

NETLab Hub Plug-in Developer Guide

Introduction

Plug-ins provide the NETLab Hub with functionality. Hub plug-ins can communicate with hardware, perform operations on the host machine's filesystem, or even grab content from web sites; in fact, a plug-in can be written to perform almost any operation that you can think of as long as you can write a Java program to do it. Through plug-ins, clients connecting to the Hub can have access to a wide array of tools.

The Plug-in API allows you to write your own plug-ins. It is always best to check the repository [todo] for existing plug-ins that will accomplish what you want to do before embarking on the process of writing your own. This document describes the process of writing your own plug-in. It starts with a Quick Start that gets you up and running quickly, then goes into detail about various aspects of the API such as testing, helper classes and frameworks, and exception handling, as well as contributing your plug-in to the repository so that others can use it.

Quick Start

This section will help you get up and running with developing your own plug-in. The basic steps to writing a plug-in are:

1. Define your main plug-in class.
2. Tell the Hub about your plug-in by writing a short XML file.
3. Deploy your plug-in to the Hub's plug-ins directory.

You can download the demo plug-in discussed below from here. [todo] For reference, you can also consult the core plug-ins source code provided with the Plug-in Developer Kit.

Prerequisites

Download and install the NETLab Hub application

You can download the latest version of the Hub at ???. In theory, you only need the core.jar library that comes with the Hub, but downloading the full version will be important for testing your plug-in.

Set up your development environment

The NETLab Hub requires Java 1.6. It is also much easier if you install an Integrated Development Environment (IDE) such as Eclipse or NetBeans. The instructions in this document assume that you have installed Eclipse.

Next, add the core.jar library included in the Plug-in Developer Kit to your development project's build path. If you downloaded the Mac version of the Hub then you can find this file by control-clicking the NETLabHub.app file, selecting "Show Package Contents", and navigating to Contents/Resources/Java.

Writing Your Plug-in

Define your plug-in's service classes

Plug-ins provide functionality by implementing one or more “service” classes. The bulk of that functionality is provided by implementing the `netlab.hub.core.IService` interface's `accept()` method, or by overriding that same method in a subclass of `netlab.hub.core.BaseService`. A simple plug-in implementation that echoes a string back to the client might look like this:

```
package net.myplugins;

import netlab.hub.core.BaseService;
import netlab.hub.core.ServiceRequest;
import netlab.hub.core.ServiceResponse;
import netlab.hub.core.ServiceException;

public class HelloWorldPlugin extends BaseService {

    public boolean accept(ServiceRequest request, ServiceResponse response)
        throws ServiceException {

        String clientName = request.getArgument(0);
        if (clientName == null) {
            response.write("Hello, anonymous!", super.outputFormat);
        } else {
            response.write("Hello, "+clientName, super.outputFormat);
        }
        return true;
    }
}
```

Tell the Hub about your plug-in

You now need to write a simple configuration file, named `config.xml`, that gives the Hub the information it needs to load your plug-in and to dispatch messages to it. The configuration file should reside in the plug-in's root directory. A configuration file for the above example looks like this:

```
<!DOCTYPE PlugIn>
<PlugIn version="1.0" build="Sep 30 2010">
  <Services>
    <!--
    =====
    Hello, world
    This service echoes a message back to the client.
    It takes an optional name argument.
    Message format: /service/myplug/hello/say [name]
    =====
    -->
    <Service enabled="true" name="hello" type="net.myplugins.HelloWorldPlugin">
      <Description>A generic Hello World plugin</Description>
    </Service>
  </Services>
</PlugIn>
```

Deploy your plug-in

You make your plug-in available to the Hub by copying the compiled class file and the XML configuration file it into a folder named after your plug-in, which you should place in the Hub's `plugins` directory (in the same directory as the `NETLabHub` application file). In the above example, the folder hierarchy would look like this:

```
[Hub root]/plugins/myplug/net/myplugins/HelloWorldPlugin.class
[Hub root]/plugins/myplug/config.xml
```

Alternatively, you can deploy your plug-in classes in a jar file, like so:

```
[Hub root]/plugins/myplug/hello.jar
[Hub root]/plugins/myplug/config.xml
```

Restart the Hub and your plug-in should now accept messages from clients. In the above example, all messages sent to the Hub that match the pattern `/service/myplug/hello/say` will be dispatched to your plugin's `accept()` method.

Getting into Details

The Plug-in API core

Addressing

Hub clients gain access to the specific services provided by a plug-in through an OSC-like addressing scheme. All messages are sent to the Hub as strings, which are then dispatched to the appropriate service, decoded and encapsulated in a `netlab.hub.core.ServiceRequest` instance.

An address string from the example above might be:

```
/service/myplugin/hello/say Frank
```

The Hub parses this message into segments according to the following scheme:

Service address (required)			Service command (optional)	
Common prefix	Plug-in name	Service name	Command	Arguments
service	myplug	hello	say	Frank

The first three segments address a specific service defined within a plug-in. This service is defined using the `Service` element in the `config.xml` document. The fourth segment and arguments string is passed to the service's `accept()` method.

Segment 1: Common prefix: this segment is a common prefix for all services. It must be included but is not currently used.

Segment 2: The second element is a plug-in's name. This name must be unique across all installed plug-ins. The plug-in's containing directory name should match this.

Segment 3: The service name, as defined by the `config.xml` document's `Service` element `name` attribute.

Segment 4 (optional): The command sent to the `accept()` method of the service class whose type is defined in the `config.xml` document's `Service` element `type` attribute. This command can be retrieved using `request.getPath().first()` or `request.getPath().get(0)`, both of which return a `java.lang.String`.

Arguments (optional): Arguments can be retrieved using `request.getArguments()` (which returns a `java.util.LinkedList` implementation) or with `request.getArgument(idx)` (which returns a `java.lang.String`). Normally, arguments are delimited with a space (as in Unix command line arguments) but the Hub also parses arguments that are enclosed in double-quotes or in curly braces. The following are thus all legal address message strings:

```
/service/myplug/hello/say
/service/myplug/hello/say Frank
/service/myplug/hello/say "Frank"
/service/myplug/hello/say "Frank" {Mary}
/service/myplug/hello/say "Frank and Mary" Janet
/service/myplug/hello/say "Frank and Mary" {Janet}
```

To parse these arguments, the `accept()` method in the example above might be changed to something like:

```
public boolean accept(ServiceRequest request, ServiceResponse response)
    throws ServiceException {
    if (request.getArguments().isEmpty()) {
        response.write("Hello, anonymous!", super.outputFormat);
    } else {
        for (Iterator<String> it=request.getArguments().iterator(); it.hasNext();) {
            response.write("Hello, "+it.next(), super.outputFormat);
        }
    }
    return true;
}
```

Plug-in design

Architecture

Plug-ins can include as many separate services and service classes as required. It is up to the plug-in designer to modularize the plug-in's functionality among separate service commands and service classes. Most simple plug-ins will contain one service class.

A single plug-in might be designed to perform a very specific set of functions, such as communicating with a specific hardware controller. Conversely, a "toolkit" plug-in might offer a general, disconnected set of utilities and tools.

Directory layout

The full directory structure for plug-ins is:

Note that the `lib` directory contents will be automatically added to the Hub application classpath when the Hub is first launched.

Finally, you should create your plug-in's directory structure. (Doing this with an Ant build file is easiest in the long run since it will simplify future deployments.) Assuming that your plug-in is called "myplugin", the directory structure is:

to-do

Frameworks

The Plug-in API includes some basic frameworks and utilities that make building plug-ins easier.