DATABASE INSTRUMENTATION

Django mastery in Nepali

Database instrumentation

To help you understand and control the queries issued by your code, Django provides a hook for installing wrapper functions around the execution of database queries. For example, wrappers can count queries, measure query duration, log queries, or even prevent query execution (them to make sure that no queries are issued while rendering a template with prefetched data). The wrappers are modeled after middleware – they are callable which take another callable as one of them arguments. They call that callable to invoke the (possibly wrapped) database query, and they can do what they want around that call. They are, however, created and installed by user code, and so don't need a separate factory like middleware do. Installing a wrapper is done in a context manager – so the wrappers are temporary and specific to some flow in your code. As mentioned above, an example of a wrapper is a query execution blocker. It could look like this:

```
def blocker(*args):
    raise Exception("No database access allowed
    here.")
```

And it would be used in a view to block queries from the template like so:

```
from django.db import connection
from django.shortcuts import render
def my_view(request):
    context = {...} # Code to generate
    context with all data.
    template_name = ...
    with connection.execute_wrapper(blocker):
        return render(request, template_name,
        context)
```

The parameters sent to the wrappers are:

- execute – a callable, which should be invoked with the rest of the parameters to execute the query.
- sql – a str, the SQL query to be sent to the database.
- params – a list/tuple of parameter values for the SQL command, or a list/tuple of lists/tuples if the wrapped call is executemany().
- many – a bool indicating whether the ultimately invoked call is execute() or executemany() (and whether params is expected to be a sequence of values, or a sequence of sequences of values).
- context – a dictionary with further data about the context of invocation. This includes the connection and cursor.

Using the parameters, a slightly more complex version of the blocker could include the connection name in the error message:

```
def blocker(execute, sql, params, many, context):
    alias = context["connection"].alias
    raise Exception("Access to database '{}' blocked here".format(alias))
```

For a more complete example, a query logger could look like this:

```
import time
class QueryLogger:
    def __init__(self):
        self.queries = []
    def __call__(self, execute, sql, params, many, context):
        current_query = {"sql": sql, "params": params, "many":  many}
        start = time.monotonic()
        try:
            result = execute(sql, params, many, context)
        except Exception as e:
            current_query["status"] = "error"
            current_query["exception"] = e
            raise
        else:
            current_query["status"] = "ok"
            return result
        finally:
            duration = time.monotonic() - start
            current_query["duration"] = duration
            self.queries.append(current_query)
```

To use this, you would create a logger object and install it as a wrapper:

```
from django.db import connection
ql = QueryLogger()
with connection.execute_wrapper(ql):
    do_queries()
    # Now we can print the log.
print(ql.queries)
```

connection.execute_wrapper()

execute_wrapper(wrapper)

Returns a context manager which, when entered, installs a wrapper around database query executions, and when exited, removes the wrapper. The wrapper is installed on the thread-local connection object.
wrapper is a callable taking five arguments. It is called for every query execution in the scope of the context manager, with arguments execute, sql, params, many, and context as described above. It's expected to call execute(sql, params, many, context) and return the return value of that call.

Thanks for watching