supercell Documentation

Release 0.4.0

Daniel Truemper

December 09, 2013

Contents

Supercell is a simple set of classes for creating domain driven RESTful APIs in Python. We use *schematics* for domain modeling, *scales* for statistics and *Tornado* as web server.

A very simple example for a supercell request handler looks like this:

```
from schematics.models import Model
from schematics.types import StringType, IntType

class Saying(Model):

    id = IntType()
    content = StringType()

@s.produces(s.MediaType.ApplicationJson)
class HelloWorld(s.RequestHandler):

    @property
    def counter(self):
        if not hasattr(self.__class__, '_counter'):
            self.__class__._counter = 0
        return self.__class__._counter or 0

    @counter.setter
    def counter(self, value):
        self.__class__._counter = value

    @s.async
    def get(self):
        self.counter += 1
        name = self.get_argument('name', self.config.default_name)
        content = self.render_string(self.config.template, name)
        raise s.Return(Saying(id=self.counter, content=content))
```

The **Getting Started** guide should help you becoming familiar with the ideas behind and the **Topics** contain a growingly part of in depth documentation on certain aspects. The **API** contains the full API documentation.

CHAPTER 1

Contents:

1.1 Getting Started

This guide will help you get started with a simple *Hello World* Supercell project.

1.1.1 Overview

Supercell applications use Tornado as a HTTP server, Schematics for dealing with representations, and Scales for metrics.

1.1.2 Project structure

A typical supercell application is structured in submodules:

- app
  - api - representations
  - core - domain implementation, i.e. crud operatios on representations
  - health - health checks
  - handler - request handler
  - provider - custom providers if any
  - consumer - custom consumers if any
  - service.py - the service class
  - config.py - the configuration

1.1.3 Configuration

Start with creating a *config.py* file:

```
from tornado.options import define

define('default_name', 'Hello %s', help='Default name')
define('template', 'main.html', help='Tornado template file')
```

Here we are only defining the configuration names and their default configuration values. Shortly we will se the different ways to really set the configuraion.

1.1.4 Create a service class

The service class is the part of the application defining it's handlers and startup behaviour. For this purpose we start with a very simple class:

```
import supercell.api as s

class MyService(s.Service):

    def bootstrap(self):
        self.environment.config_file_paths.append('./etc')
        self.environment.config_file_paths.append('/etc/hello-world/')

    def run(self):
        # nothing done yet
        pass

def main():
    MyService().main()
```

This class is not doing too much for now. Basically it only handles the order in which configuration files are being parsed. Right now supercell will parse the following files in that order:

1. $PWD/etc/config.cfg
2. $PWD/etc/$USER_$HOSTNAME.cfg
3. /etc/hello-world/config.cfg
4. /etc/hello-world/$USER_$HOSTNAME.cfg

After all these files were parsed, one may still overwrite the values using the command line parameters.

Assume we have this entry point in the *setup.py*:

```
hello-world = helloworld.service:main
```

we can start the application with something like *hello-world*. In order to debug configuration settings you have the following command line parameters at hand:

```
# see the config file name you have to generate for this machine
$ hello-world --show-config-name

# see the order in which the files would be parsed
$ hello-world --show-config-file-order

# see the effective configuration
$ hello-world --show-config
```

1.1.5 Representation

Now we create the model for the application:

```
from schematics.models import Model
from schematics.types import StringType, IntType

class Saying(Model):

    id = IntType()
    content = StringType()
```

There is nothing special to it assuming you have some knowledge on schematics. We simply have a *Saying* model that contains an *id* as integer and some *content* as a string.

1.1.6 Request Handler

The request handler is very similar to a *Tornado* handler, except it also takes care of de-/serializing in- and output:

```
@s.produces(s.MediaType.ApplicationJson)
class HelloWorld(s.RequestHandler):

    @property
    def counter(self):
        if not hasattr(self.__class__, '_counter'):
            self.__class__._counter = 0
        return self.__class__._counter or 0

    @counter.setter
    def counter(self, value):
        self.__class__._counter = value

    @s.async
    def get(self):
        self.counter += 1
        name = self.get_argument('name', self.config.default_name)
        content = self.render_string(self.config.template, name)
        raise s.Return(Saying(id=self.counter, content=content))
```

Ok, let's get through this example step by step. The *s.produces* decorator tells supercell the content type, that this handler should return. In this case a predefined one (*s.MediaType.ApplicationJson*) that will transform the returned model as *application/json*.

The *counter* property is a simple wrapper around a class level variable that stores the overall counter. Keep in mind that for each request a new instance of the handler class is created, so a simple instance variable would always be *0*.

The *s.async* decorator is a simple wrapper for the two *Tornado* decorators *web.asynchronous* and *gen.coroutine*. With the new *coroutine* decorator *Tornado* can now make use of the *concurrent.Futures* of Python 3.3 and the backported library for Python < 3.

Now we only have to add the request handler to the service implementation:

```
class MyService(s.Service):

    def run(self):
        self.environment.add_handler('/hello-world', HelloWorld)
```

Start the application and point your browser to http://localhost:8080/hello-world to see the response. The *id* is growing on every request and to change the output you may add the *name* parameter: http://localhost:8080/hello-world?name=you

See example/gettingstarted.py for the full example code.

1.2 Topics

1.2.1 Logging

Logging with *supercell* is configured with two simple configuration options: the *logfile* defines the file to which the logs are written. By default it is named *root.log* and does a daily log rotation for 10 days.

The second configuration sets the *loglevel*. This can be on of *DEBUG*, *INFO*, *WARN*, *ERROR*, i.e. any valid default Python *logging.level* value.

The default logging implementation is simply adding a *supercell.SupercellLoggingHandler* and sets the *loglevel* on the root logger. We also disable the *tornado.log* module, so that it does not add its own handler. The *SupercellLoggingHandler* is simply a *TimedRotatingFileHandler* with some default values like number of backups and rotation interval and it sets the logging format.

Custom logging

If the default logging does not fit your need, you may simply overwrite the *initialize_logging* method of your *Service* implementation.

1.2.2 Consumer

Consumers convert the client's input into an internal format defined as a *schematics* model. The mapping of incoming data to one of the available consumers is based on the client's *Content-Type* header. If this is missing and no default consumer is defined, the client will receive a HTTP 406 error (**Content not acceptable**).

Defining consumers for a request handler is done using the *consumes* decorator on the class definition:

```
@s.consumes(s.MediaType.ApplicationJson, model=Model)
class MyHandler(s.RequestHandler):
    pass
```

Create custom consumer

Creating a custom consumer is as easy as subclassing the *ConsumerBase* class. See the *JsonConsumer* for example:

```
class JsonConsumer(ConsumerBase):

    CONTENT_TYPE = ContentType('application/json')

    def consume(self, handler, model):
        return model(**json.loads(handler.request.body))
```

Content Types

The *CONTENT_TYPE* class level variable maps a certain *Content-Type* header to this consumer. The *consume(handler, model)* method converts the request body to an instance of the *model* class.

In situations where you need to accept the same content type, but the model has different versions, you can use the *vendor* and *version* parameters to the content type definition. This allows for multiple consumers for the same serialization format like *json* but different versions of the data. See for example the following two definitions and their respective content type value:

```
ContentType('application/json') == 'application/json'

ContentType('application/json', version='1.1', vendor='corp') == \
    'application/vnd.corp-v1.1+json'
```

If you create two consumers for both content types, the client can decide which version is sent.

1.2.3 Provider

A *Provider* is the equivalent to a *Consumer* only that it transforms the request handler's resulting model into a serialization requested from the client. The client is then able to request a certain serialization using the *Accept* header in her request.

Defining providers for a request handler is done using the *provides* decorator on the class definition:

```
@s.provides(s.MediaType.ApplicationJson)
class MyHandler(s.RequestHandler):
    pass
```

Create custom provider

A provider can be created equivalently to a consumer:

```
class JsonProvider(ConsumerBase):

    CONTENT_TYPE = ContentType(MediaType.ApplicationJson)

    def provide(self, model, handler):
        handler.write(model.validate())
```

For *Producers* the same remarks about the content type hold as for the *Consumers*.

1.3 API

This is the main *supercell* API reference.

1.3.1 Environment

The `Environment` is a container for request handlers, managed objects and other runtime settings as well as the `tornado.web.Application` settings.

There are two cases where you will need to work with the it: during the bootstrapping phase you may change paths to look for configuration files and you will add the request handlers to the environment. In addition to that you can also use it from within a request handler in and access managed objects, such as HTTP clients that can be used accross a number of client libraries for connection pooling, e.g.

class `supercell.api.environment.`**`Environment`**

Environment for **supercell** processes.

`add_handler`(*path*, *handler_class*, *init_dict=None*, *name=None*, *host_pattern='.*$'*, *cache=None*, *expires=None*)

Add a handler to the `tornado.web.Application`.

The environment will manage the available request handlers and managed objects. So in the `Service.run()` method you will add the handlers:

```
class MyService(s.Service):

    def run():
        self.environment.add_handler('/', Handler)
```

Parameters

- **path** (*str or re.pattern*) – The regular expression for the URL path the handler should be bound to.
- **handler_class** (*supercell.api.RequestHandler*) – The request handler class
- **init_dict** (*dict*) – The initialization dict that is passed to the *RequestHandler.initialize()* method.
- **name** (*str*) – If set the handler and its URL will be available in the *RequestHandler.reverse_url()* method.
- **host_pattern** (*str*) – A regular expression for matching the hostname the handler will be bound to. By default this will match all hosts ('.*$')
- **cache** (*supercell.api.cache.CacheConfig*) – Cache info for GET and HEAD requests to this handler defined by `supercell.api.cache.CacheConfig`.
- **expires** (*datetime.timedelta*) – Set the *Expires* header according to the provided timedelta

`add_health_check`(*name*, *check*)

Add a health check to the API.

Parameters

- **name** (*str*) – The name for the health check to add
- **check** (A `supercell.api.RequestHandler`) – The request handler performing the health check

`add_managed_object`(*name*, *instance*)

Add a managed instance to the environment.

A managed object is identified by a name and you can then access it from the environment as an attribute, so in your request handler you may:

```
class MyService(s.Service):

    def run(self):
        managed = HeavyObjectFactory.get_heavy_object()
        self.environment.add_managed_object('managed', managed)

class MyHandler(s.RequestHandler):
```

```
def get(self):
    self.environment.managed
```

Parameters

- **name** (*str*) – The managed object identifier
- **instance** (*object*) – Some arbitrary instance

`config_file_paths`
The list containing all paths to look for config files.

In order to manipulate the paths looked for configuration files just manipulate this list:

```
class MyService(s.Service):

    def bootstrap(self):
        self.environment.config_file_paths.append('/etc/myservice/')
        self.environment.config_file_paths.append('./etc/')
```

`config_name`
Determine the configuration file name for the machine this application is running on.

The filenames are generated using a combination of username and machine name. If you deploy the application as user **webapp** on host **fe01.local.dev** the configuration name would be **webapp_fe01.local.dev.cfg**.

`get_application``(`*config=None*`)`
Create the tornado application.

Parameters **config** – The configuration that will be added to the app

`get_cache_info``(`*handler*`)`
Return the `supercell.api.cache.CacheConfig` for a certain handler.

`get_expires_info``(`*handler*`)`
Return the *timedelta* for a specific handler that should define the *Expires* header for GET and HEAD requests.

`health_checks`
Simple property access for health checks.

`tornado_settings`
The dictionary passed to the `tornado.web.Application` containing all relevant tornado server settings.

1.3.2 Service

A `Service` is the main element of a *supercell* application. It will instanciate the `supercell.api.Environment` and parse the configuration files as well as the command line. In the final step the `tornado.web.Application` is created and bound to a socket.

class `supercell.api.service.`**`Service`**
Main service implementation managing the `tornado.web.Application` and taking care of configuration.

`bootstrap``()`
Implement this method in order to manipulate the configuration paths, e.g..

config
Assemble the configration files and command line arguments in order to finalize the service's configuration. All configuration values can be overwritten by the command line.

environment
The default environment instance.

get_app()
Create the `tornado.web.Appliaction` instance and return it.

In this method the `Service.bootstrap()` is called, then `Service.run()` will initialize the app.

initialize_logging()
Initialize the python logging system.

It is difficult to check whether the logging system is already initialized, so we are currently only checking if a `TimedRotatingFileHandler` has already been added to the *root* logger. This should only be necessary when running unittests though.

main()
Main method starting a **supercell** process.

This will first instantiate the `tornado.web.Application` and then bind it to the socket. There are two possibilities to bind to a socket: either by binding to a certain port and address as defined by the configuration (the *port* and *address* configuration settings) or by the *socketfd* command line parameter.

The latter is mainly used in combination with Circus (http://circus.readthedocs.org/). There you would bind the socket from circus and start the worker processes by binding to the file descriptor.

parse_command_line()
Parse the command line arguments to set different configuration values.

parse_config_files()
Parse the config files and return the *config* object, i.e. the *tornado.options.options* instance. For each entry in the *Environment.config_file_paths()* it will check for a general *config.py* and then for a file named as defined by *Environment.config_name*.

So if the config file paths are set to *['/etc/myservice', './etc/']* the following files are parsed:

```
/etc/myservice/config.cfg
/etc/myservice/user_hostname.cfg
./etc/config.cfg
./etc/user_hostname.cfg
```

Note: By default we disable the `tornado.log` module, you can enable this though using by setting the *logging* config to some valid log level string.

run()
Implement this method in order to add handlers and managed objects to the environment, before the app is started.

slog
Initialize the logging and return the logger.

1.3.3 Request handler

class `supercell.api.requesthandler.`**RequestHandler**(*application*, *request*, ***kwargs*)
supercell request handler.

The only difference to the `tornado.web.RequestHandler` is an adopted `RequestHandler._execute_method()` method that will handle the consuming and providing of request inputs and results.

`config`
Convinience method for accessing the environment.

`decode_argument`(*value*, *name=None*)
Overwrite the default `RequestHandler.decode_argument()` method in order to allow *latin1* encoded URLs.

`environment`
Convinience method for accessing the environment.

`logger`
Use this property to write to the log files.

In a request handler you would simply log messages like this:

```
def get(self):
    self.logger.info('A test')
```

`request_id`
Return a unique id per request. Collisions are allowed but should should not occur within a 10 minutes time window.

The current implementation is based on a timestamp in milliseconds substracted by a large number to make the id smaller.

1.3.4 Consumer

exception `supercell.api.consumer.`**`NoConsumerFound`**
Raised if no matching consumer for the client's *Content-Type* header was found.

class `supercell.api.consumer.`**`ConsumerBase`**
Base class for content type consumers.

In order to create a new consumer, you must create a new class that inherits from `ConsumerBase` and sets the `ConsumerBase.CONTENT_TYPE` variable:

```
class MyConsumer(s.ConsumerBase):

    CONTENT_TYPE = s.ContentType('application/xml')

    def consume(self, handler, model):
        return model(lxml.from_string(handler.request.body))
```

See Also:

`supercell.api.consumer.JsonConsumer.consume`

`CONTENT_TYPE` **= None**
The target content type for the consumer.

Type *supercell.api.ContentType*

`consume`(*handler*, *model*)
This method should return the correct representation as a parsed model.

Parameters **model** (`schematics.models.Model`) – the model to convert to a certain content type

static **`map_consumer`**(*content_type*, *handler*)
Map a given content type to the correct provider implementation.

If no provider matches, raise a *NoProviderFound* exception.

Parameters

- **accept_header** (*str*) – HTTP Accept header value
- **handler** – supercell request handler

Raises `NoConsumerFound`

class `supercell.api.consumer.`**`JsonConsumer`**
Default **application/json** provider.

`CONTENT_TYPE` **= ContentType(content_type='application/json', vendor=None, version=None)**
The **application/json** `ContentType`.

`consume`(*handler*, *model*)
Parse the body json via `json.loads()` and initialize the *model*.

See Also:

`supercell.api.provider.ProviderBase.provide`

1.3.5 Provider

exception `supercell.api.provider.`**`NoProviderFound`**
Raised if no matching provider for the client's *Accept* header was found.

class `supercell.api.provider.`**`ProviderBase`**
Base class for content type providers.

Creating a new provider is just as simple as creating new consumers:

```
class MyProvider(s.ProviderBase):

    CONTENT_TYPE = s.ContentType('application/xml')

    def provide(self, model, handler):
        self.set_header('Content-Type', 'application/xml')
        handler.write(model.to_xml())
```

`CONTENT_TYPE` **= None**
The target content type for the provider.

Type *supercell.api.ContentType*

static **`map_provider`**(*accept_header*, *handler*, *allow_default=False*)
Map a given content type to the correct provider implementation.

If no provider matches, raise a *NoProviderFound* exception.

Parameters

- **accept_header** (*str*) – HTTP Accept header value
- **handler** – supercell request handler

Raises `NoProviderFound`

provide (*model*, *handler*)
This method should return the correct representation as a simple string (i.e. byte buffer) that will be used as return value.

Parameters **model** (*supercell.schematics.Model*) – the model to convert to a certain content type

class `supercell.api.provider.`**`JsonProvider`**
Default *application/json* provider.

provide (*model*, *handler*)
Simply return the json via *json.dumps*.

See Also:

`supercell.api.provider.ProviderBase.provide`

1.3.6 Decorators

Several decorators for using with `supercell.api.RequestHandler` implementations.

`supercell.api.decorators.`**`async`** (*fn*)
Decorator that merges the `tornado.web.asynchronous()` as well as the `tornado.gen.coroutine()` decorators.

Example:

```
class MyHandler(s.RequestHandler):

    @s.async
    def get(self, user_id):
        # ...
        raise s.Return(User())
```

`supercell.api.decorators.`**`consumes`** (*content_type*, *model*, *vendor=None*, *version=None*)
Class decorator for mapping HTTP POST and PUT bodies to

Example:

```
@s.consumes(s.MediaType.ApplicationJson, model=Model)
class MyHandler(s.RequestHandler):

    def post(self, *args, **kwargs):
        # ...
        raise s.OkCreated()
```

Parameters

- **content_type** (*str*) – The base content type such as **application/json**
- **model** (`schematics.models.Model`) – The model that should be consumed.
- **vendor** (*str*) – Any vendor information for the base content type
- **version** (*float*) – The vendor version

`supercell.api.decorators.`**`provides`** (*content_type*, *vendor=None*, *version=None*, *default=False*)
Class decorator for mapping HTTP GET responses to content types and their representation.

In order to allow the **application/json** content type, create the handler class like this:

```
@s.provides(sMediaType.ApplicationJson)
class MyHandler(s.RequestHandler):
    pass
```

It is also possible to support more than one content type. The content type selection is then based on the client **Accept** header. If this is not present, ordering of the `provides()` decorators matter, i.e. the first content type is used:

```
@s.provides(s.MediaType.ApplicationJson)
class MyHandler(s.RequestHandler):
    ...
```

Parameters

- **content_type** (*str*) – The base content type such as **application/json**
- **vendor** (*str*) – Any vendor information for the base content type
- **version** (*float*) – The vendor version
- **default** (*bool*) – If **True** and no **Accept** header is present, this content type is provided

1.3.7 Health Checks

Health checks provide a way for the hoster to check if the application is still running and working as expected.

The most basic health check is enabled by default on the */_system/check* route. This is a very simple check if the process is running and provides no details to your application checks.

To demonstrate the idea we will describe a simple health check perfoming a request to a HTTP resource and return the appropriate result:

```
@s.provides(s.MediaType.ApplicationJson, default=True)
class SimpleHttpResourceCheck(s.RequestHandler):

    @s.async
    def get(self):
        result = self.environment.http_resource.ping()
        if result.code == 200:
            raise s.HealthCheckOk()
        if result.code == 599:
            raise s.HealthCheckError()
```

To enable this health check simply add this to the environment in your services run method:

```
class MyService(s.Service):

    def run(self):
        self.environment.add_health_check('http_resource',
                                          SimpleHttpResourceCheck)
```

When the service is then started you can access the check as */_system/check/http_resource*:

```
$ curl 'http://127.0.0.1/_system/check/http_resource'
{"code": "OK", "ok": true}
```

The HTTP response code will be **200** when everything is ok. Any error, **WARNING** or **ERROR** will return the HTTP code **500**. A warning will return the response:

```
$ curl 'http://127.0.0.1/_system/check/http_resource_with_warning'
{"code": "WARNING", "error": true}
```

and an error a similar one:

```
$ curl 'http://127.0.0.1/_system/check/http_resource_with_warning'
{"code": "ERROR", "error": true}
```

exception `supercell.api.healthchecks.`**`HealthCheckError`**`(`*additional=None*`)`

Exception for health checks return values indicating a **ERROR** health check.

The **ERROR** state indicates a major problem like a failed connection to a database.

exception `supercell.api.healthchecks.`**`HealthCheckOk`**`(`*additional=None*`)`

Exception for health checks return values indicating a **OK** health check.

exception `supercell.api.healthchecks.`**`HealthCheckWarning`**`(`*additional=None*`)`

Exception for health checks return values indicating a **WARNING** health check.

The **WARNING** state indicates a problem that is not critical to the application. This could involve things like long response and similar problems.

class `supercell.api.healthchecks.`**`SystemHealthCheck`**`(`*application*, *request*, ***kwargs*`)`

The default system health check.

This check is returning this JSON:

```
{"message": "API running", "code": "OK", "ok": true}
```

and its primiary use is to check if the process is still running and working as expected. If this request takes too long to respond, and all other systems are working correctly, you probably need to create more instances of the service since the current number of processes cannot deal with the number of requests coming from the outside.

`get``(`**args*, ***kwargs*`)`

Run the default **/_system** healthcheck and return it's result.

1.3.8 Statistics

`supercell.api.stats.`**`latency`**`(`*fn*`)`

Measure execution latency of a certain request method.

In order to measure latency for GET requests of a request handler you simply have to add the `latency()` decorator to the declaration:

```
@s.latency
@s.async
def get(self, *args, **kwargs):
    ...
```

The latency is recorded along the request path, i.e. if the request handler is defined like this:

```
env.add_handler('/test/this', LatencyExample)
```

the latency of GET/POST/PUT etc methods are stored with the path. In order to access the stats you may call **/_system/stats/test/this** or **/_system/stats/test**, e.g.

`supercell.api.stats.`**`metered`**`(`*fn*`)`

Meter the execution of certain requests.

The `metered()` stats will measure the 1/5/15 minutes averages for requests. This is also applied trivially:

```
@s.metered
@s.async
def get(self, *args, **kwargs):
    ...
```

As with the `latency()` stats, the `metered()` stats are recorded along the request path, i.e. you can get the stats values using the **/_system/stats/** route.

1.3.9 Caching

Helpers for dealing with HTTP level caching.

The *Cache-Control* and *Expires* header can be defined while adding a handler to the environment:

```
class MyService(Service):

    def run(self):
        self.environment.add_handler(...,
                                     cache=CacheConfig(timedelta(minutes=10)),
                                            expires=timedelta(minutes=10))
```

The details of setting the *CacheControl* header are documented in the `CacheConfig()`. The *expires* argument simply takes a `datetime.timedelta()` as input and will then generate the *Expires* header based on the current time and the `datetime.timedelta()`.

`supercell.api.cache.`**`CacheConfig`**`(`*max_age*, *s_max_age=None*, *public=False*, *private=False*, *no_cache=False*, *no_store=False*, *must_revalidate=True*, *proxy_revalidate=False*`)`

Create a `CacheConfigT` with default values. :param max_age: Number of seconds the response can be cached :type max_age: datetime.timedelta

Parameters

- **s_max_age** (*datetime.timedelta*) – Like *max_age* but only applies to shared caches
- **public** (*bool*) – Marks responses as cachable even if they contain authentication information
- **private** (*bool*) – Allows the browser to cache the result but not shared caches
- **no_cache** (*bool*) – If *True* caches will revalidate the request before delivering the cached copy
- **no_store** (*bool*) – Caches should not store any cached copy.
- **must_revalidate** (*bool*) – Tells the cache to not serve stale copies of the response
- **proxy_revalidate** (*bool*) – Like *must_revalidate* except it only applies to public caches

CHAPTER 2

Indices and tables

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

Python Module Index