
DISBi Documentation

Release 0.0.2

Rüdiger Frederik Busche

October 29, 2016

1	Setting up Django	3
1.1	Setting up a virtual environment	3
1.2	Setting up the database	3
2	Setting up a DISBi App	5
2.1	Installation and Configuration	5
2.2	Using the DISBi framework	7
3	Setting up the production environment	17
4	Contributing	19
4.1	Design Goals	19
4.2	Style Guide	19
5	Using the Filter View	21
6	Using the Data View	23
7	Using the Admin interface	25
7.1	Uploading data for BiologicalModels	25
7.2	Uploading data for MeasurementModels	26
8	disbi package	27
8.1	Subpackages	27
8.2	Submodules	28
	Python Module Index	45

DISBi is a flexible framework for setting up Django apps for managing experimental and predicted data from Systems Biology projects. A DISBi app presents an integrated online environment, that helps your team to manage the flood of data and share it across the project. DISBi dynamically adapts to its data model at runtime. Therefore, it offers a solution for the needs of many different types of projects. DISBi is open source and freely available.

Features

- Automatic constructions of a *Filter* interface, that allows you to find exactly the experiments you're interested in.
- Integration of related biological objects and the associated experimental data in the *Data View*. Data can be further filtered and downloaded in various formats.
- Preliminary analysis directly in the browser. Fold changes can be calculated and exported with the rest of the data. Histograms of the distributions of data and scatter plots comparing experiments can be generated with a one button press.
- Flexible abstract data model. Specify a data model that meets the requirement of your project. DISBi will figure the relations between the models and necessary steps to integrate the data at runtime.
- Adapt the admin to handle large datasets. With the DISBi framework the admin can be easily configured to allow uploads and export of large datasets using common formats such as CSV or Excel.

To get an impression of what DISBi can do for you, see the following screenshots:

6 experiments meet your requested conditions.

Experiment	Carbon source
4. Metabolome: GCMS -- Glucose	Glucose
5. Metabolome: GCMS -- Fucose	Fucose
6. Metabolome: CoA-method -- Fucose	Fucose
7. Metabolome: CoA-method -- Glucose	Glucose
9. Transcriptome: RNAseq -- Glucose	Glucose
10. Transcriptome: RNAseq -- Fucose	Fucose

Fig. 1: A screenshot of the *filter view* that helps in choosing the experiments of interest.

Copy CSV Excel Column visibility

Showing 1 to 10 of 3,686 entries

locus_tag	product	mean_rpkm_9	mean_rpkm_10	coa_derivativ
SSO9953	Transcriptional regulator containing HTH domain, ArsR family	2.4866e+01	2.9073e+01	
SSO9535	DNA-binding protein 7a	3.2055e+03	3.7155e+03	
SSO9500	TusA-related sulfurtransferase	9.3624e+02	6.1433e+02	
SSO9378	AbrB family transcriptional regulator	9.8170e+01	9.8805e+01	
SSO9221	DNA-directed RNA polymerase	6.3956e+00	1.0658e+01	
SSO9201	acetate-CoA ligase			Propanoyl-Cc
SSO9201	acetate-CoA ligase			CoenzymeA
SSO9201	acetate-CoA ligase			Acetyl-CoA
SSO9180	DNA-binding protein 7a	2.8390e+03	2.4718e+03	
SSO9136	HEPN domain containing protein	1.6481e+01	1.3731e+01	

Search locus_tag Search product Search mean_rpkm_9 Search mean_rpkm_10 Search coa_de

PLOT MEAN_RPKM_9 PLOT MEAN_RPKM_10

Show 10 entries Search:

First Previous 1 2 3 4 5 ... 369 Next Last

Fig. 2: A screenshot of the integrated data in an interactive table in the *Data View*.

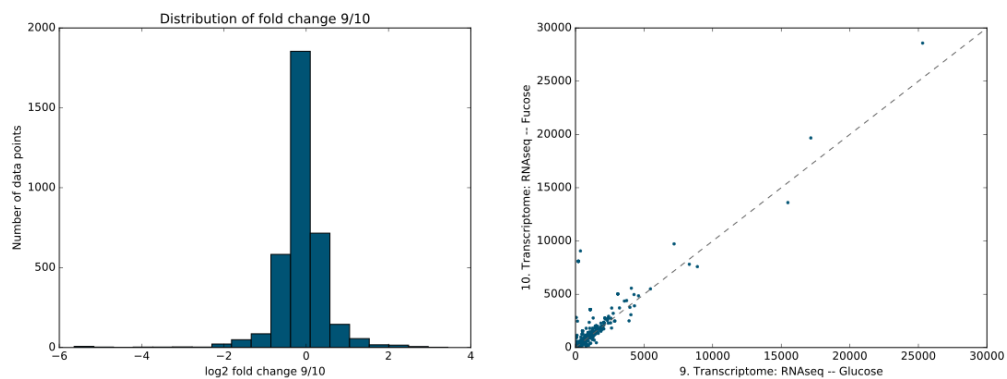


Fig. 3: A screenshot of a histogram and scatter plot, generated from transcriptome data.

Setting up Django

If you are new to Django, it is recommended to take the [tutorial](#). If you have worked with Django, but not in conjunction with a virtual environment or PostgreSQL, you can use the documentation as an opinionated guide. Programmers who have a Django environment with PostgreSQL already running can skip this part.

1.1 Setting up a virtual environment

To encapsulate your project dependencies, it is recommended to use a [virtual environment](#). A convenient way to do this is to use [Conda](#), but any environment manager will do. For a lightweight installation in production, you should use the [Miniconda](#) distribution.

Find the installer appropriate for your distribution at <http://conda.pydata.org/miniconda.html> and download it, e.g.:

```
$ wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
```

Then install from the command line. If you decide to use conda update it first:

```
$ conda update conda
```

Then create a new environment for DISBi:

```
$ conda create -n disbienv python
```

1.2 Setting up the database

A DISBi app requires [PostgreSQL](#) as database backend. This section describes how to install Postgres and set up a new user and a new database that will be used to store the data of your DISBi app. Note that it is not necessary, but only convenient to create the new user as a superuser.

Install compiler:

```
$ sudo apt-get install gcc
```

Install Postgres server and client:

```
$ sudo apt-get install postgresql postgresql-client libpq-dev
```

Login as postgres user:

```
$ sudo -u postgres psql postgres
```

From the Postgres shell create a new superuser:

```
CREATE ROLE <disbi_admin> SUPERUSER LOGIN;
```

And set a password:

```
ALTER USER <disbi_admin> WITH PASSWORD '<passwd>';
```

Exit the Postgres shell and create a database for DISBi:

```
$ sudo -u postgres createdb <disbidb>
```

Setting up a DISBi App

This guide describes both how to install DISBi and configure your Django project correctly, as well as how to use the DISBi framework to set up an app for your Systems Biology project.

2.1 Installation and Configuration

First you should install DISBi via `pip`.

Install from PyPI to get the latest release:

```
$ pip install django-disbi
```

Or install directly from GitHub to get the latest development version:

```
$ pip install -e git+https://github.com/disbi/django-disbi.git#egg=django-disbi
```

Once installed, you can create a new project for setting up a DISBi app or incorporate it in one of your existing projects.

Start project:

```
$ django-admin startproject <disbi_project>
```

Start app:

```
$ python manage.py startapp <organism>
```

Next you need to adapt a few options in your project's `settings.py`.

Add DISBi itself, your newly created DISBi app and `import_export` into intalled apps. DISBi uses `django-import-export` to enable uploads of data via Excel and CSV files:

```
INSTALLED_APPS = [  
    'disbi', # put app first to customize admin CSS  
    'organism.apps.OrganismConfig',  
    'import_export',  
    ...  
]
```

For `import_export` the following configuration is recommended to wrap uploads in transactions and skip the admin log, which speeds up the upload process:

```
IMPORT_EXPORT_USE_TRANSACTIONS = True  
IMPORT_EXPORT_SKIP_ADMIN_LOG = True
```

For global configuration of DISBi apps in your project the following settings are required. `JOINED_TABLENAME` is the name of the backbone table that is used for caching. `DATATABLE_PREFIX` is the prefix added to each cached datatable. `SEPARATOR` determines how values in experiments comparing conditions are separated. For example, a microarray experiment comparing the two mutants *mutA* and *mutB* could be specified in the admin as *mutA/mutB*, given the settings below. `EMPTY_STR` is an internal variable used to represent the empty option in case of combined experiments. It only needs to be replaced if the minus sign has another meaning in your experiments. For example, to specify an experiments that compares *mutA* to the wildtype, *mutA/-* could be given in the admin.

```
# Custom DISBi Settings
DISBI = {
    'JOINED_TABLENAME': 'joined_bio_table',
    'DATATABLE_PREFIX': 'datatable',
    'SEPARATOR': '/',
    'EMPTY_STR': '-',
}
```

Then you set up the connection to your Postgres database:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': '<disbidb>',
        'USER': '<disbi_admin>',
        'PASSWORD': '<passwd>',
        'HOST': '127.0.0.1', # Or an IP Address that your DB is hosted on
        'PORT': '5432',
    }
}
```

Now you need to set up directories and URLs for serving static files and collect those files for `import_export`.

Set up `MEDIA_ROOT` and URL and `STATIC_ROOT` and URL:

```
MEDIA_ROOT = '/home/disbi/media/disbi/'
MEDIA_URL = '/media/'
STATIC_ROOT = os.path.join('/home/disbi/disbi/project/disbi/static')
STATIC_URL = '/static/'
```

If you decide to use some of the custom templates from the boilerplate, you need to add the directories they are included in to your template directories. For example, if you include the templates at the project's root you need to add:

```
TEMPLATES = [
    {
        ...,
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        ...,
    },
]
```

Then let Django collect it static files:

```
$ python manage.py collectstatic
```

Now you are free to set up your models, admin, views and URLs. While the view and URLs can be simply copied from the boilerplate, models and admin are more complex and need to be adapted to your project's needs. A detailed description of how to configure them is available in *Specifying a data model*. Once you are finished with configuring the models and the admin, you can migrate your app, create an admin superuser and other accounts and let people start to upload their experimental data.

Make migrations and migrate:

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

Create a new Django superuser for the admin:

```
$ python manage.py createsuperuser
```

Verify that everything works as expected with the development server:

```
$ python manage.py runserver
```

2.2 Using the DISBi framework

To make DISBi useful for wide range of projects, it is designed rather as a framework than an application. Though it in fact is a Django app, that handles some basic tasks, it mostly provides you with classes that help you set up an application that meets your requirements. In this section we walk you through the necessary steps to set up a DISBi app by constructing a simple app that integrates data from flux balance predictions and metabolome analysis.

2.2.1 Specifying a data model

The data model defines what kind of information is stored in your app and how this information interrelates. DISBi uses extended versions of Django Models and Fields for the specification of its data model. Though DISBi will adapt dynamically to your data model at runtime, it requires the data model to conform to an overall general structure, the *abstract data model*. The abstract data model is an attempt at generalizing the structure of data from the domain of Systems Biology or experimental data in general. It does so by grouping models into three abstract categories and a concrete model: *BiologicalModel*, *MeasurementModel*, *MetaModel* and *Experiment*.

Every source of data (simulation or real experiment) is stored in the *Experiment* model, with its respective parameters. *MeasurementModels* store the data points generated in these experiments. *BiologicalModels* store biological objects to which data points map and *MetaModels* store information about these biological objects.

As you can see in the entity relationship model in [Fig. 2.1](#), each data point from the *MeasurementModels* can be uniquely identified by mapping to exactly one experiment and one instance of a *BiologicalModel*.

Let's consider how we would construct models for a DISBi app that integrates flux and metabolome data.

First we need to consider what parameters we will vary in our experiments and simulations. To keep things simple, we will say that we only use different *carbon sources* and different *mutants*. Additionally, we should store the *type* of the experiment, i.e. flux or metabolome, and the *date* it was performed on. Moreover, we will leave some space for *notes*.

```
# models.py
import disbi.disbimodels as dmodels
from disbi.models import (BiologicalModel, DisbiExperiment,
                          DisbiExperimentMetaInfo, MetaModel,
                          MeasurementModel,)

class Experiment(models.Model, DisbiExperiment):
    EXPERIMENT_TYPE_CHOICES = (
        ('flux', 'Predicted Flux'),
        ('metabolome', 'Metabolome'),
    )
    experiment_type = dmodels.CharField(max_length=45,
                                       choices=EXPERIMENT_TYPE_CHOICES,
                                       di_choose=True)
```

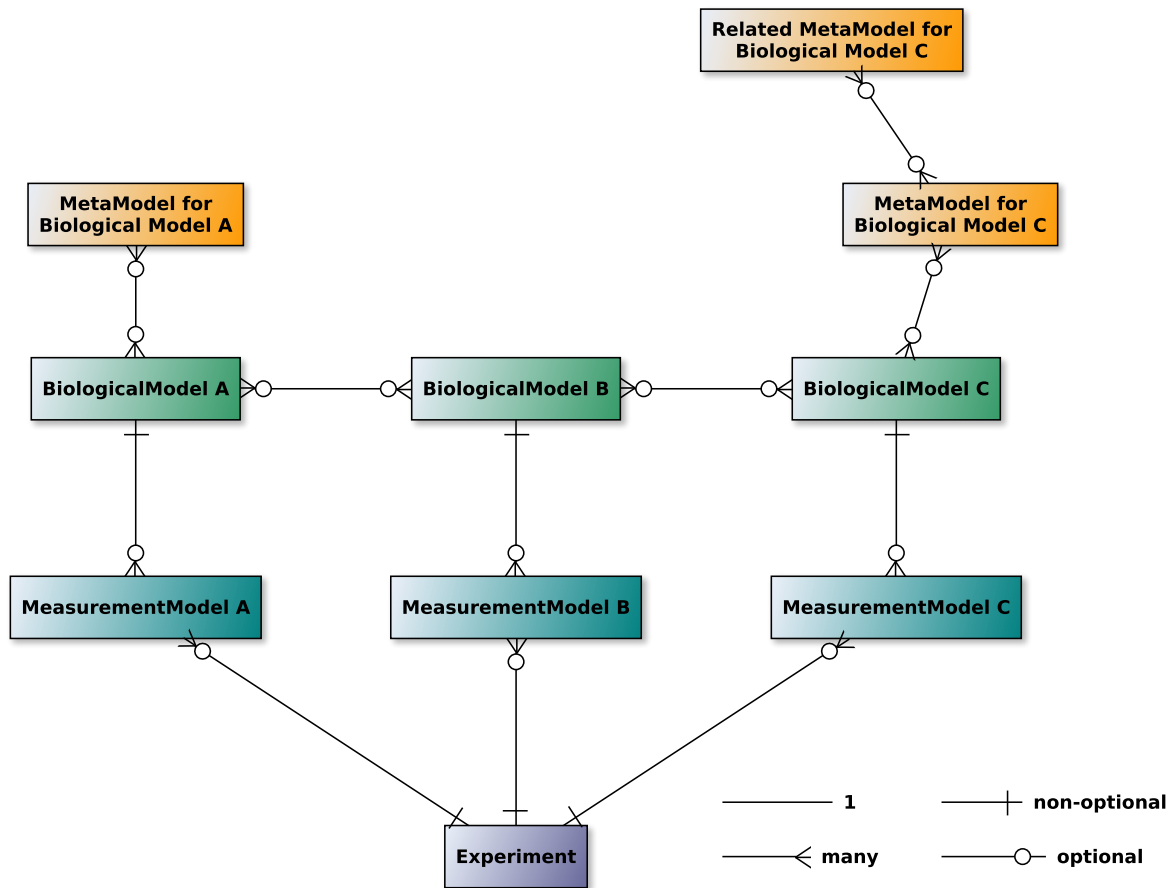


Fig. 2.1: Entity relationship model for relations between the model groups in DISBi's abstract data model.

```

carbon_source = dmodels.CharField(max_length=45, blank=True,
                                   di_choose=True, di_show=True)
mutant = dmodels.CharField(max_length=45, blank=True, di_choose=True,
                            di_show=True)
date = dmodels.DateField(max_length=45)
notes = dmodels.TextField(blank=True)

def __str__(self):
    return '{}. {}'.format(
        self.id,
        self.get_experiment_type_display()
    )

```

Your Experiment needs to be constructed by mixing in *DisbiExperiment* to the standard Django Model class. As you notice, we imported and used `dmodels.FieldClass` instead of the standard Django `models.FieldClass`. These are extended versions of Django field classes, that allow for some DISBi specific options to be passed, which always start with `di_`. Otherwise they work as the standard Django classes. Let's have a look at what those options do:

- `di_choose` Determines whether a select widget will be created for this field in the Data View. Since we only want to filter by experiment type, carbon source and mutant, we only need to set the attribute on those fields.
- `di_show` Determines whether the field will be shown in the tables summarizing the matched experiments in the filter view. In addition to these fields, the `__str__()` method of the `Experiment` class will be included in the table. Since `__str__()` includes the experiment type already, we don't need to include it again.

Next we could override the `result_view()` method, that determines the content of the table summarizing the matched experiments in the data view. However, this is only necessary if would want to include information that is not directly in the `Experiment` models fields, such as hyperlinks. So we just leave it untouched, such that it will yield the same table as in the Data View.

Finally, we need to add a class called `ExperimentMetaInfo`. This class handles determining the `MeasurementModel` and `BiologicalModel` for each experiment. We only have to create it by using a `Mixin`. No further customization is required.

```

class ExperimentMetaInfo(Experiment, DisbiExperimentMetaInfo):

    pass

```

Now we want to set up models that store information about the biological objects we measure in our experiments, the `BiologicalModels`. We will map the flux data to *Reactions* and the metabolome data to *Metabolites*. We will relate a reaction to a metabolite, whenever a metabolite occurs in the reaction equation of a reaction. This is a many-to-many relation:

```

class Reaction(BiologicalModel):
    name = dmodels.CharField(max_length=255, unique=True, di_show=True,
                              di_display_name='reaction_name')
    reaction_equation = dmodels.TextField()
    metabolite = dmodels.ManyToManyField('Metabolite', related_name='reactions')

    def __str__(self):
        return self.name

class Metabolite(BiologicalModel):
    name = dmodels.CharField(max_length=512, unique=True, di_show=True,
                              di_display_name='metabolite_name',
                              di_show=True)

```

```
def __str__(self):
    return self.name
```

As you notice, both classes derive from *BiologicalModel*. This is done to identify them as *BiologicalModels* for DISBi. Moreover, you see a new field option.

- `di_display_name` This option is the name by which the field will be included in the *result table*. It only makes sense to be set if `di_show` is set to `True`, but has to be set if the normal field name collides with field names of other models. Otherwise, the columns would be indistinguishable in the result table. (Notice that both *Reaction* and *Metabolite* have a name attribute.)
- `di_show` This option has a different meaning for *BiologicalModels*. It determines whether or not the field should be included in the result table.

When constructing your *BiologicalModels* it is always important to keep in mind the granularity of your measurement data. For example, you should not use a *Metabolite* model to map data from a measurement method that can only resolve groups of derivatives. Instead you should create a new *Derivative* model to which you map your data and relate it to your *Metabolite* model, such that each *Derivative* is related to each *Metabolite* it can derive from.

Now we also want to store more information about our *Reactions*. For example we could store all biochemical pathways in which the reaction occurs. This is a perfect case for a *MetaModel*:

```
class Pathway(MetaModel):
    name = dmodels.CharField(max_length=255, unique=True, di_show=True,
                             di_display_name='pathway')
    reaction = dmodels.ManyToManyField('Reaction', related_name='reactions')

    def __str__(self):
        return self.name
```

Notice that we could not have stored this information as a field on the original *Reaction* model, since many reactions can occur in many pathways and vice versa. The relation is therefore many-to-many.

As a final step we need to set up our *MeasurementModels*. These models need to reflect the data generated by our methods. Moreover, we need to include an explicit reference to the *BiologicalModel* the data maps to. The reference to the *Experiment* model is already included in the base class. Let's assume that our flux balance analysis program gives us a flux value and an upper and lower bound for this value. Let's further assume that we perform our metabolome method in triplets, so that we only store the mean and the standard error of each sample. This could be encoded in the models as follows:

```
class FluxData(MeasurementModel):
    flux = dmodels.FloatField(di_show=True)
    flux_min = dmodels.FloatField(di_show=True, di_display_name='lb')
    flux_max = dmodels.FloatField(di_show=True, di_display_name='ub')

    reaction = dmodels.ForeignKey('Reaction', on_delete=models.CASCADE)

    class Meta:
        unique_together = (('reaction', 'experiment',))
        verbose_name_plural = 'Fluxes'

    def __str__(self):
        return 'Flux data point'

class MetabolomeData(MeasurementModel):
    mean = dmodels.FloatField(di_show=True)
    stderr = dmodels.FloatField(di_show=True)
```

```

metabolite = dmodels.ForeignKey('Metabolite', on_delete=models.CASCADE)

class Meta:
    unique_together = (('metabolite', 'experiment'),)
    verbose_name_plural = 'Metabolome data points'

def __str__(self):
    return 'Metabolome data point'

```

If we look at our data model as a whole, we can see that it has all the features demanded by the abstract data model.

Congratulations, you have just finished making your first DISBi data model. DISBi data models can grow much more complex than described here. You can map more than one `MeasurementModel` to the same `BiologicalModel` or no `MeasurementModel` at all. You can also have more complex relation between your `BiologicalModels`. The only requirement is that the graph formed by the relations between your `BiologicalModels` and `MetaModels` is a *tree*, i.e. every model needs to be reachable from every other model and their must be no circles. This is due to the way DISBi automatically joins the data behind the scenes.

2.2.2 Configuring the admin

Once you have figured out your data model, you need to set up an admin interface so that researches can easily upload their data. Though you have full freedom in customizing the Django admin, DISBi provides a few usefull classes to set up an admin that's suitable for handling experimental datasets.

In general you'll want one `Admin` class for each of your model classes. Since normal Django `ModelAdmins` just offer an HTML form to enter new data, DISBi uses `django-import-export` to enable data upload of larger datasets from files, like CSV and Excel. The handling of the file upload is mostly done by a `Resource` class. DISBi offers the factory function `disbiresource_factory()` that produces a `Resource` class that checks data integrity before inserting the value into the database. It is recommended, though not necessary to use the factory. The admin classes for our `BiologicalModels` could look like this:

```

# admin.py
from import_export.admin import ImportExportModelAdmin
from django.contrib import admin
from disbi.admin import (DisbiDataAdmin, disbiresource_factory)
from .models import (Experiment, FluxData, MetabolomeData,
                    Metabolite, Reaction, Pathway)

@admin.register(Reaction)
class ReactionAdmin(ImportExportModelAdmin):
    resource_class = disbiresource_factory(
        mymodel=Reaction,
        myfields=('name', 'reaction_equation',
                 'metabolite',),
        myimport_id_fields=['name'],
        mywidgets={'metabolite':
                   {'field': 'name'}}
    )
    search_fields = ('name', 'reaction_equation',)
    filter_horizontal = ('metabolite',)

@admin.register(Metabolite)
class MetaboliteAdmin(ImportExportModelAdmin):
    resource_class = disbiresource_factory(
        mymodel=Metabolite,

```

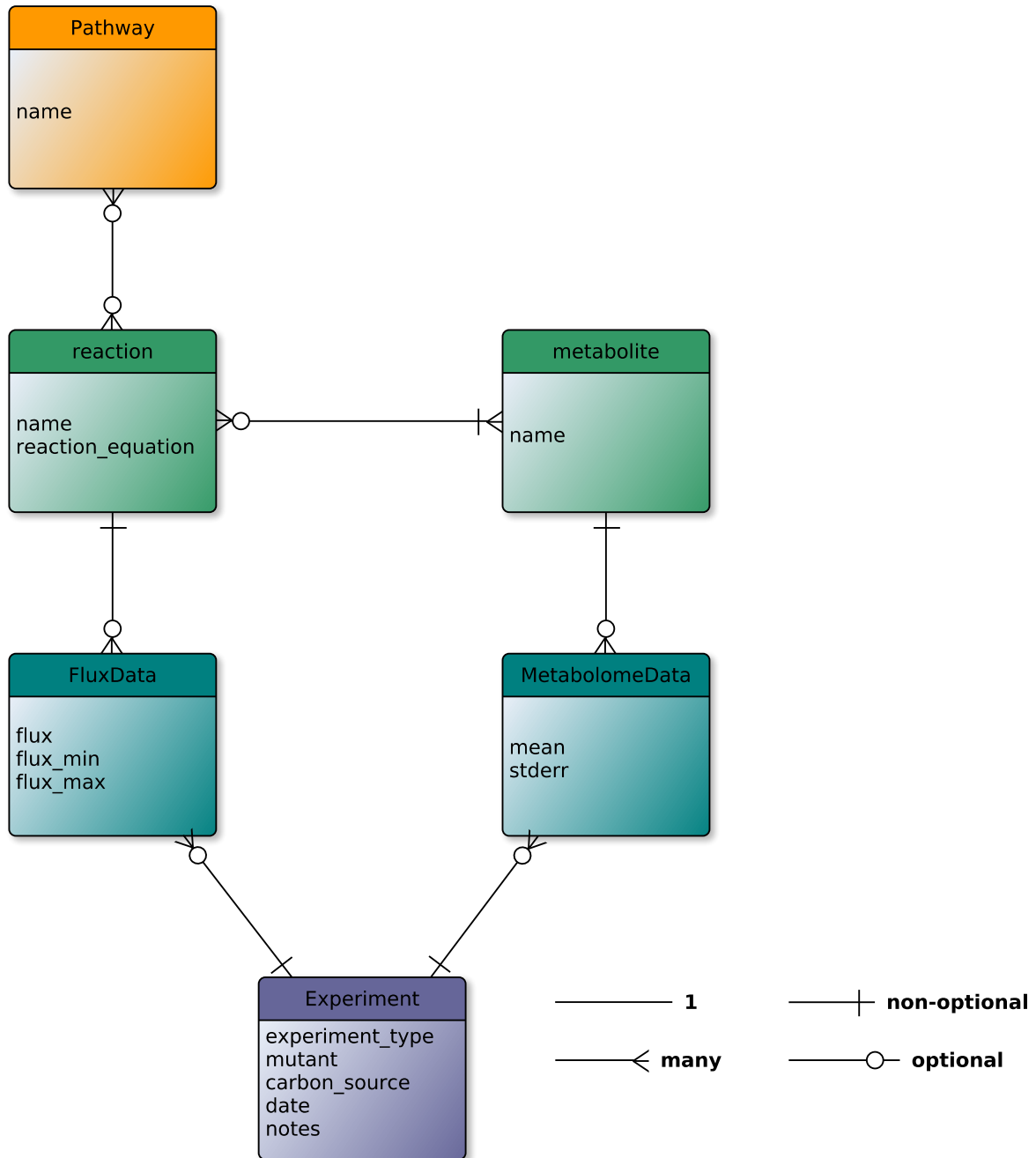


Fig. 2.2: Entity relationship model for the concrete data model.


```

        myfields=('name',),
        myimport_id_fields=['name'],
    )
    search_fields = ('name',)

```

Let's look more closely at the arguments of `disbiresource_factory()`.

- `mymodel` is the `Model` class the `Resource` is created for. This is the same class that is registered for the admin.
- `myfields` are the fields that will be imported from the uploaded file and therefore have to be present as columns in the file. The list should include all fields that were set in `models.py`.
- `myimport_id_fields` is the human readable primary key that serves for identifying the rows in the uploaded file as objects in the database. Though Django uses numerical ids internally, researchers don't talk about reactions and metabolites in terms of numbers. With this option, you can also specify compound keys (a key that consist of more than one field) and update data by changing values in your data file and re-uploading it.
- `mywidgets` is a dictionary, that passes `Meta` options to the `Widget` class used in the import. That is especially important when importing a foreign key, as the identifying attributes of the other `Model` have to be put here.

The configuration of the admin class for our `Pathway` model follows the same principle:

```

@admin.register(Pathway)
class PathwayAdmin(ImportExportModelAdmin):
    resource_class = disbiresource_factory(
        mymodel=Pathway,
        myfields=('name', 'reaction',),
        myimport_id_fields=['name'],
        mywidgets={'reaction':
                    {'field': 'name'}}
    )
    search_fields = ('name',)
    filter_horizontal = ('reaction',)

```

Now lets turn to our `MeasurementModels`. These pose a special challenge since researchers usually will produce one file per experiment. This way, each file will have to contain a column with the same value for the same experiments. To save users from the tedious process of appending a column to each file, DISBi offers a special admin class. It gives the user the opportunity to choose the experiment the data belongs to at the time the file is uploaded. This class only has to be configured with our concrete `Experiment` model. A pattern we'll encounter again when setting up the views.

```

class MeasurementAdmin(DisbiMeasurementAdmin):
    model_for_extended_form = Experiment

```

Then we can use it as a base class to define the admin classes for our `MeasurementModels`:

```

@admin.register(FluxData)
class FluxAdmin(MeasurementAdmin):
    resource_class = FluxResource

    filter_for_extended_form = {'experiment_type': 'flux'}

    list_display = ('reaction', 'flux', 'flux_min', 'flux_max',)
    search_fields = ('reaction__reaction_equation', 'reaction__name',)

@admin.register(MetabolomeData)
class MetabolomeDataAdmin(MeasurementAdmin):
    resource_class = disbiresource_factory(

```

```

mymodel=MetabolomeData,
myfields=('metabolite', 'mean', 'stderr', 'experiment',),
myimport_id_fields=['metabolite', 'experiment'],
mywidgets={'metabolite':
            {'field': 'name'},}
)

filter_for_extended_form = {'experiment_type': 'metabolome'}

list_display = ('metabolite', 'mean',)

```

Note that we don't have to specify how the experiments should be identified in `mywidgets` as this will be handled by the `MeasurementAdmin` class. We also set a the class-level attribute `filter_for_extended_form`. This dictionary will be passed as keyword arguments to the `filter()` on the `Experiment` model. It determines which of the stored experiments are eligible. It makes sense to limit those to the experiments of the corresponding type. `MeasurementAdmin` will also add a filter in the admin site for each `MeasurementModel`, so the data points can be filtered by the experiments they belong to.

Finally, we need an admin class for our `Experiment` model. This can be kept simple. Let's only set the `save_as` option to allow users to use existing experiments as templates for creating new entries:

```

@admin.register(Experiment)
class ExperimentAdmin(admin.ModelAdmin):
    save_as = True
    save_as_continue = False

```

Now you've gone through the difficult part of configuring your DISBi app. You'll be good to go after a few final steps.

2.2.3 Setting up views and URLs

The configuration of the views and URLs is simply boilerplate code. DISBi uses class-based views to allow for easy configuration. The idea is that you subclass these views and configure them with your concrete `Experiment` model, as DISBi cannot know about your model by itself. However, since the code will always look the same you can simply copy it:

```

# views.py
from disbi.views import (DisbiCalculateFoldChangeView, DisbiComparePlotView,
                        DisbiDataView, DisbiDistributionPlotView,
                        DisbiExperimentFilterView, DisbiExpInfoView,
                        DisbiGetTableData)
from .models import Experiment, ExperimentMetaInfo

class ExperimentFilterView(DisbiExperimentFilterView):
    experiment_model = Experiment

class ExperimentInfoView(DisbiExpInfoView):
    experiment_model = Experiment

class DataView(DisbiDataView):
    experiment_meta_model = ExperimentMetaInfo

class CalculateFoldChangeView(DisbiCalculateFoldChangeView):
    experiment_model = Experiment

```

```

experiment_meta_model = ExperimentMetaInfo

class ComparePlotView(DisbiComparePlotView):
    experiment_model = Experiment
    experiment_meta_model = ExperimentMetaInfo

class DistributionPlotView(DisbiDistributionPlotView):
    experiment_model = Experiment
    experiment_meta_model = ExperimentMetaInfo

class GetTableData(DisbiGetTableData):
    experiment_meta_model = ExperimentMetaInfo

```

Unless you want to modify some of the views, it is not really important to know what they do exactly. More information can be found in the [API documentation](#).

The configuration of the URLs is similarly fixed. You simply need to associate your views with the right URL patterns. As the views often take arguments from the URL patterns, you should not try to change them. The simplest thing is again to stick to the boilerplate code:

```

# urls.py
from django.conf.urls import url
from . import views

app_name = 'yourapp'
urlpatterns = [
    url(r'^filter/exp_info/', views.ExperimentInfoView.as_view(), name='exp_info'),
    url(r'^filter/', views.ExperimentFilterView.as_view(), name='experiment_filter'),
    url(r'^data/(?P<exp_id_str>\d+(?:_\d+)*)/get_distribution_plot/',
        views.DistributionPlotView.as_view(),
        name='get_distribution_plot'),
    url(r'^data/(?P<exp_id_str>\d+(?:_\d+)*)/get_compare_plot/',
        views.ComparePlotView.as_view(),
        name='get_compare_plot'),
    url(r'^data/(?P<exp_id_str>\d+(?:_\d+)*)/calculate_fold_change/',
        views.CalculateFoldChangeView.as_view(),
        name='fold_change'),
    url(r'^data/(?P<exp_id_str>\d+(?:_\d+)*)/get_table_data/',
        views.GetTableData.as_view(),
        name='get_table_data'),
    url(r'^data/(?P<exp_id_str>\d+(?:_\d+)*)/$', views.DataView.as_view(), name='data'),
]

```

Then you only need to include your apps URLs in your project's `urls.py` and you're done.

This was a quick tour through what you can accomplish with DISBi and how to do it. To help getting started even faster, there is a complete boilerplate available on [GitHub](#).

If you encounter any problems when setting up your DISBi app, feel free to contact us on [GitHub](#) and open an issue. We are happy to hear your experiences, so we can continuously improve and extend DISBi in the way the research community needs it. If you want to help to improve DISBi yourself, you can find all necessary information in [Contributing](#).

Setting up the production environment

Once you have successfully set up your DISBi application and verified that everything works as intended, you will want to make DISBi available to all people from your Systems Biology project. To do this, you need to move from the development to a **production environment**. This is a guide for using **Apache Server** on Debian/Ubuntu as a production server. Any other production environment or server recommended by the Django documentation will do as well.

Install Apache:

```
$ sudo apt-get install apache2 apache2-dev
```

Download `mod_wsgi`, but look for newer version on https://github.com/GrahamDumpleton/mod_wsgi/releases:

```
$ wget https://github.com/GrahamDumpleton/mod_wsgi/archive/4.5.7.tar.gz
$ tar xvfz 4.5.7.tar.gz
```

Install from source:

```
$ ./configure --with-python=</path/to/env/>
$ make
$ sudo make install
```

Enable `mod_wsgi` with Debian script:

```
$ sudo a2enmod wsgi
```

Set `WSGIPythonHome`, in `apache2.conf`.

See <http://modwsgi.readthedocs.io/en/develop/user-guides/installation-issues.html> in case of problems.

Set `WSGIPythonHome`, in `apache2.conf`, see also <https://docs.djangoproject.com/en/1.10/howto/deployment/wsgi/modwsgi/>.

Set up a virtual host by using the template from the boilerplate.

Enable `vhost` and reload the server:

```
$ a2ensite disbi.conf
$ service apache2 reload
```

If error occurs, probably `tkinter` is missing:

```
$ sudo apt-get install python3-tk
```

Preparing for production by adapting `settings.py`:

```
DEBUG = False
ALLOWED_HOSTS = ['myhost']
```

Contributing

DISBi is an open source project released under MIT License. Everybody is welcome to contribute! There is always a need for better documentation, cleaner code and new features. To ensure long term code quality and consistency here are a few guidelines that you should heed when committing code to the project.

4.1 Design Goals

DISBi aims at presenting a general solution for data integration in Systems Biology. To do this the software pursues several goals.

- **Data model independence** To be able to adapt to the diverse requirements of different projects, all features should rely only on the *abstract data model*.
- **Data management** Data should be easy to manage through the admin interface. Moreover, the data should be made accessible through user friendly *filter* interfaces, that do not require knowledge of the underlying database structure.
- **Data integration** Data should be made available in an integrated manner that facilitates further analysis and discovery of underlying patterns. Moreover, one should be able to export the integrated data in commonly used formats.
- **Preliminary analysis** DISBi tries to support the researcher by automating common analysis routines. These should help in identifying data that is worthwhile for more in depth analysis. However, DISBi tries not to replicate the full functional scope of mature data analysis tools. Instead, it tries to give the user the freedom to export data in interchangeable formats and let him choose his own analysis tool.

If you have a contribution that surpasses the scope of the Design Goals, but is useful nevertheless, you should consider designing your contribution as a pluggable extension.

4.2 Style Guide

4.2.1 Python

Try to follow and [PEP 8](#) the [Google Style Guide](#) as closely as possible. Everything should have at least a one-line docstring. Use [Google Style](#) for longer docstrings.

4.2.2 JavaScript

In general, follow the recommendations of the [jQuery style guide](#) and code examples presented in the jQuery documentation.

4.2.3 Sass

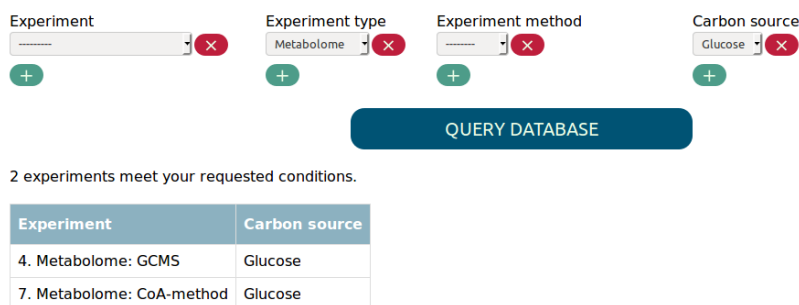
Follow the recommendations of the [Sass styleguide](#) by Hugo Giraudel.

[PEP8](#)

Using the Filter View

The Filter View provides you with an interface to find the experiments you're interested in based on experimental conditions. This way you can choose your experiments of interest, before you actually query the database for the experimental data.

You find a select widget for each experimental parameter. Whenever you change the options you selected, the table summarizing the matched experiments will be updated. For example, if you want all metabolome experiments with carbon source glucose, you will get the following result.



Experiment

Experiment type

Experiment method

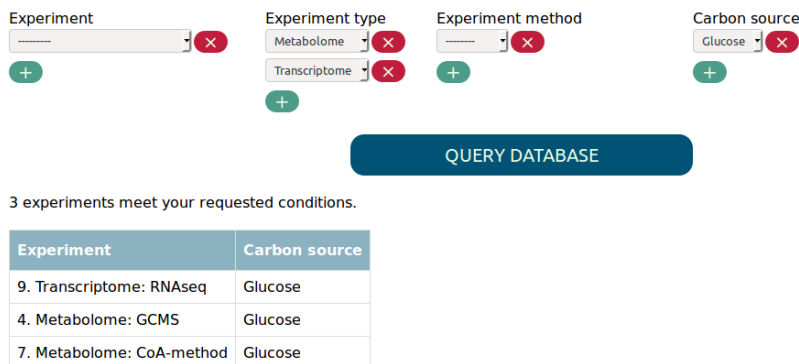
Carbon source

QUERY DATABASE

2 experiments meet your requested conditions.

Experiment	Carbon source
4. Metabolome: GCMS	Glucose
7. Metabolome: CoA-method	Glucose

If you are interested in more than one value for the same parameter, you can simply add another select box using the + button. This way you can for example find all metabolome and transcriptome experiments in which glucose was used as carbon source.



Experiment

Experiment type

Experiment method

Carbon source

QUERY DATABASE

3 experiments meet your requested conditions.

Experiment	Carbon source
9. Transcriptome: RNAseq	Glucose
4. Metabolome: GCMS	Glucose
7. Metabolome: CoA-method	Glucose

Note that multiple values for the same parameter are connected by a logical OR and values for different parameters are connected by a logical AND when looking for matching experiments. The above input, for example, would be

equivalent to the expression:

```
Experiments with (type metabolome OR type transcriptome) AND carbon source glucose
```

In order to find experiments that compare two conditions you need to select both values in the respective parameter. For example, to find the proteome experiment that compares fucose to glucose as carbon source you need to select the following:

The screenshot shows a filter interface with four columns: Experiment, Experiment type, Experiment method, and Carbon source. Each column has a dropdown menu and a red 'X' icon. Below each dropdown is a green '+' icon. A blue button labeled 'QUERY DATABASE' is centered below the filters. Below the button, it says '1 experiment meets your requested conditions.' Below that is a table with two columns: Experiment and Carbon source.

Experiment	Carbon source
8. Proteome: shotgun	Fucose/Glucose

Selecting only fucose or glucose as carbon source would not match the experiment.

Finally, you have the option to directly select experiments with the experiment tab. Experiments select in this tab get added to the set of matched experiments, regardless of the other selected parameters. This way you can, for example, add a single flux simulation that uses fucose as carbon source to the transcriptome and metabolome experiments that use glucose. Adding only a single experiments would be otherwise difficult if there are many flux experiments that use fucose.

The screenshot shows a filter interface with four columns: Experiment, Experiment type, Experiment method, and Carbon source. Each column has a dropdown menu and a red 'X' icon. Below each dropdown is a green '+' icon. A blue button labeled 'QUERY DATABASE' is centered below the filters. Below the button, it says '4 experiments meet your requested conditions.' Below that is a table with two columns: Experiment and Carbon source.

Experiment	Carbon source
9. Transcriptome: RNAseq	Glucose
12. Predicted Flux: FBA	Fucose
4. Metabolome: GCMS	Glucose
7. Metabolome: CoA-method	Glucose

Once you are satisfied with the matched experiments you can click “QUERY DATABASE”, which will take you to the Data View.

Using the Data View

Assuming you selected all transcriptome and metabolome experiments with glucose or fucose as carbon source, you would be presented the following in the Data View.

The screenshot shows a table with three columns: Name, File, and Carbon Source. Below the table are several interactive widgets: a 'HIDE EXPERIMENTS' button, a comparison selector with a red 'X' icon, a '+' icon, a 'CALCULATE FOLD CHANGE' button, a 'Plot' selector followed by 'against' and another selector, and a 'PLOT AGAINST' button.

Name	File	Carbon Source
4. Metabolome: GCMS	MMI-2016-15712_Suppo[...].xlsx	Glucose
5. Metabolome: GCMS	MMI-2016-15712_Suppo[...].xlsx	Fucose
6. Metabolome: CoA-method	MMI-2016-15712_Suppo[...].xlsx	Fucose
7. Metabolome: CoA-method	MMI-2016-15712_Suppo[...].xlsx	Glucose
9. Transcriptome: RNAseq	MMI-2016-15712_Suppo[...].xlsx	Glucose
10. Transcriptome: RNAseq	MMI-2016-15712_Suppo[...].xlsx	Fucose

At the top you see a table summarizing the matched experiments again. Then you can see some widgets that allow you to perform some preliminary analysis. We return to them later.

Farther down on the page is the *data table* that contains the actual integrated experimental data. The biological index, i.e. the biological objects the data map to, always precedes the numerical data for every type of experiment. Every column is suffixed with the id of the experiments it belongs to. The data is integrated in the way that loci and metabolites that are related to each other (e.g. by the encoded enzymes) appear in the same row. This leads of course to partial duplication of data if one locus is related to many metabolites.

The data table is searchable and columns can be completely hidden using the `Column visibility` tab. The data can also be copied to the clipboard or exported as CSV or Excel. Hidden columns and data that was filtered out with the search will not be exported.

The distribution of every column can be plotted as a histogram using the button at the bottom of the column. To plot a scatter plot comparing two experiments, the widget above the table can be used. The generated plots will be appended to the bottom of the page. You can remove them again simply by clicking on them.

Fold changes can be calculated and added to the table using the other widget above the table. If you are interested in multiple fold changes at the same time you can add more rows for specifying the experiments to be compared.

Copy CSV Excel Column visibility

Showing 1 to 10 of 3,686 entries

locus_tag	product	mean_rpkm_9	mean_rpkm_10	coa_derivativ
SSO9953	Transcriptional regulator containing HTH domain, ArsR family	2.4866e+01	2.9073e+01	
SSO9535	DNA-binding protein 7a	3.2055e+03	3.7155e+03	
SSO9500	TusA-related sulfurtransferase	9.3624e+02	6.1433e+02	
SSO9378	AbrB family transcriptional regulator	9.8170e+01	9.8805e+01	
SSO9221	DNA-directed RNA polymerase	6.3956e+00	1.0658e+01	
SSO9201	acetate-CoA ligase			Propanoyl-Cc
SSO9201	acetate-CoA ligase			CoenzymeA
SSO9201	acetate-CoA ligase			Acetyl-CoA
SSO9180	DNA-binding protein 7a	2.8390e+03	2.4718e+03	
SSO9136	HEPN domain containing protein	1.6481e+01	1.3731e+01	

Show 10 entries

[First](#)
[Previous](#)
1
[2](#)
[3](#)
[4](#)
[5](#)
[...](#)
[369](#)
[Next](#)
[Last](#)

Using the Admin interface

The Admin interface presents you a page for each `Model` that contains data in the database. You can browse the entries, add new ones or change existing ones using simple forms.

For uploading larger datasets, the Admin interface (or short Admin) is equipped with support for spreadsheet upload.

7.1 Uploading data for BiologicalModels

When you go to the `IMPORT` tab on a model's page you are prompted to upload a file and specify the file format. The page also shows you the fields that will be imported. For an exemplary `LOCUS` model the page could look like this:

Import

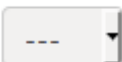
This importer will import the following fields: `locus_tag`, `product`, `ec_number`, `reaction`

File to import:

Browse...

No file selected.

Format:



The required fields must be included as column in the spreadsheet file you upload. As one locus can be related to many reaction, all related reaction must be present in the `reaction` column, separated by commas. A valid for uploading `LOCUS` data could look like this:

*Exemplary table for the data upload to the Locus model (data from *Sulfolobus solfataricus*).*

locus_tag	product	ec_number	reaction
SSO0299	transketolase	2.2.1.1	carb_ppp_2TRANSKETO-RXN,carb_ppp_2.2.1.1_1TRANSKETO-RXN,cof_thiamin_plp_2.2.1.7_DXS-RXN
SSO0302	Chorismate mutase	5.4.99.5	aa_tyr_PREPHENATEDEHYDROG-RXN
SSO0304	3-deoxy-7-phosphoheptulonate synthase	2.5.1.54	aa_phe_tyr_trp_shiki_DAHPSYN-RXN
SSO0305	3-dehydroquininate synthase	4.2.3.4	aa_phe_tyr_trp_shiki_3-DEHYDROQUINATE-SYNTHASE-RXN
SSO0306	shikimate dehydrogenase	1.1.1.25	aa_phe_tyr_trp_shik_RXN-7968_NADP
SSO0307	chorismate synthase	4.2.3.5	aa_phe_tyr_trp_shiki_CHORISMATE-SYNTHASE-RXN
SSO0308	shikimate kinase	2.7.1.71	aa_phe_tyr_trp_shiki_SHIKIMATE-KINASE-RXN
SSO0309	3-phosphoshikimate 1-carboxyvinyl transferase	2.5.1.19	aa_phe_tyr_trp_shiki_2.5.1.19-RXN
SSO0311	3-dehydroquininate dehydratase	4.2.1.10	aa_phe_tyr_trp_shiki_3-DEHYDROQUINATE-DEHYDRATASE-RXN

Note that the first row contains two entries for reaction.

7.2 Uploading data for MeasurementModels

Uploading actual experimental data follows the same principle as uploading biological data. The only difference is that you need to choose the experiment the data belongs to at the moment you upload it. Therefore you do not need to include an *experiment* column in your spreadsheet, even though it is listed as a field that is going to be imported.

Import

This importer will import the following fields: `experiment`, `flux`, `flux_min`, `flux_max`, `reaction`

File to import: No file selected.

Format:

Experiment:

Whenever your uploaded data does not make it through the validation process, you will be informed about what went wrong in which row of the dataset. This helps to ensure that the data stored in the database remains consistent.

8.1 Subpackages

8.1.1 disbi.migrations package

Submodules

`disbi.migrations.0001_initial` module

```
class disbi.migrations.0001_initial.Migration (name, app_label)
```

```
    Bases: django.db.migrations.migration.Migration
```

```
    dependencies = []
```

```
    initial = True
```

```
    operations = [<CreateModel name='Checksum', fields=[('id', <django.db.models.fields.AutoField>), ('table_name', <d
```

8.1.2 disbi.templatetags package

Custom template tags and filters used in DISBi templates.

Submodules

`disbi.templatetags.custom_filters` module

```
disbi.templatetags.custom_filters.get_item (dictionary, key)
```

```
disbi.templatetags.custom_filters.get_item_by_idx (iters, idx)
```

```
disbi.templatetags.custom_filters.get_list (qdict, key)
```

`disbi.templatetags.custom_template_tags` module

```
disbi.templatetags.custom_template_tags.nested_dict_as_table (d, make_foot,  
                                                                **kwargs)
```

Render a list of dictionaries as HTML table with keys as footer and header.

Parameters `d` – A list of dictionaries. Use an `OrderedDict` for the table to maintain order.

Keyword Arguments ****kwargs** – Valid HTML Global attributes, which will be added to the <table> tag.

`disbi.templatetags.custom_template_tags.print_none` (*obj*)

8.2 Submodules

8.2.1 disbi.admin module

Useful admin classes and factory function that can be used to configure the admin of the concrete app.

class `disbi.admin.DisbiDataAdmin` (*model, admin_site*)

Bases: `disbi._import_export.admin.RelatedImportExportModelAdmin`

Allow experimental data models to be filtered by their related experiments.

filter_for_extended_form = `None`

list_filter = (('experiment', <class 'django.contrib.admin.filters.RelatedOnlyFieldListFilter'>),)

list_per_page = `30`

media

`disbi.admin.dataframe_replace_factory` (*replace*)

Factory for creating a mixin that replaces entries globally in an uploaded dataset.

Parameters **replace** (*tuple*) – A 2-tuple with the old and the new value.

Returns *Dataset* – The new dataset with the replaced values.

`disbi.admin.disbiresource_factory` (*mymodel, myfields, myimport_id_fields, mywidgets=None*)

Return a resource class with the given meta options and the validation hook.

`disbi.admin.inline_factory` (*proxy, inline_type='tabular'*)

Create an inline class from a proxy Model.

Parameters **proxy** (*Model*) – The proxy or just the normal model from which the inline class is created.

Keyword Arguments **inline_type** (*str*) – The type of Inline that should be created. Defaults to 'tabular'.

Returns *InlineModelAdmin* – The created class.

Raises `ValueError`

8.2.2 disbi.apps module

class `disbi.apps.DisbiConfig` (*app_name, app_module*)

Bases: `django.apps.config.AppConfig`

name = 'disbi'

8.2.3 disbi.cache_table module

Handles the caching of joined tables in the DB.

`disbi.cache_table.check_for_table_change` (*exp_model*, *check_for*)

Wrapper for checking whether data in DB tables has changed.

Parameters `check_for` (*str*) – Either `bio` for checking all tables that belong to *BiologicalModel* or `data` for checking all tables that belong to *MeasurementModel*.

`disbi.cache_table.check_table` (*dbtables*)

Check whether DB tables changed since the last time.

Parameters `dbtables` (*iterable of str*) – The tables that should be checked.

Returns *bool* – True if at least one table changed, else False.

`disbi.cache_table.drop_datatables` (*app_label*)

Drop all cached datatables.

Parameters `app_label` (*str*) – The name of the app the tables belong to.

`disbi.cache_table.get_table_names_by_pattern` (*pattern*)

Get all tables from DB that match a pattern.

Parameters `pattern` (*str*) – An SQL string or pattern with placeholders.

Returns *tuple* – The matched table names.

`disbi.cache_table.reconstruct_backbone_table` (*app_label*)

Reconstruct the prejoined backbone table of the biological models.

8.2.4 disbi.db_utils module

Helper functions for performing operations circumventing the ORM layer.

`disbi.db_utils.db_table_exists` (*table_name*)

Check whether a table with a specific name exists in the DB.

Parameters `table_name` (*str*) – The name of the table to check for.

Returns *bool* – True if the table exists, else False.

`disbi.db_utils.dictfetchall` (*cursor*)

Return all rows from a cursor as a dict.

`disbi.db_utils.exec_query` (*sql*, *parameters=None*)

Execute a plain SQL query.

Use parameterized query if parameters are given.

Parameters `sql` (*str*) – The SQL query.

Keyword Arguments `parameters` (*iterable*) – An iterable of parameters, that will be autoescaped.

`disbi.db_utils.from_db` (*sql*, *parameters=None*, *fetch_as='ordereddict'*)

Fetch values from the DB, given a SQL query.

Parameters `sql` (*str*) – The SQL statement.

Keyword Arguments

- **parameters** – Parameters for a parametrized query. If given *sql* must contain the appropriate placeholders. Defaults to None.
- **fetch_as** (*str*) – The data type as which the values should be fetched. Choices are ‘ordereddict’, ‘dict’, ‘namedtuple’ and ‘tuple’.

Raises `ValueError` – If a unrecognized value for `fetch_as` is given.

`disbi.db_utils.get_columnnames(table_name)`

Get the column names of a DB table.

Parameters `table_name` (*str*) – The name of the table.

Returns *tuple* – The column names.

`disbi.db_utils.get_field_query_name(model, field)`

Format the DB column name of a field with the DB table of its model.

`disbi.db_utils.get_fk_query_name(model, related_model)`

Format the DB column name of a foreign key field of a model with the DB table of the model. Finds the foreign key relating to related model automatically, but assumes that there is only one related field.

Parameters

- **model** (*Model*) – The model for which the foreign key field is searched.
- **related_model** (*Model*) – A model related to *model*.

Returns *str* – The formatted foreign key column name.

`disbi.db_utils.get_m2m_field(intermediary_model, related_model)`

Get the field of an intermediary model, that constitutes the relation to *related_model*.

`disbi.db_utils.get_pk_query_name(model)`

Format the primary key column of a model with its DB table.

`disbi.db_utils.namedtuplefetchall(cursor)`

Return all rows from a cursor as a namedtuple.

`disbi.db_utils.orderreddictfetchall(cursor)`

Return all rows from a cursor as an OrdreDict.

8.2.5 disbi.disbimodels module

Normal Django models with a few custom options for configuration.

If you have custom model classes that need these options, add them here and create a child class of the appropriate options class and your custom model class.

```
class disbi.disbimodels.BigIntegerField(di_show=False, di_display_name=None,
                                       di_hr_primary_key=False, di_choose=False,
                                       di_combinable=False, *args, **kwargs)
```

Bases: `disbi.disbimodels.Options`, `django.db.models.fields.BigIntegerField`

BigIntegerField with custom DISBi options.

```
class disbi.disbimodels.BinaryField(di_show=False, di_display_name=None,
                                    di_hr_primary_key=False, di_choose=False,
                                    di_combinable=False, *args, **kwargs)
```

Bases: `disbi.disbimodels.Options`, `django.db.models.fields.BinaryField`

BinaryField with custom DISBi options.

```
class disbi.disbimodels.BooleanField(di_exclude=False, di_show=False,
                                     di_display_name=None, di_hr_primary_key=False,
                                     di_choose=False, di_combinable=False, *args,
                                     **kwargs)
```

Bases: `disbi.disbimodels.ExcludeOptions`, `django.db.models.fields.BooleanField`

BooleanField with custom DISBi and exclude options.

```
class disbi.disbimodels.CharField (di_show=False, di_display_name=None,
                                   di_hr_primary_key=False, di_choose=False,
                                   di_combinable=False, *args, **kwargs)
```

Bases: *disbi.disbimodels.Options*, `django.db.models.fields.CharField`

CharField with custom DISBi options.

```
class disbi.disbimodels.CommaSeparatedIntegerField (di_show=False,
                                                    di_display_name=None,
                                                    di_hr_primary_key=False,
                                                    di_choose=False,
                                                    di_combinable=False, *args,
                                                    **kwargs)
```

Bases: *disbi.disbimodels.Options*, `django.db.models.fields.CommaSeparatedIntegerField`

CommaSeparatedIntegerField with custom DISBi options.

```
class disbi.disbimodels.DateField (di_show=False, di_display_name=None,
                                     di_hr_primary_key=False, di_choose=False,
                                     di_combinable=False, *args, **kwargs)
```

Bases: *disbi.disbimodels.Options*, `django.db.models.fields.DateField`

DateField with custom DISBi options.

```
class disbi.disbimodels.DateTimeField (di_show=False, di_display_name=None,
                                          di_hr_primary_key=False, di_choose=False,
                                          di_combinable=False, *args, **kwargs)
```

Bases: *disbi.disbimodels.Options*, `django.db.models.fields.DateTimeField`

DateTimeField with custom DISBi options.

```
class disbi.disbimodels.DecimalField (di_show=False, di_display_name=None,
                                         di_hr_primary_key=False, di_choose=False,
                                         di_combinable=False, *args, **kwargs)
```

Bases: *disbi.disbimodels.Options*, `django.db.models.fields.DecimalField`

DecimalField with custom DISBi options.

```
class disbi.disbimodels.DurationField (di_show=False, di_display_name=None,
                                          di_hr_primary_key=False, di_choose=False,
                                          di_combinable=False, *args, **kwargs)
```

Bases: *disbi.disbimodels.Options*, `django.db.models.fields.DurationField`

DurationField with custom DISBi options.

```
class disbi.disbimodels.EmailField (di_show=False, di_display_name=None,
                                       di_hr_primary_key=False, di_choose=False,
                                       di_combinable=False, *args, **kwargs)
```

Bases: *disbi.disbimodels.Options*, `django.db.models.fields.EmailField`

EmailField with custom DISBi options.

```
class disbi.disbimodels.EmptyCharField (di_empty=None, di_show=True,
                                           di_display_name=None, di_hr_primary_key=False,
                                           di_choose=False, di_combinable=False, *args,
                                           **kwargs)
```

Bases: *disbi.disbimodels.Options*, `django.db.models.fields.CharField`

FloatField with custom DISBi options and the option to add an empty value displayer.

```
class disbi.disbimodels.ExcludeOptions (di_exclude=False, di_show=False,
                                          di_display_name=None, di_hr_primary_key=False,
                                          di_choose=False, di_combinable=False, *args,
                                          **kwargs)
```

Bases: `disbi.disbimodels.Options`

Adds the `exclude` option, to exclude rows where this field evaluates to `False`. Should be only used on Bool fields.

```
class disbi.disbimodels.FileField(di_show=False, di_display_name=None,
                                  di_hr_primary_key=False, di_choose=False,
                                  di_combinable=False, *args, **kwargs)
```

Bases: `disbi.disbimodels.Options`, `django.db.models.fields.files.FileField`

FileField with custom DISBi options.

```
class disbi.disbimodels.FilePathField(di_show=False, di_display_name=None,
                                       di_hr_primary_key=False, di_choose=False,
                                       di_combinable=False, *args, **kwargs)
```

Bases: `disbi.disbimodels.Options`, `django.db.models.fields.FilePathField`

FilePathField with custom DISBi options.

```
class disbi.disbimodels.FloatField(di_show=False, di_display_name=None,
                                    di_hr_primary_key=False, di_choose=False,
                                    di_combinable=False, *args, **kwargs)
```

Bases: `disbi.disbimodels.Options`, `django.db.models.fields.FloatField`

FloatField with custom DISBi options.

```
class disbi.disbimodels.ForeignKey(to, di_show=False, di_display_name=None,
                                   di_hr_primary_key=False, di_choose=False,
                                   di_combinable=False, *args, **kwargs)
```

Bases: `disbi.disbimodels.RelationshipOptions`, `django.db.models.fields.related.ForeignKey`

ForeignKey with custom DISBi options.

```
class disbi.disbimodels.GenericIPAddressField(di_show=False, di_display_name=None,
                                                di_hr_primary_key=False, di_choose=False,
                                                di_combinable=False, *args, **kwargs)
```

Bases: `disbi.disbimodels.Options`, `django.db.models.fields.GenericIPAddressField`

GenericIPAddressField with custom DISBi options.

```
class disbi.disbimodels.ImageField(di_show=False, di_display_name=None,
                                    di_hr_primary_key=False, di_choose=False,
                                    di_combinable=False, *args, **kwargs)
```

Bases: `disbi.disbimodels.Options`, `django.db.models.fields.files.ImageField`

ImageField with custom DISBi options.

```
class disbi.disbimodels.IntegerField(di_show=False, di_display_name=None,
                                       di_hr_primary_key=False, di_choose=False,
                                       di_combinable=False, *args, **kwargs)
```

Bases: `disbi.disbimodels.Options`, `django.db.models.fields.IntegerField`

IntegerField with custom DISBi options.

```
class disbi.disbimodels.ManyToManyField(to, di_show=False, di_display_name=None,
                                          di_hr_primary_key=False, di_choose=False,
                                          di_combinable=False, *args, **kwargs)
```

Bases: `disbi.disbimodels.RelationshipOptions`, `django.db.models.fields.related.ManyToManyField`

ManyToManyField with custom DISBi options.

```
class disbi.disbimodels.NullBooleanField(di_exclude=False, di_show=False,
                                           di_display_name=None, di_hr_primary_key=False,
                                           di_choose=False, di_combinable=False, *args,
                                           **kwargs)
```

Bases: `disbi.disbimodels.ExcludeOptions`, `django.db.models.fields.NullBooleanField`

NullBooleanField with custom DISBi and exclude options.

```
class disbi.disbimodels.OneToOneField(to, di_show=False, di_display_name=None,
                                       di_hr_primary_key=False, di_choose=False,
                                       di_combinable=False, *args, **kwargs)
```

Bases: *disbi.disbimodels.RelationshipOptions*, `django.db.models.fields.related.OneToOneField`

OneToOneField with custom DISBi options.

```
class disbi.disbimodels.Options(di_show=False, di_display_name=None,
                                di_hr_primary_key=False, di_choose=False,
                                di_combinable=False, *args, **kwargs)
```

Bases: `object`

```
class disbi.disbimodels.PositiveIntegerField(di_show=False, di_display_name=None,
                                             di_hr_primary_key=False, di_choose=False,
                                             di_combinable=False, *args, **kwargs)
```

Bases: *disbi.disbimodels.Options*, `django.db.models.fields.PositiveIntegerField`

PositiveIntegerField with custom DISBi options.

```
class disbi.disbimodels.PositiveSmallIntegerField(di_show=False,
                                                  di_display_name=None,
                                                  di_hr_primary_key=False,
                                                  di_choose=False,
                                                  di_combinable=False, *args,
                                                  **kwargs)
```

Bases: *disbi.disbimodels.Options*, `django.db.models.fields.PositiveSmallIntegerField`

PositiveSmallIntegerField with custom DISBi options.

```
class disbi.disbimodels.RelationshipOptions(to, di_show=False, di_display_name=None,
                                             di_hr_primary_key=False, di_choose=False,
                                             di_combinable=False, *args, **kwargs)
```

Bases: `object`

```
class disbi.disbimodels.SlugField(di_show=False, di_display_name=None,
                                   di_hr_primary_key=False, di_choose=False,
                                   di_combinable=False, *args, **kwargs)
```

Bases: *disbi.disbimodels.Options*, `django.db.models.fields.SlugField`

SlugField with custom DISBi options.

```
class disbi.disbimodels.SmallIntegerField(di_show=False, di_display_name=None,
                                           di_hr_primary_key=False, di_choose=False,
                                           di_combinable=False, *args, **kwargs)
```

Bases: *disbi.disbimodels.Options*, `django.db.models.fields.SmallIntegerField`

SmallIntegerField with custom DISBi options.

```
class disbi.disbimodels.TextField(di_show=False, di_display_name=None,
                                  di_hr_primary_key=False, di_choose=False,
                                  di_combinable=False, *args, **kwargs)
```

Bases: *disbi.disbimodels.Options*, `django.db.models.fields.TextField`

TextField with custom DISBi options.

```
class disbi.disbimodels.TimeField(di_show=False, di_display_name=None,
                                   di_hr_primary_key=False, di_choose=False,
                                   di_combinable=False, *args, **kwargs)
```

Bases: *disbi.disbimodels.Options*, `django.db.models.fields.TimeField`

TimeField with custom DISBi options.

```
class disbi.disbimodels.URLField(di_show=False, di_display_name=None,  
                                di_hr_primary_key=False, di_choose=False,  
                                di_combinable=False, *args, **kwargs)
```

Bases: `disbi.disbimodels.Options`, `django.db.models.fields.URLField`

URLField with custom DISBi options.

```
class disbi.disbimodels.UUIDField(di_show=False, di_display_name=None,  
                                  di_hr_primary_key=False, di_choose=False,  
                                  di_combinable=False, *args, **kwargs)
```

Bases: `disbi.disbimodels.Options`, `django.db.models.fields.UUIDField`

UUIDField with custom DISBi options.

8.2.6 disbi.exceptions module

Custom DISBI exceptions.

```
exception disbi.exceptions.NoRelatedMeasurementModel(exp, *args, **kwargs)
```

Bases: `Exception`

Raise if an experiment has no data attached to it.

```
exception disbi.exceptions.NotFoundError
```

Bases: `Exception`

Raise if a value for a variable could not be set.

```
exception disbi.exceptions.NotSupportedError
```

Bases: `Exception`

Raise if a requested operation is not supported.

8.2.7 disbi.experiment_filter module

Functions for filtering the experiments based on conditions.

```
disbi.experiment_filter.combine_conditions(conditions, combinable_conditions)
```

Apply `combine_on_sep()` to each value in a dictionary.

Parameters `conditions` (*dict*) – Dictionary with a list of condition values.

Returns *dict* – A dictionary with the same keys and lists of combined values.

```
disbi.experiment_filter.combine_on_sep(items, separator)
```

Combine each item with each other item on a *separator*.

Parameters

- **items** (*list*) – A list or iterable that remembers order.
- **separator** (*str*) – The SEPARATOR the items will be combined on.

Returns *list* – A list with all the combined items as strings.

```
disbi.experiment_filter.get_experiments_by_condition(conditions, experiment_model)
```

Return a set of experiments that match the conditions.

Parameters `conditions` (*dict*) – A dictionary with conditions as keys and a list of values.

Returns *set* – A set of experiments that match the conditions.

`disbi.experiment_filter.get_requested_experiments` (*formset_list*, *experiment_model*)

Return all experiments that match the request from the form.

Parameters `formset_list` (*list*) – Formsets containing POST data.

Returns *set* – The union of the set of the directly requested experiments and those that matched the requested conditons.

`disbi.experiment_filter.lookup_format` (*dic*)

Return a dictionary readily useable for filter kwargs.

Parameters `dic` (*dict*) – The dictionary to be formatted.

Returns *dict* – The same dictionary with the keys concatenated with ‘__in’

8.2.8 disbi.forms module

Forms used throughout the DISBi app.

`disbi.forms.construct_direct_select_form` (*model*)

Construct a form for directly selecting model instances.

Parameters `model` (*models.Model*) – A Django model, for which the form is constructed.

`disbi.forms.construct_foreign_select_field` (*model*, *field*)

Construct a form for directly selecting related model instances.

Parameters

- `model` (*models.Model*) – A Django model, for which the form is constructed.
- `field` (*fields.related.ForeignKey*) – The field of the realated instances.

`disbi.forms.construct_forms` (*experiment_model*)

Wrapper for `construct_modelfieldsform` with appropriate arguments.

`disbi.forms.construct_modelfieldsform` (*model*, *exclude=['id']*, *direct_select=False*)

Construct forms based on a model and available values in the DB.

Parameters `model` (*models.Model*) – A Django model, for which the forms are constructed.

Keyword Arguments

- `exclude` (*iterable*) – Fields for which no forms should be constructed.
- `direct_select` (*bool*) – Determines whether a form for directly selecting model instances is constructed.

Returns *list* – A list of namedtuples with the form classes and the prefix.

`disbi.forms.foldchange_form_factory` (*experiments*)

Create a form with two fields for selecting experiments.

The fields are select widgets allowing to calculate a fold change between the two.

Parameters `experiments` (*Queryset*) – The selectable experiments.

Returns *Form* – The constructed form.

`disbi.forms.make_ChoiceField` (*model*, *attribute*, *label=None*, *empty_choice=None*)

Return a ChoiceField and the maxiumn number of choices.

The ChoiceField contains all distinct values of an attribute of model. Additionally a NULL option is inserted as first choice.

Parameters

- **attribute** (*Field*) – The attribute of the model.
- **label** (*str*) – The label used when displaying the form (default None).
- **empty_choice** (*tuple*) – 2-tuple of a value label pair, used for enabling the user to choose an empty condition, e.g. no stress.

Returns *ChoiceField* – A Choicefield based on the available options in the DB. *int*: The number of available options.

Raises *IndexError* – Raises error when no entries are found in the DB.

8.2.9 disbi.join module

Contains the class *Relations* that stores relations between models and joins them together appropriately.

class `disbi.join.Relations` (*app_label*, *model_superclass=None*)

Bases: `object`

Stores relations of models and has the ability to join them.

create_joined_table ()

Execute the the SQL JOIN and create a table thereof.

get_related_metamodels (*model*)

Get all models related to *model* of a superclass.

is_tree

Determine whether the *relation_map* is a tree.

Perform some setup and then call the depth first search, starting from an arbitrary node.

Returns *bool* – True if the graph is a tree. If it contains at least on cycle or is not connected, return False.

start_join ()

Start the join process between all models.

8.2.10 disbi.models module

class `disbi.models.BiologicalModel` (**args*, ***kwargs*)

Bases: `django.db.models.base.Model`

Baseclass for clustering the biological entities.

class `Meta`

Bases: `object`

abstract = False

class `disbi.models.Checksum` (**args*, ***kwargs*)

Bases: `django.db.models.base.Model`

Model for storing the checksums of other tables for checking whether data has changed.

exception `DoesNotExist`

Bases: `django.core.exceptions.ObjectDoesNotExist`

exception `Checksum.MultipleObjectsReturned`

Bases: `django.core.exceptions.MultipleObjectsReturned`

Checksum.**checksum**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

Checksum.**id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

Checksum.**objects** = <django.db.models.manager.Manager object>

Checksum.**table_name**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

class disbi.models.**DisbiExperiment**

Bases: `object`

Mixin for managing experiments.

result_view()

Creates a row with information that should be displayed in the data view. Override in your app, if you need specific information in the table.

Returns *OrderedDict* – Contains the information for one row of the result table.

view()

Construct an *OrderedDict* based on a tuple of column names.

Each column name is either expected to be a method or an attribute of the object. For the keys of the dict, the *short_description* is preferred. For attributes *verbose_name* is preferred over *name*.

Parameters `cols` (*tuple*) – The column names.

Returns *OrderedDict* – The external representation or view of the experiment object.

class disbi.models.**DisbiExperimentMetaInfo** (*args, **kwargs)

Bases: `object`

Mixin for Experiment proxy model, that fetches additionaly information about the experiment when instantiated.

class **Meta**

Bases: `object`

proxy = **True**

class disbi.models.**MeasurementModel** (*args, **kwargs)

Bases: `django.db.models.base.Model`

Base class for clustering the measurement models.

class **Meta**

Bases: `object`

abstract = **False**

MeasurementModel.**experiment**

Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a `ForwardManyToOneDescriptor` instance.

MeasurementModel.**experiment_id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

class disbi.models.**MetaModel** (*args, **kwargs)

Bases: django.db.models.base.Model

Baseclass for clustering the entities with meta information.

Meta informations are considered here as entities that can not be measured in an experiment.

class **Meta**

Bases: object

abstract = False

8.2.11 disbi.option_utils module

Module for treating the custom DISBi options attached to the model fields.

disbi.option_utils.**get_display_name** (field)

Get the name used to display a field in the external representation.

Parameters **field** (*Field*) – The respective field.

Returns *str* – The display_name used for external representation.

disbi.option_utils.**get_models_of_superclass** (app_label, model_superclasses, intermediary=False)

Get all the models that derive from one superclass and include the intermediary models for N:M related models.

Parameters

- **app_label** (*str*) – The label of the app the models live in.
- **model_superclasses** (*iterable of type*) – The classes from which all models of interest derive.

Keyword Arguments **intermediary** (*bool*) – Determines whether models should be included.

Returns *list* – All models that derive from *model_superclass*. Plus their intermediary models that are no proxys if *intermediary* is True.

8.2.12 disbi.result module

Class for getting the joined table with data based on a list of experiments.

class disbi.result.**DataResult** (requested_experiments, experiment_meta_model)

Bases: object

Constructs the datatable based on the request from the filter view.

DB_FUNCTION_ZERO = 'divide_without_zeroerr'

DB_PRECISION = "9.9999EEEE"

add_foldchange (exps_for_fc, fetch_as='orderdict')

Add the fold change to a base table.

construct_SELECT_AS (exp)

Construct part of a SQL statement for alias DB columns with their display name.

Parameters **exp** (*Experiment*) – The experiment.

Returns *str* – The partial statement for aliasing the columns.

construct_base_table ()

Construct the SQL statement for creating the base table.

Returns *str* – The SQL statement for creating the base table.

construct_exptable (*exp*)

Construct the subquery for producing selection of all data points mapping to one experiment.

Only data points which foreign key matches the experiment will be selected. Data points for which an exclude column evaluates to False will be excluded.

Parameters **exp** (*disbimodels.Experiment*) – The experiment for which the data is selected.

Returns *str* – A SQL statement for the selection of the datapoints.

construct_result_table (*biomodels*)

Construct the SQL statement for getting the result table.

For each biological model, the respective experiments will be filtered. For those experiments a subquery is constructed, that is then LEFT JOINed into the prejoined backbone table. Only rows that have at least one data point are preserved.

Parameters **biomodels** (*list*) – List of Biological models.

Returns *str* – The SQL statement for the result table.

create_base_table (*table_name*)

Create the base table and write it to the DB.

Parameters **table_name** (*str*) – The name under which the table should be created.

Returns *None* – This is a procedure.

get_colnames (*model*)

Get a list of all DB columns with di_show=True.

Parameters **model** (*models.Model*) – The model for which the fields are retrieved.

Returns *list* – A list of strings with the DB column names.

get_display_names (*exp*)

Get the display names for the columns of a MeasurementModel of an experiment.

Parameters **exp** (*Experiment*) – The experiment.

Returns *tuple* – A tuple of strings containing the display names suffixed by the experiment id.

get_exp_columns (*wanted_exps*)

Get column of respective experiment.

get_foldchange (*exps_for_fc*)

Get only the fold change column.

get_notnull_column (*exp*)

Get a column that can not be NULL and will be shown.

Parameters **exp** (*Experiment*) – The experiment for which the column should be retrieved.

Returns

models.Field – The first field that will be shown and is neither NULL or blank.

Raises `NotFoundError`

get_or_create_base_table (*fetch_as='ordereddict'*)

Retrieve the base table from the DB. Create it if it does not exist.

Returns The values fetched from the DB.

get_show_columns (*model*)

Get a list of DB columns that should be shown in the result table.

Parameters **model** (*models.Model*) – A Django model with custom DISBI options.

Returns *list* – List of strings containing the DB column names of the fields to be shown.

wrap_in_func (*func, *cols*)

Pass column names as arguments to DB function.

8.2.13 disbi.utils module

Some utility functions used throughout the DISBi app.

`disbi.utils.camelize` (*string, uppercase_first_letter=True*)

Convert a string with underscores to a camelCase string.

Inspired by `inflection.camelize()` but even seems to run a little faster.

Parameters

- **string** (*str*) – The string to be converted.
- **uppercase_first_letter** (*bool*) – Determines whether the first letter of the string should be capitalized.

Returns *str* – The camelized string.

`disbi.utils.clean_set` (*cleaned_formset_data*)

Return a dictionary with keys from POST and lists as values.

The `cleaned_data` of a formset is a list of dictionaries that can have overlapping keys. The function changes that into a dictionary where each key corresponds to list of values that belonged to the same key.

Returns *dict* – Dictionary with items joined on keys.

`disbi.utils.construct_none_displayer` (*entries, placeholder='-'*)

Return a string to be used as a placeholder for an empty option.

The length of the placeholder is based on the length of the longest option in a list of entries.

`disbi.utils.get_choices` (*choice_tup, style='db'*)

Return the choices given on a model Field.

Parameters **choice_tup** – The choice tuple given in the model.

Keyword Arguments **style** – Determines whether the human readable “display” values should be returned or those from the “db”.

`disbi.utils.get_hr_val` (*choices, db_val*)

Get the human readable value for the DB value from a choice tuple.

Parameters

- **choices** (*tuple*) – The choice tuple given in the model.
- **db_val** – The respective DB value.

Returns The matching human readable value.

`disbi.utils.get_id_str(objects, delimiter='_')`

Get a string of sorted ids, separated by a delimiter.

Parameters `objects` (*Model*) – An iterable of model instances.

Keyword Arguments `delimiter` (*str*) – The string the ids will be joined on.

Returns *str* – The joined and sorted id string.

`disbi.utils.get_ids(string, delimiter='_')`

Get sorted *ids*.

`disbi.utils.get_optgroups(choice_tup, style='db')`

Parse the choices given on a model Field and map them to their groups.

Parameters `choice_tup` – The choice tuple given in the model.

Keyword Arguments `style` – Determines whether the human readable “display” values should be returned or those from the “db”.

Returns *dict* – A list of choices mapped to their optgroup.

`disbi.utils.get_unique(items)`

Get a list of unique items, even for non hashable items.

`disbi.utils.merge_dicts(a, b)`

Merge two dicts without modifying them inplace.

Parameters

- `a` (*dict*) – The first dict.
- `b` (*dict*) – The second dict. Overrides `a` on conflicts.

Returns *dict* – A merged dictionary.

`disbi.utils.object_view(obj, cols)`

Construct an external representation of an object based on a tuple of field/column names.

Each column name is either expected to be a method or an attribute of the object. For the keys of the dict, the *short_description* is preferred. For attributes *verbose_name* is preferred over *name*.

Parameters

- `obj` – Any object with *cols* as attributes.
- `cols` (*tuple*) – The column names.

Returns *OrderedDict* – The headers as keys and the entries as values.

`disbi.utils.remove_optgroups(choices)`

Remove optgroups from a choice tuple.

Parameters `choices` (*tuple*) – The choice tuple given in the model.

Returns The *n* by 2 choice tuple without optgroups.

`disbi.utils.reverse_dict(dic)`

Return a reversed dictionary.

Each former value will be the key of a list of all keys that were mapping to it in the old dict.

`disbi.utils.sort_by_other(sequence, order)`

Order a list a another list that contains the desired order.

Parameters

- `sequence` (*list*) – The list that is ordered. All items must be contained in order.

- **order** (*list*) – The list containing the order.

Returns *list* – The sequence ordered according to order.

`disbi.utils.zip_dicts(a, b)`

Merge two dicts, choose none empty value on key conflicts.

The dicts are not modified inplace, but returned.

Parameters

- **a** (*dict*) – The first dict.
- **b** (*dict*) – The second dict. Overrides a on conflicts, when both values are none empty.

Returns *dict* – A merged dictionary.

8.2.14 disbi.validators module

This file collects field validators that are common to the domain of systems biology.

`disbi.validators.validate_flux(value)`

Validate that a value is in [-1000, 1000].

`disbi.validators.validate_probabilty(value)`

Validate that a value is in [0, 1].

8.2.15 disbi.views module

DISBi views that need to subclassed and configured with the appropriate experiment models by a concrete app.

class `disbi.views.DisbiCalculateFoldChangeView(**kwargs)`

Bases: `django.views.generic.base.View`

View for calculating the fold change between two experiments.

experiment_meta_model = None

experiment_model = None

post (*request, exp_id_str*)

Return new data for the datatable with new columns for calculated fold changes.

Parameters

- **request** – The WSGI request.
- **exp_id_str** – The ids of all requested experiments from the table view joined on “_”.

Returns *JSONResponse* – The new data for the datatable or the error message.

class `disbi.views.DisbiComparePlotView(**kwargs)`

Bases: `django.views.generic.base.View`

View for generating a scatter plot that compares two experiments.

experiment_meta_model = None

experiment_model = None

post (*request, exp_id_str*)

Get the scatter plot comparing two experiments.

If the data model of the two experiments matches, a plot is generated, else an error message is raised.

Parameters

- **request** – The WSGI request.
- **exp_id_str** – The ids of all requested experiments from the table view joined on “_”.

Returns *JSONResponse* – The plot image SVG or the error message.

```
class disbi.views.DisbiDataView (**kwargs)
    Bases: django.views.generic.base.View
```

View for creating the the basic data view without the data table.

experiment_meta_model = None

```
get (request, exp_id_str)
    View for creating and displaying the result table.
```

Parameters

- **request** – The WSGI request.
- **exp_id_str** – The ids of all requested experiments from the table view joined on “_”.

Returns *TemplateResponse* – The template for the data view with the appropriate form and the result table with information about the experiments.

```
class disbi.views.DisbiDistributionPlotView (**kwargs)
    Bases: django.views.generic.base.View
```

View for generating a histogram of the distribution of a column in the data table.

experiment_meta_model = None

experiment_model = None

```
post (request, exp_id_str)
    Get the distribution of a column as a histogram.
```

If a non fold change column is plotted the matching data is fetched from the DB with the ORM. If a fold change column is selected, a new *DataResult* object is instantiated and only the fold change column is retrieved from the result table cached in the DB.

Parameters

- **request** – The WSGI request.
- **exp_id_str** – The ids of all requested experiments from the table view joined on “_”.

Returns *JSONResponse* – The plot image SVG or the error message.

```
class disbi.views.DisbiExpInfoView (**kwargs)
    Bases: django.views.generic.base.View
```

View for getting information about the experiments in the preview table.

experiment_model = None

```
post (request)
    Get information about matched experiments.
```

Parameters **request** – The WSGI request.

Returns *JSONResponse* – JSON object with number of matched experiments and a HTML table with information about those experiments.

class `disbi.views.DisbiExperimentFilterView` (**kwargs)

Bases: `django.views.generic.base.View`

View for showing dynamic formsets, allowing to choose all combinations.

experiment_model = None

get (*request*)

Returns *TemplateResponse* – The rendered forms without initial data.

post (*request*)

HttpResponseRedirect: For valid POST requests the client will be redirected to the appropriate data view.

class `disbi.views.DisbiGetTableData` (**kwargs)

Bases: `django.views.generic.base.View`

View for initially getting the data for the datatable.

experiment_meta_model = None

get (*request*, *exp_id_str*)

Return new data for the datatable with new columns for calculated fold changes.

Parameters

- **request** – The WSGI request.
- **exp_id_str** – The ids of all requested experiments from the table view joined on “_”.

Returns *JSONResponse* – The data for the datatable.

d

- disbi, 27
- disbi.admin, 28
- disbi.apps, 28
- disbi.cache_table, 28
- disbi.db_utils, 29
- disbi.disbimodels, 30
- disbi.exceptions, 34
- disbi.experiment_filter, 34
- disbi.forms, 35
- disbi.join, 36
- disbi.migrations, 27
- disbi.migrations.0001_initial, 27
- disbi.models, 36
- disbi.option_utils, 38
- disbi.result, 38
- disbi.templatetags, 27
- disbi.templatetags.custom_filters, 27
- disbi.templatetags.custom_template_tags,
27
- disbi.utils, 40
- disbi.validators, 42
- disbi.views, 42

A

abstract (disbi.models.BiologicalModel.Meta attribute), 36
 abstract (disbi.models.MeasurementModel.Meta attribute), 37
 abstract (disbi.models.MetaModel.Meta attribute), 38
 add_foldchange() (disbi.result.DataResult method), 38

B

BigIntegerField (class in disbi.disbimodels), 30
 BinaryField (class in disbi.disbimodels), 30
 BiologicalModel (class in disbi.models), 36
 BiologicalModel.Meta (class in disbi.models), 36
 BooleanField (class in disbi.disbimodels), 30

C

camelize() (in module disbi.utils), 40
 CharField (class in disbi.disbimodels), 30
 check_for_table_change() (in module disbi.cache_table), 28
 check_table() (in module disbi.cache_table), 29
 Checksum (class in disbi.models), 36
 checksum (disbi.models.Checksum attribute), 36
 Checksum.DoesNotExist, 36
 Checksum.MultipleObjectsReturned, 36
 clean_set() (in module disbi.utils), 40
 combine_conditions() (in module disbi.experiment_filter), 34
 combine_on_sep() (in module disbi.experiment_filter), 34
 CommaSeparatedIntegerField (class in disbi.disbimodels), 31
 construct_base_table() (disbi.result.DataResult method), 39
 construct_direct_select_form() (in module disbi.forms), 35
 construct_exptable() (disbi.result.DataResult method), 39
 construct_foreign_select_field() (in module disbi.forms), 35
 construct_forms() (in module disbi.forms), 35
 construct_modelfieldsform() (in module disbi.forms), 35

construct_none_displayer() (in module disbi.utils), 40
 construct_result_table() (disbi.result.DataResult method), 39
 construct_SELECT_AS() (disbi.result.DataResult method), 38
 create_base_table() (disbi.result.DataResult method), 39
 create_joined_table() (disbi.join.Relations method), 36

D

dataframe_replace_factory() (in module disbi.admin), 28
 DataResult (class in disbi.result), 38
 DateField (class in disbi.disbimodels), 31
 DateTimeField (class in disbi.disbimodels), 31
 DB_FUNCTION_ZERO (disbi.result.DataResult attribute), 38
 DB_PRECISION (disbi.result.DataResult attribute), 38
 db_table_exists() (in module disbi.db_utils), 29
 DecimalField (class in disbi.disbimodels), 31
 dependencies (disbi.migrations.0001_initial.Migration attribute), 27
 dictfetchall() (in module disbi.db_utils), 29
 disbi (module), 27
 disbi.admin (module), 28
 disbi.apps (module), 28
 disbi.cache_table (module), 28
 disbi.db_utils (module), 29
 disbi.disbimodels (module), 30
 disbi.exceptions (module), 34
 disbi.experiment_filter (module), 34
 disbi.forms (module), 35
 disbi.join (module), 36
 disbi.migrations (module), 27
 disbi.migrations.0001_initial (module), 27
 disbi.models (module), 36
 disbi.option_utils (module), 38
 disbi.result (module), 38
 disbi.templatetags (module), 27
 disbi.templatetags.custom_filters (module), 27
 disbi.templatetags.custom_template_tags (module), 27
 disbi.utils (module), 40
 disbi.validators (module), 42

disbi.views (module), 42
 DisbiCalculateFoldChangeView (class in disbi.views), 42
 DisbiComparePlotView (class in disbi.views), 42
 DisbiConfig (class in disbi.apps), 28
 DisbiDataAdmin (class in disbi.admin), 28
 DisbiDataView (class in disbi.views), 43
 DisbiDistributionPlotView (class in disbi.views), 43
 DisbiExperiment (class in disbi.models), 37
 DisbiExperimentFilterView (class in disbi.views), 43
 DisbiExperimentMetaInfo (class in disbi.models), 37
 DisbiExperimentMetaInfo.Meta (class in disbi.models), 37
 DisbiExpInfoView (class in disbi.views), 43
 DisbiGetTableData (class in disbi.views), 44
 disbiresource_factory() (in module disbi.admin), 28
 drop_datatables() (in module disbi.cache_table), 29
 DurationField (class in disbi.disbimodels), 31

E

EmailField (class in disbi.disbimodels), 31
 EmptyCharField (class in disbi.disbimodels), 31
 ExcludeOptions (class in disbi.disbimodels), 31
 exec_query() (in module disbi.db_utils), 29
 experiment (disbi.models.MeasurementModel attribute), 37
 experiment_id (disbi.models.MeasurementModel attribute), 37
 experiment_meta_model (disbi.views.DisbiCalculateFoldChangeView attribute), 42
 experiment_meta_model (disbi.views.DisbiComparePlotView attribute), 42
 experiment_meta_model (disbi.views.DisbiDataView attribute), 43
 experiment_meta_model (disbi.views.DisbiDistributionPlotView attribute), 43
 experiment_meta_model (disbi.views.DisbiGetTableData attribute), 44
 experiment_model (disbi.views.DisbiCalculateFoldChangeView attribute), 42
 experiment_model (disbi.views.DisbiComparePlotView attribute), 42
 experiment_model (disbi.views.DisbiDistributionPlotView attribute), 43
 experiment_model (disbi.views.DisbiExperimentFilterView attribute), 44
 experiment_model (disbi.views.DisbiExpInfoView attribute), 43

F

FileField (class in disbi.disbimodels), 32
 FilePathField (class in disbi.disbimodels), 32
 filter_for_extended_form (disbi.admin.DisbiDataAdmin attribute), 28
 FloatField (class in disbi.disbimodels), 32

foldchange_form_factory() (in module disbi.forms), 35
 ForeignKey (class in disbi.disbimodels), 32
 from_db() (in module disbi.db_utils), 29

G

GenericIPAddressField (class in disbi.disbimodels), 32
 get() (disbi.views.DisbiDataView method), 43
 get() (disbi.views.DisbiExperimentFilterView method), 44
 get() (disbi.views.DisbiGetTableData method), 44
 get_choices() (in module disbi.utils), 40
 get_colnames() (disbi.result.DataResult method), 39
 get_columnnames() (in module disbi.db_utils), 29
 get_display_name() (in module disbi.option_utils), 38
 get_display_names() (disbi.result.DataResult method), 39
 get_exp_columns() (disbi.result.DataResult method), 39
 get_experiments_by_condition() (in module disbi.experiment_filter), 34
 get_field_query_name() (in module disbi.db_utils), 30
 get_fk_query_name() (in module disbi.db_utils), 30
 get_foldchange() (disbi.result.DataResult method), 39
 get_hr_val() (in module disbi.utils), 40
 get_id_str() (in module disbi.utils), 41
 get_ids() (in module disbi.utils), 41
 get_item() (in module disbi.templatetags.custom_filters), 27
 get_item_by_idx() (in module disbi.templatetags.custom_filters), 27
 get_list() (in module disbi.templatetags.custom_filters), 27
 get_m2m_field() (in module disbi.db_utils), 30
 get_models_of_superclass() (in module disbi.option_utils), 38
 get_notnull_column() (disbi.result.DataResult method), 39
 get_optgroups() (in module disbi.utils), 41
 get_or_create_base_table() (disbi.result.DataResult method), 40
 get_pk_query_name() (in module disbi.db_utils), 30
 get_related_metamodels() (disbi.join.Relations method), 36
 get_requested_experiments() (in module disbi.experiment_filter), 34
 get_show_columns() (disbi.result.DataResult method), 40
 get_table_names_by_pattern() (in module disbi.cache_table), 29
 get_unique() (in module disbi.utils), 41

I

id (disbi.models.Checksum attribute), 37
 ImageField (class in disbi.disbimodels), 32
 initial (disbi.migrations.0001_initial.Migration attribute), 27
 inline_factory() (in module disbi.admin), 28

IntegerField (class in disbi.disbimodels), 32
is_tree (disbi.join.Relations attribute), 36

L

list_filter (disbi.admin.DisbiDataAdmin attribute), 28
list_per_page (disbi.admin.DisbiDataAdmin attribute), 28
lookup_format() (in module disbi.experiment_filter), 35

M

make_ChoiceField() (in module disbi.forms), 35
ManyToManyField (class in disbi.disbimodels), 32
MeasurementModel (class in disbi.models), 37
MeasurementModel.Meta (class in disbi.models), 37
media (disbi.admin.DisbiDataAdmin attribute), 28
merge_dicts() (in module disbi.utils), 41
MetaModel (class in disbi.models), 38
MetaModel.Meta (class in disbi.models), 38
Migration (class in disbi.migrations.0001_initial), 27

N

name (disbi.apps.DisbiConfig attribute), 28
namedtuplefetchall() (in module disbi.db_utils), 30
nested_dict_as_table() (in module disbi.templatetags.custom_template_tags), 27
NoRelatedMeasurementModel, 34
NotFoundError, 34
NotSupportedError, 34
NullBooleanField (class in disbi.disbimodels), 32

O

object_view() (in module disbi.utils), 41
objects (disbi.models.Checksum attribute), 37
OneToOneField (class in disbi.disbimodels), 33
operations (disbi.migrations.0001_initial.Migration attribute), 27
Options (class in disbi.disbimodels), 33
orderreddictfetchall() (in module disbi.db_utils), 30

P

PositiveIntegerField (class in disbi.disbimodels), 33
PositiveSmallIntegerField (class in disbi.disbimodels), 33
post() (disbi.views.DisbiCalculateFoldChangeView method), 42
post() (disbi.views.DisbiComparePlotView method), 42
post() (disbi.views.DisbiDistributionPlotView method), 43
post() (disbi.views.DisbiExperimentFilterView method), 44
post() (disbi.views.DisbiExpInfoView method), 43
print_none() (in module disbi.templatetags.custom_template_tags), 28

proxy (disbi.models.DisbiExperimentMetaInfo.Meta attribute), 37

Python Enhancement Proposals
PEP 8, 19

R

reconstruct_backbone_table() (in module disbi.cache_table), 29
Relations (class in disbi.join), 36
RelationshipOptions (class in disbi.disbimodels), 33
remove_optgroups() (in module disbi.utils), 41
result_view() (disbi.models.DisbiExperiment method), 37
reverse_dict() (in module disbi.utils), 41

S

SlugField (class in disbi.disbimodels), 33
SmallIntegerField (class in disbi.disbimodels), 33
sort_by_other() (in module disbi.utils), 41
start_join() (disbi.join.Relations method), 36

T

table_name (disbi.models.Checksum attribute), 37
TextField (class in disbi.disbimodels), 33
TimeField (class in disbi.disbimodels), 33

U

URLField (class in disbi.disbimodels), 33
UUIDField (class in disbi.disbimodels), 34

V

validate_flux() (in module disbi.validators), 42
validate_probabilty() (in module disbi.validators), 42
view() (disbi.models.DisbiExperiment method), 37

W

wrap_in_func() (disbi.result.DataResult method), 40

Z

zip_dicts() (in module disbi.utils), 42