# supercell Documentation

***Release 0.2.4***

**Daniel Truemper**

July 16, 2013

# CONTENTS

Contents:

CHAPTER

ONE

# API

This is the main *supercell* API reference.

## 1.1 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*, *name=None*, *host_pattern='.*$'*)

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 ('.*$')

**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**.

**health_checks**
Simple property access for health checks.

**tornado_log_function**(*logger*)
Return a function that will log tornado requests.

**Parameters** **logger** (*logging.logger*) – the logger that will be used for logging the requests

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

## 1.2 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**(*name='supercell'*)

Initialize the python logging system.

**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
```

**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 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.

**environment**
Convinience method for accessing the environment.

## 1.4 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

## 1.5 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
```

## 1.6 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.`**`cache`**(*max_age*, *s_max_age=None*, *public=False*, *private=False*, *no_cache=False*, *no_store=False*, *must_revalidate=True*, *proxy_revalidate=False*)

Set the *Cache-Control* HTTP header to allow for fine grained caching controls.

For detailed information read http://www.mnot.net/cache_docs/ and http://www.ietf.org/rfc/rfc2616.txt

Usage:

```
@cache(timedelta(minutes=10))
def get(self):
    self.finish({'ok': True})
```

This would allow public caches to store the result for 10 minutes.

**Parameters**

- **max_age** (*datetime.timedelta*) – Number of seconds the response can be cached
- **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

`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(sRequestHandler):
    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, model=MyModel)
class MyHandler(s.RequestHandler):
    ...
```

**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
- **default** (*bool*) – If **True** and no **Accept** header is present, this content type is provided

## 1.7 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* 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/http_resource*:

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

and an error a similar one:

```
$ curl 'http://127.0.0.1/_system/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.

CHAPTER

TWO

# INDICES AND TABLES

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

# PYTHON MODULE INDEX

## S