asphalt-py4j

Release 3.0.1

Sep 27, 2017

Contents

This Asphalt framework component provides the ability to run Java code directly from Python.

It is a wrapper for the Py4J library.

CHAPTER 1

Configuration

There are two principal ways to use a Java Virtual Machine with Py4J:

1. Launch a new JVM just for the application, in a subprocess
2. Connect to an existing JVM

The first method is what most people will want. The Java Virtual Machine is started along with the application and is shut down when the application is shut down.

The second method is primarly useful in special scenarios like connecting to a Java EE container. Shutting down the application has no effect in the Java side gateway then.

The minimal configuration is as follows:

```
components:
  py4j:
```

This will publish a resource of type `py4j.java_gateway.JavaGateway`, named `default`. It will appear in the context as the `java` attribute.

Connecting to an existing JVM

To connect to an existing Java Virtual Machine, specify the host address and port of the JVM that has a GatewayServer running, you can use a configuration similar to this:

```
components:
  py4j:
    launch_jvm: false
    host: 10.0.0.1
    port: 25334
```

This configuration will connect to a JVM listening on `10.0.0.1`, port 25334.

By default the JavaGateway connects to 127.0.0.1 port 25333, so you can leave out either value if you want to use the default.

Multiple gateways

If you need to configure multiple gateways, you can do so by using the `gateways` configuration option:

```
components:
  py4j:
    gateways:
      default:
        context_attr: java
      remote:
        launch_jvm: false
        host: 10.0.0.1
```

This configures two `py4j.gateway.JavaGateway` resources, named `default` and `remote`. Their corresponding context attributes are `java` and `remote`. If you omit the `context_attr` option for a gateway, its resource name will be used.

Adding jars to the class path

When you distribute your application, you often want to include some jar files with your application. But when configuring the gateway to launch a new JVM, you need to include those jar files on the class path. The problem is of course that you don't necessarily know the absolute file system path to your jar files beforehand. The solution is to define a *package relative* class path in your Py4J configuration. This feature is provided by the Py4J component and not the upstream library itself.

Suppose your project has a package named `foo.bar.baz` and a subdirectory named `javalib`. The relative path from your project root to this subdirectory would then be `foo/bar/baz/javalib`. To properly express this in your class path configuration, you can do this:

```
components:
  py4j:
    classpath: "{foo.bar.baz}/javalib/*"
```

This will add all the jars in the javalib subdirectory to the class path. The `{foo.bar.baz}` part is substituted with the computed absolute path to the `foo.bar.baz` package directory.

Note: Remember to enclose the path in quotes when specifying the class path in a YAML configuration file. Otherwise the parser may mistake it for the beginning of a dictionary definition.

```
components:
  py4j:
    classpath:
      - "{foo.bar.baz}/javalib/*"
      - "{x.y}/jars/*"
```

This specifies a class path of multiple elements in an operating system independent manner using a list. The final class path is computed by joining the elements using the operation system's path separator character.

CHAPTER 2

Using the Java gateway

Given the inherently synchronous nature of the Py4J API, it is strongly recommended that all code using Java gateways is run in a thread pool. The `JavaGateway` class is not wrapped in any way by the component so slow calls to any Java API will block the event loop if not run in a worker thread.

The following is a simple example that writes the text "Hello, Python!" to a file at `/tmp/test.txt`. More examples can be found in the `examples` directory of the source distribution.

```
async def handler(ctx):
    async with ctx.threadpool():
        f = ctx.java.jvm.java.io.File('/tmp/test.txt')
        writer = ctx.java.jvm.java.io.FileWriter(f)
        writer.write('Hello, Python!')
        writer.close()
```

CHAPTER 3

Version history

This library adheres to Semantic Versioning.

3.0.1 (2017-06-04)

- Added compatibility with Asphalt 4.0

3.0.0 (2017-04-11)

- **BACKWARD INCOMPATIBLE** Migrated to Asphalt 3.0 and py4j 0.10.4+

2.0.0 (2016-05-11)

- **BACKWARD INCOMPATIBLE** Migrated to Asphalt 2.0
- Allowed combining `gateways` with default parameters

1.1.0 (2016-01-02)

- Added typeguard checks to fail early if arguments of wrong types are passed to functions

1.0.1 (2015-11-20)

- Fixed the Asphalt dependency specification to work with setuptools older than 8.0

1.0.0 (2015-05-16)

- Initial release
- API reference