Kyoukai Documentation

Release 2.1.3

Isaac Dickinson

Apr 03, 2017

Main usage

CHAPTER 1

About

Kyoukai is a fast asynchronous Python server-side web microframework. It is built upon `asyncio` and Asphalt for an easy to use web server.

Kyoukai is Flask inspired; it attempts to be as simple as possible, but without underlying magic to make it confusing.

CHAPTER 2

Installation

Kyoukai depends heavily on the asyncio library provided by Python3.4+, and certain language features added in Python 3.5. This means the library is not compatible with code that does not use Python 3.5 or above.

Kyoukai is shipped as a PyPI package, so can be installed easily with pip.

```
$ pip install kyoukai
```

Alternatively, if you want cutting edge, you can install directly from git.

```
$ pip install git+https://github.com/SunDwarf/Kyoukai.git
```

Note that the Git version is not guarenteed to be stable, at all.

CHAPTER 3

Contents:

Your First Kyoukai App

In this tutorial, we'll go through how to write your first Kyoukai app.

Application Skeleton

Strap in with your favourite IDE, and create your first new project. Name it something silly, for example `my-first-kyokai-project`. The name doesn't matter, as you probably won't be using it for long.

Directory layout

Kyoukai projects have a very simple layout.

```
$ ls --tree

- app.py
- static
- templates
```

There are three components here:

- `app.py`
  - This contains the main code for your app. This can be named absolutely anything, but we're naming it `app` for simplicity's sake.
- `templates`
  - This contains all the templates used for rendering things server-side, or for putting your JS stack of doom inside.
- `static`

  - This contains all the static files, such as your five bootstrap theme CSS files, or the millions of JS libraries you've included.

Writing the App

Open up `app.py` and add your starting line.

```
from kyoukai import Kyoukai
```

This imports the Kyoukai application class from the library, allowing you to create a new object inside your code.

Routes

Routes in Kyoukai are very simple, and if you have ever used Flask, are similar in style to Flask routes.

Routes are made up of three parts:

- The path
  - This is a Werkzeug-based route path that uses Werkzeug to match route paths. For more information, see http://werkzeug.pocoo.org/docs/0.11/routing/ .
- The allowed methods
  - This is a list, or set, or other iterable, of allowed HTTP/1.1 methods for the route to handle. If a method (e.g `GET`) is not in the list, the route cannot handle it, and a HTTP 405 error will automatically be passed to the client.
- The route itself
  - Your route is a coroutine that accepts one argument, by default: the a new *`HTTPRequestContext`*, containing the request data and other context specific data.

```
async def some_route(ctx: HTTPRequestContext): ...
```

We are going to write a very simple route that returns a `Hello, world!` file.

Creating the route

Routes in Kyoukai are created very similarly to Flask routes: with a decorator.

```
@app.route("/path", methods=["GET", "POST"])
```

As explained above, the route decorator takes a path and a method.
This route decorator returns a Route class, but this isn't important right now.

The Route Coroutine

Your route function **must** be a coroutine. As Kyoukai is async, your code must also be async.

```
@app.route("/")
async def index(ctx): ...
```

Inside our route, we are going to return a string containing the rendered text from our template.

Templates

Templates are stored in `templates/`, obviously. They are partial HTML code, which can have parts in it replaced using code inside the template itself, or your view.

For now, we will put normal HTML in our file.

Open up `templates/index.html` and add the following code to it:

```
It's current year, and you're still using blocking code? Not <em>me!</em>
```

(note: do not replace current year with the actual current year.)

Save and close the template.

Rendering the template

Rendering the template requires an Asphalt extension - Asphalt Rendering. Once configured and installed, it can be used to render your template easily.

You can add it to your brand new route like so:

```
@app.route("/")
async def index(ctx):
    return ctx.jinja2.render("index.html")
```

Replace `jinja2` with the appropriate rendering engine you selected.

Responses

Note, how in the previous coroutine, we simply returned a `str` in our route. This is not similar to `aiohttp` and the likes who force you to return a `Response`. You can return a response object in Kyoukai as normal, but for convenience sake, you can also return simply a string or a tuple.

These are transparently converted behind the scenes:

```
r = Response(code=route_result[1] or 200, body=route_result[0], headers=route_
↪result[2] or {})
```

That is, the first item is converted to your response body, the second item (or 200 by default) is used as the response code, and the third code is used as the headers.

Note: All return params except the first is optional, if you do not return a Response object.

Running your App

The ideal way of running a Kyoukai project is through the Asphalt framework. See *Asphalt usage* for more information on how to use this.

However, Kyoukai includes a built-in way of running the app from blocking code.

```
app.run(ip="127.0.0.1", port=4444)
```

Warning: Whilst using `app.run`, you will not have Asphalt Rendering enabled in your configuration.

The args passed in here are just the default values; they are optional.
Open up your web browser and point it to http://localhost:4444/. If you have done this correctly, you should see something like this:

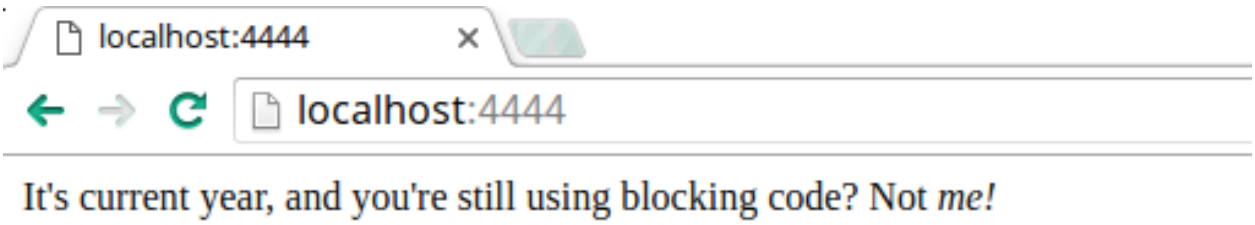

Fig. 3.1: example 1

Deploying

There's no special procedure for deploying your app. The inbuilt webserver is production ready, and you can run your application in a production environment in the same way as you would develop it.

Finishing your project

You have completed your first Kyoukai project. For maximum effectiveness, you must now publish it to GitHub.

```
$ git init
$ git remote add origin git@github.com:YourName/my-first-kyoukai-project.git
$ git add .
$ git commit -a -m "Initial commit, look how cool I am!"
$ git push -u origin master
```

Asphalt usage

The **Asphalt Framework** is a microframework for asyncio-based applications and libraries, providing useful utilities and common functions to projects built upon it.

It also provides a common interface for applications to use *components* to enhance the functionality in an easy asynchronous way.

Config File

The core part about adding Asphalt to your project is the `config.yml` file that exists at the core of every app. This defines how the application should be ran, and what settings each component within should have.

These config files are standard YAML files, with one document. An example file for a Kyoukai project would be:

```
---
component:
  type: kyoukai.asphalt:KyoukaiComponent
  app: app:kyk
```

Let's break this down.

1. First, you have the `component:` directive. This signifies to Asphalt that you wish to define a list of components to add to your project.

2. Next, you have the `type` directive. This tells Asphalt what type of component to use in the application.

   In this example, the `KyoukaiComponent` is specified directly, meaning that you wish the framework to create a single-component application, with the root component being Kyoukai's handler.

3. Finally, the `app` directive. This tells the `KyoukaiComponent` to use the app specified by the string here.

   In `app:kyk`, the first part before the : signifies the FULL IMPORT NAME (the name you would use in an import statement, e.g `import app`), and the second part signifies the object to use.

To run an app using Asphalt, you merely need to run:

```
asphalt run config.yml
```

The Asphalt runner will automatically run and load your application.

Adding Components

Components are a way of adding useful parts to your project without additional manual set up. In this example, we will add a SQLAlchemy component to the app.

The Container

First, a new **container** object is required to store the components that are added to the application. Every container is inherited from `asphalt.core.component.ContainerComponent` in order to add components to the app.

We're gonna start with a small project layout:

```
$ ls --tree

- application
|   - container.py
- static
- templates
```

This will be the basic project format from now on.

Inside `container.py`, add the following code:

```
from asphalt.core import ContainerComponent, Context
from kyoukai import Kyoukai
from kyoukai import KyoukaiComponent

app = Kyoukai("api")


class AppContainer(ContainerComponent):
     async def start(self, ctx: Context):
        self.add_component('kyoukai', KyoukaiComponent, ip="127.0.0.1", port=4444,
                           app=app)

        await super().start(ctx)
```

That's a lot of code to process. Let's break it down again.

1. First, you have the creation of the app. Nothing unusual here.
2. Next, the definition of a subclass for the app. This container contains a set of components, which are added to the app at run time, and configured appropriately.
3. The addition of the KyoukaiComponent to the app. This adds the Kyoukai handler to Asphalt, which configures the application to run with additional contexts.
4. A super call, which tells Asphalt to run our app immediately.

We're not done yet, however. Now, the config file needs to be run.

Add a basic configuration file, named `config.yml`, with this simple piece of code.

```
---
component:
  type: application.container:AppContainer
  components:
    kyoukai:
      ip: "127.0.0.1"
      port: 4444
```

This creates a new AppContainer instance, and edits the configuration of the Kyoukai component within to set the IP and port to the ones in the config file.

To run this application, it's as simple as the first Asphalt call:

```
asphalt run config.yml
```

Adding SQLAlchemy

Now that you've seen how to add basic components to your project, adding SQLAlchemy is easy.

Edit your `start` method in your `AppContainer` to add this line above your super call:

```
self.add_component('sqlalchemy', SQLAlchemyComponent)
```

Make sure to the add the import for this (`from asphalt.sqlalchemy.component import SQLAlchemyComponent`) too.

Next, in your config.yml, add a new section under `components`:

```
sqlalchemy:
   url: "sqlite3:///tmp/database.db"
   metadata: application.db:metadata
```

This will automatically configure a SQLite3 database at `/tmp/database.db` to run with your application.

Note that the reference for the metadata doesn't exist. You create your metadata like any other SQLAlchemy application, however you don't add an engine or a session. The engine and session are automatically provided.

Handling Errors Within Your Application

As with all code, eventually bugs and other exceptions will come up and risk ruining everything inside your app. Fortunately, Kyoukai handles these errors for you, and allows you to process them safely.

Error handlers are a way of handling errors easily. They are automatically called when an exception is encounted inside a route.

For example, if you have a piece of faulty code:

```
return "{}".format(a)  # 'a' is not defined
```

A `NameError` will normally be raised. However, Kyoukai will automatically catch the error, and re-raise it as a HTTP 500 exception. Normally, this exception wouldn't be handled, and would respond to the client with a `500` body. However, it is possible to catch this exception and do what you wish with it.

The `errorhandler` decorator

To create an error handler, you simply wrap an existing function with the `errorhandler` decorator, providing the integer error code that you wish to handle. So for example, to create a 500 error handler, you would do:

```
@app.root.errorhandler(500)
async def handle_500(ctx: HTTPRequestContext, exc: HTTPException):
    return repr(exception_to_handle)
```

Of course, you can have anything in the body of the error handler. Whatever is returned from this error handler is sent back to the client.

HTTP Exceptions

HTTP exceptions in Kyoukai are handled by Werkzeug, which prevents having to rewrite large amounts of the error handling internally.

For more information on Werkzeug's HTTPException, see `werkzeug.exceptions.HTTPException`.

To abort out of a function early, you can use `werkzeug.exceptions.abort()` to raise a HTTPException:

```
if something is bad:
    abort(404)
```

Blueprints

New in version 1.5.

Changed in version 2.1.2: Host Matching is now supported. See *Host Matching*.

In Kyoukai, routes are stored inside a tree structure consisting of multiple Blueprint objects with a parent and children. Each Blueprint contains a group of routes stored on it, which inherit the request hooks and the API prefix of all of its parents.

Blueprints are instantiated similar to app objects, with a name.

```
my_blueprint = Blueprint("my_blueprint")
```

Additionally, blueprints take an additional set of parameters which can be used to more finely control the behaviour.

- **`url_prefix`: The URL prefix to add to every request.** For example, if this is set to `/api/v1`, every request attached to this blueprint will only be accessible via ``/api/v1/<route>.

A note on the tree

Blueprints are stored inside a tree structure - that means that all Blueprints have a parent blueprint and 0 to N children blueprints.

When registering an error handler, or a request hook, children blueprints automatically inherit these unless they are overriden on the child level.

Routing

Routing with Blueprints is incredibly similar to routing with a bare app object. Internally, an `@app.route` maps to routing on an underlying Blueprint object used as the "root" blueprint.

```
@my_blueprint.route("/some/route")
async def some_route(ctx: HTTPRequestContext):
    return "Some route"
```

`Blueprint.`**`route`**(*routing_url*, *methods=('GET',)*, ***kwargs*)

Convenience decorator for adding a route.

This is equivalent to:

```
route = bp.wrap_route(func, **kwargs)
bp.add_route(route, routing_url, methods)
```

Error handlers

Error handlers with Blueprints are handled exactly the same as error handlers on bare app objects. The difference between these however is that error handlers are local to the Blueprint and its children.

```
@my_blueprint.errorhandler(500)
async def e500(ctx: HTTPRequestContext, err: Exception):
    return "Handled an error"
```

`Blueprint.`**`errorhandler`**(*code*)

Helper decorator for adding an error handler.

This is equivalent to:

```
route = bp.add_errorhandler(cbl, code)
```

Parameters **code** (int) – The error handler code to use.

Registering blueprints

If, after creating your blueprint, you attempt to navigate to `/some/route` you will find a 404 error living there instead. This is because you did not register the Blueprint to your application.

```
app.register_blueprint(my_blueprint)
```

Internally, this adds a child to the root blueprint, and sets the parent of the child to the root blueprint. If you have a blueprint that you wish to inherit from, you must register your Blueprint as a child of the Blueprint you wish to inherit from.

```
my_blueprint.add_child(my_other_blueprint)
```

Kyoukai.**register_blueprint**(*child*)

Registers a child blueprint to this app's root Blueprint.

This will set up the Blueprint tree, as well as setting up the routing table when finalized.

Parameters **child** (*Blueprint*) – The child Blueprint to add. This must be an instance of *Blueprint*.

Blueprint.**add_child**(*blueprint*)

Adds a Blueprint as a child of this one. This is automatically called when using another Blueprint as a parent.

Parameters **blueprint** (*Blueprint*) – The blueprint to add as a child.

Return type *Blueprint*

Requests and Responses

Requests and Responses are crucial parts of a HTTP framework - the request contains data that is received from the client, and the Response contains data that is sent to the Client.

Kyoukai piggybacks off of Werkzeug for its request and response wrappers - this means that most of the form logic and etc is handled by a well tested library used in thousands of applications across the web.

Getting the Request

The Request object for the current request is available on *request* for your route functions to use.

For example, returning a JSON blob of the headers:

```
async def my_route(ctx: HTTPRequestContext):
    headers = json.dumps(ctx.headers)
    return headers
```

Creating a Response

Responses are **automatically** created for you when you return from a route function or error handler. However, it is possible to create them manually:

```
async def my_route(ctx: HTTPRequestContext):
    return Response("Hello, world", status=200)
```

Response Helpers

New in version 2.1.3.

There are some built-in helper functions to encode data in a certain form:

kyoukai.util.**as_html**(*text*, *code=200*, *headers=None*)
Returns a HTML response.

```
return as_html("<h1>Hel Na</h1>", code=403)
```

Parameters

- **text** (str) – The text to return.
- **code** (int) – The status code of the response.
- **headers** (Optional[dict]) – Any optional headers.

Return type Response

Returns A new werkzeug.wrappers.Response representing the HTML.

kyoukai.util.**as_plaintext**(*text*, *code=200*, *headers=None*)
Returns a plaintext response.

```
return as_plaintext("hel yea", code=201)
```

Parameters

- **text** (str) – The text to return.
- **code** (int) – The status code of the response.
- **headers** (Optional[dict]) – Any optional headers.

Return type Response

Returns A new werkzeug.wrappers.Response representing the text.

kyoukai.util.**as_json**(*data*, *code=200*, *headers=None*, ***, *json_encoder=None*)
Returns a JSON response.

```
return as_json({"response": "yes", "code": 201}, code=201)
```

Parameters

- **data** (Union[dict, list]) – The data to encode.
- **code** (int) – The status code of the response.
- **headers** (Optional[dict]) – Any optional headers.

- **json_encoder** (Optional[JSONEncoder]) – The encoder class to use to encode.

Return type Response

Returns A new werkzeug.wrappers.Response representing the JSON.

Deploying Your App

Unlike some other frameworks, Kyoukai's built in web server is production ready and you do not need any specific setup to use your web application in production.

That said, if you want to get the best performance out of Kyoukai, you need to run the app with a special flag, the *-O* flag.

This flag is a builtin flag to the Python interpreter, and automatically skips costly *assert* statements that can slow down your app. This means you invoke the application with *python -O -m asphalt.core.command run config.yml*.

Request Hooks

Request hooks are a convenient way of performing actions before and after a request is processed by your code. There are several types of request hooks:

- **Global-level** request hooks, which take action on ALL routes. These can be technically seen as **root blueprint-level** hooks, since they act on the root blueprint.
- **Blueprint-level** request hooks, which take action at the blueprint level. These are registered on a blueprint, and act on **all routes** registered to that blueprint, *as well as* all routes registered to children blueprints.
- **Route-level** request hooks, which take action on individual routes.

All hooks must complete successfully. If any hook fails, then the request will fail with a 500 Internal Server Error.

Note: Global-level hooks are registered with app.add_hook and family, but actually redirect to the root blueprint.

Adding a Hook

Adding a hook can be done with *add_hook()* or *add_hook()*. These take a type param and a the hook function to add.

Alternatively, you can use the helper functions:

Blueprint.**before_request**(*func*)
Convenience decorator to add a pre-request hook.

Route.**before_request**(*func*)
Convenience decorator to add a post-request hook.

Blueprint.**after_request**(*func*)
Convenience decorator to add a post-request hook.

Route.**after_request**(*func*)
Convenience decorator to add a pre-request hook.

Pre-request hooks

Pre-request hooks are hooks that are fired before a request handler is invoked. They are fired in the order they are added.

Pre-request hooks take one param: the `HTTPRequestContext` that the request is going to be invoked with. They can either return the modified context, a new context, or None to use the previous context as the new one.

```
async def print_request(ctx: HTTPRequestContext):
    print("Request for", ctx.request.path)
    return ctx  # can be omitted to leave `ctx` in place
```

Post-request hooks

Post-request hooks are hooks that are fired after a request is invoked. They are fired in the order they are added.

Post-request hooks take two params: The `HTTPRequestContext` that the request was invoked with, and the **wrapped result** (**NOT** the final result!) of the response handler. They can either return a modified Response, or None to use the previous Response as the new one.

```
async def jsonify(ctx, response):
    if not isinstance(response.response, dict):
        return response

    r.set_data(json.dumps(response.response))
    return r
```

Route Groups

New in version 2.1.2.

Route Groups are a way of grouping routes together into a single class, where they can all access the members of the class. This is easier than having global shared state, and easily allows having "route" templates.

Creating a Route Group

All route groups descend from *RouteGroup*, or use *RouteGroupType* as the metaclass. The former uses the latter as its metaclass, which is a shorter version.

```
from kyoukai.routegroup import RouteGroup, RouteGroupType

# form 1, easiest form
class MyRouteGroup(RouteGroup):
    ...

# form 2, explicit metaclass
class MyRouteGroup(metaclass=RouteGroupType):
    ...
```

Note: By default, route groups have no magic `__init__`. You are free to implement this in whatever way you like, including passing parameters to it.

Adding Routes

To make your route group useful, you need to add some **routes** to it. The RouteGroup module includes a special decorator that marks a route function as a new *Route* during instance creation, `route()`.

This method takes the same arguments as the regular `route` decorator; the only difference is that it returns the original function in the class body rather than returning a new Route object. Instead, certain attributes are set on the new function that are picked up during scanning, such as `in_group`.

```
from kyoukai.routegroup import RouteGroup, route

class MyRouteGroup(RouteGroup):
    @route("/heck", methods=("GET", "POST"))
    async def heck_em_up(self, ctx: HTTPRequestContext):
        return "get hecked"
```

This will register `heck_em_up` as a route on the new route group.

`kyoukai.routegroup.`**`route`**(*url*, *methods=('GET',)*, ***kwargs*)

A companion function to the RouteGroup class. This follows *Blueprint.route()* in terms of arguments, and marks a function as a route inside the class.

This will return the original function, with some attributes attached:

- `in_group`: Marks the function as in the route group.
- `rg_delegate`: Internal. The type of function inside the group this is.
- `route_kwargs`: Keyword arguments to provide to `wrap_route`.
- `route_url`: The routing URL to provide to `add_route`.
- `route_methods`: The methods for the route.
- `route_hooks`: A defaultdict of route-specific hooks.

Additionally, the following methods are added.

- `hook`: A decorator that adds a hook of type `type_`.
- `before_request`: A decorator that adds a `pre` hook.
- `after_request`: A decorator that adds a `post` hook.

New in version 2.1.1.

Changed in version 2.1.3: Added the ability to add route-specific hooks.

Parameters

- **`url`** (`str`) – The routing URL of the route.
- **`methods`** (`Iterable`[`str`]) – An iterable of methods for the route.

Error Handlers

New in version 2.1.3.

Route groups can also have group-specific error handlers, using *errorhandler()*.

```
@errorhandler(500)
async def handle_errors(self, ctx, exc):
    ...
```

kyoukai.routegroup.**errorhandler**(*code*)
A companion function to the RouteGroup class. This follows *Blueprint.errorhandler()* in terms of arguments.

Parameters **code** (int) – The code for the error handler.

Request Hooks

New in version 2.1.3.

Route groups can have both Blueprint-specific error handlers, and route-specific error handlers, using the helper functions.

For Blueprint-specific, you can use *hook()* (or, better, aliases *before_request()* and *after_request()*).

```
@before_request
async def before_req(self, ctx):
    ...
```

Adding route-specific hooks is possible by calling `@route.hook` on the newly wrapped function. This is achieved by setting a special decorator function on the function object modified by the route decorator.

```
@heck_em_up.before_req
async def whatever(self, ctx):
    ...
```

kyoukai.routegroup.**hook**(*type_*)
Marks a function as a hook.

Parameters **type** (str) – The type of hook to mark.

kyoukai.routegroup.**before_request**(*func*)
Helper decorator to mark a function as a pre-request hook.

kyoukai.routegroup.**after_request**(*func*)
Helper decorator to mark a function as a post-request hook.

@func.**hook**(*type_: str*)
Marks a function as a route-specific hook.

Parameters **type** – The type of hook to add.

@func.**before_request**
Marks a function as a before-request hook.

@func.**after_request**
Marks a function as an after-request hook.

Registering the Group

Adding the group to your app is as simple as instantiating the group and calling *Blueprint.add_route_group()* with the instance.

```
rg = MyRouteGroup()
app.root.add_route_group(rg)
```

Of course, an alias for this exists on *Kyoukai* which redirects to the root blueprint.

Blueprint.**add_route_group**(*group*)
Adds a route group to the current Blueprint.

Parameters **group** (*RouteGroup*) – The *RouteGroup* to add.

Customizing the Blueprint

Route groups work by using an underlying Blueprint that is populated with all the routes from the class body during instantiation. The Blueprint can be customized by passing arguments in the class definition to the metaclass, which are stored and later used to create the new Blueprint object.

```
class MyRouteGroup(RouteGroup, prefix="/api/v1")
    ...
```

To get the blueprint object from a RouteGroup instance, you can use `get_rg_bp()`.

kyoukai.routegroup.**get_rg_bp**(*group*)
Gets the *Blueprint* created from a *RouteGroup*.

Host Matching

New in version 2.1.3.

Kyoukai comes with built-in support for Werkzeug host matching:

```
# enable host matching in the tree
# this needs to be set on the root blueprint for the blueprint tree
app = Kyoukai("my_website", host_matching=True)

# set a host on a sub-blueprint
# all sub-blueprints of `bp` will now use the host `api.myname.me`
bp = Blueprint("api", host="api.myname.me")
```

As shown above, host matching is easy to enable, requiring only two changes.

- `host_matching` MUST be set on the root Blueprint (passed here via the app) - this will enable host matching when building the final map.
- `host` is passed into the Blueprint constructor, which specifies the host that will be matched for each route in this Blueprint.

In the example above, all routes registered to `bp` will only match if the Host header is `api.myname.me`. However, all routes registered to other Blueprints will match on **any** hosts.

Relation to the Tree

Children Blueprints will copy their host from the parent, unless overridden. So, for example:

```
# only host match ``myname.me``
app = Kyoukai("my_website", host="myname.me")

# bp1 will only obey requests from `myname.me`
bp1 = Blueprint("something")
app.register_blueprint(bp1)
```

```
# bp2 will only obey requests from `something.myname.me`, overriding the global host
↪match
bp2 = Blueprint("something else", host="something.myname.me")
app.register_blueprint(bp2)

# bp3 however will inherit its parents host matching (bp2)
bp3 = Blueprint("something finally")
bp2.add_child(bp3)
```

HTTPS Support

New in version 2.1.

Kyoukai's built in web server comes with native TLS support with secure defaults. Enabling it is as simple as creating a new block in the config file:

```
# The SSL configuration for the built-in webserver
ssl:
    # Is SSL enabled?
    # If this is False, the certfile and keyfile will not be loaded.
    enabled: true

    # The public key certificate for the webserver to use.
    ssl_certfile: server.crt

    # The private keyfile for the webserver to use.
    ssl_keyfile: server.key
```

HTTPS will then automatically be enabled for this connection.

HTTP and HTTPS multiplexing

This is **not** currently supported.

HTTP/2 Support

New in version 2.1.0.

Kyoukai comes with built in support for HTTP/2, thanks to to the H2 library.

Enabling HTTP2 requires:

- TLS/SSL to be enabled
- `h2` to be installed
- The `http2` key in the config to be True, or manual switching to be enabled

Automatic switching

Kyoukai supports automatically upgrading to HTTP/2 via ALPN/NPN protocols (the default for making new connections over TLS) or with plain old h2c.

To enable **automatic upgrade**, add the `http2` key to your config file, under the `kyoukai` component, like so:

```
# Enables automatic HTTP/2 connection switching.
# This will switch to the HTTP/2 protocol parser when a connection is created.
http2: true
```

Now, when connecting over TLS (or HTTP/1.1 with h2c) the connection will be automatically upgraded to a HTTP/2 connection.

Manual switching

It is possible to enforce HTTP/2 only, or otherwise manual switching, with the usage of *H2KyoukaiProtocol*.

To switch to this component, change `KyoukaiComponent` to `H2KyoukaiComponent` in your application component container like so:

```
self.add_component('kyoukai', H2KyoukaiComponent, ip="127.0.0.1", port=4444,
                   app=app)
```

API Ref

class kyoukai.backends.http2.**H2KyoukaiComponent**(*app*, *ssl_keyfile*, *ssl_certfile*, ***, *ip='127.0.0.1'*, *port=4444*)

Bases: *kyoukai.asphalt.KyoukaiBaseComponent*

A component subclass that creates H2KyoukaiProtocol instances.

Creates a new HTTP/2 SSL-based context.

This will use the HTTP/2 protocol, disabling HTTP/1.1 support for this port. It is possible to run two servers side-by-side, one HTTP/2 and one HTTP/1.1, if you run them on different ports.

get_server_name()

Returns The server name of this app.

class kyoukai.backends.http2.**H2KyoukaiProtocol**(*component*, *parent_context*)

Bases: asyncio.protocols.Protocol

The base protocol for Kyoukai, using H2.

raw_write(*data*)

Writes to the underlying transport.

connection_made(*transport*)

Called when a connection is made.

Parameters **transport** (WriteTransport) – The transport made by the connection.

data_received(*data*)

Called when data is received from the underlying socket.

_processing_done(*environ*, *stream_id*)

Callback for when processing is done on a request.

coroutine sending_loop(*self*, *stream_id*)

This loop continues sending data to the client as it comes off of the queue.

request_received(*event*)

Called when a request has been received.

window_opened(*event*)
Called when a control flow window has opened again.

receive_data(*event*)
Called when a request has data that has been received.

stream_complete(*event*)
Called when a stream is complete.

This will invoke Kyoukai, which will handle the request.

close(*error_code=0*)
Called to terminate the connection for some reason.

This will close the underlying transport.

eof_received()
Called when the other end calls write_eof() or equivalent.

If this returns a false value (including None), the transport will close itself. If it returns a true value, closing the transport is up to the protocol.

pause_writing()
Called when the transport's buffer goes over the high-water mark.

Pause and resume calls are paired – pause_writing() is called once when the buffer goes strictly over the high-water mark (even if subsequent writes increases the buffer size even more), and eventually resume_writing() is called once when the buffer size reaches the low-water mark.

Note that if the buffer size equals the high-water mark, pause_writing() is not called – it must go strictly over. Conversely, resume_writing() is called when the buffer size is equal or lower than the low-water mark. These end conditions are important to ensure that things go as expected when either mark is zero.

NOTE: This is the only Protocol callback that is not called through EventLoop.call_soon() – if it were, it would have no effect when it's most needed (when the app keeps writing without yielding until pause_writing() is called).

resume_writing()
Called when the transport's buffer drains below the low-water mark.

See pause_writing() for details.

class kyoukai.backends.http2.**H2State**(*headers*, *stream_id*, *protocol*)
Bases: object

A temporary class that is used to store request data for a HTTP/2 connection.

This is also passed to the Werkzeug request to emit data.

insert_data(*data*)
Writes data from the stream into the body.

coroutine read_async(*self*, *to_end=True*)
There's no good way to do this - WSGI isn't async, after all.

However, you can use *read_async* on the Werkzeug request (which we subclass) to wait until the request has finished streaming.

Parameters **to_end** – If to_end is specified, then read until the end of the request. Otherwise, it will read one data chunk.

read(*size=-1*)
Reads data from the request until it's all done.

Parameters **size** (int) – The maximum amount of data to receive.

Return type bytes

get_chunk()
Gets a chunk of data from the queue.

Return type bytes

start_response(*status*, *headers*, *exc_info=None*)
The start_response callable that is plugged into a Werkzeug response.

get_response_headers()
Called by the protocol once the Response is writable to submit the request to the HTTP/2 state machine.

CHAPTER 4

Automatically generated API documentation

This API documentation is automatically generated by the Sphinx `autosummary` module.

Kyoukai Autodoc

This is **automatically generated** API documentation for the *kyoukai* module. Kyoukai is an async web framework for Python 3.5 and above.

app	The core application.
backends	Various backends that interface with the Kyoukai application.
asphalt	Asphalt wrappers for Kyoukai.
blueprint	A blueprint is a container - a collection of routes.
route	Routes are wrapped function objects that are called upon a HTTP request.
routegroup	Route groups are classes that allow you to group a set of routes together.
testing	Testing helpers for Kyoukai.
util	Misc utilities for usage inside the framework.

kyoukai.app

The core application.

Functions

`run_application`(component, *[, ...])	Configure logging and start the given root component in the default asyncio event loop.

Classes

`Blueprint`(name[, parent, prefix, ...])	A Blueprint class contains a Map of URL rules, which is checked and ran for every
`Context`([parent, default_timeout])	Contexts give request handlers and callbacks access to resources.
`HTTPRequestContext`(parent, request)	The context subclass passed to all requests within Kyoukai.
`Kyoukai`(application_name, *[, server_name])	The Kyoukai type is the core of the Kyoukai framework, and the core of your web application based upon the Kyoukai framework.
`Request`(environ[, populate_request, shallow])	Full featured request object implementing the following mixins:
`Response`([response, status, headers, ...])	Full featured response object implementing the following mixins:

Exceptions

`HTTPException`([description, response])	Baseclass for all HTTP exceptions.
`InternalServerError`([description, response])	*500 Internal Server Error*
`MethodNotAllowed`([valid_methods, description])	*405 Method Not Allowed*
`NotFound`([description, response])	*404 Not Found*
`RequestRedirect`(new_url)	Raise if the map requests a redirect.

class `kyoukai.app.`**`Kyoukai`**(*application_name*, *, *server_name=None*, ***kwargs*)

Bases: `object`

The Kyoukai type is the core of the Kyoukai framework, and the core of your web application based upon the Kyoukai framework. It acts as a central router and request processor that takes in requests from the protocols and returns responses.

The application name is currently unused, but it is good practice to set it correctly anyway in case it is used in future editions of Kyoukai.

You normally create an application instance inside your component file, like so:

```
from kyoukai.app import Kyoukai

... # setup code

kyk = Kyoukai("my_app")
kyk.register_blueprint(whatever)

... # other setup

class MyContainer(ContainerComponent):
    async def start(self, ctx):
        self.add_component('kyoukai', KyoukaiComponent, ip="127.0.0.1", port=4444,
                           app="app:app")
```

Of course, you can also embed Kyoukai inside another app, by awaiting *`Kyoukai.start()`*.

Parameters

- **application_name** (str) – The name of the application that is being created. This is passed to the *Blueprint* being created as the root blueprint.
- **server_name** (Optional[str]) – Keyword-only. The SERVER_NAME to use inside the fake WSGI environment created for url_for, if applicable.
- **host_matching** – Should host matching be enabled? This will be implicitly True if host is not None.
- **host** – The host used for host matching, to be passed to the root Blueprint. By default, no host is used, so all hosts are matched on the root Blueprint.
- **application_root** – Keyword-only. The APPLICATION_ROOT to use inside the fake WSGI environment created for url_for, if applicable.
- **loop** – Keyword-only. The asyncio event loop to use for this app. If no loop is specified it, will be automatically fetched using asyncio.get_event_loop().
- **request_class** – Keyword-only. The custom request class to instantiate requests with.
- **response_class** – Keyword-only. The custom response class to instantiate responses with.

request_class
The class of request to spawn every request. This should be a subclass of werkzeug.wrappers.Request. You can override this by passing request_class as a keyword argument to the app.

alias of Request

response_class
The class of response to wrap automatically. This should be a subclass of werkzeug.wrappers.Response. You can override this by passing response_class as a keyword argument to the app.

alias of Response

root

Return type *Blueprint*

Returns The root Blueprint for the routing tree.

register_blueprint(*child*)
Registers a child blueprint to this app's root Blueprint.

This will set up the Blueprint tree, as well as setting up the routing table when finalized.

Parameters **child** (*Blueprint*) – The child Blueprint to add. This must be an instance of *Blueprint*.

finalize()
Finalizes the app and blueprints.

This will calculate the current werkzeug.routing.Map which is required for routing to work.

log_route(*request*, *code*)
Logs a route invocation.

Parameters

- **request** (Request) – The request produced.
- **code** (int) – The response code of the route.

coroutine handle_httpexception(*self*, *ctx*, *exception*, *environ=None*)
Handle a HTTP Exception.

Parameters

- **ctx** (*HTTPRequestContext*) – The context of the request.
- **exception** (HTTPException) – The HTTPException
- **environ** (Optional[dict]) – The fake WSGI environment.

Return type Response

Returns A werkzeug.wrappers.Response that handles this response.

coroutine process_request (*self*, *request*, *parent_context*)
Processes a Request and returns a Response object.

This is the main processing method of Kyoukai, and is meant to be used by one of the HTTP server backends, and not by client code.

Parameters

- **request** (Request) – The werkzeug.wrappers.Request object to process. A new *HTTPRequestContext* will be provided to wrap this request inside of to client code.
- **parent_context** (Context) – The asphalt.core.Context that is the parent context for this particular app. It will be used as the parent for the HTTPRequestContext.

Return type Response

Returns A werkzeug.wrappers.Response object that can be written to the client as a response.

coroutine start (*self*, *ip='127.0.0.1'*, *port=4444*, ***, *component=None*, *base_context=None*)
Runs the Kyoukai component asynchronously.

This will bypass Asphalt's default runner, and allow you to run your app easily inside something else, for example.

Parameters

- **ip** (str) – The IP of the built-in server.
- **port** (int) – The port of the built-in server.
- **component** – The component to start the app with. This should be an instance of *kyoukai.asphalt.KyoukaiComponent*.
- **base_context** (Optional[Context]) – The base context that the HTTPRequestContext should be started with.

run (*ip='127.0.0.1'*, *port=4444*, ***, *component=None*)
Runs the Kyoukai server from within your code.

This is not normally invoked - instead Asphalt should invoke the Kyoukai component. However, this is here for convenience.

kyoukai.backends

Various backends that interface with the Kyoukai application.

*httptools_*	A high-performance HTTP/1.1 backend for the Kyoukai webserver using httptools.
http2	A HTTP/2 interface to Kyoukai.

kyoukai.backends.httptools_

A high-performance HTTP/1.1 backend for the Kyoukai webserver using httptools.

Functions

`get_formatted_response`(response, environment)	Transform a Werkzeug response into a HTTP response that can be sent back down the wire.
`to_wsgi_environment`(headers, method, path, ...)	Produces a new WSGI environment from a set of data that is passed in.

Classes

`BytesIO`	Buffered I/O implementation using an in-memory bytes buffer.
`Context`([parent, default_timeout])	Contexts give request handlers and callbacks access to resources.
`H2KyoukaiProtocol`(component, parent_context)	The base protocol for Kyoukai, using H2.
`KyoukaiProtocol`(component, parent_context, ...)	The base protocol for Kyoukai using httptools for a HTTP/1.0 or HTTP/1.1 interface.
`Request`(environ[, populate_request, shallow])	Full featured request object implementing the following mixins:
`Response`([response, status, headers, ...])	Full featured response object implementing the following mixins:

Exceptions

`BadRequest`([description, response])	*400 Bad Request*
`InternalServerError`([description, response])	*500 Internal Server Error*
`MethodNotAllowed`([valid_methods, description])	*405 Method Not Allowed*

class `kyoukai.backends.httptools_.`**`KyoukaiProtocol`**(*component*, *parent_context*, *server_ip*, *server_port*)

Bases: `asyncio.protocols.Protocol`

The base protocol for Kyoukai using httptools for a HTTP/1.0 or HTTP/1.1 interface.

Parameters

- **`component`** – The *`kyoukai.asphalt.KyoukaiComponent`* associated with this request.
- **`parent_context`** (`Context`) – The parent context for this request. A new HTTPRequestContext will be derived from this.

`replace`(*other*, **args*, ***kwargs*)

Replaces our type with the other.

Return type `type`

`on_message_begin`()

Called when a message begins.

on_header(*name*, *value*)
Called when a header has been received.

Parameters

- **name** (bytes) – The name of the header.
- **value** (bytes) – The value of the header.

on_headers_complete()
Called when the headers have been completely sent.

on_body(*body*)
Called when part of the body has been received.

Parameters **body** (bytes) – The body text.

on_url(*url*)
Called when a URL is received from the client.

on_message_complete()
Called when a message is complete. This creates the worker task which will begin processing the request.

connection_made(*transport*)
Called when a connection is made via asyncio.

Parameters **transport** (WriteTransport) – The transport this is using.

data_received(*data*)
Called when data is received into the connection.

handle_parser_exception(*exc*)
Handles an exception when parsing.

This will not call into the app (hence why it is a normal function, and not a coroutine). It will also close the connection when it's done.

Parameters **exc** (Exception) – The exception to handle.

coroutine **_wait**(*self*)
The main core of the protocol.

This constructs a new Werkzeug request from the headers.

write_response(*response*, *fake_environ*)
Writes a Werkzeug response to the transport.

write(*data*)
Writes data to the socket.

raw_write(*data*)
Writes data to the transport.

_raw_write(*data*)
Does a raw write to the underlying transport, if we can.

Parameters **data** (bytes) – The data to write.

eof_received()
Called when the other end calls write_eof() or equivalent.

If this returns a false value (including None), the transport will close itself. If it returns a true value, closing the transport is up to the protocol.

pause_writing()
Called when the transport's buffer goes over the high-water mark.

Pause and resume calls are paired – pause_writing() is called once when the buffer goes strictly over the high-water mark (even if subsequent writes increases the buffer size even more), and eventually resume_writing() is called once when the buffer size reaches the low-water mark.

Note that if the buffer size equals the high-water mark, pause_writing() is not called – it must go strictly over. Conversely, resume_writing() is called when the buffer size is equal or lower than the low-water mark. These end conditions are important to ensure that things go as expected when either mark is zero.

NOTE: This is the only Protocol callback that is not called through EventLoop.call_soon() – if it were, it would have no effect when it's most needed (when the app keeps writing without yielding until pause_writing() is called).

resume_writing()
Called when the transport's buffer drains below the low-water mark.

See pause_writing() for details.

kyoukai.backends.http2

A HTTP/2 interface to Kyoukai.

This uses https://python-hyper.org/projects/h2/en/stable/asyncio-example.html as a reference and a base. Massive thanks to the authors of this page.

This server has some notable pitfalls:

- It ignores any priority data that is sent by the client.
- It is not paticularly fast (unbenchmarked, but it can be assumed to be slower than the httptools backend.)
- It does not fully implement all events.

Additionally, this server is **untested** - it can and probably will fail horribly in production. Use with caution :)

Functions

create_wsgi_environment(r)	Creates a new WSGI environment from the RequestData provided.
get_header(headers, name)	Gets a header from the list of headers, or None if it doesn't exist.
urlsplit(url[, scheme, allow_fragments])	Parse a URL into 5 components: <scheme>://<netloc>/<path>?<query>#<fragment> Return a 5-tuple: (scheme, netloc, path, query, fragment).

Classes

Context([parent, default_timeout])	Contexts give request handlers and callbacks access to resources.
DataReceived()	The DataReceived event is fired whenever data is received on a stream from the remote peer.
ErrorCodes	All known HTTP/2 error codes.

Continued on next page

Table 4.10 – continued from previous page

H2Connection([config])	A low-level HTTP/2 connection object.
H2KyoukaiComponent(app, ssl_keyfile, ...[, ...])	A component subclass that creates H2KyoukaiProtocol instances.
H2KyoukaiProtocol(component, parent_context)	The base protocol for Kyoukai, using H2.
H2State(headers, stream_id, protocol)	A temporary class that is used to store request data for a HTTP/2 connection.
KyoukaiBaseComponent(app[, ip, port])	The base class for any component used by Kyoukai.
MultiDict([mapping])	A MultiDict is a dictionary subclass customized to deal with multiple values for the same key which is for example used by the parsing functions in the wrappers.
Request(environ[, populate_request, shallow])	Full featured request object implementing the following mixins:
RequestReceived()	The RequestReceived event is fired whenever request headers are received.
Response([response, status, headers, ...])	Full featured response object implementing the following mixins:
StreamEnded()	The StreamEnded event is fired whenever a stream is ended by a remote party.
StreamReset()	The StreamReset event is fired in two situations.
WindowUpdated()	The WindowUpdated event is fired whenever a flow control window changes size.
partial	partial(func, *args, **keywords) - new function with partial application

Exceptions

ProtocolError	An action was attempted in violation of the HTTP/2 protocol.

kyoukai.backends.http2.**get_header**(*headers*, *name*)
Gets a header from the list of headers, or None if it doesn't exist.

Return type str

kyoukai.backends.http2.**create_wsgi_environment**(*r*)
Creates a new WSGI environment from the RequestData provided.

Return type MultiDict

class kyoukai.backends.http2.**H2State**(*headers*, *stream_id*, *protocol*)
Bases: object

A temporary class that is used to store request data for a HTTP/2 connection.

This is also passed to the Werkzeug request to emit data.

insert_data(*data*)
Writes data from the stream into the body.

coroutine read_async(*self*, *to_end=True*)
There's no good way to do this - WSGI isn't async, after all.

However, you can use *read_async* on the Werkzeug request (which we subclass) to wait until the request has finished streaming.

Parameters **to_end** – If `to_end` is specified, then read until the end of the request. Otherwise, it will read one data chunk.

read(*size=-1*)
Reads data from the request until it's all done.

Parameters **size** (`int`) – The maximum amount of data to receive.

Return type `bytes`

get_chunk()
Gets a chunk of data from the queue.

Return type `bytes`

start_response(*status*, *headers*, *exc_info=None*)
The `start_response` callable that is plugged into a Werkzeug response.

get_response_headers()
Called by the protocol once the Response is writable to submit the request to the HTTP/2 state machine.

class `kyoukai.backends.http2.`**H2KyoukaiComponent**(*app*, *ssl_keyfile*, *ssl_certfile*, ***, *ip='127.0.0.1'*, *port=4444*)
Bases: *kyoukai.asphalt.KyoukaiBaseComponent*

A component subclass that creates H2KyoukaiProtocol instances.

Creates a new HTTP/2 SSL-based context.

This will use the HTTP/2 protocol, disabling HTTP/1.1 support for this port. It is possible to run two servers side-by-side, one HTTP/2 and one HTTP/1.1, if you run them on different ports.

get_server_name()

Returns The server name of this app.

class `kyoukai.backends.http2.`**H2KyoukaiProtocol**(*component*, *parent_context*)
Bases: `asyncio.protocols.Protocol`

The base protocol for Kyoukai, using H2.

raw_write(*data*)
Writes to the underlying transport.

connection_made(*transport*)
Called when a connection is made.

Parameters **transport** (`WriteTransport`) – The transport made by the connection.

data_received(*data*)
Called when data is received from the underlying socket.

_processing_done(*environ*, *stream_id*)
Callback for when processing is done on a request.

coroutine sending_loop(*self*, *stream_id*)
This loop continues sending data to the client as it comes off of the queue.

request_received(*event*)
Called when a request has been received.

window_opened(*event*)
Called when a control flow window has opened again.

receive_data(*event*)
Called when a request has data that has been received.

stream_complete(*event*)
Called when a stream is complete.

This will invoke Kyoukai, which will handle the request.

close(*error_code=0*)
Called to terminate the connection for some reason.

This will close the underlying transport.

eof_received()
Called when the other end calls write_eof() or equivalent.

If this returns a false value (including None), the transport will close itself. If it returns a true value, closing the transport is up to the protocol.

pause_writing()
Called when the transport's buffer goes over the high-water mark.

Pause and resume calls are paired – pause_writing() is called once when the buffer goes strictly over the high-water mark (even if subsequent writes increases the buffer size even more), and eventually resume_writing() is called once when the buffer size reaches the low-water mark.

Note that if the buffer size equals the high-water mark, pause_writing() is not called – it must go strictly over. Conversely, resume_writing() is called when the buffer size is equal or lower than the low-water mark. These end conditions are important to ensure that things go as expected when either mark is zero.

NOTE: This is the only Protocol callback that is not called through EventLoop.call_soon() – if it were, it would have no effect when it's most needed (when the app keeps writing without yielding until pause_writing() is called).

resume_writing()
Called when the transport's buffer drains below the low-water mark.

See pause_writing() for details.

kyoukai.asphalt

Asphalt wrappers for Kyoukai.

Functions

`resolve_reference`(ref)	Return the object pointed to by `ref`.

Classes

`Blueprint`(name[, parent, prefix, ...])	A Blueprint class contains a Map of URL rules, which is checked and ran for every
`Component`	This is the base class for all Asphalt components.
`ConnectionLostEvent`(source, topic, *, protocol)	Dispatched when a connection is lost from the server.
`ConnectionMadeEvent`(source, topic, *, protocol)	Dispatched when a connection is made to the server.
`Context`([parent, default_timeout])	Contexts give request handlers and callbacks access to resources.
`CtxEvent`(source, topic, *, ctx)	

Continued on next page

Table 4.13 – continued from previous page

Event(source, topic)	The base class for all events.
HTTPRequestContext(parent, request)	The context subclass passed to all requests within Kyoukai.
KyoukaiBaseComponent(app[, ip, port])	The base class for any component used by Kyoukai.
KyoukaiComponent(app[, ip, port])	A component for Kyoukai.
Request(environ[, populate_request, shallow])	Full featured request object implementing the following mixins:
Response([response, status, headers, ...])	Full featured response object implementing the following mixins:
Route(function[, reverse_hooks, ...])	A route object is a wrapped function.
RouteInvokedEvent(source, topic, *, ctx)	Dispatched when a route is invoked.
RouteMatchedEvent(source, topic, *, ctx)	Dispatched when a route is matched.
RouteReturnedEvent(source, topic, *, ctx, result)	Dispatched after a route has returned.
Rule(string[, defaults, subdomain, methods, ...])	A Rule represents one URL pattern.
Signal(event_class, *[, source, topic])	Declaration of a signal that can be used to dispatch events.
partial	partial(func, *args, **keywords) - new function with partial application

class kyoukai.asphalt.**ConnectionMadeEvent**(*source*, *topic*, *, *protocol*)
Bases: asphalt.core.event.Event

Dispatched when a connection is made to the server.

This does NOT fire when using WSGI workers.

This has the protocol as the protocol attribute.

utc_timestamp
Return a timezone aware datetime corresponding to the time variable, using the UTC timezone.

Return type datetime

class kyoukai.asphalt.**ConnectionLostEvent**(*source*, *topic*, *, *protocol*)
Bases: *kyoukai.asphalt.ConnectionMadeEvent*

Dispatched when a connection is lost from the server.

This does NOT fire when using WSGI workers.

This has the protocol as the protocol attribute.

utc_timestamp
Return a timezone aware datetime corresponding to the time variable, using the UTC timezone.

Return type datetime

class kyoukai.asphalt.**RouteMatchedEvent**(*source*, *topic*, *, *ctx*)
Bases: kyoukai.asphalt.CtxEvent

Dispatched when a route is matched.

This has the context as the ctx attribute, and the route can be accessed via ctx.route.

utc_timestamp
Return a timezone aware datetime corresponding to the time variable, using the UTC timezone.

Return type datetime

class kyoukai.asphalt.**RouteInvokedEvent**(*source*, *topic*, *, *ctx*)
Bases: kyoukai.asphalt.CtxEvent

Dispatched when a route is invoked.

This has the context as the `ctx` attribute.

utc_timestamp
Return a timezone aware `datetime` corresponding to the `time` variable, using the UTC timezone.

Return type `datetime`

class `kyoukai.asphalt.`**RouteReturnedEvent**(*source*, *topic*, *, *ctx*, *result*)
Bases: `kyoukai.asphalt.CtxEvent`

Dispatched after a route has returned.

This has the context as the `ctx` attribute and the response as the `result` attribute.

utc_timestamp
Return a timezone aware `datetime` corresponding to the `time` variable, using the UTC timezone.

Return type `datetime`

class `kyoukai.asphalt.`**KyoukaiBaseComponent**(*app*, *ip='127.0.0.1'*, *port=4444*, ***cfg*)
Bases: `asphalt.core.component.Component`

The base class for any component used by Kyoukai.

This one does not create a Server instance; it should be used when you are using a different HTTP server backend.

app = None
The application object for a this component.

ip = None
The IP address to boot the server on.

port = None
The port to boot the server on.

cfg = None
The config file to use.

server = None
The `asyncio.Server` instance that is serving us today.

base_context = None
The base context for this server.

backend = None
The backend to use for the HTTP server.

coroutine start(*self*, *ctx*)
Overridden in subclasses to spawn a new server.

get_server_name()

Returns The server name of this app.

get_protocol(*ctx*, *serv_info*)
Gets the protocol to use for this webserver.

class `kyoukai.asphalt.`**KyoukaiComponent**(*app*, *ip='127.0.0.1'*, *port=4444*, ***cfg*)
Bases: *kyoukai.asphalt.KyoukaiBaseComponent*

A component for Kyoukai.

This includes the built-in HTTP server.

Creates a new component.

Parameters

- **app** – The application object to use. This can either be the real application object, or a string that resolves to a reference for the real application object.
- **ip** (str) – If using the built-in HTTP server, the IP to bind to.
- **port** (int) – If using the built-in HTTP server, the port to bind to.
- **cfg** – Additional configuration.

get_server_name()

Returns The server name of this app.

coroutine start(*self*, *ctx*)
Starts the webserver if required.

Parameters **ctx** (Context) – The base context.

get_protocol(*ctx*, *serv_info*)
Gets the protocol to use for this webserver.

class kyoukai.asphalt.**HTTPRequestContext**(*parent*, *request*)
Bases: asphalt.core.context.Context

The context subclass passed to all requests within Kyoukai.

app = None
The *Kyoukai* object this request is handling.

request = None
The werkzeug.wrappers.Request object this request is handling.

route = None
The *Route* object this request is for.

bp = None
The *Blueprint* object this request is for.

rule = None
The werkzeug.routing.Rule object associated with this request.

environ = None
The WSGI environment for this request.

proto = None
The asyncio.Protocol protocol handling this connection.

get_resources(*type=None*, ***, *include_parents=True*)
Return the currently published resources specific to one type or all types.

Parameters

- **type** (Union[str, type, None]) – type of the resources to return, or None to return all resources
- **include_parents** (bool) – include the resources from parent contexts

Return type Sequence[Resource]

parent
Return the parent of this context or None if there is no parent context.

Return type Optional[Context]

publish_lazy_resource(*creator*, *types*, *alias='default'*, *context_attr=None*)
Publish a "lazy" or "contextual" resource and dispatch a `resource_published` event.

Instead of a concrete resource value, you supply a creator callable which is called with a context object as its argument when the resource is being requested either via *request_resource()* or by context attribute access. The return value of the creator callable will be cached so the creator will only be called once per context instance.

If the creator callable is a coroutine function or returns an awaitable, it is resolved before storing the resource value and returning it to the requester. Note that this will **NOT** work when a context attribute has been specified for the resource.

Parameters

- **creator** (Callable[[Context], Any]) – a callable taking a context instance as argument
- **types** (Union[type, Iterable[Union[str, type]]]) – type(s) to register the resource as
- **alias** (str) – name of this resource (unique among all its registered types)
- **context_attr** (Optional[str]) – name of the context attribute this resource will be accessible as

Return type Resource

Returns the resource handle

Raises **asphalt.core.context.ResourceConflict** – if there is an existing resource creator for the given types or context variable

publish_resource(*value*, *alias='default'*, *context_attr=None*, ***, *types=()*)
Publish a resource and dispatch a `resource_published` event.

Parameters

- **value** – the actual resource value
- **alias** (str) – name of this resource (unique among all its registered types)
- **context_attr** (Optional[str]) – name of the context attribute this resource will be accessible as
- **types** (Union[type, Iterable[Union[str, type]]]) – type(s) to register the resource as (omit to use the type of `value`)

Return type Resource

Returns the resource handle

Raises **asphalt.core.context.ResourceConflict** – if the resource conflicts with an existing one in any way

remove_resource(*resource*)
Remove the given resource from the collection and dispatch a `resource_removed` event.

Parameters **resource** (Resource) – the resource to be removed

Raises **LookupError** – the given resource was not in the collection

coroutine **request_resource**(*self*, *type*, *alias='default'*, ***, *timeout=None*)
Request a resource matching the given type and alias.

If no such resource was found, this method will wait `timeout` seconds for it to become available. The timeout does not apply to resolving awaitables created by lazy resource creators.

Parameters

- **type** (Union[str, type]) – type of the requested resource
- **alias** (str) – alias of the requested resource
- **timeout** (Union[int, float, None]) – the timeout (in seconds; omit to use the context's default timeout)

Returns the value contained by the requested resource (**NOT** a Resource instance)

Raises **asphalt.core.context.ResourceNotFound** – if the requested resource does not become available in the allotted time

url_for(*endpoint*, ***, *method*, ***kwargs*)
A context-local version of url_for.

For more information, see the documentation on *url_for()*.

kyoukai.blueprint

A blueprint is a container - a collection of routes.

Kyoukai uses Blueprints to create a routing tree - a tree of blueprints that are used to collect routes together and match routes easily.

Functions

get_rg_bp(group)	Gets the *Blueprint* created from a *RouteGroup*.

Classes

Blueprint(name[, parent, prefix, ...])	A Blueprint class contains a Map of URL rules, which is checked and ran for every
Map([rules, default_subdomain, charset, ...])	The map class stores all the URL rules and some configuration parameters.
Response([response, status, headers, ...])	Full featured response object implementing the following mixins:
Route(function[, reverse_hooks, ...])	A route object is a wrapped function.
RouteGroup	A route group is a class that contains multiple methods that are decorated with the route decorator.
Rule(string[, defaults, subdomain, methods, ...])	A Rule represents one URL pattern.

Exceptions

HTTPException([description, response])	Baseclass for all HTTP exceptions.

class kyoukai.blueprint.**Blueprint**(*name*, *parent=None*, *prefix=''*, ***, *host_matching=False*, *host=None*)

Bases: object

A Blueprint class contains a Map of URL rules, which is checked and ran for every

Parameters

- **name** (str) – The name of this Blueprint. This is used when generating endpoints in the finalize stage.
- **parent** (Optional[*Blueprint*]) – The parent of this Blueprint. Parent blueprints will gather the routes of their children, and return a giant werkzeug.routing.Map object that contains all of the route maps in the children
- **prefix** (str) – The prefix to be added to the start of every route name. This is inherited from parents - the parent prefix will also be added to the start of every route.
- **host_matching** (bool) – Should host matching be enabled? This is implicitly True if host is non-None.
- **host** (Optional[str]) – The host of the Blueprint. Used for custom subdomain routing. If this is None, then this Blueprint will be used for all hosts.

name = None
The name of this Blueprint.

finalized = None
If this Blueprint is finalized or not. Finalization of a blueprint means gathering all of the Maps, and compiling a routing table which stores the endpoints.

routes = None
The list of routes. This is used in finalization.

errorhandlers = None
The error handler dictionary.

parent

Return type *Blueprint*

Returns The parent Blueprint of this blueprint.

prefix

Return type str

Returns The combined prefix of this Blueprint.

tree_routes

Return type Generator[*Route*, None, None]

Returns A generator that yields all routes from the tree, from parent to children.

host

Return type str

Returns The host for this Blueprint, or the host of any parent Blueprint.

traverse_tree()
Traverses the tree for children Blueprints.

Return type Generator[*Blueprint*, None, None]

finalize(***map_options*)
Called on the root Blueprint when all Blueprints have been registered and the app is starting.

This will automatically build a werkzeug.routing.Map of werkzeug.routing.Rule objects for each Blueprint.

Note: Calling this on sub-blueprints will have no effect, apart from generating a Map. It is recommended to only call this on the root Blueprint.

Parameters **map_options** – The options to pass to the created Map.

Return type Map

Returns The werkzeug.routing.Map created from the routing tree.

add_child(*blueprint*)
Adds a Blueprint as a child of this one. This is automatically called when using another Blueprint as a parent.

Parameters **blueprint** (*Blueprint*) – The blueprint to add as a child.

Return type *Blueprint*

route(*routing_url*, *methods=('GET',)*, ***kwargs*)
Convenience decorator for adding a route.

This is equivalent to:

```
route = bp.wrap_route(func, **kwargs)
bp.add_route(route, routing_url, methods)
```

errorhandler(*code*)
Helper decorator for adding an error handler.

This is equivalent to:

```
route = bp.add_errorhandler(cbl, code)
```

Parameters **code** (int) – The error handler code to use.

wrap_route(*cbl*, **args*, ***kwargs*)
Wraps a callable in a Route.

This is required for routes to be added. :param cbl: The callable to wrap. :rtype: *Route* :return: A new *Route* object.

add_errorhandler(*cbl*, *errorcode*)
Adds an error handler to the table of error handlers.

A blueprint can only have one error handler per code. If it doesn't have an error handler for that code, it will try to fetch recursively the parent's error handler.

Parameters

- **cbl** – The callable error handler.
- **errorcode** (int) – The error code to handle, for example 404.

get_errorhandler(*exc*)
Recursively acquires the error handler for the specified error.

Parameters **exc** (Union[HTTPException, int]) – The exception to get the error handler for. This can either be a HTTPException object, or an integer.

Return type Union[None, *Route*]

Returns The *Route* object that corresponds to the error handler, or None if no error handler could be found.

get_hooks(*type_*)
Gets a list of hooks that match the current type.

These are ordered from parent to child.

Parameters **type** (str) – The type of hooks to get (currently "pre" or "post").

Returns An iterable of hooks to run.

add_hook(*type_*, *hook*)
Adds a hook to the current Blueprint.

Parameters

- **type** (str) – The type of hook to add (currently "pre" or "post").
- **hook** – The callable function to add as a hook.

after_request(*func*)
Convenience decorator to add a post-request hook.

before_request(*func*)
Convenience decorator to add a pre-request hook.

add_route(*route*, *routing_url*, *methods=('GET',)*)
Adds a route to the routing table and map.

Parameters

- **route** (*Route*) – The route object to add.

  This can be gotten from `Blueprint.wrap_route`, or by directly creating a Route object.

- **routing_url** (str) – The Werkzeug-compatible routing URL to add this route under.

  For more information, see http://werkzeug.pocoo.org/docs/0.11/routing/.

- **methods** (Iterable[str]) – An iterable of valid method this route can be called with.

Returns The unmodified *Route* object.

get_route(*endpoint*)
Gets the route associated with an endpoint.

Return type Optional[*Route*]

add_route_group(*group*)
Adds a route group to the current Blueprint.

Parameters **group** (*RouteGroup*) – The *RouteGroup* to add.

url_for(*environment*, *endpoint*, *, *method=None*, ***kwargs*)
Gets the URL for a specified endpoint using the arguments of the route.

This works very similarly to Flask's `url_for`.

It is not recommended to invoke this method directly - instead, `url_for` is set on the context object that is provided to your user function. This will allow you to invoke it with the correct environment already set.

Parameters

- **environment** (dict) – The WSGI environment to use to bind to the adapter.

- **endpoint** (str) – The endpoint to try and retrieve.
- **method** (Optional[str]) – If set, the method to explicitly provide (for similar endpoints with different allowed routes).
- **kwargs** – Keyword arguments to provide to the route.

Return type str

Returns The built URL for this endpoint.

match(*environment*)

Matches with the WSGI environment.

Parameters **environment** (dict) – The environment dict to perform matching with.

You can use the environ argument of a Request to get the environment back.

Return type Tuple[*Route*, Container[Any]]

Returns A Route object, which can be invoked to return the right response, and the parameters to invoke it with.

kyoukai.route

Routes are wrapped function objects that are called upon a HTTP request.

Functions

wrap_response(args[, response_class])	Wrap up a response, if applicable.

Classes

Response([response, status, headers, ...])	Full featured response object implementing the following mixins:
Route(function[, reverse_hooks, ...])	A route object is a wrapped function.
Rule(string[, defaults, subdomain, methods, ...])	A Rule represents one URL pattern.

Exceptions

HTTPException([description, response])	Baseclass for all HTTP exceptions.
InternalServerError([description, response])	*500 Internal Server Error*

class kyoukai.route.**Route**(*function*, *reverse_hooks=False*, *should_invoke_hooks=True*, *do_argument_checking=True*)

Bases: object

A route object is a wrapped function. They invoke this function when invoked on routing and calling.

Parameters

- **function** – The underlying callable. This can be a function, or any other callable.
- **reverse_hooks** (bool) – If the request hooks should be reversed for this request (i.e

child to parent.)

- **should_invoke_hooks** (bool) – If request hooks should be invoked. This is automatically False for error handlers.
- **do_argument_checking** (bool) – If argument type and name checking is enabled for this route.

do_argument_checking = None
If this route should do argument checking.

bp = None
The *Blueprint* this route is associated with.

rule = None
The Rule associated with this route.

methods = None
A list of methods associated with this rule.

hooks = None
Our own specific hooks.

create_rule()
Creates the rule object used by this route.

Return type Rule

Returns A new werkzeug.routing.Rule that is to be used for this route.

get_endpoint_name(*bp=None*)
Gets the endpoint name for this route.

coroutine invoke_function(*self*, *ctx*, *pre_hooks*, *post_hooks*, *params*)
Invokes the underlying callable.

This is for use in chaining routes. :param ctx: The *HTTPRequestContext* to use for this route. :type pre_hooks: list :param pre_hooks: A list of hooks to call before the route is invoked. :type post_hooks: list :param post_hooks: A list of hooks to call after the route is invoked. :param params: The parameters to pass to the function. :return: The result of the invoked function.

check_route_args(*params=None*)
Checks the arguments for a route.

Parameters **params** (Optional[dict]) – The parameters passed in, as a dict.

Raises **TypeError** – If the arguments passed in were not correct.

add_hook(*type_*, *hook*)
Adds a hook to the current Route.

Parameters

- **type** (str) – The type of hook to add (currently "pre" or "post").
- **hook** – The callable function to add as a hook.

get_hooks(*type_*)
Gets the hooks for the current Route for the type.

Parameters **type** (str) – The type to get.

Returns A list of callables.

before_request(*func*)
Convenience decorator to add a post-request hook.

after_request (*func*)
Convenience decorator to add a pre-request hook.

coroutine invoke (*self*, *ctx*, *params=None*)
Invokes a route. This will run the underlying function.

Parameters

- **ctx** – The *HTTPRequestContext* which is used in this request.
- **params** (Optional[Container[+T_co]]) – Any params that are used in this request.

Return type Response

Returns The result of the route's function.

kyoukai.routegroup

Route groups are classes that allow you to group a set of routes together.

Functions

after_request(func)	Helper decorator to mark a function as a post-request hook.
before_request(func)	Helper decorator to mark a function as a pre-request hook.
errorhandler(code)	A companion function to the RouteGroup class.
get_rg_bp(group)	Gets the *Blueprint* created from a *RouteGroup*.
hook(type_)	Marks a function as a hook.
route(url[, methods])	A companion function to the RouteGroup class.

Classes

RouteGroup	A route group is a class that contains multiple methods that are decorated with the route decorator.
RouteGroupType(name, bases, class_body, **kwargs)	The metaclass for a route group.

kyoukai.routegroup.**get_rg_bp**(*group*)
Gets the *Blueprint* created from a *RouteGroup*.

class kyoukai.routegroup.**RouteGroupType**(*name*, *bases*, *class_body*, ***kwargs*)
Bases: type

The metaclass for a route group.

This is responsible for passing the keyword arguments to the metaclass.

Override of *__init__* to store the blueprint params.

_init_blueprint(*obb*)
Initializes the Blueprint used by this route group.

Parameters **obb** – The route group instance to intialize.

mro() → list
return a type's method resolution order

kyoukai.routegroup.**route**(*url*, *methods=('GET',)*, ***kwargs*)
A companion function to the RouteGroup class. This follows *Blueprint.route()* in terms of arguments, and marks a function as a route inside the class.

This will return the original function, with some attributes attached:

- `in_group`: Marks the function as in the route group.
- `rg_delegate`: Internal. The type of function inside the group this is.
- `route_kwargs`: Keyword arguments to provide to `wrap_route`.
- `route_url`: The routing URL to provide to `add_route`.
- `route_methods`: The methods for the route.
- `route_hooks`: A defaultdict of route-specific hooks.

Additionally, the following methods are added.

- `hook`: A decorator that adds a hook of type `type_`.
- `before_request`: A decorator that adds a `pre` hook.
- `after_request`: A decorator that adds a `post` hook.

New in version 2.1.1.

Changed in version 2.1.3: Added the ability to add route-specific hooks.

Parameters

- **url** (`str`) – The routing URL of the route.
- **methods** (`Iterable[str]`) – An iterable of methods for the route.

kyoukai.routegroup.**errorhandler**(*code*)
A companion function to the RouteGroup class. This follows *Blueprint.errorhandler()* in terms of arguments.

Parameters **code** (`int`) – The code for the error handler.

kyoukai.routegroup.**hook**(*type_*)
Marks a function as a hook.

Parameters **type** (`str`) – The type of hook to mark.

kyoukai.routegroup.**before_request**(*func*)
Helper decorator to mark a function as a pre-request hook.

kyoukai.routegroup.**after_request**(*func*)
Helper decorator to mark a function as a post-request hook.

class kyoukai.routegroup.**RouteGroup**
Bases: `object`

A route group is a class that contains multiple methods that are decorated with the route decorator. They produce a blueprint that can be added to the tree that includes all methods in the route group.

```
class MyGroup(RouteGroup, url_prefix="/api/v1"):
    def __init__(self, something: str):
        self.something = something

    @route("/ping")
    async def ping(self, ctx: HTTPRequestContext):
        return '{"response": self.something}'
```

Blueprint parameters can be passed in the class call.

To add the route group as a blueprint, use `Blueprint.add_route_group(MyGroup, *args, **kwargs)()`.

kyoukai.testing

Testing helpers for Kyoukai.

Functions

`to_wsgi_environment`(headers, method, path, ...)	Produces a new WSGI environment from a set of data that is passed in.

Classes

`Blueprint`(name[, parent, prefix, ...])	A Blueprint class contains a Map of URL rules, which is checked and ran for every
`BytesIO`	Buffered I/O implementation using an in-memory bytes buffer.
`Context`([parent, default_timeout])	Contexts give request handlers and callbacks access to resources.
`Kyoukai`(application_name, *[, server_name])	The Kyoukai type is the core of the Kyoukai framework, and the core of your web application based upon the Kyoukai framework.
`Request`(environ[, populate_request, shallow])	Full featured request object implementing the following mixins:
`Response`([response, status, headers, ...])	Full featured response object implementing the following mixins:
`TestKyoukai`(*args[, base_context])	A special subclass that allows you to easily test your Kyoukai-based app.

class `kyoukai.testing.`**`_TestingBpCtxManager`**(*app*)
Bases: `object`

A context manager that is returned from *`testing_bp()`*. When entered, this will produce a new Blueprint object, that is then set onto the test application as the root blueprint.

After exiting, it will automatically restore the old root Blueprint onto the application, allowing complete isolation of individual test routes away from eachother.

class `kyoukai.testing.`**`TestKyoukai`**(**args*, *base_context=None*, ***kwargs*)
Bases: *`kyoukai.app.Kyoukai`*

A special subclass that allows you to easily test your Kyoukai-based app.

Parameters **`base_context`** (`Optional`[`Context`]) – The base context to use for all request testing.

classmethod **`wrap_existing_app`**(*other_app*, *base_context=None*)
Wraps an existing app in a test frame.

This allows easy usage of writing unit tests:

```
# main.py
kyk = Kyoukai("my_app")

# test.py
testing = TestKyoukai.wrap_existing_app(other_app)
# use testing as you would normally
```

Parameters

- **other_app** (*Kyoukai*) – The application object to wrap. Internally, this creates a new instance of ourselves, then sets the `process_request` of the subclass to the copied object.

  This means whenever `inject_request` is called, it will use the old app's process_request to run with, which will use the environment of the previous instance.

  Of course, if the old app has any side effects upon process_request, these side effects will happen when the testing application runs as well, as the old app is completely copied over.

- **base_context** (`Optional`[`Context`]) – The base context to use for this.

testing_bp()

Context handler that allows `with TestKyoukai.testing_bp() as bp:`

You can then register items onto this new root blueprint until __exit__, which will then destroy the blueprint.

Return type *_TestingBpCtxManager*

coroutine inject_request(*self*, *headers*, *url*, *method='GET'*, *body=None*)

Injects a request into the test client.

This will automatically create the correct context.

Parameters

- **headers** (`dict`) – The headers to use.
- **body** (`Optional`[`str`]) – The body to use.
- **url** (`str`) – The URL to use.
- **method** (`str`) – The method to use.

Return type `Response`

Returns The result.

finalize()

Finalizes the app and blueprints.

This will calculate the current `werkzeug.routing.Map` which is required for routing to work.

coroutine handle_httpexception(*self*, *ctx*, *exception*, *environ=None*)

Handle a HTTP Exception.

Parameters

- **ctx** (*HTTPRequestContext*) – The context of the request.
- **exception** (`HTTPException`) – The HTTPException
- **environ** (`Optional`[`dict`]) – The fake WSGI environment.

Return type `Response`

Returns A `werkzeug.wrappers.Response` that handles this response.

log_route(*request*, *code*)
Logs a route invocation.

Parameters

- **request** (`Request`) – The request produced.
- **code** (`int`) – The response code of the route.

coroutine process_request(*self*, *request*, *parent_context*)
Processes a Request and returns a Response object.

This is the main processing method of Kyoukai, and is meant to be used by one of the HTTP server backends, and not by client code.

Parameters

- **request** (`Request`) – The `werkzeug.wrappers.Request` object to process. A new *HTTPRequestContext* will be provided to wrap this request inside of to client code.
- **parent_context** (`Context`) – The `asphalt.core.Context` that is the parent context for this particular app. It will be used as the parent for the HTTPRequestContext.

Return type `Response`

Returns A `werkzeug.wrappers.Response` object that can be written to the client as a response.

register_blueprint(*child*)
Registers a child blueprint to this app's root Blueprint.

This will set up the Blueprint tree, as well as setting up the routing table when finalized.

Parameters **child** (*Blueprint*) – The child Blueprint to add. This must be an instance of *Blueprint*.

request_class
alias of `Request`

response_class
alias of `Response`

root

Return type *Blueprint*

Returns The root Blueprint for the routing tree.

run(*ip='127.0.0.1'*, *port=4444*, ***, *component=None*)
Runs the Kyoukai server from within your code.

This is not normally invoked - instead Asphalt should invoke the Kyoukai component. However, this is here for convenience.

coroutine start(*self*, *ip='127.0.0.1'*, *port=4444*, ***, *component=None*, *base_context=None*)
Runs the Kyoukai component asynchronously.

This will bypass Asphalt's default runner, and allow you to run your app easily inside something else, for example.

Parameters

- **ip** (`str`) – The IP of the built-in server.

- **port** (int) – The port of the built-in server.
- **component** – The component to start the app with. This should be an instance of *kyoukai.asphalt.KyoukaiComponent*.
- **base_context** (Optional[Context]) – The base context that the HTTPRequest-Context should be started with.

kyoukai.util

Misc utilities for usage inside the framework.

Functions

as_html(text[, code, headers])	Returns a HTML response.
as_json(data[, code, headers, json_encoder])	Returns a JSON response.
as_plaintext(text[, code, headers])	Returns a plaintext response.
wrap_response(args[, response_class])	Wrap up a response, if applicable.

Classes

Response([response, status, headers, ...])	Full featured response object implementing the following mixins:

kyoukai.util.**as_html**(*text*, *code=200*, *headers=None*)

Returns a HTML response.

```
return as_html("<h1>Hel Na</h1>", code=403)
```

Parameters

- **text** (str) – The text to return.
- **code** (int) – The status code of the response.
- **headers** (Optional[dict]) – Any optional headers.

Return type Response

Returns A new werkzeug.wrappers.Response representing the HTML.

kyoukai.util.**as_plaintext**(*text*, *code=200*, *headers=None*)

Returns a plaintext response.

```
return as_plaintext("hel yea", code=201)
```

Parameters

- **text** (str) – The text to return.
- **code** (int) – The status code of the response.
- **headers** (Optional[dict]) – Any optional headers.

Return type Response

Returns A new werkzeug.wrappers.Response representing the text.

kyoukai.util.**as_json**(*data*, *code=200*, *headers=None*, ***, *json_encoder=None*)
Returns a JSON response.

```
return as_json({"response": "yes", "code": 201}, code=201)
```

Parameters

- **data** (Union[dict, list]) – The data to encode.
- **code** (int) – The status code of the response.
- **headers** (Optional[dict]) – Any optional headers.
- **json_encoder** (Optional[JSONEncoder]) – The encoder class to use to encode.

Return type Response

Returns A new werkzeug.wrappers.Response representing the JSON.

kyoukai.util.**wrap_response**(*args*, *response_class=<class 'werkzeug.wrappers.Response'>*)
Wrap up a response, if applicable. This allows Flask-like *return "whatever"*.

Parameters

- **args** – The arguments that are being wrapped.
- **response_class** (Response) – The Response class that is being used.

Return type Response

class kyoukai.**Kyoukai**(*application_name*, ***, *server_name=None*, ***kwargs*)
Bases: object

The Kyoukai type is the core of the Kyoukai framework, and the core of your web application based upon the Kyoukai framework. It acts as a central router and request processor that takes in requests from the protocols and returns responses.

The application name is currently unused, but it is good practice to set it correctly anyway in case it is used in future editions of Kyoukai.

You normally create an application instance inside your component file, like so:

```
from kyoukai.app import Kyoukai

... # setup code

kyk = Kyoukai("my_app")
kyk.register_blueprint(whatever)

... # other setup

class MyContainer(ContainerComponent):
    async def start(self, ctx):
        self.add_component('kyoukai', KyoukaiComponent, ip="127.0.0.1", port=4444,
                           app="app:app")
```

Of course, you can also embed Kyoukai inside another app, by awaiting *Kyoukai.start()*.

Parameters

- **application_name** (str) – The name of the application that is being created. This is passed to the *Blueprint* being created as the root blueprint.
- **server_name** (Optional[str]) – Keyword-only. The SERVER_NAME to use inside the fake WSGI environment created for url_for, if applicable.
- **host_matching** – Should host matching be enabled? This will be implicitly True if host is not None.
- **host** – The host used for host matching, to be passed to the root Blueprint. By default, no host is used, so all hosts are matched on the root Blueprint.
- **application_root** – Keyword-only. The APPLICATION_ROOT to use inside the fake WSGI environment created for url_for, if applicable.
- **loop** – Keyword-only. The asyncio event loop to use for this app. If no loop is specified it, will be automatically fetched using asyncio.get_event_loop().
- **request_class** – Keyword-only. The custom request class to instantiate requests with.
- **response_class** – Keyword-only. The custom response class to instantiate responses with.

finalize()
Finalizes the app and blueprints.

This will calculate the current werkzeug.routing.Map which is required for routing to work.

coroutine handle_httpexception(*self*, *ctx*, *exception*, *environ=None*)
Handle a HTTP Exception.

Parameters

- **ctx** (*HTTPRequestContext*) – The context of the request.
- **exception** (HTTPException) – The HTTPException
- **environ** (Optional[dict]) – The fake WSGI environment.

Return type Response

Returns A werkzeug.wrappers.Response that handles this response.

log_route(*request*, *code*)
Logs a route invocation.

Parameters

- **request** (Request) – The request produced.
- **code** (int) – The response code of the route.

coroutine process_request(*self*, *request*, *parent_context*)
Processes a Request and returns a Response object.

This is the main processing method of Kyoukai, and is meant to be used by one of the HTTP server backends, and not by client code.

Parameters

- **request** (Request) – The werkzeug.wrappers.Request object to process. A new *HTTPRequestContext* will be provided to wrap this request inside of to client code.
- **parent_context** (Context) – The asphalt.core.Context that is the parent context for this particular app. It will be used as the parent for the HTTPRequestContext.

Return type Response

Returns A werkzeug.wrappers.Response object that can be written to the client as a response.

register_blueprint(*child*)
Registers a child blueprint to this app's root Blueprint.

This will set up the Blueprint tree, as well as setting up the routing table when finalized.

Parameters **child** (*Blueprint*) – The child Blueprint to add. This must be an instance of *Blueprint*.

request_class
alias of Request

response_class
alias of Response

root

Return type *Blueprint*

Returns The root Blueprint for the routing tree.

run(*ip='127.0.0.1'*, *port=4444*, ***, *component=None*)
Runs the Kyoukai server from within your code.

This is not normally invoked - instead Asphalt should invoke the Kyoukai component. However, this is here for convenience.

coroutine start(*self*, *ip='127.0.0.1'*, *port=4444*, ***, *component=None*, *base_context=None*)
Runs the Kyoukai component asynchronously.

This will bypass Asphalt's default runner, and allow you to run your app easily inside something else, for example.

Parameters

- **ip** (str) – The IP of the built-in server.
- **port** (int) – The port of the built-in server.
- **component** – The component to start the app with. This should be an instance of *kyoukai.asphalt.KyoukaiComponent*.
- **base_context** (Optional[Context]) – The base context that the HTTPRequest-Context should be started with.

class kyoukai.**HTTPRequestContext**(*parent*, *request*)
Bases: asphalt.core.context.Context

The context subclass passed to all requests within Kyoukai.

get_resources(*type=None*, ***, *include_parents=True*)
Return the currently published resources specific to one type or all types.

Parameters

- **type** (Union[str, type, None]) – type of the resources to return, or None to return all resources
- **include_parents** (bool) – include the resources from parent contexts

Return type Sequence[Resource]

parent
Return the parent of this context or `None` if there is no parent context.

Return type `Optional[Context]`

publish_lazy_resource(*creator*, *types*, *alias='default'*, *context_attr=None*)
Publish a "lazy" or "contextual" resource and dispatch a `resource_published` event.

Instead of a concrete resource value, you supply a creator callable which is called with a context object as its argument when the resource is being requested either via *request_resource()* or by context attribute access. The return value of the creator callable will be cached so the creator will only be called once per context instance.

If the creator callable is a coroutine function or returns an awaitable, it is resolved before storing the resource value and returning it to the requester. Note that this will **NOT** work when a context attribute has been specified for the resource.

Parameters

- **creator** (`Callable[[Context], Any]`) – a callable taking a context instance as argument
- **types** (`Union[type, Iterable[Union[str, type]]]`) – type(s) to register the resource as
- **alias** (`str`) – name of this resource (unique among all its registered types)
- **context_attr** (`Optional[str]`) – name of the context attribute this resource will be accessible as

Return type `Resource`

Returns the resource handle

Raises **asphalt.core.context.ResourceConflict** – if there is an existing resource creator for the given types or context variable

publish_resource(*value*, *alias='default'*, *context_attr=None*, *, *types=()*)
Publish a resource and dispatch a `resource_published` event.

Parameters

- **value** – the actual resource value
- **alias** (`str`) – name of this resource (unique among all its registered types)
- **context_attr** (`Optional[str]`) – name of the context attribute this resource will be accessible as
- **types** (`Union[type, Iterable[Union[str, type]]]`) – type(s) to register the resource as (omit to use the type of `value`)

Return type `Resource`

Returns the resource handle

Raises **asphalt.core.context.ResourceConflict** – if the resource conflicts with an existing one in any way

remove_resource(*resource*)
Remove the given resource from the collection and dispatch a `resource_removed` event.

Parameters **resource** (`Resource`) – the resource to be removed

Raises **LookupError** – the given resource was not in the collection

coroutine request_resource (*self*, *type*, *alias='default'*, ***, *timeout=None*)
Request a resource matching the given type and alias.

If no such resource was found, this method will wait `timeout` seconds for it to become available. The timeout does not apply to resolving awaitables created by lazy resource creators.

Parameters

- **type** (`Union[str, type]`) – type of the requested resource
- **alias** (`str`) – alias of the requested resource
- **timeout** (`Union[int, float, None]`) – the timeout (in seconds; omit to use the context's default timeout)

Returns the value contained by the requested resource (**NOT** a `Resource` instance)

Raises **asphalt.core.context.ResourceNotFound** – if the requested resource does not become available in the allotted time

url_for (*endpoint*, ***, *method*, ***kwargs*)
A context-local version of `url_for`.

For more information, see the documentation on *url_for()*.

class kyoukai.**KyoukaiComponent** (*app*, *ip='127.0.0.1'*, *port=4444*, ***cfg*)
Bases: *kyoukai.asphalt.KyoukaiBaseComponent*

A component for Kyoukai.

This includes the built-in HTTP server.

Creates a new component.

Parameters

- **app** – The application object to use. This can either be the real application object, or a string that resolves to a reference for the real application object.
- **ip** (`str`) – If using the built-in HTTP server, the IP to bind to.
- **port** (`int`) – If using the built-in HTTP server, the port to bind to.
- **cfg** – Additional configuration.

get_protocol (*ctx*, *serv_info*)
Gets the protocol to use for this webserver.

get_server_name ()

Returns The server name of this app.

coroutine start (*self*, *ctx*)
Starts the webserver if required.

Parameters **ctx** (`Context`) – The base context.

class kyoukai.**Blueprint** (*name*, *parent=None*, *prefix=''*, ***, *host_matching=False*, *host=None*)
Bases: `object`

A Blueprint class contains a Map of URL rules, which is checked and ran for every

Parameters

- **name** (`str`) – The name of this Blueprint. This is used when generating endpoints in the finalize stage.

- **parent** (Optional[*Blueprint*]) – The parent of this Blueprint. Parent blueprints will gather the routes of their children, and return a giant `werkzeug.routing.Map` object that contains all of the route maps in the children
- **prefix** (str) – The prefix to be added to the start of every route name. This is inherited from parents - the parent prefix will also be added to the start of every route.
- **host_matching** (bool) – Should host matching be enabled? This is implicitly True if `host` is non-None.
- **host** (Optional[str]) – The host of the Blueprint. Used for custom subdomain routing. If this is None, then this Blueprint will be used for all hosts.

add_child(*blueprint*)
Adds a Blueprint as a child of this one. This is automatically called when using another Blueprint as a parent.

Parameters **blueprint** (*Blueprint*) – The blueprint to add as a child.

Return type *Blueprint*

add_errorhandler(*cbl*, *errorcode*)
Adds an error handler to the table of error handlers.

A blueprint can only have one error handler per code. If it doesn't have an error handler for that code, it will try to fetch recursively the parent's error handler.

Parameters

- **cbl** – The callable error handler.
- **errorcode** (int) – The error code to handle, for example 404.

add_hook(*type_*, *hook*)
Adds a hook to the current Blueprint.

Parameters

- **type** (str) – The type of hook to add (currently "pre" or "post").
- **hook** – The callable function to add as a hook.

add_route(*route*, *routing_url*, *methods=('GET',)*)
Adds a route to the routing table and map.

Parameters

- **route** (*Route*) – The route object to add.

  This can be gotten from `Blueprint.wrap_route`, or by directly creating a Route object.
- **routing_url** (str) – The Werkzeug-compatible routing URL to add this route under.

  For more information, see http://werkzeug.pocoo.org/docs/0.11/routing/.
- **methods** (Iterable[str]) – An iterable of valid method this route can be called with.

Returns The unmodified *Route* object.

add_route_group(*group*)
Adds a route group to the current Blueprint.

Parameters **group** (*RouteGroup*) – The *RouteGroup* to add.

after_request(*func*)
Convenience decorator to add a post-request hook.

before_request(*func*)
Convenience decorator to add a pre-request hook.

errorhandler(*code*)
Helper decorator for adding an error handler.

This is equivalent to:

```
route = bp.add_errorhandler(cbl, code)
```

Parameters **code** (int) – The error handler code to use.

finalize(***map_options*)
Called on the root Blueprint when all Blueprints have been registered and the app is starting.

This will automatically build a werkzeug.routing.Map of werkzeug.routing.Rule objects for each Blueprint.

Note: Calling this on sub-blueprints will have no effect, apart from generating a Map. It is recommended to only call this on the root Blueprint.

Parameters **map_options** – The options to pass to the created Map.

Return type Map

Returns The werkzeug.routing.Map created from the routing tree.

get_errorhandler(*exc*)
Recursively acquires the error handler for the specified error.

Parameters **exc** (Union[HTTPException, int]) – The exception to get the error handler for. This can either be a HTTPException object, or an integer.

Return type Union[None, *Route*]

Returns The *Route* object that corresponds to the error handler, or None if no error handler could be found.

get_hooks(*type_*)
Gets a list of hooks that match the current type.

These are ordered from parent to child.

Parameters **type** (str) – The type of hooks to get (currently "pre" or "post").

Returns An iterable of hooks to run.

get_route(*endpoint*)
Gets the route associated with an endpoint.

Return type Optional[*Route*]

host

Return type str

Returns The host for this Blueprint, or the host of any parent Blueprint.

match(*environment*)
Matches with the WSGI environment.

Parameters **environment** (dict) – The environment dict to perform matching with.

You can use the environ argument of a Request to get the environment back.

Return type Tuple[*Route*, Container[Any]]

Returns A Route object, which can be invoked to return the right response, and the parameters to invoke it with.

parent

Return type *Blueprint*

Returns The parent Blueprint of this blueprint.

prefix

Return type str

Returns The combined prefix of this Blueprint.

route(*routing_url*, *methods=('GET',)*, ***kwargs*)
Convenience decorator for adding a route.

This is equivalent to:

```
route = bp.wrap_route(func, **kwargs)
bp.add_route(route, routing_url, methods)
```

traverse_tree()
Traverses the tree for children Blueprints.

Return type Generator[*Blueprint*, None, None]

tree_routes

Return type Generator[*Route*, None, None]

Returns A generator that yields all routes from the tree, from parent to children.

url_for(*environment*, *endpoint*, *, *method=None*, ***kwargs*)
Gets the URL for a specified endpoint using the arguments of the route.

This works very similarly to Flask's url_for.

It is not recommended to invoke this method directly - instead, url_for is set on the context object that is provided to your user function. This will allow you to invoke it with the correct environment already set.

Parameters

- **environment** (dict) – The WSGI environment to use to bind to the adapter.
- **endpoint** (str) – The endpoint to try and retrieve.
- **method** (Optional[str]) – If set, the method to explicitly provide (for similar endpoints with different allowed routes).
- **kwargs** – Keyword arguments to provide to the route.

Return type str

Returns The built URL for this endpoint.

wrap_route(*cbl*, **args*, ***kwargs*)
Wraps a callable in a Route.

This is required for routes to be added. :param cbl: The callable to wrap. :rtype: *Route* :return: A new *Route* object.

class kyoukai.**Route**(*function*, *reverse_hooks=False*, *should_invoke_hooks=True*, *do_argument_checking=True*)

Bases: object

A route object is a wrapped function. They invoke this function when invoked on routing and calling.

Parameters

- **function** – The underlying callable. This can be a function, or any other callable.
- **reverse_hooks** (bool) – If the request hooks should be reversed for this request (i.e child to parent.)
- **should_invoke_hooks** (bool) – If request hooks should be invoked. This is automatically False for error handlers.
- **do_argument_checking** (bool) – If argument type and name checking is enabled for this route.

add_hook(*type_*, *hook*)

Adds a hook to the current Route.

Parameters

- **type** (str) – The type of hook to add (currently "pre" or "post").
- **hook** – The callable function to add as a hook.

after_request(*func*)

Convenience decorator to add a pre-request hook.

before_request(*func*)

Convenience decorator to add a post-request hook.

check_route_args(*params=None*)

Checks the arguments for a route.

Parameters **params** (Optional[dict]) – The parameters passed in, as a dict.

Raises **TypeError** – If the arguments passed in were not correct.

create_rule()

Creates the rule object used by this route.

Return type Rule

Returns A new werkzeug.routing.Rule that is to be used for this route.

get_endpoint_name(*bp=None*)

Gets the endpoint name for this route.

get_hooks(*type_*)

Gets the hooks for the current Route for the type.

Parameters **type** (str) – The type to get.

Returns A list of callables.

coroutine invoke(*self*, *ctx*, *params=None*)

Invokes a route. This will run the underlying function.

Parameters

- **ctx** – The *HTTPRequestContext* which is used in this request.

- **params** (Optional[Container[+T_co]]) – Any params that are used in this request.

Return type Response

Returns The result of the route's function.

coroutine invoke_function(*self*, *ctx*, *pre_hooks*, *post_hooks*, *params*)
Invokes the underlying callable.

This is for use in chaining routes. :param ctx: The *HTTPRequestContext* to use for this route. :type pre_hooks: list :param pre_hooks: A list of hooks to call before the route is invoked. :type post_hooks: list :param post_hooks: A list of hooks to call after the route is invoked. :param params: The parameters to pass to the function. :return: The result of the invoked function.

class kyoukai.**RouteGroup**
Bases: object

A route group is a class that contains multiple methods that are decorated with the route decorator. They produce a blueprint that can be added to the tree that includes all methods in the route group.

```
class MyGroup(RouteGroup, url_prefix="/api/v1"):
    def __init__(self, something: str):
        self.something = something

    @route("/ping")
    async def ping(self, ctx: HTTPRequestContext):
        return '{"response": self.something}'
```

Blueprint parameters can be passed in the class call.

To add the route group as a blueprint, use `Blueprint.add_route_group(MyGroup, *args, **kwargs)()`.

class kyoukai.**TestKyoukai**(**args*, *base_context=None*, ***kwargs*)
Bases: *kyoukai.app.Kyoukai*

A special subclass that allows you to easily test your Kyoukai-based app.

Parameters **base_context** (Optional[Context]) – The base context to use for all request testing.

finalize()
Finalizes the app and blueprints.

This will calculate the current werkzeug.routing.Map which is required for routing to work.

coroutine handle_httpexception(*self*, *ctx*, *exception*, *environ=None*)
Handle a HTTP Exception.

Parameters

- **ctx** (*HTTPRequestContext*) – The context of the request.
- **exception** (HTTPException) – The HTTPException
- **environ** (Optional[dict]) – The fake WSGI environment.

Return type Response

Returns A werkzeug.wrappers.Response that handles this response.

coroutine inject_request(*self*, *headers*, *url*, *method='GET'*, *body=None*)
Injects a request into the test client.

This will automatically create the correct context.

Parameters

- **headers** (dict) – The headers to use.
- **body** (Optional[str]) – The body to use.
- **url** (str) – The URL to use.
- **method** (str) – The method to use.

Return type Response

Returns The result.

log_route(*request*, *code*)
Logs a route invocation.

Parameters

- **request** (Request) – The request produced.
- **code** (int) – The response code of the route.

coroutine process_request(*self*, *request*, *parent_context*)
Processes a Request and returns a Response object.

This is the main processing method of Kyoukai, and is meant to be used by one of the HTTP server backends, and not by client code.

Parameters

- **request** (Request) – The werkzeug.wrappers.Request object to process. A new *HTTPRequestContext* will be provided to wrap this request inside of to client code.
- **parent_context** (Context) – The asphalt.core.Context that is the parent context for this particular app. It will be used as the parent for the HTTPRequestContext.

Return type Response

Returns A werkzeug.wrappers.Response object that can be written to the client as a response.

register_blueprint(*child*)
Registers a child blueprint to this app's root Blueprint.

This will set up the Blueprint tree, as well as setting up the routing table when finalized.

Parameters **child** (*Blueprint*) – The child Blueprint to add. This must be an instance of *Blueprint*.

request_class
alias of Request

response_class
alias of Response

root

Return type *Blueprint*

Returns The root Blueprint for the routing tree.

run(*ip='127.0.0.1'*, *port=4444*, ***, *component=None*)
Runs the Kyoukai server from within your code.

This is not normally invoked - instead Asphalt should invoke the Kyoukai component. However, this is here for convenience.

coroutine start (*self*, *ip='127.0.0.1'*, *port=4444*, ***, *component=None*, *base_context=None*)
Runs the Kyoukai component asynchronously.

This will bypass Asphalt's default runner, and allow you to run your app easily inside something else, for example.

Parameters

- **ip** (str) – The IP of the built-in server.
- **port** (int) – The port of the built-in server.
- **component** – The component to start the app with. This should be an instance of *kyoukai.asphalt.KyoukaiComponent*.
- **base_context** (Optional[Context]) – The base context that the HTTPRequestContext should be started with.

testing_bp()
Context handler that allows `with TestKyoukai.testing_bp() as bp:`

You can then register items onto this new root blueprint until __exit__, which will then destroy the blueprint.

Return type *_TestingBpCtxManager*

classmethod wrap_existing_app (*other_app*, *base_context=None*)
Wraps an existing app in a test frame.

This allows easy usage of writing unit tests:

```
# main.py
kyk = Kyoukai("my_app")

# test.py
testing = TestKyoukai.wrap_existing_app(other_app)
# use testing as you would normally
```

Parameters

- **other_app** (*Kyoukai*) – The application object to wrap. Internally, this creates a new instance of ourselves, then sets the process_request of the subclass to the copied object.

  This means whenever inject_request is called, it will use the old app's process_request to run with, which will use the environment of the previous instance.

  Of course, if the old app has any side effects upon process_request, these side effects will happen when the testing application runs as well, as the old app is completely copied over.
- **base_context** (Optional[Context]) – The base context to use for this.

Kyoukai Changelog

Here you can see the list of changes between each Kyoukai release.

Version 2.1.3

- Add `errorhandler()` to mark a function inside a route group as an error handler.
- Add request hook support to route groups.
- Add `as_html()`, `as_plaintext()`, `as_json()` helper methods.
- Add **Host Matching** support. See *Host Matching*.

Version 2.1.2

- Add *RouteGroup*.

Version 2.1.1

- Fix request bodies not being read properly.
- Fix loop propagation.
- Fix http2 module for H2 3.0.0.

Version 2.1.0

- Add *Route.hooks* property to *Route*, which allows route-specific hooks.
- Add the ability to disable argument conversion on *Route* objects.
- Automatically disable argument conversion on error handlers.
- HTTP/2 is now automatically enabled in all requests over TLS, if available.
- HTTPS is now easier to configure (requires one config file change).

Version 2.0.5

- Add `REMOTE_ADDR` and `REMOTE_PORT` to WSGI environ in httptools backend.
- Add `REMOTE_ADDR` and `REMOTE_PORT` to WSGI environ in h2 backend.

Version 2.0.4.1

- Automatically stringify the response body.

Version 2.0.3

- Fix Content-Type and Content-Length header parsing.
- Add automatic JSON form parsing.
- Log when a HTTPException is raised inside a route function.

Version 2.0.2

- Automatic argument conversion now ignores functions with _empty params.

Version 2.0.1

- Error handlers can now handle errors that happen in other error handlers.

Version 2.0

Version 2.0 is a major overhaul of the library, simplifying it massively and removing a lot of redundant or otherwise overly complex code.

- Requests and responses are now based on Werkzeug data structures. Werkzeug is a much more battle tested library than Kyoukai; it ensures that there are less edge cases during HTTP parsing.
- Routing is now handled by Werkzeug and the Rule/Map based router rather than overly complex regex routes.
- The application object is now I/O blind - it will take in a Request object and produce a Response object, instead of writing to the stream directly.
- A new `gunicorn` HTTP backend has been added - using the `gaiohttp` worker, gunicorn can now be connected to Kyoukai.
- A new `uwsgi` HTTP backend has been added - uWSGI running in asyncio mode can now be connected to Kyoukai.
- A new HTTP/2 backend has been added which uses the pure Python `h2` library as a state machine for parsing HTTP frames.
- The `httptools` backend has been rewritten - it is now more reliable and supports chunked data streams.

Version 1.9.2

- Add `depth` property which signifies how deep in the tree the Blueprint is.
- The routing tree no longer considers matching routes that don't start with the prefix of the blueprint.
- Add `tree_path` property which shows the full tree path to a Blueprint.
- Add the ability to set 405 error handlers on Blueprints. The routing engine will automatically try and match the 405 on the lowest common ancestor of all routes that failed to match in the blueprint tree.
- Add `blueprint` and `route` attributes to *`HTTPRequestContext`*.
- Add `ip` and `port` attributes to `Request`.
- Correctly load cookies from the `Cookie` header from client requests.
- Converters will now handle `*args` and `**kwargs` in functions properly.
- HTTPExceptions have been overhauled to allow early exiting with a custom response. Do not abuse as a replacement for the return statement.

Version 1.9.1

- Large amount of code clean up relating to the embedded HTTP server. The HTTP server now uses httptools to create requests which is more reliable than http_parser.

Version 1.8.6

- Add a default static file handler.

Version 1.8.5

- Routing tree has been improved by allowing two routes with the same path but different methods to reside in two different blueprints.

Version 1.8.4

- Error handlers can now error themselves, and this is handled gracefully.
- If a match is invalid, it will raise a 500 error at compile time, which is usually when routes are first matched.

Version 1.8.3

- Converters can now be awaitables.

Version 1.8.2

- JSON forms are now lazy loaded when `.form` is called.

Version 1.8.1

- Fix crashing at startup without a startup function registered.
- Fix routing tree not working with multiple URL prefixes.
- Fix default converters.

Version 1.8.0

- Add the ability to override the Request and Response classes used in views with `app.request_cls` and `app.response_cls` respectively.
- Views now have the ability to change which Route class they use in the decorator.
- Implement the Werkzeug Debugger on 500 errors if the app is in debug mode.

Version 1.7.3

- Add the ability to register a callable to run on startup. This callable can be a regular function or a coroutine.

Version 1.7.2

- Form handling is now handled by Werkzeug.
- Add a new attribute, `kyoukai.request.Request.files` which stores uploaded files from the form passed in.
- Requests are no longer parsed multiple times.

Version 1.7.0

- Overhaul template renderers. This allows easier creation of a template renderer with a specific engine without having to use engine-specific code in views.
- Add a Jinja2 based renderer. This can be enabled by passing `template_renderer="jinja2"` in your application constructor.

Version 1.6.0

- Add converters. Converters allow annotations to be added to parameters which will automatically convert the argument passed in to that type, if possible.
- Exception handlers now take an `exception` param as the second arg, whcih is the HTTPException that caused this error handler to happen.

Version 1.5.0

- Large amount of internal codebase re-written.
- The Blueprint system was overhauled into a tree system which handles routes much better than before.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

k

Index

Symbols

A

B

C

D

L

M

N

O

P

R

S

T

U

W