nucleon Documentation

Release 0.1

Daniel Pope

December 23, 2014

Contents:

CHAPTER 1

Getting started with Nucleon

Starting a new nucleon app is easy. Just type:

```
$ nucleon new example_project
$ cd example_project
```

to set up a basic nucleon project (obviously, you can use any name instead of `example_project`).

You should then see several files in your project directory:

- `app.py` - the application setup - *views* can be written here.
- `app.cfg` - per-environment *configuration settings*
- `database.sql` - an SQL script to create database tables and initial data
- `tests.py` - a suitable place to write nose tests.

Then just run:

```
$ nucleon start
```

in the same directory to start the server. You should be able to see the nucleon app running by visiting http://localhost:8888/ in your web browser. In the default setup, the version of the application is displayed as a JSON document.

1.1 An example application

Let's go through the process of building a nucleon application. Let's imagine we have a small list of countries that we can trade with, and we want this information to be available as a web service for other services to query.

Having set up an application you can immediately start writing views (functions that process web requests) by editing `app.py`. Let's do that now. After the bootstrapping code that sets up the app is a suitable place to start writing views.

First, let's set up the data we are going to serve:

```
COUNTRIES = {
    'gb': {
        'name': "United Kingdom",
        'language': 'en-GB',
        'currency': 'GBP'
    },
    'fr': {
        'name': "France",
        'language': 'fr-FR',
```

```
        'currency': 'EUR'
    }
}
```

Then we can add a couple of views on this data. First, other services may want to know our country codes:

```
@app.view('/countries/')
def countries(request):
    return COUNTRIES.keys()
```

This view, which can be accessed under `/countries/` (http://localhost:8888/countries/ if you are following along!), is simply a JSON list of country codes!

Another view we might want to support is a view for getting full information on a country. Let's write that view now:

```
from nucleon.http import Http404

@app.view('/countries/([a-z]{2})/')
def country(request, code):
    try:
        return COUNTRIES[code]
    except KeyError:
        raise Http404('No such country with the code %s.' % code)
```

The regular expression in the `app.view()` decorator means that this view will be called to handle requests for `/countries/<code>/` where `code` is a 2-letter country code. For example, we can request `/countries/gb/` and the response JSON document will be

```
{
    "name": "United Kingdom",
    "language": "en-GB",
    "currency": "GBP"
}
```

1.2 Our first database app

Let's now try to write a shared to-do list. Unlike the above application, this will require persistence. nucleon can be integrated with a variety of different NoSQL stores, but particular attention has been paid to its integration with the PostgreSQL database, such that multiple greenlets can execute SQL statements in the database at the same time.

The first thing we should do is open `database.sql` and add write some SQL statements (using any PostgreSQL syntax you like) to configure the required database table:

```
CREATE TABLE tasks (
    id SERIAL PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    description TEXT,
    complete BOOLEAN NOT NULL DEFAULT FALSE;
);
```

We can have nucleon create this table by running:

```
$ nucleon syncdb
```

Writing REST/JSON Applications

Nucleon makes it easy to write JSON/REST applications with relatively little code. The URL routing and view dispatching is modelled on Django's, except that in Nucleon callables are registered directly with an app rather than being automatically loaded from a separate URL configuration module.

Views are simply methods that accept a request object as a parameter and return a response. The response is typically a structure consisting of lists, dicts, strings, etc., that will be served to the user as JSON.

The other type of response is a `nucleon.http.Response` subclass, which allows a wider variety of return values or response codes.

2.1 Processing Requests

2.2 Returning Responses

Nucleon includes several classes that can be used as responses, in addition to returning Python structures to be serialised as JSON. These are all based on WebOb Response:

A subclass of Response is provided as a convenience for returning a response in JSON format. This is necessary if you wish to customise the response - for example, by including different response headers, or to return a different status code.

Another convenience is the ability to raise particular exception classes which will cause Nucleon to serve standard error responses.

CHAPTER 3

PostgreSQL Database Access

Nucleon includes a wrapper for the psycopg2 PostgreSQL driver that shares a pool of database connections between greenlets. The number of open database connections is capped for performance.

3.1 Making database queries

A PostgreSQL connection pool can be retreived from each app's Application object.

When a greenlet wishes to make a database request, it "borrows" a connection from the pool. A context manager interface ensures that the connection is returned to the pool when the greenlet no longer needs it.

3.2 High-level API

A higher level API is available for pre-defining queries that can be executed later. This is intended to save boilerplate and allow queries to be defined in one place - by convention, a separate `queries.py`.

The style of the API is largely declarative; queries can be declared in SQL syntax but can be used as Python callables. For example, declaring and using a query might work as follows:

```
>>> db = Database('database')
>>> get_customer = db.select("SELECT id, name FROM customers WHERE id=%s")
>>> get_customer(52).unique
{'id': 52, 'name': 'Leonard Winter'}
```

The entry point to this high-level API is the `Database` class, which wraps a PostgreSQL connection corresponding to a setting defined in the application *settings file*.

When performing a query, the return value is an object that allows transformation of the results into simple Pythonic forms.

A results instance is also iterable; iterating it is equivalent to iterating `.rows`, except that it does not build a list of all results first.

3.3 Transaction Functions

Sometimes we want to do more processing in Python than is possible using the above approach - we may need to take results into Python code, operate on them, and perform database actions as a result.

We can do this using a “transaction function” - effectively a block of code in which database queries are either all committed or all rolled back. This is written as a decorated function whose first position argument is a callable that can be used to perform queries within the transaction context:

```
db = Database('database')

@db.transaction()
def add_customer(q, name)
    return q('INSERT INTO customers(name) VALUES(%s)', name)
```

Additionally, it is possible to specify that such a transaction be automatically retried a finite number of times - this is useful if there can be integrity problems but the chances of such are relatively low.

For example, we can write a transaction to insert a record with the next highest *id* value as follows (assume *id* has a uniqueness constraint):

```
@db.transaction(retries=3)
def insert_value(q):
    lastid = q('SELECT max(id) FROM test').value
    return q(
        'insert into test(id, name) values(%s, %s)',
        lastid + 1, 'a%s' % lastid
    )
```

This would be retried up to three times (4 attempts total) if there was an integrity error (ie. another client inserts the same id between the SELECT and the INSERT.

CHAPTER 4

Using Signals

Often it is necessary for operations to be performed at particular points in the lifecycle of an application.

To allow the developer to register code to be called at these points in the lifecycle, Nucleon provides a system of signal registration and dispatch.

4.1 Signal Registration

4.2 Predefined Signals

Several signals are predefined that will be called by the nucleon framework at appropriate times during the application lifecycle. They will also be called when at appropriate times when running tests, though the testing lifecycle may be subtly different [1].

`nucleon.signals.on_initialise`
  Fired before the application has started. Callbacks receive no arguments.

`nucleon.signals.on_start`
  Fired when the web application has started and is accepting requests. Callbacks receive no arguments.

4.3 Example Usage

To register a signal handler that logs that the application is accepting requests:

```
import logging
from nucleon.signals import on_start

@on_start
def log_ready():
    logging.info("Application started")
```

[1] In particular, `nucleon.signals.on_start` will always be called before any tests are executed, whereas in production requests may be processed before the `on_start` event is finished dispatching.

CHAPTER 5

Configuring Nucleon Applications

A Nucleon app's configuration is loaded from a file named `app.cfg`, which must reside in the same directory as the application's `app.py`. The configuration file is in standard Python ConfigParser format.

Settings are loaded from sections within the configuration file named by environment - different environments (demo, staging, qa, production, etc.) may well have different connection settings for databases, queues, services and so on. Each application is configured at start time to use settings for a particular environment.

5.1 Selecting an environment

The default environment is called 'default' and so is loaded from the configuration file section named `[default]`.

However, when running tests, settings are loaded from the `test` environment. This switch is currently enacted by calling `nucleon.tests.get_test_app()`.

5.2 Configuration API

In an application, settings for the current environment can be retrieved as properties of a global settings object.

`nucleon.config.`**`settings`**

A global settings object that reflects the current environment.

Config variables are available as properties of this object.

`environment`

The name of the currently active environment.

`_set_environment``(`*environment*`)`

Change to a different environment.

This method will raise ConfigurationError if any settings have been read. The alternative could allow the application to become partially configured for multiple different environments at the same time, and pose a risk of accidental data loss.

For example, reading the currently configured database is as simple as:

```
from nucleon.config import settings
print settings.database
```

5.3 Database access credentials

Databases can be configured for each environment by using the following syntax:

```
[environment]
database = postgres://username:password@host:5432/databasename
```

'database' is not a special name - just the default. Specific database connections can be requested by passing the name of their configuration setting when retrieving a connection pool from the app with `nucleon.framework.Application.get_database()`. Thus a Nucleon app can easily use connections to multiple databases (albeit with a risk of deadlocks if greenlets require exclusive use of connections on multiple databases).

Nucleon can manage the set up of database tables and inserting initial data. This is achieved using the commandline tools - see *Nucleon Management Commands* for full details.

Nucleon Management Commands

Nucleon has a basic command line interface for managing applications and databases.

Most commands operate on the nucleon application in the current directory, but will also work if called from a parent directory.

6.1 Application Management

new `<dest>`

Sets up the initial structure of a nucleon application in the named directory.

start

Start the nucleon application.

By default, nucleon's web service is available on port 8888.

6.2 Database Management

syncdb

Creates any database tables listed in `database.sql` that do not already exist, and also performs INSERTs into the tables that it creates.

resetdb

Runs the `database.sql` script. Any tables that already exist are dropped and re-created.

6.3 Graceful Application Exit

To close nucleon app in production environment please send it a SIGUSR1 message. Within 10 seconds timeout (default timeout) nucleon will:

1. Stop serving new pages. All requests during shutdown will be handled with 503 response.
2. Wait for existing requests to complete.

Coroutine-based concurrency with gevent

Nucleon is tightly integrated with the gevent library, which makes it possible to perform multiple IO operations apparently concurrently, without the overhead of operating system threads.

This model of programming can require a different way of thinking. Specifically, unless a greenlet yields control either explicitly or by blocking on something, it will block everything else happening anywhere else in the nucleon application. For example an operation that takes 1 second to complete prevents any other requests from being served for a whole second, even if those operations would consume a tiny amount of CPU time. This could cause severe performance problems.

On the other hand, blocking network operations are very efficient. This includes database operations, REST calls, or communication with AMQP. gevent patches Python so that instead of blocking, other operations will be processed *as if in the background*. This includes Python operations that block, as well as pure Python libraries that use those operations.

> **Warning: Native libraries can completely block gevent**. If an external library performs some blocking operation, your entire application will grind to a halt. You should identify whether the library supports non-blocking IO or can be integrated with an external IO loop, before attempting to integrate the library.
> You might need to be particularly careful if a library performs I/O as a hidden side-effect of it normal operation. Some XML-processing libraries, for example, may make web requests for DTDs in order to correctly process an XML document.
> You should also watch out for unexpected DNS activity.

To use gevent to best effect you should try to ensure that CPU is used in very short bursts so that the processing of other requests can be interleaved.

7.1 Diagnosing blocking calls

The Linux `strace` command can be used to print out the system calls used by a nucleon application.

```
$ strace -T nucleon start
```

The `-T` option will make strace display the time spent in each system call - pay attention to any calls with particularly large values, other than `epoll_wait()` (which is how gevent stops when all greenlets are blocked).

7.2 Undertaking more expensive processing

If you do need to use more CPU time very rarely, then it's possible to mitigate the impact to other requests running at the same time.

The most direct way to do this is to explicitly yield control from within a greenlet. gevent will run any other greenlets that can run before returning control to the yielding greenlet. This is most similar to conventional threading.

A more elegant way to do this is to use a map-reduce model. In the map phase, a greenlet breaks up a task into many component tasks. These are each put onto a queue. Other greenlets pick up a task and execute them. The results are also put back into a queue. In the reduce phase some greenlet blocks waiting for responses and combines the results. Writing a task in this way can give extremely good scalability.

7.3 In-memory Persistence

Nucleon runs in a single native thread in a single process, with all greenlets sharing the same memory space. Because of this, Nucleon apps can store data in application memory. No synchronisation primitives are required, so long as your application code never performs leaves the memory space in an inconsistent state while blocking IO operations are being performed.

Ensuring this is the case is preferable to using `gevent.coros` classes for locking, as this will simply reduce the number of greenlets eligible to run to completion while the greenlet holding the lock is blocked on I/O.

Changelog

Changes to Nucleon:

8.1 Version 0.1

- Removed AMQP support from core - there is now a new standalone package to provide AMQP support.

8.2 Version 0.0.2

- Now uses gevent v1.0b4
- Removed dependency on Beautiful Soup
- Added support for type and sequences in SQL scripts
- Added transaction support for database API
- Reverted the signature of make_reinitialize_script to avoid backwards-incompatible changes
- Returned the row count for UPDATE and DELETE queries
- Fix: Make nucleon.validation importable
- Allow no-op change of settings environment

8.3 Version 0.0.1-gevent

Initial Version

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*

Symbols

E

N

R

S