Flask Unchained Documentation

Release v0.8.1

Brian Cappello

Jan 18, 2021

CONTENTS

CHAPTER

ONE

INTRODUCING FLASK UNCHAINED

The quickest and easiest way to build large web apps and APIs with Flask and SQLAlchemy

Flask Unchained is a fully integrated, declarative, object-oriented web framework for Flask and its optional-batteries-included extension ecosystem. Flask Unchained is powerful, consistent, highly extensible and completely customizable. Flask Unchained stays true to the spirit and API of Flask while simultaneously introducing powerful new building blocks that enable you to rapidly take your Flask apps to the next level:

- clean and predictable application structure that encourage good design patterns by organizing code as bundles
- no integration headaches between supported libraries and extensions
- no plumbing or boilerplate; everything just works straight out-of-the-box
- simple and consistent patterns for customizing and/or overriding *everything*
- designed with code reuse in mind; your bundles can be distributed as their own Python packages to automatically integrate with other Flask Unchained apps

---

Included bundles & integrated extensions (mostly optional)

- **Controller Bundle**: Enhanced class-based views, enhanced blueprints, declarative routing, and Flask WTF for forms and CSRF protection. The only required bundle.
- **SQLAlchemy Bundle** Flask SQLAlchemy and Flask Migrate for database models and migrations, plus some optional "sugar" on top of SQLAlchemy to make the best ORM in existence even quicker and easier to use with SQLAlchemy Unchained.
- **API Bundle**: RESTful APIs with Flask Marshmallow serializers for SQLAlchemy models.
- **Graphene Bundle**: Flask GraphQL with Graphene for SQLAlchemy models.
- **Security Bundle**: Flask Login for authentication and Flask Principal for authorization.
- **Celery Bundle**: Celery distributed tasks queue.
- Flask Admin, Flask BabelEx, Flask Mail, Flask Session, ...
- Don't like the default stack? With Flask Unchained you can bring your own! Flask Unchained is designed to be so flexible that you could even use it to create your own works-out-of-the-box web framework for Flask with an entirely different stack.

Thanks and acknowledgements

The architecture of how Flask Unchained and its bundles works is only possible thanks to Python 3. The concepts and design patterns Flask Unchained introduces are inspired by the Symfony Framework, which is enterprise-proven and

awesome, aside from the fact that it isn't Python ;)

1.1 Install Flask Unchained

Requires **Python 3.6+**

```
pip install "flask-unchained[dev]"
```

Or, to use **asyncio** by running atop Quart instead of Flask **(experimental!)**:

```
pip install "flask-unchained[asyncio,dev]"  # Requires Python 3.7+
```

Attention

This software is somewhere between alpha and beta quality. It works for me, the design patterns are proven and the core is solid, but especially at the edges there will probably be bugs - and possibly some breaking API changes too. Flask Unchained needs you: please file issues on GitHub if you encounter any problems, have any questions, or have any feedback!

1.2 Hello World

As simple as it gets:

```
# project-root/app.py
from flask_unchained import AppBundle, Controller route

class App(AppBundle):
    pass

class SiteController(Controller):
    @route('/')
    def index(self):
        return 'Hello World from Flask Unchained!'
```

1.2.1 Running the Development Server

And just like that we can run it:

```
cd project-root
UNCHAINED="app" flask run
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

You can now browse to http://127.0.0.1:5000 to see it in action!

Under the easily accessible hood lives Flask Unchained's fully customizable App Factory: the good old call to `app = Flask(__name__)` and everything else necessary to correctly initialize, register, and run your app's code. **No plumbing, no boilerplate, everything just works.**

1.2.2 Testing with pytest

Python's best testing framework comes integrated out-of-the-box:

```
# project-root/test_app.py
from flask_unchained.pytest import HtmlTestClient

class TestSiteController:
    def test_index(self, client: HtmlTestClient):
        r = client.get('site_controller.index')
        assert r.status_code == 200
        assert r.html.count('Hello World from Flask Unchained!') == 1
```

Tests can be run like so:

```
cd project-root
UNCHAINED="app" pytest
============================= test session starts ==============================
platform linux -- Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy-0.13.1
rootdir: /home/user/dev/project-root
plugins: Faker-4.1.1, flask-1.1.0, Flask-Unchained-0.7.9
collected 1 item

test_app.py .                                                            [100%]

============================== 1 passed in 0.05s ===============================
```

1.2.3 The Production App Factory

In development and testing the app factory is automatically used, while in production you call it yourself:

```
# project-root/wsgi.py
from flask_unchained import AppFactory, PROD

app = AppFactory().create_app(env=PROD)
```

We've just shown how Flask Unchained keeps the minimal simplicity micro-frameworks like Flask are renowned for, but to really begin to grasp the power of using Flask Unchained, we need to go bigger than this simple example!

1.3 Hello World for Real

Let's take a peak at some of what this baby can really do to see just how quickly you can start building something more useful:

```
cd project-root
mkdir -p templates/site
pip install "flask-unchained[dev,sqlalchemy]"
```

1.3.1 Quotes App

We're going to create a simple app to store authors and quotes in an SQLite database, and to display them to the user in their browser.

```
# project-root/app.py
from flask_unchained import (FlaskUnchained, AppBundle, BundleConfig,
                             unchained, injectable, generate_csrf)
from flask_unchained.views import Controller, route, param_converter
from flask_unchained.bundles.sqlalchemy import db, ModelManager

# configuration ------------------------------------------------------------------
BUNDLES = ['flask_unchained.bundles.sqlalchemy']

class Config(BundleConfig):
    SECRET_KEY = 'super-secret-key'
    WTF_CSRF_ENABLED = True
    SQLALCHEMY_DATABASE_URI = 'sqlite://'  # memory

class TestConfig(Config):
    WTF_CSRF_ENABLED = False

@unchained.after_request
def set_csrf_token_cookie(response):
    if response:
        response.set_cookie('csrf_token', generate_csrf())
    return response


# database models ----------------------------------------------------------------
class Author(db.Model):
    # models get a primary key (id) and created_at/updated_at columns by default
    name = db.Column(db.String(length=64))
    quotes = db.relationship('Quote', back_populates='author')

class Quote(db.Model):
    text = db.Column(db.Text)
    author = db.relationship('Author', back_populates='quotes')
    author_id = db.foreign_key('Author', nullable=False)


# model managers (dependency-injectable services for database CRUD operations) ------
class AuthorManager(ModelManager):
    class Meta:
        model = Author

class QuoteManager(ModelManager):
    class Meta:
        model = Quote


# views (controllers) ------------------------------------------------------------
class SiteController(Controller):
    class Meta:
        template_folder = 'site'  # the default, auto-determined from class name

    # get the app's instance of the QuoteManager service injected into us
    quote_manager: QuoteManager = injectable
```

(continues on next page)

(continued from previous page)

```
    @route('/')
    def index(self):
        return self.render('index', quotes=self.quote_manager.all())

    @route('/authors/<int:id>')
    @param_converter(id=Author)  # use `id` in the URL to query that Author in the DB
    def author(self, author: Author):
        return self.render('author', author=author)


# declare this module (file) is a Flask Unchained Bundle by subclassing AppBundle ---
class App(AppBundle):
    def before_init_app(self, app: FlaskUnchained) -> None:
        app.url_map.strict_slashes = False

    @unchained.inject()
    def after_init_app(self,
                       app: FlaskUnchained,
                       author_manager: AuthorManager = injectable,
                       quote_manager: QuoteManager = injectable,
                       ) -> None:
        # typically you should use DB migrations and fixtures to perform these steps
        db.create_all()
        quote_manager.create(
            text="Happiness is not a station you arrive at, "
                 "but rather a manner of traveling.",
            author=author_manager.create(name="Margaret Lee Runbeck"))
        quote_manager.create(
            text="Things won are done; joy's soul lies in the doing.",
            author=author_manager.create(name="Shakespeare"))
        db.session.commit()
```

That's the complete app code right there! Hopefully this helps show what is meant by Flask Unchained minimizing plumbing and boilerplate by being *declarative* and *object-oriented*. We just need to add the template files before starting the server:

```
<!-- project-root/templates/layout.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Flask Unchained Quotes</title>
</head>
<body>
    <nav>
        <a href="{{ url_for('site_controller.index') }}">Home</a>
    </nav>
    {% block body %}
    {% endblock %}
</body>
</html>
```

```
<!-- project-root/templates/site/index.html -->
{% extends "layout.html" %}

{% block body %}
```

(continues on next page)

(continued from previous page)

```
    <h1>Flask Unchained Quotes</h1>
    {% for quote in quotes %}
        <blockquote>
            {{ quote.text }}<br />
            <a href="{{ url_for('site_controller.author', id=quote.author.id) }}">
                {{ quote.author.name }}
            </a>
        </blockquote>
    {% endfor %}
{% endblock %}
```

```
<!-- project-root/templates/site/author.html -->
{% extends "layout.html" %}

{% block body %}
    <h1>{{ author.name }} Quotes</h1>
    {% for quote in author.quotes %}
        <blockquote>{{ quote.text }}</blockquote>
    {% endfor %}
{% endblock %}
```

Fire it up:

```
export UNCHAINED="app"
flask urls
Method(s)  Rule                          Endpoint                View
-------------------------------------------------------------------------------------
↪-----
      GET  /                           site_controller.index   app.SiteController.
↪index
      GET  /authors/<int:id>           site_controller.author  app.SiteController.
↪author
```

```
export UNCHAINED="app"
flask run
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

1.3.2 Adding a RESTful API

Flask Unchained includes an API Bundle integrating RESTful support atop the Controller Bundle with SQLAlchemy models and Marshmallow serializers. Basic out-of-the-box usage is dead simple.

Install dependencies for the API Bundle:

```
pip install "flask-unchained[api]"
```

And add the following code to the bottom of `project-root/app.py`:

```
# append to project-root/app.py
from flask_unchained.bundles.api import ma, ModelResource, ModelSerializer
from flask_unchained.routes import controller, resource, prefix
```

(continues on next page)

(continued from previous page)

```
BUNDLES += ['flask_unchained.bundles.api']

# db model serializers ------------------------------------------------------------
class AuthorSerializer(ModelSerializer):
    class Meta:
        model = Author
        url_prefix = '/authors'  # the default, auto-determined from model class name

    quotes = ma.Nested('QuoteSerializer', only=('id', 'text'), many=True)

class QuoteSerializer(ModelSerializer):
    class Meta:
        model = Quote

    author = ma.Nested('AuthorSerializer', only=('id', 'name'))

# api views -----------------------------------------------------------------------
class AuthorResource(ModelResource):
    class Meta:
        model = Author
        include_methods = ('get', 'list')

class QuoteResource(ModelResource):
    class Meta:
        model = Quote
        exclude_methods = ('create', 'patch', 'put', 'delete')

# use declarative routing for specifying views with fine-grained control over URLs
routes = lambda: [
    controller(SiteController),
    prefix('/api/v1', [
        resource(AuthorResource),
        resource(QuoteResource),
    ]),
]
```

We can take a look at the new URLs:

```
flask urls
Method(s)  Rule                          Endpoint                View
-------------------------------------------------------------------------------------
↪-----
      GET  /                            site_controller.index   app.SiteController.index
      GET  /authors/<int:id>            site_controller.author  app.SiteController.
↪author
      GET  /api/v1/authors              author_resource.list    app.AuthorResource.list
      GET  /api/v1/authors/<int:id>     author_resource.get     app.AuthorResource.get
      GET  /api/v1/quotes               quote_resource.list     app.QuoteResource.list
      GET  /api/v1/quotes/<int:id>      quote_resource.get      app.QuoteResource.get
```

And run it:

```
flask run
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

1.3.3 Securing the App

Flask Unchained also includes the Security Bundle as a foundation for handling authentication and authorization in your apps. It is designed to be extended and customized to your needs - like everything in Flask Unchained! - but it also works out-of-the-box for when all it provides is sufficient for your needs. Let's set things up to require an authenticated user to use the app's API.

Install dependencies for the Security Bundle:

```
pip install "flask-unchained[session,security]"
```

And add the following to the bottom of your `project-root/app.py`:

```
# append to project-root/app.py
from flask_unchained.bundles.security import SecurityController, auth_required
from flask_unchained.bundles.security import SecurityService, UserManager
from flask_unchained.bundles.security.models import User as BaseUser
from flask_unchained.bundles.sqlalchemy import db

# enable the session and security bundles
BUNDLES += ['flask_unchained.bundles.session',
            'flask_unchained.bundles.security']

# configure server-side sessions
Config.SESSION_TYPE = 'sqlalchemy'
Config.SESSION_SQLALCHEMY_TABLE = 'flask_sessions'

# configure security
Config.SECURITY_REGISTERABLE = True  # enable user registration
AuthorResource.Meta.decorators = (auth_required,)
QuoteResource.Meta.decorators = (auth_required,)

# want to add fields to the database model for users? no problem!
# just subclass it, keeping the same original class name
class User(BaseUser):
    favorite_color = db.Column(db.String)

# add the Security Controller views to our app
routes = lambda: [
    controller(SiteController),
    controller(SecurityController),
    prefix('/api/v1', [
        resource('/authors', AuthorResource),
        resource('/quotes', QuoteResource),
    ]),
]

# create a demo user and log them in when the dev server starts
@unchained.before_first_request()
@unchained.inject()
def create_and_login_demo_user(user_manager: UserManager = injectable,
                               security_service: SecurityService = injectable):
    user = user_manager.create(email='demo@example.com',
                               password='password',
                               favorite_color='magenta',
                               is_active=True,
                               commit=True)
    security_service.login_user(user)
```

By default the Security Bundle only comes with the `/login` and `/logout` URLs enabled. Let's confirm we've also enabled `/register`:

```
flask urls
Method(s)  Rule                          Endpoint                       View
-------------------------------------------------------------------------------------------
↪----------------------------------------
      GET  /                           site_controller.index          quotes.
↪SiteController.index
      GET  /authors/<int:id>           site_controller.author         quotes.
↪SiteController.author
      GET  /api/v1/authors             author_resource.list           quotes.
↪AuthorResource.list
      GET  /api/v1/authors/<int:id>    author_resource.get            quotes.
↪AuthorResource.get
      GET  /api/v1/quotes              quote_resource.list            quotes.
↪QuoteResource.list
      GET  /api/v1/quotes/<int:id>     quote_resource.get             quotes.
↪QuoteResource.get
GET, POST  /login                      security_controller.login      flask_unchained.
↪bundles.security.SecurityController.login
      GET  /logout                     security_controller.logout     flask_unchained.
↪bundles.security.SecurityController.logout
GET, POST  /register                   security_controller.register   flask_unchained.
↪bundles.security.SecurityController.register
```

```
flask run
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

NOTE: you'll need to logout the demo user by visiting http://127.0.0.1:5000/logout before the login and register endpoints will work.

1.4 Going Big (Project Layout)

When you want to expand beyond a single file, Flask Unchained defines a standardized (but configurable) folder structure for you so that everything just works. A typical structure looks like this:

```
/home/user/dev/project-root
├── unchained_config.py # the Flask Unchained config
├── app                 # the app bundle Python package
│   ├── admins          # Flask-Admin model admins
│   ├── commands        # Click CLI groups/commands
│   ├── extensions      # Flask extensions
│   ├── models          # SQLAlchemy models
│   ├── fixtures        # SQLAlchemy model fixtures (for seeding the dev db)
│   ├── serializers     # Marshmallow serializers (aka schemas)
│   ├── services        # dependency-injectable Services
│   ├── tasks           # Celery tasks
│   ├── templates       # Jinja2 templates
│   ├── views           # Controllers, Resources and ModelResources
│   ├── __init__.py     # your AppBundle subclass
│   ├── config.py       # your app config
│   └── routes.py       # declarative routes
```

(continues on next page)

(continued from previous page)

```
├── bundles              # custom bundles and/or bundle extensions/overrides
│   └── security         # a customized/extended Security Bundle
│       ├── models
│       ├── serializers
│       ├── services
│       ├── templates
│       └── __init__.py
├── db
│   └── migrations       # migrations generated by Flask-Migrate
├── static               # the top-level static assets folder
├── templates            # the top-level templates folder
└── tests                # your pytest tests
```

Want to start building now? Check out the *Tutorial*! There are also some example open source apps available:

- Flask React SPA
- Flask Techan Unchained
- Open a PR to add yours!

1.5 Features

1.5.1 Bundles

Bundles are powerful and flexible. They are standalone Python packages that can do anything from integrate Flask extensions to be full-blown apps your app can integrate and extend (like, say, a blog or web store). Conceptually, a bundle *is* a blueprint, and Flask Unchained gives you complete control to configure not only which views from each bundle get registered with your app and at what routes, but also to extend and/or override anything else you might want to from the bundles you enable.

Some examples of what you can customize from bundles include configuration, controllers, resources, and routes, templates, extensions and services, and models and serializers. Each uses simple and consistent patterns that work the same way across every bundle. Extended/customized bundles can themselves also be distributed as their own projects, and support the same patterns for customization, ad infinitum.

Bundle Structure

The example "hello world" app bundle lived in a single file, while a "full" bundle package typically consists of many modules (as shown just above under Project Layout). The module locations for your code are customizable on a per-bundle basis by setting class attributes on your `Bundle` subclass, for example:

```
# your_custom_bundle/__init__.py

from flask_unchained import Bundle

class YourCustomBundle(Bundle):
    config_module_name = 'settings'
    routes_module_name = 'urls'
    views_module_names = ['controllers', 'resources', 'views']
```

You can see the default module names and the override attribute names to set on your `Bundle` subclass by printing the ordered list of hooks that will run for your app using `flask unchained hooks`:

```
flask unchained hooks

Hook Name              Default Bundle Module  Bundle Module Override Attr
-------------------------------------------------------------------------
register_extensions    extensions             extensions_module_names
models                 models                 models_module_names
configure_app          config                 config_module_name
init_extensions        extensions             extensions_module_names
services               services               services_module_names
commands               commands               commands_module_names
routes                 routes                 routes_module_name
bundle_blueprints      (None)                 (None)
blueprints             views                  blueprints_module_names
views                  views                  views_module_names
model_serializers      serializers            model_serializers_module_names
model_resources        views                  model_resources_module_names
celery_tasks           tasks                  celery_tasks_module_names
```

Bundle Blueprints

Bundles *are* blueprints, so if you want to define request/response functions that should only run for views from a specific bundle, you can do that like so:

```python
from flask_unchained import Bundle, unchained

class YourCoolBundle(Bundle):
    name = 'your_cool_bundle'  # the default (snake_cased class name)

@unchained.your_cool_bundle.before_request
def this_only_runs_before_requests_to_views_from_your_cool_bundle():
    pass

# the other supported decorators are also available:
@unchained.your_cool_bundle.after_request
@unchained.your_cool_bundle.teardown_request
@unchained.your_cool_bundle.context_processor
@unchained.your_cool_bundle.url_defaults
@unchained.your_cool_bundle.url_value_preprocessor
@unchained.your_cool_bundle.errorhandler
```

The API here is the same as `flask.Blueprint`, however, its methods must be accessed via the *Unchained* extension. The syntax is `@unchained.bundle_name.blueprint_method_name`.

Wait but why?

Sadly, there are some very serious technical limitations with the implementation of `flask.Blueprint` such that its direct usage breaks the power and flexibility of Flask Unchained. Under the hood, Flask Unchained does indeed use a blueprint for each bundle - you just never interact with them directly.

You can *technically* continue using `flask.Blueprint` **strictly for views in your app bundle**, however this support is only kept around for porting purposes. Note that even in your app bundle, views from blueprints unfortunately will not work with declarative routing.

Extending and Overriding Bundles

Extending and overriding bundles is pretty simple. All you need to do is subclass the bundle you want to extend in its own Python package, and include that package in your `unchained_config.BUNDLES` instead of the original bundle. There is no limit to the depth of the bundle hierarchy (other than perhaps your sanity). So, for example, to extend the Security Bundle, it would look like this:

```
# project-root/bundles/security/__init__.py

from flask_unchained.bundles.security import SecurityBundle as BaseSecurityBundle

class SecurityBundle(BaseSecurityBundle):
    pass
```

```
# project-root/unchained_config.py

BUNDLES = [
    # ...
    'bundles.security',
    'app',
]
```

The App Bundle

When defining the app bundle, you must subclass *AppBundle* instead of *Bundle*:

```
# project-root/app/__init__.py

from flask_unchained import AppBundle

class App(AppBundle):
    pass
```

Everything about your app bundle is otherwise the same as regular bundles, except **the app bundle can extend and/or override anything from any bundle**.

1.5.2 The Unchained Extension

The "orchestrator" that ties everything together. It handles dependency injection and enables access to much of the public API of `flask.Flask` and `flask.Blueprint`:

```
# project-root/app.py
from flask_unchained import unchained, injectable

@unchained.inject()
def print_hello(name: str, hello_service: HelloService = injectable):
    print(hello_service.hello_world(name))

@unchained.before_first_request
def runs_once_at_startup():
    print_hello("App")

@unchained.app.after_request
def runs_after_each_request_to_an_app_bundle_view(response):
```

(continues on next page)

(continued from previous page)

```
    print_hello("Response")
    return response
```

The *Unchained* extension also plays a role in the app factory:

1.5.3 The App Factory

The *AppFactory* discovers all the code from your app and its bundles, and then with it automatically initializes, configures, and "boots up" the Flask `app` instance for you. I know that sounds like magic, but it's actually quite easy to understand, and every step it takes can be customized by you if necessary. In barely-pseudo-code, the app factory looks like this:

```
from flask import Flask
from flask_unchained import DEV, PROD, STAGING, TEST

class AppFactory:
    APP_CLASS = Flask

    def create_app(self, env: Union[DEV, PROD, STAGING, TEST]) -> Flask:
        # load the Unchained Config and configured bundles
        unchained_config = self.load_unchained_config(env)
        app_bundle, bundles = self.load_bundles(unchained_config.BUNDLES)

        # instantiate the Flask app instance
        app = self.APP_CLASS(app_bundle.name, **kwargs_from_unchained_config)

        # let bundles configure the app pre-initialization
        for bundle in bundles:
            bundle.before_init_app(app)

        # discover code from bundles and boot the app using hooks
        unchained.init_app(app, bundles)
            # the Unchained extension runs hooks in their correct order:
            # (there may be more hooks depending on which bundles you enable)
            RegisterExtensionsHook.run_hook(app, bundles)
            ConfigureAppHook.run_hook(app, bundles)
            InitExtensionsHook.run_hook(app, bundles)
            RegisterServicesHook.run_hook(app, bundles)
            RegisterCommandsHook.run_hook(app, bundles)
            RegisterRoutesHook.run_hook(app, bundles)
            RegisterBundleBlueprintsHook.run_hook(app, bundles)

        # let bundles configure the app post-initialization
        for bundle in bundles:
            bundle.after_init_app(app)

        # return the app instance ready to rock'n'roll
        return app
```

The `flask` and `pytest` CLI commands automatically use the app factory for you, while in production you have to call it yourself:

```
# project-root/wsgi.py
from flask_unchained import AppFactory, PROD
```

(continues on next page)

(continued from previous page)

```
app = AppFactory().create_app(env=PROD)
```

For a deeper look check out *How Flask Unchained Works*.

1.5.4 Controllers, Resources, and Templates

The controller bundle includes two base classes that all of your views should extend. The first is *Controller*, and the second is *Resource*, meant for building RESTful APIs.

Controller

Chances are *Controller* is the base class you want to extend, unless you're building a RESTful API. Under the hood, the implementation is actually very similar to `flask.views.View`, however, they're not compatible. Controllers include a bit of magic:

```python
# your_bundle/views.py

from flask_unchained import Controller, route, injectable

class SiteController(Controller):
    # all of class Meta is optional (automatic defaults shown)
    class Meta:
        abstract: bool = False
        url_prefix = Optional[str] = '/'          # aka no prefix
        endpoint_prefix: str = 'site_controller'  # snake_cased class name
        template_folder: str = 'site'             # snake_cased class name prefix
        template_file_extension: Optional[str] = '.html'
        decorators: List[callable] = ()

    # controllers automatically support dependency injection
    name_service: NameService = injectable

    @route('/foobaz', methods=['GET', 'POST'])
    def foo_baz():
        return self.render('site/foo_baz.html')  # template paths can be explicit

    # defaults to @route('/view-one', methods=['GET'])
    def view_one():
        # or just the filename
        return self.render('one')  # equivalent to 'site/one.html'

    # defaults to @route('/view-two', methods=['GET'])
    def view_two():
        return self.render('two')

    # utility function (gets no route)
    def _protected_function():
        return 'not a view'
```

On any subclass of `Controller` that isn't abstract, all public methods are automatically assigned default routing rules. In the example above, `foo_baz` has a route decorator, but `view_one` and `view_two` do not. The undecorated views will be assigned default routing rules of `/view-one` and `/view-two` respectively (the default is to convert the method name to kebab-case). Protected methods (those prefixed with _) are not assigned routes.

Templates

Flask Unchained uses the Jinja templating language, just like Flask.

By default bundles are configured to use a `templates` subfolder. This is customizable per-bundle:

```
# your_bundle/__init__.py

from flask_unchained import Bundle

class YourBundle(Bundle):
    template_folder = 'templates'  # the default
```

Controllers each have their own template folder within `Bundle.template_folder`. It defaults to the snake_cased class name, with the suffixes `Controller` or `View` stripped (if any). You can customize it using `Controller.Meta.template_folder`.

The default file extension used for templates is configured by setting `TEMPLATE_FILE_EXTENSION` in your app config. It defaults to `.html`, and is also configurable on a per-controller basis by setting `Controller.Meta.template_file_extension`.

Therefore, the above controller corresponds to the following templates folder structure:

```
./your_bundle
├── templates
│   └── site
│       ├── foo_baz.html
│       ├── one.html
│       └── two.html
├── __init__.py
└── views.py
```

Extending and Overriding Templates

Templates can be overridden by placing an equivalently named template higher up in the bundle hierarchy (i.e. in a bundle extending another bundle, or in your app bundle).

So for example, the Security Bundle includes default templates for all of its views. They are located at `security/login.html`, `security/register.html`, and so on. Thus, to override them, you would make a `security` folder in your app bundle's `templates` folder and put your customized templates with the same names in it. You can even extend the template you're overriding, using the standard Jinja syntax (this doesn't work in regular Flask apps):

```
{# your_app_or_security_bundle/templates/security/login.html #}

{% extends 'security/login.html' %}

{% block content %}
   <h1>Login</h1>
   {{ render_form(login_user_form, endpoint='security_controller.login') }}
{% endblock %}
```

If you encounter problems, you can set the `EXPLAIN_TEMPLATE_LOADING` config option to `True` to help debug what's going on.

Resources (API Controllers)

The *Resource* class extends *Controller* to add support for building RESTful APIs. The implementation draws much inspiration from Flask-RSETful (specifically, the Resource and Api classes). Using *Resource* adds a bit more magic to controllers around specific methods:

Method name on your Resource subclass	HTTP Method	URL Rule
list	GET	/
create	POST	/
get	GET	/<cls.Meta.member_param>
patch	PATCH	/<cls.Meta.member_param>
put	PUT	/<cls.Meta.member_param>
delete	DELETE	/<cls.Meta.member_param>

If you implement any of these methods, then the shown URL rules will automatically be used.

So, for example:

```
from http import HTTPStatus
from flask_unchained import Resource, injectable, param_converter, request
from flask_unchained.bundles.security import User, UserManager

class UserResource(Resource):
    # class Meta is optional on resources (automatic defaults shown)
    class Meta:
        url_prefix = '/users'
        member_param = '<int:id>'
        unique_member_param = '<int:user_id>'

    # resources are controllers, so they support dependency injection
    user_manager: UserManager = injectable

    def list():
        return self.jsonify(dict(users=self.user_manager.all()))
        # NOTE: returning SQLAlchemy models directly like this is
        # only supported by ModelResource from the API Bundle

    def create():
        data = request.get_json()
        user = self.user_manager.create(**data, commit=True)
        return self.jsonify(dict(user=user), code=HTTPStatus.CREATED)

    @param_converter(id=User)
    def get(user):
        return self.jsonify(dict(user=user)

    @param_converter(id=User)
    def patch(user):
        data = request.get_json()
        user = self.user_manager.update(user, **data, commit=True)
        return self.jsonify(dict(user=user))

    @param_converter(id=User)
    def put(user):
        data = request.get_json()
        user = self.user_manager.update(user, **data, commit=True)
```

(continues on next page)

(continued from previous page)

```
        return self.jsonify(dict(user=user))

    @param_converter(id=User)
    def delete(user):
        self.user_manager.delete(user, commit=True)
        return self.make_response('', code=HTTPStatus.NO_CONTENT)
```

Registered like so:

```
routes = lambda: [
    resource(UserResource),
]
```

Results in the following routes:

```
GET     /users              UserResource.list
POST    /users              UserResource.create
GET     /users/<int:id>     UserResource.get
PATCH   /users/<int:id>     UserResource.patch
PUT     /users/<int:id>     UserResource.put
DELETE  /users/<int:id>     UserResource.delete
```

1.5.5 Declarative Routing

Using declarative routing, your app bundle has final say over which views (from all bundles) should get registered with the app, as well as their routing rules. By default, it uses the rules decorated on views:

```
# project-root/app/routes.py

from flask_unchained import (controller, resource, func, include, prefix,
                             delete, get, patch, post, put, rule)

from flask_unchained.bundles.security import SecurityController

from .views import SiteController

routes = lambda: [
    controller(SiteController),
    controller(SecurityController),
]
```

By running `flask urls`, we can verify it does what we want:

```
flask urls
Method(s)  Rule                      Endpoint                    View
-----------------------------------------------------------------------------------
 ↪----------------------------------------
      GET  /                        site_controller.index       app.views.
 ↪SiteController.index
GET, POST  /login                   security_controller.login   flask_unchained.
 ↪bundles.security.views.SecurityController.login
      GET  /logout                  security_controller.logout  flask_unchained.
 ↪bundles.security.views.SecurityController.logout
```

Declarative routing can also be *much* more powerful when you want it to be. For example, to build a RESTful SPA with the Security Bundle, your routes might look like this:

```
# project-root/app/routes.py

from flask_unchained import (controller, resource, func, include, prefix,
                             delete, get, patch, post, put, rule)

from flask_unchained.bundles.security import SecurityController, UserResource

from .views import SiteController

routes = lambda: [
    controller(SiteController),

    controller('/auth', SecurityController, rules=[
        get('/reset-password/<token>', SecurityController.reset_password,
            endpoint='security_api.reset_password'),
    ]),
    prefix('/api/v1', [
        controller('/auth', SecurityController, rules=[
            get('/check-auth-token', SecurityController.check_auth_token,
                endpoint='security_api.check_auth_token', only_if=True),
            post('/login', SecurityController.login,
                 endpoint='security_api.login'),
            get('/logout', SecurityController.logout,
                endpoint='security_api.logout'),
            post('/send-confirmation-email',
                 SecurityController.send_confirmation_email,
                 endpoint='security_api.send_confirmation_email'),
            post('/forgot-password', SecurityController.forgot_password,
                 endpoint='security_api.forgot_password'),
            post('/reset-password/<token>', SecurityController.reset_password,
                 endpoint='security_api.post_reset_password'),
            post('/change-password', SecurityController.change_password,
                 endpoint='security_api.change_password'),
        ]),
        resource('/users', UserResource),
    ]),
]
```

Which results in the following:

```
flask urls
Method(s)  Rule                                  Endpoint                                                                ␣
↪ View
------------------------------------------------------------------------------------------
↪--------------------------------------------------------------------------------------
      GET  /                                     site_controller.index                                                   ␣
↪ app.views.SiteController.index
      GET  /auth/reset-password/<token>          security_api.reset_password                                             ␣
↪ flask_unchained.bundles.security.views.SecurityController.reset_password
      GET  /api/v1/auth/check-auth-token         security_api.check_auth_token                                           ␣
↪ flask_unchained.bundles.security.views.SecurityController.check_auth_token
     POST  /api/v1/auth/login                    security_api.login                                                      ␣
↪ flask_unchained.bundles.security.views.SecurityController.login
      GET  /api/v1/auth/logout                   security_api.logout                                                     ␣
↪ flask_unchained.bundles.security.views.SecurityController.logout
     POST  /api/v1/auth/send-confirmation-email  security_api.send_confirmation_email                                    ␣
↪ flask_unchained.bundles.security.views.SecurityController.send_confirmation_email
```

(continues on next page)

(continued from previous page)

```
    POST  /api/v1/auth/forgot-password          security_api.forgot_password
↪ flask_unchained.bundles.security.views.SecurityController.forgot_password
    POST  /api/v1/auth/reset-password/<token>   security_api.post_reset_password
↪ flask_unchained.bundles.security.views.SecurityController.reset_password
    POST  /api/v1/auth/change-password          security_api.change_password
↪ flask_unchained.bundles.security.views.SecurityController.change_password
    POST  /api/v1/users                         user_resource.create
↪ flask_unchained.bundles.security.views.UserResource.create
     GET  /api/v1/users/<int:id>                user_resource.get
↪ flask_unchained.bundles.security.views.UserResource.get
   PATCH  /api/v1/users/<int:id>                user_resource.patch
↪ flask_unchained.bundles.security.views.UserResource.patch
```

Here is a summary of the functions imported at the top of the `routes.py` module:

Table 1: Declarative Routing Functions

Function	Description
include()	Include all of the routes from the specified module at that point in the tree.
prefix()	Prefixes all of the child routing rules with the given prefix.
func()	Registers a function-based view with the app, optionally specifying the routing rules.
controller()	Registers a controller and its views with the app, optionally customizing the routes to register.
resource()	Registers a resource and its views with the app, optionally customizing the routes to register.
rule()	Define customizations to a controller/resource method's route rules.
get(), *patch()*, *post()*, *put()*, and *delete()*	Like *rule()* except specifically for each HTTP method.

1.5.6 Dependency Injection and Services

Flask Unchained supports dependency injection of services and extensions (by default).

Services

For services to be automatically discovered, they must subclass `Service` and (by default) live in a bundle's `services` or `managers` modules. You can however manually register anything as a "service", even plain values if you really wanted to, using the `unchained.service` decorator and/or the `unchained.register_service` method:

```
from flask_unchained import unchained

@unchained.service(name='something')
class SomethingNotExtendingService:
    pass

A_CONST = 'a constant'
unchained.register_service('A_CONST', A_CONST)
```

Services can request other services be injected into them, and as long as there are no circular dependencies, it will work:

```
from flask_unchained import Service, injectable

class OneService(Service):
    something: SomethingNotExtendingService = injectable
    A_CONST: str = injectable

class TwoService(Service):
    one_service: OneService = injectable
```

By setting the default value of a class attribute or function/method argument to the `flask_unchained.injectable` constant, you are informing the *Unchained* extension that it should inject those arguments.

Important

The names of services must be unique across *all* of the bundles in your app (by default services are named as the snake_cased class name). If there are any conflicting class names then you will need to use the `unchained.service` decorator or the `unchained.register_service` method to customize the name the service gets registered under:

```
from flask_unchained import Service, unchained

@unchained.service('a_unique_name')
class ServiceWithNameConflict(Service):
    pass
```

Automatic Dependency Injection

Dependency injection works automatically on all classes extending `Service` and `Controller`. The easiest way is with class attributes:

```
from flask_unchained import Controller, injectable
from flask_unchained.bundles.security import Security, SecurityService
from flask_unchained.bundles.sqlalchemy import SessionManager

class SecurityController(Controller):
    security: Security = injectable
    security_service: SecurityService = injectable
    session_manager: SessionManager = injectable
```

It also works on the constructor, which is functionally equivalent, just more verbose:

```
class SiteController(Controller):
    def __init__(self, security: Security = injectable):
        self.security = security
```

Manual Dependency Injection

You can use the `unchained.inject` decorator just about anywhere else you want to inject something:

```
from flask_unchained import unchained, injectable

# decorate a class to use class attributes injection
@unchained.inject()
class Foobar:
    some_service: SomeService = injectable

    # or you can decorate individual methods
    @unchained.inject()
    def a_method(self, another_service: AnotherService = injectable):
        pass

# it works on regular functions too
@unchained.inject()
def a_function(some_service: SomeService = injectable):
    pass
```

Alternatively, you can also use `unchained.get_local_proxy`:

```
from flask_unchained import unchained

db = unchained.get_local_proxy('db')
```

Extending and Overriding Services

Services are just classes, so they follow the normal Python inheritance rules. All you need to do is name your service the same as the one you want to customize, placed in the `services` module higher up in the bundle hierarchy (i.e. in a bundle extending another bundle, or in your app bundle).

1.5.7 Integrating Flask Extensions

Extensions that can be used in Flask Unchained bundles have a few limitations. The primary one being, the extension must implement `init_app`, and its signature must take a single argument: `app`. Some extensions fit this restriction out of the box, but often times you will need to subclass the extension to make sure its `init_app` signature matches. You can create new config options to replace arguments that were originally passed into the extension's constructor and/or `init_app` method.

In order for Flask Unchained to actually discover and initialize the extension you want to include, they must be placed in your bundle's `extensions` module. It looks like this:

```
# your_bundle/extensions.py

from flask_whatever import WhateverExtension

whatever = WhateverExtension()

EXTENSIONS = {
    'whatever': whatever,
}
```

The keys of the `EXTENSIONS` dictionary serve as the name that will be used to reference the extension at runtime (and for dependency injection). There can be multiple extensions per bundle, and you can also declare other extensions as dependencies that must be initialized before yours:

```
EXTENSIONS = {
    'whatever': (whatever, ['dep_ext_one', 'dep_ext_two']),
}
```

CHAPTER

TWO

TUTORIAL

This tutorial will walk you through creating a basic portfolio application for monitoring your investments. Users will be able to register, log in, create portfolios and manage the stocks in them. You will be able to package and install the application on other computers.

It is assumed you're already familiar with:

- Python 3.6+. The official tutorial is a great way to learn or review.
- The Jinja2 templating engine. See the official docs for more info.

Table of Contents

2.1 Getting Started

2.1.1 Install Flask Unchained

Create a new directory and enter it:

```
mkdir hello-flask-unchained && cd hello-flask-unchained
```

The tutorial will assume you're working from the `hello-flask-unchained` directory from now on. All commands are assumed to be run from this top-level project directory, and the file names at the top of each code block are also relative to this directory.

Next, let's create a new virtualenv, install Flask Unchained into it, and activate it:

```
# create our virtualenv and activate it
python3 -m venv venv && . venv/bin/activate

# install flask-unchained
pip install "flask-unchained[dev]"

# reactivate the virtualenv so that pytest will work correctly
deactivate && . venv/bin/activate
```

Python Virtual Environments

There are other ways to create virtualenvs for Python, and if you have a different preferred method that's fine, but you should always use a virtualenv by some way or another.

2.1.2 Project Layout

Just like Flask, Flask Unchained apps can be written either as a single file or in multiple files following a (configurable) naming convention. A large project might have a folder structure that looks like this:

```
/home/user/dev/hello-flask-unchained
├── app                  # your app bundle package
│   ├── admins           # model admins
│   ├── commands         # click groups/commands
│   ├── extensions       # extension instances
│   ├── models           # sqlalchemy models
│   ├── serializers      # marshmallow serializers (aka schemas)
│   ├── services         # dependency-injectable services
│   ├── tasks            # celery tasks
│   ├── templates        # jinja templates
│   ├── views            # controllers and resources
│   ├── __init__.py
│   ├── config.py        # app config
│   └── routes.py        # declarative routes
├── assets               # static assets to be handled by Webpack
│   ├── images
│   ├── scripts
│   └── styles
├── bundles              # third-party bundle extensions/overrides
│   └── security         # a customized/extended Security Bundle
│       ├── models
│       ├── serializers
│       ├── templates
│       └── __init__.py
├── db
│   ├── fixtures         # sqlalchemy model fixtures (for seeding the dev db)
│   └── migrations       # alembic migrations (generated by flask-migrate)
├── static               # static assets (Webpack compiles to here, and Flask
│                        #  serves this folder at /static (by default))
├── templates            # the top-level templates folder
├── tests                # your pytest tests
├── webpack              # Webpack configs
└── unchained_config.py # the flask unchained config
```

By the end of this tutorial, we'll have built something very close. But for now, let's start with the basics.

2.1.3 A Minimal Hello World App

The starting project layout of our hello world app is three files:

```
/home/user/dev/hello-flask-unchained
├── unchained_config.py
├── app.py
└── test_app.py
```

Let's create them:

```
touch unchained_config.py app.py test_app.py
```

And the code:

```python
# unchained_config.py

BUNDLES = [
    'app',
]
```

```python
# app.py

from flask_unchained import AppBundle, Controller, route

class App(AppBundle):
    pass

class SiteController(Controller):
    @route('/')
    def index(self):
        return 'Hello World!'
```

Whenever you create a new app in Flask Unchained, you start by creating a new "app bundle": This is an overloaded term. The app bundle, conceptually, *is* your app. Literally, the app bundle is a subclass of *AppBundle* that must live in your app bundle's module root (*app.py* here).

We can now start the development server with `flask run` and you should see your site running at http://localhost:5000:

```
flask run
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Let's add a quick test before we continue.

```python
# test_app.py

class TestSiteController:
    def test_index(self, client):
        r = client.get('site_controller.index')
        assert r.status_code == 200
        assert r.html.count('Hello World!') == 1
```

Here, we're using the HTTP `client` pytest fixture to request the URL for the endpoint `"site_controller.index"`, verifying the response has a status code of `200`, and lastly checking that the string `"Hello World!"` is in the response.

Let's make sure it passes:

```
pytest
======================== test session starts ========================
platform linux -- Python 3.6.6, pytest-3.6.4, py-1.5.4, pluggy-0.7.1
rootdir: /home/user/dev/hello-flask-unchained, inifile:
plugins: flask-0.10.0, Flask-Unchained-0.8.0
collected 1 item

test_app.py .                                                  [100%]
===================== 1 passed in 0.18 seconds ======================
```

NOTE: If you get any errors, you may need to deactivate and reactivate your virtualenv if you haven't already since installing `pytest`.

If you haven't already, now would be a good time to initialize a git repo and make our first commit. Before we do that though, let's add a `.gitignore` file to make sure we don't commit anything that shouldn't be.

```
# .gitignore

*.egg-info
*.pyc
.coverage
.cache/
.pytest_cache/
.tox/
__pycache__/
build/
coverage_html_report/
db/*.sqlite
dist/
docs/_build
venv/
```

Initialize the repo and make our first commit:

```
git init
git add .

# review to make sure it's not going to do anything you don't want it to:
git status

git commit -m 'initial hello world commit'
```

OK, everything works, but this is about as basic as it gets. Let's make things a bit more interesting by moving on to *Views, Templates, and Static Assets*.

2.2 Views, Templates, and Static Assets

So far we've been returning a raw string from our view function. This works fine for demo purposes, however, in the real world you'll most often use template files. Let's create a directory each for our templates and static assets.

```
mkdir -p templates static \
  && touch templates/layout.html templates/_navbar.html templates/_flashes.html
```

These directories are the standard locations the `Flask` constructor expects, so they will automatically be used. (Note however that if just created either of these directories, then you will need to restart the development server for them to be picked up by it.)

Our site looks pretty weak as it stands. Let's add Bootstrap to spruce things up a bit:

```
wget https://stackpath.bootstrapcdn.com/bootstrap/4.1.2/js/bootstrap.min.js -O static/
↪bootstrap-v4.1.2.min.js \
  && wget https://stackpath.bootstrapcdn.com/bootstrap/4.1.2/css/bootstrap.min.css -O␣
↪static/bootstrap-v4.1.2.min.css \
  && wget https://code.jquery.com/jquery-3.3.1.slim.min.js -O static/jquery-v3.3.1.
↪slim.min.js \
  && wget https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js -
↪O static/popper-v1.14.3.min.js
```

2.2.1 Layout Template

In order to share as much code as possible between templates, it's best practice to abstract away the shared boilerplate into `templates/layout.html`. Just like vanilla Flask, Flask Unchained uses the Jinja2 templating engine. If you're unfamiliar with what anything below is doing, I recommend checking out the excellent official Jinja2 documentation.

Now let's write our `templates/layout.html` file:

```
{# templates/layout.html #}

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-
↪fit=no">

    <title>
      {% block title %}Hello Flask Unchained!{% endblock %}
    </title>

    {% block stylesheets %}
      <link rel="stylesheet" href="{{ url_for('static', filename='bootstrap-v4.1.2.
↪min.css') }}">
    {% endblock stylesheets %}

    {% block extra_head %}
    {% endblock extra_head %}
  </head>

  <body>
    {% block header %}
      <header>
        {% block navbar %}
          {% include '_navbar.html' %}
        {% endblock %}
      </header>
    {% endblock%}

    {% block body %}
      <div class="container">
        {% include '_flashes.html' %}
        {% block content %}
        {% endblock content %}
      </div>
    {% endblock body %}

    {% block javascripts %}
      <script src="{{ url_for('static', filename='jquery-v3.3.1.slim.min.js') }}"></
↪script>
      <script src="{{ url_for('static', filename='popper-v1.14.3.min.js') }}"></
↪script>
      <script src="{{ url_for('static', filename='bootstrap-v4.1.2.min.js') }}"></
↪script>
    {% endblock javascripts %}
  </body>
</html>
```

And also the included `templates/_flashes.html` and `templates/_navbar.html` templates:

```
{# templates/_flashes.html #}

{% with messages = get_flashed_messages(with_categories=True) %}
{% if messages %}
  <div class="row flashes">
    <div class="col">
      {% for category, message in messages %}
        <div class="alert alert-{{ category }} alert-dismissable fade show" role=
↪"alert">
          {{ message }}
          <button type="button" class="close" data-dismiss="alert" aria-label="Close">
            <span aria-hidden="true">&times;</span>
          </button>
        </div>
      {% endfor %}
    </div>
  </div>
{% endif %}
{% endwith %}
```

```
{# templates/_navbar.html #}

{% macro nav_link(label) %}
  {% set href = kwargs.get('href', url_for(kwargs['endpoint'])) %}
  <li class="nav-item {% if kwargs is active %}active{% endif %}">
    <a class="nav-link" href="{{ href }}">
      {{ label }}
      {% if kwargs is active %}
        <span class="sr-only">(current)</span>
      {% endif %}
    </a>
  </li>
{% endmacro %}

<nav class="navbar navbar-expand-md navbar-dark bg-dark">
  <a class="navbar-brand" href="{{ url_for('site_controller.index') }}">
    Hello Flask Unchained
  </a>
  <button type="button"
          class="navbar-toggler"
          data-toggle="collapse"
          data-target="#navbarCollapse"
          aria-controls="navbarCollapse"
          aria-expanded="false"
          aria-label="Toggle navigation"
  >
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarCollapse">
    <ul class="navbar-nav mr-auto">
      {{ nav_link('Home', endpoint='site_controller.index') }}
    </ul>
  </div>
</nav>
```

The `nav_link` macro perhaps deserves some explanation. This is a small utility function that renders a navigation

item in the bootstrap navbar. We do this to make our code more DRY, because every navigation link needs to contain logic to determine whether or not it is the currently active view. The `{% if endpoint is active %}` bit is special - Flask Unchained adds the `active` template test by default to make this easier.

And now let's update our `app/templates/site/index.html` template to use our new layout template:

```
{# app/templates/site/index.html #}

{% extends 'layout.html' %}

{% block title %}Hello World!{% endblock %}

{% block content %}
  <div class="row">
    <div class="col">
      <h1>Hello World!</h1>
    </div>
  </div>
{% endblock %}
```

Tests should still pass...

```
pytest
==================================== test session starts␣
↪=====================================
platform linux -- Python 3.6.6, pytest-3.6.4, py-1.5.4, pluggy-0.7.1
rootdir: /home/user/dev/hello-flask-unchained, inifile:
plugins: flask-0.10.0, Flask-Unchained-0.8.0
collected 1 item

tests/app/test_views.py .                                                  ␣
↪[100%]

================================= 1 passed in 0.10 seconds␣
↪==================================
```

This seems like a good place to make a commit:

```
git add .
git status
git commit -m 'refactor templates to extend a base layout template'
```

2.2.2 Customizing Styles

If you take a look at how our new template looks, it's pretty good, but the `h1` tag is now very close to the navbar. Let's fix that by adding some style customizations:

```
mkdir static/vendor \
   && mv static/*.min.* static/vendor \
   && touch static/main.css
```

Let's update the `stylesheets` and `javascripts` blocks in our layout template to reference the changed locations of the vendor assets, and our new `main.css` stylesheet:

```
{# templates/layout.html #}
```

(continues on next page)

(continued from previous page)

```
{% block stylesheets %}
  <link rel="stylesheet" href="{{ url_for('static', filename='vendor/bootstrap-v4.1.2.
↪min.css') }}">
  <link rel="stylesheet" href="{{ url_for('static', filename='main.css') }}">
{% endblock stylesheets %}

{% block javascripts %}
  <script src="{{ url_for('static', filename='vendor/jquery-v3.3.1.slim.min.js') }}">
↪</script>
  <script src="{{ url_for('static', filename='vendor/popper-v1.14.3.min.js') }}"></
↪script>
  <script src="{{ url_for('static', filename='vendor/bootstrap-v4.1.2.min.js') }}"></
↪script>
{% endblock javascripts %}
```

And of course, the custom rule for our `h1` tags:

```
/* static/main.css */

h1 {
  padding-top: 0.5em;
  margin-top: 0.5em;
}
```

Let's commit our changes:

```
git add .
git status
git commit -m 'add a custom stylesheet'
```

2.2.3 Adding a Landing Page

OK, let's refactor our views so we have a landing page and a separate page for the hello view. We're also going to introduce `flask_unchained.decorators.param_converter()` here so that we can (optionally) customizable the name we're saying hello to via the query string:

```
# app/views.py

from flask_unchained import Controller, route, param_converter


class SiteController(Controller):
    @route('/')
    def index(self):
        return self.render('index')

    @route('/hello')
    @param_converter(name=str)
    def hello(self, name=None):
        name = name or 'World'
        return self.render('hello', name=name)
```

The `param_converter` converts arguments passed in via the query string to arguments that get passed to the decorated view function. It can make sure you get the right type via a callable (like here), or as we'll cover later, it can even convert unique identifiers from the URL directly into database models. But that's getting ahead of ourselves.

Now that we've added another view/route, our templates need some work again. Let's update the navbar, move our existing `index.html` template to `hello.html` (adding support for the `name` template context variable), and lastly add a new `index.html` template for the landing page.

```
{# templates/_navbar.html #}

<ul class="navbar-nav mr-auto">
  {{ nav_link('Home', endpoint='site_controller.index') }}
  {{ nav_link('Hello', endpoint='site_controller.hello') }}  <!-- add this line -->
</ul>
```

```
{# app/templates/site/hello.html #}

{% extends 'layout.html' %}

{% block title %}Hello {{ name }}!{% endblock %}

{% block content %}
  <div class="row">
    <div class="col">
      <h1>Hello {{ name }}!</h1>
    </div>
  </div>
{% endblock %}
```

```
{# app/templates/site/index.html #}

{% extends 'layout.html' %}

{% block body %}
  <div class="jumbotron">
    <div class="container">
      <div class="row">
        <div class="col">
          <h1 class="display-3">Hello Flask Unchained!</h1>
        </div>
      </div>
    </div>
  </div>
{% endblock %}
```

We need to update our tests:

```
# tests/app/test_views.py

class TestSiteController:
    def test_index(self, client, templates):
        r = client.get('site_controller.index')
        assert r.status_code == 200
        assert templates[0].template.name == 'site/index.html'
        assert r.html.count('Hello Flask Unchained!') == 2

    def test_hello(self, client, templates):
        r = client.get('site_controller.hello')
        assert r.status_code == 200
        assert templates[0].template.name == 'site/hello.html'
        assert r.html.count('Hello World!') == 2
```

(continues on next page)

(continued from previous page)

```
    def test_hello_with_name_parameter(self, client, templates):
        r = client.get('site_controller.hello', name='User')
        assert r.status_code == 200
        assert templates[0].template.name == 'site/hello.html'
        assert r.html.count('Hello User!') == 2
```

A couple things to note here. Most obviously, we added another view, and therefore need to add methods to test it. Also of note is the `templates` pytest fixture, which we're using to verify the correct template got rendered for each of the views.

Let's make sure they pass:

```
pytest
================================== test session starts␣
↪===================================
platform linux -- Python 3.6.6, pytest-3.6.4, py-1.5.4, pluggy-0.7.1
rootdir: /home/user/dev/hello-flask-unchained, inifile:
plugins: flask-0.10.0, Flask-Unchained-0.8.0
collected 3 items

tests/app/test_views.py ...                                                  ␣
↪[100%]

================================= 3 passed in 0.17 seconds␣
↪==================================
```

Cool. You guessed it, time to make a commit!

```
git add .
git status
git commit -m 'add landing page, parameterize hello view to accept a name'
```

2.2.4 Adding a Form to the Hello View

We've parameterized our hello view take a `name` argument, however, it's not exactly discoverable by users (unless perhaps they're a developer with good variable naming intuition). One way to improve this is by using a form. First, we'll add a form the old-school way, followed by a refactor to use Flask-WTF form classes.

Let's update our hello template:

```
{# app/templates/site/hello.html #}

{% extends 'layout.html' %}

{% block title %}Hello {{ name }}!{% endblock %}

{% block content %}
  <div class="row">
    <div class="col">
      <h1>Hello {{ name }}!</h1>

      <h2>Enter your name:</h2>
      <form name="hello_form" action="{{ url_for('site_controller.hello') }}" method=
↪"POST">
```

(continues on next page)

(continued from previous page)

```
        {% if error %}
          <ul class="errors">
            <li class="error">{{ error }}</li>
          </ul>
        {% endif %}
        <div class="form-group">
          <label for="name">Name</label>
          <input type="text" id="name" name="name" class="form-control" />
        </div>
        <button type="submit" class="btn btn-primary">Submit</button>
      </form>
    </div>
  </div>
{% endblock %}
```

And the corresponding view code:

```
# app/views.py

# import request from flask_unchained
from flask_unchained import Controller, request, route, param_converter

class SiteController(Controller):
    # and update the code for our hello view
    @route('/hello', methods=['GET', 'POST'])
    @param_converter(name=str)
    def hello(self, name=None):
        if request.method == 'POST':
            name = request.form['name']
            if not name:
                return self.render('hello', error='Name is required.', name='World')
            return self.redirect('hello', name=name)
        return self.render('hello', name=name or 'World')
```

A wee styling update to also put some spacing above `h2` headers:

```
/* static/main.css */

h1, h2 {
  padding-top: 0.5em;
  margin-top: 0.5em;
}
```

And let's fix our tests:

```
# tests/app/test_views.py

# add this import
from flask_unchained import url_for

class TestSiteController:
    # and add this method
    def test_hello_with_form_post(self, client, templates):
        r = client.post('site_controller.hello', data=dict(name='User'))
        assert r.status_code == 302
        assert r.path == url_for('site_controller.hello')
```

(continues on next page)

(continued from previous page)

```
        r = client.follow_redirects(r)
        assert r.status_code == 200
        assert r.html.count('Hello User!') == 2

        # note: when request is a POST, the templates fixture only works after␣
↪redirecting
        assert templates[0].template.name == 'site/hello.html'
```

Make sure they pass,

```
pytest
================================== test session starts␣
↪===================================
platform linux -- Python 3.6.6, pytest-3.7.1, py-1.5.4, pluggy-0.7.1
rootdir: /home/user/dev/hello-flask-unchained, inifile:
plugins: flask-0.10.0, Flask-Unchained-0.8.0
collected 4 items

tests/app/test_views.py ....                                                ␣
↪[100%]

================================= 4 passed in 0.16 seconds␣
↪==================================
```

And commit our changes once satisfied:

```
git add .
git status
git commit -m 'add a form to the hello view'
```

2.2.5 Converting to a Flask-WTF Form

The above method works, as far as it goes, but both our view code and our template code are very verbose, and the form verification/error handling is awfully manual. Luckily the Flask ecosystem has a solution to this problem, in the awesomely named `Flask-WTF` package (it's installed by default as a dependency of Flask Unchained). With it, our new form looks like this:

```
touch app/forms.py
```

```
# app/forms.py

from flask_unchained.forms import FlaskForm, fields, validators


class HelloForm(FlaskForm):
    name = fields.StringField('Name', validators=[
        validators.DataRequired('Name is required.')])
    submit = fields.SubmitField('Submit')
```

The updated view code:

```
# app/views.py

from flask_unchained import Controller, request, route, param_converter
```

(continues on next page)

(continued from previous page)

```
from .forms import HelloForm


class SiteController(Controller):
    @route('/')
    def index(self):
        return self.render('index')

    @route('/hello', methods=['GET', 'POST'])
    @param_converter(name=str)
    def hello(self, name=None):
        form = HelloForm(request.form)
        if form.validate_on_submit():
            return self.redirect('hello', name=form.name.data)
        return self.render('hello', hello_form=form, name=name or 'World')
```

And the updated template:

```
{# app/templates/site/hello.html #}

{% extends 'layout.html' %}

{% from '_macros.html' import render_form %}

{% block title %}Hello {{ name }}!{% endblock %}

{% block content %}
  <div class="row">
    <div class="col">
      <h1>Hello {{ name }}!</h1>

      <h2>Enter your name:</h2>
      {{ render_form(hello_form, endpoint='site_controller.hello') }}
    </div>
  </div>
{% endblock %}
```

What is this mythical `render_form` macro? Well, we need to write it ourselves. But luckily once it's written, it should work on the majority of `FlaskForm` subclasses. Here's the code for it:

```
touch templates/_macros.html
```

```
{% macro render_form(form) %}
  {% set action = kwargs.get('action', url_for(kwargs['endpoint'])) %}
  <form name="{{ form._name }}" {% if action %}action="{{ action }}"{% endif %}␣
↪method="POST">
    {{ render_errors(form.errors.get('_error', [])) }}
    {% for field in form %}
      {{ render_field(field) }}
    {% endfor %}
  </form>
{% endmacro %}

{% macro render_field(field) %}
  {% set input_type = field.widget.input_type %}
```

(continues on next page)

(continued from previous page)

```
  {% if input_type == 'hidden' %}
    {{ field(**kwargs)|safe }}
  {% elif input_type == 'submit' %}
    <div class="form-group">
      {{ field(class='btn btn-primary', **kwargs)|safe }}
    </div>
  {% else %}
    <div class="form-group">
      {% if input_type == 'checkbox' %}
        <label for="{{ field.id }}">
          {{ field(**kwargs)|safe }} {{ field.label.text }}
        </label>
      {% else %}
        {{ field.label }}
        {{ field(class='form-control', **kwargs)|safe }}
      {% endif %}

      {# always render description and/or errors if they are present #}
      {% if field.description %}
        <small class="form-text text-muted form-field-description">
          {{ field.description }}
        </small>
      {% endif %}
      {{ render_errors(field.errors) }}
    </div> {# /.form-group #}
  {% endif %}
{% endmacro %}

{% macro render_errors(errors) %}
  {% if errors %}
    <ul class="errors">
    {% for error in errors %}
      <li class="error">{{ error }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endmacro %}
```

More complicated forms, for example those with multiple submit buttons or multiple pages, that require more manual control over the presentation can use `render_field` directly for each field in the form.

As usual, let's update our tests and make sure they pass:

```
# tests/app/test_views.py

class TestSiteController:
    # add this method
    def test_hello_errors_with_empty_form_post(self, client, templates):
        r = client.post('site_controller.hello')
        assert r.status_code == 200
        assert templates[0].template.name == 'site/hello.html'
        assert r.html.count('Name is required.') == 1
```

```
pytest
================================== test session starts␣
↪===================================
```

(continues on next page)

(continued from previous page)

```
platform linux -- Python 3.6.6, pytest-3.7.1, py-1.5.4, pluggy-0.7.1
rootdir: /home/user/dev/hello-flask-unchained, inifile:
plugins: flask-0.10.0, Flask-Unchained-0.8.0
collected 5 items

tests/app/test_views.py .....                                            ␣
↪[100%]

================================= 5 passed in 0.19 seconds ␣
↪=================================
```

Once your tests are passing, it's time to make commit:

```
git add .
git status
git commit -m 'refactor hello form to use flask-wtf'
```

2.2.6 Enabling CSRF Protection

By default, CSRF protection is disabled. However, any time you're using forms or have enabled authentication (covered later), you should also enable CSRF protection. There are two requirements:

The first is to update our configuration:

```
touch app/config.py
```

```
# app/config.py

from flask_unchained import BundleConfig

class Config(BundleConfig):
    SECRET_KEY = 'some-secret-key'
    WTF_CSRF_ENABLED = True

class TestConfig(Config):
    WTF_CSRF_ENABLED = False
```

And secondly, we need to actually send the CSRF token in the cookie with every response:

```
# app/__init__.py

from flask_unchained import AppBundle, generate_csrf

class App(AppBundle):
    def after_init_app(self, app) -> None:
        @app.after_request
        def set_csrf_token_cookie(response):
            if response:
                response.set_cookie('csrf_token', generate_csrf())
            return response
```

Tests should still pass, so it's time to make commit:

```
git add .
git status
git commit -m 'enable CSRF protection'
```

Cool. Let's move on to *Setting up the Database* in preparation for installing the Security Bundle.

2.3 Setting up the Database

Flask Unchained comes integrated with the SQLAlchemy ORM.

2.3.1 Install Dependencies

First we need to install SQLAlchemy and related dependencies:

```
pip install "flask-unchained[sqlalchemy]" py-yaml-fixtures
```

We also need to update our tests so that they load the pytest fixtures from the SQLAlchemy Bundle:

```
touch tests/conftest.py
```

```
# tests/conftest.py

from flask_unchained.bundles.sqlalchemy.pytest import *
```

There are two fixtures that it includes: `db` and `db_session`. They're both automatically used; `db` is scoped `session` and will create/drop all necessary tables once per test session while `db_session` is function scoped, and thus will run for every test. Its responsibility is to create an isolated session of transactions for each individual test, to make sure that every test starts with a clean slate database without needing to drop and recreate all the tables for each and every test.

Next, enable the SQLAlchemy and Py YAML Fixtures bundles so we can begin using them:

```
# unchained_config.py

BUNDLES = [
    # ...
    'flask_unchained.bundles.sqlalchemy',
    'py_yaml_fixtures',
    'app',
]
```

2.3.2 Configuration

By default, the SQLAlchemy Bundle is configured to use an SQLite database. For the sake of simplicity, we'll leave the defaults as-is. The SQLite file will be stored at `db/<env>.sqlite`.

If you'd like to change the path, it would look like this:

```
# app/config.py

from flask_unchained import BundleConfig
```

(continues on next page)

(continued from previous page)

```
class Config(BundleConfig):
    # ...
    SQLALCHEMY_DATABASE_URI = 'sqlite:///db/hello-flask-unchained.sqlite'
```

If you're fine using SQLite, continue to *Initialize Migrations*.

If instead you'd like to use MariaDB/MySQL or PostgreSQL, now would be the time to configure it. For example, to use PostgreSQL with `psycopg2`:

```
# app/config.py

from flask_unchained import BundleConfig

class Config(BundleConfig):
    # ...
    SQLALCHEMY_DATABASE_URI = '{engine}://{user}:{pw}@{host}:{port}/{db}'.format(
        engine=os.getenv('FLASK_DATABASE_ENGINE', 'postgresql+psycopg2'),
        user=os.getenv('FLASK_DATABASE_USER', 'hello_fun'),
        pw=os.getenv('FLASK_DATABASE_PASSWORD', 'hello_fun'),
        host=os.getenv('FLASK_DATABASE_HOST', '127.0.0.1'),
        port=os.getenv('FLASK_DATABASE_PORT', 5432),
        db=os.getenv('FLASK_DATABASE_NAME', 'hello_fun'))

class TestConfig:
    # ...
    SQLALCHEMY_DATABASE_URI = '{engine}://{user}:{pw}@{host}:{port}/{db}'.format(
        engine=os.getenv('FLASK_DATABASE_ENGINE', 'postgresql+psycopg2'),
        user=os.getenv('FLASK_DATABASE_USER', 'hello_fun_test'),
        pw=os.getenv('FLASK_DATABASE_PASSWORD', 'hello_fun_test'),
        host=os.getenv('FLASK_DATABASE_HOST', '127.0.0.1'),
        port=os.getenv('FLASK_DATABASE_PORT', 5432),
        db=os.getenv('FLASK_DATABASE_NAME', 'hello_fun_test'))
```

Or for MariaDB/MySQL, replace the `engine` parameter with `mysql+mysqldb` and the `port` parameter with `3306`.

Note that you'll probably need to install the relevant driver package, eg:

```
# for psycopg2
pip install psycopg2-binary

# for mysql
pip install mysqlclient
```

See the upstream docs on SQLAlchemy dialects for details.

2.3.3 Initialize Migrations

The last step is to initialize the database migrations folder:

```
flask db init
```

We should commit our changes before continuing:

```
git add .
git status
git commit -m 'install sqlalchemy and py-yaml-fixtures bundles'
```

Next, in order to demonstrate using migrations, and also as preparation for installing the Security Bundle, let's continue to setting up *Server Side Sessions* using the Session Bundle.

2.4 Server Side Sessions

By default, Flask will store user session information in a client-side cookie. This works, however, it has some drawbacks. For instance, it doesn't allow sessions to be terminated by the server, implementation and user details get exposed in the cookie, and logout isn't fully implemented (Flask asks the browser to delete the cookie, and it will, but if the cookie had been saved/intercepted and later resubmitted, eg by a nefarious party, the cookie would continue to work even though the user thought they logged out). Using server side sessions solves these problems for us, and it's really easy to set up thanks to Flask-Session.

2.4.1 Install Dependencies

```
pip install "flask-unchained[session]"
```

Enable the Session Bundle:

```
# unchained_config.py

BUNDLES = [
    # ...
    'flask_unchained.bundles.session',
    'app',
]
```

2.4.2 Configuration

Let's configure the Session Bundle to use SQLAlchemy:

```
# app/config.py

class Config(BundleConfig):
    # ...
    SESSION_TYPE = 'sqlalchemy'
    SESSION_SQLALCHEMY_TABLE = 'flask_sessions'
```

Because the Session Bundle is just a thin wrapper around Flask Session, configuration options are the same as documented in the official docs.

2.4.3 Database Migrations

And now we need to create the table, using a database migration:

```
flask db migrate -m 'create sessions table'
```

It's always a good idea to inspect the migration to make sure it's going to do exactly what you expect it to do before running it. It should look something about like this:

```
# db/migrations/versions/[hash]_create_sessions_table.py

"""create sessions table

Revision ID: 17c5247038a6
Revises:
Create Date: 2018-07-30 11:50:19.709960

"""
from alembic import op
import sqlalchemy as sa
import flask_unchained.bundles.sqlalchemy.sqla.types as sqla_bundle

# revision identifiers, used by Alembic.
revision = '17c5247038a6'
down_revision = None
branch_labels = None
depends_on = None

def upgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.create_table('flask_sessions',
        sa.Column('id', sa.Integer(), nullable=False),
        sa.Column('session_id', sa.String(length=255), nullable=False),
        sa.Column('data', sa.LargeBinary(), nullable=False),
        sa.Column('expiry', sa.DateTime(), nullable=True),
        sa.Column('created_at', sqla_bundle.DateTime(timezone=True),
                  server_default=sa.text('CURRENT_TIMESTAMP'), nullable=False),
        sa.Column('updated_at', sqla_bundle.DateTime(timezone=True),
                  server_default=sa.text('CURRENT_TIMESTAMP'), nullable=False),
        sa.PrimaryKeyConstraint('id', name=op.f('pk_flask_sessions')),
        sa.UniqueConstraint('session_id', name=op.f('uq_flask_sessions_session_id'))
    )
    # ### end Alembic commands ###

def downgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.drop_table('flask_sessions')
    # ### end Alembic commands ###
```

Once you're satisfied, run the migration:

```
flask db upgrade
```

That's it for setting up server side sessions! Let's make a commit before we continue:

```
git add .
git status
git commit -m 'install session bundle'
```

And proceed to *Authentication and Authorization* using the Security Bundle.

2.5 Authentication and Authorization

Flask Unchained currently has one officially supported bundle for securing your app. It's a refactored and cleaned up fork of the Flask Security project, and includes support for session and token authentication. (All of the core security logic remains unchanged.) Adding support for JWT authentication is on the roadmap, but isn't implemented yet.

2.5.1 Install Security Bundle

```
pip install "flask-unchained[security]"
```

Let's update our test fixtures configuration file to include the fixtures provided by the Security Bundle:

```
# tests/conftest.py

from flask_unchained.bundles.sqlalchemy.pytest import *
from flask_unchained.bundles.security.pytest import *  # add this line
```

The security bundle overrides the `client` and `api_client` test fixtures to add support for logging in and logging out.

Now we can enable the Security Bundle by adding it to `unchained_config.py`:

```
# unchained_config.py

BUNDLES = [
    # ...
    'flask_unchained.bundles.security',
    'app',
]
```

2.5.2 Database Models and Migrations

Let's start with configuring our database models, because the views will be broken until we implement our database models and create tables in the database for them. The security bundle includes default model implementations that, for now, will be sufficient for our needs:

```
# flask_unchained/bundles/security/models/user.py

from flask_unchained.bundles.sqlalchemy import db
from flask_unchained import unchained, injectable, lazy_gettext as _
from flask_unchained.bundles.security.models.user_role import UserRole
from flask_unchained.bundles.security.validators import EmailValidator

MIN_PASSWORD_LENGTH = 8


class User(db.Model):
    """
    Base :class:`User` model. Includes :attr:`email`, :attr:`password`, :attr:`is_
↪active`,
```

(continues on next page)

(continued from previous page)

```
    and :attr:`confirmed_at` columns, and a many-to-many relationship to the
    :class:`Role` model via the intermediary :class:`UserRole` join table.
    """
    class Meta:
        lazy_mapped = True
        repr = ('id', 'email', 'is_active')

    email = db.Column(db.String(64), unique=True, index=True, info=dict(
        required=_('flask_unchained.bundles.security:email_required'),
        validators=[EmailValidator]))
    _password = db.Column('password', db.String, info=dict(
        required=_('flask_unchained.bundles.security:password_required')))
    is_active = db.Column(db.Boolean(name='is_active'), default=False)
    confirmed_at = db.Column(db.DateTime(), nullable=True)

    user_roles = db.relationship('UserRole', back_populates='user',
                                 cascade='all, delete-orphan')
    roles = db.association_proxy('user_roles', 'role',
                                 creator=lambda role: UserRole(role=role))

    @db.hybrid_property
    def password(self):
        return self._password

    @password.setter
    @unchained.inject('security_utils_service')
    def password(self, password, security_utils_service=injectable):
        self._password = security_utils_service.hash_password(password)

    @classmethod
    def validate_password(cls, password):
        if password and len(password) < MIN_PASSWORD_LENGTH:
            raise db.ValidationError(f'Password must be at least '
                                     f'{MIN_PASSWORD_LENGTH} characters long.')

    @unchained.inject('security_utils_service')
    def get_auth_token(self, security_utils_service=injectable):
        """
        Returns the user's authentication token.
        """
        return security_utils_service.get_auth_token(self)

    def has_role(self, role):
        """
        Returns `True` if the user identifies with the specified role.

        :param role: A role name or :class:`Role` instance
        """
        if isinstance(role, str):
            return role in (role.name for role in self.roles)
        else:
            return role in self.roles

    @property
    def is_authenticated(self):
        return True
```

(continues on next page)

(continued from previous page)

```
    @property
    def is_anonymous(self):
        return False
```

```
# flask_unchained/bundles/security/models/role.py

from flask_unchained.bundles.sqlalchemy import db
from flask_unchained.bundles.security.models.user_role import UserRole


class Role(db.Model):
    """
    Base :class`Role` model. Includes an :attr:`name` column and a many-to-many
    relationship with the :class:`User` model via the intermediary :class:`UserRole`
    join table.
    """
    class Meta:
        lazy_mapped = True
        repr = ('id', 'name')

    name = db.Column(db.String(64), unique=True, index=True)

    role_users = db.relationship('UserRole', back_populates='role',
                                 cascade='all, delete-orphan')
    users = db.association_proxy('role_users', 'user',
                                 creator=lambda user: UserRole(user=user))

    def __hash__(self):
        return hash(self.name)
```

```
# flask_unchained/bundles/security/models/user_role.py

from flask_unchained.bundles.sqlalchemy import db


class UserRole(db.Model):
    """
    Join table between the :class:`User` and :class:`Role` models.
    """
    class Meta:
        lazy_mapped = True
        pk = None
        repr = ('user_id', 'role_id')

    user_id = db.foreign_key('User', primary_key=True)
    user = db.relationship('User', back_populates='user_roles')

    role_id = db.foreign_key('Role', primary_key=True)
    role = db.relationship('Role', back_populates='role_users')

    def __init__(self, user=None, role=None, **kwargs):
        super().__init__(**kwargs)
        if user:
            self.user = user
        if role:
            self.role = role
```

We're going to leave them as-is for now, but in preparation for later customizations, let's subclass `User` and `Role` in our app bundle:

```
touch app/models.py
```

```
# app/models.py

from flask_unchained.bundles.security import User as BaseUser, Role as BaseRole,␣
↪UserRole


class User(BaseUser):
    pass


class Role(BaseRole):
    pass
```

Time to generate some migrations:

```
flask db migrate -m 'add security bundle models'
```

And review them to make sure it's going to do what we want:

```
# db/migrations/versions/[hash]_add_security_bundle_models.py

"""add security bundle models

Revision ID: 839865db0b53
Revises: eb0448e9a537
Create Date: 2018-08-07 16:55:40.180962

"""
from alembic import op
import sqlalchemy as sa
import flask_unchained.bundles.sqlalchemy.sqla.types as sqla_bundle

# revision identifiers, used by Alembic.
revision = '839865db0b53'
down_revision = 'eb0448e9a537'
branch_labels = None
depends_on = None


def upgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.create_table('role',
        sa.Column('name', sa.String(length=64), nullable=False),
        sa.Column('id', sqla_bundle.BigInteger(), nullable=False),
        sa.Column('created_at', sqla_bundle.DateTime(timezone=True),
                  server_default=sa.text('CURRENT_TIMESTAMP'), nullable=False),
        sa.Column('updated_at', sqla_bundle.DateTime(timezone=True),
                  server_default=sa.text('CURRENT_TIMESTAMP'), nullable=False),
        sa.PrimaryKeyConstraint('id', name=op.f('pk_role'))
    )
    op.create_index(op.f('ix_role_name'), 'role', ['name'], unique=True)

    op.create_table('user',
```

(continues on next page)

(continued from previous page)

```
        sa.Column('email', sa.String(length=64), nullable=False),
        sa.Column('password', sa.String(), nullable=False),
        sa.Column('is_active', sa.Boolean(name='is_active'), nullable=False),
        sa.Column('confirmed_at', sqla_bundle.DateTime(timezone=True), nullable=True),
        sa.Column('id', sqla_bundle.BigInteger(), nullable=False),
        sa.Column('created_at', sqla_bundle.DateTime(timezone=True),
                  server_default=sa.text('CURRENT_TIMESTAMP'), nullable=False),
        sa.Column('updated_at', sqla_bundle.DateTime(timezone=True),
                  server_default=sa.text('CURRENT_TIMESTAMP'), nullable=False),
        sa.PrimaryKeyConstraint('id', name=op.f('pk_user'))
    )
    op.create_index(op.f('ix_user_email'), 'user', ['email'], unique=True)

    op.create_table('user_role',
        sa.Column('user_id', sqla_bundle.BigInteger(), nullable=False),
        sa.Column('role_id', sqla_bundle.BigInteger(), nullable=False),
        sa.Column('created_at', sqla_bundle.DateTime(timezone=True),
                  server_default=sa.text('CURRENT_TIMESTAMP'), nullable=False),
        sa.Column('updated_at', sqla_bundle.DateTime(timezone=True),
                  server_default=sa.text('CURRENT_TIMESTAMP'), nullable=False),
        sa.ForeignKeyConstraint(['role_id'], ['role.id'], name=op.f(
            'fk_user_role_role_id_role')),
        sa.ForeignKeyConstraint(['user_id'], ['user.id'], name=op.f(
            'fk_user_role_user_id_user')),
        sa.PrimaryKeyConstraint('user_id', 'role_id', name=op.f('pk_user_role'))
    )
    # ### end Alembic commands ###


def downgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.drop_table('user_role')
    op.drop_index(op.f('ix_user_email'), table_name='user')
    op.drop_table('user')
    op.drop_index(op.f('ix_role_name'), table_name='role')
    op.drop_table('role')
    # ### end Alembic commands ###
```

Looks good.

```
flask db upgrade
```

2.5.3 Seeding the Database

There is of course the manual method of creating users, either via the command line interface using `flask users create`, or via the register endpoint (which we'll set up just after this). But the problem with those methods is that they're not reproducible. Database fixtures are one common solution to this problem, and the SQLAlchemy Bundle includes support for them.

First we need to create our fixtures directory and files. The file names must match the class name of the model each fixture corresponds to (`Role` and `User` in our case):

```
mkdir db/fixtures && touch db/fixtures/Role.yaml db/fixtures/User.yaml
```

```yaml
# db/fixtures/Role.yaml

ROLE_USER:
  name: ROLE_USER

ROLE_ADMIN:
  name: ROLE_ADMIN
```

```yaml
# db/fixtures/User.yaml

admin:
  email: your_email@somewhere.com
  password: 'a secure password'
  is_active: True
  confirmed_at: utcnow
  roles: ['Role(ROLE_ADMIN, ROLE_USER)']

user:
  email: user@flaskr.com
  password: password
  is_active: True
  confirmed_at: utcnow
  roles: ['Role(ROLE_USER)']
```

The keys in the yaml files, `admin`, `user`, `ROLE_USER` and `ROLE_ADMIN`, must each be unique across all of your fixtures. This is because they are used to specify relationships. The syntax there is `'ModelClassName(key1, Optional[key2, ...])'`. If the relationship is on the many side, as it is in our case, then the relationship specifier must also be surrounded by `[]` square brackets (yaml syntax to specify it's a list).

It's not shown above, but the fixture files are actually *Jinja2 templates that generate yaml*. Fixtures also have access to the excellent faker library to generate random data, for example we could have written `email: {{ faker.free_email() }}` in the `user` fixture. Between access to faker and the power of Jinja2, it's quite easy to build up a bunch of fake content when you need to quickly.

Running the fixtures should create two users and two roles in our dev db:

```
flask db import-fixtures
Loading fixtures from `db/fixtures` directory
Created ROLE_USER: Role(id=1, name='ROLE_USER')
Created ROLE_ADMIN: Role(id=2, name='ROLE_ADMIN')
Created admin: User(id=1, email='your_email@somewhere.com', is_active=True)
Created user: User(id=2, email='user@flaskr.com', is_active=True)
Finished adding fixtures
```

Sweet. Let's set up our views so we can actually login to our site!

2.5.4 Configuring and Customizing Views

The first thing we need to do is to include the `SecurityController` in our `routes.py`:

```python
# app/routes.py

from flask_unchained import (controller, resource, func, include, prefix,
                             get, delete, post, patch, put, rule)

from flask_unchained.bundles.security import SecurityController
```

(continues on next page)

(continued from previous page)

```
from .views import SiteController


routes = lambda: [
    controller(SiteController),
    controller(SecurityController),
]
```

By default, Security Bundle only comes with the login and logout endpoints enabled. Let's confirm:

```
flask urls
Method(s)  Rule                                Endpoint
↪  View
↪             Options
-------------------------------------------------------------------------------
↪-------------------------------------------------------------------------------
↪----------------------------
      GET  /static/<path:filename>          static
↪  flask.helpers :: send_static_file
↪             strict_slashes
      GET  /                                site_controller.index
↪  app.views :: SiteController.index
↪             strict_slashes
      GET  /hello                           site_controller.hello
↪  app.views :: SiteController.hello
↪             strict_slashes
GET, POST  /login                           security_controller.login
↪  flask_unchained.bundles.security.views.security_controller :: SecurityController.
↪login                    strict_slashes
      GET  /logout                          security_controller.logout
↪  flask_unchained.bundles.security.views.security_controller :: SecurityController.
↪logout                   strict_slashes
```

The security bundle comes with optional support for registration, required email confirmation, change password functionality, and last but not least, forgot password functionality. For now, let's just enable registration:

```
# app/config.py

from flask_unchained import BundleConfig

class Config(BundleConfig):
    # ...
    SECURITY_REGISTERABLE = True
```

Rerunning `flask urls`, you should see the following line added:

```
Method(s)  Rule                                Endpoint
↪  View
↪             Options
-------------------------------------------------------------------------------
↪-------------------------------------------------------------------------------
↪----------------------------
GET, POST  /register                        security_controller.register
↪  flask_unchained.bundles.security.views.security_controller :: SecurityController.
↪register                 strict_slashes
```

Let's add these routes to our navbar:

```
{# templates/_navbar.html #}

<div class="collapse navbar-collapse" id="navbarCollapse">
  <ul class="navbar-nav mr-auto">
    {{ nav_link('Home', endpoint='site_controller.index') }}
    {{ nav_link('Hello', endpoint='site_controller.hello') }}
  </ul>
  <ul class="navbar-nav">
    {% if not current_user.is_authenticated %}
      {{ nav_link('Login', endpoint='security_controller.login') }}
      {{ nav_link('Register', endpoint='security_controller.register') }}
    {% else %}
      {{ nav_link('Logout', endpoint='security_controller.logout') }}
    {% endif %}
  </ul>
</div>
```

Cool. You should now be able to login with the credentials you created in the `User.yaml` fixture. If you take a look at the login and/or register views, however, you'll notice that things aren't rendering "the bootstrap way." Luckily all the default templates in the security bundle extend the `security/layout.html` template, so we can override just this template to fix integrating the layout of all security views into our site.

We're going to completely override the layout template. In order to make sure the layout works correctly, we need to wrap the content block with a row and a column. Therefore, our version looks like this:

```
mkdir -p app/templates/security \
   && touch app/templates/security/layout.html \
   && touch app/templates/security/_macros.html
```

```
{# app/templates/security/layout.html #}

{% extends 'layout.html' %}

{% block body %}
  <div class="container">
    {% include '_flashes.html' %}
    <div class="row">
      <div class="col">
        {% block content %}
        {% endblock content %}
      </div>
    </div>
  </div>
{% endblock body %}
```

But even after this change, our forms are still using the browser's default form styling. Once again, the security bundle makes it easy to fix this, by overriding the `render_form` macro in the `security/_macros.html` template. You'll note we've already written this macro, so all we need to do is the following:

```
{# app/templates/security/_macros.html #}

{% from '_macros.html' import render_form as _render_form %}

{# the above is *only* an import, and Jinja doesn't re-export it, so we #}
{# work around that by proxying to the original macro under the same name #}
```

(continues on next page)

(continued from previous page)

```
{% macro render_form(form) %}
  {{ _render_form(form, **kwargs) }}
{% endmacro %}
```

2.5.5 Testing the Security Views

Unlike all of our earlier tests, testing the security bundle views requires that we have valid users in the database. Perhaps the most powerful way to accomplish this is by using Factory Boy, which Flask Unchained comes integrated with out of the box. If you aren't familiar with Factory Boy, I recommend you read more about how it works in the official docs. The short version is, it makes it incredibly easy to dynamically create and customize models on-the-fly.

```
pip install factory_boy
```

```
# tests/conftest.py

import pytest

from flask_unchained.bundles.sqlalchemy.pytest import *
from flask_unchained.bundles.security.pytest import *

from datetime import datetime, timezone
from app.models import User, Role, UserRole


class UserFactory(ModelFactory):
    class Meta:
        model = User

    email = 'user@example.com'
    password = 'password'
    is_active = True
    confirmed_at = datetime.now(timezone.utc)


class RoleFactory(ModelFactory):
    class Meta:
        model = Role

    name = 'ROLE_USER'


class UserRoleFactory(ModelFactory):
    class Meta:
        model = UserRole

    user = factory.SubFactory(UserFactory)
    role = factory.SubFactory(RoleFactory)


class UserWithRoleFactory(UserFactory):
    user_role = factory.RelatedFactory(UserRoleFactory, 'user')


@pytest.fixture()
```

(continues on next page)

(continued from previous page)

```
def user(request):
    kwargs = getattr(request.node.get_closest_marker('user'), 'kwargs', {})
    return UserWithRoleFactory(**kwargs)


@pytest.fixture()
def role(request):
    kwargs = getattr(request.node.get_closest_marker('role'), 'kwargs', {})
    return RoleFactory(**kwargs)
```

The `ModelFactory` subclasses define the default values, and the `user` and `role` fixtures at the bottom make it possible to customize the values by marking the test, for example:

```
@pytest.mark.user(email='foo@bar.com')
def test_something(user):
    assert user.email == 'foo@bar.com'
```

And our tests look like this:

```
# tests/app/test_security_controller.py

import pytest

from flask_unchained.bundles.security import AnonymousUser, current_user
from flask_unchained import url_for


class TestSecurityController:
    def test_login_get(self, client, templates):
        r = client.get('security_controller.login')
        assert r.status_code == 200
        assert templates[0].template.name == 'security/login.html'

    @pytest.mark.user(password='password')
    def test_login_post(self, client, user, templates):
        r = client.post('security_controller.login', data=dict(
            email=user.email,
            password='password'))

        assert r.status_code == 302
        assert r.path == url_for('site_controller.index')
        assert current_user == user

        r = client.follow_redirects(r)
        assert r.status_code == 200
        assert templates[0].template.name == 'site/index.html'

    def test_logout(self, client, user):
        client.login_user()
        assert current_user == user

        r = client.get('security_controller.logout')
        assert r.status_code == 302
        assert r.path == url_for('site_controller.index')
        assert isinstance(current_user._get_current_object(), AnonymousUser)
```

(continues on next page)

(continued from previous page)

```
    def test_register_get(self, client, templates):
        r = client.get('security_controller.register')
        assert r.status_code == 200
        assert templates[0].template.name == 'security/register.html'

    def test_register_post_errors(self, client, templates):
        r = client.post('security_controller.register')
        assert r.status_code == 200
        assert templates[0].template.name == 'security/register.html'
        assert 'Email is required.' in r.html
        assert 'Password is required.' in r.html

    def test_register_post(self, client, registrations, user_manager):
        r = client.post('security_controller.register', data=dict(
            email='a@a.com',
            password='password',
            password_confirm='password'))
        assert r.status_code == 302
        assert r.path == url_for('site_controller.index')

        assert len(registrations) == 1
        user = user_manager.get_by(email='a@a.com')
        assert registrations[0]['user'] == user
```

Running them should pass:

```
pytest --maxfail=1
================================== test session starts␣
↪===================================
platform linux -- Python 3.6.6, pytest-3.7.1, py-1.5.4, pluggy-0.7.1
rootdir: /home/user/dev/hello-flask-unchained, inifile:
plugins: flask-0.10.0, Flask-Unchained-0.8.0, Flask-Security-Bundle-0.3.0
collected 11 items

tests/app/test_views.py .....                                          [ 45%]
tests/security/test_security_controller.py ......                                ␣
↪[100%]

============================== 11 passed in 0.74 seconds␣
↪===============================
```

You can learn more about how to use all of the features the security bundle supports in its documentation.

Let's commit our changes:

```
git add .
git status
git commit -m 'install and configure security bundle'
```

CHAPTER

THREE

HOW FLASK UNCHAINED WORKS

3.1 The Application Factory

The app factory might sound like magic, but it's actually quite easy to understand, and every step it takes is customizable by you, if necessary. The just-barely pseudo code looks like this:

```
class AppFactory:
    def create_app(self, env):
        # first load the user's unchained config
        unchained_config = self.load_unchained_config(env)

        # next load configured bundles
        app_bundle, bundles = self.load_bundles(unchained_config.BUNDLES)

        # instantiate the Flask app instance
        app = Flask(app_bundle.name, **kwargs_from_unchained_config)

        # let bundles configure the app pre-initialization
        for bundle in bundles:
            bundle.before_init_app(app)

        # discover code from bundles and boot up the Flask app using hooks
        unchained.init_app(app, bundles)
            # the Unchained extension runs hooks in their correct order:
            RegisterExtensionsHook.run_hook(app, bundles)
            ConfigureAppHook.run_hook(app, bundles)
            InitExtensionsHook.run_hook(app, bundles)
            RegisterServicesHook.run_hook(app, bundles)
            RegisterCommandsHook.run_hook(app, bundles)
            RegisterRoutesHook.run_hook(app, bundles)
            RegisterBundleBlueprintsHook.run_hook(app, bundles)
            # (there may be more depending on which bundles you enable)

        # let bundles configure the app post-initialization
        for bundle in bundles:
            bundle.after_init_app(app)

        # return the finalized app ready to rock'n'roll
        return app
```

Advanced

You can subclass *flask_unchained.AppFactory* if you need to customize any of its behavior. If you want to take things a step further, Flask Unchained can even be used to create your own redistributable batteries-included web

framework for Flask using whichever stack of Flask extensions and Python libraries you prefer.

3.2 The Unchained Config

The first thing the app factory does is to load your "Unchained Config". The unchained config is used to declare which bundles should be loaded, as well as for passing any optional keyword arguments to the `Flask` constructor.

3.2.1 Dedicated Module Mode

The most common way to configure Flask Unchained apps is with a module named `unchained_config` in the project root:

```
/home/user/project-root
├── unchained_config.py     # your Unchained Config
└── app.py                  # your App Bundle
```

The Unchained Config itself doesn't actually contain a whole lot. The only required setting is the `BUNDLES` list:

```
# project-root/unchained_config.py

import os

# kwargs to pass to the Flask constructor (must be uppercase, all optional)
ROOT_PATH = os.path.dirname(__file__)   # determined automatically by default
STATIC_FOLDER = "static"                # determined automatically by default
STATIC_URL_PATH = "/static"             # determined automatically by default
STATIC_HOST = None                      # None by default
TEMPLATE_FOLDER = "templates"           # determined automatically by default
HOST_MATCHING = False                   # False by default
SUBDOMAIN_MATCHING = False              # False by default

# the ordered list of bundles to load for your app (in dot-module notation)
BUNDLES = [
    'flask_unchained.bundles.babel',        # always enabled, optional to list here
    'flask_unchained.bundles.controller',   # always enabled, optional to list here
    'app',  # your app bundle *must* be last
]
```

When `unchained_config.py` exists in the `project-root` directory, exporting `UNCHAINED` is not required, and the app can be run like so:

```
cd project-root
flask run
```

3.2.2 Single-File App Mode

For single-file apps, you can "double purpose" the `app` module as your Unchained Config. This is as simple as it gets:

```
# project-root/app.py

from flask_unchained import AppBundle, Controller, route

class App(AppBundle):
    pass

class SiteController(Controller):
    @route('/')
    def index(self):
        return 'hello world'
```

It can be run like so:

```
export UNCHAINED="app"  # the module where the unchained config resides
flask run
```

In the above example, we're essentially telling the app factory, "just use the defaults with my app bundle". In single-file mode, the app bundle is automatically detected, so there aren't actually any Unchained Config settings in the above file. To set them looks as you would expect:

```
# project-root/app.py

import os
from flask_unchained import AppBundle, Controller, route

# kwargs to pass to the Flask constructor (must be uppercase, all optional)
ROOT_PATH = os.path.dirname(__file__)  # determined automatically by default
STATIC_FOLDER = "static"               # determined automatically by default
STATIC_URL_PATH = "/static"            # determined automatically by default
STATIC_HOST = None                     # None by default
TEMPLATE_FOLDER = "templates"          # determined automatically by default
HOST_MATCHING = False                  # False by default
SUBDOMAIN_MATCHING = False             # False by default

# the ordered list of bundles to load for your app (in dot-module notation)
BUNDLES = [
    'flask_unchained.bundles.babel',       # always enabled, optional to list here
    'flask_unchained.bundles.controller',  # always enabled, optional to list here
    'app',  # your app bundle *must* be last
]

class App(AppBundle):
    pass

class SiteController(Controller):
    @route('/')
    def index(self):
        return 'hello world'
```

And once again, just be sure to `export UNCHAINED="app"`:

```
export UNCHAINED="app"  # the module where the unchained config resides
flask run
```

3.3 Bundle.before/after_init_app

The most obvious place you can hook into the app factory is with your *Bundle* subclass, for example:

```
# project-root/app.py

from flask import Flask
from flask_unchained import AppBundle

class App(AppBundle):
    def before_init_app(app: Flask):
        app.url_map.strict_slashes = False

    def after_init_app(app: Flask):
        @app.after_request
        def do_stuff(response):
            return response
```

Using the *Unchained* extension is another way to plug into the app factory, so let's look at that next.

3.4 The Unchained Extension

As an alternative to using `Bundle.before_init_app` and `Bundle.after_init_app`, the *Unchained* extension also acts as a drop-in replacement for some of the public API of `Flask`:

```
from flask_unchained import unchained

@unchained.before_first_request
def called_once_before_the_first_request():
    pass

# the other familiar decorators are also available:
@unchained.url_value_preprocessor
@unchained.url_defaults
@unchained.before_request
@unchained.after_request
@unchained.errorhandler
@unchained.teardown_request
@unchained.teardown_appcontext
@unchained.context_processor
@unchained.shell_context_processor
@unchained.template_filter
@unchained.template_global
@unchained.template_tag
@unchained.template_test
```

These decorators all work exactly the same as if you were using them from the `Flask` `app` instance itself.

The `Unchained` extension first forwards these calls to the `Flask` instance itself, and then it calls `RunHooksHook.run_hook(app, bundles)`. Hooks are where the real action of actually booting up the app happens.

3.5 App Factory Hooks

App Factory Hooks are what make sure all of the code from your configured list of bundles gets discovered and registered correctly with both the Flask `app` instance and the Unchained extension.

Important

Hooks are what define the patterns to load and customize everything in bundles. By default, to override something, you just place it in your bundle with the same name and in the same location (module) as whatever you want to override, or to extend something, do the same while also subclassing whatever you wish to extend. In other words, you just use standard object-oriented Python while following consistent naming conventions.

Advanced

While it shouldn't be necessary, you can even extend and/or override hooks themselves if you need to customize their behavior.

These are some of the hooks Flask Unchained includes:

InitExtensionsHook Discovers Flask extensions in bundles and initializes them with the app.

RegisterServicesHook Discovers services in bundles and registers them with the *Unchained* extension. Both services and extensions are dependency-injectable at runtime into just about anything that can be wrapped with the *inject()* decorator.

ConfigureAppHook Discovers configuration options in bundles and registers them with the app.

RegisterCommandsHook Discovers CLI commands in bundles and registers them with the app.

Hooks are also loaded from bundles, for example the Controller Bundle includes these:

RegisterRoutesHook Discovers all views/routes in bundles and registers any "top-level" ones with the app.

RegisterBundleBlueprintsHook Registers the views/routes in bundles as blueprints with the app. Each bundle gets (conceptually, is) its own blueprint.

RegisterBlueprintsHook Discovers legacy Flask Blueprints and registers them with the app.

For our simple "hello world" app, most of these are no-ops, with the exception of the hook to register bundle blueprints. This is the essence of it:

```
# flask_unchained/bundles/controller/hooks/register_bundle_blueprints_hook.py

from flask_unchained import AppFactoryHook, Bundle, FlaskUnchained
from flask_unchained.bundles.controller.bundle_blueprint import BundleBlueprint

class RegisterBundleBlueprintsHook(AppFactoryHook):
    def run_hook(self,
                 app: FlaskUnchained,
                 bundles: List[Bundle],
                 unchained_config: Optional[Dict[str, Any]] = None,
                 ) -> None:
        for bundle in bundles:
            bp = BundleBlueprint(bundle)
            for route in bundle.routes:
                bp.add_url_rule(route.full_rule,
```

(continues on next page)

(continued from previous page)

```
                                defaults=route.defaults,
                                endpoint=route.endpoint,
                                methods=route.methods,
                                **route.rule_options)
            app.register_blueprint(bp)
```

And the result can be seen by running `flask urls`:

```
flask urls
Method(s)  Rule                       Endpoint                    View
-----------------------------------------------------------------------------------------
↪--------
      GET  /static/<path:filename>  static                      flask.helpers.send_
↪static_file
      GET  /                        site_controller.index       app.SiteController.
↪index
      GET  /hello                   site_controller.hello       app.SiteController.
↪hello
```

3.6 The Bundle Hierarchy

FIXME: Expand on the bundle hierarchy and inheritance concept! Show examples.

CHAPTER

FOUR

INCLUDED BUNDLES

4.1 Admin Bundle

Integrates Flask Admin with Flask Unchained.

4.1.1 Installation

Install dependencies:

```
pip install "flask-unchained[admin]"
```

Enable the bundle in your `unchained_config.py`:

```
# project-root/unchained_config.py

BUNDLES = [
    # ...
    'flask_unchained.bundles.admin',
    'app',
]
```

And include the Admin Bundle's routes:

```
# project-root/your_app_bundle/routes.py

routes = lambda: [
    include('flask_unchained.bundles.admin.routes'),
    # ...
]
```

4.1.2 Config

class flask_unchained.bundles.admin.config.**Config**

Config class for the Admin bundle. Defines which configuration values this bundle supports, and their default values.

ADMIN_NAME = 'Admin'

The title of the admin section of the site.

ADMIN_BASE_URL = '/admin'

Base url of the admin section of the site.

ADMIN_INDEX_VIEW = <flask_unchained.bundles.admin.views.dashboard.AdminDashboardView o
The `AdminIndexView` (or subclass) instance to use for the index view.

ADMIN_SUBDOMAIN = None
Subdomain of the admin section of the site.

ADMIN_BASE_TEMPLATE = 'admin/base.html'
Base template to use for other admin templates.

ADMIN_TEMPLATE_MODE = 'bootstrap4'
Which version of bootstrap to use. (bootstrap2, bootstrap3, or bootstrap4)

ADMIN_CATEGORY_ICON_CLASSES = {}
Dictionary of admin category icon classes. Keys are category names, and the values depend on which version of bootstrap you're using.

For example, with bootstrap4:

```
ADMIN_CATEGORY_ICON_CLASSES = {
    'Mail': 'fa fa-envelope',
    'Security': 'fa fa-lock',
}
```

ADMIN_ADMIN_ROLE_NAME = 'ROLE_ADMIN'
The name of the Role which represents an admin.

ADMIN_LOGIN_ENDPOINT = 'admin.login'
Name of the endpoint to use for the admin login view.

ADMIN_POST_LOGIN_REDIRECT_ENDPOINT = 'admin.index'
Name of the endpoint to redirect to after the user logs into the admin.

ADMIN_LOGOUT_ENDPOINT = 'admin.logout'
Name of the endpoint to use for the admin logout view.

ADMIN_POST_LOGOUT_REDIRECT_ENDPOINT = 'admin.login'
Endpoint to redirect to after the user logs out of the admin.

4.1.3 API Docs

See *Admin Bundle API*

4.2 API Bundle

Integrates Marshmallow and APISpec with SQLAlchemy and Flask Unchained.

4.2.1 Installation

Install dependencies:

```
pip install "flask-unchained[api]"
```

And enable the API bundle in your `unchained_config.py`:

```
# your_project_root/unchained_config.py

BUNDLES = [
    # ...
    'flask_unchained.bundles.api',
    'app',
]
```

4.2.2 Usage

The API bundle includes two extensions, *Api* and *Marshmallow*. The *Api* extension is used for generating OpenAPI documentation and the *Marshmallow* extension is used for serialization. These should be imported like so:

```
from flask_unchained.bundles.api import api, ma
```

Model Serializers

ModelSerializer is very similar to Flask Marshmallow's `ModelSchema`. There are two differences:

- dependency injection is automatically set up, and
- we automatically convert field names to/from camel case when dumping/loading (although this is customizable)

Let's say you have the following model:

```
# your_bundle/models.py

from flask_unchained.bundles.sqlalchemy import db

class User(db.Model):
    name = db.Column(db.String)
    email = db.Column(db.String)
    password = db.Column(db.String)
```

A simple serializer for it would look like this:

```
# your_bundle/serializers.py

from flask_unchained.bundles.api import ma

from .models import User


class UserSerializer(ma.ModelSerializer):
    class Meta:
        model = User
```

One gotchya here is that Marshmallow has no way to know that the email column should use an email field. Therefore, we need to help it out a bit:

```
# your_bundle/serializers.py

from flask_unchained.bundles.api import ma

from .models import User


class UserSerializer(ma.ModelSerializer):
    class Meta:
        model = User
        load_only = ('password',)

    email = ma.Email(required=True)
```

There are three separate contexts for (de)serialization:

- standard: dumping/loading a single object
- many: dumping/loading multiple objects
- create: creating a new object

By default, any serializers you define will be used for all three. This can be customized:

```
# your_bundle/serializers.py

from flask_unchained.bundles.api import ma

@ma.serializer(many=True)
class UserSerializerMany(ma.ModelSerializer):
    # ...

@ma.serializer(create=True)
class UserSerializerCreate(ma.ModelSerializer):
    # ...
```

Let's make a model resource so we'll have API routes for it:

Model Resources

```
# your_bundle/views.py

from flask_unchained.bundles.api import ModelResource

from .models import User


class UserResource(ModelResource):
    class Meta:
        model = User
```

Add it to your routes:

```
# your_app_bundle/routes.py
```

(continues on next page)

(continued from previous page)

```
routes = lambda: [
    prefix('/api/v1', [
        resource('/users', UserResource),
    ],
]
```

And that's it, unless you need to customize any behavior.

Model Resource Meta Options

ModelResource inherits all of the meta options from *Controller* and *Resource*, and it adds some options of its own:

meta option name	description	default value
model	The model class to use for the resource.	None (it's required to be set by you)
serializer	The serializer instance to use for (de)serializing an individual model.	Determined automatically by the model name. Can be set manually to override the automatic discovery.
serializer_create	The serializer instance to use for loading data for creation of a new model.	Determined automatically by the model name. Can be set manually to override the automatic discovery.
serializer_many	The serializer instance to use for (de)serializing a list of models.	Determined automatically by the model name. Can be set manually to override the automatic discovery.
include_methods	A list of resource methods to automatically include.	('list', 'create', 'get', 'patch', 'put', 'delete')
exclude_methods	A list of resource methods to exclude.	()
include_decorators	A list of resource methods for which to automatically apply the default decorators.	('list', 'create', 'get', 'patch', 'put', 'delete')
exclude_decorators	A list of resource methods for which to *not* automatically apply the default decorators.	()
method_decorators	This can either be a list of decorators to apply to *all* methods, or a dictionary of method names to a list of decorators to apply for each method. In both cases, decorators specified here are run *before* the default decorators.	()

4.2.3 API Docs

See *API Bundle API*

4.3 Babel Bundle

The babel bundle provides support for internationalization and localization by integrating Flask BabelEx with Flask Unchained.

4.3.1 Installation

The babel bundle comes enabled by default.

4.3.2 Config

class flask_unchained.bundles.babel.config.**Config**
Default configuration options for the Babel Bundle.

LANGUAGES = ['en']
The language codes supported by the app.

BABEL_DEFAULT_LOCALE = 'en'
The default language to use if none is specified by the client's browser.

BABEL_DEFAULT_TIMEZONE = 'UTC'
The default timezone to use.

DEFAULT_DOMAIN = <flask_babelex.Domain object>
The default `Domain` to use.

DATE_FORMATS = {'date': 'medium', 'date.full': None, 'date.long': None, 'date.mediu
A dictionary of date formats.

ENABLE_URL_LANG_CODE_PREFIX = False
Whether or not to enable the capability to specify the language code as part of the URL.

class flask_unchained.bundles.babel.config.**DevConfig**

LAZY_TRANSLATIONS = False
Do not use lazy translations in development.

class flask_unchained.bundles.babel.config.**ProdConfig**

LAZY_TRANSLATIONS = True
Use lazy translations in production.

class flask_unchained.bundles.babel.config.**StagingConfig**

LAZY_TRANSLATIONS = True
Use lazy translations in staging.

4.3.3 Commands

flask babel

Babel translations commands.

```
flask babel COMMAND [<args>...] [OPTIONS]
```

compile

Compile translations into a distributable `.mo` file.

```
flask babel compile [OPTIONS]
```

Options

-d, --domain <domain>

extract

Extract newly added translations keys from source code.

```
flask babel extract [OPTIONS]
```

Options

-d, --domain <domain>

init

Initialize translations for a language code.

```
flask babel init <lang> [OPTIONS]
```

Options

-d, --domain <domain>

Arguments

LANG
Required argument

update

Update language-specific translations files with new keys discovered by `flask babel extract`.

```
flask babel update [OPTIONS]
```

Options

-d, --domain <domain>

4.3.4 API Docs

See *Babel Bundle API*

4.4 Celery Bundle

Integrates Celery with Flask Unchained.

4.4.1 Dependencies

- A broker of some sort; Redis or RabbitMQ are popular choices.

4.4.2 Installation

Install dependencies:

```
pip install "flask-unchained[celery]" <broker-of-choice>
```

And enable the celery bundle in your `unchained_config.py`:

```
# your_project_root/unchained_config.py

BUNDLES = [
    # ...
    'flask_unchained.bundles.celery',
    'app',
]
```

NOTE: If you have enabled the *Mail Bundle*, and want to send emails asynchronously using celery, then you must list the celery bundle *after* the mail bundle in `BUNDLES`.

4.4.3 Config

class flask_unchained.bundles.celery.config.**Config**
Default configuration options for the Celery Bundle.

CELERY_BROKER_URL = 'redis://127.0.0.1:6379/0'
The broker URL to connect to.

CELERY_RESULT_BACKEND = 'redis://127.0.0.1:6379/0'
The result backend URL to connect to.

CELERY_ACCEPT_CONTENT = ('json', 'pickle', 'dill')
Tuple of supported serialization strategies.

MAIL_SEND_FN(*to=None*, *template=None*, ***kwargs*)
If the celery bundle is listed *after* the mail bundle in `unchained_config.BUNDLES`, then this configures the mail bundle to send emails asynchronously.

4.4.4 Commands

flask celery

Celery commands.

```
flask celery COMMAND [<args>...] [OPTIONS]
```

beat

Start the celery beat.

```
flask celery beat [OPTIONS]
```

worker

Start the celery worker.

```
flask celery worker [OPTIONS]
```

4.4.5 API Docs

See *Celery Bundle API*

4.5 Controller Bundle

The controller bundle provides the primary means for defining views and registering their routes with Flask. It is also the controller bundle that makes all of your other bundles blueprints.

4.5.1 Installation

The controller bundle comes enabled by default.

4.5.2 Usage

Config

class flask_unchained.bundles.controller.config.**Config**
Default configuration options for the controller bundle.

FLASH_MESSAGES = True
Whether or not to enable flash messages.

NOTE: This only works for messages flashed using the *flask_unchained.Controller.flash()* method; using the `flask.flash()` function directly will not respect this setting.

TEMPLATE_FILE_EXTENSION = '.html'
The default file extension to use for templates.

WTF_CSRF_ENABLED = False
Whether or not to enable CSRF protection.

Views and Controllers

Like stock Flask, Flask Unchained supports using views defined the "standard" way, ie:

```
# your_app/views.py

from flask import Blueprint, render_template


bp = Blueprint('bp', __name__)

@bp.route('/foo')
def foo():
   return render_template('bp/foo.html')
```

View functions defined this way have a major drawback, however. And that is, as soon as this code gets imported, the view's route gets registered with the app. Most of the time this is the desired behavior, and it will work as expected with code placed in your app bundle, however bundles meant for distribution to third parties must avoid defining views in this way, because it makes it impossible for the view to be overridden and/or its route to be customized.

The recommended way to define views with Flask Unchained is by using controller classes:

```
# your_app_bundle/views.py

from flask_unchained import Controller, route

```

(continues on next page)

(continued from previous page)

```
class SiteController(Controller):
    @route('/foo')
    def foo():
        return self.render('foo')
```

This view will do the same thing as the prior example view, however, using *Controller* as the base class has the advantage of making views and their routes easily and independently customizable. Controllers also support automatic dependency injection and include some convenience methods such as *flash()*, *jsonify()*, *redirect()*, and *render()*.

Unlike when using the stock Flask decorators to register a view's routes, in Flask Unchained we must explicitly enable the routes we want:

```
# your_app_bundle/routes.py

from flask_unchained import (controller, resource, func, include, prefix,
                             get, delete, post, patch, put, rule)

from .views import SiteController


routes = lambda: [
    controller(SiteController),
]
```

By default this code will register all of the view functions on `SiteController` with the app, using the default routing rules as defined on the view methods of the controller.

Declarative Routing

Declaration of routes in Flask Unchained can be uncoupled from the definition of the views themselves. Using function- and class-based views is supported. You already saw a simple example above, that would register a single route at `/foo` for the `SiteController.foo` method. Let's say we wanted to change it to `/site/foobar`:

```
# your_app_bundle/routes.py

from flask_unchained import (controller, resource, func, include, prefix,
                             get, delete, post, patch, put, rule)

from .views import SiteController

routes = lambda: [
    # this is probably the clearest way to do it:
    controller('/site', SiteController, rules=[
        get('/foobar', SiteController.foo),
    ]),

    # or you can provide the method name of the controller as a string:
    controller('/site', SiteController, rules=[
        get('/foobar', 'foo'),
    ]),

    # or use nesting to produce the same result (this is especially useful when
    # you want to prefix more than one view/route with the same url prefix)
```

(continues on next page)

(continued from previous page)

```
    prefix('/site', [
        controller(SiteController, rules=[
            get('/foobar', SiteController.foo),
        ]),
    ]),
]
```

Here is a summary of the functions imported at the top of your `routes.py`:

Table 1: Declarative Routing Functions

Function	Description
include()	Include all of the routes from the specified module at that point in the tree.
prefix()	Prefixes all of the child routing rules with the given prefix.
func()	Registers a function-based view with the app, optionally specifying the routing rules.
controller()	Registers a controller and its views with the app, optionally customizing the routes to register.
resource()	Registers a resource and its views with the app, optionally customizing the routes to register.
rule()	Define customizations to a controller/resource method's route rules.
get()	Defines a controller/resource method as only accepting the HTTP `GET` method, otherwise the same as *rule()*.
patch()	Defines a controller/resource method as only accepting the HTTP `PATCH` method, otherwise the same as *rule()*.
post()	Defines a controller/resource method as only accepting the HTTP `POST` method, otherwise the same as *rule()*.
put()	Defines a controller/resource method as only accepting the HTTP `PUT` method, otherwise the same as *rule()*.
delete()	Defines a controller/resource method as only accepting the HTTP `DELETE` method, otherwise the same as *rule()*.

Class-Based Views

The controller bundle includes two base classes that all of your views should extend. The first is *Controller*, which is implemented very similarly to `View`, however they're not compatible. The second is *Resource*, which extends *Controller*, and whose implementation draws a lot of inspiration from Flask-RSETful (specifically, the Resource and Api classes).

Controller

Unless you're building an API, chances are *Controller* is the base class you want to extend. Controllers include a bit of magic that deserves some explanation:

On any Controller subclass that doesn't specify itself as abstract, all methods not designated as protected by prefixing them with an _ are automatically assigned default routing rules. In the example below, `foo_baz` has a route decorator, but `foo` and `foo_bar` do not. The undecorated views will be assigned default route rules of `/foo` and `/foo-bar` respectively (the default is to convert the method name to kebab-case).

```
# your_bundle/views.py

from flask_unchained import Controller, route

class MyController(Controller):
```

(continues on next page)

(continued from previous page)

```
    def foo():
        return self.render('foo')

    def foo_bar():
        return self.render('foo_bar')

    @route('/foobaz')
    def foo_baz():
        return self.render('foo_baz')

    def _protected_function():
        return 'stuff'
```

Controller Meta Options

Controllers have a few meta options that you can use to customize their behavior:

```
# your_bundle/views.py

from flask_unchained import Controller, route

class SiteController(Controller):
    class Meta:
        abstract: bool = False                          # default is False
        decorators: List[callable] = ()                 # default is an empty tuple
        template_folder: str = 'site'                   # see explanation below
        template_file_extension: Optional[str] = None   # default is None
        url_prefix = Optional[str] = None               # default is None
        endpoint_prefix = 'site_controller'             # see explanation below
```

meta option name	description	default value
abstract	Whether or not this controller should be abstract. Abstract controller classes do not get any routes assigned to their view methods (if any exist).	False
decorators	A list of decorators to apply to all views in this controller.	()
template_folder	The name of the folder containing the templates for this controller's views.	Defaults to the snake_cased class name (with the `Controller` or `View` suffixes stripped).
template_file_extension	The filename extension to use for templates for this controller's views.	Defaults to your app config's `TEMPLATE_FILE_EXTENSION` setting, and overrides it if set.
url_prefix	The url prefix to use for all routes from this controller.	Defaults to `'/'` (aka no prefix).
endpoint_prefix	The endpoint prefix to use for all routes from this controller.	Defaults to the snake_cased class name.

Overriding Controllers

Controllers can be extended or overridden by creating an equivalently named class higher up in the bundle hierarchy. (In other words, either in a bundle that extends another bundle, or in your app bundle.) As an example, the security bundle includes *SecurityController*. To extend it, you would simply subclass it like any other class in Python and change what you need to:

```
# your_app_or_security_bundle/views.py

from flask_unchained.bundles.security import SecurityController as␣
↪BaseSecurityController

# to extend BaseSecurityController
class SecurityController(BaseSecurityController):
    pass

# to completely override it, just use the same name without extending the base class
class SecurityController:
    pass
```

Resource

The *Resource* class extends *Controller* to add support for building RESTful APIs. It adds a bit of magic around specific methods:

HTTP Method	Resource class method name	URL Rule
GET	list	/
POST	create	/
GET	get	/<cls.Meta.member_param>
PATCH	patch	/<cls.Meta.member_param>
PUT	put	/<cls.Meta.member_param>
DELETE	delete	/<cls.Meta.member_param>

If you implement any of these methods, then the shown URL rules will automatically be used.

So, for example:

```
from http import HTTPStatus
from flask_unchained import Resource, injectable, param_converter, request
from flask_unchained.bundles.security import User, UserManager


class UserResource(Resource):
    class Meta:
        url_prefix = '/users'
        member_param = '<int:id>'
        unique_member_param = '<int:user_id>'

    user_manager: UserManager = injectable

    def list():
        return self.jsonify(dict(users=self.user_manager.all()))
```

(continues on next page)

(continued from previous page)

```
    def create():
        data = request.get_json()
        user = self.user_manager.create(**data, commit=True)
        return self.jsonify(dict(user=user), code=HTTPStatus.CREATED)

    @param_converter(id=User)
    def get(user):
        return self.jsonify(dict(user=user)

    @param_converter(id=User)
    def patch(user):
        data = request.get_json()
        user = self.user_manager.update(user, **data, commit=True)
        return self.jsonify(dict(user=user))

    @param_converter(id=User)
    def put(user):
        data = request.get_json()
        user = self.user_manager.update(user, **data, commit=True)
        return self.jsonify(dict(user=user))

    @param_converter(id=User)
    def delete(user):
        self.user_manager.delete(user, commit=True)
        return self.make_response('', code=HTTPStatus.NO_CONTENT)
```

Registered like so:

```
routes = lambda: [
    resource(UserResource),
]
```

Results in the following routes:

```
GET     /users              UserResource.list
POST    /users              UserResource.create
GET     /users/<int:id>     UserResource.get
PATCH   /users/<int:id>     UserResource.patch
PUT     /users/<int:id>     UserResource.put
DELETE  /users/<int:id>     UserResource.delete
```

Resource Meta Options

Resources have a few extra meta options on top of those that Controller includes:

```
# your_bundle/views.py

from flask_unchained import Controller, route

class UserResource(Resource):
    class Meta:
        abstract: bool = False                      # the default
        decorators: List[callable] = ()             # the default
        url_prefix = Optional[str] = '/users'       # automatically determined
```

(continues on next page)

(continued from previous page)

```
        member_param: str = '<int:id>'              # the default
        unique_member_parm: str = '<int:user_id>'   # automatically determined
```

meta option name	description	default value
`url_prefix`	The url prefix to use for all routes from this resource.	Defaults to the pluralized, kebab-cased class name (without the Resource suffix)
`member_param`	The url parameter rule to use for the special member functions (`get`, `patch`, `put`, and `delete`) of this resource.	`<int:id>`
`unique_member_param`	The url parameter rule to use for the special member methods (`get`, `patch`, `put`, and `delete`) of this resource when `member_param` conflicts with a subresource's `member_param`.	`<{type}:{prefix}_{name}>` where `type` and `name` come from `member_param` and `prefix` is the snake_cased class name (without the Resource suffix)

Overriding Resources

Because *`Resource`* is already a subclass of *`Controller`*, overriding resources works the same way as for controllers.

Templating

Flask Unchained uses the Jinja templating language, just like stock Flask.

By default bundles are configured to use a `templates` subfolder. This is configurable by setting the `template_folder` attribute on your `Bundle` subclass to a custom path.

Controllers each have their own template folder within `Bundle.template_folder`. It defaults to the snake_cased class name, with the suffixes `Controller` or `View` stripped (if any). You can customize it using `flask_unchained.Controller.Meta.template_folder`.

The default file extension used for templates is configured by setting `TEMPLATE_FILE_EXTENSION` in your app config. It defaults to `.html`, and is also configurable on a per-controller basis by setting `flask_unchained.Controller.Meta.template_file_extension`.

Taking the above into account, given the following controller:

```
class SiteController(Controller):
    @route('/')
    def index():
        return self.render('index')
```

Then the corresponding folder structure would look like this:

```
./your_bundle
├── templates
│   └── site
│       └── index.html
├── __init__.py
└── views.py
```

Extending and Overriding Templates

Templates can be overridden by placing an equivalently named template higher up in the bundle hierarchy.

So for example, the Security Bundle includes default templates for all of its views. They are located at `security/login.html`, `security/register.html`, and so on. Thus, to extend or override them, you would make a `security` folder in your app bundle's `templates` folder and put your customized templates with the same names in it. You can even extend the template you're overriding, using the standard Jinja syntax (this doesn't work in regular Flask apps):

```
{# your_app_or_security_bundle/templates/security/login.html #}

{% extends 'security/login.html' %}

{% block content %}
   <h1>Login</h1>
   {{ render_form(login_user_form, endpoint='security_controller.login') }}
{% endblock %}
```

If you encounter problems, you can set the `EXPLAIN_TEMPLATE_LOADING` config option to `True` to help debug what's going on.

Dependency Injection

Controllers are configured with dependency injection set up on them automatically. You can use class attributes or the constructor (or both).

Here's an example of using class attributes:

```
from flask_unchained import Controller, injectable
from flask_unchained.bundles.security import Security, SecurityService,␣
↪SecurityUtilsService
from flask_unchained.bundles.sqlalchemy import SessionManager

class SecurityController(Controller):
    security: Security = injectable
    security_service: SecurityService = injectable
    security_utils_service: SecurityUtilsService = injectable
    session_manager: SessionManager = injectable
```

And here's what the same thing using the constructor looks like:

```
from flask_unchained import Controller, injectable
from flask_unchained.bundles.security import Security, SecurityService,␣
↪SecurityUtilsService
from flask_unchained.bundles.sqlalchemy import SessionManager

class SecurityController(Controller):
    def __init__(self,
                 security: Security = injectable,
                 security_service: SecurityService = injectable,
                 security_utils_service: SecurityUtilsService = injectable,
                 session_manager: SessionManager = injectable):
        self.security = security
        self.security_service = security_service
        self.security_utils_service = security_utils_service
        self.session_manager = session_manager
```

4.5.3 API Docs

See *Controller Bundle API*

4.6 Graphene Bundle

Integrates Flask GraphQL and Graphene-SQLAlchemy with Flask Unchained.

4.6.1 Installation

Install dependencies:

```
pip install "flask-unchained[graphene]"
```

And enable the graphene bundle in your `unchained_config.py`:

```
# project-root/unchained_config.py

BUNDLES = [
    # ...
    'flask_unchained.bundles.graphene',
    'app',
]
```

4.6.2 Usage

Create a `graphene` module in your bundle:

```
cd your_bundle
mkdir graphene && touch graphene/__init__.py
touch graphene/mutations.py graphene/queries.py graphene/types.py
```

Let's say you have some models you want to create a GraphQL schema for:

```
# your_bundle/models.py

from flask_unchained.bundles.sqlalchemy import db

class Parent(db.Model):
    name = db.Column(db.String)

    children = db.relationship('Child', back_populates='parent',
                               cascade='all,delete,delete-orphan')

class Child(db.Model):
    name = db.Column(db.String)

    parent_id = db.foreign_key('Parent')
    parent = db.relationship('Parent', back_populates='children')
```

Object Types

The first step is to define your object types for your models:

```
# your_bundle/graphene/types.py

import graphene
from flask_unchained.bundles.graphene import SQLAlchemyObjectType

from .. import models

class Parent(SQLAlchemyObjectType):
    class Meta:
        model = models.Parent
        only_fields = ('id', 'name', 'created_at', 'updated_at')

    children = graphene.List(lambda: Child)

class Child(SQLAlchemyObjectType):
    class Meta:
        model = models.Child
        only_fields = ('id', 'name', 'created_at', 'updated_at')

    parent = graphene.Field(Parent)
```

Queries

Next define the queries for your types:

```
# your_bundle/graphene/queries.py

import graphene

from flask_unchained.bundles.graphene import QueriesObjectType

from . import types

class Queries(QueriesObjectType):
    parent = graphene.Field(types.Parent, id=graphene.ID(required=True))
    parents = graphene.List(types.Parent)

    child = graphene.Field(types.Child, id=graphene.ID(required=True))
    children = graphene.List(types.Child)
```

When subclassing `QueriesObjectType`, it automatically adds default resolvers for you. But these can be overridden if you want, eg:

```
# your_bundle/graphene/queries.py

import graphene

from flask_unchained import unchained
from flask_unchained.bundles.graphene import QueriesObjectType

from .. import services
from . import types
```

(continues on next page)

(continued from previous page)

```
child_manager: services.ChildManager = unchained.get_local_proxy('child_manager')

class Queries(QueriesObjectType):
    # ...
    children = graphene.List(types.Child, parent_id=graphene.ID())

    def resolve_children(self, info, parent_id=None, **kwargs):
        if not parent_id:
            return child_manager.all()
        return child_manager.filter_by_parent_id(parent_id).all()
```

Note

Unfortunately, dependency injection does not work with graphene classes. (That said, you can still decorate individual methods with `@unchained.inject()`.)

Mutations

Graphene mutations, per the `RegisterGrapheneMutationsHook`, by default live in the `graphene.mutations` module of bundles. This can be customized by setting the `graphene_mutations_module_names` attribute on your bundle class.

```
import graphene

from flask_unchained import unchained, injectable, lazy_gettext as _
from flask_unchained.bundles.graphene import MutationsObjectType,␣
↪MutationValidationError

from flask_unchained.bundles.security.exceptions import AuthenticationError
from flask_unchained.bundles.security.services import SecurityService,␣
↪SecurityUtilsService
from flask_unchained.bundles.security.graphene.types import UserInterface

class LoginUser(graphene.Mutation):
    class Arguments:
        email = graphene.String(required=True)
        password = graphene.String(required=True)

    success = graphene.Boolean(required=True)
    message = graphene.String()
    user = graphene.Field(UserInterface)

    @unchained.inject('security_service')
    def mutate(
        self,
        info,
        email: str,
        password: str,
        security_service: SecurityService = injectable,
        **kwargs,
    ):
        try:
```

(continues on next page)

(continued from previous page)

```
            user = LoginUser.validate(email, password)
        except MutationValidationError as e:
            return LoginUser(success=False, message=e.args[0])

        try:
            security_service.login_user(user)
        except AuthenticationError as e:
            return LoginUser(success=False, message=e.args[0])

        return LoginUser(success=True, user=user)

    @staticmethod
    @unchained.inject('security_utils_service')
    def validate(
        email: str,
        password: str,
        security_utils_service: SecurityUtilsService = injectable,
    ):
        user = security_utils_service.user_loader(email)
        if user is None:
            raise MutationValidationError(
                _('flask_unchained.bundles.security:error.user_does_not_exist'))

        if not security_utils_service.verify_password(user, password):
            raise MutationValidationError(
                _('flask_unchained.bundles.security:error.invalid_password'))

        return user

class LogoutUser(graphene.Mutation):
    success = graphene.Boolean(required=True)

    @unchained.inject('security_service')
    def mutate(self, info, security_service: SecurityService = injectable, **kwargs):
        security_service.logout_user()
        return LogoutUser(success=True)

class SecurityMutations(MutationsObjectType):
    login_user = mutations.LoginUser.Field(description="Login with email and password.
↪")
    logout_user = mutations.LogoutUser.Field(description="Logout the current user.")
```

4.6.3 API Docs

See *Graphene Bundle API*

4.7 Mail Bundle

Integrates Flask Mail with Flask Unchained.

4.7.1 Installation

Install dependencies:

```
pip install "flask-unchained[mail]"
```

And enable the bundle in your `unchained_config.py`:

```
# your_project_root/unchained_config.py

BUNDLES = [
    # ...
    'flask_unchained.bundles.mail',
    'app',
]
```

NOTE: If you have enabled the *Celery Bundle*, and want to send emails asynchronously using Celery, then you must list the celery bundle *after* the mail bundle in `BUNDLES`.

4.7.2 Config

class flask_unchained.bundles.mail.config.**Config**

Default configuration options for the mail bundle.

MAIL_SERVER = '127.0.0.1'

The hostname/IP of the mail server.

MAIL_PORT = 25

The port the mail server is running on.

MAIL_USERNAME = None

The username to connect to the mail server with, if any.

MAIL_PASSWORD = None

The password to connect to the mail server with, if any.

MAIL_USE_TLS = False

Whether or not to use TLS.

MAIL_USE_SSL = False

Whether or not to use SSL.

MAIL_DEFAULT_SENDER = 'Flask Mail <noreply@localhost>'

The default sender to use, if none is specified otherwise.

MAIL_SEND_FN(*to: Union[str, List[str], None] = None, template: Optional[str] = None, **kwargs*)

The function to use for sending emails. Defaults to `_send_mail()`. Any customized send function must implement the same function signature:

```
def send_mail(subject_or_message: Optional[Union[str, Message]] = None,
              to: Optional[Union[str, List[str]] = None,
              template: Optional[str] = None,
```

(continues on next page)

(continued from previous page)

```
            **kwargs):
    # ...
```

NOTE: *subject_or_message* is optional because you can also pass *subject* as a keyword argument, and *to* is optional because you can also pass *recipients* as a keyword argument. These are artifacts of backwards-compatibility with vanilla Flask-Mail.

MAIL_DEBUG = 0
The debug level to set for interactions with the mail server.

MAIL_MAX_EMAILS = None
The maximum number of emails to send per connection with the mail server.

MAIL_SUPPRESS_SEND = False
Whether or not to actually send emails, or just pretend to. This is mainly useful for testing.

MAIL_ASCII_ATTACHMENTS = False
Whether or not to coerce attachment filenames to ASCII.

class flask_unchained.bundles.mail.config.**DevConfig**
Development-specific config options for the mail bundle.

MAIL_DEBUG = 1
Set the mail server debug level to 1 in development.

MAIL_PORT = 1025
In development, the mail bundle is configured to connect to MailHog.

class flask_unchained.bundles.mail.config.**ProdConfig**
Production-specific config options for the mail bundle.

MAIL_PORT = 465
In production, the mail bundle is configured to connect using SSL.

MAIL_USE_SSL = True
Set use SSL to `True` in production.

class flask_unchained.bundles.mail.config.**TestConfig**
Test-specific config options for the mail bundle.

MAIL_SUPPRESS_SEND = True
Do not actually send emails when running tests.

4.7.3 Usage

After configuring the bundle, usage is simple:

```
from flask_unchained.bundles.mail import mail

mail.send_message('hello world', to='foo@bar.com')
```

`mail` is an instance of the *`Mail`* extension, and *`send_message()`* is the only public method on it. Technically, it's an alias for `send()`, which you can also use. (The `send()` method is maintained for backwards compatibility with the stock Flask Mail extension, although it has a different but compatible function signature than the original - we don't require that you manually create `Message` instances yourself before calling `send()`.)

4.7.4 Commands

flask mail

Mail commands.

```
flask mail COMMAND [<args>...] [OPTIONS]
```

send-test-email

Attempt to send a test email to the given email address.

```
flask mail send-test-email [OPTIONS]
```

Options

--to <to>
Email address of the recipient.

--subject <subject>
Email subject.

4.7.5 pytest fixtures

The mail bundle includes one pytest fixture, `outbox()`, that you can use to verify that emails were sent:

```
def test_something(client, outbox):
    r = client.get('endpoint.that.sends.an.email')
    assert len(outbox) == 1
    assert outbox[0].subject == 'hello world'
    assert 'hello world' in outbox[0].html
```

4.7.6 API Docs

See *Mail Bundle API*

4.8 OAuth Bundle

Integrates Flask OAuthlib with Flask Unchained. This allows OAuth authentication to any OAuth Provider supported by Flask OAuthlib.

4.8.1 Installation

The OAuth Bundle depends on the Security Bundle, as well as a few third-party libraries:

```
pip install "flask-unchained[oauth,security,sqlalchemy]"
```

And enable the bundles in your `unchained_config.py`:

```
# your_project_root/unchained_config.py

BUNDLES = [
    # ...
    'flask_unchained.bundles.sqlalchemy',
    'flask_unchained.bundles.security',
    'flask_unchained.bundles.oauth',
    'app',
]
```

And set the OAuthController to your app's `routes.py`:

```
# your_app_bundle_root/routes.py

from flask_unchained.bundles.oauth.views import OAuthController

routes = lambda: [
    # ...
    controller(OAuthController),
]
```

4.8.2 Config

The OAuth bundle includes support for two remote providers by default:

- amazon
- github

To configure these, would look like this:

```
# your_app_bundle_root/config.py

import os

from flask_unchained import BundleConfig

class Config(BundleConfig):
    OAUTH_GITHUB_CONSUMER_KEY = os.getenv('OAUTH_GITHUB_CONSUMER_KEY', '')
    OAUTH_GITHUB_CONSUMER_SECRET = os.getenv('OAUTH_GITHUB_CONSUMER_SECRET', '')

    OAUTH_AMAZON_CONSUMER_KEY = os.getenv('OAUTH_AMAZON_CONSUMER_KEY', '')
    OAUTH_AMAZON_CONSUMER_SECRET = os.getenv('OAUTH_AMAZON_CONSUMER_SECRET', '')
```

You can also add other remote providers, for example to add support for the (made up) `abc` provider:

```
# your_app_bundle_root/config.py

import os
```

(continues on next page)

(continued from previous page)

```
from flask_unchained import BundleConfig

class Config(BundleConfig):
    OAUTH_REMOTE_APP_ABC = dict(
        consumer_key=os.getenv('OAUTH_ABC_CONSUMER_KEY', ''),
        consumer_secret=os.getenv('OAUTH_ABC_CONSUMER_SECRET', ''),
        base_url='https://api.abc.com/',
        access_token_url='https://abc.com/login/oauth/access_token',
        access_token_method='POST',
        authorize_url='https://abc.com/login/oauth/authorize'
        request_token_url=None,
        request_token_params={'scope': 'user:email'},
    )
```

Each remote provider is available at its respective endpoint: `/login/<remote-app-name>`

For more information and OAuth config examples see: Flask OAuthlib Examples

4.9 Security Bundle

Integrates Flask Login and Flask Principal with Flask Unchained. Technically speaking, this bundle is actually a heavily refactored fork of the Flask Security project. As of this writing, it is at approximate feature parity with Flask Security, and supports session and token authentication. (We've removed support for HTTP Basic Auth, tracking users' IP addresses and similar, as well as the experimental password-less login support.)

4.9.1 Installation

The Security Bundle depends on the SQLAlchemy Bundle, as well as a few third-party libraries:

```
pip install "flask-unchained[security,sqlalchemy]"
```

And enable the bundles in your `unchained_config.py`:

```
# your_project_root/unchained_config.py

BUNDLES = [
    # ...
    'flask_unchained.bundles.sqlalchemy',
    'flask_unchained.bundles.security',
    'app',
]
```

4.9.2 Config

class flask_unchained.bundles.security.config.**AuthenticationConfig**
Config options for logging in and out.

SECURITY_LOGIN_FORM
alias of *flask_unchained.bundles.security.forms.LoginForm*

SECURITY_DEFAULT_REMEMBER_ME = False
Whether or not the login form should default to checking the "Remember me?" option.

SECURITY_USER_IDENTITY_ATTRIBUTES = ['email']
List of attributes on the user model that can used for logging in with. Each must be unique.

SECURITY_POST_LOGIN_REDIRECT_ENDPOINT = '/'
The endpoint or url to redirect to after a successful login.

SECURITY_POST_LOGOUT_REDIRECT_ENDPOINT = '/'
The endpoint or url to redirect to after a user logs out.

class flask_unchained.bundles.security.config.**ChangePasswordConfig**
Config options for changing passwords

SECURITY_CHANGEABLE = False
Whether or not to enable change password functionality.

SECURITY_CHANGE_PASSWORD_FORM
alias of *flask_unchained.bundles.security.forms.ChangePasswordForm*

SECURITY_POST_CHANGE_REDIRECT_ENDPOINT = None
Endpoint or url to redirect to after the user changes their password.

SECURITY_SEND_PASSWORD_CHANGED_EMAIL = False
Whether or not to send the user an email when their password has been changed. Defaults to True, and it's strongly recommended to leave this option enabled.

class flask_unchained.bundles.security.config.**EncryptionConfig**
Config options for encryption hashing.

SECURITY_PASSWORD_SALT = 'security-password-salt'
Specifies the HMAC salt. This is only used if the password hash type is set to something other than plain text.

SECURITY_PASSWORD_HASH = 'bcrypt'
Specifies the password hash algorithm to use when hashing passwords. Recommended values for production systems are `argon2`, `bcrypt`, or `pbkdf2_sha512`. May require extra packages to be installed.

SECURITY_PASSWORD_SINGLE_HASH = False
Specifies that passwords should only be hashed once. By default, passwords are hashed twice, first with SECURITY_PASSWORD_SALT, and then with a random salt. May be useful for integrating with other applications.

SECURITY_PASSWORD_SCHEMES = ['argon2', 'bcrypt', 'pbkdf2_sha512', 'plaintext']
List of algorithms that can be used for hashing passwords.

SECURITY_PASSWORD_HASH_OPTIONS = {}
Specifies additional options to be passed to the hashing method.

SECURITY_DEPRECATED_PASSWORD_SCHEMES = ['auto']
List of deprecated algorithms for hashing passwords.

SECURITY_HASHING_SCHEMES = ['sha512_crypt']
List of algorithms that can be used for creating and validating tokens.

SECURITY_DEPRECATED_HASHING_SCHEMES = []
List of deprecated algorithms for creating and validating tokens.

class flask_unchained.bundles.security.config.**ForgotPasswordConfig**
Config options for recovering forgotten passwords

SECURITY_RECOVERABLE = False
Whether or not to enable forgot password functionality.

SECURITY_FORGOT_PASSWORD_FORM
alias of *flask_unchained.bundles.security.forms.ForgotPasswordForm*

SECURITY_RESET_PASSWORD_FORM
alias of *flask_unchained.bundles.security.forms.ResetPasswordForm*

SECURITY_RESET_PASSWORD_WITHIN = '5 days'
Specifies the amount of time a user has before their password reset link expires. Always pluralized the time unit for this value. Defaults to 5 days.

SECURITY_POST_RESET_REDIRECT_ENDPOINT = None
Endpoint or url to redirect to after the user resets their password.

SECURITY_INVALID_RESET_TOKEN_REDIRECT = 'security_controller.forgot_password'
Endpoint or url to redirect to if the reset token is invalid.

SECURITY_EXPIRED_RESET_TOKEN_REDIRECT = 'security_controller.forgot_password'
Endpoint or url to redirect to if the reset token is expired.

SECURITY_API_RESET_PASSWORD_HTTP_GET_REDIRECT = None
Endpoint or url to redirect to if a GET request is made to the reset password view. Defaults to None, meaning no redirect. Useful for single page apps.

SECURITY_SEND_PASSWORD_RESET_NOTICE_EMAIL = False
Whether or not to send the user an email when their password has been reset. Defaults to True, and it's strongly recommended to leave this option enabled.

class flask_unchained.bundles.security.config.**RegistrationConfig**
Config options for user registration

SECURITY_REGISTERABLE = False
Whether or not to enable registration.

SECURITY_REGISTER_FORM
alias of *flask_unchained.bundles.security.forms.RegisterForm*

SECURITY_POST_REGISTER_REDIRECT_ENDPOINT = None
The endpoint or url to redirect to after a user completes the registration form.

SECURITY_SEND_REGISTER_EMAIL = False
Whether or not send a welcome email after a user completes the registration form.

SECURITY_CONFIRMABLE = False
Whether or not to enable required email confirmation for new users.

SECURITY_SEND_CONFIRMATION_FORM
alias of *flask_unchained.bundles.security.forms.SendConfirmationForm*

SECURITY_LOGIN_WITHOUT_CONFIRMATION = False
Allow users to login without confirming their email first. (This option only applies when *SECURITY_CONFIRMABLE* is True.)

SECURITY_CONFIRM_EMAIL_WITHIN = '5 days'
How long to wait until considering the token in confirmation emails to be expired.

SECURITY_POST_CONFIRM_REDIRECT_ENDPOINT = None
Endpoint or url to redirect to after the user confirms their email. Defaults to SECURITY_POST_LOGIN_REDIRECT_ENDPOINT.

SECURITY_CONFIRM_ERROR_REDIRECT_ENDPOINT = None
Endpoint to redirect to if there's an error confirming the user's email.

class flask_unchained.bundles.security.config.**TokenConfig**
Config options for token authentication.

SECURITY_TOKEN_AUTHENTICATION_KEY = 'auth_token'
Specifies the query string parameter to read when using token authentication.

SECURITY_TOKEN_AUTHENTICATION_HEADER = 'Authentication-Token'
Specifies the HTTP header to read when using token authentication.

SECURITY_TOKEN_MAX_AGE = None
Specifies the number of seconds before an authentication token expires. Defaults to None, meaning the token never expires.

class flask_unchained.bundles.security.config.**Config**
Config options for the Security Bundle.

SECURITY_ANONYMOUS_USER
alias of flask_unchained.bundles.security.models.anonymous_user.AnonymousUser

SECURITY_UNAUTHORIZED_CALLBACK()
This callback gets called when authorization fails. By default we abort with an HTTP status code of 401 (UNAUTHORIZED).

SECURITY_DATETIME_FACTORY()
Factory function to use when creating new dates. By default we use datetime.now(timezone.utc) to create a timezone-aware datetime.

class flask_unchained.bundles.security.config.**TestConfig**
Default test settings for the Security Bundle.

SECURITY_PASSWORD_HASH = 'plaintext'
Disable password-hashing in tests (shaves about 30% off the test-run time)

4.9.3 Commands

flask users

User model commands.

```
flask users COMMAND [<args>...] [OPTIONS]
```

activate

Activate a user.

```
flask users activate <query> [OPTIONS]
```

Arguments

QUERY
Required argument

add-role

Add a role to a user.

```
flask users add-role [OPTIONS]
```

Options

-u, --user <user>
The query to search for a user by. For example, *id=5*, *email=a@a.com* or *first_name=A,last_name=B*.

-r, --role <role>
The query to search for a role by. For example, *id=5* or *name=ROLE_USER*.

confirm

Confirm a user account.

```
flask users confirm <query> [OPTIONS]
```

Arguments

QUERY
Required argument

create

Create a new user.

```
flask users create [OPTIONS]
```

Options

--email <email>
The user's email address.

--password <password>
The user's password.

--active, --inactive
Whether or not the new user should be active.

Default False

--confirmed-at <confirmed_at>
The date stamp the user was confirmed at (or enter "now") [default: None]

--send-email, --no-email
Whether or not to send the user a welcome email.

Default False

deactivate

Deactivate a user.

```
flask users deactivate <query> [OPTIONS]
```

Arguments

QUERY
Required argument

delete

Delete a user.

```
flask users delete <query> [OPTIONS]
```

Arguments

QUERY
Required argument

list

List users.

```
flask users list [OPTIONS]
```

remove-role

Remove a role from a user.

```
flask users remove-role [OPTIONS]
```

Options

-u, --user <user>
The query to search for a user by. For example, *id=5*, *email=a@a.com* or *first_name=A,last_name=B*.

-r, --role <role>
The query to search for a role by. For example, *id=5* or *name=ROLE_USER*.

set-password

Set a user's password.

```
flask users set-password <query> [OPTIONS]
```

Options

--password <password>
The new password to assign to the user.

--send-email, --no-email
Whether or not to send the user a notification email.

Default False

Arguments

QUERY
Required argument

flask roles

Role commands.

```
flask roles COMMAND [<args>...] [OPTIONS]
```

create

Create a new role.

```
flask roles create [OPTIONS]
```

Options

--name <name>
The name of the role to create, eg *ROLE_USER*.

delete

Delete a role.

```
flask roles delete <query> [OPTIONS]
```

Arguments

QUERY
Required argument

list

List roles.

```
flask roles list [OPTIONS]
```

4.9.4 API Docs

See *Security Bundle API*

4.10 Session Bundle

Integrates Flask Session with Flask Unchained. This bundle is only a thin wrapper around Flask Session, and usage in Flask Unchained is as simple as it gets. Enable the bundle, configure `SESSION_TYPE`, and you're off running.

4.10.1 Installation

Install dependencies:

```
pip install "flask-unchained[session]"
```

And enable the bundle in your `unchained_config.py`:

```
# your_project_root/unchained_config.py

BUNDLES = [
    # ...
    'flask_unchained.bundles.session',
    'app',
]
```

4.10.2 Config

You must configure `SESSION_TYPE`, and depending upon what you set it to, any other required options for that type:

SESSION_TYPE	Extra Required Options
`'null'`	(none)
`'redis'`	• *SESSION_REDIS*
`'memcached'`	• *SESSION_MEMCACHED*
`'filesystem'`	• *SESSION_FILE_DIR* • *SESSION_FILE_THRESHOLD* • *SESSION_FILE_MODE*
`'mongodb'`	• *SESSION_MONGODB* • *SESSION_MONGODB_DB* • *SESSION_MONGODB_COLLECT*
`'sqlalchemy'`	• *SESSION_SQLALCHEMY* • *SESSION_SQLALCHEMY_TABLE* • *SESSION_SQLALCHEMY_MODEL*

class flask_unchained.bundles.session.config.**DefaultFlaskConfigForSessions**

Default configuration options for sessions in Flask.

SESSION_COOKIE_NAME = 'session'

The name of the session cookie.

Defaults to `'session'`.

SESSION_COOKIE_DOMAIN = None
The domain for the session cookie. If this is not set, the cookie will be valid for all subdomains of `SERVER_NAME`.

Defaults to `None`.

SESSION_COOKIE_PATH = None
The path for the session cookie. If this is not set the cookie will be valid for all of `APPLICATION_ROOT` or if that is not set for '/'.

Defaults to `None`.

SESSION_COOKIE_HTTPONLY = True
Controls if the cookie should be set with the `httponly` flag. Browsers will not allow JavaScript access to cookies marked as `httponly` for security.

Defaults to `True`.

SESSION_COOKIE_SECURE = False
Controls if the cookie should be set with the `secure` flag. Browsers will only send cookies with requests over HTTPS if the cookie is marked `secure`. The application must be served over HTTPS for this to make sense.

Defaults to `False`.

PERMANENT_SESSION_LIFETIME = datetime.timedelta(31)
The lifetime of a permanent session as `datetime.timedelta` object or an integer representing seconds.

Defaults to 31 days.

SESSION_COOKIE_SAMESITE = None
Restrict how cookies are sent with requests from external sites. Limits the scope of the cookie such that it will only be attached to requests if those requests are "same-site". Can be set to `'Lax'` (recommended) or `'Strict'`.

Defaults to `None`.

SESSION_REFRESH_EACH_REQUEST = True
Controls the set-cookie behavior. If set to `True` a permanent session will be refreshed each request and get their lifetime extended, if set to `False` it will only be modified if the session actually modifies. Non permanent sessions are not affected by this and will always expire if the browser window closes.

Defaults to `True`.

class flask_unchained.bundles.session.config.**Config**
Default configuration options for the Session Bundle.

See Flask Session for more information.

SESSION_TYPE = 'null'
Specifies which type of session interface to use. Built-in session types:

- `'null'`: `NullSessionInterface` (default)
- `'redis'`: `RedisSessionInterface`
- `'memcached'`: `MemcachedSessionInterface`
- `'filesystem'`: `FileSystemSessionInterface`
- `'mongodb'`: `MongoDBSessionInterface`
- `'sqlalchemy'`: `SqlAlchemySessionInterface`

Defaults to `'null'`.

SESSION_PERMANENT = True
Whether use permanent session or not.

Defaults to `True`.

SESSION_USE_SIGNER = False
Whether sign the session cookie sid or not. If set to `True`, you have to set `SECRET_KEY`.

Defaults to `False`.

SESSION_KEY_PREFIX = 'session:'
A prefix that is added before all session keys. This makes it possible to use the same backend storage server for different apps.

Defaults to `'session:'`.

SESSION_REDIS = None
A `redis.Redis` instance.

By default, connect to `127.0.0.1:6379`.

SESSION_MEMCACHED = None
A `memcached.Client` instance.

By default, connect to `127.0.0.1:11211`.

SESSION_FILE_DIR = '/home/docs/checkouts/readthedocs.org/user_builds/flask-unchained/c
The folder where session files are stored.

Defaults to using a folder named `flask_sessions` in your current working directory.

SESSION_FILE_THRESHOLD = 500
The maximum number of items the session stores before it starts deleting some.

Defaults to 500.

SESSION_FILE_MODE = 384
The file mode wanted for the session files. Should be specified as an octal, eg `0o600`.

Defaults to `0o600`.

SESSION_MONGODB = None
A `pymongo.MongoClient` instance.

By default, connect to `127.0.0.1:27017`.

SESSION_MONGODB_DB = 'flask_session'
The MongoDB database you want to use.

Defaults to `'flask_session'`.

SESSION_MONGODB_COLLECT = 'sessions'
The MongoDB collection you want to use.

Defaults to `'sessions'`.

SESSION_SQLALCHEMY = <SQLAlchemyUnchained engine=None>
A `SQLAlchemy` extension instance.

SESSION_SQLALCHEMY_TABLE = 'flask_sessions'
The name of the SQL table you want to use.

Defaults to `flask_sessions`.

SESSION_SQLALCHEMY_MODEL = None
Set this if you need to customize the `BaseModel` subclass used for storing sessions in the database.

4.10.3 API Docs

See *Session Bundle API*

4.11 SQLAlchemy Bundle

Integrates Flask SQLAlchemy Unchained and Flask Migrate with Flask Unchained.

4.11.1 Dependencies

- Flask SQLAlchemy
- Flask Migrate
- SQLAlchemy
- Alembic
- Depending on your database of choice, you might also need a specific driver library.

4.11.2 Installation

Install dependencies:

```
pip install "flask-unchained[sqlalchemy]"
```

And enable the bundle in your `unchained_config.py`:

```
# your_project_root/unchained_config.py

BUNDLES = [
    # ...
    'flask_unchained.bundles.sqlalchemy',
    'app',
]
```

4.11.3 Config

class flask_unchained.bundles.sqlalchemy.config.**Config**
The default configuration options for the SQLAlchemy Bundle.

SQLALCHEMY_DATABASE_URI = 'sqlite:///db/development.sqlite'
The database URI that should be used for the connection. Defaults to using SQLite with the database file stored at `ROOT_PATH/db/<env>.sqlite`. See the SQLAlchemy Dialects documentation for more info.

SQLALCHEMY_TRANSACTION_ISOLATION_LEVEL = None
Set the engine-wide transaction isolation level.

See the docs for more info.

SQLALCHEMY_ECHO = False
If set to `True` SQLAlchemy will log all the statements issued to stderr which can be useful for debugging.

SQLALCHEMY_TRACK_MODIFICATIONS = False
If set to `True`, Flask-SQLAlchemy will track modifications of objects and emit signals. The default is `False`. This requires extra memory and should be disabled if not needed.

SQLALCHEMY_RECORD_QUERIES = None
Can be used to explicitly disable or enable query recording. Query recording automatically happens in debug or testing mode. See `get_debug_queries()` for more information.

SQLALCHEMY_BINDS = None
A dictionary that maps bind keys to SQLAlchemy connection URIs.

SQLALCHEMY_NATIVE_UNICODE = None
Can be used to explicitly disable native unicode support. This is required for some database adapters (like PostgreSQL on some Ubuntu versions) when used with improper database defaults that specify encoding-less databases.

SQLALCHEMY_POOL_SIZE = None
The size of the database pool. Defaults to the engine's default (usually 5).

SQLALCHEMY_POOL_TIMEOUT = None
Specifies the connection timeout in seconds for the pool.

SQLALCHEMY_POOL_RECYCLE = None
Number of seconds after which a connection is automatically recycled. This is required for MySQL, which removes connections after 8 hours idle by default. Note that Flask-SQLAlchemy automatically sets this to 2 hours if MySQL is used.

Certain database backends may impose different inactive connection timeouts, which interferes with Flask-SQLAlchemy's connection pooling.

By default, MariaDB is configured to have a 600 second timeout. This often surfaces hard to debug, production environment only exceptions like `2013: Lost connection to MySQL server` during query.

If you are using a backend (or a pre-configured database-as-a-service) with a lower connection timeout, it is recommended that you set *SQLALCHEMY_POOL_RECYCLE* to a value less than your backend's timeout.

SQLALCHEMY_MAX_OVERFLOW = None
Controls the number of connections that can be created after the pool reached its maximum size. When those additional connections are returned to the pool, they are disconnected and discarded.

SQLALCHEMY_COMMIT_ON_TEARDOWN = False
Whether or not to automatically commit on app context teardown. Defaults to False.

ALEMBIC = {'script_location': 'db/migrations'}
Used to set the directory where migrations are stored. *ALEMBIC* should be set to a dictionary, using the key *script_location* to set the directory. Defaults to `ROOT_PATH/db/migrations`.

ALEMBIC_CONTEXT = {'render_item': <function render_migration_item>, 'template_args':
Extra kwargs to pass to the constructor of the Flask-Migrate extension. If you need to change this, make sure to merge the defaults with your settings!

class flask_unchained.bundles.sqlalchemy.config.**TestConfig**
Default configuration options for testing.

SQLALCHEMY_DATABASE_URI = 'sqlite://'
The database URI to use for testing. Defaults to SQLite in memory.

4.11.4 Usage

Usage of the SQLAlchemy bundle starts with an import:

```
from flask_unchained.bundles.sqlalchemy import db
```

From there, usage is extremely similar to stock Flask SQLAlchemy, and aside from a few minor gotchyas, you should just be able to copy your models and they should work (please file a bug report if this doesn't work!). The gotchyas you should be aware of are:

- the automatic table naming has slightly different behavior if the model class name has sequential upper-case characters
- any models defined in not-app-bundles must declare themselves as `class Meta: lazy_mapped = True`
- you must use `back_populates` instead of `backref` in `relationship` declarations (it may be possible to implement support for backrefs, but honestly, using `back_populates` is more Pythonic anyway, because Zen of Python)
- many-to-many join tables must be declared as model classes

Furthermore, models in SQLAlchemy Unchained include three columns by default:

column name	description
`id`	the primary key
`created_at`	a timestamp of when the model got inserted into the database
`updated_at`	a timestamp of the last time the model was updated in the database

These are customizable by declaring meta options. For example to disable timestamping and to rename the primary key column to `pk`, it would look like this:

```
from flask_unchained.bundles.sqlalchemy import db


class Foo(db.Model):
    class Meta:
        pk = 'pk'
        created_at = None
        updated_at = None
```

Models support the following meta options:

meta option name	default	description
table	snake_cased model class name	The database tablename to use for this model.
pk	`'id'`	The name of the primary key column.
created_at	`'created_at'`	The name of the row creation timestamp column.
updated_at	`'updated_at'`	The name of the most recent row update timestamp column.
repr	`('id',)`	Column attributes to include in the automatic `__repr__`
validation	`True`	Whether or not to enable validation on the model for CRUD operations.
mv_for	`None`	Used for specifying the name of the model a `MaterializedView` is for, as a string.
relationships	`{}`	This is an automatically determined meta option, and is used for determining whether or not a model has the same relationships as its base model. This is useful when you want to override a model from a bundle but change its relationships. The code that determines this is rather experimental, and may not do the right thing. Please report any bugs you come across!

FIXME: Polymorphic Models

4.11.5 Commands

flask db

Perform database migrations.

```
flask db [OPTIONS] COMMAND [ARGS]...
```

branches

Show current branch points

```
flask db branches [OPTIONS]
```

Options

-d, --directory <directory>
Migration script directory (default is "migrations")

-v, --verbose
Use more verbose output

current

Display the current revision for each database.

```
flask db current [OPTIONS]
```

Options

-d, --directory <directory>
Migration script directory (default is "migrations")

-v, --verbose
Use more verbose output

--head-only
Deprecated. Use –verbose for additional output

downgrade

Revert to a previous version

```
flask db downgrade [OPTIONS] [REVISION]
```

Options

-d, --directory <directory>
Migration script directory (default is "migrations")

--sql
Don't emit SQL to database - dump to standard output instead

--tag <tag>
Arbitrary "tag" name - can be used by custom env.py scripts

-x, --x-arg <x_arg>
Additional arguments consumed by custom env.py scripts

Arguments

REVISION
Optional argument

drop

Drop database tables.

```
flask db drop [OPTIONS]
```

Options

--force

edit

Edit a revision file

```
flask db edit [OPTIONS] [REVISION]
```

Options

-d, --directory <directory>
Migration script directory (default is "migrations")

Arguments

REVISION
Optional argument

heads

Show current available heads in the script directory

```
flask db heads [OPTIONS]
```

Options

-d, --directory <directory>
Migration script directory (default is "migrations")

-v, --verbose
Use more verbose output

--resolve-dependencies
Treat dependency versions as down revisions

history

List changeset scripts in chronological order.

```
flask db history [OPTIONS]
```

Options

-d, --directory <directory>
Migration script directory (default is "migrations")

-r, --rev-range <rev_range>
Specify a revision range; format is [start]:[end]

-v, --verbose
Use more verbose output

-i, --indicate-current
Indicate current version (Alembic 0.9.9 or greater is required)

init

Creates a new migration repository.

```
flask db init [OPTIONS]
```

Options

-d, --directory <directory>
Migration script directory (default is "migrations")

--multidb
Support multiple databases

merge

Merge two revisions together, creating a new revision file

```
flask db merge [OPTIONS] [REVISIONS]...
```

Options

-d, --directory <directory>
Migration script directory (default is "migrations")

-m, --message <message>
Merge revision message

--branch-label <branch_label>
Specify a branch label to apply to the new revision

--rev-id <rev_id>
Specify a hardcoded revision id instead of generating one

Arguments

REVISIONS
Optional argument(s)

migrate

Autogenerate a new revision file (Alias for 'revision –autogenerate')

```
flask db migrate [OPTIONS]
```

Options

-d, --directory <directory>
Migration script directory (default is "migrations")

-m, --message <message>
Revision message

--sql
Don't emit SQL to database - dump to standard output instead

--head <head>
Specify head revision or <branchname>@head to base new revision on

--splice
Allow a non-head revision as the "head" to splice onto

--branch-label <branch_label>
Specify a branch label to apply to the new revision

--version-path <version_path>
Specify specific path from config for version file

--rev-id <rev_id>
Specify a hardcoded revision id instead of generating one

-x, --x-arg <x_arg>
Additional arguments consumed by custom env.py scripts

reset

Drop database tables and run migrations.

```
flask db reset [OPTIONS]
```

Options

--force

revision

Create a new revision file.

```
flask db revision [OPTIONS]
```

Options

-d, --directory <directory>
Migration script directory (default is "migrations")

-m, --message <message>
Revision message

--autogenerate
Populate revision script with candidate migration operations, based on comparison of database to model

--sql
Don't emit SQL to database - dump to standard output instead

--head <head>
Specify head revision or <branchname>@head to base new revision on

--splice
Allow a non-head revision as the "head" to splice onto

--branch-label <branch_label>
Specify a branch label to apply to the new revision

--version-path <version_path>
Specify specific path from config for version file

--rev-id <rev_id>
Specify a hardcoded revision id instead of generating one

show

Show the revision denoted by the given symbol.

```
flask db show [OPTIONS] [REVISION]
```

Options

-d, --directory <directory>
Migration script directory (default is "migrations")

Arguments

REVISION
Optional argument

stamp

'stamp' the revision table with the given revision; don't run any migrations

```
flask db stamp [OPTIONS] [REVISION]
```

Options

-d, --directory <directory>
Migration script directory (default is "migrations")

--sql
Don't emit SQL to database - dump to standard output instead

--tag <tag>
Arbitrary "tag" name - can be used by custom env.py scripts

Arguments

REVISION
Optional argument

upgrade

Upgrade to a later version

```
flask db upgrade [OPTIONS] [REVISION]
```

Options

-d, --directory <directory>
Migration script directory (default is "migrations")

--sql
Don't emit SQL to database - dump to standard output instead

--tag <tag>
Arbitrary "tag" name - can be used by custom env.py scripts

-x, --x-arg <x_arg>
Additional arguments consumed by custom env.py scripts

Arguments

REVISION
Optional argument

4.11.6 API Docs

See *SQLAlchemy Bundle API*

4.12 Webpack Bundle

CHAPTER

FIVE

COMMANDS

5.1 flask new

Generate new code for your Flask Unchained projects.

```
flask new COMMAND [<args>...] [OPTIONS]
```

5.1.1 project

Create a new Flask Unchained project.

```
flask new project <dest> [OPTIONS]
```

Options

-a, --app-bundle <app_bundle>
The module name to use for your app bundle.

--force, --no-force
Whether or not to force creation if project folder is not empty.

Default False

--prompt, --no-prompt
Whether or not to skip prompting and just use the defaults.

Default False

--dev, --no-dev
Whether or not to install development dependencies.

Default <function <lambda> at 0x7fa7fe5186a8>

--admin, --no-admin
Whether or not to install the Admin Bundle.

Default <function <lambda> at 0x7fa7fe5187b8>

--api, --no-api
Whether or not to install the API Bundle.

Default <function <lambda> at 0x7fa7fe5188c8>

--celery, --no-celery
Whether or not to install the Celery Bundle.

Default <function <lambda> at 0x7fa7fe5189d8>

--graphene, --no-graphene
Whether or not to install the Graphene Bundle.

Default <function <lambda> at 0x7fa7fe518ae8>

--mail, --no-mail
Whether or not to install the Mail Bundle.

Default <function <lambda> at 0x7fa7fe518bf8>

--oauth, --no-oauth
Whether or not to install the OAuth Bundle.

Default <function <lambda> at 0x7fa7fe518d08>

--security, --no-security
Whether or not to install the Security Bundle.

Default <function <lambda> at 0x7fa7fe518e18>

--session, --no-session
Whether or not to install the Session Bundle.

Default <function <lambda> at 0x7fa7fe518f28>

--sqlalchemy, --no-sqlalchemy
Whether or not to install the SQLAlchemy Bundle.

Default <function <lambda> at 0x7fa7fe5220d0>

--webpack, --no-webpack
Whether or not to install the Webpack Bundle.

Default <function <lambda> at 0x7fa7fe5221e0>

Arguments

DEST
Required argument

5.2 flask run

Run a local development server.

This server is for development purposes only. It does not provide the stability, security, or performance of production WSGI servers.

The reloader and debugger are enabled by default if FLASK_ENV=development or FLASK_DEBUG=1.

```
flask run [OPTIONS]
```

Options

-h, --host <host>
The interface to bind to.

-p, --port <port>
The port to bind to.

--cert <cert>
Specify a certificate file to use HTTPS.

--key <key>
The key file to use when specifying a certificate.

--reload, --no-reload
Enable or disable the reloader. By default the reloader is active if debug is enabled.

--debugger, --no-debugger
Enable or disable the debugger. By default the debugger is active if debug is enabled.

--eager-loading, --lazy-loader
Enable or disable eager loading. By default eager loading is enabled if the reloader is disabled.

--with-threads, --without-threads
Enable or disable multithreading.

--extra-files <extra_files>
Extra files that trigger a reload on change. Multiple paths are separated by ':'.

5.3 flask shell

Runs a shell in the app context. If `IPython` is installed, it will be used, otherwise the default Python shell is used.

```
flask shell [OPTIONS]
```

5.4 flask unchained

Flask Unchained commands.

```
flask unchained COMMAND [<args>...] [OPTIONS]
```

5.4.1 bundles

List registered bundles.

```
flask unchained bundles [OPTIONS]
```

5.4.2 config

Show current app config (or optionally just the options for a specific bundle).

```
flask unchained config [<bundle_name>] [OPTIONS]
```

Arguments

BUNDLE_NAME
Optional argument

5.4.3 extensions

List extensions.

```
flask unchained extensions [OPTIONS]
```

5.4.4 hooks

List registered hooks (in the order they run).

```
flask unchained hooks [OPTIONS]
```

5.4.5 services

List services.

```
flask unchained services [OPTIONS]
```

5.5 flask urls

List all URLs registered with the app.

```
flask urls [OPTIONS]
```

Options

--order-by <order_by>
Property to order by: methods, rule, endpoint, view, or priority (aka registration order with the app)

5.6 flask url

Show details for a specific URL.

```
flask url <url> [OPTIONS]
```

Options

--method <method>
Method for url to match (default: GET)

Arguments

URL
Required argument

5.7 flask clean

Recursively remove *.pyc and *.pyo files.

```
flask clean [OPTIONS]
```

5.8 flask lint

Run flake8.

```
flask lint [OPTIONS]
```

Options

-f, --fix-imports
Fix imports using isort, before linting

5.9 flask qtconsole

Starts qtconsole in the app context. **!!EXPERIMENTAL!!**

Only available if `Ipython`, `PyQt5` and `qtconsole` are installed.

```
flask qtconsole [OPTIONS]
```

CHAPTER

SIX

TESTING

6.1 Included pytest fixtures

6.1.1 app

flask_unchained.pytest.**app**(*request*)
Automatically used test fixture. Returns the application instance-under-test with a valid app context.

6.1.2 maybe_inject_extensions_and_services

flask_unchained.pytest.**maybe_inject_extensions_and_services**(*app*, *request*)
Automatically used test fixture. Allows for using services and extensions as if they were test fixtures:

```
def test_something(db, mail, security_service, user_manager):
    # assert important stuff
```

NOTE: This only works on tests themselves; it will *not* work on test fixtures

6.1.3 cli_runner

flask_unchained.pytest.**cli_runner**(*app*)
Yields an instance of *FlaskCliRunner*. Example usage:

```
from your_package.commands import some_command

def test_some_command(cli_runner):
    result = cli_runner.invoke(some_command)
    assert result.exit_code == 0
    assert result.output.strip() == 'output of some_command'
```

6.1.4 client

flask_unchained.pytest.**client**(*app*)
Yields an instance of *HtmlTestClient*. Example usage:

```
def test_some_view(client):
    r = client.get('some.endpoint')

    # r is an instance of :class:`HtmlTestResponse`
    assert r.status_code == 200
    assert 'The Page Title' in r.html
```

6.1.5 api_client

flask_unchained.pytest.**api_client**(*app*)
Yields an instance of *ApiTestClient*. Example usage:

```
def test_some_view(api_client):
    r = api_client.get('some.endpoint.returning.json')

    # r is an instance of :class:`ApiTestResponse`
    assert r.status_code == 200
    assert 'some_key' in r.json
```

6.1.6 templates

flask_unchained.pytest.**templates**(*app*)
Fixture to record which templates (if any) got rendered during a request. Example Usage:

```
def test_some_view(client, templates):
    r = client.get('some.endpoint')
    assert r.status_code == 200
    assert templates[0].template.name == 'some/template.html'
    assert templates[0].context.get('some_ctx_var') == 'expected value'
```

6.2 Testing related classes

6.2.1 FlaskCliRunner

class flask_unchained.pytest.**FlaskCliRunner**(*app*, ***kwargs*)
Extended from upstream to run commands within the Flask app context.

The CLI runner provides functionality to invoke a Click command line script for unit testing purposes in a isolated environment. This only works in single-threaded systems without any concurrency as it changes the global interpreter state.

Parameters

- **charset** – the character set for the input and output data. This is UTF-8 by default and should not be changed currently as the reporting to Click only works in Python 2 properly.
- **env** – a dictionary with environment variables for overriding.

- **echo_stdin** – if this is set to *True*, then reading from stdin writes to stdout. This is useful for showing examples in some circumstances. Note that regular prompts will automatically echo the input.

invoke(*cli=None*, *args=None*, ***kwargs*)

Invokes a command in an isolated environment. The arguments are forwarded directly to the command line script, the *extra* keyword arguments are passed to the `main()` function of the command.

This returns a `Result` object.

Parameters

- **cli** – the command to invoke
- **args** – the arguments to invoke
- **input** – the input data for *sys.stdin*.
- **env** – the environment overrides.
- **catch_exceptions** – Whether to catch any other exceptions than `SystemExit`.
- **extra** – the keyword arguments to pass to `main()`.
- **color** – whether the output should contain color codes. The application can still override this explicitly.

6.2.2 HtmlTestClient

class flask_unchained.pytest.**HtmlTestClient**(**args*, ***kwargs*)

Like `FlaskClient`, except it supports passing an endpoint as the first argument directly to the HTTP get/post/etc methods (no need to use `url_for`, unless your URL rule has parameter names that conflict with the keyword arguments of `EnvironBuilder`). It also adds support for following redirects. Example usage:

```
def test_something(client: HtmlTestClient):
    r = client.get('site_controller.index')
    assert r.status_code == 200
```

follow_redirects(*response*)

Follow redirects on a response after inspecting it. Example usage:

```
def test_some_view(client):
    r = client.post('some.endpoint.that.redirects', data=data)
    assert r.status_code == 302
    assert r.path == url_for('some.endpoint')

    r = client.follow_redirects(r)
    assert r.status_code == 200
```

6.2.3 HtmlTestResponse

class flask_unchained.pytest.**HtmlTestResponse**(*response=None*, *status=None*, *headers=None*, *mimetype=None*, *content_type=None*, *direct_passthrough=False*)

Like `flask.wrappers.Response`, except extended with methods for inspecting the parsed URL and automatically decoding the response to a string.

property scheme
Returns the URL scheme specifier of the response's url, eg http or https.

property netloc
Returns the network location part the response's url.

property path
Returns the path part of the response's url.

property params
Returns the parameters for the last path element in the response's url.

property query
Returns the query component from the response's url.

property fragment
Returns the fragment identifier from the response's url.

property html
Returns the response's data parsed to a string of html.

6.2.4 ApiTestClient

class flask_unchained.pytest.**ApiTestClient**(**args*, ***kwargs*)

Like *HtmlTestClient* except it supports automatic serialization to json of data, as well as setting the `Accept` and `Content-Type` headers to `application/json`.

open(**args*, ***kwargs*)

Takes the same arguments as the `EnvironBuilder` class with some additions: You can provide a `EnvironBuilder` or a WSGI environment as only argument instead of the `EnvironBuilder` arguments and two optional keyword arguments (*as_tuple*, *buffered*) that change the type of the return value or the way the application is executed.

Changed in version 0.5: If a dict is provided as file in the dict for the *data* parameter the content type has to be called *content_type* now instead of *mimetype*. This change was made for consistency with `werkzeug.FileWrapper`.

The *follow_redirects* parameter was added to *open()*.

Additional parameters:

Parameters

- **as_tuple** – Returns a tuple in the form `(environ, result)`
- **buffered** – Set this to True to buffer the application run. This will automatically close the application for you as well.
- **follow_redirects** – Set this to True if the *Client* should follow HTTP redirects.

6.2.5 ApiTestResponse

class flask_unchained.pytest.**ApiTestResponse**(*response=None*, *status=None*, *headers=None*, *mimetype=None*, *content_type=None*, *direct_passthrough=False*)

Like *HtmlTestResponse* except it adds methods for automatically parsing the response data as json and retrieving errors from the response data.

property json

Returns the response's data parsed from json.

property errors

If the response contains the key `errors`, return its value, otherwise returns an empty dictionary.

6.2.6 RenderedTemplate

class flask_unchained.pytest.**RenderedTemplate**(*template*, *context*)

A `namedtuple` returned by the *templates()* fixture.

property context

Alias for field number 1

property template

Alias for field number 0

CHAPTER

SEVEN

API REFERENCE

7.1 Flask Unchained API

flask_unchained.app_factory

AppFactory	The Application Factory Pattern for Flask Unchained.

flask_unchained.app_factory_hook

AppFactoryHook	Base class for hooks.

flask_unchained.bundles

AppBundle	Like *Bundle*, except used for the top-most application bundle.
Bundle	Base class for bundles.

flask_unchained.config

BundleConfig	Base class for configuration settings.

flask_unchained.di

Service	Base class for services.

flask_unchained.flask_unchained

FlaskUnchained	A simple subclass of `flask.Flask`.

flask_unchained.forms

FlaskForm	Base form class extending `flask_wtf.FlaskForm`.

flask_unchained.hooks

ConfigureAppHook	Updates `app.config` with the settings from each bundle.
InitExtensionsHook	Initializes extensions found in bundles with the current app.
RegisterCommandsHook	Registers commands and command groups from bundles.
RegisterExtensionsHook	Registers extensions found in bundles with the `unchained` extension.
RegisterServicesHook	Registers services for dependency injection.
RunHooksHook	An internal hook to discover and run all the other hooks.
ViewsHook	Allows configuring bundle views modules.

flask_unchained.string_utils

right_replace(string, old, new[, count])	Right replaces `count` occurrences of `old` with `new` in `string`.
slugify(string)	Converts a string into a url-safe slug.
pluralize(word[, pos, custom, classical])	Returns the plural of a given word, e.g., child => children.
singularize(word[, pos, custom])	Returns the singular of a given word.
camel_case(string)	Converts a string to camel case.
class_case(string)	Converts a string to class case.
kebab_case(string)	Converts a string to kebab case.
snake_case(string)	Converts a string to snake case.
title_case(string)	Converts a string to title case.

flask_unchained.unchained

Unchained	The `Unchained` extension.

flask_unchained.utils

AttrDict	A dictionary that allows using the dot operator to get and set keys.
ConfigProperty	Allows extension classes to create properties that proxy to the config value, eg `app.config[key]`.
ConfigPropertyMetaclass	Use this metaclass to enable config properties on extension classes.

cwd_import(module_name)	Attempt to import a module from the current working directory.
get_boolean_env(name, default)	Converts environment variables to boolean values.
safe_import_module(module_name)	Like `importlib.import_module()`, except it does not raise `ImportError` if the requested `module_name` is not found.
utcnow()	Returns a current timezone-aware `datetime.datetime` in UTC.

7.1.1 Constants

DEV

flask_unchained.constants.**DEV**
Used to specify the development environment.

PROD

flask_unchained.constants.**PROD**
Used to specify the production environment.

STAGING

flask_unchained.constants.**STAGING**
Used to specify the staging environment.

TEST

flask_unchained.constants.**TEST**
Used to specify the test environment.

injectable

flask_unchained.di.**injectable = 'INJECTABLE_PARAMETER'**
Use this to mark a service parameter as injectable. For example:

```
class MyService(Service):
    a_dependency: ADependency = injectable
```

This allows MyService to be used in two ways:

```
# 1. using dependency injection with Flask Unchained
my_service = MyService()

# 2. overriding the dependency injection (or used without Flask Unchained)
a_dependency = ADependency()
my_service = MyService(a_dependency)

# but, if you try to use it without Flask Unchained and without parameters:
my_service = MyService()  # raises ServiceUsageError
```

7.1.2 AppFactory

class flask_unchained.**AppFactory**
The Application Factory Pattern for Flask Unchained.

APP_CLASS
alias of flask_unchained.flask_unchained.FlaskUnchained

create_app(*env: Union[development, production, staging, test], bundles: Optional[List[str]] = None, *, _config_overrides: Optional[Dict[str, Any]] = None, _load_unchained_config: bool = True, **app_kwargs*) → flask_unchained.flask_unchained.FlaskUnchained
Flask Unchained Application Factory. Returns an instance of *APP_CLASS* (by default, *FlaskUnchained*).

Example Usage:

```
app = AppFactory().create_app(PROD)
```

Parameters

- **env** (*str*) – Which environment the app should run in. Should be one of "development", "production", "staging", or "test" (you can import them: from flask_unchained import DEV, PROD, STAGING, TEST)
- **bundles** (*List[str]*) – An optional list of bundle modules names to use. Overrides unchained_config.BUNDLES (mainly useful for testing).
- **app_kwargs** (*Dict[str, Any]*) – keyword argument overrides for the *APP_CLASS* constructor
- **_config_overrides** – a dictionary of config option overrides; meant for test fixtures (for internal use only).
- **_load_unchained_config** – Whether or not to try to load unchained_config (for internal use only).

Returns The initialized *APP_CLASS* app instance, ready to rock'n'roll

static load_unchained_config(*env: Union[development, production, staging, test]*) → module
Load the unchained config from the current working directory for the given environment. If env == "test", look for tests._unchained_config, otherwise check the value of the UNCHAINED environment variable, falling back to loading the unchained_config module.

get_app_kwargs(*app_kwargs: Dict[str, Any], bundles: List[flask_unchained.bundles.Bundle], env: Union[development, production, staging, test], unchained_config: Dict[str, Any]*) → Dict[str, Any]
Returns app_kwargs with default settings applied from unchained_config.

classmethod load_bundles(*bundle_package_names: Optional[List[str]] = None, unchained_config_module: Optional[module] = None*) → Tuple[Union[None, flask_unchained.bundles.AppBundle], List[flask_unchained.bundles.Bundle]]
Load bundle instances from the given list of bundle packages. If unchained_config_module is given and there was no app bundle listed in bundle_package_names, attempt to load the app bundle from the unchained config.

classmethod load_bundle(*bundle_package_name: str*) → flask_unchained.bundles.Bundle
Attempt to load the bundle instance from the given package.

classmethod bundle_from_module(*module: module*) → Optional[flask_unchained.bundles.Bundle]

Attempt to instantiate the bundle class from the given module.

7.1.3 AppFactoryHook

class flask_unchained.**AppFactoryHook**(*unchained: flask_unchained.unchained.Unchained*, *bundle: Optional[flask_unchained.bundles.Bundle] = None*)

Base class for hooks.

Hooks have one entry point, *run_hook()*, which can be overridden to completely customize the behavior of the subclass. The default behavior is to look for objects in *bundle_module_names* which pass the result of *type_check()*. These objects are collected from all bundles into a dictionary with keys the result of *key_name()*, starting from the base-most bundle, allowing bundle subclasses to override objects with the same name from earlier bundles.

Subclasses should implement at a minimum *bundle_module_names*, *process_objects()*, and *type_check()*. You may also need to set one or both of *run_before* or *run_after*. Also of interest, hooks can store objects on their bundle's instance, using *bundle*. Hooks can also modify the shell context using *update_shell_context()*.

name: str = 'app_factory_hook'

The name of this hook. Defaults to the snake_cased class name.

run_before: Union[List[str], Tuple[str, ...]] = ()

An optional list of hook names that this hook must run *before*.

run_after: Union[List[str], Tuple[str, ...]] = ()

An optional list of hook names that this hook must run *after*.

bundle_module_name: Optional[str] = None

If *require_exactly_one_bundle_module* is True, only load from this module name in bundles. Should be set to `None` if your hook does not use that default functionality.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = None

A list of the default module names this hook will load from in bundles. Should be set to `None` if your hook does not use that default functionality (or *require_exactly_one_bundle_module* is True).

require_exactly_one_bundle_module: bool = False

Whether or not to require that there must be exactly one module name to load from in bundles.

bundle_override_module_names_attr: str = None

The attribute name that bundles can set on themselves to override the module(s) this hook will load from for that bundle. The defaults are as follows:

If *require_exactly_one_bundle_module* and *bundle_module_name* are set, use `f'{YourHook.bundle_module_name}_module_name'`.

Otherwise if *bundle_module_names* is set, we use the same f-string, just with the first module name listed in *bundle_module_names*.

If neither of *bundle_module_name* or *bundle_module_names* is set, then this will be `None`.

discover_from_bundle_superclasses: bool = True

Whether or not to search the whole bundle hierarchy for objects.

limit_discovery_to_local_declarations: bool = True

Whether or not to only include objects declared within bundles (ie not imported from other places, like third-party code).

unchained = None
The *Unchained* extension instance.

bundle = None
The *Bundle* instance this hook is from (if any).

run_hook(*app: flask_unchained.flask_unchained.FlaskUnchained, bundles: List[flask_unchained.bundles.Bundle], unchained_config: Optional[Dict[str, Any]] = None*) → None
Hook entry point. Override to disable standard behavior of iterating over bundles to discover objects and processing them.

process_objects(*app: flask_unchained.flask_unchained.FlaskUnchained, objects: Dict[str, Any]*) → None
Implement to do stuff with discovered objects (eg, registering them with the app instance).

collect_from_bundles(*bundles: List[flask_unchained.bundles.Bundle], *, _initial_objects: Optional[Dict[str, Any]] = None*) → Dict[str, Any]
Collect objects where *type_check()* returns `True` from bundles. Discovered names (keys, typically the class names) are expected to be unique across bundle hierarchies, except for the app bundle, which can override anything from other bundles.

collect_from_bundle(*bundle: flask_unchained.bundles.Bundle*) → Dict[str, Any]
Collect objects where *type_check()* returns `True` from bundles. Bundle subclasses can override objects discovered in superclass bundles.

key_name(*name: str*, *obj: Any*) → str
Override to use a custom key to determine uniqueness/overriding.

type_check(*obj: Any*) → bool
Implement to determine which objects in a module should be processed by this hook.

classmethod import_bundle_modules(*bundle: flask_unchained.bundles.Bundle*) → List[module]
Safe-import the modules in a bundle for this hook to load from.

classmethod get_module_names(*bundle: flask_unchained.bundles.Bundle*) → List[str]
The list of fully-qualified module names for a bundle this hook should load from.

classmethod get_bundle_module_names(*bundle: flask_unchained.bundles.Bundle*) → List[str]
The list of module names inside a bundle this hook should load from.

update_shell_context(*ctx: Dict[str, Any]*) → None
Implement to add objects to the CLI shell context.

7.1.4 AppBundle

class flask_unchained.**AppBundle**
Like *Bundle*, except used for the top-most application bundle.

name: str = 'app'
Name of the bundle. Defaults to the snake_cased class name, excluding any "Bundle" suffix.

7.1.5 Bundle

class flask_unchained.**Bundle**

Base class for bundles.

Should be placed in your package's root or its `bundle` module:

```
# your_bundle_package/__init__.py or your_bundle_package/bundle.py

class YourBundle(Bundle):
    pass
```

name: str = 'bundle'

Name of the bundle. Defaults to the snake_cased class name.

module_name: str = 'flask_unchained.bundles'

Top-level module name of the bundle (dot notation).

Automatically determined; read-only.

root_path: str = '/home/docs/checkouts/readthedocs.org/user_builds/flask-unchained/en

Root directory path of the bundle's package.

Automatically determined; read-only.

template_folder

Root directory path of the bundle's template folder. By default, if there exists a folder named `templates` in the bundle package *root_path*, it will be used, otherwise `None`.

static_folder

Root directory path of the bundle's static assets folder. By default, if there exists a folder named `static` in the bundle package *root_path*, it will be used, otherwise `None`.

static_url_path

Url path where this bundle's static assets will be served from. If *static_folder* is set, this will default to `/<bundle.name>/static`, otherwise `None`.

is_single_module: bool = False

Whether or not the bundle is a single module (Python file).

Automatically determined; read-only.

default_load_from_module_name: Optional[str] = None

The default module name for hooks to load from. Set hooks' bundle modules override attributes for the modules you want in separate files.

WARNING - EXPERIMENTAL

Using this feature may cause mysterious exceptions to be thrown!!

Best practice is to organize your code in separate modules.

---

before_init_app(*app: flask_unchained.flask_unchained.FlaskUnchained*) → None

Override this method to perform actions on the *FlaskUnchained* app instance *before* the `unchained` extension has initialized the application.

after_init_app(*app: flask_unchained.flask_unchained.FlaskUnchained*) → None

Override this method to perform actions on the *FlaskUnchained* app instance *after* the `unchained` extension has initialized the application.

7.1.6 BundleConfig

class flask_unchained.**BundleConfig**

Base class for configuration settings. Allows access to the app-under-construction as it's currently configured. Example usage:

```
# your_bundle_root/config.py

import os

from flask_unchained import BundleConfig

class Config(BundleConfig):
    SHOULD_PRETTY_PRINT_JSON = BundleConfig.current_app.config.DEBUG
```

7.1.7 FlaskUnchained

class flask_unchained.**FlaskUnchained**(*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*)

A simple subclass of `flask.Flask`. Overrides *`register_blueprint()`* and *`add_url_rule()`* to support automatic (optional) registration of URLs prefixed with a language code.

config_class

alias of `AttrDictFlaskConfig`

env: str = None

The environment the application is running in. Will be one of `development`, `production`, `staging`, or `test`.

unchained

The *`Unchained`* extension instance.

register_blueprint(*blueprint: flask.blueprints.Blueprint*, ***, *register_with_babel: bool = True*, ***options: Any*) → None

The same as `flask.Flask.register_blueprint()`, but if `register_with_babel` is True, then we also allow the Babel Bundle an opportunity to register language code prefixed URLs.

add_url_rule(*rule: str*, *endpoint: Optional[str] = None*, *view_func: Optional[function] = None*, *provide_automatic_options: Optional[bool] = None*, ***, *register_with_babel: bool = False*, ***options: Any*) → None

The same as `flask.Flask.add_url_rule()`, but if `register_with_babel` is True, then we also allow the Babel Bundle an opportunity to register a language code prefixed URL.

7.1.8 Service

class flask_unchained.**Service**
Base class for services. Automatically sets up dependency injection on the constructor of the subclass, and allows for your service to be automatically detected and used.

7.1.9 Hooks

ConfigureAppHook

class flask_unchained.hooks.**ConfigureAppHook**(*unchained: flask_unchained.unchained.Unchained, bundle: Optional[flask_unchained.bundles.Bundle] = None*)
Updates `app.config` with the settings from each bundle.

name: str = 'configure_app'
The name of this hook.

bundle_module_name: Optional[str] = 'config'
The default module this hook loads from.

Override by setting the `config_module_name` attribute on your bundle class.

run_hook(*app: flask_unchained.flask_unchained.FlaskUnchained, bundles: List[flask_unchained.bundles.Bundle], unchained_config: Optional[Dict[str, Any]] = None*) → None
For each bundle in `unchained_config.BUNDLES`, iterate through that bundle's class hierarchy, starting from the base-most bundle. For each bundle in that order, look for a `config` module, and if it exists, update `app.config` with the options first from a base `Config` class, if it exists, and then also if it exists, from an env-specific config class: one of `DevConfig`, `ProdConfig`, `StagingConfig`, or `TestConfig`.

get_bundle_config(*bundle: flask_unchained.bundles.Bundle, env: Union[development, production, staging, test]*) → flask.config.Config
Get the config settings from a bundle hierarchy.

InitExtensionsHook

class flask_unchained.hooks.**InitExtensionsHook**(*unchained: flask_unchained.unchained.Unchained, bundle: Optional[flask_unchained.bundles.Bundle] = None*)
Initializes extensions found in bundles with the current app.

name: str = 'init_extensions'
The name of this hook.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['extensions']
The default module this hook loads from.

Override by setting the `extensions_module_names` attribute on your bundle class.

process_objects(*app: flask_unchained.flask_unchained.FlaskUnchained, extensions: Dict[str, object]*) → None
Initialize each extension with `extension.init_app(app)`.

update_shell_context(*ctx: Dict[str, Any]*) → None
Add extensions to the CLI shell context.

RegisterCommandsHook

class flask_unchained.hooks.**RegisterCommandsHook**(*unchained: flask_unchained.unchained.Unchained*, *bundle: Optional[flask_unchained.bundles.Bundle] = None*)
Registers commands and command groups from bundles.

name: str = 'commands'
The name of this hook.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['commands']
The default module this hook loads from.

Override by setting the `commands_module_names` attribute on your bundle class.

run_hook(*app: flask_unchained.flask_unchained.FlaskUnchained*, *bundles: List[flask_unchained.bundles.Bundle]*, *unchained_config: Optional[Dict[str, Any]] = None*) → Dict[str, click.core.Command]
Discover CLI commands and command groups from bundles and register them with the app.

RegisterExtensionsHook

class flask_unchained.hooks.**RegisterExtensionsHook**(*unchained: flask_unchained.unchained.Unchained*, *bundle: Optional[flask_unchained.bundles.Bundle] = None*)
Registers extensions found in bundles with the `unchained` extension.

name: str = 'register_extensions'
The name of this hook.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['extensions']
The default module this hook loads from.

Override by setting the `extensions_module_names` attribute on your bundle class.

process_objects(*app: flask_unchained.flask_unchained.FlaskUnchained*, *extensions: Dict[str, object]*) → None
Discover extensions in bundles and register them with the Unchained extension.

collect_from_bundle(*bundle: flask_unchained.bundles.Bundle*) → Dict[str, object]
Collect declared extensions from a bundle hierarchy.

RegisterServicesHook

class flask_unchained.hooks.**RegisterServicesHook**(*unchained: flask_unchained.unchained.Unchained, bundle: Optional[flask_unchained.bundles.Bundle] = None*)

Registers services for dependency injection.

name: str = 'services'
The name of this hook.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['services', 'managers
The default modules this hook loads from.

Override by setting the `services_module_names` attribute on your bundle class.

process_objects(*app: flask_unchained.flask_unchained.FlaskUnchained, services: Dict[str, flask_unchained.di.Service]*) → None
Register services with the Unchained extension, initialize them, and inject any requested into extensions.

key_name(*name*, *obj*) → str
Returns the service's dependency injection name.

type_check(*obj*) → bool
Returns True if `obj` is a concrete subclass of *Service*.

update_shell_context(*ctx: Dict[str, Any]*) → None
Add services to the CLI shell context.

RunHooksHook

class flask_unchained.hooks.**RunHooksHook**(*unchained: flask_unchained.unchained.Unchained, bundle: Optional[flask_unchained.bundles.Bundle] = None*)

An internal hook to discover and run all the other hooks.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['hooks']
The default module this hook loads from.

Override by setting the `hooks_module_names` attribute on your bundle class.

run_hook(*app: flask_unchained.flask_unchained.FlaskUnchained, bundles: List[flask_unchained.bundles.Bundle], unchained_config: Optional[Dict[str, Any]] = None*) → None
Collect hooks from Flask Unchained and the list of bundles, resolve their correct order, and run them in that order to build (boot) the app instance.

collect_from_bundle(*bundle: flask_unchained.bundles.Bundle*) → Dict[str, flask_unchained.hooks.run_hooks_hook.HookTuple]
Collect hooks from a bundle hierarchy.

collect_unchained_hooks() → Dict[str, flask_unchained.hooks.run_hooks_hook.HookTuple]
Collect hooks built into Flask Unchained that should always run.

type_check(*obj: Any*) → bool
Returns True if `obj` is a subclass of *AppFactoryHook*.

ViewsHook

class flask_unchained.hooks.**ViewsHook**(*unchained: flask_unchained.unchained.Unchained*, *bundle: Optional[flask_unchained.bundles.Bundle] = None*)

Allows configuring bundle views modules.

name: str = 'views'
The name of this hook.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['views']
The default module this hook loads from.

Override by setting the `views_module_names` attribute on your bundle class.

run_hook(*app*, *bundles*, *unchained_config=None*) → None
Hook entry point. Override to disable standard behavior of iterating over bundles to discover objects and processing them.

7.1.10 Unchained

class flask_unchained.**Unchained**(*env: Union[development, production, staging, test, None] = None*)

The `Unchained` extension. Responsible for initializing the app by loading all the things from bundles, keeping references to all of the various discovered bundles and things inside them, and for doing dependency injection. To get access to the `unchained` extension instance:

```
from flask_unchained import unchained
```

Also acts as a replacement for some of the public API of `flask.Flask`. (The part that allows registering url rules, functions to run for handling errors, functions to run during the normal request response cycle, and methods for setting up the Jinja templating environment.)

get_local_proxy(*name*)
Returns a `LocalProxy` to the extension or service with `name` as registered with the current app.

service(*name: str = None*)
Decorator to mark something as a service.

register_service(*name: str*, *service: Any*)
Method to register a service.

inject(**args*)
Decorator to mark a class, method, or function as needing dependencies injected.

Example usage:

```
from flask_unchained import unchained, injectable

# automatically figure out which params to inject
@unchained.inject()
def my_function(not_injected, some_service: SomeService = injectable):
    # do stuff

# or declare injectables explicitly (makes the ``injectable`` default␣
↪optional)
@unchained.inject('some_service')
def my_function(not_injected, some_service: SomeService):
    # do stuff
```

(continues on next page)

(continued from previous page)

```
# use it on a class to set up class attributes injection (and the constructor)
@unchained.inject()
class MyClass:
    some_service: SomeService = injectable

    def __init__(self, another_service: AnotherService = injectable):
        self.another_service = another_service
```

add_url_rule(*rule*, *endpoint=None*, *view_func=None*, ***options*)
Register a new url rule. Acts the same as `flask.Flask.add_url_rule()`.

before_request(*fn=None*)
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_first_request(*fn=None*)
Registers a function to be run before the first request to this instance of the application.

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

after_request(*fn=None*)
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.

teardown_request(*fn=None*)
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_appcontext (*fn=None*)
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.

context_processor (*fn=None*)
Registers a template context processor function.

shell_context_processor (*fn=None*)
Registers a shell context processor function.

url_value_preprocessor (*fn=None*)
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_defaults (*fn=None*)
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.

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
```

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

template_filter(*arg: Optional[Callable] = None, *, name: Optional[str] = None, pass_context: bool = False, inject: Union[bool, Iterable[str], None] = None, safe: bool = False*) → Callable

Decorator to mark a function as a Jinja template filter.

Parameters

- **name** – The name of the filter, if different from the function name.
- **pass_context** – Whether or not to pass the template context into the filter. If `True`, the first argument must be the context.
- **inject** – Whether or not this filter needs any dependencies injected.
- **safe** – Whether or not to mark the output of this filter as html-safe.

template_global(*arg: Optional[Callable] = None, *, name: Optional[str] = None, pass_context: bool = False, inject: Union[bool, Iterable[str], None] = None, safe: bool = False*) → Callable

Decorator to mark a function as a Jinja template global (tag).

Parameters

- **name** – The name of the tag, if different from the function name.
- **pass_context** – Whether or not to pass the template context into the tag. If `True`, the first argument must be the context.
- **inject** – Whether or not this tag needs any dependencies injected.
- **safe** – Whether or not to mark the output of this tag as html-safe.

template_tag(*arg: Optional[Callable] = None, *, name: Optional[str] = None, pass_context: bool = False, inject: Union[bool, Iterable[str], None] = None, safe: bool = False*) → Callable

Alias for *template_global()*.

Parameters

- **name** – The name of the tag, if different from the function name.
- **pass_context** – Whether or not to pass the template context into the tag. If `True`, the first argument must be the context.
- **inject** – Whether or not this tag needs any dependencies injected.
- **safe** – Whether or not to mark the output of this tag as html-safe.

template_test(*arg: Optional[Callable] = None, *, name: Optional[str] = None, inject: Union[bool, Iterable[str], None] = None, safe: bool = False*) → Callable

Decorator to mark a function as a Jinja template test.

Parameters

- **name** – The name of the test, if different from the function name.
- **inject** – Whether or not this test needs any dependencies injected.

- **safe** – Whether or not to mark the output of this test as html-safe.

7.1.11 string_utils

flask_unchained.string_utils.**right_replace**(*string*, *old*, *new*, *count=1*)
Right replaces `count` occurrences of `old` with `new` in `string`. For example:

```
right_replace('one_two_two', 'two', 'three') -> 'one_two_three'
```

flask_unchained.string_utils.**slugify**(*string*)
Converts a string into a url-safe slug. For example:

```
slugify('Hello World') -> 'hello-world'
```

flask_unchained.string_utils.**pluralize**(*word*, *pos='NN'*, *custom=None*, *classical=True*)
Returns the plural of a given word, e.g., child => children. Handles nouns and adjectives, using classical inflection by default (i.e., where "matrix" pluralizes to "matrices" and not "matrixes"). The custom dictionary is for user-defined replacements.

flask_unchained.string_utils.**singularize**(*word*, *pos='NN'*, *custom=None*)
Returns the singular of a given word.

flask_unchained.string_utils.**camel_case**(*string*)
Converts a string to camel case. For example:

```
camel_case('one_two_three') -> 'oneTwoThree'
```

flask_unchained.string_utils.**class_case**(*string*)
Converts a string to class case. For example:

```
class_case('one_two_three') -> 'OneTwoThree'
```

flask_unchained.string_utils.**kebab_case**(*string*)
Converts a string to kebab case. For example:

```
kebab_case('one_two_three') -> 'one-two-three'
```

NOTE: To generate valid slugs, use *slugify()*

flask_unchained.string_utils.**snake_case**(*string*)
Converts a string to snake case. For example:

```
snake_case('OneTwoThree') -> 'one_two_three'
```

flask_unchained.string_utils.**title_case**(*string*)
Converts a string to title case. For example:

```
title_case('one_two_three') -> 'One Two Three'
```

7.1.12 utils

class flask_unchained.utils.**AttrDict**
A dictionary that allows using the dot operator to get and set keys.

class flask_unchained.utils.**ConfigProperty**(*key=None*)
Allows extension classes to create properties that proxy to the config value, eg `app.config[key]`.

If key is left unspecified, in will be injected by `ConfigPropertyMetaclass`, defaulting to `f'{ext_class_name}_{property_name}'.upper()`.

class flask_unchained.utils.**ConfigPropertyMetaclass**(*class_name*, *bases*, *clsdict*)
Use this metaclass to enable config properties on extension classes. I'm not sold on this being a good idea for *new* extensions, but for backwards compatibility with existing extensions that have silly `__getattr__` magic, I think it's a big improvement. (NOTE: this only works when the application context is available, but that's no different than the behavior of what it's meant to replace.)

Example usage:

```
class MyExtension(metaclass=ConfigPropertyMetaclass):
    __config_prefix__ = 'MY_EXTENSION'
    # if __config_prefix__ is unspecified, default is class_name.upper()

    foobar: Optional[FunctionType] = ConfigProperty()
    _custom: Optional[str] = ConfigProperty('MY_EXTENSION_CUSTOM')

my_extension = MyExtension(app)
my_extension.foobar == current_app.config.MY_EXTENSION_FOOBAR
my_extension._custom == current_app.config.MY_EXTENSION_CUSTOM
```

flask_unchained.utils.**cwd_import**(*module_name*)
Attempt to import a module from the current working directory.

Raises `ImportError` if not found, or the found module isn't from the current working directory.

flask_unchained.utils.**get_boolean_env**(*name*, *default*)
Converts environment variables to boolean values. Truthy is defined as: `value.lower() in {'true', 'yes', 'y', '1'}` (everything else is falsy).

flask_unchained.utils.**safe_import_module**(*module_name*)
Like `importlib.import_module()`, except it does not raise `ImportError` if the requested `module_name` is not found.

flask_unchained.utils.**utcnow**()
Returns a current timezone-aware `datetime.datetime` in UTC.

7.2 Admin Bundle API

flask_unchained.bundles.admin

AdminBundle	The Admin Bundle.

flask_unchained.bundles.admin.config

Config	Config class for the Admin bundle.

flask_unchained.bundles.admin.extensions

Admin	The *Admin* extension.

flask_unchained.bundles.admin.hooks

RegisterModelAdminsHook	Registers ModelAdmins with the *Admin* extension.

flask_unchained.bundles.admin.forms

ReorderableForm	Like BaseForm, except it supports re-ordering fields by setting the field_order class attribute to a list of field names.
EnumField	An extension of Select2Field, adding support for Enum.

flask_unchained.bundles.admin.macro

macro	Replaces macro(), adding support for using macros imported from another file.

flask_unchained.bundles.admin.model_admin

ModelAdmin	Base class for SQLAlchemy model admins.

flask_unchained.bundles.admin.views

AdminDashboardView	Default admin dashboard view.
AdminSecurityController	Extends *SecurityController*, to customize the template folder to use admin-specific templates.

7.2.1 AdminBundle

class flask_unchained.bundles.admin.**AdminBundle**

The Admin Bundle.

name: str = 'admin_bundle'

The name of the Admin Bundle.

after_init_app(*app: flask_unchained.flask_unchained.FlaskUnchained*) → None

Override this method to perform actions on the *FlaskUnchained* app instance *after* the unchained extension has initialized the application.

7.2.2 Config

class flask_unchained.bundles.admin.config.**Config**

Config class for the Admin bundle. Defines which configuration values this bundle supports, and their default values.

ADMIN_NAME = 'Admin'

The title of the admin section of the site.

ADMIN_BASE_URL = '/admin'

Base url of the admin section of the site.

ADMIN_INDEX_VIEW = <flask_unchained.bundles.admin.views.dashboard.AdminDashboardView ob

The `AdminIndexView` (or subclass) instance to use for the index view.

ADMIN_SUBDOMAIN = None

Subdomain of the admin section of the site.

ADMIN_BASE_TEMPLATE = 'admin/base.html'

Base template to use for other admin templates.

ADMIN_TEMPLATE_MODE = 'bootstrap4'

Which version of bootstrap to use. (bootstrap2, bootstrap3, or bootstrap4)

ADMIN_CATEGORY_ICON_CLASSES = {}

Dictionary of admin category icon classes. Keys are category names, and the values depend on which version of bootstrap you're using.

For example, with bootstrap4:

```
ADMIN_CATEGORY_ICON_CLASSES = {
    'Mail': 'fa fa-envelope',
    'Security': 'fa fa-lock',
}
```

ADMIN_ADMIN_ROLE_NAME = 'ROLE_ADMIN'

The name of the Role which represents an admin.

ADMIN_LOGIN_ENDPOINT = 'admin.login'

Name of the endpoint to use for the admin login view.

ADMIN_POST_LOGIN_REDIRECT_ENDPOINT = 'admin.index'

Name of the endpoint to redirect to after the user logs into the admin.

ADMIN_LOGOUT_ENDPOINT = 'admin.logout'

Name of the endpoint to use for the admin logout view.

ADMIN_POST_LOGOUT_REDIRECT_ENDPOINT = 'admin.login'

Endpoint to redirect to after the user logs out of the admin.

MAPBOX_MAP_ID = None

7.2.3 The Admin Extension

class flask_unchained.bundles.admin.**Admin**(*app=None*, *name=None*, *url=None*, *subdomain=None*, *index_view=None*, *translations_path=None*, *endpoint=None*, *static_url_path=None*, *base_template=None*, *template_mode=None*, *category_icon_classes=None*)

The *Admin* extension:

```
from flask_unchained.bundles.admin import admin
```

init_app(*app: flask_unchained.flask_unchained.FlaskUnchained*)
Register all views with the Flask application.

Parameters **app** – Flask application instance

7.2.4 RegisterModelAdminsHook

class flask_unchained.bundles.admin.hooks.**RegisterModelAdminsHook**(*unchained: flask_unchained.unchained.Unchain bundle: Optional[flask_unchained.bundles.Bun = None*)

Registers ModelAdmins with the *Admin* extension.

name: str = 'admins'
The name of this hook.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['admins']
The default module this hook loads from.

Override by setting the `admins_module_names` attribute on your bundle class.

process_objects(*app*, *objects*)
Register discovered ModelAdmins with the Admin extension.

type_check(*obj*)
Returns True if `obj` is a subclass of *ModelAdmin*.

7.2.5 Forms

class flask_unchained.bundles.admin.forms.**ReorderableForm**(*formdata=None*, *obj=None*, *prefix=''*, ***kwargs*)

Like `BaseForm`, except it supports re-ordering fields by setting the `field_order` class attribute to a list of field names.

class flask_unchained.bundles.admin.forms.**EnumField**(*column*, ***kwargs*)
An extension of `Select2Field`, adding support for `Enum`.

pre_validate(*form*)
Override if you need field-level validation. Runs before any other validators.

Parameters **form** – The form the field belongs to.

7.2.6 Macro

flask_unchained.bundles.admin.**macro**(*name*)

Replaces `macro()`, adding support for using macros imported from another file. For example:

```
{# templates/admin/column_formatters.html #}

{% macro email(model, column) %}
  {% set address = model[column] %}
  <a href="mailto:{{ address }}">{{ address }}</a>
{% endmacro %}
```

```
class FooAdmin(ModelAdmin):
    column_formatters = {
        'col_name': macro('column_formatters.email')
    }
```

Also required for this to work, is to add the following to the top of your master admin template:

```
{# templates/admin/master.html #}

{% import 'admin/column_formatters.html' as column_formatters with context %}
```

7.2.7 ModelAdmin

class flask_unchained.bundles.admin.**ModelAdmin**(*model*, *session*, *name=None*, *category=None*, *endpoint=None*, *url=None*, *static_folder=None*, *menu_class_name=None*, *menu_icon_type=None*, *menu_icon_value=None*)

Base class for SQLAlchemy model admins. More or less the same as `ModelView`, except we set some different defaults.

form_base_class
alias of *flask_unchained.bundles.admin.forms.ReorderableForm*

model_form_converter
alias of flask_unchained.bundles.admin.forms.AdminModelFormConverter

action_view()
Mass-model action view.

ajax_update()
Edits a single column of a record in list view.

create_view()
Create model view

delete_view()
Delete model view. Only POST method is allowed.

details_view()
Details model view

edit_view()
Edit model view

index_view()
List view

7.2.8 AdminDashboardView

class flask_unchained.bundles.admin.**AdminDashboardView**
Default admin dashboard view. Renders the `admin/dashboard.html` template.

is_visible()
Overridden to hide this view from the menu by default.

7.2.9 AdminSecurityController

class flask_unchained.bundles.admin.**AdminSecurityController**
Extends *SecurityController*, to customize the template folder to use admin-specific templates.

logout()
View function to log a user out. Supports html and json requests.

7.3 API Bundle API

flask_unchained.bundles.api

ApiBundle	The API Bundle.

flask_unchained.bundles.api.config

Config	Default config settings for the API Bundle.

flask_unchained.bundles.api.extensions

Api	The *Api* extension.
Marshmallow	The *Marshmallow* extension.

flask_unchained.bundles.api.hooks

RegisterModelResourcesHook	Registers ModelResources and configures ModelSerializers on them.
RegisterModelSerializersHook	Registers ModelSerializers.

flask_unchained.bundles.api.model_resource

ModelResource	Base class for model resources.

flask_unchained.bundles.api.model_serializer

ModelSerializer	Base class for SQLAlchemy model serializers.

7.3.1 ApiBundle

class flask_unchained.bundles.api.**ApiBundle**

The API Bundle.

name: str = 'api_bundle'

The name of the API Bundle.

resources_by_model = None

Lookup of resource classes by class name.

serializers = None

Lookup of serializer classes by class name.

serializers_by_model = None

Lookup of serializer classes by model class name

create_by_model = None

Lookup of serializer classes by model class name, as set by `@ma.serializer(create=True)` (see `serializer()`)

many_by_model = None

Lookup of serializer classes by model class name, as set by `@ma.serializer(many=True)` (see `serializer()`)

after_init_app(*app: flask_unchained.flask_unchained.FlaskUnchained*)

Configure the JSON encoder for Flask to be able to serialize Enums, LocalProxy objects, and SQLAlchemy models.

7.3.2 Config

class flask_unchained.bundles.api.config.**Config**

Default config settings for the API Bundle.

API_OPENAPI_VERSION = '2.0'

API_REDOC_SOURCE_URL = 'https://cdn.jsdelivr.net/npm/redoc@next/bundles/redoc.standalo

API_REDOC_URL_PREFIX = '/api-docs'

API_REDOC_PATH = '/'

API_OPENAPI_JSON_PATH = 'openapi.json'

API_TITLE = None

API_VERSION = 1

API_DESCRIPTION = None

API_APISPEC_PLUGINS = None

DUMP_KEY_FN()

An optional function to use for converting keys when dumping data to send over the wire. By default, we convert snake_case to camelCase.

LOAD_KEY_FN()

An optional function to use for converting keys received over the wire to the backend's representation. By default, we convert camelCase to snake_case.

ACCEPT_HANDLERS = {'application/json': <function jsonify>}

Functions to use for converting response data for Accept headers.

7.3.3 Extensions

Api

class flask_unchained.bundles.api.**Api**

The *Api* extension:

```
from flask_unchained.bundles.api import api
```

Allows interfacing with apispec.

register_serializer(*serializer*, *name=None*, ***kwargs*)

Method to manually register a `Serializer` with APISpec.

Parameters

- **serializer** –
- **name** –
- **kwargs** –

register_model_resource(*resource: flask_unchained.bundles.api.model_resource.ModelResource*)

Method to manually register a *ModelResource* with APISpec.

Parameters **resource** –

register_converter(*converter*, *conv_type*, *conv_format=None*, *, *name=None*)

Register custom path parameter converter.

Parameters

- **converter** (*BaseConverter*) – Converter Subclass of werkzeug's BaseConverter
- **conv_type** (*str*) – Parameter type
- **conv_format** (*str*) – Parameter format (optional)
- **name** (*str*) – Name of the converter. If not None, this name is used to register the converter in the Flask app. Example:

```
api.register_converter(
    UUIDConverter, 'string', 'UUID', name='uuid')
@blp.route('/pets/{uuid:pet_id}')
    # ...
api.register_blueprint(blp)
```

This registers the converter in the Flask app and in the internal APISpec instance.

Once the converter is registered, all paths using it will have corresponding path parameter documented with the right type and format. The *name* parameter need not be passed if the converter is already registered in the app, for instance if it belongs to a Flask extension that already registers it in the app.

register_field(*field*, **args*)

Register custom Marshmallow field.

Registering the Field class allows the Schema parser to set the proper type and format when documenting parameters from Schema fields.

Parameters **field** (*Field*) – Marshmallow Field class

Param args: - a pair of the form `(type, format)` to map to - a core marshmallow field type (then that type's mapping is used)

Examples:

```
# Map to ('string', 'UUID')
api.register_field(UUIDField, 'string', 'UUID')
# Map to ('string')
api.register_field(URLField, 'string', None)
# Map to ('integer, 'int32')
api.register_field(CustomIntegerField, ma.fields.Integer)
```

Marshmallow

class flask_unchained.bundles.api.**Marshmallow**

The *Marshmallow* extension:

```
from flask_unchained.bundles.api import ma
```

Allows decorating a *ModelSerializer* with *serializer()* to specify it should be used for creating objects, listing them, or as the fallback.

Also provides aliases from the following modules:

flask_unchained.bundles.api

ModelSerializer(*args, **kwargs)	Base class for SQLAlchemy model serializers.

marshmallow.decorators

pre_load([fn, pass_many])	Register a method to invoke before deserializing an object.
post_load([fn, pass_many, pass_original])	Register a method to invoke after deserializing an object.
pre_dump([fn, pass_many])	Register a method to invoke before serializing an object.
post_dump([fn, pass_many, pass_original])	Register a method to invoke after serializing an object.
validates(field_name)	Register a field validator.
validates_schema([fn, pass_many, ...])	Register a schema-level validator.

marshmallow.exceptions

ValidationError(message[, field_name, data, ...])	Raised when validation fails on a field or schema.

marshmallow.fields

Bool	alias of marshmallow.fields.Boolean
Boolean(*[, truthy, falsy])	A boolean field.
Constant(constant, **kwargs)	A field that (de)serializes to a preset constant.
Date([format])	ISO8601-formatted date string.
DateTime([format])	A formatted datetime string.
NaiveDateTime([format, timezone])	A formatted naive datetime string.

Continued on next page

Table 30 – continued from previous page

AwareDateTime([format, default_timezone])	A formatted aware datetime string.
Decimal([places, rounding, allow_nan, as_string])	A field that (de)serializes to the Python decimal.Decimal type.
Dict([keys, values])	A dict field.
Email(*args, **kwargs)	A validated email field.
Field(*[, default, missing, data_key, ...])	Basic field from which other fields should extend.
Float(*[, allow_nan, as_string])	A double as an IEEE-754 double precision string.
Function([serialize, deserialize])	A field that takes the value returned by a function.
Int	alias of marshmallow.fields.Integer
Integer(*[, strict])	An integer field.
List(cls_or_instance, **kwargs)	A list field, composed with another *Field* class or instance.
Mapping([keys, values])	An abstract class for objects with key-value pairs.
Method([serialize, deserialize])	A field that takes the value returned by a *Schema* method.
Nested(nested, *[, default, only, exclude, ...])	Allows you to nest a Schema inside a field.
Number(*[, as_string])	Base class for number fields.
Pluck(nested, field_name, **kwargs)	Allows you to replace nested data with one of the data's fields.
Raw(*[, default, missing, data_key, ...])	Field that applies no formatting.
Str	alias of marshmallow.fields.String
String(*[, default, missing, data_key, ...])	A string field.
Time([format])	A formatted time string.
TimeDelta([precision])	A field that (de)serializes a datetime.timedelta object to an integer and vice versa.
Tuple(tuple_fields, *args, **kwargs)	A tuple field, composed of a fixed number of other *Field* classes or instances
UUID(*[, default, missing, data_key, ...])	A UUID field.
Url(*[, relative, schemes, require_tld])	A validated URL field.
URL	alias of marshmallow.fields.Url

flask_marshmallow.fields

AbsoluteUrlFor	alias of flask_marshmallow.fields.AbsoluteURLFor
AbsoluteURLFor(endpoint[, values])	Field that outputs the absolute URL for an endpoint.
UrlFor	alias of flask_marshmallow.fields.URLFor
URLFor(endpoint[, values])	Field that outputs the URL for an endpoint.
Hyperlinks(schema, **kwargs)	Field that outputs a dictionary of hyperlinks, given a dictionary schema with URLFor objects as values.

flask_marshmallow.sqla

HyperlinkRelated(endpoint[, url_key, external])	Field that generates hyperlinks to indicate references between models, rather than primary keys.

serializer(*create=False*, *many=False*)

Decorator to mark a Serializer subclass for a specific purpose, ie, to be used during object creation **or** for serializing lists of objects.

Parameters

- **create** – Whether or not this serializer is for object creation.
- **many** – Whether or not this serializer is for lists of objects.

7.3.4 Hooks

RegisterModelResourcesHook

class flask_unchained.bundles.api.hooks.**RegisterModelResourcesHook**(*unchained: flask_unchained.unchained.Unchai bundle: Optional[flask_unchained.bundles.Bu = None*)

Registers ModelResources and configures ModelSerializers on them.

name: str = 'model_resources'
The name of this hook.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['views']
The default module this hook loads from.

Override by setting the `model_resources_module_names` attribute on your bundle class.

process_objects(*app*, *objects*)
Configures ModelSerializers on ModelResources.

type_check(*obj*)
Returns True if `obj` is a subclass of *ModelResource*.

RegisterModelSerializersHook

class flask_unchained.bundles.api.hooks.**RegisterModelSerializersHook**(*unchained: flask_unchained.unchained.Unc bundle: Optional[flask_unchained.bundles. = None*)

Registers ModelSerializers.

name: str = 'model_serializers'
The name of this hook.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['serializers']
The default module this hook loads from.

Override by setting the `model_serializers_module_names` attribute on your bundle class.

process_objects(*app: flask_unchained.flask_unchained.FlaskUnchained, serializers: Dict[str, Type[flask_unchained.bundles.api.model_serializer.ModelSerializer]]*) → None
Registers model serializers onto the API Bundle instance.

type_check(*obj: Any*) → bool
Returns True if `obj` is a subclass of *ModelSerializer*.

update_shell_context(*ctx: Dict[str, Any]*) → None
Adds model serializers to the CLI shell context.

7.3.5 ModelResource

class flask_unchained.bundles.api.**ModelResource**
Base class for model resources. This is intended for building RESTful APIs with SQLAlchemy models and Marshmallow serializers.

list(*instances*)
List model instances.

Parameters **instances** – The list of model instances.

Returns The list of model instances.

create(*instance*, *errors*)
Create an instance of a model.

Parameters

- **instance** – The created model instance.
- **errors** – Any errors.

Returns The created model instance, or a dictionary of errors.

get(*instance*)
Get an instance of a model.

Parameters **instance** – The model instance.

Returns The model instance.

patch(*instance*, *errors*)
Partially update a model instance.

Parameters

- **instance** – The model instance.
- **errors** – Any errors.

Returns The updated model instance, or a dictionary of errors.

put(*instance*, *errors*)
Update a model instance.

Parameters

- **instance** – The model instance.
- **errors** – Any errors.

Returns The updated model instance, or a dictionary of errors.

delete(*instance*)
Delete a model instance.

Parameters **instance** – The model instance.

Returns HTTPStatus.NO_CONTENT

created(*instance*, *commit=True*)
Convenience method for saving a model (automatically commits it to the database and returns the object with an HTTP 201 status code)

deleted(*instance*)
Convenience method for deleting a model (automatically commits the delete to the database and returns with an HTTP 204 status code)

updated(*instance*)
Convenience method for updating a model (automatically commits it to the database and returns the object with with an HTTP 200 status code)

7.3.6 ModelSerializer

class flask_unchained.bundles.api.**ModelSerializer**(**args*, ***kwargs*)
Base class for SQLAlchemy model serializers. This is pretty much a stock `flask_marshmallow.sqla.ModelSchema`, except:

- dependency injection is set up automatically on ModelSerializer
- when loading to update an existing instance, validate the primary keys are the same
- automatically make fields named `slug`, `model.Meta.created_at`, and `model.Meta.updated_at` dump-only

For example:

```
from flask_unchained.bundles.api import ModelSerializer
from flask_unchained.bundles.security.models import Role

class RoleSerializer(ModelSerializer):
    class Meta:
        model = Role
```

Is roughly equivalent to:

```
from marshmallow import Schema, fields

class RoleSerializer(Schema):
    id = fields.Integer(dump_only=True)
    name = fields.String()
    description = fields.String()
    created_at = fields.DateTime(dump_only=True)
    updated_at = fields.DateTime(dump_only=True)
```

OPTIONS_CLASS
alias of `ModelSerializerOptionsClass`

is_create()
Check if we're creating a new object. Note that this context flag must be set from the outside, ie when the class gets instantiated.

load(*data: Mapping, *, many: bool = None, partial: Union[bool, Sequence[str], Set[str]] = None, unknown: str = None, **kwargs*)
Deserialize a dict to an object defined by this ModelSerializer's fields.

A `ValidationError` is raised if invalid data is passed.

Parameters

- **data** – The data to deserialize.
- **many** – Whether to deserialize *data* as a collection. If *None*, the value for *self.many* is used.

- **partial** – Whether to ignore missing fields and not require any fields declared. Propagates down to `Nested` fields as well. If its value is an iterable, only missing fields listed in that iterable will be ignored. Use dot delimiters to specify nested fields.
- **unknown** – Whether to exclude, include, or raise an error for unknown fields in the data. Use *EXCLUDE*, *INCLUDE* or *RAISE*. If *None*, the value for *self.unknown* is used.

Returns Deserialized data

dump(*obj*, *, *many: bool = None*)

Serialize an object to native Python data types according to this ModelSerializer's fields.

Parameters

- **obj** – The object to serialize.
- **many** – Whether to serialize *obj* as a collection. If *None*, the value for *self.many* is used.

Returns A dict of serialized data

Return type dict

handle_error(*error: marshmallow.exceptions.ValidationError*, *data: Any*, ***kwargs*) → None

Customize the error messages for required/not-null validators with dynamically generated field names. This is definitely a little hacky (it mutates state, uses hardcoded strings), but unsure how better to do it

7.4 Babel Bundle API

flask_unchained.bundles.babel

BabelBundle	The Babel Bundle.

gettext(*args, **kwargs)	Return the localized translation of message, based on the language, and locale directory of the domain specified in the translation key (or the current global domain).
ngettext(*args, **kwargs)	Like `gettext()`, except it supports pluralization.
lazy_gettext(*args, **kwargs)	Like `gettext()`, except lazy.
lazy_ngettext(*args, **kwargs)	Like `ngettext()`, except lazy.

flask_unchained.bundles.babel.config

Config	Default configuration options for the Babel Bundle.
DevConfig	
ProdConfig	
StagingConfig	

7.4.1 BabelBundle

class flask_unchained.bundles.babel.**BabelBundle**

The Babel Bundle. Responsible for configuring the correct gettext callables with Jinja, as well as optionally registering endpoints for language-specific URLs (if enabled).

name: str = 'babel_bundle'

The name of the Babel Bundle.

command_group_names = ('babel',)

Names of the command groups included in this bundle.

language_code_key = 'lang_code'

Default Werkzeug parameter name to be used when registering language-specific URLs.

before_init_app(*app: flask_unchained.flask_unchained.FlaskUnchained*)

Override this method to perform actions on the *FlaskUnchained* app instance *before* the unchained extension has initialized the application.

after_init_app(*app: flask_unchained.flask_unchained.FlaskUnchained*)

Override this method to perform actions on the *FlaskUnchained* app instance *after* the unchained extension has initialized the application.

7.4.2 gettext functions

flask_unchained.**gettext**(**args*, ***kwargs*)

Return the localized translation of message, based on the language, and locale directory of the domain specified in the translation key (or the current global domain). This function is usually aliased as _:

```
from flask_unchained import gettext as _
```

flask_unchained.**ngettext**(**args*, ***kwargs*)

Like *gettext()*, except it supports pluralization. This function is usually aliased as _:

```
from flask_unchained import ngettext as _
```

flask_unchained.**lazy_gettext**(**args*, ***kwargs*)

Like *gettext()*, except lazy. This function is usually aliased as _:

```
from flask_unchained import lazy_gettext as _
```

flask_unchained.**lazy_ngettext**(**args*, ***kwargs*)

Like *ngettext()*, except lazy. This function is usually aliased as _:

```
from flask_unchained import lazy_ngettext as _
```

7.4.3 Config

class flask_unchained.bundles.babel.config.**Config**
Default configuration options for the Babel Bundle.

LANGUAGES = ['en']
The language codes supported by the app.

BABEL_DEFAULT_LOCALE = 'en'
The default language to use if none is specified by the client's browser.

BABEL_DEFAULT_TIMEZONE = 'UTC'
The default timezone to use.

DEFAULT_DOMAIN = <flask_babelex.Domain object>
The default `Domain` to use.

DATE_FORMATS = {'date': 'medium', 'date.full': None, 'date.long': None, 'date.mediu
A dictionary of date formats.

ENABLE_URL_LANG_CODE_PREFIX = False
Whether or not to enable the capability to specify the language code as part of the URL.

class flask_unchained.bundles.babel.config.**DevConfig**

LAZY_TRANSLATIONS = False
Do not use lazy translations in development.

class flask_unchained.bundles.babel.config.**ProdConfig**

LAZY_TRANSLATIONS = True
Use lazy translations in production.

class flask_unchained.bundles.babel.config.**StagingConfig**

LAZY_TRANSLATIONS = True
Use lazy translations in staging.

7.5 Celery Bundle API

flask_unchained.bundles.celery

CeleryBundle	The Celery Bundle.

flask_unchained.bundles.celery.config

Config	Default configuration options for the Celery Bundle.

flask_unchained.bundles.celery.extensions

Celery	The *Celery* extension.

flask_unchained.bundles.celery.hooks

DiscoverTasksHook	Discovers celery tasks.

7.5.1 CeleryBundle

class flask_unchained.bundles.celery.**CeleryBundle**

The Celery Bundle.

name: str = 'celery_bundle'

The name of the Celery Bundle.

command_group_names = ['celery']

Click groups for the Celery Bundle.

7.5.2 Config

class flask_unchained.bundles.celery.config.**Config**

Default configuration options for the Celery Bundle.

CELERY_BROKER_URL = 'redis://127.0.0.1:6379/0'

The broker URL to connect to.

CELERY_RESULT_BACKEND = 'redis://127.0.0.1:6379/0'

The result backend URL to connect to.

CELERY_ACCEPT_CONTENT = ('json', 'pickle', 'dill')

Tuple of supported serialization strategies.

MAIL_SEND_FN(*to=None*, *template=None*, ***kwargs*)

If the celery bundle is listed *after* the mail bundle in `unchained_config.BUNDLES`, then this configures the mail bundle to send emails asynchronously.

7.5.3 The Celery Extension

class flask_unchained.bundles.celery.**Celery**(**args*, ***kwargs*)

The *Celery* extension:

```
from flask_unchained.bundles.celery import celery
```

task(**args*, ***opts*)

Decorator to create a task class out of any callable.

See Task options for a list of the arguments that can be passed to this decorator.

Examples:

```
@app.task
def refresh_feed(url):
    store_feed(feedparser.parse(url))
```

with setting extra options:

```
@app.task(exchange='feeds')
def refresh_feed(url):
    return store_feed(feedparser.parse(url))
```

Note: App Binding: For custom apps the task decorator will return a proxy object, so that the act of creating the task is not performed until the task is used or the task registry is accessed.

If you're depending on binding to be deferred, then you must not access any attributes on the returned object until the application is fully set up (finalized).

7.5.4 DiscoverTasksHook

class flask_unchained.bundles.celery.hooks.**DiscoverTasksHook**(*unchained: flask_unchained.unchained.Unchained, bundle: Optional[flask_unchained.bundles.Bundle] = None*)

Discovers celery tasks.

name: str = 'celery_tasks'
The name of this hook.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['tasks']
The default module this hook loads from.

Override by setting the `celery_tasks_module_names` attribute on your bundle class.

process_objects(*app: flask_unchained.flask_unchained.FlaskUnchained, objects: Dict[str, Any]*)
Implement to do stuff with discovered objects (eg, registering them with the app instance).

type_check(*obj: Any*)
Implement to determine which objects in a module should be processed by this hook.

7.6 Controller Bundle API

flask_unchained.bundles.controller

ControllerBundle	The Controller Bundle.

flask_unchained.bundles.controller.config

Config	Default configuration options for the controller bundle.

flask_unchained.bundles.controller.controller

Controller	Base class for views.

flask_unchained.bundles.controller.decorators

param_converter	Convert arguments from the URL and/or query string.
route	Decorator to set default route rules for a view function.
no_route	Decorator to mark a `Controller` or `Resource` method as *not* a route.

flask_unchained.bundles.controller.hooks

RegisterBlueprintsHook	Registers legacy Flask blueprints with the app.
RegisterBundleBlueprintsHook	Registers a bundle blueprint for each bundle with views and/or template/static folders.
RegisterRoutesHook	Registers routes.

flask_unchained.bundles.controller.resource

Resource	Base class for resources.

flask_unchained.bundles.controller.route

Route	This is a semi-private class that you most likely shouldn't use directly.

flask_unchained.bundles.controller.routes

controller	This function is used to register a controller class's routes.
resource	This function is used to register a `Resource`'s routes.
func	This function allows to register legacy view functions as routes, eg.
include	Include the routes from another module at that point in the tree.
prefix	Sets a prefix on all of the child routes passed to it.
rule	Used to specify customizations to the route settings of class-based view function.
get	Like `rule()`, except specifically for HTTP GET requests.
patch	Like `rule()`, except specifically for HTTP PATCH requests.
post	Like `rule()`, except specifically for HTTP POST requests.
put	Like `rule()`, except specifically for HTTP PUT requests.
delete	Like `rule()`, except specifically for HTTP DELETE requests.

flask_unchained.bundles.controller.utils

redirect	An improved version of flask's redirect function
url_for	An improved version of flask's url_for function
abort	Raises an `HTTPException` for the given status code or WSGI application.
generate_csrf	Generate a CSRF token.

7.6.1 ControllerBundle

class flask_unchained.bundles.controller.**ControllerBundle**
The Controller Bundle.

name: str = 'controller_bundle'
The name of the Controller Bundle.

endpoints: Dict[str, List[Route]] = None
Lookup of routes by endpoint name.

controller_endpoints: Dict[str, List[Route]] = None
Lookup of routes by keys: f'{ControllerClassName}.{view_method_name}'

bundle_routes: Dict[str, List[Route]] = None
Lookup of routes belonging to each bundle by bundle name.

other_routes: List[Route] = None
List of routes not associated with any bundles.

before_init_app(*app: flask_unchained.flask_unchained.FlaskUnchained*)
Configure the Jinja environment and template loader.

7.6.2 Config

class flask_unchained.bundles.controller.config.**Config**
Default configuration options for the controller bundle.

FLASH_MESSAGES = True
Whether or not to enable flash messages.

NOTE: This only works for messages flashed using the *flask_unchained.Controller.flash()* method; using the `flask.flash()` function directly will not respect this setting.

TEMPLATE_FILE_EXTENSION = '.html'
The default file extension to use for templates.

WTF_CSRF_ENABLED = False
Whether or not to enable CSRF protection.

7.6.3 Views

Controller

class flask_unchained.**Controller**
Base class for views.

Concrete controllers should subclass this and their public methods will used as the views. By default view methods will be assigned routing defaults with the HTTP method GET and paths as the kebab-cased method name. For example:

```
from flask_unchained import Controller, injectable, route, no_route
from flask_unchained.bundles.sqlalchemy import SessionManager

class SiteController(Controller):
    class Meta:
        abstract = False  # this is the default; no need to set explicitly
```

(continues on next page)

(continued from previous page)

```
        decorators = ()  # a list of decorators to apply to all view methods
                         # on the controller (defaults to an empty tuple)
        template_folder = 'site'  # defaults to the snake_cased class name,
                                  # minus any Controller/View suffix
        template_file_extension = app.config.TEMPLATE_FILE_EXTENSION = '.html'
        url_prefix = None  # optional url prefix to use for all routes
        endpoint_prefix = 'site_controller'  # defaults to snake_cased class name

    # dependency injection works automatically on controllers
    session_manager: SessionManager = injectable

    @route('/')  # change the default path of '/index' to '/'
    def index():
        return self.render('index')  # site/index.html

    # use the defaults, equivalent to @route('/about-us', methods=['GET'])
    def about_us():
        return self.render('about_us.html')  # site/about_us.html

    # change the path, HTTP methods, and the endpoint
    @route('/contact', methods=['GET', 'POST'], endpoint='site_controller.contact
↪')
    def contact_us():
        # ...
        return self.render('site/contact.html.j2')  # site/contact.html.j2

    @no_route
    def public_utility_method():
        return 'not a view'

    def _protected_utility_method():
        return 'not a view'
```

How do the calls to render know which template to use? They look in `Bundle.template_folder` for a folder with the controller's `Meta.template_folder` and a file with the passed name and `Meta.template_file_extension`. For example:

```
class SiteController(Controller):
    # these defaults are automatically determined, unless you override them
    class Meta:
        template_folder = 'site'  # snake_cased class name (minus Controller␣
↪suffix)
        template_file_extension = '.html'  # from Config.TEMPLATE_FILE_EXTENSION

    def about_us():
        return self.render('about_us')  # site/about_us.html

    def contact():
        return self.render('contact')  # site/contact.html

    def index():
        return self.render('index')  # site/index.html

# your_bundle_root
├── __init__.py
├── templates
```

(continues on next page)

(continued from previous page)

```
│   └── site
│       ├── about_us.html
│       ├── contact.html
│       └── index.html
└── views
    └── site_controller.py
```

flash(*msg: str*, *category: Optional[str] = None*)
Convenience method for flashing messages.

Parameters

- **msg** – The message to flash.
- **category** – The category of the message.

render(*template_name: str*, ***ctx*)
Convenience method for rendering a template.

Parameters

- **template_name** – The template's name. Can either be a full path, or a filename in the controller's template folder. (The file extension can be omitted.)
- **ctx** – Context variables to pass into the template.

redirect(*where: Optional[str] = None*, *default: Optional[str] = None*, *override: Optional[str] = None*, ***url_kwargs*)
Convenience method for returning redirect responses.

Parameters

- **where** – A method name from this controller, a URL, an endpoint, or a config key name to redirect to.
- **default** – A method name from this controller, a URL, an endpoint, or a config key name to redirect to if `where` is invalid.
- **override** – Explicitly redirect to a method name from this controller, a URL, an endpoint, or a config key name (takes precedence over the `next` value in query strings or forms)
- **url_kwargs** – the variable arguments of the URL rule
- **_anchor** – if provided this is added as anchor to the URL.
- **_external** – if set to `True`, an absolute URL is generated. Server address can be changed via `SERVER_NAME` configuration variable which defaults to *localhost*.
- **_external_host** – if specified, the host of an external server to generate urls for (eg https://example.com or localhost:8888)
- **_method** – if provided this explicitly specifies an HTTP method.
- **_scheme** – a string specifying the desired URL scheme. The *_external* parameter must be set to `True` or a `ValueError` is raised. The default behavior uses the same scheme as the current request, or `PREFERRED_URL_SCHEME` from the app configuration if no request context is available. As of Werkzeug 0.10, this also can be set to an empty string to build protocol-relative URLs.

jsonify(*data: Any*, *code: Union[int, Tuple[int, str, str]] = <HTTPStatus.OK: 200>*, *headers: Optional[Dict[str, str]] = None*)
Convenience method to return json responses.

Parameters

- **data** – The python data to jsonify.
- **code** – The HTTP status code to return.
- **headers** – Any optional headers.

errors(*errors: List[str], code: Union[int, Tuple[int, str, str]] = <HTTPStatus.BAD_REQUEST: 400>, key: str = 'errors', headers: Optional[Dict[str, str]] = None*)
Convenience method to return errors as json.

Parameters

- **errors** – The list of errors.
- **code** – The HTTP status code.
- **key** – The key to return the errors under.
- **headers** – Any optional headers.

after_this_request(*fn*)
Register a function to run after this request.

Parameters **fn** – The function to run. It should accept one argument, the response, which it should also return

Resource

class flask_unchained.**Resource**
Base class for resources. This is intended for building RESTful APIs. Following the rules shown below, if the given class method is defined, this class will automatically set up the shown routing rule for it.

HTTP Method	Resource class method name	URL Rule
GET	list	/
POST	create	/
GET	get	/<cls.Meta.member_param>
PATCH	patch	/<cls.Meta.member_param>
PUT	put	/<cls.Meta.member_param>
DELETE	delete	/<cls.Meta.member_param>

So, for example:

```
from flask_unchained import Resource, injectable, param_converter
from flask_unchained.bundles.security import User, UserManager


class UserResource(Resource):
    class Meta:
        member_param: '<string:username>'

    user_manager: UserManager = injectable

    def list():
        return self.jsonify(dict(users=self.user_manager.all()))
        # NOTE: returning SQLAlchemy models directly like this is
        # only supported by ModelResource from the API Bundle
```

(continues on next page)

(continued from previous page)

```
    def create():
        user = self.user_manager.create(**data, commit=True)
        return self.jsonify(dict(user=user), code=201)

    @param_converter(username=User)
    def get(user):
        return self.jsonify(dict(user=user)

    @param_converter(username=User)
    def patch(user):
        user = self.user_manager.update(user, **data, commit=True)
        return self.jsonify(dict(user=user))

    @param_converter(username=User)
    def put(user):
        user = self.user_manager.update(user, **data, commit=True)
        return self.jsonify(dict(user=user))

    @param_converter(username=User)
    def delete(user):
        self.user_manager.delete(user, commit=True)
        return self.make_response('', code=204)
```

Registered like so:

```
routes = lambda: [
    resource('/users', UserResource),
]
```

Would register the following routes:

```
GET     /users                      UserResource.list
POST    /users                      UserResource.create
GET     /users/<string:username>    UserResource.get
PATCH   /users/<string:username>    UserResource.patch
PUT     /users/<string:username>    UserResource.put
DELETE  /users/<string:username>    UserResource.delete
```

See also `ModelResource` from the API bundle.

7.6.4 View Decorators

param_converter

`flask_unchained.`**`param_converter`**(**decorator_args*, ***decorator_kwargs*)

Convert arguments from the URL and/or query string.

For parsing arguments from the query string, pass their names as keyword argument keys where the value is a lookup (dict, Enum) or callable used to convert the query string argument's value:

```
@route('/users/<int:id>')
@param_converter(id=User, foo=str, optional=int)
def show_user(user, foo, optional=10):
```

(continues on next page)

(continued from previous page)

```
    # GET /users/1?foo=bar
    # calls show_user(user=User.query.get(1), foo='bar')
```

It also supports loading SQLAlchemy models from the database. Call with the url parameter names as keyword argument keys, their values being the model class to convert to.

Models will be looked up by the url param names. If a url param name is prefixed with the snake_cased model name, the prefix will be stripped. If a model isn't found, abort with a 404.

The view function's argument names must match the snake_cased model names.

For example:

```
@route('/users/<int:user_id>/posts/<int:id>')
@param_converter(user_id=User, id=Post)
def show_post(user, post):
    # the param converter does the database lookups:
    # user = User.query.get(id=user_id)
    # post = Post.query.get(id=id)
    # and calls the decorated view: show_post(user, post)

# or to customize the argument names passed to the view:
@route('/users/<int:user_id>/posts/<int:post_id>')
@param_converter(user_id={'user_arg_name': User},
                 post_id={'post_arg_name': Post})
def show_post(user_arg_name, post_arg_name):
    # do stuff
```

route

flask_unchained.**route**(*rule=None*, *blueprint=None*, *defaults=None*, *endpoint=None*, *is_member=False*, *methods=None*, *only_if=<py_meta_utils._missing_cls object>*, ***rule_options*)

Decorator to set default route rules for a view function. The arguments this function accepts are very similar to Flask's `route()`, however, the `is_member` perhaps deserves an example:

```
class UserResource(ModelResource):
    class Meta:
        model = User
        member_param = '<int:id>'
        include_methods = ['list', 'get']

    @route(is_member=True, methods=['POST'])
    def set_profile_pic(user):
        # do stuff

# registered like so in your ``app_bundle/routes.py``:
routes = lambda: [
    resource(UserResource),
]

# results in the following routes:
# UserResource.list             => GET  /users
# UserResource.get              => GET  /users/<int:id>
# UserResource.set_profile_pic  => POST /users/<int:id>/set-profile-pic
```

Parameters

- **rule** – The URL rule.
- **defaults** – Any default values for parameters in the URL rule.
- **endpoint** – The endpoint name of this view. Determined automatically if left unspecified.
- **is_member** – Whether or not this view is for a `Resource` member method.
- **methods** – A list of HTTP methods supported by this view. Defaults to `['GET']`.
- **only_if** – A boolean or callable to dynamically determine whether or not to register this route with the app.
- **rule_options** – Other kwargs passed on to `Rule`.

no_route

`flask_unchained.`**no_route**(*arg=None*)

Decorator to mark a `Controller` or `Resource` method as *not* a route. For example:

```
class SiteController(Controller):
    @route('/')
    def index():
        return self.render('index')

    def about():
        return self.render('about', stuff=self.utility_method())

    @no_route
    def utility_method():
        return 'stuff'

# registered like so in ``your_app_bundle/routes.py``
routes = lambda: [
    controller(SiteController),
]

# results in the following routes
SiteController.index            => GET /
SiteController.about            => GET /about

# but without the @no_route decorator, it would have also added this route:
SiteController.utility_method   => GET /utility-method
```

NOTE: The perhaps more Pythonic way to accomplish this is to make all non-route methods protected by prefixing them with an underscore, eg `_utility_method`.

7.6.5 Declarative Routing

controller

flask_unchained.**controller**(*url_prefix_or_controller_cls: Union[str, Type[flask_unchained.bundles.controller.controller.Controller]], controller_cls: Optional[Type[flask_unchained.bundles.controller.controller.Controller]] = None, *, rules: Optional[Iterable[Union[flask_unchained.bundles.controller.route.Route, Iterable[flask_unchained.bundles.controller.route.Route]]]] = None*) → Iterable[flask_unchained.bundles.controller.route.Route]

This function is used to register a controller class's routes.

Example usage:

```
routes = lambda: [
    controller(SiteController),
]
```

Or with the optional prefix argument:

```
routes = lambda: [
    controller('/products', ProductController),
]
```

Specify `rules` to only include those routes from the controller:

```
routes = lambda: [
    controller(SecurityController, rules=[
       # these inherit all unspecified kwargs from the decorated view methods
       rule('/login', SecurityController.login),  # methods=['GET', 'POST']
       rule('/logout', SecurityController.logout),  # methods=['GET']
       rule('/sign-up', SecurityController.register),  # methods=['GET', 'POST']
    ]),
]
```

Parameters

- **url_prefix_or_controller_cls** – The controller class, or a url prefix for all of the rules from the controller class passed as the second argument
- **controller_cls** – If a url prefix was given as the first argument, then the controller class must be passed as the second argument
- **rules** – An optional list of rules to limit/customize the routes included from the controller

resource

flask_unchained.**resource**(*url_prefix_or_resource_cls: Union[str, Type[flask_unchained.bundles.controller.resource.Resource]], resource_cls: Optional[Type[flask_unchained.bundles.controller.resource.Resource]] = None, *, member_param: Optional[str] = None, unique_member_param: Optional[str] = None, rules: Optional[Iterable[Union[flask_unchained.bundles.controller.route.Route, Iterable[flask_unchained.bundles.controller.route.Route]]]] = None, subresources: Optional[Iterable[Iterable[flask_unchained.bundles.controller.route.Route]]] = None*) → Iterable[flask_unchained.bundles.controller.route.Route]

This function is used to register a *Resource*'s routes.

Example usage:

```
routes = lambda: [
    resource(ProductResource),
]
```

Or with the optional prefix argument:

```
routes = lambda: [
    resource('/products', ProductResource),
]
```

Specify `rules` to only include those routes from the resource:

```
routes = lambda: [
    resource('/users', UserResource, rules=[
       get('/', UserResource.list),
       get('/<int:id>', UserResource.get),
    ]),
]
```

Specify `subresources` to nest resource routes:

```
routes = lambda: [
    resource('/users', UserResource, subresources=[
       resource('/roles', RoleResource)
    ]),
]
```

Subresources can be nested as deeply as you want, however it's not recommended to go more than two or three levels deep at the most, otherwise your URLs will become unwieldy.

Parameters

- **url_prefix_or_resource_cls** – The resource class, or a url prefix for all of the rules from the resource class passed as the second argument.
- **resource_cls** – If a url prefix was given as the first argument, then the resource class must be passed as the second argument.
- **member_param** – Optionally override the controller's member_param attribute.
- **rules** – An optional list of rules to limit/customize the routes included from the resource.
- **subresources** – An optional list of subresources.

func

flask_unchained.**func**(*rule_or_view_func: Union[str, Callable], view_func: Optional[Callable] = <py_meta_utils._missing_cls object>, blueprint: Optional[flask.blueprints.Blueprint] = <py_meta_utils._missing_cls object>, defaults: Optional[Dict[str, Any]] = <py_meta_utils._missing_cls object>, endpoint: Optional[str] = <py_meta_utils._missing_cls object>, methods: Union[List[str], Tuple[str], Set[str], None] = <py_meta_utils._missing_cls object>, only_if: Union[bool, Callable[flask_unchained.flask_unchained.FlaskUnchained, bool], None] = <py_meta_utils._missing_cls object>, **rule_options*) → Iterable[flask_unchained.bundles.controller.route.Route]

This function allows to register legacy view functions as routes, eg:

```
@route('/')
def index():
    return render_template('site/index.html')

routes = lambda: [
    func(index),
]
```

It accepts an optional url rule argument:

```
routes = lambda: [
    func('/products', product_list_view),
]
```

As well as supporting the same kwargs as Werkzeug's `Rule`, eg:

```
routes = lambda: [
    func('/', index, endpoint='home', methods=['GET', 'POST']),
]
```

Parameters

- **rule_or_view_func** – The view function, or an optional url rule for the view function given as the second argument
- **view_func** – The view function if passed a url rule as the first argument
- **only_if** – An optional function to decide at runtime whether or not to register the route with Flask. It gets passed the configured app as a single argument, and should return a boolean.
- **rule_options** – Keyword arguments that ultimately end up getting passed on to `Rule`

include

flask_unchained.**include**(*url_prefix_or_module_name: str, module_name: Optional[str] = None, *, attr: str = 'routes', exclude: Union[List[str], Tuple[str], Set[str], None] = None, only: Union[List[str], Tuple[str], Set[str], None] = None*) → Iterable[flask_unchained.bundles.controller.route.Route]

Include the routes from another module at that point in the tree. For example:

```
# project-root/bundles/primes/routes.py
routes = lambda: [
    controller('/two', TwoController),
    controller('/three', ThreeController),
    controller('/five', FiveController),
]


# project-root/bundles/blog/routes.py
routes = lambda: [
    func('/', index),
    controller('/authors', AuthorController),
    controller('/posts', PostController),
]
```

(continues on next page)

(continued from previous page)

```
# project-root/your_app_bundle/routes.py
routes = lambda: [
    include('bundles.primes.routes'),

    # these last two are equivalent
    include('/blog', 'bundles.blog.routes'),
    prefix('/blog', [
        include('bundles.blog.routes'),
    ]),
]
```

Parameters

- **url_prefix_or_module_name** – The module name, or a url prefix for all of the included routes in the module name passed as the second argument.
- **module_name** – The module name of the routes to include if a url prefix was given as the first argument.
- **attr** – The attribute name in the module, if different from `routes`.
- **exclude** – An optional list of endpoints to exclude.
- **only** – An optional list of endpoints to only include.

prefix

flask_unchained.**prefix**(*url_prefix: str, children: Iterable[Union[flask_unchained.bundles.controller.route.Route, Iterable[flask_unchained.bundles.controller.route.Route]]]*) → Iterable[flask_unchained.bundles.controller.route.Route]

Sets a prefix on all of the child routes passed to it. It also supports nesting, eg:

```
routes = lambda: [
    prefix('/foobar', [
        controller('/one', OneController),
        controller('/two', TwoController),
        prefix('/baz', [
            controller('/three', ThreeController),
            controller('/four', FourController),
        ])
    ])
]
```

Parameters

- **url_prefix** – The url prefix to set on the child routes
- **children** –

rule

flask_unchained.**rule**(*rule: str, cls_method_name_or_view_fn: Union[str, Callable, None] = None, *, defaults: Optional[Dict[str, Any]] = <py_meta_utils._missing_cls object>, endpoint: Optional[str] = <py_meta_utils._missing_cls object>, is_member: Optional[bool] = <py_meta_utils._missing_cls object>, methods: Union[List[str], Tuple[str], Set[str], None] = <py_meta_utils._missing_cls object>, only_if: Union[bool, Callable[flask_unchained.flask_unchained.FlaskUnchained, bool], None] = <py_meta_utils._missing_cls object>, **rule_options*) → Iterable[flask_unchained.bundles.controller.route.Route]

Used to specify customizations to the route settings of class-based view function. Unspecified kwargs will be inherited from the route decorated on each view. For example:

```
routes = lambda: [
    prefix('/api/v1', [
        controller(SecurityController, rules=[
            rule('/login', SecurityController.login,
                 endpoint='security_api.login'),  # methods=['GET', 'POST']
            rule('/logout', SecurityController.logout,
                 endpoint='security_api.logout'),  # methods=['GET']
            rule('/sign-up', SecurityController.register,
                 endpoint='security_api.register'),  # methods=['GET', 'POST']
        ]),
    ],
]
```

Parameters

- **rule** – The URL rule.
- **cls_method_name_or_view_fn** – The view function.
- **defaults** – Any default values for parameters in the URL rule.
- **endpoint** – The endpoint name of this view. Determined automatically if left unspecified.
- **is_member** – Whether or not this view is for a `Resource` member method.
- **methods** – A list of HTTP methods supported by this view. Defaults to `['GET']`.
- **only_if** – A boolean or callable to dynamically determine whether or not to register this route with the app.
- **rule_options** – Other kwargs passed on to `Rule`.

get

flask_unchained.**get**(*rule: str, cls_method_name_or_view_fn: Union[str, Callable, None] = None, *, defaults: Optional[Dict[str, Any]] = <py_meta_utils._missing_cls object>, endpoint: Optional[str] = <py_meta_utils._missing_cls object>, is_member: Optional[bool] = <py_meta_utils._missing_cls object>, only_if: Union[bool, Callable[flask_unchained.flask_unchained.FlaskUnchained, bool], None] = <py_meta_utils._missing_cls object>, **rule_options*) → Iterable[flask_unchained.bundles.controller.route.Route]

Like *rule()*, except specifically for HTTP GET requests.

Parameters

- **rule** – The url rule for this route.

- **cls_method_name_or_view_fn** – The view function for this route.
- **is_member** – Whether or not this route is a member function.
- **only_if** – An optional function to decide at runtime whether or not to register the route with Flask. It gets passed the configured app as a single argument, and should return a boolean.
- **rule_options** – Keyword arguments that ultimately end up getting passed on to Rule

patch

flask_unchained.**patch**(*rule: str, cls_method_name_or_view_fn: Union[str, Callable, None] = None, *, defaults: Optional[Dict[str, Any]] = <py_meta_utils._missing_cls object>, endpoint: Optional[str] = <py_meta_utils._missing_cls object>, is_member: Optional[bool] = <py_meta_utils._missing_cls object>, only_if: Union[bool, Callable[flask_unchained.flask_unchained.FlaskUnchained, bool], None] = <py_meta_utils._missing_cls object>, **rule_options*) → Iterable[flask_unchained.bundles.controller.route.Route]

Like *rule()*, except specifically for HTTP PATCH requests.

Parameters

- **rule** – The url rule for this route.
- **cls_method_name_or_view_fn** – The view function for this route.
- **is_member** – Whether or not this route is a member function.
- **only_if** – An optional function to decide at runtime whether or not to register the route with Flask. It gets passed the configured app as a single argument, and should return a boolean.
- **rule_options** – Keyword arguments that ultimately end up getting passed on to Rule

post

flask_unchained.**post**(*rule: str, cls_method_name_or_view_fn: Union[str, Callable, None] = None, *, defaults: Optional[Dict[str, Any]] = <py_meta_utils._missing_cls object>, endpoint: Optional[str] = <py_meta_utils._missing_cls object>, is_member: Optional[bool] = <py_meta_utils._missing_cls object>, only_if: Union[bool, Callable[flask_unchained.flask_unchained.FlaskUnchained, bool], None] = <py_meta_utils._missing_cls object>, **rule_options*) → Iterable[flask_unchained.bundles.controller.route.Route]

Like *rule()*, except specifically for HTTP POST requests.

Parameters

- **rule** – The url rule for this route.
- **cls_method_name_or_view_fn** – The view function for this route.
- **is_member** – Whether or not this route is a member function.
- **only_if** – An optional function to decide at runtime whether or not to register the route with Flask. It gets passed the configured app as a single argument, and should return a boolean.
- **rule_options** – Keyword arguments that ultimately end up getting passed on to Rule

put

flask_unchained.**put**(*rule: str, cls_method_name_or_view_fn: Union[str, Callable, None] = None, *, defaults: Optional[Dict[str, Any]] = <py_meta_utils._missing_cls object>, endpoint: Optional[str] = <py_meta_utils._missing_cls object>, is_member: Optional[bool] = <py_meta_utils._missing_cls object>, only_if: Union[bool, Callable[flask_unchained.flask_unchained.FlaskUnchained, bool], None] = <py_meta_utils._missing_cls object>, **rule_options*) → Iterable[flask_unchained.bundles.controller.route.Route]

Like *rule()*, except specifically for HTTP PUT requests.

Parameters

- **rule** – The url rule for this route.
- **cls_method_name_or_view_fn** – The view function for this route.
- **is_member** – Whether or not this route is a member function.
- **only_if** – An optional function to decide at runtime whether or not to register the route with Flask. It gets passed the configured app as a single argument, and should return a boolean.
- **rule_options** – Keyword arguments that ultimately end up getting passed on to Rule

delete

flask_unchained.**delete**(*rule: str, cls_method_name_or_view_fn: Union[str, Callable, None] = None, *, defaults: Optional[Dict[str, Any]] = <py_meta_utils._missing_cls object>, endpoint: Optional[str] = <py_meta_utils._missing_cls object>, is_member: Optional[bool] = <py_meta_utils._missing_cls object>, only_if: Union[bool, Callable[flask_unchained.flask_unchained.FlaskUnchained, bool], None] = <py_meta_utils._missing_cls object>, **rule_options*) → Iterable[flask_unchained.bundles.controller.route.Route]

Like *rule()*, except specifically for HTTP DELETE requests.

Parameters

- **rule** – The url rule for this route.
- **cls_method_name_or_view_fn** – The view function for this route.
- **is_member** – Whether or not this route is a member function.
- **only_if** – An optional function to decide at runtime whether or not to register the route with Flask. It gets passed the configured app as a single argument, and should return a boolean.
- **rule_options** – Keyword arguments that ultimately end up getting passed on to Rule

Route

class flask_unchained.bundles.controller.route.**Route**(*rule: Optional[str], view_func: Union[str, function], blueprint: Optional[flask.blueprints.Blueprint] = None, defaults: Optional[Dict[str, Any]] = None, endpoint: Optional[str] = None, is_member: bool = False, methods: Union[List[str], Tuple[str, ...], None] = None, only_if: Union[bool, function, None] = <py_meta_utils._missing_cls object>, **rule_options*)

This is a semi-private class that you most likely shouldn't use directly. Instead, you should use the public functions in *Declarative Routing*, and the `route()` and `no_route()` decorators.

This class is used to store an *intermediate* representation of route details as an attribute on view functions and class view methods. Most notably, this class's `rule` and `full_rule` attributes may not represent the final url rule that gets registered with `Flask`.

Further gotchas with `Controller` and `Resource` routes include that their view_func must be finalized from the outside using `TheControllerClass.method_as_view`.

should_register(*app: flask_unchained.flask_unchained.FlaskUnchained*) → bool

Determines whether or not this route should be registered with the app, based on `only_if`.

property defaults

The URL defaults for this route.

property endpoint

The endpoint for this route.

property is_member

Whether or not this route is for a resource member route.

property method_name

The string name of this route's view function.

property methods

The HTTP methods supported by this route.

property module_name

The module where this route's view function was defined.

property rule

The (partial) url rule for this route.

property full_rule

The full url rule for this route, including any blueprint prefix.

property full_name

The full name of this route's view function, including the module path and controller name, if any.

7.6.6 Hooks

RegisterBlueprintsHook

class flask_unchained.bundles.controller.hooks.**RegisterBlueprintsHook**(*unchained: flask_unchained.unchained.Un bundle: Optional[flask_unchained.bundle = None*)

Registers legacy Flask blueprints with the app.

name: str = 'blueprints'
The name of this hook.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['views']
The default module this hook loads from.

Override by setting the `blueprints_module_names` attribute on your bundle class.

process_objects(*app: flask_unchained.flask_unchained.FlaskUnchained, blueprints: List[flask.blueprints.Blueprint]*)
Registers discovered blueprints with the app.

collect_from_bundles(*bundles: List[flask_unchained.bundles.Bundle]*) → List[flask.blueprints.Blueprint]
Find blueprints in bundles.

collect_from_bundle(*bundle: flask_unchained.bundles.Bundle*) → Iterable[flask.blueprints.Blueprint]
Finds blueprints in a bundle hierarchy.

type_check(*obj*)
Returns True if `obj` is an instance of `flask.Blueprint`.

RegisterBundleBlueprintsHook

class flask_unchained.bundles.controller.hooks.**RegisterBundleBlueprintsHook**(*unchained: flask_unchained.unch bundle: Optional[flask_unchaine = None*)

Registers a bundle blueprint for each bundle with views and/or template/static folders.

name: str = 'bundle_blueprints'
The name of this hook.

run_hook(*app: flask_unchained.flask_unchained.FlaskUnchained, bundles: List[flask_unchained.bundles.Bundle], unchained_config: Optional[Dict[str, Any]] = None*) → None
Register blueprints for bundles, where necessary.

RegisterRoutesHook

class flask_unchained.bundles.controller.hooks.**RegisterRoutesHook**(*unchained: flask_unchained.unchained.Unchain bundle: Optional[flask_unchained.bundles.Bund = None*)

Registers routes.

name: str = 'routes'
The name of this hook.

bundle_module_name: Optional[str] = 'routes'
The default module this hook loads from.

Override by setting the `routes_module_name` attribute on your bundle class.

run_hook(*app: flask_unchained.flask_unchained.FlaskUnchained, bundles: List[flask_unchained.bundles.Bundle], unchained_config: Optional[Dict[str, Any]] = None*) → None
Discover and register routes.

process_objects(*app: flask_unchained.flask_unchained.FlaskUnchained, routes: Iterable[flask_unchained.bundles.controller.route.Route]*)
Organize routes by where they came from, and then register them with the app.

get_explicit_routes(*bundle: flask_unchained.bundles.Bundle*)
Collect routes from a bundle using declarative routing.

collect_from_bundle(*bundle: flask_unchained.bundles.Bundle*)
Collect routes from a bundle when not using declarative routing.

type_check(*obj*)
Returns True if `obj` was decorated with *`route()`* or if `obj` is a controller or resource with views.

7.6.7 FlaskForm

Using forms in Flask Unchained is exactly the same as it is with Flask-WTF.

class flask_unchained.**FlaskForm**(*formdata=None*, *obj=None*, *prefix=''*, *data=None*, *meta=None*, ***kwargs*)
Base form class extending `flask_wtf.FlaskForm`. Adds support for specifying the field order via a `field_order` class attribute.

Parameters

- **formdata** – Used to pass data coming from the end user, usually `request.POST` or equivalent. formdata should be some sort of request-data wrapper which can get multiple parameters from the form input, and values are unicode strings, e.g. a Werkzeug/Django/WebOb MultiDict
- **obj** – If `formdata` is empty or not provided, this object is checked for attributes matching form field names, which will be used for field values.
- **prefix** – If provided, all fields will have their name prefixed with the value.
- **data** – Accept a dictionary of data. This is only used if `formdata` and *obj* are not present.
- **meta** – If provided, this is a dictionary of values to override attributes on this form's meta instance.

- ****kwargs** – If *formdata* is empty or not provided and *obj* does not contain an attribute named the same as a field, form will assign the value of a matching keyword argument to the field, if one exists.

field_order = ()
An ordered list of field names. Fields not listed here will be rendered first.

7.6.8 Utility Functions

redirect

flask_unchained.**redirect**(*where: Optional[str] = None*, *default: Optional[str] = None*, *override: Optional[str] = None*, *_anchor: Optional[str] = None*, *_cls: object = None*, *_external: Optional[bool] = False*, *_external_host: Optional[str] = None*, *_method: Optional[str] = None*, *_scheme: Optional[str] = None*, ***values*) → flask.wrappers.Response

An improved version of flask's redirect function

Parameters

- **where** – A URL, endpoint, or config key name to redirect to
- **default** – A URL, endpoint, or config key name to redirect to if `where` is invalid
- **override** – explicitly redirect to a URL, endpoint, or config key name (takes precedence over the `next` value in query strings or forms)
- **values** – the variable arguments of the URL rule
- **_anchor** – if provided this is added as anchor to the URL.
- **_cls** – if specified, allows a method name to be passed to where, default, and/or override
- **_external** – if set to `True`, an absolute URL is generated. Server address can be changed via `SERVER_NAME` configuration variable which defaults to *localhost*.
- **_external_host** – if specified, the host of an external server to generate urls for (eg https://example.com or localhost:8888)
- **_method** – if provided this explicitly specifies an HTTP method.
- **_scheme** – a string specifying the desired URL scheme. The *_external* parameter must be set to `True` or a `ValueError` is raised. The default behavior uses the same scheme as the current request, or `PREFERRED_URL_SCHEME` from the app configuration if no request context is available. As of Werkzeug 0.10, this also can be set to an empty string to build protocol-relative URLs.

url_for

flask_unchained.**url_for**(*endpoint_or_url_or_config_key: Optional[str], _anchor: Optional[str] = None, _cls: object = None, _external: Optional[bool] = False, _external_host: Optional[str] = None, _method: Optional[str] = None, _scheme: Optional[str] = None, **values*) → Optional[str]

An improved version of flask's url_for function

Parameters

- **endpoint_or_url_or_config_key** – what to lookup. it can be an endpoint name, an app config key, or an already-formed url. if _cls is specified, it also accepts a method name.

- **values** – the variable arguments of the URL rule
- **_anchor** – if provided this is added as anchor to the URL.
- **_cls** – if specified, can also pass a method name as the first argument
- **_external** – if set to `True`, an absolute URL is generated. Server address can be changed via `SERVER_NAME` configuration variable which defaults to *localhost*.
- **_external_host** – if specified, the host of an external server to generate urls for (eg https://example.com or localhost:8888)
- **_method** – if provided this explicitly specifies an HTTP method.
- **_scheme** – a string specifying the desired URL scheme. The *_external* parameter must be set to `True` or a `ValueError` is raised. The default behavior uses the same scheme as the current request, or `PREFERRED_URL_SCHEME` from the app configuration if no request context is available. As of Werkzeug 0.10, this also can be set to an empty string to build protocol-relative URLs.

abort

flask_unchained.**abort**(*status*, **args*, ***kwargs*)

Raises an `HTTPException` for the given status code or WSGI application.

If a status code is given, it will be looked up in the list of exceptions and will raise that exception. If passed a WSGI application, it will wrap it in a proxy WSGI exception and raise that:

```
abort(404)  # 404 Not Found
abort(Response('Hello World'))
```

generate_csrf

flask_unchained.**generate_csrf**(*secret_key=None*, *token_key=None*)

Generate a CSRF token. The token is cached for a request, so multiple calls to this function will generate the same token.

During testing, it might be useful to access the signed token in `g.csrf_token` and the raw token in `session['csrf_token']`.

Parameters

- **secret_key** – Used to securely sign the token. Default is `WTF_CSRF_SECRET_KEY` or `SECRET_KEY`.
- **token_key** – Key where token is stored in session for comparision. Default is `WTF_CSRF_FIELD_NAME` or `'csrf_token'`.

7.7 Graphene Bundle API

flask_unchained.bundles.graphene

GrapheneBundle	The Graphene Bundle.

flask_unchained.bundles.graphene.config

Config	
DevConfig	

flask_unchained.bundles.graphene.hooks

RegisterGrapheneMutationsHook	Registers Graphene Mutations with the Graphene Bundle.
RegisterGrapheneQueriesHook	Registers Graphene Queries with the Graphene Bundle.
RegisterGrapheneRootSchemaHook	Creates the root `graphene.Schema` to register with Flask-GraphQL.
RegisterGrapheneTypesHook	Registers SQLAlchemyObjectTypes with the Graphene Bundle.

flask_unchained.bundles.graphene.object_types

QueriesObjectType	Base class for `query` schema definitions.
MutationsObjectType	Base class for `mutation` schema definitions.
SQLAlchemyObjectType	Base class for SQLAlchemy model object types.

7.7.1 GrapheneBundle

class flask_unchained.bundles.graphene.**GrapheneBundle**
The Graphene Bundle. Integrates Flask-GraphQL with SQLAlchemy.

7.7.2 Config

class flask_unchained.bundles.graphene.config.**Config**

GRAPHENE_URL = '/graphql'
The URL where graphene should be served from. Set to `None` to disable.

GRAPHENE_BATCH_URL = None
The URL where graphene should be served from in batch mode. Set to `None` to disable.

GRAPHENE_ENABLE_GRAPHIQL = False
Whether or not to enable GraphIQL.

GRAPHENE_PRETTY_JSON = False
Whether or not to pretty print the returned JSON.

class flask_unchained.bundles.graphene.config.**DevConfig**

GRAPHENE_ENABLE_GRAPHIQL = True
Whether or not to enable GraphIQL.

GRAPHENE_PRETTY_JSON = True
Whether or not to pretty print the returned JSON.

7.7.3 RegisterGrapheneMutationsHook

class flask_unchained.bundles.graphene.hooks.**RegisterGrapheneMutationsHook**(*unchained: flask_unchained.uncha bun- dle: Op- tional[flask_unchained = None*)

Registers Graphene Mutations with the Graphene Bundle.

name: str = 'graphene_mutations'
The name of this hook.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['graphene.mutations',
The default module this hook loads from.

Override by setting the `graphene_mutations_module_names` attribute on your bundle class.

process_objects(*app: flask_unchained.flask_unchained.FlaskUnchained, mutations: Dict[str, flask_unchained.bundles.graphene.object_types.MutationsObjectType]*)
Register discovered mutations with the Graphene Bundle.

type_check(*obj: Any*)
Returns True if `obj` is a subclass of *MutationsObjectType*.

7.7.4 RegisterGrapheneQueriesHook

class flask_unchained.bundles.graphene.hooks.**RegisterGrapheneQueriesHook**(*unchained: flask_unchained.unchaine bun- dle: Op- tional[flask_unchained.bu = None*)

Registers Graphene Queries with the Graphene Bundle.

name: str = 'graphene_queries'
The name of this hook.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['graphene.queries', '
The default module this hook loads from.

Override by setting the `graphene_queries_module_names` attribute on your bundle class.

process_objects(*app: flask_unchained.flask_unchained.FlaskUnchained, queries: Dict[str, flask_unchained.bundles.graphene.object_types.QueriesObjectType]*)
Register discovered queries with the Graphene Bundle.

type_check(*obj: Any*)
Returns True if `obj` is a subclass of *QueriesObjectType*.

7.7.5 RegisterGrapheneRootSchemaHook

class flask_unchained.bundles.graphene.hooks.**RegisterGrapheneRootSchemaHook**(*unchained: flask_unchained.unch bun- dle: Op- tional[flask_unchaine = None*)

Creates the root `graphene.Schema` to register with Flask-GraphQL.

name: str = 'graphene_root_schema'
The name of this hook.

run_hook(*app: flask_unchained.flask_unchained.FlaskUnchained, bundles: List[flask_unchained.bundles.Bundle], unchained_config: Optional[Dict[str, Any]] = None*) → None
Create the root `graphene.Schema` from queries, mutations, and types discovered by the other hooks and register it with the Graphene Bundle.

7.7.6 RegisterGrapheneTypesHook

class flask_unchained.bundles.graphene.hooks.**RegisterGrapheneTypesHook**(*unchained: flask_unchained.unchained.U bun- dle: Op- tional[flask_unchained.bundl = None*)

Registers SQLAlchemyObjectTypes with the Graphene Bundle.

name: str = 'graphene_types'
The name of this hook.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['graphene.types', 'gr
The default module this hook loads from.

Override by setting the `graphene_types_module_names` attribute on your bundle class.

process_objects(*app: flask_unchained.flask_unchained.FlaskUnchained, types: Dict[str, flask_unchained.bundles.graphene.object_types.SQLAlchemyObjectType]*)
Implement to do stuff with discovered objects (eg, registering them with the app instance).

type_check(*obj: Any*)
Implement to determine which objects in a module should be processed by this hook.

7.7.7 QueriesObjectType

class flask_unchained.bundles.graphene.**QueriesObjectType**(**args*, ***kwargs*)

Base class for `query` schema definitions. `graphene.Field` and `graphene.List` fields are automatically resolved (but you can write your own and it will disable the automatic behavior for that field).

Example usage:

```
# your_bundle/graphql/schema.py

from flask_unchained.bundles.graphene import QueriesObjectType

from . import types

class YourBundleQueries(QueriesObjectType):
    parent = graphene.Field(types.Parent, id=graphene.ID(required=True))
    parents = graphene.List(types.Parent)

    child = graphene.Field(types.Child, id=graphene.ID(required=True))
    children = graphene.List(types.Child)

    # this is what the default resolvers do, and how you would override them:
    def resolve_child(self, info, **kwargs):
        return types.Child._meta.model.query.get_by(**kwargs)

    def resolve_children(self, info, **kwargs):
        return types.Child._meta.model.query.all()
```

7.7.8 MutationsObjectType

class flask_unchained.bundles.graphene.**MutationsObjectType**(**args*, ***kwargs*)

Base class for `mutation` schema definitions.

Example usage:

```
#  your_bundle/graphql/mutations.py

import graphene

from flask_unchained import unchained
from flask_unchained.bundles.sqlalchemy import SessionManager, db
from graphql import GraphQLError

from . import types

session_manager: SessionManager = unchained.get_local_proxy('session_manager')

class CreateParent(graphene.Mutation):
    class Arguments:
        name = graphene.String(required=True)
        children = graphene.List(graphene.ID)

    parent = graphene.Field(types.Parent)
    success = graphene.Boolean()

    def mutate(self, info, children, **kwargs):
        if children:
```

(continues on next page)

(continued from previous page)

```
            children = (session_manager
                        .query(types.Child._meta.model)
                        .filter(types.Child._meta.model.id.in_(children))
                        .all())
        try:
            parent = types.Parent._meta.model(children=children, **kwargs)
        except db.ValidationErrors as e:
            raise GraphQLError(str(e))

        session_manager.save(parent, commit=True)
        return CreateParent(parent=parent, success=True)

class DeleteParent(graphene.Mutation):
    class Arguments:
        id = graphene.ID(required=True)

    id = graphene.Int()
    success = graphene.Boolean()

    def mutate(self, info, id):
        parent = session_manager.query(types.Parent._meta.model).get(id)
        session_manager.delete(parent, commit=True)
        return DeleteParent(id=id, success=True)

class EditParent(graphene.Mutation):
    class Arguments:
        id = graphene.ID(required=True)
        name = graphene.String()
        children = graphene.List(graphene.ID)

    parent = graphene.Field(types.Parent)
    success = graphene.Boolean()

    def mutate(self, info, id, children, **kwargs):
        parent = session_manager.query(types.Parent._meta.model).get(id)

        try:
            parent.update(**{k: v for k, v in kwargs.items() if v})
        except db.ValidationErrors as e:
            raise GraphQLError(str(e))

        if children:
            parent.children = (session_manager
                               .query(types.Child._meta.model)
                               .filter(types.Child._meta.model.id.in_
↪(children))
                               .all())

        session_manager.save(parent, commit=True)
        return EditParent(parent=parent, success=True)


# your_bundle/graphql/schema.py

from flask_unchained.bundles.graphene import MutationsObjectType

from . import mutations
```

(continues on next page)

(continued from previous page)

```
class YourBundleMutations(MutationsObjectType):
    create_parent = mutations.CreateParent.Field()
    delete_parent = mutations.DeleteParent.Field()
    edit_parent = mutations.EditParent.Field()
```

7.7.9 SQLAlchemyObjectType

class flask_unchained.bundles.graphene.**SQLAlchemyObjectType**(**args*, ***kwargs*)

Base class for SQLAlchemy model object types. Acts exactly the same as `graphene_sqlalchemy.SQLAlchemyObjectType`, except we've added compatibility with the SQLAlchemy Bundle.

Example usage:

```
# your_bundle/models.py

from flask_unchained.bundles.sqlalchemy import db

class Parent(db.Model):
    name = db.Column(db.String)

    children = db.relationship('Child', back_populates='parent',
                               cascade='all,delete,delete-orphan')

class Child(db.Model):
    name = db.Column(db.String)

    parent_id = db.foreign_key('Parent')
    parent = db.relationship('Parent', back_populates='children')


# your_bundle/graphql/types.py

import graphene

from flask_unchained.bundles.graphene import SQLAlchemyObjectType

from .. import models

class Parent(SQLAlchemyObjectType):
    class Meta:
        model = models.Parent
        only_fields = ('id', 'name', 'created_at', 'updated_at')

    children = graphene.List(lambda: Child)

class Child(SQLAlchemyObjectType):
    class Meta:
        model = models.Child
        only_fields = ('id', 'name', 'created_at', 'updated_at')

    parent = graphene.Field(Parent)
```

7.8 Mail Bundle API

flask_unchained.bundles.mail

MailBundle	The Mail Bundle.

flask_unchained.bundles.mail.config

Config	Default configuration options for the mail bundle.
DevConfig	Development-specific config options for the mail bundle.
ProdConfig	Production-specific config options for the mail bundle.
StagingConfig	Inherit settings from production.
TestConfig	Test-specific config options for the mail bundle.

flask_unchained.bundles.mail.extensions

Mail	The *Mail* extension.

7.8.1 MailBundle

class flask_unchained.bundles.mail.**MailBundle**

The Mail Bundle.

name: str = 'mail_bundle'

The name of the Mail Bundle.

command_group_names = ['mail']

Click groups for the Mail Bundle.

7.8.2 Config

class flask_unchained.bundles.mail.config.**Config**

Default configuration options for the mail bundle.

MAIL_SERVER = '127.0.0.1'

The hostname/IP of the mail server.

MAIL_PORT = 25

The port the mail server is running on.

MAIL_USERNAME = None

The username to connect to the mail server with, if any.

MAIL_PASSWORD = None

The password to connect to the mail server with, if any.

MAIL_USE_TLS = False

Whether or not to use TLS.

MAIL_USE_SSL = False

Whether or not to use SSL.

MAIL_DEFAULT_SENDER = 'Flask Mail <noreply@localhost>'
The default sender to use, if none is specified otherwise.

MAIL_SEND_FN(*to: Union[str, List[str], None] = None, template: Optional[str] = None, **kwargs*)
The function to use for sending emails. Defaults to `_send_mail()`. Any customized send function must implement the same function signature:

```
def send_mail(subject_or_message: Optional[Union[str, Message]] = None,
              to: Optional[Union[str, List[str]] = None,
              template: Optional[str] = None,
              **kwargs):
    # ...
```

NOTE: *subject_or_message* is optional because you can also pass *subject* as a keyword argument, and *to* is optional because you can also pass *recipients* as a keyword argument. These are artifacts of backwards-compatibility with vanilla Flask-Mail.

MAIL_DEBUG = 0
The debug level to set for interactions with the mail server.

MAIL_MAX_EMAILS = None
The maximum number of emails to send per connection with the mail server.

MAIL_SUPPRESS_SEND = False
Whether or not to actually send emails, or just pretend to. This is mainly useful for testing.

MAIL_ASCII_ATTACHMENTS = False
Whether or not to coerce attachment filenames to ASCII.

class flask_unchained.bundles.mail.config.**DevConfig**
Development-specific config options for the mail bundle.

MAIL_DEBUG = 1
Set the mail server debug level to 1 in development.

MAIL_PORT = 1025
In development, the mail bundle is configured to connect to MailHog.

class flask_unchained.bundles.mail.config.**ProdConfig**
Production-specific config options for the mail bundle.

MAIL_PORT = 465
In production, the mail bundle is configured to connect using SSL.

MAIL_USE_SSL = True
Set use SSL to `True` in production.

class flask_unchained.bundles.mail.config.**StagingConfig**
Inherit settings from production.

class flask_unchained.bundles.mail.config.**TestConfig**
Test-specific config options for the mail bundle.

MAIL_SUPPRESS_SEND = True
Do not actually send emails when running tests.

7.8.3 The Mail Extension

class flask_unchained.bundles.mail.**Mail**
The *Mail* extension:

```
from flask_unchained.bundles.mail import mail
```

send_message(*subject_or_message: Union[flask_mail.Message, str, None] = None, to: Union[str, List[str], None] = None, **kwargs*)
Send an email using the send function as configured by *MAIL_SEND_FN*.

Parameters

- **subject_or_message** – The subject line, or an instance of flask_mail.Message.
- **to** – The message recipient(s).
- **kwargs** – Extra values to pass on to Message

7.9 OAuth Bundle API

flask_unchained.bundles.oauth

OAuthBundle	The OAuth Bundle.

flask_unchained.bundles.oauth.config

Config

flask_unchained.bundles.oauth.services

OAuthService

flask_unchained.bundles.oauth.views

OAuthController

7.9.1 OAuthBundle

class flask_unchained.bundles.oauth.**OAuthBundle**
The OAuth Bundle.

name: str = 'oauth_bundle'
The name of the OAuth Bundle.

7.9.2 Config

class flask_unchained.bundles.oauth.config.**Config**

OAUTH_REMOTE_APP_GITHUB = {'access_token_method': 'POST', 'access_token_url': 'https

OAUTH_REMOTE_APP_AMAZON = {'access_token_method': 'POST', 'access_token_url': 'https

OAUTH_REMOTE_APP_GITLAB = {'access_token_method': 'POST', 'access_token_url': 'https

7.9.3 OAuthService

class flask_unchained.bundles.oauth.**OAuthService**

get_user_details(*provider: flask_oauthlib.client.OAuthRemoteApp*) → Tuple[str, dict]
For the given `provider`, return the user's email address and any extra data to create the user model with.

on_authorized(*provider: flask_oauthlib.client.OAuthRemoteApp*) → None
Optional callback to add custom behavior upon OAuth authorized.

7.9.4 OAuthController

class flask_unchained.bundles.oauth.**OAuthController**

7.10 Security Bundle API

flask_unchained.bundles.security

SecurityBundle	The Security Bundle.

flask_unchained.bundles.security.config

Config	Config options for the Security Bundle.
AuthenticationConfig	Config options for logging in and out.
TokenConfig	Config options for token authentication.
RegistrationConfig	Config options for user registration
ChangePasswordConfig	Config options for changing passwords
ForgotPasswordConfig	Config options for recovering forgotten passwords
EncryptionConfig	Config options for encryption hashing.

flask_unchained.bundles.security.extensions

Security	The *Security* extension.

flask_unchained.bundles.security.views

SecurityController	The controller for the security bundle.
UserResource	RESTful API resource for the *User* model.

flask_unchained.bundles.security.decorators

auth_required	Decorator for requiring an authenticated user, optionally with roles.
auth_required_same_user	Decorator for requiring an authenticated user to be the same as the user in the URL parameters.
anonymous_user_required	Decorator requiring that there is no user currently logged in.

flask_unchained.bundles.security.models

User	Base user model.
Role	Base role model.
UserRole	Join table between the *User* and *Role* models.

flask_unchained.bundles.security.serializers

UserSerializer	Marshmallow serializer for the *User* model.
RoleSerializer	Marshmallow serializer for the *Role* model.

flask_unchained.bundles.security.services

SecurityService	The security service.
SecurityUtilsService	The security utils service.
UserManager	*ModelManager* for the *User* model.
RoleManager	*ModelManager* for the *Role* model.

flask_unchained.bundles.security.forms

LoginForm	The default login form.
RegisterForm	The default register form.
ChangePasswordForm	The default change password form.
ForgotPasswordForm	The default forgot password form.
ResetPasswordForm	The default reset password form.
SendConfirmationForm	The default resend confirmation email form.

7.10.1 SecurityBundle

class flask_unchained.bundles.security.**SecurityBundle**

The Security Bundle. Integrates Flask Login and Flask Principal with Flask Unchained.

name: str = 'security_bundle'

The name of the Security Bundle.

command_group_names = ['users', 'roles']

Click groups for the Security Bundle.

7.10.2 Config

Config	Config options for the Security Bundle.
AuthenticationConfig	Config options for logging in and out.
TokenConfig	Config options for token authentication.
RegistrationConfig	Config options for user registration
ChangePasswordConfig	Config options for changing passwords
ForgotPasswordConfig	Config options for recovering forgotten passwords
EncryptionConfig	Config options for encryption hashing.

General

class flask_unchained.bundles.security.config.**Config**
Config options for the Security Bundle.

SECURITY_ANONYMOUS_USER
alias of `flask_unchained.bundles.security.models.anonymous_user.AnonymousUser`

SECURITY_UNAUTHORIZED_CALLBACK()
This callback gets called when authorization fails. By default we abort with an HTTP status code of 401 (UNAUTHORIZED).

SECURITY_DATETIME_FACTORY()
Factory function to use when creating new dates. By default we use `datetime.now(timezone.utc)` to create a timezone-aware datetime.

Authentication

class flask_unchained.bundles.security.config.**AuthenticationConfig**
Config options for logging in and out.

SECURITY_LOGIN_FORM
alias of *flask_unchained.bundles.security.forms.LoginForm*

SECURITY_DEFAULT_REMEMBER_ME = False
Whether or not the login form should default to checking the "Remember me?" option.

SECURITY_USER_IDENTITY_ATTRIBUTES = ['email']
List of attributes on the user model that can used for logging in with. Each must be unique.

SECURITY_POST_LOGIN_REDIRECT_ENDPOINT = '/'
The endpoint or url to redirect to after a successful login.

SECURITY_POST_LOGOUT_REDIRECT_ENDPOINT = '/'
The endpoint or url to redirect to after a user logs out.

Token Authentication

class flask_unchained.bundles.security.config.**TokenConfig**
Config options for token authentication.

SECURITY_TOKEN_AUTHENTICATION_KEY = 'auth_token'
Specifies the query string parameter to read when using token authentication.

SECURITY_TOKEN_AUTHENTICATION_HEADER = 'Authentication-Token'
Specifies the HTTP header to read when using token authentication.

SECURITY_TOKEN_MAX_AGE = None
Specifies the number of seconds before an authentication token expires. Defaults to None, meaning the token never expires.

Registration

class flask_unchained.bundles.security.config.**RegistrationConfig**
Config options for user registration

SECURITY_REGISTERABLE = False
Whether or not to enable registration.

SECURITY_REGISTER_FORM
alias of *flask_unchained.bundles.security.forms.RegisterForm*

SECURITY_POST_REGISTER_REDIRECT_ENDPOINT = None
The endpoint or url to redirect to after a user completes the registration form.

SECURITY_SEND_REGISTER_EMAIL = False
Whether or not send a welcome email after a user completes the registration form.

SECURITY_CONFIRMABLE = False
Whether or not to enable required email confirmation for new users.

SECURITY_SEND_CONFIRMATION_FORM
alias of *flask_unchained.bundles.security.forms.SendConfirmationForm*

SECURITY_LOGIN_WITHOUT_CONFIRMATION = False
Allow users to login without confirming their email first. (This option only applies when *SECURITY_CONFIRMABLE* is True.)

SECURITY_CONFIRM_EMAIL_WITHIN = '5 days'
How long to wait until considering the token in confirmation emails to be expired.

SECURITY_POST_CONFIRM_REDIRECT_ENDPOINT = None
Endpoint or url to redirect to after the user confirms their email. Defaults to SECURITY_POST_LOGIN_REDIRECT_ENDPOINT.

SECURITY_CONFIRM_ERROR_REDIRECT_ENDPOINT = None
Endpoint to redirect to if there's an error confirming the user's email.

Change Password

class flask_unchained.bundles.security.config.**ChangePasswordConfig**
Config options for changing passwords

SECURITY_CHANGEABLE = False
Whether or not to enable change password functionality.

SECURITY_CHANGE_PASSWORD_FORM
alias of *flask_unchained.bundles.security.forms.ChangePasswordForm*

SECURITY_POST_CHANGE_REDIRECT_ENDPOINT = None
Endpoint or url to redirect to after the user changes their password.

SECURITY_SEND_PASSWORD_CHANGED_EMAIL = False
Whether or not to send the user an email when their password has been changed. Defaults to True, and it's strongly recommended to leave this option enabled.

Forgot Password

class flask_unchained.bundles.security.config.**ForgotPasswordConfig**
Config options for recovering forgotten passwords

SECURITY_RECOVERABLE = False
Whether or not to enable forgot password functionality.

SECURITY_FORGOT_PASSWORD_FORM
alias of *flask_unchained.bundles.security.forms.ForgotPasswordForm*

SECURITY_RESET_PASSWORD_FORM
alias of *flask_unchained.bundles.security.forms.ResetPasswordForm*

SECURITY_RESET_PASSWORD_WITHIN = '5 days'
Specifies the amount of time a user has before their password reset link expires. Always pluralized the time unit for this value. Defaults to 5 days.

SECURITY_POST_RESET_REDIRECT_ENDPOINT = None
Endpoint or url to redirect to after the user resets their password.

SECURITY_INVALID_RESET_TOKEN_REDIRECT = 'security_controller.forgot_password'
Endpoint or url to redirect to if the reset token is invalid.

SECURITY_EXPIRED_RESET_TOKEN_REDIRECT = 'security_controller.forgot_password'
Endpoint or url to redirect to if the reset token is expired.

SECURITY_API_RESET_PASSWORD_HTTP_GET_REDIRECT = None
Endpoint or url to redirect to if a GET request is made to the reset password view. Defaults to None, meaning no redirect. Useful for single page apps.

SECURITY_SEND_PASSWORD_RESET_NOTICE_EMAIL = False
Whether or not to send the user an email when their password has been reset. Defaults to True, and it's strongly recommended to leave this option enabled.

Encryption

class flask_unchained.bundles.security.config.**EncryptionConfig**
Config options for encryption hashing.

SECURITY_PASSWORD_SALT = 'security-password-salt'
Specifies the HMAC salt. This is only used if the password hash type is set to something other than plain text.

SECURITY_PASSWORD_HASH = 'bcrypt'
Specifies the password hash algorithm to use when hashing passwords. Recommended values for production systems are `argon2`, `bcrypt`, or `pbkdf2_sha512`. May require extra packages to be installed.

SECURITY_PASSWORD_SINGLE_HASH = False
Specifies that passwords should only be hashed once. By default, passwords are hashed twice, first with SECURITY_PASSWORD_SALT, and then with a random salt. May be useful for integrating with other applications.

SECURITY_PASSWORD_SCHEMES = ['argon2', 'bcrypt', 'pbkdf2_sha512', 'plaintext']
List of algorithms that can be used for hashing passwords.

SECURITY_PASSWORD_HASH_OPTIONS = {}
Specifies additional options to be passed to the hashing method.

SECURITY_DEPRECATED_PASSWORD_SCHEMES = ['auto']
List of deprecated algorithms for hashing passwords.

SECURITY_HASHING_SCHEMES = ['sha512_crypt']
List of algorithms that can be used for creating and validating tokens.

SECURITY_DEPRECATED_HASHING_SCHEMES = []
List of deprecated algorithms for creating and validating tokens.

7.10.3 The Security Extension

class flask_unchained.bundles.security.**Security**
The *Security* extension:

```
from flask_unchained.bundles.security import security
```

context_processor(*fn*)
Add a context processor that runs for every view with a template in the security bundle.

Parameters **fn** – A function that returns a dictionary of template context variables.

forgot_password_context_processor(*fn*)
Add a context processor for the *SecurityController.forgot_password()* view.

Parameters **fn** – A function that returns a dictionary of template context variables.

login_context_processor(*fn*)
Add a context processor for the *SecurityController.login()* view.

Parameters **fn** – A function that returns a dictionary of template context variables.

register_context_processor(*fn*)
Add a context processor for the *SecurityController.register()* view.

Parameters **fn** – A function that returns a dictionary of template context variables.

reset_password_context_processor(*fn*)
Add a context processor for the *SecurityController.reset_password()* view.

Parameters **fn** – A function that returns a dictionary of template context variables.

change_password_context_processor(*fn*)
Add a context processor for the *SecurityController.change_password()* view.

Parameters **fn** – A function that returns a dictionary of template context variables.

send_confirmation_context_processor(*fn*)
Add a context processor for the *SecurityController.send_confirmation_email()* view.

Parameters **fn** – A function that returns a dictionary of template context variables.

mail_context_processor(*fn*)
Add a context processor to be used when rendering all the email templates.

Parameters **fn** – A function that returns a dictionary of template context variables.

7.10.4 Views

Decorators

flask_unchained.bundles.security.**auth_required**(*decorated_fn=None*, ***role_rules*)
Decorator for requiring an authenticated user, optionally with roles.

Roles are passed as keyword arguments, like so:

```
@auth_required(role='REQUIRE_THIS_ONE_ROLE')
@auth_required(roles=['REQUIRE', 'ALL', 'OF', 'THESE', 'ROLES'])
@auth_required(one_of=['EITHER_THIS_ROLE', 'OR_THIS_ONE'])
```

Either of the *role* or *roles* kwargs can also be combined with one_of:

```
@auth_required(role='REQUIRED', one_of=['THIS', 'OR_THIS'])
```

Aborts with `HTTP 401: Unauthorized` if no user is logged in, or `HTTP 403: Forbidden` if any of the specified role checks fail.

flask_unchained.bundles.security.**auth_required_same_user**(**args*, ***kwargs*)
Decorator for requiring an authenticated user to be the same as the user in the URL parameters. By default the user url parameter name to lookup is `id`, but this can be customized by passing an argument:

```
@auth_require_same_user('user_id')
@bp.route('/users/<int:user_id>/foo/<int:id>')
def get(user_id, id):
    # do stuff
```

Any keyword arguments are passed along to the @auth_required decorator, so roles can also be specified in the same was as it, eg:

```
@auth_required_same_user('user_id', role='ROLE_ADMIN')
```

Aborts with `HTTP 403: Forbidden` if the user-check fails.

flask_unchained.bundles.security.**anonymous_user_required**(**decorator_args*, *msg=None*, *category=None*, *redirect_url=None*)

Decorator requiring that there is no user currently logged in.

Aborts with `HTTP 403: Forbidden` if there is an authenticated user.

SecurityController

class flask_unchained.bundles.security.**SecurityController**

The controller for the security bundle.

check_auth_token(***kwargs*)

View function to check a token, and if it's valid, log the user in.

Disabled by default; must be explicitly enabled in your `routes.py`.

login()

View function to log a user in. Supports html and json requests.

logout()

View function to log a user out. Supports html and json requests.

register()

View function to register user. Supports html and json requests.

send_confirmation_email()

View function which sends confirmation token and instructions to a user.

confirm_email(*token*)

View function to confirm a user's token from the confirmation email send to them. Supports html and json requests.

forgot_password()

View function to request a password recovery email with a reset token. Supports html and json requests.

reset_password(*token*)

View function verify a users reset password token from the email we sent to them. It also handles the form for them to set a new password. Supports html and json requests.

change_password(***kwargs*)

View function for a user to change their password. Supports html and json requests.

UserResource

class flask_unchained.bundles.security.**UserResource**

RESTful API resource for the *User* model.

create(*user*, *errors*)

Create an instance of a model.

Parameters

- **instance** – The created model instance.
- **errors** – Any errors.

Returns The created model instance, or a dictionary of errors.

7.10.5 Models and Managers

User

class flask_unchained.bundles.security.**User**(***kwargs*)
Base user model. Includes `email`, `password`, `is_active`, and `confirmed_at` columns, and a many-to-many relationship to the *Role* model via the intermediary *UserRole* join table.

get_auth_token(*security_utils_service='INJECTABLE_PARAMETER'*)
Returns the user's authentication token.

has_role(*role*)
Returns *True* if the user identifies with the specified role.

Parameters **role** – A role name or *Role* instance

UserManager

class flask_unchained.bundles.security.**UserManager**
ModelManager for the *User* model.

Role

class flask_unchained.bundles.security.**Role**(***kwargs*)
Base role model. Includes an `name` column and a many-to-many relationship with the *User* model via the intermediary *UserRole* join table.

RoleManager

class flask_unchained.bundles.security.**RoleManager**
ModelManager for the *Role* model.

UserRole

class flask_unchained.bundles.security.**UserRole**(*user=None*, *role=None*, ***kwargs*)
Join table between the *User* and *Role* models.

7.10.6 Serializers

UserSerializer

class flask_unchained.bundles.security.serializers.**UserSerializer**
Marshmallow serializer for the *User* model.

RoleSerializer

class flask_unchained.bundles.security.serializers.**RoleSerializer**(**args*, ***kwargs*)

Marshmallow serializer for the *Role* model.

7.10.7 Services

SecurityService

class flask_unchained.bundles.security.**SecurityService**(*mail: Optional[flask_unchained.bundles.mail.extensions.mail.* = *None*)

The security service.

Contains shared business logic that doesn't belong in controllers, but isn't so low level that it belongs in *SecurityUtilsService*.

login_user(*user: flask_unchained.bundles.security.models.user.User*, *remember: Optional[bool]* = *None*, *duration: Optional[datetime.timedelta]* = *None*, *force: bool* = *False*, *fresh: bool* = *True*) → bool

Logs a user in. You should pass the actual user object to this. If the user's *is_active* property is `False`, they will not be logged in unless *force* is `True`.

This will return `True` if the log in attempt succeeds, and `False` if it fails (i.e. because the user is inactive).

Parameters

- **user** (*object*) – The user object to log in.
- **remember** (*bool*) – Whether to remember the user after their session expires. Defaults to `False`.
- **duration** (`datetime.timedelta`) – The amount of time before the remember cookie expires. If `None` the value set in the settings is used. Defaults to `None`.
- **force** (*bool*) – If the user is inactive, setting this to `True` will log them in regardless. Defaults to `False`.
- **fresh** (*bool*) – setting this to `False` will log in the user with a session marked as not "fresh". Defaults to `True`.

logout_user()

Logs out the current user and cleans up the remember me cookie (if any).

Sends signal *identity_changed* (from flask_principal). Sends signal *user_logged_out* (from flask_login).

register_user(*user*, *allow_login=None*, *send_email=None*, *_force_login_without_confirmation=False*)

Service method to register a user.

Sends signal *user_registered*.

Returns True if the user has been logged in, False otherwise.

change_password(*user*, *password*, *send_email=None*)

Service method to change a user's password.

Sends signal *password_changed*.

Parameters

- **user** – The *User*'s password to change.

- **password** – The new password.
- **send_email** – Whether or not to override the config option SECURITY_SEND_PASSWORD_CHANGED_EMAIL and force either sending or not sending an email.

reset_password(*user*, *password*)
Service method to reset a user's password. The same as *change_password()* except we this method sends a different notification email.

Sends signal *password_reset*.

Parameters

- **user** –
- **password** –

Returns

send_email_confirmation_instructions(*user*)
Sends the confirmation instructions email for the specified user.

Sends signal *confirm_instructions_sent*.

Parameters **user** – The user to send the instructions to.

send_reset_password_instructions(*user*)
Sends the reset password instructions email for the specified user.

Sends signal *reset_password_instructions_sent*.

Parameters **user** – The user to send the instructions to.

confirm_user(*user*)
Confirms the specified user. Returns False if the user has already been confirmed, True otherwise.

Parameters **user** – The user to confirm.

send_mail(*subject*, *to*, *template*, ***template_ctx*)
Utility method to send mail with the *mail* template context.

SecurityUtilsService

class flask_unchained.bundles.security.**SecurityUtilsService**
The security utils service. Mainly contains lower-level encryption/token handling code.

get_hmac(*password*)
Returns a Base64 encoded HMAC+SHA512 of the password signed with the salt specified by SECURITY_PASSWORD_SALT.

Parameters **password** – The password to sign.

get_auth_token(*user*)
Returns the user's authentication token.

verify_password(*user*, *password*)
Returns True if the password is valid for the specified user.

Additionally, the hashed password in the database is updated if the hashing algorithm happens to have changed.

Parameters

- **user** – The user to verify against

- **password** – The plaintext password to verify

hash_password(*password*)
Hash the specified plaintext password.

It uses the configured hashing options.

Parameters **password** – The plaintext password to hash

hash_data(*data*)
Hash data in the security token hashing context.

verify_hash(*hashed_data*, *compare_data*)
Verify a hash in the security token hashing context.

use_double_hash(*password_hash=None*)
Return a bool indicating whether a password should be hashed twice.

generate_confirmation_token(*user*)
Generates a unique confirmation token for the specified user.

Parameters **user** – The user to work with

confirm_email_token_status(*token*)
Returns the expired status, invalid status, and user of a confirmation token. For example:

```
expired, invalid, user = confirm_email_token_status('...')
```

Parameters **token** – The confirmation token

generate_reset_password_token(*user*)
Generates a unique reset password token for the specified user.

Parameters **user** – The user to work with

reset_password_token_status(*token*)
Returns the expired status, invalid status, and user of a password reset token. For example:

```
expired, invalid, user, data = reset_password_token_status('...')
```

Parameters **token** – The password reset token

get_token_status(*token*, *serializer*, *max_age=None*, *return_data=False*)
Get the status of a token.

Parameters

- **token** – The token to check
- **serializer** – The name of the serializer. Can be one of the following: `confirm`, `login`, `reset`
- **max_age** – The name of the max age config option. Can be one of the following: `SECURITY_CONFIRM_EMAIL_WITHIN` or `SECURITY_RESET_PASSWORD_WITHIN`

get_within_delta(*key*)
Get a timedelta object from the application configuration following the internal convention of:

```
<Amount of Units> <Type of Units>
```

Examples of valid config values:

```
5 days
10 minutes
```

Parameters **key** – The config value key

7.10.8 Forms

LoginForm

class flask_unchained.bundles.security.forms.**LoginForm**(**args*, ***kwargs*)
The default login form.

validate()
Validate the form by calling `validate` on each field. Returns `True` if validation passes.

If the form defines a `validate_<fieldname>` method, it is appended as an extra validator for the field's `validate`.

Parameters **extra_validators** – A dict mapping field names to lists of extra validator methods to run. Extra validators run after validators passed when creating the field. If the form has `validate_<fieldname>`, it is the last extra validator.

RegisterForm

class flask_unchained.bundles.security.forms.**RegisterForm**(**args*, ***kwargs*)
The default register form.

ChangePasswordForm

class flask_unchained.bundles.security.forms.**ChangePasswordForm**(**args*, ***kwargs*)
The default change password form.

validate()
Validate the form by calling `validate` on each field. Returns `True` if validation passes.

If the form defines a `validate_<fieldname>` method, it is appended as an extra validator for the field's `validate`.

Parameters **extra_validators** – A dict mapping field names to lists of extra validator methods to run. Extra validators run after validators passed when creating the field. If the form has `validate_<fieldname>`, it is the last extra validator.

ForgotPasswordForm

class flask_unchained.bundles.security.forms.**ForgotPasswordForm**(**args*, ***kwargs*)
The default forgot password form.

ResetPasswordForm

class flask_unchained.bundles.security.forms.**ResetPasswordForm**(**args*, ***kwargs*)

The default reset password form.

SendConfirmationForm

class flask_unchained.bundles.security.forms.**SendConfirmationForm**(**args*, ***kwargs*)

The default resend confirmation email form.

validate()

Validate the form by calling `validate` on each field. Returns `True` if validation passes.

If the form defines a `validate_<fieldname>` method, it is appended as an extra validator for the field's `validate`.

Parameters **extra_validators** – A dict mapping field names to lists of extra validator methods to run. Extra validators run after validators passed when creating the field. If the form has `validate_<fieldname>`, it is the last extra validator.

Validators

flask_unchained.bundles.security.forms.**password_equal**(*form*, *field*)

Compares the values of two fields.

Parameters

- **fieldname** – The name of the other field to compare to.
- **message** – Error message to raise in case of a validation error. Can be interpolated with *%(other_label)s* and *%(other_name)s* to provide a more helpful error.

flask_unchained.bundles.security.forms.**new_password_equal**(*form*, *field*)

Compares the values of two fields.

Parameters

- **fieldname** – The name of the other field to compare to.
- **message** – Error message to raise in case of a validation error. Can be interpolated with *%(other_label)s* and *%(other_name)s* to provide a more helpful error.

flask_unchained.bundles.security.forms.**unique_user_email**(*form*, *field*)

flask_unchained.bundles.security.forms.**valid_user_email**(*form*, *field*)

7.11 Session Bundle API

flask_unchained.bundles.session

SessionBundle	The Session Bundle.

flask_unchained.bundles.session.config

DefaultFlaskConfigForSessions	Default configuration options for sessions in Flask.
Config	Default configuration options for the Session Bundle.

flask_unchained.bundles.session.hooks

RegisterSessionModelHook	If using `sqlalchemy` as the SESSION_TYPE, register the `Session` model with the SQLAlchemy Bundle.

7.11.1 SessionBundle

class flask_unchained.bundles.session.**SessionBundle**
The Session Bundle. Integrates Flask Session with Flask Unchained.

7.11.2 Config

class flask_unchained.bundles.session.config.**DefaultFlaskConfigForSessions**
Default configuration options for sessions in Flask.

SESSION_COOKIE_NAME = 'session'
The name of the session cookie.

Defaults to `'session'`.

SESSION_COOKIE_DOMAIN = None
The domain for the session cookie. If this is not set, the cookie will be valid for all subdomains of `SERVER_NAME`.

Defaults to `None`.

SESSION_COOKIE_PATH = None
The path for the session cookie. If this is not set the cookie will be valid for all of `APPLICATION_ROOT` or if that is not set for '/'.

Defaults to `None`.

SESSION_COOKIE_HTTPONLY = True
Controls if the cookie should be set with the `httponly` flag. Browsers will not allow JavaScript access to cookies marked as `httponly` for security.

Defaults to `True`.

SESSION_COOKIE_SECURE = False
Controls if the cookie should be set with the `secure` flag. Browsers will only send cookies with requests over HTTPS if the cookie is marked `secure`. The application must be served over HTTPS for this to make sense.

Defaults to `False`.

PERMANENT_SESSION_LIFETIME = datetime.timedelta(31)
The lifetime of a permanent session as `datetime.timedelta` object or an integer representing seconds.

Defaults to 31 days.

SESSION_COOKIE_SAMESITE = None
Restrict how cookies are sent with requests from external sites. Limits the scope of the cookie such that it

will only be attached to requests if those requests are "same-site". Can be set to `'Lax'` (recommended) or `'Strict'`.

Defaults to `None`.

SESSION_REFRESH_EACH_REQUEST = True
Controls the set-cookie behavior. If set to `True` a permanent session will be refreshed each request and get their lifetime extended, if set to `False` it will only be modified if the session actually modifies. Non permanent sessions are not affected by this and will always expire if the browser window closes.

Defaults to `True`.

class flask_unchained.bundles.session.config.**Config**
Default configuration options for the Session Bundle.

See Flask Session for more information.

SESSION_TYPE = 'null'
Specifies which type of session interface to use. Built-in session types:

- `'null'`: `NullSessionInterface` (default)
- `'redis'`: `RedisSessionInterface`
- `'memcached'`: `MemcachedSessionInterface`
- `'filesystem'`: `FileSystemSessionInterface`
- `'mongodb'`: `MongoDBSessionInterface`
- `'sqlalchemy'`: `SqlAlchemySessionInterface`

Defaults to `'null'`.

SESSION_PERMANENT = True
Whether use permanent session or not.

Defaults to `True`.

SESSION_USE_SIGNER = False
Whether sign the session cookie sid or not. If set to `True`, you have to set `SECRET_KEY`.

Defaults to `False`.

SESSION_KEY_PREFIX = 'session:'
A prefix that is added before all session keys. This makes it possible to use the same backend storage server for different apps.

Defaults to `'session:'`.

SESSION_REDIS = None
A `redis.Redis` instance.

By default, connect to `127.0.0.1:6379`.

SESSION_MEMCACHED = None
A `memcached.Client` instance.

By default, connect to `127.0.0.1:11211`.

SESSION_FILE_DIR = '/home/docs/checkouts/readthedocs.org/user_builds/flask-unchained/c
The folder where session files are stored.

Defaults to using a folder named `flask_sessions` in your current working directory.

SESSION_FILE_THRESHOLD = 500
The maximum number of items the session stores before it starts deleting some.

Defaults to 500.

SESSION_FILE_MODE = 384
The file mode wanted for the session files. Should be specified as an octal, eg `0o600`.

Defaults to `0o600`.

SESSION_MONGODB = None
A `pymongo.MongoClient` instance.

By default, connect to `127.0.0.1:27017`.

SESSION_MONGODB_DB = 'flask_session'
The MongoDB database you want to use.

Defaults to `'flask_session'`.

SESSION_MONGODB_COLLECT = 'sessions'
The MongoDB collection you want to use.

Defaults to `'sessions'`.

SESSION_SQLALCHEMY = <SQLAlchemyUnchained engine=None>
A `SQLAlchemy` extension instance.

SESSION_SQLALCHEMY_TABLE = 'flask_sessions'
The name of the SQL table you want to use.

Defaults to `flask_sessions`.

SESSION_SQLALCHEMY_MODEL = None
Set this if you need to customize the `BaseModel` subclass used for storing sessions in the database.

7.11.3 RegisterSessionModelHook

class flask_unchained.bundles.session.hooks.**RegisterSessionModelHook**(*unchained: flask_unchained.unchained.Unc bundle: Optional[flask_unchained.bundles. = None*)
If using `sqlalchemy` as the SESSION_TYPE, register the `Session` model with the SQLAlchemy Bundle.

name: str = 'register_session_model'
The name of this hook.

run_hook(*app: flask_unchained.flask_unchained.FlaskUnchained, bundles: List[flask_unchained.bundles.Bundle], unchained_config: Optional[Dict[str, Any]] = None*) → None
If using `sqlalchemy` as the *SESSION_TYPE*, register the `Session` model with the SQLAlchemy Bundle.

update_shell_context(*ctx: dict*)
Add the `Session` model to the CLI shell context if necessary.

7.12 SQLAlchemy Bundle API

flask_unchained.bundles.sqlalchemy

SQLAlchemyBundle	The SQLAlchemy Bundle.

flask_unchained.bundles.sqlalchemy.config

Config	The default configuration options for the SQLAlchemy Bundle.
TestConfig	Default configuration options for testing.

flask_unchained.bundles.sqlalchemy.extensions

SQLAlchemyUnchained	The *SQLAlchemyUnchained* extension.

flask_unchained.bundles.sqlalchemy.hooks

RegisterModelsHook	Discovers SQLAlchemy models.

flask_unchained.bundles.sqlalchemy.services

SessionManager	The database session manager service.
ModelManager	Base class for database model manager services.

flask_unchained.bundles.sqlalchemy.sqla

Column	Overridden to make nullable False by default
attach_events	Class decorator for SQLAlchemy models to attach listeners for class methods decorated with *on()*
on	Class method decorator for SQLAlchemy models.
slugify	Class decorator to specify a field to slugify.
foreign_key	Helper method to add a foreign key column to a model.

flask_unchained.bundles.sqlalchemy.forms

ModelForm	Base class for SQLAlchemy model forms.

7.12.1 SQLAlchemyBundle

class flask_unchained.bundles.sqlalchemy.**SQLAlchemyBundle**

The SQLAlchemy Bundle. Integrates SQLAlchemy and Flask-Migrate with Flask Unchained.

name: str = 'sqlalchemy_bundle'

The name of the SQLAlchemy Bundle.

command_group_names = ['db']

Click groups for the SQLAlchemy Bundle.

models = None
A lookup of model classes keyed by class name.

7.12.2 Config

class flask_unchained.bundles.sqlalchemy.config.**Config**
The default configuration options for the SQLAlchemy Bundle.

SQLALCHEMY_DATABASE_URI = 'sqlite:///db/development.sqlite'
The database URI that should be used for the connection. Defaults to using SQLite with the database file stored at `ROOT_PATH/db/<env>.sqlite`. See the SQLAlchemy Dialects documentation for more info.

SQLALCHEMY_TRANSACTION_ISOLATION_LEVEL = None
Set the engine-wide transaction isolation level.

See the docs for more info.

SQLALCHEMY_ECHO = False
If set to `True` SQLAlchemy will log all the statements issued to stderr which can be useful for debugging.

SQLALCHEMY_TRACK_MODIFICATIONS = False
If set to `True`, Flask-SQLAlchemy will track modifications of objects and emit signals. The default is `False`. This requires extra memory and should be disabled if not needed.

SQLALCHEMY_RECORD_QUERIES = None
Can be used to explicitly disable or enable query recording. Query recording automatically happens in debug or testing mode. See `get_debug_queries()` for more information.

SQLALCHEMY_BINDS = None
A dictionary that maps bind keys to SQLAlchemy connection URIs.

SQLALCHEMY_NATIVE_UNICODE = None
Can be used to explicitly disable native unicode support. This is required for some database adapters (like PostgreSQL on some Ubuntu versions) when used with improper database defaults that specify encoding-less databases.

SQLALCHEMY_POOL_SIZE = None
The size of the database pool. Defaults to the engine's default (usually 5).

SQLALCHEMY_POOL_TIMEOUT = None
Specifies the connection timeout in seconds for the pool.

SQLALCHEMY_POOL_RECYCLE = None
Number of seconds after which a connection is automatically recycled. This is required for MySQL, which removes connections after 8 hours idle by default. Note that Flask-SQLAlchemy automatically sets this to 2 hours if MySQL is used.

Certain database backends may impose different inactive connection timeouts, which interferes with Flask-SQLAlchemy's connection pooling.

By default, MariaDB is configured to have a 600 second timeout. This often surfaces hard to debug, production environment only exceptions like `2013: Lost connection to MySQL server` during query.

If you are using a backend (or a pre-configured database-as-a-service) with a lower connection timeout, it is recommended that you set *SQLALCHEMY_POOL_RECYCLE* to a value less than your backend's timeout.

SQLALCHEMY_MAX_OVERFLOW = None
Controls the number of connections that can be created after the pool reached its maximum size. When those additional connections are returned to the pool, they are disconnected and discarded.

SQLALCHEMY_COMMIT_ON_TEARDOWN = False
Whether or not to automatically commit on app context teardown. Defaults to False.

ALEMBIC = {'script_location': 'db/migrations'}
Used to set the directory where migrations are stored. *ALEMBIC* should be set to a dictionary, using the key *script_location* to set the directory. Defaults to `ROOT_PATH/db/migrations`.

ALEMBIC_CONTEXT = {'render_item': <function render_migration_item>, 'template_args':
Extra kwargs to pass to the constructor of the Flask-Migrate extension. If you need to change this, make sure to merge the defaults with your settings!

class flask_unchained.bundles.sqlalchemy.config.**TestConfig**
Default configuration options for testing.

SQLALCHEMY_DATABASE_URI = 'sqlite://'
The database URI to use for testing. Defaults to SQLite in memory.

7.12.3 The SQLAlchemy Extension

class flask_unchained.bundles.sqlalchemy.**SQLAlchemyUnchained**(*app=None*, *use_native_unicode=True*, *session_options=None*, *metadata=None*, *query_class=<class 'flask_sqlalchemy_unchained.BaseQuery'>*, *model_class=<class 'flask_unchained.bundles.sqlalchemy.base_m*

The *SQLAlchemyUnchained* extension:

```
from flask_unchained.bundles.sqlalchemy import db
```

Provides aliases for common SQLAlchemy stuffs:

sqlalchemy.schema: Columns & Tables

Column	Represents a column in a database table.
Computed	Defines a generated column, i.e.
ColumnDefault	A plain default value on a column.
DefaultClause	A DDL-specified DEFAULT column value.
FetchedValue	A marker for a transparent database-side default.
ForeignKey	Defines a dependency between two columns.
Index	A table-level INDEX.
Sequence	Represents a named database sequence.
Table	Represent a table in a database.

sqlalchemy.schema: Constraints

CheckConstraint	A table- or column-level CHECK constraint.
Constraint	A table-level SQL constraint.
ForeignKeyConstraint	A table-level FOREIGN KEY constraint.
PrimaryKeyConstraint	A table-level PRIMARY KEY constraint.
UniqueConstraint	A table-level UNIQUE constraint.

sqlalchemy.types: Column types

BigInteger	A type for bigger `int` integers.
Boolean	A bool datatype.
Date	A type for `datetime.date()` objects.
DateTime	A type for `datetime.datetime()` objects.
Enum	Generic Enum Type.
Float	Type representing floating point types, such as `FLOAT` or `REAL`.
Integer	A type for `int` integers.
Interval	A type for `datetime.timedelta()` objects.
LargeBinary	A type for large binary byte data.
Numeric	A type for fixed precision numbers, such as `NUMERIC` or `DECIMAL`.
PickleType	Holds Python objects, which are serialized using pickle.
SmallInteger	A type for smaller `int` integers.
String	The base for all string and character types.
Text	A variably sized string type.
Time	A type for `datetime.time()` objects.
TypeDecorator	Allows the creation of types which add additional functionality to an existing type.
Unicode	A variable length Unicode string type.
UnicodeText	An unbounded-length Unicode string type.

relationship helpers

association_proxy	Return a Python property implementing a view of a target attribute which references an attribute on members of the target.
declared_attr	Mark a class-level method as representing the definition of a mapped property or special declarative member name.
foreign_key	Helper method to add a foreign key column to a model.
hybrid_method	A decorator which allows definition of a Python object method with both instance-level and class-level behavior.
hybrid_property	A decorator which allows definition of a Python descriptor with both instance-level and class-level behavior.
relationship	Provide a relationship between two mapped classes.

sqlalchemy.types: SQL types

ARRAY	Represent a SQL Array type.
BIGINT	The SQL BIGINT type.
BINARY	The SQL BINARY type.
BLOB	The SQL BLOB type.
BOOLEAN	The SQL BOOLEAN type.
CHAR	The SQL CHAR type.

Continued on next page

Table 84 – continued from previous page

CLOB	The CLOB type.
DATE	The SQL DATE type.
DATETIME	The SQL DATETIME type.
DECIMAL	The SQL DECIMAL type.
FLOAT	The SQL FLOAT type.
INT	alias of sqlalchemy.sql.sqltypes.INTEGER
INTEGER	The SQL INT or INTEGER type.
JSON	Represent a SQL JSON type.
NCHAR	The SQL NCHAR type.
NUMERIC	The SQL NUMERIC type.
NVARCHAR	The SQL NVARCHAR type.
REAL	The SQL REAL type.
SMALLINT	The SQL SMALLINT type.
TEXT	The SQL TEXT type.
TIME	The SQL TIME type.
TIMESTAMP	The SQL TIMESTAMP type.
VARBINARY	The SQL VARBINARY type.
VARCHAR	The SQL VARCHAR type.

sqlalchemy.schema

DDL	A literal DDL statement.
MetaData	A collection of _schema.Table objects and their associated schema constructs.
ThreadLocalMetaData	A MetaData variant that presents a different bind in every thread.

sqlalchemy.sql.expression

alias	Return an _expression.Alias object.
all_	Produce an ALL expression.
and_	Produce a conjunction of expressions joined by AND.
any_	Produce an ANY expression.
asc	Produce an ascending ORDER BY clause element.
between	Produce a BETWEEN predicate clause.
bindparam	Produce a “bound expression”.
case	Produce a CASE expression.
cast	Produce a CAST expression.
collate	Return the clause expression COLLATE collation.
column	Produce a ColumnClause object.
delete	Construct _expression.Delete object.
desc	Produce a descending ORDER BY clause element.
distinct	Produce an column-expression-level unary DISTINCT clause.
except_	Return an EXCEPT of multiple selectables.
except_all	Return an EXCEPT ALL of multiple selectables.
exists	Construct a new _expression.Exists against an existing _expression.Select object.

Continued on next page

Table 86 – continued from previous page

extract	Return a Extract construct.
false	Return a False_ construct.
func	Generate SQL function expressions.
funcfilter	Produce a FunctionFilter object against a function.
insert	Construct an _expression.Insert object.
intersect	Return an INTERSECT of multiple selectables.
intersect_all	Return an INTERSECT ALL of multiple selectables.
join	Produce a _expression.Join object, given two _expression.FromClause expressions.
lateral	Return a _expression.Lateral object.
literal	Return a literal clause, bound to a bind parameter.
literal_column	Produce a ColumnClause object that has the **:paramref:`_expression.column.is_literal`** flag set to True.
not_	Return a negation of the given clause, i.e.
null	Return a constant Null construct.
nullsfirst	Produce the NULLS FIRST modifier for an ORDER BY expression.
nullslast	Produce the NULLS LAST modifier for an ORDER BY expression.
or_	Produce a conjunction of expressions joined by OR.
outerjoin	Return an OUTER JOIN clause element.
outparam	Create an 'OUT' parameter for usage in functions (stored procedures), for databases which support them.
over	Produce an Over object against a function.
select	Construct a new _expression.Select.
subquery	Return an _expression.Alias object derived from a _expression.Select.
table	Produce a new _expression.TableClause.
tablesample	Return a _expression.TableSample object.
text	Construct a new _expression.TextClause clause, representing a textual SQL string directly.
true	Return a constant True_ construct.
tuple_	Return a Tuple.
type_coerce	Associate a SQL expression with a particular type, without rendering CAST.
union	Return a UNION of multiple selectables.
union_all	Return a UNION ALL of multiple selectables.
update	Construct an _expression.Update object.
within_group	Produce a WithinGroup object against a function.

init_app(*app*)

This callback can be used to initialize an application for the use with this database setup. Never use a database in the context of an application not initialized that way or connections will leak.

7.12.4 RegisterModelsHook

class flask_unchained.bundles.sqlalchemy.hooks.**RegisterModelsHook**(*unchained: flask_unchained.unchained.Unchain bundle: Optional[flask_unchained.bundles.Bund = None*)

Discovers SQLAlchemy models.

name: str = 'models'
The name of this hook.

bundle_module_names: Union[List[str], Tuple[str, ...], None] = ['models']
The default module this hook loads from.

Override by setting the `models_module_names` attribute on your bundle class.

process_objects(*app: flask_unchained.flask_unchained.FlaskUnchained, objects: Dict[str, Any]*) → None
Finalize SQLAlchemy model mappings from the registry.

type_check(*obj: Any*) → bool
Returns True if `obj` is a concrete subclass of `BaseModel`.

update_shell_context(*ctx: dict*)
Add models to the CLI shell context.

import_bundle_modules(*bundle*)
Safe-import the modules in a bundle for this hook to load from.

7.12.5 Services

class flask_unchained.bundles.sqlalchemy.**SessionManager**
The database session manager service.

class flask_unchained.bundles.sqlalchemy.**ModelManager**
Base class for database model manager services.

7.12.6 ModelForm

class flask_unchained.bundles.sqlalchemy.forms.**ModelForm**(**args*, ***kwargs*)
Base class for SQLAlchemy model forms.

validate()
Validate the form by calling `validate` on each field. Returns `True` if validation passes.

If the form defines a `validate_<fieldname>` method, it is appended as an extra validator for the field's `validate`.

Parameters **extra_validators** – A dict mapping field names to lists of extra validator methods to run. Extra validators run after validators passed when creating the field. If the form has `validate_<fieldname>`, it is the last extra validator.

7.12.7 SQLAlchemy

Column

class flask_unchained.bundles.sqlalchemy.sqla.**Column**(**args*, *nullable=False*, ***kwargs*)

Overridden to make nullable False by default

foreign_key

flask_unchained.bundles.sqlalchemy.sqla.**foreign_key**(**args*, *fk_col: Optional[str] = None*, *primary_key: bool = False*, *nullable: bool = False*, *ondelete: Optional[str] = None*, *onupdate: Optional[str] = None*, ***kwargs*) → flask_unchained.bundles.sqlalchemy.sqla.column.Column

Helper method to add a foreign key column to a model.

For example:

```
class Post(db.Model):
    category_id = db.foreign_key('Category')
    category = db.relationship('Category', back_populates='posts')
```

Is equivalent to:

```
class Post(db.Model):
    category_id = db.Column(db.BigInteger, db.ForeignKey('category.id'),
                            nullable=False)
    category = db.relationship('Category', back_populates='posts')
```

Customizing all the things:

```
class Post(db.Model):
    _category_id = db.foreign_key('category_id',  # db column name
                                  db.String,      # db column type
                                  'categories',   # foreign table name
                                  fk_col='pk')    # foreign key col name
```

Is equivalent to:

```
class Post(db.Model):
    _category_id = db.Column('category_id',
                             db.String,
                             db.ForeignKey('categories.pk'),
                             nullable=False)
```

Parameters **args** – *foreign_key()* takes up to three positional arguments.

Most commonly, you will only pass one argument, which should be the model name, the model class, or table name you're linking to. If you want to customize the column name the foreign key gets stored in the database under, then *it must be the first string argument*, and you must *also* supply the model name, class or table name. You can also customize the column type (eg `sa.Integer` or `sa.String(36)`) by passing it as an arg.

Parameters

- **fk_col** (*str*) – The column name of the primary key on the *opposite* side of the relationship (defaults to `sqlalchemy_unchained.ModelRegistry.default_primary_key_column`).
- **primary_key** (*bool*) – Whether or not this `Column` is a primary key.
- **nullable** (*bool*) – Whether or not this `Column` should be nullable.
- **kwargs** – Any other kwargs to pass the `Column` constructor.

events

`flask_unchained.bundles.sqlalchemy.sqla.`**`on`**(**args*, ***listen_kwargs*)

Class method decorator for SQLAlchemy models. Must be used in conjunction with the *attach_events()* class decorator

Usage:

```
@attach_events
class Post(Model):
    uuid = Column(String(36))
    post_tags = relationship('PostTag', back_populates='post')  # m2m

    # instance event (only one positional argument, the event name)
    # kwargs are passed on to the sqlalchemy.event.listen function
    @on('init', once=True)
    def generate_uuid(self, args, kwargs):
        self.uuid = str(uuid.uuid4())

    # attribute event (two positional args, field name and event name)
    @on('post_tags', 'append')
    def set_tag_order(self, post_tag, initiating_event):
        if not post_tag.order:
            post_tag.order = len(self.post_tags) + 1
```

`flask_unchained.bundles.sqlalchemy.sqla.`**`attach_events`**(**args*)

Class decorator for SQLAlchemy models to attach listeners for class methods decorated with *on()*

Usage:

```
@attach_events
class User(Model):
    email = Column(String(50))

    @on('email', 'set')
    def lowercase_email(self, new_value, old_value, initiating_event):
        self.email = new_value.lower()
```

`flask_unchained.bundles.sqlalchemy.sqla.`**`slugify`**(*field_name*, *slug_field_name=None*, *mutable=False*)

Class decorator to specify a field to slugify. Slugs are immutable by default unless mutable=True is passed.

Usage:

```
@slugify('title')
def Post(Model):
    title = Column(String(100))
    slug = Column(String(100))
```

(continues on next page)

(continued from previous page)

```
# pass a second argument to specify the slug attribute field:
@slugify('title', 'title_slug')
def Post(Model)
    title = Column(String(100))
    title_slug = Column(String(100))

# optionally set mutable to True for a slug that changes every time
# the slugified field changes:
@slugify('title', mutable=True)
def Post(Model):
    title = Column(String(100))
    slug = Column(String(100))
```

7.13 Webpack Bundle API

flask_unchained.bundles.webpack

WebpackBundle	The Webpack Bundle.

flask_unchained.bundles.webpack.config

Config	Default configuration options for the Webpack Bundle.
ProdConfig	Default production configuration options for the Webpack Bundle.
StagingConfig	Inherit production settings.

flask_unchained.bundles.webpack.extensions

Webpack	The *Webpack* extension.

7.13.1 WebpackBundle

class flask_unchained.bundles.webpack.**WebpackBundle**
The Webpack Bundle.

name: str = 'webpack_bundle'
The name of the Webpack Bundle.

7.13.2 Config

class flask_unchained.bundles.webpack.config.**Config**
Default configuration options for the Webpack Bundle.

WEBPACK_MANIFEST_PATH = None
The full path to the `manifest.json` file generated by Webpack Manifest Plugin.

class flask_unchained.bundles.webpack.config.**ProdConfig**
Default production configuration options for the Webpack Bundle.

WEBPACK_ASSETS_HOST = ''
The host where Webpack assets are served from. Defaults to the same server as the backend.

class flask_unchained.bundles.webpack.config.**StagingConfig**
Inherit production settings.

7.13.3 Extensions

class flask_unchained.bundles.webpack.**Webpack**
The *Webpack* extension:

```
from flask_unchained.bundles.webpack import webpack
```

webpack_asset_url(*asset*)
Returns the Webpack URL for the given asset (a key from manifest.json).

style_tag(*href_or_bundle_name*)
Include a stylesheet.

script_tag(*src_or_bundle_name*)
Include a script.

CHAPTER

EIGHT

CHANGELOG

8.1 v0.8.1 (2021/01/17)

8.1.1 Features

- upgrade Flask-Admin templates to bootstrap4
- add Admin-specific post login/logout redirect endpoints
- add default Model Admins for the User and Role models

8.1.2 Bug Fixes

- fix default config settings for `ADMIN_LOGIN_ENDPOINT` and `ADMIN_LOGOUT_ENDPOINT`
- defer initialization of the Admin extension to fix template overriding
- do not register duplicate templates folder for single-module app bundle

8.1.3 Breaking Changes

- rename `User.active` to `User.is_active` for compatibility with Flask-Login v0.5

8.2 v0.8.0 (2020/12/20)

8.2.1 Features

- **experimental asyncio support by installing Quart**
- **major improvements to `AppFactory` and `AppFactoryHook`**
  - **support single-file app bundles** (just export the app bundle as `UNCHAINED`)
  - support using a custom subclass of `FlaskUnchained` using `AppFactory.APP_CLASS`
  - support using a custom subclass of `AppFactory`
  - support passing all kwargs to `Flask` by setting the same names upper-cased in `unchained_config`
  - support automatic defaults for the Flask app kwargs `root_path`, `template_folder`, `static_folder`, and `static_url_path`
  - support extending and overriding hooks with the same consistent object-oriented patterns

  - support using a custom module name for `unchained_config` by setting the `UNCHAINED` environment variable
  - make it possible to define multiple modules hooks should load from (excluding config and routes, as those only make sense to live inside a single module within bundles)
    * very experimental: add `Bundle.default_load_from_module_name` to ease migration from single-file app bundles to individual modules for different types (ie grouped by base class)
- set up automatic dependency injection on commands (use `from flask_unchained.cli import cli, click` and define command groups for your commands using `@cli.group()`)
- add `flask unchained config` command for listing the current config (optionally filtered by bundle)
- add `flask unchained extensions` command for listing extensions discovered by the app
- add `flask unchaiend services` command for listing services discovered by the app
- alias `flask.abort` (`werkzeug.exceptions.abort`) as `flask_unchained.abort`
- alias `flask_wtf.csrf.generate_csrf` as `flask_unchained.generate_csrf`
- alias `flask.Request` and `flask.Response` into `flask_unchained`
- support `Accept` headers for handling responses in the API bundle
- allow customizing the endpoint prefix for controllers using `Controller.Meta.endpoint_prefix`

8.2.2 General Improvements

- document the rest of SQLAlchemy's config options
- automatically discover services in the `services` and `managers` modules of bundles
- bump sqlalchemy-unchained to v0.11.0
- add compatibility with pytest 5

8.2.3 Bug Fixes

- fix grouping routes by which bundle they're from
- fix registration of resource method routes so the order is deterministic
- fix `ConfigureAppHook` to load configs from every bundle in the hierarchy, not just the top-most one
- fix resolving extension initiation order to only happen once instead of twice
- fix passing explicit rule overrides to `routes.resource`
- fix automatic endpoint names for resource routes using the default implementations for create/list/get/delete/patch/put
- fix using default url rule from view function when no explicit rule passed to `func`
- fix `flask urls` command when no URLs found
- make sure hooks don't resolve local proxies
- fix `ModelResource.Meta.url_prefix` to use `Meta.model.__name__` instead of the resource's class name
- allow using `AnonymousUser` as if it were a SQLAlchemy model in queries

8.2.4 Breaking Changes

- rename `flask_unchained.BaseService` to `flask_unchained.Service`
- rename `PROJECT_ROOT` to `ROOT_PATH` for consistency with upstream `Flask`
- rename `Bundle.folder` to `Bundle.root_path` for consistency with `Flask`
- rename `Controller.Meta.template_folder_name` to `Controller.Meta.template_folder` for consistency with `Flask`
- `AppFactory` is now a `Singleton` that must be instantiated (ie change `AppFactory.create_app(env)` to `AppFactory().create_app(env)` in `wsgi.py`)
- no longer automatically set up dependency injection on all the methods from classes (you can still decorate them manually with `unchained.inject()`, but the preferred approach is to use class attributes to define what to inject into classes)
- default endpoint name for simple view functions is now just the function name
- rename Resource method name constants to reduce confusion with HTTP method names
- remove `AppBundleConfig.ROOT_PATH` and `AppBundleConfig.APP_ROOT` as they didn't always work correctly (use `BundleConfig.current_app.root_path` instead)
- moved `flask_unchained.commands.utils.print_table` to `flask_unchained.cli.print_table`
- if using the API bundle, require `marshmallow>=3.0`, `marshmallow-sqlalchemy>=0.23`, and `flask-marshmallow>=0.12`
- if using the SQLAlchemy bundle, require `sqlalchemy-unchained>=0.10`
- CSRF protection is no longer enabled by default. To re-enable it:

```
from flask_unchained import BundleConfig, unchained, generate_csrf

class Config(BundleConfig):
    SECRET_KEY = 'some-secret-key'
    WTF_CSRF_ENABLED = True

class TestConfig(Config):
    WTF_CSRF_ENABLED = False

@unchained.after_request
def set_csrf_token_cookie(response):
    if response:
        response.set_cookie('csrf_token', generate_csrf())
    return response
```

- customizing bundle module locations changed:

```
class YourBundle(Bundle):
    extensions_module_name = 'custom'  # before
    extensions_module_names = ['custom']  # after

    services_module_name = 'custom'  # before
    services_module_names = ['custom']  # after

    commands_module_name = 'custom'  # before
    commands_module_names = ['custom']  # after
```

(continues on next page)

(continued from previous page)

```
    blueprints_module_name = 'custom'  # before
    blueprints_module_names = ['custom']  # after

    models_module_name = 'custom'  # before
    models_module_names = ['custom']  # after

    admins_module_name = 'custom'  # before
    admins_module_names = ['custom']  # after

    resources_module_name = 'custom'  # before
    model_resources_module_names = ['custom']  # after

    serializers_module_name = 'custom'  # before
    model_serializers_module_names = ['custom']  # after

    celery_tasks_module_name = 'custom'  # before
    celery_tasks_module_names = ['custom']  # after

    graphene_queries_module_name = 'custom'  # before
    graphene_queries_module_names = ['custom']  # after

    graphene_mutations_module_name = 'custom'  # before
    graphene_mutations_module_names = ['custom']  # after

    graphene_types_module_name = 'custom'  # before
    graphene_types_module_names = ['custom']  # after
```

8.2.5 Internals

- move `Bundle` and `AppBundle` into the `flask_unchained.bundles` module
- move `BundleBlueprint` into the `flask_unchained.bundles.controller.bundle_blueprint` module
- move `_DeferredBundleFunctions` into `flask_unchained.unchained`, rename it to `DeferredBundleFunctions`
- make a bunch more protected internal classes public
- make `_has_views`, `_blueprint_names`, `_static_folders`, `is_top_bundle` and `_has_hierarchy_name_conflicts` methods on `Bundle` properties
- rename double-negative `reverse_mro` parameter for `Bundle._iter_class_hierarchy` to `mro`
- warn when identical routes are registered
- make `AppFactory` methods `load_bundles`, `load_bundle` and `is_bundle` classmethods
- add a noop `ViewsHook` to consolidate logic for defining and loading `views_module_names`
- move `param_converter` into the controller bundle

8.3 v0.7.9 (2019/05/19)

- compatibility with `sqlalchemy-unchained >= 0.7.6`

8.4 v0.7.8 (2019/04/21)

- bump required `alembic` version to 1.0.9, fixes `immutabledict is not defined` error

8.5 v0.7.7 (2019/04/11)

- bump requirements
- change behavior of `flask new project` command to use defaults unless `--prompt` is given
- NOTE: broken with sqlalchemy bundle, must install alembic using:

```
pip install git+https://github.com/sqlalchemy/alembic.
↪git@d46de05b8b3281a85e6b107ef3f3407e232eb9e9#egg=alembic
```

8.6 v0.7.6 (2019/03/24)

- NOTE: broken with sqlalchemy bundle, must install alembic using:

```
pip install git+https://github.com/sqlalchemy/alembic.
↪git@d46de05b8b3281a85e6b107ef3f3407e232eb9e9#egg=alembic
```

8.7 v0.7.5 (2019/03/24)

- NOTE: broken with sqlalchemy bundle, must install alembic using:

```
pip install git+https://github.com/sqlalchemy/alembic.
↪git@d46de05b8b3281a85e6b107ef3f3407e232eb9e9#egg=alembic
```

8.8 v0.7.4 (2019/03/17)

- support injecting current app config into services
- extend the `string` url parameter converter to support `upper=True/False`
- add `ModelForm.make_instance` convenience method
- fix `ModelForm.name` to return `bytes`
- add `.gitignore` to `flask new project` command
- improve error message when no `config` module found in the app bundle

8.9 v0.7.3 (2019/02/26)

- do not generate `celery_app.py` for new projects without the celery bundle enabled
- improve user warnings when mail bundle is enabled but lxml or beautifulsoup isn't installed
- bump required versions of py-meta-utils and sqlalchemy-unchained

8.10 v0.7.2 (2019/02/25)

- fix the project's registered name on PyPI so it doesn't contain spaces

8.11 v0.7.1 (2019/02/04)

8.11.1 Bug fixes

- support multiple routing rules with the same endpoint per view function
- fix type error in dependency injection when comparing parameter values with string

8.12 v0.7.0 (2019/01/30)

8.12.1 Features

- **fire:OAuth:fire** support with the new OAuth Bundle (many thanks to @chriamue!)
- **fire:GraphQL:fire** support with the new Graphene Bundle
- add support for specifying parameters to inject into classes as class attributes
- when using `unchained.inject()` on a class, or subclassing a class that supports automatic dependency injection, all non-dunderscore methods now support having dependencies injected
- the `include` function used in `routes.py` now supports specifying the url prefix as the first argument
- support distributing and loading database fixture files with/from bundles
- implement proper support for `ModelForm` (it now adds fields for columns by default)

Configuration Improvements

- add a way for bundle configs to get access to the current app-under-construction
- make options in `app.config` accessible as attributes, eg `app.config.SECRET_KEY` is now the same as `app.config['SECRET_KEY']`
- apply any settings from the app bundle config not already present in `app.config` as defaults before loading bundles

8.12.2 General

- improve documentation of how Flask Unchained works
- update to py-meta-utils 0.7.4 and sqlalchemy-unchained 0.7.0
- update to marshmallow 2.16
- update to marshmallow-sqlalchemy 0.15

8.12.3 Breaking Changes

- move database fixture loading code into the `py_yaml_fixtures` package (which is now a bundle as of v0.4.0)
- consolidate `unchained.get_extension_local_proxy` and `unchained.get_service_local_proxy` into a single function, `unchained.get_local_proxy`
- rename `AppConfig` to `BundleConfig`
- rename the `SQLAlchemy` extension class to `SQLAlchemyUnchained`
- rename `flask_unchained.bundles.sqlalchemy.model_form` to `flask_unchained.bundles.sqlalchemy.forms`
- rename the Graphene Bundle's `QueryObjectType` to `QueriesObjectType` and `MutationObjectType` to `MutationsObjectType`
- rename the Security Bundle's `SecurityUtilsService.verify_and_update_password` method to `verify_password`
- (internal) descriptors, metaclasses, meta options, and meta option factories are now protected
- (internal) rename the `flask_unchained.app_config` module to `flask_unchained.config`
- (internal) remove the `Bundle.root_folder` descriptor as it made no sense (`Bundle.folder` is the bundle package's root folder)
- (internal) rename `ConfigPropertyMeta` to `ConfigPropertyMetaclass`

8.12.4 Bug fixes

- fix the `Api` extension so it only generates docs for model resources that are registered with the app
- fix setting of `Route._controller_cls` when controllers extend another concrete controller with routes
- fix `Bundle.static_url_path` descriptor
- specify required minimum package versions in `setup.py`, and pin versions in `requirements.txt`
- fix the `UnchainedModelRegistry.reset` method so it allows using factory_boy from `conftest.py`
- fix the `flask celery` commands so that they gracefully terminate instead of leaving zombie processes running
- fix `param_converter` to allowing converting models from optional query parameters
- add support to graphene for working with SQLAlchemy BigInteger columns

8.13 v0.6.6 (2018/10/09)

- ship `_templates` folder with the distribution so that the `flask new <tempate>` command works when Flask Unchained gets installed via `pip`

8.14 v0.6.0 - v0.6.5 (2018/10/09)

IMPORTANT: these releases are broken, use `v0.6.6`

- export `get_boolean_env` from core `flask_unchained` package
- export `param_converter` from core `flask_unchained` package
- fix discovery of user-app `tests._unchained_config`
- improve the output of commands that display their information in a table
- improve the output of custom sqlalchemy types in generated migrations files
- improve the output of the Views column from the `flask urls` command
- improve the `--order-by` option of the `flask urls` command
- rename command `flask db import_fixtures` to `flask db import-fixtures`
- add a `FlaskForm` base class extending `FlaskForm` that adds support for specifying the rendered field order
- automatically set the `csrf_token` cookie on responses
- override the `click` module to also support documenting arguments with `help`
  - also make the default help options `-h` and `--help` instead of just `--help`
- refactor the hook store to be a class attribute of the bundle the hook(s) belong to
- add an `env` attribute on the `FlaskUnchained` app instance
- make the `bundles` attribute on the `Unchained` extension an `AttrDict`
  - bundles are now instantiated instead of passing the classes around directly
- add default config options, making `DEBUG` and `TESTING` unnecessary to set manually
- add a `_name` attribute to `FlaskForm` to automatically name forms when rendering them programmatically
- add `get_extension_local_proxy` and `get_service_local_proxy` methods to the `Unchained` extension
- add support for overriding static files from bundles
- minor refactor of the declarative routing for `Controller` and `Resource` classes
  - consolidate default route rule generation into the `Route` class
  - make it possible to override the `member_param` of a `Resource` with the `resource` routes function
- add a `TEMPLATE_FILE_EXTENSION` option to `AppConfig` that controllers will respect by default. Controllers can still set their `template_file_extension` attribute to override the application-wide default.
- implement missing `delete` routing function
- preliminary support for customizing the generated unique member param
- fix setting of `Route._controller_cls` to automatically always happen

- refactor the SQLAlchemy Bundle to split most of it out into its own package, so that it can be used on its own (without Flask).
- fix the resource url prefix descriptor to convert to kebab-case instead of snake-case
- rename `Controller.template_folder` to `Controller.template_folder_name`
- add `Controller.make_response` as an alias for `flask.make_response`
- convert attributes on `Controller`, `Resource`, and `ModelResource` to be `class Meta` options
- rename `_meta` to `Meta` per py-meta-utils v0.3
- rename `ModelManager.find_all` to `ModelManager.all` and `ModelManager.find_by` to `ModelManager.filter_by` for consistency with the `Query` api
- move instantiation of the `CSRFProtect` extension from the security bundle into the controller bundle, where it belongs, so that it always gets used
- improve registration of request cycle functions meant to run only for a specific bundle blueprint
- update `BaseService` to use a MetaOptionsFactory
- make the `ModelManager.model` class attribute a meta option
- rename the `flask db drop --drop` option to `flask db drop --force` to skip prompting
- rename the `flask db reset --reset` option to `flask db reset --force` to skip prompting
- add `no_autoflush` to `SessionManager`

8.15 v0.5.1 (2018/07/25)

- include html templates in the distribution
- add bundles to the shell context

8.16 v0.5.0 (2018/07/25)

- export `FlaskUnchained` from the root package
- export Flask's `current_app` from the root package
- never register static assets routes with babel bundle
- integrate the admin bundle into the `flask_unchained` package
- integrate the api bundle into the `flask_unchained` package
- integrate the celery bundle into the `flask_unchained` package
- integrate the mail bundle into the `flask_unchained` package
- integrate the sqlalchemy bundle into the `flask_unchained` package
- integrate the webpack bundle into the `flask_unchained` package

8.17 v0.4.2 (2018/07/21)

- fix tests when babel_bundle isn't loaded

8.18 v0.4.1 (2018/07/20)

- fix infinite recursion error when registering urls and blueprints with babel

8.19 v0.4.0 (2018/07/20)

- make `tests._unchained_config` optional if `unchained_config` exists
- fix discovery of bundle views to include any bundle in the hierarchy with views
- subclass Flask to improve handling of adding blueprints and url rules in conjunction with the babel bundle
- rename `unchained.BUNDLES` to `unchained.bundles`

8.20 v0.3.2 (2018/07/16)

- fix naming of bundle static endpoints

8.21 v0.3.1 (2018/07/16)

- support loading bundles as the app bundle for development
- refactor the babel commands to work with both app and regular bundles
- fix discovery of tests._unchained_config module

8.22 v0.3.0 (2018/07/14)

- add `flask qtconsole` command
- rename Bundle.iter_bundles to Bundle.iter_class_hierarchy
- add `cli_runner` pytest fixture for testing click commands
- fix register commands hook to support overriding groups and commands
- support registering request hooks, template tags/filters/tests, and context processors via deferred decorators on the Unchained and Bundle classes
- ship the controller bundle, session bundle, and babel bundles as part of core
  - the babel and controller bundles are now mandatory, and will be included automatically

8.23 v0.2.2 (2018/04/08)

- bugfix: Bundle.static_url_prefix renamed to Bundle.static_url_path

8.24 v0.2.1 (2018/04/08)

- bugfix: check for `FunctionType` in `set_up_class_dependency_injection`

8.25 v0.2.0 (2018/04/06)

- rename `BaseConfig` to `Config`
- add utilities for dealing with optional dependencies:
  - `OptionalClass`: generic base class that can also be used as a substitute for extensions that have base classes defined as attributes on them
  - `optional_pytest_fixture`: allows to conditionally register test fixtures
- hooks now declare their dependencies by hook name, as opposed to using an integer priority

8.26 v0.1.x

- early releases

PYTHON MODULE INDEX

f

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W