The Request Context

The request context keeps track of the request-level data during a request. Rather than passing the request object to each function that runs during a request, the **`request`** and **`session`** proxies are accessed instead.

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

Purpose of the Context

When the **`Flask`** application handles a request, it creates a **`Request`** object based on the environment it received from the WSGI server. Because a *worker* (thread, process, or coroutine depending on the server) handles only one request at a time, the request data can be considered global to that worker during that request. Flask uses the term *context local* for this.

Flask automatically *pushes* a request context when handling a request. View functions, error handlers, and other functions that run during a request will have access to the **`request`** proxy, which points to the request object for the current request.

Lifetime of the Context

When a Flask application begins handling a request, it pushes a request context, which also pushes an The Application Context. When the request ends it pops the request context then the application context.

The context is unique to each thread (or other worker type). **`request`** cannot be passed to another thread, the other thread will have a different context stack and will not know about the request the parent thread was pointing to.

Context locals are implemented in Werkzeug. See Context Locals for more information on how this works internally.

Manually Push a Context

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

```
RuntimeError: Working outside of request context.

This typically means that you attempted to use functionality that
needed an active HTTP request. Consult the documentation on testing
for information about how to avoid this problem.
```

This should typically only happen when testing code that expects an active request. One option is to use the **`test_client`** to simulate a full request. Or you can use **`test_request_context()`** in a `with` block, and everything that runs in the block will have access to **`request`**, populated with your test data.

```
def generate_report(year):
    format = request.args.get('format')
    ...

with app.test_request_context(
        '/make_report/2017', data={'format': 'short'}):
    generate_report()
```

If you see that error somewhere else in your code not related to testing, it most likely indicates that you should move that code into a view function.

For information on how to use the request context from the interactive Python shell, see Working with the Shell.

How the Context Works

The **`Flask.wsgi_app()`** method is called to handle each request. It manages the contexts during the request. Internally, the request and application contexts work as stacks, **`_request_ctx_stack`** and **`_app_ctx_stack`**. When contexts are pushed onto the stack, the proxies that depend on them are available and point at information from the top context on the stack.

When the request starts, a **`RequestContext`** is created and pushed, which creates and pushes an **`AppContext`** first if a context for that application is not already the top context. While these contexts are pushed, the **`current_app`**, **`g`**, **`request`**, and **`session`** proxies are available to the original thread handling the request.

Because the contexts are stacks, other contexts may be pushed to change the proxies during a request. While this is not a common pattern, it can be used in advanced applications to, for example, do internal redirects or chain different applications together.

After the request is dispatched and a response is generated and sent, the request context is popped, which then pops the application context. Immediately before they are popped, the **`teardown_request()`** and **`teardown_appcontext()`** functions are are executed. These execute even if an unhandled exception occurred during dispatch.

Callbacks and Errors

Flask dispatches a request in multiple stages which can affect the request, response, and how errors are handled. The contexts are active during all of these stages.

A **`Blueprint`** can add handlers for these events that are specific to the blueprint. The handlers for a blueprint will run if the blueprint owns the route that matches the request.

1. Before each request, **`before_request()`** functions are called. If one of these functions return a value, the other functions are skipped. The return value is treated as the response and the view function is not called.
2. If the **`before_request()`** functions did not return a response, the view function for the matched route is called and returns a response.
3. The return value of the view is converted into an actual response object and passed to the **`after_request()`** functions. Each function returns a modified or new response object.
4. After the response is returned, the contexts are popped, which calls the **`teardown_request()`** and **`teardown_appcontext()`** functions. These functions are called even if an unhandled exception was raised at any point above.

If an exception is raised before the teardown functions, Flask tries to match it with an **`errorhandler()`** function to handle the exception and return a response. If no error handler is found, or the handler itself raises an exception, Flask returns a generic `500 Internal Server Error` response. The teardown functions are still called, and are passed the exception object.

If debug mode is enabled, unhandled exceptions are not converted to a `500` response and instead are propagated to the WSGI server. This allows the development server to present the interactive debugger with the traceback.

Teardown Callbacks

The teardown callbacks are independent of the request dispatch, and are instead called by the contexts when they are popped. The functions are called even if there is an unhandled exception during dispatch, and for manually pushed contexts. This means there is no guarantee that any other parts of the request dispatch have run first. Be sure to write these functions in a way that does not depend on other callbacks and will not fail.

During testing, it can be useful to defer popping the contexts after the request ends, so that their data can be accessed in the test function. Use the **`test_client()`** as a `with` block to preserve the contexts until the `with` block exits.

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/')
def hello():
    print('during view')
    return 'Hello, World!'

@app.teardown_request
def show_teardown(exception):
    print('after with block')

with app.test_request_context():
    print('during with block')

# teardown functions are called after the context with block exits

with app.test_client() as client:
    client.get('/')
    # the contexts are not popped even though the request ended
    print(request.path)

# the contexts are popped and teardown functions are called after
# the client with block exits
```

Signals

If **`signals_available`** is true, the following signals are sent:

1. **`request_started`** is sent before the **`before_request()`** functions are called.
2. **`request_finished`** is sent after the **`after_request()`** functions are called.
3. **`got_request_exception`** is sent when an exception begins to be handled, but before an **`errorhandler()`** is looked up or called.
4. **`request_tearing_down`** is sent after the **`teardown_request()`** functions are called.

Context Preservation on Error

At the end of a request, the request context is popped and all data associated with it is destroyed. If an error occurs during development, it is useful to delay destroying the data for

debugging purposes.

When the development server is running in development mode (the `FLASK_ENV` environment variable is set to `'development'`), the error and data will be preserved and shown in the interactive debugger.

This behavior can be controlled with the **`PRESERVE_CONTEXT_ON_EXCEPTION`** config. As described above, it defaults to `True` in the development environment.

Do not enable **`PRESERVE_CONTEXT_ON_EXCEPTION`** in production, as it will cause your application to leak memory on exceptions.

Notes On Proxies

Some of the objects provided by Flask are proxies to other objects. The proxies are accessed in the same way for each worker thread, but point to the unique object bound to each worker behind the scenes as described on this page.

Most of the time you don't have to care about that, but there are some exceptions where it is good to know that this object is actually a proxy:

- The proxy objects cannot fake their type as the actual object types. If you want to perform instance checks, you have to do that on the object being proxied.
- The reference to the proxied object is needed in some situations, such as sending Signals or passing data to a background thread.

If you need to access the underlying object that is proxied, use the **`_get_current_object()`** method:

```
app = current_app._get_current_object()
my_signal.send(app)
```