Working with the Shell

▶ *Changelog*

One of the reasons everybody loves Python is the interactive shell. It basically allows you to execute Python commands in real time and immediately get results back. Flask itself does not come with an interactive shell, because it does not require any specific setup upfront, just import your application and start playing around.

There are however some handy helpers to make playing around in the shell a more pleasant experience. The main issue with interactive console sessions is that you're not triggering a request like a browser does which means that **`g`**, **`request`** and others are not available. But the code you want to test might depend on them, so what can you do?

This is where some helper functions come in handy. Keep in mind however that these functions are not only there for interactive shell usage, but also for unit testing and other situations that require a faked request context.

Generally it's recommended that you read the The Request Context chapter of the documentation first.

Command Line Interface

Starting with Flask 0.11 the recommended way to work with the shell is the `flask shell` command which does a lot of this automatically for you. For instance the shell is automatically initialized with a loaded application context.

For more information see Command Line Interface.

Creating a Request Context

The easiest way to create a proper request context from the shell is by using the **`test_request_context`** method which creates us a **`RequestContext`**:

```
>>> ctx = app.test_request_context()
```

Normally you would use the `with` statement to make this request object active, but in the shell it's easier to use the **`push()`** and **`pop()`** methods by hand:

```
>>> ctx.push()
```

From that point onwards you can work with the request object until you call *pop*:

```
>>> ctx.pop()
```

Firing Before/After Request

By just creating a request context, you still don't have run the code that is normally run before a request. This might result in your database being unavailable if you are connecting to the database in a before-request callback or the current user not being stored on the **g** object etc.

This however can easily be done yourself. Just call **preprocess_request()**:

```
>>> ctx = app.test_request_context()
>>> ctx.push()
>>> app.preprocess_request()
```

Keep in mind that the **preprocess_request()** function might return a response object, in that case just ignore it.

To shutdown a request, you need to trick a bit before the after request functions (triggered by **process_response()**) operate on a response object:

```
>>> app.process_response(app.response_class())
<Response 0 bytes [200 OK]>
>>> ctx.pop()
```

The functions registered as **teardown_request()** are automatically called when the context is popped. So this is the perfect place to automatically tear down resources that were needed by the request context (such as database connections).

Further Improving the Shell Experience

If you like the idea of experimenting in a shell, create yourself a module with stuff you want to star import into your interactive session. There you could also define some more helper methods for common things such as initializing the database, dropping tables etc.

Just put them into a module (like *shelltools*) and import from there:

```
>>> from shelltools import *
```