Coaster Documentation

Release 0.7.0

Hasgeek

Jul 23, 2021

Contents

1 Coaster documentation **1**
- 1.1 Coaster types . . . 1
- 1.2 App configuration . . . 1
- 1.3 Logger . . . 19
- 1.4 Assets . . . 20
- 1.5 Utilities . . . 21
- 1.6 Miscellaneous utilities . . . 22
- 1.7 Date, time and timezone utilities . . . 29
- 1.8 Text processing utilities . . . 30
- 1.9 Markdown processor . . . 30
- 1.10 PostgreSQL query processor . . . 31
- 1.11 Utility classes . . . 32
- 1.12 Authentication management . . . 35
- 1.13 View helpers . . . 36
- 1.14 Miscellaneous view helpers . . . 36
- 1.15 View decorators . . . 37
- 1.16 Class-based views . . . 42
- 1.17 SQLAlchemy patterns . . . 46
- 1.18 SQLAlchemy mixin classes . . . 47
- 1.19 SQLAlchemy column types . . . 54
- 1.20 Helper functions . . . 57
- 1.21 Role-based access control . . . 58
- 1.22 SQLAlchemy attribute annotations . . . 65
- 1.23 Immutable annotation . . . 66
- 1.24 Model helper registry . . . 66
- 1.25 Enhanced query and custom comparators . . . 67
- 1.26 States and transitions . . . 69
- 1.27 Database session and instance . . . 76
- 1.28 Natural language processing . . . 76

2 Indices and tables **77**

Python Module Index **79**

Index **81**

CHAPTER 1

Coaster documentation

Coaster contains functions and db models for recurring patterns in Flask apps. Coaster is available under the BSD license, the same license as Flask.

1.1 Coaster types

`coaster.typing.`**`SimpleDecorator = typing.Callable[[typing.Callable], typing.Callable]`**

Type for a simple function decorator that does not accept options

1.2 App configuration

`class` `coaster.app.`**`KeyRotationWrapper`**(*cls*, *secret_keys*, ***kwargs*)

Wrapper to support multiple secret keys in itsdangerous.

The first secret key is used for all operations, but if it causes a BadSignature exception, the other secret keys are tried in order.

Parameters

- **`cls`** – Signing class from itsdangerous (eg: URLSafeTimedSerializer)
- **`secret_keys`** – List of secret keys
- **`kwargs`** – Arguments to pass to each signer/serializer

`class` `coaster.app.`**`RotatingKeySecureCookieSessionInterface`**

Replaces the serializer with key rotation support

`class` `coaster.app.`**`Flask`**(*import_name: str*, *static_url_path: Optional[str] = None*, *static_folder: Optional[str] = 'static'*, *static_host: Optional[str] = None*, *host_matching: bool = False*, *subdomain_matching: bool = False*, *template_folder: Optional[str] = 'templates'*, *instance_path: Optional[str] = None*, *instance_relative_config: bool = False*, *root_path: Optional[str] = None*)

The flask object implements a WSGI application and acts as the central object. It is passed the name of the module or package of the application. Once it is created it will act as a central registry for the view functions, the URL rules, template configuration and much more.

The name of the package is used to resolve resources from inside the package or the folder the module is contained in depending on if the package parameter resolves to an actual python package (a folder with an `__init__.py` file inside) or a standard module (just a `.py` file).

For more information about resource loading, see `open_resource()`.

Usually you create a *`Flask`* instance in your main module or in the `__init__.py` file of your package like this:

```
from flask import Flask
app = Flask(__name__)
```

About the First Parameter

The idea of the first parameter is to give Flask an idea of what belongs to your application. This name is used to find resources on the filesystem, can be used by extensions to improve debugging information and a lot more.

So it's important what you provide there. If you are using a single module, *__name__* is always the correct value. If you however are using a package, it's usually recommended to hardcode the name of your package there.

For example if your application is defined in `yourapplication/app.py` you should create it with one of the two versions below:

```
app = Flask('yourapplication')
app = Flask(__name__.split('.')[0])
```

Why is that? The application will work even with *__name__*, thanks to how resources are looked up. However it will make debugging more painful. Certain extensions can make assumptions based on the import name of your application. For example the Flask-SQLAlchemy extension will look for the code in your application that triggered an SQL query in debug mode. If the import name is not properly set up, that debugging information is lost. (For example it would only pick up SQL queries in *yourapplication.app* and not *yourapplication.views.frontend*)

New in version 0.7: The *static_url_path*, *static_folder*, and *template_folder* parameters were added.

New in version 0.8: The *instance_path* and *instance_relative_config* parameters were added.

New in version 0.11: The *root_path* parameter was added.

New in version 1.0: The `host_matching` and `static_host` parameters were added.

New in version 1.0: The `subdomain_matching` parameter was added. Subdomain matching needs to be enabled manually now. Setting `SERVER_NAME` does not implicitly enable it.

Parameters

- **`import_name`** – the name of the application package
- **`static_url_path`** – can be used to specify a different path for the static files on the web. Defaults to the name of the *static_folder* folder.
- **`static_folder`** – The folder with static files that is served at `static_url_path`. Relative to the application `root_path` or an absolute path. Defaults to `'static'`.
- **`static_host`** – the host to use when adding the static route. Defaults to None. Required when using `host_matching=True` with a `static_folder` configured.
- **`host_matching`** – set `url_map.host_matching` attribute. Defaults to False.

- **subdomain_matching** – consider the subdomain relative to `SERVER_NAME` when matching routes. Defaults to False.
- **template_folder** – the folder that contains the templates that should be used by the application. Defaults to `'templates'` folder in the root path of the application.
- **instance_path** – An alternative instance path for the application. By default the folder `'instance'` next to the package or module is assumed to be the instance path.
- **instance_relative_config** – if set to `True` relative filenames for loading the config are assumed to be relative to the instance path instead of the application root.
- **root_path** – The path to the root of the application files. This should only be set manually when it can't be detected automatically, such as for namespace packages.

add_template_filter (*f: Callable[[Any], str], name: Optional[str] = None*) → None
Register a custom template filter. Works exactly like the *`template_filter()`* decorator.

Parameters **name** – the optional name of the filter, otherwise the function name will be used.

add_template_global (*f: Callable[[], Any], name: Optional[str] = None*) → None
Register a custom template global function. Works exactly like the *`template_global()`* decorator.

New in version 0.10.

Parameters **name** – the optional name of the global function, otherwise the function name will be used.

add_template_test (*f: Callable[[Any], bool], name: Optional[str] = None*) → None
Register a custom template test. Works exactly like the *`template_test()`* decorator.

New in version 0.10.

Parameters **name** – the optional name of the test, otherwise the function name will be used.

add_url_rule (*rule: str, endpoint: Optional[str] = None, view_func: Optional[Callable] = None, provide_automatic_options: Optional[bool] = None, **options*) → None
Register a rule for routing incoming requests and building URLs. The `route()` decorator is a shortcut to call this with the `view_func` argument. These are equivalent:

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

```
def index():
    ...

app.add_url_rule("/", view_func=index)
```

See URL Route Registrations.

The endpoint name for the route defaults to the name of the view function if the `endpoint` parameter isn't passed. An error will be raised if a function has already been registered for the endpoint.

The `methods` parameter defaults to `["GET"]`. `HEAD` is always added automatically, and `OPTIONS` is added automatically by default.

`view_func` does not necessarily need to be passed, but if the rule should participate in routing an endpoint name must be associated with a view function at some point with the `endpoint()` decorator.

```
app.add_url_rule("/", endpoint="index")
```

(continues on next page)

(continued from previous page)

```
@app.endpoint("index")
def index():
    ...
```

If `view_func` has a `required_methods` attribute, those methods are added to the passed and automatic methods. If it has a `provide_automatic_methods` attribute, it is used as the default if the parameter is not passed.

Parameters

- **`rule`** – The URL rule string.
- **`endpoint`** – The endpoint name to associate with the rule and view function. Used when routing and building URLs. Defaults to `view_func.__name__`.
- **`view_func`** – The view function to associate with the endpoint name.
- **`provide_automatic_options`** – Add the `OPTIONS` method and respond to `OPTIONS` requests automatically.
- **`options`** – Extra options passed to the `Rule` object.

`app_context``()` → flask.ctx.AppContext

Create an `AppContext`. Use as a `with` block to push the context, which will make `current_app` point at this application.

An application context is automatically pushed by `RequestContext.push()` when handling a request, and when running a CLI command. Use this to manually create a context outside of these situations.

```
with app.app_context():
    init_db()
```

See /appcontext.

New in version 0.9.

`app_ctx_globals_class`

alias of `flask.ctx._AppCtxGlobals`

`async_to_sync``(`*func: Callable[[...], Coroutine[T_co, T_contra, V_co]]*`)` → Callable[[...], Any]

Return a sync function that will run the coroutine function.

```
result = app.async_to_sync(func)(*args, **kwargs)
```

Override this method to change how the app converts async code to be synchronously callable.

New in version 2.0.

`auto_find_instance_path``()` → str

Tries to locate the instance path if it was not provided to the constructor of the application class. It will basically calculate the path to a folder named `instance` next to your main file or the package.

New in version 0.8.

`before_first_request``(`*f: Callable[[], None]*`)` → Callable[[], None]

Registers a function to be run before the first request to this instance of the application.

The function will be called without any arguments and its return value is ignored.

New in version 0.8.

before_first_request_funcs = None
A list of functions that will be called at the beginning of the first request to this instance. To register a function, use the `before_first_request()` decorator.

New in version 0.8.

blueprints = None
Maps registered blueprint names to blueprint objects. The dict retains the order the blueprints were registered in. Blueprints can be registered multiple times, this dict does not track how often they were attached.

New in version 0.7.

config = None
The configuration dictionary as `Config`. This behaves exactly like a regular dictionary but supports additional methods to load a config from files.

config_class
alias of `flask.config.Config`

create_global_jinja_loader() → flask.templating.DispatchingJinjaLoader
Creates the loader for the Jinja2 environment. Can be used to override just the loader and keeping the rest unchanged. It's discouraged to override this function. Instead one should override the `jinja_loader()` function instead.

The global loader dispatches between the loaders of the application and the individual blueprints.

New in version 0.7.

create_jinja_environment() → flask.templating.Environment
Create the Jinja environment based on `jinja_options` and the various Jinja-related methods of the app. Changing `jinja_options` after this will have no effect. Also adds Flask-related globals and filters to the environment.

Changed in version 0.11: `Environment.auto_reload` set in accordance with `TEMPLATES_AUTO_RELOAD` configuration option.

New in version 0.5.

create_url_adapter(*request: Optional[flask.wrappers.Request]*) → Optional[werkzeug.routing.MapAdapter]
Creates a URL adapter for the given request. The URL adapter is created at a point where the request context is not yet set up so the request is passed explicitly.

New in version 0.6.

Changed in version 0.9: This can now also be called without a request object when the URL adapter is created for the application context.

Changed in version 1.0: `SERVER_NAME` no longer implicitly enables subdomain matching. Use `subdomain_matching` instead.

debug
Whether debug mode is enabled. When using `flask run` to start the development server, an interactive debugger will be shown for unhandled exceptions, and the server will be reloaded when code changes. This maps to the `DEBUG` config key. This is enabled when `env` is `'development'` and is overridden by the `FLASK_DEBUG` environment variable. It may not behave as expected if set in code.

Do not enable debug mode when deploying in production.

Default: `True` if `env` is `'development'`, or `False` otherwise.

default_config = {'APPLICATION_ROOT': '/', 'DEBUG': None, 'ENV': None, 'EXPLAIN_TEMPLA
Default configuration parameters.

dispatch_request() → Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], Union[Headers, Dict[str, Union[str, List[str], Tuple[str, ...]]], List[Tuple[str, Union[str, List[str], Tuple[str, ...]]]]]], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], int], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], int, Union[Headers, Dict[str, Union[str, List[str], Tuple[str, ...]]], List[Tuple[str, Union[str, List[str], Tuple[str, ...]]]]]], WSGIApplication]

Does the request dispatching. Matches the URL and returns the return value of the view or error handler. This does not have to be a response object. In order to convert the return value to a proper response object, call `make_response()`.

Changed in version 0.7: This no longer does the exception handling, this code was moved to the new `full_dispatch_request()`.

do_teardown_appcontext(*exc: Optional[BaseException] = <object object>*) → None

Called right before the application context is popped.

When handling a request, the application context is popped after the request context. See `do_teardown_request()`.

This calls all functions decorated with `teardown_appcontext()`. Then the `appcontext_tearing_down` signal is sent.

This is called by `AppContext.pop()`.

New in version 0.9.

do_teardown_request(*exc: Optional[BaseException] = <object object>*) → None

Called after the request is dispatched and the response is returned, right before the request context is popped.

This calls all functions decorated with `teardown_request()`, and `Blueprint.teardown_request()` if a blueprint handled the request. Finally, the `request_tearing_down` signal is sent.

This is called by `RequestContext.pop()`, which may be delayed during testing to maintain access to resources.

Parameters **exc** – An unhandled exception raised while dispatching the request. Detected from the current exception information if not passed. Passed to each teardown function.

Changed in version 0.9: Added the `exc` argument.

ensure_sync(*func: Callable*) → Callable

Ensure that the function is synchronous for WSGI workers. Plain `def` functions are returned as-is. `async def` functions are wrapped to run and wait for the response.

Override this method to change how the app runs async views.

New in version 2.0.

env

What environment the app is running in. Flask and extensions may enable behaviors based on the environment, such as enabling debug mode. This maps to the `ENV` config key. This is set by the `FLASK_ENV` environment variable and may not behave as expected if set in code.

Do not enable development when deploying in production.

Default: `'production'`

extensions = None

a place where extensions can store application specific state. For example this is where an extension could store database engines and similar things.

The key must match the name of the extension module. For example in case of a "Flask-Foo" extension in *flask_foo*, the key would be `'foo'`.

New in version 0.7.

finalize_request(*rv: Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], Union[Headers, Dict[str, Union[str, List[str], Tuple[str, ...]]], List[Tuple[str, Union[str, List[str], Tuple[str, ...]]]]]], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], int], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], int, Union[Headers, Dict[str, Union[str, List[str], Tuple[str, ...]]], List[Tuple[str, Union[str, List[str], Tuple[str, ...]]]]]], WSGIApplication, werkzeug.exceptions.HTTPException], from_error_handler: bool = False*) → flask.wrappers.Response

Given the return value from a view function this finalizes the request by converting it into a response and invoking the postprocessing functions. This is invoked for both normal request dispatching as well as error handlers.

Because this means that it might be called as a result of a failure a special safe mode is available which can be enabled with the *from_error_handler* flag. If enabled, failures in response processing will be logged and otherwise ignored.

Internal

full_dispatch_request() → flask.wrappers.Response

Dispatches the request and on top of that performs request pre and postprocessing as well as HTTP exception catching and error handling.

New in version 0.7.

got_first_request

This attribute is set to `True` if the application started handling the first request.

New in version 0.8.

handle_exception(*e: Exception*) → flask.wrappers.Response

Handle an exception that did not have an error handler associated with it, or that was raised from an error handler. This always causes a 500 `InternalServerError`.

Always sends the `got_request_exception` signal.

If *`propagate_exceptions`* is `True`, such as in debug mode, the error will be re-raised so that the debugger can display it. Otherwise, the original exception is logged, and an `InternalServerError` is returned.

If an error handler is registered for `InternalServerError` or `500`, it will be used. For consistency, the handler will always receive the `InternalServerError`. The original unhandled exception is available as `e.original_exception`.

Changed in version 1.1.0: Always passes the `InternalServerError` instance to the handler, setting `original_exception` to the unhandled error.

Changed in version 1.1.0: `after_request` functions and other finalization is done even for the default 500 response when there is no handler.

New in version 0.3.

handle_http_exception(*e: werkzeug.exceptions.HTTPException*) → Union[werkzeug.exceptions.HTTPException, Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], Union[Headers, Dict[str, Union[str, List[str], Tuple[str, ...]]], List[Tuple[str, Union[str, List[str], Tuple[str, ...]]]]]], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], int], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], int, Union[Headers, Dict[str, Union[str, List[str], Tuple[str, ...]]], List[Tuple[str, Union[str, List[str], Tuple[str, ...]]]]]], WSGIApplication]

Handles an HTTP exception. By default this will invoke the registered error handlers and fall back to returning the exception as response.

Changed in version 1.0.3: `RoutingException`, used internally for actions such as slash redirects during routing, is not passed to error handlers.

Changed in version 1.0: Exceptions are looked up by code *and* by MRO, so `HTTPExcpetion` subclasses can be handled with a catch-all handler for the base `HTTPException`.

New in version 0.3.

handle_url_build_error(*error: Exception*, *endpoint: str*, *values: dict*) → str

Handle `BuildError` on `url_for()`.

handle_user_exception(*e: Exception*) → Union[werkzeug.exceptions.HTTPException, Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], Union[Headers, Dict[str, Union[str, List[str], Tuple[str, ...]]], List[Tuple[str, Union[str, List[str], Tuple[str, ...]]]]]], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], int], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], int, Union[Headers, Dict[str, Union[str, List[str], Tuple[str, ...]]], List[Tuple[str, Union[str, List[str], Tuple[str, ...]]]]]], WSGIApplication]

This method is called whenever an exception occurs that should be handled. A special case is `HTTPException` which is forwarded to the *`handle_http_exception()`* method. This function will either return a response value or reraise the exception with the same traceback.

Changed in version 1.0: Key errors raised from request data like `form` show the bad key in debug mode rather than a generic bad request message.

New in version 0.7.

inject_url_defaults(*endpoint: str*, *values: dict*) → None

Injects the URL defaults for the given endpoint directly into the values dictionary passed. This is used internally and automatically called on URL building.

New in version 0.7.

instance_path = None

Holds the path to the instance folder.

New in version 0.8.

iter_blueprints() → ValuesView[Blueprint]

Iterates over all blueprints by the order they were registered.

New in version 0.11.

jinja_env
The Jinja environment used to load templates.

The environment is created the first time this property is accessed. Changing `jinja_options` after that will have no effect.

jinja_environment
alias of `flask.templating.Environment`

jinja_options = {}
Options that are passed to the Jinja environment in `create_jinja_environment()`. Changing these options after the environment is created (accessing `jinja_env`) will have no effect.

Changed in version 1.1.0: This is a `dict` instead of an `ImmutableDict` to allow easier configuration.

json_decoder
alias of `flask.json.JSONDecoder`

json_encoder
alias of `flask.json.JSONEncoder`

log_exception(*exc_info: Union[Tuple[type, BaseException, traceback], Tuple[None, None, None]]*) → None
Logs an exception. This is called by `handle_exception()` if debugging is disabled and right before the handler is called. The default implementation logs the exception as error on the `logger`.

New in version 0.8.

logger
A standard Python `Logger` for the app, with the same name as `name`.

In debug mode, the logger's `level` will be set to `DEBUG`.

If there are no handlers configured, a default handler will be added. See /logging for more information.

Changed in version 1.1.0: The logger takes the same name as `name` rather than hard-coding `"flask.app"`.

Changed in version 1.0.0: Behavior was simplified. The logger is always named `"flask.app"`. The level is only set during configuration, it doesn't check `app.debug` each time. Only one format is used, not different ones depending on `app.debug`. No handlers are removed, and a handler is only added if no handlers are already configured.

New in version 0.3.

make_config(*instance_relative: bool = False*) → flask.config.Config
Used to create the config attribute by the Flask constructor. The *instance_relative* parameter is passed in from the constructor of Flask (there named *instance_relative_config*) and indicates if the config should be relative to the instance path or the root path of the application.

New in version 0.8.

make_default_options_response() → flask.wrappers.Response
This method is called to create the default `OPTIONS` response. This can be changed through subclassing to change the default behavior of `OPTIONS` responses.

New in version 0.7.

make_response(*rv: Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], Union[Headers, Dict[str, Union[str, List[str], Tuple[str, ...]]], List[Tuple[str, Union[str, List[str], Tuple[str, ...]]]]]], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], int], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], int, Union[Headers, Dict[str, Union[str, List[str], Tuple[str, ...]]], List[Tuple[str, Union[str, List[str], Tuple[str, ...]]]]]], WSGIApplication]*) → flask.wrappers.Response

Convert the return value from a view function to an instance of *response_class*.

Parameters **rv** – the return value from the view function. The view function must return a response. Returning `None`, or the view ending without returning, is not allowed. The following types are allowed for `view_rv`:

str A response object is created with the string encoded to UTF-8 as the body.

bytes A response object is created with the bytes as the body.

dict A dictionary that will be jsonify'd before being returned.

tuple Either `(body, status, headers)`, `(body, status)`, or `(body, headers)`, where `body` is any of the other types allowed here, `status` is a string or an integer, and `headers` is a dictionary or a list of `(key, value)` tuples. If `body` is a *response_class* instance, `status` overwrites the exiting value and `headers` are extended.

response_class The object is returned unchanged.

other Response class The object is coerced to *response_class*.

callable() The function is called as a WSGI application. The result is used to create a response object.

Changed in version 0.9: Previously a tuple was interpreted as the arguments for the response object.

make_shell_context() → dict

Returns the shell context for an interactive shell for this application. This runs all the registered shell context processors.

New in version 0.11.

name

The name of the application. This is usually the import name with the difference that it's guessed from the run file if the import name is main. This name is used as a display name when Flask needs the name of the application. It can be set and overridden to change the value.

New in version 0.8.

open_instance_resource(*resource: str*, *mode: str = 'rb'*) → IO[AnyStr]

Opens a resource from the application's instance folder (*instance_path*). Otherwise works like `open_resource()`. Instance resources can also be opened for writing.

Parameters

- **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.
- **mode** – resource file opening mode, default is 'rb'.

permanent_session_lifetime

A `timedelta` which is used to set the expiration date of a permanent session. The default is 31 days which makes a permanent session survive for roughly one month.

This attribute can also be configured from the config with the `PERMANENT_SESSION_LIFETIME` configuration key. Defaults to `timedelta(days=31)`

`preprocess_request``()` → Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], Union[Headers, Dict[str, Union[str, List[str], Tuple[str, ...]]], List[Tuple[str, Union[str, List[str], Tuple[str, ...]]]]]], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], int], Tuple[Union[Response, AnyStr, Dict[str, Any], Generator[AnyStr, None, None]], int, Union[Headers, Dict[str, Union[str, List[str], Tuple[str, ...]]], List[Tuple[str, Union[str, List[str], Tuple[str, ...]]]]]], WSGIApplication, None]

Called before the request is dispatched. Calls `url_value_preprocessors` registered with the app and the current blueprint (if any). Then calls `before_request_funcs` registered with the app and the blueprint.

If any `before_request()` handler returns a non-None value, the value is handled as if it was the return value from the view, and further request handling is stopped.

`preserve_context_on_exception`

Returns the value of the `PRESERVE_CONTEXT_ON_EXCEPTION` configuration value in case it's set, otherwise a sensible default is returned.

New in version 0.7.

`process_response``(`*response: flask.wrappers.Response*`)` → flask.wrappers.Response

Can be overridden in order to modify the response object before it's sent to the WSGI server. By default this will call all the `after_request()` decorated functions.

Changed in version 0.5: As of Flask 0.5 the functions registered for after request execution are called in reverse order of registration.

Parameters **`response`** – a `response_class` object.

Returns a new response object or the same, has to be an instance of `response_class`.

`propagate_exceptions`

Returns the value of the `PROPAGATE_EXCEPTIONS` configuration value in case it's set, otherwise a sensible default is returned.

New in version 0.7.

`raise_routing_exception``(`*request: flask.wrappers.Request*`)` → te.NoReturn

Exceptions that are recording during routing are reraised with this method. During debug we are not reraising redirect requests for non `GET`, `HEAD`, or `OPTIONS` requests and we're raising a different error instead to help debug situations.

Internal

`register_blueprint``(`*blueprint: Blueprint*, ***options*`)` → None

Register a `Blueprint` on the application. Keyword arguments passed to this method will override the defaults set on the blueprint.

Calls the blueprint's `register()` method after recording the blueprint in the application's `blueprints`.

Parameters

- **`blueprint`** – The blueprint to register.
- **`url_prefix`** – Blueprint routes will be prefixed with this.
- **`subdomain`** – Blueprint routes will match on this subdomain.

- **url_defaults** – Blueprint routes will use these default values for view arguments.
- **options** – Additional keyword arguments are passed to `BlueprintSetupState`. They can be accessed in `record()` callbacks.

Changed in version 2.0.1: The `name` option can be used to change the (pre-dotted) name the blueprint is registered with. This allows the same blueprint to be registered multiple times with unique names for `url_for`.

New in version 0.7.

request_class
alias of `flask.wrappers.Request`

request_context(*environ: dict*) → flask.ctx.RequestContext
Create a `RequestContext` representing a WSGI environment. Use a `with` block to push the context, which will make `request` point at this request.

See /reqcontext.

Typically you should not call this from your own code. A request context is automatically pushed by the *`wsgi_app()`* when handling a request. Use *`test_request_context()`* to create an environment and context instead of this method.

Parameters **environ** – a WSGI environment

response_class
alias of `flask.wrappers.Response`

run(*host: Optional[str] = None*, *port: Optional[int] = None*, *debug: Optional[bool] = None*, *load_dotenv: bool = True*, ***options*) → None
Runs the application on a local development server.

Do not use `run()` in a production setting. It is not intended to meet security and performance requirements for a production server. Instead, see /deploying/index for WSGI server recommendations.

If the *`debug`* flag is set the server will automatically reload for code changes and show a debugger in case an exception happened.

If you want to run the application in debug mode, but disable the code execution on the interactive debugger, you can pass `use_evalex=False` as parameter. This will keep the debugger's traceback screen active, but disable code execution.

It is not recommended to use this function for development with automatic reloading as this is badly supported. Instead you should be using the **flask** command line script's `run` support.

Keep in Mind

Flask will suppress any server error with a generic error page unless it is in debug mode. As such to enable just the interactive debugger without the code reloading, you have to invoke *`run()`* with `debug=True` and `use_reloader=False`. Setting `use_debugger` to `True` without being in debug mode won't catch any exceptions because there won't be any to catch.

Parameters

- **host** – the hostname to listen on. Set this to `'0.0.0.0'` to have the server available externally as well. Defaults to `'127.0.0.1'` or the host in the `SERVER_NAME` config variable if present.
- **port** – the port of the webserver. Defaults to `5000` or the port defined in the `SERVER_NAME` config variable if present.

- **debug** – if given, enable or disable debug mode. See *debug*.
- **load_dotenv** – Load the nearest `.env` and `.flaskenv` files to set environment variables. Will also change the working directory to the directory containing the first file found.
- **options** – the options to be forwarded to the underlying Werkzeug server. See `werkzeug.serving.run_simple()` for more information.

Changed in version 1.0: If installed, python-dotenv will be used to load environment variables from `.env` and `.flaskenv` files.

If set, the `FLASK_ENV` and `FLASK_DEBUG` environment variables will override *env* and *debug*.

Threaded mode is enabled by default.

Changed in version 0.10: The default port is now picked from the `SERVER_NAME` variable.

secret_key

If a secret key is set, cryptographic components can use this to sign cookies and other things. Set this to a complex random value when you want to use the secure cookie for instance.

This attribute can also be configured from the config with the `SECRET_KEY` configuration key. Defaults to `None`.

select_jinja_autoescape (*filename: str*) → bool

Returns `True` if autoescaping should be active for the given template name. If no template name is given, returns *True*.

New in version 0.5.

send_file_max_age_default

A `timedelta` or number of seconds which is used as the default `max_age` for `send_file()`. The default is `None`, which tells the browser to use conditional requests instead of a timed cache.

Configured with the `SEND_FILE_MAX_AGE_DEFAULT` configuration key.

Changed in version 2.0: Defaults to `None` instead of 12 hours.

session_cookie_name

The secure cookie uses this for the name of the session cookie.

This attribute can also be configured from the config with the `SESSION_COOKIE_NAME` configuration key. Defaults to `'session'`

session_interface = <flask.sessions.SecureCookieSessionInterface object>

the session interface to use. By default an instance of `SecureCookieSessionInterface` is used here.

New in version 0.8.

shell_context_processor (*f: Callable*) → Callable

Registers a shell context processor function.

New in version 0.11.

shell_context_processors = None

A list of shell context processor functions that should be run when a shell context is created.

New in version 0.11.

should_ignore_error (*error: Optional[BaseException]*) → bool

This is called to figure out if an error should be ignored or not as far as the teardown system is concerned. If this function returns `True` then the teardown handlers will not be passed the error.

New in version 0.10.

teardown_appcontext(*f: Callable[[Optional[BaseException]], Response]*) → Callable[[Optional[BaseException]], flask.wrappers.Response]

Registers a function to be called when the application context ends. These functions are typically also called when the request context is popped.

Example:

```
ctx = app.app_context()
ctx.push()
...
ctx.pop()
```

When `ctx.pop()` is executed in the above example, the teardown functions are called just before the app context moves from the stack of active contexts. This becomes relevant if you are using such constructs in tests.

Since a request context typically also manages an application context it would also be called when you pop a request context.

When a teardown function was called because of an unhandled exception it will be passed an error object. If an `errorhandler()` is registered, it will handle the exception and the teardown will not receive it.

The return values of teardown functions are ignored.

New in version 0.9.

teardown_appcontext_funcs = None

A list of functions that are called when the application context is destroyed. Since the application context is also torn down if the request ends this is the place to store code that disconnects from databases.

New in version 0.9.

template_filter(*name: Optional[str] = None*) → Callable[[Callable[[Any], str]], Callable[[Any], str]]

A decorator that is used to register custom template filter. You can specify a name for the filter, otherwise the function name will be used. Example:

```
@app.template_filter()
def reverse(s):
    return s[::-1]
```

Parameters **name** – the optional name of the filter, otherwise the function name will be used.

template_global(*name: Optional[str] = None*) → Callable[[Callable[[], Any]], Callable[[], Any]]

A decorator that is used to register a custom template global function. You can specify a name for the global function, otherwise the function name will be used. Example:

```
@app.template_global()
def double(n):
    return 2 * n
```

New in version 0.10.

Parameters **name** – the optional name of the global function, otherwise the function name will be used.

template_test(*name: Optional[str] = None*) → Callable[[Callable[[Any], bool]], Callable[[Any], bool]]

A decorator that is used to register custom template test. You can specify a name for the test, otherwise the function name will be used. Example:

```
@app.template_test()
def is_prime(n):
    if n == 2:
        return True
    for i in range(2, int(math.ceil(math.sqrt(n))) + 1):
        if n % i == 0:
            return False
    return True
```

New in version 0.10.

Parameters **name** – the optional name of the test, otherwise the function name will be used.

templates_auto_reload

Reload templates when they are changed. Used by *create_jinja_environment()*.

This attribute can be configured with TEMPLATES_AUTO_RELOAD. If not set, it will be enabled in debug mode.

New in version 1.0: This property was added but the underlying config and behavior already existed.

test_cli_runner(***kwargs*) → FlaskCliRunner

Create a CLI runner for testing CLI commands. See Testing CLI Commands.

Returns an instance of *test_cli_runner_class*, by default FlaskCliRunner. The Flask app object is passed as the first argument.

New in version 1.0.

test_cli_runner_class = None

The CliRunner subclass, by default FlaskCliRunner that is used by *test_cli_runner()*. Its __init__ method should take a Flask app object as the first argument.

New in version 1.0.

test_client(*use_cookies: bool = True*, ***kwargs*) → FlaskClient

Creates a test client for this application. For information about unit testing head over to /testing.

Note that if you are testing for assertions or exceptions in your application code, you must set app.testing = True in order for the exceptions to propagate to the test client. Otherwise, the exception will be handled by the application (not visible to the test client) and the only indication of an AssertionError or other exception will be a 500 status code response to the test client. See the *testing* attribute. For example:

```
app.testing = True
client = app.test_client()
```

The test client can be used in a with block to defer the closing down of the context until the end of the with block. This is useful if you want to access the context locals for testing:

```
with app.test_client() as c:
    rv = c.get('/?vodka=42')
    assert request.args['vodka'] == '42'
```

Additionally, you may pass optional keyword arguments that will then be passed to the application's *test_client_class* constructor. For example:

```
from flask.testing import FlaskClient

class CustomClient(FlaskClient):
    def __init__(self, *args, **kwargs):
        self._authentication = kwargs.pop("authentication")
        super(CustomClient,self).__init__( *args, **kwargs)

app.test_client_class = CustomClient
client = app.test_client(authentication='Basic ....')
```

See `FlaskClient` for more information.

Changed in version 0.4: added support for `with` block usage for the client.

New in version 0.7: The *use_cookies* parameter was added as well as the ability to override the client to be used by setting the *test_client_class* attribute.

Changed in version 0.11: Added ***kwargs* to support passing additional keyword arguments to the constructor of *test_client_class*.

test_client_class = None
the test client that is used with when *test_client* is used.

New in version 0.7.

test_request_context(**args*, ***kwargs*) → flask.ctx.RequestContext
Create a `RequestContext` for a WSGI environment created from the given values. This is mostly useful during testing, where you may want to run a function that uses request data without dispatching a full request.

See /reqcontext.

Use a `with` block to push the context, which will make `request` point at the request for the created environment.

```
with test_request_context(...):
    generate_report()
```

When using the shell, it may be easier to push and pop the context manually to avoid indentation.

```
ctx = app.test_request_context(...)
ctx.push()
...
ctx.pop()
```

Takes the same arguments as Werkzeug's `EnvironBuilder`, with some defaults from the application. See the linked Werkzeug docs for most of the available arguments. Flask-specific behavior is listed here.

Parameters

- **path** – URL path being requested.
- **base_url** – Base URL where the app is being served, which `path` is relative to. If not given, built from `PREFERRED_URL_SCHEME`, `subdomain`, `SERVER_NAME`, and `APPLICATION_ROOT`.
- **subdomain** – Subdomain name to append to `SERVER_NAME`.
- **url_scheme** – Scheme to use instead of `PREFERRED_URL_SCHEME`.
- **data** – The request body, either as a string or a dict of form keys and values.

- **json** – If given, this is serialized as JSON and passed as `data`. Also defaults `content_type` to `application/json`.
- **args** – other positional arguments passed to `EnvironBuilder`.
- **kwargs** – other keyword arguments passed to `EnvironBuilder`.

testing

The testing flag. Set this to `True` to enable the test mode of Flask extensions (and in the future probably also Flask itself). For example this might activate test helpers that have an additional runtime cost which should not be enabled by default.

If this is enabled and PROPAGATE_EXCEPTIONS is not changed from the default it's implicitly enabled.

This attribute can also be configured from the config with the `TESTING` configuration key. Defaults to `False`.

trap_http_exception(*e: Exception*) → bool

Checks if an HTTP exception should be trapped or not. By default this will return `False` for all exceptions except for a bad request key error if `TRAP_BAD_REQUEST_ERRORS` is set to `True`. It also returns `True` if `TRAP_HTTP_EXCEPTIONS` is set to `True`.

This is called for all HTTP exceptions raised by a view function. If it returns `True` for any exception the error handler for this exception is not called and it shows up as regular exception in the traceback. This is helpful for debugging implicitly raised HTTP exceptions.

Changed in version 1.0: Bad request errors are not trapped by default in debug mode.

New in version 0.8.

try_trigger_before_first_request_functions() → None

Called before each request and will ensure that it triggers the *`before_first_request_funcs`* and only exactly once per application instance (which means process usually).

Internal

update_template_context(*context: dict*) → None

Update the template context with some commonly used variables. This injects request, session, config and g into the template context as well as everything template context processors want to inject. Note that the as of Flask 0.6, the original values in the context will not be overridden if a context processor decides to return a value with the same key.

Parameters **context** – the context as a dictionary that is updated in place to add extra variables.

url_build_error_handlers = None

A list of functions that are called when `url_for()` raises a `BuildError`. Each function registered here is called with *error*, *endpoint* and *values*. If a function returns `None` or raises a `BuildError` the next function is tried.

New in version 0.9.

url_map = None

The `Map` for this instance. You can use this to change the routing converters after the class was created but before any routes are connected. Example:

```python
from werkzeug.routing import BaseConverter

class ListConverter(BaseConverter):
    def to_python(self, value):
        return value.split(',')
    def to_url(self, values):
```

(continues on next page)

(continued from previous page)

```
        return ','.join(super(ListConverter, self).to_url(value)
                        for value in values)

app = Flask(__name__)
app.url_map.converters['list'] = ListConverter
```

url_map_class
alias of `werkzeug.routing.Map`

url_rule_class
alias of `werkzeug.routing.Rule`

use_x_sendfile
Enable this if you want to use the X-Sendfile feature. Keep in mind that the server has to support this. This only affects files sent with the `send_file()` method.

New in version 0.2.

This attribute can also be configured from the config with the `USE_X_SENDFILE` configuration key. Defaults to `False`.

wsgi_app(*environ: dict*, *start_response: Callable*) → Any
The actual WSGI application. This is not implemented in `__call__()` so that middlewares can be applied without losing a reference to the app object. Instead of doing this:

```
app = MyMiddleware(app)
```

It's a better idea to do this instead:

```
app.wsgi_app = MyMiddleware(app.wsgi_app)
```

Then you still have the original application object around and can continue to call methods on it.

Changed in version 0.7: Teardown events for the request and app contexts are called even if an unhandled error occurs. Other events may not be called depending on when an error occurs during dispatch. See Callbacks and Errors.

Parameters

- **environ** – A WSGI environment.
- **start_response** – A callable accepting a status code, a list of headers, and an optional exception context to start the response.

`coaster.app.`**init_app**(*app*, *init_logging=True*)
Configure an app depending on the environment. Loads settings from a file named `settings.py` in the instance folder, followed by additional settings from one of `development.py`, `production.py` or `testing.py`. Typical usage:

```
from flask import Flask
import coaster.app

app = Flask(__name__, instance_relative_config=True)
coaster.app.init_app(app)   # Guess environment automatically
```

`init_app()` also configures logging by calling *`coaster.logger.init_app()`*.

Parameters

- **app** – App to be configured

- **init_logging** (*bool*) – Call *coaster.logger.init_app* (default *True*)

1.3 Logger

Coaster can help your application log errors at run-time. Initialize with *coaster.logger.init_app()*. If you use *coaster.app.init_app()*, this is done automatically for you.

class coaster.logger.**FilteredValueIndicator**

Represent a filtered value.

class coaster.logger.**LocalVarFormatter**(**args*, ***kwargs*)

Log the contents of local variables in the stack frame.

format(*record*)

Format the specified record as text.

Overrides `logging.Formatter.format()` to remove cache of `record.exc_text` unless it was produced by this formatter.

formatException(*ei*) → str

Render a stack trace with local variables in each stack frame.

class coaster.logger.**RepeatValueIndicator**(*key*)

Represent a repeating value.

class coaster.logger.**SlackHandler**(*app_name*, *webhooks*)

Custom logging handler to post error reports to Slack.

emit(*record*)

Emit an event.

class coaster.logger.**TelegramHandler**(*app_name*, *chatid*, *apikey*)

Custom logging handler to report errors to a Telegram chat.

emit(*record*)

Emit an event.

coaster.logger.**configure**(*app*)

Enable logging for an app using *LocalVarFormatter*.

Requires the app to be configured and checks for the following configuration parameters. All are optional:

- `LOGFILE`: Name of the file to log to (default `error.log`)
- `LOGFILE_LEVEL`: Logging level to use for file logger (default *WARNING*)
- `ADMINS`: List of email addresses of admins who will be mailed error reports
- **MAIL_DEFAULT_SENDER: From address of email. Can be an address or a tuple with** name and address
- `MAIL_SERVER`: SMTP server to send with (default `localhost`)
- `MAIL_USERNAME` and `MAIL_PASSWORD`: SMTP credentials, if required
- **SLACK_LOGGING_WEBHOOKS: If present, will send error logs to all specified** Slack webhooks
- **TELEGRAM_ERROR_CHATID and TELEGRAM_ERROR_APIKEY: If present, will use the** specified API key to post a message to the specified chat

Format for `SLACK_LOGGING_WEBHOOKS`:

```
SLACK_LOGGING_WEBHOOKS = [{
    'levelnames': ['WARNING', 'ERROR', 'CRITICAL'],
    'url': 'https://hooks.slack.com/...'
    }]
```

`coaster.logger.`**`filtered_value`**`(`*key*, *value*`)`
Find and mask sensitive values based on key names.

`coaster.logger.`**`init_app`**`(`*app*`)`
Enable logging for an app using *`LocalVarFormatter`*.

Requires the app to be configured and checks for the following configuration parameters. All are optional:

- `LOGFILE`: Name of the file to log to (default `error.log`)
- `LOGFILE_LEVEL`: Logging level to use for file logger (default *WARNING*)
- `ADMINS`: List of email addresses of admins who will be mailed error reports
- **`MAIL_DEFAULT_SENDER`: From address of email. Can be an address or a tuple with** name and address
- `MAIL_SERVER`: SMTP server to send with (default `localhost`)
- `MAIL_USERNAME` and `MAIL_PASSWORD`: SMTP credentials, if required
- **`SLACK_LOGGING_WEBHOOKS`: If present, will send error logs to all specified** Slack webhooks
- **`TELEGRAM_ERROR_CHATID` and `TELEGRAM_ERROR_APIKEY`: If present, will use the** specified API key to post a message to the specified chat

Format for `SLACK_LOGGING_WEBHOOKS`:

```
SLACK_LOGGING_WEBHOOKS = [{
    'levelnames': ['WARNING', 'ERROR', 'CRITICAL'],
    'url': 'https://hooks.slack.com/...'
    }]
```

`coaster.logger.`**`pprint_with_indent`**`(`*dictlike*, *outfile*, *indent=4*`)`
Filter values and pprint with indent to create a Markdown code block.

1.4 Assets

Coaster provides a simple asset management system for semantically versioned assets using the semantic_version and webassets libraries. Many popular libraries such as jQuery are not semantically versioned, so you will have to be careful about assumptions you make around them.

`class` `coaster.assets.`**`SimpleSpec`**`(`*expression*`)`

`class` `coaster.assets.`**`VersionedAssets`**
Semantic-versioned assets. To use, initialize a container for your assets:

```
from coaster.assets import VersionedAssets, Version
assets = VersionedAssets()
```

And then populate it with your assets. The simplest way is by specifying the asset name, version number, and path to the file (within your static folder):

```
assets['jquery.js'][Version('1.8.3')] = 'js/jquery-1.8.3.js'
```

You can also specify one or more *requirements* for an asset by supplying a list or tuple of requirements followed by the actual asset:

```
assets['jquery.form.js'][Version('2.96.0')] = (
    'jquery.js', 'js/jquery.form-2.96.js')
```

You may have an asset that provides replacement functionality for another asset:

```
assets['zepto.js'][Version('1.0.0-rc1')] = {
    'provides': 'jquery.js',
    'bundle': 'js/zepto-1.0rc1.js',
    }
```

Assets specified as a dictionary can have three keys:

Parameters

- **provides** (*string or list*) – Assets provided by this asset
- **requires** (*string or list*) – Assets required by this asset (with optional version specifications)
- **bundle** (*string or Bundle*) – The asset itself

To request an asset:

```
assets.require('jquery.js', 'jquery.form.js==2.96.0', ...)
```

This returns a webassets Bundle of the requested assets and their dependencies.

You can also ask for certain assets to not be included even if required if, for example, you are loading them from elsewhere such as a CDN. Prefix the asset name with '!':

```
assets.require('!jquery.js', 'jquery.form.js', ...)
```

To use these assets in a Flask app, register the assets with an environment:

```
from flask_assets import Environment
appassets = Environment(app)
appassets.register('js_all', assets.require('jquery.js', ...))
```

And include them in your master template:

```
{% assets "js_all" -%}
  <script type="text/javascript" src="{{ ASSET_URL }}"></script>
{%- endassets -%}
```

require (**namespecs*)
Return a bundle of the requested assets and their dependencies.

exception coaster.assets.**AssetNotFound**
No asset with this name

1.5 Utilities

These functions are not dependent on Flask. They implement common patterns in Flask-based applications.

1.6 Miscellaneous utilities

`coaster.utils.misc.`**`base_domain_matches`**`(`*d1*, *d2*`)`

Check if two domains have the same base domain, using the Public Suffix List.

```
>>> base_domain_matches('https://hasjob.co', 'hasjob.co')
True
>>> base_domain_matches('hasgeek.hasjob.co', 'hasjob.co')
True
>>> base_domain_matches('hasgeek.com', 'hasjob.co')
False
>>> base_domain_matches('static.hasgeek.co.in', 'hasgeek.com')
False
>>> base_domain_matches('static.hasgeek.co.in', 'hasgeek.co.in')
True
>>> base_domain_matches('example@example.com', 'example.com')
True
```

`coaster.utils.misc.`**`buid`**`()`

Legacy name

`coaster.utils.misc.`**`buid2uuid`**`(`*value*`)`

Legacy name

`coaster.utils.misc.`**`domain_namespace_match`**`(`*domain*, *namespace*`)`

Checks if namespace is related to the domain because the base domain matches.

```
>>> domain_namespace_match('hasgeek.com', 'com.hasgeek')
True
>>> domain_namespace_match('funnel.hasgeek.com', 'com.hasgeek.funnel')
True
>>> domain_namespace_match('app.hasgeek.com', 'com.hasgeek.peopleflow')
True
>>> domain_namespace_match('app.hasgeek.in', 'com.hasgeek.peopleflow')
False
>>> domain_namespace_match('peopleflow.local', 'local.peopleflow')
True
```

`coaster.utils.misc.`**`format_currency`**`(`*value*, *decimals=2*`)`

Return a number suitably formatted for display as currency, with thousands separated by commas and up to two decimal points.

```
>>> format_currency(1000)
'1,000'
>>> format_currency(100)
'100'
>>> format_currency(999.95)
'999.95'
>>> format_currency(99.95)
'99.95'
>>> format_currency(100000)
'100,000'
>>> format_currency(1000.00)
'1,000'
>>> format_currency(1000.41)
'1,000.41'
>>> format_currency(23.21, decimals=3)
```

(continues on next page)

(continued from previous page)

```
'23.210'
>>> format_currency(1000, decimals=3)
'1,000'
>>> format_currency(123456789.123456789)
'123,456,789.12'
```

`coaster.utils.misc.`**`get_email_domain`**`(`*emailaddr*`)`

Return the domain component of an email address. Returns None if the provided string cannot be parsed as an email address.

```
>>> get_email_domain('test@example.com')
'example.com'
>>> get_email_domain('test+trailing@example.com')
'example.com'
>>> get_email_domain('Example Address <test@example.com>')
'example.com'
>>> get_email_domain('foobar')
>>> get_email_domain('foobar@')
>>> get_email_domain('@foobar')
```

`coaster.utils.misc.`**`getbool`**`(`*value*`)`

Returns a boolean from any of a range of values. Returns None for unrecognized values. Numbers other than 0 and 1 are considered unrecognized.

```
>>> getbool(True)
True
>>> getbool(1)
True
>>> getbool('1')
True
>>> getbool('t')
True
>>> getbool(2)
>>> getbool(0)
False
>>> getbool(False)
False
>>> getbool('n')
False
```

`coaster.utils.misc.`**`is_collection`**`(`*item*`)`

Returns True if the item is a collection class: list, tuple, set, frozenset or any other class that resembles one of these (using abstract base classes).

```
>>> is_collection(0)
False
>>> is_collection(0.1)
False
>>> is_collection('')
False
>>> is_collection(b'')
False
>>> is_collection({})
False
>>> is_collection({}.keys())
True
```

(continues on next page)

(continued from previous page)

```
>>> is_collection([])
True
>>> is_collection(())
True
>>> is_collection(set())
True
>>> is_collection(frozenset())
True
>>> from coaster.utils import InspectableSet
>>> is_collection(InspectableSet({1, 2}))
True
```

`coaster.utils.misc.`**`make_name`**(*text*, *delim='-'*, *maxlength=50*, *checkused=None*, *counter=2*)

Generate an ASCII name slug. If a checkused filter is provided, it will be called with the candidate. If it returns True, make_name will add counter numbers starting from 2 until a suitable candidate is found.

Parameters

- **`delim`** (*`string`*) – Delimiter between words, default '-'
- **`maxlength`** (*`int`*) – Maximum length of name, default 50
- **`checkused`** – Function to check if a generated name is available for use
- **`counter`** (*`int`*) – Starting position for name counter

```
>>> make_name('This is a title')
'this-is-a-title'
>>> make_name('Invalid URL/slug here')
'invalid-url-slug-here'
>>> make_name('this.that')
'this-that'
>>> make_name('this:that')
'this-that'
>>> make_name("How 'bout this?")
'how-bout-this'
>>> make_name("How’s that?")
'hows-that'
>>> make_name('K & D')
'k-d'
>>> make_name('billion+ pageviews')
'billion-pageviews'
>>> make_name(' slug!')
'hindii-slug'
>>> make_name('Talk in español, Kiswahili,  and  too.', maxlength=250)
'talk-in-espanol-kiswahili-guang-zhou-hua-and-asmiiyaa-too'
>>> make_name('__name__', delim='_')
'name'
>>> make_name('how_about_this', delim='_')
'how_about_this'
>>> make_name('and-that', delim='_')
'and_that'
>>> make_name('Umlauts in Mötörhead')
'umlauts-in-motorhead'
>>> make_name('Candidate', checkused=lambda c: c in ['candidate'])
'candidate2'
>>> make_name('Candidate', checkused=lambda c: c in ['candidate'], counter=1)
'candidate1'
```

(continues on next page)

(continued from previous page)

```
>>> make_name('Candidate',
...   checkused=lambda c: c in ['candidate', 'candidate1', 'candidate2'],␣
↪counter=1)
'candidate3'
>>> make_name('Long title, but snipped', maxlength=20)
'long-title-but-snipp'
>>> len(make_name('Long title, but snipped', maxlength=20))
20
>>> make_name('Long candidate', maxlength=10,
...   checkused=lambda c: c in ['long-candi', 'long-cand1'])
'long-cand2'
>>> make_name('Lnkran')
'lankaran'
>>> make_name('example@example.com')
'example-example-com'
>>> make_name('trailing-delimiter', maxlength=10)
'trailing-d'
>>> make_name('trailing-delimiter', maxlength=9)
'trailing'
>>> make_name('''test this
... newline''')
'test-this-newline'
>>> make_name("testing an emoji")
'testing-an-emoji'
>>> make_name('''testing\t\nmore\r\nslashes''')
'testing-more-slashes'
>>> make_name('What if a HTML <tag/>')
'what-if-a-html-tag'
>>> make_name('These are equivalent to \x01 through \x1A')
'these-are-equivalent-to-through'
>>> make_name("feedback;\x00")
'feedback'
```

coaster.utils.misc.**md5sum**(*data*)

Return md5sum of data as a 32-character string.

```
>>> md5sum('random text')
'd9b9bec3f4cc5482e7c5ef43143e563a'
>>> md5sum('random text')
'd9b9bec3f4cc5482e7c5ef43143e563a'
>>> len(md5sum('random text'))
32
```

coaster.utils.misc.**namespace_from_url**(*url*)

Construct a dotted namespace string from a URL.

coaster.utils.misc.**nary_op**(*f*, *doc=None*)

Decorator to convert a binary operator into a chained n-ary operator.

coaster.utils.misc.**newpin**(*digits=4*)

Return a random numeric string with the specified number of digits, default 4.

```
>>> len(newpin())
4
>>> len(newpin(5))
5
>>> newpin().isdigit()
```

(continues on next page)

(continued from previous page)

```
True
```

`coaster.utils.misc.`**`newsecret`**`()`

Make a secret key for non-cryptographic use cases like email account verification. Mashes two UUID4s into a Base58 rendering, between 42 and 44 characters long. The resulting string consists of only ASCII strings and so will typically not be word-wrapped by email clients.

```
>>> len(newsecret()) in (42, 43, 44)
True
>>> newsecret() == newsecret()
False
```

`coaster.utils.misc.`**`nullint`**`(`*value*`)`

Return int(value) if bool(value) is not False. Return None otherwise. Useful for coercing optional values to an integer.

```
>>> nullint('10')
10
>>> nullint('') is None
True
```

`coaster.utils.misc.`**`nullstr`**`(`*value*`)`

Return unicode(value) if bool(value) is not False. Return None otherwise. Useful for coercing optional values to a string.

```
>>> nullstr(10) == '10'
True
>>> nullstr('') is None
True
```

`coaster.utils.misc.`**`require_one_of`**`(`*_return=False*, ***kwargs*`)`

Validator that raises `TypeError` unless one and only one parameter is not `None`. Use this inside functions that take multiple parameters, but allow only one of them to be specified:

```
def my_func(this=None, that=None, other=None):
    # Require one and only one of `this` or `that`
    require_one_of(this=this, that=that)

    # If we need to know which parameter was passed in:
    param, value = require_one_of(True, this=this, that=that)

    # Carry on with function logic
    pass
```

Parameters

- **`_return`** – Return the matching parameter
- **`kwargs`** – Parameters, of which one and only one is mandatory

Returns If *_return*, matching parameter name and value

Return type tuple

Raises **`TypeError`** – If the count of parameters that aren't `None` is not 1 `coaster.utils.misc.`**`unicode_http_header`**`(`*value*`)`

Convert an ASCII HTTP header string into a unicode string with the appropriate encoding applied. Expects headers to be RFC 2047 compliant.

```
>>> unicode_http_header('=?iso-8859-1?q?p=F6stal?=') == 'p\xf6stal'
True
>>> unicode_http_header(b'=?iso-8859-1?q?p=F6stal?=') == 'p\xf6stal'
True
>>> unicode_http_header('p\xf6stal') == 'p\xf6stal'
True
```

`coaster.utils.misc.`**`uuid1mc`**`()`

Return a UUID1 with a random multicast MAC id.

```
>>> isinstance(uuid1mc(), uuid.UUID)
True
```

`coaster.utils.misc.`**`uuid1mc_from_datetime`**`(`*dt*`)`

Return a UUID1 with a random multicast MAC id and with a timestamp matching the given datetime object or timestamp value.

Warning: This function does not consider the timezone, and is not guaranteed to return a unique UUID. Use under controlled conditions only.

```
>>> dt = datetime.now()
>>> u1 = uuid1mc()
>>> u2 = uuid1mc_from_datetime(dt)
>>> # Both timestamps should be very close to each other but not an exact match
>>> u1.time > u2.time
True
>>> u1.time - u2.time < 5000
True
>>> d2 = datetime.fromtimestamp((u2.time - 0x01b21dd213814000) * 100 / 1e9)
>>> d2 == dt
True
```

`coaster.utils.misc.`**`uuid2buid`**`(`*value*`)`

Legacy name

`coaster.utils.misc.`**`uuid_b58`**`()`

Return a UUID4 encoded in base58 and rendered as a string. Will be 21 or 22 characters long

```
>>> len(uuid_b58()) in (21, 22)
True
>>> uuid_b58() == uuid_b58()
False
>>> isinstance(uuid_b58(), str)
True
```

`coaster.utils.misc.`**`uuid_b64`**`()`

Return a new random id that is exactly 22 characters long, by encoding a UUID4 in URL-safe Base64. See http://en.wikipedia.org/wiki/Base64#Variants_summary_table

```
>>> len(buid())
22
```

(continues on next page)

(continued from previous page)

```
>>> buid() == buid()
False
>>> isinstance(buid(), str)
True
```

`coaster.utils.misc.`**`uuid_from_base58`**(*value*)

Convert a Base58-encoded UUID back into a UUID object

```
>>> uuid_from_base58('7KAmj837MyuJWUYPwtqAfz')
UUID('33203dd2-f2ef-422f-aeb0-058d6f5f7089')
>>> # The following UUID to Base58 encoding is from NPM uuid-base58, for␣
↪comparison
>>> uuid_from_base58('TedLUruK7MosG1Z88urTkk')
UUID('d7ce8475-e77c-43b0-9dde-56b428981999')
```

`coaster.utils.misc.`**`uuid_from_base64`**(*value*)

Convert a 22-char URL-safe Base64 string (BUID) to a UUID object

```
>>> uuid_from_base64('MyA90vLvQi-usAWNb19wiQ')
UUID('33203dd2-f2ef-422f-aeb0-058d6f5f7089')
```

`coaster.utils.misc.`**`uuid_to_base58`**(*value*)

Render a UUID in Base58 and return as a string

```
>>> uuid_to_base58(uuid.UUID('33203dd2-f2ef-422f-aeb0-058d6f5f7089'))
'7KAmj837MyuJWUYPwtqAfz'
>>> # The following UUID to Base58 encoding is from NPM uuid-base58, for␣
↪comparison
>>> uuid_to_base58(uuid.UUID('d7ce8475-e77c-43b0-9dde-56b428981999'))
'TedLUruK7MosG1Z88urTkk'
```

`coaster.utils.misc.`**`uuid_to_base64`**(*value*)

Convert a UUID object to a 22-char URL-safe Base64 string (BUID)

```
>>> uuid_to_base64(uuid.UUID('33203dd2-f2ef-422f-aeb0-058d6f5f7089'))
'MyA90vLvQi-usAWNb19wiQ'
```

`coaster.utils.misc.`**`valid_username`**(*candidate*)

Check if a username is valid.

```
>>> valid_username('example person')
False
>>> valid_username('example_person')
False
>>> valid_username('exampleperson')
True
>>> valid_username('example-person')
True
>>> valid_username('a')
True
>>> (valid_username('a-') or valid_username('ab-') or valid_username('-a') or
...   valid_username('-ab'))
False
```

1.7 Date, time and timezone utilities

`coaster.utils.datetime.`**`utcnow`**`()`

Returns the current time at UTC with *tzinfo* set

`coaster.utils.datetime.`**`parse_isoformat`**`(`*text*, *naive=True*, *delimiter='T'*`)`

Attempts to parse an ISO 8601 timestamp as generated by *datetime.isoformat()*. Timestamps without a timezone are assumed to be at UTC. Raises `ParseError` if the timestamp cannot be parsed.

Parameters **`naive`** (`bool`) – If *True*, strips timezone and returns datetime at UTC.

`coaster.utils.datetime.`**`isoweek_datetime`**`(`*year*, *week*, *timezone='UTC'*, *naive=False*`)`

Returns a datetime matching the starting point of a specified ISO week in the specified timezone (default UTC). Returns a naive datetime in UTC if requested (default False).

```
>>> isoweek_datetime(2017, 1)
datetime.datetime(2017, 1, 2, 0, 0, tzinfo=<UTC>)
>>> isoweek_datetime(2017, 1, 'Asia/Kolkata')
datetime.datetime(2017, 1, 1, 18, 30, tzinfo=<UTC>)
>>> isoweek_datetime(2017, 1, 'Asia/Kolkata', naive=True)
datetime.datetime(2017, 1, 1, 18, 30)
>>> isoweek_datetime(2008, 1, 'Asia/Kolkata')
datetime.datetime(2007, 12, 30, 18, 30, tzinfo=<UTC>)
```

`coaster.utils.datetime.`**`midnight_to_utc`**`(`*dt*, *timezone=None*, *naive=False*`)`

Returns a UTC datetime matching the midnight for the given date or datetime.

```
>>> from datetime import date
>>> midnight_to_utc(datetime(2017, 1, 1))
datetime.datetime(2017, 1, 1, 0, 0, tzinfo=<UTC>)
>>> midnight_to_utc(pytz.timezone('Asia/Kolkata').localize(datetime(2017, 1, 1)))
datetime.datetime(2016, 12, 31, 18, 30, tzinfo=<UTC>)
>>> midnight_to_utc(datetime(2017, 1, 1), naive=True)
datetime.datetime(2017, 1, 1, 0, 0)
>>> midnight_to_utc(pytz.timezone('Asia/Kolkata').localize(datetime(2017, 1, 1)),
...   naive=True)
datetime.datetime(2016, 12, 31, 18, 30)
>>> midnight_to_utc(date(2017, 1, 1))
datetime.datetime(2017, 1, 1, 0, 0, tzinfo=<UTC>)
>>> midnight_to_utc(date(2017, 1, 1), naive=True)
datetime.datetime(2017, 1, 1, 0, 0)
>>> midnight_to_utc(date(2017, 1, 1), timezone='Asia/Kolkata')
datetime.datetime(2016, 12, 31, 18, 30, tzinfo=<UTC>)
>>> midnight_to_utc(datetime(2017, 1, 1), timezone='Asia/Kolkata')
datetime.datetime(2016, 12, 31, 18, 30, tzinfo=<UTC>)
>>> midnight_to_utc(pytz.timezone('Asia/Kolkata').localize(datetime(2017, 1, 1)),
...   timezone='UTC')
datetime.datetime(2017, 1, 1, 0, 0, tzinfo=<UTC>)
```

`coaster.utils.datetime.`**`sorted_timezones`**`()`

Return a list of timezones sorted by offset from UTC.

`coaster.utils.datetime.`**`ParseError`**

alias of `aniso8601.exceptions.ISOFormatError`

1.8 Text processing utilities

coaster.utils.text.**compress_whitespace**(*text*)
Reduce all space-like characters into single spaces and strip from ends.

coaster.utils.text.**deobfuscate_email**(*text*)
Deobfuscate email addresses in provided text.

coaster.utils.text.**normalize_spaces**(*text*)
Replace whitespace characters with regular spaces.

coaster.utils.text.**normalize_spaces_multiline**(*text*)
Replace whitespace characters with regular spaces, in multiline text.

Line break characters like newlines are not considered whitespace.

coaster.utils.text.**sanitize_html**(*value*, *valid_tags=None*, *strip=True*, *linkify=False*)
Strip unwanted markup out of HTML.

coaster.utils.text.**simplify_text**(*text*)
Simplify text to allow comparison.

```
>>> simplify_text("Awesome Coder wanted at Awesome Company")
'awesome coder wanted at awesome company'
>>> simplify_text("Awesome Coder, wanted  at Awesome Company! ")
'awesome coder wanted at awesome company'
>>> simplify_text("Awesome Coder, wanted  at Awesome Company! ") == (
...   'awesome coder wanted at awesome company')
True
```

coaster.utils.text.**text_blocks**(*html_text*, *skip_pre=True*)
Extracts a list of paragraphs from a given HTML string.

coaster.utils.text.**ulstrip**(*text*)
Strip Unicode extended whitespace from the left side of a string.

coaster.utils.text.**urstrip**(*text*)
Strip Unicode extended whitespace from the right side of a string.

coaster.utils.text.**ustrip**(*text*)
Strip Unicode extended whitespace from a string.

1.9 Markdown processor

Markdown parser with a number of sane defaults that resembles GitHub-Flavoured Markdown (GFM).

GFM exists because normal markdown has some vicious gotchas. Further reading: http://blog.stackoverflow.com/2009/10/markdown-one-year-later/

This Markdown processor is used by `MarkdownColumn()` to auto-render HTML from Markdown text.

coaster.utils.markdown.**markdown**(*text: Optional[str], html: bool = False, linkify: bool = True, valid_tags: Union[List[str], Mapping[str, List[T]], None] = None, extensions: Optional[List[Union[str, markdown.extensions.Extension]]] = None, extension_configs: Optional[Mapping[str, Mapping[str, Any]]] = None*) → Optional[markupsafe.Markup]
Markdown parser with a number of sane defaults that resemble GFM.

Parameters

- **html** (*bool*) – Allow known-safe HTML tags in text (this disables code syntax highlighting and task lists)
- **linkify** (*bool*) – Whether to convert naked URLs into links
- **valid_tags** (*dict*) – Valid tags and attributes if HTML is allowed
- **extensions** (*list*) – List of Markdown extensions to be enabled
- **extension_configs** (*dict*) – Config for Markdown extensions

1.10 PostgreSQL query processor

`coaster.utils.tsquery.`**`for_tsquery`**`(`*text*`)`

Tokenize text into a valid PostgreSQL to_tsquery query.

```
>>> for_tsquery(" ")
''
>>> for_tsquery("This is a test")
"'This is a test'"
>>> for_tsquery('Match "this AND phrase"')
"'Match this'&'phrase'"
>>> for_tsquery('Match "this & phrase"')
"'Match this'&'phrase'"
>>> for_tsquery("This NOT that")
"'This'&!'that'"
>>> for_tsquery("This & NOT that")
"'This'&!'that'"
>>> for_tsquery("This > that")
"'This > that'"
>>> for_tsquery("Ruby AND (Python OR JavaScript)")
"'Ruby'&('Python'|'JavaScript')"
>>> for_tsquery("Ruby AND NOT (Python OR JavaScript)")
"'Ruby'&!('Python'|'JavaScript')"
>>> for_tsquery("Ruby NOT (Python OR JavaScript)")
"'Ruby'&!('Python'|'JavaScript')"
>>> for_tsquery("Ruby (Python OR JavaScript) Golang")
"'Ruby'&('Python'|'JavaScript')&'Golang'"
>>> for_tsquery("Ruby (Python OR JavaScript) NOT Golang")
"'Ruby'&('Python'|'JavaScript')&!'Golang'"
>>> for_tsquery("Java*")
"'Java':*"
>>> for_tsquery("Java**")
"'Java':*"
>>> for_tsquery("Android || Python")
"'Android'|'Python'"
>>> for_tsquery("Missing (bracket")
"'Missing'&('bracket')"
>>> for_tsquery("Extra bracket)")
"('Extra bracket')"
>>> for_tsquery("Android (Python ())")
"'Android'&('Python')"
>>> for_tsquery("Android (Python !())")
"'Android'&('Python')"
>>> for_tsquery("()")
''
```

(continues on next page)

(continued from previous page)

```
>>> for_tsquery("(")
''
>>> for_tsquery("() Python")
"'Python'"
>>> for_tsquery("!() Python")
"'Python'"
>>> for_tsquery("*")
''
>>> for_tsquery("/etc/passwd\x00")
"'/etc/passwd'"
```

1.11 Utility classes

class coaster.utils.classes.**NameTitle**(*name*, *title*)

name
Alias for field number 0

title
Alias for field number 1

class coaster.utils.classes.**LabeledEnum**
Labeled enumerations. Declare an enumeration with values and labels (for use in UI):

```
>>> class MY_ENUM(LabeledEnum):
...     FIRST = (1, "First")
...     THIRD = (3, "Third")
...     SECOND = (2, "Second")
```

LabeledEnum will convert any attribute that is a 2-tuple into a value and label pair. Access values as direct attributes of the enumeration:

```
>>> MY_ENUM.FIRST
1
>>> MY_ENUM.SECOND
2
>>> MY_ENUM.THIRD
3
```

Access labels via dictionary lookup on the enumeration:

```
>>> MY_ENUM[MY_ENUM.FIRST]
'First'
>>> MY_ENUM[2]
'Second'
>>> MY_ENUM.get(3)
'Third'
>>> MY_ENUM.get(4) is None
True
```

Retrieve a full list of values and labels with `.items()`. Definition order is preserved in Python 3.x, but not in 2.x:

```
>>> sorted(MY_ENUM.items())
[(1, 'First'), (2, 'Second'), (3, 'Third')]
>>> sorted(MY_ENUM.keys())
[1, 2, 3]
>>> sorted(MY_ENUM.values())
['First', 'Second', 'Third']
```

However, if you really want ordering in Python 2.x, add an __order__ list. Anything not in it will default to Python's ordering:

```
>>> class RSVP(LabeledEnum):
...     RSVP_Y = ('Y', "Yes")
...     RSVP_N = ('N', "No")
...     RSVP_M = ('M', "Maybe")
...     RSVP_U = ('U', "Unknown")
...     RSVP_A = ('A', "Awaiting")
...     __order__ = (RSVP_Y, RSVP_N, RSVP_M, RSVP_A)

>>> RSVP.items()
[('Y', 'Yes'), ('N', 'No'), ('M', 'Maybe'), ('A', 'Awaiting'), ('U', 'Unknown')]
```

Three value tuples are assumed to be (value, name, title) and the name and title are converted into NameTitle(name, title):

```
>>> class NAME_ENUM(LabeledEnum):
...     FIRST = (1, 'first', "First")
...     THIRD = (3, 'third', "Third")
...     SECOND = (2, 'second', "Second")
...     __order__ = (FIRST, SECOND, THIRD)

>>> NAME_ENUM.FIRST
1
>>> NAME_ENUM[NAME_ENUM.FIRST]
NameTitle(name='first', title='First')
>>> NAME_ENUM[NAME_ENUM.SECOND].name
'second'
>>> NAME_ENUM[NAME_ENUM.THIRD].title
'Third'
```

To make it easier to use with forms and to hide the actual values, a list of (name, title) pairs is available:

```
>>> NAME_ENUM.nametitles()
[('first', 'First'), ('second', 'Second'), ('third', 'Third')]
```

Given a name, the value can be looked up:

```
>>> NAME_ENUM.value_for('first')
1
>>> NAME_ENUM.value_for('second')
2
```

Values can be grouped together using a set, for performing "in" operations. These do not have labels and cannot be accessed via dictionary access:

```
>>> class RSVP_EXTRA(LabeledEnum):
...     RSVP_Y = ('Y', "Yes")
...     RSVP_N = ('N', "No")
```

(continues on next page)

(continued from previous page)

```
...     RSVP_M = ('M', "Maybe")
...     RSVP_U = ('U', "Unknown")
...     RSVP_A = ('A', "Awaiting")
...     __order__ = (RSVP_Y, RSVP_N, RSVP_M, RSVP_U, RSVP_A)
...     UNCERTAIN = {RSVP_M, RSVP_U, 'A'}

>>> isinstance(RSVP_EXTRA.UNCERTAIN, set)
True
>>> sorted(RSVP_EXTRA.UNCERTAIN)
['A', 'M', 'U']
>>> 'N' in RSVP_EXTRA.UNCERTAIN
False
>>> 'M' in RSVP_EXTRA.UNCERTAIN
True
>>> RSVP_EXTRA.RSVP_U in RSVP_EXTRA.UNCERTAIN
True
```

Labels are stored internally in a dictionary named `__labels__`, mapping the value to the label. Symbol names are stored in `__names__`, mapping name to the value. The label dictionary will only contain values processed using the tuple syntax, which excludes grouped values, while the names dictionary will contain both, but will exclude anything else found in the class that could not be processed (use `__dict__` for everything):

```
>>> list(RSVP_EXTRA.__labels__.keys())
['Y', 'N', 'M', 'U', 'A']
>>> list(RSVP_EXTRA.__names__.keys())
['RSVP_Y', 'RSVP_N', 'RSVP_M', 'RSVP_U', 'RSVP_A', 'UNCERTAIN']
```

class `coaster.utils.classes.`**`InspectableSet`**(*members=()*)

Given a set, mimics a read-only dictionary where the items are keys and have a value of `True`, and any other key has a value of `False`. Also supports attribute access. Useful in templates to simplify membership inspection:

```
>>> myset = InspectableSet({'member', 'other'})
>>> 'member' in myset
True
>>> 'random' in myset
False
>>> myset.member
True
>>> myset.random
False
>>> myset['member']
True
>>> myset['random']
False
>>> joinset = myset | {'added'}
>>> isinstance(joinset, InspectableSet)
True
>>> joinset = joinset | InspectableSet({'inspectable'})
>>> isinstance(joinset, InspectableSet)
True
>>> 'member' in joinset
True
>>> 'other' in joinset
True
>>> 'added' in joinset
True
```

(continues on next page)

(continued from previous page)

```
>>> 'inspectable' in joinset
True
>>> emptyset = InspectableSet()
>>> len(emptyset)
0
```

class coaster.utils.classes.**classmethodproperty**(*func*)

Class method decorator to make class methods behave like properties:

```
>>> class Foo:
...     @classmethodproperty
...     def test(cls):
...         return repr(cls)
...
```

Works on classes:

```
>>> Foo.test
"<class 'coaster.utils.classes.Foo'>"
```

Works on class instances:

```
>>> Foo().test
"<class 'coaster.utils.classes.Foo'>"
```

Works on subclasses too:

```
>>> class Bar(Foo):
...     pass
...
>>> Bar.test
"<class 'coaster.utils.classes.Bar'>"
>>> Bar().test
"<class 'coaster.utils.classes.Bar'>"
```

Due to limitations in Python's descriptor API, `classmethodproperty` can block write and delete access on an instance…

```
>>> Foo().test = 'bar'
Traceback (most recent call last):
AttributeError: test is read-only
>>> del Foo().test
Traceback (most recent call last):
AttributeError: test is read-only
```

…but not on the class itself:

```
>>> Foo.test = 'bar'
>>> Foo.test
'bar'
```

1.12 Authentication management

Coaster provides a `current_auth` for handling authentication. Login managers must comply with its API for Coaster's view handlers to work.

If a login manager installs itself as `current_app.login_manager` and provides a `_load_user()` method, it will be called when *`current_auth`* is invoked for the first time in a request. Login managers can call *`add_auth_attribute()`* to load the actor (typically the authenticated user) and any other relevant authentication attributes.

For compatibility with Flask-Login, a user object loaded at `_request_ctx_stack.top.user` will be recognised and made available via *`current_auth`*.

`coaster.auth.`**`add_auth_attribute`**(*attr*, *value*, *actor=False*)

Helper function for login managers. Adds authorization attributes to *`current_auth`* for the duration of the request.

Parameters

- **`attr`** (*`str`*) – Name of the attribute
- **`value`** – Value of the attribute
- **`actor`** (*`bool`*) – Whether this attribute is an actor (user or client app accessing own data)

If the attribute is an actor and *`current_auth`* does not currently have an actor, the attribute is also made available as `current_auth.actor`, which in turn is used by `current_auth.is_authenticated`.

The attribute name `user` is special-cased:

1. `user` is always treated as an actor
2. `user` is also made available as `_request_ctx_stack.top.user` for compatibility with Flask-Login

`coaster.auth.`**`add_auth_anchor`**(*anchor*)

Helper function for login managers and view handlers to add a new auth anchor. This is a placeholder until anchors are properly specified.

`coaster.auth.`**`request_has_auth`**()

Helper function that returns True if *`current_auth`* was invoked during the current request. A login manager can use this during request teardown to set cookies or perform other housekeeping functions.

`coaster.auth.`**`current_auth = CurrentAuth(None)`**

A proxy object that hosts state for user authentication, attempting to load state from request context if not already loaded. Returns a `CurrentAuth`. Typical use:

```python
from coaster.auth import current_auth

@app.route('/')
def user_check():
    if current_auth.is_authenticated:
        return "We have a user"
    else:
        return "User not logged in"
```

1.13 View helpers

Coaster provides classes, functions and decorators for common scenarios in view handlers.

1.14 Miscellaneous view helpers

Helper functions for view handlers.

All items in this module can be imported directly from `coaster.views`.

`coaster.views.misc.`**`get_current_url`**`()`

Return the current URL including the query string as a relative path. If the app uses subdomains, return an absolute path

`coaster.views.misc.`**`get_next_url`**`(`*referrer=False*, *external=False*, *session=False*, *default=<object object>*`)`

Get the next URL to redirect to. Don't return external URLs unless explicitly asked for. This is to protect the site from being an unwitting redirector to external URLs. Subdomains are okay, however.

This function looks for a `next` parameter in the request or in the session (depending on whether parameter `session` is True). If no `next` is present, it checks the referrer (if enabled), and finally returns either the provided default (which can be any value including `None`) or the script root (typically `/`).

`coaster.views.misc.`**`jsonp`**`(`**args*, ***kw*`)`

Returns a JSON response with a callback wrapper, if asked for. Consider using CORS instead, as JSONP makes the client app insecure. See the `cors()` decorator.

`coaster.views.misc.`**`endpoint_for`**`(`*url*, *method=None*, *return_rule=False*, *follow_redirects=True*`)`

Given an absolute URL, retrieve the matching endpoint name (or rule) and view arguments. Requires a current request context to determine runtime environment.

Parameters

- **`method`** (`str`) – HTTP method to use (defaults to GET)
- **`return_rule`** (`bool`) – Return the URL rule instead of the endpoint name
- **`follow_redirects`** (`bool`) – Follow redirects to final endpoint

Returns Tuple of endpoint name or URL rule or *None*, view arguments

1.15 View decorators

Decorators for view handlers.

All items in this module can be imported directly from `coaster.views`.

`exception` `coaster.views.decorators.`**`RequestTypeError`**`(`*description: Optional[str] = None*, *response: Optional[Response] = None*`)`

Exception that combines TypeError with BadRequest. Used by `requestargs()`.

`exception` `coaster.views.decorators.`**`RequestValueError`**`(`*description: Optional[str] = None*, *response: Optional[Response] = None*`)`

Exception that combines ValueError with BadRequest. Used by `requestargs()`.

`coaster.views.decorators.`**`requestargs`**`(`**args*, ***config*`)`

Decorator that loads parameters from request.values if not specified in the function's keyword arguments. Usage:

```
@requestargs('param1', ('param2', int), 'param3[]', ...)
def function(param1, param2=0, param3=None):
    ...
```

requestargs takes a list of parameters to pass to the wrapped function, with an optional filter (useful to convert incoming string request data into integers and other common types). If a required parameter is missing and your function does not specify a default value, Python will raise TypeError. requestargs recasts this as `RequestTypeError`, which returns HTTP 400 Bad Request.

If the parameter name ends in `[]`, requestargs will attempt to read a list from the incoming data. Filters are applied to each member of the list, not to the whole list.

If the filter raises a ValueError, this is recast as a *RequestValueError*, which also returns HTTP 400 Bad Request.

Tests:

```
>>> from flask import Flask
>>> app = Flask(__name__)
>>>
>>> @requestargs('p1', ('p2', int), ('p3[]', int))
... def f(p1, p2=None, p3=None):
...     return p1, p2, p3
...
>>> f(p1=1)
(1, None, None)
>>> f(p1=1, p2=2)
(1, 2, None)
>>> f(p1='a', p2='b')
('a', 'b', None)
>>> with app.test_request_context('/?p2=2'):
...     f(p1='1')
...
('1', 2, None)
>>> with app.test_request_context('/?p3=1&p3=2'):
...     f(p1='1', p2='2')
...
('1', '2', [1, 2])
>>> with app.test_request_context('/?p2=100&p3=1&p3=2'):
...     f(p1='1', p2=200)
...
('1', 200, [1, 2])
```

`coaster.views.decorators.`**`requestform`**(**args*)

Like *requestargs()*, but loads from request.form (the form submission).

`coaster.views.decorators.`**`requestquery`**(**args*)

Like *requestargs()*, but loads from request.args (the query string).

`coaster.views.decorators.`**`load_model`**(*model, attributes=None, parameter=None, kwargs=False, permission=None, addlperms=None, urlcheck=()*)

Decorator to load a model given a query parameter.

Typical usage:

```
@app.route('/<profile>')
@load_model(Profile, {'name': 'profile'}, 'profileob')
def profile_view(profileob):
    # 'profileob' is now a Profile model instance.
    # The load_model decorator replaced this:
    # profileob = Profile.query.filter_by(name=profile).first_or_404()
    return "Hello, %s" % profileob.name
```

Using the same name for request and parameter makes code easier to understand:

```
@app.route('/<profile>')
@load_model(Profile, {'name': 'profile'}, 'profile')
```

(continues on next page)

(continued from previous page)

```
def profile_view(profile):
    return "Hello, %s" % profile.name
```

`load_model` aborts with a 404 if no instance is found.

Parameters

- **model** – The SQLAlchemy model to query. Must contain a `query` object (which is the default with Flask-SQLAlchemy)
- **attributes** – A dict of attributes (from the URL request) that will be used to query for the object. For each key:value pair, the key is the name of the column on the model and the value is the name of the request parameter that contains the data
- **parameter** – The name of the parameter to the decorated function via which the result is passed. Usually the same as the attribute. If the parameter name is prefixed with 'g.', the parameter is also made available as g.<parameter>
- **kwargs** – If True, the original request parameters are passed to the decorated function as a `kwargs` parameter
- **permission** – If present, `load_model` calls the `permissions()` method of the retrieved object with `current_auth.actor` as a parameter. If `permission` is not present in the result, `load_model` aborts with a 403. The permission may be a string or a list of strings, in which case access is allowed if any of the listed permissions are available
- **addlperms** – Iterable or callable that returns an iterable containing additional permissions available to the user, apart from those granted by the models. In an app that uses Lastuser for authentication, passing `lastuser.permissions` will pass through permissions granted via Lastuser
- **urlcheck** (*list*) – If an attribute in this list has been used to load an object, but the value of the attribute in the loaded object does not match the request argument, issue a redirect to the corrected URL. This is useful for attributes like `url_id_name` and `url_name_uuid_b58` where the `name` component may change

`coaster.views.decorators.`**`load_models`**`(*chain, **kwargs)`

Decorator to load a chain of models from the given parameters. This works just like *`load_model()`* and accepts the same parameters, with some small differences.

Parameters

- **chain** – The chain is a list of tuples of (`model`, `attributes`, `parameter`). Lists and tuples can be used interchangeably. All retrieved instances are passed as parameters to the decorated function
- **permission** – Same as in *`load_model()`*, except `permissions()` is called on every instance in the chain and the retrieved permissions are passed as the second parameter to the next instance in the chain. This allows later instances to revoke permissions granted by earlier instances. As an example, if a URL represents a hierarchy such as `/<page>/<comment>`, the `page` can assign `edit` and `delete` permissions, while the `comment` can revoke `edit` and retain `delete` if the current user owns the page but not the comment

In the following example, load_models loads a Folder with a name matching the name in the URL, then loads a Page with a matching name and with the just-loaded Folder as parent. If the Page provides a 'view' permission to the current user, the decorated function is called:

```
@app.route('/<folder_name>/<page_name>')
@load_models(
```

(continues on next page)

(continued from previous page)

```
    (Folder, {'name': 'folder_name'}, 'folder'),
    (Page, {'name': 'page_name', 'parent': 'folder'}, 'page'),
    permission='view')
def show_page(folder, page):
    return render_template('page.html', folder=folder, page=page)
```

`coaster.views.decorators.`**`render_with`**(*template=None*, *json=False*, *jsonp=False*)

Decorator to render the wrapped function with the given template (or dictionary of mimetype keys to templates, where the template is a string name of a template file or a callable that returns a Response). The function's return value must be a dictionary and is passed to the template as parameters. Callable templates get a single parameter with the function's return value. Usage:

```
@app.route('/myview')
@render_with('myview.html')
def myview():
    return {'data': 'value'}

@app.route('/myview_with_json')
@render_with('myview.html', json=True)
def myview_no_json():
    return {'data': 'value'}

@app.route('/otherview')
@render_with({
    'text/html': 'otherview.html',
    'text/xml': 'otherview.xml'})
def otherview():
    return {'data': 'value'}

@app.route('/404view')
@render_with('myview.html')
def myview():
    return {'error': '404 Not Found'}, 404

@app.route('/headerview')
@render_with('myview.html')
def myview():
    return {'data': 'value'}, 200, {'X-Header': 'Header value'}
```

When a mimetype is specified and the template is not a callable, the response is returned with the same mimetype. Callable templates must return Response objects to ensure the correct mimetype is set.

If a dictionary of templates is provided and does not include a handler for `*/*`, render_with will attempt to use the handler for (in order) `text/html`, `text/plain` and the various JSON types, falling back to rendering the value into a unicode string.

If the method is called outside a request context, the wrapped method's original return value is returned. This is meant to facilitate testing and should not be used to call the method from within another view handler as the presence of a request context will trigger template rendering.

Rendering may also be suspended by calling the view handler with `_render=False`.

render_with provides JSON and JSONP handlers for the `application/json`, `text/json` and `text/x-json` mimetypes if `json` or `jsonp` is True (default is False).

Parameters

- **`template`** – Single template, or dictionary of MIME type to templates. If the template is a callable, it is called with the output of the wrapped function

- **json** – Helper to add a JSON handler (default is False)
- **jsonp** – Helper to add a JSONP handler (if True, also provides JSON, default is False)

coaster.views.decorators.**cors**(*origins*, *methods=('HEAD', 'OPTIONS', 'GET', 'POST', 'PUT', 'PATCH', 'DELETE')*, *headers=('Accept', 'Accept-Language', 'Content-Language', 'Content-Type', 'X-Requested-With')*, *max_age=None*)

Adds CORS headers to the decorated view function.

Parameters

- **origins** – Allowed origins (see below)
- **methods** – A list of allowed HTTP methods
- **headers** – A list of allowed HTTP headers
- **max_age** – Duration in seconds for which the CORS response may be cached

The `origins` parameter may be one of:

1. A callable that receives the origin as a parameter.
2. A list of origins.
3. `*`, indicating that this resource is accessible by any origin.

Example use:

```
from flask import Flask, Response
from coaster.views import cors

app = Flask(__name__)

@app.route('/any')
@cors('*')
def any_origin():
    return Response()

@app.route('/static', methods=['GET', 'POST'])
@cors(
    ['https://hasgeek.com'],
    methods=['GET'],
    headers=['Content-Type', 'X-Requested-With'],
    max_age=3600)
def static_list():
    return Response()

def check_origin(origin):
    # check if origin should be allowed
    return True

@app.route('/callable')
@cors(check_origin)
def callable_function():
    return Response()
```

coaster.views.decorators.**requires_permission**(*permission*)

View decorator that requires a certain permission to be present in `current_auth.permissions` before the view is allowed to proceed. Aborts with `403 Forbidden` if the permission is not present.

The decorated view will have an `is_available` method that can be called to perform the same test.

Parameters **`permission`** – Permission that is required. If a collection type is provided, any one permission must be available

1.16 Class-based views

Group related views into a class for easier management.

`coaster.views.classview.`**`rulejoin`**`(`*class_rule*, *method_rule*`)`

Join class and method rules. Used internally by *`ClassView`* to combine rules from the *`route()`* decorators on the class and on the individual view handler methods:

```
>>> rulejoin('/', '')
'/'
>>> rulejoin('/', 'first')
'/first'
>>> rulejoin('/first', '/second')
'/second'
>>> rulejoin('/first', 'second')
'/first/second'
>>> rulejoin('/first/', 'second')
'/first/second'
>>> rulejoin('/first/<second>', '')
'/first/<second>'
>>> rulejoin('/first/<second>', 'third')
'/first/<second>/third'
```

`coaster.views.classview.`**`current_view = None`**

A proxy object that holds the currently executing *`ClassView`* instance, for use in templates as context. Exposed to templates by *`coaster.app.init_app()`*. Note that the current view handler method within the class is named `current_handler`, so to examine it, use `current_view.current_handler`.

`class` `coaster.views.classview.`**`ClassView`**

Base class for defining a collection of views that are related to each other. Subclasses may define methods decorated with *`route()`*. When *`init_app()`* is called, these will be added as routes to the app.

Typical use:

```
@route('/')
class IndexView(ClassView):
    @viewdata(title="Homepage")
    @route('')
    def index():
        return render_template('index.html.jinja2')

    @route('about')
    @viewdata(title="About us")
    def about():
        return render_template('about.html.jinja2')

IndexView.init_app(app)
```

The *`route()`* decorator on the class specifies the base rule, which is prefixed to the rule specified on each view method. This example produces two view handlers, for `/` and `/about`. Multiple *`route()`* decorators may be used in both places.

The *viewdata()* decorator can be used to specify additional data, and may appear either before or after the *route()* decorator, but only adjacent to it. Data specified here is available as the `data` attribute on the view handler, or at runtime in templates as `current_view.current_handler.data`.

A rudimentary CRUD view collection can be assembled like this:

```
@route('/doc/<name>')
class DocumentView(ClassView):
    @route('')
    @render_with('mydocument.html.jinja2', json=True)
    def view(self, name):
        document = MyDocument.query.filter_by(name=name).first_or_404()
        return document.current_access()

    @route('edit', methods=['POST'])
    @requestform('title', 'content')
    def edit(self, name, title, content):
        document = MyDocument.query.filter_by(name=name).first_or_404()
        document.title = title
        document.content = content
        return 'edited!'

DocumentView.init_app(app)
```

See *ModelView* for a better way to build views around a model.

classmethod add_route_for(*_name*, *rule*, ***options*)

Add a route for an existing method or view. Useful for modifying routes that a subclass inherits from a base class:

```
class BaseView(ClassView):
    def latent_view(self):
        return 'latent-view'

    @route('other')
    def other_view(self):
        return 'other-view'

@route('/path')
class SubView(BaseView):
    pass

SubView.add_route_for('latent_view', 'latent')
SubView.add_route_for('other_view', 'another')
SubView.init_app(app)

# Created routes:
# /path/latent -> SubView.latent (added)
# /path/other -> SubView.other (inherited)
# /path/another -> SubView.other (added)
```

Parameters

- **_name** – Name of the method or view on the class
- **rule** – URL rule to be added
- **options** – Additional options for `add_url_rule()`

after_request(*response*)
This method is called with the response from the view handler method. It must return a valid response object. Subclasses and mixin classes may override this to perform any necessary post-processing:

```
class MyView(ClassView):
    ...
    def after_request(self, response):
        response = super().after_request(response)
        ...  # Process here
        return response
```

Parameters **response** – Response from the view handler method

Returns Response object

before_request()
This method is called after the app's `before_request` handlers, and before the class's view method. Subclasses and mixin classes may define their own *before_request()* to pre-process requests. This method receives context via *self*, in particular via *current_handler* and *view_args*.

current_handler = None
When a view is called, this will point to the current view handler, an instance of `ViewHandler`.

dispatch_request(*view*, *view_args*)
View dispatcher that calls before_request, the view, and then after_request. Subclasses may override this to provide a custom flow. *ModelView* does this to insert a model loading phase.

Parameters

- **view** – View method wrapped in specified decorators. The dispatcher must call this
- **view_args** (*dict*) – View arguments, to be passed on to the view method

classmethod init_app(*app*, *callback=None*)
Register views on an app. If `callback` is specified, it will be called after `app.add_url_rule()`, with the same parameters.

is_always_available = False
Indicates whether meth:*is_available* should simply return *True* without conducting a test. Subclasses should not set this flag. It will be set by *init_app()* if any view handler is missing an `is_available` method, as it implies that view is always available.

is_available()
Returns *True* if *any* view handler in the class is currently available via its *is_available* method.

view_args = None
When a view is called, this will be replaced with a dictionary of arguments to the view.

class coaster.views.classview.**ModelView**(*obj=None*)
Base class for constructing views around a model. Functionality is provided via mixin classes that must precede *ModelView* in base class order. Two mixins are provided: *UrlForView* and *InstanceLoader*. Sample use:

```
@route('/doc/<document>')
class DocumentView(UrlForView, InstanceLoader, ModelView):
    model = Document
    route_model_map = {
        'document': 'name'
        }
```

(continues on next page)

(continued from previous page)

```
    @route('')
    @render_with(json=True)
    def view(self):
        return self.obj.current_access()

Document.views.main = DocumentView
DocumentView.init_app(app)
```

Views will not receive view arguments, unlike in `ClassView`. If necessary, they are available as *self.view_args*.

`dispatch_request`(*view*, *view_args*)
View dispatcher that calls `before_request()`, `loader()`, `after_loader()`, the view, and then `after_request()`.

Parameters

- **`view`** – View method wrapped in specified decorators.
- **`view_args`** (`dict`) – View arguments, to be passed on to the view method

`loader`(***view_args*)
Subclasses or mixin classes may override this method to provide a model instance loader. The return value of this method will be placed at `self.obj`.

Returns Object instance loaded from database

`model = None`
The model that this view class represents, to be specified by subclasses.

`query = None`
A base query to use if the model needs special handling.

`route_model_map = {}`
A mapping of URL rule variables to attributes on the model. For example, if the URL rule is `/<parent>/<document>`, the attribute map can be:

```
model = MyModel
route_model_map = {
    'document': 'name',        # Map 'document' in URL to MyModel.name
    'parent': 'parent.name',   # Map 'parent' to MyModel.parent.name
    }
```

The `InstanceLoader` mixin class will convert this mapping into SQLAlchemy attribute references to load the instance object.

`coaster.views.classview.`**`route`**(*rule*, ***options*)
Decorator for defining routes on a `ClassView` and its methods. Accepts the same parameters that Flask's `app.route()` accepts. See `ClassView` for usage notes.

`coaster.views.classview.`**`viewdata`**(***kwargs*)
Decorator for adding additional data to a view method, to be used alongside `route()`. This data is accessible as the `data` attribute on the view handler.

`coaster.views.classview.`**`url_change_check`**(*f*)
View method decorator that checks the URL of the loaded object in `self.obj` against the URL in the request (using `self.obj.url_for(__name__)`). If the URLs do not match, and the request is a `GET`, it issues a redirect to the correct URL. Usage:

```
@route('/doc/<document>')
class MyModelView(UrlForView, InstanceLoader, ModelView):
    model = MyModel
    route_model_map = {'document': 'url_id_name'}

    @route('')
    @url_change_check
    @render_with(json=True)
    def view(self):
        return self.obj.current_access()
```

If the decorator is required for all view handlers in the class, use *UrlChangeCheck*.

This decorator will only consider the URLs to be different if:

- Schemes differ (`http` vs `https` etc)
- Hostnames differ (apart from a case difference, as user agents use lowercase)
- Paths differ

The current URL's query will be copied to the redirect URL. The URL fragment (`#target_id`) is not available to the server and will be lost.

`coaster.views.classview.`**`requires_roles`**(*roles*)
Decorator for *ModelView* views that limits access to the specified roles.

class `coaster.views.classview.`**`UrlChangeCheck`**
Mixin class for *ModelView* and *UrlForMixin* that applies the *url_change_check()* decorator to all view handler methods. Subclasses *UrlForView*, which it depends on to register the view with the model so that URLs can be generated. Usage:

```
@route('/doc/<document>')
class MyModelView(UrlChangeCheck, InstanceLoader, ModelView):
    model = MyModel
    route_model_map = {'document': 'url_id_name'}

    @route('')
    @render_with(json=True)
    def view(self):
        return self.obj.current_access()
```

class `coaster.views.classview.`**`UrlForView`**
Mixin class for *ModelView* that registers view handler methods with *UrlForMixin*'s *is_url_for()*.

class `coaster.views.classview.`**`InstanceLoader`**
Mixin class for *ModelView* that provides a `loader()` that attempts to load an instance of the model based on attributes in the *route_model_map* dictionary.

InstanceLoader will traverse relationships (many-to-one or one-to-one) and perform a SQL `JOIN` with the target class.

1.17 SQLAlchemy patterns

Coaster provides a number of SQLAlchemy helper functions and mixin classes that add standard columns or special functionality.

All functions and mixins are importable from the *coaster.sqlalchemy* namespace.

1.18 SQLAlchemy mixin classes

Coaster provides a number of mixin classes for SQLAlchemy models. To use in your Flask app:

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from coaster.sqlalchemy import BaseMixin

app = Flask(__name__)
db = SQLAlchemy(app)

class MyModel(BaseMixin, db.Model):
    __tablename__ = 'my_model'
```

Mixin classes must always appear *before* `db.Model` in your model's base classes.

class `coaster.sqlalchemy.mixins.`**`IdMixin`**

Provides the `id` primary key column

`query_class`

alias of *`coaster.sqlalchemy.comparators.Query`*

`url_id`

The URL id, integer primary key rendered as a string

class `coaster.sqlalchemy.mixins.`**`TimestampMixin`**

Provides the `created_at` and `updated_at` audit timestamps

`query_class`

alias of *`coaster.sqlalchemy.comparators.Query`*

class `coaster.sqlalchemy.mixins.`**`PermissionMixin`**

Provides the *`permissions()`* method used by BaseMixin and derived classes

`current_permissions`

`InspectableSet` containing currently available permissions from this object, using *`current_auth`*.

`permissions`(*actor*, *inherited=None*)

Return permissions available to the given user on this object

class `coaster.sqlalchemy.mixins.`**`UrlDict`**(*obj*)

Provides dictionary access to an object's URLs.

class `coaster.sqlalchemy.mixins.`**`UrlForMixin`**

Provides a *`url_for()`* method used by BaseMixin-derived classes

`classview_for`(*action='view'*)

Return the classview that contains the viewhandler for the specified action

classmethod `is_url_for`(*_action*, *_endpoint=None*, *_app=None*, *_external=None*, ***paramattrs*)

View decorator that registers the view as a *`url_for()`* target.

Parameters

- **`_action`** (*`str`*) – Action to register a URL under
- **`_endpoint`** (*`str`*) – View endpoint name to pass to Flask's `url_for`
- **`_app`** – The app to register this action on (if your repo has multiple apps)
- **`_external`** – If *True*, URLs are assumed to be external-facing by default

- **paramattrs** (*dict*) – Mapping of URL parameter to attribute on the object

classmethod register_endpoint(*action*, *endpoint*, *app=None*, *external=None*, *roles=None*, *paramattrs=None*)
Helper method for registering an endopint to a `url_for()` action.

Parameters

- **view_func** – View handler to be registered
- **action** (*str*) – Action to register a URL under
- **endpoint** (*str*) – View endpoint name to pass to Flask's `url_for`
- **app** – Flask app (default: *None*)
- **external** – If *True*, URLs are assumed to be external-facing by default
- **roles** – Roles to which this URL is available, required by `UrlDict`
- **paramattrs** (*dict*) – Mapping of URL parameter to attribute on the object

classmethod register_view_for(*app*, *action*, *classview*, *attr*)
Register a classview and viewhandler for a given app and action

url_for(*action='view'*, ***kwargs*)
Return public URL to this instance for a given action (default 'view').

url_for_endpoints = {None: {}}
Mapping of {app: {action: UrlEndpointData}}, where attr is a string or tuple of strings. The same action can point to different endpoints in different apps. The app may also be None as fallback. Each subclass will get its own dictionary. This particular dictionary is only used as an inherited fallback.

urls
Dictionary of URLs available on this object

view_for(*action='view'*)
Return the classview viewhandler that handles the specified action

view_for_endpoints = {}
Mapping of {app: {action: (classview, attr)}}

class coaster.sqlalchemy.mixins.**NoIdMixin**
Mixin that combines all mixin classes except `IdMixin`, for use anywhere the timestamp columns and helper methods are required, but an id column is not.

class coaster.sqlalchemy.mixins.**BaseMixin**
Base mixin class for all tables that have an id column.

class coaster.sqlalchemy.mixins.**BaseNameMixin**(**args*, ***kw*)
Base mixin class for named objects

Changed in version 0.5.0: If you used BaseNameMixin in your app before Coaster 0.5.0: `name` can no longer be a blank string in addition to being non-null. This is configurable and enforced with a SQL CHECK constraint, which needs a database migration:

```
for tablename in ['named_table1', 'named_table2', ...]:
    # Drop CHECK constraint first in case it was already present
    op.drop_constraint(tablename + '_name_check', tablename)
    # Create CHECK constraint
    op.create_check_constraint(
        tablename + '_name_check',
        tablename,
        "name <> ''")
```

classmethod get(*name*)
Get an instance matching the name

make_name(*reserved=()*)
Autogenerates a `name` from the `title`. If the auto-generated name is already in use in this model, `make_name()` tries again by suffixing numbers starting with 2 until an available name is found.

Parameters reserved – List or set of reserved names unavailable for use

reserved_names = []
Prevent use of these reserved names

title_for_name
The version of the title used for `make_name()`

classmethod upsert(*name*, ***fields*)
Insert or update an instance

class coaster.sqlalchemy.mixins.**BaseScopedNameMixin**(**args*, ***kw*)
Base mixin class for named objects within containers. When using this, you must provide an model-level attribute "parent" that is a synonym for the parent object. You must also create a unique constraint on 'name' in combination with the parent foreign key. Sample use case in Flask:

```
class Event(BaseScopedNameMixin, db.Model):
    __tablename__ = 'event'
    organizer_id = db.Column(None, db.ForeignKey('organizer.id'))
    organizer = db.relationship(Organizer)
    parent = db.synonym('organizer')
    __table_args__ = (db.UniqueConstraint('organizer_id', 'name'),)
```

Changed in version 0.5.0: If you used BaseScopedNameMixin in your app before Coaster 0.5.0: `name` can no longer be a blank string in addition to being non-null. This is configurable and enforced with a SQL CHECK constraint, which needs a database migration:

```
for tablename in ['named_table1', 'named_table2', ...]:
    # Drop CHECK constraint first in case it was already present
    op.drop_constraint(tablename + '_name_check', tablename)
    # Create CHECK constraint
    op.create_check_constraint(
        tablename + '_name_check',
        tablename,
        "name <> ''")
```

classmethod get(*parent*, *name*)
Get an instance matching the parent and name

make_name(*reserved=()*)
Autogenerates a `name` from the `title`. If the auto-generated name is already in use in this model, `make_name()` tries again by suffixing numbers starting with 2 until an available name is found.

permissions(*actor*, *inherited=None*)
Permissions for this model, plus permissions inherited from the parent.

reserved_names = []
Prevent use of these reserved names

short_title
Generates an abbreviated title by subtracting the parent's title from this instance's title.

title_for_name
The version of the title used for `make_name()`

classmethod upsert(*parent*, *name*, ***fields*)
Insert or update an instance

class coaster.sqlalchemy.mixins.**BaseIdNameMixin**(**args*, ***kw*)
Base mixin class for named objects with an id tag.

Changed in version 0.5.0: If you used BaseIdNameMixin in your app before Coaster 0.5.0: `name` can no longer be a blank string in addition to being non-null. This is configurable and enforced with a SQL CHECK constraint, which needs a database migration:

```
for tablename in ['named_table1', 'named_table2', ...]:
    # Drop CHECK constraint first in case it was already present
    op.drop_constraint(tablename + '_name_check', tablename)
    # Create CHECK constraint
    op.create_check_constraint(
        tablename + '_name_check',
        tablename,
        "name <> ''")
```

make_name()
Autogenerates a `name` from *title_for_name*

title_for_name
The version of the title used for *make_name()*

url_id_name
Returns a URL name combining `url_id` and `name` in id-name syntax. This property is also available as *url_name* for legacy reasons.

url_name
Returns a URL name combining `url_id` and `name` in id-name syntax. This property is also available as *url_name* for legacy reasons.

url_name_uuid_b58
Returns a URL name combining `name` and `uuid_b58` in name-uuid_b58 syntax. To use this, the class must derive from *UuidMixin*.

class coaster.sqlalchemy.mixins.**BaseScopedIdMixin**(**args*, ***kw*)
Base mixin class for objects with an id that is unique within a parent. Implementations must provide a 'parent' attribute that is either a relationship or a synonym to a relationship referring to the parent object, and must declare a unique constraint between url_id and the parent. Sample use case in Flask:

```
class Issue(BaseScopedIdMixin, db.Model):
    __tablename__ = 'issue'
    event_id = db.Column(None, db.ForeignKey('event.id'))
    event = db.relationship(Event)
    parent = db.synonym('event')
    __table_args__ = (db.UniqueConstraint('event_id', 'url_id'),)
```

classmethod get(*parent*, *url_id*)
Get an instance matching the parent and url_id

make_id()
Create a new URL id that is unique to the parent container

permissions(*actor*, *inherited=None*)
Permissions for this model, plus permissions inherited from the parent.

class coaster.sqlalchemy.mixins.**BaseScopedIdNameMixin**(**args*, ***kw*)
Base mixin class for named objects with an id tag that is unique within a parent. Implementations must provide

a 'parent' attribute that is a synonym to the parent relationship, and must declare a unique constraint between url_id and the parent. Sample use case in Flask:

```
class Event(BaseScopedIdNameMixin, db.Model):
    __tablename__ = 'event'
    organizer_id = db.Column(None, db.ForeignKey('organizer.id'))
    organizer = db.relationship(Organizer)
    parent = db.synonym('organizer')
    __table_args__ = (db.UniqueConstraint('organizer_id', 'url_id'),)
```

Changed in version 0.5.0: If you used BaseScopedIdNameMixin in your app before Coaster 0.5.0: `name` can no longer be a blank string in addition to being non-null. This is configurable and enforced with a SQL CHECK constraint, which needs a database migration:

```
for tablename in ['named_table1', 'named_table2', ...]:
    # Drop CHECK constraint first in case it was already present
    op.drop_constraint(tablename + '_name_check', tablename)
    # Create CHECK constraint
    op.create_check_constraint(
        tablename + '_name_check',
        tablename,
        "name <> ''")
```

classmethod get(*parent*, *url_id*)
Get an instance matching the parent and name

make_name()
Autogenerates a title from the name

title_for_name
The version of the title used for *make_name()*

url_id_name
Returns a URL name combining `url_id` and `name` in id-name syntax

url_name
Returns a URL name combining `url_id` and `name` in id-name syntax

url_name_uuid_b58
Returns a URL name combining `name` and `uuid_b58` in name-uuid_b58 syntax. To use this, the class must derive from *UuidMixin*.

class coaster.sqlalchemy.mixins.**CoordinatesMixin**
Adds `latitude` and `longitude` columns with a shorthand *coordinates* property that returns both.

coordinates
Tuple of (latitude, longitude).

has_coordinates
Return *True* if both latitude and longitude are present.

has_missing_coordinates
Return *True* if one or both of latitude and longitude are missing.

class coaster.sqlalchemy.mixins.**UuidMixin**
Provides a `uuid` attribute that is either a SQL UUID column or an alias to the existing `id` column if the class uses UUID primary keys. Also provides hybrid properties `uuid_hex`, `buid` and `uuid_b58` that provide hex, URL-safe Base64 and Base58 representations of the `uuid` column.

buid
Retain *buid* as a public attribute for backward compatibility

uuid_b58
URL-friendly UUID representation, using Base58 with the Bitcoin alphabet

uuid_b64
URL-friendly UUID representation, using URL-safe Base64 (BUID)

uuid_hex
URL-friendly UUID representation as a hex string

class coaster.sqlalchemy.mixins.**RoleMixin**
Provides methods for role-based access control.

Subclasses must define a `__roles__` dictionary with roles and the attributes they have call, read and write access to:

```
__roles__ = {
    'role_name': {
        'call': {'meth1', 'meth2'},
        'read': {'attr1', 'attr2'},
        'write': {'attr1', 'attr2'},
        'grant': {'rel1', 'rel2'},
        },
    }
```

The `grant` key works in reverse: if the actor is present in any of the attributes in the set, they are granted that role via `roles_for()`. Attributes must be SQLAlchemy relationships and can be scalar, a collection or dynamic.

The `with_roles()` decorator is recommended over `__roles__`.

access_for(*roles=None*, *actor=None*, *anchors=()*, *datasets=None*)
Return a proxy object that limits read and write access to attributes based on the actor's roles.

Warning: If the *roles* parameter is provided, it overrides discovery of the actor's roles in both the current object and related objects. It should only be used when roles are pre-determined and related objects are not required.

Parameters

- **roles** (*set*) – Roles to limit access to (not recommended)
- **actor** – Limit access to this actor's roles
- **anchors** – Retrieve additional roles from anchors
- **datasets** (*tuple*) – Limit enumeration to the attributes in the dataset

If a *datasets* sequence is provided, the first dataset is applied to the current object and subsequent datasets are applied to objects accessed via relationships. Datasets limit the attributes available via enumeration when the proxy is cast into a dict or JSON. This can be used to remove unnecessary data or bi-directional relationships, which JSON can't handle.

Attributes must be specified in a `__datasets__` dictionary on the object:

```
__datasets__ = {
    'primary': {'uuid', 'name', 'title', 'children', 'parent'},
    'related': {'uuid', 'name', 'title'}
}
```

Objects and related objects can be safely enumerated like this:

```
proxy = obj.access_for(user, datasets=('primary', 'related'))
proxydict = dict(proxy)
proxyjson = json.dumps(proxy)  # This needs a custom JSON encoder
```

If a dataset includes an attribute the role doesn't have access to, it will be skipped. If it includes a relationship for which no dataset is specified, it will be rendered as an empty dict.

actors_with(*roles*, *with_role=False*)

Return actors who have the specified roles on this object, as an iterator.

Uses: 1. `__roles__[role]['granted_by']` 2. `__roles__[role]['granted_via']`

Subclasses of *RoleMixin* that have custom role granting logic in *roles_for()* must provide a matching *actors_with()* implementation.

Parameters

- **roles** (*set*) – Iterable specifying roles to find actors with. May be an ordered type if ordering is important
- **with_role** (*bool*) – If True, yields a tuple of the actor and the role they were found with. The actor may have more roles, but only the first match is returned

current_access(*datasets=None*)

Wraps *access_for()* with *current_auth* to return a proxy for the currently authenticated user.

Parameters **datasets** (*tuple*) – Datasets to limit enumeration to

current_roles

InspectableSet containing currently available roles on this object, using *current_auth*. Use in the view layer to inspect for a role being present:

if obj.current_roles.editor: pass

{% if obj.current_roles.editor %}…{% endif %}

This property is also available in `RoleAccessProxy`.

> **Warning:** *current_roles* maintains a cache for efficient use in a template where it may be consulted multiple times. It is therefore not safe to use before *and* after code that modifies role assignment. Use *roles_for()* instead, or use *current_roles* only after roles are changed.

roles_for(*actor=None*, *anchors=()*)

Return roles available to the given `actor` or `anchors` on this object. The data type for both parameters are intentionally undefined here. Subclasses are free to define them in any way appropriate. Actors and anchors are assumed to be valid.

The role `all` is always granted. If `actor` is specified, the role `auth` is granted. If not, `anon` is granted.

Subclasses overriding *roles_for()* must always call `super()` to ensure they are receiving the standard roles. Recommended boilerplate:

```
def roles_for(self, actor=None, anchors=()):
    roles = super().roles_for(actor, anchors)
    # 'roles' is a set. Add more roles here
    # ...
    return roles
```

class `coaster.sqlalchemy.mixins.`**`RegistryMixin`**

Adds common registries to a model.

Included:

- `forms` registry, for WTForms forms
- `views` registry for view classes and helper functions
- `features` registry for feature availability test functions.

The forms registry passes the instance to the registered form as an `obj` keyword parameter. The other registries pass it as the first positional parameter.

1.19 SQLAlchemy column types

class `coaster.sqlalchemy.columns.`**`JsonDict`**(**args*, ***kwargs*)

Represents a JSON data structure. Usage:

```
column = Column(JsonDict)
```

The column will be represented to the database as a `JSONB` column if the server is PostgreSQL 9.4 or later, `JSON` if PostgreSQL 9.2 or 9.3, and `TEXT` for everything else. The column behaves like a JSON store regardless of the backing data type.

`impl`

alias of `sqlalchemy.sql.sqltypes.TEXT`

`load_dialect_impl`(*dialect*)

Return a `TypeEngine` object corresponding to a dialect.

This is an end-user override hook that can be used to provide differing types depending on the given dialect. It is used by the `TypeDecorator` implementation of `type_engine()` to help determine what type should ultimately be returned for a given `TypeDecorator`.

By default returns `self.impl`.

`process_bind_param`(*value*, *dialect*)

Receive a bound parameter value to be converted.

Subclasses override this method to return the value that should be passed along to the underlying `TypeEngine` object, and from there to the DBAPI `execute()` method.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

This operation should be designed with the reverse operation in mind, which would be the process_result_value method of this class.

Parameters

- **`value`** – Data to operate upon, of any type expected by this method in the subclass. Can be `None`.
- **`dialect`** – the `Dialect` in use.

`process_result_value`(*value*, *dialect*)

Receive a result-row column value to be converted.

Subclasses should implement this method to operate on data fetched from the database.

Subclasses override this method to return the value that should be passed back to the application, given a value that is already processed by the underlying `TypeEngine` object, originally from the DBAPI cursor method `fetchone()` or similar.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

Parameters

- **`value`** – Data to operate upon, of any type expected by this method in the subclass. Can be `None`.
- **`dialect`** – the `Dialect` in use.

This operation should be designed to be reversible by the "process_bind_param" method of this class.

class `coaster.sqlalchemy.columns.`**`UUIDType`**(*binary=True, native=True*)

Stores a UUID in the database natively when it can and falls back to a BINARY(16) or a CHAR(32) when it can't.

```
from sqlalchemy_utils import UUIDType
import uuid

class User(Base):
    __tablename__ = 'user'

    # Pass `binary=False` to fallback to CHAR instead of BINARY
    id = sa.Column(UUIDType(binary=False), primary_key=True)
```

`load_dialect_impl`(*dialect*)

Return a `TypeEngine` object corresponding to a dialect.

This is an end-user override hook that can be used to provide differing types depending on the given dialect. It is used by the `TypeDecorator` implementation of `type_engine()` to help determine what type should ultimately be returned for a given `TypeDecorator`.

By default returns `self.impl`.

`process_bind_param`(*value, dialect*)

Receive a bound parameter value to be converted.

Subclasses override this method to return the value that should be passed along to the underlying `TypeEngine` object, and from there to the DBAPI `execute()` method.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

This operation should be designed with the reverse operation in mind, which would be the process_result_value method of this class.

Parameters

- **`value`** – Data to operate upon, of any type expected by this method in the subclass. Can be `None`.
- **`dialect`** – the `Dialect` in use.

`process_literal_param`(*value, dialect*)

Receive a literal parameter value to be rendered inline within a statement.

This method is used when the compiler renders a literal value without using binds, typically within DDL such as in the "server default" of a column or an expression within a CHECK constraint.

The returned string will be rendered into the output string.

New in version 0.9.0.

process_result_value(*value*, *dialect*)

Receive a result-row column value to be converted.

Subclasses should implement this method to operate on data fetched from the database.

Subclasses override this method to return the value that should be passed back to the application, given a value that is already processed by the underlying `TypeEngine` object, originally from the DBAPI cursor method `fetchone()` or similar.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be `None`.
- **dialect** – the `Dialect` in use.

This operation should be designed to be reversible by the "process_bind_param" method of this class.

python_type

alias of `uuid.UUID`

class coaster.sqlalchemy.columns.**UrlType**(*schemes=('http', 'https')*, *optional_scheme=False*, *optional_host=False*)

Extension of URLType from SQLAlchemy-Utils that adds basic validation to ensure URLs are well formed. Parses the value into a `furl` object, allowing manipulation of

Parameters

- **schemes** – Valid URL schemes. Use *None* to allow any scheme, *()* for no scheme
- **optional_scheme** – Schemes are optional (allows URLs starting with `//`)
- **optional_host** – Allow URLs without a hostname (required for `mailto` and `file` schemes)

impl

alias of `sqlalchemy.sql.sqltypes.UnicodeText`

process_bind_param(*value*, *dialect*)

Receive a bound parameter value to be converted.

Subclasses override this method to return the value that should be passed along to the underlying `TypeEngine` object, and from there to the DBAPI `execute()` method.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

This operation should be designed with the reverse operation in mind, which would be the process_result_value method of this class.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be `None`.
- **dialect** – the `Dialect` in use.

process_result_value(*value*, *dialect*)

Receive a result-row column value to be converted.

Subclasses should implement this method to operate on data fetched from the database.

Subclasses override this method to return the value that should be passed back to the application, given a value that is already processed by the underlying `TypeEngine` object, originally from the DBAPI cursor method `fetchone()` or similar.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

Parameters

- **`value`** – Data to operate upon, of any type expected by this method in the subclass. Can be `None`.
- **`dialect`** – the `Dialect` in use.

This operation should be designed to be reversible by the "process_bind_param" method of this class.

`url_parser`
alias of `furl.furl.furl`

1.20 Helper functions

`coaster.sqlalchemy.functions.`**`make_timestamp_columns`**`(`*timezone=False*`)`
Return two columns, *created_at* and *updated_at*, with appropriate defaults

`coaster.sqlalchemy.functions.`**`failsafe_add`**`(`*_session*, *_instance*, ***filters*`)`
Add and commit a new instance in a nested transaction (using SQL SAVEPOINT), gracefully handling failure in case a conflicting entry is already in the database (which may occur due to parallel requests causing race conditions in a production environment with multiple workers).

Returns the instance saved to database if no error occurred, or loaded from database using the provided filters if an error occurred. If the filters fail to load from the database, the original IntegrityError is re-raised, as it is assumed to imply that the commit failed because of missing or invalid data, not because of a duplicate entry.

However, when no filters are provided, nothing is returned and IntegrityError is also suppressed as there is no way to distinguish between data validation failure and an existing conflicting record in the database. Use this option when failures are acceptable but the cost of verification is not.

Usage: `failsafe_add(db.session, instance, **filters)` where filters are the parameters passed to `Model.query.filter_by(**filters).one()` to load the instance.

You must commit the transaction as usual after calling `failsafe_add`.

Parameters

- **`_session`** – Database session
- **`_instance`** – Instance to commit
- **`filters`** – Filters required to load existing instance from the database in case the commit fails (required)

Returns Instance that is in the database

`coaster.sqlalchemy.functions.`**`add_primary_relationship`**`(`*parent*, *childrel*, *child*, *parentrel*, *parentcol*`)`
When a parent-child relationship is defined as one-to-many, *`add_primary_relationship()`* lets the parent refer to one child as the primary, by creating a secondary table to hold the reference. Under PostgreSQL, a trigger is added as well to ensure foreign key integrity.

A SQLAlchemy relationship named `parent.childrel` is added that makes usage seamless within SQLAlchemy.

The secondary table is named after the parent and child tables, with `_primary` appended, in the form `parent_child_primary`. This table can be found in the metadata in the `parent.metadata.tables` dictionary.

Multi-column primary keys on either parent or child are unsupported at this time.

Parameters

- **`parent`** – The parent model (on which this relationship will be added)
- **`childrel`** – The name of the relationship to the child that will be added
- **`child`** – The child model
- **`parentrel`** (*`str`*) – Name of the existing relationship on the child model that refers back to the parent model
- **`parentcol`** (*`str`*) – Name of the existing table column on the child model that refers back to the parent model

Returns Secondary table that was created

`coaster.sqlalchemy.functions.`**`auto_init_default`**(*column*)

Set the default value for a column when it's first accessed rather than first committed to the database.

1.21 Role-based access control

Coaster provides a *`RoleMixin`* class that can be used to define role-based access control to the attributes and methods of any SQLAlchemy model. *`RoleMixin`* is a base class for *`BaseMixin`* and applies to all derived classes. Access is defined as one of 'call' (for methods), 'read' or 'write' (both for attributes).

Roles are freeform string tokens. A model may freely define and grant roles to actors (users and sometimes client apps) based on internal criteria. The following standard tokens are recommended. Required tokens are granted by *`RoleMixin`* itself.

1. `all`: Any actor, authenticated or anonymous (required)
2. `anon`: Anonymous actor (required)
3. `auth`: Authenticated actor (required)
4. `creator`: The creator of an object (may or may not be the current owner)
5. `owner`: The current owner of an object
6. `author`: Author of the object's contents (all creators are authors)
7. `editor`: Someone authorised to edit the object
8. `reader`: Someone authorised to read the object (assuming it's not public)
9. `subject`: User who is described by an object, typically having limited rights

Example use:

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from coaster.sqlalchemy import BaseMixin, with_roles

app = Flask(__name__)
db = SQLAlchemy(app)

```

(continues on next page)

(continued from previous page)

```
class ColumnMixin:
    '''
    Mixin class that offers some columns to the RoleModel class below,
    demonstrating two ways to use `with_roles`.
    '''
    @with_roles(rw={'owner'})
    def mixed_in1(cls):
        return db.Column(db.Unicode(250))

    @declared_attr
    def mixed_in2(cls):
        return with_roles(db.Column(db.Unicode(250)),
            rw={'owner'})


class RoleModel(ColumnMixin, RoleMixin, db.Model):
    __tablename__ = 'role_model'

    # The low level approach is to declare roles all at once.
    # 'all' is a special role that is always granted from the base class.
    # Avoid this approach in a parent or mixin class as definitions will
    # be lost if the subclass does not copy `__roles__`.

    __roles__ = {
        'all': {
            'read': {'id', 'name', 'title'},
        },
        'owner': {
            'granted_by': ['user'],
        },
    }

    # Recommended for parent and mixin classes: annotate roles on the attributes
    # using `with_roles`. These annotations are added to `__roles__` when
    # SQLAlchemy configures mappers.

    id = db.Column(db.Integer, primary_key=True)
    name = with_roles(db.Column(db.Unicode(250)),
        rw={'owner'})  # Specify read+write access

    user_id = db.Column(None, db.ForeignKey('user.id'), nullable=False)
    user = with_roles(
        db.relationship(User),
        grants={'owner'},  # Use `grants` here or `granted_by` in `__roles__`
        )

    # `with_roles` can also be called later. This is required for
    # properties, where roles must be assigned after the property is
    # fully described:

    _title = db.Column('title', db.Unicode(250))

    @property
    def title(self):
        return self._title

    @title.setter
```

(continues on next page)

(continued from previous page)

```
    def title(self, value):
        self._title = value

    # This grants 'owner' and 'editor' write but not read access
    title = with_roles(title, write={'owner', 'editor'})

    # `with_roles` can be used as a decorator on methods, in which case
    # access is controlled with the 'call' action.

    @with_roles(call={'all'})
    def hello(self):
        return "Hello!"

    # `RoleMixin` will grant roles by examining relationships specified in the
    # `granted_by` list under each role in `__roles__`. The `actor` parameter
    # to `roles_for` must be present in the relationship. You can augment this
    # by providing a custom `roles_for` method:

    def roles_for(self, actor=None, anchors=()):
        # Calling super gives us a LazyRoleSet with the standard roles
        # and with lazy evaluation of of other roles from `granted_by`
        roles = super().roles_for(actor, anchors)

        # We can manually add a role to override lazy evaluation
        if 'owner-secret' in anchors:
            roles.add('owner')
        return roles
```

class `coaster.sqlalchemy.roles.`**`RoleGrantABC`**

Base class for an object that grants roles to an actor

`offered_roles`

Roles offered by this object

class `coaster.sqlalchemy.roles.`**`LazyRoleSet`**(*obj*, *actor*, *initial=()*)

Set that provides lazy evaluations for whether a role is present

`add`(*value*)

Add role *value* to the set.

`copy`()

Return a shallow copy of the `LazyRoleSet`.

`discard`(*value*)

Remove role *value* from the set if it is present.

`has_any`(*roles*)

Convenience method for checking if any of the given roles is present in the set.

Equivalent of evaluating using either of these approaches:

1. `not roles.isdisjoint(lazy_role_set)`
2. `any(role in lazy_role_set for role in roles)`

This implementation optimizes for cached roles before evaluating role granting sources that may cause a database hit.

class `coaster.sqlalchemy.roles.`**`RoleAccessProxy`**(*obj*, *roles*, *actor*, *anchors*, *datasets*)

A proxy interface that wraps an object and provides pass-through read and write access to attributes that the specified roles have access to. Consults the `__roles__` dictionary on the object for determining which roles can access which attributes. Provides both attribute and dictionary interfaces.

Note that if the underlying attribute is a callable and is specified with the 'call' action, it will be available via attribute access but not dictionary access.

`RoleAccessProxy` is typically accessed directly from the target object via *`access_for()`* (from *`RoleMixin`*).

Example:

```
proxy = RoleAccessProxy(obj, roles={'writer'})
proxy.attr1
proxy.attr1 = 'new value'
proxy['attr2'] = 'new value'
dict(proxy)
```

Parameters

- **`obj`** – The object that should be wrapped with the proxy
- **`roles`** – A set of roles to determine what attributes are accessible
- **`actor`** – The actor this proxy has been constructed for
- **`anchors`** – The anchors this proxy has been constructed with
- **`datasets`** – Datasets to limit attribute enumeration to

The *actor* and *anchors* parameters are not used by the proxy, but are used to construct proxies for objects accessed via relationships.

class `coaster.sqlalchemy.roles.`**`DynamicAssociationProxy`**`(`*rel*, *attr*`)`

Association proxy for dynamic relationships. Use this instead of SQLAlchemy's *association_proxy* when the underlying relationship uses *lazy='dynamic'*.

Usage:

```
# Assuming a relationship like this:
Document.child_relationship = db.relationship(ChildDocument, lazy='dynamic')

# Proxy to an attribute on the target of the relationship:
Document.child_attributes = DynamicAssociationProxy(
    'child_relationship', 'attribute')
```

This proxy does not provide access to the query capabilities of dynamic relationships. It merely optimises for containment queries. A query like this:

```
Document.child_relationship.filter_by(attribute=value).exists()
```

Can be reduced to this:

```
value in Document.child_attributes
```

Parameters

- **`rel`** (*`str`*) – Relationship name (must use `lazy='dynamic'`)
- **`attr`** (*`str`*) – Attribute on the target of the relationship

class coaster.sqlalchemy.roles.**RoleMixin**

Provides methods for role-based access control.

Subclasses must define a `__roles__` dictionary with roles and the attributes they have call, read and write access to:

```
__roles__ = {
    'role_name': {
        'call': {'meth1', 'meth2'},
        'read': {'attr1', 'attr2'},
        'write': {'attr1', 'attr2'},
        'grant': {'rel1', 'rel2'},
        },
    }
```

The `grant` key works in reverse: if the actor is present in any of the attributes in the set, they are granted that role via *roles_for()*. Attributes must be SQLAlchemy relationships and can be scalar, a collection or dynamic.

The *with_roles()* decorator is recommended over `__roles__`.

access_for(*roles=None*, *actor=None*, *anchors=()*, *datasets=None*)

Return a proxy object that limits read and write access to attributes based on the actor's roles.

Warning: If the *roles* parameter is provided, it overrides discovery of the actor's roles in both the current object and related objects. It should only be used when roles are pre-determined and related objects are not required.

Parameters

- **roles** (*set*) – Roles to limit access to (not recommended)
- **actor** – Limit access to this actor's roles
- **anchors** – Retrieve additional roles from anchors
- **datasets** (*tuple*) – Limit enumeration to the attributes in the dataset

If a *datasets* sequence is provided, the first dataset is applied to the current object and subsequent datasets are applied to objects accessed via relationships. Datasets limit the attributes available via enumeration when the proxy is cast into a dict or JSON. This can be used to remove unnecessary data or bi-directional relationships, which JSON can't handle.

Attributes must be specified in a `__datasets__` dictionary on the object:

```
__datasets__ = {
    'primary': {'uuid', 'name', 'title', 'children', 'parent'},
    'related': {'uuid', 'name', 'title'}
}
```

Objects and related objects can be safely enumerated like this:

```
proxy = obj.access_for(user, datasets=('primary', 'related'))
proxydict = dict(proxy)
proxyjson = json.dumps(proxy)  # This needs a custom JSON encoder
```

If a dataset includes an attribute the role doesn't have access to, it will be skipped. If it includes a relationship for which no dataset is specified, it will be rendered as an empty dict.

actors_with(*roles*, *with_role=False*)

Return actors who have the specified roles on this object, as an iterator.

Uses: 1. `__roles__[role]['granted_by']` 2. `__roles__[role]['granted_via']`

Subclasses of `RoleMixin` that have custom role granting logic in `roles_for()` must provide a matching `actors_with()` implementation.

Parameters

- **roles** (`set`) – Iterable specifying roles to find actors with. May be an ordered type if ordering is important
- **with_role** (`bool`) – If True, yields a tuple of the actor and the role they were found with. The actor may have more roles, but only the first match is returned

current_access(*datasets=None*)

Wraps `access_for()` with `current_auth` to return a proxy for the currently authenticated user.

Parameters **datasets** (`tuple`) – Datasets to limit enumeration to

current_roles

`InspectableSet` containing currently available roles on this object, using `current_auth`. Use in the view layer to inspect for a role being present:

if obj.current_roles.editor: pass

{% if obj.current_roles.editor %}…{% endif %}

This property is also available in `RoleAccessProxy`.

> **Warning:** *current_roles* maintains a cache for efficient use in a template where it may be consulted multiple times. It is therefore not safe to use before *and* after code that modifies role assignment. Use `roles_for()` instead, or use *current_roles* only after roles are changed.

roles_for(*actor=None*, *anchors=()*)

Return roles available to the given `actor` or `anchors` on this object. The data type for both parameters are intentionally undefined here. Subclasses are free to define them in any way appropriate. Actors and anchors are assumed to be valid.

The role `all` is always granted. If `actor` is specified, the role `auth` is granted. If not, `anon` is granted.

Subclasses overriding `roles_for()` must always call `super()` to ensure they are receiving the standard roles. Recommended boilerplate:

```python
def roles_for(self, actor=None, anchors=()):
    roles = super().roles_for(actor, anchors)
    # 'roles' is a set. Add more roles here
    # ...
    return roles
```

`coaster.sqlalchemy.roles.`**with_roles**(*obj=None*, *rw=None*, *call=None*, *read=None*, *write=None*, *grants=None*, *grants_via=None*, *datasets=None*)

Convenience function and decorator to define roles on an attribute. Only works with `RoleMixin`, which reads the annotations made by this function and populates `__roles__`.

Examples:

```
id = db.Column(Integer, primary_key=True)
with_roles(id, read={'all'})

title = with_roles(db.Column(db.UnicodeText), read={'all'})

@with_roles(read={'all'})
@hybrid_property
def url_id(self):
    return str(self.id)
```

When used with properties, with_roles must always be applied after the property is fully described:

```
@property
def title(self):
    return self._title

@title.setter
def title(self, value):
    self._title = value

# Either of the following is fine, since with_roles annotates objects
# instead of wrapping them. The return value can be discarded if it's
# already present on the host object:

title = with_roles(title, read={'all'}, write={'owner', 'editor'})
with_roles(title, read={'all'}, write={'owner', 'editor'})
```

Parameters

- **`rw`** (*`set`*) – Roles which get read and write access to the decorated attribute
- **`call`** (*`set`*) – Roles which get call access to the decorated method
- **`read`** (*`set`*) – Roles which get read access to the decorated attribute
- **`write`** (*`set`*) – Roles which get write access to the decorated attribute
- **`grants`** (*`set`*) – The decorated attribute contains actors with the given roles
- **`grants_via`** (*`dict`*) – The decorated attribute is a relationship to another object type which contains one or more actors who are granted roles here
- **`datasets`** (*`set`*) – Datasets to include the attribute in

`grants_via` is typically used like this:

```
class RoleModel(db.Model):
    user_id = db.Column(None, db.ForeignKey('user.id'))
    user = db.relationship(UserModel)

    document_id = db.Column(None, db.ForeignKey('document.id'))
    document = db.relationship(DocumentModel)

DocumentModel.rolemodels = with_roles(db.relationship(RoleModel),
    grants_via={'user': {'role1', 'role2'}})
```

In this example, a user gets roles 'role1' and 'role2' on DocumentModel via the secondary RoleModel. Grants are recorded in `__roles__['role1']['granted_via']` and are honoured by the *`LazyRoleSet`* used in *`roles_for()`*.

grants_via supports an additional advanced definition for when the role granting model has variable roles and offers them via a property named offered_roles:

```
class RoleModel(db.Model):
    user_id = db.Column(None, db.ForeignKey('user.id'))
    user = db.relationship(UserModel)

    has_role1 = db.Column(db.Boolean)
    has_role2 = db.Column(db.Boolean)

    document_id = db.Column(None, db.ForeignKey('document.id'))
    document = db.relationship(DocumentModel)

    @property
    def offered_roles(self):
        roles = set()
        if self.has_role1:
            roles.add('role1')
        if self.has_role2:
            roles.add('role2')
        return roles

DocumentModel.rolemodels = with_roles(db.relationship(RoleModel),
    grants_via={'user': {
        'role1': 'renamed_role1,
        'role2': {'renamed_role2', 'also_role2'}
    }}
)
```

coaster.sqlalchemy.roles.**declared_attr_roles**(*rw=None*, *call=None*, *read=None*, *write=None*)

Equivalent of *with_roles()* for use with @declared_attr:

```
@declared_attr
@declared_attr_roles(read={'all'})
def my_column(cls):
    return Column(Integer)
```

While *with_roles()* is always the outermost decorator on properties and functions, *declared_attr_roles()* must appear below @declared_attr to work correctly.

Deprecated since version 0.6.1: Use *with_roles()* instead. It works for declared_attr since 0.6.1

1.22 SQLAlchemy attribute annotations

Annotations are strings attached to attributes that serve as a programmer reference on how those attributes are meant to be used. They can be used to indicate that a column's value should be immutable and should never change, or that it's a cached copy of a value from another source that can be safely discarded in case of a conflict.

This module's exports may be imported via *coaster.sqlalchemy*.

Sample usage:

```
from coaster.db import db
from coaster.sqlalchemy import annotation_wrapper, immutable

natural_key = annotation_wrapper('natural_key', "Natural key for this model")
```

(continues on next page)

(continued from previous page)

```
class MyModel(db.Model):
    __tablename__ = 'my_model'
    id = immutable(db.Column(db.Integer, primary_key=True))
    name = natural_key(db.Column(db.Unicode(250), unique=True))

    @classmethod
    def get(cls, **kwargs):
        for key in kwargs:
            if key in cls.__column_annotations__[natural_key.name]:
                return cls.query.filter_by(**{key: kwargs[key]}).one_or_none()
```

Annotations are saved to the model's class as a `__column_annotations__` dictionary, mapping annotation names to a list of attribute names, and to a reverse lookup `__column_annotations_by_attr__` of attribute names to annotations.

`coaster.sqlalchemy.annotations.`**`annotation_wrapper`**(*annotation*, *doc=None*)
Define an annotation, which can be applied to attributes in a database model.

1.23 Immutable annotation

`coaster.sqlalchemy.immutable_annotation.`**`immutable`**(*attr*)
Marks a column as immutable once set. Only blocks direct changes; columns may still be updated via relationships or SQL

`coaster.sqlalchemy.immutable_annotation.`**`cached`**(*attr*)
Marks the column's contents as a cached value from another source

exception `coaster.sqlalchemy.immutable_annotation.`**`ImmutableColumnError`**(*class_name*, *column_name*, *old_value*, *new_value*, *message=None*)
Exception raised when an immutable column is set.

1.24 Model helper registry

Provides a *Registry* type and a *RegistryMixin* base class with three registries, used by other mixin classes.

Helper classes such as forms and views can be registered to the model and later accessed from an instance:

```
class MyModel(BaseMixin, db.Model):
    ...

class MyForm(Form):
    ...

class MyView(ModelView):
    ...
```

(continues on next page)

(continued from previous page)

```
MyModel.forms.main = MyForm
MyModel.views.main = MyView
```

When accessed from an instance, the registered form or view will receive the instance as an `obj` parameter:

```
doc = MyModel()
doc.forms.main() == MyForm(obj=doc)
doc.views.main() == MyView(obj=doc)
```

The name `main` is a recommended default, but an app that has separate forms for `new` and `edit` actions could use those names instead.

class `coaster.sqlalchemy.registry.`**`Registry`**(*param: Optional[str] = None*, *property: bool = False*, *cached_property: bool = False*)

Container for items registered to a model.

`clear_cache_for`(*obj*) → bool

Clear cached instance registry from an object.

Returns *True* if cache was cleared, *False* if it wasn't needed.

class `coaster.sqlalchemy.registry.`**`InstanceRegistry`**(*registry*, *obj*)

Container for accessing registered items from an instance of the model.

Used internally by *`Registry`*. Returns a partial that will pass in an `obj` parameter when called.

`clear_cache`()

Clear cache from this registry.

class `coaster.sqlalchemy.registry.`**`RegistryMixin`**

Adds common registries to a model.

Included:

- `forms` registry, for WTForms forms
- `views` registry for view classes and helper functions
- `features` registry for feature availability test functions.

The forms registry passes the instance to the registered form as an `obj` keyword parameter. The other registries pass it as the first positional parameter.

1.25 Enhanced query and custom comparators

class `coaster.sqlalchemy.comparators.`**`Query`**(*entities*, *session=None*)

Extends flask_sqlalchemy.BaseQuery to add additional helper methods.

`isempty`()

Returns the equivalent of `not bool(query.count())` but using an efficient SQL EXISTS function, so the database stops counting after the first result is found.

`notempty`()

Returns the equivalent of `bool(query.count())` but using an efficient SQL EXISTS function, so the database stops counting after the first result is found.

`one_or_404`()

Extends `one_or_none()` to raise a 404 if no result is found. This method offers a safety net over `first_or_404()` as it helps identify poorly specified queries that could have returned more than one result.

class `coaster.sqlalchemy.comparators.`**`SplitIndexComparator`**(*expression*, *splitindex=None*)

Base class for comparators that support splitting a string and comparing with one of the split values.

`in_`(*other*)

Implement the `in` operator.

In a column context, produces the clause `column IN <other>`.

The given parameter `other` may be:

- A list of literal values, e.g.:

```
stmt.where(column.in_([1, 2, 3]))
```

In this calling form, the list of items is converted to a set of bound parameters the same length as the list given:

```
WHERE COL IN (?, ?, ?)
```

- A list of tuples may be provided if the comparison is against a `tuple_()` containing multiple expressions:

```
from sqlalchemy import tuple_
stmt.where(tuple_(col1, col2).in_([(1, 10), (2, 20), (3, 30)]))
```

- An empty list, e.g.:

```
stmt.where(column.in_([]))
```

In this calling form, the expression renders an "empty set" expression. These expressions are tailored to individual backends and are generally trying to get an empty SELECT statement as a subquery. Such as on SQLite, the expression is:

```
WHERE col IN (SELECT 1 FROM (SELECT 1) WHERE 1!=1)
```

Changed in version 1.4: empty IN expressions now use an execution-time generated SELECT subquery in all cases.

- A bound parameter, e.g. `bindparam()`, may be used if it includes the **:paramref:'.bindparam.expanding'** flag:

```
stmt.where(column.in_(bindparam('value', expanding=True)))
```

In this calling form, the expression renders a special non-SQL placeholder expression that looks like:

```
WHERE COL IN ([EXPANDING_value])
```

This placeholder expression is intercepted at statement execution time to be converted into the variable number of bound parameter form illustrated earlier. If the statement were executed as:

```
connection.execute(stmt, {"value": [1, 2, 3]})
```

The database would be passed a bound parameter for each value:

```
WHERE COL IN (?, ?, ?)
```

New in version 1.2: added "expanding" bound parameters

If an empty list is passed, a special "empty list" expression, which is specific to the database in use, is rendered. On SQLite this would be:

```
WHERE COL IN (SELECT 1 FROM (SELECT 1) WHERE 1!=1)
```

New in version 1.3: "expanding" bound parameters now support empty lists

- a `_expression.select()` construct, which is usually a correlated scalar select:

```
stmt.where(
    column.in_(
        select(othertable.c.y).
        where(table.c.x == othertable.c.x)
    )
)
```

In this calling form, `ColumnOperators.in_()` renders as given:

```
WHERE COL IN (SELECT othertable.y
FROM othertable WHERE othertable.x = table.x)
```

Parameters **`other`** – a list of literals, a `_expression.select()` construct, or a `bindparam()` construct that includes the **:paramref:'.bindparam.expanding'** flag set to True.

class `coaster.sqlalchemy.comparators.`**`SqlSplitIdComparator`**(*expression*, *splitindex=None*)

Allows comparing an id value with a column, useful mostly because of the splitindex feature, which splits an incoming string along the - character and picks one of the splits for comparison.

class `coaster.sqlalchemy.comparators.`**`SqlUuidHexComparator`**(*expression*, *splitindex=None*)

Allows comparing UUID fields with hex representations of the UUID

class `coaster.sqlalchemy.comparators.`**`SqlUuidB64Comparator`**(*expression*, *splitindex=None*)

Allows comparing UUID fields with URL-safe Base64 (BUID) representations of the UUID

class `coaster.sqlalchemy.comparators.`**`SqlUuidB58Comparator`**(*expression*, *splitindex=None*)

Allows comparing UUID fields with Base58 representations of the UUID

1.26 States and transitions

`StateManager` wraps a SQLAlchemy column with a *`LabeledEnum`* to facilitate state inspection, and to control state change via transitions. Sample usage:

```
class MY_STATE(LabeledEnum):
    DRAFT = (0, "Draft")
    PENDING = (1, 'pending', "Pending")
    PUBLISHED = (2, "Published")

    UNPUBLISHED = {DRAFT, PENDING}
```

(continues on next page)

(continued from previous page)

```
# Classes can have more than one state variable
class REVIEW_STATE(LabeledEnum):
    UNSUBMITTED = (0, "Unsubmitted")
    PENDING = (1, "Pending")
    REVIEWED = (2, "Reviewed")


class MyPost(BaseMixin, db.Model):
    __tablename__ = 'my_post'

    # The underlying state value columns
    # (more than one state variable can exist)
    _state = db.Column('state', db.Integer,
        StateManager.check_constraint('state', MY_STATE),
        default=MY_STATE.DRAFT, nullable=False)
    _reviewstate = db.Column('reviewstate', db.Integer,
        StateManager.check_constraint('state', REVIEW_STATE),
        default=REVIEW_STATE.UNSUBMITTED, nullable=False)

    # The state managers controlling the columns
    state = StateManager('_state', MY_STATE, doc="The post's state")
    reviewstate = StateManager('_reviewstate', REVIEW_STATE,
        doc="Reviewer's state")

    # Datetime for the additional states and transitions
    datetime = db.Column(db.DateTime, default=datetime.utcnow, nullable=False)

    # Additional states:

    # RECENT = PUBLISHED + in the last one hour
    state.add_conditional_state('RECENT', state.PUBLISHED,
        lambda post: post.datetime > datetime.utcnow() - timedelta(hours=1))

    # REDRAFTABLE = DRAFT or PENDING or RECENT
    state.add_state_group('REDRAFTABLE',
        state.DRAFT, state.PENDING, state.RECENT)

    # Transitions change FROM one state TO another, and can have
    # an additional if_ condition (a callable) that must return True
    @state.transition(state.DRAFT, state.PENDING, if_=reviewstate.UNSUBMITTED)
    def submit(self):
        pass

    # Transitions can coordinate across state managers. All of them
    # must be in a valid FROM state for the transition to be available.
    # Transitions can also specify arbitrary metadata such as this `title`
    # attribute (on any of the decorators). These are made available in a
    # `data` dictionary, accessible here as `publish.data`
    @state.transition(state.UNPUBLISHED, state.PUBLISHED, title="Publish")
    @reviewstate.transition(reviewstate.UNSUBMITTED, reviewstate.PENDING)
    def publish(self):
        # A transition can do additional housekeeping
        self.datetime = datetime.utcnow()

    # A transition can use a conditional state. The condition is evaluated
    # before the transition can proceed
    @state.transition(state.RECENT, state.PENDING)
```

(continues on next page)

(continued from previous page)

```
    @reviewstate.transition(reviewstate.PENDING, reviewstate.UNSUBMITTED)
    def undo(self):
        pass

    # Transitions can be defined FROM a group of states, but the TO
    # state must always be an individual state
    @state.transition(state.REDRAFTABLE, state.DRAFT)
    def redraft(self):
        pass

    # Transitions can abort without changing state, with or without raising
    # an exception to the caller
    @state.transition(state.REDRAFTABLE, state.DRAFT)
    def faulty_transition_examples(self):
        # Cancel the transition, but don't raise an exception to the caller
        raise AbortTransition()
        # Cancel the transition and return a result to the caller
        raise AbortTransition('failed')
        # Need to return a data structure? That works as well
        raise AbortTransition((False, 'faulty_failure'))
        raise AbortTransition({'status': 'error', 'error': 'faulty_failure'})
        # If any other exception is raised, it is passed up to the caller
        raise ValueError("Faulty transition")

    # The requires decorator specifies a transition that does not change
    # state. It can be used to limit a method's availability
    @state.requires(state.PUBLISHED)
    def send_email_alert(self):
        pass
```

1.26.1 Defining states and transitions

Adding a *StateManager* to the class links the underlying column (specified as a string) to the *LabeledEnum* (specified as an object). The *StateManager* is read-only and state can only be mutated via transitions. The *LabeledEnum* is not required after this point. All symbol names in it are available as attributes on the state manager henceforth (as instances of *ManagedState*).

Conditional states can be defined with *add_conditional_state()* as a combination of an existing state and a validator that receives the object (the instance of the class the StateManager is present on). This can be used to evaluate for additional conditions. For example, to distinguish between a static "published" state and a dynamic "recently published" state. *add_conditional_state()* also takes an optional `class_validator` parameter that is used for queries against the class (see below for query examples).

State groups can be defined with *add_state_group()*. These are similar to grouped values in a LabeledEnum, but can also contain conditional states, and are stored as instances of *ManagedStateGroup*. Grouped values in a *LabeledEnum* are more efficient for testing state against, so those should be preferred if the group does not contain a conditional state.

Transitions connect one managed state or group to another state (but not group). Transitions are defined as methods and decorated with *transition()*, which transforms them into instances of *StateTransition*, a callable class. If the transition raises an exception, the state change is aborted. Transitions may also abort without changing state using *AbortTransition*. Transitions have two additional attributes, *is_available*, a boolean property which indicates if the transition is currently available, and `data`, a dictionary that contains all additional parameters passed to the *transition()* decorator.

Transitions can be chained to coordinate a state change across state managers if the class has more than one. All state managers must be in a valid `from` state for the transition to be available. A dictionary of currently available transitions can be obtained from the state manager using the *`transitions()`* method.

1.26.2 Queries

The current state of the object can be retrieved by calling the state attribute or reading its `value` attribute:

```
post = MyPost(_state=MY_STATE.DRAFT)
post.state() == MY_STATE.DRAFT
post.state.value == MY_STATE.DRAFT
```

The label associated with the state value can be accessed from the `label` attribute:

```
post.state.label == "Draft"           # This is the string label from MY_STATE.DRAFT
post.submit()                         # Change state from DRAFT to PENDING
post.state.label.name == 'pending'    # Is the NameTitle tuple from MY_STATE.PENDING
post.state.label.title == "Pending"   # The title part of NameTitle
```

States can be tested by direct reference using the names they were originally defined with in the *`LabeledEnum`*:

```
post.state.DRAFT        # True
post.state.is_draft     # True (is_* attrs are lowercased aliases to states)
post.state.PENDING      # False (since it's a draft)
post.state.UNPUBLISHED  # True (grouped state values work as expected)
post.publish()          # Change state from DRAFT to PUBLISHED
post.state.RECENT       # True (calls the validator if the base state matches)
```

States can also be used for database queries when accessed from the class:

```
# Generates MyPost._state == MY_STATE.DRAFT
MyPost.query.filter(MyPost.state.DRAFT)

# Generates MyPost._state.in_(MY_STATE.UNPUBLISHED)
MyPost.query.filter(MyPost.state.UNPUBLISHED)

# Generates and_(MyPost._state == MY_STATE.PUBLISHED,
#     MyPost.datetime > datetime.utcnow() - timedelta(hours=1))
MyPost.query.filter(MyPost.state.RECENT)
```

This works because *`StateManager`*, *`ManagedState`* and *`ManagedStateGroup`* behave in three different ways, depending on context:

1. During class definition, the state manager returns the managed state. All methods on the state manager recognise these managed states and handle them appropriately.
2. After class definition, the state manager returns the result of calling the managed state instance. If accessed via the class, the managed state returns a SQLAlchemy filter condition.
3. After class definition, if accessed via an instance, the managed state returns itself wrapped in *`ManagedStateWrapper`* (which holds context for the instance). This is an object that evaluates to `True` if the state is active, `False` otherwise. It also provides pass-through access to all attributes of the managed state.

States can be changed via transitions, defined as methods with the *`transition()`* decorator. They add more power and safeguards over direct state value changes:

1. Original and final states can be specified, prohibiting arbitrary state changes.
2. The transition method can do additional validation and housekeeping.

3. Combined with the `with_roles()` decorator and `RoleMixin`, transitions provide access control for state changes.
4. Signals are raised before and after a successful transition, or in case of failures, allowing for the attempts to be logged.

class coaster.sqlalchemy.statemanager.**StateManager**(*propname*, *lenum*, *doc=None*)
Wraps a property with a `LabeledEnum` to facilitate state inspection and control state changes.

This is the main export of this module.

Parameters

- **propname** (`str`) – Name of the property that is to be wrapped
- **lenum** (`LabeledEnum`) – The `LabeledEnum` containing valid values
- **doc** (`str`) – Optional docstring

add_conditional_state(*name*, *state*, *validator*, *class_validator=None*, *cache_for=None*, *label=None*)
Add a conditional state that combines an existing state with a validator that must also pass. The validator receives the object on which the property is present as a parameter.

Parameters

- **name** (`str`) – Name of the new state
- **state** (`ManagedState`) – Existing state that this is based on
- **validator** – Function that will be called with the host object as a parameter
- **class_validator** – Function that will be called when the state is queried on the class instead of the instance. Falls back to `validator` if not specified. Receives the class as the parameter
- **cache_for** – Integer or function that indicates how long `validator`'s result can be cached (not applicable to `class_validator`). `None` implies no cache, `0` implies indefinite cache (until invalidated by a transition) and any other integer is the number of seconds for which to cache the assertion
- **label** – Label for this state (string or 2-tuple)

TODO: *cache_for*'s implementation is currently pending a test case demonstrating how it will be used.

add_state_group(*name*, **states*)
Add a group of managed states. Groups can be specified directly in the `LabeledEnum`. This method is only useful for grouping a conditional state with existing states. It cannot be used to form a group of groups.

Parameters

- **name** (`str`) – Name of this group
- **states** – `ManagedState` instances to be grouped together

static check_constraint(*column*, *lenum*, ***kwargs*)
Returns a SQL CHECK constraint string given a column name and a `LabeledEnum`.

Alembic may not detect the CHECK constraint when autogenerating migrations, so you may need to do this manually using the Python console to extract the SQL string:

```
from coaster.sqlalchemy import StateManager
from your_app.models import YOUR_ENUM
```

(continues on next page)

(continued from previous page)

```
print str(StateManager.check_constraint('your_column', YOUR_ENUM).sqltext)
```

Parameters

- **column** (*str*) – Column name
- **lenum** (LabeledEnum) – *LabeledEnum* to retrieve valid values from
- **kwargs** – Additional options passed to CheckConstraint

requires (*from_*, *if_=None*, ***data*)
Decorates a method that may be called if the given state is currently active. Registers a transition internally, but does not change the state.

Parameters

- **from** – Required state to allow this call (can be a state group)
- **if** – Validator(s) that, given the object, must all return True for the call to proceed
- **data** – Additional metadata, stored on the *StateTransition* object as a `data` attribute

transition (*from_*, *to*, *if_=None*, ***data*)
Decorates a method to transition from one state to another. The decorated method can accept any necessary parameters and perform additional processing, or raise an exception to abort the transition. If it returns without an error, the state value is updated automatically. Transitions may also abort without raising an exception using *AbortTransition*.

Parameters

- **from** – Required state to allow this transition (can be a state group)
- **to** – The state of the object after this transition (automatically set if no exception is raised)
- **if** – Validator(s) that, given the object, must all return True for the transition to proceed
- **data** – Additional metadata, stored on the *StateTransition* object as a `data` attribute

class coaster.sqlalchemy.statemanager.**ManagedState** (*name*, *statemanager*, *value*, *label=None*, *validator=None*, *class_validator=None*, *cache_for=None*)
Represents a state managed by a *StateManager*. Do not use this class directly. Use *add_conditional_state()* instead.

is_conditional
This is a conditional state

is_direct
This is a direct state (scalar state without a condition)

is_scalar
This is a scalar state (not a group of states, and may or may not have a condition)

class coaster.sqlalchemy.statemanager.**ManagedStateGroup** (*name*, *statemanager*, *states*)
Represents a group of managed states in a *StateManager*. Do not use this class directly. Use *add_state_group()* instead.

class `coaster.sqlalchemy.statemanager.`**`StateTransition`**(*func*, *statemanager*, *from_*, *to*, *if_=None*, *data=None*)

Helper for transitions from one state to another. Do not use this class directly. Use the `StateManager.transition()` decorator instead, which creates instances of this class.

To access the decorated function with `help()`, use `help(obj.func)`.

class `coaster.sqlalchemy.statemanager.`**`StateManagerWrapper`**(*statemanager*, *obj: Optional[T]*, *cls: Optional[Type[T]]*)

Wraps `StateManager` with the context of the containing object. Automatically constructed when a `StateManager` is accessed from either a class or an instance.

`bestmatch`()

Best matching current scalar state (direct or conditional), only applicable when accessed via an instance.

`current`()

All states and state groups that are currently active.

`group`(*items*, *keep_empty=False*)

Given an iterable of instances, groups them by state using `ManagedState` instances as dictionary keys. Returns a dict that preserves the order of states from the source `LabeledEnum`.

Parameters **`keep_empty`** (`bool`) – If `True`, empty states are included in the result

`label`

Label for the current state's value (using `bestmatch()`).

`transitions`(*current=True*)

Returns available transitions for the current state, as a dictionary of name: `StateTransitionWrapper`.

Parameters **`current`** (`bool`) – Limit to transitions available in `obj. current_access()`

`transitions_for`(*roles=None*, *actor=None*, *anchors=()*)

For use on `RoleMixin` classes: returns currently available transitions for the specified roles or actor as a dictionary of name: `StateTransitionWrapper`.

`value`

The current state value.

class `coaster.sqlalchemy.statemanager.`**`ManagedStateWrapper`**(*mstate*, *obj*, *cls=None*)

Wraps a `ManagedState` or `ManagedStateGroup` with an object or class, and otherwise provides transparent access to contents.

This class is automatically constructed by `StateManager`.

class `coaster.sqlalchemy.statemanager.`**`StateTransitionWrapper`**(*statetransition*, *obj*)

Wraps `StateTransition` with the context of the object it is accessed from. Automatically constructed by `StateTransition`.

`data`

Dictionary containing all additional parameters to the `transition()` decorator.

`is_available`

Property that indicates whether this transition is currently available.

exception coaster.sqlalchemy.statemanager.**StateTransitionError**(*description: Optional[str] = None, response: Optional[Response] = None*)

Raised if a transition is attempted from a non-matching state

exception coaster.sqlalchemy.statemanager.**AbortTransition**(*result=None*)

Transitions may raise *AbortTransition* to return without changing state. The parameter to this exception is returned as the transition's result.

This exception is a signal to *StateTransition* and will not be raised to the transition's caller.

Parameters **result** – Value to return to the transition's caller

coaster.sqlalchemy.statemanager.**transition_error = <blinker.base.NamedSignal object at 0x7**

Signal raised when a transition fails validation

coaster.sqlalchemy.statemanager.**transition_before = <blinker.base.NamedSignal object at 0x**

Signal raised before a transition (after validation)

coaster.sqlalchemy.statemanager.**transition_after = <blinker.base.NamedSignal object at 0x7**

Signal raised after a successful transition

coaster.sqlalchemy.statemanager.**transition_exception = <blinker.base.NamedSignal object at**

Signal raised when a transition raises an exception

1.27 Database session and instance

Coaster provides an instance of Flask-SQLAlchemy. If your app has models distributed across modules, you can use coaster's instance instead of creating a new module solely for a shared dependency. Some Hasgeek libraries like nodular and Flask-Commentease depend on this instance for their models.

coaster.db.**db**

Instance of SQLAlchemy

Caution: This instance is process-global. Your database models will be shared across all apps running in the same process. Do not run unrelated apps in the same process.

1.28 Natural language processing

Provides a wrapper around NLTK to extract named entities from HTML text:

```
from coaster.utils import text_blocks
from coaster.nlp import extract_named_entities

html = "<p>This is some HTML-formatted text.</p><p>In two paragraphs.</p>"
textlist = text_blocks(html)  # Returns a list of paragraphs.
entities = extract_named_entities(textlist)
```

coaster.nlp.**extract_named_entities**(*text_blocks*)

Return a list of named entities extracted from provided text blocks (list of text strings).

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

c

coaster.app, 1
coaster.assets, 20
coaster.auth, 35
coaster.db, 76
coaster.logger, 19
coaster.nlp, 76
coaster.sqlalchemy, 46
coaster.sqlalchemy.annotations, 65
coaster.sqlalchemy.columns, 54
coaster.sqlalchemy.comparators, 67
coaster.sqlalchemy.functions, 57
coaster.sqlalchemy.immutable_annotation, 66
coaster.sqlalchemy.mixins, 46
coaster.sqlalchemy.registry, 66
coaster.sqlalchemy.roles, 58
coaster.sqlalchemy.statemanager, 69
coaster.typing, 1
coaster.utils, 21
coaster.utils.classes, 32
coaster.utils.datetime, 28
coaster.utils.markdown, 30
coaster.utils.misc, 21
coaster.utils.text, 29
coaster.utils.tsquery, 31
coaster.views, 36
coaster.views.classview, 42
coaster.views.decorators, 37
coaster.views.misc, 36

Index

A

AbortTransition, 76
access_for() (*coaster.sqlalchemy.mixins.RoleMixin method*), 52
access_for() (*coaster.sqlalchemy.roles.RoleMixin method*), 62
actors_with() (*coaster.sqlalchemy.mixins.RoleMixin method*), 53
actors_with() (*coaster.sqlalchemy.roles.RoleMixin method*), 62
add() (*coaster.sqlalchemy.roles.LazyRoleSet method*), 60
add_auth_anchor() (*in module coaster.auth*), 36
add_auth_attribute() (*in module coaster.auth*), 36
add_conditional_state() (*coaster.sqlalchemy.statemanager.StateManager method*), 73
add_primary_relationship() (*in module coaster.sqlalchemy.functions*), 57
add_route_for() (*coaster.views.classview.ClassView class method*), 43
add_state_group() (*coaster.sqlalchemy.statemanager.StateManager method*), 73
add_template_filter() (*coaster.app.Flask method*), 3
add_template_global() (*coaster.app.Flask method*), 3
add_template_test() (*coaster.app.Flask method*), 3
add_url_rule() (*coaster.app.Flask method*), 3
after_request() (*coaster.views.classview.ClassView method*), 43
annotation_wrapper() (*in module coaster.sqlalchemy.annotations*), 66
app_context() (*coaster.app.Flask method*), 4
app_ctx_globals_class (*coaster.app.Flask attribute*), 4
AssetNotFound, 21
async_to_sync() (*coaster.app.Flask method*), 4
auto_find_instance_path() (*coaster.app.Flask method*), 4
auto_init_default() (*in module coaster.sqlalchemy.functions*), 58

B

base_domain_matches() (*in module coaster.utils.misc*), 22
BaseIdNameMixin (*class in coaster.sqlalchemy.mixins*), 50
BaseMixin (*class in coaster.sqlalchemy.mixins*), 48
BaseNameMixin (*class in coaster.sqlalchemy.mixins*), 48
BaseScopedIdMixin (*class in coaster.sqlalchemy.mixins*), 50
BaseScopedIdNameMixin (*class in coaster.sqlalchemy.mixins*), 50
BaseScopedNameMixin (*class in coaster.sqlalchemy.mixins*), 49
before_first_request() (*coaster.app.Flask method*), 4
before_first_request_funcs (*coaster.app.Flask attribute*), 4
before_request() (*coaster.views.classview.ClassView method*), 44
bestmatch() (*coaster.sqlalchemy.statemanager.StateManagerWrapper method*), 75
blueprints (*coaster.app.Flask attribute*), 5
buid (*coaster.sqlalchemy.mixins.UuidMixin attribute*), 51
buid() (*in module coaster.utils.misc*), 22
buid2uuid() (*in module coaster.utils.misc*), 22

C

cached() (*in module coaster.sqlalchemy.immutable_annotation*), 66

check_constraint() (*coaster.sqlalchemy.statemanager.StateManager static method*), 73
classmethodproperty (*class in coaster.utils.classes*), 35
ClassView (*class in coaster.views.classview*), 42
classview_for() (*coaster.sqlalchemy.mixins.UrlForMixin method*), 47
clear_cache() (*coaster.sqlalchemy.registry.InstanceRegistry method*), 67
clear_cache_for() (*coaster.sqlalchemy.registry.Registry method*), 67
coaster.app (*module*), 1
coaster.assets (*module*), 20
coaster.auth (*module*), 35
coaster.db (*module*), 76
coaster.logger (*module*), 19
coaster.nlp (*module*), 76
coaster.sqlalchemy (*module*), 46
coaster.sqlalchemy.annotations (*module*), 65
coaster.sqlalchemy.columns (*module*), 54
coaster.sqlalchemy.comparators (*module*), 67
coaster.sqlalchemy.functions (*module*), 57
coaster.sqlalchemy.immutable_annotation (*module*), 66
coaster.sqlalchemy.mixins (*module*), 46
coaster.sqlalchemy.registry (*module*), 66
coaster.sqlalchemy.roles (*module*), 58
coaster.sqlalchemy.statemanager (*module*), 69
coaster.typing (*module*), 1
coaster.utils (*module*), 21
coaster.utils.classes (*module*), 32
coaster.utils.datetime (*module*), 28
coaster.utils.markdown (*module*), 30
coaster.utils.misc (*module*), 21
coaster.utils.text (*module*), 29
coaster.utils.tsquery (*module*), 31
coaster.views (*module*), 36
coaster.views.classview (*module*), 42
coaster.views.decorators (*module*), 37
coaster.views.misc (*module*), 36
compress_whitespace() (*in module coaster.utils.text*), 30
config (*coaster.app.Flask attribute*), 5
config_class (*coaster.app.Flask attribute*), 5
configure() (*in module coaster.logger*), 19
coordinates (*coaster.sqlalchemy.mixins.CoordinatesMixin attribute*), 51
CoordinatesMixin (*class in coaster.sqlalchemy.mixins*), 51
copy() (*coaster.sqlalchemy.roles.LazyRoleSet method*), 60
cors() (*in module coaster.views.decorators*), 41
create_global_jinja_loader() (*coaster.app.Flask method*), 5
create_jinja_environment() (*coaster.app.Flask method*), 5
create_url_adapter() (*coaster.app.Flask method*), 5
current() (*coaster.sqlalchemy.statemanager.StateManagerWrapper method*), 75
current_access() (*coaster.sqlalchemy.mixins.RoleMixin method*), 53
current_access() (*coaster.sqlalchemy.roles.RoleMixin method*), 63
current_auth (*in module coaster.auth*), 36
current_handler (*coaster.views.classview.ClassView attribute*), 44
current_permissions (*coaster.sqlalchemy.mixins.PermissionMixin attribute*), 47
current_roles (*coaster.sqlalchemy.mixins.RoleMixin attribute*), 53
current_roles (*coaster.sqlalchemy.roles.RoleMixin attribute*), 63
current_view (*in module coaster.views.classview*), 42

D

data (*coaster.sqlalchemy.statemanager.StateTransitionWrapper attribute*), 75
db (*in module coaster.db*), 76
debug (*coaster.app.Flask attribute*), 5
declared_attr_roles() (*in module coaster.sqlalchemy.roles*), 65
default_config (*coaster.app.Flask attribute*), 5
deobfuscate_email() (*in module coaster.utils.text*), 30
discard() (*coaster.sqlalchemy.roles.LazyRoleSet method*), 60
dispatch_request() (*coaster.app.Flask method*), 5
dispatch_request() (*coaster.views.classview.ClassView method*), 44
dispatch_request() (*coaster.views.classview.ModelView method*), 45
do_teardown_appcontext() (*coaster.app.Flask method*), 6
do_teardown_request() (*coaster.app.Flask method*), 6
domain_namespace_match() (*in module coaster.utils.misc*), 22
DynamicAssociationProxy (*class in coaster.sqlalchemy.roles*), 61

E

emit() (*coaster.logger.SlackHandler method*), 19
emit() (*coaster.logger.TelegramHandler method*), 19
endpoint_for() (*in module coaster.views.misc*), 37
ensure_sync() (*coaster.app.Flask method*), 6
env (*coaster.app.Flask attribute*), 6
environment variable
    FLASK_DEBUG, 13
    FLASK_ENV, 6, 13
extensions (*coaster.app.Flask attribute*), 6
extract_named_entities() (*in module coaster.nlp*), 76

F

failsafe_add() (*in module coaster.sqlalchemy.functions*), 57
filtered_value() (*in module coaster.logger*), 20
FilteredValueIndicator (*class in coaster.logger*), 19
finalize_request() (*coaster.app.Flask method*), 7
Flask (*class in coaster.app*), 1
FLASK_DEBUG, 13
FLASK_ENV, 6, 13
for_tsquery() (*in module coaster.utils.tsquery*), 31
format() (*coaster.logger.LocalVarFormatter method*), 19
format_currency() (*in module coaster.utils.misc*), 22
formatException() (*coaster.logger.LocalVarFormatter method*), 19
full_dispatch_request() (*coaster.app.Flask method*), 7

G

get() (*coaster.sqlalchemy.mixins.BaseNameMixin class method*), 48
get() (*coaster.sqlalchemy.mixins.BaseScopedIdMixin class method*), 50
get() (*coaster.sqlalchemy.mixins.BaseScopedIdNameMixin class method*), 51
get() (*coaster.sqlalchemy.mixins.BaseScopedNameMixin class method*), 49
get_current_url() (*in module coaster.views.misc*), 37
get_email_domain() (*in module coaster.utils.misc*), 23
get_next_url() (*in module coaster.views.misc*), 37
getbool() (*in module coaster.utils.misc*), 23
got_first_request (*coaster.app.Flask attribute*), 7
group() (*coaster.sqlalchemy.statemanager.StateManagerWrapper method*), 75

H

handle_exception() (*coaster.app.Flask method*), 7
handle_http_exception() (*coaster.app.Flask method*), 7
handle_url_build_error() (*coaster.app.Flask method*), 8
handle_user_exception() (*coaster.app.Flask method*), 8
has_any() (*coaster.sqlalchemy.roles.LazyRoleSet method*), 60
has_coordinates (*coaster.sqlalchemy.mixins.CoordinatesMixin attribute*), 51
has_missing_coordinates (*coaster.sqlalchemy.mixins.CoordinatesMixin attribute*), 51

I

IdMixin (*class in coaster.sqlalchemy.mixins*), 47
immutable() (*in module coaster.sqlalchemy.immutable_annotation*), 66
ImmutableColumnError, 66
impl (*coaster.sqlalchemy.columns.JsonDict attribute*), 54
impl (*coaster.sqlalchemy.columns.UrlType attribute*), 56
in_() (*coaster.sqlalchemy.comparators.SplitIndexComparator method*), 68
init_app() (*coaster.views.classview.ClassView class method*), 44
init_app() (*in module coaster.app*), 18
init_app() (*in module coaster.logger*), 20
inject_url_defaults() (*coaster.app.Flask method*), 8
InspectableSet (*class in coaster.utils.classes*), 34
instance_path (*coaster.app.Flask attribute*), 8
InstanceLoader (*class in coaster.views.classview*), 46
InstanceRegistry (*class in coaster.sqlalchemy.registry*), 67
is_always_available (*coaster.views.classview.ClassView attribute*), 44
is_available (*coaster.sqlalchemy.statemanager.StateTransitionWrappe attribute*), 75
is_available() (*coaster.views.classview.ClassView method*), 44
is_collection() (*in module coaster.utils.misc*), 23
is_conditional (*coaster.sqlalchemy.statemanager.ManagedState attribute*), 74
is_direct (*coaster.sqlalchemy.statemanager.ManagedState attribute*), 74
is_scalar (*coaster.sqlalchemy.statemanager.ManagedState attribute*), 74
is_url_for() (*coaster.sqlalchemy.mixins.UrlForMixin class method*), 47

isempty() (*coaster.sqlalchemy.comparators.Query method*), 67
isoweek_datetime() (*in module coaster.utils.datetime*), 29
iter_blueprints() (*coaster.app.Flask method*), 8

J

jinja_env (*coaster.app.Flask attribute*), 8
jinja_environment (*coaster.app.Flask attribute*), 9
jinja_options (*coaster.app.Flask attribute*), 9
json_decoder (*coaster.app.Flask attribute*), 9
json_encoder (*coaster.app.Flask attribute*), 9
JsonDict (*class in coaster.sqlalchemy.columns*), 54
jsonp() (*in module coaster.views.misc*), 37

K

KeyRotationWrapper (*class in coaster.app*), 1

L

label (*coaster.sqlalchemy.statemanager.StateManagerWrapper attribute*), 75
LabeledEnum (*class in coaster.utils.classes*), 32
LazyRoleSet (*class in coaster.sqlalchemy.roles*), 60
load_dialect_impl() (*coaster.sqlalchemy.columns.JsonDict method*), 54
load_dialect_impl() (*coaster.sqlalchemy.columns.UUIDType method*), 55
load_model() (*in module coaster.views.decorators*), 38
load_models() (*in module coaster.views.decorators*), 39
loader() (*coaster.views.classview.ModelView method*), 45
LocalVarFormatter (*class in coaster.logger*), 19
log_exception() (*coaster.app.Flask method*), 9
logger (*coaster.app.Flask attribute*), 9

M

make_config() (*coaster.app.Flask method*), 9
make_default_options_response() (*coaster.app.Flask method*), 9
make_id() (*coaster.sqlalchemy.mixins.BaseScopedIdMixin method*), 50
make_name() (*coaster.sqlalchemy.mixins.BaseIdNameMixin method*), 50
make_name() (*coaster.sqlalchemy.mixins.BaseNameMixin method*), 49
make_name() (*coaster.sqlalchemy.mixins.BaseScopedIdNameMixin method*), 51
make_name() (*coaster.sqlalchemy.mixins.BaseScopedNameMixin method*), 49
make_name() (*in module coaster.utils.misc*), 24
make_response() (*coaster.app.Flask method*), 9
make_shell_context() (*coaster.app.Flask method*), 10
make_timestamp_columns() (*in module coaster.sqlalchemy.functions*), 57
ManagedState (*class in coaster.sqlalchemy.statemanager*), 74
ManagedStateGroup (*class in coaster.sqlalchemy.statemanager*), 74
ManagedStateWrapper (*class in coaster.sqlalchemy.statemanager*), 75
markdown() (*in module coaster.utils.markdown*), 30
md5sum() (*in module coaster.utils.misc*), 25
midnight_to_utc() (*in module coaster.utils.datetime*), 29
model (*coaster.views.classview.ModelView attribute*), 45
ModelView (*class in coaster.views.classview*), 44

N

name (*coaster.app.Flask attribute*), 10
name (*coaster.utils.classes.NameTitle attribute*), 32
namespace_from_url() (*in module coaster.utils.misc*), 25
NameTitle (*class in coaster.utils.classes*), 32
nary_op() (*in module coaster.utils.misc*), 25
newpin() (*in module coaster.utils.misc*), 25
newsecret() (*in module coaster.utils.misc*), 26
NoIdMixin (*class in coaster.sqlalchemy.mixins*), 48
normalize_spaces() (*in module coaster.utils.text*), 30
normalize_spaces_multiline() (*in module coaster.utils.text*), 30
notempty() (*coaster.sqlalchemy.comparators.Query method*), 67
nullint() (*in module coaster.utils.misc*), 26
nullstr() (*in module coaster.utils.misc*), 26

O

offered_roles (*coaster.sqlalchemy.roles.RoleGrantABC attribute*), 60
one_or_404() (*coaster.sqlalchemy.comparators.Query method*), 67
open_instance_resource() (*coaster.app.Flask method*), 10

P

parse_isoformat() (*in module coaster.utils.datetime*), 29
ParseError (*in module coaster.utils.datetime*), 29
permanent_session_lifetime (*coaster.app.Flask attribute*), 10
PermissionMixin (*class in coaster.sqlalchemy.mixins*), 47

permissions() (*coaster.sqlalchemy.mixins.BaseScopedIdMixin method*), 50
permissions() (*coaster.sqlalchemy.mixins.BaseScopedNameMixin method*), 49
permissions() (*coaster.sqlalchemy.mixins.PermissionMixin method*), 47
pprint_with_indent() (*in module coaster.logger*), 20
preprocess_request() (*coaster.app.Flask method*), 11
preserve_context_on_exception (*coaster.app.Flask attribute*), 11
process_bind_param() (*coaster.sqlalchemy.columns.JsonDict method*), 54
process_bind_param() (*coaster.sqlalchemy.columns.UrlType method*), 56
process_bind_param() (*coaster.sqlalchemy.columns.UUIDType method*), 55
process_literal_param() (*coaster.sqlalchemy.columns.UUIDType method*), 55
process_response() (*coaster.app.Flask method*), 11
process_result_value() (*coaster.sqlalchemy.columns.JsonDict method*), 54
process_result_value() (*coaster.sqlalchemy.columns.UrlType method*), 56
process_result_value() (*coaster.sqlalchemy.columns.UUIDType method*), 56
propagate_exceptions (*coaster.app.Flask attribute*), 11
python_type (*coaster.sqlalchemy.columns.UUIDType attribute*), 56

Q

Query (*class in coaster.sqlalchemy.comparators*), 67
query (*coaster.views.classview.ModelView attribute*), 45
query_class (*coaster.sqlalchemy.mixins.IdMixin attribute*), 47
query_class (*coaster.sqlalchemy.mixins.TimestampMixin attribute*), 47

R

raise_routing_exception() (*coaster.app.Flask method*), 11
register_blueprint() (*coaster.app.Flask method*), 11
register_endpoint() (*coaster.sqlalchemy.mixins.UrlForMixin class method*), 48
register_view_for() (*coaster.sqlalchemy.mixins.UrlForMixin class method*), 48
Registry (*class in coaster.sqlalchemy.registry*), 67
RegistryMixin (*class in coaster.sqlalchemy.mixins*), 53
RegistryMixin (*class in coaster.sqlalchemy.registry*), 67
render_with() (*in module coaster.views.decorators*), 40
RepeatValueIndicator (*class in coaster.logger*), 19
request_class (*coaster.app.Flask attribute*), 12
request_context() (*coaster.app.Flask method*), 12
request_has_auth() (*in module coaster.auth*), 36
requestargs() (*in module coaster.views.decorators*), 37
requestform() (*in module coaster.views.decorators*), 38
requestquery() (*in module coaster.views.decorators*), 38
RequestTypeError, 37
RequestValueError, 37
require() (*coaster.assets.VersionedAssets method*), 21
require_one_of() (*in module coaster.utils.misc*), 26
requires() (*coaster.sqlalchemy.statemanager.StateManager method*), 74
requires_permission() (*in module coaster.views.decorators*), 41
requires_roles() (*in module coaster.views.classview*), 46
reserved_names (*coaster.sqlalchemy.mixins.BaseNameMixin attribute*), 49
reserved_names (*coaster.sqlalchemy.mixins.BaseScopedNameMixin attribute*), 49
response_class (*coaster.app.Flask attribute*), 12
RoleAccessProxy (*class in coaster.sqlalchemy.roles*), 60
RoleGrantABC (*class in coaster.sqlalchemy.roles*), 60
RoleMixin (*class in coaster.sqlalchemy.mixins*), 52
RoleMixin (*class in coaster.sqlalchemy.roles*), 61
roles_for() (*coaster.sqlalchemy.mixins.RoleMixin method*), 53
roles_for() (*coaster.sqlalchemy.roles.RoleMixin method*), 63
RotatingKeySecureCookieSessionInterface (*class in coaster.app*), 1
route() (*in module coaster.views.classview*), 45
route_model_map (*coaster.views.classview.ModelView attribute*), 45

`rulejoin()` (*in module coaster.views.classview*), 42
`run()` (*coaster.app.Flask method*), 12

S

`sanitize_html()` (*in module coaster.utils.text*), 30
`secret_key` (*coaster.app.Flask attribute*), 13
`select_jinja_autoescape()` (*coaster.app.Flask method*), 13
`send_file_max_age_default` (*coaster.app.Flask attribute*), 13
`session_cookie_name` (*coaster.app.Flask attribute*), 13
`session_interface` (*coaster.app.Flask attribute*), 13
`shell_context_processor()` (*coaster.app.Flask method*), 13
`shell_context_processors` (*coaster.app.Flask attribute*), 13
`short_title` (*coaster.sqlalchemy.mixins.BaseScopedNameMixin attribute*), 49
`should_ignore_error()` (*coaster.app.Flask method*), 13
`SimpleDecorator` (*in module coaster.typing*), 1
`SimpleSpec` (*class in coaster.assets*), 20
`simplify_text()` (*in module coaster.utils.text*), 30
`SlackHandler` (*class in coaster.logger*), 19
`sorted_timezones()` (*in module coaster.utils.datetime*), 29
`SplitIndexComparator` (*class in coaster.sqlalchemy.comparators*), 68
`SqlSplitIdComparator` (*class in coaster.sqlalchemy.comparators*), 69
`SqlUuidB58Comparator` (*class in coaster.sqlalchemy.comparators*), 69
`SqlUuidB64Comparator` (*class in coaster.sqlalchemy.comparators*), 69
`SqlUuidHexComparator` (*class in coaster.sqlalchemy.comparators*), 69
`StateManager` (*class in coaster.sqlalchemy.statemanager*), 73
`StateManagerWrapper` (*class in coaster.sqlalchemy.statemanager*), 75
`StateTransition` (*class in coaster.sqlalchemy.statemanager*), 74
`StateTransitionError`, 75
`StateTransitionWrapper` (*class in coaster.sqlalchemy.statemanager*), 75

T

`teardown_appcontext()` (*coaster.app.Flask method*), 14
`teardown_appcontext_funcs` (*coaster.app.Flask attribute*), 14
`TelegramHandler` (*class in coaster.logger*), 19
`template_filter()` (*coaster.app.Flask method*), 14
`template_global()` (*coaster.app.Flask method*), 14
`template_test()` (*coaster.app.Flask method*), 14
`templates_auto_reload` (*coaster.app.Flask attribute*), 15
`test_cli_runner()` (*coaster.app.Flask method*), 15
`test_cli_runner_class` (*coaster.app.Flask attribute*), 15
`test_client()` (*coaster.app.Flask method*), 15
`test_client_class` (*coaster.app.Flask attribute*), 16
`test_request_context()` (*coaster.app.Flask method*), 16
`testing` (*coaster.app.Flask attribute*), 17
`text_blocks()` (*in module coaster.utils.text*), 30
`TimestampMixin` (*class in coaster.sqlalchemy.mixins*), 47
`title` (*coaster.utils.classes.NameTitle attribute*), 32
`title_for_name` (*coaster.sqlalchemy.mixins.BaseIdNameMixin attribute*), 50
`title_for_name` (*coaster.sqlalchemy.mixins.BaseNameMixin attribute*), 49
`title_for_name` (*coaster.sqlalchemy.mixins.BaseScopedIdNameMixin attribute*), 51
`title_for_name` (*coaster.sqlalchemy.mixins.BaseScopedNameMixin attribute*), 49
`transition()` (*coaster.sqlalchemy.statemanager.StateManager method*), 74
`transition_after` (*in module coaster.sqlalchemy.statemanager*), 76
`transition_before` (*in module coaster.sqlalchemy.statemanager*), 76
`transition_error` (*in module coaster.sqlalchemy.statemanager*), 76
`transition_exception` (*in module coaster.sqlalchemy.statemanager*), 76
`transitions()` (*coaster.sqlalchemy.statemanager.StateManagerWrappe method*), 75
`transitions_for()` (*coaster.sqlalchemy.statemanager.StateManagerWrapper method*), 75
`trap_http_exception()` (*coaster.app.Flask method*), 17
`try_trigger_before_first_request_functions()` (*coaster.app.Flask method*), 17

U

`ulstrip()` (*in module coaster.utils.text*), 30
`unicode_http_header()` (*in module coaster.utils.misc*), 26
`update_template_context()` (*coaster.app.Flask method*), 17
`upsert()` (*coaster.sqlalchemy.mixins.BaseNameMixin class method*), 49

upsert() (*coaster.sqlalchemy.mixins.BaseScopedNameMixin class method*), 49
url_build_error_handlers (*coaster.app.Flask attribute*), 17
url_change_check() (*in module coaster.views.classview*), 45
url_for() (*coaster.sqlalchemy.mixins.UrlForMixin method*), 48
url_for_endpoints (*coaster.sqlalchemy.mixins.UrlForMixin attribute*), 48
url_id (*coaster.sqlalchemy.mixins.IdMixin attribute*), 47
url_id_name (*coaster.sqlalchemy.mixins.BaseIdNameMixin attribute*), 50
url_id_name (*coaster.sqlalchemy.mixins.BaseScopedIdNameMixin attribute*), 51
url_map (*coaster.app.Flask attribute*), 17
url_map_class (*coaster.app.Flask attribute*), 18
url_name (*coaster.sqlalchemy.mixins.BaseIdNameMixin attribute*), 50
url_name (*coaster.sqlalchemy.mixins.BaseScopedIdNameMixin attribute*), 51
url_name_uuid_b58 (*coaster.sqlalchemy.mixins.BaseIdNameMixin attribute*), 50
url_name_uuid_b58 (*coaster.sqlalchemy.mixins.BaseScopedIdNameMixin attribute*), 51
url_parser (*coaster.sqlalchemy.columns.UrlType attribute*), 57
url_rule_class (*coaster.app.Flask attribute*), 18
UrlChangeCheck (*class in coaster.views.classview*), 46
UrlDict (*class in coaster.sqlalchemy.mixins*), 47
UrlForMixin (*class in coaster.sqlalchemy.mixins*), 47
UrlForView (*class in coaster.views.classview*), 46
urls (*coaster.sqlalchemy.mixins.UrlForMixin attribute*), 48
UrlType (*class in coaster.sqlalchemy.columns*), 56
urstrip() (*in module coaster.utils.text*), 30
use_x_sendfile (*coaster.app.Flask attribute*), 18
ustrip() (*in module coaster.utils.text*), 30
utcnow() (*in module coaster.utils.datetime*), 29
uuid1mc() (*in module coaster.utils.misc*), 27
uuid1mc_from_datetime() (*in module coaster.utils.misc*), 27
uuid2buid() (*in module coaster.utils.misc*), 27
uuid_b58 (*coaster.sqlalchemy.mixins.UuidMixin attribute*), 51
uuid_b58() (*in module coaster.utils.misc*), 27
uuid_b64 (*coaster.sqlalchemy.mixins.UuidMixin attribute*), 52
uuid_b64() (*in module coaster.utils.misc*), 27
uuid_from_base58() (*in module coaster.utils.misc*), 28
uuid_from_base64() (*in module coaster.utils.misc*), 28
uuid_hex (*coaster.sqlalchemy.mixins.UuidMixin attribute*), 52
uuid_to_base58() (*in module coaster.utils.misc*), 28
uuid_to_base64() (*in module coaster.utils.misc*), 28
UuidMixin (*class in coaster.sqlalchemy.mixins*), 51
UUIDType (*class in coaster.sqlalchemy.columns*), 55

V

valid_username() (*in module coaster.utils.misc*), 28
value (*coaster.sqlalchemy.statemanager.StateManagerWrapper attribute*), 75
VersionedAssets (*class in coaster.assets*), 20
view_args (*coaster.views.classview.ClassView attribute*), 44
view_for() (*coaster.sqlalchemy.mixins.UrlForMixin method*), 48
view_for_endpoints (*coaster.sqlalchemy.mixins.UrlForMixin attribute*), 48
viewdata() (*in module coaster.views.classview*), 45

W

with_roles() (*in module coaster.sqlalchemy.roles*), 63
wsgi_app() (*coaster.app.Flask method*), 18