# tangled.web Documentation

*Release 1.0a13.dev0*

**Wyatt Baldwin**

**Dec 25, 2017**

# Contents

A resource oriented, Python 3 only Web framework.

No controllers. No views. Just resources and representations.

Also, there are no magic globals. There is an *Application* object that contains your app's configuration. It is passed to resources along with the current request.

CHAPTER 1

# Links

- Project Home Page
- Source Code (GitHub)

# CHAPTER 2

# Contents

## 2.1 Hello, World

Here's a really simple Tangled Web app:

```python
from wsgiref.simple_server import make_server

from tangled.web import Application, Resource


class Hello(Resource):

    def GET(self):
        if 'name' in self.urlvars:
            content = 'Hello, {name}'.format(**self.urlvars)
        else:
            content = 'Hello'
        return content


if __name__ == '__main__':
    settings = {
        'debug': True,
    }
    app = Application(settings)
    app.mount_resource('hello', Hello, '/')
    app.mount_resource('hello_name', Hello, '/<name>')
    server = make_server('0.0.0.0', 6666, app)
    server.serve_forever()
```

**Note:** This is a copy of `examples/hello_world.py`. If you're in the top level of `tangled.web` checkout, you can run it with `python examples/hello_world.py` (assuming `tangled.web` is already installed).

## 2.2 Quick Start

This is a short guide that will show you the basics of creating a Web application based on `tangled.web`.

### 2.2.1 Install Python 3.3+

First, install Python 3.3. Older versions of Python 3 *will not* work. Mainly, this is because of the use of built-in namespace package support that was added in Python 3.3.

You can download Python 3.3 from here. If you're using Mac OS, Homebrew is an easy way to install Python:

```
brew install python3
```

**Note:** Python 2.x is *not* supported, and there are no plans to support it.

### 2.2.2 Virtual Env

Next, set up an isolated virtual environment. Since we're using Python 3, this is built in. The command for creating a virtual env looks like this:

```
python3 -m venv helloworld.venv
```

Change into the `helloworld.venv` directory and download the following file there:

https://raw.github.com/pypa/pip/master/contrib/get-pip.py

Then run the following command:

```
./bin/python get-pip.py
```

### 2.2.3 Install Dependencies

A couple of Tangled dependencies need to be installed so that the `tangled scaffold` command and `basic` scaffold are available:

```
./bin/pip install tangled.web==VERSION
```

Replace *VERSION* with the version you want to install. The current version is 1.0a13.dev0.

If you want to use the latest code, you can do this instead (requires git to be installed):

```
./bin/pip install -e git+git://github.com/TangledWeb/tangled#egg=tangled
./bin/pip install -e git+git://github.com/TangledWeb/tangled.web#egg=tangled.web
```

### 2.2.4 Create a Basic Tangled Web App

Now that the virtual environment is set up and the Tangled dependencies have been installed, a project can be created. Run the following commands in the `helloworld.venv` directory:

```
./bin/tangled scaffold basic helloworld
./bin/pip install -e helloworld
```

### 2.2.5 Serve it Up

Now that everything's installed, it's time to run the app:

```
./bin/tangled serve -f helloworld/development.ini
```

Now you can visit http://localhost:6666/ and http://localhost:6666/name.

### 2.2.6 Next Steps

Take a look at the app configuration in `helloworld/helloworld/__init__.py` and the `Hello` resource in `helloworld/helloworld/resources.py`.

The *Application API* documentation currently has the most comprehensive info on creating and configuring Tangled Web apps.

**Note:** This is all still very much a work in progress. Please feel free to make suggestions or report issues on GitHub.

## 2.3 Installation

**Note:** Python 3.3+ is required. Older versions of Python 3 will not work. No version of Python 2 will work.

All `tangled.*` packages are standard setuptools distributions that can be installed via easy_install or pip.

## 2.4 Contributing

### 2.4.1 Issues

Bugs and other issues can be reported on GitHub.

### 2.4.2 Patches

To contribute patches, go to the TangledWeb project on GitHub, fork a package, and send a pull request. All new code must be 100% covered by tests and be PEP8 compliant.

### 2.4.3 Creating an Extension Package

To create your own extension package, you can use the `tangled.contrib` namespace. If you install the `tangled.contrib` package, you will be able to create a contrib package easily using the `tangled scaffold` command:

```
tangled scaffold contrib tangled.contrib.{name}
```

## 2.5 Main Documentation

### 2.5.1 Displaying Errors

By default, errors will be displayed using the plain error templates provided by WebOb. To customize the display of errors, an error resource needs to be created. The simplest error resource looks like this:

```python
from tangled.web import Resource, config


class Error(Resource):

    @config('text/html', template='/error.html')
    def GET(self):
        return {}
```

`error.html` would contain contents like this:

```
<%inherit file="/layout.html"/>

<h1>Error</h1>

<div class="error">
  The request failed with status code ${request.status_code}
</div>
```

To activate the error resource, point the `tangled.app.error_resource` setting at it:

```ini
[app]
tangled.app.error_resource = my.pkg.resources.error:Error
```

## 2.6 Application API

This documents the API that's typically used by application developers.

### 2.6.1 Application

**class** `tangled.web.app.`**`Application`**(*settings*, ***extra_settings*)

Application container.

The application container handles configuration and provides the WSGI interface. It is passed to components such as handlers, requests, and resources so they can inspect settings, retrieve items from the registry, etc...

**Registry:**

Speaking of which, the application instance acts as a registry (it's a subclass of `tangled.registry.Registry`). This provides a means for extensions and application code to set application level globals.

**Settings:**

`settings` can be passed as either a file name pointing to a settings file or as a dict.

File names can be specified as absolute, relative, or asset paths:

- development.ini
- /some/where/production.ini
- some.package:some.ini

A plain dict can be used when no special handling of settings is required. For more control of how settings are parsed (or to disable parsing), pass a `AAppSettings` instance instead (typically, but not necessarily, created by calling `tangled.web.settings.make_app_settings()`).

Extra settings can be passed as keyword args. These settings will override *all* other settings. They will be parsed along with other settings.

NOTE: If `settings` is an `AppSettings` instance, extra settings passed here will be ignored; pass them to the `AppSettings` instead.

**Logging:**

If settings are loaded from a file and that file (or one of the files it extends) contains logging config sections (`formatters`, `handlers`, `loggers`), that logging configuration will automatically be loaded via `logging.config.fileConfig`.

**`add_helper`**(*helper*, *name=None*, *static=False*, *package=None*, *replace=False*)

Add a "helper" function.

`helper` can be a string pointing to the helper or the helper itself. If it's a string, `helper` and `package` will be passed to `load_object()`.

Helper functions can be methods that take a `Helpers` instance as their first arg or they can be static methods. The latter is useful for adding third party functions as helpers.

Helper functions can be accessed via `request.helpers`. The advantage of this is that helpers added as method have access to the application and the current request.

**`add_subscriber`**(*event_type*, *func*, *priority=None*, *once=False*, ***args*)

Add a subscriber for the specified event type.

`args` will be passed to `func` as keyword args. (Note: this functionality is somewhat esoteric and should perhaps be removed.)

You can also use the `subscriber` decorator to register subscribers.

**`get_setting`**(*key*, *default=NOT_SET*)

Get a setting; return `default` *if* one is passed.

If `key` isn't in settings, try prepending `'tangled.app.'`.

If the `key` isn't present, return the `default` if one was passed; if a `default` wasn't passed, a KeyError will be raised.

**`get_settings`**(*settings=None*, *prefix='tangled.app.'*, ***kwargs*)

Get settings with names that start with `prefix`.

This is a front end for `tangled.util.get_items_with_key_prefix()` that sets defaults for `settings` and `prefix`.

By default, this will get the settings from `self.settings` that have a `'tangled.app.'` prefix.

Alternate `settings` and/or `prefix` can be specified.

**`on_created`**(*func*, *priority=None*, *once=True*, ***args*)

Add an `ApplicationCreated` subscriber.

Sets `once` to `True` by default since `ApplicationCreated` is only emitted once per application.

This can be used as a decorator in the simple case where no args other than `func` need to be passed along to *`add_subscriber()`*.

### 2.6.2 Settings

`tangled.web.settings.`**`make_app_settings`**(*settings*, *conversion_map={}*, *defaults={}*, *required=()*, *prefix=None*, *strip_prefix=True*, *parse=True*, *section='app'*, ***extra_settings*)

Create a properly initialized application settings dict.

In simple cases, you don't need to call this directly–you can just pass a settings file name or a plain dict to *`tangled.web.app.Application`*, and this will be called for you.

If you need to do custom parsing (e.g., if your app has custom settings), you can call this function with a conversion map, defaults, &c. It's a wrapper around `parse_settings()` that adds a bit of extra functionality:

- A file name can be passed instead of a settings dict, in which case the settings will be extracted from the specified `section` of that file.
- Core tangled.web defaults are *always* added because *`tangled.web.app.Application`* assumes they are always set.
- Settings parsing can be disabled by passing `parse=False`. This only applies to *your* settings, including defaults and extra settings (*core* defaults are always parsed).
- Extra settings can be passed as keyword args; they will override all other settings, and they will be parsed (or not) along with other settings.
- Required settings are checked for after all the settings are merged.

In really special cases you can create a subclass of `AAppSettings` and then construct your settings dict by hand (eschewing the use of this function).

### 2.6.3 Events

Events are registered in the context of an application via *`tangled.web.app.Application.add_subscriber()`*.

Subscribers typically have the signature `subscriber(event)`. If subscriber keyword args were passed to `add_subscriber`, then the signature for the subscriber would be `subscriber(event, **kwargs)`.

Every event object will have an `app` attribute. Other attributes are event dependent.

**`class`** `tangled.web.events.`**`ApplicationCreated`**(*app*)

Emitted when an application is fully configured.

These events can be registered in the usual way by calling *`tangled.web.app.Application.add_subscriber()`*. There's also a convenience method for this: *`tangled.web.app.Application.on_created()`*.

Attributes: `app`.

**`class`** `tangled.web.events.`**`NewRequest`**(*app*, *request*)

Emitted when an application receives a new request.

This is *not* emitted for static file requests.

Attributes: `app`, `request`.

**class** tangled.web.events.**NewResponse**(*app*, *request*, *response*)

Emitted when the response for a request is created.

This is *not* emitted for static file requests.

If there's in exception during request handling, this will *not* be emitted.

Attributes: `app`, `request`, `response`.

**class** tangled.web.events.**ResourceFound**(*app*, *request*, *resource*)

Emitted when the resource is found for a request.

Attributes: `app`, `request`, `resource`.

**class** tangled.web.events.**TemplateContextCreated**(*app*, *request*, *context*)

Emitted when the context for a template is created.

The template `context` is whatever data will passed to the template. E.g., for Mako, it's a dict.

This is emitted just before the template is rendered. Its purpose is to allow additional data to be injected into the template context.

Attributes: `app`, `request`, `context`

tangled.web.events.**subscriber**(*event_type*, **args*, ***kw*)

Decorator for adding event subscribers.

Subscribers registered this way won't be activated until *tangled.web.app.Application.load_config()* is called.

Example:

```
@subscriber('tangled.web.events:ResourceFound')
def on_resource_found(event):
    log.debug(event.resource.name)
```

### 2.6.4 Request factory

**class** tangled.web.request.**Request**(*environ*, *app*, **args*, ***kwargs*)

Default request factory.

Every request has a reference to its application context (i.e., `request.app`).

**abort**(*status_code*, **args*, ***kwargs*)

Abort the request by raising a WSGIHTTPException.

This is a convenience so resource modules don't need to import exceptions from `webob.exc`.

**get_setting**(**args*, ***kwargs*)

Get an app setting.

Simply delegates to *tangled.web.app.Application.get_setting()*.

**helpers**

Get helpers for this request.

Returns a `Helpers` instance; all the helpers added via *tangled.web.app.Application.add_helper()* will be accessible as methods of this instance.

**make_url**(*path*, *query=None*, *fragment=None*, ***, *_fully_qualified=True*)

Generate a URL.

`path` should be application-relative (that is, it should *not* include SCRIPT_NAME).

`query` can be a string, a dict, or a sequence. See `make_query_string()` for details.

If `fragment` is passed it will be quoted using `urllib.parse.quote()` with no “safe” characters (i.e., all special characters will be quoted).

**`on_finished`**(*callback*, **args*, ***kwargs*)
Add a finished callback.

Callbacks must have the signature `(app, response)`. They can also take additional positional and keyword args–`*args` and `**kwargs` will be passed along to the `callback`.

Finished callbacks are always called regardless of whether an error occurred while processing the request. They are called just before the Tangled application returns to its caller.

*All* finished callbacks will be called. If any of them raises an exception, a `RequestFinishedException` will be raised and a “500 Internal Server Error” response will be returned in place of the original response.

Raising instances of `webob.exc.WSGIHTTPException` in finished callbacks is an error.

The `response` object can be inspected to see if an error occurred while processing the request. If the `response` is `None`, the request failed hard (i.e., there was an uncaught exception before the response could be created).

This can be used as a decorator in the simple case where the `callback` doesn’t take any additional args.

**`resource_config`**
Get info for the resource associated with this request.

**Note:** This can’t be safely accessed until after the resource has been found and set for this request.

**`resource_path`**(*resource*, *urlvars=None*, ***kwargs*)
Generate a URL path (with SCRIPT_NAME) for a resource.

**`resource_url`**(*resource*, *urlvars=None*, ***kwargs*)
Generate a URL for a resource.

**`response`**
Create the default response object for this request.

The response is initialized with attributes set via `@config`: `status`, `location`, and `response_attrs`.

If no status code was set via `@config`, we try our best to set it to something sane here based on content type and method.

If `location` is set but `status` isn’t, the response’s status is set to `DEFAULT_REDIRECT_STATUS`.

The location can also be set to one of the special values ‘REFERER’ or ‘CAME_FROM’. The former redirects back to the refering page. The latter redirects to whatever is set in the `came_from` request parameter.

TODO: Check origin of referer and came from.

**Note:** See note in *`resource_config()`*.

**`response_content_type`**
Get the content type to use for the response.

This retrieves the content types the resource is configured to handle then selects the best match for the requested content type. If the resource isn't explicitly configured to handle any types or of there's no best match, the default content type will be used.

**Note:** This can't be safely accessed until after the resource has been found and set for this request.

**`static_url`**(*path*, *query=None*, ***kwargs*)
Generate a static URL from `path`.

`path` should always be an application-relative path like '/static/images/logo.png'. SCRIPT_NAME will be prepended by *`make_url()`*.

**`update_response`**(***kwargs*)
Set multiple attributes on *request.response*.

## 2.6.5 Resources

- *Creating resources*
- *Configuring resources*
- *Mounting resources*

### Creating resources

**class** `tangled.web.resource.resource.`**`Resource`**(*app*, *request*, *name=None*, *urlvars=None*)
Base resource class.

Usually, you will want to subclass *`Resource`* when creating your own resources. Doing so will ensure your resources are properly initialized.

Subclasses will automatically return a `405 Method Not Allowed` response for unimplemented methods.

Subclasses also have *`url()`* and *`path()`* methods that generate URLs and paths to the "current resource". E.g., in a template, you can do `resource.path()` to generate the application-relative path to the current resource. You can also pass in query parameters and alternate URL vars to generate URLs and paths based on the current resource.

**`DELETE`**()
Delete resource.

Return

- 204 if no body
- 200 if body
- 202 if accepted but not yet deleted

**`GET`**()
Get resource.

Return:

- 200 body

**`HEAD`**()
Get resource metadata.

Return:

- 204 no body (same headers as GET)

**OPTIONS**()
Get resource options.

By default, this will add an `Allow` header to the response that lists the methods implemented by the resource.

**PATCH**()
Update resource.

Return:

- 200 (body)
- 204 (no body)
- 303 (instead of 204)

**POST**()
Create a new child resource.

Return:

- **If resource created and identifiable w/ URL:**
  - 201 w/ body and Location header (for XHR?)
  - 303 w/ Location header (for browser?)
- **If resource not identifiable:**
  - 200 if body
  - 204 if no body

**PUT**()
Update resource or create if it doesn't exist.

Return:

- If new resource created, same as *POST()*
- **If updated:**
  - 200 (body)
  - 204 (no body)
  - 303 (instead of 204)

**path**(*urlvars=None*, ***kwargs*)
Generate an application-relative URL path for this resource.

You can pass `urlvars`, `query`, and/or `fragment` to generate a path based on this resource.

**url**(*urlvars=None*, ***kwargs*)
Generate a fully qualified URL for this resource.

You can pass `urlvars`, `query`, and/or `fragment` to generate a URL based on this resource.

### Configuring resources

**class** `tangled.web.resource.config.`**config**
Decorator for configuring resources methods.

When used on a resource class, the class level configuration will be applied to all methods.

Example:

```
class MyResource:

    @config('text/html', template='my_resource.mako')
    def GET(self):
        pass
```

Example of defaults and overrides:

```
@config('*/*', status=303, response_attrs={'location': '/'})
class MyResource:

    @config('*/*', status=302)
    @config('text/html', status=None, response_attrs={})
    def GET(self):
        pass
```

## Mounting Resources

`Application.`**`mount_resource`**`(`*name*, *factory*, *path*, *methods=()*, *method_name=None*, *add_slash=False*, *_level=3*`)`

Mount a resource at the specified path.

Basic example:

```
app.mount_resource('home', 'mypackage.resources:Home', '/')
```

Specifying URL vars:

```
app.mount_resource(
    'user', 'mypackage.resources:User', '/user/<id>')
```

A unique `name` for the mounted resource must be specified. This can be *any* string. It's used when generating resource URLs via *`request.Request.resource_url()`*.

A `factory` must also be specified. This can be any class or function that produces objects that implement the resource interface (typically a subclass of *`resource.resource.Resource`*). The factory may be passed as a string with the following format: `package.module:factory`.

The `path` is an application relative path that may or may not include URL vars.

A list of HTTP `methods` can be passed to constrain which methods the resource will respond to. By default, it's assumed that a resource will respond to all methods. Note however that when subclassing *`resource.resource.Resource`*, unimplemented methods will return a `405 Method Not Allowed` response, so it's often unnecessary to specify the list of allowed methods here; this is mainly useful if you want to mount different resources at the same path for different methods.

If `path` ends with a slash or `add_slash` is True, requests to `path` without a trailing slash will be redirected to the `path` with a slash appended.

About URL vars:

The format of a URL var is `<(converter)identifier:regex>`. Angle brackets delimit URL vars. Only the `identifier` is required; it can be any valid Python identifier.

If a `converter` is specified, it can be a built-in name, the name of a converter in `tangled.util.converters`, or a `package.module:callable` path that points to a callable that accepts a single argument. URL vars found in a request path will be converted automatically.

The `regex` can be *almost* any regular expression. The exception is that < and > can't be used. In practice, this means that named groups (`(?P<name>regex)`) can't be used (which would be pointless anyway), nor can "look behinds".

**Mounting Subresources**

Subresources can be mounted like this:

```
parent = app.mount_resource('parent', factory, '/parent')
parent.mount('child', 'child')
```

or like this:

```
with app.mount_resource('parent', factory, '/parent') as parent:
    parent.mount('child', 'child')
```

In either case, the subresource's `name` will be prepended with its parent's name plus a slash, and its `path` will be prepended with its parent's path plus a slash. If no `factory` is specified, the parent's factory will be used. `methods` will be propagated as well. `method_name` and `add_slash` are *not* propagated.

In the examples above, the child's name would be `parent/child` and its path would be `/parent/child`.

### 2.6.6 Static Files

`Application.`**`mount_static_directory`**(*prefix*, *directory*, *remote=False*, *index_page=None*)

Mount a local or remote static directory.

`prefix` is an alias referring to `directory`.

If `directory` is just a path, it should be a local directory. Requests to `/{prefix}/{path}` will look in this directory for the file indicated by `path`.

If `directory` refers to a remote location (i.e., it starts with `http://` or `https://`), URLs generated via `reqeust.static_url` and `request.static_path` will point to the remote directory.

`remote` can also be specified explicitly. In this context, "remote" means not served by the application itself. E.g., you might be mapping an alias in Nginx to a local directory.

**Note:** It's best to always use *`tangled.web.request.Request.static_url()`* `tangled.web.request.Request.static_path()` to generate static URLs.

## 2.7 Extension API

This documents the API that is typically used by extension developers.

### 2.7.1 Configuration

#### Including other configuration

`Application.`**`include`**(*obj*)

Include some other code.

If a callable is passed, that callable will be called with this app instance as its only argument.

If a module is passed, it must contain a function named `include`, which will be called as described above.

#### Loading configuration registered via decorators

`Application.`**`load_config`**`(`*where*`)`
Load config registered via decorators.

### 2.7.2 Adding @config Args

#### Fields

`Application.`**`add_config_field`**`(`*content_type*, *name*, **args*, ***kwargs*`)`
Add a config field that can be passed via `@config`.

This allows extensions to add additional keyword args for `@config`. These args will be accessible as attributes of the `resource.config.Config` object returned by `request.resource_config`.

These fields can serve any purpose. For example, a `permission` field could be added, which would be accessible as `request.resource_config.permission`. This could be checked in an auth handler to verify the user has the specified permission.

See `_add_config_arg()` for more detail.

#### Representation Args

`Application.`**`add_representation_arg`**`(`**args*, ***kwargs*`)`
Add a representation arg that can be specified via @config.

This allows extensions to add additional keyword args for `@config`. These args will be passed as keyword args to the representation type that is used for the request.

These args are accessible via the `representation_args` dict of the `resource.config.Config` object returned by `request.resource_config` (but generally would not be accessed directly).

See `_add_config_arg()` for more detail.

### 2.7.3 Request Handlers

#### Adding request handlers

Handlers are callables with the signature `(app, request, next_handler)`.

`Application.`**`add_handler`**`(`*handler*`)`
Add a handler to the handler chain.

Handlers added via this method are inserted into the system handler chain above the main handler. They will be called in the order they are added (the last handler added will be called directly before the main handler).

Handlers are typically functions but can be any callable that accepts `app`, `request`, and `next_handler` args.

Each handler should either call its `next_handler`, return a response object, or raise an exception.

TODO: Allow ordering?

## System handler chain

*Adding request handlers* System handlers.

Requests are processed through a chain of handlers. This module contains the "system" handlers. These are handlers that always run in a specific order.

Most of the system handlers *always* run. They can't be turned off, but you can swap in different implementations via settings. Take a look at `tangled/web/defaults.ini` to see how you would do this.

Some handlers are only enabled when certain settings are enabled or when certain configuration takes place. For example, to enable CSRF protection, the `tangled.app.csrf.enabled` setting needs to be set to `True`. Another example: the static files handlers is only enabled when at least one static directory has been mounted.

If an auth handler is enabled, it will run directly before any (other) handlers added by the application developer.

All added handlers are called in the order they were added. The last handler to run is always the `main()` handler; it calls into application code (i.e., it calls a resource method to get data or a response).

`tangled.web.handlers.`**`error_handler`**(*app*, *request*, *main_handler*, *original_response*)

Handle error response.

If an error resource is configured, its `GET` method will be called to get the final response. This is accomplished by setting the error resource as the resource for the request and then passing the request back into the main handler.

If CORS is enabled, the main handler will be wrapped in the CORS handler so that error responses will have the appropriate headers.

If no error resource is configured, the original error response will be returned as is.

`tangled.web.handlers.`**`request_finished_handler`**(*app*, *request*, *_*)

Call request finished callbacks in exc handling context.

This calls the request finished callbacks in the same exception handling context as the request. This way, if exceptions occur in finished callbacks, they can be logged and displayed as usual.

**Note:** Finished callbacks are not called for static requests.

`tangled.web.handlers.`**`tweaker`**(*app*, *request*, *next_handler*)

Tweak the request based on special request parameters.

`tangled.web.handlers.`**`resource_finder`**(*app*, *request*, *next_handler*)

Find resource for request.

Sets `request.resource` and notifies `ResourceFound` subscribers.

If a resource isn't found, a 404 response is immediatley returned. If a resource is found but doesn't respond to the request's method, a `405 Method Not Allowed` response is returned.

`tangled.web.handlers.`**`timer`**(*app*, *request*, *next_handler*)

Log time taken to handle a request.

`tangled.web.handlers.`**`main`**(*app*, *request*, *_*)

Get data from resource method and return response.

If the resource method returns a response object (an instance of `Response`), that response will be returned without further processing.

If the status of `request.response` has been set to 3xx (either via @config or in the body of the resource method) AND the resource method returns no data, the response will will be returned as is without further processing.

Otherwise, a representation will be generated based on the request's Accept header (unless a representation type has been set via @config, in which case that type will be used instead of doing a best match guess).

If the representation returns a response object as its content, that response will be returned without further processing.

Otherwise, *request.response* will be updated according to the representation type (the response's content_type, charset, and body are set from the representation).

## 2.7.4 Request

### Adding request methods

`Application.`**`add_request_attribute`**(*attr*, *name=None*, *decorator=None*, *reify=False*)

Add dynamic attribute to requests.

This is mainly intended so that extensions can easily add request methods and properties.

Functions can already be decorated, or a `decorator` can be specified. If `reify` is `True`, the function will be decorated with `tangled.decorators.cached_property()`. If a `decorator` is passed and `reify` is `True`, `cached_property` will be applied as the outermost decorator.

### Request factories

These two methods make it easy to create properly configured requests. In particular, they set the request's `app` attribute, and they create request instances with the attributes added via *`tangled.web.app.Application.add_request_attribute()`*.

`Application.`**`make_request`**(*environ*, ***kwargs*)

Make a request using the registered request factory.

`Application.`**`make_blank_request`**(**args*, ***kwargs*)

Make a blank request using the registered request factory.

# CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index