Lazily Loading Views

Flask is usually used with the decorators. Decorators are simple and you have the URL right next to the function that is called for that specific URL. However there is a downside to this approach: it means all your code that uses decorators has to be imported upfront or Flask will never actually find your function.

This can be a problem if your application has to import quick. It might have to do that on systems like Google's App Engine or other systems. So if you suddenly notice that your application outgrows this approach you can fall back to a centralized URL mapping.

The system that enables having a central URL map is the **`add_url_rule()`** function. Instead of using decorators, you have a file that sets up the application with all URLs.

Converting to Centralized URL Map

Imagine the current application looks somewhat like this:

```
from flask import Flask
app = Flask(__name__)

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

@app.route('/user/<username>')
def user(username):
    pass
```

Then, with the centralized approach you would have one file with the views (`views.py`) but without any decorator:

```
def index():
    pass

def user(username):
    pass
```

And then a file that sets up an application which maps the functions to URLs:

```
from flask import Flask
from yourapplication import views
app = Flask(__name__)
```

```
app.add_url_rule('/', view_func=views.index)
app.add_url_rule('/user/<username>', view_func=views.user)
```

Loading Late

So far we only split up the views and the routing, but the module is still loaded upfront. The trick is to actually load the view function as needed. This can be accomplished with a helper class that behaves just like a function but internally imports the real function on first use:

```
from werkzeug.utils import import_string, cached_property

class LazyView(object):

    def __init__(self, import_name):
        self.__module__, self.__name__ = import_name.rsplit('.', 1)
        self.import_name = import_name

    @cached_property
    def view(self):
        return import_string(self.import_name)

    def __call__(self, *args, **kwargs):
        return self.view(*args, **kwargs)
```

What's important here is is that *__module__* and *__name__* are properly set. This is used by Flask internally to figure out how to name the URL rules in case you don't provide a name for the rule yourself.

Then you can define your central place to combine the views like this:

```
from flask import Flask
from yourapplication.helpers import LazyView
app = Flask(__name__)
app.add_url_rule('/',
                 view_func=LazyView('yourapplication.views.index'))
app.add_url_rule('/user/<username>',
                 view_func=LazyView('yourapplication.views.user'))
```

You can further optimize this in terms of amount of keystrokes needed to write this by having a function that calls into **`add_url_rule()`** by prefixing a string with the project name and a dot, and by wrapping *view_func* in a *LazyView* as needed.

```
def url(import_name, url_rules=[], **options):
    view = LazyView('yourapplication.' + import_name)
    for url_rule in url_rules:
        app.add_url_rule(url_rule, view_func=view, **options)

# add a single route to the index view
url('views.index', ['/'])

# add two routes to a single function endpoint
url_rules = ['/user/','/user/<username>']
url('views.user', url_rules)
```

One thing to keep in mind is that before and after request handlers have to be in a file that is imported upfront to work properly on the first request. The same goes for any kind of remaining decorator.