Snorky Documentation

Release 0.1.0-a1

Juan Luis Boya García

February 01, 2017

Contents

CHAPTER 1

User's guide

1.1 Installation

In order to run the Snorky server you need a system with a working Python installation and a series of dependencies.

1.1.1 Supported systems

Snorky will work on any platform supported by Tornado[1]. These include Windows, Mac, Linux and BSD.

In order to achieve the maximum performance in a production server, Linux and BSD are recommended, as they have fast event selection system calls which are supported by Tornado (`epoll` in Linux and `kqueue` in BSD). At the moment of writing Tornado does not support `IOCP` on Windows.

For development, any system is fine.

1.1.2 Which Python version should I choose

There are two versions of the Python language in use, the 2.x branch and the 3.x one.

Python 3.x changes a number of things, for example...

- In Python 2 strings were byte-based and had arbitrary encodings, whilst strings in Python 3 are character-based by default in order to manage Unicode text better.
- Some old features of the language were removed and other slightly modified. For example, `print` was an statement in Python 2 but it is a function in Python 3 and as a consequence, requires parentheses (i.e. `print("Hello World")`, not `print "Hello World"`).
- Although Python 3 has been out there for a long time, there are still some libraries that only work in Python 2. Still, it is possible to write code that works in both systems without changes, and many Python packages do this.

Snorky can work with both branches of Python. If you need to use any library which works only in Python 2, use that; in other case, use Python 3.

[1] http://www.tornadoweb.org/

1.1.3 Installing a Python interpreter

You can download a Python installer for your platform at the official Python web site[2].

If you use Windows, it is recommended that you check *Add Python to PATH* during the installation.

In Linux based systems it is often either already installed or available through the usual distribution channels.

Supported versions

Snorky works in either:

- Python 3.3 or later
- Python 2.7 or later.

You can check what version of Python you have writing `python --version` in a command shell.

Note: Got "not recognized as an internal or external command" error on Windows?

In that case you probably either did not installed a Python interpreter or, if you did, you did not add it to the system `PATH`.

Try running `C:\Python34\python.exe --version` instead, changing `Python34` with the version of Python that you installed (in this case it would be 3.4).

If that works, you can either run Python everytime using the full path every time or add it to the environmental variables of the system[3].

1.1.4 Installing the Python package manager

Snorky carries a series of dependencies. In order to be able to install them, you need *pip*, the Python package manager.

If you use Python 3.4 or later, you already have it installed. Otherwise you can install it following its official guide[4].

1.1.5 Installing Snorky from the tarball

Download and the last Snorky package[5] and extract it. In a terminal, change to the directory where you extracted the package and run the following command.

```
python setup.py install
```

That's it.

[2] https://www.python.org/
[3] http://stackoverflow.com/a/6318188
[4] http://pip.readthedocs.org/en/latest/installing.html
[5] http://snorkyproject.org/

1.2 Running the examples

Snorky bundles with a few small demonstration applications that you can use to verify it works.

You can find them in `snorky/demos` inside the project root.

1.2.1 Snorky ToDo demo

In `snorky/demos/snorky_todo_angular` you can find a simple note taking application. The demo is based in TodoMVC[6]. It works with Django and AngularJS.

In order to run it, first you must install its dependencies (e.g. Django). To do so you can run the following command:

```
pip install -r requirements.txt
```

Note: If you are on Windows and you get "not recognized as an internal or external command" error, check your PATH.

You must add both the Python directory (e.g. `C:\Python34`) and the Python script directory where pip is installed (e.g. `C:\Python34\Script`).

Django requires a database to run. The demo uses SQLite, which is included in Python by default. In order to create the database run:

```
python manage.py syncdb
```

Reply `no` when it asks you to create a super user, you don't need it.

After that, you can run the Django production server.

```
python manage.py runserver
```

Open another terminal, go to the demo directory and run the Snorky server.

```
python run_snorky_server.py
```

You can open http://localhost:8000/ in your browser and try adding some notes. Opening it in several browser windows should show how changes are applied on both automatically.

1.3 Overview of Snorky

The following is an example server with Snorky exposing a simple Pub Sub service. The goal of this section is to explain each of the parts involved on it, both from Snorky and from Tornado.

[6] http://todomvc.com/

```
import os
from tornado.ioloop import IOLoop
from tornado.web import Application
from snorky import ServiceRegistry

from snorky.request_handlers.websocket import SnorkyWebSocketHandler
from snorky.services.pubsub import PubSubService

if __name__ == "__main__":
    service_registry = ServiceRegistry()
    service_registry.register_service(PubSubService("messaging"))

    application = Application([
        SnorkyWebSocketHandler.get_route(service_registry, "/ws"),
    ])
    application.listen(8002, address="")

    try:
        print("Snorky running...")
        IOLoop.instance().start()
    except KeyboardInterrupt:
        pass
```

1.3.1 The I/O loop

`tornado.ioloop.IOLoop`[7] is the Tornado event loop. Being an asynchronous server, there are no separate threads for each connection. Instead, all sockets are managed by this class.

`IOLoop` performs and endless loop in which tells the operating system to notify it of any event occurred in the managed sockets and in turn dispatches the event to the class that owns the socket. This loop starts when `IOLoop.instance().start()` is called.

`IOLoop` is a singleton class. In normal usage you even don't need to keep references to it, the framework manages this automatically. Most of the times you only need call `IOLoop.instance().start()` and go on.

1.3.2 Services

Services are classes which process messages. Messages are JSON entities, that is: they can be any data type representable with JSON, like strings, arrays (also called *lists*) or objects (also called *dictionaries*). Services often are stateful and track client connections.

Seen in the heading example, `snorky.services.pubsub.PubSubService` is a service which accepts several methods: `join`, `leave` and `publish`. Clients can connect to the server and ask `PubSubService` to join them to a certain *channel* (which is also JSON entity, typically a string). An user can send a `publish` command to a channel which would trigger notifications in all those clients which joined that channel.

`PubSubService` is a simple service which can be useful in a number of situations, but often you will also write your own services, either from scratch or subclassing other services.

[7] http://www.tornadoweb.org/en/stable/ioloop.html#tornado.ioloop.IOLoop

Writing services explains this in detail.

1.3.3 Service registries

The `ServiceRegistry` class tracks a fixed number of services, each identified with a name, which should be a string.

`ServiceRegistry` is also responsible for delivery of messages to its services. There can be several instances tracking different services at the same time.

Wiring a backend explains how this can be used to offer both a *public* set of services on an interface, and a *private* one in other, often firewalled. This can be used for example to only allow a trusted machine to send certain types of events to the clients subscribed to the public services.

1.3.4 Request handlers

Request handlers are Tornado classes which are intended to respond to HTTP and WebSocket requests. They inherit from `tornado.web.RequestHandler`[8] and they are explained in detail in the Tornado documentation.

Snorky request handlers are associated with a service registry. Their job is to receive messages from the outside and forward them to the service registry, providing also the service with means to send messages in the other way.

Several request handlers can be attached to the same service registry or to independent registries.

Currently there are three request handlers bundled with Snorky:

- `snorky.request_handlers.websocket.SnorkyWebSocketHandler`: Handles WebSocket connections.
- `snorky.request_handlers.sockjs.SnorkySockJSHandler`: Handles SockJS connections, which are an abstraction layer of WebSocket providing fallbacks for old browsers which do not support it natively.
- `snorky.request_handlers.http.BackendHTTPHandler`: This is a more limited request handler. It works over plain HTTP and each connection can only exchange one message from each party, one for the request, and one for the response. It's usually used in order to expose a backend interface.

1.3.5 Application

In Tornado, a `tornado.web.Application`[9] is *a collection of request handlers that make up a web application.*

This class manages a set of routes, each one consisting of an URL pattern, a request handler class and optionally a set of parameters which are fed to the

[8] http://www.tornadoweb.org/en/stable/web.html#tornado.web.RequestHandler
[9] http://www.tornadoweb.org/en/stable/web.html#tornado.web.Application

`tornado.web.RequestHandler`[10] `__init__` function. `SnorkyWebSocketHandler.get_route()` returns such a route for a WebSocket request handler.

`tornado.web.Application.listen()` sets up an HTTP server listening on the specified port and address. If no address is specified, it will listen in all interfaces, both in IPv4 and IPv6, if supported.

1.3.6 Conclusion

The UML diagram in Fig. 1.1 resumes the collaborations explained above.

The next chapters will cover further details on the inner working of each of the components and how they can be extended.

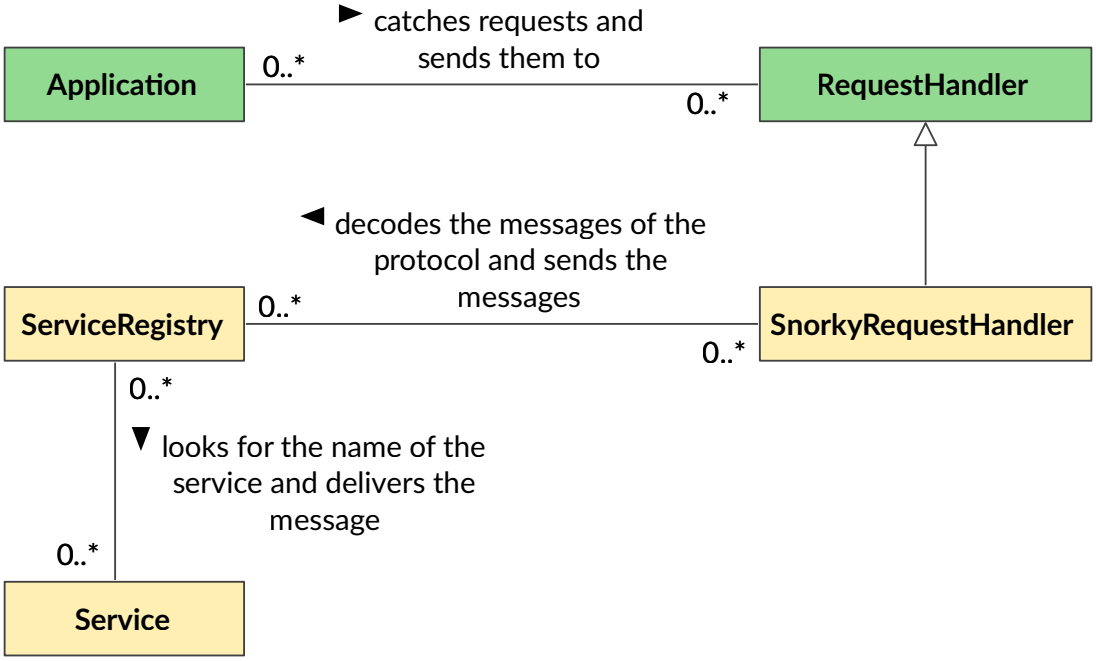

Fig. 1.1: The overview of Snorky core classes as a simplistic UML diagram.

1.4 Writing services

Services are the main construction blocks of Snorky. In this section you will learn how services work and how they are created.

1.4.1 The Snorky service protocol

Every message send through Snorky must be directed to a service, identified by a service *name*, which shall be an string.

[10] http://www.tornadoweb.org/en/stable/web.html#tornado.web.RequestHandler

The message itself must be a JSON entity. That includes strings, numbers, arrays, objects and the `null` value. The JSON web site[11] contains the full specification of the language, describing each data type in detail.

Often in this documentation, JSON arrays will be referred as **lists** and JSON objects will be referred as **dictionaries**, matching the Python data types those primitive types are transformed into.

The following is an example of a message sent to an `echo` service, as it could be sent through WebSocket:

```
{"service":"echo","message":"Hello"}
```

1.4.2 The Service definition

A service class must inherit from *Service* or one of their descendants. It must provide an implementation for the method *Service.process_message_from()*.

The following example service sends back to the client each message it receives:

```
from snorky.services.base import Service

class EchoService(Service):
    def process_message_from(self, client, msg):
        self.send_message_to(client, msg)
```

class snorky.services.base.Service(*name*)

Subclass this class and redefine `process_message_from()` in order to create a new service.

process_message_from(*client*, *msg*)

Called when a message is received.

`msg` contains the message as a JSON decoded entity. msg and all their descedants are always hashable.

send_message_to(*client*, *msg*)

Sends a message to a client through the current service.

Services should use this method instead of calling directly to `client.send()` in order to add the service header.

client_connected(*client*)

Called each time a client connects to Snorky through a channel which is connected to the same `snorky.ServiceRegistry` than this service.

Exceptionally, this method is not called when a client connects from a short-lived channel like `snorky.request_handlers.http.BackendHTTPHandler`.

client_disconnected(*client*)

Called each time a client disconnects from Snorky through a channel which is connected to the same `snorky.ServiceRegistry` than this service.

Exceptionally, this method is not called when a client connects from a short-lived channel like `snorky.request_handlers.http.BackendHTTPHandler`.

[11] http://json.org/

1.4.3 RPC services

Although you could write services inheriting directly from *Service* and using its simple methods, they often fall short.

Most services often work, at least partially, in a request-response fashion, ocasionally sending notifications to the client that are not part of the response.

Snorky leverages this pattern through the subclass *RPCService*. Currently, all instanciable Snorky services are RPC services, and chances are yours will be too.

class snorky.services.base.RPCService(*name*)

Subclass this class to make RPC services.

RPC services expose a more convenient interface than bare Snorky services.

Commands

Each RPC service has a series of *commands* which are defined as methods with the `rpc_command()` decorator.

Commands accept a set of parameters which is specified in the signature of the method. They may have default values.

The return value of a command is sent automatically to the requester client. Every entity that can be serialized as JSON is a valid return value.

If the method does not return anything, `null` is sent as response. This is usually done in order to signal that the request has been processed successfully but there is nothing interesting to send in return.

The following service calculates sums and logarithms in response to client requests:

```
import math
from snorky.services.base import RPCService, rpc_command

class CalculatorService(RPCService):
    @rpc_command
    def sum(self, req, number1, number2):
        return number1 + number2

    @rpc_command
    def log(self, req, number, base=2.718):
        return math.log(number, base)
```

Note: The names of RPC commands and their parameters are usually written in `camelCase` instead of `snake_case` because they are exposed in Javascript with the same name.

Exceptions

Sometimes you want to signal an error condition. In this cases, instead of returning, raise an instance of `RPCError`. For example, `raise RPCError("Not authorized")`.

Snorky already signals some error conditions by default:

- If a client requests a non existing command, `Unknown command` is raised.
- If the request params don't fit the ones specified in the method, i.e. nonexistent parameters are used or required parameters are ommited, `Invalid params` is raised.
- If an exception different from `RPCError` is raised while the command is being handled, `Internal error` is raised.

Asynchronous commands

Sometimes the processing of a command has to be temporarily suspended until a certain event occurs.

For example, the command may need to perform a HTTP request. It's undesirable for the command to block the entire server, as that would kill performance. Instead, asynchronous requests shall be used.

Such RPC commands must use the decorator `rpc_asynchronous()`, in addition to `rpc_command()`.

Asynchronous RPC commands do not send a response when the method call returns nothing. Instead, is expected that the request will be replied eventually as a response to another event.

The `req` parameter in RPC commands contains a *`Request`* object with methods to send either a successful reply or signal an error to the client.

1.4.4 The Request class

class `snorky.services.base.Request`(*service*, *client*, *msg*)

Represents a request against an RPC service and provides methods to resolve it.

`reply`(*data*)

Sends a successful response.

Each request can be resolved one time. Calling this method twice or calling both `reply()` and `error()` will trigger a server error.

`error`(*msg*)

Sends an error response.

Each request can be resolved one time. Calling this method twice or calling both `reply()` and `error()` will trigger a server error.

`client`

The client which initiated this requests.

The requester client. It complies with the interface defined in `snorky.client.Client`.

`command`

The requested command.

The requested command name.

`params`
: The specified parameters as a dictionary.

  The params supplied by the client, as a dictionary.

`resolved`
: Whether the request has been resolved either with success or failure.

  `True` if the request has been resolved either with a successful reply or with an error.

1.4.5 Sending notifications

At any moment you can send an arbitrary message to any client of your service. These messages should be JSON objects and should contain a `type` attribute which must be neither `response` or `error`, since these types are used by RPC calls.

Messages which are neither of type `response` or `error` are called *notifications*.

The following example shows a simple PubSub service in which any client can publish a message to every client subscribed (including itself):

```
from snorky.services.base import RPCService, rpc_command

class MinimalPubSubService(RPCService):
    def __init__(self, name):
        # Call parent constructor
        RPCService.__init__(self, name)

        self.clients = set()

    @rpc_command
    def subscribe(self, req):
        if req.client not in self.clients:
            self.clients.add(req.client)

    @rpc_command
    def unsubscribe(self, req):
        if req.client in self.clients:
            self.clients.remove(req.client)

    def client_disconnected(self, client):
        # Never forget to remove the client from the set after disconnection!
        if client in self.clients:
            self.clients.remove(client)

    @rpc_command
    def publish(self, req, message):
        for client in self.clients:
            self.send_message_to(client, {
                "type": "publication",
                "message": message,
            })
```

1.4.6 Conclusion

This chapter has explained how to built services with Snorky.

Although Snorky comes with a few services, often you will need to extend them or create small specific services for your application. Nevertheless, Snorky utilities should not make this task difficult.

The next chapter will explain how to connect to Snorky from a web application and how service connectors are created in Javascript.

1.5 Connecting to services

More often than not, you will want to connect to Snorky from Javascript in a web application. For this purpose, there is an official connector, **Snorky.js**.

1.5.1 Dependencies

Snorky.js has a few dependencies that must be included before it:

- my.class.js[12]: A lightweight class library for Javascript. Javascript does not have a declarative syntax to define classes, which makes writing them a tedious and error-prone process. This is one of many libraries trying to fill that gap.

  Once initialized, this class factory is stored in `Snorky.Class()`.

- js-signals[13]: An event library for Javascript. All events defined in Snorky are created with this, which allows you to easily add and remove any number of handlers for each event.

  Once initialized, the event class is stored in `Snorky.Signal()`. Although in vanilla usage this class will be the same as js-signal's `signals.Signal()`, additional initialization code may replace it with a subclass which provides additional functionality.

  For example, Angular-Snorky does this in order to request a `$digest` cycle to AngularJS every time an event is dispatched in order to update the view.

Promises

Additionally, Snorky returns promises for service requests. In order for this to work, Snorky needs a `Promise` class.

By default it will use the Promise class defined in ECMAScript 6[14], but it can be changed to any other class with a similar interface running the following code after `snorky.js` has been included:

```
Snorky.Promise = <your promise class>;
```

[12] https://github.com/jiem/my-class
[13] http://millermedeiros.github.io/js-signals/
[14] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Particularly, Angular-Snorky does this as part of its initialization in order to make Snorky work with Angular promises[15] instead of ES6 promises.

If you choose to use ES6 promises, chances are you will need a polyfill since, at the time of writing, most browsers don't support them out of the box yet. Jake Archibald[16] has published such polyfill at GitHub[17].

Socket class

Snorky needs a socket class in order to connect to the server. This class must have the WebSocket interface[18].

Nowadays most browsers in use support WebSocket natively, making it an excellent choice for a number of applications. In cases where older browser support is needed (e.g. IE9 or older) SockJS[19] provides a viable alternative, providing fallback transports for old browsers but using WebSocket under the hood in modern browsers.

1.5.2 Connecting

Once `snorky.js` and all the dependencies needed have been included, the `Snorky` class will be available in the global namespace.

class `Snorky`(*socketClass*, *address*, *services*[, *options*])

Manages a connection to Snorky and its associated service connectors. Connection is automatically made on object creation.

Arguments

- `socketClass` (*class*) – The socket class which will be used for the connection.

  It must provide the same interface as `WebSocket()`. It's usually either `WebSocket` or `SockJS`.

  Note that this parameter requests the class itself, not an instance.

- `address` (*String*) – The URL which will be passed to the constructor of `socketClass`. Note that WebSocket uses `ws://` or `wss://` as protocol while `SockJS` uses `http://` or `https://` instead.

- `services` (*Object*) – A dictionary matching each service name with a class. Those classes will be used to interact with the services from Javascript.

- `options` (*Object*) – An optional dictionary with additional options. At the moment only one option is supported:

  `debug`
  Whether to print debugging information to the console when the connection is made or lost.

[15] https://docs.angularjs.org/api/ng/service/$q
[16] https://github.com/jakearchibald
[17] https://github.com/jakearchibald/es6-promise
[18] http://www.w3.org/TR/2011/WD-websockets-20110419/
[19] https://github.com/sockjs/sockjs-client

`address`
The address to which the socket has been connected.

`socketClass`
The socket class used.

`debug`
Whether the debug mode is enabled.

`services`
Dictionary of **service instances** available, indexed by name.

`isConnected`
Whether there is an active connection to the server.

`isConnecting`
Whether a connection to the server is being attempted.

`logDebug`(*format*[, *...*])
Logs text in the console, but only if the debug mode is active.

Accepts either any object or a format string with arguments like `console.debug()`.

Events

`connected`
Event raised when a connection is successfully made to the server.

`disconnected`
Event raised when a connection is closed, either voluntarily or due to a network failure.

1.5.3 Service connectors

Snorky.js has a number of classes which provide an interface to the Snorky services from the client side. They should inherit from *Snorky.Service()*, or, more frequently, from *Snorky.RPCService()*.

class `Snorky.Service`(*name*, *snorky*)
Provides an interface to a Snorky service.

Usually you don't need to create instances of this class directly, *Snorky()* does automatically.

Arguments

- `name` (*String*) – The name of the service instance in the server side.
- `snorky` (Snorky) – The Snorky object this service connector belongs.

`name`
The service name.

`snorky`
The Snorky object this service connector belongs.

`init()`
Initialization hook. You usually extend it in your subclasses in order to listen for events, register attributes or do other initialization work.

`sendMessage(`*message*`)`
Sends a message to the service. `message` must be serializable to JSON.

Events

`packetReceived`
Event raised when a new message arrives to this service.

class `Snorky.RPCService(`*name*, *snorky*`)`
Provides a convenient connector for RPC services.

If you are writing a connector to an RPC service you should subclass this class instead of *`Snorky.Service()`*.

`rpcCall(`*command*, *params*`)`

Arguments

- `command` (*`String`*) – The command to request
- `params` (*`Object`*) – A **dictionary** of parameters.

Returns

An A+ Promise[20].

If the remote call is successful, the promise will be fulfilled with the value returned from the remote call.

If the remote call fails, the promise will be rejected with the error message sent by Snorky.

Makes an RPC call.

`addRPCMethods(`*methods*`)`

Arguments

- `methods` (*`Array`*) – A list of command names

This is an static method.

Adds the specified RPC commands as methods to the class. For each command a method with the same name will be generated which will accept one argument with the RPC parameters and will return a Promise.

Internally these methods will call to `Snorky.RPCService.addRPCMethods()`.

Events

`notificationReceived`
Event raised when a notification is received; that is, a message whose `type` is neither `response` or `error`.

[20] http://promisesaplus.com/

1.5.4 SimplePubSub connector

The following service connector makes easy to use the `SimplePubSubService` example service described in *Sending notifications*:

```
var SimplePubSub = new Snorky.Class(Snorky.RPCService, {
  init: function() {
    // Call the superclass init()
    Snorky.RPCService.prototype.init();

    // Listen notificationReceived events
    this.notificationReceived.add(this.onNotification, this);

    // Create new event
    this.publicationReceived = new Snorky.Signal();
  },

  onNotification: function(notification) {
    if (notification.type == "publication") {
      // Dispatch the publicationReceived event
      this.publicationReceived.dispatch(notification.message);
    }
  }
});
SimplePubSub.addRPCMethods([
  "subscribe",
  "unsubscribe",
  "publish"
]);
```

1.5.5 Connection example

The following example connects to an Snorky server with the `SimplePubSub` service described above.

```
var snorky = new Snorky(WebSocket, "ws://localhost:8001/ws", {
    "pubsub": SimplePubSub
});
```

`snorky.services` contains the instantiated services. The following code would request a subscription.

```
snorky.services.pubsub.subscribe({ /* no parameters */ })
  .then(function (returnValue) {
    console.log("Subscribed!");
  });
```

Note: The services can be used even before the connection has been established. Snorky will store the messages in a buffer and send them when the connection is made.

Event handlers can be added too at this stage. The following would log messages published.

```
snorky.services.pubsub.publicationReceived.add(
  function(publishedText) {
    console.log(publishedText)
  });
```

1.5.6 Conclusion

This chapter has shown the basics of the Javascript Snorky connector, including how to include it, how connect to it and how services connector can be written and used.

The following chapter will cover the backend interface.

1.6 Wiring a backend

Sometimes there are some actions in your system that you only want to allow to certain parties, like trusted servers in your network. In Snorky, those are called *the backend*.

Special services are often exposed to the backend servers, which allow controlling restricted aspects of other services. This is possible due to Snorky allowing to have several service registries.

1.6.1 Pub Sub with backend

Sometimes you want a Pub Sub service when the end clients are not allowed to publish, just to subscribe, and publications can only be made by a trusted computer.

Snorky comes with a Pub Sub service, `snorky.services.pubsub.PubSubService` which already implements this with another backend service `snorky.services.pubsub.PubSubBackend`.

Both could be used like this:

```
from tornado.ioloop import IOLoop
from tornado.web import Application
from snorky import ServiceRegistry

from snorky.request_handlers.http import BackendHTTPHandler
from snorky.request_handlers.websocket import SnorkyWebSocketHandler

from snorky.services.pubsub import PubSubService, PubSubBackend

class PrivatePubSub(PubSubService):
    def can_publish(self, client, channel):
        # Publishing is only allowed from the backend
        return False

if __name__ == "__main__":
    # Create separate service registries, providing separate interfaces to
    # services
    frontend_registry = ServiceRegistry()
    backend_registry = ServiceRegistry()
```

```
# Create a PubSub service and register it in the frontend registry
pubsub = PrivatePubSub("pubsub")
frontend_registry.register_service(pubsub)

# Create a PubSub backend service and register it in the backend registry
pubsub_backend = PubSubBackend("pubsub_backend", pubsub)
backend_registry.register_service(pubsub_backend)

# Make frontend_registry attend requests from WebSocket through part 5800
frontend_application = Application([
    SnorkyWebSocketHandler.get_route(frontend_registry, "/ws"),
])
frontend_application.listen(5800)

# Make backend_registry attend requests from HTTP through port 5801
backend_application = Application([
    (r"/backend", BackendHTTPHandler, {
        "service_registry": backend_registry,
        "api_key": "swordfish"
    }),
])
backend_application.listen(5801)

try:
    print("Snorky running...")
    IOLoop.instance().start()
except KeyboardInterrupt:
    pass
```

Note the backend and the frontend interface are exposed in **different ports**. This is not a requirement, but makes firewalling easier. For example, the backend port may only be exposed to the local network while the frontend port will usually be exposed to the entire world.

1.6.2 The API key

In order to provide additional security, the `BackendHTTPHandler` requires a API key or password in order to interact with its associated services. This key will be sent as an HTTP header with name `X-Backend-Key`.

Make sure to choose an API key and make it secret.

Even if you make the backend interface available only to computers in a restricted network or only to the same machine that runs Snorky, the API key still provides a security benefit, avoiding successful attacks to other server processes to escalate into Snorky.

1.6.3 Exposing the backend interface only to the local machine

If you only need to communicate with the backend interface within the same machine Snorky runs, you can bind `backend_application` to the local address, thus avoiding it to be reachable from the outside.

In order to do this, replace the `listen` call for `backend_application` in the code above with this:

```
backend_application.listen(5801, address="127.0.0.1")
```

1.6.4 Communicating with the backend interface

A JSON HTTP request is enough to send a command to the Snorky backend interface. Virtually every programming language has support for this kind of communication.

The request must contain in the body the JSON message including the service header, encoded in UTF-8.

The following code shows an example which sends a `publish` command in Python using requests[21].

```
import json
import requests

# Publish "Hello world"
response_obj = requests.post("http://localhost:8001/backend", headers={
    "X-Backend-Key": "swordfish",
    "Content-Type": "application/json",
    "Accept": "application/json",
}, data=json.dumps({
    "service": "pubsub_backend",
    "message": {
        "command": "publish",
        "params": {
            "channel": "announcements",
            "message": "Hello World",
        }
    }
}).encode("UTF-8"))
```

The body of the response will also have a JSON object encoded in UTF-8. It will consist on the service response wrapped in the service header.

The following code would print the returned value from the service or signal a failure:

```
if response_obj.status_code != 200:
    print("Non service related error")
else:
    response = json.loads(response.content.decode("UTF-8"))
    # Remove the service envelop
    response = response["message"]
    if response["type"] == "response":
        print("The service replied: " + repr(response["data"]))
    else:
        print("The service signaled an error: " + response["message"])
```

Note: The Snorky HTTP transport is not aware of the RPC system so service failures will

[21] http://docs.python-requests.org/

still send responses with the `200 OK` code, yet they will appoint the error message in the body of the RPC response.

Note that in this case the call returns nothing, so the command response would be `None` (equivalent to JSON `null`).

1.6.5 Using the backend connector

Sending the requests with HTTP libraries is prone to code repetition, so it's advised to write helper functions or classes that take care of the low level communication.

If the application that you are connecting to Snorky is written in Python, you can use the Snorky backend connector for this purpose. For example, the code before would be reduced to this:

```
from snorky.backend import SnorkyBackend, SnorkyHTTPTransport, SnorkyError

backend_http = SnorkyHTTPTransport("http://localhost:8001/backend",
                                   key="swordfish")
backend = SnorkyBackend(backend_http)

response = backend.call("pubsub_backend", "publish",
                        channel="announcements", message="Hello World")
```

The Snorky backend connector will automatically serialize the request into JSON, send it to the HTTP endpoint, receive the response, deserialize it, remove the service header and return the RPC call return value.

If the service returns with error, a `SnorkyError`, specifying the error message as an argument. If there is another kind of error, e.g. the Snorky server is not available, a `RuntimeError` is thrown, with more details in the error argument also.

CHAPTER 2

The DataSync service

2.1 Introduction

The DataSync service provides a systematic way to allow browser clients to fetch data from a database and keep it synchronized as changes occur.

It's important to note that DataSync **is not a database**. You still have to provide the database, the DataSync service only manages routing of change notifications from the source of the changes to the browser clients.

In order to use DataSync in your system you need to fulfill the following requirements:

- Establish which data models you have and a systematic JSON representation for each one.

  For example, in a simple SQL database each relevant table could be a model, and the JSON representation could be a dictionary with all the values of the columns, labeled by field name.

```
{
    "title": "Something that needs to be done",
    "completed": false
}
```

- Find the code paths that make changes to the database and hook them to send notifications to Snorky.

  You could do this with database triggers, listening to ORM events (if you use an ORM) or simply looking for the functions who make changes in the database and modifying them.

  On the Snorky side, in order to work, the only thing that matters is that the change notifications arrive, no matter where they come from.

- Write the dealers. These are small Snorky classes that receive both the data change notifications and client subscriptions. They job is to match every change notification to the adequate subscribed clients.

- Make subscription endpoints in your web application. Browser clients cannot ask Snorky directly for a subscription. Instead, they should ask the server which has access to the database. It's the duty of this server to only allow subscriptions to data the client has access right to access.

2.2 Sending change notifications

Change notifications, also called *deltas*, must be sent to Snorky for each change that may be subscribed by a user.

These notifications must be sent to the `snorky.services.datasync.DataSyncBackend` service, which must be connected to a `snorky.services.datasync.DataSyncService`.

The services will be explained later in more detail, but for now, this is how they could be added to a Snorky server.

```
from snorky.services.datasync import DataSyncService, DataSyncBackend

datasync = DataSyncService("datasync", [
    # dealer list (will be explained in the next chapter)
])
datasync_backend = DataSyncBackend("datasync_backend", datasync)
```

2.2.1 Delta types

There are three delta types:

- **Insertion**: A new element of the model class, e.g. a row, was created.
- **Update**: An already existing element was changed, e.g. a row was edited.
- **Delete**: An element was removed.

2.2.2 Sending deltas

In order to send one or more deltas, an RPC call to `publishDeltas` must be made.

`DataSyncBackend.publishDeltas`(*req*, *deltas*)

Distributes one or more deltas to the appropriate dealers which, in turn, will distribute them to browser clients.

Parameters `deltas` (*list*[22]) – A list of deltas represented as dictionaries.

Each delta dictionary must have the following fields:

- `type`: It must be `"insert"`, `"update"` or `"delete"`, depending on the nature of the change.
- `model`: A name for the model class, e.g. the table name. Different data kinds with different fields should have different values for this property.
- If the delta is an insertion or deletion delta:
  - `data`: The object created or removed, encoded as a JSON entity. It must contain all the fields that may be required to be displayed by the end application.
- If the delta in an update delta:
  - `newData`: The object as a JSON object, after the update was made. It must contain all the fields that may be required to be displayed by the end application.

[22] http://docs.python.org/library/functions.html#list

- `oldData`: The object as a JSON object, before the update was made. It must contain at least enough fields to identify the element that was updated.

  If the model can be subscribed filtered by some fields, the fields used as filter must also be present in order for Snorky to be able to know whether the matched the filters before and after the update.

2.2.3 When to send deltas

Deltas must be sent after the database has been modified. If several changes are being made in an atomic transaction, it's advised not to send the deltas until the transaction has been committed.

2.2.4 Example

The following code sends an insertion delta using the Snorky Python backend connector.

```
from snorky.backend import SnorkyBackend, SnorkyHTTPTransport, SnorkyError

backend_http = SnorkyHTTPTransport("http://localhost:8001/backend",
                                   key="swordfish")
backend = SnorkyBackend(backend_http)

backend.call("datasync_backend", "publish", deltas=[{
    "type": "insert",
    "model": "Task",
    "data": {
        "title": "Send a delta",
        "completed": true
    }
}])
```

2.3 Choosing dealers

Once Snorky receives all relevant deltas, the next step is to re-send them to the interested clients, if any. This is controlled with dealer classes.

2.3.1 What is a dealer

Dealers are classes which track client subscriptions to certain kinds of models.

Dealers also manage the delivery of deltas, by determining which clients are subscribed to the information that they carry.

2.3.2 What is a subscription

In Snorky, clients acquire *subscriptions* to dealers. Each subscription conforms one or more *subscription items*. Each *subscription item* specifies a dealer, and a query to that dealer.

For example, a dealer may be called `CommentsByBlogEntry`. A subscription may contain one subscription item having `CommentsByBlogEntry` as dealer and `15` as query, in order to get notified of new comments in the blog entry with id 15.

2.3.3 The Dealer API

The most basic dealer API is the *`Dealer`* class. You can subclass it to make new dealers.

class `snorky.services.datasync.dealers.Dealer`
Matches dealer data with subscriptions in order to deliver deltas to clients.

`name`
The name of the dealer. If not provided will default to the name of the class.

`model`
The name of the model class that is handled by this dealer. Usually specified as an static attribute.

`add_subscription_item(`*item*`)`
Called everytime a subscription item referring this Dealer is authorized.

`remove_subscription_item(`*item*`)`
Called everytime a subscription is cancelled, once for each subscription item which refers to this Dealer.

`get_subscription_items_for_model(`*model*`)`
Called every time a delta arrives. If the delta is of `update` type, it's called twice, once with the old data and another time with the new data.

It must return an iterable set of the subscription items which represent subscriptions to the provided model.

2.3.4 Simple dealers

Often your dealer only has to filter models by a certain field which clients subscribe to.

For example, the `CommentsByBlogEntry` dealer would receive subscriptions that specify a blog entry id as query, and each time it receives a delta of model `Comment`, it would look which blog entry id it is for, and forward it to those clients which subscribed to it.

Snorky comes with a *`SimpleDealer`* class that leverages this pattern.

class `snorky.services.datasync.dealers.SimpleDealer`
This dealer uses a key function in order to determine which subscription items match which models.

`get_key_for_model(`*model*`)`
A subclass must define a function here that for each model returns a value, usually the value of a certain field.

The dealer will forward deltas to those subscriptions whose query equals the value returned by this function.

Example

```
from snorky.services.datasync.dealers import SimpleDealer

class CommentsByBlogEntry(SimpleDealer):
    name = "CommentsByBlogEntry" # optional
    model = "Comment"

    def get_key_for_model(self, model):
        return model["entryId"]
```

2.3.5 Broadcast dealers

Sometimes you want the clients to receive all deltas for a certain model class, unfiltered. For this purpose there is the *BroadcastDealer* class.

class snorky.services.datasync.dealers.BroadcastDealer
Dealer that matches all deltas with all subscription items, without filters.

Example

```
from snorky.services.datasync.dealers import BroadcastDealer

class AllTasks(BroadcastDealer):
    name = "AllTasks" # optional
    model = "Task"
```

2.3.6 Filter dealers

For cases where clients need to ask for data filtered to complex criteria `FilterDealer` provides an advanced dealer which supports complex filter expressions.

Filter syntax

The subscription query for this dealer must be a JSON list specifying a filter expression in prefix notation. These are some examples:

- `['==', 'color', 'blue']`

  color is 'blue'.

- `['<', 'age', 21]`

  age is less than 21.

- `['>=', 'age', 21]`

  age is greater than or equal to 21.

- `['and', ['==', 'service', 'prosody'], ['>=', 'severity_level', 3]]`

  service is 'prosody' and *severity_level* is greater than or equal to 3.

- `['or', ['not', ['==', 'service', 'java']], ['>=', 'severity_level', 3]]`

  service is not 'java' or *severity_level* is greater than or equal to 3.

- `['==', 'player.color', 'blue']`

  player is a dictionary which contains a property `color`, and the value of that property is 'blue'.

Example

```
from snorky.services.datasync.dealers import BroadcastDealer

class FilteredTasks(FilterDealer):
    name = "FilteredTasks" # optional
    model = "Task"
```

2.4 Authorizing subscriptions

In order to maintain the system secure and to avoid race conditions, browser clients cannot directly ask for subscriptions to Snorky. Instead, they need to ask to another party, usually your web application, to fetch the data and authorize a subscription.

2.4.1 The subscription process

1. The browser client requests both the current data and a subscription. This can be done in a RESTful way with the header `X-Snorky`.
2. The web application sends to Snorky a subscription authorization request to one or more dealers. The web application receives a *subscription token* in return.
3. The web application queries the database.
4. The web application sends to the client both the data and the subscription token.
5. The browser client shows the received data on the user interface, connects to Snorky and sends it the subscription token in order to receive updates.

2.4.2 The authorization request

`DataSyncBackend` provides the RPC command *`DataSyncBackend.authorizeSubscription()`* to request a subscription token.

`DataSyncBackend.authorizeSubscription`(*req*, *items*)

Requests a subscription authorization token.

Parameters `items` (*dict*[23]) – Subscription items to be authorized.

Each item must be a dictionary with two properties: `dealer` and `query`.

[23] http://docs.python.org/library/stdtypes.html#dict

Warning: In order to avoid race conditions, the database query must not be made until the subscription token has been received.

2.4.3 Snorky headers

The subscription mechanism fits well into RESTful APIs. The recommended way to do this is with additional headers.

When a client wants to get both certain data and a subscription to updates in that data, it must send a header `X-Snorky: Subscribe`.

When the server detects this header it must check client permissions and, if it is allowed to do so, it will ask Snorky for a subscription token for the kind of data requested.

For example, if the client requested the comments for the blog post with id 15, it will put in the subscription the dealer `CommentsByBlogEntry` with query `15`.

Once received the token, the server will send it to the client in the response header `X-Subscription`.

The server will query the database for the current comments in the blog entry `15` and write them in the response body.

Note this protocol is merely conventional. You can use whatever protocol you want to ask for subscriptions and return them later.

2.5 Acquiring subscriptions

In order to acquire the subscriptions in the browser clients and handle the updates, Snorky.js provides a DataSync connector.

2.5.1 Basic usage

The DataSync service usage is very simple, just tell *Snorky()* that you need a *Snorky.DataSync()* service.

```
var snorky = new Snorky(WebSocket, "ws://localhost:8001/ws", {
  "datasync": Snorky.DataSync
});
```

class `Snorky.DataSync()`

DataSync service connector.

Events

`deltaReceived`

Event raised when a delta arrives.

The event is dispatched with the delta as argument, being it a dictionary with the following fields:

`model`
: The model class over the change occurred.

`type`
: The type of the delta. Will be either `"insert"`, `"update"` or `"delete"`.

`data`
: The element added or removed. Only in `insert` and `delete` deltas.

`oldData`
: The element before the update. Only in `update` deltas.

`newData`
: The element after the update. Only in `update` deltas.

You can bind the *deltaReceived* event and process the deltas as required by your application.

```
snorky.services.datasync.deltaReceived.add(
  function (delta) {
    if (delta.type == "insert") {
      /* code for insertions */
    } else if (delta.type == "update") {
      /* code for updates */
    } else if (delta.type == "delete") {
      /* code for deletions */
    }
  });
```

2.6 Updating collections

Often an integral part of the delta processing is modifying a collection stored in JS, e.g. in an array. This is specially true if you use a MV* framework like AngularJS.

For example, you may store the initially received elements in an array and later modify it as deltas come from Snorky.

In order to leverage this pattern there is the *Snorky.DataSync.CollectionDeltaProcessor()* class.

class `Snorky.DataSync.CollectionDeltaProcessor(`*collections*, *options*`)`
: Updates one or more collections with deltas received from Snorky.

  Arguments

  - `collections` (*Object*) – A dictionary where each key is a model class name and each value is a collection.
  - `options` (*Object*) – An optional dictionary with additional options.

    `itemsAreEqual`
    : Provides a custom item comparison (see below).

`itemsAreEqual(`*item*, *other*, *delta*`)`
: If return true, two items from a collection will be considered the same. This function is used for processing update and deletion deltas.

The updated or deleted element will be the one that makes this function returns true when compared with the element in the delta.

By default compares the `id` field in both `item` and `other` and returns true if they are equal.

`processDelta(`*delta*`)`
Checks if the delta is associated with any registered collection. If it is, updates the collection adding, updating or removing the matching element.

2.6.1 Collections

A *collection*, as understood by *Snorky.DataSync.CollectionDeltaProcessor()* is a class which allows inserting methods and getting an iterator with update and delete capabilities.

class `Snorky.DataSync.Collection()`
An abstract interface for a collection.

`insert(`*value*`)`
Inserts a new element in the collection.

`getIterator()`
Returns an iterator to the collection.

class `Snorky.DataSync.Iterator()`
An abstract interface for an iterator.

`next()`
Advances the iterator to the next element and returns it.

This method should also be called to retrieve the first element in the collection.

`hasNext()`
True if there are elements in the collection which the iterator has not explored.

`remove()`
Remove the last element returned by *next()* from the collection.

`update(`*newValue*`)`
Replace the last element returned by *next()* with the value provided as argument.

2.6.2 Array collection

The most common collection is the one that is backend by a JavaScript array.

class `Snorky.DataSync.ArrayCollection(`*array*, *options*`)`

Arguments

- `array` (*Array*) – An array, to which this class will expose a collection interface.
- `options` (*Object*) – An optional dictionary of options:

`transformItem`
When an element is inserted or updated, this function will be called with the element to insert or update, and the object returned will be inserted or updated instead.

This is often used when you use *fat models*, that is, you extend the JSON objects that you receive from the server in order to provide helper methods that calculate additional data or perform special operations.

This function gives you the opportunity to add additional methods or perform transformations in the models received from Snorky.

2.6.3 Single item collection

Sometimes the data you synchronize with Snorky is not a list but a single element. *`Snorky.DataSync.SingleItemCollection()`* allows you to update it by providing an update callback.

class `Snorky.DataSync.SingleItemCollection`(*readHandler*, *updateHandler*, *removeHandler*)

Virtual collection of a single item.

Arguments

- `readHandler` (*function*) – Called to get the current item value.
- `updateHandler` (*function*) – Called to update the item value.
- `removeHandler` (*function*) – Optional, called when the item is deleted.

2.6.4 Example usage

The following code would update the array `comments` with the deltas received from Snorky.

```
var collectionProcessor = new Snorky.DataSync.CollectionDeltaProcessor({
  "Comment": new Snorky.DataSync.ArrayCollection(comments)
});

// Delegate delta processing to the collection processor
snorky.services.datasync.deltaReceived.add(function(delta) {
  collectionProcessor.processDelta(delta);
});
```

2.7 Using Snorky with AngularJS

If you use AngularJS you can benefit from Angular Snorky. If you don't, you can skip this.

Angular Snorky is a small library which modifies Snorky to use `$q` promises instead of ES6 promises and automatically triggers `$digest` cycles on event dispatching.

2.7.1 Using Angular Snorky

In order to use Angular Snorky you need to include `angular-snorky.js` after both `angular.js` and `snorky.js`.

Then, you need to add it to the dependencies of your application.

```
angular.module("my-app", [
  /* other dependencies */
  "Snorky"
])
```

After that, the usage of Snorky is exactly the same, you keep on using the Snorky object exported in the global object.

2.8 Using Snorky with Django

Snorky comes with a Django connector which can prove useful if you develop the server side of your application using Django.

2.8.1 Subscribable models

Adding the `snorky.backend.django.subscribable()` decorator to a model class will automatically take care of sending notifications to Snorky with each change.

You only need to provide a `jsonify()` method in the model which returns the representation of the model in a format which can be transformed into JSON.

Example

```
from django.db import models
from snorky.backend.django import subscribable

@subscribable
class Task(models.Model):
    title = models.CharField(max_length=100)
    completed = models.BooleanField(default=False)

    def jsonify(self):
        return {
            "title": self.title,
            "completed": self.completed,
        }
```

2.8.2 Subscribable REST views

If you use Django REST Framework[24] for offering a REST API, you can also use the `ListSubscribeModelMixin` which extends `ListModelMixin` to provide Snorky subscription support.

[24] http://www.django-rest-framework.org/

class snorky.backend.django.rest_framework.ListSubscribeModelMixin

Provides a `list()` method which understands the `X-Snorky` header.

`get_subscription_items()`

Returns a list of dictionaries of dealer and queries which will be sent to Snorky to authorize a subscription.

By default it returns a list of only one item, with *get_dealer()* as dealer and *get_dealer_query()* as query.

`get_dealer()`

Returns the dealer this model is associated with.

By default it returns the value of the property *dealer*, if any.

`get_dealer_query()`

Returns the query which will be sent to the dealer.

By default it returns the value of the property *dealer_query*.

`dealer`

The dealer name to whom subscription will be bound, if *get_dealer()* is not redefined.

`dealer_query`

The query which will be sent to the dealer, if *get_dealer_query()* is not redefined.

Example

```
from rest_framework import viewsets
import snorky.backend.django.rest_framework as snorky


class TaskViewSet(snorky.ListSubscribeModelMixin,
                  viewsets.ModelViewSet):
    model = models.Task
    dealer = "AllTasks"
    dealer_query = None
```

CHAPTER 3

Release notes

3.1 What's new in Snorky 0.1.0-a5

3.1.1 March 28, 2016

@Alternhuman[25] contributed several features and bug fixes:

- Fixed bug #3[26] that affected Django 1.8+ integration.
- Added a new React.js demo.

On top of that:

- Both ToDo demo applications have been updated to Django 1.9 and Django REST Framework 3.3.

  This also has allowed them to work in Python 3.5, which is incompatible with older Django versions.

3.2 What's new in Snorky 0.1.0-a4

3.2.1 Jan 30, 2016

Bugfix release, which handles a `bytes` vs `str` issue in the ping pong feature introduced in `0.1.0-a3` when running in Python 3.

3.3 What's new in Snorky 0.1.0-a3

3.3.1 Apr 14, 2015

Here is a new release with new features.

[25] https://github.com/Alternhuman
[26] https://github.com/ntrrgc/snorky/issues/3

- `compare_digest`, a function that provides constant-time string comparison in Python 2.7.7+ and Python 3.3+ has been replaced with streql[27], a third party module that performs the same task, but it is compatible with older Python versions.

  This is required to use Snorky in CentOS 7 with Python 2 without building the interpreter from source.

- Added a `SNORKY_JSON_ENCODER` setting in the Django connector, allowing to change the JSON encoder class (as if `cls` parameter of `json.dumps()` was specified).
- Added debug logging to the Python backend connector. The logging channel is 'snorky'. If the logging level is `DEBUG` it will emit a line for each message sent or received between the client and Snorky.
- When `DataSyncBackend` does not receive a field now it will include its name in the error message (before only *Missing field* was returned).
- Now `SnorkyWebSocketHandler` pings clients each 90 seconds by default. This is needed in order to not let NAT routers drop long lived otherwise inactive connections.

  The interval can be customized with the `ping_pong_interval` option of `SnorkyWebSocketHandler`. For example, in order to ping the clients with a frequency of 3 minutes (180 seconds) you could use this code:

```
app_frontend = Application([
    SnorkyWebSocketHandler.get_route(frontend, "/ws",
          ping_pong_interval=180),
])
```

3.4 What's new in Snorky 0.1.0-a2

3.4.1 Apr 12, 2015

Updated and finally published in the open!

- MPL 2.0 license has been added.
- This project is now in GitHub.
- Added bundled and minified builds for the JavaScript client-side connector.
- A bug fix in PubSub connector.
- Trivial API change: `@subscriptable` decorator renamed to `@subscribable`.

3.5 First release: Snorky 0.1.0-a1

3.5.1 Sep 4, 2014

First version, released as a final year project for the University of Salamanca.

[27] https://pypi.python.org/pypi/streql/3.0.2

This version establishes the main components of the framework:

- Basic service architecture.
- Data synchronization with databases using `DataSyncService`.
- JavaScript client-side connector.
  - With additional support for AngularJS framework.
- Python server-side connector:
  - With additional support for Django and Django REST Framework.