Keg

Release 0.11.0

Mar 03, 2023

Contents

1 Keg's Goal **3**

2 Installation **5**

3 Upgrade Notes **7**

4 Contents **9**

4.1 App 9
4.2 Components 10
4.3 Utils 12
4.4 Signals 12
4.5 Testing Utils 13
4.6 Keg Web 15
4.7 Views 15
4.8 Features 19
4.9 App Configuration 20
4.10 Internationalization 21

Python Module Index **23**

Index **25**

Keg is an opinionated but flexible web framework built on Flask and SQLAlchemy.

CHAPTER 1

Keg's Goal

The goal for this project is to encapsulate Flask best practices and libraries so devs can avoid boilerplate and work on the important stuff.

We will lean towards being opinionated on the big things (like SQLAlchemy as our ORM) while supporting hooks and customizations as much as possible.

Think North of Flask but South of Django.

CHAPTER 2

Installation

```
pip install keg
```

CHAPTER 3

Upgrade Notes

While we attempt to preserve backward compatibility, some Keg versions do introduce breaking changes. This list should provide information on needed app changes.

- 0.11.0
  - App context is no longer pushed as part of test suite setup
    - Having the app context pushed there was problematic because extensions can rely on `flask.g` refreshing for each request
    - Generally, when processing requests, flask will push a fresh app context. However, this does not occur if a context is already present.
    - While that concern is resolved most directly in flask-webtest 0.1.1 by directly pushing an app context as part of the request process, keg needs to stop tying the test framework and the context so closely together. An app's test suite needs to be able to set up and tear down contexts as needed.
    - pytest does not yet support passing fixtures to setup methods. Thus, for the time being, to have database use available in setup methods (since flask-sqlalchemy ties that session to an app context), auto-used fixtures will be needed. Ensure the scope of the auto-used fixture matches the level of setup methods needed in the suite (module, class, etc.)
    - Mocking can be affected in the test suite as well. Any mocks requiring app context will need to happen at run time, rather than import time
      - E.g. `@mock.patch.dict(flask.current_app.config, ...)` becomes `@mock.patch.dict('flask.current_app.config', ...)`
- 0.10.0
  - `rule`: default class view route no longer generated when any rules are present
    - Absolute route had been provided automatically from the class name, but in some situations this would not be desired. Views that still need that route can use a couple of solutions:
      - Provide an absolute route rule: `rule('/my-route')`
      - Use an empty relative route rule: `rule()`

    - All of an app's routes may be shown on CLI with the `<app> develop routes` command

- Removed `keg` blueprint along with `ping` and `exception-test` routes
- DB manager `prep_empty` method no longer called (had been deprecated)
- Python 2 support removed
- Flask changed app's `json_encoder/json_decoder` attributes to `_json_encoder/_json_decoder`

CHAPTER 4

Contents

4.1 App

class keg.app.**Keg**(*import_name=None*, **args*, ***kwargs*)

cli_loader_class
alias of `keg.cli.CLILoader`

config_class
alias of `keg.config.Config`

logger
Standard logger for the app.

make_config(*instance_relative=False*)
Needed for Flask <= 0.10.x so we can set the configuration class being used. Once 0.11 comes out, Flask supports setting the config_class on the app.

on_config_complete()
For subclasses to override

on_init_complete()
For subclasses to override

request_context(*environ*)
Request context for the app.

route(*rule*, ***options*)
Enable .route() to be used in a class context as well. E.g.:

```
KegApp.route('/something'):
def view_something():
    pass
```

classmethod testing_prep(***config*)

1. Instantiate the app class.
2. **Cache the app instance after creation so that it's only instantiated once per Python** process.
3. **Trigger *signal.testing_run_start* the first time this method is called for an app** class.

4.2 Components

Keg components follow the paradigm of flask extensions, and provide some defaults for the purpose of setting up model/view structure. Using components, a project may be broken down into logical blocks, each having their own entities, blueprints, templates, tests, etc.

- Components need to be registered in config at `KEG_REGISTERED_COMPONENTS`
  - The path given here should be a full dotted path to the top level of the component
    - e.g. `my_app.components.blog`
  - At the top level of the component, `__component__` must be defined as an instance of KegComponent
    - Depending on the needs of the component, model and view discovery may be driven by the subclasses of KegComponent that have path defaults
    - Examples:
      - `__component__ = KegModelComponent('blog')`
      - `__component__ = KegViewComponent('blog')`
      - `__component__ = KegModelViewComponent('blog')`
- Component discovery
  - A component will attempt to load model and blueprints on app init
  - The default paths relative to the component may be modified or extended on the component's definition
  - Default model path in "model" components: `.model.entities`
    - Override via the component's `db_visit_modules` list of relative import paths
  - Default blueprint path for "view" components: `.views.component_bp`
    - Use the `create_named_blueprint` or `create_blueprint` helpers on the component's `__component__` to create blueprints with configured template folders
    - Override via the component's `load_blueprints` list
      - List elements are a tuple of the relative import path and the name of the blueprint attribute
    - Components have their own template stores, in a `templates` folder
      - Override the component's template path via the `template_folder` attribute
  - Paths may also be supplied to the constructor
    - e.g. `__component__ = KegComponent('blog', db_visit_modules=('.somewhere.else', ))`

4.2.1 Documentation

class `keg.component.`**`KegComponent`**(*name*, *app=None*, *db_visit_modules=None*, *load_blueprints=None*, *template_folder=None*, *parent_path=None*)

Keg components follow the paradigm of flask extensions, and provide some defaults for the purpose of setting up model/view structure. Using components, a project may be broken down into logical blocks, each having their own entities, blueprints, templates, tests, etc.

Setup involves: - KEG_REGISTERED_COMPONENTS config setting: assumed to be an iterable of importable dotted paths - *__component__*: at the top level of each dotted path, this attribute should point to an instance of *KegComponent*. E.g. *__component__* = *KegComponent('widgets')*

By default, components will load entities from *db_visit_modules* into metadata and register any blueprints specified by *load_blueprints*.

Blueprints can be created with the helper methods *create_named_blueprint* or *create_blueprint* in order to have a configured template folder relative to the blueprint path.

Use KegModelComponent, KegViewComponent, or KegModelViewComponent for some helpful defaults for model/blueprint discovery.

db_visit_modules: an iterable of dotted paths (e.g. *.mycomponent.entities*, *app.component.extraentities*)
where Keg can find the entities for this component to load them into the metadata.

Note: Normally this is not explicitly required but can be useful in cases where imports won't reach that file.

Note: This can accept relative dotted paths (starts with .) and it will prepend the component python package determined by Keg when instantiating the component. You can also pass absolute dotted paths and no alterations will be performed.

load_blueprints: an iterable of tuples, each having a dotted path (e.g. *.mycomponent.views*,
app.component.extraviews) and the blueprint attribute name to load and register on the app. E.g. *(('.views', 'component_bp'),)*

Note: This can accept relative dotted paths (starts with .) and it will prepend the component python package determined by Keg when instantiating the component. You can also pass absolute dotted paths and no alterations will be performed.

template_folder: string to be passed for template config to blueprints created via the component

`create_blueprint`(**args*, ***kwargs*)
Make a flask blueprint having a template folder configured.

Generally, args and kwargs provided will be passed to the blueprint constructor, with the following exceptions:

- template_folder kwarg defaults to the component's template_folder if not provided
- blueprint_cls kwarg may be used to specify an alternative to flask.Blueprint

`create_named_blueprint`(**args*, ***kwargs*)

`db_visit_modules = ()`

`init_app`(*app*, *parent_path=None*)

init_blueprints(*app*, *parent_path*)

init_config(*app*)

init_db(*parent_path*)

load_blueprints = ()

template_folder = 'templates'

4.3 Utils

class keg.utils.**ClassProperty**(*fget*, **arg*, ***kw*)
A decorator that behaves like @property except that operates on classes rather than instances.

class keg.utils.**HybridMethod**(*func*, *cm_func=None*)
A decorator which allows definition of a Python object method with both instance-level and class-level behavior:

```
Class Bar:
    @hybridmethod
    def foo(self, rule, **options):
        # this is used in an instance context

    @foo.classmethod
    def foo(cls, rule, **options):
        # this is used in class context
```

classmethod(*cm_func*)
Provide a modifying decorator that is used as a classmethod decorator.

keg.utils.**app_environ_get**(*app_import_name*, *key*, *default=None*)

keg.utils.**ensure_dirs**(*newdir*, *mode=<class 'keg.utils.NotGiven'>*)
A "safe" verision of Path.makedir(. . . , parents=True) that will only create the directory if it doesn't already exist. We also manually create parents so that mode is set correctly. Python docs say that mode is ignored when using Path.mkdir(. . . , parents=True)

keg.utils.**pymodule_fpaths_to_objects**(*fpaths*)
Takes an iterable of file paths reprenting possible python modules and will return an iterable of tuples with the file path along with the contents of that file if the file exists.

If the file does not exist or cannot be accessed, the third term of the tuple stores the exception.

keg.utils.**visit_modules**(*dotted_paths*, *base_path=None*)

4.4 Signals

As Keg is based on Flask architecture, signals are used to set up and execute callback methods upon certain events.

Attaching a callback to a signal involves the connect decorator:

```
from keg.signals import init_complete

@init_complete.connect
def init_navigation(app):
    pass
```

Take care: some signals fire before an app is fully set up and ready. See the definitions below for when the signals are fired, and what can be counted upon to be available.

4.4.1 Keg Events

- `init_complete`
  - All of the app's init tasks have run
  - App's `on_init_complete` method has run
  - At this point in the process, it should be safe to assume all app-related objects are present
- `config_complete`
  - App config has been loaded
  - Config is the first property of the app to be initialized. *app.config* will be available, but do not count on anything else.
- `db_before_import`
  - Database options have been configured, and the app is about to visit modules containing entities
  - Config, logging, and error handling have been loaded, but no other extensions, and the app's `visit_modules` has not yet been processed
  - Some SQLAlchemy metadata attributes, such as naming convention, need to be set prior to entities loading. Attaching a method on this signal is an ideal way to set these properties.
  - A common practice with signals is to attach handlers in a separate module, and then list that module in the app's visit_modules. This works with many signals, however, the database layer gets set up very early in app init to make it available in other steps of the init process.
    * As a result, `db_before_import` happens long before visit_modules is processed.
    * Instead, use `db_before_import` somewhere that gets loaded at import time for the app (e.g. in the module containing the app itself, or something it imports).
  - If customization of the db object, metadata, engine options, etc. is needed, ensure that no modules containing entities are imported before the connected callback runs.
- `testing_run_start`
  - `app.testing_prep` has set up necessary context and is about to return the test app
  - Not run during normal operation
  - Provides a hook to inject necessary test objects
- `db_clear_pre`, `db_clear_post`, `db_init_pre`, `db_init_post`
  - Called during the database initialization process, which occurs in test setup and from CLI commands

4.5 Testing Utils

class `keg.testing.`**`CLIBase`**

Test class base for testing Keg click commands.

Creates a CLI runner instance, and allows subclass to call `self.invoke` with command args.

Class attributes: - app_cls: Optional, will default to `flask.current_app` class. - cmd_name: Optional, provides default in `self.invoke` for `cmd_name` kwarg.

invoke(**args*, ***kwargs*)
Run a command, perform some assertions, and return the result for testing.

class `keg.testing.`**ContextManager**(*appcls*)
Facilitates having a single instance of an application ready for testing.

By default, this is used in `Keg.testing_prep`.

Constructor arg is the Keg app class to manage for tests.

classmethod get_for(*appcls*)
Return the ContextManager instance for the given app class. Only one ContextManager instance will be created in a Python process for any given app.

is_ready()
Indicates the manager's app instance exists.

The instance should be created with `get_for`. Only one ContextManager instance will get created in a Python process for any given app. But, `get_for` may be called multiple times. The first call to `ensure_current` will set up the application and bring the manager to a ready state.

`keg.testing.`**app_config**(***kwargs*)
Set config values on any apps instantiated while the context manager is active. This is intended to be used with cli tests where the `current_app` in the test will be different from the `current_app` when the CLI command is invoked, making it very difficult to dynamically set app config variables using mock.patch.dict like we normally would.

Example:

```
class TestCLI(CLIBase):
    app_cls = MyApp
    def test_it(self):
        with testing.app_config(FOO_NAME='Bar'):
            result = self.invoke('echo-foo-name')
        assert 'Bar' in result.output
```

`keg.testing.`**inrequest**(**req_args*, *args_modifier=None*, ***req_kwargs*)
A decorator/context manager to add the flask request context to a test function.

Allows test to assume a request context without running a full view stack. Use for unit-testing a view instance without setting up a webtest instance for the app and running requests.

Flask's `request.args` is normally immutable, but in test cases, it can be helpful to patch in args without needing to construct the URL. But, we don't want to leave them mutable, because potential app bugs could be masked in doing so. To modify args, pass in a callable as `args_modifier` that takes the args dict to be modified in-place. Args will only be mutable for executing the modifier, then returned to immutable for the remainder of the scope.

Assumes that `flask.current_app` is pointing to the desired app.

Example:

```
@inrequest('/mypath?foo=bar&baz=boo')
def test_in_request_args(self):
    assert flask.request.args['foo'] == 'bar'

def test_request_args_mutated(self):
    def args_modifier(args_dict):
        args_dict['baz'] = 'custom-value'

    with inrequest('/mypath?foo=bar&baz=boo', args_modifier=args_modifier):
```

(continues on next page)

(continued from previous page)

```
        assert flask.request.args['foo'] == 'bar'
        assert flask.request.args['baz'] == 'custom-value'
```

keg.testing.**invoke_command**(*app_cls*, **args*, ***kwargs*)

Invoke a command using a CLI runner and return the result.

Optional kwargs: - exit_code: Default 0. Process exit code to assert. - runner: Default `click.testing.CliRunner()`. CLI runner instance to use for invocation. - use_test_profile: Default True. Drive invoked app to use test profile instead of default.

4.6 Keg Web

4.7 Views

While generic Flask views will certainly work in this framework, Keg provides a BaseView that applies a certain amount of magic around route and blueprint setup. BaseView is based on Flask's MethodView. Best practice is to set up a blueprint and attach the views to it via the `blueprint` attribute. Be aware, BaseView will set up some route, endpoint, and template location defaults, but these can be configured if needed.

4.7.1 Blueprint Setup

Adding views to a blueprint is accomplished via the `blueprint` attribute on the view. Note, BaseView magic kicks in when the class is created, so assigning the blueprint later on will not currently have the desired effect:

```
import flask
from keg.web import BaseView

blueprint = flask.Blueprint('routing', __name__)


class VerbRouting(BaseView):
    blueprint = blueprint

    def get(self):
        return 'method get'
```

Once the blueprint is created, you must attach it to the app via the `use_blueprints` app attribute:

```
from keg.app import Keg
from my_app.views import blueprint


class MyApp(Keg):
    import_name = 'myapp'
    use_blueprints = (blueprint, )
```

Blueprints take some parameters for URL prefix and template path. BaseView will respect these when generating URLs and finding templates:

```
blueprint = flask.Blueprint(
    'custom',
```

(continues on next page)

(continued from previous page)

```
    __name__,
    template_folder='../templates/specific-path',
    url_prefix='/tanagra')

class BlueprintTest(BaseView):
    # template "blueprint_test.html" will be expected in specific-path
    # endpoint is custom.blueprint-test
    # URL is /tanagra/blueprint-test
    blueprint = blueprint

    def get(self):
        return self.render()
```

4.7.2 Template Discovery

To avoid requiring the developer to configure all the things, BaseView will attempt to discover the correct template for a view, based on the view class name. Generally, this is a camel-case to underscore-notation conversion. Blueprint name is included in the path, unless the blueprint has its own `template_path` defined.

- `class MyBestView` in blueprint named “public” -> `<app>/templates/public/my_best_view.html`
- `class View2` in blueprint named “other” with template path “foo” -> `<app>/foo/view2.html`

A view may be given a `template_name` attribute to override the default filename, although the same path is used for discovery:

```
class TemplateOverride(BaseView):
    blueprint = blueprint
    template_name = 'my-special-template.html'

    def get(self):
        return self.render()
```

4.7.3 URL and Endpoint Calculation

BaseView has `calc_url` and `calc_endpoint` class methods which will allow the developer to avoid hard-coding those types of values throughout the code. These methods will both produce the full URL/endpoint, including the blueprint prefix (if any).

4.7.4 Route Generation

BaseView will, by default, create rules for views on their respective blueprints. Generally, this is based on the view class name as a camel-case to dash-notation conversion:

- `class MyBestView` in blueprint named “public”: `/my-best-view` -> `public.my-best-view`
- `class View2` in blueprint named “other” with URL prefix “foo”: `/foo/view2` -> `other.view2`

Note that BaseView is a MethodView implementation, so methods named `get`, `post`, etc. will be respected as the appropriate targets in the request/response cycle.

A view may be given a `url` attribute to override the default:

```
class RouteOverride(BaseView):
    blueprint = blueprint
    url = '/something-other-than-the-default'

    def get(self):
        return self.render()
```

See `keg_apps/web/views/routing.py` for other routing possibilities that BaseView supports.

4.7.5 Class View Lifecycle

Keg views use Flask's `dispatch_request` to call several methods walking a view through its response cycle. As the methods progress, assumptions may be built for access, availability, etc. Many of these methods will not normally be present on a view.

The view lifecycle is as follows:

- `process_calling_args`
  - Gather arguments from the route definition and the query string
  - If `expected_qs_args` is set on the view, look for these arguments in the query string
  - URL arguments from the route definition have precedence over GET args in the query string
  - Arguments are processed once, then stored on the view
- `pre_auth`
  - Meant for actions that should take place before a user/session has been verified
  - Assumptions: calling args
- `check_auth`
  - Meant to verify the user/session has access to this resource
  - Failure at this point should take appropriate action in the method itself (403, 401, etc.)
  - Extensions such as keg-auth leverage this method to insert permission-based authorization into the view cycle
  - Assumptions: calling args
- `pre_loaders`
  - Authentication/authorization has passed, but we haven't loaded any related view dependencies
  - Assumptions: calling args, auth
- Loader methods
  - Any method on the view ending with `_loader` is called with args
  - Return value of the method is stored with the calling args, keyed by the method name
    - e.g. a method named `record_loader` will set a value in calling args for `record`
  - Methods folliwng this in the lifecycle can use the newly-set arg
  - If no value is returned, Keg assumes a required dependency could not be loaded and returns a 404 response
  - Order of execution of a view's loaders may not be assumed
  - Assumptions: calling args, auth

- `pre_method`
  - Ideal method for running code shared by all response methods (e.g. `get`, `post`, etc.)
  - Assumptions: calling args, auth, loader args
- Responding method
  - The method used here is generally the lowercase of the request method (e.g. `get`, `post`, etc.)
  - If the request method is HEAD, but there is no `head` method, Keg looks for `get` instead
  - This method may return the view's response
  - Assumptions: calling args, auth, loader args
- If responding method does not return a reponse:
  - I.e. the responding method returned something falsy that isn't an empty string
  - `pre_render`
    - Assumptions: calling args, auth, loader args
  - `render`
    - Returns a response object
    - By default, renders the template with args assigned on the view
    - See Template Discovery above
- `pre_response`
  - A response has been generated, but has not been sent yet
  - The response is included as the `_response` arg for this method
  - The response should not be assumed to be mutable
  - If a different response should be sent, return that response from this method
  - Assumptions: calling args, auth, loader args, response (from responding method or render)

4.7.6 Documentation

class keg.web.**BaseView**(*responding_method=None*)

Base class for all Keg views to inherit from. *BaseView* automatically calculates and installs routing, templating, and responding methods for HTTP verb named functions.

```
# Example usage of `keg.web.BaseView`
import flask
from keg.web import BaseView

core_bp = flask.Blueprint('core', __name__)

class FooView(BaseView):
    url = '/foo'
    template_name = 'foo.html'
    blueprint = core_bp

    def get(self):
        context = {
            "bar": "baz",
```

(continues on next page)

(continued from previous page)

```
        }

        return flask.render_template(self.calc_template_name(), **context)
```

assign (*key*, *value*)

classmethod assign_blueprint (*blueprint*)

auto_assign = ()

blueprint = None

calc_class_fname (*use_us=False*)

classmethod calc_endpoint (*use_blueprint=True*)

calc_responding_method ()

calc_template_name (*use_us=False*)

classmethod calc_url (*use_blueprint=True*)

call_loaders (*calling_args*)

check_auth ()

dispatch_request (***kwargs*)

The actual view function behavior. Subclasses must override this and return a valid response. Any variables from the URL rule are passed as keyword arguments.

expected_qs_args = []

classmethod init_blueprint (*rules*)

classmethod init_routes ()

process_auto_assign ()

process_calling_args (*urlargs*)

render ()

require_authentication = False

template_name = None

url = None

keg.web.**redirect** (*endpoint*, **args*, ***kwargs*)

4.8 Features

4.8.1 Default Logging Configuration

We highly recommend good logging practices and, as such, a Keg application does basic setup of the Python logging system:

- Sets the log level on the root logger to INFO
- Creates two handlers and assigns them to the root logger:
  - outputs to stderr

- – outputs to syslog
- Provides an optional json formatter

The thinking behind that is:

- In development, a developer will see log messages on stdout and doesn't have to monitor a file.
- Log messages will be in syslog by default and available for review there if no other action is taken by the developer or sysadmin. This avoids the need to manage log placement, permissions, rotation, etc.
- It's easy to configure syslog daemons to forward log messages to different files or remote log servers and it's better to handle that type of need at the syslog level than in the app.
- Structured log files (json) provide metadata details in a easy-to-parse format and should be easy to generate.
- The options and output should be easily configurable from the app to account for different needs in development and deployed scenarios.
- Keg's logging setup should be easy to turn off and/or completely override for situations where it hurts more than it helps.

4.9 App Configuration

4.9.1 Configuration Variables

- `KEG_DB_DIALECT_OPTIONS`: Dict of options to provide to the db manager. E.g. "postgresql.schemas".
  - Options keys can target either the dialect or a specific bind.
    - Dialect: `postgresql.schemas`
    - Bind: `bind.<bind-name>.schemas`
  - If the bind option is present, it will override a corresponding dialect option.
  - Options supported:
    - `schemas`: Tuple of schema names to create. Supported for postgresql and mssql dialects.
- `KEG_DIR_MODE`: Mode used by `ensure_dirs`. Default 0o777.
- `KEG_ENDPOINTS`: Keys/endpoints usable via `keg.web.redirect`.
- `KEG_LOG_AUTO_CLEAR_HANDLERS`: Remove existing handlers before creating new ones. Default True.
- `KEG_LOG_JSON_FORMAT_STR`: Format string to use for JSON log output (option for syslog)
- `KEG_LOG_JSON_FORMATTER_KWARGS`: Args to provide to JSON formatter
- `KEG_LOG_LEVEL`: Default log level, defaults to INFO and can be modified with CLI options
- `KEG_LOG_MANAGED_LOGGERS`: Keg creates loggers for these paths. Defaults to no paths
- `KEG_LOG_STDOUT_FORMAT_STR`: Format string to use for stdout log output
- `KEG_LOG_STREAM_ENABLED`: Directs Keg to set up a StreamHandler. Default True.
- `KEG_LOG_SYSLOG_ENABLED`: Directs Keg to set up a SysLogHandler. Default True.
- `KEG_LOG_SYSLOG_FORMAT_STR`: Format string to use for syslog log output
- `KEG_LOG_SYSLOG_IDENT`: Log ident for syslog. Defaults to "<app_import_name>.app"
- `KEG_LOG_SYSLOG_JSON`: Directs Keg to output to syslog with JSON. Default False.

- KEG_LOG_SYSLOG_JSON_PREFIX: Prefix to set in JSON output. Default "@cee:"
- KEG_REGISTERED_COMPONENTS: List of paths to import as component extensions
- KEG_SQLITE_ENABLE_FOREIGN_KEYS: Configure SQLite to enforce foreign keys by default

4.9.2 CLI Command

The command `<myapp> develop config` will give detailed information about the files and objects being used to configure an application.

4.9.3 Profile Priority

All configuration classes with the name `DefaultProfile` will be applied to the app's config first.

Then, the configuration classes that match the "selected" profile will be applied on top of the app's existing configuration. This makes the settings from the "selected" profile override any settings from the `DefaultProfile`.

Practically speaking, any configuration that applies to the entire app regardless of what context it is being used in will generally go in `myapp.config` in the `DefaultProfile` class.

4.9.4 Selecting a Configuration Profile

The "selected" profile is the name of the objects that the Keg configuration handling code will look for. It should be a string.

A Keg app considers the "selected" profile as follows:

- If `config_profile` was passed into `myapp.init()` as an argument, use it as the selected profile. The `--profile` cli option uses this method to set the selected profile and therefore has the highest priority.
- Look in the app's environment namespace for "CONFIG_PROFILE". If found, use it.
- If running tests, use "TestProfile". Whether or not the app is operating in this mode is controlled by the use of:
  - `myapp.init(use_test_profile=True)` which is used by `MyApp.testing_prep()`
  - looking in the app's environment namespace for "USE_TEST_PROFILE" which is used by `keg.testing.invoke_command()`
- Look in the app's main config file (`app.config`) and all it's other config files for the variable `DEFAULT_PROFILE`. If found, use the value from the file with highest priority.

4.10 Internationalization

Keg can optionally be installed with the `morphi` library to use `babel` for internationalization:

```
pip install keg[i18n]
```

The `setup.cfg` file is configured to handle the standard message extraction commands. For ease of development and ensuring that all marked strings have translations, a tox environment is defined for testing i18n. This will run commands to update and compile the catalogs, and specify any strings which need to be added.

The desired workflow here is to run tox, update strings in the PO files as necessary, run tox again (until it passes), and then commit the changes to the catalog files.

```
tox -e i18n
```

Python Module Index

k

keg.testing, 13

Index

A

app_config() (*in module keg.testing*), 14
app_environ_get() (*in module keg.utils*), 12
assign() (*keg.web.BaseView method*), 19
assign_blueprint() (*keg.web.BaseView class method*), 19
auto_assign (*keg.web.BaseView attribute*), 19

B

BaseView (*class in keg.web*), 18
blueprint (*keg.web.BaseView attribute*), 19

C

calc_class_fname() (*keg.web.BaseView method*), 19
calc_endpoint() (*keg.web.BaseView class method*), 19
calc_responding_method() (*keg.web.BaseView method*), 19
calc_template_name() (*keg.web.BaseView method*), 19
calc_url() (*keg.web.BaseView class method*), 19
call_loaders() (*keg.web.BaseView method*), 19
check_auth() (*keg.web.BaseView method*), 19
classmethod() (*keg.utils.HybridMethod method*), 12
ClassProperty (*class in keg.utils*), 12
cli_loader_class (*keg.app.Keg attribute*), 9
CLIBase (*class in keg.testing*), 13
config_class (*keg.app.Keg attribute*), 9
ContextManager (*class in keg.testing*), 14
create_blueprint() (*keg.component.KegComponent method*), 11
create_named_blueprint() (*keg.component.KegComponent method*), 11

D

db_visit_modules (*keg.component.KegComponent attribute*), 11
dispatch_request() (*keg.web.BaseView method*), 19

E

ensure_dirs() (*in module keg.utils*), 12
expected_qs_args (*keg.web.BaseView attribute*), 19

G

get_for() (*keg.testing.ContextManager class method*), 14

H

HybridMethod (*class in keg.utils*), 12

I

init_app() (*keg.component.KegComponent method*), 11
init_blueprint() (*keg.web.BaseView class method*), 19
init_blueprints() (*keg.component.KegComponent method*), 12
init_config() (*keg.component.KegComponent method*), 12
init_db() (*keg.component.KegComponent method*), 12
init_routes() (*keg.web.BaseView class method*), 19
inrequest() (*in module keg.testing*), 14
invoke() (*keg.testing.CLIBase method*), 13
invoke_command() (*in module keg.testing*), 15
is_ready() (*keg.testing.ContextManager method*), 14

K

Keg (*class in keg.app*), 9
keg.testing (*module*), 13
KegComponent (*class in keg.component*), 11

L

load_blueprints (*keg.component.KegComponent attribute*), 12 `logger` (*keg.app.Keg attribute*), 9

M

`make_config()` (*keg.app.Keg method*), 9

O

`on_config_complete()` (*keg.app.Keg method*), 9
`on_init_complete()` (*keg.app.Keg method*), 9

P

`process_auto_assign()` (*keg.web.BaseView method*), 19
`process_calling_args()` (*keg.web.BaseView method*), 19
`pymodule_fpaths_to_objects()` (*in module keg.utils*), 12

R

`redirect()` (*in module keg.web*), 19
`render()` (*keg.web.BaseView method*), 19
`request_context()` (*keg.app.Keg method*), 9
`require_authentication` (*keg.web.BaseView attribute*), 19
`route()` (*keg.app.Keg method*), 9

T

`template_folder` (*keg.component.KegComponent attribute*), 12
`template_name` (*keg.web.BaseView attribute*), 19
`testing_prep()` (*keg.app.Keg class method*), 9

U

`url` (*keg.web.BaseView attribute*), 19

V

`visit_modules()` (*in module keg.utils*), 12