Pyramid Avro Documentation

Release 0.0.2

Alex Milstead

Sep 01, 2017

Contents

This project is a Pyramid plugin built for integrating Avro protocol files into a pyramid application.

Be sure to check out the *Quickstart Project* documentation to get started.

PyPi https://pypi.python.org/pypi/pyramid-avro/

Coverage

Build Status

License Apache License (2.0)

CHAPTER 1

Python Version Support

Current Supported Versions:

- 2.7
- 3.3
- 3.4
- 3.5

CHAPTER 2

A Note On Compilation

While this plugin provides for auto-compiling your avro protocol into an avro schema, this is rarely something you'd want in all environments your application might be deployed into.

When defining a service configuration, you **must have at least a schema defined**. This means that the protocol file itself isn't actually ever required **UNLESS** the auto_compile flag is turned on.

For non-development configs, we suggest compiling your schema files prior to deployments and simply specifying them in your config rather turning auto_compile on.

Lastly, the tools jar must be provided by you, the developer, not this plugin. In addition to not wanting a compilation at runtime in non-dev environments, you probably don't want that jar hanging around either.

Contents:

Quickstart

Below are the minimal steps needed to get up and running with an avro-based pyramid app.

Setup Pyramid

Setup a virtualenv:

```
virtualenv avro-project
source avro-project/bin/activate
```

Install pyramid:

```
pip install pyramid pyramid-avro
```

Create starter scaffold:

```
pcreate avro-project
```

Then make sure your app is setup:

```
cd avro-project && python setup.py develop
```

Avro

Download the avro-tools jar here: avro-tools, and put it in a *lib* directory.

Make a *protocols* directory and add a simple "Hello, World!" avro protocol, called *hello.avdl*:

```
protocol HelloProtocol {
    error Exception {
        string message;
    }

    string hello_world(string arg) throws Exception;
}
```

Your file tree should now look like this:

```
avro-project
- avro_project
|   - __init__.py
|   - protocols
|   |   - hello.avdl
|   - static
|   |   - pyramid-16x16.png
|   |   - pyramid.png
|   |   - theme.css
|   - templates
|   |   - mytemplate.pt
|   - tests.py
|   - views.py
- CHANGES.txt
- development.ini
- lib
|   - avro-tools.jar
- MANIFEST.in
- production.ini
- pytest.ini
- README.txt
- setup.py
```

Configure Routes

In your development.ini file, add these options:

```
pyramid_includes =
    pyramid_avro

# Set the base URL path prefix, other wise it's /<service-name>
avro.default_path_prefix = /avro
```

```
# Set up base protocol dir.
avro.protocol_dir = %(here)s/avro_project/protocols

# Set auto-compile to true.
avro.auto_compile = true

# Where tools jar lives:
avro.tools_jar = $(here)s/lib/avro-tools.jar

# Begin service definitions:
avro.service.hello =
    protocol = hello.avdl
```

Now we need to add an implementation for the *hello* message. In the *views.py* file, add this:

```
from pyramid_avro.decorators import avro_message


@avro_message(service_name="hello", message="hello_world")
def hello_handler(request):
    return "Hello, {}!".format(request.avro_data["arg"])
```

Now run the server in one terminal:

```
pserve development.ini
```

Congratulations! You now have an avro service running on the */avro/hello* endpoint of your pyramid application!

Client Integration

A simple *test_client.py* would look like the following:

```
import os

from avro import ipc
from avro import protocol


here = os.path.abspath(os.path.dirname(__file__))
protocol_file = os.path.join(here, "avro_project", "protocols", "hello.avpr")


if __name__ == "__main__":
    with open(protocol_file) as _file:
        protocol_object = protocol.parse(_file.read())
    driver = ipc.HTTPTransceiver("localhost", 6543, "/avro/hello")
    client = ipc.Requestor(protocol_object, driver)

    response = client.request("hello_world", {"arg": "World"})
    print response
```

And upon execution, you'd see:

```
$ python test_client.py
Hello, World!
```

Configuring pyramid-avro

Core configuration options:

- default_path_prefix: A default URL path prefix.
- protocol_dir: A path to a base directory for protocol files.
- auto_compile: Whether or not to automatically compile protocol -> schema on config commit.
- tools_jar: A path to an avro-tools (look for *avro-tools-X.Y.Z.jar*).
- service objects
  - schema: A path to a schema file.
  - protocol: A path to a protocol file.
  - pattern: A URL pattern.

Configuration Files

Protocols can be configured inside pyramid config files as follows:

```
# Make sure it's included:
pyramid_includes =
    pyramid_avro

# Run compilation of protocol -> schema on start-up.
avro.auto_compile = false

# All avro routes will be prepended by "/avro"
avro.default_path_prefix = /avro

# A base protocol directory.
avro.protocol_dir = %(here)s/protocols

# A tools jar reference.
avro.tools_jar = %(here)s/avro_project/lib/avro-tools-1.7.7.jar

# Service definitions.
avro.service.foo =
    pattern = /avro/{val}/foo
    protocol = foo.avdl

avro.service.bar =
    schema = bar.avpr
    pattern = /avro/other-bar

avro.service.baz =
    schema = baz.avpr
```

Config Object/Programmatic

Protocols and message implementations can also be configured directly by calling pyramid-avro's config directives:

```
from pyramid.config import Configurator


def main(global_config, **settings):

    config = Configurator(settings=settings)
    config.include("pyramid_avro")

    config.add_avro_route("foo", pattern="/avro/{val}/foo",
        protocol="foo.avdl")
    config.add_avro_route("bar", pattern="/avro/other-bar",
        schema="bar.avpr")
    config.add_avro_route("baz", schema="baz.avpr")

    config.register_avro_message("foo", "my_project.views:impl", message="my_message")
    config.scan()
    return config.make_wsgi_app()
```

Writing Implementations

Message implementations can be done as free-standing functions or two different forms of object instance methods.

Pure Configuration

You can register a message implementation through pure config:

```
from pyramid.config import Configurator


def main(global_config, **settings):
    config = Configurator(settings=settings)

    config.include("pyramid_avro")

    # The message name can be derived from the function
    config.register_avro_route("hello", "avro_project.views:hello_world")

    # Or set explicitly
    config.register_avro_route("hello", "avro_project.views:other_message",
        message="other_message")
```

Using Decorators

Free-standing functions

Define a simple implementation function is as easy as follows:

```
# The message name can be derived from the function
@avro_message(service_name="hello")
def hello_world(request):
    return "Hello, {}!".format(request.avro_data["arg"])
```

```
# Or set explicitly
@avro_message(service_name="hello", message="other_message")
def other_message_impl(request):
    return "Hello, other {}!".format(request.avro_data["arg"])
```

It's possible to use an object instance for implementing your service methods, too.

There are two ways service routes can be defined.

Classes: Service Name Specified

The service name can be setup under the class as a property:

```
class HelloProtocol(object):

    service_name = "hello"

    # The message name can be derived from the function
    @avro_message()
    def hello_world(self, request):
        return "Hello, {}!".format(request.avro_data["arg"])

    # Or set explicitly
    @avro_message(message_name="other_message")
    def other_message_impl(self, request):
        return "Hello, other {}!".format(request.avro_data["arg"])
```

Classes: Service Name Derived

Or the name can be derived from the class name (it's *.lower()*'d):

```
class Hello(object):

    # The message name can be derived from the function
    @avro_message()
    def hello_world(self, request):
        return "Hello, {}!".format(request.avro_data["arg"])

    # Or set explicitly
    @avro_message(message_name="other_message")
    def other_message_impl(self, request):
        return "Hello, other {}!".format(request.avro_data["arg"])
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search