
supercell Documentation

Release 0.6.3

Daniel Truemper

August 24, 2015

1	Contents:	3
1.1	Getting Started	3
1.2	Topics	6
1.3	API	8
2	Indices and tables	15
	Python Module Index	17

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.provides(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.

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 operations 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 see the different ways to really set the configuration.

1.1.4 Create a service class

The service class is the part of the application defining its 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.provides(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.provides* 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 one 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 Rendering HTML

HTML is just another output format supported by *supercell*. The default implementation is using tornado's built in template mechanism for rendering HTML. A *RequestHandler* supporting HTML only has to implement the *get_template()* method, that will return the template file name based on the final model. By setting the *template_path* configuration variable one may define the directory containing all templates.

A minimum example would look like this:

```
@s.provides(s.MediaType.TextHtml, default=True)
class SimpleHtml(s.RequestHandler):

    def get_template(self, model):
        return 'simple.html'

    def get(self, *args, **kwargs):
        raise s.Return(Saying({'id': self.counter, 'content': content}))

class MyService(s.Service):

    def bootstrap(self):
        self.environment.tornado_settings['template_path'] = 'templates/'

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

1.2.3 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.

1.2.4 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

1.3.2 Service

A *Service* is the main element of a *supercell* application. It will instantiate 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.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 configuration 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.Application` 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 `SupercellLoggingHandler` has already been added to the *root* logger. This should only be necessary when running unittests though.

main (with_signals=True)

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.

shutdown()

Graceful shutdown of the server.

In this method we stop the `tornado.httpserver` in order to stop accepting new connections. During a period of `max_grace_seconds` current requests are allowed to finish. After this period the `IOLoop` is stopped.

slog

Initialize the logging and return the logger.

1.3.3 Request handler

class `supercell.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

Convenience 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

Convenience method for accessing the environment.

get_template(model)

Method to determine which template to use for rendering the html.

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 not occur within a 10 minutes time window.

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

1.3.4 Consumer

exception `supercell.consumer.NoConsumerFound`

Raised if no matching consumer for the client's *Content-Type* header was found.

class `supercell.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`

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.consumer.JsonConsumer`

Default *application/json* provider.

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.provider.NoProviderFound`

Raised if no matching provider for the client's *Accept* header was found.

class `supercell.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())
```

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.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 Query Parameter

Simple decorator for dealing with typed query parameters.

class `supercell.queryparam.QueryParams` (*params, kwargs_name='query'*)

Simple middleware for ensuring types in query parameters.

A simple example:

```
@QueryParams((
    ('limit', IntType()),
    ('q', StringType())
))
@s.async
def get(self, *args, **kwargs):
    limit = kwargs.get('limit', 0)
    q = kwargs.get('q', None)
    ...
```

If a param is required, simply set the *required* property for the schematics type definition:

```
@QueryParams((
    ('limit', IntType(required=True)),
    ('q', StringType())
))
...

```

If the parameter is missing, a HTTP 400 error is raised.

By default the dictionary containing the typed query parameters is added to the *kwargs* of the method with the key *query*. In order to change that, simply change the key in the definition:

```
@QueryParams((
    ...
),
    kwargs_name='myquery'
)
...

```

1.3.7 Decorators

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

`supercell.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.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(s.MediaType.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

1.3.8 Health Checks

1.3.9 Statistics

`supercell.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.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.

1.3.10 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.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

Indices and tables

- `genindex`
- `modindex`
- `search`

S

- `supercell.cache`, [13](#)
- `supercell.consumer`, [10](#)
- `supercell.decorators`, [12](#)
- `supercell.provider`, [10](#)
- `supercell.queryparam`, [11](#)
- `supercell.requesthandler`, [9](#)
- `supercell.service`, [8](#)
- `supercell.stats`, [13](#)

B

bootstrap() (supercell.service.Service method), 8

C

CacheConfig() (in module supercell.cache), 14

config (supercell.requesthandler.RequestHandler attribute), 9

config (supercell.service.Service attribute), 8

consume() (supercell.consumer.ConsumerBase method), 10

consume() (supercell.consumer.JsonConsumer method), 10

ConsumerBase (class in supercell.consumer), 10

consumes() (in module supercell.decorators), 12

D

decode_argument() (supercell.requesthandler.RequestHandler method), 9

E

environment (supercell.requesthandler.RequestHandler attribute), 9

environment (supercell.service.Service attribute), 8

G

get_app() (supercell.service.Service method), 8

get_template() (supercell.requesthandler.RequestHandler method), 9

I

initialize_logging() (supercell.service.Service method), 8

J

JsonConsumer (class in supercell.consumer), 10

JsonProvider (class in supercell.provider), 11

L

latency() (in module supercell.stats), 13

logger (supercell.requesthandler.RequestHandler attribute), 9

M

main() (supercell.service.Service method), 8

map_consumer() (supercell.consumer.ConsumerBase static method), 10

map_provider() (supercell.provider.ProviderBase static method), 11

metered() (in module supercell.stats), 13

N

NoConsumerFound, 10

NoProviderFound, 10

P

parse_command_line() (supercell.service.Service method), 8

parse_config_files() (supercell.service.Service method), 8

provide() (supercell.provider.JsonProvider method), 11

provide() (supercell.provider.ProviderBase method), 11

ProviderBase (class in supercell.provider), 10

provides() (in module supercell.decorators), 12

Q

QueryParams (class in supercell.queryparam), 11

R

request_id (supercell.requesthandler.RequestHandler attribute), 9

RequestHandler (class in supercell.requesthandler), 9

run() (supercell.service.Service method), 9

S

Service (class in supercell.service), 8

shutdown() (supercell.service.Service method), 9

slog (supercell.service.Service attribute), 9

supercell.cache (module), 13

supercell.consumer (module), 10

supercell.decorators (module), 12

[supercell.provider \(module\)](#), [10](#)
[supercell.queryparam \(module\)](#), [11](#)
[supercell.requesthandler \(module\)](#), [9](#)
[supercell.service \(module\)](#), [8](#)
[supercell.stats \(module\)](#), [13](#)