# n6sdk Documentation

***Release 0.6.1***

**NASK**

April 27, 2016

## Contents

CHAPTER 1

# *n6sdk*: Server-side Software Development Kit for *n6*

*n6* (Network Security Incident eXchange) is a system to collect, manage and distribute security information on a large scale (see: http://www.cert.pl/projekty/langswitch_lang/en). Distribution is realized through a simple REST API that authorized users can use to receive various types of data, in particular information on threats to their networks.

The *n6sdk* library was created to foster exchange of information across organizations. The library makes it easier to implement an *n6*-like REST API that provides access to your own source of security-related data.

## 1.1 Basic References

- **Home page:** https://github.com/CERT-Polska/n6sdk
- **Documentation:** http://n6sdk.readthedocs.org/

## 1.2 Copyright, License and Authors

### 1.2.1 Basic Legal Information

### 1.2.2 Main Authors

The developers of *n6sdk* are:

- Jan Kaliszewski,
- Mateusz Kukawski,
- Łukasz Michalak,

- Marcin Ptak,
- Mariusz Szot,

*NASK, Software Development Department*.

The project is developed for CERT Polska. Contact us via e-mail: n6@cert.pl.

### 1.2.3 Supplementary Attributions

Small portions of the software have been created and are copyrighted by other parties; appropriate notes are placed in the concerned source code files.

Moreover,

- major portions of the `n6sdk/scaffolds/basic_n6sdk_scaffold/*_tmpl` files were generated by the tools provided by the Pyramid web framework;
- major portions of the `docs/Makefile` and `docs/source/conf.py` files were generated by the Sphinx documentation tool.

### 1.2.4 Acknowledgements

This work is partially supported by the Strategic International Collaborative R&D Promotion Project of the Ministry of Internal Affairs and Communication, Japan, and by the European Union Seventh Framework Programme (FP7/2007–2013) under grant agreement No. 608533 (NECOMA).

For more information on the NECOMA project see: http://www.necoma-project.eu/

CHAPTER 2

# Tutorial

This tutorial describes how to use the *n6sdk* library to implement an *n6*-like REST API that provides access to your own network incident data source.

## 2.1 Setting up the development environment

### 2.1.1 Prerequisites

You need to have:

- A Linux system + the *bash* shell used to interact with it + basic Unix-like OS tools such as *mkdir*, *cat* etc. (other platforms and tools could also be used – but this tutorial assumes using the aforementioned ones) + your favorite text editor installed;
- the *Python 2.7* language interpreter installed (on Debian GNU/Linux it can be installed with the command: `sudo apt-get install python2.7`);
- The *git* version control system installed (on Debian GNU/Linux it can be installed with the command: `sudo apt-get install git`);
- the *virtualenv* tool installed (see: http://virtualenv.readthedocs.org/en/latest/virtualenv.html; on Debian GNU/Linux it can be installed with the command: `sudo apt-get install python-virtualenv`);
- Internet access.

### 2.1.2 Obtaining the *n6sdk* source code

We will start with creating the "workbench" directory for all our activities:

```
$ mkdir <the workbench directory>
```

(Of course, `<the workbench directory>` needs to be replaced with the actual name (absolute path) of the directory you want to create.)

Then, we need to clone the *n6sdk* source code repository:

```
$ cd <the workbench directory>
$ git clone https://github.com/CERT-Polska/n6sdk.git
```

Now, in the `<the workbench directory>/n6sdk/` subdirectory we have the source code of the *n6sdk* library.

### 2.1.3 Installing the necessary stuff

Next, we will create and activate our Python *virtual environment*:

```
$ virtualenv dev-venv
$ source dev-venv/bin/activate
```

Then, we can install the *n6sdk* library:

```
$ cd n6sdk
$ python setup.py install
```

Then, we need to create our project:

```
$ cd ..
$ pcreate -s n6sdk Using_N6SDK
```

– where `Using_N6SDK` is the name of our new *n6sdk*-based project. Obviously, when creating your real project you will want to pick another name. Anyway, for the rest of this tutorial we will use `Using_N6SDK` as the project name (and, consequently, `using_n6sdk` as the "technical" package name, automatically derived from the given project name).

Now, we have the skeleton of our new project. You may want to customize some details in the newly created files, especially the *version* and *description* fields in `Using_N6SDK/setup.py`.

Then, we need to install our new project *for development*:

```
$ cd Using_N6SDK
$ python setup.py develop
$ cd ..
```

We can check whether everything up to now went well by running the Python interpreter...

```
$ python
```

...and trying to import some of the installed components:

```
>>> import n6sdk
>>> import n6sdk.data_spec.fields
>>> n6sdk.data_spec.fields.Field
<class 'n6sdk.data_spec.fields.Field'>
>>> import using_n6sdk
>>> exit()
```

## 2.2 Data processing and architecture overview

When a client sends a **HTTP request** to the *n6 REST API*, the following data processing is performed on the server side:

1. **Receiving the HTTP request**

   *n6sdk* uses the *Pyramid* library (see: http://docs.pylonsproject.org/projects/pyramid/en/1.5-branch/) to perform processing related to HTTP communication, request data (for example, extracting query parameters from the URL's query string) and routing (deciding what function shall be invoked with what arguments depending on the given URL) – however there are the *n6sdk*-specific wrappers and helpers used to adjust some important factors: `n6sdk.pyramid_commons.DefaultStreamViewBase`, `n6sdk.pyramid_commons.HttpResource` and `n6sdk.pyramid_commons.ConfigHelper` (see below: *Gluing it together*). These three classes can be customized by subclassing them and extending selected methods, however it is beyond the scope of this tutorial.

2. **Authentication**

   Authentication is performed using a mechanism provided by the *Pyramid* library: *authentication policies*. The simplest policy is implemented as the `n6sdk.pyramid_commons.AnonymousAuthenticationPolicy` class (it is a dummy policy: all clients are identified as `"anonymous"`); it can be replaced with a custom one (see below: *Custom authentication policy*).

   The result is an object containing authentication data.

3. **Cleaning query parameters provided by the client**

   Here "cleaning" means: validation and adjustment (normalization) of the parameters (already extracted from the request's URL).

   An instance of a *data specification class* (see below: *Data specification class*) is responsible for doing that.

   The result is a dictionary containing the cleaned query parameters.

4. **Retrieving result data from the data backend API**

   The *data backend API*, responsible for interacting with the actual data storage, needs to be implemented as a class (see below: *Implementing the data backend API*).

   For a client request (see above: *1. Receiving the HTTP request*), an appropriate method of the sole instance of this class is called with the authentication data (see above: *2. Authentication*) and the cleaned client query parameters dictionary (see above: *3. Cleaning query parameters...*) as call arguments.

   The result of the call is an iterator which yields dictionaries, each containing the data of one network incident.

5. **Cleaning the result data**

   Each of the yielded dictionaries is cleaned. Here "cleaning" means: validation and adjustment (normalization) of the result data.

   An instance of a *data specification class* (see below: *Data specification class*) is responsible for doing that.

   The result is another iterator (which yields dictionaries, each containing cleaned data of one network incident).

6. **Rendering the HTTP response**

   The yielded cleaned dictionaries are processed to produce consecutive fragments of the HTTP response which are successively sent to the client. The key component responsible for transforming the dictionaries into the response body is a *renderer*. Note that *n6sdk* renderers (being a custom *n6sdk* concept, different from *Pyramid* renderers) are able to process data in an iterator ("stream-like") manner, so even if the resultant response body is huge it does not have to fit as a whole in the server's memory.

   The *n6sdk* library provides two standard renderers: `json` (to render JSON-formatted responses) and `sjson` (to render responses in a format similar to JSON but more convenient for "stream-like" or "pipeline" data processing).

   Implementing and registering custom renderers is possible, however it is beyond the scope of this tutorial.

## 2.3 Data specification class

### 2.3.1 Basics

A *data specification* determines:

- how query parameters (already extracted from the query string part of the URL of a client HTTP request) are cleaned (before being passed in to the data backend API) – that is:
  - what are the legal parameter names;
  - whether particular parameters are required or optional;
  - what are valid values of particular parameters (e.g.: a `time.min` value must be a valid *ISO-8601*-formatted date and time);
  - whether, for a particular parameter, there can be many alternative values or only one value (e.g.: `time.min` can have only one value, and `ip` can have multiple values);
  - how particular parameter values are normalized (e.g.: a `time.min` value is always transformed to a Python `datetime.datetime` object, converting any time zone information to UTC);
- how result dictionaries (each containing data of one incident) yielded by the data backend API are cleaned (before being passed in to a response renderer) – that is:
  - what are the legal result keys;
  - whether particular items are required or optional;
  - what are valid types and values of particular items (e.g.: a `time` value must be either a `datetime.datetime` object or a string being a valid *ISO-8601*-formatted date and time);
  - how particular items are normalized (e.g.: a `time` value is always transformed to a Python `datetime.datetime` object, converting any time zone information to UTC).

The declarative way of defining a *data specification* is somewhat similar to domain-specific languages known from ORMs (such as the *SQLAlchemy*'s or *Django*'s ones): a data specification class (`n6sdk.data_spec.DataSpec` or some subclass of it) looks like an ORM "model" class and particular query parameter and result item specifications (being instances of `n6sdk.data_spec.fields.Field` or of subclasses of it) are declared similarly to ORM "fields" or "columns".

For example, consider the following simple data specification class:

```
class MyDataSpecFromScratch(n6sdk.data_spec.BaseDataSpec):

    id = UnicodeLimitedField(
        in_params='optional',
        in_result='required',
        max_length=64,
    )

    time = DateTimeField(
        in_params=None,
        in_result='required',

        extra_params=dict(
            min=DateTimeField(             # `time.min`
                in_params='optional',
                single_param=True,
            ),
            max=DateTimeField(             # `time.max`
                in_params='optional',
                single_param=True,
            ),
            until=DateTimeField(           # `time.until`
                in_params='optional',
                single_param=True,
            ),
        ),
    )

    address = ExtendedAddressField(
        in_params=None,
        in_result='optional',
    )

    ip = IPv4Field(
        in_params='optional',
        in_result=None,

        extra_params=dict(
            net=IPv4NetField(            # `ip.net`
                in_params='optional',
            ),
        ),
    )

    asn = ASNField(
        in_params='optional',
        in_result=None,
    )

    cc = CCField(
        in_params='optional',
        in_result=None,
    )

    count = IntegerField(
        in_params=None,
        in_result='optional',
        min_value=0,
        max_value=(2 ** 15 - 1),
    )
```

**Note:** In a real project you should inherit from `DataSpec` rather than from `BaseDataSpec`. See the following sections, especially *Your first data specification class*.

What do we see in the above listing is that:

1. `id` is a text field: its values are strings, not longer than 64 characters (as its declaration is an instance of `n6sdk.data_spec.fields.UnicodeLimitedField` created with the constructor argument *max_length* set to `64`). It is **optional** as a query parameter and **required** (obligatory) as an item of a result dictionary.
2. `time` is a date-and-time field (as its declaration is an instance of `n6sdk.data_spec.fields.DateTimeField`). It is **not** a legal query parameter, and it is **required** as an item of a result dictionary.
3. `time.min`, `time.max` and `time.until` are date-and-time fields (as their declarations are instances of `n6sdk.data_spec.fields.DateTimeField`). They are **optional** as query parameters, and they are **not** legal items of a result dictionary. Unlike most of other fields, these three fields do not allow to specify multiple query parameter values (note the constructor argument *single_param* set to `True`).
4. `address` is a field whose values are lists of dictionaries containing `ip` and optionally `asn` and `cc` (as the declaration of `address` is an instance of `n6sdk.data_spec.fields.AddressField`). It is **not** a

legal query parameter, and it is **optional** as an item of a result dictionary.

5. `ip` is an IPv4 address field (as its declaration is an instance of `n6sdk.data_spec.fields.IPv4Field`). It is **optional** as a query parameter and it is **not** a legal item of a result dictionary (note that in a result dictionary the `address` field contains the corresponding data).
6. `ip.net` is an IPv4 network definition (as its declaration is an instance of `n6sdk.data_spec.fields.IPv4NetField`). It is **optional** as a query parameter and it is **not** a legal item of a result dictionary.
7. `asn` is an autonomous system number (ASN) field (as its declaration is an instance of `n6sdk.data_spec.fields.ASNField`). It is **optional** as a query parameter and it is **not** a legal item of a result dictionary (note that in a result dictionary the `address` field contains the corresponding data).
8. `cc` is 2-letter country code field (as its declaration is an instance of `n6sdk.data_spec.fields.CCField`). It is **optional** as a query parameter and it is **not** a legal item of a result dictionary (note that in a result dictionary the `address` field contains the corresponding data).
9. `count` is an integer field: its values are integer numbers, not less than 0 and not greater than 32767 (as the declaration of `count` is an instance of `n6sdk.data_spec.fields.IntegerField` created with the constructor arguments: *min_value* set to 0 and *max_value* set to 32767). It is **not** a legal query parameter, and it is **optional** as an item of a result dictionary.

To create your data specification class you will, most probably, want to inherit from `n6sdk.data_spec.DataSpec`. In its subclass you can:

- add new field specifications as well as modify (extend), replace or remove (mask) field specifications defined in `DataSpec`;
- extend the `DataSpec`'s cleaning methods.

(See comments in `Using_N6SDK/using_n6sdk/data_spec.py` as well as descriptions in the following sections of this tutorial.)

You may also want to subclass `n6sdk.data_spec.fields.Field` (or any of its subclasses, such as `UnicodeLimitedField`, `IPv4Field` or `IntegerField`) to create new kinds of fields whose instances can be used as field specifications in your data specification class (see below...).

## 2.3.2 Your first data specification class

**Let us open the** `<the workbench directory>/Using_N6SDK/using_n6sdk/data_spec.py` **file with our favorite text editor and uncomment the following lines in it** (within the body of the `UsingN6sdkDataSpec` class):

```
id = Ext(in_params='optional')

source = Ext(in_params='optional')

restriction = Ext(in_params='optional')

confidence = Ext(in_params='optional')

category = Ext(in_params='optional')

time = Ext(
    extra_params=Ext(
        min=Ext(in_params='optional'),    # search for >= than...
        max=Ext(in_params='optional'),    # search for <= than...
        until=Ext(in_params='optional'),  # search for <  than...
    ),
)

ip = Ext(
    in_params='optional',
)

url = Ext(
    in_params='optional',
)
```

Our `UsingN6sdkDataSpec` data specification class is a subclass of `n6sdk.data_spec.DataSpec` which, by default, has all query parameters **disabled** – so here we **enabled** *some* of them by uncommenting these lines. (We can remove the rest of commented lines.)

**Note:** You should always ensure that you *do not* enable in your *data specification class* any query parameters that are *not* supported by your *data backend API* (see: *Implementing the data backend API*).

Apart from changing (extending) inherited field specifications, we can also add some new fields. For example, **let us add, near the beginning of our data specification class definition, a new field specification:** `mac_address`.

```
from n6sdk.data_spec import DataSpec, Ext
from n6sdk.data_spec.fields import UnicodeRegexField  # remember to add this line


class UsingN6sdkDataSpec(DataSpec):

    """
    The data specification class for the `Using_N6SDK` project.
    """

    mac_address = UnicodeRegexField(
        in_params='optional',  # *can* be in query params
        in_result='optional',  # *can* be in result data

        regex=r'^(?:[0-9A-F]{2}(?:[:-]|$)){6}$',
        error_msg_template=u'"{}" is not a valid MAC address',
    )
```

(Of course, we *do not remove* the lines uncommented earlier.)

### 2.3.3 More knowledge about data specification...

#### The data specification's cleaning methods

The most important methods of any *data specification* (typically, an instance of `n6sdk.data_spec.DataSpec` or of its subclass) are:

- `clean_param_dict()` – used to clean client query parameters;
- `clean_result_dict()` – used to clean results yielded by the data backend API.

Normally, these methods are called automatically by the *n6sdk* machinery.

Each of these methods takes *exactly one positional argument* which is respectively:

- for `clean_param_dict()` – a **dictionary of query parameters** (representing one client request); the dictionary maps field names (query parameter names) to **lists of their raw values** (lists – because, as it was said, for most fields there can be more than one query parameter value);
- for `clean_result_dict()` – a **single result dictionary** (representing one network incident); the dictionary maps field names (result keys) to **their raw values**.

(Here "raw" is a synonym of "uncleaned".)

Each of these methods also accepts the following *optional keyword-only arguments*:

- *ignored_keys* – an iterable (e.g., a set or a list) of keys that will be completely ignored (i.e., the processed dictionary that has been given as the positional argument will be treated as it did not contain any of these keys; therefore, the resultant dictionary will not contain them either);
- *forbidden_keys* – an iterable of keys that *must not apperar* in the processed dictionary;
- *extra_required_keys* – an iterable of keys that *must appear* in the processed dictionary;
- *discarded_keys* – an iterable of keys that will be removed (discarded) *after* validation of the processed dictionary keys (but *before* cleaning the values).

If a raw value is not valid and cannot be cleaned (see below: *The field's cleaning methods*) or any other data specification constraint is violated (including those specified with the *forbidden_keys* and *extra_required_keys* arguments mentioned above) an exception – respectively: `ParamKeyCleaningError` or `ParamValueCleaningError`, or `ResultKeyCleaningError`, or `ResultValueCleaningError` – is raised.

Otherwise, *a new dictionary* is returned (the input dictionary given as the positional argument *is not modified*). Regarding returned dictionaries:

- a dictionary returned by `clean_param_dict()` maps field names (query parameter names) to **lists of cleaned query parameter values**;
- a dictionary returned by `clean_result_dict()` (containing cleaned data of exactly one network incident) maps field names (result keys) to **cleaned result values**.

### The field's cleaning methods

The most important methods of any *field* (an instance of `n6sdk.data_spec.fields.Field` or of its subclass) are:

- `clean_param_value()` – called to clean a single query parameter value;
- `clean_result_value()` – called to clean a single result value.

Each of these methods takes exactly *one positional argument*: a single uncleaned (raw) value.

Each of these methods returns *a single value*: a cleaned one.

These methods are called by the data specification machinery in the following way:

- The data specification's method `clean_param_dict()` (described above in the *The data specification's cleaning methods* section) calls the `clean_param_value()` method of the appropriate field – separately **for each element of each of the raw value lists taken from the dictionary passed as the argument**.

  If the field's method raises (or propagates) an exception being an instance/subclass of `Exception` (i.e., practically *any* exception, excluding `KeyboardInterrupt`, `SystemExit` and a few others), the data specification's method `clean_param_dict()` catches and collects it (doing the same for any such exceptions raised for other values, possibly for other fields) and then raises `ParamValueCleaningError`.

**Note:** If the exception raised (or propagated) by the field's method is *FieldValueError* (or any other exception derived from *_ErrorWithPublicMessageMixin*) its *public_message* will be included in the *ParamValueCleaningError*'s `public_message`).

- the data specification's method `clean_result_dict()` (described above in the *The data specification's cleaning methods* section) calls the `clean_result_value()` method of the appropriate field – **for each raw value from the dictionary passed as the argument**.

  If the field's method raises (or propagates) an exception being an instance/subclass of `Exception` (i.e., practically *any* exception, excluding `KeyboardInterrupt`, `SystemExit` and a few others), the data specification's method `clean_result_dict()` catches and collects it (doing the same for any such exceptions raised for other fields) and then raises *ResultValueCleaningError*.

  **Note:** Unlike *ParamValueCleaningError* raised by `clean_param_dict()`, the *ResultValueCleaningError* exception raised by `clean_result_dict()` in reaction to exception(s) from `clean_result_value()` *does not* include in its `public_message` any information from the underlying exception(s) (instead of that, *ResultValueCleaningError*'s `public_message` is set to the safe default: `u"Internal error."`).

  The rationale for this behaviour is that any exceptions related to *result cleaning* are strictly internal (contrary to those related to *query parameter cleaning*).

  Thanks to this behaviour, much of the code of field classes that is related to parameter value cleaning can also be used for result value cleaning without concern about disclosing some sensitive details in `public_message` of *ResultValueCleaningError*.

  **Warning:** For security sake, when extending `n6sdk.data_spec.BaseDataSpec.clean_result_dict()` ensure that your implementation behaves in the same way as described in this *note*.

## Overview of the basic data specification classes

The `n6sdk.data_spec.DataSpec` and `n6sdk.data_spec.AllSearchableDataSpec` classes are two variants of a base class for your own data specification class.

Each of them defines all standard *n6-like* REST API fields – but:

- `DataSpec` – has *all query parameters* **disabled**. This makes the class suitable for most *n6sdk* uses: in your subclass of `DataSpec` you will *need to enable* (typically, with a `<field name> = Ext(in_params='optional')` declaration) only those query parameters that your data backend supports.
- `AllSearchableDataSpec` – has *all query parameters* **enabled**. This makes the class suitable for cases when your data backend supports all or most of standard *n6* query parameters. In your subclass of `AllSearchableDataSpec` you will need to *disable* (typically, with a `<field name> = Ext(in_params=None)` declaration) those query parameters that your data backend *does not* support.

The following list describes briefly all field specifications defined in these two classes.

- basic event data fields:
  - `id`:
    - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    - *in result:* **required**

    * *field class:* `UnicodeLimitedField`
    * *specific field constructor arguments:* `max_length=64`
    * *param/result cleaning example:*
        · *raw value:* `"abcDEF... \xc5\x81"`
        · *cleaned value:* `u"abcDEF... \u0141"`

  Unique incident identifier being an arbitrary text. Maximum length: 64 characters (after cleaning).

- `source`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* **required**
    * *field class:* `SourceField`
    * *param/result cleaning example:*
        · *raw value:* `"some-org.some-type"`
        · *cleaned value:* `u"some-org.some-type"`

  Incident data source identifier. Consists of two parts separated with a dot (`.`). Allowed characters (apart from the dot) are: ASCII lower-case letters, digits and hyphen (`-`). Maximum length: 32 characters (after cleaning).

- `restriction`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* **required**
    * *field class:* `UnicodeEnumField`
    * *specific field constructor arguments:* `enum_values=n6sdk.data_spec.RESTRICTION_ENUMS`
    * *param/result cleaning example:*
        · *raw value:* `"public"`
        · *cleaned value:* `u"public"`

  Data distribution restriction qualifier. One of: `"public"`, `"need-to-know"` or `"internal"`.

- `confidence`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* **required**
    * *field class:* `UnicodeEnumField`
    * *specific field constructor arguments:* `enum_values=n6sdk.data_spec.CONFIDENCE_ENUMS`
    * *param/result cleaning example:*
        · *raw value:* `"medium"`
        · *cleaned value:* `u"medium"`

  Data confidence qualifier. One of: `"high"`, `"medium"` or `"low"`.

- `category`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* **required**

    * *field class:* `UnicodeEnumField`
    * *specific field constructor arguments:* `enum_values=n6sdk.data_spec.CATEGORY_ENUMS`
    * *param/result cleaning example:*
        · *raw value:* `"bots"`
        · *cleaned value:* `u"bots"`

  Incident category label (some examples: `"bots"`, `"phish"`, `"scanning"`...).

- `time`
    * *in params:* N/A
    * *in result:* **required**
    * *field class:* `DateTimeField`
    * *result cleaning examples:*
        · *example synonymous raw values:*
        · `"2014-11-05T23:13:00.000000"` or
        · `"2014-11-06 01:13+02:00"` or
        · `datetime.datetime(2014, 11, 5, 23, 13, 0)` or
        · `datetime.datetime(2014, 11, 6, 1, 13, 0, 0, <tzinfo with UTC offset 2h>)`
        · *cleaned value:* `datetime.datetime(2014, 11, 5, 23, 13, 0)`

  Incident *occurrence* time (**not** *when-entered-into-the-database*). Value cleaning includes conversion to UTC time.

- `time.min`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`, marked as **single_param** in both
    * *in result:* N/A
    * *field class:* `DateTimeField`
    * *param cleaning examples:*
        · *example synonymous raw values:*
        · `"2014-11-06T01:13+02:00"` or
        · `u"2014-11-05 23:13:00.000000"`
        · *cleaned value:* `datetime.datetime(2014, 11, 5, 23, 13, 0)`

  The *earliest* time the queried incidents *occurred* at. Value cleaning includes conversion to UTC time.

- `time.max`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`, marked as **single_param** in both
    * *in result:* N/A
    * *field class:* `DateTimeField`
    * *param cleaning examples:*

- *example synonymous raw values:*
- `u"2014-11-06T01:13+02:00"` or
- `"2014-11-05 23:13:00.000000"`
- *cleaned value:* `datetime.datetime(2014, 11, 5, 23, 13, 0)`

The *latest* time the queried incidents *occurred* at. Value cleaning includes conversion to UTC time.

- `time.until`:
  - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`, marked as **single_param** in both
  - *in result:* N/A
  - *field class:* `DateTimeField`
  - *param cleaning examples:*
    - *example synonymous raw values:*
    - `u"2014-11-06T01:13+02:00"` or
    - `"2014-11-05 23:13:00.000000"`
    - *cleaned value:* `datetime.datetime(2014, 11, 5, 23, 13, 0)`

  The time the queried incidents *occurred before* (i.e., exclusive; a handy replacement for `time.max` in some cases). Value cleaning includes conversion to UTC time.

- `address`-related fields:
  - `address`
    - *in params:* N/A
    - *in result:* **optional**
    - *field class:* `ExtendedAddressField`
    - *result cleaning examples:*
      - *example synonymous raw values:*
      - `[{"ipv6": "::1"}, {"ip": "123.10.234.169", "asn": 999998}]` or
      - `[{u"ipv6": "::0001"}, {"ip": "123.10.234.169", u"asn": "999998"}]` or
      - `[{"ipv6": "0000:0000::0001"}, {u"ip": "123.10.234.169", u"asn": "15.16958"}]`
      - *cleaned value:* `[{u"ipv6": u"::1"}, {u"ip": "123.10.234.169", u"asn": 999998}]`

    Set of network addresses related to the returned incident (e.g., for malicious web sites: taken from DNS *A* or *AAAA* records; for sinkhole/scanning: communication source addresses) – in the form of a list of dictionaries, each containing:

    - obligatorily:
      - either `"ip"` (IPv4 address in quad-dotted decimal notation, cleaned using a subfield being an instance of `IPv4Field`)

    · or `"ipv6"` (IPv6 address in the standard text representation, cleaned using a subfield being an instance of `IPv6Field`)

  – but *not* both `"ip"` and `"ipv6"`;

* plus optionally – all or some of:

    · `"asn"` (autonomous system number in the form of a number or two numbers separated with a dot, cleaned using a subfield being an instance of `ASNField`),

    · `"cc"` (two-letter country code, cleaned using a subfield being an instance of `CCField`),

    · `"dir"` (the indicator of the address role in terms of the direction of the network flow in layers 3 or 4; one of: `"src"`, `"dst"`; cleaned using a subfield being an instance of `DirField`),

    · `"rdns"` (the domain name from the PTR record of the `.in-addr-arpa` domain associated with the IP address, without the trailing dot; cleaned using a subfield being an instance of `DomainNameField`).

**Note:** The cleaned IPv6 addresses is in the "condensed" form – in contrast to the "exploded" form used for *param cleaning* of *ipv6* and *ipv6.net*. .

– `ip`:

  * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
  * *in result:* N/A
  * *field class:* `IPv4Field`
  * *param cleaning example:*
    · *raw value:* `"123.10.234.168"`
    · *cleaned value:* `u"123.10.234.168"`

  IPv4 address (in quad-dotted decimal notation) related to the queried incidents.

– `ip.net`:

  * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
  * *in result:* N/A
  * *field class:* `IPv4NetField`
  * *param cleaning example:*
    · *raw value:* `"123.10.234.0/24"`
    · *cleaned value:* `(u"123.10.234.0", 24)`

  IPv4 network (in CIDR notation) containing IP addresses related to the queried incidents.

– `ipv6`:

  * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
  * *in result:* N/A
  * *field class:* `IPv6Field`
  * *param cleaning examples:*

            · *example synonymous raw values:*
            · `u"abcd::1"` or
            · `"ABCD::1"` or
            · `u"ABCD:0000:0000:0000:0000:0000:0000:0001"`
            · `"abcd:0000:0000:0000:0000:0000:0000:0001"` or
            · *cleaned value:* `u"abcd:0000:0000:0000:0000:0000:0000:0001"`

    IPv6 address (in the standard text representation) related to the queried incidents.

    **Note:** Cleaned values are in the "exploded" form – in contrast to the "condensed" form used for *result cleaning* of *address*.

  – `ipv6.net`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* N/A
    * *field class:* `IPv6NetField`
    * *param cleaning examples:*
      · *example synonymous raw values:*
      · `"abcd::1/128"` or
      · `u"ABCD::1/128"` or
      · `"ABCD:0000:0000:0000:0000:0000:0000:0001/128"`
      · `u"abcd:0000:0000:0000:0000:0000:0000:0001/128"` or
      · *cleaned value:* `(u"abcd:0000:0000:0000:0000:0000:0000:0001", 128)`

    IPv6 network (in CIDR notation) containing IPv6 addresses related to the queried incidents.

    **Note:** The address part of each cleaned value is in the "exploded" form – in contrast to the "condensed" form used for *result cleaning* of *address*.

  – `asn`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* N/A
    * *field class:* `ASNField`
    * *param cleaning examples:*
      · *example synonymous raw values:*
      · `u"999998"` or
      · `u"15.16958"`
      · *cleaned value:* `999998`

    Autonomous system number of IP addresses related to the queried incidents; in the form of a number or two numbers separated with a dot (see the examples above).

    - `cc`:
        - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
        - *in result:* N/A
        - *field class:* `CCField`
        - *param cleaning example:*
            - *raw value:* `"US"`
            - *cleaned value:* `u"US"`

      Two-letter country code related to IP addresses related to the queried incidents.

- fields related to *black list* events:
    - `expires`:
        - *in params:* N/A
        - *in result:* **optional**
        - *field class:* `DateTimeField`
        - *result cleaning examples:*
            - *example synonymous raw values:*
            - `"2014-11-05T23:13:00.000000"` or
            - `"2014-11-06 01:13+02:00"` or
            - `datetime.datetime(2014, 11, 5, 23, 13, 0)` or
            - `datetime.datetime(2014, 11, 6, 1, 13, 0, 0, <tzinfo with UTC offset 2h>)`
            - *cleaned value:* `datetime.datetime(2014, 11, 5, 23, 13, 0)`

      Black list item *expiry* time. Value cleaning includes conversion to UTC time.
    - `active.min`:
        - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`, marked as **single_param** in both
        - *in result:* N/A
        - *field class:* `DateTimeField`
        - *param cleaning examples:*
            - *example synonymous raw values:*
            - `"2014-11-05T23:13:00.000000"` or
            - `"2014-11-06 01:13+02:00"`
            - *cleaned value:* `datetime.datetime(2014, 11, 5, 23, 13, 0)`

      The *earliest* expiry-or-occurrence time of the queried black list items. Value cleaning includes conversion to UTC time.
    - `active.max`:
        - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`, marked as **single_param** in both

    * *in result:* N/A
    * *field class:* `DateTimeField`
    * *param cleaning examples:*
        - *example synonymous raw values:*
        - `u"2014-11-05T23:13:00.000000"` or
        - `u"2014-11-06 01:13+02:00"`
        - *cleaned value:* `datetime.datetime(2014, 11, 5, 23, 13, 0)`

  The *latest* expiry-or-occurrence time of the queried black list items. Value cleaning includes conversion to UTC time.

- `active.until`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`, marked as **single_param** in both
    * *in result:* N/A
    * *field class:* `DateTimeField`
    * *param cleaning examples:*
        - *example synonymous raw values:*
        - `u"2014-11-06T01:13+02:00"` or
        - `"2014-11-05 23:13:00.000000"`
        - *cleaned value:* `datetime.datetime(2014, 11, 5, 23, 13, 0)`

  The time the queried incidents *expired or occurred before* (i.e., exclusive; a handy replacement for `active.max` in some cases). Value cleaning includes conversion to UTC time.

- `replaces`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* **optional**
    * *field class:* `UnicodeLimitedField`
    * *specific field constructor arguments:* `max_length=64`
    * *param/result cleaning example:*
        - *raw value:* `"abcDEF"`
        - *cleaned value:* `u"abcDEF"`

  `id` of the black list item replaced by the queried/returned one. Maximum length: 64 characters (after cleaning).

- `status`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* **optional**
    * *field class:* `UnicodeEnumField`
    * *specific field constructor arguments:* `enum_values=n6sdk.data_spec.STATUS_ENUMS`
    * *param/result cleaning example:*

        · *raw value:* `"active"`
        · *cleaned value:* `u"active"`

    *Black list* item status qualifier. One of: `"active"` (item currently in the list), `"delisted"` (item removed from the list), `"expired"` (item expired, so treated as removed by the n6 system) or `"replaced"` (e.g.: IP address changed for the same URL).

- fields related to *aggregated (high frequency)* events
  - `count`:
    - *in params:* N/A
    - *in result:* **optional**
    - *field class:* `IntegerField`
    - *specific field constructor arguments:* `min_value=0, max_value=32767`
    - *result cleaning examples:*
      - *example synonymous raw values:* `42` or `42.0` or `"42"`
      - *cleaned value:* `42`

    Number of events represented by the returned incident data record. It must be a positive integer number not greater than 32767.
  - `until`:
    - *in params:* N/A
    - *in result:* **optional**
    - *field class:* `DateTimeField`
    - *result cleaning examples:*
      - *example synonymous raw values:*
      - `"2014-11-05T23:13:00.000000"` or
      - `"2014-11-06 01:13+02:00"` or
      - `datetime.datetime(2014, 11, 5, 23, 13, 0)` or
      - `datetime.datetime(2014, 11, 6, 1, 13, 0, 0, <tzinfo with UTC offset 2h>)`
      - *cleaned value:* `datetime.datetime(2014, 11, 5, 23, 13, 0)`

    The occurrence time of the *latest* [newest] aggregated event represented by the returned incident data record (*note:* `time` is the occurrence time of the *first* [oldest] aggregated event). Value cleaning includes conversion to UTC time.
- the rest of the standard *n6* fields:
  - `action`:
    - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    - *in result:* **optional**
    - *field class:* `UnicodeLimitedField`
    - *specific field constructor arguments:* `max_length=32`
    - *param/result cleaning example:*

        · *raw value:* `"Some Text"`
        · *cleaned value:* `u"Some Text"`

  Action taken by malware (e.g. `"redirect"`, `"screen grab"`...). Maximum length: 32 characters (after cleaning).

- `adip`:
  * *in params:* N/A
  * *in result:* **optional**
  * *field class:* `AnonymizedIPv4Field`
  * *result cleaning example:*
    · *raw value:* `"x.X.234.168"`
    · *cleaned value:* `u"x.x.234.168"`

  Anonymized destination IPv4 address: in quad-dotted decimal notation, with one or more segments replaced with `"x"`, for example: `"x.168.0.1"` or `"x.x.x.1"` (*note:* at least the leftmost segment must be replaced with `"x"`).

- `dip`:
  * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
  * *in result:* **optional**
  * *field class:* `IPv4Field`
  * *param/result cleaning example:*
    · *raw value:* `"123.10.234.168"`
    · *cleaned value:* `u"123.10.234.168"`

  Destination IPv4 address (for sinkhole, honeypot etc.; does not apply to malicious web sites) in quad-dotted decimal notation.

- `dport`:
  * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
  * *in result:* **optional**
  * *field class:* `PortField`
  * *param cleaning example:*
    · *raw value:* `"80"`
    · *cleaned value:* `80`
  * *result cleaning examples:*
    · *example synonymous raw values:* `80` or `80.0` or `u"80"`
    · *cleaned value:* `80`

  TCP/UDP destination port (non-negative integer number, less than 65536).

- `email`
  * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
  * *in result:* **optional**

    * *field class:* `EmailSimplifiedField`
    * *param/result cleaning example:*
        · *raw value:* `"Foo@example.com"`
        · *cleaned value:* `u"Foo@example.com"`

  E-mail address associated with the threat (e.g. source of spam, victim of a data leak).

- `fqdn`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* **optional**
    * *field class:* `DomainNameField`
    * *param/result cleaning examples:*
        · *example synonymous raw values:*
        · `u"WWW.ŁÓDKA.ORG.EXAMPLE"` or
        · `"WWW.\xc5\x81\xc3\x93DKA.ORG.EXAMPLE"` or
        · `u"wwW.łódka.org.Example"` or
        · `"www.\xc5\x82\xc3\xb3dka.org.Example"` or
        · `u"www.xn--dka-fna80b.org.example"` or
        · `"www.xn--dka-fna80b.example.org"`
        · *cleaned value:* `u"www.xn--dka-fna80b.example.org"`

  Fully qualified domain name related to the queried/returned incidents (e.g., for malicious web sites: from the site's URL; for sinkhole/scanning: the domain used for communication). Maximum length: 255 characters (after cleaning).

  **Note:** During cleaning, the `IDNA` encoding is applied (see: https://docs.python.org/2.7/library/codecs.html#module-encodings.idna and http://en.wikipedia.org/wiki/Internationalized_domain_name; see also the above examples), then all remaining upper-case letters are converted to lower-case.

- `fqdn.sub`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* N/A
    * *field class:* `DomainNameSubstringField`
    * *param cleaning example:*
        · *raw value:* `"mple.c"`
        · *cleaned value:* `u"mple.c"`

  Substring of fully qualified domain names related to the queried incidents. Maximum length: 255 characters (after cleaning).

  **See also:**

  The above *fqdn* description.

- `iban`

    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* **optional**
    * *field class:* `IBANSimplifiedField`
    * *param/result cleaning example:*
        · *raw value:* `"gB82weST12345698765432"`
        · *cleaned value:* `u"GB82WEST12345698765432"`

  International Bank Account Number associated with fraudulent activity.

- `injects`:
    * *in params:* N/A
    * *in result:* **optional**
    * *field class:* `ListOfDictsField`

  List of dictionaries containing data that describe a set of injects performed by banking trojans when a user loads a targeted website. (Exact structure of the dictionaries is dependent on malware family and not specified at this time.)

- `md5`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* **optional**
    * *field class:* `MD5Field`
    * *param/result cleaning example:*
        · *raw value:* `"b555773768bc1a672947d7f41f9c247f"`
        · *cleaned value:* `u"b555773768bc1a672947d7f41f9c247f"`

  MD5 hash of the binary file related to the (queried/returned) incident. In the form of a string of 32 hexadecimal digits.

- `modified`
    * *in params:* N/A
    * *in result:* **optional**
    * *field class:* `DateTimeField`
    * *result cleaning examples:*
        · *example synonymous raw values:*
        · `"2014-11-05T23:13:00.000000"` or
        · `"2014-11-06 01:13+02:00"` or
        · `datetime.datetime(2014, 11, 5, 23, 13, 0)` or
        · `datetime.datetime(2014, 11, 6, 1, 13, 0, 0, <tzinfo with UTC offset 2h>)`
        · *cleaned value:* `datetime.datetime(2014, 11, 5, 23, 13, 0)`

  The time when the incident data was *made available through the API or modified.* Value cleaning includes conversion to UTC time.

- `modified.min`:

    - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`, marked as **single_param** in both
    - *in result:* N/A
    - *field class:* `DateTimeField`
    - *param cleaning examples:*
        - *example synonymous raw values:*
        - `"2014-11-06T01:13+02:00"` or
        - `u"2014-11-05 23:13:00.000000"`
        - *cleaned value:* `datetime.datetime(2014, 11, 5, 23, 13, 0)`

  The *earliest* time the queried incidents were *made available through the API or modified* at. Value cleaning includes conversion to UTC time.

- `modified.max`:
    - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`, marked as **single_param** in both
    - *in result:* N/A
    - *field class:* `DateTimeField`
    - *param cleaning examples:*
        - *example synonymous raw values:*
        - `u"2014-11-06T01:13+02:00"` or
        - `"2014-11-05 23:13:00.000000"`
        - *cleaned value:* `datetime.datetime(2014, 11, 5, 23, 13, 0)`

  The *latest* time the queried incidents were *made available through the API or modified* at. Value cleaning includes conversion to UTC time.

- `modified.until`:
    - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`, marked as **single_param** in both
    - *in result:* N/A
    - *field class:* `DateTimeField`
    - *param cleaning examples:*
        - *example synonymous raw values:*
        - `u"2014-11-06T01:13+02:00"` or
        - `"2014-11-05 23:13:00.000000"`
        - *cleaned value:* `datetime.datetime(2014, 11, 5, 23, 13, 0)`

  The time the queried incidents were *made available through the API or modified* before (i.e., exclusive; a handy replacement for `modified.max` in some cases). Value cleaning includes conversion to UTC time.

- `name`:
    - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`

    * *in result:* **optional**
    * *field class:* `UnicodeLimitedField`
    * *specific field constructor arguments:* `max_length=255`
    * *param/result cleaning example:*
        · *raw value:* `"LoremIpsuM"`
        · *cleaned value:* `u"LoremIpsuM"`

  Threat's exact name, such as `"virut"`, `"Potential SSH Scan"` or any other... Maximum length: 255 characters (after cleaning).

- `origin`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* **optional**
    * *field class:* `UnicodeEnumField`
    * *specific field constructor arguments:* `enum_values=n6sdk.data_spec.ORIGIN_ENUMS`
    * *param/result cleaning example:*
        · *raw value:* `"honeypot"`
        · *cleaned value:* `u"honeypot"`

  Incident origin label (some examples: `"p2p-crawler"`, `"sinkhole"`, `"honeypot"`...).

- `phone`
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* **optional**
    * *field class:* `UnicodeLimitedField`
    * *specific field constructor arguments:* `max_length=20`

  Telephone number (national or international). Maximum length: 20 characters (after cleaning).

- `proto`:
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* **optional**
    * *field class:* `UnicodeEnumField`
    * *specific field constructor arguments:* `enum_values=n6sdk.data_spec.PROTO_ENUMS`
    * *param/result cleaning example:*
        · *raw value:* `"tcp"`
        · *cleaned value:* `u"tcp"`

  Layer #4 protocol label – one of: `"tcp"`, `"udp"`, `"icmp"`.

- `registrar`
    * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    * *in result:* **optional**

    - *field class:* `UnicodeLimitedField`
    - *specific field constructor arguments:* `max_length=100`

  Name of the domain registrar. Maximum length: 100 characters (after cleaning).

- `sha1`:
    - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    - *in result:* **optional**
    - *field class:* `SHA1Field`
    - *param/result cleaning example:*
        - *raw value:* `u"7362d67c4f32ba5cd9096dcefc81b28ca04465b1"`
        - *cleaned value:* `u"7362d67c4f32ba5cd9096dcefc81b28ca04465b1"`

  SHA-1 hash of the binary file related to the (queried/returned) incident. In the form of a string of 40 hexadecimal digits.

- `sport`:
    - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    - *in result:* **optional**
    - *field class:* `PortField`
    - *param cleaning example:*
        - *raw value:* `u"80"`
        - *cleaned value:* `80`
    - *result cleaning examples:*
        - *example synonymous raw values:* `80` or `80.0` or `"80"`
        - *cleaned value:* `80`

  TCP/UDP source port (non-negative integer number, less than 65536).

- `target`:
    - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    - *in result:* **optional**
    - *field class:* `UnicodeLimitedField`
    - *specific field constructor arguments:* `max_length=100`
    - *param/result cleaning example:*
        - *raw value:* `"LoremIpsuM"`
        - *cleaned value:* `u"LoremIpsuM"`

  Name of phishing target (organization, brand etc.). Maximum length: 100 characters (after cleaning).

- `url`:
    - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
    - *in result:* **optional**
    - *field class:* `URLField`

  * *param/result cleaning examples:*
    - *example synonymous raw values:*
    - `"ftp://example.com/non-utf8-\xdd"` or
    - `u"ftp://example.com/non-utf8-\udcdd"` or
    - `"ftp://example.com/non-utf8-\xed\xb3\x9d"`
    - *cleaned value:* `u"ftp://example.com/non-utf8-\udcdd"`

  URL related to the queried/returned incidents. Maximum length: 2048 characters (after cleaning).

  **Note:** Cleaning involves decoding byte strings using the `surrogateescape` error handler backported from Python 3.x (see: `n6sdk.encoding_helpers.provide_surrogateescape()`).

- `url.sub`:
  * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
  * *in result:* N/A
  * *field class:* `URLSubstringField`
  * *param cleaning example:*
    - *raw value:* `"/example.c"`
    - *cleaned value:* `u"/example.c"`

  Substring of URLs related to the queried incidents. Maximum length: 2048 characters (after cleaning).

  **See also:**

  The above *url* description.

- `url_pattern`
  * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
  * *in result:* **optional**
  * *field class:* `UnicodeLimitedField`
  * *specific field constructor arguments:* `max_length=255, disallow_empty=True`

  Wildcard pattern or regular expression triggering injects used by banking trojans. Maximum length: 255 characters (after cleaning).

- `username`
  * *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
  * *in result:* **optional**
  * *field class:* `UnicodeLimitedField`
  * *specific field constructor arguments:* `max_length=64`

  Local identifier (login) of the affected user. Maximum length: 64 characters (after cleaning).

- `x509fp_sha1`

  - *in params:* **optional** in `AllSearchableDataSpec`, `None` in `DataSpec`
  - *in result:* **optional**
  - *field class:* `SHA1Field`
  - *param/result cleaning example:*
    - *raw value:* `u"7362d67c4f32ba5cd9096dcefc81b28ca04465b1"`
    - *cleaned value:* `u"7362d67c4f32ba5cd9096dcefc81b28ca04465b1"`

SHA-1 fingerprint of an SSL certificate. In the form of a string of 40 hexadecimal digits.

**Note:** **Generally**, byte strings (if any), when converted to Unicode strings, are – by default – decoded using the `utf-8` encoding.

### Adding, modifying, replacing and getting rid of particular fields...

As you already now, typically you create your own data specification class by subclassing `n6sdk.data_spec.DataSpec` or, alternatively, `n6sdk.data_spec.AllSearchableDataSpec`.

For variety's sake, this time we will subclass `AllSearchableDataSpec` (it has all relevant fields marked as legal query parameters).

Let us prepare a temporary module for our experiments:

```
$ cd <the workbench directory>/Using_N6SDK/using_n6sdk
$ touch experimental_data_spec.py
```

Then, we can open the newly created file (`experimental_data_spec.py`) with our favorite text editor and place the following code in it:

```
from n6sdk.data_spec import AllSearchableDataSpec
from n6sdk.data_spec.fields import UnicodeEnumField

class ExperimentalDataSpec(AllSearchableDataSpec):

    weekday = UnicodeEnumField(
        in_result='optional',
        enum_values=(
            'Monday', 'Tuesday', 'Wednesday', 'Thursday',
            'Friday', 'Saturday', 'Sunday'),
        ),
    )
```

We just made a new *data specification class* – very similar to `AllSearchableDataSpec` but with one additional field specification: `weekday`.

We could also modify (extend) within our subclass some of the field specifications inherited from `AllSearchableDataSpec`. For example:

```
from n6sdk.data_spec import (
    AllSearchableDataSpec,
    Ext,
)

class ExperimentalDataSpec(AllSearchableDataSpec):
    # ...

    id = Ext(
        # here: changing the `max_length` property
        # of the `id` field -- from 64 to 32
        max_length=32,
    )
    time = Ext(
        # here: enabling bare `time` as a query parameter
        # (in AllSearchableDataSpec, by default, the `time.min`,
        # `time.max`, `time.until` query params are enabled but
        # bare `time` is not)
        in_params='optional',

        # here: making `time.min` a required query parameter
        # (*required* -- that is: a client *must* specify it
        # or they will get HTTP-400)
        extra_params=Ext(
            min=Ext(in_params='required'),
        ),
    )
```

Please note how `n6sdk.data_spec.Ext` is used above to extend existing (inherited) field specifications (see also: the *Your first data specification class* section).

It is also possible to replace existing (inherited) field specifications with completely new definitions...

```
# ...
from n6sdk.data_spec.fields import MD5Field
# ...

class ExperimentalDataSpec(AllSearchableDataSpec):
    # ...
    id = MD5Field(
        in_params='optional',
        in_result='required',
    )
    # ...
```

...as well as to remove (mask) them:

```
# ...
class ExperimentalDataSpec(AllSearchableDataSpec):
    # ...
    count = None
```

You can also extend the `clean_param_dict()` and/or `clean_result_dict()` method:

```
# ...

def _is_april_fools_day():
    now = datetime.datetime.utcnow()
    return now.month == 4 and now.day == 1


class ExperimentalDataSpec(AllSearchableDataSpec):

    def clean_param_dict(self, params, ignored_keys=(), **kwargs):
        if _is_april_fools_day():
            ignored_keys = set(ignored_keys) | {'joke'}
        return super(ExperimentalDataSpec, self).clean_param_dict(
            params,
            ignored_keys=ignored_keys,
            **kwargs)

    def clean_result_dict(self, result, **kwargs):
        if _is_april_fools_day():
            result['time'] = '1810-03-01T13:13'
        return super(ExperimentalDataSpec, self).clean_result_dict(
            result,
            **kwargs)
```

**Note:** Manipulating the optional keyword-only arguments (*ignored_keys*, *forbidden_keys*, *extra_required_keys*, *discarded_keys* – see above: *The data specification's cleaning methods*) of these methods can be useful, for example, when you need to implement some authentication-driven data anonymization or param/result-key-focused access rules (however, in such a case you may also need to add some additional keyword-only arguments to the signatures of these methods, e.g. *auth_data*; then you will also need to extend the `get_clean_param_dict_kwargs()` and/or `get_clean_result_dict_kwargs()` methods of your custom subclass of `DefaultStreamViewBase`; generally that matter is beyond the scope of this tutorial).

### Standard *n6sdk* field classes

The following list briefly describes all field classes defined in the `n6sdk.data_spec.fields` module:

- `Field`:

  The top-level base class for field specifications.

- `DateTimeField`:

  - *raw (uncleaned) result value type:* `str`/`unicode` or `datetime.datetime`
  - *cleaned value type:* `datetime.datetime`
  - *example cleaned value:* `datetime.datetime(2014, 11, 6, 13, 30, 1)`

  For date-and-time (timestamp) values, automatically normalized to UTC.

- `UnicodeField`:

  - *base classes:* `Field`
  - *most useful constructor arguments or subclass attributes:*
    - **encoding** (default: `"utf-8"`)
    - **decode_error_handling** (default: `"strict"`)
    - **disallow_empty** (default: `True`)
  - *raw (uncleaned) result value type:* `str` or `unicode`
  - *cleaned value type:* `unicode`
  - *example cleaned value:* `u"Some text value.  Zażółć gęślą jaźń."`

  For arbitrary text data.

- `HexDigestField`:

   - *base classes:* `UnicodeField`
   - **obligatory** *constructor arguments or subclass attributes:*
     * **num_of_characters** (exact number of characters)
     * **hash_algo_descr** (hash algorithm label, such as `"MD5"` or `"SHA256"`...)
   - *raw (uncleaned) result value type:* `str` or `unicode`
   - *cleaned value type:* `unicode`

  For hexadecimal digests (hashes), such as *MD5*, *SHA256* or any other...

- `MD5Field`:
   - *base classes:* `HexDigestField`
   - *raw (uncleaned) result value type:* `str` or `unicode`
   - *cleaned value type:* `unicode`
   - *example cleaned value:* `u"b555773768bc1a672947d7f41f9c247f"`

  For hexadecimal MD5 digests (hashes).

- `SHA1Field`:
   - *base classes:* `HexDigestField`
   - *raw (uncleaned) result value type:* `str` or `unicode`
   - *cleaned value type:* `unicode`
   - *example cleaned value:* `u"7362d67c4f32ba5cd9096dcefc81b28ca04465b1"`

  For hexadecimal SHA-1 digests (hashes).

- `UnicodeEnumField`:
   - *base classes:* `UnicodeField`
   - **obligatory** *constructor arguments or subclass attributes:*
     * **enum_values** (a sequence or set of strings)
   - *raw (uncleaned) result value type:* `str` or `unicode`
   - *cleaned value type:* `unicode`
   - *example cleaned value:* `u"Some selected text value"`

  For text data limited to a finite set of possible values.

- `UnicodeLimitedField`:
   - *base classes:* `UnicodeField`
   - **obligatory** *constructor arguments or subclass attributes:*
     * **max_length** (maximum number of characters)
   - *raw (uncleaned) result value type:* `str` or `unicode`
   - *cleaned value type:* `unicode`
   - *example cleaned value:* `u"Some not-too-long text value"`

  For text data with limited length.

- `UnicodeRegexField`:

  - *base classes:* `UnicodeField`
  - **obligatory** *constructor arguments or subclass attributes:*
    * **regex** (regular expression – as a string or compiled regular expression object)
  - *raw (uncleaned) result value type:* `str` or `unicode`
  - *cleaned value type:* `unicode`
  - *example cleaned value:* `u"Some matching text value"`

  For text data limited by the specified regular expression.

- `SourceField`:
  - *base classes:* `UnicodeLimitedField`, `UnicodeRegexField`
  - *raw (uncleaned) result value type:* `str` or `unicode`
  - *cleaned value type:* `unicode`
  - *example cleaned value:* `u"some-organization.some-type"`

  For dot-separated source specifications, such as `organization.type`.

- `IPv4Field`:
  - *base classes:* `UnicodeLimitedField`, `UnicodeRegexField`
  - *raw (uncleaned) result value type:* `str` or `unicode`
  - *cleaned value type:* `unicode`
  - *example cleaned value:* `u"123.10.234.168"`

  For IPv4 addresses (in decimal dotted-quad notation).

- `IPv6Field`:
  - *base classes:* `UnicodeField`
  - *raw (uncleaned) result value type:* `str` or `unicode`
  - *cleaned value type:* `unicode`
  - *example cleaned values:*
    * **cleaned param value:** `u"abcd:0000:0000:0000:0000:0000:0000:0001` [note the “exploded” form]
    * **cleaned result value:** `u"abcd::1"` [note the “condensed” form]

  For IPv6 addresses (in the standard text representation).

- `AnonymizedIPv4Field`:
  - *base classes:* `UnicodeLimitedField`, `UnicodeRegexField`
  - *raw (uncleaned) result value type:* `str` or `unicode`
  - *cleaned value type:* `unicode`
  - *example cleaned value:* `u"x.10.234.168"`

  For anonymized IPv4 addresses (in decimal dotted-quad notation, with the leftmost octet – and possibly any other octets – replaced with `"x"`).

- `IPv4NetField`:
  - *base classes:* `UnicodeLimitedField`, `UnicodeRegexField`

  - *raw (uncleaned) result value type:* `str`/`unicode` or 2-`tuple`: `(<str/unicode>, <int>)`
  - *cleaned value types:*
    * **of cleaned param values:** 2-`tuple`: `(<unicode>, <int>)`
    * **of cleaned result values:** `unicode`
  - *example cleaned values:*
    * **cleaned param value:** `(u"123.10.0.0", 16)`
    * **cleaned result value:** `u"123.10.0.0/16"`

  For IPv4 network specifications (in CIDR notation).

- `IPv6NetField`:
  - *base classes:* `UnicodeField`
  - *raw (uncleaned) result value type:* `str`/`unicode` or 2-`tuple`: `(<str/unicode>, <int>)`
  - *cleaned value types:*
    * **of cleaned param values:** 2-`tuple`: `(<unicode>, <int>)`
    * **of cleaned result values:** `unicode`
  - *example cleaned values:*
    * **cleaned param value:** `(u"abcd:0000:0000:0000:0000:0000:0000:0001, 128)` [note the "exploded" form of the address part]
    * **cleaned result value:** `(u"abcd::1", 128)` [note the "condensed" form of the address part]

  For IPv6 network specifications (in CIDR notation).

- `CCField`:
  - *base classes:* `UnicodeLimitedField`, `UnicodeRegexField`
  - *raw (uncleaned) result value type:* `str` or `unicode`
  - *cleaned value type:* `unicode`
  - *example cleaned value:* `u"JP"`

  For 2-letter country codes.

- `URLSubstringField`:
  - *base classes:* `UnicodeLimitedField`
  - *most useful constructor arguments or subclass attributes:*
    * **decode_error_handling** (default: `'surrogateescape'`)
  - *raw (uncleaned) result value type:* `str` or `unicode`
  - *cleaned value type:* `unicode`
  - *example cleaned value:* `u"/xyz.example.c"`

  For substrings of URLs.

- `URLField`:
  - *base classes:* `URLSubstringField`
  - *most useful constructor arguments or subclass attributes:*

    * **decode_error_handling** (default: `'surrogateescape'`)
  - *raw (uncleaned) result value type:* `str` or `unicode`
  - *cleaned value type:* `unicode`
  - *example cleaned value:* `u"http://xyz.example.com/path?query=foo#bar"`

  For URLs.

- `DomainNameSubstringField`:
  - *base classes:* `UnicodeLimitedField`
  - *raw (uncleaned) result value type:* `str` or `unicode`
  - *cleaned value type:* `unicode`
  - *example cleaned value:* `u"xample.or"`

  For substrings of domain names, automatically IDNA-encoded and lower-cased.

- `DomainNameField`:
  - *base classes:* `DomainNameSubstringField`, `UnicodeRegexField`
  - *raw (uncleaned) result value type:* `str` or `unicode`
  - *cleaned value type:* `unicode`
  - *example cleaned value:* `u"www.xn--w-uga1v8h.example.org"`

  For domain names, automatically IDNA-encoded and lower-cased.

- `EmailSimplifiedField`:
  - *base classes:* `UnicodeLimitedField`, `UnicodeRegexField`
  - *raw (uncleaned) result value type:* `str` or `unicode`
  - *cleaned value type:* `unicode`
  - *example cleaned value:* `u"Foo@example.com"`

  For e-mail addresses (validation is rather rough).

- `IBANSimplifiedField`:
  - *base classes:* `UnicodeLimitedField`, `UnicodeRegexField`
  - *raw (uncleaned) result value type:* `str` or `unicode`
  - *cleaned value type:* `unicode`
  - *example cleaned value:* `u"GB82WEST12345698765432"`

  For International Bank Account Numbers.

- `IntegerField`:
  - *base classes:* `Field`
  - *most useful constructor arguments or subclass attributes:*
    * **min_value** (*optional* minimum value)
    * **max_value** (*optional* maximum value)
  - *raw (uncleaned) result value type:* `str`/`unicode` or an **integer number** of *any numeric type*
  - *cleaned value type:* `int` or (for bigger numbers) `long`

  - *example cleaned value:* `42`

  For integer numbers (optionally with minimum/maximum limits defined).

- `ASNField`:
  - *base classes:* `IntegerField`
  - *raw (uncleaned) result value type:* `str`/`unicode` or `int`/`long`
  - *cleaned value type:* `int` or (possibly, for bigger numbers) `long`
  - *example cleaned value:* `123456789`

  For autonomous system numbers, such as `12345` or `123456789`, or `12345.65432`.

- `PortField`:
  - *base classes:* `IntegerField`
  - *raw (uncleaned) result value type:* `str`/`unicode` or an **integer number** of *any numeric type*
  - *cleaned value type:* `int`
  - *example cleaned value:* `12345`

  For TCP/UDP port numbers.

- `ResultListFieldMixin`:
  - *base classes:* `Field`
  - *most useful constructor arguments or subclass attributes:*
    - **allow_empty** (default: `False` which means that an empty sequence causes a cleaning error)

  A mix-in class for fields whose result values are supposed to be a *sequence of values* and not single values. Its `clean_result_value()` checks that its argument is a *non-string sequence* (`list` or `tuple`, or any other `collections.Sequence` not being `str` or `unicode`) and performs result cleaning (as defined in a superclass) for *each item* of it.

  **See also:**

  The *ListOfDictsField* description below.

- `DictResultField`:
  - *base classes:* `Field`
  - *most useful constructor arguments or subclass attributes:*
    - **key_to_subfield_factory** (`None` or a dictionary that maps subfield names to field classes or field factory functions)
  - *raw (uncleaned) result value type:* `collections.Mapping`
  - *cleaned value type:* `dict`

  A base class for fields whose result values are supposed to be dictionaries (their structure can be constrained by specifying the *key_to_subfield_factory* property, described above).

  **Note:** This is a result-only field class, i.e. its `clean_param_value()` raises `TypeError`.

  **See also:**

  The *ListOfDictsField* description below.

- `ListOfDictsField`:
  - *base classes:* `ResultListFieldMixin`, `DictResultField`
  - *raw (uncleaned) result value type:* `collections.Sequence` of `collections.Mapping` instances
  - *cleaned value type:* `list` of `dict` instances
  - *example cleaned values:*
    - **cleaned param value:** N/A (`clean_param_value()` raises `TypeError`)
    - **cleaned result value:** `[{u"a": u"b", u"c": 4, u"e": [1, 2, 3]}]`

  For lists of dictionaries containing arbitrary values.

  **See also:**

  The *AddressField* and *ExtendedAddressField* descriptions below.

- `AddressField`:
  - *base classes:* `ListOfDictsField`
  - *raw (uncleaned) result value type:* `collections.Sequence` of `collections.Mapping` instances
  - *cleaned value type:* `list` of `dict` instances
  - *example cleaned values:*
    - **cleaned param value:** N/A (`clean_param_value()` raises `TypeError`)
    - **cleaned result value:** `[{u"ip": u"123.10.234.169", u"cc": u"UA", u"asn": 12345}]`

  For lists of dictionaries – each containing `"ip"` and optionally `"cc"` and/or `"asn"`.

- `DirField`:
  - *base classes:* `UnicodeEnumField`
  - *raw (uncleaned) result value type:* `str` or `unicode`
  - *cleaned value type:* `unicode`
  - *the only possible cleaned values:* `u"src"` or `u"dst"`

  For `dir` values in items cleaned by of `ExtendedAddressField` instances (`dir` marks role of the address in terms of the direction of the network flow in layers 3 or 4).

- `ExtendedAddressField`:
  - *base classes:* `ListOfDictsField`
  - *raw (uncleaned) result value type:* `collections.Sequence` of `collections.Mapping` instances
  - *cleaned value type:* `list` of `dict` instances
  - *example cleaned values:*
    - **cleaned param value:** N/A (`clean_param_value()` raises `TypeError`)
    - **cleaned result value:** `[{u"ipv6": u"abcd::1", u"cc": u"PL", u"asn": 12345, u"dir": u"dst"}]`

For lists of dictionaries – each containing either `"ip"` or `"ipv6"` (but not both), and optionally all or some of: `"cc"`, `"asn"`, `"dir"`, `"rdns"`.

**Note: Generally –**

- constructor arguments, when specified, must be provided as *keyword arguments*;
- "constructor argument or a subclass attribute" means that a certain field property can be specified in two alternative ways: either when creating a field instance (using a keyword argument for the constructor) or when subclassing the field class (using an attribute of the subclass; see below: *Custom field classes*);
- raw (uncleaned) *parameter* value type is *always* `str`/`unicode`;
- all these classes are *cooperative-inheritance*-friendly (i.e., `super()` in subclasses' `clean_param_value()` and `clean_result_value()` will work properly, also with multiple inheritance).

**See also:**

The *Overview of the basic data specification classes* section above.

## Custom field classes

You may want to subclass any of the *n6sdk* field classes (described above in the *Standard n6sdk field classes* section):

- to override class attributes,
- to extend the `clean_param_value()` and/or `clean_result_value()` method.

Please, consider the beggining of our `<the workbench directory>/Using_N6SDK/using_n6sdk/data_spec.py` file:

```
from n6sdk.data_spec import DataSpec, Ext
from n6sdk.data_spec.fields import UnicodeRegexField


class UsingN6sdkDataSpec(DataSpec):

    """
    The data specification class for the `Using_N6SDK` project.
    """

    mac_address = UnicodeRegexField(
        in_params='optional',  # *can* be in query params
        in_result='optional',  # *can* be in result data

        regex=r'^(?:[0-9A-F]{2}(?:[:-]|$)){6}$',
        error_msg_template=u'"{}" is not a valid MAC address',
    )
```

It can be rewritten in a more self-documenting and code-reusability-friendly way:

```
from n6sdk.data_spec import DataSpec, Ext
from n6sdk.data_spec.fields import UnicodeRegexField


class MacAddressField(UnicodeRegexField):
    regex = r'^(?:[0-9A-F]{2}(?:[:-]|$)){6}$'
    error_msg_template = u'"{}" is not a valid MAC address'


class UsingN6sdkDataSpec(DataSpec):

    """
    The data specification class for the `Using_N6SDK` project.
    """

    mac_address = MacAddressField(
        in_params='optional',  # *can* be in query params
        in_result='optional',  # *can* be in result data
    )
```

Another technique – extending the value cleaning methods (see above: *The field's cleaning methods*) – offers more possibilities. For example, we could create an integer number field that accepts parameter values with such suffixes as `"m"` (*meters*), `"kg"` (*kilograms*) and `"s"` (*seconds*), ignoring the suffixes:

```
from n6sdk.data_spec.fields import IntegerField

class SuffixedIntegerField(IntegerField):

    # the `legal_suffixes` class attribute we create here
    # can be overridden with a `legal_suffixes` constructor
    # argument or a `legal_suffixes` subclass attribute
    legal_suffixes = 'm', 'kg', 's'

    def clean_param_value(self, value):
        """
        >>> SuffixedIntegerField().clean_param_value('123 kg')
        123
        """
        value = value.strip()
        for suffix in self.legal_suffixes:
            if value.endswith(suffix):
                value = value[:(-len(suffix))]
                break
        value = super(SuffixedIntegerField,
                      self).clean_param_value(value)
        return value
```

If – in your implementation of `clean_param_value()` or `clean_result_value()` – you need to raise a cleaning error (to signal that a value is invalid and cannot be cleaned) just raise any exception being an instance of standard `Exception` (or of its subclass); it *can* (but *does not have to*) be *`n6sdk.exceptions.FieldValueError`*.

When subclassing *n6sdk* field classes, please do not be afraid to look into the source code of the `n6sdk.data_spec.fields` module.

## 2.4 Implementing the data backend API

### 2.4.1 The interface

The network incident data can be stored in various ways: using text files, in an SQL database, using some distributed storage such as Hadoop etc. Implementation of obtaining data from any of such backends is beyond the scope of this document. What we do concern here is the API the *n6sdk*'s machinery needs to use to get the data.

Therefore, for the purposes of this tutorial, we will assume that our network incident data is stored in the simplest possible way: *in one file in the JSON format*. You will have to replace any implementation details related to this particular way of keeping and querying for data with an implementation appropriate for the data store you use (file reads, SQL queries or whatever is needed for the particular storage backend) – see the next section: *Guidelines for the real implementation*.

First, we will **create the example JSON data file**:

```
$ cat << EOF > /tmp/our-data.json
     [
       {
         "id": "1",
         "address": [
           {
             "ip": "11.22.33.44"
           },
           {
             "asn": 12345,
             "cc": "US",
             "ip": "123.124.125.126"
           }
         ],
         "category": "phish",
         "confidence": "low",
         "mac_address": "00:11:22:33:44:55",
         "restriction": "public",
         "source": "test.first",
         "time": "2015-04-01 10:00:00",
         "url": "http://example.com/?spam=ham"
       },
       {
         "id": "2",
         "adip": "x.2.3.4",
         "category": "server-exploit",
         "confidence": "medium",
         "restriction": "need-to-know",
         "source": "test.first",
         "time": "2015-04-01 23:59:59"
       },
       {
         "id": "3",
         "address": [
           {
             "ip": "11.22.33.44"
           },
           {
             "asn": 87654321,
             "cc": "PL",
             "ip": "111.122.133.144"
           }
         ],
         "category": "server-exploit",
         "confidence": "high",
         "restriction": "public",
         "source": "test.second",
         "time": "2015-04-01 23:59:59",
         "url": "http://example.com/?spam=ham"
```

```
        }
      ]
EOF
```

Then, we need to **open the file** `<the workbench directory>/Using_N6SDK/using_n6sdk/data_backend_api.py` with our favorite text editor and **modify it so that it will contain the following code** (however, it is recommented not to remove the comments and docstrings the file already contains – as they can be valuable hints for future code maintainers):

```
import json

from n6sdk.class_helpers import singleton
from n6sdk.datetime_helpers import parse_iso_datetime_to_utc
from n6sdk.exceptions import AuthorizationError


@singleton
class DataBackendAPI(object):

    def __init__(self, settings):
        ## [...existing docstring + comments...]
        # Implementation for our example JSON-file-based "storage":
        with open(settings['json_data_file_path']) as f:
            self.data = json.load(f)

    ## [...existing comments...]

    def generate_incidents(self, auth_data, params):
        ## [...existing docstring + comments...]
        # This is a naive implementation for our example
        # JSON-file-based "storage" (some efficient database
        # query needs to be performed instead, in case of any
        # real-world implementation...):
        for incident in self.data:
            for key, value_list in params.items():
                if key == 'ip':
                    address_seq = incident.get('address', [])
                    if not any(addr.get(key) in value_list
                               for addr in address_seq):
                        break    # incident does not match the query params
                elif key in ('time.min', 'time.max', 'time.until'):
                    [param_val] = value_list  # must be exactly one value
                    db_val = parse_iso_datetime_to_utc(incident['time'])
                    if not ((key == 'time.min' and db_val >= param_val) or
                            (key == 'time.max' and db_val <= param_val) or
                            (key == 'time.until' and db_val < param_val)):
                        break    # incident does not match the query params
                elif incident.get(key) not in value_list:
                    break        # incident does not match the query params
            else:
                # (the inner for loop has not been broken)
                yield incident   # incident *matches* the query params
```

What is important:

1. The constructor of the class is supposed to be called exactly once per application run. The constructor must take exactly one argument:

   - *settings* – a dictionary containing settings from the `*.ini` file (e.g., `development.ini` or `production.ini`).

2. The class can have one or more data query methods, with arbitrary names (in the above example there is only one: `generate_incidents()`; to learn how URLs are mapped to particular data query method names – see below: *Gluing it together*).

   Each data query method must take two positional arguments:

   - *auth_data* – authentication data, relevant only if you need to implement in your data query methods some kind of authorization based on the authentication data; its type and format depends on the authentication policy you use (see below: *Custom authentication policy*);
   - *params* – a dictionary containing cleaned (validated and normalized with `clean_param_dict()`) client query parameters; the dictionary maps parameter names (strings) to lists of parameter values (see above: *Data specification class*).

3. Each data query method must be a *generator* (see: https://docs.python.org/2/glossary.html#term-generator) or any other callable that returns an *iterator* (see: https://docs.python.org/2/glossary.html#term-iterator). Each of the generated items should be a dictionary containing the data of one network incident (the *n6sdk* machinery will use it as the argument for the `clean_result_dict()` data specification method).

### 2.4.2 Guidelines for the real implementation

Typically, the following activities are performed **in the __init__() method** of the data backend API class:

1. Get the storage backend settings from the *settings* dictionary (apropriate items should have been placed in the `[app:main]` section of the `*.ini` file – see below: *Gluing it together*).
2. Configure the storage backend (for example, create the database connection).

Typically, the following activities are performed **in a data query method** of the data backend API class:

1. If needed: do any authorization checks based on the *auth_data* and *params* arguments; raise *`n6sdk.exceptions.AuthorizationError`* on failure.
2. Translate the contents of the *params* argument to some storage-specific queries. (Obviously, when doing the translation you may need, for example, to map *params* keys to some storage-specific keys...).

   **Note:** If the data specification includes dotted "extra params" (such as `time.min`, `time.max`, `time.until`, `fqdn.sub`, `ip.net` etc.) their semantics should be implemented carefully.

3. If needed: perform a necessary storage-specific maintenance activity (e.g., re-new the database connection).
4. Perform a storage-specific query (or queries).

   Sometimes you may want to limit the number of allowed results – then, raise *`n6sdk.exceptions.TooMuchDataError`* if the limit is exceeded.

5. Translate the results of the storage-specific query (queries) to result dictionaries and *yield* each of these dictionaries (each of them should be a dictionary ready to be passed to the `clean_result_dict()` method of data specification).

   (Obviously, when doing the translation, you may need, for example, to map some storage-specific keys to the result keys accepted by the `clean_result_dict()` method of your data specificaton class...)

   If there are no results – just do not yield any items (the caller will obtain an empty iterator).

In case of an internal error, do not be afraid to raise an exception – any instance of `Exception` (or of its subclass) will be handled automatically by the *n6sdk* machinery: logged (including the traceback) using the `n6sdk.pyramid_commons` logger and transformed into `pyramid.httpexceptions.HTTPServerError` which will break generation of the HTTP response body (note, however, that there will be no *HTTP-500* response – because it is not possible to send such an "error response" when some parts of the body of the "data response" have already been sent out).

It is recommended to decorate your data backend API class with the *`n6sdk.class_helpers.singleton()`* decorator (it ensures that the class is instantiated only once; any attempt to repeat that causes `TypeError`).

## 2.5 Custom authentication policy

A description of the concept of *Pyramid authentication policies* is beyond the scope of this tutorial. Please read the appropriate paragraph and example from the documentation of the *Pyramid* library: http://docs.pylonsproject.org/projects/pyramid/en/1.5-branch/narr/security.html#creating-your-own-authentication-policy (you may also want to search the *Pyramid* documentation for the term `authentication policy`).

The *n6sdk* library requires that the authentication policy class has the additional static (decorated with `staticmethod()`) method `get_auth_data()` that takes exactly one positional argument: a *Pyramid request* object. The method is expected to return a value that is **not** `None` in case of authentication success, and `None` otherwise. Apart from this simple rule there are no constraints what exactly the return value should be – the implementer decides about that. The return value will be available as the `auth_data` attribute of the *Pyramid request* as well as is passed into data backend API methods as the *auth_data* argument.

Typically, the `authenticated_userid()` method implementation makes use of the request's attribute `auth_data` (being return value of `get_auth_data()`), and the `get_auth_data()` implementation makes some use of the request's attribute `unauthenticated_userid` (being return value of the `unauthenticated_userid()` policy method). It is possible because `get_auth_data()` is called (by the *Pyramid* machinery) *after* the `unauthenticated_userid()` method and *before* the `authenticated_userid()` method.

The *n6sdk* library provides `n6sdk.pyramid_commons.BaseAuthenticationPolicy` – an authentication policy base class that makes it easier to implement your own authentication policies. Please consult its source code.

## 2.6 Gluing it together

We can inspect the `__init__.py` file of our application (`<the workbench directory>/Using_N6SDK/using_n6sdk/__init__.py`) with our favorite text editor. It contains a lot of useful comments that suggest how to customize the code – however, if we omitted them, the actual Python code would be:

```python
from n6sdk.pyramid_commons import (
    AnonymousAuthenticationPolicy,
    ConfigHelper,
    HttpResource,
)

from .data_spec import UsingN6sdkDataSpec
from .data_backend_api import DataBackendAPI


# (this is how we map URLs to particular data query methods...)
RESOURCES = [
    HttpResource(
        resource_id='/incidents',
        url_pattern='/incidents.{renderer}',
        renderers=('json', 'sjson'),
        data_spec=UsingN6sdkDataSpec(),
        data_backend_api_method='generate_incidents',
    ),
]


def main(global_config, **settings):
    helper = ConfigHelper(
        settings=settings,
        data_backend_api_class=DataBackendAPI,
        authentication_policy=AnonymousAuthenticationPolicy(),
        resources=RESOURCES,
    )
    return helper.make_wsgi_app()
```

(In the context of descriptions the previous sections contain, this boilerplate code should be rather self-explanatory. If not, please consult the comments in the actual `<the workbench directory>/Using_N6SDK/using_n6sdk/__init__.py` file.)

Now, yet another important step needs to be completed: **customization of the settings** in the `<the workbench directory>/Using_N6SDK/*.ini` files: `development.ini` and `production.ini` – to match the environment, database configuration (if any) etc.

**Warning:** You should **not** place any sensitive settings (such as real database passwords) in these files – as they are still just configuration templates (which your will want, for example, to add to your version control system) and **not** real configuration files for production.
**See also:**
The *Installation for production (using Apache server)* section.

In case of our naive JSON-file-based data backend implementation (see above: *The interface*) we need to **add the following line in the** `[app:main]` **section of each of the two settings files** (`development.ini` and `production.ini`):

```
json_data_file_path = /tmp/our-data.json
```

Finally, let us run the application (still in the development environment):

```
$ cd <the workbench directory>
$ source dev-venv/bin/activate   # ensuring the virtualenv is active
$ pserve Using_N6SDK/development.ini
```

Our application should be being served now. Try visiting the following URLs (with any web browser or, for example, with the `wget` command-line tool):

- `http://127.0.0.1:6543/incidents.json`
- `http://127.0.0.1:6543/incidents.json?ip=11.22.33.44`
- `http://127.0.0.1:6543/incidents.json?ip=11.22.33.44&time.min=2015-04-01T23:00:00`
- `http://127.0.0.1:6543/incidents.json?category=phish`
- `http://127.0.0.1:6543/incidents.json?category=server-exploit`
- `http://127.0.0.1:6543/incidents.json?category=server-exploit&ip=11.22.33.44`
- `http://127.0.0.1:6543/incidents.json?category=bots&category=server-exploit`

- `http://127.0.0.1:6543/incidents.json?category=bots,dos-attacker,phish,server-exploit`
- `http://127.0.0.1:6543/incidents.sjson?mac_address=00:11:22:33:44:55`
- `http://127.0.0.1:6543/incidents.sjson?source=test.first`
- `http://127.0.0.1:6543/incidents.sjson?source=test.second`
- `http://127.0.0.1:6543/incidents.sjson?source=some.non-existent`
- `http://127.0.0.1:6543/incidents.sjson?source=some.non-existent&source=test.second`
- `http://127.0.0.1:6543/incidents.sjson?time.min=2015-04-01T23:00`
- `http://127.0.0.1:6543/incidents.sjson?time.max=2015-04-01T23:59:59&confidence=medium,l`
- `http://127.0.0.1:6543/incidents.sjson?time.until=2015-04-01T23:59:59`

...as well as those causing (expected) errors:

- `http://127.0.0.1:6543/incidents`
- `http://127.0.0.1:6543/incidents.jsonnn`
- `http://127.0.0.1:6543/incidents.json?some-illegal-key=1&another-one=foo`
- `http://127.0.0.1:6543/incidents.json?category=wrong`
- `http://127.0.0.1:6543/incidents.json?category=bots,wrong`
- `http://127.0.0.1:6543/incidents.json?category=bots&category=wrong`
- `http://127.0.0.1:6543/incidents.json?ip=11.22.33.44.55`
- `http://127.0.0.1:6543/incidents.sjson?mac_address=00:11:123456:33:44:55`
- `http://127.0.0.1:6543/incidents.sjson?time.min=2015-04-01T23:00,2015-04-01T23:30`
- `http://127.0.0.1:6543/incidents.sjson?time.min=2015-04-01T23:00&time.min=2015-04-01T23`
- `http://127.0.0.1:6543/incidents.sjson?time.min=blablabla`
- `http://127.0.0.1:6543/incidents.sjson?time.max=blablabla&ip=11.22.33.444`
- `http://127.0.0.1:6543/incidents.sjson?time.until=2015-04-01T23:59:59&ip=11.22.33.444`

Now, it can be a good idea to try the *helper script for automatized basic API testing*.

## 2.7 Installation for production (using Apache server)

**Warning:** This section is intended to be a brief and rough explanation how you can glue relevant configuration stuff together. It is **not** intended to be used as a step-by-step recipe for a secure production configuration. **The final configuration (including but not limited to file access permissions) should always be carefully reviewed by an experienced system administrator – BEFORE it is deployed on a publicly available server**.

Prerequisites are similar to those concerning the development environment, listed near the beginning of this tutorial. The same applies to the way of obtaining the source code of *n6sdk*.

**See also:**

The sections: *Prerequisites* and ref:*obtaining_source_code*.

The Debian GNU/Linux operating system in the version 7.9 or newer is recommended to follow the guides presented below. Additional prerequisite is that the Apache2 HTTP server is installed and configured together with `mod_wsgi` (the `apache2` and `libapache2-mod-wsgi` Debian packages).

First, we will create a directory structure and a *virtualenv* for our server, e.g. under `/opt`:

```
$ sudo mkdir /opt/myn6-srv
$ cd /opt/myn6-srv
$ sudo virtualenv prod-venv
$ sudo chown -R $(echo $USER) prod-venv
$ source prod-venv/bin/activate
```

Then, let us install the necessary packages:

```
$ cd <the workbench directory>/n6sdk
$ python setup.py install
$ cd <the workbench directory>/Using_N6SDK
$ python setup.py install
```

(Of course, `<the workbench directory>/n6sdk` needs to be replaced with the actual name (absolute path) of the directory containing the source code of the *n6sdk* library; and `<the workbench directory>/Using_N6SDK` needs to be replaced with the actual name (absolute path) of the directory containing the source code of our *n6sdk*-based project.)

Now, we will copy the template of the configuration file for production:

```
$ cd /opt/myn6-srv
$ sudo cp <the workbench directory>/Using_N6SDK/production.ini ./
```

For security sake, let us restrict access to the `production.ini` file before we will place any real passwords and other sensitive settings in it:

```
$ sudo chown root ./production.ini
$ sudo chmod 600 ./production.ini
```

We need to ensure that the Apache's user group has read-only access to the file. On Debian GNU/Linux it can be done by executing:

```
$ sudo chgrp www-data ./production.ini
$ sudo chmod g+r ./production.ini
```

You may want to customize the settings that the file contains, especially to match your production environment, database configuration etc. Just edit the `/opt/myn6-srv/production.ini` file.

Then, we will create the WSGI script:

```
$ cat << EOF > prod-venv/myn6-app.wsgi
from pyramid.paster import get_app, setup_logging
ini_path = '/opt/myn6-srv/production.ini'
setup_logging(ini_path)
application = get_app(ini_path, 'main')
EOF
```

...and provide the directory for the *egg cache*:

```
$ sudo mkdir /opt/myn6-srv/.python-eggs
```

We need to ensure that the Apache's user has write access to it. On Debian GNU/Linux it can be done by executing:

```
$ sudo chown www-data /opt/myn6-srv/.python-eggs
```

Now, we need to adjust the Apache configuration. On Debian GNU/Linux it can be done by executing:

```
$ cat << EOF > prod-venv/myn6.apache
<VirtualHost *:80>
  # Only one Python sub-interpreter should be used
  # (multiple ones do not cooperate well with C extensions).
  WSGIApplicationGroup %{GLOBAL}

  # Remove the following line if you use native Apache authorization.
  WSGIPassAuthorization On

  WSGIDaemonProcess myn6_srv \\
    python-path=/opt/myn6-srv/prod-venv/lib/python2.7/site-packages \\
    python-eggs=/opt/myn6-srv/.python-eggs
  WSGIScriptAlias /myn6 /opt/myn6-srv/prod-venv/myn6-app.wsgi

  <Directory /opt/myn6-srv/prod-venv>
    WSGIProcessGroup myn6_srv
    Order allow,deny
    Allow from all
  </Directory>

  # Logging of errors and other events:
  ErrorLog \${APACHE_LOG_DIR}/error.log
  # Possible values for the LogLevel directive include:
  # debug, info, notice, warn, error, crit, alert, emerg.
  LogLevel warn

  # Logging of client requests:
  CustomLog \${APACHE_LOG_DIR}/access.log combined

  # It is recommended to uncomment and adjust the following line.
  #ServerAdmin webmaster@yourserver.example.com
</VirtualHost>
EOF
$ sudo chmod 640 prod-venv/myn6.apache
$ sudo chown root:root prod-venv/myn6.apache
$ sudo mv prod-venv/myn6.apache /etc/apache2/sites-available/myn6
$ cd /etc/apache2/sites-enabled
$ sudo ln -s ../sites-available/myn6 001-myn6
```

You may want or need to adjust the contents of the newly created file (`/etc/apache2/sites-available/myn6`) – especially regarding the following directives (see the comments accompanying them in the file):

- `WSGIPassAuthorization`,
- `ErrorLog` and `LogLevel`,
- `CustomLog`,
- `ServerAdmin`.

**See also:**

- About general configuration of Apache: http://httpd.apache.org/docs/2.2/configuring.html

- About `modwsgi`-specific configuration: http://code.google.com/p/modwsgi/wiki/ConfigurationGuidelines

If we have the default Apache configuration on Debian, we need to disable the default site by removing the symbolic link:

```
$ rm 000-default
```

Finally, let us restart the Apache daemon. On Debian GNU/Linux it can be done by executing:

```
$ sudo service apache2 restart
```

Our application should be being served now. Try visiting the following URL (with any web browser or, for example, with the `wget` command-line tool):

`http://<your apache server address>/myn6/incidents.json`

(Of course, `<your apache server address>` needs to be replaced with the actual host address of your Apache server, for example `127.0.0.1` or `localhost`.)

CHAPTER 3

# `n6sdk_api_test`: API testing tool

## 3.1 Overview

The `n6sdk_api_test` script is a simple tool to perform basic validation of your *n6sdk*-based REST API.

In the current version of the tool, *validation* consist of the following steps:

1. inferring basic information about the tested API + testing essential compliance with the general *n6* specification;
2. testing a query containing two (randomly selected) *legal* parameters;
3. testing queries containing some *illegal* parameters;
4. testing queries containing (single) *legal* parameters;
5. testing queries containing one parameter, using various values of it.

API testing tool provides feedback printed in a plain text format. The report is structured in sections for every test case category. The output is more informative when the `--verbose` option is used.

The test data set is prepared automatically; how is it done depends on the query parameters placed in the tool's config file. Hence, it is the user's responsibility to select the base URL containing such query parameters that the response will reflect the internal structure of data records from the database as well as possible. In other words, the user is responsible for selecting a query that allows to pick out the most diverse data sample.

Because of simplicity of the `n6sdk_api_test` tool – and considering that the script employs a lot of randomization – it may be worth running the tool more than once. Experimenting with different settings in the `[constant_params]` section of the tool's config can also be a good idea.

## 3.2 Installation

The script is automatically installed in the appropriate place when you install *n6sdk* by running `python setup.py ...` (see: *Installing the necessary stuff* or *Installation for production (using Apache server)*).

## 3.3 Configuration and usage

To use `n6sdk_api_test` follow these steps:

1. Generate the config file base:

```
$ n6sdk_api_test --generate-config > config.ini
```

2. Adjust the generated `config.ini` file:
   - provide the base URL of the tested API resource;
   - specify mandatory query parameters (`time.min` etc.; see the comment in the generated config file);
   - specify SSL certificate/key paths in case of SSL-based method of authentication, or username/password in case of basic HTTP authentication (if required by the tested API).
3. Run the script, e.g.:

```
$ n6sdk_api_test -c config.ini
```

Note that the resultant report is printed to the standard output so one can easily write it to a file:

```
$ n6sdk_api_test -c config.ini > report.txt
```

To see the available options:

```
$ n6sdk_api_test -h
```

# Library Reference

## 4.1 Core modules

### 4.1.1 n6sdk.data_spec

### 4.1.2 n6sdk.data_spec.fields

### 4.1.3 n6sdk.exceptions

**class** `n6sdk.exceptions.`**`_ErrorWithPublicMessageMixin`**(**args*, ***kwargs*)

Bases: `object`

A mix-in class that provides the *`public_message`* property.

The value of this property is a unicode string. It is taken either from the *public_message* constructor keyword argument (which should be a unicode string or an UTF-8-decodable str string) or – if the argument was not specified – from the value of the *`default_public_message`* attribute (which should also be a unicode string or an UTF-8-decodable str string).

The public message should be a complete sentence (or several sentences): first word capitalized (if not being an identifier that begins with a lower case letter) + the period at the end.

**Warning:** Generally, the message is intended to be presented to clients. **Ensure that you do not disclose any sensitive details in the message.**

**See also:**

The documentation of the public exception classes provided by this module.

The `str` and `unicode` conversions that are provided by the class use the value of *`public_message`*:

```
>>> class SomeError(_ErrorWithPublicMessageMixin, Exception):
...     pass
...
>>> str(SomeError('a', 'b'))  # using attribute default_public_message
'Internal error.'
>>> str(SomeError('a', 'b', public_message='Sp\xc4\x85m.'))
'Sp\xc4\x85m.'
>>> str(SomeError('a', 'b', public_message=u'Sp\u0105m.'))
'Sp\xc4\x85m.'
>>> unicode(SomeError('a', 'b'))  # using attribute default_public_message
```

```
u'Internal error.'
>>> unicode(SomeError('a', 'b', public_message='Sp\xc4\x85m.'))
u'Sp\u0105m.'
>>> unicode(SomeError('a', 'b', public_message=u'Sp\u0105m.'))
u'Sp\u0105m.'
```

The `repr()` conversion results in a programmer-readable representation (containing the class name, `repr()`-formatted constructor arguments and the *public_message* property):

```
>>> SomeError('a', 'b')    # using class's default_public_message
<SomeError: args=('a', 'b'); public_message=u'Internal error.'>
>>> SomeError('a', 'b', public_message='Spam.')
<SomeError: args=('a', 'b'); public_message=u'Spam.'>
```

**`default_public_message`** **= u'Internal error.'**
(overridable in subclasses)

**`public_message`**
The aforementioned property.

**class** `n6sdk.exceptions.`**`_KeyCleaningErrorMixin`**(*illegal_keys*, *missing_keys*)
Bases: `object`

Mix-in for *key cleaning*-related exception classes.

Each instance of such a class:

- should be initialized with two (positional or keyword) arguments: *illegal_keys* and *missing_keys* that should be sets of – respectively – illegal or missing keys (each key being a string);
- exposes these arguments as the `illegal_keys` and `missing_keys` attributes (for possible later inspection).

**class** `n6sdk.exceptions.`**`_ValueCleaningErrorMixin`**(*error_info_seq*)
Bases: `object`

Mix-in for *value cleaning*-related exception classes.

Each instance of such a class:

- should be initialized with one argument being a list of (<*key*>, <*offending value or list of offending values*>, <*actual exception*>) tuples – where <*actual exception*> is the exception instance that caused the error (e.g. a `ValueError` or an instance of some *`_ErrorWithPublicMessageMixin`* subclass);
- exposes that argument as the `error_info_seq` attribute (for possible later inspection).

**exception** `n6sdk.exceptions.`**`FieldValueError`**(**args*, ***kwargs*)
Bases: *`n6sdk.exceptions._ErrorWithPublicMessageMixin`*, `exceptions.ValueError`

Intended to be raised in `clean_param_value()` and `clean_result_value()` methods of `n6sdk.data_spec.fields.Field` subclasses.

When using it in a `clean_param_value()`'s implementation it is recommended (though not required) to insantiate the exception specifying the *public_message* keyword argument.

Typically, this exception (as any other `Exception` subclass/instance raised in a field's `clean_*_value()` method) is caught by the *n6sdk* machinery – then, appropriately, *`ParamValueCleaningError`* (with `public_message` including `public_message` of this exception – see: the *`ParamValueCleaningError`* documentation) or *`ResultValueCleaningError`* (with `public message` being just the default and safe `"Internal error."`) is raised.

**See also:**

*`_ErrorWithPublicMessageMixin`* as well as the *`ParamValueCleaningError`* and *`ResultValueCleaningError`* documentation.

**exception** `n6sdk.exceptions.`**`FieldValueTooLongError`**`(`**args*, ***kwargs*`)`

Bases: *`n6sdk.exceptions.FieldValueError`*

Intended to be raised when the length of the given value is too big.

Instances *must* be initialized with the following keyword-only arguments:

- *field* (`n6sdk.data_spec.fields.Field` instance): the field whose method raised the exception;
- *checked_value*: the value which caused the exception (possibly already partially processed by methods of *field*);
- *max_length*: the length limit that was exceeded (what caused the exception).

They become attributes of the exception instance – respectively: `field`, `checked_value`, `max_length`.

```
>>> exc = FieldValueTooLongError(
...     field='sth', checked_value=['foo'], max_length=42)
>>> exc.field
'sth'
>>> exc.checked_value
['foo']
>>> exc.max_length
42
```

```
>>> FieldValueTooLongError(
...     checked_value=['foo'], max_length=42)
Traceback (most recent call last):
  ...
TypeError: __init__() needs keyword-only argument field
```

```
>>> FieldValueTooLongError(
...     field='sth', max_length=42)
Traceback (most recent call last):
  ...
TypeError: __init__() needs keyword-only argument checked_value
```

```
>>> FieldValueTooLongError(
...     field='sth', checked_value=['foo'])
Traceback (most recent call last):
  ...
TypeError: __init__() needs keyword-only argument max_length
```

**exception** `n6sdk.exceptions.`**`DataAPIError`**`(`**args*, ***kwargs*`)`

Bases: *`n6sdk.exceptions._ErrorWithPublicMessageMixin`*, `exceptions.Exception`

The base class for *client-data-or-backend-API*-related exceptions.

(They are **not** intended to be raised in `clean_*_value()` of `Field` subclasses – use *`FieldValueError`* instead.)

```
>>> exc = DataAPIError('a', 'b')
>>> exc.args
('a', 'b')
>>> exc.public_message   # using attribute default_public_message
u'Internal error.'
>>> unicode(exc)
u'Internal error.'
>>> str(exc)
'Internal error.'
>>> u'{}'.format(exc)
```

```
u'Internal error.'
>>> '{}'.format(exc)
'Internal error.'
```

```
>>> exc = DataAPIError('a', 'b', public_message='Spam.')
>>> exc.args
('a', 'b')
>>> exc.public_message   # the message passed into constructor
u'Spam.'
>>> unicode(exc)
u'Spam.'
>>> str(exc)
'Spam.'
>>> u'{}'.format(exc)
u'Spam.'
>>> '{}'.format(exc)
'Spam.'
```

**exception** `n6sdk.exceptions.`**`AuthorizationError`**`(`**args*, ***kwargs*`)`

Bases: *`n6sdk.exceptions.DataAPIError`*

Intended to be raised by *data backend API* to signal authorization problems.

**`default_public_message`** **= u'Access not allowed.'**

**exception** `n6sdk.exceptions.`**`TooMuchDataError`**`(`**args*, ***kwargs*`)`

Bases: *`n6sdk.exceptions.DataAPIError`*

Intended to be raised by *data backend API* when too much data have been requested.

**`default_public_message`** **= u'Too much data requested.'**

**exception** `n6sdk.exceptions.`**`ParamCleaningError`**`(`**args*, ***kwargs*`)`

Bases: *`n6sdk.exceptions.DataAPIError`*

The base class for exceptions raised when query parameter cleaning fails.

Instances of its subclasses are raised by the *data specification* machinery.

**`default_public_message`** **= u'Invalid parameter(s).'**

**exception** `n6sdk.exceptions.`**`ParamKeyCleaningError`**`(`*illegal_keys*, *missing_keys*`)`

Bases: *`n6sdk.exceptions._KeyCleaningErrorMixin`*, *`n6sdk.exceptions.ParamCleaningError`*

This exception should be raised by the *data specification* machinery (in particular, it is raised in `n6sdk.data_spec.BaseDataSpec.clean_param_dict()`) when some client-specified parameter keys (names) are illegal and/or missing.

This exception class provides *`default_public_message`* (see: *`_ErrorWithPublicMessageMixin`*) as a property whose value is a nice, user-readable message that includes all illegal and missing keys.

```
>>> try:
...     raise ParamKeyCleaningError({'zz', 'x'}, {'Ę', 'b'})
... except ParamCleaningError as exc:
...     pass
...
>>> exc.public_message == (
...     u'Illegal query parameters: "x", "zz". ' +
...     u'Required but missing query parameters: "\\u0118", "b".')
True
>>> exc.illegal_keys == {'zz', 'x'}
```

```
True
>>> exc.missing_keys == {'Ę', 'b'}
True
```

**illegal_keys_msg_template = u'Illegal query parameters: {}.'**

**missing_keys_msg_template = u'Required but missing query parameters: {}.'**

**default_public_message**
The aforementioned property.

**exception** n6sdk.exceptions.**ParamValueCleaningError**(*error_info_seq*)
Bases: *n6sdk.exceptions._ValueCleaningErrorMixin*, *n6sdk.exceptions.ParamCleaningError*

Raised when query parameter value(s) cannot be cleaned (are not valid).

Especially, this exception should be raised by the *data specification* machinery (in particular, it is raised in `n6sdk.data_spec.BaseDataSpec.clean_param_dict()`) when any `Exception` subclass(es)/instance(s) (possibly, *FieldValueError*) have been *caught after being raised by data specification fields'* `clean_param_value()`.

This exception class provides *default_public_message* (see: *_ErrorWithPublicMessageMixin*) as a property whose value is a nice, user-readable message that includes, *for each contained exception*: the key, the offending value(s) and the `public_message` attribute of that *contained exception* (the latter only for instances of *_ErrorWithPublicMessageMixin* subclasses).

```
>>> err1 = TypeError('foo', 'bar')
>>> err2 = FieldValueError('foo', 'bar', public_message='Message.')
>>> try:
...     raise ParamValueCleaningError([
...         ('k1', 'ł-1', err1),
...         ('k2', ['ł-2', 'xyz'], err2),
...     ])
... except ParamCleaningError as exc:
...     pass
...
>>> exc.public_message == (
...     u'Problem with value(s) ("\\u0142-1") of query parameter "k1". ' +
...     u'Problem with value(s) ("\\u0142-2", "xyz")' +
...     u' of query parameter "k2" (Message).')
True
>>> exc.error_info_seq == [
...     ('k1', 'ł-1', err1),
...     ('k2', ['ł-2', 'xyz'], err2),
... ]
True
```

**msg_template = u'Problem with value(s) ({values_repr}) of query parameter "{key}"{optional_exc_public_message}.'**

**default_public_message**
The aforementioned property.

**exception** n6sdk.exceptions.**ResultCleaningError**(**args*, ***kwargs*)
Bases: *n6sdk.exceptions.DataAPIError*

The base class for exceptions raised when result data cleaning fails.

Instances of its subclasses are raised by the *data specification* machinery.

**exception** n6sdk.exceptions.**ResultKeyCleaningError**(*illegal_keys*, *missing_keys*)
Bases: *n6sdk.exceptions._KeyCleaningErrorMixin*, *n6sdk.exceptions.ResultCleaningError*

This exception should be raised by the *data specification* machinery (in particular, it is raised in `n6sdk.data_spec.BaseDataSpec.clean_result_dict()`) when some keys in a data-backend-API-produced *result dictionary* are illegal and/or missing.

**Note:** `default_public_message` (see: *`_ErrorWithPublicMessageMixin`*) is consciously left as the default and safe `u'Internal error.'`.

**exception** `n6sdk.exceptions.`**`ResultValueCleaningError`**(*error_info_seq*)

Bases: *`n6sdk.exceptions._ValueCleaningErrorMixin`*, *`n6sdk.exceptions.ResultCleaningError`*

Raised when result item value(s) cannot be cleaned (are not valid).

Especially, this exception should be raised by the *data specification* machinery (in particular, it is raised in `n6sdk.data_spec.BaseDataSpec.clean_result_dict()`) when any `Exception` subclass(es)/instance(s) have been *caught after being raised by data specification fields'* `clean_result_value()`.

**Note:** `default_public_message` (see: *`_ErrorWithPublicMessageMixin`*) is consciously left as the default and safe `u'Internal error.'` – so (**unlike** for *`ParamValueCleaningError`* and fields' `clean_param_value()`) no information from underlying *`FieldValueError`* or other exceptions raised in fields' `clean_result_value()` is disclosed in the `default_public_message` value.

### 4.1.4 n6sdk.pyramid_commons

### 4.1.5 n6sdk.pyramid_commons.renderers

## 4.2 Helper modules

### 4.2.1 n6sdk.addr_helpers

`n6sdk.addr_helpers.`**`ip_network_as_tuple`**(*ip_network_str*)

```
>>> ip_network_as_tuple('10.20.30.40/24')
('10.20.30.40', 24)
```

`n6sdk.addr_helpers.`**`ip_network_tuple_to_min_max_ip`**(*ip_network_tuple*)

```
>>> ip_network_tuple_to_min_max_ip(('10.20.30.41', 24))
(169090560, 169090815)
>>> ip_network_tuple_to_min_max_ip(('10.20.30.41', 32))
(169090601, 169090601)
>>> ip_network_tuple_to_min_max_ip(('10.20.30.41', 0))
(0, 4294967295)
```

`n6sdk.addr_helpers.`**`ip_str_to_int`**(*ip_str*)

```
>>> ip_str_to_int('10.20.30.41')
169090601
```

### 4.2.2 n6sdk.class_helpers

`n6sdk.class_helpers.`**`attr_required`**`(`**attr_names*, ***kwargs*`)`

A method decorator: provides a check for presence of specified attributes.

**Some positional args:** Names of attributes that are required to be present *and* not to be the *dummy_placeholder* object (see below) when the decorated method is called.

**Kwargs:**

*dummy_placeholder* **(default: `None`):** The object that is not treated as a required value.

**Raises:**

`NotImplementedError`**:** When at least one of the specified attributes is set to the *dummy_placeholder* object or does not exist.

```
>>> class XX(object):
...     a = 1
...
...     @attr_required('a')
...     def meth_a(self):
...         print 'OK'
...
...     @attr_required('a', 'b')
...     def meth_ab(self):
...         print 'Excellent'
...
...     @attr_required('z', dummy_placeholder=NotImplemented)
...     def meth_z(self):
...         print 'Nice'
...
>>> x = XX()
>>> x.meth_a()
OK
>>> x.meth_ab()
Traceback (most recent call last):
  ...
NotImplementedError: ...
>>> x.b = 42
>>> x.meth_ab()
Excellent
>>> del XX.a
>>> x.meth_ab()
Traceback (most recent call last):
  ...
NotImplementedError: ...
>>> XX.a = None
>>> x.meth_ab()
Traceback (most recent call last):
  ...
NotImplementedError: ...
>>> x.meth_z()
Traceback (most recent call last):
  ...
NotImplementedError: ...
>>> x.z = None
>>> x.meth_z()  # OK as here `dummy_placeholder` is not None
Nice
>>> x.z = NotImplemented
>>> x.meth_z()
Traceback (most recent call last):
  ...
```

```
NotImplementedError: ...
```

n6sdk.class_helpers.**singleton**(*cls*)

A class decorator ensuring that the class can be instantiated only once.

**Args:** *cls*: the decorated class.

**Returns:** The same class (*cls*).

Trying to instantiate the decorated class more than once causes `RuntimeError` – unless, during provious instantiations, `__init__()` of the decorated class did not succeed (caused an exception).

Subclasses are also bound by this restriction (i.e. the decorated class and its subclasses are "counted" as one entity) – unless their `__init__()` is overridden in such a way that the `__init__()` of the decorated class is not called.

The check is thread-safe (protected with a lock).

```
>>> @singleton
... class X(object):
...     pass
...
>>> o = X()
>>> o = X()
Traceback (most recent call last):
  ...
RuntimeError: ...
```

```
>>> @singleton
... class X2(object):
...     def __init__(self, exc=None):
...         if exc is not None:
...             raise exc
...
>>> o = X2(ValueError('foo'))
Traceback (most recent call last):
  ...
ValueError: foo
>>> o = X2()
>>> o = X2()
Traceback (most recent call last):
  ...
RuntimeError: ...
>>> o = X2(ValueError('foo'))
Traceback (most recent call last):
  ...
RuntimeError: ...
```

```
>>> @singleton
... class Y(object):
...     def __init__(self, a, b, c=42):
...         print a, b, c
...
>>> class Z(Y):
...     pass
...
>>> class ZZZ(Y):
...     def __init__(self, a, b):
...         # will *not* call Y.__init__
...         print 'zzz', a, b
```

```
...
>>> o = Y('spam', b='ham')
spam ham 42
>>> o = Y('spam', b='ham')
Traceback (most recent call last):
  ...
RuntimeError: ...
>>> o = Z('spam', b='ham')
Traceback (most recent call last):
  ...
RuntimeError: ...
>>> o = ZZZ('spam', b='ham')
zzz spam ham
```

```
>>> @singleton
... class Y2(object):
...     def __init__(self, a, b, c=42):
...         print a, b, c
...
>>> class Z2(Y2):
...     pass
...
>>> class ZZZZZ(Y):
...     def __init__(self, a, b):
...         # *will* call Y.__init__
...         super(ZZZZZ, self).__init__(a, b=b)
...
>>> o = Z2('spam', b='ham')
spam ham 42
>>> o = Z2('spam', b='ham')
Traceback (most recent call last):
  ...
RuntimeError: ...
>>> o = Y2('spam', b='ham')
Traceback (most recent call last):
  ...
RuntimeError: ...
>>> o = ZZZZZ('spam', b='ham')
Traceback (most recent call last):
  ...
RuntimeError: ...
```

```
>>> class A(object):
...     def __init__(self, a, b, c=42):
...         print a, b, c
...
>>> @singleton
... class B(A):
...     pass
...
>>> o = A('spam', b='ham')
spam ham 42
>>> o = B('spam', b='ham')
spam ham 42
>>> o = B('spam', b='ham')
Traceback (most recent call last):
  ...
```

```
RuntimeError: ...
>>> o = A('spam', b='ham')
spam ham 42
>>> o = B('spam', b='ham')
Traceback (most recent call last):
  ...
RuntimeError: ...
```

### 4.2.3 n6sdk.datetime_helpers

**class** `n6sdk.datetime_helpers.`**`FixedOffsetTimezone`**(*offset*)

Bases: `datetime.tzinfo`

TZ-info to represent fixed offset in minutes east from UTC.

The source code of the class has been copied from http://docs.python.org/2.7/library/datetime.html#tzinfo-objects, then adjusted, enriched and documented.

```
>>> tz = FixedOffsetTimezone(180)
>>> tz
FixedOffsetTimezone(180)
```

```
>>> import copy
>>> tz is copy.copy(tz)
True
>>> tz is copy.deepcopy(tz)
True
```

```
>>> dt = datetime.datetime(2014, 5, 31, 1, 2, 3, tzinfo=tz)
>>> dt.utcoffset()
datetime.timedelta(0, 10800)
>>> dt.dst()
datetime.timedelta(0)
>>> dt.tzname()
'<UTC Offset: +180>'
>>> dt.astimezone(FixedOffsetTimezone(-60))
datetime.datetime(2014, 5, 30, 21, 2, 3, tzinfo=FixedOffsetTimezone(-60))
```

**`dst`**(*dt*)

**`tzname`**(*dt*)

**`utcoffset`**(*dt*)

`n6sdk.datetime_helpers.`**`date_by_isoweekday`**(*isoyear*, *isoweek*, *isoweekday*)

**Returns:** An equivalent `datetime.date` instance (see: http://en.wikipedia.org/wiki/ISO_week_date).

`n6sdk.datetime_helpers.`**`date_by_ordinalday`**(*year*, *ordinalday*)

**Returns:** An equivalent `datetime.date` instance.

`n6sdk.datetime_helpers.`**`datetime_to_utc_timestamp`**(*dt*)

Convert a `datetime.datetime` to a UTC timestamp.

**Args:** *dt*: A `datetime.datetime` instance (naive or TZ-aware).

**Returns:** The equivalent timestamp as a `float` number.

```
>>> naive_dt = datetime.datetime(2013, 6, 6, 12, 13, 57, 251211)
>>> t = datetime_to_utc_timestamp(naive_dt)
>>> t
1370520837.251211
>>> datetime.datetime.utcfromtimestamp(t)
datetime.datetime(2013, 6, 6, 12, 13, 57, 251211)
```

```
>>> tzinfo = FixedOffsetTimezone(120)
>>> tz_aware_dt = datetime.datetime(2013, 6, 6, 14, 13, 57, 251211,
...                                 tzinfo=tzinfo)
>>> t2 = datetime_to_utc_timestamp(tz_aware_dt)
>>> t2 == t
True
>>> utc_naive_dt = datetime.datetime.utcfromtimestamp(t2)
>>> utc_tzinfo = FixedOffsetTimezone(0)  # just UTC
>>> utc_tz_aware_dt = utc_naive_dt.replace(tzinfo=utc_tzinfo)
>>> utc_tz_aware_dt.hour
12
>>> tz_aware_dt.hour
14
>>> utc_tz_aware_dt == tz_aware_dt
True
```

n6sdk.datetime_helpers.**datetime_utc_normalize**(*dt*)

Normalize a `datetime.datetime` to a naive UTC one.

**Args:** *dt*: A `datetime.datetime` instance (naive or TZ-aware).

**Returns:** An equivalent `datetime.datetime` instance (a naive one).

```
>>> naive_dt = datetime.datetime(2013, 6, 6, 12, 13, 57, 251211)
>>> datetime_utc_normalize(naive_dt)
datetime.datetime(2013, 6, 6, 12, 13, 57, 251211)
```

```
>>> tzinfo = FixedOffsetTimezone(120)
>>> tz_aware_dt = datetime.datetime(2013, 6, 6, 14, 13, 57, 251211,
...                                 tzinfo=tzinfo)
>>> datetime_utc_normalize(tz_aware_dt)
datetime.datetime(2013, 6, 6, 12, 13, 57, 251211)
```

n6sdk.datetime_helpers.**is_datetime_format_normalized**(*s*)

```
>>> is_datetime_format_normalized('2013-06-13 10:02:00')
True
>>> is_datetime_format_normalized('2013-06-13 10:02:00.123400')
True
```

```
>>> is_datetime_format_normalized('2013-06-13 10:02')
False
>>> is_datetime_format_normalized('2013-06-13 10:02:00.000000')
False
>>> is_datetime_format_normalized('2013-06-13 10:02:00.1234')
False
>>> is_datetime_format_normalized('2013-06-13 10:02:00.12345678')
False
>>> is_datetime_format_normalized('2013-06-13T10:02:00')
False
```

```
>>> is_datetime_format_normalized('2013-06-13 10:02:00Z')
False
```

`n6sdk.datetime_helpers.`**`parse_iso_date`**(*s*, *prestrip=True*)

Parse *ISO-8601*-formatted date.

**Args:** *s*: *ISO-8601*-formatted date as a string.

**Kwargs:**

*prestrip* **(default: `True`):** Whether the `strip()` method should be called on the input string before performing the actual processing.

**Returns:** A `datetime.date` instance.

**Raises:** `ValueError` for invalid input.

Intentional limitation: specified date must include unambiguous day specification (inputs such as `'2013-05'` or `'2013'` are not supported).

```
>>> parse_iso_date('2013-06-12')
datetime.date(2013, 6, 12)
```

```
>>> parse_iso_date('99991231')
datetime.date(9999, 12, 31)
```

```
>>> parse_iso_date('2013-W24-3')
datetime.date(2013, 6, 12)
>>> datetime.date(2013, 6, 12).isocalendar()    # checking this was OK...
(2013, 24, 3)
```

```
>>> parse_iso_date('2013-W01-1')
datetime.date(2012, 12, 31)
>>> datetime.date(2012, 12, 31).isocalendar()   # checking this was OK...
(2013, 1, 1)
```

```
>>> parse_iso_date('2011-W52-7')
datetime.date(2012, 1, 1)
>>> datetime.date(2012, 1, 1).isocalendar()     # checking this was OK...
(2011, 52, 7)
```

```
>>> parse_iso_date('2013-001')
datetime.date(2013, 1, 1)
>>> parse_iso_date('2013-365')
datetime.date(2013, 12, 31)
>>> parse_iso_date('2012-366')   # 2012 was a leap year
datetime.date(2012, 12, 31)
```

```
>>> parse_iso_date('0000-01-01')
Traceback (most recent call last):
  ...
ValueError: ...
>>> parse_iso_date('13-01-01')
Traceback (most recent call last):
  ...
ValueError: ...
>>> parse_iso_date('01-01-2013')
Traceback (most recent call last):
  ...
ValueError: ...
```

```
>>> parse_iso_date('2013-6-01')
Traceback (most recent call last):
  ...
ValueError: ...
>>> parse_iso_date('2013-02-31')
Traceback (most recent call last):
  ...
ValueError: ...
>>> parse_iso_date('2013-W54-1')
Traceback (most recent call last):
  ...
ValueError: ...
>>> parse_iso_date('2013-W22-8')
Traceback (most recent call last):
  ...
ValueError: ...
>>> parse_iso_date('2013-W1-1')
Traceback (most recent call last):
  ...
ValueError: ...
>>> parse_iso_date('2013-W01-01')
Traceback (most recent call last):
  ...
ValueError: ...
>>> parse_iso_date('2013-000')
Traceback (most recent call last):
  ...
ValueError: ...
>>> parse_iso_date('2013-366')
Traceback (most recent call last):
  ...
ValueError: ...
>>> parse_iso_date('2013-1')
Traceback (most recent call last):
  ...
ValueError: ...
```

`n6sdk.datetime_helpers.`**`parse_iso_datetime`**(*s*, *prestrip=True*)

Parse *ISO-8601*-formatted combined date and time.

**Args:** *s*: *ISO-8601*-formatted combined date and time – as a string.

**Kwargs:**

*prestrip* **(default: `True`):** Whether the `strip()` method should be called on the input string before performing the actual processing.

**Returns:** A `datetime.datetime` instance (a TZ-aware one if the input does include time zone information, otherwise a naive one).

**Raises:** `exceptions.ValueError` for invalid input.

For notes about some limitations – see *`parse_iso_date()`* and *`parse_iso_time()`*.

`n6sdk.datetime_helpers.`**`parse_iso_datetime_to_utc`**(*s*, *prestrip=True*)

Parse *ISO-8601*-formatted combined date and time, and normalize it to UTC.

**Args:** *s*: *ISO-8601*-formatted combined date and time – as a string.

**Kwargs:**

*prestrip* **(default: `True`):** Whether the `strip()` method should be called on the input string before performing the actual processing.

**Returns:** A `datetime.datetime` instance (a naive one, normalized to UTC).

**Raises:** `exceptions.ValueError` for invalid input.

This function processes input by calling *`parse_iso_datetime()`* and *`datetime_utc_normalize()`*.

```
>>> parse_iso_datetime_to_utc('2013-06-13T10:02Z')
datetime.datetime(2013, 6, 13, 10, 2)
```

```
>>> parse_iso_datetime_to_utc('2013-06-13 10:02')
datetime.datetime(2013, 6, 13, 10, 2)
```

```
>>> parse_iso_datetime_to_utc('2013-06-13 10:02+02:00')
datetime.datetime(2013, 6, 13, 8, 2)
```

```
>>> parse_iso_datetime_to_utc('2013-06-13T22:02:04.1234-07:00')
datetime.datetime(2013, 6, 14, 5, 2, 4, 123400)
```

```
>>> parse_iso_datetime_to_utc('2013-06-13 10:02:04.123456789Z')
datetime.datetime(2013, 6, 13, 10, 2, 4, 123456)
```

```
>>> parse_iso_datetime_to_utc('  2013-06-13T10:02Z          ')
datetime.datetime(2013, 6, 13, 10, 2)
```

```
>>> parse_iso_datetime_to_utc('  2013-06-13T10:02Z          ', prestrip=False)
...
Traceback (most recent call last):
...
ValueError: ...
```

`n6sdk.datetime_helpers.`**`parse_iso_time`**(*s*, *prestrip=True*)

Parse *ISO-8601*-formatted time.

**Args:** *s*: *ISO-8601*-formatted time as a string.

**Kwargs:**

*prestrip* **(default: `True`):** Whether the strip() method should be called on the input string before performing the actual processing.

**Returns:** A `datetime.time` instance (a TZ-aware one if the input does include time zone information, otherwise a naive one).

**Raises:** `exceptions.ValueError` for invalid input.

Intentional limitation: specified time must include at least hour and minute. Second, microsecond and timezone information are optional.

*ISO-8601*-enabled "leap second" (60) is accepted but silently converted to 59 seconds + 999999 microseconds.

The optional fractional-part-of-second part can be specified with bigger or smaller precision – it will always be transformed to microseconds.

`n6sdk.datetime_helpers.`**`parse_python_formatted_datetime`**(*s*)

A limited version of *`parse_iso_datetime()`*: accepts only a string in the format: `%Y-%m-%d %H:%M:%S` or `%Y-%m-%d %H:%M:%S.%f` in terms of `datetime.datetime.strptime()`.

### 4.2.4 n6sdk.encoding_helpers

**class** `n6sdk.encoding_helpers.`**`AsciiMixIn`**

Bases: `object`

A mix-in class that provides the `__str__()`, `__unicode__()` and `__format__()` special methods based on *`ascii_str()`*.

```
>>> class SomeBase(object):
...     def __str__(self):
...         return 'Cośtam-cośtam'
...     def __format__(self, fmt):
...         return 'Nó i ' + fmt
...
>>> class MyClass(AsciiMixIn, SomeBase):
...     pass
...
>>> obj = MyClass()
```

```
>>> str(obj)
'Co\\u015btam-co\\u015btam'
>>> unicode(obj)
u'Co\\u015btam-co\\u015btam'
>>> format(obj)
'N\\xf3 i '
```

```
>>> 'Oto {0:ś}'.format(obj)
'Oto N\\xf3 i \\u015b'
>>> u'Oto {0:\\u015b}'.format(obj)   # unicode format string
u'Oto N\\xf3 i \\u015b'
>>> 'Oto {0!s}'.format(obj)
'Oto Co\\u015btam-co\\u015btam'
```

```
>>> 'Oto %s' % obj
'Oto Co\\u015btam-co\\u015btam'
>>> u'Oto %s' % obj                  # unicode format string
u'Oto Co\\u015btam-co\\u015btam'
```

`n6sdk.encoding_helpers.`**`as_unicode`**`(`*obj*`)`

Convert the given object to a `unicode` string.

Unlike `ascii_str()`, this function is not decoding-error-proof and does not apply any escaping.

The function requires that the given object is one of the following:

- a `unicode` string,
- a UTF-8-decodable `str` string,
- an object that produces one of the above kinds of strings when converted using `unicode` or `str`, or `repr()` (the conversions are tried in this order);

if not – `UnicodeDecodeError` is raised.

```
>>> as_unicode(u'')
u''
>>> as_unicode('')
u''
```

```
>>> as_unicode(u'O\u0142\xf3wek') == u'O\u0142\xf3wek'
True
>>> as_unicode('O\xc5\x82\xc3\xb3wek') == u'O\u0142\xf3wek'
True
>>> as_unicode(ValueError(u'O\u0142\xf3wek')) == u'O\u0142\xf3wek'
True
>>> as_unicode(ValueError('O\xc5\x82\xc3\xb3wek')) == u'O\u0142\xf3wek'
True
```

```
>>> class Hard(object):
...     def __str__(self): raise UnicodeError
...     def __unicode__(self): raise UnicodeError
...     def __repr__(self): return 'foo'
...
>>> as_unicode(Hard())
u'foo'
```

```
>>> as_unicode('\xdd')
Traceback (most recent call last):
  ...
UnicodeDecodeError: ...
```

`n6sdk.encoding_helpers.`**`ascii_str`**(*obj*)

Safely convert the given object to an ASCII-only string.

This function does its best to obtain a pure-ASCII string representation (possibly `str`/`unicode()`-like, though `repr()` can also be used as the last-resort fallback) – *not raising* any encoding/decoding exceptions.

The result is an ASCII str, with non-ASCII characters escaped using Python literal notation (`\x...`, `\u...`, `\U...`).

```
>>> ascii_str('')
''
>>> ascii_str(u'')
''
>>> ascii_str('Ala ma kota\nA kot?\n2=2 ')   # pure ASCII str => unchanged
'Ala ma kota\nA kot?\n2=2 '
>>> ascii_str(u'Ala ma kota\nA kot?\n2=2 ')
'Ala ma kota\nA kot?\n2=2 '
```

```
>>> ascii_str(ValueError('Ech, ale błąd!'))  # UTF-8 str => decoded
'Ech, ale b\\u0142\\u0105d!'
>>> ascii_str(ValueError(u'Ech, ale b\u0142\u0105d!'))
'Ech, ale b\\u0142\\u0105d!'
```

```
>>> ascii_str('\xee\xdd \t jaźń')  # non-UTF-8 str => using surrogateescape
'\\udcee\\udcdd \t ja\\u017a\\u0144'
>>> ascii_str(u'\udcee\udcdd \t ja\u017a\u0144')
'\\udcee\\udcdd \t ja\\u017a\\u0144'
```

```
>>> class Nasty(object):
...     def __str__(self): raise UnicodeError
...     def __unicode__(self): raise UnicodeError
...     def __repr__(self): return 'really nas\xc5\xa7y! \xaa'
...
>>> ascii_str(Nasty())
'really nas\\u0167y! \\udcaa'
```

`n6sdk.encoding_helpers.`**`provide_surrogateescape`**()

Provide the `surrogateescape` error handler for bytes-to-unicode decoding.

The source code of the function has been copied from https://bitbucket.org/haypo/misc/src/d76f4ff5d27c746c883d40160c8b4fb089 and then adjusted, optimized and commented. Original code was created by Victor Stinner and released by him under the Python license and the BSD 2-clause license.

The surrogateescape error handler is provided out-of-the-box in Python 3 but not in Python 2. It can be used to convert arbitrary binary data to Unicode in a practically non-destructive way.

**See also:**

https://www.python.org/dev/peps/pep-0383.

This implementation (for Python 2) covers only the decoding part of the handler, i.e. the str-to-unicode conversion. The encoding (unicode-to-str) part is not implemented. Note, however, that once we transformed a binary data into a *surrogate-escaped* Unicode data we can (in Python 2) freely encode/decode it (unicode-to/from-str), not using surrogateescape anymore, e.g.:

```
>>> # We assume that the function has already been called --
>>> # as it is imported and called in N6SDK/n6sdk/__init__.py
>>> b = 'ołówek \xee\xdd'          # utf-8 text + some non-utf-8 mess
>>> b
'o\xc5\x82\xc3\xb3wek \xee\xdd'
>>> u = b.decode('utf-8', 'surrogateescape')
>>> u
u'o\u0142\xf3wek \udcee\udcdd'
>>> b2 = u.encode('utf-8')
>>> b2                             # now all stuff is utf-8 encoded
'o\xc5\x82\xc3\xb3wek \xed\xb3\xae\xed\xb3\x9d'
>>> u2 = b2.decode('utf-8')
>>> u2 == u
True
```

```
>>> u.encode('latin2',
...           'surrogateescape')    # does not work for *encoding*
Traceback (most recent call last):
  ...
TypeError: don't know how to handle UnicodeEncodeError in error callback
```

This function is idempotent (i.e., it can be called safely multiple times – because if the handler is already registered the function does not try to register it again) though it is not thread-safe (typically it does not matter as the function is supposed to be called somewhere at the begginning of program execution).

**Note:** This function is called automatically on first import of n6sdk module or any of its submodules.

**Warning:** In Python 3 (if you were using a Python-3-based application or script to handle data produced with Python 2), the utf-8 codec (as well as other utf-... codecs) does not decode *surrogate-escaped* data encoded to bytes with the Python 2's utf-8 codec unless the surrogatepass error handler is used for decoding (on the Python 3 side).

### 4.2.5 n6sdk.regexes

This module contains several regular expression objects (most of them are used in other parts of the *n6sdk* library).

n6sdk.regexes.**CC_SIMPLE_REGEX = <_sre.SRE_Pattern object>**

Two-character country code.

Used by n6sdk.data_spec.fields.CCField.

`n6sdk.regexes.`**`DOMAIN_ASCII_LOWERCASE_REGEX`** **= <_sre.SRE_Pattern object>**
Domain name – with the underscore character allowed (as life is more eventful than RFCs, especially when it comes to maliciously constructed domain names).

Used by `n6sdk.data_spec.fields.DomainNameField` and `n6sdk.data_spec.fields.DomainNameSubstr`

`n6sdk.regexes.`**`DOMAIN_ASCII_LOWERCASE_STRICT_REGEX`** **= <_sre.SRE_Pattern object at 0x3032e30>**
Domain name – more strict (hopefully RFC-compliant) variant.

`n6sdk.regexes.`**`EMAIL_SIMPLIFIED_REGEX`** **= <_sre.SRE_Pattern object>**
E-mail address (very rough validation).

Used by `n6sdk.data_spec.fields.EmailSimplifiedField`.

`n6sdk.regexes.`**`IBAN_REGEX`** **= <_sre.SRE_Pattern object>**
International Bank Account Number.

Used by `n6sdk.data_spec.fields.IBANSimplifiedField`.

`n6sdk.regexes.`**`IPv4_ANONYMIZED_REGEX`** **= <_sre.SRE_Pattern object at 0x3096cf0>**
Anonymized IPv4 address.

Used by `n6sdk.data_spec.fields.AnonymizedIPv4Field`.

`n6sdk.regexes.`**`IPv4_CIDR_NETWORK_REGEX`** **= <_sre.SRE_Pattern object at 0x305e5b0>**
IPv4 network specification in CIDR notation.

Used by `n6sdk.data_spec.fields.IPv4NetField`.

`n6sdk.regexes.`**`IPv4_STRICT_DECIMAL_REGEX`** **= <_sre.SRE_Pattern object at 0x30586d0>**
IPv4 address in decimal dotted-quad notation.

Used by `n6sdk.data_spec.fields.IPv4Field`.

CHAPTER 5

# Release Notes

## 5.1 0.6.1 (2015-10-21)

Documentation-related changes:

- The *0.6.0* release made the contents of the `examples/BasicExample` directory out-of-date. Now, the directory has been completely removed and relevant documentation parts have been adjusted.

  Note: to generate a (richly commented) template for your *n6sdk*-based project you can use the `pcreate -s n6sdk YourProjectName` command (within the virtualenv in which *n6sdk* has been installed). See also: the *Tutorial*.

- Several documentation fixes and improvements (including fixes and rearrangements in these *release notes*).

## 5.2 0.6.0 (2015-10-13)

Significant or backward incompatible changes:

- Documentation: some **security-related** fixes in the *Installation for production...* and *Gluing it together* sections of the *Tutorial* – especially, related to necessary access restrictions on `production.ini` file.

- A new utility script added: `n6sdk_api_test`. It is – a tool to perform basic validation of your *n6sdk*-based API. (The script is automatically installed in the appropriate place when you install *n6sdk*.)

  See: the *n6sdk_api_test: API testing tool* chapter of the documentation.

- A new Pyramid *scaffold* added that makes it easy to create a skeleton of a new *n6sdk*-based REST API project – just run the shell command: `pcreate -s n6sdk YourProjectName` (within the *virtualenv* in which *n6sdk* has been installed).

  See: the updated *Tutorial*.

- The *n6sdk.data_spec.DataSpec* data specification class has been deeply changed as well as a new data specification class has been added: *n6sdk.data_spec.AllSearchableDataSpec* (being a subclass of *DataSpec*).

  Now, all field specifications defined within *DataSpec* have the flag `in_params` set to `None`, that is, are *not* marked as query parameters.

  It means that you need to create a subclass of *DataSpec* and, in that subclass, extend desired field specifications to *enable* them explicitly as query parameters (e.g., by using `Ext(in_params='optional')`).

  Alternatively, you can create a subclass of *AllSearchableDataSpec* (which is a *DataSpec* subclass similar to the previous version of *DataSpec*, i.e., with most of fields marked as possible query parameters, using `in_params='optional'`) – then you may want to extend some field specifications (by using `Ext(in_params=None)`) to *disable* them as query parameters.

See: the updated *Tutorial*.

- Now, the field specification *DataSpec.url_pattern* requires that `url_pattern` values (concerning query parameter values as well as result values), if present, are *not* empty.

  To implement this restriction, a new boolean flag has been added to *n6sdk.data_spec.fields.UnicodeField* (and all its subclasses): `disallow_empty`, settable as a constructor argument or a subclass attribute. If the flag is true, values are not allowed to be empty. The default value of the flag is `False`.

- A fix that affects *n6sdk.data_spec.DataSpec.fqdn*, *n6sdk.data_spec.fields.DomainNameField* as well as *n6sdk.regexes.DOMAIN_ASCII_LOWERCASE_REGEX* and *n6sdk.regexes.DOMAIN_ASCII_LOWERCASE_STRICT_REGEX*: now, the top-level part of a domain name (TLD) is no longer allowed to consist of digits only (thanks to that, the possibility to erroneously accept an IPv4 address as a domain name has been suppressed).

- A new category added to *DataSpec.category.enum_values*: `'malware-action'`.

- A new field specification added to *DataSpec*: `action`, intented to be used for events whose category is `'malware-action'` (mentioned above).

- Exception handling has been revamped. Among others, now `content-type` of HTTP error pages is set to `text/plain` (not to `text/html`).

  A few significant changes related to that novelties have been applied to the following *n6sdk.pyramid_commons* classes: *DefaultStreamViewBase*, *HttpResource* and *ConfigHelper* (please analyze their code if you need detailed information); most notably:

  – the *DefaultStreamViewBase.concrete_view_class()* class method takes the fifth obligatory argument (and obligatory subclass attribute): `adjust_exc` (see the documentation of the method);

  – the *HttpResource.configure_views()* method (which calls *DefaultStreamViewBase.concrete_view_class()*) takes the second obligatory argument: `adjust_exc`;

  – the *ConfigHelper.complete()* method (called, if needed, by *ConfigHelper.make_wsgi_app()*) registers appropriate Pyramid *exception views* as well as specifies appropriate callable as the `adjust_exc` argument for *HttpResource.configure_views()* (if you are interested in details, please, see the *ConfigHelper* source code, especially the code of *complete()* and of two new class methods: *exception_view()* and *exc_to_http_exc()*).

- Now, all *ConfigHelper* constructor arguments are *officially required* to be specified as keyword arguments (not as positional ones).

- The *pyramid* library (an existing external dependency) is now restricted to be not newer than version *1.5.7*.

Other changes:

- New external dependencies added: python-cjson and requests (used by the `n6sdk_api_test` tool mentioned above).

- A bugfix: now, log messages from the *n6sdk.pyramid_commons* module are emitted using the `'n6sdk.pyramid_commons'` logger rather than the root logger.

- A new public helper function added: *n6sdk.pyramid_commons.renderers.data_dict_to_json()*; it defines how the standard renderers `json` and `sjson` serialize each data record (for details, see the documentation of the function);

- Various minor code cleanups, refactorizations and improvements.

- New and improved unit tests and doctests.

- A lot of documentation improvements and fixes.

## 5.3 0.5.0 (2015-04-18)

Significant or backward incompatible changes:

- Now, multiple values for a client query parameter can be specified in URL query strings in two alternative ways:
  - separated with commas, within one query string item (as in past *n6sdk* versions), e.g.: `category=bots,dos-attacker,phish`;
  - as individual query string items (the way introduced in this *n6sdk* release), e.g.: `category=bots&category=dos-attacker&category=phish`.

  Implementation of the extension caused the following changes in the *n6sdk* programming interfaces:

  - now, the argument for *<data specification>.clean_param_dict()* is a dictionary that maps query parameter names to *lists of individual uncleaned parameter values* (in past *n6sdk* versions it mapped to *strings consisting of comma-separated uncleaned parameter values*);
  - extraction of individual query parameter values from the URL's query string – including splitting comma-separated sequences of values – is now *entirely outside* of the data specification machinery and field classes; the *n6sdk.data_spec.fields.Field._split_raw_param_value()* non-public method has been removed.
  - the interface of the *n6sdk.exceptions.ParamValueCleaningError* constructor has been extended a bit: now the second item of a 3-tuple being an item of an *error_info_seq* argument can be either a single value (as previously) or a list of values.

  The *Tutorial* and other parts of the documentation have been adjusted appropriately.

- A lot of changes related to *data specification fields*:
  - New field classes in the *n6sdk.data_spec.fields* module:
    - *IPv6Field* (for IPv6 addresses),
    - *IPv6NetField* (for IPv6 network specifications),
    - *EmailSimplifiedField* (for e-mail addresses),
    - *IBANSimplifiedField* (for International Bank Account Numbers),
    - *ListOfDictsField* (for lists of dictionaries containing arbitrary data),
    - *DirField* (two-value enumeration: `'src'` or `'dst'`),
    - *ExtendedAddressField* (for lists of address data items – see the change in the `address` field specification, mentioned below).
  - Modified field classes in the *n6sdk.data_spec.fields* module:
    - *DictResultField*:
      - the `key_to_subfield_factory` attribute is no longer obligatory;
      - the `required_keys` attribute is gone;
      - the *clean_param_value()* method now raises *TypeError* instead of *NotImplementedError*;
    - *AddressField*:
      - now inherits from *ListOfDictsField*, not directly from *ResultListFieldMixin* and *DictResultField*;
      - the `required_keys` attribute is gone; `ip` subfield is still obligatory – but now this requirement is implemented internally;

        · the *clean_param_value()* method now raises *TypeError* instead of *NotImplementedError*.

- New field specifications added to the *n6sdk.data_spec.DataSpec* class:
    * `time.until` (*DateTimeField*, params-only),
    * `active.until` (*DateTimeField*, params-only),
    * `modified` (*DateTimeField*, results-only),
    * `modified.min` (*DateTimeField*, params-only),
    * `modified.max` (*DateTimeField*, params-only),
    * `modified.until` (*DateTimeField*, params-only),
    * `ipv6` (*IPv6Field*, params-only),
    * `ipv6.net` (*IPv6NetField*, params-only),
    * `injects` (*ListOfDictsField*, results-only),
    * `registrar` (*UnicodeLimitedField*),
    * `url_pattern` (*UnicodeLimitedField*),
    * `username` (*UnicodeLimitedField*),
    * `x509fp_sha1` (*SHA1Field*),
    * `email` (*EmailSimplifiedField*),
    * `iban` (*IBANSimplifiedField*),
    * `phone` (*UnicodeLimitedField*).
- The `address` field specification (at *n6sdk.data_spec.DataSpec*) has been changed: now it is an *ExtendedAddressField* instance – its subfields include:
    * `ip`/`ipv6` (*IPv4Field*/*IPv6Field*, obligatory – which means that either `'ip'` or `'ipv6'`, but *not* both, must be present in each member dictionary),
    * `cc` (*CCField*),
    * `asn` (*ASNField*),
    * `dir` (*DirField*),
    * `rdns` (*DomainNameField*).
- New categories added to *DataSpec.category.enum_values*:
    * `'amplifier'`,
    * `'backdoor'`,
    * `'dns-query'`,
    * `'flow'`,
    * `'flow-anomaly'`,
    * `'fraud'`,
    * `'leak'`,
    * `'vulnerable'`,
    * `'webinject'`.

The *Tutorial* has been adjusted appropriately.

- Both standard renderers (`json` and `sjson`) now add the `"Z"` suffix (indicating the UTC time) to all *date+time* values.
- The `sjson` renderer now generates an additional empty line to indicate the end of data stream.

Other changes:

- A new external dependency: the ipaddr library.
- New and improved unit tests and doctests.
- Several documentation improvements and fixes.

## 5.4 0.4.0 (2014-12-23)

This is the first public, *free/open-source*-licensed release of *n6sdk*.

Backward incompatible (though rather minor) changes:

- Changed behaviour of the standard `json` and `sjson` renderers (defined in *n6sdk.pyramid_commons.renderers* as the *StreamRenderer_json* and *StreamRenderer_sjson* classes): now they make use of a new helper function, *dict_with_nulls_removed()*, that replaces the old mechanism of recursive removing of `None`-or-empty values from result dictionaries: previously, values equal to zero (such as `0`, `0.0` or `False`) were also removed; now they are kept (note that values being `None`, empty containers and empty strings are still removed).
- Now, in the *n6sdk.pyramid_commons.DefaultStreamViewBase.call_api()* method, an *n6sdk.exceptions.TooMuchDataError* exception from *call_api_method()* or from data specification's *clean_result_dict()* causes *pyramid.httpexceptions.HTTPForbidden* and not *pyramid.httpexceptions.HTTPServerError*.
- The *n6sdk.class_helpers.singleton()* class decorator is now more lenient: instantiation does not count if *__init__()* of a decorated class raised (or propagated) an exception.

Other changes:

- Bugfix in the *n6sdk.pyramid_commons.DefaultStreamViewBase.concrete_view_class()* class method: now the check of the given renderer labels against the set of registered renderers works properly; previously it behaved nonsensically: accepted unregistered labels (causing further *KeyError* exceptions) and at the same time demanded that all registeted labels had to be used.
- Furthermore, *n6sdk.pyramid_commons.DefaultStreamViewBase* has a new class attribute: *break_on_result_cleaning_error*, by default set to `True`. In custom subclasses it can be set to `False` – then result dictionaries that cannot be cleaned will be skipped (and a proper warning will be recorded to the logs) instead of causing *pyramid.httpexceptions.HTTPServerError*.
- The *n6sdk.pyramid_commons.renderers.dict_with_nulls_removed()* function (mentioned above) is exposed as a public helper (it may be useful when implementing custom renderers).
- The *n6sdk.data_spec.fields.Field* class (and its subclasses) as well as *n6sdk.datetime_helpers.FixedOffsetTimezone* – have custom implementations of the *__repr__()* method (producing more readable results).
- Various minor code cleanups, refactorizations and improvements.
- New and improved unit tests and doctests.

Documentation-related news (including big ones!):

- Now the documentation is generated with Sphinx.
- A new, long *Tutorial* has been added.

- A bunch of docstrings have been added.
- Contents of many docstrings have been improved.
- All docstrings are now *reStructuredText*-formatted and used as a part of the *Sphinx*-generated documentation.
- The former `CHANGES.txt` file has been *reStructuredText*-formatted, renamed to `NEWS.rst` and used as a part of the *Sphinx*-generated documentation. There is also a new `README.rst` file, also included in the generated documentation.
- The former `README.txt` file has been moved to `examples/BasicExample` and sligthly improved.
- Furthermore, some other *BasicExample* improvements have been made (cleanups, refactorizations and minor fixes; among others, the *version* field in the *BasicExample*'s `setup.py` file no longer follows the *n6sdk* version; from now it is just `"0.0.1"`).

## 5.5 0.3.0 (2014-08-12)

Significant or backward incompatible changes:

- Network incident category `"ddos"` has been replaced with two separate categories: `"dos-attacker"` and `"dos-victim"` (see: *n6sdk.data_spec.CATEGORY_ENUMS*).
- *n6sdk.data_spec.fields.ResultListFieldMixin.clean_result_value()* no longer accepts *collections.Set* instances (now it accepts only *collection.Sequence* instances that are not *str*/*unicode* instances).

## 5.6 0.2.0 (2014-08-08)

Significant or backward incompatible changes:

- Changes in the base data specification class (*n6sdk.data_spec.DataSpec*) and/or in the classes defined in the *n6sdk.data_spec.fields* module:
  - the *source* field is now an instance of a new class: *n6sdk.data_spec.fields.SourceField* – which implements more restricted validation of values; now each value not only needs to be at most 32-characters long, but also it must consist of two non-empty parts, separated with exactly one dot character (`'.'`), containing only lowercase ASCII letters, digits and hyphens (`'-'`).
  - a change in *n6sdk.data_spec.fields.DateTimeField* that affects the *time*, *expires* and *until* fields of *DataSpec*: the *clean_result_value()* method now accepts also *ISO*-formatted date-and-time strings (not only *datetime.datetime* instances);
  - a change in *n6sdk.data_spec.fields.IntegerField* that affects the *sport*, *dport* and *count* fields of *DataSpec*: in *clean_result_value()*, the former strict is-instance check (*int*/*long*) has been replaced with a duck-typed coercion, accepting anything that can be converted using *int()* without information loss (e.g. a *float* being an integer number, such as `42.0`, or a string being a decimal representation of an integer number, such as `'42'` – but not `'42.0'`);
  - a change in *n6sdk.data_spec.fields.ASNField* that affects the *address* (namely: *asn* of its subitems) and *asn* fields of *DataSpec*: the *clean_*_value()* methods now accept strings (*str*/*unicode*):
    * either being a decimal representation of an integer number in range `0`..‘‘2**32-1‘‘, e.g. `'98765432'` (formely only *clean_param_value()* accepted such strings),
    * or consisting of two dot-separated decimal representations of integer numbers in range `0`..‘‘2**16-1‘‘, e.g. `'34567.65432'` (formely such a notation was not accepted at all);

note: `clean_result_value()` still accepts also *int* and *long* values in range `0`..‘‘2**32-1‘‘ (and still does not accept instances of *float* and other types).

- a change in *n6sdk.data_spec.fields.CCField* that affects the *address* (namely: *cc* of its subitems) and *cc* fields of *DataSpec*: the *clean_*_value()* methods now accept also lowercase letters (which are automatically uppercased);
- a change in *n6sdk.data_spec.fields.DomainNameSubstringField* that affects the *fqdn* (note: *DomainNameField* is a subclass of *DomainNameSubstringField*) and *fqdn.sub* fields of *DataSpec*: the value of *max_length* has been changed from `253` to `255`;
- a change in *n6sdk.data_spec.fields.DomainNameField* that affects the *fqdn* field of *DataSpec*: the regular expression the values are matched against is now more liberal (especially, underscores are now allowed; rationale: real-life domain names – especially those maliciously constructed – are not necessarily RFC-compliant; see: *n6sdk.regexes.DOMAIN_ASCII_LOWERCASE_REGEX* for details);
- a change in *n6sdk.data_spec.fields.AnonymizedIPv4Field* that affects the *adip* field of *DataSpec*: the *clean_*_value()* methods now accept also `'X'` (uppercased `'x'`) segments which are automatically lowercased;
- the *adip* field is no longer enabled as a query parameter (field’s *in_params* is now set to `None`);
- a change in *n6sdk.data_spec.fields.HexDigestField* that affects the *md5* and *sha1* fields of *DataSpec*: the *clean_*_value()* methods now accept also non-lowercase hexadecimal digit letters (which are automatically lowercased);
- the former *hash_algo* attribute of *UnicodeField* class/subclasses/instances has been renamed to *hash_algo_descr*;
- *n6sdk.data_spec.fields.URLField* is now a subclass of *n6sdk.data_spec.fields.URLSubstringField*;
- *n6sdk.data_spec.fields.ListField* has been removed (use *ResultListFieldMixin* instead);
- the former *n6sdk.data_spec.fields.AddressField* implementation has been replaced with a new one, especially the implementation of the methods has been factored out to new generic base classes: *ResultListFieldMixin* and *DictResultField*; some details have changed in a backwards-incompatible way – notably: *key_to_subfield_class* has been renamed to *key_to_subfield_factory*.

• Changes in signatures of the *n6sdk.data_spec.BaseDataSpec* methods: *clean_param_dict()*, *clean_param_keys()*, *clean_result_dict()*, *clean_result_keys()*:

- replaced the optional argument *keys_to_ignore* with the *ignored_keys* keyword-only argument (still optional),
- added other optional arguments: *forbidden_keys*, *extra_required_keys*, *discarded_keys*.

• Changes in *n6sdk.pyramid_commons*:

- functions *init_pyramid_config()* and *complete_pyramid_config()* have been removed; use the new *ConfigHelper* class instead (for details – see its documentation, its code and the examples in `examples/BasicExample`...);
- a new function added: *register_stream_renderer()* (see below);
- the signature of the *StreamResponse* class constructor changed: *renderer* has been renamed to *renderer_name*; also, now the value of that argument can be any name registered with the new function *register_stream_renderer()* (see its documentation for details); `'json'` and `'sjson'` are registered out-of-the-box;
- the *DefaultStreamViewBase* class has been revamped in a backward-incompatibile way (please analyze its code if you need detailed information); most notably:

    - now the *concrete_view_class()* class method has completely different signature (see its documentation for details; note that *data_spec* now must be an instance, not a class); now each concrete subclass must have specified the *resource_id*, *renderers*, *data_spec* and *data_backend_api_method* attributes (for more information, also see the documentation of the *concrete_view_class()* class method mentioned above);
    - formely, the data specification's *clean_param_dict()* call performed in *prepare_params()* was guarded only against *ParamCleaningError* (transformed into *pyramid.httpexceptions.HTTPBadRequest*, when caught); now, also other exceptions are handled: *n6sdk.exceptions.AuthorizationError* (transformed into *pyramid.httpexceptions.HTTPForbidden*) and generic *n6sdk.exceptions.DataAPIError* (logged as an error and transformed into *pyramid.httpexceptions.HTTPServerError*) [note the symmetry between the *prepare_params()* and *call_api()* methods];
    - the possibility of specifying keyword arguments for data specification's *clean_*_dict()* calls as well as for data backend API's method call has been added (see the *get_clean_param_dict_kwargs()*, *get_clean_result_dict_kwargs()* and *get_extra_api_kwargs()* hook methods; the default implementation of each of them returns just an empty dict);
  - backward-incompatibile chages in the signature of the constructor of the *HttpResource* class:
    - now all arguments should be specified as keyword ones (never positional, i.e. you cannot rely on argument order any more);
    - now *data_spec* must be an instance, not a class;

    note: see the documentation of this class for details.
- The module *n6sdk.data_backend_api* (together with the decorator *n6sdk.data_backend_api.data_backend_api_method*) has been removed. It is no longer required to decorate or mark your custom data backend API class or its methods in any special way.
- Unused *n6sdk.exceptions.InvalidCallError* has been removed.
- *n6sdk.exceptions.FieldValueTooLongError* has been added (see below).

Other changes:

- Appropriate adjustments in `examples/BasicExample`.
- Some non-essential changes related to *n6sdk.data_spec.fields*:
  - if the given value is too long, the *clean_*_value()* methods of *n6sdk.data_spec.fields.UnicodeLimitedField* (and of its subclasses) now raise a new exception *n6sdk.exceptions.FieldValueTooLongError* (which is a subclass of *n6sdk.exceptions.FieldValueError* that was formely raised) – see its documentation for details about attributes of its instances (that attributes can be useful, for example, when implementing external trimming of too long values...);
  - it is now explicitly required for *n6sdk.data_spec.fields.HexDigestField* instances (and for instances of its subclasses) that *num_of_characters* and *hash_algo_descr* are specified (as subclass attributes or constructor arguments);
  - it is now explicitly required for *n6sdk.data_spec.fields.UnicodeLimitedField* instances (and for instances of its subclasses) that *max_length* is not less than 1.
- Module *n6sdk.addr_helpers* added.
- Major refactorings and several minor additions, improvements, fixes and cleanups.
- Improvements in the documentation (a lot of improved/added docstrings, improved `README.txt`, added `CHANGES.txt`...) and code comments.
- `MANIFEST.in` and other package setup improvements and cleanups.

- New and improved unit tests and doctests.

## 5.7 0.0.1 (2014-04-25)

Initial release.

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search