Guide" [http://documentation.fabricengine.com/latest/FabricEngine-DependencyGraphProgrammingGuide.pdf] and the "Fabric Engine Map Reduce Programming Guide" [http://documentation.fabricengine.com/latest/FabricEngine-MapReduceProgrammingGuide.pdf] . You can find a few more examples of the use of Fabric Engine with Node.js in our `benchmark repository` [https://github.com/fabric-engine/Benchmarks]. And finally, you can learn about using Fabric Engine for in-browser high-end 3D effects by learning about the Fabric JavaScript scene graph.

Chapter 3. Python Module

3.1. Supported Platforms

The Fabric Engine module for Python currently runs on:

- Windows Vista or Windows 7
- Mac OS X running on Intel processors, version 10.6 ("Snow Leopard") or later
- 32-bit and 64-bit Linux. We do our development on Ubuntu 10.04 and above but it should work with any modern distribution

We target Python version 2.6.x or later, but may work with older versions of Python as well.

3.2. Installation

3.2.1. Step One: Download

From the `Fabric distributions page` [http://dist.fabricengine.com/latest/], download the version of the Node.js module that applies for your operating system.

For Linux, choose the file based on whether you are running a 32-bit or 64-bit operating system. If you're not sure which version to download, run `uname -m` which will report either `i686` or `x86_64`.

- `FabricEngine-PythonModule-Linux-i686-VERSION.tar.bz2`
- `FabricEngine-PythonModule-Linux-x86_64-VERSION.tar.bz2`

For Mac OS X choose:

- `FabricEngine-PythonModule-Darwin-universal-VERSION.tar.bz2`

For Windows choose:

- `FabricEngine-PythonModule-Windows-x86-VERSION.zip`

3.2.2. Step Two: Install

On Mac OS X and Linux, unpack the module by running `tar jxf FabricEngine-PythonModule-XXX.tar.bz2` from a command line. On Windows, simply double-click the .zip file to unpack it.

The module can be installed anywhere you would normally install Python modules on your system. If you do not already have a location where you normally install Python modules you can create one. Assuming that your local user is called `myuser`, the recommended paths to create are as follows:

- Windows: `c:\Users\myuser\python_modules`
- Mac OS X: `/Users/myuser/python_modules`
- Linux: `/home/myuser/python_modules`

Next, inside the module that you just unpacked there should be a subfolder called `python_modules` with a subfolder of its own called `fabric`. Move this `fabric` folder to the folder you created above. In Mac OS X or Linux you can also create a symlink if it makes more sense for your setup.

If you created a new folder above you will also need to set an environment variable to tell Python where to look. On Mac OS X or Linux this can be done by running `export PYTHONPATH=~/python_modules`. You may want to add this to your `~/.bashrc` or `~/.bash_profile` as well.

Under Windows you'll want to set it as a permanent environment variable by going to Control Panel -> System -> Advanced System Settings -> Advanced -> Environment Variables. Create a new variable called `PYTHONPATH` and set it to `c:\Users\myuser\python_modules`.

3.2.3. Step Three: Test

Now we will test that the installation works. Run `python` from a command line and try this:

```
user@host~$ python
Python 2.7.2+ (default, Jan 21 2012, 23:31:34)
[GCC 4.6.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import fabric
[FABRIC] Fabric Engine version 1.0.22-release
>>> fabric.createClient().build.getName()
[FABRIC] Searching extension directory '/home/andrew/.fabric/Exts'
[FABRIC] Warning: unable to open extension directory '/home/andrew/.fabric/Exts'
[FABRIC] Searching extension directory '/usr/lib/fabric/Exts'
[FABRIC] Warning: unable to open extension directory '/usr/lib/fabric/Exts'
u'Fabric Engine'
>>>
```

This confirms that the Fabric Engine Python module is installed and working. You can press `Ctrl-D` and then `Ctrl-C` to exit.

3.2.4. (Optional) Step Four: Install Extensions

To install the optional Fabric extensions, see Chapter 4, *Fabric Extensions*.

3.3. Getting Started with Fabric Engine on Python

This is a tutorial intended to help get you started with using the Fabric Engine Python module. It assumes that you've already installed the Fabric Engine Python module as described above.

All of the examples in this tutorial are provided in the `Fabric Engine PublicDev github repository` [https://github.com/fabric-engine/PublicDev/] under `Examples/Python`. The filenames given in this tutorial refer to the filenames within a checkout of this repository.

3.3.1. Conventions

We will be showing some examples of direct command-line usage, e.g.:

```
user@host~$ python
Python 2.7.2+ (default, Jan 21 2012, 23:31:34)
[GCC 4.6.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import fabric
[FABRIC] Fabric Engine version 1.0.22-release
>>> client = fabric.createClient()
[FABRIC] Searching extension directory '/home/andrew/.fabric/Exts'
[FABRIC] Warning: unable to open extension directory '/home/andrew/.fabric/Exts'
[FABRIC] Searching extension directory '/usr/lib/fabric/Exts'
[FABRIC] Warning: unable to open extension directory '/usr/lib/fabric/Exts'
>>>
```

In general, we may omit the Fabric Engine startup messages (prefixed by `[FABRIC]`) from example output. In cases when it's obvious, we may also omit the Python startup and the creation of the Fabric Engine client:

```
>>> client.build.getFullVersion()
u'1.0.22-release'
>>>
```

In other examples, we will be working with Python sources files that will be run directly:

Example 3.1. `Examples/Python/Tutorial/trivialHTTPServer.py`

```
import fabric
import BaseHTTPServer

class FabricHandler( BaseHTTPServer.BaseHTTPRequestHandler ):
  def do_GET( self ):
    client = fabric.createClient()
    self.send_response( 200 )
    self.send_header( 'Content-type', 'text/plain' )
    self.end_headers()
    self.wfile.write( client.build.getName() + ' version ' + client.build.getFullVersion() + '\n'
 )
    client.close()

port = 1337
httpd = BaseHTTPServer.HTTPServer( ('', port), FabricHandler )
print('Server running at http://127.0.0.1:' + str(port) + '/')
httpd.serve_forever()
```

We will then generally show the output of running the command directly:

```
user@host~$ python trivialHTTPServer.py
[FABRIC] Fabric Engine version 1.0.22-release
Server running at http://127.0.0.1:1337/
```

In cases of using a server (as in the example above), we will also show examples of accessing the server, generally using `curl` or some other command line tool:

```
andrew@variable:~$ curl http://127.0.0.1:1337/
Fabric Engine version 1.0.22-release
andrew@variable:~$
```

3.3.2. A Standalone Python Application

We will start by creating a standalone Python application that simply computes and prints a result; think of it as a kind of "Hello, world!" for using Fabric Engine with Python.

Since Fabric Engine augments an existing platform (in this case Python) to perform heavy-duty computation, the simplest meaningful Fabric Engine examples are still fairly complex. The examples here should be fairly obvious if you read through the code and the comments, and we encourage you to refer to the “Fabric Engine KL Programming Guide” [http://documentation.fabricengine.com/latest/FabricEngine-KLProgrammingGuide.pdf], the “Fabric Engine Dependency Graph Programming Guide” [http://documentation.fabricengine.com/latest/FabricEngine-DependencyGraphProgrammingGuide.pdf] and the “Fabric Engine Map Reduce Programming Guide” [http://documentation.fabricengine.com/latest/FabricEngine-MapReduceProgrammingGuide.pdf] for more detail as to what's possible with the Fabric Engine core and the KL programming language.

Our first example will compute the sum of the first N terms of the series `1/1 + 1/4 + 1/9+....` by computing all of the terms in parallel and them summing the results. This provides a simple example of using Fabric Engine for a parallel computation.

First, the source code for the Python source file:

Example 3.2. `Examples/Python/Tutorial/Series/Series-Display.py`

```python
import fabric
import sys

fabricClient = fabric.createClient()

# Take the number of terms to compute on the command
# line default to 10

numTerms = 10
if len( sys.argv ) > 1:
  numTerms = int( sys.argv[1] )

# Create the operator that computes the value for
# each term of the series

computeTermOp = fabricClient.DG.createOperator("computeTermOp")
computeTermOp.setSourceCode('computeTerm.kl', open('computeTerm.kl').read())
computeTermOp.setEntryPoint('computeTerm')

# Create the binding that binds the computeTermOp to the
# terms node.  A binding binds the members of the node
# to the arguments to the operator

computeTermBinding = fabricClient.DG.createBinding()
computeTermBinding.setOperator(computeTermOp)
computeTermBinding.setParameterLayout([
  "self.index",   # self.index is special: the index of the
                  # slice being operated on
  "self.result"
])

# Create the node that holds the terms in the series.
# The number of terms is the "count" of the node,
# ie. the SIMD multiplicity
```

```
termsNode = fabricClient.DG.createNode("termsNode")
termsNode.setCount(numTerms)
termsNode.addMember("result", "Scalar")
termsNode.bindings.append(computeTermBinding)

# Create the operator that sums the terms of the series

sumTermsOp = fabricClient.DG.createOperator("sumTermsOp")
sumTermsOp.setSourceCode('sumTerms.kl', open('sumTerms.kl').read())
sumTermsOp.setEntryPoint('sumTerms')

# Create the binding that binds sumTermsOp to the members of
# sumNode

sumTermsBinding = fabricClient.DG.createBinding()
sumTermsBinding.setOperator(sumTermsOp)
sumTermsBinding.setParameterLayout([
  "terms",              # terms is special: it is an object that
                        # allows you to get and set the number of
                        # slices of the node
  "terms.result<>",     # the <> syntax specifies that we want to bind
                        # to all the slices at once
  "self.result"
])

# Create the node to hold the result, add termsNode as a
# dependency and append the binding for sumTermsOp

sumNode = fabricClient.DG.createNode("sumNode")
sumNode.addMember("result", "Scalar")
sumNode.setDependency(termsNode, "terms")
sumNode.bindings.append(sumTermsBinding)

# Evaluate the sumNode (which evalutes its dependecy, the
# termsNode, first) and then print the result.

errors = sumNode.getErrors()
for error in errors:
  print error
sumNode.evaluate()
print(sumNode.getData('result', 0))

# Close the Fabric Engine client.  If the client isn't closed
# then the Python module will keep this script alive!

fabricClient.close()
```

Before we can do anything with the Fabric Core, we need to call `import fabric`. The result of the import statement is an object that allows you to create Fabric Core client connections by calling its `createClient()` method.

Important Note: As of this writing, it is not safe to perform multiple computations on the same client connection at the same time. As such, it is highly recommended for request server architectures (which we will demonstrate below) that one client connection is created per server request!

Once we've created the client connection, we use it to build a dependency graph. We load the operator source code from external files (whose contents are given below), create the operators and bindings, set the number of terms, perform the evaluation and print the result. Once we are done we close the client connection to tell the Fabric Engine core that we are done using it.

Important Note: If you do not close a client connection the Python script will never exit. For testing you can use Ctrl-C to close a Python script where you forgot to close the Fabric client but you should always make sure that the client is correctly closed in production scripts.

The source code for the two operators follow:

Example 3.3. `Examples/Python/Tutorial/Series/computeTerm.kl`

```
operator computeTerm(
  Size index,
  io Scalar result
  )
{
  result = 1.0 / (Scalar(index+1) * Scalar(index+1));
}
```

Example 3.4. `Examples/Python/Tutorial/Series/sumTerms.kl`

```
operator sumTerms(
  Size numTerms,
  io Scalar terms<>,
  io Scalar result
  )
{
  result = 0.0;
  for (Size i=0; i<numTerms; ++i)
    result += terms[i];
}
```

And the output from running the program to request the first 1000 terms of the series:

```
user@host~$ python Series-Display.py 1000
[FABRIC] Fabric Engine version 1.0.22-release
1.64393484592
user@host~$
```

3.3.3. An HTTP Server Using Fabric Engine

We now modify our series computation example to show how it is easily transformed into an HTTP server for the same result.

The key modification that is made to the program is to perform the computation asynchronously. To do so, we replace the call to `sumNode.evalute()` with a call to `sumNode.evaluateAync(...)`, passing a callback function that gets executed when the computation is finished. While the computation is being performed by Fabric Engine, control returns to Python which will continue to service other incoming HTTP requests. Fabric Engine provides generic, high-performance, parallel computation as an asynchronous service, similar to an asynchronous web service or database query.

The other modifications that are made are:

- We will use the Python BaseHTTPServer and pass the number of terms to compute as the GET variable `n`
- In order to server multiple requests at once, we will create one Fabric Engine client connection per incoming HTTP request

The actual "meat" of the computation is exactly as before. The source code is:

Example 3.5. `Examples/Python/Tutorials/Series/Series-HTTP.py`

```
import fabric
import BaseHTTPServer
import urlparse

class FabricHandler( BaseHTTPServer.BaseHTTPRequestHandler ):
  def do_GET( self ):

    query = urlparse.parse_qs(urlparse.urlparse(self.path).query)
    numTerms = 10
    if 'n' in query:
      numTerms = int( query[ 'n' ][ 0 ] )

    fabricClient = fabric.createClient()

    # Create the operator that computes the value for
    # each term of the series

    computeTermOp = fabricClient.DG.createOperator("computeTermOp")
    computeTermOp.setSourceCode('computeTerm.kl', open('computeTerm.kl').read())
    computeTermOp.setEntryPoint('computeTerm')

    # Create the binding that binds the computeTermOp to the
    # terms node.  A binding binds the members of the node
    # to the arguments to the operator

    computeTermBinding = fabricClient.DG.createBinding()
    computeTermBinding.setOperator(computeTermOp)
    computeTermBinding.setParameterLayout([
      "self.index",    # self.index is special: the index of the
                       # slice being operated on
      "self.result"
    ])

    # Create the node that holds the terms in the series.
    # The number of terms is the "count" of the node,
    # ie. the SIMD multiplicity

    termsNode = fabricClient.DG.createNode("termsNode")
    termsNode.setCount(numTerms)
    termsNode.addMember("result", "Scalar")
    termsNode.bindings.append(computeTermBinding)

    # Create the operator that sums the terms of the series

    sumTermsOp = fabricClient.DG.createOperator("sumTermsOp")
    sumTermsOp.setSourceCode('sumTerms.kl', open('sumTerms.kl').read())
    sumTermsOp.setEntryPoint('sumTerms')

    # Create the binding that binds sumTermsOp to the members of
    # sumNode

    sumTermsBinding = fabricClient.DG.createBinding()
    sumTermsBinding.setOperator(sumTermsOp)
    sumTermsBinding.setParameterLayout([
      "terms",             # terms is special: it is an object that
                           # allows you to get and set the number of slices
      "terms.result<>",    # the <> syntax specifies that we want to bind
                           # to all the slices at once
      "self.result"
    ])

    # Create the node to hold the result, add termsNode as a
    # dependency and append the binding for sumTermsOp
    sumNode = fabricClient.DG.createNode("sumNode")
    sumNode.addMember("result", "Scalar")
    sumNode.setDependency(termsNode, "terms")
    sumNode.bindings.append(sumTermsBinding)

    # Evaluate the sumNode

    sumNode.evaluate()

    # Return the result as the HTTP content

    self.send_response( 200 )
    self.send_header( 'Content-type', 'text/plain' )
    self.end_headers()
    self.wfile.write( str(sumNode.getData('result', 0)) + '\n' )

    # Close the Fabric Engine client.  If the client isn't closed
    # then Fabric Python client will keep this script alive!

    fabricClient.close()

# Create the HTTP server
port = 1337
httpd = BaseHTTPServer.HTTPServer( ('', port), FabricHandler )
print('Server running at http://127.0.0.1:' + str(port) + '/')
httpd.serve_forever()
```

The code for the operators is exactly as before. When run, the HTTP server starts:

```
user@host~$ python Series-HTTP.py
[FABRIC] Fabric Engine version 1.0.22-release
Server running at http://127.0.0.1:1337/
```

From another command line, we can request computation, passing the number of terms as the `n` query parameter:

```
user@host~$ curl http://127.0.0.1:1337/?n=1000
1.64393484592
user@host~$
```

3.3.4. Supported and Unsupported Features of Fabric Engine Python Module

Fabric Engine was originally developed as a browser plug-in that could be used to render high-end 3D effects inside of normal web pages. Many of the features that are available in the context of high-end browser 3D are not yet available or available differently when Fabric Engine is being used as a Python module.

Features that are not supported or that are partially supported:

- OpenGL viewports are not yet supported, but as of this writing, a PyQT port is being developed. For up-to-date information please consult the `Fabric Engine` [http://www.fabricengine.com] website.

3.3.5. Where to Go from Here

You have learned the basics of how to use Fabric Engine with Python and how to use it for asynchronous computation. As mentioned above, much more information can be obtained by reading the “Fabric Engine

KL Programming Guide” [http://documentation.fabricengine.com/latest/FabricEngine-KLProgrammingGuide.pdf], the “Fabric Engine Dependency Graph Programming Guide” [http://documentation.fabricengine.com/latest/FabricEngine-DependencyGraphProgrammingGuide.pdf] and the “Fabric Engine Map Reduce Programming Guide” [http://documentation.fabricengine.com/latest/FabricEngine-MapReduceProgrammingGuide.pdf] . You can find a few more examples of the use of Fabric Engine with Node.js in our `benchmark repository` [https://github.com/fabric-engine/Benchmarks]. And finally, you can learn about using Fabric Engine for in-browser high-end 3D effects by learning about the Fabric JavaScript scene graph.

Chapter 4. Fabric Extensions

Each module should also contain the Fabric Extensions. These are optional but expose additional functionality within Fabric and KL. To install the Extensions, look for a subfolder called `Exts` under the archive for your module that you downloaded and unpacked. The contents of this folder should be copied to a specific location depending on your operating system:

- Windows: `c:\Users\myuser\AppData\Roaming\Fabric\Exts`
- Mac OS X: `/Users/myuser/Library/Fabric/Exts`
- Linux: `/home/myuser/.fabric/Exts`

This folder may not exist and will need to be created (including any missing parent folders) if not.

If you are upgrading the extensions from a previous version of Fabric, you should remove the contents of this folder before copying over the new extensions. This ensure that if any extensions are removed from the Fabric distribution (which may be because they are no longer compatible with the latest version of Fabric) then it won't cause problems.

After copying the files into the folder specified you can verify that they are seen by Fabric by running the command as specified in the Test section for your module. After the `createClient()` call, you should now see output that looks something like this:

```
>>> fabric.createClient()
[FABRIC] Searching extension directory '/home/user/.fabric/Exts'
[FABRIC] [FabricTEEM] Extension registered
[FABRIC] [FabricOGL] Extension registered
[FABRIC] [FabricMath] Extension registered
[FABRIC] [FabricFILESTREAM] Extension registered
[FABRIC] [FabricLIDAR] Extension registered
[FABRIC] [FabricOBJ] Extension registered
[FABRIC] [TimeSample] Extension registered
[FABRIC] [FabricHDR] Extension registered
[FABRIC] [FabricPNG] Extension registered
[FABRIC] [FabricVIDEO] Extension registered
[FABRIC] [FabricBULLET] Extension registered
[FABRIC] [ExceptSample] Extension registered
[FABRIC] [FabricFILESYSTEM] Extension registered
[FABRIC] [FabricALEMBIC] Extension registered
[FABRIC] [FabricCIMG] Extension registered
[FABRIC] [FabricTGA] Extension registered
[FABRIC] [FabricEXR] Extension registered
[FABRIC] Searching extension directory '/usr/lib/fabric/Exts'
[FABRIC] Warning: unable to open extension directory '/usr/lib/fabric/Exts'
>>>
```

If your output is similar to that then the extensions have been successfully installed.