EYNNYD

Release 0.1.0

Chad Befus

Oct 24, 2020

CONTENTS

Eynnyd (pronounced [Ey-nahyd]) is an acronym for **Everything You Need, Nothing You Don't**. It is a light-weight WSGI compliant python 3 web framework. Eynnyd was designed with the primary goal to not impose bad engineering decisions on it's users. It is also designed to not overstep or assume the wants of it's user.

Other frameworks weigh you down with extraneous dependencies, unnecessary inheritance, highly coupled design, forced restrictions into REST, forced function naming, magic global singletons, and so much more.

```
from eynnyd import RoutesBuilder
from eynnyd import EynnydWebappBuilder
from eynnyd import ResponseBuilder

 def hello_world(request):
     return ResponseBuilder() \
         .set_utf8_body("Hello World") \
         .build()

 routes = \
     RoutesBuilder() \
         .add_handler("GET", "/hello", hello_world) \
         .build()

 application = \
     EynnydWebappBuilder() \
         .set_routes(routes) \
         .build()
```

CHAPTER

ONE

WHY USE EYNNYD?

Eynnyd was created to provide all the power of a WSGI web framework without imposing extraneous decision restricting your design. You can use Eynnyd with minimal interaction with the framework.

The priorities when building Eynnyd are as follows:

1. **Readable:** By using intention revealing names, small functions with single levels of abstraction, pushing conditionals up the stack, avoiding exceptions for control flow, and all the other core principles of good software engineering (found in Clean Code), the code in Eynnyd is designed to be easily read and understood by other developers (or even just ourselves in the future). Given that it should be easy to read our code, it should be easy to extend, fix bugs, test, and improve over time; thus why this is our top priority.

2. **Flexible:** Our framework was fueled from a frustration of restrictive engineering decisions being imposed upon users from other frameworks. Our goal was freedom for you to design your own implementations, your way. With our framework you can:

- Use Object-oriented, procedural, or functional design.
- Use REST, REST-like, or completely ad-hoc API design.
- Name your functions anything you like, allowing you to be expressive with your naming.
- Use dependency injection, or don't.
- Have as many interceptors (both request and response) as you like and restrict the routes they wrap.
- Pick whatever libraries are best for your dependencies (Not some selection we made for you).
- And pretty much anything else you want to do.

3. **Extensible:** Designed following the **SOLID** principles, adding new features should be easily done without having to dig through miles of coupled code. This means that we can address feature requests quickly and maintain confidence in our backwards compatibility.

4. **Reliable:** To add confidence to our code we have an extensive suite of unittests covering our framework. While we don't actually believe in the validity of code coverage measurements (bad tests are worse than no tests) we purposely pushed our coverage to 100% (Largely this was for marketing purposes). But the important paths of execution in our code are heavily covered in layers of well designed tests. In doing so we wrote highly decoupled tests which all follow the F.I.R.S.T. principles of good unittests. With these tests we can address bugs and add features with high confidence that we are not introducing new bugs.

5. **Predictable:** With Eynnyd you get pretty-much-what-you-expect. We don't bury exceptions, or assume implementation. We wont parse your body into JSON for you or any other magic. We also work hard to provide you with a strong-borders-free-society environment; by doing validation at the edge we can raise errors as close to their implementation as possible, instead of deep in the belly of unrelated logic.

6. **Fast:** Let's be honest, when dealing with HTTP overhead speed is a luxury. Certainly the various frameworks range in their speed, and handle very specific situation better or worse than each other, but the best any WSGI framework can do is match the speed of raw WSGI. With the other priorities on this list maximized we will continue to re-evaluate our

design and speed to make progress here, but never at the cost of higher priority items. Our goal is to not be wasteful with our speed. We will follow up with actual comparisons between our framework and others soon. However, if you absolutely need something faster than Eynnyd, you probably should look into non HTTP type frameworks or at least non WSGI frameworks.

CHAPTER

TWO

DOCUMENTATION

2.1 User's Guide

2.1.1 Install

Eynnyd is published to pypi and can be installed using pip via:

```
pip install eynnyd
```

2.1.2 Deploy

Eynnyd is a WSGI framework, which means to serve an Eynnyd application you need to run a WSGI server. We recommend Gunicorn (as it is what we use) but there are many worth looking into.

Local Serving

If you used Gunicorn then you can run your application via:

```
gunicorn hello_world_app
```

This assumes that you have a file named *hellow_world_app.py* where inside you have a variable named *application* which returns the built Eynnyd Webapp (See *the hello world tutorial* for an example).

Deploying a Server

Due to Eynnyd being a new framework we don't have a ton of documentation on deploying to various cloud systems.

The good news is that because it is a WSGI framework, Eynnyd can be used as a drop in replacement for any tutorial on how to deploy any other WSGI frameworks.

Keep watching though for coming documentation on deploying Eynnyd specifically.

2.1.3 Glossary

Throughout these docs and our code we use a series of terms which we hope make sense inherently but just in case we have compiled this glossary:

Handler A Handler is the code executed for a request which converts a request into a response. In Eynnyd it is simply a function which takes a single argument (the request) and returns a response.

Interceptor An Interceptor is code which is executed before or after a *Handler* for a request. Request Interceptors happen before the *Handler* and Response Interceptors happen after. In Eynnyd you can have as many Interceptors as you like executed around a *Handler*. In some frameworks this is also called Middleware.

Route A route is a path of execution to some code. *Handler* Routes use an HTTP method like "GET", "POST", "PUT", "DELETE", etc., and a path like "/foo/bar" to decide what handlers to execute. *Interceptor* Routes only require a path and execute for every request along that path.

Error Handler An Error Handler is code that gets executed when an associated exception is thrown. There are two types of pre response error handlers and post response error handler. Pre response error handlers which execute when an exception is thrown from a request *Interceptor* or a *Handler*. Post response error handlers execute when an exception is thrown during response *Interceptor*s.

2.1.4 Tutorials

We will build up our collection of Tutorials over time to include as many real world situations as possible. Our tutorials use Eynnyd the way we prefer **but this is not the only way to use the framework.**

For example, we like to using Fluent Interfaces and our framework allows for that, but it also allows for you to not work this way.

A Fluent usage might look like:

```
routes = \
    RoutesBuilder() \
        .add_request_interceptor("/hello", log_request) \
        .add_handler("GET", "/hello", hello_world) \
        .add_response_interceptor("/hello", log_response) \
        .build()
```

but this would work just as well if you wrote it as:

```
routes_builder = RoutesBuilder()
routes_builder.add_request_interceptor("/hello", log_request)
routes_builder.add_handler("GET", "/hello", hello_world)
routes_builder.add_response_interceptor("/hello", log_response)
routes = routes_builder.build()
```

Use the framework the way you prefer, who are we to judge?

Tutorial: Hello World

This is our most basic of tutorials. How can we create an endpoint which when hit with a blank **GET** request it returns the text "Hello World". First we will show you the code and then we will explain the parts.

```
# hello_world_app.py
from eynnyd import RoutesBuilder
from eynnyd import EynnydWebappBuilder
from eynnyd import ResponseBuilder
from http import HTTPStatus

def hello_world(request):
    return ResponseBuilder()\
        .set_status(HTTPStatus.OK)\
        .set_utf8_body("Hello World")\
        .build()
```

(continues on next page)

(continued from previous page)

```

def build_application():
    routes = \
        RoutesBuilder()\
            .add_handler("GET", "/hello", hello_world)\
            .build()

    return EynnydWebappBuilder()\
            .set_routes(routes)\
            .build()

application = build_application()
```

Simple right? Lets look at the various parts.

Simple Request Handler

Our request *Handler* is called **hello_world**. It looks like:

```
def hello_world(request):
    return ResponseBuilder()\
        .set_status(HTTPStatus.OK)\
        .set_utf8_body("Hello World")\
        .build()
```

It's simply a function which takes a *request* and returns a *response*. Many other frameworks provide you with both a request and response as inputs to your *Handler*s. This is exploiting output parameters and is generally a violation of Clean Code. We prefer to use returns for outputs and reserve parameters for inputs. For the purposes of this *Handler*, we don't care anything about the request, all we want to do is return a response with the content "Hello World".

We are using Eynnyds built in *ResponseBuilder*. to construct a response. It is possible to build responses yourself, but the `ResponseBuilder` is a convenient tool for doing it succinctly.

For our response we are returning a status of `OK`, which evaluates to a code of `200`. We could have left this `set_status(HTTPStatus.OK)` line out though, because Eynnyd uses `OK` as the default response status. We added it here to give you a clear example of how to set the status.

For our body we are using the `set_utf8_body("Hello World")` method. You can set several different kinds of bodies on your response using the `ResponseBuilder` depending on what content you want to send (for example: Steaming, Byte, Iterable, etc.).

Finally we call the `build()` method on the `ResponseBuilder`. This method tells the `ResponseBuilder` to create the response. You might note that we use the Builder Pattern a lot in Eynnyd. For things that have default values, require validation on inputs, and can be built up over time, we believe the Builder pattern to be a valid way of separating the concerns of building something, from using something. This ties into the Clean Code Philosophy that objects should do one thing.

Building Routes

Organizing and wiring up your *Route*s to the code they execute is a single responsibility. This is why we don't like then *Route*s are defined at the definition site (in some frameworks) or as part of the building of the webapp itself (in other frameworks). In our code above, building the *Route*s looks like:

```
routes = \
    RoutesBuilder()\
        .add_handler("GET", "/hello", hello_world)\
        .build()
```

The key here is that we have added a *Handler* for any request using the HTTP method `GET` on path `/hello` will execute the *Handler* code inside our `hello_world` method.

After this we call the `build()` method and our `routes` variable now is assigned to a built routing system.

Building the Webapp

Next we have to build the actual Web Application itself. We do this with code that looks like:

```
return EynnydWebappBuilder()\
        .set_routes(routes)\
        .build()
```

Here we use the `set_routes` method to pass our built *Route*s from above to the webapp so that it can direct requests to the right place.

After this we call the `build()` method and return a fully ready to use Web Application.

Setting the Application Variable

The last line of our code assigns the global variable named `application` to the result of our `build_application()` method (which is a built `EynnydWebapp`). This is a WSGI standard allowing the server to connect into your application.

Tutorial: Request Interceptors

This tutorial builds on what we saw in our *hello world tutorial* so you probably want to read that if you haven't yet. All we are going to do is add a `request interceptor` which logs every request coming into the application. First we will show you the code and then we will explain the *relevant* parts (AKA the parts not in the hello world tutorial).

```
# hello_world_app.py
import logging

from eynnyd import RoutesBuilder
from eynnyd import EynnydWebappBuilder
from eynnyd import ResponseBuilder
from http import HTTPStatus

LOG = logging.getLogger("hello_world_app")

def hello_world(request):
    return ResponseBuilder()\
        .set_status(HTTPStatus.OK)\
        .set_utf8_body("Hello World")\
        .build()

def log_request(request):
    LOG.info("Got Request: {r}".format(r=request))
```

(continues on next page)

(continued from previous page)

```
    return request

def build_application():
    routes = \
        RoutesBuilder()\
            .add_request_interceptor("/hello", log_request)\
            .add_handler("GET", "/hello", hello_world)\
            .build()

    return EynnydWebappBuilder()\
            .set_routes(routes)\
            .build()

application = build_application()
```

Only minor changes to the hello world tutorial are added here.

The Request Interceptor Code

Our request *Interceptor* is called `log_request` and it looks like:

```
def log_request(request):
    LOG.info("Got Request: {r}".format(r=request))
    return request
```

As you can see our *Interceptor* is a function which takes a request and returns a request. In this case we are returning the same request we were given. All we are doing here is logging the string representation of the request to an info level log line. However, if we had mutated the request here (or better yet, created a new request with slightly updated values), **the request we return is the request which will be passed on from this point** through the code**. For example, perhaps you would like to attach a new header to your request, you could do that here and all following code would be given that new request object.

Routing Requests Through The Request Interceptor

The other relevant piece of code here is the registering of the *Interceptor* to specific *Route*s. It looks like:

```
routes = \
    RoutesBuilder()\
        .add_request_interceptor("/hello", log_request)\
        .add_handler("GET", "/hello", hello_world)\
        .build()
```

As you can see, we are adding a request *Interceptor* which should run for any request on the path `/hello`. This includes *Route*s like `/hello/more/path/parts`.

The request *Interceptor*s will run before a matching *Handler* is run. You can register many request *Interceptor*s, even at the same path level. This allows you to have small, single purpose *Interceptor*s, that are easy to test and maintain. Other frameworks only allow you to have a single *Interceptor* for all requests which leads to messy implementations.

Request *Interceptor*s run in priority of outside in (so *Interceptor*s at the base path will run before *Interceptor*s at a more specific path) and then first in first out (the order added to the RoutesBuilder).

Tutorial: Response Interceptors

This tutorial builds on what we saw in our *request interceptor tutorial* so you probably want to read that if you haven't yet. All we are going to do is add a response interceptor which logs every response leaving the application. First we will show you the code and then we will explain the *relevant* parts (AKA the parts not in the prior tutorials).

```python
# hello_world_app.py
import logging

from eynnyd import RoutesBuilder
from eynnyd import EynnydWebappBuilder
from eynnyd import ResponseBuilder
from http import HTTPStatus

LOG = logging.getLogger("hello_world_app")

def hello_world(request):
    return ResponseBuilder() \
        .set_status(HTTPStatus.OK) \
        .set_utf8_body("Hello World")\
        .build()

def log_request(request):
    LOG.info("Got Request: {r}".format(r=request))
    return request

def log_response(request, response):
    LOG.info("Built Response: {s} for Request: {r}".format(s=response, r=request))
    return response

def build_application():
    routes = \
        RoutesBuilder() \
            .add_request_interceptor("/hello", log_request) \
            .add_handler("GET", "/hello", hello_world) \
            .add_response_interceptor("/hello", log_response)\
            .build()

    return EynnydWebappBuilder() \
            .set_routes(routes) \
            .build()

application = build_application()
```

Only minor changes are made from the request *Interceptor* tutorial, in particular the response *Interceptor* method and the routing connection of the response *Interceptor*.

The Response Interceptor Code

Our response *Interceptor* is called `log_response` and it looks like:

```python
def log_response(request, response):
    LOG.info("Built Response: {s} for Request: {r}".format(s=response, r=request))
    return response
```

Response *Interceptor*s are passed both the request (modified by any request *Interceptor*s it has passed through) and the response (either from the *Handler* which created it or any prior response *Interceptor*s who may have changed it).

It returns a response and this case we are simply returning the response we were passed.

The response returned is the response that will be passed to any follow up response *Interceptor*s or, if this is the final one, sent to the client. You can use this to either modify the response you are given (Ideally through building a clone) or return a completely different response.

Routing Responses Through The Response Interceptor

The other relevant change to prior tutorials is the adding of the response *Interceptor*s *Route* to `RoutesBuilder`.

```
routes = \
    RoutesBuilder() \
        .add_request_interceptor("/hello", log_request) \
        .add_handler("GET", "/hello", hello_world) \
        .add_response_interceptor("/hello", log_response) \
        .build()
```

Here we have set it up so that any response from a *Route* down the `/hello` path would be logged. This includes *Route*s like `/hello/more/path/parts`.

The response *Interceptor*s run after a *Handler* has created a response from the request. You can have as many response *Interceptor*s as you please, even at the same level. This allows you to have small, single purpose, *Interceptor*s that are easy to test and maintain.

Response *Interceptor*s run in priority of inside out (more specific first to less specific) and first in first out (the order they are registered with the builder).

Tutorial: Error Handlers

*Error Handler*s are code that should execute if an exception is raised somewhere in the web application. Eynnyd provides you with a way to associate an Exception type with a function to execute if that exception is raised.

An error handling function takes two forms: pre_response and post_response. More on this below.

This tutorial builds on the prior one on *response interceptors*, so if there is a piece you don't understand here it was likely covered there.

First we will show you the code and then we will explain the *relevant* parts (AKA the parts not in the prior tutorials).

```
# hello_world_app.py
import logging

from eynnyd import RoutesBuilder
from eynnyd import EynnydWebappBuilder
from eynnyd import ResponseBuilder
from eynnyd import ErrorHandlersBuilder
from http import HTTPStatus

LOG = logging.getLogger("hello_world_app")

def hello_world(request):
    return ResponseBuilder() \
        .set_status(HTTPStatus.OK) \
        .set_utf8_body("Hello World")\
        .build()

def log_request(request):
```

(continues on next page)

(continued from previous page)

```
    LOG.info("Got Request: {r}".format(r=request))
    return request

def log_response(request, response):
    LOG.info("Built Response: {s} for Request: {r}".format(s=response, r=request))
    return response

class UnAuthorizedAccessAttemptException(Exception):
    pass

def raise_unauth_for_missing_header(request):
    if "AUTH" not in request.headers:
        raise UnAuthorizedAccessAttemptException("Must have an auth header to be␣
↪granted access.")
    return request

def handle_unauthorized_error(error_thrown, request):
    LOG.warn("Unauthorized attempt on url: {u}".format(u=request.request_uri))
    return ResponseBuilder() \
        .set_status(HTTPStatus.UNAUTHORIZED) \
        .set_utf8_body("Authorization failed with error: {e}".format(e=str(error_
↪thrown))) \
        .build()

def build_application():
    routes = \
        RoutesBuilder() \
            .add_request_interceptor("/", raise_unauth_for_missing_header) \
            .add_request_interceptor("/hello", log_request) \
            .add_handler("GET", "/hello", hello_world) \
            .add_response_interceptor("/hello", log_response)\
            .build()

    error_handlers = \
        ErrorHandlersBuilder() \
            .add_pre_response_error_handler(UnAuthorizedAccessAttemptException,␣
↪handle_unauthorized_error) \
            .build()

    return EynnydWebappBuilder() \
            .set_routes(routes) \
            .set_error_handlers(error_handlers) \
            .build()

application = build_application()
```

It should be noted that, so far in our tutorials, we have been using stand alone functions for all of our code. There functions can of course be encapsulated into objects so that a function like `def log_request(request)` could instead be `def log_request(self, request)` on a class and we would then use it at our callsites as `logging_interceptors.log_request`.

Building a Python Named Exception

The first relevant new piece to this tutorial is a custom named exception. Named Exceptions are superior to built in exceptions for a variety of reasons, mainly readability and allowing for explicit handling. That being said, you can use the built in python exceptions for this same purpose.

We haven't done anything special with this named exception so it should look like your typical usage of python:

```
class UnAuthorizedAccessAttemptException(Exception):
    pass
```

Here we have defined an exception to be used when access is attempted which should be denied as unauthorized.

Raising an Exception

Now that we have an exception we need somewhere to raise it. For this tutorial we are going to do that in a new request *Interceptor*.

```
def raise_unauth_for_missing_header(request):
    if "AUTH" not in request.headers:
        raise UnAuthorizedAccessAttemptException("Must have an auth header to be␣
↪granted access.")
    return request
```

Our new request *Interceptor* checks if there is a header keyed on "auth". If not it raises our named exception. Of course we probably want to do more validation on this header to confirm that even if it is present it is valid, but we can leave that to other *Interceptor*s (and out of this tutorial for simplicity).

The other thing we need to do, as expected is to register this request *Interceptor* into our *Route*s:

```
routes = \
    RoutesBuilder() \
        .add_request_interceptor("/", raise_unauth_for_missing_header) \
        .add_request_interceptor("/hello", log_request) \
        .add_handler("GET", "/hello", hello_world) \
        .add_response_interceptor("/hello", log_response)\
        .build()
```

As you can see, this *Interceptor* should run for all requests by using the root path "/".

Writing a Error Handling Method

Next we need code that we want to run if this error is thrown. That looks like:

```
def handle_unauthorized_error(error_thrown, request):
    LOG.warn("Unauthorized attempt on url: {u}".format(u=request.request_uri))
    return ResponseBuilder() \
        .set_status(HTTPStatus.UNAUTHORIZED) \
        .set_utf8_body("Authorization failed with error: {e}".format(e=str(error_
↪thrown))) \
        .build()
```

This function is built to handle errors thrown prior to having a response object (which is why it only takes parameters for the `error_thrown` and the `request`. If we threw our error from a *Handler* this code would look exactly the same. However, if we threw an error from a response *Interceptor* then this code would be different (the function would take a third parameter for the response).

*Error Handler*s return responses. In this case the response we are going to return is an `UNAUTHORIZED` status with a body of text describing the errors message.

Associating an Error Type with An Error Handler

Next we need to associate our named exception with the code we just wrote to handle that exception being thrown. We do this using the Eynnyd `ErrorHandlersBuilder` class:

```
error_handlers = \
    ErrorHandlersBuilder() \
        .add_pre_response_error_handler(UnAuthorizedAccessAttemptException, handle_
↪unauthorized_error) \
        .build()
```

You can see we are associating our new handler `handle_unauthorized_error` to the named exception `UnAuthorizedAccessAttemptException` by calling `add_pre_response_error_handler`. It should be obvious that this method only works for errors raised from a pre response location (request *Interceptor*s and handlers). Once there is a response (in response *Interceptor*s) you would want to associate your exception with the code to call using a similar method called: `add_post_response_error_handler`.

Adding Error Handlers To The Web App

Finally we can add our *Error Handler*s to the Eynnyd webapp using the `EynnydWebappBuilder`:

```
return EynnydWebappBuilder() \
        .set_routes(routes) \
        .set_error_handlers(error_handlers) \
        .build()
```

Very similar to setting our routing object from earlier tutorials.

Tutorial: Decorators

In the previous tutorial on *Tutorial: Error Handlers* we saw the use of request *Interceptor*s to do simple authorization. The truth is throwing exceptions like this may be pythonic but it it's a violation of *using exceptions for control flow*. Having requests which are not authorized isn't really exceptional behaviour.

In this Tutorial we will build out an authorization validator using both request *Interceptor*s and python decorators. Some frameworks implement their own versions of decorators (often calling them hooks or muddying them with their *Interceptor*s) but this is needless because python decorators are pretty awesome.

Building out a proper authorization cycle also requires talking to a database. For this tutorial we won't show the database code as it isn't relevant to the point.

A final change to our prior tutorials is that we will be taking more of an OO approach to our code this time. We do this for two reasons, first to show you how everything we've done thus far is just as easy in an OO mindset, and second, because it allows for dependency injection.

With all of that said, first we will show you the full code for this tutorial and then we will work through the parts piece by piece to explain them further.

```
# auth_example_app.py
import functools
import os
import sys

from eynnyd import RoutesBuilder
from eynnyd import EynnydWebappBuilder
```

(continues on next page)

(continued from previous page)

```
from eynnyd import ResponseBuilder
from eynnyd import ErrorHandlersBuilder
from http import HTTPStatus

from src.config_loader import ConfigLoader
from src.mysql_db_connection_pool import MySQLDBConnectionPool
from src.mysql_messages_dao import MySQLMessagesDAO
from src.mysql_sessions_dao import MySQLSessionsDAO

class MessagesHandler:

    def __init__(self, messages_dao):
        self._messages_dao = messages_dao

    @request_secured_by_session
    @requires_json_body
    @request_json_field_existence_validation(["user_id"])
    def get_user_messages(request):
        messages = self._messages_dao.get_messages_for_user_id(request.json_body[
↪"user_id"])
        return ResponseBuilder() \
            .set_status(HTTPStatus.OK) \
            .set_utf8_body(json.dumps(messages)) \
            .build()


class RequestSessionBuildingInterceptor:

    def __init__(self, sessions_dao):
        self._sessions_dao = sessions_dao

    def load_session_onto_request(self, request):
        request.session = None
        if "AUTH" not in request.headers:
            return request

        if not request.headers["auth"]:
            return request

        request.session = self._sessions_dao.get_valid_session_or_none(request.
↪headers["auth"])
        return request


def request_secured_by_session(decorated_function):
    @functools.wraps(decorated_function)
    def decorator(handler, request, *args, **kwargs):
        if not request.session:
            return ResponseBuilder() \
                .set_status(HTTPStatus.UNAUTHORIZED) \
                .set_utf8_body("Request require a valid session. Please login.") \
                .build()
        return decorated_function
    return decorator


def build_application():
```

(continues on next page)

(continued from previous page)

```
    configuration = ConfigLoader(os.environ, sys.argv).load()
    database_pool = MySQLDBConnectionPool(configuration.get_database_config())

    messages_dao = MySQLMessagesDAO(database_pool)
    sessions_dao = MySQLSessionsDAO(database_pool)

    request_session_building_interceptor = RequestSessionBuildingInterceptor(sessions_
↪dao)
    messages_handler = MessagesHandler(messages_dao)

    routes = \
        RoutesBuilder() \
            .add_request_interceptor("/", request_session_building_interceptor.load_
↪session_onto_request) \
            .add_handler("GET", "/messages", messages_handler.get_user_messages) \
            .build()

    return EynnydWebappBuilder() \
            .set_routes(routes) \
            .build()

application = build_application()
```

So what we have is an application with a single *Route* which returns a list of messages from our database given a `user_id`. This *Route* is secured by an authorization header. We use the request *Interceptor* `request_session_building_interceptor.load_session_onto_request` to load a valid session onto the request object and then use the `@request_secured_by_session` decorator to make the decision what to do if it isn't there. The value here is that we can now wrap any *Handler* we want to be secured using the `@request_secured_by_session` but if we have a non secured endpoint (for example a register endpoint) then we can simply leave off the decorator and it is not secured. The information about the endpoint being secured is at the definition site of the function, where it should be. Because the *Interceptor* is built ahead of time, database access can be injected into it (where as this would involve something hackish to do inside the decorator).

Now the *Interceptor* has one job: loading the session onto the request. The decorator has one job: returning an error response if the valid session does not exist. The *Handler* method has one job: getting the messages for the user id.

We will discuss all the parts of this code in much further detail below.

The Handler

First we have our *Handler* who's responsibility is to get messages for a user. Ideally all other code isn't in the *Handler* so that we don't obfuscate the code.

```
class MessagesHandler:

    def __init__(self, messages_dao):
        self._messages_dao = messages_dao

    @request_secured_by_session
    @requires_json_body
    @request_json_field_existence_validation(["user_id"])
    def get_user_messages(request):
        messages = self._messages_dao.get_messages_for_user_id(request.json_body[
↪"user_id"])
```

(continues on next page)

(continued from previous page)

```
        return ResponseBuilder() \
            .set_status(HTTPStatus.OK) \
            .set_utf8_body(json.dumps(messages)) \
            .build()
```

Note that the code in the *Handler* function clearly states how we get the messages for the user and nothing else. However, using decorators we can see that before this function executes we:

1. Secure our request for sessions
2. Validates the body has json content (and in this case loads the json into request.json_body).
3. Validates that the json contains a field keyed on "user_id"

This is a lot of logic that is no longer muddying what our *Handler* does, but is still clearly visible as being executed for this *Handler*. More importantly, the many other *Handler* who would need this same functionality can have it, in a readable fashion, without obfuscating their logic either.

Also different from the other tutorials, this *Handler* is inside an object. We do this so that we can take advantage of dependency injection. We injected a messages data access object (DAO) into this handling class. This class does not care that this DAO is connecting us to a MySQL database, only that it has a method called `get_messages_for_user_id` that takes a `user_id` and returns a list of messages.

The Interceptor

The next piece of code to look at is the class holding our *Interceptor*:

```
class RequestSessionBuildingInterceptor:

    def __init__(self, sessions_dao):
        self._sessions_dao = sessions_dao

    def load_session_onto_request(self, request):
        request.session = None
        if "auth" not in request.headers:
            return request

        if not request.headers["auth"]:
            return request

        request.session = self._sessions_dao.get_valid_session_or_none(request.
↪headers["auth"])
        return request
```

As in the *Handler* above we have put this method inside a class because we want to exploit dependency injection of our sessions data access object.

You can quickly see that all this method does is either load a session onto the request from the database or it sets the value to None. We actually wouldn't use `None` for this generally, but rather optionals, but we figured this tutorial was not the platform to discuss that.

As should be expected, this *Interceptor* has nothing to do with getting a response back to the user, it simply mutates the request, loading new values onto it. We have removed the unnecessary exception raising from our *Interceptor* and saved ourselves one less violation of exceptions as control flow.

The Decorator

Instead of throwing exceptions and using *Error Handler*s to return a bad response we instead have a python decorator wrap our *Handler* function. The code for this decorator looks like:

```
def request_secured_by_session(decorated_function):
    @functools.wraps(decorated_function)
    def decorator(handler, request, *args, **kwargs):
        if not request.session:
            return ResponseBuilder() \
                .set_status(HTTPStatus.UNAUTHORIZED) \
                .set_utf8_body("Request require a valid session. Please login.") \
                .build()
        return decorated_function
    return decorator
```

All this decorator does is check if the *Interceptor* put a valid session onto the request. If it didn't we return an UNAUTHORIZED status response. If a valid session is present we call through to the wrapped function.

Wiring Up Dependencies

Another change you might have seen in this tutorial is that we build up a series of objects before we start building our *Route*s. These objects are our dependency chain. The code looks like:

```
configuration = ConfigLoader(os.environ, sys.argv).load()
database_pool = MySQLDBConnectionPool(configuration.get_database_config())

messages_dao = MySQLMessagesDAO(database_pool)
sessions_dao = MySQLSessionsDAO(database_pool)

request_session_building_interceptor = RequestSessionBuildingInterceptor(sessions_dao)
messages_handler = MessagesHandler(messages_dao)
```

First we have an object which loads configuration from various sources (the environment, command line, and any configuration files we happen to read in). We need this configuration to build other dependencies.

Next we have a database pool connection which requires a selection of values from our configuration result.

Then we have two DAOss, the `messages_dao` and the `sessions_dao`. Note that on the right side of the assignment here we care that this is a MySQL implementation but on the left we just care that it is a DAO. In a statically typed language we would be using an interface on the left, but this is python, so life is easier. Note that into the DAOs we inject our database pool. These DAOs dont care about the specifics of our MySQL driver, only that they can execute sql commands against a database.

Now that we have our DAOs we can build our *Interceptor*s and *Handler*s. For this tutorial we just have the one of each. Into each of these we inject our built DAOs.

This kind of dependency build up allows code to be easy to read, debug, extend, and maintain. In fact, in his book **:ref:'Clean Architecture <https://www.amazon.com/Clean-Architecture-Craftsmans-Software-Structure/dp/0134494164>'__** Robert C. Martin makes a very strong argument that dependency inversion like this is the only real advantage OO gave us. Several other WSGI frameworks prevent this kind of dependency injection.

Setting Up The Routes

Finally we have code which should look pretty familiar at this point throughout the tutorials. We build our *Route*s:

```
routes = \
    RoutesBuilder() \
        .add_request_interceptor("/", request_session_building_interceptor.load_
↪session_onto_request) \
        .add_handler("GET", "/messages", messages_handler.get_user_messages) \
        .build()
```

The only reason to call attention to it here is so that you see how the function assignment works with *Interceptor*s and *Handler*s which have been encapsulated into classes.

Tutorial: Adding Values To Requests

Requests are preloaded in Eynnyd with all the values from the raw WSGI request. However, as you are processing your request, you may wish to add additional details to it. For example, on routes secured by a session, you may want to load that session from your database in an *request interceptor* and put that loaded value onto your request for later use (without reloading it). Because python allows you to manipulate objects after they have been built, you could do this simply by doing `request.session = session_dao.get_session(request.headers["session"])` but that wouldn't be very explicit and mutation like this can lead to hidden bugs, surprised readers, and much more. A more explicit way of doing this is shown below.

Some might argue that the mutation method we just talked about is more "pythonic" than the explicit version we are about to show you, however we would direct them to the zen of python (type `import this` into any python terminal) which specifically (and correctly) states: "Explicit is better than implicit".

As a simple example, let's assume we want to add a random ID to every request so that when we log things about it the ID can be matched up.

This tutorial builds on the *response interceptors tutorial* so if you have not read that yet, and you find something confusing in here, it is recommended you look there for your answer. First we will show you the code and then we will explain the *relevant* parts (AKA the parts not in the prior tutorials).

```
# hello_world_app.py
import logging

from eynnyd import AbstractRequest
from eynnyd import RoutesBuilder
from eynnyd import EynnydWebappBuilder
from eynnyd import ResponseBuilder
from eynnyd import ErrorHandlersBuilder

from http import HTTPStatus
import uuid

LOG = logging.getLogger("hello_world_app")

class IDEnhancedRequest(AbstractRequest):

    def __init__(self, original_request, request_id):
        self._request_id = request_id
        self._original_request = original_request

    @property
    def request_id(self):
        return self._request_id

    @property
```

(continues on next page)

(continued from previous page)

```
    def http_method(self):
        return self._original_request.http_method

    @property
    def request_uri(self):
        return self._original_request.request_uri

    @property
    def forwarded_request_uri(self):
        return self._original_request.forwarded_request_uri

    @property
    def headers(self):
        return self._original_request.headers

    @property
    def client_ip_address(self):
        return self._original_request.client_ip_address

    @property
    def cookies(self):
        return self._original_request.cookies

    @property
    def query_parameters(self):
        return self._original_request.query_parameters

    @property
    def path_parameters(self):
        return self._original_request.path_parameters

    @property
    def byte_body(self):
        return self._original_request.byte_body

    @property
    def utf8_body(self):
        return self._original_request.utf8_body

    def __str__(self):
        return "[{i}]<{m} {p}>".format(i=self._request_id, m=self.http_method, p=self.
↪request_uri)

def hello_world(request):
    return ResponseBuilder() \
        .set_status(HTTPStatus.OK) \
        .set_utf8_body("Hello World")\
        .build()

def add_id_to_request(request):
    return IDEnhancedRequest(request, uuid.uuid4())

def log_request(request):
    LOG.info("Got Request: {r}".format(r=request))
    return request

def log_response(request, response):
```

(continues on next page)

(continued from previous page)

```
    LOG.info("Built Response: {s} for Request: {r}".format(s=response, r=request))
    return response

def build_application():
    routes = \
        RoutesBuilder() \
            .add_request_interceptor("/", add_id_to_request) \
            .add_request_interceptor("/", log_request) \
            .add_handler("GET", "/hello", hello_world) \
            .add_response_interceptor("/", log_response)\
            .build()

    return EynnydWebappBuilder() \
            .set_routes(routes) \
            .build()

application = build_application()
```

New Imports

For our new work we need two new imports.

```
from eynnyd import AbstractRequest
...
import uuid
```

The AbstractRequest represents all the functionality a request must contain (at minimum). These values are already provided to your code, loaded from the WSGI server. We are also using the uuid module here to generate us random IDs. Collisions this way are pretty uncommon and since these IDs are short lived (only for the duration of a request) we feel this method is pretty reasonable.

Building An Explicit Request Wrapper Class

We now build a class which implements our Eynnyd `AbstractRequest` from above explicitly and it also provides us with a `request_id property`.

```
class IDEnhancedRequest(AbstractRequest):

    def __init__(self, original_request, request_id):
        self._request_id = request_id
        self._original_request = original_request

    @property
    def request_id(self):
        return self._request_id

    @property
    def http_method(self):
        return self._original_request.http_method

    @property
    def request_uri(self):
        return self._original_request.request_uri
```

(continues on next page)

(continued from previous page)

```
    @property
    def forwarded_request_uri(self):
        return self._original_request.forwarded_request_uri

    @property
    def headers(self):
        return self._original_request.headers

    @property
    def client_ip_address(self):
        return self._original_request.client_ip_address

    @property
    def cookies(self):
        return self._original_request.cookies

    @property
    def query_parameters(self):
        return self._original_request.query_parameters

    @property
    def path_parameters(self):
        return self._original_request.path_parameters

    @property
    def byte_body(self):
        return self._original_request.byte_body

    @property
    def utf8_body(self):
        return self._original_request.utf8_body

    def __str__(self):
        return "[{i}]<{m} {p}>".format(i=self._request_id, m=self.http_method, p=self.
↪request_uri)
```

There are 3 unique things to note about this class. The first is that every property except the `request_id` property just returns the value from the original request object. The second note-worthy item is the `request_id` property itself, which just returns any id set in the constructor. And finally, it is also worth noting we have updated our `__str__` method, which means that any logging of this request will now start with a prefixed request id value.

Updating the Request

Now we just need a *request interceptor* to update our incoming request with new values.

```
def add_id_to_request(request):
    return IDEnhancedRequest(request, uuid.uuid4())
```

Because `IDEnhancedRequest` extends `AbstractRequest` this code is legal (wont fail Eynnyd's request interceptor validation). All we are doing is returning the wrapped request with a newly added, random id.

Adding the Interceptor

Finally, we just need to add this interceptor to our routes at the root level and make sure it runs before all other interceptors.

```
def build_application():
    routes = \
        RoutesBuilder() \
            .add_request_interceptor("/", add_id_to_request) \
            .add_request_interceptor("/", log_request) \
            .add_handler("GET", "/hello", hello_world) \
            .add_response_interceptor("/", log_response)\
            .build()
```

Note we added this interceptor before the other root interceptors to insure it runs first. With this change both the `log_request` request interceptor and the `log_response` response interceptor will log out the request including our new id value.

2.1.5 Contributing

Bugs and Features

If you have ideas for new features or bug fixes that you would like done in Eynnyd please feel free to open an issue in our Github issue page.

Open for Contributions

Eynnyd is a project of passion, not one to make money, so we welcome any help we can get. If you want to contribute, feel free to fork our repo, and clone the fork to your machine. From there you will want to be able to run the test suite.

```
pip install -r test_requirements.txt
```

Will install everything you need to run the tests.

```
python -m unittest discover tests
```

Will run all of the tests.

You will also want to make sure you maintain 100% test coverage for anything you add. Once you have installed the test requirements (as above) you can run a coverage report via:

```
coverage run --source eynnyd/ -m unittest discover tests/
coverage report
```

Generating Documentation

If you want to build a local copy of these documents to your own machine you can run:

```
pip install -r documents_requirements.txt
cd docs/
make html
```

This will generate a file at `docs/build/html/index.html` that can be opened using your browser.

2.1.6 Frequently Asked Questions

- **Another WSGI web framework? Why not just fix (or add on to) an existing one?**

We tried that first. We pared our merge request down to only a minor element of Eynnyd and still our contribution was deemed "too big" to merge. To get the framework we always wished we had we decided the more direct route was to build our own.

- **What about speed? How fast are you compared to other frameworks?**

We are first prioritizing code quality and correctness. Once we nail those we will worry about speed. Our goal is not now, nor will it ever be, to be the fastest framework out there, however we do aim to perform within the range of other frameworks, such that Eynnyd is an acceptable solution for production applications.

- **What python versions do you support and why?**

We use functools.lru_cache() which limits us to python >3.2 but since python 2 is about to be deprecated we feel this isn't a limiting constraint. We may add a different caching system, or backwards compatibility for python 2 in the near future, but this does not feel like a major priority.

2.2 API Documentation

All of our public api is documented. Our internal code is intentionally, mostly, not documented (see Clean Code Chapter 4 for why.

2.2.1 Request

class `eynnyd.abstract_request.`**`AbstractRequest`**

The expected interface for a request.

If you want to build your own request object it needs to meet the requirements set out in this class.

`abstract property http_method`

The HTTP method used for the request. This gets matched against the handler routes and must be an exact match.

Returns a string like: "GET", "PUT", "POST", "DELETE", etc.

`abstract property request_uri`

The request uri encapsulates the scheme, host, port, path, and query.

Returns An Eynnyd RequestURI object with properties for scheme, host, port, path, and query

`abstract property forwarded_request_uri`

The request uri but using forwarded info.

Returns An Eynnyd RequestURI object with properties for scheme, host, port, path, and query

`abstract property headers`

The HTTP headers from the request

Returns A dictionary of header names to header values

`abstract property client_ip_address`

The ip address of the remote user

Returns a string ip address

`abstract property cookies`

The cookies from the request

Returns A dictionary of cookie name to Eynnyd Cookie objects with name and value properties

abstract property query_parameters
The query part of the request

Returns a dictionary of parameter names to lists of parameter values

abstract property path_parameters
The parameters set in the path of request matching against pattern matching path parameters.

Returns A dictionary of path variable name to request path value.

abstract property byte_body
The raw request body

Returns The body of the request left encoded as bytes.

abstract property utf8_body
The encoded request body

Returns The request body after being encoded to utf-8

2.2.2 Response

class `eynnyd.abstract_response.`**AbstractResponse**
The expected interface for a response.

If you want to build your own response objects it must meet the contract of this class

abstract property status
The status of the response

Returns An instance of an Eynnyd HTTPStatus object with code and phrase properties

abstract property body
The body of the response

Returns An instance of an Eynnyd ResponseBody with type and content properties

abstract property headers
The headers of the response

Returns A dictionary of header name to header value

abstract property cookies
The cookies of the response

Returns a list of Eynnyd ResponseCookie objects

2.2.3 Error Handlers Builder

class `eynnyd.error_handlers_builder.`**ErrorHandlersBuilder**
An object for setting handlers for any exceptions that can come up.

There are two times an error can be thrown in the process of turning a request into a response. Either the error occurs before we have a response or after. Handlers should be set based on where the exception is expected (or in both places if it can come up anywhere).

Handling will prefer the most specific exception but will execute against a base exception if one was set.

Several default handlers are set if they are not set manually. The defaults registered are for RouteNotFound, InvalidCookieHeader, and Exception.

`__init__`()
Initialize self. See help(type(self)) for accurate signature.

`add_pre_response_error_handler`(*error_class*, *handler*)
Add error handlers which happen before we have built a response (returned from a handler).

Parameters

- **`error_class`** – The class to execute the handler for.
- **`handler`** – A function which takes a request as a parameter.

Returns This builder so that fluent design can optionally be used.

`add_post_response_error_handler`(*error_class*, *handler*)
Add error handlers which happen after we have built a response.

Parameters

- **`error_class`** – The class to execute the handler for.
- **`handler`** – A function which takes both a request and response parameter.

Returns This builder so that fluent design can optionally be used.

`build`()
set defaults and build the error handlers for setting into the Eynnyd WebAppBuilder.

return The ErrorHandlers required by the Eynnyd WebAppBuilder set_error_handlers method.

2.2.4 Exceptions

exception `eynnyd.exceptions.`**`EynnydWebappBuildException`**
Raised if there is a problem with the configured webapp when it attempts to build.

exception `eynnyd.exceptions.`**`ErrorHandlingBuilderException`**
Raised when there is a problem with adding an error handler.

exception `eynnyd.exceptions.`**`NoGenericErrorHandlerException`**
Raised when attempting to handle an exception and there are no configured handlers (even the base Exception). This should not happen.

exception `eynnyd.exceptions.`**`HandlerNotFoundException`**
This is raised when a handler is not found for a request. Note that this is caught and rethrown as a more generic RouteNotFoundException which is what is used for retuning 404 responses.

exception `eynnyd.exceptions.`**`DuplicateHandlerRoutesException`**
Raised when trying to add a handler to the routes which is already registered for that method/path pair. Note this is caught and rethrown as a more generic RouteBuildException.

exception `eynnyd.exceptions.`**`RouteNotFoundException`**
Raised when no route could be found for a request. Indicates a 404.

exception `eynnyd.exceptions.`**`RouteBuildException`**
Raised when there is a problem with route building.

exception `eynnyd.exceptions.`**`InvalidHTTPStatusException`**
Raised when the status given to a response cannot be translated into a recognizable format.

exception `eynnyd.exceptions.`**`SettingNonTypedStatusWithContentTypeException`**
Raised when a response has a content type but trying to set a non content type status.

exception eynnyd.exceptions.**SettingContentTypeWithNonTypedStatusException**
Raised when a response has a non content type status but trying to set a content-type.

exception eynnyd.exceptions.**SettingBodyWithNonBodyStatusException**
Raised when a response has a non body type status but trying to set a body.

exception eynnyd.exceptions.**SettingNonBodyStatusWithBodyException**
Raised when a response has a body but trying to set a non body status.

exception eynnyd.exceptions.**InvalidURIException**
Raised when a uri (for example the path given to a route) is badly formatted.

exception eynnyd.exceptions.**InvalidCookieBuildException**
Raised when attempting to build a cookie with bad values.

exception eynnyd.exceptions.**InvalidCookieHeaderException**
Raised when a request cookie is not rfc compliant (Ideally, should not happen).

exception eynnyd.exceptions.**InvalidHeaderException**
Raised when a response header uses invalid characters.

exception eynnyd.exceptions.**InvalidBodyTypeException**
Raised when a body being set on a response is invalid (ex. setting a utf-8 body using the set_byte_body method).

exception eynnyd.exceptions.**UnknownResponseBodyTypeException**
Raised when a body is set on the response with an unknown type (should not happen).

exception eynnyd.exceptions.**RequestInterceptorReturnedNonRequestException**
Raised when a request interceptor does not return a valid request object.

exception eynnyd.exceptions.**HandlerReturnedNonResponseException**
Raised when a handler does not return a valid response object.

exception eynnyd.exceptions.**ResponseInterceptorReturnedNonResponseException**
Raised when a response interceptor does not return a valid response object.

exception eynnyd.exceptions.**NonCallableInterceptor**
Raised when a interceptor is registered without being a callable.

exception eynnyd.exceptions.**NonCallableHandler**
Raised when a handler is registered without being a callable.

exception eynnyd.exceptions.**CallbackIncorrectNumberOfParametersException**
Raised when a callback doesn't match the correct number of parameters.

exception eynnyd.exceptions.**NonCallableExceptionHandlerException**
Raised when an exception handler is registered but is not a callable.

exception eynnyd.exceptions.**InvalidResponseCookieException**
Raised when a response cookie is of a non rfc compliant format.

exception eynnyd.exceptions.**ExecutionPlanBuildException**
Raised when an execution plan is finished but cannot build. (should not happen)

2.2.5 Eynnyd Webapp Builder

class eynnyd.eynnyd_webapp_builder.**EynnydWebappBuilder**

__init__()
Initialize self. See help(type(self)) for accurate signature.

set_routes(*root_tree_node*)
Sets routes built by the Eynnyd RoutesBuilder

Parameters **root_tree_node** – the result from the Eynnyd RoutesBuilder build method

Returns This builder so that fluent design can be used.

set_error_handlers(*error_handlers*)
Sets the error handlers built by Eynnyd ErrorHandlersBuilder :param error_handlers: the result from the Eynnyd ErrorHandlersBuilder create method :return: This builder so that fluent design can be used

build()
Builds the webapp

Returns the WSGI compliant webapp

2.2.6 Response Builder

class eynnyd.response_builder.**ResponseBuilder**
A builder allowing for the easy and validated building of a Response.

__init__()
Initialize self. See help(type(self)) for accurate signature.

set_status(*status*)
Sets the HTTP status of the response. Raises if the type conflicts with other attributes of the response.

Parameters **status** – a value indicating response status (can be an int, http.HTTPStatus or Eynnyd HttpStatus)

Returns This builder to allow for fluent design.

set_utf8_body(*body*)
Sets a utf8 body on the request (overwriting any other set body). Raises if setting the body conflicts with the status. Sets a content-length header if one has not already been set.

Parameters **body** – The utf-8 encoded body

Returns This builder to allow for fluent design.

set_byte_body(*body*)
Sets a byte body on the request (overwriting any other set body). Raises if setting the body conflicts with the status. Sets a content-length header if one has not already been set.

Parameters **body** – The bytes encoded body

Returns This builder to allow for fluent design.

set_stream_body(*body*)
Sets a streaming body on the request (overwriting any other set body). Raises if setting the body conflicts with the status.

Parameters **body** – A streamable object with a read method taking 1 parameter (and an optional close method)

Returns This builder to allow for fluent design.

set_iterable_body(*body*)
Sets an iterable body on the request (overwriting any other set body). Raises if setting the body conflicts with the status.

For simple strings you should use the utf-8 body as using this would be highly inefficient.

Parameters **body** – The iterable body

Returns This builder to allow for fluent design.

`unset_body`()
Unsets the body on the request.

Returns This builder to allow for fluent design.

`set_headers`(*headers_by_name*)
Sets the headers on the response (deleting/overwriting all current headers).

Raises if this method is attempted to be used to set cookies. Use the set_cookies/add_cookie methods for that.

Parameters **`headers_by_name`** – a dictionary of header values keyed by name

Returns This builder to allow for fluent design.

`add_header`(*name*, *value*)
Adds a single header to the response.

Raises if this method is attempted to be used to set cookies. Use the set_cookies/add_cookie methods for that.

Parameters

- **`name`** – the name to use for the header
- **`value`** – the value to store in the header

Returns This builder to allow for fluent design.

`remove_header`(*name*)
Removes a header from the response by name.

Parameters **`name`** – The name for the header to remove.

Returns This builder to allow for fluent design.

`set_cookies`(*cookies*)
Sets all the cookies on the response (deleting/overwriting any previously set).

Parameters **`cookies`** – An iterable of Eynnyd ResponseCookie objects.

Returns This builder to allow for fluent design.

`add_cookie`(*cookie*)
Adds a single cookie to the response.

Parameters **`cookie`** – an Eynnyd ResponseCookie object.

Returns This builder to allow for fluent design.

`add_basic_cookie`(*name*, *value*)
Adds a simple cookie to the response (allowing the skipping of using Eynnyd specific objects).

Parameters

- **`name`** – An rfc valid name for the cookie.
- **`value`** – An rfc valid value for the cookie.

Returns This builder to allow for fluent design.

`remove_cookie`(*name*)
Removes a cookie from the response by name.

Parameters **`name`** – the name of the cookie to remove.

Returns This builder to allow for fluent design.

build()

Returns A response ready for returning from the webapp.

2.2.7 Response Cookie Builder

class eynnyd.response_cookie_builder.**ResponseCookieBuilder**(*name*, *value*)

Response cookies are generally just key-value pairs but can be more complicated. Using this builder allows for simple creation of generic or complex cookies with validation.

__init__(*name*, *value*)

Constructs an initial ResponseCookieBuilder with common defaults.

Parameters

- **name** – a valid cookie name (via rfc spec)
- **value** – a valid cookie value (via rfc spec)

set_expires(*expires*)

Sets the cookie to include an expiry date.

Parameters **expires** – a date, parsable by python Arrow

Returns This builder to allow for fluent design.

set_expires_in_days(*days_till_expiry*)

Sets the cookie to include an expiry date in days from now.

Parameters **days_till_expiry** – number of days from now to the expiry time.

Returns This builder to allow for fluent design.

set_max_age(*max_age*)

Sets the max age of the cookie.

Parameters **max_age** – an rfc compliant max age

Returns This builder to allow for fluent design.

set_domain(*domain*)

Sets the limiting domain for the cookie.

Parameters **domain** – an rfc compliant domain

Returns This builder to allow for fluent design.

set_path(*path*)

Sets the limiting path for the cookie.

Parameters **path** – an rfc compliant path

Returns This builder to allow for fluent design.

set_secure(*secure*)

Sets whether the cookie is secure or not.

Parameters **secure** – a boolean to indicate if we are setting secure or insecure

Returns This builder to allow for fluent design.

set_http_only(*http_only*)

Sets whether the cookie is http only or not.

Parameters **`http_only`** – a boolean to indicate if we are setting http only or not

Returns This builder to allow for fluent design.

`build`()

Validates the name and value and builds the ResponseCookie object. :return: a valid ResponseCookie object.

2.2.8 RoutesBuilder

class `eynnyd.routes_builder.`**`RoutesBuilder`**

A builder for registering request interceptors, handlers, and response interceptors.

`__init__`()

Initialize self. See help(type(self)) for accurate signature.

`add_request_interceptor`(*uri_path*, *interceptor*)

Adds a request interceptor to be run (before handler execution) given a uri path for when to execute it

Parameters

- **`uri_path`** – The path dictating what requests this interceptor is run against
- **`interceptor`** – A function which takes a request parameter and returns a request

Returns This builder to allow fluent design

`add_response_interceptor`(*uri_path*, *interceptor*)

Adds a response interceptor to be run (after handler execution) given a uri path for when to execute it

Parameters

- **`uri_path`** – The path dictating what requests this interceptor is run against
- **`interceptor`** – A function which takes a request and a response and returns a response

Returns This builder to allow for fluent design

`add_handler`(*http_method*, *uri_path*, *handler*)

Adds a handler to be run (after request interceptors and before response interceptors) given a http method and uri path for when to execute it

Parameters

- **`http_method`** – the method to match to execute this handler against a request
- **`uri_path`** – The path dictating what requests this handler is run against
- **`handler`** – A function taking a request and returning a response

Returns This handler to allow for fluent design

`build`()

Builds out the route tree for processing requests into responses.

Returns The route tree for usage in the Eynnyd WebAppBuilder

PYTHON MODULE INDEX

INDEX