Signals

The signals are utilities that allow us to associate events with actions.

Signals allow certain senders to notify a set of receivers that some action has taken place.

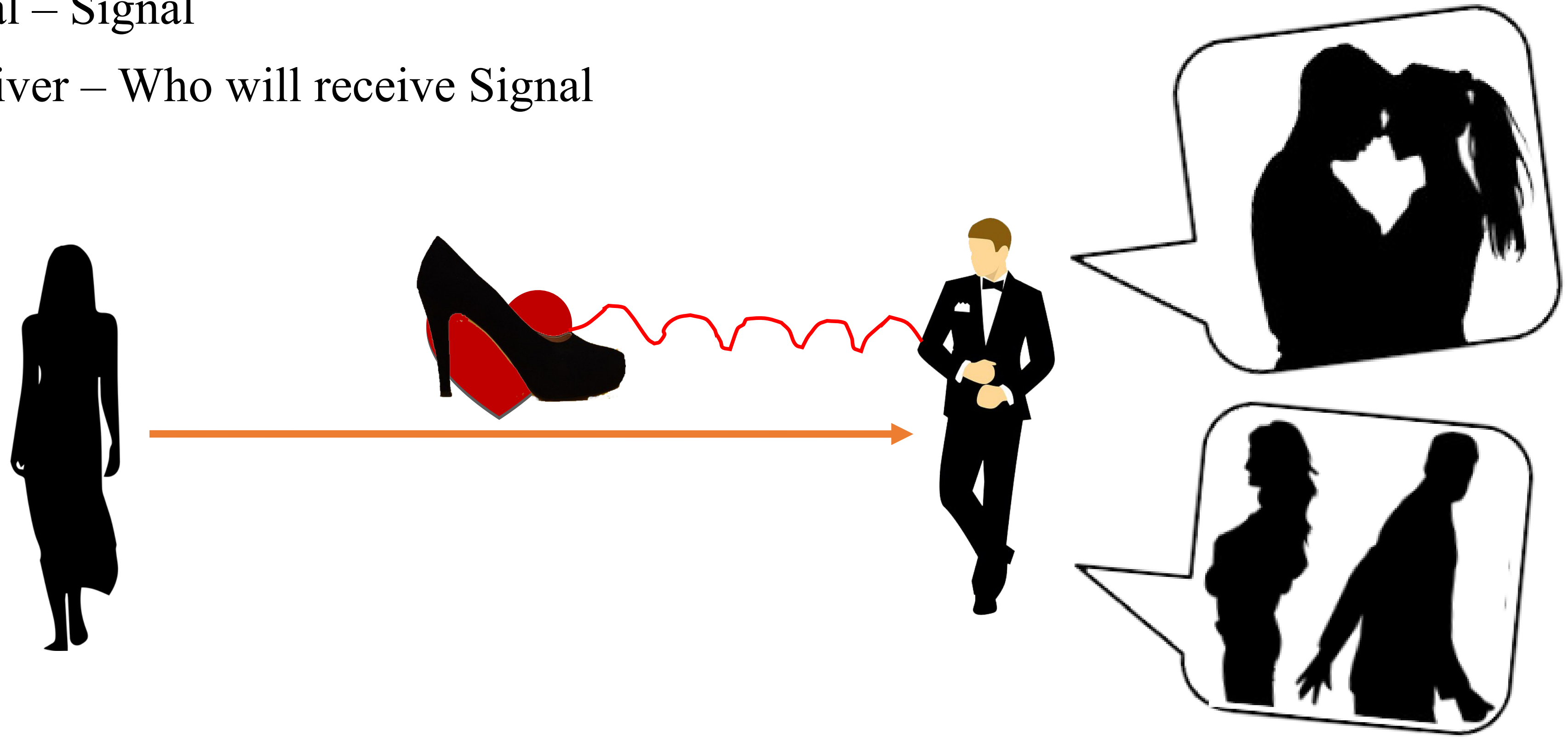
- Login and Logout Signals
- Model Signals
- Management Signals
- Request/Response Signals
- Test Signals
- Database Wrappers

Signals

Sender – Who will send Signal

Signal – Signal

Receiver – Who will receive Signal



Signals

Receiver Function - This function takes a sender argument, along with wildcard keyword arguments (**kwargs); all signal handlers must take these arguments. A receiver can be any Python function or method.

```
def receiver_func(sender, request, user, **kwargs):
    pass
```

Connecting/Registering Receiver Function - There are two ways you can connect a receiver to a signal:-

- Manual Connect Route
- Decorator

Signals

Manual Connect Route - To receive a signal, register a receiver function using the Signal.connect() method. The receiver function is called when the signal is sent. All of the signal's receiver functions are called one at a time, in the order they were registered.

Signal.connect(receiver_func, sender=None, weak=True, dispatch_uid=None)

Where,

receiver_func – The callback function which will be connected to signal.

sender – Specifies a particular sender to receive signals from.

weak – Django stores signal handlers as weak references by default. Thus, if your receiver is a local function, it may be garbage collected. To prevent this, pass weak=False when you call the signal's connect() method.

dispatch_uid – A unique identifier for a signal receiver in cases where duplicate signals may be sent.

Decorator - @receiver(signal or list of signal, sender)

Built-in Signals

Django provides a set of built-in signals that let user code get notified by Django itself of certain actions.

Login and Logout Signals - The auth framework uses the following signals that can be used for notification when a user logs in or out.

django.contrib.auth.signals

user_logged_in(sender, request, user) - Sent when a user logs in successfully.

sender - The class of the user that just logged in.

request - The current HttpRequest instance.

user - The user instance that just logged in.

user_logged_out(sender, request, user) - Sent when the logout method is called.

sender - The class of the user that just logged out or None if the user was not authenticated.

request - The current HttpRequest instance.

user - The user instance that just logged out or None if the user was not authenticated.

Built-in Signals

user_login_failed(sender, credentials, request) - Sent when the user failed to login successfully.

sender - The name of the module used for authentication.

credentials - A dictionary of keyword arguments containing the user credentials that were passed to authenticate() or your own custom authentication backend. Credentials matching a set of 'sensitive' patterns, (including password) will not be sent in the clear as part of the signal.

request - The HttpRequest object, if one was provided to authenticate()

<u>Built-in Signals</u>

<u>**Model signals**</u> - A set of signals sent by the model system.

django.db.models.signals

pre_init (sender, args, kwargs) - Whenever you instantiate a Django model, this signal is sent at the beginning of the model’s __init__() method.

sender - The model class that just had an instance created.

args - A list of positional arguments passed to __init__().

kwargs - A dictionary of keyword arguments passed to __init__().

post_init (sender, instance) - Like pre_init, but this one is sent when the __init__() method finishes.

sender - The model class that just had an instance created.

instance - The actual instance of the model that’s just been created.

Built-in Signals

pre_save (sender, instance, raw, using, update_fields) - This is sent at the beginning of a model's save() method.

sender - The model class.

instance - The actual instance being saved.

raw - A boolean; True if the model is saved exactly as presented (i.e. when loading a fixture). One should not query/modify other records in the database as the database might not be in a consistent state yet.

using - The database alias being used.

update_fields - The set of fields to update as passed to Model.save(), or None if update_fields wasn't passed to save().

Built-in Signals

post_save(sender, instance, created, raw, using, update_fields) - Like pre_save, but sent at the end of the save() method.

sender - The model class.

instance - The actual instance being saved.

created - A boolean; True if a new record was created.

raw - A boolean; True if the model is saved exactly as presented (i.e. when loading a fixture). One should not query/modify other records in the database as the database might not be in a consistent state yet.

using - The database alias being used.

update_fields - The set of fields to update as passed to Model.save(), or None if update_fields wasn't passed to save().

Built-in Signals

pre_delete(sender, instance, using) - Sent at the beginning of a model’s delete() method and a queryset’s delete() method.

sender - The model class.

instance - The actual instance being deleted.

using - The database alias being used.

post_delete(sender, instance, using) - Like pre_delete, but sent at the end of a model’s delete() method and a queryset’s delete() method.

sender - The model class.

instance - The actual instance being deleted.

Note that the object will no longer be in the database, so be very careful what you do with this instance.

using - The database alias being used.

Built-in Signals

m2m_changed(sender, instance, action, reverse, model, pk_set, using) - Sent when a ManyToManyField is changed on a model instance. Strictly speaking, this is not a model signal since it is sent by the ManyToManyField, but since it complements the pre_save/post_save and pre_delete/post_delete when it comes to tracking changes to models, it is included here.

sender - The intermediate model class describing the ManyToManyField. This class is automatically created when a many-to-many field is defined; you can access it using the through attribute on the many-to-many field.

instance - The instance whose many-to-many relation is updated. This can be an instance of the sender, or of the class the ManyToManyField is related to.

action - A string indicating the type of update that is done on the relation. This can be one of the following:

"pre_add" - Sent before one or more objects are added to the relation.

"post_add" - Sent after one or more objects are added to the relation.

"pre_remove" - Sent before one or more objects are removed from the relation.

"post_remove" - Sent after one or more objects are removed from the relation.

"pre_clear" - Sent before the relation is cleared.

"post_clear" - Sent after the relation is cleared.

Built-in Signals

reverse - Indicates which side of the relation is updated (i.e., if it is the forward or reverse relation that is being modified).

model - The class of the objects that are added to, removed from or cleared from the relation.

pk_set - For the pre_add and post_add actions, this is a set of primary key values that will be, or have been, added to the relation. This may be a subset of the values submitted to be added, since inserts must filter existing values in order to avoid a database IntegrityError.

For the pre_remove and post_remove actions, this is a set of primary key values that was submitted to be removed from the relation. This is not dependent on whether the values actually will be, or have been, removed. In particular, non-existent values may be submitted, and will appear in pk_set, even though they have no effect on the database.

For the pre_clear and post_clear actions, this is None.

using - The database alias being used.

Built-in Signals

class_prepared(sender) - Sent whenever a model class has been "prepared" – that is, once model has been defined and registered with Django's model system. Django uses this signal internally; it's not generally used in third-party applications.

Since this signal is sent during the app registry population process, and AppConfig.ready() runs after the app registry is fully populated, receivers cannot be connected in that method. One possibility is to connect them AppConfig.__init__() instead, taking care not to import models or trigger calls to the app registry.

sender - The model class which was just prepared.

Built-in Signals

Request/Response Signals - Signals sent by the core framework when processing a request.

django.core.signals

request_started(sender, environ) - Sent when Django begins processing an HTTP request.

sender - The handler class – e.g. django.core.handlers.wsgi.WsgiHandler – that handled the request.

environ - The environ dictionary provided to the request.

request_finished(sender) - Sent when Django finishes delivering an HTTP response to the client.

sender - The handler class.

got_request_exception(sender, request) - This signal is sent whenever Django encounters an exception while processing an incoming HTTP request.

sender - Unused (always None).

request - The HttpRequest object.

Built-in Signals

Management signals – Signals sent by Django-admin

django.db.models.signals

pre_migrate(sender, app_config, verbosity, interactive, using, plan, apps) - Sent by the migrate command before it starts to install an application. It's not emitted for applications that lack a models module.

sender - An AppConfig instance for the application about to be migrated/synced.

app_config - Same as sender.

verbosity - Indicates how much information manage.py is printing on screen.

Functions which listen for pre_migrate should adjust what they output to the screen based on the value of this argument.

interactive - If interactive is True, it's safe to prompt the user to input things on the command line. If interactive is False, functions which listen for this signal should not try to prompt for anything.

For example, the django.contrib.auth app only prompts to create a superuser when interactive is True.

using - The alias of database on which a command will operate.

Built-in Signals

plan - The migration plan that is going to be used for the migration run. While the plan is not public API, this allows for the rare cases when it is necessary to know the plan. A plan is a list of two-tuples with the first item being the instance of a migration class and the second item showing if the migration was rolled back (True) or applied (False).

apps - An instance of Apps containing the state of the project before the migration run. It should be used instead of the global apps registry to retrieve the models you want to perform operations on

Built-in Signals

post_migrate(sender, aap_config, verbosity, interactive, using, plan, apps) - Sent at the end of the migrate (even if no migrations are run) and flush commands. It’s not emitted for applications that lack a models module.

Handlers of this signal must not perform database schema alterations as doing so may cause the flush command to fail if it runs during the migrate command.

sender - An AppConfig instance for the application that was just installed.

app_config - Same as sender.

verbosity - Indicates how much information manage.py is printing on screen.

Functions which listen for post_migrate should adjust what they output to the screen based on the value of this argument.

interactive - If interactive is True, it’s safe to prompt the user to input things on the command line. If interactive is False, functions which listen for this signal should not try to prompt for anything.

For example, the django.contrib.auth app only prompts to create a superuser when interactive is True.

Built-in Signals

using - The database alias used for synchronization. Defaults to the default database.

plan - The migration plan that was used for the migration run. While the plan is not public API, this allows for the rare cases when it is necessary to know the plan. A plan is a list of two-tuples with the first item being the instance of a migration class and the second item showing if the migration was rolled back (True) or applied (False).

apps - An instance of Apps containing the state of the project after the migration run. It should be used instead of the global apps registry to retrieve the models you want to perform operations on.

Built-in Signals

Test Signals - Signals only sent when running tests.

django.test.signals

setting_changed(sender, setting, value, enter) - This signal is sent when the value of a setting is changed through the django.test.TestCase.settings() context manager or the django.test.override_settings() decorator/context manager.

It's actually sent twice: when the new value is applied ("setup") and when the original value is restored ("teardown"). Use the enter argument to distinguish between the two.

You can also import this signal from django.core.signals to avoid importing from django.test in non-test situations.

sender - The settings handler.

setting - The name of the setting.

value - The value of the setting after the change. For settings that initially don't exist, in the "teardown" phase, value is None.

enter - A boolean; True if the setting is applied, False if restored.

Built-in Signals

template_rendered(sender, template, context) - Sent when the test system renders a template. This signal is not emitted during normal operation of a Django server – it is only available during testing.

sender - The Template object which was rendered.

template - Same as sender

context - The Context with which the template was rendered.

Built-in Signals

Database Wrappers - Signals sent by the database wrapper when a database connection is initiated.

django.db.backends.signals

connection_created - Sent when the database wrapper makes the initial connection to the database. This is particularly useful if you'd like to send any post connection commands to the SQL backend.

sender - The database wrapper class – i.e. django.db.backends.postgresql.DatabaseWrapper or django.db.backends.mysql.DatabaseWrapper, etc.

connection - The database connection that was opened. This can be used in a multiple-database configuration to differentiate connection signals from different databases.