The Application Context

The application context keeps track of the application-level data during a request, CLI command, or other activity. Rather than passing the application around to each function, the **`current_app`** and **`g`** proxies are accessed instead.

This is similar to the The Request Context, which keeps track of request-level data during a request. A corresponding application context is pushed when a request context is pushed.

Purpose of the Context

The **`Flask`** application object has attributes, such as **`config`**, that are useful to access within views and CLI commands. However, importing the `app` instance within the modules in your project is prone to circular import issues. When using the app factory pattern or writing reusable blueprints or extensions there won't be an `app` instance to import at all.

Flask solves this issue with the *application context*. Rather than referring to an `app` directly, you use the **`current_app`** proxy, which points to the application handling the current activity.

Flask automatically *pushes* an application context when handling a request. View functions, error handlers, and other functions that run during a request will have access to **`current_app`**.

Flask will also automatically push an app context when running CLI commands registered with **`Flask.cli`** using `@app.cli.command()`.

Lifetime of the Context

The application context is created and destroyed as necessary. When a Flask application begins handling a request, it pushes an application context and a request context. When the request ends it pops the request context then the application context. Typically, an application context will have the same lifetime as a request.

See The Request Context for more information about how the contexts work and the full life cycle of a request.

Manually Push a Context

If you try to access **`current_app`**, or anything that uses it, outside an application context, you'll get this error message:

```
RuntimeError: Working outside of application context.

This typically means that you attempted to use functionality that
needed to interface with the current application object in some way.
To solve this, set up an application context with app.app_context().
```

If you see that error while configuring your application, such as when initializing an extension, you can push a context manually since you have direct access to the `app`. Use **`app_context()`** in a `with` block, and everything that runs in the block will have access to **`current_app`**.

```
def create_app():
    app = Flask(__name__)

    with app.app_context():
        init_db()

    return app
```

If you see that error somewhere else in your code not related to configuring the application, it most likely indicates that you should move that code into a view function or CLI command.

Storing Data

The application context is a good place to store common data during a request or CLI command. Flask provides the **`g object`** for this purpose. It is a simple namespace object that has the same lifetime as an application context.

Note:

The `g` name stands for “global”, but that is referring to the data being global *within a context*. The data on `g` is lost after the context ends, and it is not an appropriate place to store data between requests. Use the **`session`** or a database to store data across requests.

A common use for **`g`** is to manage resources during a request.

1. `get_X()` creates resource `X` if it does not exist, caching it as `g.X`.
2. `teardown_X()` closes or otherwise deallocates the resource if it exists. It is registered as a **`teardown_appcontext()`** handler.

For example, you can manage a database connection using this pattern:

```
from flask import g

def get_db():
    if 'db' not in g:
        g.db = connect_to_database()

    return g.db

@app.teardown_appcontext
def teardown_db():
    db = g.pop('db', None)

    if db is not None:
        db.close()
```

During a request, every call to `get_db()` will return the same connection, and it will be closed automatically at the end of the request.

You can use **`LocalProxy`** to make a new context local from `get_db()`:

```
from werkzeug.local import LocalProxy
db = LocalProxy(get_db)
```

Accessing `db` will call `get_db` internally, in the same way that **`current_app`** works.

If you're writing an extension, **`g`** should be reserved for user code. You may store internal data on the context itself, but be sure to use a sufficiently unique name. The current context is accessed with **`_app_ctx_stack.top`**. For more information see Flask Extension Development.

Events and Signals

The application will call functions registered with **`teardown_appcontext()`** when the application context is popped.

If **`signals_available`** is true, the following signals are sent: **`appcontext_pushed`**, **`appcontext_tearing_down`**, and **`appcontext_popped`**.