# ievv$_{o}pensourceDocumentation$

***Release 5.21.0***

**ievv**$_{o}pensource$

**Apr 22, 2020**

# CONTENTS

CHAPTER

ONE

# INSTALL

```
$ pip install ievv_opensource
```

CHAPTER

TWO

# DEVELOPMENT

## 2.1 Development guide

### 2.1.1 Install the requirements

Install the following:

1. Python
2. PIP
3. VirtualEnv
4. pipenv

### 2.1.2 Install development requirements

Install the development requirements:

```
$ pipenv install --dev
```

**Note:** The commands below assume you are in the virtualenv. You activate the virtualenv with:

```
$ pipenv shell
```

You can also run all the commands with:

```
$ pipenv run <command>
```

### 2.1.3 Build the docs

*Enable the virtualenv*, and run:

```
$ ievv docs --build --open
```

### 2.1.4 Create a development database

*Enable the virtualenv*, and run:

```
$ ievv recreate_devdb
```

### 2.1.5 Running tests

To run the tests, we need to use a different settings file. We tell ievvtasks to do this using the `DJANGOENV` environent variable:

```
$ DJANGOENV=test python manage.py test
```

### 2.1.6 Adding more dependencies

Just add dependencies as normal with pipenv, BUT make sure you run:

```
$ pipenv lock -r > requirements.txt
```

when you merge into master before you push IF you add any non-dev dependencies. This is because readthedocs requires `requirements.txt` to build.

CHAPTER

# THREE

# APPS

## 3.1 *ievv_tagframework* — General purpose tagging framework

The intention of this module is to provide a re-usable tagging framework for Django apps.

### 3.1.1 Data model API

**class** `ievv_opensource.ievv_tagframework.models.`**`Tag`**`(`**args*, ***kwargs*`)`

A single tag.

A tag has a unique name, and data models is added to a tag via `TaggedObject`.

**`taglabel`**

The label for the tag.

**`tagtype`**

The tagtype is a way for applications to group tags by type. No logic is assigned to this field by default, other than that is is `db_indexed`. The valid choices for the field is configured via the `IEVV_TAGFRAMEWORK_TAGTYPE_CHOICES` setting.

**`static get_tagtype_choices`**`()`

Returns choices for `tagtype`.

This is not set as choices on the field (because changing that would trigger a migration), but it should be used in any form displaying tagtype.

You configure the return value via the `IEVV_TAGFRAMEWORK_TAGTYPE_CHOICES` Django setting.

**`classmethod get_tagtype_valid_values`**`()`

Returns an iterator over the choices returned by `get_tagtype_choices()`, but only the values (not the labels) as a flat list.

**`clean`**`()`

Hook for doing any extra model-wide validation after clean() has been called on every field by self.clean_fields. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field defined by NON_FIELD_ERRORS.

**`exception DoesNotExist`**

**`exception MultipleObjectsReturned`**

**class** `ievv_opensource.ievv_tagframework.models.`**`TaggedObject`**`(`**args*, ***kwargs*`)`

Represents a many-to-many relationship between any data model object and a `Tag`.

**`tag`**

The `Tag`.

**content_type**
The ContentType of the tagged object.

**object_id**
The ID of the tagged object.

**content_object**
The GenericForeignKey using `content_type` and `object_id` to create a generic foreign key to the tagged object.

**exception DoesNotExist**

**exception MultipleObjectsReturned**

## 3.2 *ievv_batchframework* — Framework for batch/bulk tasks

The intention of this module is to make it easier to write code for background tasks and some kinds of bulk operations.

### 3.2.1 Configuration

Add the following to your `INSTALLED_APPS`-setting:

‘ievv_opensource.ievv_batchframework.apps.BatchOperationAppConfig’

### 3.2.2 Batchregistry - the high level API

### 3.2.3 Recommended Celery setup

#### Install Redis

Redis is very easy to install and use, and it is one of the recommended broker and result backends for Celery, so we recommend that you use this when developing with Celery. You may want to use the Django database instead, but that leaves you with a setup that is further from a real production environment, and using Redis is very easy if you use *ievv devrun* as shown below.

On Mac OSX, you can install redis with Homebrew using:

```
$ brew install redis
```

and most linux systems have Redis in their package repository. For other systems, go to http://redis.io/, and follow their install guides.

#### Configure Celery

First, you have to create a Celery Application for your project. Create a file named `celery.py` within a module that you know is loaded when Django starts. The safest place is in the root of your project module. So if you have:

```
myproject/
    __init__.py
    myapp/
        __init__.py
        models.py
```

(continues on next page)

(continued from previous page)

```
    mysettings/
        settings.py
```

You should add the celery configuration in `myproject/celery.py`. The rest of this guide will assume you put it at this location.

Put the following code in `myproject/celery.py`:

```
from __future__ import absolute_import
import os
from celery import Celery

# Ensure this matches your
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myproject.settings')

# The ``main``-argument is used as prefix for celery task names.
app = Celery(main='myproject')

# We put all the celery settings in out Django settings so we use
# this line to load Celery settings from Django settings.
# You could also add configuration for celery directly in this
# file using app.conf.update(...)
app.config_from_object('django.conf:settings')

# This debug task is only here to make it easier to verify that
# celery is working properly.
@app.task(bind=True)
def debug_add_task(self, a, b):
    print('Request: {0!r} - Running {} + {}, and returning the result.'.format(
        self.request, a, b))
    return a + b
```

And put the following code in `myproject/__init__.py`:

```
from __future__ import absolute_import

# This will make sure the Celery app is always imported when
# Django starts so that @shared_task will use this app.
from .celery import app as celery_app
```

Add the following to your Django settings:

```
# Celery settings
BROKER_URL = 'redis://localhost:6379'
CELERY_RESULT_BACKEND = 'redis://localhost:6379'
CELERY_ACCEPT_CONTENT = ['application/json']
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'
CELERY_TIMEZONE = 'Europe/Oslo'  # Change to your preferred timezone!
CELERY_IMPORTS = [
    'ievv_opensource.ievv_batchframework.celery_tasks',
]
CELERYD_TASK_LOG_FORMAT = '[%(asctime)s: %(levelname)s/%(processName)s] ' \
                          '[%(name)s] ' \
                          '[%(task_name)s(%(task_id)s)] ' \
                          '%(message)s'
```

(continues on next page)

(continued from previous page)

```
# ievv_batchframework settings
IEVV_BATCHFRAMEWORK_CELERY_APP = 'myproject.celery_app'
```

Setup *ievv devrun — All your development servers in one command*, and add `ievvdevrun.runnables.redis_server.RunnableThread()` and `` to your `IEVVTASKS_DEVRUN_RUNNABLES`. You should end up with something like this:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        # ievvdevrun.runnables.dbdev_runserver.RunnableThread(),  # Uncomment if␣
↪using django_dbdev
        ievvdevrun.runnables.django_runserver.RunnableThread(),
        ievvdevrun.runnables.redis_server.RunnableThread(),
        ievvdevrun.runnables.celery_worker.RunnableThread(app='myproject'),
    ),
}
```

At this point, you should be able to run:

```
$ ievv devrun
```

to start the Django server, redis and the celery worker. To test that everything is working:

1. Take a look at the output from `ievv devrun`, and make sure that your `debug_add_task` (from celery.py) is listed as a task in the `[tasks]` list printed by the celery worker on startup. If it is not, this probably means you did not put the code in the `myproject/__init__.py` example above in a place that Django reads at startup. You may want to try to move it into the same module as your `settings.py` and restart `ievv devrun`.

2. Start up the django shell and run the `debug_add_task`:

   ```
   $ python manage.py shell
   >>> from myproject.celery import debug_add_task
   >>> result = debug_add_task.delay(10, 20)
   >>> result.wait()
   30
   ```

   If this works, Celery is configured correctly.

### 3.2.4 Developing with asyncronous actions

When developing with asyncronous tasks with the setup from the introduction guide above, you need to restart `ievv devrun` each time you change some code used by an asyncronous action. This means that if you add an Action or ActionGroup, or change any code used within an Action or ActionGroup, you have to stop `ievv devrun`, and start it again. We provide two options for avoiding this.

#### Option 1: Run all actions synchronously

This is great for unit tests, and for developing and debugging code in your `ievv_opensource.ievv_batchframework.batchregistry.Action.executable()` methods. To enable this, add the following to your settings:

```
IEVV_BATCHFRAMEWORK_ALWAYS_SYNCRONOUS = True
```

#### Option 2: Run celery worker manually

Restarting `ievv devrun` can take some time if you have lots of commands that have to stop and start again. You can save some time for each change if you remove/comment out the `ievvdevrun.runnables.celery_worker.RunnableThread` line from the `IEVVTASKS_DEVRUN_RUNNABLES` setting (restart `ievv devrun` after this change), and run Celery manually with the following command instead:

```
$ celery -A myproject worker -l debug
```

Now you can start and stop only the Celery worker instead of restarting `ievv devrun`.

### 3.2.5 Batchregistry API

### 3.2.6 The BatchOperation model

The BatchOperation model is at the hearth of `ievv_batchframework`. Each time you start a batch process, you create an object of `ievv_opensource.ievv_batchframework.models.BatchOperation` and use that to communicate the status, success/error data and other metadata about the batch operation.

#### Asynchronous operations

An asynchronous operation is the most common use case for the BatchOperation model. It is used to track a task that is handled (E.g.: completed) by some kind of asynchronous service such as a cron job or a Celery task.

Lets say you have want to send an email 15 minutes after a blog post has been created unless the user cancels the email sending within within 15 minutes. You would then need to:

- Create a BatchOperation object each time a blog post is created.
- Use some kind of batching service, like Celery, to poll for BatchOperation objects that asks it to send out email.
- Delete the BatchOperation if a user clicks "cancel" within 15 minutes of the creation timestamp.

The code for this would look something like this:

```
from ievv_opensource.ievv_batchframework.models import BatchOperation

myblogpost = Blog.objects.get(...)
BatchOperation.objects.create_asynchronous(
    context_object=myblogpost,
    operationtype='new-blogpost-email')
# code to send the batch operation to the batching service (like celery)
# with a 15 minute delay, or just a service that polls for
# BatchOperation.objects.filter(operationtype='new-blogpost-email',
#                               created_datetime__lt=timezone.now() -␣
↪timedelta(minutes=15))


# The batching service code
def my_batching_service(...):
    batchoperation = BatchOperation.objects.get(...)
    batchoperation.mark_as_running()
    # ... send out the emails ...
    batchoperation.finish()


# In the view for cancelling email sending
BatchOperation.objects\
    .filter(operationtype='new-blogpost-email',
            context_object=myblogpost)\
    .remove()
```

### Synchronous operations

You may also want to use BatchOperation for synchronous operations. This is mainly useful for complex bulk create and bulk update operations.

Lets say you have Game objects with a one-to-many relationship to Player objects with a one-to-many relationship to Card objects. You want to start all players in a game with a card. How to you batch create all the players with a single card?

You can easily batch create players with `bulk_create`, but you can not batch create the cards because they require a Player. So you need to a way of retrieving the players you just batch created.

If you create a BatchOperation with `context_object` set to the Game, you will get a unique identifier for the operation (the id of the BatchOperation). Then you can set that identifier as an attribute on all the batch-created Player objects (preferrably as a foreign-key), and retrieve the batch created objects by filtering on the id of the BatchOperation. After this, you can iterate through all the created Player objects, and create a list of Card objects for your batch create operation for the cards.

Example:

```
game = Game.objects.get(...)
batchoperation = BatchOperation.objects.create_synchronous(
    context_object=game)

players = []
for index in range(1000):
    player = Player(
        game=game,
        name='player{}'.format(index),
        batchoperation=batchoperation)
```

(continues on next page)

(continued from previous page)

```
    players.append(player)
Player.objects.bulk_create(players)
created_players = Player.objects.filter(batchoperation=batchoperation)

cards = []
available_cards = [...]  # A list of available card IDs
for player in created_players:
    card = Card(
        player=player,
        cardid=random.choice(available_cards)
    )
    cards.append(card)
Card.objects.bulk_create(cards)
batchoperation.finish()
```

As you can see in the example above, instead of having to perform 2000 queries (one for each player, and one for each card), we now only need 5 queries no matter how many players we have (or a few more on database servers that can not bulk create 1000 items at a time).

### 3.2.7 Data model API

## 3.3 *ievv_customsql* — Framework for adding custom SQL to Django apps

The intention of this module is to make it easier to add and update custom SQL in a Django app. We do this by providing a registry of all custom SQL, and a management command to add and update custom SQL and any refresh data that is maitained by triggers.

### 3.3.1 Configuration

Add the following to your `INSTALLED_APPS`-setting:

```
'ievv_opensource.ievv_customsql'
```

### 3.3.2 Add custom SQL to your app

#### Create the class containing your custom SQL

First you need to create a subclass of `AbstractCustomSql`.

Lets say you have a Person model with name and description. You want to maintain a fulltext search vector to efficiently search the person model. You would then create a subclass of AbstractCustomSql that looks something like this:

```
from ievv_opensource.ievv_customsql import customsql_registry


class PersonCustomSql(customsql_registry.AbstractCustomSql):
    def initialize(self):
        self.execute_sql("""
```

(continues on next page)

(continued from previous page)

```
            -- Add search_vector column to the Person model
            ALTER TABLE myapp_person DROP COLUMN IF EXISTS search_vector;
            ALTER TABLE myapp_person ADD COLUMN search_vector tsvector;

            -- Function used to create the search_vector value both in the trigger,
            -- and in the UPDATE statement (in recreate_data()).
            CREATE OR REPLACE FUNCTION myapp_person_get_search_vector_value(param_
↪table myapp_person)
            RETURNS tsvector AS $$
            BEGIN
                RETURN setweight(to_tsvector(param_table.name), 'A') ||
                    setweight(to_tsvector(param_table.description), 'C');
            END
            $$ LANGUAGE plpgsql;

            -- Trigger function called on insert or update to keep the search_vector␣
↪column
            -- in sync.
            CREATE OR REPLACE FUNCTION myapp_person_set_search_vector() RETURNS␣
↪trigger AS $$
            BEGIN
                NEW.search_vector := myapp_person_get_search_vector_value(NEW);
              return NEW;
            END
            $$ LANGUAGE plpgsql;

            DROP TRIGGER IF EXISTS myapp_person_set_search_vector_trigger ON myapp_
↪person;
            CREATE TRIGGER myapp_person_set_search_vector_trigger BEFORE INSERT OR␣
↪UPDATE
                ON myapp_person FOR EACH ROW
                EXECUTE PROCEDURE myapp_person_set_search_vector();
        """)

    def recreate_data(self):
        self.execute_sql("""
            UPDATE myapp_person SET
                search_vector = myapp_person_get_search_vector_value(myapp_person);
        """)
```

You can put this code anywhere in your app, but the recommended location is to put it in a file named `customsql.py` in the root of your app.

### Add your custom SQL to the registry

Next, you need to register your PersonCustomSql class in the registry. Create an AppConfig for your app with the following code:

```
from django.apps import AppConfig

from ievv_opensource.ievv_customsql import customsql_registry
from myproject.myapp.customsqldemo.customsql import PersonCustomSql
```

(continues on next page)

(continued from previous page)

```
class CustomSqlDemoAppConfig(AppConfig):
    name = 'myproject.myapp'
    verbose_name = "My APP"

    def ready(self):
        registry = customsql_registry.Registry.get_instance()
        registry.add('myapp', PersonCustomSql)
```

### Using your custom SQL

During development and as part of production releases, you use the `ievvtasks_customsql` command to update your custom SQL. Run the following to execute both:

- `initialize()`
- `recreate_data()`

for all the custom SQL classes in the registry:

```
$ python manage.py ievvtasks_customsql -i -r
```

Since this is an ievvtasks command, you can also run it as:

```
$ ievv customsql -i -r
```

### Writing tests using your custom SQL

The custom SQL is not added automatically, so you need to use it explicitly in your tests. You have three choices:

1. Call `PersonCustomSql().initialize()` in your setUp() method, or in your test method(s). You will probably also want to call `PersonCustomSql().recreate_data()` when required. This is **normally the recommented method**, since it provides the largest amount of control. See `AbstractCustomSql.initialize()` and `AbstractCustomSql.recreate_data()` for more info.
2. Call `ievv_customsql.Registry.get_instance().run_all_in_app('myapp')`. This may be useful to test views and other code that require all the custom SQL in your app. See `Registry.run_all_in_app()` for more info.
3. Call `ievv_customsql.Registry.get_instance().run_all()`. This is **not recommended** because it runs SQL from ALL apps in `INSTALLED_APPS`. See See `Registry.run_all()` for more info.

Example of using option (1) to create a TestCase:

```
class TestPersonCustomSql(test.TestCase):
    def test_add_person_and_search(self):
        PersonCustomSql().initialize()
        jack = mommy.make('myapp.Person', name='Jack The Man', description='Also␣
↪called john by some.')
        mommy.make('myapp.Person', name='NoMatch Man')
        john = mommy.make('myapp.Person', name='John Peterson', description='Hello␣
↪world')

        tsquery = 'john'
        queryset = Person.objects.extra(
            select={
```

(continues on next page)

(continued from previous page)

```
            'rank': 'ts_rank_cd(search_vector, to_tsquery(%s))',
        },
        select_params=[tsquery],
        where=['search_vector @@ to_tsquery(%s)'],
        params=[tsquery],
        order_by=['-rank']
    )
    self.assertEqual([john, jack], list(queryset))
```

#### Demo

See `ievv_opensource/demo/customsqldemo/` for a full demo of everything explained above.

### 3.3.3 API

#### AbstractCustomSql

**class** `ievv_opensource.ievv_customsql.customsql_registry.`**`AbstractCustomSql`**(*appname=None*)

Defines custom SQL that can be executed by the `ievv_customsql` framework.

You typically override `initialize()` and use `execute_sql()` to add triggers and functions, and override `recreate_data()` to rebuild the data maintained by the triggers, but many other use-cases are also possible.

**Parameters appname** – Not required - it is added automatically by *`Registry`*, and used by `__str__`()` for easier debugging / prettier output.

#### Registry

**class** `ievv_opensource.ievv_customsql.customsql_registry.`**`Registry`**

Registry of *`AbstractCustomSql`* objects.

##### Examples

First, define a subclass of *`AbstractCustomSql`*.

Register the custom SQL class with the registry via an AppConfig for your Django app:

```
from django.apps import AppConfig
from ievv_opensource.ievv_customsql import customsql_registry
from myapp import customsql

class MyAppConfig(AppConfig):
    name = 'myapp'

    def ready(self):
        customsql_registry.Registry.get_instance().add(customsql.MyCustomSql)
```

See `ievv_opensource/demo/customsql/apps.py` for a complete demo.

Ensures there is only one instance created. Make sure to use super() in subclasses.

#### MockableRegistry

**class** ievv_opensource.ievv_customsql.customsql_registry.**MockableRegistry**

A non-singleton version of *Registry*. For tests.

Typical usage in a test:

```
from ievv_opensource.ievv_customsql import customsql_registry

class MockCustomSql(customsql_registry.AbstractCustomSql):
    # ...

mockregistry = customsql_registry.MockableRegistry()
mockregistry.add(MockCustomSql())

with mock.patch('ievv_opensource.ievv_customsql.customsql_registry.Registry.get_
↪instance',
                lambda: mockregistry):
    pass  # ... your code here ...
```

Ensures there is only one instance created. Make sure to use super() in subclasses.

## 3.4 *ievv_jsbase* — Javascript library with general purpose functionality

### 3.4.1 Javascript API docs

The API docs for the javascript library is available here: IEVV jsbase API.

## 3.5 *ievv_developemail* — Develop mail backend that lets you view mails in django admin

### 3.5.1 Setup

Add it to `INSTALLED_APPS` setting, and set the `EMAIL_BACKEND` setting (typically only in develop settings):

```
INSTALLED_APPS = [
    # ...
    'ievv_opensource.ievv_developemail',
]

EMAIL_BACKEND = 'ievv_opensource.ievv_developemail.email_backend.DevelopEmailBackend'
```

Migrate:

```
$ python manage.py migrate
```

You should now get new emails both logged to the terminal, and added as a DevelopEmail object in the database which you can browse in Django admin.

### 3.5.2 How it works

We have a custom email backend that sends emails to the DevelopEmail database model (and to log).

### 3.5.3 Management script for sending test emails

We provide the `ievv_developemail_send_testmail` management script for sending test emails. It can be useful just to check that emails are sent, but also for testing `cradmin_email` styling etc.

In its simplest form:

```
$ python manage.py ievv_developemail_send_testmail --to "test@example.com"
```

The same, but with an HTML message:

```
$ python manage.py ievv_developemail_send_testmail --to "test@example.com" --html
```

With a custom message instead of the default lorem ipsum message:

```
$ python manage.py ievv_developemail_send_testmail --to "test@example.com" --html --
↪message-html "Dette er <em>en test lizzm</em>"
```

Send using django_cradmin.apps.cradmin_email.emailutils.AbstractEmail:

```
$ python manage.py ievv_developemail_send_testmail --to "test@example.com" --html --
↪use-cradmin-email
.. or with custom message ..
$ python manage.py ievv_developemail_send_testmail --to "test@example.com" --html  --
↪use-cradmin-email --message-html "Dette er <em>en test lizzm</em>"
```

From email can be set too! Defaults to the DEFAULT_FROM_EMAIL setting:

```
$ python manage.py ievv_developemail_send_testmail --to "test@example.com" --from
↪"from@example.com"
```

## 3.6 *ievv_model_mommy_extras* — Add support for more fields types to model-mommy

Model-mommy does not support all the fields in Django. To remidy this, we provide the `ievv_opensource.ievv_model_mommy_extras`.

For now it only adds support for:

```
- django.contrib.postgres.fields.ranges.DateTimeRangeField
- django.contrib.postgres.fields.ranges.DateRangeField
```

### 3.6.1 Setup

To use this, you only need to add it to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
    # Other apps ...
    'ievv_opensource.ievv_model_mommy_extras'
]
```

**Note:** You only need this for testing, so if you have split out your test settings, you should only add it to `INSTALLED_APPS` there.

## 3.7 *ievv_restframework_helpers* — Helpers for working with Django rest framework

### 3.7.1 FullCleanModelSerializer

## 3.8 *ievv_sms* — SMS sending - multiple backends supported

### 3.8.1 Requirements

```
$ pip install gsm0338
```

### 3.8.2 Setup

Add it to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
    # Other apps ...
    'ievv_opensource.ievv_sms'
]
```

For development, you probably also want to use the `ievv_opensource.ievv_sms.backends.debugprint.Backend` backend. You configure that with the following setting:

```
IEVV_SMS_DEFAULT_BACKEND_ID = 'debugprint'
```

### 3.8.3 Sending SMS

Send an SMS using the default backend with:

```
from ievv_opensource.ievv_sms.sms_registry import send_sms
send_sms(phone_number='12345678', message='This is a test')
```

Send using another backend using the `backend_id` argument:

```
from ievv_opensource.ievv_sms.sms_registry import send_sms
send_sms(phone_number='12345678', message='This is a test',
         backend_id='some_backend_id')
```

See `ievv_opensource.ievv_sms.sms_registry.send_sms()` for more details.

## 3.8.4 Creating a custom backend

See the example in `AbstractSmsBackend`.

## 3.8.5 Specifying the default backend

Just set the backend ID (see `get_backend_id()`) of a backend in the `IEVV_SMS_DEFAULT_BACKEND_ID` setting.

## 3.8.6 Core API

### send_sms()

`ievv_opensource.ievv_sms.sms_registry.`**`send_sms`**(*phone_number*, *message*, *backend_id=None*, ***kwargs*)

Send SMS message.

Just a shortcut for `Registry.send()` (`Registry.get_instance().send(...)`).

**Parameters**

- **`phone_number`** (`str`) – The phone number to send the message to.
- **`message`** (`str`) – The message to send.
- **`backend_id`** (`str`) – The ID of the backend to use for sending. If this is `None`, we use the default backend (see `get_default_backend_id()`).
- **`**kwargs`** – Extra kwargs for the `AbstractSmsBackend` constructor.

**Returns** An instance of a subclass of `AbstractSmsBackend`.

**Return type** *AbstractSmsBackend*

### AbstractSmsBackend

**`class`** `ievv_opensource.ievv_sms.sms_registry.`**`AbstractSmsBackend`**(*phone_number*, *message*, *send_as=None*, ***kwargs*)

Base class for SMS backends.

An instance of this class is created each time an SMS is created.

This means that you can store temporary information related to building the SMS on `self`.

Example (simple print SMS backend):

```
class PrintBackend(sms_registry.AbstractSmsBackend):
    @classmethod
    def get_backend_id(cls):
        return 'print'

    def send(self):
        print(
            'Phone number: {phone_number}. '
            'Message: {message}'.format(
                phone_number=self.cleaned_phone_number,
                message=self.cleaned_message
            )
        )
```

To use the PrintBackend, add it to the registry via an AppConfig for your Django app:

```
from django.apps import AppConfig

class MyAppConfig(AppConfig):
    name = 'myapp'

    def ready(self):
        from ievv_opensource.ievv_sms import sms_registry
        from myapp import sms_backends
        sms_registry.Registry.get_instance().add(sms_backends.PrintBackend)
```

Now you can use the backend to send an SMS:

```
from ievv_opensource.ievv_sms import sms_registry
sms_registry.Registry.get_instance().send(
    phone_number='12345678',
    message='This is a test',
    backend_id='print')
```

You can also set the backend as the default backend for SMS sending by adding `IEVV_SMS_DEFAULT_BACKEND_ID = 'print'` to your django settings. With this setting you can call send() without the `backend_id` argument, and the SMS will be sent with the print backend.

All the arguments are forwarded from `Registry.send()` / `Registry.make_backend_instance()`.

**Parameters**

- **phone_number** (*str*) – The phone number to send the message to.
- **message** (*str*) – The message to send.
- ****kwargs** – Extra kwargs. Both for future proofing, and to make it possible for backends to support extra kwargs.

#### Registry

**class** `ievv_opensource.ievv_sms.sms_registry.`**`Registry`**
Registry of *`AbstractSmsBackend`* objects.

Ensures there is only one instance created. Make sure to use super() in subclasses.

### 3.8.7 Backends

#### Debug/develop backends

#### PSWIN backend

#### DebugSmsMessage backend

## 3.9 *ievv_i18n_url* — i18n support for various ways of routing i18n views

Hosting a page in different languages can only be achived in a finite number of ways. These are the most common:

- The default django way where the selected language is stored in session.
- Some prefix to the URL path like `/<languagecode>/path/to/my/view`, `/l/<languagecode>/path/to/my/view`, etc. typically with the default translation at `/path/to/my/view`
- Based on domain name. E.g. *myapp.com* for english, *myapp.no* for norwegian, etc., or *<languagecode>.myapp.com*.
- Combinations of prefix and separate domains (e.g.: you have custom domains for the most important languages, and just hos the rest on the “main” domain with prefix in the URL path).

*ievv_i18n_url* supports all these and more through:

- Swappable URL handlers where all the logic happens (this is where *these and more* comes in since you can write your own handler class).
- Template tags that use the URL handlers.
- A library that provides a superset of the functionality of what the template tags provide.
- Middleware that uses the handler to handle whatever you write a handler for.

Things this handle that the built-in locale URL support in Django does not handle:

- Different default/fallback language per domain.
- Support for domains to define the languagecode.
- Locale aware URL reversing (e.g.: link to a page in a specific language).

### 3.9.1 Setup

Add it to settings:

```
INSTALLED_APPS = [
   # ...
   'ievv_opensource.ievv_i18n_url',
]

MIDDLEWARE = [
   # ...
   # Instead of django.middleware.locale.LocaleMiddleware, or any other␣
↪LocaleMiddleware,
   'ievv_opensource.ievv_i18n_url.middleware.LocaleMiddleware'
]

# Fallback base URL - used everywhere that we can not determine the "current" domain␣
↪(management scripts that does not specify a base url etc).
IEVV_I18N_URL_FALLBACK_BASE_URL = https://mydomain.com/

# The handler class - see further down for the available handler classes
IEVV_I18N_URL_HANDLER = 'ievv_opensource.ievv_i18n_url.handlers.UrlpathPrefixHandler'
```

In your urls.py, wrap your URLs with i18n_patterns():

```
from ievv_opensource.ievv_i18n_url import i18n_url_utils

# Instead of:
# urlpatterns = [
#     url(r'^my/view$', myview),
#     url(r'^another/view$', anotherview),
# ]

# Wrap it in i18n_urlpatterns like this:
urlpatterns = i18n_url_utils.i18n_patterns(
   url(r'^my/view$', myview),
   url(r'^another/view$', anotherview),
)
```

**Warning:** You should not have ANY urls that are not wrapped with i18n_patterns - this will just make the middleware and handlers work in an undeterministic manner. If you want to exclude URLs from being translated, create a subclass of your handler and override *ievv_opensource.ievv_i18n_url.handlers.AbstractHandler.is_translatable_urlpath()*.

### 3.9.2 How it works

A very high level overview of how it works is that we have swappable handlers that decide what the current language is based on information they get from the request and URL.

### The handlers

The handlers serve two puposes: - They have some classmehods that the LocaleMiddleware uses to set the current language. I.e.: The handlers

basically implement what the middleware should do.

- They have a lot of helper methods to make it easy to work with locale aware URL schemes, and some methods to help generalize locale handling (such as labels and icons for languages).

### The LocaleMiddleware

The middleware, `ievv_opensource.ievv_i18n_url.middleware.LocaleMiddleware`, is very simple. It just calls `ievv_opensource.ievv_i18n_url.handlers.AbstractHandler.activate_languagecode_from_request()` on the configured handler, and then it sets some information about the detected language on the request:

- `request.LANGUAGE_CODE`: Set to the languagecode we activated django translations for.
- `request.session['LANGUAGE_CODE']`: Same value as request.LANGUAGE_CODE - this is just set for compatibility with code that is written explicitly for session based language selection.
- `request.IEVV_I18N_URL_DEFAULT_LANGUAGE_CODE`: Set to the detected default language code.
- `request.IEVV_I18N_URL_ACTIVE_LANGUAGE_CODE`: Set to the active languagecode. This is normally the same as request.LANGUAGE_CODE, but if `ievv_opensource.ievv_i18n_url.handlers.AbstractHandler.get_translation_to_activate_for_languagecode()` is overridden on the handler they may differ. E.g.: Multiple languages may use the same Django translation.

### Url pattern handling

The `ievv_opensource.ievv_i18n_url.i18n_url_utils.i18n_patterns()` function that you wrap around all your URL patterns uses a custom URL resolver that simply gets the languagecode that the middleware set, and ignores any URL path prefix that the handler has said that we have for the current language. E.g.: we use the same "hack/smart solution" as the built in i18n url routing handler in Django.

## 3.9.3 A warning about session based translations

We provide support for session based translations, BUT it is not fully supported. Things like generating an URL for a specific languagecode, or finding the translation for the current URL in a different languagecode is NOT possible to do in a safe manner.

The reason for this limitation is that ALL translations live at the same URL, and the only way to safely change the languagecode is to HTTP POST a change to the languagecode. You may want to implement a handler that ignores this and actually provides full support with session based translations. This will require some kind of redirect view that changes the session language from a HTTP GET request.

Our recommendation is to NOT use session based translations, and instead use a URL path or domain based translation handler.

### 3.9.4 Utilities

#### i18n_url_utils

`ievv_opensource.ievv_i18n_url.i18n_url_utils.`**`get_handler_class`**`()`

Get the configured *ievv_i18n_url* handler class.

E.g. import the handler class from the class path configured in the `IEVV_I18N_URL_HANDLER` setting.

**Returns** A handler class.

**Return type** ievv_opensource.ievv_i18n_url.handlers.abstract_handler.AbstractHandler

`ievv_opensource.ievv_i18n_url.i18n_url_utils.`**`i18n_reverse`**`(`*viewname*, *url-conf=None*, *args=None*, *kwargs=None*, *cur-rent_app=None*, *lan-guagecode=None*`)`

Serves kind of the same use case as the `django.urls.reverse` function, but with i18n URL support, AND this function returns an absolute URL.

The reason why it returns absolute URL is because i18n URLs may be based on domains, not just URL paths.

NOTE: Session based *ievv_i18n_url* handlers will ignore the languagecode argument and just return the URL for the default translation. This is because all their translations live at the same URL. See the *A warning about session based translations* in the docs for more details.

**Parameters**

- **`viewname`** – See the docs for `django.urls.reverse`.
- **`urlconf`** – See the docs for `django.urls.reverse`. Defaults to None.
- **`args`** – See the docs for `django.urls.reverse`. Defaults to None.
- **`kwargs`** – See the docs for `django.urls.reverse`. Defaults to None.
- **`current_app`** – See the docs for `django.urls.reverse`. Defaults to None.
- **`languagecode`** (*`str, optional`*) – The languagecode to reverse the URL in. Defaults to None, which means we reverse the URL in the current languagecode.

**Returns** An URL.

**Return type** str

`ievv_opensource.ievv_i18n_url.i18n_url_utils.`**`transform_url_to_languagecode`**`(`*url*, *to_languagecode*, *from_languagecode=N*

Takes an URL, and finds the URL to the same content within a different languagecode.

NOTE: Session based *ievv_i18n_url* handlers will ignore the languagecode argument and just return provided url. This is because all their translations live at the same URL. See the *A warning about session based translations* in the docs for more details.

**Parameters**

- **`url`** (*`str`*) – The URL to transform.
- **`to_languagecode`** (*`str`*) – The languagecode to transform the url into.
- **`from_languagecode`** (*`str`*) – The languagecode to transform the url from.

**Returns**

**The transformed URL. If from_languagecode and to_languagecode is the same,** the provided `url` is returned unchanged.

**Return type** str

`ievv_opensource.ievv_i18n_url.i18n_url_utils.`**`i18n_patterns`**(**urls*, *include_redirect_view=True*)

Adds the language code prefix to every URL pattern within this function. This may only be used in the root URLconf, not in an included URLconf.

**`class`** `ievv_opensource.ievv_i18n_url.i18n_url_utils.`**`I18nRegexURLResolver`**(*urlconf_name*, *default_kwargs=None*, *app_name=None*, *namespace=None*, *prefix_default_language=False*

A URL resolver that always matches the active language code as URL prefix.

Rather than taking a regex argument, we just override the `regex` function to always return the active language-code as regex.

## base_url

**`class`** `ievv_opensource.ievv_i18n_url.base_url.`**`BaseUrl`**(*url_or_urllib_parseresult*)

Defines ievv_i18n_url base url.

The base URL is the URL to the root of the domain e.g: https://example.com, http://www.example.com:8080, etc. (NOT https://example.com/my/path, https://example.com/en, etc.).

The constructor can take any valid absolute URL, or None (in which case it falls back on the IEVV_I18N_URL_FALLBACK_BASE_URL setting), but all the methods and properties work on a `urllib.parse.ParseResult` that does not have any of the URL parts after

**`parsed_url`**

Get the parsed URL.

The returned URL only has scheme+domain+port (e.g.: scheme+netloc).

**Returns** A parsed URL on the same format as urllib.parse.urlparse returns.

**Return type** urllib.parse.ParseResult

**`property scheme`**

Shortcut for parsed_url.scheme.

See *`parsed_url`*.

**Returns** The url scheme (e.g.: https, http, …)

**Return type** str

**`property netloc`**

Shortcut for parsed_url.netloc.

See *`parsed_url`*.

**Returns** The url netloc (e.g.: www.example.com:9090)

**Return type** str

**property hostname**

Shortcut for parsed_url.hostname.

See *parsed_url*.

**Returns** The url hostname (e.g.: www.example.com)

**Return type** str

**property port**

Shortcut for parsed_url.port.

See *parsed_url*.

**Returns** The url port (e.g.: 8080, 80, . . . )

**Return type** str

**build_absolute_url** (*path_or_url*)

Build absolute URL from the provided `path_or_url`.

If the provided `path_or_url` is an URL, it is returned unchanged. If the provided `path_or_url` is a URL path, it is appended to the base url.

**Parameters** **path_or_url** (*str*) – URL path or an URL.

**Returns** [description]

**Return type** [type]

### i18n_url_settings

`ievv_opensource.ievv_i18n_url.i18n_url_settings.`**get_fallback_base_url_setting**()

Get the IEVV_I18N_URL_FALLBACK_BASE_URL.

**Raises** **Exception** – With a nice description if the setting is missing.

**Returns** The value of the setting

**Return type** str

### active_i18n_url_translation

`ievv_opensource.ievv_i18n_url.active_i18n_url_translation.`**get_default_languagecode**()

Get the default language code activated within the current thread.

I.e.: This returns the default languagecode that the *ievv_opensource.ievv_i18n_url.middleware.LocaleMiddleware* sets as default.

If this is called without using the middleware, or in management scripts etc. where the middleware is not applied, we fall back on `settings.LANGUAGE_CODE`.

**Returns** The default languagecode for the current thread.

**Return type** str

`ievv_opensource.ievv_i18n_url.active_i18n_url_translation.`**set_default_languagecode**(*default_lan*

Used by *ievv_opensource.ievv_i18n_url.middleware.LocaleMiddleware* to set the default languagecode in the current thread.

> **Warning:** You will normally not want to use this, but it may be useful in management scripts along with calling `activate()`.

`ievv_opensource.ievv_i18n_url.active_i18n_url_translation.`**`get_active_languagecode`**`()`

Get the active language code activated within the current thread.

I.e.: This returns the active languagecode that the *`ievv_opensource.ievv_i18n_url.middleware.LocaleMiddleware`* sets as active. This may not be the same as the languagecode django.utils.translation.get_language() returns if the handler overrides `get_translation_to_activate_for_languagecode()`.

If this is called without using the middleware, or in management scripts etc. where the middleware is not applied, we fall back on `settings.LANGUAGE_CODE`.

**Returns** The active languagecode for the current thread.

**Return type** str

`ievv_opensource.ievv_i18n_url.active_i18n_url_translation.`**`set_active_languagecode`**`(`*active_langu*

Used by *`ievv_opensource.ievv_i18n_url.middleware.LocaleMiddleware`* to set the active languagecode in the current thread.

> **Warning:** You will normally not want to use this, but it may be useful in management scripts along with calling `activate()`.

`ievv_opensource.ievv_i18n_url.active_i18n_url_translation.`**`get_active_language_urlpath_pref`**

Get the active URL path prefix within the current thread.

I.e.: This returns the language url path prefix that the *`ievv_opensource.ievv_i18n_url.middleware.LocaleMiddleware`* sets as active.

If this is called without using the middleware, or in management scripts etc. where the middleware is not applied, we fall back on empty string.

**Returns** The URL path prefix for active language in the current thread.

**Return type** str

`ievv_opensource.ievv_i18n_url.active_i18n_url_translation.`**`set_active_language_urlpath_pref`**

Used by *`ievv_opensource.ievv_i18n_url.middleware.LocaleMiddleware`* to set the active language url prefix in the current thread.

> **Warning:** You will normally not want to use this, but it may be useful in management scripts along with calling `activate()`.

`ievv_opensource.ievv_i18n_url.active_i18n_url_translation.`**`get_active_base_url`**`()`

Get the default language code activated within the current thread.

I.e.: This returns the language url path prefix that the *`ievv_opensource.ievv_i18n_url.middleware.LocaleMiddleware`* sets as active.

If this is called without using the middleware, or in management scripts etc. where the middleware is not applied, we fall back on empty string.

**Returns** The URL path prefix for active language in the current thread.

**Return type** str

`ievv_opensource.ievv_i18n_url.active_i18n_url_translation.`**`set_active_base_url`**(*active_base_url*)

Used by `ievv_opensource.ievv_i18n_url.middleware.LocaleMiddleware` to set the active language url prefix in the current thread.

**Warning:** You will normally not want to use this, but it may be useful in management scripts along with calling `activate()`.

`ievv_opensource.ievv_i18n_url.active_i18n_url_translation.`**`activate`**(*active_languagecode*, *de-fault_languagecode*, *ac-tive_translation_languagecode=No* *ac-tive_base_url=None*, *ac-tive_language_urlpath_prefix=Non*

Activate a translation.

This works much like `django.utils.translation.activate()` (and it calls that function), but it stores all of the stuff *ievv_i18n_url* needs to function in the current thread context.

**Parameters**

- **`active_languagecode`** (*`str`*) – Language code to set as the active languagecode in the current thread.
- **`default_languagecode`** (*`str`*) – Default language code for the current thread.
- **`active_translation_languagecode`** (*`str`*) – Language code to set as the active translation in the current thread. I.e.: The languagecode we send to `django.utils.translation.activate()`. Defaults to `active_languagecode`.
- **`active_base_url`** (*`urllib.parse.ParseResult`*) – The active base URL (E.g.: https://example.com/). Defaults to settings.IEVV_I18N_URL_FALLBACK_BASE_URL. Can be provided as a urllib.parse.ParseResult or as a string.
- **`active_language_urlpath_prefix`** (*`str`*) – URL path prefix for the active language.

### 3.9.5 Template tags

`ievv_opensource.ievv_i18n_url.templatetags.ievv_i18n_url_tags.`**`transform_url_to_languagecode`**

Template tag for the ievv_i18n_utils.transform_url_to_languagecode function.

See `transform_url_to_languagecode()` for the available arguments, but do not provide the `request` argument - we get that from `context["request"]`.

**Returns** An url.

**Return type** str

### 3.9.6 handlers

#### UrlpathPrefixHandler

**class** `ievv_opensource.ievv_i18n_url.handlers.`**`UrlpathPrefixHandler`**

I18n url handler that matches languages based on a URL prefix.

#### DjangoSessionHandler

**class** `ievv_opensource.ievv_i18n_url.handlers.`**`DjangoSessionHandler`**

Django session based i18n url handler.

> **Warning:** Please read the *A warning about session based translations* in the docs before deciding to use this handler.

#### AbstractHandler

**class** `ievv_opensource.ievv_i18n_url.handlers.`**`AbstractHandler`**

Base class for *ievv_i18n_url* handlers.

**`is_default_languagecode`**(*languagecode*)

Is the provided languagecode the default language code?

Note that this may be per domain etc. depending on the handler class.

**Parameters** **`languagecode`** (*str*) – Language code.

**Returns** True if the provided languagecode is the default.

**Return type** bool

**property default_languagecode**

Get the default language code.

Note that this may be per domain etc. depending on the handler class.

Defaults to `settings.LANGUAGE_CODE`.

**Returns** Default languagecode.

**Return type** str

**property active_languagecode**

Get the active languagecode.

**Returns** The active languagecode.

**Return type** str

**property active_languagecode_or_none_if_default**

Get the active languagecode, but returns None if the active languagecode is the default languagecode.

**Returns** The active languagecode, or None if the active languagecode is the default languagecode.

**Return type** str

**property active_base_url**

Get the active base url.

**Returns** The active base url.

**Return type** str

**classmethod is_supported_languagecode**(*languagecode*)

Is the provided languagecode a supported languagecode?

**Parameters** **languagecode** (*str*) – Language code.

**Returns** True if the provided languagecode is supported.

**Return type** bool

**get_translated_label_for_languagecode**(*languagecode*)

Get the translated label for the languagecode (the name of the language) in the currently active language.

This defaults to the english name for the language fetched via `django.utils.translation.get_language_info()`.

This is typically used in subclasses that override *get_label_for_languagecode()* and change to translated labels by default.

**Parameters** **languagecode** (*str*) – Language code.

**Returns** Translated label for the languagecode.

**Return type** str

**get_untranslated_label_for_languagecode**(*languagecode*)

Get the untranslated label for the languagecode (the name of the language).

Should return a label for the languagecode in a commonly used language that most users of your site will understand.

This defaults to the english name for the language fetched via `django.utils.translation.get_language_info()`.

This is the what the default implementation of *get_label_for_languagecode()* uses.

**Parameters** **languagecode** (*str*) – Language code.

**Returns** Unstranslated label for the languagecode.

**Return type** str

**get_local_label_for_languagecode**(*languagecode*)

Get the local label for the languagecode (the name of the language in that language).

This defaults to the english name for the language fetched via `django.utils.translation.get_language_info()`.

This is typically used in subclasses that override *get_label_for_languagecode()* and change to labels in the native language by default.

**Parameters** **languagecode** (*str*) – Language code.

**Returns** Local label for the languagecode.

**Return type** str

**get_label_for_languagecode**(*languagecode*)

Get the label for the languagecode (the name of the language).

Defaults to using *get_local_label_for_languagecode()*. I.e.: We use the native/local translation of the language name as language label by default.

**Parameters** **languagecode** (*str*) – Language code.

**Returns** Label for the languagecode.

**Return type** str

**classmethod activate_languagecode_from_request**(*request*)

Activate the detected languagecode.

This is what *ievv_opensource.ievv_i18n_url.middleware.LocaleMiddleware* uses to process the request.

What this does:

- Builds a base_url using `request.build_absolute_uri('/')`.
- Finds the default language code using *detect_default_languagecode()*
- Finds the current language code using *detect_current_languagecode()*
- Handles fallback to default languagecode if the current languagecode is unsupported or the requested url path is not not translatable (using *is_supported_languagecode()* and *is_translatable_urlpath()*).
- Finds the language URL prefix *get_urlpath_prefix_for_languagecode()*.
- Activates the current language/translation using *ievv_opensource.ievv_i18n_url.active_i18n_url_translation.activate()*.

**Warning:** Do not override this method, and you should normally not call this method. I.e.: This is for the middleware.

**get_icon_cssclass_for_languagecode**(*languagecode*)

Get an icon CSS class for the language code.

This is typically implemented on a per app basis. I.e.: The application creates a subclass of one of the built-in handlers and override this to provide icons for their supported languages. This is provided as part of the handler to make it possible to generalize things like rendering language selects with icons.

This icon must be possible to use in HTML like this:

```
<span class="ICON_CSS_CLASS_HERE"></span>
```

**Parameters** **languagecode** (*str*) – Language code.

**Returns** Icon css class

**Return type** str

**get_icon_svg_image_url_for_languagecode**(*languagecode*)

Get an icon SVG image URL for the language code.

This is typically implemented on a per app basis. I.e.: The application creates a subclass of one of the built-in handlers and override this to provide icons for their supported languages. This is provided as part of the handler to make it possible to generalize things like rendering language selects with icons.

**Parameters** **languagecode** (*str*) – Language code.

**Returns** SVG image URL.

**Return type** str

**build_absolute_url** (*path*, *languagecode=None*)

Build absolute uri for the provided path within the provided languagecode.

MUST be implemented in subclasses.

---

**Note:** Session based handlers will ignore the languagecode argument and just return the URL for the default translation. This is because all their translations live at the same URL. See the *A warning about session based translations* in the docs for more details.

---

**Parameters**

- **path** (*str*) – The path (same format as HttpRequest.get_full_path() returns - e.g: `"/my/path?option1&option2"`)
- **languagecode** (*str, optional*) – The languagecode to build the URI for. Defaults to None, which means we build the URI within the current languagecode.

**build_urlpath** (*path*, *languagecode=None*)

Build URL path for the provided path within the provided languagecode.

This is a compatibility layer to make it possible to work with older code that considers a URL path as fully qualified. Most handlers will just do nothing with the path, or prepend a prefix, but some handlers (those that work with multiple domains), will use the `ievv_i18n_url_redirect_to_languagecode` redirect view here to return an URL that will redirect the user to the correct URL.

MUST be implemented in subclasses.

---

**Note:** Session based handlers will ignore the languagecode argument and just return the PATH for the default translation. This is because all their translations live at the same URL. See the *A warning about session based translations* in the docs for more details.

---

**Parameters**

- **path** (*str*) – The path (same format as HttpRequest.get_full_path() returns - e.g: `"/my/path?option1&option2"`)
- **languagecode** (*str, optional*) – The languagecode to build the path for. Defaults to None, which means we build the URI within the current languagecode.

**transform_url_to_languagecode** (*url*, *languagecode*)

Transform the provided url into the "same" url, but in the provided languagecode.

MUST be implemented in subclasses.

---

**Note:** This is not possible to implement in a safe manner for session based handlers (I.e.: multiple translation live at the same URI), so for these kind of handler this method will just return the provided *url*. See the *A warning about session based translations* in the docs for more details.

---

**Parameters**

- **url** (*str*) – The URL to transform.
- **languagecode** (*str*) – The languagecode to transform the URL into.

**classmethod get_translation_to_activate_for_languagecode**(*languagecode*)

Get the languagecode to actually activate for the provided languagecode.

Used by the middleware provided by *ievv_i18n_url* to activate the translation for the provided languagecode.

This is here to make it possible for applications to have languages that has their own domains or URLs, but show translations from another language code. E.g.: you may have content in a language, but be OK with translation strings from another language.

**Returns** The languagecode to activate with the django translation system for the provided `languagecode`. Defaults to the provided `languagecode`.

**Return type** str

**classmethod get_supported_languagecodes**()

Get supported language codes.

Defaults to the language codes in `settings.LANGUAGES`.

**Returns** A set of the supported language codes.

**Return type** set

**classmethod is_translatable_urlpath**(*base_url*, *path*)

Is the provided URL path translatable within the current base_url?

We default to consider the paths in settings.MEDIA_URL and settings.STATIC_URL as untranslatable.

If this returns `False`, the middleware will use the default translation when serving the path.

If you subclass this, you should not write code that parses the querystring part of the path. This will not work as intended (will not work the same everywhere) since the middleware does not call this with the querystring included, but other code using this may pass in the path with the querystring.

**Parameters**

- **base_url** (`ievv_opensource.ievv_i18n_url.base_url.BaseUrl`) – The base URL - see *`ievv_opensource.ievv_i18n_url.base_url.BaseUrl`* for more info.
- **path** (*`str`*) – An URL path (e.g: `/my/path`, `/my/path?a=1&b=2`, etc.)

**Returns** Is the provided URL translatable?

**Return type** bool

**classmethod detect_preferred_languagecode_for_user**(*user*)

Detect the preferred languagecode for the provided user.

This is normally NOT used by *`detect_current_languagecode()`* except for handlers like the session handler where the URL does not change based on language code. E.g.: It would be strange to serve a language based on user preferences when the URL explicitly says what language code we are serving.

This is mostly a convenience for management scripts, and a utility if you want to redirect users based on the preferred language code.

**Parameters** **user** – A user model object.

**Returns** The preferred language code for the provided user, or None. None means that the user has no preferred language code, or that the handler does not respect user preferences. Returns `None` by default.

**Return type** str

**classmethod detect_current_languagecode**(*base_url*, *request*)

Detect the current languagecode from the provided request and/or base_url.

Used by the middleware provided by *ievv_i18n_url* find the current language code and set it on the current request.

DO NOT USE THIS - it is for the middleware. Use `current_languagecode`.

MUST be overridden in subclasses.

If this returns None, it means that we should use the default languagecode. I.e.: Do not handle fallback to default languagecode when implementing this method in subclasses - just return None.

**Parameters**

- **base_url** (`ievv_opensource.ievv_i18n_url.base_url.BaseUrl`) – The base URL - see *`ievv_opensource.ievv_i18n_url.base_url.BaseUrl`* for more info.
- **request** (*`django.http.HttpRequest`*) – The HttpRequest

**Returns** The current languagecode or None (None means default languagecode is detected).

**Return type** str

**classmethod detect_default_languagecode**(*base_url*)

Detect the default languagecode for the provided base_url.

This is here so that handlers can override it to support different default languagecode per domain or perhaps more fancy stuff based on the provided url.

**Parameters base_url** (*`django.http.HttpRequest`*) – The base URL - e.g: https://example.com, http://www.example.com:8080, . . . (NOT https://example.com/my/path, https://example.com/en, . . . ).

**Returns** The default languagecode. Defaults to `settings.LANGUAGE_CODE`.

**Return type** str

**classmethod get_urlpath_prefix_for_languagecode**(*base_url*, *languagecode*)

Get the URL path prefix for the provided languagecode within the current base_url.

**Parameters**

- **base_url** (`ievv_opensource.ievv_i18n_url.base_url.BaseUrl`) – The base URL - see *`ievv_opensource.ievv_i18n_url.base_url.BaseUrl`* for more info.
- **languagecode** (*`str`*) – The language code to find the prefix for.

**Returns** The url path prefix. **Can not** start or end with `/`.

**Return type** str

### 3.9.7 Middleware

**class** `ievv_opensource.ievv_i18n_url.middleware.`**`LocaleMiddleware`**(*get_response=None*)

*ievv_js_i18n_url* locale middleware.

**`response_redirect_class`**

alias of `django.http.response.HttpResponseRedirect`

**`process_request`**(*request*)

Initializes the ievv_i18n_url handler from the request, and calls *`ievv_opensource.ievv_i18n_url.handlers.AbstractHandler.activate_languagecode_from_request()`*.

**Parameters** **`request`** – The request-object.

CHAPTER

# FOUR

# SETTINGS

## 4.1 Settings

### 4.1.1 ievvtasks_dump_db_as_json

#### IEVVTASKS_DUMPDATA_DIRECTORY

The directory where we put dumps created by the `ievvtasks_dump_db_as_json` management command. Typically, you put something like this in your develop settings:

```
THIS_DIR = os.path.dirname(__file__)

IEVVTASKS_DUMPDATA_DIRECTORY = os.path.join(THIS_DIR, 'dumps')
```

#### IEVVTASKS_DUMPDATA_ADD_EXCLUDES

Use this setting to add models and apps to exclude from the dumped json. We exclude:

- contenttypes
- auth.Permission
- sessions.Session

By default, and we exclude `thumbnail.KVStore` by default if `sorl.thumbnail` is in installed apps, and the `THUMBNAIL_KVSTORE` setting is configured to use the database (`sorl.thumbnail.kvstores.cached_db_kvstore.KVStore`).

Example:

```
IEVVTASKS_DUMPDATA_ADD_EXCLUDES = [
    'auth.Group',
    'myapp.MyModel',
]
```

#### IEVVTASKS_DUMPDATA_EXCLUDES

If you do not want to get the default excludes, you can use this instead of *IEVVTASKS_DUMPDATA_ADD_EXCLUDES* to specify exactly what to exclude.

### 4.1.2 ievvtasks_makemessages

#### IEVVTASKS_MAKEMESSAGES_LANGUAGE_CODES

The languages to build translations for. Example:

```
IEVVTASKS_MAKEMESSAGES_LANGUAGE_CODES = ['en', 'nb']
```

#### IEVVTASKS_MAKEMESSAGES_IGNORE

The patterns to ignore when making translations. Defaults to:

```
IEVVTASKS_MAKEMESSAGES_IGNORE = [
    'static/*'
]
```

#### IEVVTASKS_MAKEMESSAGES_DIRECTORIES

Directories to run makemessages and compilemessages in. Defaults to a list with the current working directory as the only item. The use case for this setting is if you have your translation files split over multiple apps or directories. Then you can use this setting to specify the parent directories of all your `locale` directories.

Lets say you have this structure:

```
myproject/
    usersapp/
        locale/
    untranslatedapp/
    themeapp/
        locale/
```

You can then configure `ievv makemessages` and `ievv compilemessages` to build the translations in `myproject.usersapp` and `myproject.themeapp` with the following setting:

```
IEVVTASKS_MAKEMESSAGES_DIRECTORIES = [
    'myproject/usersapp',
    'myproject/themeapp',
]
```

Just adding strings to `IEVVTASKS_MAKEMESSAGES_DIRECTORIES` is just a shortcut. You can add dicts instead:

```
IEVVTASKS_MAKEMESSAGES_DIRECTORIES = [
    {
        'directory': 'myproject/usersapp',
    },
    {
        'directory': 'myproject/themeapp',
        'python': True,  # Build python translations
```

(continues on next page)

(continued from previous page)

```
        'javascript': True,  # Build javascript translations
        # 'javascript_ignore': ['something/*'],  # Override IEVVTASKS_MAKEMESSAGES_
↪JAVASCRIPT_IGNORE for the directory
        # 'python_ignore': ['something/*'],  # Override IEVVTASKS_MAKEMESSAGES_IGNORE␣
↪for the directory
    }
]
```

### IEVVTASKS_MAKEMESSAGES_BUILD_JAVASCRIPT_TRANSLATIONS

Set this to `True` if you want to built translations for javascript code. Defaults to `False`.

### IEVVTASKS_MAKEMESSAGES_JAVASCRIPT_IGNORE

The patterns to ignore when making javascript translations. Defaults to:

```
IEVVTASKS_MAKEMESSAGES_JAVASCRIPT_IGNORE = [
    'node_modules/*',
    'bower_components/*',
    'not_for_deploy/*',
]
```

### IEVVTASKS_MAKEMESSAGES_PRE_MANAGEMENT_COMMANDS

Iterable of managemement commands to run before running makemessages. Example:

```
IEVVTASKS_MAKEMESSAGES_PRE_MANAGEMENT_COMMANDS = [
    {
        'name': 'ievvtasks_buildstatic',
        'options': {
            'includegroups': ['i18n']
        }
    }
]
```

Defaults to empty list.

The items in the iterable can be one of:

- A string with the name of a management command (for commands without any arguments or options).
- A dict with `name`, `args`, and `options` keys. The `name` key is required, but `args` and `options` are optional. `args` and `options` is just forwarded to `django.core.management.call_command`.

#### IEVVTASKS_MAKEMESSAGES_EXTENSIONS

Extensions to look for strings marked for translations in normal python/django code (for the `django` –domain for makemessages).

Defaults to `['py', 'html', 'txt']`.

#### IEVVTASKS_MAKEMESSAGES_JAVASCRIPT_EXTENSIONS

Extensions to look for strings marked for translations in javascript code (for the `djangojs` –domain for makemessages).

Defaults to `['js']`.

### 4.1.3 ievvtasks_docs

#### IEVVTASKS_DOCS_DIRECTORY

The directory where your sphinx docs resides (the directory where you have your sphinx `conf.py`). Defaults to `not_for_deploy/docs/`.

#### IEVVTASKS_DOCS_BUILD_DIRECTORY

The directory where your sphinx docs should be built. Defaults to `not_for_deploy/docs/_build`.

### 4.1.4 ievvtasks_recreate_devdb

#### IEVVTASKS_RECREATE_DEVDB_POST_MANAGEMENT_COMMANDS

Iterable of managemement commands to after creating/restoring and migrating the database in `ievv recreate_devdb`. Example:

```
IEVVTASKS_RECREATE_DEVDB_POST_MANAGEMENT_COMMANDS = [
    {
        'name': 'createsuperuser',
        'args': ['test@example.com'],
        'options': {'verbosity': 3}
    },
    'ievvtasks_set_all_passwords_to_test',
]
```

The items in the iterable can be one of:

- A string with the name of a management command (for commands without any arguments or options).
- A dict with `name`, `args`, and `options` keys. The `name` key is required, but `args` and `options` are optional. `args` and `options` is just forwarded to `django.core.management.call_command`.

### 4.1.5 ievv_tagframework

#### IEVV_TAGFRAMEWORK_TAGTYPE_CHOICES

The legal values for *ievv_opensource.ievv_tagframework.models.Tag.tagtype*.

Example:

```
IEVV_TAGFRAMEWORK_TAGTYPE_CHOICES = [
    ('', ''),
    ('mytype', 'My tag type'),
]
```

### 4.1.6 ievv devrun

#### IEVVTASKS_DEVRUN_RUNNABLES

Dict mapping `ievv devrun` target names to `ievv_opensource.utils.ievvdevrun.config.RunnableThreadList` objects. Must contain the `"default"` key.

Documented in *ievv devrun — All your development servers in one command*.

### 4.1.7 ievv_batchframework

#### IEVV_BATCHFRAMEWORK_ALWAYS_SYNCRONOUS

If this is `True`, all actions will be executed syncronously. Read more about this in ievv_batchframework_develop_asyncronous.

### 4.1.8 ievv_sms

#### IEVV_SMS_DEFAULT_BACKEND_ID

The default backend ID to use for SMS sending. Example:

```
IEVV_SMS_DEFAULT_BACKEND_ID = 'debugprint'
```

### 4.1.9 ievv_db

. . . setting:: IEVV_DB_POSTGRES_COLLATION

### IEVV_DB_POSTGRES_COLLATION

The collation to use for Django order_by that `ievv_opensource.ievv_db.postgres.collate_utils.collate()` uses as fallback if not passed as a parameter. Example:

Example setting the default collation in settings:

```
IEVV_DB_POSTGRES_COLLATION = 'en_US.UTF-8'
```

**Example usage in query::** from ievv_opensource.ievv_db.postgres.collate_util import collate

# Ascending order ExampleModel.objects.order_by(collate('some_field'))

# Descending order ExampleModel.objects.order_by(collate('-some_field'))

## 4.1.10 utils

### IEVV_SLUGIFY_CHARACTER_REPLACE_MAP

Custom character replacement map for the `ievv_slugify` function

### IEVV_COLORIZE_USE_COLORS

Colorize output from `ievv_opensource.utils.ievv_colorize.colorize()`? Defaults to `True`.

### IEVV_VALID_REDIRECT_URL_REGEX

Valid redirect URLs for *utils.validate_redirect_url — Validate redirect urls*.

Defaults to `^/.*$`, which means that only urls without domain is allowed by default.

Example for only allowing redirect urls that does not contain a domain, or redirect urls within the example.com domain:

```
IEVV_VALID_REDIRECT_URL_REGEX = r'^(https?://example\.com/|/).*$'
```

**Warning:** Do not use `^https://example\.com.*$` (no / after the domain). This can potentially lead to attacks using subdomains. `^https://example\.com.*$` would allow `example.com.spamdomain.io/something`, but `^https://example\.com/.*$` would not allow this.

CHAPTER

FIVE

# THE IEVV COMMAND

## 5.1 The `ievv` command

The `ievv` command does two things:

1. It avoids having to write `python manange.py appressotaks_something` and lets you write `ievv something` istead.
2. It provides commands that are not management commands, such as the commands for building docs and creating new projects.

When we add the command for initializing a new project, the ievv command will typically be installed globally instead of as a requirement of each project.

You find the source code for the command in `ievv_opensource/ievvtasks_common/cli.py`.

Some of the commands has required settings. See *Settings*.

## 5.2 *ievv buildstatic* — A static file builder (less, coffee, . . . )

The `ievv buildstatic` command is a fairly full featured general purpose static asset builder that solves the same general tasks as Grunt and Gulp, but in a very Django friendly and pythonic way.

You extend the system with object oriented python programming, and you configure the system using python classes.

### 5.2.1 Getting started

First of all, make sure you have the following in your `INSTALLED_APPS` setting:

```
'ievv_opensource.ievvtasks_common',
```

`ievv buildstatic` assumes you have the sources for your static files in the `staticsources/<appname>/` directory within each Django app.

For this example, we will assume your Django app is named `myapp`, and that it is located in `myproject/myapp/`.

First you need to create the staticsources directory:

```
$ mkdir -p myproject/myapp/staticsource/myapp/
```

### Setup building of LESS files

To kick things off, we will configure building of LESS sources into CSS. Create `myproject/myapp/staticsource/myapp/styles/theme.less`, and add some styles:

```
.myclass {
    color: red;
}
```

Now we need to add configurations to `settings.py` for building this less file. Add the following Django setting:

```
from ievv_opensource.utils import ievvbuildstatic
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='myapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.lessbuild.Plugin(sourcefile='theme.less'),
        ]
    ),
)
```

Now run the following command in your terminal:

```
$ ievv buildstatic
```

This should create `myproject/myapp/static/myapp/1.0.0/styles/theme.css`.

### Add media files

You will probably need some media files for your styles (fonts, images, ect.). First, put an image in `myproject/myapp/staticsource/myapp/media/images/`, then add the following to the `plugins` list in the `IEVVTASKS_BUILDSTATIC_APPS` Django setting:

```
ievvbuildstatic.mediacopy.Plugin()
```

Run:

```
$ ievv buildstatic
```

This should add your image to `myproject/myapp/static/myapp/1.0.0/media/images/`.

### Watch for changes

Re-running `ievv buildstatic` each time you make changes is tedious. You can watch for changes using:

```
$ ievv buildstatic --watch
```

### 5.2.2 Advanced topics

#### Using multiple apps

Using multiple apps is easy. You just add another `ievv_opensource.utils.ievvbuildstatic.config.App` to the `IEVVTASKS_BUILDSTATIC_APPS` setting:

```
from ievv_opensource.utils import ievvbuildstatic
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='myapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.lessbuild.Plugin(sourcefile='theme.less'),
        ]
    ),
    ievvbuildstatic.config.App(
        appname='anotherapp',
        version='2.3.4',
        plugins=[
            ievvbuildstatic.lessbuild.Plugin(sourcefile='theme.less'),
        ]
    ),
)
```

### 5.2.3 NPM packages and ievv buildstatic

#### How ievv buildstatic interracts with package.json

Many of the ievv buildstatic plugins install their own npm packages, so they will modify `package.json` if needed. Most plugins do not specify a specific version of a package, but they will not override your versions if you specify them in `package.json`.

#### Yarn or NPM?

ievv buildstatic uses `yarn` by default, but you can configure it to use `npm` with the following settings:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='myapp',
        version='1.0.0',
        installers_config={
            'npm': {
                'installer_class': ievvbuildstatic.installers.npm.NpmInstaller
            }
        },
        plugins=[
            # ...
        ]
    )
)
```

### 5.2.4 Working with npm packages guide

**Note:** You can create your `package.json` manually, using `npm init`/`yarn init`. If you do not create a package.json, ievv buildstatic will make one for you if you use any plugins that require npm packages.

You work with npm/yarn just like you would for any javascript project. The package.json file must be in:

```
<django appname>/staticsources/<django appname>/package.json
```

So you should be in `<django appname>/staticsources/<django appname>/` when running npm or yarn commands.

#### Example

1. Create the `staticsources/myapp/` directory in your django app. Replace myapp with your django app name.
2. Create `staticsources/myapp/scripts/javascript/app.js`.
3. Configure ievv buildstatic with the following in your django settings:

```
IEVVTASKS_BUILDSTATIC_APPS = ievvbuildstatic.config.Apps(
    ievvbuildstatic.config.App(
        appname='myapp',
        version='1.0.0',
        plugins=[
            ievvbuildstatic.browserify_jsbuild.Plugin(
                sourcefile='app.js',
                destinationfile='app.js',
            ),
        ]
    )
)
```

4. Run `ievv buildstatic` to build this app. This will create a `package.json` file.
5. Lets add momentjs as a dependency:

```
$ cd /path/to/myapp/staticsources/myapp/
$ yarn add momentjs
```

6. … and so on …

### 5.2.5 Plugins

#### Overview

| | |
|---|---|
| `pluginbase.Plugin`([group]) | Base class for all plugins in `ievvbuildstatic`. |
| `cssbuildbaseplugin.AbstractPlugin`([lint, …]) | Base class for builders that produce CSS. |
| `sassbuild.Plugin`(sourcefile[, sourcefolder, …]) | SASS build plugin — builds .scss files into css, and supports watching for changes. |

continues on next page

Table 1 – continued from previous page

| | |
|---|---|
| `lessbuild.Plugin`(sourcefile[, sourcefolder, ...]) | LESS build plugin — builds .less files into css, and supports watching for changes. |
| `mediacopy.Plugin`([sourcefolder, ...]) | Media copy plugin — copies media files from staticsources into the static directory, and supports watching for file changes. |
| `bowerinstall.Plugin`(packages, **kwargs) | Bower install plugin — installs bower packages. |
| `npminstall.Plugin`(packages, **kwargs) | NPM install plugin — installs NPM packages. |
| `browserify_jsbuild.Plugin`(sourcefile, ...[, ...]) | Browserify javascript bundler plugin. |
| `browserify_babelbuild.Plugin`([...]) | Browserify javascript babel build plugin. |
| `browserify_reactjsbuild.Plugin`([...]) | Browserify javascript babel build plugin. |
| `autosetup_jsdoc.Plugin`([group]) | Autosetup jsdoc config and npm script. |
| `autosetup_esdoc.Plugin`([title]) | Autosetup esdoc config and npm script. |
| `npmrun.Plugin`(script[, script_args]) | Run a script from package.json (I.E.: `npm run <something>`. |
| `npmrun_jsbuild.Plugin`([extra_import_paths, ...]) | Webpack builder plugin. |
| `run_jstests.Plugin`(**kwargs) | Run javascript tests by running `npm test` if the `package.json` has a `"test"` entry in `"scripts"`. |

#### Details

### 5.2.6 Apps and App

#### Overview

| | |
|---|---|
| `App`(appname, version, plugins[, ...]) | Configures how `ievv buildstatic` should build the static files for a Django app. |
| `Apps`(*apps, **kwargs) | Basically a list around `App` objects. |

#### Details

### 5.2.7 Utils

### 5.2.8 Installers

#### Overview

| | |
|---|---|
| `base.AbstractInstaller`(app) | Base class for installers. |
| `npm.NpmInstaller`(*args, **kwargs) | NPM installer. |
| `yarn.YarnInstaller`(*args, **kwargs) | Yarn installer. |

#### Details

### 5.2.9 Low level API

## 5.3 *ievv devrun* — All your development servers in one command

The `ievv devrun` command makes it easy to run (start/stop) all your development servers with a single command. It uses multithreading, background processes to run all your servers in a single blocking process that stops all the processes when `CTRL-C` is hit.

### 5.3.1 Getting started

First of all, make sure you have the following in your `INSTALLED_APPS` setting:

```
'ievv_opensource.ievvtasks_common',
'ievv_opensource.ievvtasks_development',
```

Next, you need to configure what to run when you run `ievv devrun`. You do this with the `IEVVTASKS_DEVRUN_RUNNABLES`-setting.

For the first example, we will use `ievv devrun` to just run the Django development server, just like `python manage.py runserver`. Add the following you your Django settings:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        ievvdevrun.runnables.django_runserver.RunnableThread()
    )
}
```

With this configured, you can run:

```
$ ievv devrun
```

to start the Django development. Hit `CTRL-C` to stop the server.

#### Using Django dbdev

For this example, we will setup *Django runserver* and *Django dbdev database server*:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        ievvdevrun.runnables.dbdev_runserver.RunnableThread(),
        ievvdevrun.runnables.django_runserver.RunnableThread()
    )
}
```

With this configured, you can run:

```
$ ievv devrun
```

to start both the Django development server and your django_dbdev database. Hit `CTRL-C` to stop both the servers.

### Multiple run configurations

You may already have guessed that you can add multiple configurations since we only add a `default`-key to `IEVVTASKS_DEVRUN_RUNNABLES`. To add multiple configurations, just add another key to the dict. For this example, we will add an `design` key that also runs `ievv buildstatic --watch`:

```
IEVVTASKS_DEVRUN_RUNNABLES = {
    'default': ievvdevrun.config.RunnableThreadList(
        # ... same as above ...
    ),
    'design': ievvdevrun.config.RunnableThreadList(
        ievvdevrun.runnables.dbdev_runserver.RunnableThread(),
        ievvdevrun.runnables.django_runserver.RunnableThread(),
        ievvdevrun.runnables.ievv_buildstatic.RunnableThread(),
    )
}
```

To run the `design`-set of runnables, use:

```
$ ievv devrun -n design
```

## 5.3.2 Adding ievv devrun as PyCharm run config

If you use PyCharm, you can do the following to add `ievv devrun` as a run target:

- Select `Run -> Edit configurations ....`
- **Select + `-> Python`.**
  - Give it a name (E.g.: `ievv devrun default`).
  - Check *Single instance*.
  - Check *Share* if you want to share it with your co-workers.
  - Select your `manage.py` as the script.
  - Set `ievvtasks_devrun -n default` as *Script parameters*.

If you want to create a target for the `design` config shown above, you just create another PyCharm run target with `ievvtasks_devrun -n design`.

## 5.3.3 Custom runnables

We bundle a fairly limited set of runnables, but adding one is really easy. Check out the docs for:

- `ievv_opensource.utils.ievvdevrun.runnables.base.ShellCommandRunnableThread`
- `ievv_opensource.utils.ievvdevrun.runnables.base.AbstractRunnableThread`

### 5.3.4 Bundled runnables

#### Overview

| | |
|---|---|
| `base.AbstractRunnableThread`([name]) | Abstract class for a thread that runs something for the `ievvdevrun` framework. |
| `base.ShellCommandRunnableThread`([name, . . . ]) | Makes it very easy to implement `AbstractRunnableThread` for system/shell commands. |
| `django_runserver.RunnableThread`([host, . . . ]) | Django runserver runnable thread. |
| `dbdev_runserver.RunnableThread`([name, . . . ]) | Django-DBdev run database runnable thread. |
| `redis_server.RunnableThread`([port, config_path]) | redis-server runnable thread. |

#### Details

### 5.3.5 Low level API

CHAPTER

SIX

# UTILITIES

## 6.1 *virtualenvutils* — Utilities for virtualenvs

`ievv_opensource.utils.virtualenvutils.`**`is_in_virtualenv`**`()`

Returns `True` if we are in a virtualenv.

`ievv_opensource.utils.virtualenvutils.`**`get_virtualenv_directory`**`()`

Get the root directory of the current virtualenv.

Raises OSError if not in a virtualenv.

`ievv_opensource.utils.virtualenvutils.`**`add_virtualenv_bin_directory_to_path`**`()`

Add *`get_virtualenv_directory()`* to `os.environ['PATH']`.

Why do we need this? This is used to work around limitations in how certain IDE's implement virtualenv support. They may add the virtualenv to PYTHONPATH, but to to PATH.

## 6.2 *utils.desktopnotifications* — Very simple desktop notification system

## 6.3 *utils.logmixin* — Colorized logging

## 6.4 *utils.shellcommandmixin* — Simplifies shell commands

## 6.5 *utils.singleton* — Singleton

**`class`** `ievv_opensource.utils.singleton.`**`Singleton`**

Implements the singleton pattern.

#### Example

Create a singleton class:

```
class MySingleton(Singleton):
    def __init__(self):
        super().__init__()
        self.value = 0

    def add(self):
        self.value += 1
```

Use the singleton:

```
MySingleton.get_instance().add()
MySingleton.get_instance().add()
print(MySingleton.get_instance().value)
```

Ensures there is only one instance created. Make sure to use super() in subclasses.

**classmethod get_instance**()
    Get an instance of the singleton.

## 6.6 *utils.class_registry_singleton* — Framework for swappable classes

### 6.6.1 What is this for?

If you are creating a library where you need to enable apps using the library to replace or add some classes with injection. There are two main use-cases:

1. You have a choice field, and you want to bind the choices to values backed by classes (for validation, etc.), AND you want apps using the library to be able to add more choices and/or replace the default choices.
2. You have some classes, such as adapters working with varying user models, and you need to be able to allow apps to inject their own implementations.

See *ievv_opensource.utils.class_registry_singleton.ClassRegistrySingleton* examples.

### 6.6.2 API docs

**exception** ievv_opensource.utils.class_registry_singleton.**DuplicateKeyError**(*registry*, *key*)
    Raised when adding a key already in a *ClassRegistrySingleton*

**class** ievv_opensource.utils.class_registry_singleton.**AbstractRegistryItem**
    Base class for *ClassRegistrySingleton* items.

**classmethod on_add_to_registry**(*registry*)
    Called automatically when the class is added to a *ClassRegistrySingleton*.

**classmethod on_remove_from_registry**(*registry*)
    Called automatically when the class is removed from a *ClassRegistrySingleton*.

**class** ievv_opensource.utils.class_registry_singleton.**RegistryItemWrapper**(*cls, default_instance_kwargs*)

Registry item wrapper.

When you add a *AbstractRegistryItem* to a *ClassRegistrySingleton*, it is stored as an instance of this class.

You can use a subclass of this class with your registry singleton by overriding *ClassRegistrySingleton.get_registry_item_wrapper_class()*. This enables you to store extra metadata along with your registry items, and provided extra helper methods.

**cls = None**
The *AbstractRegistryItem* class.

**default_instance_kwargs = None**
The default kwargs for instance created with *get_instance()*.

**make_instance_kwargs**(*kwargs*)
Used by *get_instance()* to merge `kwargs` with `default_instance_kwargs`.

**Returns** The full kwargs fo the instance.

**Return type** dict

**get_instance**(***kwargs*)
Get an instance of the *AbstractRegistryItem* class initialized with the provided `**kwargs`.

The provided `**kwargs` is merged with the `default_instance_kwargs`, with `**kwargs` overriding any keys also in `default_instance_kwargs`.

**Parameters** ****kwargs** – Kwargs for the class constructor.

**Returns** A class instance.

**Return type** *AbstractRegistryItem*

**class** ievv_opensource.utils.class_registry_singleton.**ClassRegistrySingleton**
Base class for class registry singletons - for having a singleton of swappable classes. Useful when creating complex libraries with classes that the apps using the libraries should be able to swap out with their own classes.

Example:

```
class AbstractMessageGenerator(class_registry_singleton.AbstractRegistryItem):
    def get_message(self):
        raise NotImplementedError()


class SimpleSadMessageGenerator(AbstractMessageGenerator):
    @classmethod
    def get_registry_key(cls):
        return 'sad'

    def get_message(self):
        return 'A sad message'

class ComplexSadMessageGenerator(AbstractMessageGenerator):
    @classmethod
    def get_registry_key(cls):
        return 'sad'

    def get_message(self):
```

(continues on next page)

(continued from previous page)

```
        return random.choice([
            'Most people are smart, but 60% of people think they are smart.',
            'Humanity will probably die off before we become a multi-planet␣
↪spiecies.',
            'We could feed everyone in the world - if we just bothered to share␣
↪resources.',
        ])

class SimpleHappyMessageGenerator(AbstractMessageGenerator):
    @classmethod
    def get_registry_key(cls):
        return 'happy'

    def get_message(self):
        return 'A happy message'

class ComplexHappyMessageGenerator(AbstractMessageGenerator):
    @classmethod
    def get_registry_key(cls):
        return 'happy'

    def get_message(self):
        return random.choice([
            'Almost every person you will ever meet are good people.',
            'You will very likely live to see people land on mars.',
            'Games are good now - just think how good they will be in 10 years!',
        ])

class MessageGeneratorSingleton(class_registry_singleton.ClassRegistrySingleton):
    ''''
    We never use ClassRegistrySingleton directly - we always create a subclass.␣
↪This is because
    of the nature of singletons. If you ise ClassRegistrySingleton directly as␣
↪your singleton,
    everything added to the registry would be in THE SAME singleton.
    ''''


class DefaultAppConfig(AppConfig):
    def ready(self):
        registry = MessageGeneratorSingleton.get_instance()
        registry.add(SimpleSadMessageGenerator)
        registry.add(SimpleHappyMessageGenerator)


class SomeCustomAppConfig(AppConfig):
    def ready(self):
        registry = MessageGeneratorSingleton.get_instance()
        registry.add_or_replace(ComplexSadMessageGenerator)
        registry.add_or_replace(ComplexHappyMessageGenerator)


# Using the singleton in code
registry = MessageGeneratorSingleton.get_instance()
print(registry.get_registry_item_instance('sad').get_message())
print(registry.get_registry_item_instance('happy').get_message())
```

Ensures there is only one instance created. Make sure to use super() in subclasses.

**`get_registry_item_wrapper_class`**`()`
Get the registry item wrapper class.

Defaults to *`RegistryItemWrapper`* which should work well for most use cases.

**`get`**`(`*key*, *fallback=None*`)`
Get a class (wrapper) stored in the registry by its key.

**Parameters**

- **`key`** (*`str`*) – A registry item class key.
- **`fallback`** – Fallback value of the `key` is not in the registry

Returns:

**Returns** You can use this to get the class or to get an instance of the class.

**Return type** *RegistryItemWrapper*

**`items`**`()`
Iterate over all the items in the registry yielding (key, RegistryItemWrapper) tuples.

**`iterwrappers`**`()`
Iterate over all the items in the registry yielding RegistryItemWrapper objects.

**`iterchoices`**`()`
Iterate over the the classes in the in the registry yielding two-value tuples where both values are the `get_registry_key()`.

Useful when rendering in a ChoiceField.

**Returns** An iterator that yields `(<key>, <key>)` tuples for each *`AbstractRegistryItem`* in the registry. The iterator is sorted by `get_registry_key()`.

**`add`**`(`*cls*, ***default_instance_kwargs*`)`
Add the provided `cls` to the registry. :param cls: A *`AbstractRegistryItem`* class (NOT AN OBJECT/INSTANCE). :param **default_instance_kwargs: Default instance kwargs.

**Raises** ***`DuplicateKeyError`*** – When a class with the same

:raises `get_registry_key()` is already: :raises in the registry.:

**`add_or_replace`**`(`*cls*, ***default_instance_kwargs*`)`
Insert the provided `cls` in registry. If another `cls` is already registered with the same `key`, this will be replaced.

**Parameters**

- **`cls`** – A *`AbstractRegistryItem`* class (NOT AN OBJECT/INSTANCE).
- **`**default_instance_kwargs`** – Default instance kwargs.

**`replace`**`(`*cls*, ***default_instance_kwargs*`)`
Replace the class currently in the registry with the same key as the provided `cls.get_registry_key()`.

**Parameters**

- **`cls`** – A *`AbstractRegistryItem`* class (NOT AN OBJECT/INSTANCE).
- **`**default_instance_kwargs`** – Default instance kwargs.

**Raises** **`KeyError`** – If the `cls.get_registry_key()` is NOT in the registry.

**remove**(*key*)
Remove the class provided `key` from the registry.

**Parameters** **key** (*str*) – A

:param `get_registry_key()`.:

**Raises** **KeyError** – When the `key` is not in the registry.

**remove_if_in_registry**(*key*)
Works just like *remove()*, but if the `key` is not in the registry this method just does nothing instead of raising KeyError.

**get_registry_item_instance**(*key*, ***kwargs*)
Just a shortcut for `singleton[key].get_instance(**kwargs)`.

## 6.7 *utils.text* — Text utils for simple text transformations

### 6.7.1 Settings for character replacement map

If you want to control the character replacements. You may add a custom map in your settings

In your **settings**, add:

```
IEVV_SLUGIFY_CHARACTER_REPLACE_MAP = {'<character>': '<replacement_character>'}
```

Where you replace your characters matching your need

## 6.8 *utils.ievv_colorize* — Colorized output for terminal stdout/stderr

## 6.9 *utils.choices_with_meta* — Object oriented choices for choice fields

## 6.10 *utils.validate_redirect_url* — Validate redirect urls

A very small set of utilities for validating a redirect URL. You will typically use this to secure URLs that you get as input from an insecure source, such as a `?next=<anything>` argument for a login view.

### 6.10.1 Configure

The only configuration is via the *IEVV_VALID_REDIRECT_URL_REGEX*, where you configure the valid redirect URLs.

### 6.10.2 Typical use case

Lets say you have a login view that supports `?next=<some url or path>`. This could lead to attacks for mining personal information if someone shares a link where the next URL points to an external domain. What you want is probably to allow redirects within your domain. To achieve this, you only need to set the *IEVV_VALID_REDIRECT_URL_REGEX* setting to match your domain and paths without a domain, and do something like this in your login view:

```
from ievv_opensource.utils import validate_redirect_url

class MyLoginView(...):
    # ... other code ...

    def get_success_url(self):
        nexturl = self.request.GET.get('next')
        if nexturl:
            validate_redirect_url.validate_url(nexturl)
            return nexturl
        else:
            # return some default
```

This will raise a ValidationError if the validation fails. You may cach this exception if you want to handle this with something other than a crash message in your server log.

### 6.10.3 Functions

## 6.11 *utils.testhelpers* — Testing utilities

### 6.11.1 Testing Django migrations

**Warning:** You can not test migrations if you have a `MIGRATION_MODULES` setting that disabled migrations. So make sure you remove that setting if you have it in your test settings.

#### Guide

Lets say you have the following model:

```
class Node(models.Model):
    name = models.CharField(max_length=255)
```

You have an initial migration, and you have created a migration named `0002_suffix_name_with_stuff` which looks like this:

```
suffix = ' STUFF'

def add_stuff_to_all_node_names(apps, schema_editor):
    Node = apps.get_model('myapp', 'Node')
    for node in Node.objects.all():
        node.name = '{}{}'.format(node.name, suffix)
        node.save()

def reverse_add_stuff_to_all_node_names(apps, schema_editor):
```

(continues on next page)

(continued from previous page)

```
    Node = apps.get_model('myapp', 'Node')
    for node in Node.objects.all():
        if node.name.endswith(suffix):
            node.name = node.name[:-len(suffix)]
            node.save()

class Migration(migrations.Migration):
    dependencies = [
        ('myapp', '0001_initial'),
    ]

    operations = [
        migrations.RunPython(add_stuff_to_all_node_names, reverse_code=reverse_add_
↪stuff_to_all_node_names),
    ]
```

**Note:** You can not test migrations that can not be reversed, so you **MUST** write reversible migrations if you want to be able to test them. Think of this as a good thing - it forces you to write reversible migrations.

To test this, you can write a test case like this:

```
from ievv_opensource.utils.testhelpers import testmigrations

class TestSomeMigrations(testmigrations.MigrationTestCase):
    app_label = 'myapp'
    migrate_from = '0001_initial'
    migrate_to = '0002_suffix_name_with_stuff'

    def test_migrate_works(self):

        # Add some data to the model using the ``apps_before`` model state
        Node = self.apps_before.get_model('myapp', 'Node')
        node1_id = Node.objects.create(
            name='Node1'
        ).id
        node2_id = Node.objects.create(
            name='Node2'
        ).id

        # Migrate (run the 0002_suffix_name_with_stuff migration)
        self.migrate()

        # Test using the ``apps_after`` model state.
        Node = self.apps_after.get_model('myapp', 'Node')
        self.assertEqual(Node.objects.get(id=node1_id).name, 'Node1 STUFF')
        self.assertEqual(Node.objects.get(id=node2_id).name, 'Node2 STUFF')

    def test_reverse_migrate_works(self):

        # First, we migrate to get to a state where we can reverse the migration
        self.migrate()

        # Add some data to the model using the ``apps_after`` model state
        Node = self.apps_after.get_model('myapp', 'Node')
        node1_id = Node.objects.create(
```

(continues on next page)

(continued from previous page)

```
            name='Node1 STUFF'
        ).id
        node2_id = Node.objects.create(
            name='Node2 STUFF'
        ).id

        # Reverse the migration
        self.reverse_migrate()

        # Test using the ``apps_before`` model state.
        Node = self.apps_before.get_model('myapp', 'Node')
        self.assertEqual(Node.objects.get(id=node1_id).name, 'Node1')
        self.assertEqual(Node.objects.get(id=node2_id).name, 'Node2')
```

#### The MigrationTestCase class

## 6.12 *utils.validation_error_util* — Util for working with ValidationError

## 6.13 *utils.datetime_format* — Utilities for formatting datetimes

## 6.14 *utils.model_field_choices* — Utils for validating model choice fields

## 6.15 *utils.progress_print_iterator* — Util for printing progress for long running scripts

## 6.16 *utils.ievv_json* — Json encoder/decoder

### 6.16.1 Usage

You use the module just like the python json module for all simple use-cases:

```
from ievv_opensource.utils import ievv_json
import datetime
import decimal
import arrow
from django.contrib.auth import get_user_model

json_data = ievv_json.dumps({
    'name': 'Test',
    'score_weight': decimal.Decimal('2.333342'),
    'created_datetime': arrow.get(datetime.datetime(2012, 12, 24, 12, 33), 'Europe/
↪Oslo').datetime,
    'last_changed_by': get_user_model().objects.get(email='someone@example.com')
})
print(json_data)
```

(continues on next page)

(continued from previous page)

```
decoded_back_to_dict = ievv_json.loads(json_data)
print(decoded_back_to_dict)
```

See `ievv_opensource.utils.ievv_json.Encoder` for overview over the extra types we support.

### 6.16.2 Classes and functions

CHAPTER

SEVEN

# RELEASENOTES

## 7.1 ievv_opensource 1.1.0 releasenotes

### 7.1.1 What is new?

- **Django 1.10b1 support.**
    - Minumum Django version is still 1.8.
- Minimum version of django-cradmin updated to 1.1.1.

### 7.1.2 Breaking changes

No breaking changes.

## 7.2 ievv_opensource 5.0.0 releasenotes

### 7.2.1 What is new?

- Make `ievv docs` not run `ievv buildstatic` by default. Have to use `--buildstatic-docs` to get the old default behavior.
- `ievv_developemail`: New module that provides a develop email backend that makes all sent emails browsable through Django admin. See *ievv_developemail — Develop mail backend that lets you view mails in django admin* for more details.

### 7.2.2 Breaking changes

Removed all traces of celery. Should not break anything since the only code using celery was `ievv_batchframework`, and that has been updated to using django-rq a few releases ago.

## 7.3 ievv_opensource 5.0.1 releasenotes

### 7.3.1 What is new?

- Bugfix in `ievv_developemail`. Did not handle unicode correctly when parsing the emails.

## 7.4 ievv_opensource 5.1.0 releasenotes

### 7.4.1 What is new?

- New module for writing tests for Django database migrations. See *Testing Django migrations*.

## 7.5 ievv_opensource 5.10.0 releasenotes

### 7.5.1 New features

- New `utils.datetime_format` module - see *utils.datetime_format — Utilities for formatting datetimes*.

## 7.6 ievv_opensource 5.11.0 releasenotes

### 7.6.1 New features

- New `utils.model_field_choices` module - see *utils.model_field_choices — Utils for validating model choice fields*.

## 7.7 ievv_opensource 5.12.0 releasenotes

### 7.7.1 New features

- New `utils.class_registry_singleton` module - see *utils.class_registry_singleton — Framework for swappable classes*.
- New `utils.progress_print_iterator` module - see *utils.progress_print_iterator — Util for printing progress for long running scripts*.

## 7.8 ievv_opensource 5.13.0 releasenotes

### 7.8.1 New features

- Better handling of unicode in pswin backend for ievv_sms.

## 7.9 ievv_opensource 5.15.0

### 7.9.1 New features

- Script for semantic versioning added (Not implemented anywhere, but for later use if needed).
- Add utility for forcing collation in Django order_by (ievv_db).

## 7.10 ievv_opensource 5.2.0 releasenotes

### 7.10.1 What is new?

- Extend the `iterchoices()`, `iter_as_django_choices_short()` and `iter_as_django_choices_long()` methods of `ievv_opensource.utils.choices_with_meta.ChoicesWithMeta` with support for optional extra choices.

## 7.11 ievv_opensource 5.2.1 releasenotes

### 7.11.1 What is new?

- ievv_developemail support for python 2.x.

## 7.12 ievv_opensource 5.2.2 releasenotes

### 7.12.1 Notes

This is a quickfix for release 5.2.2.

#### Bug-fixes

It did not break anything for python 3.x, but python 2.x was not fully supported.

## 7.13 ievv_opensource 5.21.0

### 7.13.1 New features

- New `ievv_i18n_url` package. This is a fairly rough first version, but it is usable.

## 7.14 ievv_opensource 5.3.0 releasenotes

### 7.14.1 New features

- Add `ievv_opensource.utils.validation_error_util.ValidationErrorUtil`.

## 7.15 ievv_opensource 5.4.0 releasenotes

### 7.15.1 New features

- Add *ievv_model_mommy_extras — Add support for more fields types to model-mommy*.

## 7.16 ievv_opensource 5.5.0 releasenotes

### 7.16.1 New features

- Add *ievv_restframework_helpers — Helpers for working with Django rest framework*.

## 7.17 ievv_opensource 5.6.0 releasenotes

### 7.17.1 New features

- Add support for per directory config for *ievv makemessages*.

## 7.18 ievv_opensource 5.7.0 releasenotes

### 7.18.1 New features

- Add *ievv_sms — SMS sending - multiple backends supported*.

## 7.19 ievv_opensource 5.8.0 releasenotes

### 7.19.1 New features

- Add new `debug_dbstore` backend to *ievv_sms — SMS sending - multiple backends supported*.

## 7.20 ievv_opensource 5.9.0 releasenotes

### 7.20.1 New features

- New ievvtask - `ievv make_source_dist`.

# CHAPTER EIGHT

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

# INDEX