Upgrading to Newer Releases

Flask itself is changing like any software is changing over time. Most of the changes are the nice kind, the kind where you don't have to change anything in your code to profit from a new release.

However every once in a while there are changes that do require some changes in your code or there are changes that make it possible for you to improve your own code quality by taking advantage of new features in Flask.

This section of the documentation enumerates all the changes in Flask from release to release and how you can change your code to have a painless updating experience.

Use the **pip** command to upgrade your existing Flask installation by providing the `--upgrade` parameter:

```
$ pip install --upgrade Flask
```

Version 0.12

Changes to send_file

The `filename` is no longer automatically inferred from file-like objects. This means that the following code will no longer automatically have `X-Sendfile` support, etag generation or MIME-type guessing:

```
response = send_file(open('/path/to/file.txt'))
```

Any of the following is functionally equivalent:

```
fname = '/path/to/file.txt'

# Just pass the filepath directly
response = send_file(fname)

# Set the MIME-type and ETag explicitly
response = send_file(open(fname), mimetype='text/plain')
response.set_etag(...)

# Set `attachment_filename` for MIME-type guessing
# ETag still needs to be manually set
```

```
response = send_file(open(fname), attachment_filename=fname)
response.set_etag(...)
```

The reason for this is that some file-like objects have an invalid or even misleading `name` attribute. Silently swallowing errors in such cases was not a satisfying solution.

Additionally the default of falling back to `application/octet-stream` has been restricted. If Flask can't guess one or the user didn't provide one, the function fails if no filename information was provided.

Version 0.11

0.11 is an odd release in the Flask release cycle because it was supposed to be the 1.0 release. However because there was such a long lead time up to the release we decided to push out a 0.11 release first with some changes removed to make the transition easier. If you have been tracking the master branch which was 1.0 you might see some unexpected changes.

In case you did track the master branch you will notice that **flask --app** is removed now. You need to use the environment variable to specify an application.

Debugging

Flask 0.11 removed the `debug_log_format` attribute from Flask applications. Instead the new `LOGGER_HANDLER_POLICY` configuration can be used to disable the default log handlers and custom log handlers can be set up.

Error handling

The behavior of error handlers was changed. The precedence of handlers used to be based on the decoration/call order of **`errorhandler()`** and **`register_error_handler()`**, respectively. Now the inheritance hierarchy takes precedence and handlers for more specific exception classes are executed instead of more general ones. See Error handlers for specifics.

Trying to register a handler on an instance now raises **`ValueError`**.

Note:

There used to be a logic error allowing you to register handlers only for exceptio
stances. This was unintended and plain wrong, and therefore was replaced with the in-

tended behavior of registering handlers only using exception classes and HTTP error codes.

Templating

The **`render_template_string()`** function has changed to autoescape template variables by default. This better matches the behavior of **`render_template()`**.

Extension imports

Extension imports of the form `flask.ext.foo` are deprecated, you should use `flask_foo`.

The old form still works, but Flask will issue a `flask.exthook.ExtDeprecationWarning` for each extension you import the old way. We also provide a migration utility called flask-ext-migrate that is supposed to automatically rewrite your imports for this.

Version 0.10

The biggest change going from 0.9 to 0.10 is that the cookie serialization format changed from pickle to a specialized JSON format. This change has been done in order to avoid the damage an attacker can do if the secret key is leaked. When you upgrade you will notice two major changes: all sessions that were issued before the upgrade are invalidated and you can only store a limited amount of types in the session. The new sessions are by design much more restricted to only allow JSON with a few small extensions for tuples and strings with HTML markup.

In order to not break people's sessions it is possible to continue using the old session system by using the Flask-OldSessions extension.

Flask also started storing the **`flask.g`** object on the application context instead of the request context. This change should be transparent for you but it means that you now can store things on the `g` object when there is no request context yet but an application context. The old `flask.Flask.request_globals_class` attribute was renamed to **`flask.Flask.app_ctx_globals_class`**.

Version 0.9

The behavior of returning tuples from a function was simplified. If you return a tuple it no longer defines the arguments for the response object you're creating, it's now always a tuple in the form `(response, status, headers)` where at least one item has to be provided. If you depend on the old behavior, you can add it easily by subclassing Flask:

```
class TraditionalFlask(Flask):
    def make_response(self, rv):
        if isinstance(rv, tuple):
            return self.response_class(*rv)
        return Flask.make_response(self, rv)
```

If you maintain an extension that was using **`_request_ctx_stack`** before, please consider changing to **`_app_ctx_stack`** if it makes sense for your extension. For instance, the app context stack makes sense for extensions which connect to databases. Using the app context stack instead of the request context stack will make extensions more readily handle use cases outside of requests.

Version 0.8

Flask introduced a new session interface system. We also noticed that there was a naming collision between `flask.session` the module that implements sessions and **`flask.session`** which is the global session object. With that introduction we moved the implementation details for the session system into a new module called **`flask.sessions`**. If you used the previously undocumented session support we urge you to upgrade.

If invalid JSON data was submitted Flask will now raise a **`BadRequest`** exception instead of letting the default **`ValueError`** bubble up. This has the advantage that you no longer have to handle that error to avoid an internal server error showing up for the user. If you were catching this down explicitly in the past as **`ValueError`** you will need to change this.

Due to a bug in the test client Flask 0.7 did not trigger teardown handlers when the test client was used in a with statement. This was since fixed but might require some changes in your test suites if you relied on this behavior.

Version 0.7

In Flask 0.7 we cleaned up the code base internally a lot and did some backwards incompatible changes that make it easier to implement larger applications with Flask. Because we want to make upgrading as easy as possible we tried to counter the problems arising from these changes by providing a script that can ease the transition.

The script scans your whole application and generates a unified diff with changes it assumes are safe to apply. However as this is an automated tool it won't be able to find all use cases and it might miss some. We internally spread a lot of deprecation warnings all over the place to make it easy to find pieces of code that it was unable to upgrade.

We strongly recommend that you hand review the generated patchfile and only apply the chunks that look good.

If you are using git as version control system for your project we recommend applying the patch with `path -p1 < patchfile.diff` and then using the interactive commit feature to only apply the chunks that look good.

To apply the upgrade script do the following:

1. Download the script: flask-07-upgrade.py
2. Run it in the directory of your application:

```
$ python flask-07-upgrade.py > patchfile.diff
```

3. Review the generated patchfile.
4. Apply the patch:

```
$ patch -p1 < patchfile.diff
```

5. If you were using per-module template folders you need to move some templates around. Previously if you had a folder named `templates` next to a blueprint named `admin` the implicit template path automatically was `admin/index.html` for a template file called `templates/index.html`. This no longer is the case. Now you need to name the template `templates/admin/index.html`. The tool will not detect this so you will have to do that on your own.

Please note that deprecation warnings are disabled by default starting with Python 2.7. In order to see the deprecation warnings that might be emitted you have to enabled them with the **`warnings`** module.

If you are working with windows and you lack the `patch` command line utility you can get it as part of various Unix runtime environments for windows including cygwin, msysgit or ming32. Also source control systems like svn, hg or git have builtin support for applying unified diffs as generated by the tool. Check the manual of your version control system for more information.

Bug in Request Locals

Due to a bug in earlier implementations the request local proxies now raise a **`RuntimeError`** instead of an **`AttributeError`** when they are unbound. If you caught these exceptions with **`AttributeError`** before, you should catch them with **`RuntimeError`** now.

Additionally the **`send_file()`** function is now issuing deprecation warnings if you depend on functionality that will be removed in Flask 0.11. Previously it was possible to use etags and mimetypes when file objects were passed. This was unreliable and caused issues for a few setups. If you get a deprecation warning, make sure to update your application to work with either filenames there or disable etag attaching and attach them yourself.

Old code:

```
return send_file(my_file_object)
return send_file(my_file_object)
```

New code:

```
return send_file(my_file_object, add_etags=False)
```

Upgrading to new Teardown Handling

We streamlined the behavior of the callbacks for request handling. For things that modify the response the **`after_request()`** decorators continue to work as expected, but for things that absolutely must happen at the end of request we introduced the new **`teardown_request()`** decorator. Unfortunately that change also made after-request work differently under error conditions. It's not consistently skipped if exceptions happen whereas previously it might have been called twice to ensure it is executed at the end of the request.

If you have database connection code that looks like this:

```
@app.after_request
def after_request(response):
    g.db.close()
    return response
```

You are now encouraged to use this instead:

```
@app.teardown_request
def after_request(exception):
    if hasattr(g, 'db'):
        g.db.close()
```

On the upside this change greatly improves the internal code flow and makes it easier to customize the dispatching and error handling. This makes it now a lot easier to write unit tests as you can prevent closing down of database connections for a while. You can take advantage of the fact that the teardown callbacks are called when the response context is removed from the stack so a test can query the database after request handling:

```
with app.test_client() as client:
    resp = client.get('/')
    # g.db is still bound if there is such a thing

# and here it's gone
```

Manual Error Handler Attaching

While it is still possible to attach error handlers to **`Flask.error_handlers`** it's discouraged to do so and in fact deprecated. In general we no longer recommend custom error handler attaching via assignments to the underlying dictionary due to the more complex internal handling to support arbitrary exception classes and blueprints. See **`Flask.errorhandler()`** for more information.

The proper upgrade is to change this:

```
app.error_handlers[403] = handle_error
```

Into this:

```
app.register_error_handler(403, handle_error)
```

Alternatively you should just attach the function with a decorator:

```
@app.errorhandler(403)
def handle_error(e):
    ...
```

(Note that **`register_error_handler()`** is new in Flask 0.7)

Blueprint Support

Blueprints replace the previous concept of "Modules" in Flask. They provide be[illegible] tics for various features and work better with large applications. The update script provid-

ed should be able to upgrade your applications automatically, but there might be some cases where it fails to upgrade. What changed?

- Blueprints need explicit names. Modules had an automatic name guessing scheme where the shortname for the module was taken from the last part of the import module. The upgrade script tries to guess that name but it might fail as this information could change at runtime.
- Blueprints have an inverse behavior for **`url_for()`**. Previously `.foo` told **`url_for()`** that it should look for the endpoint `foo` on the application. Now it means "relative to current module". The script will inverse all calls to **`url_for()`** automatically for you. It will do this in a very eager way so you might end up with some unnecessary leading dots in your code if you're not using modules.
- Blueprints do not automatically provide static folders. They will also no longer automatically export templates from a folder called `templates` next to their location however but it can be enabled from the constructor. Same with static files: if you want to continue serving static files you need to tell the constructor explicitly the path to the static folder (which can be relative to the blueprint's module path).
- Rendering templates was simplified. Now the blueprints can provide template folders which are added to a general template searchpath. This means that you need to add another subfolder with the blueprint's name into that folder if you want `blueprintname/template.html` as the template name.

If you continue to use the `Module` object which is deprecated, Flask will restore the previous behavior as good as possible. However we strongly recommend upgrading to the new blueprints as they provide a lot of useful improvement such as the ability to attach a blueprint multiple times, blueprint specific error handlers and a lot more.

Version 0.6

Flask 0.6 comes with a backwards incompatible change which affects the order of after-request handlers. Previously they were called in the order of the registration, now they are called in reverse order. This change was made so that Flask behaves more like people expected it to work and how other systems handle request pre- and post-processing. If you depend on the order of execution of post-request functions, be sure to change the order.

Another change that breaks backwards compatibility is that context processors will no longer override values passed directly to the template rendering function. If for example `request` is as variable passed directly to the template, the default context processor will not override it with the current request object. This makes it easier to extend context processors later to inject additional variables without breaking existing template n[illegible] them.

Version 0.5

Flask 0.5 is the first release that comes as a Python package instead of a single module. There were a couple of internal refactoring so if you depend on undocumented internal details you probably have to adapt the imports.

The following changes may be relevant to your application:

- autoescaping no longer happens for all templates. Instead it is configured to only happen on files ending with `.html`, `.htm`, `.xml` and `.xhtml`. If you have templates with different extensions you should override the **`select_jinja_autoescape()`** method.
- Flask no longer supports zipped applications in this release. This functionality might come back in future releases if there is demand for this feature. Removing support for this makes the Flask internal code easier to understand and fixes a couple of small issues that make debugging harder than necessary.
- The `create_jinja_loader` function is gone. If you want to customize the Jinja loader now, use the **`create_jinja_environment()`** method instead.

Version 0.4

For application developers there are no changes that require changes in your code. In case you are developing on a Flask extension however, and that extension has a unittest-mode you might want to link the activation of that mode to the new `TESTING` flag.

Version 0.3

Flask 0.3 introduces configuration support and logging as well as categories for flashing messages. All these are features that are 100% backwards compatible but you might want to take advantage of them.

Configuration Support

The configuration support makes it easier to write any kind of application that requires some sort of configuration. (Which most likely is the case for any application out there).

If you previously had code like this:

```
app.debug = DEBUG
app.secret_key = SECRET_KEY
```

You no longer have to do that, instead you can just load a configuration into the config object. How this works is outlined in Configuration Handling.

Logging Integration

Flask now configures a logger for you with some basic and useful defaults. If you run your application in production and want to profit from automatic error logging, you might be interested in attaching a proper log handler. Also you can start logging warnings and errors into the logger when appropriately. For more information on that, read Application Errors.

Categories for Flash Messages

Flash messages can now have categories attached. This makes it possible to render errors, warnings or regular messages differently for example. This is an opt-in feature because it requires some rethinking in the code.

Read all about that in the Message Flashing pattern.