asphalt-serialization

Release 4.0.2.post3

Oct 01, 2017

Contents

1 Configuration **3**
- 1.1 Multiple serializers . . . 3

2 Using serializers **5**
- 2.1 Registering custom types with serializers . . . 5
- 2.2 Disabling the default wrapping of marshalled custom types . . . 7
- 2.3 Serializing built-in custom types . . . 7

3 Writing new serializer backends **9**

4 Version history **11**

This Asphalt framework component provides a standardized interface for a number of different serialization algorithms:

- CBOR (using cbor2)
- JSON (using the Python standard library `json` module)
- msgpack (using msgpack-python)
- Pickle (using the Python standard library `pickle` module)
- YAML (using PyYAML)

Additional backends may be provided through third party plugins.

Some serializers also provide hooks for safely (un)marshalling custom types and this mechanism can easily be plugged into a third party marshalling library.

CHAPTER 1

Configuration

To configure a serializer for your application, you need to choose a backend and then specify any necessary configuration values for it. The following backends are provided out of the box:

- `cbor` (**recommended**)
- `json`
- `msgpack`
- `pickle`
- `yaml`

Other backends may be provided by other components.

Once you've selected a backend, see its specific documentation to find out what configuration values you need to provide, if any. Configuration values are expressed as constructor arguments for the backend class:

```
components:
  serialization:
    backend: json
```

This configuration publishes a `asphalt.serialization.api.Serializer` resource named `default` using the JSON backend, accessible as `ctx.json`. The same can be done directly in Python code as follows:

```
class ApplicationComponent(ContainerComponent):
    async def start(ctx: Context):
        self.add_component('serialization', backend='json')
        await super().start()
```

1.1 Multiple serializers

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

```
components:
  serialization:
    serializers:
      cbor:
      json:
      msgpack:
      pickle:
      foobar:
        backend: json
        context_attr: foo
        encoding: iso-8859-15
```

The above configuration creates 5 resources of type `asphalt.serialization.api.Serializer`:

- `cbor` as `ctx.cbor`
- `json` as `ctx.json`
- `msgpack` as `ctx.msgpack`
- `pickle` as `ctx.pickle`
- `foobar` as `ctx.foo`

CHAPTER 2

Using serializers

Using serializers is quite straightforward:

```
async def handler(ctx):
    serialized = ctx.json.serialize({'foo': 'example JSON object'})
    original = ctx.json.deserialize(payload)
```

This example assumes a configuration where a JSON serializer is present as `ctx.json`.

To see what Python types can be serialized by every serializer, consult the documentation of the abstract `Serializer` class.

2.1 Registering custom types with serializers

An application may sometimes need to send over the wire instances of classes that are not normally handled by the chosen serializer. In order to do that, a process called *marshalling* is used to reduce the object to something the serializer can natively handle. Conversely, the process of restoring the original object from a natively serializable object is called *unmarshalling*.

The `pickle` serializer obtains the serializable state of an object from the `__dict__` attribute, or alternatively, calls its `__getstate__()` method. Conversely, when deserializing it creates a new object using `__new__()` and either sets its `__dict__` or calls its `__setstate__` method. While this is convenient, pickle has an important drawback that limits its usefulness. Pickle's deserializer automatically imports arbitrary modules and can trivially be made to execute any arbitrary code by maliciously constructing the datastream.

A better solution is to use one of the `cbor`, `msgpack` or `json` serializers and register each type intended for serialization using `register_custom_type()`. This method lets the user register marshalling/unmarshalling functions that are called whenever the serializer encounters an instance of the registered type, or when the deserializer needs to reconstitute an object of that type using the state object previously returned by the marshaller callback.

The default marshalling callback mimics pickle's behavior by returning the `__dict__` of an object or the return value of its `__getstate__()` method, if available. Likewise, the default unmarshalling callback either updates the `__dict__` attribute of the uninitialized instance, or calls its `__setstate__()` method, if available, with the state object.

The vast majority of classes are directly compatible with the default marshaller and unmarshaller so registering them is quite straightforward:

```
from asphalt.serialization.serializers.json import JSONSerializer


class User:
    def __init__(self, name, email, password):
        self.name = name
        self.email = email
        self.password = password

serializer = JSONSerializer()
serializer.register_custom_type(User)
```

If the class defines `__slots__` or requires custom marshalling/unmarshalling logic, the easiest way is to implement `__getstate__` and/or `__setstate__` in the class:

```
class User:
    def __init__(self, name, email, password):
        self.name = name
        self.email = email
        self.password = password

    def __getstate__(self):
        # Omit the "password" attribute
        dict_copy = self.__dict__.copy()
        del dict_copy['password']
        return dict_copy

    def __setstate__(self, state):
        state['password'] = None
        self.__dict__.update(state)

serializer = JSONSerializer()
serializer.register_custom_type(User)
```

If you are unable to modify the class itself, you can instead use standalone functions for that:

```
def marshal_user(user):
    # Omit the "password" attribute
    dict_copy = user.__dict__.copy()
    del dict_copy['password']
    return dict_copy


def unmarshal_user(user, state):
    state['password'] = None
    user.__dict__.update(state)

serializer.register_custom_type(User, marshal_user, unmarshal_user)
```

The callbacks can be a natural part of the class too if you want:

```
class User:
    def __init__(self, name, email, password):
        self.name = name
        self.email = email
```

```
        self.password = password

    def marshal(self):
        # Omit the "password" attribute
        dict_copy = self.__dict__.copy()
        del dict_copy['password']
        return dict_copy

    def unmarshal(self, state):
        state['password'] = None
        self.__dict__.update(state)

serializer.register_custom_type(User, User.marshal, User.unmarshal)
```

Hint: If a component depends on the ability to register custom types, it can request a resource of type `CustomizableSerializer` instead of `Serializer`.

2.2 Disabling the default wrapping of marshalled custom types

When you register a custom type with a serializer, it by default wraps its marshalled instances during serialization in a way specific to each serializer in order to include the type name necessary for automatic deserialization. For example, the `json` serializer wraps the state of a marshalled object in a JSON object like `{"__type__": "MyTypeName", "state": {"some_attribute": "some_value"}}`.

In situations where you need to serialize objects for a recipient that does not understand this special wrapping, you can forego the wrapping step by passing the `wrap_state=False` option to the serializer. Doing so will cause the naked state object to be directly serialized. Of course, this will disable the automatic deserialization, since the required metadata is no longer available.

2.3 Serializing built-in custom types

If you need to (de)serialize types that have mandatory arguments for their `__new__()` method, you will need to supply a specialized unmarshaller callback that returns a newly created instance of the target class. Likewise, if the class has neither a `__dict__` or a `__getstate__()` method, a specialized marshaller callback is required.

For example, to successfully marshal instances of `datetime.timedelta`, you could use the following (un)marshalling callbacks:

```
from datetime import timedelta


def marshal_timedelta(td):
    return td.total_seconds()


def unmarshal_timedelta(seconds):
    return timedelta(seconds=seconds)

serializer.register_custom_type(timedelta, marshal_timedelta, unmarshal_timedelta)
```

As usual, so long as the marshaller and unmarshaller callbacks agree on the format of the state object, it can be anything natively serializable.

CHAPTER 3

Writing new serializer backends

If you wish to implement an alternate method of serialization, you can do so by subclassing the `Serializer` class. There are three methods implementors must override:

- `serialize()`
- `deserialize()`
- `mimetype()`

The `mimetype` method is a `@property` that simply returns the MIME type appropriate for the serialization scheme. This property is used by certain other components. If you cannot find an applicable MIME type, you can use `application/octet-stream`.

Note: Serializers must always serialize to bytes; never serialize to strings!

If you want your serializer to be available as a backend for `SerializationComponent`, you need to add the corresponding entry point for it. Suppose your serializer class is named `AwesomeSerializer`, lives in the package `foo.bar.awesome` and you want to give it the alias `awesome`, add this line to your project's `setup.py` under the `entry_points` argument in the `asphalt.serialization.serializers` namespace:

```
setup(
    # (...other arguments...)
    entry_points={
        'asphalt.serialization.serializers': [
            'awesome = foo.bar.awesome:AwesomeSerializer'
        ]
    }
)
```

CHAPTER 4

Version history

This library adheres to Semantic Versioning.

4.0.2 (2017-06-04)

- Added compatibility with Asphalt 4.0

4.0.1 (2017-05-11)

- Fixed `None` not being accepted in place of a serializer configuration dictionary

4.0.0 (2017-04-24)

- **BACKWARD INCOMPATIBLE** Migrated to Asphalt 3.0
- **BACKWARD INCOMPATIBLE** Upgraded cbor2 dependency to v4
- **BACKWARD INCOMPATIBLE** Improved the ability to customize the serialization of custom types in serializers implementing the `CustomizableSerializer` interface by specifying a value for the `custom_type_codec` option. This replaces the `custom_type_key` and `wrap_state` options.

3.2.0 (2016-11-24)

- Added the ability to skip wrapping custom marshalled objects (by setting `wrap_state=False` in any of the customizable serializers)

3.1.0 (2016-09-25)

- Allow parameterless unmarshaller callbacks that return a new instance of the target class
- Switched YAML serializer to use ruamel.yaml instead of PyYAML

3.0.0 (2016-07-03)

- **BACKWARD INCOMPATIBLE** Switched the CBOR implementation to *cbor2 <https://pypi.io/project/cbor2/>*
- **BACKWARD INCOMPATIBLE** Switched msgpack's MIME type to `application/msgpack`
- **BACKWARD INCOMPATIBLE** Switched the default context attribute name to the backend name, for consistency with asphalt-templating

- Added custom type handling for CBOR, msgpack and JSON serializers
- Serializer resources are now also published using their actual types (in addition the interfaces)

2.0.0 (2016-05-09)

- **BACKWARD INCOMPATIBLE** Migrated to Asphalt 2.0
- **BACKWARD INCOMPATIBLE** A backend must be specified explicitly (it no longer defaults to JSON)
- Allowed combining `serializers` with default parameters

1.1.0 (2016-01-02)

- Added support for CBOR (Concise Binary Object Representation)
- Added typeguard checks to fail early if arguments of wrong types are passed to functions

1.0.0 (2015-05-31)

- Initial release
- API reference