Coaster Documentation

Release 0.6.dev1

Hasgeek

Jan 06, 2021

Contents

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 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*, *static_url_path=None*, *static_folder='static'*, *static_host=None*, *host_matching=False*, *subdomain_matching=False*, *template_folder='templates'*, *instance_path=None*, *instance_relative_config=False*, *root_path=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** – Flask by default will automatically calculate the path to the root of the application. In certain situations this cannot be achieved (for instance if the package is a Python 3 namespace package) and needs to be manually defined.

add_template_filter(*f*, *name=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*, *name=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*, *name=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*, *endpoint=None*, *view_func=None*, *provide_automatic_options=None*, ***options*)
Connects a URL rule. Works exactly like the *route()* decorator. If a view_func is provided it will be registered with the endpoint.

Basically this example:

```
@app.route('/')
def index():
    pass
```

Is equivalent to the following:

```
def index():
    pass
app.add_url_rule('/', 'index', index)
```

If the view_func is not provided you will need to connect the endpoint to a view function like so:

```
app.view_functions['index'] = index
```

Internally *route()* invokes *add_url_rule()* so if you want to customize the behavior via subclassing you only need to change this method.

For more information refer to URL Route Registrations.

Changed in version 0.2: *view_func* parameter added.

Changed in version 0.6: `OPTIONS` is added automatically as method.

Parameters

- **rule** – the URL rule as string
- **endpoint** – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint

- **view_func** – the function to call when serving a request to the provided endpoint
- **provide_automatic_options** – controls whether the OPTIONS method should be added automatically. This can also be controlled by setting the view_func.provide_automatic_options = False before adding the rule.
- **options** – the options to be forwarded to the underlying Rule object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (GET, POST etc.). By default a rule just listens for GET (and implicitly HEAD). Starting with Flask 0.6, OPTIONS is implicitly added and handled by the standard request handling.

after_request(*f*)
Register a function to be run after each request.

Your function must take one parameter, an instance of *response_class* and return a new response object or the same (see *process_response()*).

As of Flask 0.7 this function might not be executed at the end of the request in case an unhandled exception occurred.

after_request_funcs = None
A dictionary with lists of functions that should be called after each request. The key of the dictionary is the name of the blueprint this function is active for, None for all requests. This can for example be used to close database connections. To register a function here, use the *after_request()* decorator.

app_context()
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

auto_find_instance_path()
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*)
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.

before_request(*f*)
Registers a function to run before each request.

For example, this can be used to open a database connection, or to load the logged in user from the session.

The function will be called without any arguments. If it 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.

before_request_funcs = None
A dictionary with lists of functions that will be called at the beginning of each request. The key of the dictionary is the name of the blueprint this function is active for, or `None` for all requests. To register a function, use the *`before_request()`* decorator.

blueprints = None
all the attached blueprints in a dictionary by name. Blueprints can be attached multiple times so this dictionary does not tell you how often they got 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`

context_processor(*f*)
Registers a template context processor function.

create_global_jinja_loader()
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()
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*)
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()
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=<object object>*)
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=<object object>*)
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.

endpoint(*endpoint*)
A decorator to register a function as an endpoint. Example:

```
@app.endpoint('example.endpoint')
def example():
    return "example"
```

Parameters **endpoint** – the name of the endpoint

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'`

error_handler_spec = None
A dictionary of all registered error handlers. The key is `None` for error handlers active on the application, otherwise the key is the name of the blueprint. Each key points to another dictionary where the key is the status code of the http exception. The special key `None` points to a list of tuples where the first item is the class for the instance check and the second the error handler function.

To register an error handler, use the *errorhandler()* decorator.

errorhandler(*code_or_exception*)
Register a function to handle errors by code or exception class.

A decorator that is used to register a function given an error code. Example:

```
@app.errorhandler(404)
def page_not_found(error):
    return 'This page does not exist', 404
```

You can also register handlers for arbitrary exceptions:

```
@app.errorhandler(DatabaseError)
def special_exception_handler(error):
    return 'Database connection failed', 500
```

New in version 0.7: Use *register_error_handler()* instead of modifying *error_handler_spec* directly, for application wide error handlers.

New in version 0.7: One can now additionally also register custom exception types that do not necessarily have to be a subclass of the `HTTPException` class.

Parameters **code_or_exception** – the code as integer for the handler, or an arbitrary exception

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. For backwards compatibility extensions should register themselves like this:

```
if not hasattr(app, 'extensions'):
    app.extensions = {}
app.extensions['extensionname'] = SomeObject()
```

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*, *from_error_handler=False*)
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()
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*)
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`.

Note: Prior to Werkzeug 1.0.0, `InternalServerError` will not always have an `original_exception` attribute. Use `getattr(e, "original_exception", None)` to simulate the behavior for compatibility.

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*)
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*, *endpoint*, *values*)
Handle `BuildError` on `url_for()`.

handle_user_exception(*e*)
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.

import_name = None
The name of the package or module that this app belongs to. Do not change this once it is set by the constructor.

inject_url_defaults(*endpoint*, *values*)
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()
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 = {'extensions': ['jinja2.ext.autoescape', 'jinja2.ext.with_']}
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*)
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=False*)
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()
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_null_session()
Creates a new instance of a missing session. Instead of overriding this method we recommend replacing the *session_interface*.

New in version 0.7.

make_response(*rv*)
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 (unicode in Python 2) A response object is created with the string encoded to UTF-8 as the body.

bytes (str in Python 2) 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()
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*, *mode='rb'*)
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'.

open_session(*request*)
Creates or opens a new session. Default implementation stores all session data in a signed cookie. This requires that the *secret_key* is set. Instead of overriding this method we recommend replacing the *session_interface*.

Parameters **request** – an instance of *request_class*.

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()
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*)
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*)
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*, ***options*)
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.

New in version 0.7.

register_error_handler(*code_or_exception*, *f*)

Alternative error attach function to the *errorhandler()* decorator that is more straightforward to use for non decorator usage.

New in version 0.7.

request_class

alias of `flask.wrappers.Request`

request_context(*environ*)

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`

root_path = None

Absolute path to the package on the filesystem. Used to look up resources contained in the package.

route(*rule*, ***options*)

A decorator that is used to register a view function for a given URL rule. This does the same thing as *add_url_rule()* but is intended for decorator usage:

```
@app.route('/')
def index():
    return 'Hello World'
```

For more information refer to URL Route Registrations.

Parameters

- **rule** – the URL rule as string
- **endpoint** – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint
- **options** – the options to be forwarded to the underlying `Rule` object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (`GET`, `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`).

Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling.

run(*host=None*, *port=None*, *debug=None*, *load_dotenv=True*, ***options*)
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 Deployment Options 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.

save_session(*session*, *response*)
Saves the session if it needs updates. For the default implementation, check *open_session()*. Instead of overriding this method we recommend replacing the *session_interface*.

Parameters

- **session** – the session to be saved (a `SecureCookie` object)

- **response** – an instance of *response_class*

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*)
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 which is used as default cache_timeout for the send_file() functions. The default is 12 hours.

This attribute can also be configured from the config with the SEND_FILE_MAX_AGE_DEFAULT configuration key. This configuration variable can also be set with an integer value used as seconds. Defaults to timedelta(hours=12)

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*)
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*)
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*)
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.

teardown_request(*f*)

Register a function to be run at the end of each request, regardless of whether there was an exception or not. These functions are executed when the request context is popped, even if not an actual request was performed.

Example:

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

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

Generally teardown functions must take every necessary step to avoid that they will fail. If they do execute code that might fail they will have to surround the execution of these code by try/except statements and log occurring errors.

When a teardown function was called because of an exception it will be passed an error object.

The return values of teardown functions are ignored.

Debug Note

In debug mode Flask will not tear down a request on an exception immediately. Instead it will keep it alive so that the interactive debugger can still access it. This behavior can be controlled by the `PRESERVE_CONTEXT_ON_EXCEPTION` configuration variable.

---

teardown_request_funcs = None

A dictionary with lists of functions that are called after each request, even if an exception has occurred. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. These functions are not allowed to modify the request, and their return values are ignored. If an exception occurred while processing the request, it gets passed to each teardown_request function. To register a function here, use the *teardown_request()* decorator.

New in version 0.7.

template_context_processors = None
A dictionary with list of functions that are called without argument to populate the template context. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. Each returns a dictionary that the template context is updated with. To register a function here, use the *context_processor()* decorator.

template_filter(*name=None*)
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_folder = None
Location of the template files to be added to the template lookup. `None` if templates should not be added.

template_global(*name=None*)
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=None*)
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*)
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=True*, ***kwargs*)
Creates a test client for this application. For information about unit testing head over to Testing Flask Applications.

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*)
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*)

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()
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*)
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_default_functions = None
A dictionary with lists of functions that can be used as URL value preprocessors. The key `None` here is used for application wide callbacks, otherwise the key is the name of the blueprint. Each of these functions has the chance to modify the dictionary of URL values before they are used as the keyword arguments of the view function. For each function registered this one should also provide a *url_defaults()* function that adds the parameters automatically again that were removed that way.

New in version 0.7.

url_defaults(*f*)
Callback function for URL defaults for all view functions of the application. It's called with the endpoint and values and should update the values passed in place.

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:

```
from werkzeug.routing import BaseConverter

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

url_value_preprocessor(*f*)
Register a URL value preprocessor function for all view functions in the application. These functions will be called before the *before_request()* functions.

The function can modify the values captured from the matched url before they are passed to the view. For example, this can be used to pop a common language code value and place it in `g` rather than pass it to every view.

The function is passed the endpoint name and values dict. The return value is ignored.

url_value_preprocessors = None
A dictionary with lists of functions that are called before the *before_request_funcs* functions. The key of the dictionary is the name of the blueprint this function is active for, or `None` for all requests. To register a function, use *url_value_preprocessor()*.

New in version 0.7.

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`.

view_functions = None
A dictionary of all view functions registered. The keys will be function names which are also used to generate URLs and the values are the function objects themselves. To register a view function, use the *route()* decorator.

wsgi_app(*environ*, *start_response*)
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.

class coaster.app.**SandboxedFlask**(*import_name*, *static_url_path=None*, *static_folder='static'*, *static_host=None*, *host_matching=False*, *subdomain_matching=False*, *template_folder='templates'*, *instance_path=None*, *instance_relative_config=False*, *root_path=None*)
Flask with a sandboxed Jinja2 environment, for when your app's templates need sandboxing. Useful when your app works with externally provided templates:

```
from coaster.app import SandboxedFlask
app = SandboxedFlask(__name__)
```

create_jinja_environment()
Creates the Jinja2 environment based on `jinja_options` and `select_jinja_autoescape()`. Since 0.7 this also adds the Jinja2 globals and filters after initialization. Override this function to customize the behavior.

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*)

1.2 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.**LocalVarFormatter**(*fmt=None*, *datefmt=None*, *style='%'*)
Custom log formatter that logs 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*)
Format and return the specified exception information as a string.

This default implementation just uses traceback.print_exception()

class coaster.logger.**SMSHandler**(*app_name*, *exotel_sid*, *exotel_token*, *exotel_from*, *twilio_sid*, *twilio_token*, *twilio_from*, *phonenumbers*)
Custom logging handler to send SMSes to admins

emit(*record*)
Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

class coaster.logger.**SlackHandler**(*app_name*, *webhooks*)
Custom logging handler to post error reports to Slack.

emit(*record*)
Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

class coaster.logger.**TelegramHandler**(*app_name*, *chatid*, *apikey*)
Custom logging handler to report errors to a Telegram chat

emit(*record*)
Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

coaster.logger.**configure**(*app*)
Enables 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
- **`ADMIN_NUMBERS`: List of mobile numbers of admin to send SMS alerts. Requires** the following values too
- `SMS_EXOTEL_SID`: Exotel SID for Indian numbers (+91 prefix)
- `SMS_EXOTEL_TOKEN`: Exotel token
- `SMS_EXOTEL_FROM`: Exotel sender's number
- `SMS_TWILIO_SID`: Twilio SID for non-Indian numbers
- `SMS_TWILIO_TOKEN`: Twilio token
- `SMS_TWILIO_FROM`: Twilio sender's number

Format for `SLACK_LOGGING_WEBHOOKS`:

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

coaster.logger.**init_app**(*app*)
Enables 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
- **`ADMIN_NUMBERS`: List of mobile numbers of admin to send SMS alerts. Requires** the following values too

- `SMS_EXOTEL_SID`: Exotel SID for Indian numbers (+91 prefix)
- `SMS_EXOTEL_TOKEN`: Exotel token
- `SMS_EXOTEL_FROM`: Exotel sender's number
- `SMS_TWILIO_SID`: Twilio SID for non-Indian numbers
- `SMS_TWILIO_TOKEN`: Twilio token
- `SMS_TWILIO_FROM`: Twilio sender's number

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.

1.3 App management script

Coaster provides a Flask-Script-based manage.py with common management functions. To use in your Flask app, create a `manage.py` with this boilerplate:

```
from coaster.manage import manager, init_manager
from yourapp import app, db

@manager.command
def my_app_command()
    print "Hello!"

if __name__ == "__main__":
    # If you have no custom commands, you can skip importing manager
    # and use the return value from init_manager
    manager = init_manager(app, db)
    manager.run()
```

To see all available commands:

```
$ python manage.py --help
```

`coaster.manage.`**`alembic_table_metadata`**()
Returns SQLAlchemy metadata and Alembic version table

`coaster.manage.`**`createdb`**()
Create database tables from sqlalchemy models

`coaster.manage.`**`dropdb`**()
Drop database tables

`coaster.manage.`**`init_manager`**(*app*, *db*, ***kwargs*)
Initialise Manager

Parameters

- **app** – Flask app object

- **kwargs** – Additional keyword arguments to be made available as shell context

Parm db db instance

coaster.manage.**set_alembic_revision**(*path=None*)
Create/Update alembic table to latest revision number

coaster.manage.**shell_context**()
Supplies context variables for the shell

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.**check_password**(*reference*, *attempt*)
Compare a reference password with the user attempt.

```
>>> check_password('{PLAIN}foo', 'foo')
True
>>> check_password(u'{PLAIN}bar', 'bar')
True
>>> check_password(u'{UNKNOWN}baz', 'baz')
False
>>> check_password(u'no-encoding', u'no-encoding')
False
>>> check_password(u'{SSHA}q/uVU8r15k/9QhRi92CWUwMJu2DM6TUSpp25', u're-foo')
True
>>> check_password(u'{BCRYPT}$2b$12$NfKivgz7njR3/rWZ56EsDe7..PPum.fcmFLbdkbP.'
...   'chtMTcS1s01C', 'foo')
True
```

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)
'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({})
False
>>> is_collection({}.keys())
True
>>> 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(u"How's that?")
'hows-that'
>>> make_name(u'K & D')
'k-d'
>>> make_name('billion+ pageviews')
'billion-pageviews'
>>> make_name(u' slug!')
'hindii-slug'
>>> make_name(u'Talk in español, Kiswahili,  and  too.', maxlength=250)
u'talk-in-espanol-kiswahili-guang-zhou-hua-and-asmiiyaa-too'
>>> make_name(u'__name__', delim=u'_')
'name'
>>> make_name(u'how_about_this', delim=u'_')
'how_about_this'
>>> make_name(u'and-that', delim=u'_')
'and_that'
>>> make_name(u'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'
>>> 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(u'Lnkran')
'lankaran'
>>> make_name(u'example@example.com')
'example-example-com'
>>> make_name('trailing-delimiter', maxlength=10)
```

(continues on next page)

(continued from previous page)

```
'trailing-d'
>>> make_name('trailing-delimiter', maxlength=9)
'trailing'
>>> make_name('''test this
... newline''')
'test-this-newline'
>>> make_name(u"testing an emoji")
u'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(u"feedback;\x00")
u'feedback'
```

coaster.utils.misc.**make_password**(*password*, *encoding='BCRYPT'*)

Make a password with PLAIN, SSHA or BCRYPT (default) encoding.

```
>>> make_password('foo', encoding='PLAIN')
'{PLAIN}foo'
>>> make_password(u're-foo', encoding='SSHA')[:6]
'{SSHA}'
>>> make_password(u're-foo')[:8]
'{BCRYPT}'
>>> make_password('foo') == make_password('foo')
False
```

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

Return md5sum of data as a 32-character string.

```
>>> md5sum('random text')
'd9b9bec3f4cc5482e7c5ef43143e563a'
>>> md5sum(u'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()
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?=') == u'p\xf6stal'
True
```

(continues on next page)

(continued from previous page)

```
>>> unicode_http_header(b'=?iso-8859-1?q?p=F6stal?=') == u'p\xf6stal'
True
>>> unicode_http_header('p\xf6stal') == u'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(), six.text_type)
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
>>> buid() == buid()
False
>>> isinstance(buid(), six.text_type)
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.**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, but ignoring characters that are relevant to multiline text, like tabs and newlines.

coaster.utils.text.**sanitize_html**(*value*, *valid_tags=None*, *strip=True*, *linkify=False*)
Strips 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(u"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

coaster.utils.text.**word_count**(*text*, *html=True*)
Return the count of words in the given text. If the text is HTML (default True), tags are stripped before counting. Handles punctuation and bad formatting like.this when counting words, but assumes conventions for Latin script languages. May not be reliable for other languages.

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*, *html=False*, *valid_tags=None*)
Markdown parser with a number of sane defaults that resembles GitHub-Flavoured Markdown.

Parameters **html** (*bool*) – Allow known-safe HTML tags in text (this disables code syntax highlighting)

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("()")
''
>>> 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")
...     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
>>> '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(object):
...     @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:

```
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=None*, *response=None*)

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

exception coaster.views.decorators.**RequestValueError**(*description=None*, *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')
```

(continues on next page)

(continued from previous page)

```
...
('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')
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(
    (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'}
```

(continues on next page)

(continued from previous page)

```
@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'
```

(continues on next page)

(continued from previous page)

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

coaster.views.classview.**current_view = False**

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!'
```

(continues on next page)

(continued from previous page)

```
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(MyView, self).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'
        }

    @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 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.

1.18 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).

1.19 Document workflows

Coaster provides versions of the main Docflow classes where workflow exceptions map to HTTP 403 Forbidden (via werkzeug.exceptions.Forbidden).

exception coaster.docflow.**WorkflowStateException**(*description=None*, *response=None*)

exception coaster.docflow.**WorkflowTransitionException**(*description=None*, *response=None*)

exception coaster.docflow.**WorkflowPermissionException**(*description=None*, *response=None*)

class coaster.docflow.**WorkflowState**(*value*, *title=''*, *description=''*)
State in a workflow.

exception_permission
alias of *WorkflowPermissionException*

exception_state
alias of *WorkflowStateException*

exception_transition
alias of *WorkflowTransitionException*

class coaster.docflow.**WorkflowStateGroup**(*value*, *title=''*, *description=''*)
Group of states in a workflow. The value parameter is a list of values or WorklowState instances.

exception_permission
alias of *WorkflowPermissionException*

exception_state
alias of *WorkflowStateException*

exception_transition
alias of *WorkflowTransitionException*

class coaster.docflow.**InteractiveTransition**(*workflow*)
Multipart workflow transitions. Subclasses of this class may provide methods to return a form, validate the form and submit the form. Implementing a `submit()` method is mandatory. `submit()` will be wrapped by the `transition()` decorator to automatically update the document's state value.

Instances of *InteractiveTransition* will receive `workflow` and `document` attributes pointing to the workflow instance and document respectively.

validate()
Validate self.form, assuming Flask-WTF Form

class coaster.docflow.**DocumentWorkflow**(*document*, *context=None*)
Base class for document workflows.

exception_state
alias of *WorkflowStateException*

permissions()
Permissions for this workflow. Plays nice with `coaster.views.load_models()` and `coaster.sqlalchemy.PermissionMixin` to determine the available permissions to the current user.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W