Django Automations

Fabian Braun

Nov 28, 2021

CONTENTS:

CHAPTER

ONE

WHY DJANGO AUTOMATIONS

1.1 Business logic

The Django framework rightfully is an extremely popular web development framework. Django makes it easier to build better Web apps more quickly and with less code, as they state themselves.

The key elements are **models**, **views** and **templates**. Models represent the persistent data which is stored in a database backend. Views convert these data into human readable form: contexts. The context is rendered as html using templates.

This setup leads to business logic being scattered around in a project: Bits and pieces of the same process appear in different views which in turn access several models and their logic.

Django automations aims to add another layer where business logic can be maintained centrally. Just like models live in `models.py`, views in `views.py` automations are made to live in an app's `automations.py`.

Automations connect different tasks, may they be automatic or require user-interaction, to lead to a predefined result. Conditionals allow to branch according to the specific needs.

The objective is to integrate and automate business processes with less code. Certain tasks either can be assigned to specific users or user groups or automatically carried out by your Django app.

1.2 Implementation

The implementation is done with a few objectives in mind:

- **Lightweight:** Django automations builds on proven Django elements: Models to keep the state of the processes and forms to manage user interaction. Ability to bind to other Django models, however, no need for migrations if bindings are changed or state information is added.
- **Python syntax:** Just like models or forms are Python classes, automations are Python classes built in a similar way (from Nodes instead of ModelFields or FormFields)
- **Easy extensibility:** To keep the core light, it is designed to allow for easy customization in a project.

1.3 Benefits

- **Transparency:** Business logic in one place
- **Maintainability:** Changes in business logic do not happen in several models or views, just in `automations.py`.
- **Time savings:** Less code to write
- **Extendability:** Automations can be extended while "running" as long as the names of existing states remain the same.

1.4 Enjoy!

CHAPTER

TWO

GETTING STARTED

2.1 Automations

In Django, a model is the single, definitive source of information about your data. View encapsulate the logic responsible for processing a user's request and for returning the response. Templates are used to create HTML from views. Forms define how model instances can be edited and created.

Automations describe (business) processes that define what model manipulations or form interactions have to be executed to reach a specific goal in your applications. Automations glue together different parts of your Django project while keeping all required code in one place. Just like models live in `models.py`, views in `views.py`, forms in `forms.py` automations live in an apps's `automations.py`.

The basics:

1. Automations are Python subclasses of `flow.Automation`
2. Each attribute represents a task node, quite similar to Django models
3. All instances of automations are executed. Their states are kept in two models used by all Automations, quite in contrast to Django models where there is a one-to-one correspondence between model and table.

2.2 Preparing your Django project

Before using the Django automations app, you need to install the package using pip. Then *automations has to be added to your project's `INSTALLED_APPS`* in `settings.py`:

```
INSTALLED_APPS = (
    ...,
    'automations',
    'automations.cms_automations',   # ONLY IF YOU USE DJANGO-CMS!
)
```

Note: Only include the "sub app" `automations.cms_automations` if you are using Django CMS. This sub-application will add additional plugins for Django CMS that integrate nicely with Django Automations.

Finally, run

```
python manage.py migrate automations
python manage.py migrate cms_automations
```

to create Django Automations' database tables.

2.3 Simple example: WebinarWorkflow

This automation executes four consecutive tasks before terminating. These tasks have a timely pattern: The reminder is only sent shortly before the webinar begins. The replay offer is sent after the webinar, 2.5 hours after the reminder.

```
import datetime

from automations import flow
from automations.flow import this

from . import webinar

class WebinarWorkflow(flow.Automation):
    start =              flow.Execute(this.init)
    send_welcome_mail = flow.Execute(webinar.send_welcome_mail)
    send_reminder =     (flow.Execute(webinar.send_reminder_mail)
                             .AfterWaitingUntil(webinar.reminder_time))
    send_replay_offer = (flow.Execute(webinar.send_replay_mail)
                             .AfterWaitingFor(datetime.timedelta(minutes=150)))
    end =                flow.End()

    def init(self, task):
        ...
```

This defines the WebinarWorkflow. Only once a class object is created, the `WebinarWorkflow` automation will be executed. Programmatically, you can create an object by saying `webinar_workflow = WebinarWorkflow()`.

2.4 Nodes

Each task of an automation is expressed by a `flow.Node`. In the example above two node classes are used: `flow.Execute` and `flow.End()`. By making a node an attribute of an `Automation` class it gets bound to it. Some nodes take parameters, like `flow.Execute`, some do not, like `flow.End()`.

Note:

- Nodes are processed in their order of declaration in the automation class (unless specified differently, see below).
- Each node has a name (`start`, `send_welcome_mail`, ...). Each running instance of the automation has a state (or program counter) which corresponds to the name of the node which is to be executed next.
- Since at the declaration of the `Automation` attributes no object has been created there is no `self` reference. The `this` object replaces `self` during the declaration of the automation class. (`this` objects are replaced by `self`-references at the time of execution of the automation.)
- To allow for timed execution, some sort of scheduler is needed in the project.

2.4.1 Node types

Django Automations has some built-in node types (see reference).

- `flow.Execute()` executes a Python callable, typically a method of the automation class to perform the task.
- `flow.End()` terminates the execution of the current automation object.

More nodes are:

- `flow.Repeat()` declares an infinite loop to define regular worker processes.
- `flow.If` allows conditional branching within the automation.
- `flow.Split()` allows to split the execution of the automation in 2 or more concurring paths.
- `flow.Join()` waits until all paths that have started at the same previous `Split()` have converged again. (All splitted paths must be join before ending an automation!)
- `flow.Form()` requires a specific user or a user of a group of users to fill in a form before the automation continues.
- `flow.ModelForm()` is a simplified front end of `flow.Form()` to create or edit model instances.
- `flow.SendMessage()` allows to communicate with other automations.

2.4.2 Modifier

Each node can be modified using modifiers. Modifiers are methods of the `Node` class which return `self` and therefore can be chained together. This well-known pattern from JavaScript allows a node to be modified multiple times.

Modifiers can add conditions which have to be fulfilled before the execution of the task begins. Typical conditions include passing of a certain amount of time or reaching a certain date and time. Other uses include defining the next node that is to be executed (a little bit like goto).

Modifiers for all nodes (with the exception for `flow.Form` and `flow.ModelForm`) are

- `.Next(node)` sets the node to continue with after finishing this node. If omitted the automation continues with the chronologically next node of the class. `.Next` resembles a goto statement. `.Next` takes a string or a `This` object as a parameter. A string denotes the name of the next node. The this object allows for a different syntax. `.Next("next_node")` and `.Next(this.next_node)` are equivalent.
- `.AsSoonAs(condition)` waits for condition before continuing the automation. If condition is `False` the automation is interrupted and `condition` is checked the next time the automation instance is run.
- `.AfterWaitingUntil(datetime)` stops the automation until the specific datetime has passed. Note that depending on how the scheduler runs the automation there might be a significant time slip between `datetime` and the real execution time. It is only guaranteed that the node is not executed before. `datetime` may be a callable.
- `.AfterWaitingFor(timedelta)` stops the automation for a specific amount of time. This is equivalent to `.AfterWaitingUntil(lambda x: now()+timedelta)`.
- `.SkipIf` leaves a node unprocessed if a condition is fulfilled.

Other nodes implement additional modifiers, e.g., `.Then()` and `.Else()` in the `If()` node. A different example is `.OnError(next_node)` in the `flow.Execute()` node which defines where to jump should the execution of the specified method raise an exception.

2.4.3 Node inheritance

Especially the `flow.Execute` node can be easily subclassed to create specific and speaking nodes. E.g., in the above example it might be useful to create a node `SendMail`:

```
class SendMail(flow.Execute):
    def method(self, task_instance, mail_id):
        """here goes the code to be executed"""
```

2.4.4 Meta options

Similar to Django's meta options, Django Automations allows to define verbose names for each automation.

```
class WebinarWorkflow(flow.Automation):
    class Meta:
        verbose_name = _("Webinar preparation")

    start =             flow.Execute(this.init)
    ...
```

Verbose names can appear in Django Automations' views. If no verbose name is given the standard name "Automation " plus the class name is used. In this example it is `Automation WebinarWorkflow`.

CHAPTER

THREE

CORE CONCEPTS

3.1 Automations are processes

Automations are nothing but well-defined processes designed to reach a goal. While often it is not technically difficult to implement processes with Django projects, maintenance can become quite complex over time if

- different tasks of the process are hidden at different places in the code base.
- documentation of the automations is not complete or out of sync
- multiple users or user groups are involved in one process and need to share information.

Commonly, business processes are described using events, tasks, and gateways.

3.1.1 Events

Events can start an automation or can change its course once it is running. Django automations can be started by Django signals (see Django documentation), programmatically by instantiating an automation, or by sending it a message.

3.1.2 Tasks

Tasks describe work that has to be done, either by the Django application or by an user: Send an email or update a model instance are respective examples.

Tasks are the single units of work that is not or cannot be broken down to a further level. It is referred to as an atomic activity. A task is the lowest level activity illustrated on a process diagram. A set of tasks may represent a high-level procedure. Tasks are are atomic transactions from a Django viewpoint.

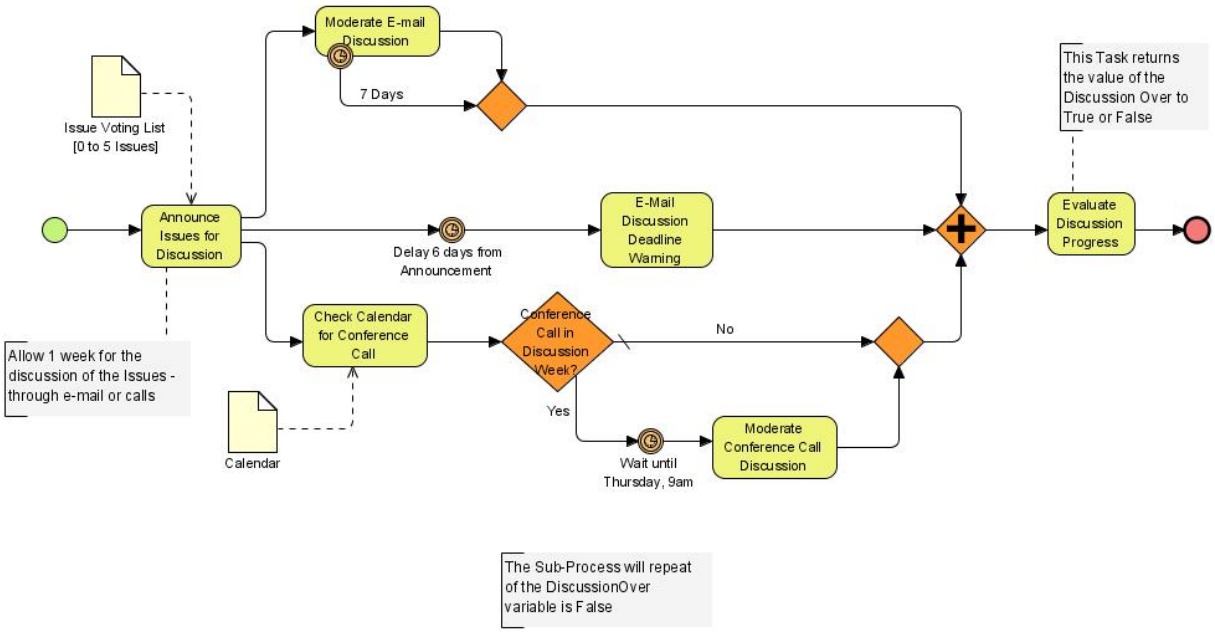
Work to be done by a user are represented by Django forms where the user has to fill in the results of her work, e.g., do an approval, fill in billing details etc.

3.1.3 Gateways

Gateways change the flow of an automation, e.g., depending on a condition. Django automations offers `If()`, `.Then()`, `.Else()` constructs. The `Split()`/`Join()` gateway allows to parallelize the automation.

3.2 Example issue discussion cycle

Assume you have a Django app that collects issues on a list and each week it creates an issue list for discussion.



Django Automations allows to describe and document the process in one place, a python class (what else?).

This process can be translated into an automation like this

```
class IssueDiscussion(flow.Automation):
    issue_list = models.IssueList

    start =          flow.Execute(this.publish_announcement)
    parallel =       flow.Split().Next(this.moderation).Next(this.warning).Next(this.conf_
↪call)

    moderation =     flow.If(this.no_new_mails).Then(this.repeat)
    moderate =       flow.Form(forms.MailModeration).Group(name='Moderators')
    repeat =         (flow.If(this.moderation_deadline_reached)
                         .Then(this.join)
                         .Else(this.moderation)
                     ).AfterWaitingFor(datetime.timedelta(hours=1))

    warning =        (flow.Execute(this.send_deadline_warning)
                         .AfterWaitingFor(datetime.timedelta(days=6))
                         .Next(this.join))

    conf_call =      flow.If(this.conf_call_this_week).Then(this.moderate_call).Else(this.
↪join)
    moderate_call = flow.Execute(this.block_calendar)

    join           = flow.Join()
    evaluate       = flow.Execute(this.evaluate)
```

(continues on next page)

(continued from previous page)

```
    end             = flow.End()
```

3.3 Automation states

Automations have a state, i.e. they execute at one or more tasks. All execution points share the same attached model instances and (simple) state data. As many instances of an automation may be executed concurrently as necessary each instance has its own state.

Let's say you wanted to automate the signup process for a webinar. Then a single session of a webinar with date and time might be the model instance you wanted to attach to the automation. This means each session of the webinar would also have an independent automation instance managing the signup process including sending out the webinar link, reminding people when the webinar starts or offering a recording after the webinar. While during the process the session does not change the list of people who have signed up changes but still always refers to the same webinar session.

Django automations has two optional ways of storing state data. The first one is **binding model instances to an automation instance** allowing for all form of data Django models can handle. Additionally each automation instance has **a json-serializable dictionary attached** called `data`. Since it is stored in a Django `JSONField` it only may contain combination of basic types (like `int`, `real`, `list`, `dict`, `None`). This data dictionary is also used to store results of form interactions or for automation methods to keep intermediate states to communicate with each other.

3.4 Modifiers

When interacting with humans, an automation will have to wait for input but also give humans time to digest and react. Modifiers control the speed at which an automation is executed: How long to wait before sending a reminder, or how long to give time before escalating the need for important information to the user's superior. The timing of each step is controlled by "modifiers" which, e.g., pause an automation before continuing.

3.5 Request-response cycle and scheduling

Practically all automations pause or wait for other processes to finish most of the time.

From time to time, the automations have to be checked if they can advance. This is the task of a scheduler outside this package. The scheduler may, e.g., call the class method `models.AutomationModel.run`. Additionally, Django Automations offers a new management command `python manage.py automation_step` that can be invoked by an external scheduler.

Also, an automation may advance, e.g., after an processing form has been filled and validated. Then the automation may advance within the request-response cycle of the POST request of the form. To keep the web app responsive, all automation steps need to be fast. Optionally, Django Automations allows to spawn threads for the background processes, or if serious calculations have to be done, outsourced to a worker task.

Note: Django Automations is not a task scheduler for background worker processes. It is a framework that defines what tasks have to be done at what time under what conditions. It works, however, well with background worker processes.

Currently, there is no native integration with, say, Celery. However, this might be an extension which would be welcomed.

CHAPTER

FOUR

HOW-TO GUIDES

4.1 How to modify automations already running

Automations can be updated, improved and changed even when they are running if a few rules are followed.

Each automation instance is represented by a model instance of `models.AutomationModel`. For each task that is executed an instance of `models.AutomationTaskModel` is created. For each unfinished automation there is at least one unfinished task. If the automations contains `Split()` nodes there might be more than one unfinished task. Typically these tasks are waiting either for a user interaction, a condition to become true, or until a certain amount of time passes.

This implies that **it is always possible to add nodes** to the automation. New nodes will be executed as soon as an previous task is finished and the automation pointer moves forward to the new node.

Also, **it is possible to change existing nodes**. However, this will only affect automation instances that have not yet processed the node. This leaves a record for the automations where the same node name corresponds to a different task and may render evaluation of automation results difficult.

> **Warning:** Nodes can only be removed from an automation if no instance is pointing to that node. Since this is difficult to guarantee the following process ensures integrity.

Hence, to remove a node from an automation with existing instances follow this process:

1. Change the node you want to remove from an automation to `flow.Execute()` without any modifiers. This is a no-operation.
2. Run `./manage.py automation_step`. This causes all automation instances with an open task at the node you want to delete to process the no-op and move to the next task.
3. Remove the node. Removing is safe now since an automation instance coming to the node immediately will execute to no-op and move to the next task.

CHAPTER

FIVE

REFERENCE

5.1 Automation class

`flow.Automation` is a class that provides the core functionality of Django Automations. To create an automation `flow.Automation` is subclassed with a list of properties of type `flow.Node` which are to executed one after the other.

5.1.1 flow.Automation

Creating a subclass of `flow.Automation` is simple and similar to creating forms or models in Django. The order of the properties defines the order of execution. All nodes can be adapted using modifiers.

Example:

```
from automations import flow

this = flow.This()

class IssueDiscussion(flow.Automation):
    issue_list = models.IssueList

    start =          flow.Execute(this.publish_announcement)
    parallel =       flow.Split().Next(this.moderation).Next(this.warning).Next(this.conf_
↪call)

    moderation =     flow.If(this.no_new_mails).Then(this.repeat)
    moderate =       flow.Form(forms.MailModeration).Group(name='Moderators')
    repeat =         (flow.If(this.moderation_deadline_reached)
                         .Then(this.join)
                         .Else(this.moderation)
                     ).AfterWaitingFor(datetime.timedelta(hours=1))

    warning =        (flow.Execute(this.send_deadline_warning)
                         .AfterWaitingFor(datetime.timedelta(days=6))
                         .Next(this.join))

    conf_call =      flow.If(this.conf_call_this_week).Then(this.moderate_call).Else(this.
↪join)
    moderate_call = flow.Execute(this.block_calendar)

    join           = flow.Join()
```

(continues on next page)

(continued from previous page)

```
    evaluate      = flow.Execute(this.evaluate)
    end           = flow.End()

    def publish_announcement(self, task_instance):
        ...

    def no_new_mails(self, task_instance):
        # Do something to check mails
        return not new_mail_found

    ...
```

Note that each node has a name (start, parallel, ...). Execution of the automation starts at the first node and continues node by node as long as no branches are set (`.Next()`, `.Then()`, or `.Else()` modifiers). Nodes need to have a name even if they are not the target of a branch.

It is customary to bracket nodes that continue over line breaks.

The `.End()` node denotes the end of the automation. Once it is reached the automation's instances' finished flag is set.

Short programs to execute are realized as methods which take one argument besides `self`: `task` is an instance of `models.AutomationTaskModel` and has access to the automations json data field through `task.data`. This is a convenience solution since it allows lambda expressions, e.g., as parameters for the `If()` node:

```
branch = (If(lambda x: x.data.get('has_signed_contract', False))
          .Then(this.process_contract)
          .Else(this.send_reminder))
```

Self-referencing can be achieved using the `this` instance of the `This()` object or by denoting the reference as string: `"self.last_node"` is equivalent to `this.last_node`.

Warning: The property names might shadow attributes or methods of the `flow.Automations` class. Avoid using existing names like `id`, `data`, `save`, etc. See the list below. Shadowing might lead to unexpected side-effects. Names beginning with underscore _ are reserved and not to used for nodes.

Here is a list of names to avoid:

- `broadcast_message`
- `create_on_message`
- `data`
- `dispatch_message`
- `finished`
- `get_automation_class_name`
- `get_key`
- `get_model_instance`
- `get_verbose_name`
- `get_verbose_name_plural`
- `id`
- `kill`

- model_class
- nice
- on
- on_signal
- run
- satisfies_data_requirements
- save
- send_message
- unique

Automations are started when instantiated, e.g., by `instance = IssueDiscussion(issue_list=this_weeks_list)`.

Note: Parameters to the `__init__` method are stored in the instance's data json field. The values need to be json-serializable. the only exception are Django model instances. If a model instance is passed the id will be stored in the data field. Also, a property will be created where the respective instance of the model is available.

There are three special parameters when creating an instance:

- `automation` denotes the `models.AutomationModel` instance to bind this automation to. Hence, not a new automation will be created but an existing automation instance will be created from the database data.
- `automation_id` is an integer, denoting the id of an `models.AutomationModel` instance. The effect is the same as binding directly to the automation.
- `autorun` is a boolean value indicating whether the execution shall start immediately when creating the instance. Its default is `True`. Set it `False` if you need to do additional initialization work.

`Automation.`**`unique`**

The unique attribute is declared when subclassing `flow.Automation`. It takes either a boolean value or is a list or tuple of strings.

If `True` it makes the automation a singleton, i.e. only one instance can run at a time. If a singleton is instantiated and already an instance is running it will return this running instance.

If `.unique` is a list of strings it declares a set of parameters for the automation which are unique for any instance of it. Parameters of an automation instance are stored in its `.data` json field. For example, if you want to avoid sending the same e-mails to an email address multiple times, you can use `unique = ('email', )` to only alow one instance of the automation per email.

`.unique` defaults to `False`.

`Automation.`**`id`**

Gives the id of the automation instance. It can be used, e.g., to send messages to this instance. Since it is an integer, it can easily serialized and, e.g., passed as a GET parameter.

`Automation.`**`data`**

Gives the automation instance's `data` json field. It is a dictionary of json-serializable data: an instance of Django's `JSONField`.

`Automation.`**`save`**`()`

Saves the data field back to the database. This method needs to be called after modifying the `.data` attribute.

`Automation.run()`
Starts the execution loop of the automation and runs until the automation

1. Finishes
2. Reaches a node which requires user interaction (subclass of `flow.Flow()`)
3. Reaches a node which requires waiting for a condition or a certain amount of time

Automations should only contain nodes that do not need more than a few milliseconds to reach one of these conditions. Complex algorithms are supposed to be coded in Python directly. If an automation needs to do complex calculations these calculations should use the `threaded=True` option fo the `Execute()` node.

`run()` returns the node at which one of the three conditions was reached.

`Automation.nice()`
Starts the execution loop in a new thread using Python's `threading` library and returns immediately.

`Automation.is_finished()`
Returns `True` if the automation has finished, `False` if it is still running. Finished automations remain in the database for analytics.

`Automation.kill()`
Deletes the instance entry in `models.AutomationModel`.

This implies that the execution of the automation is stopped and its history and status are removed from the database. Use this method only if an instance has been created in error, e.g., if you detect invalid arguments after creation. Killing an instance is also removing it from all analytics.

Note: To prematurely stop the execution of an automation consider using `If()` nodes to branch to an `End()` node. This makes the stopping condition explicit in the declaration of an automation.

`Automation.get_key()`
Retrieves a unique key (hash) to be used to identify an automation instance. This has can be used as a `key` parameter to send messages if, e.g., a page is viewed. Just add `?key={{ atm.get_key }}` to the page's url.

5.1.2 flow.Automation.Meta

Meta data on automations can be stored in a nested `Meta` class.

```
class MyAutomation(flow.Automation):
    class Meta:
        ...
```

Currently the following attributes are used.

`Automation.Meta.verbose_name`

`Automation.Meta.verbose_name_plural`
This is a human-readable verbose name of the automation which can be used, e.g., in templates to refer the user to which automation she is for example filling a form.

If unset it will be `Automation <<class__name>>`.

`Automation.get_verbose_name()`

`Automation.get_verbose_name_plural()`
Returns the verbose name set in the automation's meta class, or, if unset, the standard verbose name `"Automation <<class_name>>"` and `"Automations <<class_name>>"`, respectively.

`Automation.Meta.dashboard_template`
: specifies a Django template for rendering the dashboard content of this specififc `Automation` subclass. If not specified the standard template will be used: `automations/includes/dashboard_item.html`. A simple implementation is part of the module but may be overwritten in the projects template path.

`Automation.get_dashboard_context(`*queryset*`)`
: If present this method returns a context dictionary to be added to the rendering context for the automation's dashboard item. It gets passed a queryset of `AutomationModel` instances if the specific automation.

5.1.3 Messages

Automations can receive messages. Messages are used to update an automation instance once it has started, e.g., when a user visits a certain page of your Django project.

Also, messages can be used create an instance of an automation and start it.

Declaring receivers

To receive messages automations have to declare receivers. Receivers are special methods of an automation class. Messages are always received by instances (and not the class itself).

Receivers have names that begin with `receive_` followed by the message name. They take three parameters: `self`, `token`, and `data`.

They have access to the automation's `data` property using `self.data`. After updating `self.data` receivers need to call `self.save()` to avoid changes to be lost.

`token` specifies either an expected action or specifics about the sender of the message. It is either None or of type `str`.

`data` is either a dictionary of additional information or - if called by a template tag or a CMS plugin - a request object. Receivers are not to change the dictionary and only to keep copies and not references to avoid side effects.

Example:

```
def receive_update_subscriber(self, token, data):
    if token == "subscribe":
        self.data['subscriber_list'].append(data['details'])
        self.save()
    elif token == "unsubscribe":
        ...
```

This receiver can be sent the message `"update_subscriber"` and will require a token to specify the expected action.

`Automation.publish_receivers()`
: If this attribute is set to `True` the receivers of an automation are supposed to be visible to the outside. For now this implies that CMS plugins offer the receivers in their forms. Messages can be sent despite the setting on `.publish_receivers`.

  It is common practice not to define this attribute in base classes other classes inherit from to avoid receivers to be offered to users that are present only in base classes.

Sending messages

instance.**send_message**(*message*, *token*, *data*)
automation.send_message() sends the message message to the automation instance automation. Its class needs to have declared a receiver by providing a method named receive_<<message>> where <<message>> is to be replaced by the string message.

token is a string parameter which may be used to give the receiver additional information on, e.g., the sender or the specific content of the message. Sender and receiver are free to agree on its meaning. data typically is a dict of additional data passed to the receiver. The receiving part is supposed not to alter it and to make a copy of it if it is to be stored.

If the message is sent from a template tag or CMS plugin data is the request object.

classmethod Automation.**dispatch_message**(*automation*, *message*, *token*, *data*)
If the first parameter automation is an instance of an automation this is equivalent to automation.send_message(..). If automation is of type int it is interpreted as the id of the automation and the instance is created before it is sent the message. Hence this class method can be used as a shortcut if only an automation's id is known.

classmethod Automation.**create_on_message**(*message*, *token*, *data*)
The class method creates an instance of the automation and immediately sends the message. If the automation is a singleton with respect to certain properties these property values must be given in the data dict or request object.

The message is sent before the automation's run method is called the first time. This means the first Node will not have been executed yet.

classmethod Automation.**broadcast_message**(*message*, *token*, *data*)
The class method sends the message to all running instances of the automation. The order is undefined.

An instance can "catch" a message by returning the string "received". This will stop the broadcast and not all instances might get the message. All other return values do not influence the broadcast.

The class method returns a list of all return values. If the broadcast was caught then the last element in this list will be the string "received".

flow.require_data_parameters

@flow.require_data_parameters(kwargs)**
This decorator for receivers (i.e., methods the name of which starts with receive_) declares that the receiver needs certain parameters of certain type, e.g. email=str denotes that it requires an parameter named email which has the type str (string). The parameters must be present in the data dict or - if data is the request object - in the requests GET parameters.

If a sender does not provide the listed parameters the message will not be sent to the receiver in the first place. Using this decorator avoids that a message is sent if, e.g., the required GET parameters are not present.

classmethod Automation.**satisfies_data_requirements**(*message*, *data*)
This class method checks if data satisfies the declaration of require_data_parameters of the message receiver. If the receiver does not have required data parameters defined, it will return True.

5.1.4 Singletons

work in progress

5.2 flow.This and flow.this

Nodes for an automation are specified as class attributes. To refer to other nodes in the definition of a node, Django automations offers two options:

1. Reference by string literal: `.Next("next_node")` will continue the automation with the node named `next_node`
2. Reference using the global `This` object instance `this`: `.Next(this.next_node)` refers to the Automation objects `next_node` attribute. Since the classes' attributes are not accessible at definition time the this object buffers the reference. It is resolved when an instance is created and executed.

The `this` object serves to avoid unnecessary strings and keep the automation definition less bloated with strings. To use the global `this` instance use `from automations.flow import this`. Alternatively define your own app-specific `this`:

```
from automations import flow

this = flow.This()  # alternative to from automations.flow import this

class TestAutomation(flow.Automation):
    singleton = True

    start = flow.Execute(this.worker_job).Next(this.next). # this notation
    next = flow.Execute("self.worker_job").Next("self.start")  # alternative notation␣
↪with string literals

    def worker_job(self, task_instance):
        ...
```

Alternatively, forward references can be denoted by a string starting with `"self."`. Both forms are equivalent and may be used interchangeably.

5.3 Node class

5.3.1 flow.Node

class flow.**Node**(**args*, ***kwargs*)

Base class for all nodes. Nodes are only functional if bound to a `flow.Automation` subclass as attributes. Other than the `description` `kwarg`, `*args` and `**kwargs` are ignored. Node inherits from `object`.

Nodes use the concept of modifiers to come to a somewhat human readable syntax. Modifiers are methods that return `self`, the node's instance. This implies that modifier be chained just as in Javascript. `SomeNode().AsSoonAs(this.ready).Next(this.end)` is a valid node with two modifiers.

`flow.Node` **is never directly used in any automation,** since it is a base class.

Modifiers for all subclasses of flow.Node

The `flow.Node` class defines the following **modifiers** common to all subclasses. Some subclasses, however, add specific modifiers for their use.

`Node.`**`Next`**`(`*node*`)`
: Sets the node to continue with after finishing this node. If omitted the automation continues with the chronologically next node of the class. `.Next` resembles a goto statement. `.Next` takes a string or a `This` object as a parameter. A string denotes the name of the next node. The this object allows for a different syntax. `.Next("next_node")` and `this.next_node` are equivalent.

`Node.`**`AsSoonAs`**`(`*condition*`)`
: Waits for condition before continuing the automation. If condition is `False` the automation is interrupted and `condition` is checked the next time the automation instance is run.

  If `condition` is callable it will be called every time the condition needs to be evaluated.

`Node.`**`AfterWaitingUntil`**`(`*datetime*`)`
: stops the automation until the specific datetime has passed. Note that depending on how the scheduler runs the automation there might be a significant time slip between `datetime` and the real execution time. It is only guaranteed that the node is not executed before. `datetime` may be a callable.

`Node.`**`AfterWaitingFor`**`(`*timedelta*`)`
: stops the automation for a specific amount of time. This is roughly equivalent to `.AfterWaitingUntil(lambda x: now()+timedelta)`. `timedelta` may be a callable.

`Node.`**`SkipIf`**`(`*condition*`)`
: Skips the current node if `condition` is truthy (i.e., `bool(condition)` evaluates to `True`) or evaluates to a truthy value. The node is left with `"skipped"` in the message field of the `AutomationTaskModel`.

  The `SkipIf()` modifier is useful to skip, e.g., user interactions or sending emails under a certain condition.

`Node.`**`SkipAfter`**`(`*timedelta*`)`
: Skips the current node `timedelta` after its creation. This modifier allows, e.g., to continue after a user interaction has not been received after a certain amount of time.

Note: `.SkipIf()` and `.SkipAfter()` have precedence over waiting/pausing modifiers. If a node is skipped, e.g., it is not guaranteed that the `condition` of `.AsSoonAs()` is fulfilled. If the condition has to be fulfilled separate the modifiers and add them to different nodes.

Attributes

`Node.`**`data`**
: References a JsonField of the node's automation instance. Each instance of an automation can carry additional data in form of a JsonField. This data is shared by all nodes of the automation instance. The node's attribute returns the common JsonField. Any changes in the field need to be saved using `.data.save()` or they might be lost.

  Attached model objects will be referenced by their id in the `.data` attribute. Beyond this the automation may use the data field to save its state in any way it prefers **as long as the dict is json serializable**. This excludes `datetime` objects or `timedelta` objects.

`Node.`**`description`**
: Can be used to provide information about the purpose of a Node. Set the description when defining properties within the automation, e.g.: `start = flow.Execute(this.init, description="This is the first node in my automation")`

Additional methods

Additional methods differ from modifiers since they do **not** return `self`.

`Node.`**`ready`**`(`*automation_instance*`)`
Is called by the newly initialized Automation instance to bind the nodes to the instance. Typically, there is no need to call it from other apps.

`Node.`**`get_automation_name`**`()`
Returns the (dotted) name of the Automation instance class the node is bound to. Automations are identified by this name.

`Node.`**`resolve`**`(`*value*`)`
Resolves the value to the node's automation attribute if `value` is either a `This` object or a string with the name of a node's automation attribute.

5.3.2 flow.End

`class flow.`**`End`**
ends an automation. All finite automations need an `.End()` node. An automation instance that has ended cannot be executed. If you call its `run` method it will throw an error. As long as the automation is not a singleton you can of course at any time instantiate a new instance of the same automation which will run from the start.

5.3.3 flow.Repeat

`class flow.`**`Repeat`**`(`*start=None*`)`
allows for repetitive automations (which do not need an `flow.End()` node). The automation will resume at node given by the `start` argument, or - if omitted - from the first node.

The repetition pattern is described by **modifiers**:

`Repeat.`**`At`**`(`*hour*, *minute*`)`
for daily automations which need to run at a certain hour and minute each day.

`Repeat.`**`EveryHour`**`(`*hours=1*`)`
for hourly automations or automations that need to run at an interval of `hours` hours, repeating based on the time the node initially executes.

`Repeat.`**`EveryNMinutes`**`(`*minutes*`)`
for periodic automations that need to run at an interval of `minutes` minutes, repeating based on the time the node initially executes.

`Repeat.`**`EveryNDays`**`(`*days*`)`
for periodic automations that need to run at an interval of `days` days, repeating based on the time the node initially executes.

`Repeat.`**`EveryDay`**`()`
for daily automations that need to run once each day, repeating based on the time the node initially executes.

5.3.4 flow.Execute

class flow.**Execute**(*func*, *threaded=False*, **args*, ***kwargs*)
runs a callable, typically a method of the automation. The method gets passed a parameter, called `task_instance` which is an instance of the `AutomationTaskModel`. It gives the method access to the processes json database.

The `*args` and `**kwargs` are passed to `func`. If the function returns a json-serializable result it will be stored in the task instance in the database.

Subclass `flow.Execute` to create your own executable nodes, e.g. `class SendEMail(flow.Execute)`. Implement a method named `method`. It gets passed a `task_instance` and all parameters of the node.

`flow.Execute` has one specific modifier.

Execute.**OnError**(*next_node*)
defines a node to continue with in case the `Execute` node fails with an exception. If no `.OnError` modifier is given the automation will stop if an error occurs. The error information is kept in the task instance in the database.

5.3.5 flow.If

class flow.**If**(*condition*)
is a conditional node which needs at least the `.Then()` modifier and optionally can contain an `.Else()` modifier.

If.**Then**(*parameter*)
contains either a callable that is Executed (see `flow.Execute`) or a reference to another node where the automation is continued, if the condition is `True`.

If.**Else**(*parameter*)
specifies what is to be done in case the condition is `False`. If it is omitted the automation continues with the next node.

5.3.6 flow.Split

class flow.**Split**
spawns two or more paths which are to be executed independently. These nodes are given by one or more `.Next()` modifiers. (Example `flow.Split().Next(this.path1).Next(this.path2).Next(this.path3)`). These paths all need to end in the same `flow.Join()` node.

5.3.7 flow.Join

class flow.**Join**
stops the automation until all paths spawned by the same `flow.Split()` have arrived at this node.

5.3.8 flow.SendMessage

class flow.**SendMessage**(*target*, *message*, *token=None*, *data=None*)

Sends a message to other (unfinished) automation instances. `target` can either be an `int` giving the automation id of the automation instance the message is sent to. It can be an Automation instance that receives the message, or it can be an Automation calls. Then the message is sent to all running instances of that class. The class can be replaced by a string with the dotted path to the class definition.

A message is nothing but a method of the receiving class called `receive_<<message>>`. This method will be called for the target instance giving the optional parameters `token` and `data`. Token typically is a string to define more specifically what the message is supposed to mean. `data` can be any python object.

Note: The message is the same mechanism used by the template tags or CMS plugins to send a message when a specific page is rendered. If the message comes from the template tag or plugin `data` is the request object.

5.3.9 flow.Form

class flow.**Form**(*form*, *template_name=None*, *description=''*, *context={}*)

Represents a user interaction with a Django Form. The form's class is passed as `form`. It will be rendered using the optional `template_name`. If `template_name` is not provided, Django automations looks for the `default_template_name` attribute of the automation class. Use the `default_template_name` attribute if all forms of an automation share the same template. If neither is given Django Automations will fall back to `"automations/form_view.html"`.

Also optional is `description`, a text that explains what the user is expected to do with the form, e.g., validate its entries. The description can, e.g., be shown to a user when editing the form, or in her task list.

The form is redered by a Django `FormView`. Additional context for the template is provided by the `FormView`

- project-wide using `settings.ATM_FORM_VIEW_CONTEXT` in the *project's settings file*,
- defining the `context` attribute for the whole Automation class, and
- specifying the `context` parameter in an individual `flow.Form`.

The `flow.Form` has two extra modifiers to assign the task to a user or a group of users:

Form.**User**(***kwargs*)
: assigns the form to a single user who will have to process it. For the time being the user needs to be unique.

Form.**Group**(***kwargs*)
: assigns the form to all members of a user group. Selectors typically are only `id=1` or `name="admins"`.

From.**Permission**(*str*)
: assigns the form to all users who have the permission given by a string dot-formatted: `app_name.codename`. `app_name` is the name of the Django app which provides the permission and `codename` is the permission's name. An example could be `my_app.add_mymodel`. This permission allows an user to add an instance of My_App's `MyModel` model. For details on permissions see Django's Permission documentation. Multiple `.Permission(str)` modifiers can be added implying the a user would require **all** permissions requested.

If more than one modifier is given, `.User`, `.Group`, and `.Permission` have all to be satisfied. If a user loses a required group membership he cannot process the form any more. The same is true for permissions. Superusers can always process the form.

The automation will continue as soon as the form is submitted and validated, i.e. in the request response cycle. If you need to execute an action after this step consider using a threaded `Execute()` not to keep the user waiting for too long.

5.3.10 flow.ModelForm

class flow.**ModelForm**(*form*, *key*, *template_name=None*, *description=''*)

Represents a user interaction with a model. `form` needs to be a subclass of Django's `models.ModelForm`. The model is fixed in the form's `Meta` class (see Django's ModelForm documentation)

5.4 flow.get_automations

flow.**get_automations**(*app=None*)

Returns all automations in the current project (including those in dependencies if they are loaded). All modules or submodules named `automations.py` are searched. If the `app` parameter is given only `app.automations` is searched. Other submodules of `app` are ignored.

The result is a list of tuples, the first one being the automations dotted path, the second one its human readable name. It differs only from the path if `verbose_name` is set in the automations `Meta` subclass.

Note: Because of the way Python's `sys` checks for module modules names, automations (either the automation class or the automations.py file) must be imported somewhere within your project in order to be discoverable. Typically if you are using automations from within the project, it will already require imports, but if you are using an automation solely from the commandline, you can import its class in the `AppConfig.ready()` method within one of your project's apps.

5.5 Models

All automations share the same two models.

5.5.1 models.AutomationModel

All automation instances share a Django model class called `models.AutomationModel`. To distinguish different automations each instance has a field `automation_class` which contains the dotted path to the declaration of the automation class.

classmethod models.AutomationModel.**run**()

This class method is to be called by the scheduler (e.g., through the management command `./manage.py automation_step`) regularly. It will check any unfinished automation instances and process them as appropriate.

classmethod models.AutomationModel.**delete_history**(*days=30*)

Deletes all history of automations finished longer than `days` ago. Once deleted, those automation instances will not be available for analysis any more.

`models.AutomationModel.delete_history` returns a tuple with two entries: The first is the number of deleted objects, the second a dictionary specifying how many automations and how many automation task objects have been deleted from the database.

This class method can be called through the management command `./manage.py automation_delete_history`

Note: Regular deletion of the history keeps the database size from growing endlessly. It also might be required for privacy reasons. Alternatives include removing all person-related data from the automation.

Warning: `models.AutomationModel.delete_history` only affects the database not any instances of `Automation` objects.

All interactions with automations go through their classes and instances. Since the views provide querysets, templates use some additional automation model attributes and methods.

`AutomationModel.`**`automation_class`**
: The `automation_class` field contains the dotted path to the automation class declaration.

`AutomationModel.`**`data`**
: `data` is as json field to store state information of the process it is shared between the tasks, i.e. later tasks see results of earlier tasks if they are retained in the data field. If a Django model is bound to an automation the data field will contain its bound instance id. Bound models are accessed through the automation instance not the automation model (see below).

`AutomationModel.`**`instance`**
: The `instance` property yields the corresponding automation instance. It is often used in templates since the views provide querysets of the `AutomationModel` and access to bound Django models is through the automation instance. To avoid unnecessary instantiations keep the instance if it is needed more than once. In templates this is achieved using the `{% with %}` template tag.

`AutomationModel.`**`get_automation_class`**`()`
: Returns the class of an automation model instance without instantiating it. The class is imported and buffered in the model instance.

`AutomationModel.`**`finished`**
: `True` if the automation instance is finished executing, `False` otherwise.

5.5.2 models.AutomationTaskModel

The automation task model stores start time, end time and result of each task executed. Open tasks have no end time assigned.

Execution errors are stored as task results. If not caught by `.OnError` an error will stop the task **and** automation.

`AutomationTaskModel.`**`created`**
: `DateTimeField` when the task is first entered and created.

`AutomationTaskModel.`**`finished`**
: `DateTimeField` when the task was finished. This field is `None` for open tasks.

`AutomationTaskModel.`**`status`**
: Name of the node corresponding to the task.

`AutomationTaskModel.`**`message`**
: Short message on the result of the task.

- `OK` for `Execute()` nodes which did execute without error. The result returned by the executed function is json serialized in `result` if possible.
- `skipped` if execution did not happen after a `.SkipIf()` modifier

- An error message (i.e., `TypeError(...)`) if an `Execute()` node did fail. *result* ` will contain an dict with a key `error` that contains the traceback.

`AutomationTaskModel.`**`result`**
A json field with the result of an `Execute()` node. See above.

`AutomationTaskModel.`**`automation`**
`ForeignKey` to the `AutomationModel`

`AutomationTaskModel.`**`data`**
Short for `AutomationTaskModel.automation.data`.

`AutomationTaskModel.`**`hours_since_created`**`()`
returns a float indicating the number of hours since the task has been created and has not been finished. Once finished the method returns 0. This is useful if, e.g., the urgency of a task needs to be shown, e.g. by coloring the task item in the task list yellow or red.

5.6 Views

5.6.1 TaskView

5.6.2 TaskListView

5.6.3 DashboardView

5.7 Templates

Django Automations comes with simplistic templates. They are largely thought to be a reference for implementing your project-specific set of templates which probably include some more markup to adapt to your project's look and feel.

All template can be replaced simply by offering alternatives in your project's template folder. This is the structure:

```
└── automations
    ├── base.html
    ├── dashboard.html
    ├── form_view.html
    ├── includes
    │   ├── dashboard.html
    │   ├── dashboard_item.html
    │   ├── form_view.html
    │   ├── task_item.html
    │   └── task_list.html
    └── task_list.html
```

The templates can be replaced individually. It is not necessary (though certainly possible) to replicate the whole tree.

The templates in the `includes` subdirectory are also used by the *Django-CMS plugins*.

5.7.1 base.html

All other templates extend automation's base template. Modify this template to bind into your project's template hierarchy.

5.7.2 cms/empty_template.html

Literally an empty file. Only necessary for the *Django-CMS plugin AutomationHook*. The automation hook does not render anything by using this template.

5.7.3 form_view.html

This is a simple fall-back template if no templates are given in a `Form()` node. Ideally, you specify the correct template by note or process. See *flow.Form*.

5.7.4 task_list.html

This is the template used by the `TaskListView`.

5.8 Template tags

5.9 Management commands

```
python manage.py automation_step
```

This wrapper calls the class method `models.AutomationModel.run()` which in turn lets all automations run which are not waiting for a response (filled form, other condition) or a certain point in time.

```
python manage.py automation_delete_history 14
```

This wrapper calls the class method `models.AutomationModel.delete_history()` which in turn deletes all automations older than the specified number of days. Defaults to 30 days if no argument is provided.

5.10 Settings in settings.py

settings.**ATM_FORM_VIEW_CONTEXT**
: The `Form()` nodes and its subclasses present the forms to the user using a Django `FormView`. This attribute is an dictionary which will be added to the template's context when rendering. The dictionary items may be overwritten by an automation classes' `context` attribute or by a node's `context` parameter. Hence, this setting in practice is used to set default context elements.

settings.**ATM_GROUP_MODEL**
: If your project uses a different model than the built-in `auth.Group` model for grouping users (e.g. using a package like django-organizations or other custom groupings of users), use this setting to specify the model that will be used in place of `auth.Group`. The setting should be specified as `app_label.Model`, and defaults to `auth.Group` if not set in the project-level settings. See *Non-standard Group and Permissions* for additional details.

settings.**ATM_USER_WITH_PERMISSIONS_FORM_METHOD**
: If your project uses non-standard permissions, specify a method for the Form Node that will return a QuerySet of Users based on a the list of Permission codename strings, the User, and Group within the Form Node itself. This setting should be a string of the dotted-path to the location of the replacement method. See *Non-standard Group and Permissions* for additional details.

settings.**ATM_USER_WITH_PERMISSIONS_MODEL_METHOD**
: If your project uses non-standard permissions, specify a method for the AutomationTaskModel that will return a QuerySet of Users based on a the list of Permission codename strings, the User, and Group within the model instance itself. This setting should be a string of the dotted-path to the location of the replacement method. See *Non-standard Group and Permissions* for additional details.

Warning: Either all or none of the following must be present in your project's settings: ATM_GROUP_MODEL, ATM_USER_WITH_PERMISSIONS_FORM_METHOD, ATM_USER_WITH_PERMISSIONS_MODEL_METHOD. Setting only one or two will raise `ImproperlyConfigured`

5.11 Non-standard Group and Permissions

By default, Django Automations assumes your project uses Django's default Group and Permission models, with their default associations to the User model (either `auth.User` or the model specified in `settings.AUTH_USER_MODEL` if you use a custom User model).

To provide flexibility for projects that group users in a non-standard way, Django Automations allows you to swap out the `Group` model and the two methods within the package that retrieve a QuerySet of users that should be allowed access based on the user, group, and permissions specified.

5.11.1 Getting users with access to Form Node instances

`flow.Form` contains a method for retrieving the users who have access to a given Form node. This method, which has access to a list of permission codenames (self._permissions), the assigned user(self._user), and assigned group (self._group), returns a QuerySet of users with applicable permissions that meet the requirements for access. The default method can be overridden in `settings.ATM_USER_WITH_PERMISSIONS_FORM_METHOD`.

The default method used within `form.Flow`:

```
def get_users_with_permission(self):
    from django.contrib.auth.models import Permission

    perm = Permission.objects.filter(codename__in=self._permissions)
    filter_args = Q(groups__permissions__in=perm) | Q(user_permissions__in=perm)
    if self._user is not None:
        filter_args = filter_args & Q(**self._user)
    if self._group is not None:
        filter_args = filter_args & Q(group_set__contains=self._group)
    users = User.objects.filter(filter_args).distinct()
    return users
```

Be sure to include the `self` argument in your replacement method definition and return a Queryset of users.

5.11.2 Getting users with access to `AutomationTaskModel` instances

`AutomationTaskModel` contains a model method for retrieving the users who have access to a given task. This method, which has access to a list of permission codenames (self.interaction_permissions), the assigned user (self.interaction_user), and assigned group (self.interaction_group), returns a QuerySet of users with applicable permissions that meet the requirements for access. The default method can be overridden in `settings.ATM_USER_WITH_PERMISSIONS_MODEL_METHOD`.

The default method used within `AutomationTaskModel`:

```
def get_users_with_permission(
    self,
    include_superusers=True,
    backend="django.contrib.auth.backends.ModelBackend",
):
    users = User.objects.all()
    for permission in self.interaction_permissions:
        users &= User.objects.with_perm(permission, include_superusers=False,␣
↪backend=backend)
    if self.interaction_user is not None:
        users = users.filter(id=self.interaction_user_id)
    if self.interaction_group is not None:
        users = users.filter(groups=self.interaction_group)
    if include_superusers:
        users |= User.objects.filter(is_superuser=True)
    return users
```

Be sure to include the `self`, `include_superusers`, and `backend` arguments in your replacement method definition and return a Queryset of users. The value of `backend` can be modified to match the backend your project uses for authentication.

5.12 Django-CMS integration

The Django-CMS dependency is weak. Installing Django Automations will not require or trigger the installation of Django-CMS.

Note: If you want to use Django Automations's CMS plugins, be sure to include `automations.cms_automations` in your `INSTALLED_APPS` settings.

Alternatively pure Django users can use *template tags* instead.

5.12.1 CMS Plugins

Task List Plugin

class AutomationTaskList

This plugin shows all interactions required for automations to continue their work from the current user. It will never show tasks for the anonymous user (nobody logged in).

With this plugin the task list can be part of any CMS page. It is helpful if the user's tasks are to be shown as a part of a page, say, a dashboard.

In the project settings a choice of template can be defined. CMS page authors can chose the appropriate template do adjust the plugin's look and feel.

Status Plugin

class AutomationStatus

This plugin allows a user to see a detailed status of an automation instance. The automation instance is defined by a get parameter: `key` is an unique identifier for an automation instance.

Automations may chose to offer status templates. They have to be declared in the Automations Meta class:

```
class MyAutomation(flow.Automation):
    class Meta:
        status_template = "my_automation/status.html", _("Current status")
        issue_template = "my_automation/issues.html", _("Problem sheet")
```

Any property with a name that ends on `_template` in the automation's Meta class is considered to be a template path for some sort of status view. For user friendliness a verbose name can be added. Once declared the plugin will offer all status templates.

The templates receive the automation instance in the context with the key `automation` and the corresponding automation model instance with the key `automation_model`.

Send Message Plugin

class AutomationHook

The automation hook does not display or render anything. Its purpose is to send a message to the automation, if a page is viewed. If on this page this plugin should be included. It offers all receiving automations and its receiver ports.

An automation declares an receiving slot by defining a method with a name starting with `receive_`, e.g., `receive_add_participant_to_webinar`. All such slots are open for the Send Message Plugin and the example will appear as "Add participant to webinar" (capitalized, and underscores replaced by spaces) if the `Automation.publish_receivers` is set to `True`.

The receiver will be passed an optional token and a data object which in this case is the request object.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search

INDEX

M

N

O

P

R

S

T

U

V