Pluggable Views

▶ *Changelog*

Flask 0.7 introduces pluggable views inspired by the generic views from Django which are based on classes instead of functions. The main intention is that you can replace parts of the implementations and this way have customizable pluggable views.

Basic Principle

Consider you have a function that loads a list of objects from the database and renders into a template:

```
@app.route('/users/')
def show_users(page):
    users = User.query.all()
    return render_template('users.html', users=users)
```

This is simple and flexible, but if you want to provide this view in a generic fashion that can be adapted to other models and templates as well you might want more flexibility. This is where pluggable class-based views come into place. As the first step to convert this into a class based view you would do this:

```
from flask.views import View

class ShowUsers(View):

    def dispatch_request(self):
        users = User.query.all()
        return render_template('users.html', objects=users)

app.add_url_rule('/users/', view_func=ShowUsers.as_view('show_users'))
```

As you can see what you have to do is to create a subclass of **`flask.views.View`** and implement **`dispatch_request()`**. Then we have to convert that class into an actual view function by using the **`as_view()`** class method. The string you pass to that function is the name of the endpoint that view will then have. But this by itself is not helpful, so let's refactor the code a bit:

```
from flask.views import View

class ListView(View):
```

```
    def get_template_name(self):
        raise NotImplementedError()

    def render_template(self, context):
        return render_template(self.get_template_name(), **context)

    def dispatch_request(self):
        context = {'objects': self.get_objects()}
        return self.render_template(context)

class UserView(ListView):

    def get_template_name(self):
        return 'users.html'

    def get_objects(self):
        return User.query.all()
```

This of course is not that helpful for such a small example, but it's good enough to explain the basic principle. When you have a class-based view the question comes up what `self` points to. The way this works is that whenever the request is dispatched a new instance of the class is created and the **`dispatch_request()`** method is called with the parameters from the URL rule. The class itself is instantiated with the parameters passed to the **`as_view()`** function. For instance you can write a class like this:

```
class RenderTemplateView(View):
    def __init__(self, template_name):
        self.template_name = template_name
    def dispatch_request(self):
        return render_template(self.template_name)
```

And then you can register it like this:

```
app.add_url_rule('/about', view_func=RenderTemplateView.as_view(
    'about_page', template_name='about.html'))
```

Method Hints

Pluggable views are attached to the application like a regular function by either using **`route()`** or better **`add_url_rule()`**. That however also means that you would have to provide the names of the HTTP methods the view supports when you attach this. In order to move that information to the class you can provide a **`methods`** attribute that has this information:

```
class MyView(View):
    methods = ['GET', 'POST']

    def dispatch_request(self):
        if request.method == 'POST':
            ...
        ...

app.add_url_rule('/myview', view_func=MyView.as_view('myview'))
```

Method Based Dispatching

For RESTful APIs it's especially helpful to execute a different function for each HTTP method. With the **`flask.views.MethodView`** you can easily do that. Each HTTP method maps to a function with the same name (just in lowercase):

```
from flask.views import MethodView

class UserAPI(MethodView):

    def get(self):
        users = User.query.all()
        ...

    def post(self):
        user = User.from_form_data(request.form)
        ...

app.add_url_rule('/users/', view_func=UserAPI.as_view('users'))
```

That way you also don't have to provide the **`methods`** attribute. It's automatically set based on the methods defined in the class.

Decorating Views

Since the view class itself is not the view function that is added to the routing system it does not make much sense to decorate the class itself. Instead you either have to decorate the return value of **`as_view()`** by hand:

```
def user_required(f):
    """Checks whether user is logged in or raises error 401."""
    def decorator(*args, **kwargs):
        if not g.user:
            abort(401)
```

```
        return f(*args, **kwargs)
    return decorator

view = user_required(UserAPI.as_view('users'))
app.add_url_rule('/users/', view_func=view)
```

Starting with Flask 0.8 there is also an alternative way where you can specify a list of decorators to apply in the class declaration:

```
class UserAPI(MethodView):
    decorators = [user_required]
```

Due to the implicit self from the caller's perspective you cannot use regular view decorators on the individual methods of the view however, keep this in mind.

Method Views for APIs

Web APIs are often working very closely with HTTP verbs so it makes a lot of sense to implement such an API based on the **`MethodView`**. That said, you will notice that the API will require different URL rules that go to the same method view most of the time. For instance consider that you are exposing a user object on the web:

URL	Method	Description
`/users/`	`GET`	Gives a list of all users
`/users/`	`POST`	Creates a new user
`/users/<id>`	`GET`	Shows a single user
`/users/<id>`	`PUT`	Updates a single user
`/users/<id>`	`DELETE`	Deletes a single user

So how would you go about doing that with the **`MethodView`**? The trick is to take advantage of the fact that you can provide multiple rules to the same view.

Let's assume for the moment the view would look like this:

```
class UserAPI(MethodView):

    def get(self, user_id):
        if user_id is None:
            # return a list of users
            pass
        else:
```

```
            # expose a single user
            pass

    def post(self):
        # create a new user
        pass

    def delete(self, user_id):
        # delete a single user
        pass

    def put(self, user_id):
        # update a single user
        pass
```

So how do we hook this up with the routing system? By adding two rules and explicitly mentioning the methods for each:

```
user_view = UserAPI.as_view('user_api')
app.add_url_rule('/users/', defaults={'user_id': None},
                 view_func=user_view, methods=['GET',])
app.add_url_rule('/users/', view_func=user_view, methods=['POST',])
app.add_url_rule('/users/<int:user_id>', view_func=user_view,
                 methods=['GET', 'PUT', 'DELETE'])
```

If you have a lot of APIs that look similar you can refactor that registration code:

```
def register_api(view, endpoint, url, pk='id', pk_type='int'):
    view_func = view.as_view(endpoint)
    app.add_url_rule(url, defaults={pk: None},
                     view_func=view_func, methods=['GET',])
    app.add_url_rule(url, view_func=view_func, methods=['POST',])
    app.add_url_rule('%s<%s:%s>' % (url, pk_type, pk), view_func=view_f
                     methods=['GET', 'PUT', 'DELETE'])

register_api(UserAPI, 'user_api', '/users/', pk='user_id')
```