Chapter 12. 11¾ A Little Bit of Threading: Dealing with Waiting

When they said: "wait," I'd no idea this is what they had in mind...

Your code can sometimes take a long time to execute. Depending on who notices, this may or may not be an issue. If some code takes 30 seconds to do its thing "behind the scenes," the wait may not be an issue. However, if your user is waiting for your application to respond, and it takes 30 seconds, everyone notices. What you should do to fix this problem depends on what you're trying to do (and who's doing the waiting). In this short chapter, we'll briefly discuss some options, then look at one solution to the issue at hand: *what happens if something takes too long?*

Waiting: What to Do?

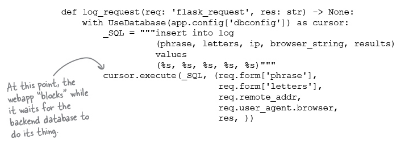
When you write code that has the potential to make your users wait, you need to think carefully about what it is you are trying to do. Let's consider some points of view.

Listen, dude, if you have to wait, there's nothing else for it: you wait...
Maybe how you wait is different when you're writing than when you're reading?
Like everything else, it all depends on what you're trying to do, and the user experience you're shooting for...

Maybe it is the case that waiting for a write is *different* from waiting for a read, especially as it relates to how your webapp works?

Let's take another look at the SQL queries in `log_request` and `view_the_log` to see how you're using them.

In the `log_request` function, we are using an SQL `INSERT` to add details of the request to our backend database. When `log_request` is called, it **waits** while the `INSERT` is executed by `cursor.execute`:



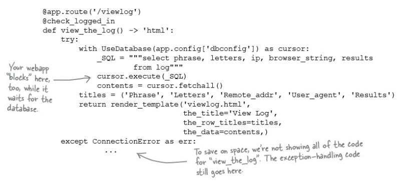
```
def log_request(req: 'flask_request', res: str) -> None:
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into log
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        cursor.execute(_SQL, (req.form['phrase'],
                              req.form['letters'],
                              req.remote_addr,
                              req.user_agent.browser,
                              res, ))
```

At this point, the webapp "blocks" while it waits for the backend database to do its thing.

GEEK BITS

Code that waits for something external to complete is referred to as "blocking code," in that the execution of your program is **blocked** from continuing until the wait is over. As a general rule, blocking code that takes a noticeable length of time is bad.

The same holds for the `view_the_log` function, which also **waits** whenever the SQL `SELECT` query is executed:



```
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """select phrase, letters, ip, browser_string, results
                    from log"""
            cursor.execute(_SQL)
            contents = cursor.fetchall()
        titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
        return render_template('viewlog.html',
                               the_title='View Log',
                               the_row_titles=titles,
                               the_data=contents,)
    except ConnectionError as err:
          ...
```

Your webapp "blocks" here, too, while it waits for the database.
To save on space, we're not showing all of the code for "view_the_log". The exception-handling code still goes here.

Both functions block. However, look closely at what happens *after* the call to `cursor.execute` in both functions. In `log_request`, the `cursor.execute` call is the last thing that function does, whereas in `view_the_log`, the results of `cursor.execute` are used by the rest of the function.

Let's consider the implications of this difference.

Database INSERTs and SELECTs Are Different

If you're reading the title to this page and thinking "Of course they are!", be assured that (this late in this book) we haven't lost our marbles.

Yes: an SQL `INSERT` is *different* from an SQL `SELECT`, but, as it relates to your use of both queries in your webapp, it turns out that the `INSERT` in `log_request` doesn't need to block, whereas the `SELECT` in `view_the_log` does, which makes the queries *very* different.

This is a key observation.

If the `SELECT` in `view_the_log` doesn't wait for the data to return from the backend database, the code that follows `cursor.execute` will likely fail (as it'll have no data to work with). The `view_the_log` function **must** block, as it has to wait for data *before* proceeding.

When your webapp calls `log_request`, it wants the function to log the details of the current web request to the database. The calling code doesn't really care *when* this happens, just that it does. The `log_request` function returns no value, nor data; the calling code isn't waiting for a response. All the calling code cares about is that the web request is logged *eventually*.

Which begs the question: why does `log_request` force its callers to wait?

Are you about to suggest that the "log_request" code could somehow run concurrently with the webapp's code?

Yes. That's our madcap idea.

When users of your webapp enter a new search, they couldn't care less that the request details are logged to some backend database, so let's not make them wait while your webapp does that work.

Instead, let's arrange for some other process to do the logging *eventually* and independently of the webapp's main function (which is to allow your users to perform searches).

Doing More Than One Thing at Once

Here's the plan: you're going to arrange for the `log_request` function to execute independently of your main webapp. To do this, you're going to adjust your webapp's code so each call to `log_request` runs concurrently. This will mean that your webapp no longer has to wait for `log_request` to complete before servicing another request from another user (i.e., no more delays).

If `log_request` takes an instant, a few seconds, a minute, or even hours to execute, your webapp doesn't care (and neither does your user). What you care about is that the code eventually executes.

CONCURRENT CODE: YOU HAVE OPTIONS

When it comes to arranging for some of your application's code to run concurrently, Python has a few options. As well as lots of support from third-party modules, the standard library comes with some built-in goodies that can help here.

One of the most well known is the `threading` library, which provides a high-level interface to the threading implementation provided by the operating system hosting your webapp. To use the library, all you need to do is `import` the `Thread` class from the `threading` module near the top of your program code:

```
from threading import Thread
```

Go ahead and add this line of code near the top of your `vsearch4web.py` file.

For the full list of (and all the details about) Python's standard library concurrency options, see https://docs.python.org/3/library/concurrency.html.

Now the fun starts.

To create a new thread, you create a `Thread` object, assigning the name of the function you want the thread to execute to a named argument called `target`, and providing any arguments as a tuple to another named argument called `args`. The created `Thread` object is then assigned to a variable of your choosing.

As an example, let's assume that you have a function called `execute_slowly`, which takes three arguments, which we'll assume are three numbers. The code that invokes `execute_slowly` has assigned the three values to variables called `glacial`, `plodding`, and `leaden`. Here's how `execute_slowly` is invoked normally (i.e., without our worrying about concurrent execution):

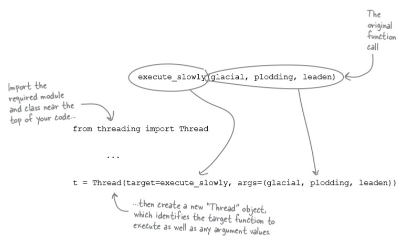
```
execute_slowly(glacial, plodding, leaden)
```

If `execute_slowly` takes 30 seconds to do what it has to do, the calling code blocks and waits for 30 seconds before doing anything else. Bummer.

Don't Get Bummed Out: Use Threads

In the big scheme of things, waiting 30 seconds for the `execute_slowly` function to complete doesn't sound like the end of the world. But, if your user is sitting and waiting, they'll be wondering what's gone wrong.

If your application can continue to run while `execute_slowly` goes about its business, you can create a `Thread` to run `execute_slowly` concurrently. Here's the normal function call once more, together with the code that turns the function call into a request for threaded execution:



```
execute_slowly(glacial, plodding, leaden)

from threading import Thread

        ...

t = Thread(target=execute_slowly, args=(glacial, plodding, leaden))
```

The original function call
Import the required module and class near the top of your code...
...then create a new "Thread" object, which identifies the target function to execute as well as any argument values.

Granted, this use of `Thread` looks a little strange, but it's not really. The key to understanding what's going on here is to note that the `Thread` object has been assigned to a variable (`t` in this example), and that the `execute_slowly` function has yet to execute.

Assigning the `Thread` object to `t` *prepares* it for execution. To ask Python's threading technology to run `execute_slowly`, start the thread like this:

```
t.start()
```

When you call "start", the function associated with the "t" thread is scheduled for execution by the "threading" module.

At this point, the code that called `t.start` continues to run. The 30-second wait that results from running `execute_slowly` has no effect on the calling code, as `execute_slowly`'s execution is handled by Python's `threading` module, not by you. The threading module conspires with the Python interpreter to run `execute_slowly` *eventually*.

SHARPEN YOUR PENCIL

When it comes to calling `log_request` in your webapp code, there's only one place you need to look: in the `do_search` function. Recall that you've already put your call to `log_request` inside a `try/except` to guard against unexpected runtime errors.

Note, too, that we've added a 15-second delay—using `sleep(15)`—to our `log_request` code (making it slow). Here's the current code to `do_search`:

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    try:
        log_request(request, results)
    except Exception as err:
        print('***** Logging failed with this error:',
str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Here's how you currently invoke "log_request".

We are going to assume that you have already added `from threading import Thread` to the top of your webapp's code.

Grab your pencil, and in the space provided below, write the code you'd insert into `do_search` instead of the standard call to `log_request`.

Remember: you are to use a `Thread` object to run `log_request`, just like we did with the `execute_slowly` example from the last page.

Add the threading code you'd use to eventually execute "log_request".

SHARPEN YOUR PENCIL SOLUTION

When it comes to calling `log_request` in your webapp code, there's only one place you need to look: in the `do_search` function. Recall that you've already put your call to `log_request` inside a `try/except` to guard against unexpected run-time errors.

Note, too, that we've added a 15 second delay - using `sleep(15)` - to our `log_request` code (making it slow). Here's the current code to `do_search`:

Here's how you currently invoke "log_request".

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    try:
        log_request(request, results)
    except Exception as err:
        print('***** Logging failed with this error:',
str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

We assumed that you had already added `from threading import Thread` to the top of your webapp's code.

In the space provided below, you were to write the code you'd insert into `do_search` instead of the standard call to `log_request`.

You were to use a `Thread` object to run `log_request`, just like we did with the recent `execute_slowly` example.

We're keeping the "try" statement (for now).

```
try:
    t = Thread(target=log_request, args=(request, results))
    t.start()
except ...
```

The "except" suite is unchanged, so we aren't showing it here.

Just like the earlier example, identify the target function to run, supply any arguments it needs, and don't forget to schedule your thread to run.

TEST DRIVE

With these edits applied to `vsearch4web.py`, you are ready for another test run. What you're expecting to see here is next-to-no wait when you enter a search into your webapp's search page (as the `log_request` code is being run concurrently by the `threading` module).

Go ahead and give it a go.

Sure enough, the instant you click on the "Do it!" button, your webapp returns with your results. The assumption is that the `threading` module is now executing `log_request`, and waiting however long it takes to run that function's code to completion (approximately 15 seconds).

You're just about to give yourself a pat on the back (for a job well done) when, out of nowhere and after about 15 seconds, your webapp's terminal window erupts with error messages, not unlike these:

Take a look at this message.

The last request was a success.

```
    ...
127.0.0.1 - - [29/Jul/2016 19:43:31] "POST /search4 HTTP/1.1" 200 -
Exception in thread Thread-6:
Traceback (most recent call last):
  File "vsearch4web.not.slow.with.threads.but.broken.py", line 42, in log_request
    cursor.execute(_SQL, (req.form['phrase'],
  File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/
werkzeug/local.py", line 343, in __getattr__
        ...
    raise RuntimeError(_request_ctx_err_msg)
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.

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/threading.py",
line 914, in _bootstrap_inner
    self.run()
        ...
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.
```

Whoops! An uncaught exception.

Lots (!!) more traceback messages here

And another one...yikes!

If you check your backend database, you'll learn that the details of your web request were **not** logged. Based on the messages above, it appears the `threading` module isn't at all happy with your code. A lot of the second group of traceback messages refer to `threading.py`, whereas the first group of traceback messages refer to code in the `werkzeug` and `flask` folders. What's clear is that adding in the threading code has resulted in a **huge mess**. What's going on?

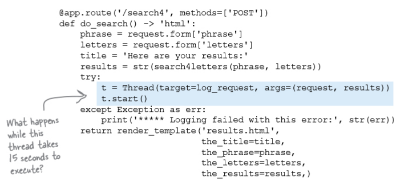
First Things First: Don't Panic

Your first instinct may be to back out the code you added to run `log_request` in its own thread (and get yourself back to a known good state). But let's not panic, and let's **not** do that. Instead, let's take a look at that descriptive paragraph that appeared twice in the traceback messages:

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

This message is coming from Flask, not from the `threading` module. We know this because the `threading` module couldn't care less about what you use it for, and definitely has no interest in what you're trying to do with HTTP.

Let's take another look at the code that schedules the thread for execution, which we know takes 15 seconds to run, as that's how long `log_request` takes. While you're looking at this code, think about what happens during that 15 seconds:



```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    try:
        t = Thread(target=log_request, args=(request, results))
        t.start()
    except Exception as err:
        print('***** Logging failed with this error:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

The instant the thread is scheduled for execution, the calling code (the `do_search` function) continues to execute. The `render_template` function executes (in the blink of an eye), and then the `do_search` function *ends*.

When `do_search` ends, all of the data associated with the function (its *context*) is reclaimed by the interpreter. The variables `request`, `phrase`, `letters`, `title`, and `results` cease to be. However, the `request` and `results` variables are passed as arguments to `log_request`, which tries to access them 15 seconds later. Unfortunately, at that point in time, the variables no longer exist, as `do_search` has ended. Bummer.

Don't Get Bummed Out: Flask Can Help

Based on what you've just learned, it appears the `log_request` function (when executed within a thread) can no longer "see" its argument data. This is due to the fact that the interpreter has long since cleaned up after itself, and reclaimed the memory used by these variables (as `do_search` has ended). Specifically, the `request` object is no longer active, and when `log_request` goes looking for it, it can't be found.

So, what can be done? Don't fret: help is at hand.

I'm just going to pencil you in for next week, when I know you're going to ask me to rewrite the "log_request" function. OK?

There's really no need for a rewrite.

At first glance, it might appear that you'd need to rewrite `log_request` to somehow rely less on its arguments... assuming that's even possible. But it turns out that Flask comes with a decorator that can help here.

The decorator, `copy_current_request_context`, ensures that the HTTP request that is active when a function is called *remains* active even when the function is subsequently executed in a thread. To use it, you need to add `copy_current_request_context` to the list of imports at the top of your webapp's code.

As with any other decorator, you apply it to an existing function using the usual `@` syntax. However, there is a caveat: the function being decorated has to be defined *within* the function that calls it; the decorated function must be nested inside its caller (as an inner function).

EXERCISE

Here's what we want you to do (after updating the list of imports from Flask):

1. Take the `log_request` function and nest it inside the `do_search` function.

2. Decorate `log_request` with `@copy_current_request_context`.

3. Confirm that the runtime errors from the last *Test Drive* have gone away.

EXERCISE SOLUTION

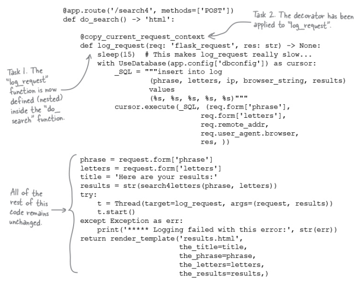
We asked you to do three things:

1. Take the `log_request` function and nest it inside the `do_search` function.

2. Decorate `log_request` with `@copy_current_request_context`.

3. Confirm that the runtime errors from the last *Test Drive* have gone away.

Here's what our `do_search` code looks like after we perform tasks 1 and 2 (note: we'll discuss task 3 over the page):



```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':

    @copy_current_request_context
    def log_request(req: 'flask_request', res: str) -> None:
        sleep(15)  # This makes log_request really slow...
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """insert into log
                    (phrase, letters, ip, browser_string, results)
                    values
                    (%s, %s, %s, %s, %s)"""
            cursor.execute(_SQL, (req.form['phrase'],
                                  req.form['letters'],
                                  req.remote_addr,
                                  req.user_agent.browser,
                                  res, ))

    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    try:
        t = Thread(target=log_request, args=(request, results))
        t.start()
    except Exception as err:
        print('***** Logging failed with this error:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Task 2. The decorator has been applied to "log_request".
Task 1. The "log_request" function is now defined (nested) inside the "do_search" function.
All of the rest of this code remains unchanged.

THERE ARE NO DUMB QUESTIONS

Q:	**Q: Does it still make sense to protect the threaded invocation of `log_request` with `try`/`except`?**
A:	**A:** *Not if you are hoping to react to a runtime issue with* `log_request`, *as the* `try/except` *will have ended before the thread starts. However, your system may fail trying to create a new thread, so we figure it can't hurt to leave* `try/except` *in* `do_search`.

TEST DRIVE

Task 3: Taking this latest version of `vsearch4web.py` for a spin confirms that the runtime errors from the last *Test Drive* are a thing of the past. Your webapp's terminal window confirms that all is well:

```
     ...
127.0.0.1 - - [30/Jul/2016 20:42:46] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [30/Jul/2016 20:43:10] "POST /search4 HTTP/1.1" 200 -
127.0.0.1 - - [30/Jul/2016 20:43:14] "GET /login HTTP/1.1" 200 -
127.0.0.1 - - [30/Jul/2016 20:43:17] "GET /viewlog HTTP/1.1" 200 -
127.0.0.1 - - [30/Jul/2016 20:43:37] "GET /viewlog HTTP/1.1" 200 -
```

No more scary runtime exceptions. All those 200s mean all is well with your webapp. And, 15 seconds after you submit a new search, your webapp eventually logs the details to your backend database WITHOUT requiring your webapp user to wait. ☺

According to this card, I get to ask one last question. Is there any downside to defining "log_request" within "do_search"?
?

No. Not in this case.

For this webapp, the `log_request` function was only ever called by `do_search`, so nesting `log_request`'s within `do_search` isn't an issue.

If you later decide to invoke `log_request` from some other function, you may have an issue (and you'll have to rethink things). But, for now, you're golden.

Is Your Webapp Robust Now?

Here are the four questions posed at the start of Chapter 11:

1. **What happens if the database connection fails?**
2. **Is our webapp protected from web attacks?**
3. **What happens if something takes a long time?**
4. **What happens if a function call fails?**

Your webapp now handles a number of runtime exceptions, thanks to your use of `try`/`except` and some custom exceptions that you can `raise` and catch as required.

When you know something can go wrong at runtime, fortify your code against any exceptions that might occur. This improves the overall robustness of your application, which is a good thing.

Note that there are other areas where robustness could be improved. You spent a lot of time adding `try`/`except` code to `view_the_log`'s code, which took advantage of the `UseDatabase` context manager. `UseDatabase` is *also* used within `log_request`, and should probably be protected, too (and doing so is left as a homework exercise for you).

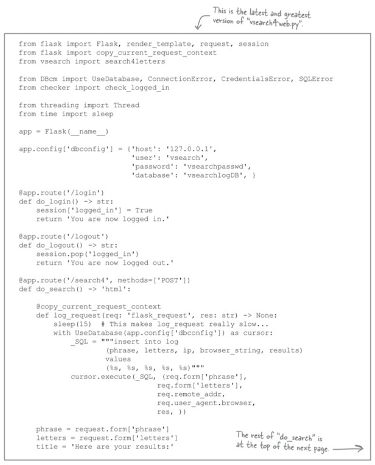
Your webapp is more responsive due to your use of threading to handle a task that has to be performed eventually, but not right away. This is a good design strategy, although you do need to be careful not to go overboard with threads: the threading example in this chapter is very straightforward. However, it is very easy to create threading code that nobody can understand, and which will drive you mad when you have to debug it. **Use threads with care.**

In answering question 3—*what happens if something takes a long time?*—the use of threads improved the performance of the database write, but not the database read. It is a case of just having to wait for the data to arrive after the read, no matter how long it takes, as the webapp wasn't able to proceed without the data.

To make the database read go faster (assuming it's actually slow in the first place), you may have to look at using an alternative (faster) database setup. But that's a worry for another day that we won't concern ourselves with further in this book.

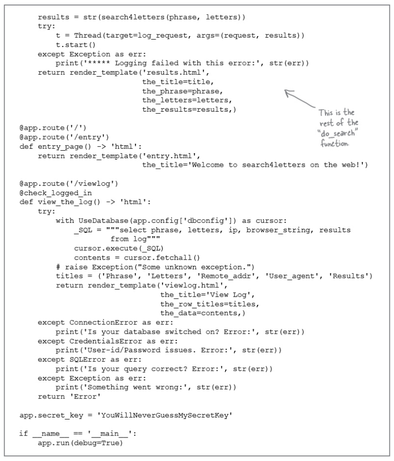
However, having said that, in the next and last chapter, we do indeed consider performance, but we'll be doing so while discussing a topic everyone understands, and which we've already discussed in this book: looping.

Chapter 11¾'s Code, 1 of 2


This is the latest and greatest version of "vsearch4web.py".
The rest of "do_search" is at the top of the next page.

```
from flask import Flask, render_template, request, session
from flask import copy_current_request_context
from vsearch import search4letters

from DBcm import UseDatabase, ConnectionError, CredentialsError, SQLError
from checker import check_logged_in

from threading import Thread
from time import sleep

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':

    @copy_current_request_context
    def log_request(req: 'flask_request', res: str) -> None:
        sleep(15)  # This makes log_request really slow...
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """insert into log
                    (phrase, letters, ip, browser_string, results)
                    values
                    (%s, %s, %s, %s, %s)"""
            cursor.execute(_SQL, (req.form['phrase'],
                                  req.form['letters'],
                                  req.remote_addr,
                                  req.user_agent.browser,
                                  res, ))

    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
```

Chapter 11¾'s Code, 2 of 2


This is the rest of the "do_search" function.

```
    results = str(search4letters(phrase, letters))
    try:
        t = Thread(target=log_request, args=(request, results))
        t.start()
    except Exception as err:
        print('***** Logging failed with this error:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """select phrase, letters, ip, browser_string, results
                    from log"""
            cursor.execute(_SQL)
            contents = cursor.fetchall()
        # raise Exception("Some unknown exception.")
        titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
        return render_template('viewlog.html',
                               the_title='View Log',
                               the_row_titles=titles,
                               the_data=contents,)
    except ConnectionError as err:
        print('Is your database switched on? Error:', str(err))
    except CredentialsError as err:
        print('User-id/Password issues. Error:', str(err))
    except SQLError as err:
        print('Is your query correct? Error:', str(err))
    except Exception as err:
        print('Something went wrong:', str(err))
    return 'Error'

app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
    app.run(debug=True)
```

PREV
11. Exception Handling: What to Do When Things Go Wrong

NEXT
13. Advanced Iteration: Looping Like Crazy