MouseDB Documentation

Release 0.1

Dave Bridges, Ph.D.

December 27, 2009

CONTENTS

Contents:

CHAPTER

ONE

MOUSEDB CONCEPTS

Data storage for MouseDB is separated into packages which contain information about animals, and information collected about animals. There is also a separate module for timed matings of animals. This document will describe the basics of how data is stored in each of these modules.

1.1 Animal Module

Animals are tracked as individual entities, and given associations to breeding cages to follow ancestry, and strains.

1.1.1 Animal

1.1.2 Breeding Cages

A breeding cage is defined as a set of one or more male and one or more female mice. Because of this, it is not always clear who the precise parentage of an animal is. If the parentage is known, then the Mother and Father fields can be set for a particular animal.

1.1.3 Strains

1.2 Data Module

Data (or measurements) can be stored for any type of measurement. Conceptually, several pieces of data belong to an experiment (for example several mice are measured at some time) and several experiments belong to a study. Measurements can be stored independent of experiments and experiments can be performed outside of the context of a study. It is however, perfered that measurements are stored within an experiment and experiments are stored within studies as this will greatly facilitate the organization of the data.

1.2.1 Studies

In general studies are a collection of experiments. These can be grouped together on the basis of animals and/or treatment groups. A study must have at least one treatment group, which defines the animals and their conditions.

1.2.2 Experiments

An experiment is a collection of measurements for a given set of animals. In general, an experiment is defined as a number of measurements take in a given day.

1.2.3 Measurements

A measurement is an animal, an assay and a measurement value. It can be associated with an experiment, or can stand alone as an individual value. Measurements can be viewed in the context of a study, an experiment, a treatment group or an animal by going to the appropriate page.

CHAPTER

TWO

MOUSEDB INSTALLATION

2.1 Configuration

MouseDB requires both a database and a webserver to be set up. Ideally, the database should be hosted separately from the webserver and MouseDB installation, but this is not necessary, as both can be used from the same server. If you are using a remote server for the database, it is best to set up a user for this database that can only be accessed from the webserver. If you want to set up several installations (ie for different users or different laboratories), you need separate databases and MouseDB installations for each. You will also need to set up the webserver with different addresses for each installation.

2.2 Software Dependencies

1. **MouseDB source code**. Download from one of the following:

1. http://github.com/davebridges/mousedb/downloads for the current release
2. http://github.com/davebridges/mousedb for the source code via Git

Downloading and/or unzipping will create a directory named mousedb. You can update to the newest revision at any time either using git or downloading and re-installing the newer version. Changing or updating software versions will not alter any saved data, but you will have to update the localsettings.py file (described below).

1. **Python**. Requires Version 2.6, is not yet compatible with Python 3.0. Download from http://www.python.org/download/.
2. **Django**. Download from http://www.djangoproject.com/download/
3. **Database software**. Typically MySQL is used, but PostgreSQL, Oracle or SQLite can also be used. You also need to install the python driver for this database (unless you are using SQLite, which is internal to Python 2.5+). See http://docs.djangoproject.com/en/dev/topics/install/database-installation - Django Database Installation Documentation for more information.

2.3 Database Setup

1. Create a new database. You need to record the user, password, host and database name. If you are using SQLite this step is not required.
2. Go to localsettings_empty.py and edit the settings:

- DATABASE_ENGINE: 'mysql', 'postgresql_psycopg2' or 'sqlite3 depending on the database software used.

- DATABASE_NAME: database name
- DATABASE_USER: database user
- DATABASE_PASSWORD: database password
- DATABASE_HOST: database host

1. Save this file as localsettings.py in the main MouseDB directory.

2.4 Web Server Setup

You need to set up a server to serve both the django installation and saved files. For the saved files. I recommend using apache for both. The preferred setup is to use Apache2 with mod_python. The following is a httpd.conf example where the code is placed in /usr/src/mousedb:

```
Alias /static /usr/src/mousedb/media
Alias /media /usr/src/mousedb/media
<Directory /usr/mousedb/media>
   Order allow,deny
   Allow from all
</Directory>
<Location "/mousedb/">
   SetHandler python-program
   PythonHandler django.core.handlers.modpython
   SetEnv DJANGO_SETTINGS_MODULE mousedb.settings
   SetEnv PYTHON_EGG_CACHE /var/www/eggs
   PythonOption django.root /mousedb
   PythonDebug On
   PythonPath "['/usr/src'] + sys.path"
   PythonInterpreter mousedb
</Location>
```

If you want to restrict access to these files, change the Allow from all directive to specific domains or ip addresses (for example Allow from 192.168.0.0/99 would allow from 192.168.0.0 to 192.168.0.99)

2.5 Final Configuration and User Setup

1. Go to mousedb/admin/auth/users/ and create users, selecting usernames, full names, password (or have the user set the password) and then choose group permissions.

CHAPTER

THREE

ANIMAL DATA ENTRY

3.1 Newborn Mice or Newly Weaned Mice

1. Go to Breeding Cages Tab
2. Click on Add/Wean Pups Button
3. Each row is a new animal. If you accidentaly enter an extra animal, check off the delete box then submit.
4. Leave extra lines blank if you have less than 10 mice to enter
5. If you need to enter more than 10 mice, enter the first ten and submit them. Go back and enter up to 10 more animals (10 more blank spaces will appear)

3.2 Newborn Mice

1. Enter Breeding Cage under Cage
2. Enter Strain
3. Enter Background (normally Mixed or C57BL/6-BA unless from the LY breeding cages in which case it is C57BL/6-LY5.2)
4. Enter Birthdate in format YYYY-MM-DD
5. Enter Generation and Backcross

3.3 Weaning Mice

1. If not previously entered, enter data as if newborn mice
2. Enter gender
3. Enter Wean Date in format YYYY-MM-DD
4. Enter new Cage number for Cage

3.4 Cage Changes (Not Weaning)

1. Find mouse either from animal list or strain list

2. Click the edit mouse button
3. Change the Cage, Rack and Rack Position as Necessary

3.5 Genotyping or Ear Tagging

1. Find mouse either from animal list or strain list, or through breeding cage
2. Click the edit mouse button or the Eartag/Genotype/Cage Change/Death Button
3. Enter the Ear Tag and/or select the Genotype from the Pull Down List

3.6 Marking Mice as Dead

3.6.1 Dead Mice (Single Mouse)

1. Find mouse from animal list or strain list
2. Click the edit mouse button
3. Enter the death date in format YYYY-MM-DD
4. Choose Cause of Death from Pull Down List

3.6.2 Dead Mice (Several Mice)

1. Find mice from breeding cages
2. Click the Eartag/Genotype/Cage Change/Death Button
3. Enter the death date in format YYYY-MM-DD
4. Choose the Cause of Death from Pull Down List

CHAPTER

FOUR

STUDIES AND EXPERIMENTAL SETUP

Set up a new study at /mousedb/admin/data/study/ selecting animals

You must put a description and select animals in one or more treatment groups

If you have more than 2 treatment groups save the first two, then two more empty slots will appear. For animals, click on the magnifying glass then find the animal in that treatment group and click on the MouseID. The number displayed now in that field will not be the MouseID, but don't worry its just a different number to describe the mouse. To add more animals, click on the magnifying glass again and select the next animal. There should be now two numbers, separated by commas in this field. Repeat to fill all your treatment groups. You must enter a diet and environment for each treatment. The other fields are optional, and should only be used if appropriate. Ensure for pharmaceutical, you include a saline treatment group.

CHAPTER

FIVE

MEASURMENT ENTRY

5.1 Studies

If this measurement is part of a study (ie a group of experiments) then click on the plus sign beside the study field and enter in the details about the study and treatment groups. Unfortunately until i can figure out how to filter the treatment group animals in the admin interface, at each of the subsequent steps you will see all the animals in the database (soon hopefully it will only be the ones as part of the study group).

5.2 Experiment Details

- Pick experiment date, feeding state and resarchers
- Pick animals used in this experiment (the search box will filter results)
- Fasting state, time, injections, concentration, experimentID and notes are all optional

5.3 Measurements

- There is room to enter 14 measurements. If you need more rows, enter the first 14 and select "Save and Continue Editing" and 14 more blank spots will appear.
- Each row is a measurement, so if you have glucose and weight for some animal that is two rows entered.
- For animals, click on the magnifying glass then find the animal in that treatment group and click on the MouseID. The number displayed now in that field will not be the MouseID, but don't worry its just a different number to describe the mouse.
- For values, the standard units (defined by each assay) are mg for weights, mg/dL for glucose and pg/mL for insulin). You must enter integers here (no decimal places). If you have several measurements (ie several glucose readings during a GTT, enter them all in one measurement row, separated by commas and *NO spaces*).

CHAPTER

SIX

AUTOMATED DOCUMENTATION

6.1 Data Package

6.1.1 Models

class **Assay** (**args, **kwargs*)

Assay(id, assay, assay_slug, notes, measurement_units)

exception **DoesNotExist**

args

message

exception **MultipleObjectsReturned**

args

message

delete ()

measurement_set

pk

prepare_database_save (*unused*)

save (*force_insert=False, force_update=False*)

Saves the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base (*raw=False, cls=None, origin=None, force_insert=False, force_update=False*)

Does the heavy-lifting involved in saving. Subclasses shouldn't need to override this method. It's separate from save() in order to hide the need for overrides of save() to pass around internal-only parameters ('raw', 'cls', and 'origin').

serializable_value (*field_name*)

Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

class **Diet** (**args, **kwargs*)

Diet(id, vendor_id, description, product_id, fat_content, protein_content, carb_content, irradiated, notes)

exception **DoesNotExist**

args

message

exception **MultipleObjectsReturned**

args

message

delete()

pk

prepare_database_save(*unused*)

save(*force_insert=False, force_update=False*)

Saves the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base(*raw=False, cls=None, origin=None, force_insert=False, force_update=False*)

Does the heavy-lifting involved in saving. Subclasses shouldn't need to override this method. It's separate from save() in order to hide the need for overrides of save() to pass around internal-only parameters ('raw', 'cls', and 'origin').

serializable_value(*field_name*)

Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

treatment_set

vendor

class **Environment** (**args, **kwargs*)

Environment(id, building, room, temperature, humidity, notes)

exception **DoesNotExist**

args

message

exception **MultipleObjectsReturned**

args

message

contact

delete()

pk

prepare_database_save(*unused*)

save(*force_insert=False, force_update=False*)
Saves the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base(*raw=False, cls=None, origin=None, force_insert=False, force_update=False*)
Does the heavy-lifting involved in saving. Subclasses shouldn't need to override this method. It's separate from save() in order to hide the need for overrides of save() to pass around internal-only parameters ('raw', 'cls', and 'origin').

serializable_value(*field_name*)
Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

treatment_set

class Experiment(**args, **kwargs*)
Experiment(id, date, notes, experimentID, feeding_state, fasting_time, injection, concentration, study_id)

exception DoesNotExist

args

message

exception MultipleObjectsReturned

args

message

animals

delete()

get_feeding_state_display(**moreargs, **morekwargs*)

get_injection_display(**moreargs, **morekwargs*)

get_next_by_date(**moreargs, **morekwargs*)

get_previous_by_date(**moreargs, **morekwargs*)

measurement_set

pk

prepare_database_save(*unused*)

researchers

save (*force_insert=False, force_update=False*)
Saves the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base (*raw=False, cls=None, origin=None, force_insert=False, force_update=False*)
Does the heavy-lifting involved in saving. Subclasses shouldn't need to override this method. It's separate from save() in order to hide the need for overrides of save() to pass around internal-only parameters ('raw', 'cls', and 'origin').

serializable_value (*field_name*)
Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

study

class Implantation (**args, **kwargs*)
Implantation(id, implant, vendor_id, product_id, notes)

exception DoesNotExist

args

message

exception MultipleObjectsReturned

args

message

delete ()

pk

prepare_database_save (*unused*)

save (*force_insert=False, force_update=False*)
Saves the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base (*raw=False, cls=None, origin=None, force_insert=False, force_update=False*)
Does the heavy-lifting involved in saving. Subclasses shouldn't need to override this method. It's separate from save() in order to hide the need for overrides of save() to pass around internal-only parameters ('raw', 'cls', and 'origin').

serializable_value (*field_name*)
Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

surgeon

treatment_set

vendor

class Measurement (**args, **kwargs*)
Measurement(id, animal_id, experiment_id, assay_id, values)

exception DoesNotExist

args

message

exception MultipleObjectsReturned

args

message

animal

assay

delete()

experiment

pk

prepare_database_save(*unused*)

save(*force_insert=False, force_update=False*)
Saves the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base(*raw=False, cls=None, origin=None, force_insert=False, force_update=False*)
Does the heavy-lifting involved in saving. Subclasses shouldn't need to override this method. It's separate from save() in order to hide the need for overrides of save() to pass around internal-only parameters ('raw', 'cls', and 'origin').

serializable_value(*field_name*)
Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

class Pharmaceutical (**args, **kwargs*)
Pharmaceutical(id, drug, dose, recurrance, mode, vendor_id, notes)

exception DoesNotExist

args

message

exception MultipleObjectsReturned

args

message

delete()

get_mode_display(**moreargs, **morekwargs*)

pk

prepare_database_save(*unused*)

save(*force_insert=False, force_update=False*)
Saves the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base(*raw=False, cls=None, origin=None, force_insert=False, force_update=False*)
Does the heavy-lifting involved in saving. Subclasses shouldn't need to override this method. It's separate from save() in order to hide the need for overrides of save() to pass around internal-only parameters ('raw', 'cls', and 'origin').

serializable_value(*field_name*)
Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

treatment_set

vendor

class Researcher(**args, **kwargs*)
Researcher(id, first_name, last_name, name_slug, email, active)

exception DoesNotExist

args

message

exception MultipleObjectsReturned

args

message

delete()

environment_set

experiment_set

get_absolute_url(**moreargs, **morekwargs*)

implantation_set

pk

prepare_database_save(*unused*)

save(*force_insert=False, force_update=False*)
Saves the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base (*raw=False, cls=None, origin=None, force_insert=False, force_update=False*)
Does the heavy-lifting involved in saving. Subclasses shouldn't need to override this method. It's separate from save() in order to hide the need for overrides of save() to pass around internal-only parameters ('raw', 'cls', and 'origin').

serializable_value (*field_name*)
Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

transplantation_set

treatment_set

class Study (**args, **kwargs*)
Study(id, description, start_date, stop_date, notes)

exception DoesNotExist

args

message

exception MultipleObjectsReturned

args

message

delete ()

experiment_set

pk

prepare_database_save (*unused*)

save (*force_insert=False, force_update=False*)
Saves the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base (*raw=False, cls=None, origin=None, force_insert=False, force_update=False*)
Does the heavy-lifting involved in saving. Subclasses shouldn't need to override this method. It's separate from save() in order to hide the need for overrides of save() to pass around internal-only parameters ('raw', 'cls', and 'origin').

serializable_value (*field_name*)
Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

strain

treatment_set

class Transplantation(**args, **kwargs*)
Transplantation(id, tissue, transplant_date, notes)

exception DoesNotExist

args

message

exception MultipleObjectsReturned

args

message

delete()

donor

get_next_by_transplant_date(**moreargs, **morekwargs*)

get_previous_by_transplant_date(**moreargs, **morekwargs*)

pk

prepare_database_save(*unused*)

save(*force_insert=False, force_update=False*)
Saves the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base(*raw=False, cls=None, origin=None, force_insert=False, force_update=False*)
Does the heavy-lifting involved in saving. Subclasses shouldn't need to override this method. It's separate from save() in order to hide the need for overrides of save() to pass around internal-only parameters ('raw', 'cls', and 'origin').

serializable_value(*field_name*)
Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

surgeon

treatment_set

class Treatment(**args, **kwargs*)
Treatment(id, treatment, study_id, diet_id, environment_id, transplantation_id, notes)

exception DoesNotExist

args

message

exception MultipleObjectsReturned

args

message

animals

delete()

diet

environment

implantation

pharmaceutical

pk

prepare_database_save(*unused*)

researchers

save(*force_insert=False, force_update=False*)
Saves the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base(*raw=False, cls=None, origin=None, force_insert=False, force_update=False*)
Does the heavy-lifting involved in saving. Subclasses shouldn't need to override this method. It's separate from save() in order to hide the need for overrides of save() to pass around internal-only parameters ('raw', 'cls', and 'origin').

serializable_value(*field_name*)
Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

study

transplantation

class Vendor(**args, **kwargs*)
Vendor(id, vendor, website, email, ordering, notes)

exception DoesNotExist

args

message

exception MultipleObjectsReturned

args

message

delete()

diet_set

get_ordering_display(**moreargs, **morekwargs*)

implantation_set

pharmaceutical_set

pk

prepare_database_save(*unused*)

save(*force_insert=False, force_update=False*)
Saves the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base(*raw=False, cls=None, origin=None, force_insert=False, force_update=False*)
Does the heavy-lifting involved in saving. Subclasses shouldn't need to override this method. It's separate from save() in order to hide the need for overrides of save() to pass around internal-only parameters ('raw', 'cls', and 'origin').

serializable_value(*field_name*)
Returns the value of the field name for this instance. If the field is a foreign key, returns the id value, instead of the object. If there's no Field object with this name on the model, the model attribute's value is returned directly.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

6.1.2 Forms

class **ExperimentForm**(*data=None, files=None, auto_id='id_%s', prefix=None, initial=None, error_class=<class 'django.forms.util.ErrorList'>, label_suffix=':', empty_permitted=False, instance=None*)

class **Meta**()

model
alias of `Experiment`

add_initial_prefix(*field_name*)
Add a 'initial' prefix for checking dynamic initial values

add_prefix(*field_name*)
Returns the field name with a prefix appended, if this Form has a prefix set.

Subclasses may wish to override.

as_p()
Returns this form rendered as HTML <p>s.

as_table()
Returns this form rendered as HTML <tr>s – excluding the <table></table>.

as_ul()
Returns this form rendered as HTML s – excluding the .

changed_data

clean()

date_error_message(*lookup_type, field, unique_for*)

errors
Returns an ErrorDict for the data provided for the form

full_clean()
Cleans all of self.data and populates self._errors and self.cleaned_data.

has_changed()
Returns True if data differs from initial.

hidden_fields()
Returns a list of all the BoundField objects that are hidden fields. Useful for manual form layout in templates.

is_multipart()
Returns True if the form needs to be multipart-encrypted, i.e. it has FileInput. Otherwise, False.

is_valid()
Returns True if the form has no errors. Otherwise, False. If errors are being ignored, returns False.

media

non_field_errors()
Returns an ErrorList of errors that aren't associated with a particular field – i.e., from Form.clean(). Returns an empty ErrorList if there are none.

save(*commit=True*)
Saves this `form`'s cleaned_data into model instance `self.instance`.

If commit=True, then the changes to `instance` will be saved to the database. Returns `instance`.

unique_error_message(*unique_check*)

validate_unique()

visible_fields()
Returns a list of BoundField objects that aren't hidden fields. The opposite of the hidden_fields() method.

class **MeasurementForm**(*data=None, files=None, auto_id='id_%s', prefix=None, initial=None, error_class=<class 'django.forms.util.ErrorList'>, label_suffix=':', empty_permitted=False, instance=None*)

class **Meta**()

model
alias of `Measurement`

add_initial_prefix(*field_name*)
Add a 'initial' prefix for checking dynamic initial values

add_prefix(*field_name*)
Returns the field name with a prefix appended, if this Form has a prefix set.

Subclasses may wish to override.

as_p()
Returns this form rendered as HTML <p>s.

as_table()
Returns this form rendered as HTML <tr>s – excluding the <table></table>.

as_ul()
Returns this form rendered as HTML s – excluding the .

changed_data

clean()

date_error_message(*lookup_type, field, unique_for*)

errors
Returns an ErrorDict for the data provided for the form

full_clean()
Cleans all of self.data and populates self._errors and self.cleaned_data.

has_changed()
Returns True if data differs from initial.

hidden_fields()
Returns a list of all the BoundField objects that are hidden fields. Useful for manual form layout in templates.

is_multipart()
Returns True if the form needs to be multipart-encrypted, i.e. it has FileInput. Otherwise, False.

is_valid()
Returns True if the form has no errors. Otherwise, False. If errors are being ignored, returns False.

media

non_field_errors()
Returns an ErrorList of errors that aren't associated with a particular field – i.e., from Form.clean(). Returns an empty ErrorList if there are none.

save(*commit=True*)
Saves this `form`'s cleaned_data into model instance `self.instance`.

If commit=True, then the changes to `instance` will be saved to the database. Returns `instance`.

unique_error_message(*unique_check*)

validate_unique()

visible_fields()
Returns a list of BoundField objects that aren't hidden fields. The opposite of the hidden_fields() method.

class **StudyExperimentForm**(*data=None, files=None, auto_id='id_%s', prefix=None, initial=None, error_class=<class 'django.forms.util.ErrorList'>, label_suffix=':', empty_permitted=False, instance=None*)

class **Meta**()

model
alias of `Experiment`

add_initial_prefix(*field_name*)
Add a 'initial' prefix for checking dynamic initial values

add_prefix(*field_name*)
Returns the field name with a prefix appended, if this Form has a prefix set.

Subclasses may wish to override.

as_p()
Returns this form rendered as HTML <p>s.

as_table()
Returns this form rendered as HTML <tr>s – excluding the <table></table>.

as_ul()
Returns this form rendered as HTML s – excluding the .

changed_data

clean()

date_error_message(*lookup_type, field, unique_for*)

errors
Returns an ErrorDict for the data provided for the form

full_clean()
Cleans all of self.data and populates self._errors and self.cleaned_data.

has_changed()
Returns True if data differs from initial.

hidden_fields()
Returns a list of all the BoundField objects that are hidden fields. Useful for manual form layout in templates.

is_multipart()
Returns True if the form needs to be multipart-encrypted, i.e. it has FileInput. Otherwise, False.

is_valid()
Returns True if the form has no errors. Otherwise, False. If errors are being ignored, returns False.

media

non_field_errors()
Returns an ErrorList of errors that aren't associated with a particular field – i.e., from Form.clean(). Returns an empty ErrorList if there are none.

save(*commit=True*)
Saves this `form`'s cleaned_data into model instance `self.instance`.

If commit=True, then the changes to `instance` will be saved to the database. Returns `instance`.

unique_error_message(*unique_check*)

validate_unique()

visible_fields()
Returns a list of BoundField objects that aren't hidden fields. The opposite of the hidden_fields() method.

class StudyForm(*data=None, files=None, auto_id='id_%s', prefix=None, initial=None, error_class=<class 'django.forms.util.ErrorList'>, label_suffix=':', empty_permitted=False, instance=None*)

class Meta()

model
alias of `Study`

add_initial_prefix(*field_name*)
Add a 'initial' prefix for checking dynamic initial values

add_prefix(*field_name*)
Returns the field name with a prefix appended, if this Form has a prefix set.

Subclasses may wish to override.

as_p()
Returns this form rendered as HTML <p>s.

as_table()
Returns this form rendered as HTML <tr>s – excluding the <table></table>.

as_ul()
Returns this form rendered as HTML s – excluding the .

changed_data

clean()

date_error_message(*lookup_type, field, unique_for*)

errors
Returns an ErrorDict for the data provided for the form

full_clean()
Cleans all of self.data and populates self._errors and self.cleaned_data.

has_changed()
Returns True if data differs from initial.

hidden_fields()
Returns a list of all the BoundField objects that are hidden fields. Useful for manual form layout in templates.

is_multipart()
Returns True if the form needs to be multipart-encrypted, i.e. it has FileInput. Otherwise, False.

is_valid()
Returns True if the form has no errors. Otherwise, False. If errors are being ignored, returns False.

media

non_field_errors()
Returns an ErrorList of errors that aren't associated with a particular field – i.e., from Form.clean(). Returns an empty ErrorList if there are none.

save(*commit=True*)
Saves this `form`'s cleaned_data into model instance `self.instance`.

If commit=True, then the changes to `instance` will be saved to the database. Returns `instance`.

unique_error_message(*unique_check*)

validate_unique()

visible_fields()
Returns a list of BoundField objects that aren't hidden fields. The opposite of the hidden_fields() method.

6.1.3 Views and URLs

add_measurement

experiment_detail

experiment_detail_all

experiment_list

study_experiment

6.1.4 Administrative Site Configuration

class AssayAdmin(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

class DietAdmin(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

class **EnvironmentAdmin**(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

class **ExperimentAdmin**(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

class **ImplantationAdmin**(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

class **MeasurementAdmin**(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

class MeasurementInline(*parent_model, admin_site*)

declared_fieldsets

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

formset
alias of `BaseInlineFormSet`

get_fieldsets(*request, obj=None*)

get_formset(*request, obj=None, **kwargs*)
Returns a BaseInlineFormSet class for use in admin add/change views.

media

model
alias of `Measurement`

class PharmaceuticalAdmin(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

class **ResearcherAdmin**(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of ActionForm

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of ModelForm

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

class StudyAdmin(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

class **TransplantationAdmin**(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

class **TreatmentAdmin**(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

class **TreatmentInline**(*parent_model, admin_site*)

declared_fieldsets

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

formset
alias of `BaseInlineFormSet`

get_fieldsets(*request, obj=None*)

get_formset(*request, obj=None, **kwargs*)
Returns a BaseInlineFormSet class for use in admin add/change views.

media

model
alias of `Treatment`

class **VendorAdmin**(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

6.2 Animals Package

6.2.1 Models

6.2.2 Forms

class **AnimalChangeForm**(*data=None, files=None, auto_id='id_%s', prefix=None, initial=None, error_class=<class 'django.forms.util.ErrorList'>, label_suffix=':', empty_permitted=False, instance=None*)

class **Meta**()

model
alias of `Animal`

add_initial_prefix(*field_name*)
Add a 'initial' prefix for checking dynamic initial values

add_prefix(*field_name*)
Returns the field name with a prefix appended, if this Form has a prefix set.

Subclasses may wish to override.

as_p()
Returns this form rendered as HTML <p>s.

as_table()
Returns this form rendered as HTML <tr>s – excluding the <table></table>.

as_ul()
Returns this form rendered as HTML s – excluding the .

changed_data

clean()

date_error_message(*lookup_type, field, unique_for*)

errors
Returns an ErrorDict for the data provided for the form

full_clean()
Cleans all of self.data and populates self._errors and self.cleaned_data.

has_changed()
Returns True if data differs from initial.

hidden_fields()
Returns a list of all the BoundField objects that are hidden fields. Useful for manual form layout in templates.

is_multipart()
Returns True if the form needs to be multipart-encrypted, i.e. it has FileInput. Otherwise, False.

is_valid()
Returns True if the form has no errors. Otherwise, False. If errors are being ignored, returns False.

media

non_field_errors()
Returns an ErrorList of errors that aren't associated with a particular field – i.e., from Form.clean(). Returns an empty ErrorList if there are none.

save(*commit=True*)
Saves this `form`'s cleaned_data into model instance `self.instance`.

If commit=True, then the changes to `instance` will be saved to the database. Returns `instance`.

unique_error_message(*unique_check*)

validate_unique()

visible_fields()
Returns a list of BoundField objects that aren't hidden fields. The opposite of the hidden_fields() method.

class **AnimalForm**(*data=None, files=None, auto_id='id_%s', prefix=None, initial=None, error_class=<class 'django.forms.util.ErrorList'>, label_suffix=':', empty_permitted=False, instance=None*)

class **Meta**()

model
alias of `Animal`

add_initial_prefix(*field_name*)
Add a 'initial' prefix for checking dynamic initial values

add_prefix(*field_name*)
Returns the field name with a prefix appended, if this Form has a prefix set.

Subclasses may wish to override.

as_p()
Returns this form rendered as HTML <p>s.

as_table()
Returns this form rendered as HTML <tr>s – excluding the <table></table>.

as_ul()
Returns this form rendered as HTML s – excluding the .

changed_data

clean()

date_error_message(*lookup_type, field, unique_for*)

errors
Returns an ErrorDict for the data provided for the form

full_clean()
Cleans all of self.data and populates self._errors and self.cleaned_data.

has_changed()
Returns True if data differs from initial.

hidden_fields()
Returns a list of all the BoundField objects that are hidden fields. Useful for manual form layout in templates.

is_multipart()
Returns True if the form needs to be multipart-encrypted, i.e. it has FileInput. Otherwise, False.

is_valid()
Returns True if the form has no errors. Otherwise, False. If errors are being ignored, returns False.

media

non_field_errors()
Returns an ErrorList of errors that aren't associated with a particular field – i.e., from Form.clean(). Returns an empty ErrorList if there are none.

save(*commit=True*)
Saves this `form`'s cleaned_data into model instance `self.instance`.

If commit=True, then the changes to `instance` will be saved to the database. Returns `instance`.

unique_error_message(*unique_check*)

validate_unique()

visible_fields()
Returns a list of BoundField objects that aren't hidden fields. The opposite of the hidden_fields() method.

6.2.3 Views and URLs

animal_change

animal_detail

animal_new

breeding

breeding_all

breeding_change

breeding_detail

breeding_pups

strain_detail

strain_detail_all

strain_list

6.2.4 Administrative Site Configuration

class AnimalAdmin(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

mark_sacrificed(*request, queryset*)

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

class AnimalInline(*parent_model, admin_site*)

declared_fieldsets

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

formset
alias of `BaseInlineFormSet`

get_fieldsets(*request, obj=None*)

get_formset(*request, obj=None, **kwargs*)
Returns a BaseInlineFormSet class for use in admin add/change views.

media

model
alias of `Animal`

class BreedingAdmin(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

class CageAdmin(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

class StrainAdmin(*model, admin_site*)

action_checkbox(*obj*)
A list_display column containing a checkbox widget.

action_form
alias of `ActionForm`

add_view(**args, **kw*)
The 'add' admin view for this model.

change_view(**args, **kw*)
The 'change' admin view for this model.

changelist_view(*request, extra_context=None*)
The 'change list' admin view for this model.

construct_change_message(*request, form, formsets*)
Construct a change message from a changed object.

declared_fieldsets

delete_view(*request, object_id, extra_context=None*)
The 'delete' admin view for this model.

form
alias of `ModelForm`

formfield_for_choice_field(*db_field, request=None, **kwargs*)
Get a form Field for a database Field that has declared choices.

formfield_for_dbfield(*db_field, **kwargs*)
Hook for specifying the form Field instance for a given database Field instance.

If kwargs are given, they're passed to the form Field's constructor.

formfield_for_foreignkey(*db_field, request=None, **kwargs*)
Get a form Field for a ForeignKey.

formfield_for_manytomany(*db_field, request=None, **kwargs*)
Get a form Field for a ManyToManyField.

get_action(*action*)
Return a given action from a parameter, which can either be a callable, or the name of a method on the ModelAdmin. Return is a tuple of (callable, name, description).

get_action_choices(*request, default_choices=, [('', '———')]*)
Return a list of choices for use in a form object. Each choice is a tuple (name, description).

get_actions(*request*)
Return a dictionary mapping the names of all actions for this ModelAdmin to a tuple of (callable, name, description) for each action.

get_changelist_form(*request, **kwargs*)
Returns a Form class for use in the Formset on the changelist page.

get_changelist_formset(*request, **kwargs*)
Returns a FormSet class for use on the changelist page if list_editable is used.

get_fieldsets(*request, obj=None*)
Hook for specifying fieldsets for the add form.

get_form(*request, obj=None, **kwargs*)
Returns a Form class for use in the admin add view. This is used by add_view and change_view.

get_formsets(*request, obj=None*)

get_model_perms(*request*)
Returns a dict of all perms for this model. This dict has the keys `add`, `change`, and `delete` mapping to the True/False for each of those actions.

get_urls()

has_add_permission(*request*)
Returns True if the given request has permission to add an object.

has_change_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to change *any* object of the given type.

has_delete_permission(*request, obj=None*)
Returns True if the given request has permission to change the given Django model instance.

If *obj* is None, this should return True if the given request has permission to delete *any* object of the given type.

history_view(*request, object_id, extra_context=None*)
The 'history' admin view for this model.

log_addition(*request, object*)
Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

log_change(*request, object, message*)
Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

log_deletion(*request, object, object_repr*)
Log that an object has been successfully deleted. Note that since the object is deleted, it might no longer be safe to call *any* methods on the object, hence this method getting object_repr.

The default implementation creates an admin LogEntry object.

media

message_user(*request, message*)
Send a message to the user. The default implementation posts a message using the auth Message object.

queryset(*request*)
Returns a QuerySet of all model instances that can be edited by the admin site. This is used by changelist_view.

render_change_form(*request, context, add=False, change=False, form_url='', obj=None*)

response_action(*request, queryset*)
Handle an admin action. This is called if a request is POSTed to the changelist; it returns an HttpResponse if the action was handled, and None otherwise.

response_add(*request, obj, post_url_continue='../%s/'*)
Determines the HttpResponse for the add_view stage.

response_change(*request, obj*)
Determines the HttpResponse for the change_view stage.

save_form(*request, form, change*)
Given a ModelForm return an unsaved instance. `change` is True if the object is being changed, and False if it's being added.

save_formset(*request, form, formset, change*)
Given an inline formset save it to the database.

save_model(*request, obj, form, change*)
Given a model instance save it to the database.

urls

CHAPTER

SEVEN

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

A

D

INDEX

A

B

C

D

E

F

G

H

I

L

M

N

P

Q

R

S