Towel Documentation

Release 0.8.1

Feinheit GmbH and contributors

Aug 25, 2018

Contents

Towel is a collection of tools which make your life easier if you are building a web application using Django. It contains helpers and templates for creating paginated, searchable object lists, CRUD functionality helping you safely and easily create and update objects, and using Django's own proofed machinery to see what happens when you want to safely delete objects.

CHAPTER 1

Contents

1.1 Installation instructions

This document describes the steps needed to get Towel up and running.

Towel is based on Django, so you need a working Django installation first. Towel is mainly developed using the newest release of Django, but should work with Django 1.4 up to the upcoming 1.7 and with Python 2.7 and 3.3. Towel does not currently support Python 3.2 but patches adding support are welcome.

Towel can be installed using the following command:

```
$ pip install Towel
```

Towel has no dependencies apart from Django.

You should add `towel` to `INSTALLED_APPS` if you want to use the bundled templates and template tags. This isn't strictly required though.

1.2 Towel API

`towel.api` is a set of classes which facilitate building a RESTful API. In contrast to other, well known projects such as django-piston and tastypie it does not try to cover all HTTP verbs out of the box, and does not come with as many configuration knobs and classes for everything, and tries staying small and simple instead.

The API consists of the following classes and methods, which are explained in more depth further down this page:

- *`API`*: A collection of resources.
- *`Resource`*: A single resource which exposes a Django model instance.
- *`Serializer`*: The API response serializer, responsible for content type negotiation and creation of `HttpResponse` instances.
- *`RequestParser`*: Understands requests in various formats (JSON, urlencoded, etc.) and handles the differences.

- *APIException*: An exception which can be raised deep down in the API / resource machinery and will be converted into a nicely formatted response in the requested content format.
- *Objects* and *Page*: Containers for objects related to a particular resource and / or URI. They are returned by the method `Resource.objects()`.
- *api_reverse()*: Helper for reversing URLs inside a particular API instance.
- *serialize_model_instance()*: The default Django model serializer.
- *querystring()*: Helper for constructing querystrings.

1.2.1 The `API` class

class towel.api.**API**(*name, decorators=[csrf_exempt]*)

This class acts as a collection of resources. The arguments are:

- `name`: The name of this API. If you don't know what to use here, simply use `'v1'`.
- `decorators`: A list of decorators which should be applied to the root API view and to all resources (if you don't override it upon resource registration). The list of decorators is applied in reverse, which means that you should follow the same order as if you were using the `@decorator` notation. It's recommended to always use `csrf_exempt()` here, otherwise API requests other than GET, HEAD, OPTIONS and TRACE (the HTTP verbs defined as safe by RFC2616) will have to include a valid CSRF middleware token.

Example:

```
api_v1 = API('v1')
```

name

The name of this API.

decorators

The decorators passed upon initialization.

resources

A list of dictionaries holding resource configuration.

serializers

A dictionary mapping models to serialization functions. If a model does not exist inside this dictionary, the default serialization function *serialize_model_instance()* is used.

urls

This property returns a URL pattern instance suitable for including inside your main URLconf:

```
from .views import api_v1

urlpatterns = patterns('',
    url(r'^api/v1/', include(api_v1.urls)),
)
```

register(*self*, *model*, *view_class=None*, *canonical=True*, *decorators=None*, *prefix=None*, *view_init=None*, *serializer=None*)

Resources are normally not created by hand. This method should be used instead. The arguments are:

- `model`: The Django model used in this resource.
- `view_class`: The resource view class used, defaults to *Resource*.

- `canonical`: Whether this resource is the canonical location of the model in this API. Allows registering the same model several times in the API (only one location should be the canonical location!)
- `decorators`: A list of decorators which should be applied to the view. Function decorators only, method decorators aren't supported. The list is applied in reverse, the order is therefore the same as with the `@decorator` notation. If unset, the set of decorators is determined from the API initialization. Pass an empty list if you want no decorators at all.
- `prefix`: The prefix for this model, defaults to the model name in lowercase. You should include a caret and a trailing slash if you specify this yourself (`prefix=r'^library/'`).
- `view_init`: Python dictionary which contains keyword arguments used during the instantiation of the `view_class`.
- `serializer`: Function which takes a model instance, the API instance and additional keyword arguments (accept `**kwargs` for forward compatibility) and returns the serialized representation as a Python dictionary.

serialize_instance(*self*, *instance*, ***kwargs*)
Returns a serialized version of the passed model instance.

This method should always be used for serialization, because it knows about custom serializers specified when registering resources with this API.

root(*self*, *request*)
Main API view, returns a list of all available resources

1.2.2 Resources

class towel.api.**Resource**(*self*, ***kwargs*)
This is a `View` subclass with additional goodies for exposing a Django model in a RESTful way. You should not instantiate this class yourself, but use *`API.register()`* instead.

api
The *`API`* instance to which this resource is bound to.

model
The model exposed by this resource.

queryset
Prefiltered queryset for this resource or `None` if all objects accessible through the first defined manager on the model should be exposed (or if you do the limiting yourself in `Resource.get_query_set()`)

limit_per_page
Standard count of items in a single request. Defaults to 20. This can be overridden by sending a different value with the `limit` querystring parameter.

max_limit_per_page
Maximal count of items in a single request. `limit` query values higher than this are not allowed. Defaults to 1000.

http_method_names
Allowed HTTP method names. The *`Resource`* only comes with implementations for GET, HEAD and OPTIONS. You have to implement all other handlers yourself.

A typical request-response cycle

`Resource`.**dispatch**(*self*, *request*, **args*, ***kwargs*)
This method is the primary entry point for requests. It is similar to the base class implementation but has a few

important differences:

- It uses `self.request`, `self.args` and `self.kwargs` in all places.
- It calls *`unserialize_request()`* after assigning the aforementioned variables on `self` which may modify all aspects and all variables (f.e. deserialize a JSON request and serialize it again to look like a standard POST request) and only then determines whether the request should be handled by this view at all.
- The return value of the *`get()`*, *`post()`* etc. methods is passed to *`serialize_response()`* and only then returned to the client. The processing methods should return a dictionary which is then serialized into the requested format. If the format is unknown or unsupported, a 406 Not acceptable HTTP error is returned instead.
- *`APIException`* and `Http404` exceptions are caught and transformed into appropriate responses according to the content type requested.

`Resource.`**`unserialize_request`**(*self*)

This method's intent is to standardize various aspects of the incoming request so that the following code does not have to care about the format of the incoming data. It might decode incoming JSON data and reformat it as a standard HTTP POST.

Currently, this method does nothing, and because of that, content is only accepted in two forms:

- urlencoded in the request body
- multipart in the request body

`Resource.`**`get`**(*self*, *request*, **args*, ***kwargs*)

`Resource.`**`head`**(*self*, *request*, **args*, ***kwargs*)

These methods return serialized lists, sets or details depending upon the request URI.

All of the following are valid URIs for a fictional resource for books:

- `/api/v1/book/`: Returns 20 books.
- `/api/v1/book/?offset=20&limit=20`: Returns books 21-40.
- `/api/v1/book/42/`: Returns the book with the primary key of 42.
- `/api/v1/book/1;3;15/`: Returns a set of three books.

The *`get()`* method offloads processing into three distinct methods depending upon the URI:

`get_single`(*self*, *request*, *objects*, **args*, ***kwargs*)

`Resource.`**`get_set`**(*self*, *request*, *objects*, **args*, ***kwargs*)

`Resource.`**`get_page`**(*self*, *request*, *objects*, **args*, ***kwargs*)

These methods receive an *`Objects`* instance containing all instances they have to process. The default implementation of all these methods use *`API.serialize_instance()`* to do the serialization work (using the *`API`* instance at *`Resource.api`*).

If any of the referenced objects do not exist for the single and the set case, a HTTP 404 is returned instead of returning a partial response.

The list URI does not only return a list of objects, but another mapping containing metadata about the response such as URIs for the previous and next page (if they exist) and the total object count.

`Resource.`**`options`**(*self*, *request*, **args*, ***kwargs*)

Returns a list of allowed HTTP verbs in the `Allow` response header. The response is otherwise empty.

Note: URIs inside the resource might still return 405 Method not allowed erorrs if a particular HTTP verb is only implemented for a subset of URIs, for example only for single instances.

`Resource.`**`post`**(*self*, *request*, **args*, ***kwargs*)

`Resource.`**`put`**(*self*, *request*, **args*, ***kwargs*)

`Resource.`**`delete`**(*self*, *request*, **args*, ***kwargs*)

`Resource.`**`patch`**(*self*, *request*, **args*, ***kwargs*)

`Resource.`**`trace`**(*self*, *request*, **args*, ***kwargs*)
Default implementations do not exist, that means that if you do not provide your own, the only answer will ever be a HTTP 405 Method not allowed error.

`Resource.`**`serialize_response`**(*self*, *response*, *status=httplib.OK*, *headers={}*)
This method is a thin wrapper around `Serializer.serialize()`. If `response` is already a `HttpResponse` instance, it is returned directly.

The content types supported by *`Serializer`* are JSON, but more on that later.

1.2.3 The serializer

`class` `towel.api.`**`Serializer`**

The API supports output as JSON. The format is determined by looking at the HTTP `Accept` header first. If no acceptable encoding is found, a HTTP 406 Not acceptable error is returned to the client.

The detection of supported content types can be circumvented by adding a querystring parameter naemd `format`. The supported values are as follows:

- `?format=json` or `?format=application/json` for JSON output

1.2.4 The request parser

`class` `towel.api.`**`RequestParser`**
Parses the request body into a format independent of its content type.

Does nothing for the following HTTP methods because they are not supposed to have a request body:

- `GET`
- `HEAD`
- `OPTIONS`
- `TRACE`
- `DELETE`

Otherwise, the code tries determining a parser for the request. The following content types are supported:

- `application/x-www-form-urlencoded` (the default)
- `multipart/form-data`
- `application/json`

The two former content types are supported directly by Django, all capabilities and restrictions are inherited directly. When using JSON, file uploads are not supported.

The parsed data is available as `request.POST` and `request.FILES`. `request.POST` is used instead of something else even for `PUT` and `PATCH` requests (among others), because most code written for Django expects data to be provided under that name.

parse(*self*, *request*)
Decides whether the request body should be parsed, and if yes, decides which parser to use. Returns a HTTP 415 Unsupported media type if the request isn't understood.

parse_form(*self*, *request*)

parse_json(*self*, *request*)
The actual work horses.

1.2.5 Additional classes and exceptions

exception towel.api.**APIException**(*error_message=None*, *status=None*, *data={}*)
Custom exception which signals a problem detected somewhere inside the API machinery.

Usage:

```
# Use official W3C error names from ``httplib.responses``
raise ClientError(status=httplib.NOT_ACCEPTABLE)
```

or:

```
raise ServerError('Not implemented, go away',
    status=httplib.NOT_IMPLEMENTED)
```

Additional information can be passed through by setting the `data` argument to a dict instance. The *APIException* handler will merge the dict into the default error data and return everything to the client:

```
raise APIException('Validation failed', data={
    'form': form.errors,
    })
```

class towel.api.**Objects**(*queryset*, *page*, *set*, *single*)
A `namedtuple` holding the return value of `Resource.objects()`.

class towel.api.**Page**(*queryset*, *offset*, *limit*, *total*)
A `namedtuple` for the `page` object from *Objects* above.

1.2.6 Utility functions

towel.api.**api_reverse**(*model*, *ident*, *api_name='api'*, *fail_silently=False*, ***kwargs*)
Determines the canonical URL of API endpoints for arbitrary models.

- `model` is the Django model you want to use,
- `ident` should be one of `list`, `set` or `detail` at the moment
- Additional keyword arguments are forwarded to the `reverse()` call.

Usage:

```
api_reverse(Product, 'detail', pk=42)
```

Passing an instance works too:

```
api_reverse(instance, 'detail', pk=instance.pk)
```

towel.api.**serialize_model_instance**(*instance*, *api*, *inline_depth=0*, *exclude=()*, *only_registered=True*, *build_absolute_uri=lambda uri: uri*, ***kwargs*)

Serializes a single model instance.

If `inline_depth` is a positive number, `inline_depth` levels of related objects are inlined. The performance implications of this feature might be severe! Note: Additional arguments specified when calling `serialize_model_instance` such as `exclude`, `only_registered` and further keyword arguments are currently **not** forwarded to inlined objects. Those parameters should be set upon resource registration time as documented in the `API` docstring above.

The `exclude` parameter is especially helpful when used together with `functools.partial`.

Set `only_registered=False` if you want to serialize models which do not have a canonical URI inside this API.

`build_absolute_uri` should be a callable which transforms any passed URI fragment into an absolute URI including the protocol and the hostname, for example `request.build_absolute_uri`.

This implementation has a few characteristics you should be aware of:

- Only objects which have a canonical URI inside this particular API are serialized; if no such URI exists, this method returns `None`. This behavior can be overridden by passing `only_registered=False`.
- Many to many relations are only processed if `inline_depth` has a positive value. The reason for this design decision is that the database has to be queried for showing the URIs of related objects anyway and because of that we either show the full objects or nothing at all.
- Some fields (currently only fields with choices) have a machine readable and a prettified value. The prettified values are delivered inside the `__pretty__` dictionary for your convenience.
- The primary key of the model instance is always available as `__pk__`.

towel.api.**querystring**(*data*, *exclude=()*, ***kwargs*)

Returns a properly encoded querystring

The supported arguments are as follows:

- `data` should be a `MultiValueDict` instance (i.e. `request.GET`)
- `exclude` is a list of keys from `data` which should be skipped
- Additional key-value pairs are accepted as keyword arguments

Usage:

```
next_page_url = querystring(
    request.GET,
    exclude=('page',),
    page=current + 1,
    )
```

1.2.7 API behavior

Resource list

The available resources can be determined by sending a request to the root URI of this API, `/api/v1/`. Resources can either be canonical or not.

All resources are returned in a list, the canonical URIs for objects are additionally returned as a hash.

The individual resources are described by a hash containing two values (as do most objects returned by the API):

- `__uri__`: The URI of the particular object
- `__str__`: A string containing the 'name' of the object, whatever that would be (it's the return value of the `__str__` method for Django models, and the lowercased class name of the model registered with the resource).

In the list of resources, a particular `__str__` value will exist several times if a model is exposed through more than one resource; `__uri__` values will always be unique.

Listing endpoints

All API endpoints currently support GET, HEAD and OPTIONS requests.

All listing endpoints support the following parameters:

- `?limit=<integer>`: Determines how many objects will be shown on a single page. The default value is 20. The lower limit is zero, the upper limit is determined by the variable `max_limit_per_page` which defaults to 1000.
- `?offset=<integer>`: Can be used for retrieving a different page of objects. Passing `?offset=20` with a limit of 20 will return the next page. The offset is zero-indexed.

Note: You won't have to construct query strings containing these parameters yourself in most cases. All list views return a mapping with additional information about the current request and `next` and `previous` links for your convenience as well.

List views return two data structures, `objects` and `meta`. The former is a list of all objects for the current request, the latter a mapping of additional information about the current set of objects:

- `offset`: The offset value as described above.
- `limit`: The limit value as described above.
- `total`: The total count of objects.
- `previous`: A link to the previous page or `null`.
- `next`: A link to the next page or `null`.

Object representation

The following fields should always be available on objects returned:

- `__uri__`: The URI.
- `__pk__`: The primary key of this object.
- `__str__`: The return value of the `__str__` or `__unicode__` method.

A few fields' values have to be treated specially, because their values do not have an obvious representation in an JSON document. The fields and their representations are as follows:

- `date` and `datetime` objects are converted into strings using `str()`.
- `Decimal` is converted into a string without (lossy) conversion to `float` first.
- `FileField` and `ImageField` are shown as the URL of the file.
- `ForeignKey` fields are shown as their canonical URI (if there exists such a URI inside this API) or even inlined if `?full=1` is passed when requesting the details of an object.

1.3 ModelView

We'll start with simple object list and object detail pages, explaining many provided tools along the way. Next, this guide covers the CRUD part of Towel, talk about batch processing a bit and end up with explaining a few components in more detail.

Warning: Please note that Towel's ModelView could be considered similar to Django's own generic views. However, they do not have the same purpose and software design: Django's generic views use one class per view, and every instance only processes one request. Towel's ModelView is more similar to Django's admin site in that one instance is responsible for many URLs and handles many requests. You have to take care not to modify ModelView itself during request processing, because this will not be thread-safe.

1.3.1 Preparing your models, views and URLconfs for ModelView

ModelView has a strong way of how Django-based web applications should be written. The rigid structure is necessary to build a well-integrated toolset which will bring you a long way towards successful completion of your project. If you do not like the design decisions made, ModelView offers hooks to customize the behavior, but that's not covered in this guide.

For this guide, we assume the following model structure and relationships:

```
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=100)
    address = models.TextField()

class Author(models.Model):
    name = models.CharField(max_length=100)
    date_of_birth = models.DateField(blank=True, null=True)

class Book(models.Model):
    title = models.CharField(max_length=100)
    topic = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    published_on = models.DateField()
    publisher = models.ForeignKey(Publisher)
```

ModelView works with an URL structure similar to the following:

- `/books/`
- `/books/add/`

- /books/<pk>/
- /books/<pk>/edit/
- /books/<pk>/delete/

The regular expression used to match the detail page (here <pk>) can be customized. If you'd rather match on the slug, on a combination of several fields (separated by dashes or slashes, whatever you want) or on something else, you can do this by modifying *urlconf_detail_re*. You only have to make sure that *get_object()* will know what to do with the extracted parameters.

If you want to use the primary key-based URL configuration, you do not need to add a `get_absolute_url()` method to your model, because *ModelView* will add one itself. It isn't considered good practice to put primary keys on the web for everyone to see but it might be okay for your use case.

1.3.2 The main `ModelView` class

class towel.modelview.**ModelView**(*model*[, ...])

The first and only required argument when instantiating a model view is the Django model. Additional keyword arguments may be used to override attribute values of the model view class. It is not allowed to pass keyword arguments which do not exist as attributes on the class already.

urlconf_detail_re
The regular expression used for detail pages. Defaults to a regular expression which only accepts a numeric primary key.

paginate_by
Objects per page for list views. Defaults to `None` which means that all objects are shown on one page (usually a bad idea).

pagination_all_allowed
Pagination can be deactivated by passing `?all=1` in the URL. If you expect having lots of objects in the table showing all on one page can lead to a very slow and big page being shown. Set this attribute to `False` to disallow this behavior.

paginator_class
Paginator class which should have the same interface as `django.core.paginator.Paginator`. Defaults to `towel.paginator.Paginator` which is almost the same as Django's, but offers additional methods for outputting Digg-style pagination links.

template_object_name
The name used for the instance in detail and edit views. Defaults to `object`.

template_object_list_name
The name used for instances in list views. Defaults to `object_list`.

base_template
The template which all standard modelview templates extend. Defaults to `base.html`.

form_class
The form class used to create and update models. The method *get_form()* returns this value instead of invoking `modelform_factory()` if it is set. Defaults to `None`.

search_form
The search form class to use in list views. Should be a subclass of `towel.forms.SearchForm`. Defaults to `None`, which deactivates search form handling.

search_form_everywhere
Whether a search form instance should be added to all views, not only to list views. Useful if the search form is shown on detail pages as well.

batch_form
The batch form class used for batch editing in list views. Should be a subclass of `towel.forms.BatchForm`. Defaults to `None`.

default_messages
A set of default messages for various success and error conditions. You should not modify this dictionary, but instead override messages by adding them to *custom_messages* below. The current set of messages is:

- `object_created`
- `adding_denied`
- `object_updated`
- `editing_denied`
- `object_deleted`
- `deletion_denied`
- `deletion_denied_related`

Note that by modifying this dictionary you are modifying it for all model view instances!

custom_messages
A set of custom messages for custom actions or for overriding messages from *custom_messages*.

Note that by modifying this dictionary you are modifying it for all model view instances! If you want to override a few messages only for a particular model view instance, you have to set this attribute to a new dictionary instance, not update the existing dictionary.

view_decorator(*self*, *func*)

crud_view_decorator(*self*, *func*)
The default implementation of `get_urls()` uses those two methods to decorate all views, the former for list and detail views, the latter for add, edit and delete views.

Models and querysets

`towel.modelview.ModelView.`**get_query_set**(*self*, *request*, **args*, ***kwargs*)
This method should return a queryset with all objects this modelview is allowed to see. If a certain user should only ever see a subset of all objects, add the permission checking here. Example:

```
class UserModelView(ModelView):
    def get_query_set(self, request, *args, **kwargs):
        return self.model.objects.filter(created_by=request.user)
```

`towel.modelview.ModelView.`**get_object**(*self*, *request*, **args*, ***kwargs*)
Returns a single object for the query parameters passed as `args` and `kwargs` or raises a `ObjectDoesNotExist` exception. The default implementation passes all args and kwargs to a `get()` call, which means that all parameters extracted by the *urlconf_detail_re* regular expression should uniquely identify the object in the queryset returned by *get_query_set()* above.

`towel.modelview.ModelView.`**get_object_or_404**(*self*, *request*, **args*, ***kwargs*)
Wraps *get_object()*, but raises a `Http404` instead of a `ObjectDoesNotExist`.

Object lists

Towel's object lists are handled by *list_view()*. By default, all objects are shown on one page but this can be modified through *paginate_by*. The following code puts a paginated list of books at /books/:

```
from myapp.models import Book
from towel.modelview import ModelView

class BookModelView(ModelView):
    paginate_by = 20

book_views = BookModelView(Book)

urlpatterns = patterns('',
    url(r'^books/', include(book_views.urls)),
)
```

This can even be written shorter if you do not want to override any ModelView methods:

```
from myapp.models import Book
from towel.modelview import ModelView

urlpatterns = patterns('',
    url(r'^books/', include(ModelView(Book, paginate_by=20).urls)),
)
```

The model instances are passed as object_list into the template by default. This can be customized by setting *template_object_list_name* to a different value.

The *list_view()* method does not contain much code, and simply defers to other methods who do most of the grunt-work. Those methods are shortly explained here.

towel.modelview.ModelView.**list_view**(*self*, *request*)
: Main entry point for object lists, calls all other methods.

towel.modelview.ModelView.**handle_search_form**(*self*, *request*, *ctx*, *queryset=None*)

towel.modelview.ModelView.**handle_batch_form**(*self*, *request*, *ctx*, *queryset*)
: These methods are discussed later, under *List Searchable* and *Batch processing*.

towel.modelview.ModelView.**paginate_object_list**(*self*, *request*, *queryset*, *paginate_by=10*)
: If paginate_by``is given paginates the object list using the ``page GET parameter. Pagination can be switched off by passing all=1 in the GET request. If you have lots of objects and want to disable the all=1 parameter, set pagination_all_allowed to False.

towel.modelview.ModelView.**render_list**(*self*, *request*, *context*)
: The rendering of object lists is done inside render_list. This method calls get_template to assemble a list of templates to try, and get_context to build the context for rendering the final template. The templates tried are as follows:

  - <app_label>/<model_name>_list.html (in our case, myapp/book_list.html)
  - modelview/object_list.html

  The additional variables passed into the context are documented in *Standard context variables*.

1.3.3 List Searchable

Please refer to the *Search and Filter* page for information about filtering lists.

1.3.4 Object detail pages

Object detail pages are handled by `detail_view()`. All parameters captured in the *`urlconf_detail_re`* regex are passed on to *`get_object_or_404()`*, which passes them to *`get_object()`*. *`get_object()`* first calls *`get_query_set()`*, and tries finding a model thereafter.

The rendering is handled by `render_detail()`; the templates tried are

- `<app_label>/<model_name>_detail.html` (in our case, `myapp/book_detail.html`)
- `modelview/object_detail.html`

The model instance is passed as `object` into the template by default. This can be customized by setting `template_object_name` to a different value.

1.3.5 Adding and updating objects

Towel offers several facilities to make it easier to build and process complex forms composed of forms and formsets. The code paths for adding and updating objects are shared for a big part.

`add_view` and `edit_view` are called first. They defer most of their work to helper methods.

`towel.modelview.ModelView.`**`add_view`**(*self*, *request*)
`add_view` does not accept any arguments.

`towel.modelview.ModelView.`**`edit_view`**(*self*, *request*, **args*, ***kwargs*)
`args` and `kwargs` are passed as they are directly into *`get_object()`*.

`towel.modelview.ModelView.`**`process_form`**(*self*, *request*, *intance=None*, *change=None*)
These are the common bits of *`add_view()`* and *`edit_view()`*.

`towel.modelview.ModelView.`**`get_form`**(*self*, *request*, *instance=None*, *change=None*, ***kwargs*)
Return a Django form class. The default implementation returns the result of calling `modelform_factory()`. Keyword arguments are forwarded to the factory invocation.

`towel.modelview.ModelView.`**`get_form_instance`**(*self*, *request*, *form_class*, *instance=None*, *change=None*, ***kwargs*)
Instantiate the form, for the given instance in the editing case.

The arguments passed to the form class when instantiating are determined by `extend_args_if_post` and `**kwargs`.

`towel.modelview.ModelView.`**`extend_args_if_post`**(*self*, *request*, *args*)
Inserts `request.POST` and `request.FILES` at the beginning of `args` if `request.method` is `POST`.

`towel.modelview.ModelView.`**`get_formset_instances`**(*self*, *request*, *instance=None*, *change=None*, ***kwargs*)
Returns an empty `dict` by default. Construct your formsets if you want any in this method:

```
BookFormSet = inlineformset_factory(Publisher, Book)

class PublisherModelView(ModelView):
    def get_formset_instances(self, request, instance=None, change=None,␣
↪**kwargs):
        args = self.extend_args_if_post(request, [])
        kwargs.setdefault('instance', instance)

        return {
            'books': BookFormSet(prefix='books', *args, **kwargs),
            }
```

towel.modelview.ModelView.**save_form**(*self*, *request*, *form*, *change*)
Return an unsaved instance when editing an object. `change` is `True` if editing an object.

towel.modelview.ModelView.**save_model**(*self*, *request*, *instance*, *form*, *change*)
Save the instance to the database. `change` is `True` if editing an object.

towel.modelview.ModelView.**save_formsets**(*self*, *request*, *form*, *formsets*, *change*)
Iterates through the `formsets` `dict`, calling `save_formset` on each.

towel.modelview.ModelView.**save_formset**(*self*, *request*, *form*, *formset*, *change*)
Actually saves the formset instances.

towel.modelview.ModelView.**post_save**(*self*, *request*, *form*, *formsets*, *change*)
Hook for adding custom processing after forms, formsets and m2m relations have been saved. Does nothing by default.

towel.modelview.ModelView.**render_form**(*self*, *request*, *context*, *change*)
Offloads work to `get_template`, `get_context` and `render_to_response`. The templates tried when rendering are:

- `<app_label>/<model_name>_form.html`
- `modelview/object_form.html`

towel.modelview.ModelView.**response_add**()

towel.modelview.ModelView.**response_edit**()
They add a message using the `django.contrib.messages` framework and redirect the user to the appropriate place, being the detail page of the edited object or the editing form if `_continue` is contained in the POST request.

1.3.6 Object deletion

Object deletion through ModelView is forbidden by default as a safety measure. However, it is very easy to allow deletion globally:

```
class AuthorModelView(ModelView):
    def deletion_allowed(self, request, instance):
        return True
```

If you wanted to allow deletion only for the creator, you could use something like this:

```
class AuthorModelView(ModelView):
    def deletion_allowed(self, request, instance):
        # Our author model does not have a created_by field, therefore this
        # does not work.
        return request.user == instance.created_by
```

Often, you want to allow deletion, but only if no related objects are affected by the deletion. ModelView offers a helper to do that:

```
class PublisherModelView(ModelView):
    def deletion_allowed(self, request, instance):
        return self.deletion_allowed_if_only(request, instance, [Publisher])
```

If there are any books in our system published by the given publisher instance, the deletion would not be allowed. If there are no related objects for this instance, the user is asked whether he really wants to delete the object. If he confirms, the instance is or the instances are deleted for good, depending on whether there are related objects or not.

Deletion of inline formset instances

Django's inline formsets are very convenient to edit a set of related objects on one page. When deletion of inline objects is enabled, it's much too easy to lose related data because of Django's cascaded deletion behavior. Towel offers helpers to allow circumventing Django's inline formset deletion behavior.

Note: The problem is that `formset.save(commit=False)` deletes objects marked for deletion right away even though `commit=False` might be interpreted as not touching the database yet.

The models edited through inline formsets have to be changed a bit:

```
from django.db import models
from towel import deletion

class MyModel(deletion.Model):
    field = models.CharField(...) # whatever
```

`deletion.Model` only consists of a customized `Model.delete` method which does not delete the model under certain circumstances. See the *Deletion* API documentation if you need to know more.

Next, you have to override `save_formsets`:

```
class MyModelView(modelview.ModelView):
    def get_formset_instances(self, request, instance=None, change=None, **kwargs):
        args = self.extend_args_if_post(request, [])
        kwargs['instance'] = instance

        return {
            'mymodels': InlineFormSet(*args, **kwargs),
            }

    def save_formsets(self, request, form, formsets, change):
        # Only delete MyModel instances if there are no related objects
        # attached to them
        self.save_formset_deletion_allowed_if_only(
            request, form, formsets['mymodels'], change, [MyModel])
```

Warning: `save_formset_deletion_allowed_if_only` calls `save_formset` do actually save the formset. If you need this customized behavior, you must not call `save_formset_deletion_allowed_if_only` in `save_formset` or you'll get infinite recursion.

1.3.7 Standard context variables

The following variables are always added to the context:

- `verbose_name`
- `verbose_name_plural`
- `list_url`
- `add_url`
- `base_template`
- `search_form` if *`search_form_everywhere`* is `True`

`RequestContext` is used, therefore all configured context processors are executed too.

1.3.8 Permissions

`get_urls()` assumes that there are two groups of users with potentially differing permissions: Those who are only allowed to view and those who may add, change or update objects.

To restrict viewing to authenticated users and editing to managers, you could do the following:

```
from django.contrib.admin.views.decorators import staff_member_required
from django.contrib.auth.decorators import login_required

book_views = BookModelView(Book,
    search_form=BookSearchForm,
    paginate_by=20,
    view_decorator=login_required,
    crud_view_decorator=staff_member_required,
    )
```

If *`crud_view_decorator()`* is not provided, it defaults to *`view_decorator()`*, which defaults to returning the function as-is. This means that by default, you do not get any view decorators.

Additionally, ModelView offers the following hooks for customizing permissions:

`towel.modelview.ModelView.`**`adding_allowed`**(*self*, *request*)

`towel.modelview.ModelView.`**`editing_allowed`**(*self*, *request*, *instance*)
Return `True` by default.

`towel.modelview.ModelView.`**`deletion_allowed`**(*self*, *request*, *instance*)
Was already discussed under *Object deletion*. Returns `False` by default.

1.3.9 Batch processing

Suppose you want to change the publisher for a selection of books. You could do this by editing each of them by hand, or by thinking earlier and doing this:

```
from django import forms
from django.contrib import messages
from towel import forms as towel_forms
from myapp.models import Book, Publisher

class BookBatchForm(towel_forms.BatchForm):
    publisher = forms.ModelChoiceField(Publisher.objects.all(), required=False)

    formfield_callback = towel_forms.towel_formfield_callback

    def _context(self, batch_queryset):
        data = self.cleaned_data

        if data.get('publisher'):
            messages.success(request, 'Updated %s books.' % (
                batch_queryset.update(publisher=data.get('publisher')),
                ))

        return {
```

(continues on next page)

(continued from previous page)

```
            'batch_items': batch_queryset,
            }
```

Activate the batch form like this:

```
book_views = BookModelView(Book,
    batch_form=BookBatchForm,
    search_form=BookSearchForm,
    paginate_by=20,
    )
```

If you have to return a response from the batch form (f.e. because you want to generate sales reports for a selection of books), you can return a response in `_context` using the special-cased key `response`:

```
def _context(self, batch_queryset):
    # [...]

    return {
        'response': HttpResponse(your_report,
            content_type='application/pdf'),
        }
```

1.4 Search and Filter

Towel does not distinguish between searching and filtering. There are different layers of filtering applied during a request and depending on your need you have to hook in your filter at the right place.

1.4.1 Making lists searchable using the search form

Pagination is not enough for many use cases, we need more! Luckily, Towel has a pre-made solution for searching object lists too.

`towel.forms.SearchForm` can be used together with `towel.managers.SearchManager` to build a low-cost implementation of full text search and filtering by model attributes.

The method used to implement full text search is a bit stupid and cannot replace mature full text search solutions such as Apache Solr. It might just solve 80% of the problems with 20% of the effort though.

Code talks. First, we extend our models definition with a `Manager` subclass with a simple search implementation:

```
from django.db import models
from towel.managers import SearchManager

class BookManager(SearchManager):
    search_fields = ('title', 'topic', 'authors__name',
        'publisher__name', 'publisher__address')

class Book(models.Model):
    # [...]

    objects = BookManager()
```

`SearchManager` supports queries with multiple clauses; terms may be grouped using apostrophes, plus and minus signs may be optionally prepended to the terms to determine whether the given term should be included or not. Example:

```
+Django "Shop software" -Satchmo
```

Please note that you can search fields from other models too. You should be careful when traversing many-to-many or reverse foreign key relations however, because you will get duplicated results if you do not call `distinct()` on the resulting queryset.

The method `_search()` does the heavy lifting when constructing a queryset. You should not need to override this method. If you want to customize the results further, f.e. apply a site-wide limit for the objects a certain logged in user may see, you should override `search()`.

Next, we have to create a `SearchForm` subclass:

```
from django import forms
from towel import forms as towel_forms
from myapp.models import Author, Book, Publisher

class BookSearchForm(towel_forms.SearchForm):
    publisher = forms.ModelChoiceField(Publisher.objects.all(), required=False)
    authors = forms.ModelMultipleChoiceField(Author.objects.all(), required=False)
    published_on__lte = forms.DateField(required=False)
    published_on__gte = forms.DateField(required=False)

    formfield_callback = towel_forms.towel_formfield_callback
```

You have to add `required=False` to every field if you do not want validation errors on the first visit to the form (which would not make a lot of sense, but isn't actively harmful).

As long as you only use search form fields whose names correspond to the keywords used in Django's `.filter()` calls or `Q()` objects you do not have to do anything else.

The `formfield_callback` simply substitutes a few fields with whitespace-stripping equivalents, and adds CSS classes to `DateInput` and `DateTimeInput` so that they can be easily augmented by javascript code.

Warning: If you want to be able to filter by multiple items, i.e. publishers 1 and 2, you have to define the `publisher` field in the `SearchForm` as `ModelMultipleChoiceField`. Even if the model itself only has a simple ForeignKey Field. Otherwise only the last element of a series is used for filtering.

To activate a search form, all you have to do is add an additional parameter when you instantiate a ModelView subclass:

```
from myapp.forms import BookSearchForm
from myapp.models import Book
from towel.modelview import ModelView

urlpatterns = patterns('',
    url(r'^books/', include(ModelView(Book,
        search_form=BookSearchForm,
        paginate_by=20,
        ).urls)),
)
```

You can now filter the list by providing the search keys as GET parameters:

```
localhost:8000/books/?author=2
localhost:8000/books/?publisher=4&o=authors
localhost:8000/books/?authors=4&authors=5&authors=6
```

Advanced SearchForm features

The `SearchForm` has a `post_init` method, which receives the request and is useful if you have to further modify the queryset i.e. depending on the current user:

```
def post_init(self, request):
    self.access = getattr(request.user, 'access', None)
    self.fields['publisher'].queryset = Publisher.objects.for_user(request.user)
```

The ordering is also defined in the `SearchForm`. You have to specify a dict called `orderings` which has the ordering key as first parameter. The second parameter can be a field name, an iterable of field names or a callable. The ordering keys are what is used in the URL:

```
class AddressSearchForm(SearchForm):
    orderings = {
        '': ('last_name', 'first_name'),  # Default
        'dob': 'dob',  # Sort by date of birth
        'random': lambda queryset: queryset.order_by('?'),
        }
```

1.4.2 Persistent queries

When you pass the parameter `s`, the search is stored in the session for that path. If the user returns to the object list, the filtering is applied again.

The field is included in the SearchForm by default, but don't forget to add it to your template if you are using a custom form render method.

To reset the filters, you have to pass `?clear=1` or `?n`.

1.4.3 Quick Rules

Another option for filtering are *Quick rules*. This allows for field-independent filtering like `is:cool`. Quick rules are mapped to filter attributes using regular expressions. They go into the search form and are parsed automatically (as long as `query_data` is used inside the `queryset` method:

```
class BookSearchForm(towel_forms.SearchForm):
    quick_rules = [
        (re.compile(r'has:publisher'), quick.static(publisher__isnull=False)),
        (re.compile(r'is:published'), quick.static(published_on__lt=timezone.now)),
    ]
```

1.5 Template tags

1.5.1 ModelView detail tags

towel.templatetags.modelview_detail.**model_details**()
Yields a list of (verbose_name, value) tuples for all local model fields:

```
{% load modelview_detail %}

<table>
{% for title, value in object|model_details %}
    <tr>
        <th>{{ title }}</th>
        <td>{{ value }}<td>
    </tr>
{% endfor %}
</table>
```

1.5.2 ModelView list tags

towel.templatetags.modelview_list.**model_row**()
Requires a list of fields which should be shown in columns on a list page. The fields may also be callables. ForeignKey fields are automatically converted into links:

```
{% load modelview_list %}

<table>
{% for object in object_list %}
    <tr>
        {% for title, value in object|model_row:"__unicode__,author %}
            <td>{{ value }}</td>
        {% endfor %}
    </tr>
{% endfor %}
</table>
```

towel.templatetags.modelview_list.**pagination**()
Uses towel/_pagination.html to display a nicely formatted pagination section. An additional parameter may be provided if the pagination should behave differently depending on where it is shown; it is passed to towel/_pagination.html as where:

```
{% load modelview_list %}

{% if paginator %}{% pagination page paginator "top" %}{% endif %}

{# list / table code ... #}

{% if paginator %}{% pagination page paginator "bottom" %}{% endif %}
```

As long as paginate_by is set on the ModelView, a paginator object is always provided. The {% if paginator %} is used because you cannot be sure that pagination is used at all in a generic list template.

This template tag needs the django.core.context_processors.request context processor.

towel.templatetags.modelview_list.**querystring**()
URL-encodes the passed dict in a format suitable for pagination. page and all are excluded by default:

```
{% load modelview_list %}

<a href="?{{ request.GET|querystring }}&page=1">Back to first page</a>

{# equivalent, but longer: #}
<a href="?{{ request.GET|querystring:"page,all" }}&page=1">Back to first page</a>
```

towel.templatetags.modelview_list.**ordering_link**()
Shows a table column header suitable for use as a link to change the ordering of objects in a list:

```
{% ordering_link "" request title=_("Edition") %} {# default order #}
{% ordering_link "customer" request title=_("Customer") %}
{% ordering_link "state" request title=_("State") %}
```

Required arguments are the field and the request. It is very much recommended to add a title too of course.

`ordering_link` has an optional argument, `base_url` which is useful if you need to customize the link part before the question mark. The default behavior is to only add the query string, and nothing else to the `href` attribute.

It is possible to specify a set of CSS classes too. The CSS classes `'asc'` and `'desc'` are added automatically by the code depending upon the ordering which would be selected if the ordering link were clicked (NOT the current ordering):

```
{% ordering_link "state" request title=_("State") classes="btn" %}
```

The `classes` argument defaults to `'ordering'`.

1.5.3 Batch tags

towel.templatetags.towel_batch_tags.**batch_checkbox**()
Returns the checkbox for batch processing:

```
{% load towel_batch_tags %}

{% for object in object_list %}
    {# ... #}
    {% batch_checkbox batch_form object.id %}
    {# ... #}
{% endfor %}
```

1.5.4 Form tags

towel.templatetags.towel_form_tags.**form_items**()
Returns the concatenated result of running `{% form_item field %}` on every form field.

towel.templatetags.towel_form_tags.**form_item**()
Uses `towel/_form_item.html` to render a form field. The default template renders a table row, and includes:

- `help_text` after the form field in a `p.help`
- `invalid` and `required` classes on the row

towel.templatetags.towel_form_tags.**form_item_plain**()
Uses `towel/_form_item_plain.html` to render a form field, f.e. inside a table cell. The default template

puts the form field inside a tag with various classes depending on the state of the form field such as invalid and required.

towel.templatetags.towel_form_tags.**form_errors**()
Shows form and formset errors using towel/_form_errors.html. You can pass a list of forms, formsets, lists containing forms and formsets and dicts containing forms and formsets as values.

Variables which do not exist are silently ignored:

```
{% load towel_form_tags %}

{% form_errors publisher_form books_formset %}
```

towel.templatetags.towel_form_tags.**form_warnings**()
Shows form and formset warnings using towel/_form_warnings.html. You can pass a list of forms, formsets, lists containing forms and formsets and dicts containing forms and formsets as values. Also shows a checkbox which can be used to ignore warnings. This template tag does not work with Django's standard forms because they have do not have support for warnings. Use WarningsForm instead.

Variables which do not exist are silently ignored:

```
{% load towel_form_tags %}

{% form_warnings publisher_form books_formset %}
```

towel.templatetags.towel_form_tags.**dynamic_formset**()
This is a very convenient block tag which can be used to build dynamic formsets, which means formsets where new forms can be added with javascript (jQuery):

```
{% load towel_form_tags %}

<script type="text/javascript" src="PATH_TO_JQUERY.JS"></script>
<script type="text/javascript" src="{{ STATIC_URL }}towel/towel.js"></script>
<style type="text/css">.empty { display: none; }</style>

<form method="post" action=".">{% csrf_token %}
    {% form_errors form formset %}

    <table>
    {% for field in form %}{% form_item field %}{% endfor %}
    </table>

    <h2>Formset</h2>

    <table>
        <thead><tr>
            <th>Field 1</th>
            <th>Field 2</th>
            <th></th>
        </tr></thead>
        <tbody>
        {% dynamic_formset formset "formset-prefix" %}
            <tr id="{{ form_id }}" {% if empty %}class="empty"{% endif %}>
                <td>
                    {{ form.id }}
                    {% form_item_plain form.field1 %}
                </td>
                <td>{% form_item_plain form.field2 %}</td>
```

(continues on next page)

(continued from previous page)

```
                <td>{{ form.DELETE }}</td>
            </tr>
        {% enddynamic_formset %}
        </tbody>
    </table>

    <button type="button" onclick="towel_add_subform('formset-prefix')">
        Add row to formset</button>

    <button type="submit">Save</button>
</form>
```

The formset-prefix must correspond to the prefix used when initializing the FormSet in your Python code. You should pass `extra=0` when creating the FormSet class; any additional forms are better created using `towel_add_subform`.

CHAPTER 2

Autogenerated API Documentation

2.1 API programming

2.2 Deletion

2.3 Forms

class towel.forms.**BatchForm**(*request*, *queryset*, **args*, ***kwargs*)

This form class can be used to provide batch editing functionality in list views, similar to Django's admin actions.

You have to implement your batch processing in the `_context()` method. This method only receives one parameter, a queryset which is already filtered according to the selected items on the list view. Additionally, the current request is available as an attribute of the form instance, `self.request`.

The method `process(self)` may have the following return values:

- A `dict` instance: Will be merged into the template context.
- A `HttpResponse` instance: Will be returned directly to the client.
- An iterable: The handler assumes successful processing of all objects contained in the iterable.
- Nothing: Nothing happens.

Usage example:

```
class AddressBatchForm(BatchForm):
    subject = forms.CharField()
    body = forms.TextField()

    def process(self):
        # Form validation has already been taken care of
        subject = self.cleaned_data.get('subject')
```

(continues on next page)

(continued from previous page)

```
        body = self.cleaned_data.get('body')

        if not (subject and body):
            return {}

        sent = 0
        for item in self.batch_queryset:
            send_mail(subject, body, settings.DEFAULT_SENDER,
                [item.email])
            sent += 1
        if sent:
            messages.success(self.request, 'Sent %s emails.' % sent)

        return self.batch_queryset

def addresses(request):
    queryset = Address.objects.all()
    batch_form = AddressBatchForm(request, queryset)
    ctx = {'addresses': queryset}

    if batch_form.should_process():
        result = form.process()
        if isinstance(result, HttpResponse):
            return result
        elif isinstance(result, dict):
            ctx.update(result)
        elif hasattr(result, '__iter__'):
            messages.success(request,
                _('Processed the following items: %s') % (
                    ', '.join(force_text(item) for item in result)))

        return HttpResponseRedirect('.')

    return render(request, 'addresses.html', ctx)
```

Template code:

```
{% load towel_batch_tags %}
<form method="post" action=".">
    <ul>
    {% for address in addresses %}
        <li>
        {% batch_checkbox address.id batch_form %}
        {{ address }}
        </li>
    {% endfor %}
    </ul>

    {# Required! Otherwise, ``BatchForm.process`` does nothing. #}
    <input type="hidden" name="batchform" value="1" />

    <table>
        {{ batch_form }}
    </table>
    <button type="submit">Send mail to selected</button>
</form>
```

batch_queryset
Returns the queryset containing only items that have been selected for batch processing.

clean()
Cleans the batch form fields and checks whether at least one item had been selected.

process()
Actually processes the batch form submission. Override this with your own behavior.

Batch forms may return the following types here (they are handled by `ModelView.handle_batch_form`:

- A `HttpResponse`: Will be returned directly to the user.
- An iterable: A success message will be generated containing all items in the iterable.

should_process()
Returns true when the submitted form was the batch form, and the batch form is valid.

class towel.forms.**ModelAutocompleteWidget**(*attrs=None*, *url=None*, *queryset=None*)
Model autocompletion widget using jQuery UI Autocomplete

Supports both querysets and JSON-returning AJAX handlers as data sources. Use as follows:

```
class MyForm(forms.ModelForm):
    customer = forms.ModelChoiceField(Customer.objects.all(),
        widget=ModelAutocompleteWidget(url='/customers/search_ajax/'),
        )
    type = forms.ModelChoiceField(Type.objects.all(),
        widget=ModelAutocompleteWidget(queryset=Type.objects.all()),
        )
```

You need to make sure that the jQuery UI files are loaded correctly yourself.

class towel.forms.**MultipleAutocompletionWidget**(*attrs=None*, *queryset=None*)
You should probably use harvest chosen instead.

class towel.forms.**SearchForm**(*data*, **args*, ***kwargs*)
Supports persistence of searches (stores search in the session). Requires not only the GET parameters but the request object itself to work correctly.

Usage example:

```
class AddressManager(SearchManager):
    search_fields = ('first_name', 'last_name', 'address', 'email',
        'city', 'zip_code', 'created_by__email')

class Address(models.Model):
    ...

    objects = AddressManager()

class AddressSearchForm(SearchForm):
    orderings = {
        '': ('last_name', 'first_name'), # Default
        'dob': 'dob', # Sort by date of birth
        'random': lambda queryset: queryset.order_by('?'),
        }
    is_person = forms.NullBooleanField()

def addresses(request):
```

(continues on next page)

(continued from previous page)

```
    search_form = AddressSearchForm(request.GET, request=request)
    queryset = search_form.queryset(Address)
    ctx = {
        'addresses': queryset,
        'search_form': search_form,
        }
    return render(request, 'addresses.html', ctx)
```

Warning: All fields in the form need to have `required=False` set. Otherwise, form validation would already fail on the first visit on the list page (which would kind of defeat the purpose of a search form).

Template code:

```
<form method="get" action=".">
    <input type="hidden" name="s" value="1"> <!-- SearchForm search -->
    <table>
        {{ search_form }}
    </table>
    <button type="submit">Search</button>
</form>

{% for address in addresses %}
    ...
{% endfor %}
```

always_exclude = (u's', u'query', u'o')
Fields which are always excluded from automatic filtering in `apply_filters`

apply_filters(*queryset*, *data*, *exclude=()*)
Automatically apply filters

Uses form field names for `filter()` argument construction.

apply_ordering(*queryset*, *ordering=None*)
Applies ordering if the value in `o` matches a key in `self.orderings`. The ordering may also be reversed, in which case the `o` value should be prefixed with a minus sign.

default = {}
Default field values - used if not overridden by the user

fields_iterator()
Yield all additional search fields.

o = None
Current ordering

orderings = {}
Ordering specification

persist(*request*)
Persist the search in the session, or load saved search if user isn't searching right now.

post_init(*request*)
Hook for customizations.

prepare_data(*data*, *request*)
Fill in default values from `default` if they aren't provided by the user.

query = None
Full text search query

query_data()
Return a fulltext query and structured data which can be converted into simple filter() calls

queryset(*model*)
Return the result of the search

quick_rules = []
Quick rules, a list of (regex, mapper) tuples

s = None
Search form active?

safe_cleaned_data
Safely return a dictionary of values, even if search form isn't valid.

searching()
Returns `searching` for use as CSS class if results are filtered by this search form in any way.

class towel.forms.**StrippedTextInput**(*attrs=None*)
`TextInput` form widget subclass returning stripped contents only

class towel.forms.**StrippedTextarea**(*attrs=None*)
`Textarea` form widget subclass returning stripped contents only

class towel.forms.**WarningsForm**(**args*, ***kwargs*)
Form subclass which allows implementing validation warnings

In contrast to Django's `ValidationError`, these warnings may be ignored by checking a checkbox.

The warnings support consists of the following methods and properties:

- `WarningsForm.add_warning(<warning>)`: Adds a new warning message
- `WarningsForm.warnings`: A list of warnings or an empty list if there are none.
- `WarningsForm.is_valid()`: Overridden `Form.is_valid()` implementation which returns `False` for otherwise valid forms with warnings, if those warnings have not been explicitly ignored (by checking a checkbox or by passing `ignore_warnings=True` to `is_valid()`.
- An additional form field named `ignore_warnings` is available - this field should only be displayed if `WarningsForm.warnings` is non-emtpy.

add_warning(*warning*)
Adds a new warning, should be called while cleaning the data

is_valid(*ignore_warnings=False*)
`is_valid()` override which returns `False` for forms with warnings if these warnings haven't been explicitly ignored

towel.forms.**autocompletion_response**(*queryset*, *limit=10*)
Helper which returns a `HttpResponse` list of instances in a format suitable for consumption by jQuery UI Autocomplete, respectively `towel.forms.ModelAutocompleteWidget`.

towel.forms.**towel_formfield_callback**(*field*, ***kwargs*)
Use this callback as `formfield_callback` if you want to use stripped text inputs and textareas automatically without manually specifying the widgets. Adds a `dateinput` class to date and datetime fields too.

2.4 Managers

class towel.managers.**SearchManager**

Stupid searching manager

Does not use fulltext searching abilities of databases. Constructs a query searching specified fields for a freely definable search string. The individual terms may be grouped by using apostrophes, and can be prefixed with + or - signs to specify different searching modes:

```
+django "shop software" -satchmo
```

Usage example:

```
class MyModelManager(SearchManager):
    search_fields = ('field1', 'name', 'related__field')

class MyModel(models.Model):
    # ...

    objects = MyModelManager()

MyModel.objects.search('yeah -no')
```

search(*query*)

This implementation stupidly forwards to _search, which does the gruntwork.

Put your customizations in here.

towel.managers.**normalize_query**(*query_string*, *findterms=<built-in method findall of _sre.SRE_Pattern object>*, *normspace=<built-in method sub of _sre.SRE_Pattern object>*)

Splits the query string in invidual keywords, getting rid of unecessary spaces and grouping quoted words together.

Example:

```
>>> normalize_query(' some random  words "with   quotes  " and spaces')
['some', 'random', 'words', 'with quotes', 'and', 'spaces']
```

2.5 ModelView

2.6 Multitenancy

2.6.1 Assumptions

- The following settings are required:
  - `TOWEL_MT_CLIENT_MODEL`: The tenant model, e.g. `clients.Client`.
  - `TOWEL_MT_ACCESS_MODEL`: The model linking a Django user with a client, must have the following fields:
    - `user`: Foreign key to `auth.User`.
    - `access`: An integer describing the access level of the given user. Higher numbers mean higher access. You have to define those numbers yourself.

    * The lowercased class name of the client model above as a foreign key to the client model. If your client model is named `Customer`, the name of this foreign key must be `customer`.

- All model managers have a `for_access()` method with a single argument, an instance of the access model, which returns a queryset containing only the objects the current user is allowed to see. The access model should be available as `request.access`, which means that you are free to put anything there which can be understood by the `for_access()` methods. The `request.access` attribute is made available by the `towel.mt.middleware.LazyAccessMiddleware` middleware.
- `towel.mt.modelview.ModelView` automatically fills in a `created_by` foreign key pointing to `auth.User` if it exists.
- The form classes in `towel.mt.forms`, those being `ModelForm`, `Form` and `SearchForm` all require the request (the two former on initialization, the latter on `post_init`). Model choice fields are postprocessed to only contain values from the current tenant. This does not work if you customize the `choices` field at the same time as setting the `queryset`. If you do that you're on your own.
- The model authentication backend `towel.mt.auth.ModelBackend` also allows email addresses as username. It preloads the access and client model and assigns it to `request.user` if possible. This is purely a convenience – you are not required to use the backend.

2.6.2 Forms

These three form subclasses will automatically add limitation by tenant to all form fields with a `queryset` attribute.

Warning: If you customized the dropdown using `choices` you have to limit the choices by the current tenant yourself.

2.6.3 Middleware for a lazy `request.access` attribute

class `towel.mt.middleware.`**LazyAccessMiddleware**

This middleware (or something equivalent providing a `request.access` attribute must be put in `MIDDLEWARE_CLASSES` to use the helpers in `towel.mt`.

2.6.4 Models for multitenant Django projects

The models for `towel.mt` have to be provided by the project where `towel.mt` is used, that's why this file is empty.

The simplest models might look like that:

```
from django.contrib.auth.models import User
from django.db import models


class Client(models.Model):
    name = models.CharField(max_length=100)


class Access(models.Model):
    EMPLOYEE = 10
    MANAGEMENT = 20

    ACCESS_CHOICES = (
```

(continues on next page)

(continued from previous page)

```
        (EMPLOYEE, 'employee'),
        (MANAGEMENT, 'management'),
        )

    client = models.ForeignKey(Client)
    user = models.OneToOneField(User)
    access = models.SmallIntegerField(choices=ACCESS_CHOICES)
```

API methods can be protected as follows:

```
from towel.api import API
from towel.api.decorators import http_basic_auth
from towel.mt.api import Resource, api_access

# Require a valid login and an associated Access model:
api_v1 = API('v1', decorators=[
    csrf_exempt,
    http_basic_auth,
    api_access(Access.EMPLOYEE),
    ])
api_v1.register(SomeModel,
    view_class=Resource,
    )
```

Other views:

```
from towel.mt import AccessDecorator

# Do this once somewhere in your project
access = AccessDecorator()


@access(Access.MANAGEMENT)
def management_only_view(request):
    # ...
```

2.7 Paginator

Drop-in replacement for Django's `django.core.paginator` with additional goodness

Django's paginator class has a `page_range` method returning a list of all available pages. If you got lots and lots of pages this is not very helpful. Towel's page class (**not** paginator class!) sports a `page_range` method too which only returns a few pages at the beginning and at the end of the page range and a few pages around the current page.

All you have to do to use this module is replacing all imports from `django.core.paginator` with `towel.paginator`. All important classes and all exceptions are available inside this module too.

The page range parameters can be customized by adding a `PAGINATION` setting. The defaults are as follows:

```
PAGINATION = {
    'START': 6, # pages at the beginning of the range
    'END': 6, # pages at the end of the range
    'AROUND': 5, # pages around the current page
    }
```

exception towel.paginator.**InvalidPage**

exception towel.paginator.**PageNotAnInteger**

exception towel.paginator.**EmptyPage**

class towel.paginator.**Paginator**(*object_list*, *per_page*, *orphans=0*, *allow_empty_first_page=True*)

Custom paginator returning a Page object with an additional page_range method which can be used to implement Digg-style pagination

page(*number*)

Returns a Page object for the given 1-based page number.

class towel.paginator.**Page**(*page*)

Page object for Digg-style pagination

page_range

Generates a list for displaying Digg-style pagination

The page numbers which are left out are indicated with a `None` value. Please note that Django's paginator own `page_range` method isn't overwritten – Django's `page_range` is a method of the `Paginator` class, not the `Page` class.

Usage:

```
{% for p in page.page_range %}
    {% if p == page.number %}
        {{ p }} <!-- current page -->
    {% else %}
        {% if p is None %}
            &hellip;
        {% else %}
            <a href="?page={{ p }}">{{ p }}</a>
        {% endif %}
    {% endif %}
{% endfor %}
```

2.8 Queryset transform

2.8.1 django_queryset_transform

Allows you to register a transforming map function with a Django QuerySet that will be executed only when the QuerySet itself has been evaluated.

This allows you to build optimisations like "fetch all tags for these 10 rows" while still benefiting from Django's lazy QuerySet evaluation.

For example:

```
def lookup_tags(item_qs):
    item_pks = [item.pk for item in item_qs]
    m2mfield = Item._meta.get_field('tags')[0]
    tags_for_item = Tag.objects.filter(
        item__in = item_pks
    ).extra(select = {
        'item_id': '%s.%s' % (
            m2mfield.m2m_db_table(), m2mfield.m2m_column_name()
        )
```

(continues on next page)

(continued from previous page)

```
    })
    tag_dict = {}
    for tag in tags_for_item:
        tag_dict.setdefault(tag.item_id, []).append(tag)
    for item in item_qs:
        item.fetched_tags = tag_dict.get(item.pk, [])

qs = Item.objects.filter(name__contains = 'e').transform(lookup_tags)

for item in qs:
    print(item, item.fetched_tags)
```

Prints:

```
Winter comes to Ogglesbrook [<sledging>, <snow>, <winter>, <skating>]
Summer now [<skating>, <sunny>]
```

But only executes two SQL queries - one to fetch the items, and one to fetch ALL of the tags for those items.

Since the transformer function can transform an evaluated QuerySet, it doesn't need to make extra database calls at all - it should work for things like looking up additional data from a cache.multi_get() as well.

Originally inspired by http://github.com/lilspikey/django-batch-select/

2.8.2 LICENSE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of Django nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

class towel.queryset_transform.**TransformQuerySet**(**args*, ***kwargs*)

iterator()
An iterator over the results from applying this QuerySet to the database.

2.9 Quick

This module beefs up the default full text search field to be a little bit more versatile. It allows specifying patterns such as `is:unread` or `!important` which are extracted from the query string and returned as standalone values allowing the implementation of a search syntax known from f.e. Google Mail.

Quick rules always consist of two parts: A regular expression pulling values out of the query string and a mapper which maps the values from the regex to something else which may be directly usable by forms.

Usage example:

```
QUICK_RULES = [
    (re.compile(r'!!'), quick.static(important=True)),
    (re.compile(r'@(?P<username>\w+)'),
        quick.model_mapper(User.objects.all(), 'assigned_to')),
    (re.compile(r'\^\+(?P<due>\d+)'),
        lambda v: {'due': date.today() + timedelta(days=int(v['due']))}),
    (re.compile(r'=(?P<estimated_hours>[\d\.]+)h'),
        quick.identity()),
    ]

data, rest = quick.parse_quickadd(
    request.POST.get('quick', ''),
    QUICK_RULES)

data['notes'] = ' '.join(rest)  # Everything which could not be parsed
                                # is added to the ``notes`` field.
form = TicketForm(data)
```

Note: The mappers always get the regex matches `dict` and return a `dict`.

`towel.quick.`**`bool_mapper`**(*attribute*)
Maps `yes`, `1` and `on` to `True` and `no`, `0` and `off` to `False`.

`towel.quick.`**`due_mapper`**(*attribute*)
Understands `Today`, `Tomorrow`, the following five localized week day names or (partial) dates such as `20.12.` and `01.03.2012`.

`towel.quick.`**`identity`**()
Identity mapper. Returns the values from the regular expression directly.

`towel.quick.`**`model_choices_mapper`**(*data*, *attribute*)
Needs a `value` provided by the regular expression and returns the corresponding `key` value.

Example:

```
class Ticket(models.Model):
    VISIBILITY_CHOICES = (
        ('public', _('public')),
        ('private', _('private')),
        )
    visibility = models.CharField(choices=VISIBILITY_CHOICES)

QUICK_RULES = [
    (re.compile(r'~(?P<value>[^\s]+)'), quick.model_choices_mapper(
        Ticket.VISIBILITY_CHOICES, 'visibility')),
    ]
```

towel.quick.**model_mapper**(*queryset*, *attribute*)
The regular expression needs to return a dict which is directly passed to `queryset.get()`. As a speciality, this mapper returns both the primary key of the instance under the `attribute` name, and the instance itself as `attribute_`.

towel.quick.**parse_quickadd**(*quick*, *regexes*)
The main workhorse. Named `parse_quickadd` for historic reasons, can be used not only for adding but for searching etc. too. In fact, `towel.forms.SearchForm` supports quick rules out of the box when they are specified in `quick_rules`.

towel.quick.**static**(***kwargs*)
Return a predefined `dict` when the given regex matches.

2.10 Template tags

2.10.1 `ModelView` template tags

towel.templatetags.modelview_detail.**model_details**(*instance*, *fields=None*)
Returns a stream of `verbose_name`, `value` pairs for the specified model instance:

```
<table>
{% for verbose_name, value in object|model_details %}
    <tr>
        <th>{{ verbose_name }}</th>
        <td>{{ value }}</td>
    </tr>
{% endfor %}
</table>
```

towel.templatetags.modelview_list.**model_row**(*instance*, *fields*)
Shows a row in a modelview object list:

```
{% for object in object_list %}
    <tr>
        {% for verbose_name, field in object|model_row:"name,url" %}
            <td>{{ field }}</td>
        {% endfor %}
    </tr>
{% endfor %}
```

2.10.2 Batch form template tags

towel.templatetags.towel_batch_tags.**batch_checkbox**(*form*, *id*)
Checkbox which allows selecting objects for batch processing:

```
{% for object in object_list %}
    {% batch_checkbox batch_form object.id %}
    {{ object }} etc...
{% endfor %}
```

This tag returns an empty string if `batch_form` does not exist for some reason. This makes it easier to write templates when you don't know if the batch form will be available or not (f.e. because of a permissions requirement).

2.10.3 Generally helpful form tags

towel.templatetags.towel_form_tags.**dynamic_formset**(*parser*, *token*)
Implements formsets where subforms can be added using the `towel_add_subform` javascript method:

```
{% dynamic_formset formset "activities" %}
    ... form code
{% enddynamic_formset %}
```

towel.templatetags.towel_form_tags.**form_errors**(*parser*, *token*)
Show all form and formset errors:

```
{% form_errors form formset1 formset2 %}
```

Silently ignores non-existant variables.

towel.templatetags.towel_form_tags.**form_item**(*item*, *additional_classes=None*)
Helper for easy displaying of form items:

```
{% for field in form %}
    {% form_item field %}
{% endfor %}
```

towel.templatetags.towel_form_tags.**form_item_plain**(*item*, *additional_classes=None*)
Helper for easy displaying of form items without any additional tags (table cells or paragraphs) or labels:

```
{% form_item_plain field %}
```

towel.templatetags.towel_form_tags.**form_items**(*form*)
Render all form items:

```
{% form_items form %}
```

towel.templatetags.towel_form_tags.**form_warnings**(*parser*, *token*)
Show all form and formset warnings:

```
{% form_warnings form formset1 formset2 %}
```

Silently ignores non-existant variables.

2.10.4 Template tags for pulling out the `verbose_name(_plural)?` from almost any object

towel.templatetags.verbose_name_tags.**verbose_name**(*item*)
Pass in anything and it tries hard to return its `verbose_name`:

```
{{ form|verbose_name }}
{{ object|verbose_name }}
{{ formset|verbose_name }}
{{ object_list|verbose_name }}
```

towel.templatetags.verbose_name_tags.**verbose_name_plural**(*item*)
Pass in anything and it tries hard to return its `verbose_name_plural`:

```
{{ form|verbose_name_plural }}
{{ object|verbose_name_plural }}
{{ formset|verbose_name_plural }}
{{ object_list|verbose_name_plural }}
```

2.11 Utils

towel.utils.**app_model_label**(*model*)
Stop those deprecation warnings

towel.utils.**changed_regions**(*regions*, *fields*)
Returns a subset of regions which have to be updated when fields have been edited. To be used together with the `{% regions %}` template tag.

Usage:

```
regions = {}
render(request, 'detail.html', {
    'object': instance,
    'regions': regions,
    })
return HttpResponse(
    json.dumps(changed_regions(regions, ['emails', 'phones'])),
    content_type='application/json')
```

towel.utils.**parse_args_and_kwargs**(*parser*, *bits*)
Parses template tag arguments and keyword arguments

Returns a tuple `args, kwargs`.

Usage:

```
@register.tag
def custom(parser, token):
    return CustomNode(*parse_args_and_kwargs(parser,
        token.split_contents()[1:]))

class CustomNode(template.Node):
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs

    def render(self, context):
        args, kwargs = resolve_args_and_kwargs(context, self.args,
            self.kwargs)
        return self._render(context, *args, **kwargs):

    def _render(self, context, ...):
        # The real workhorse
```

towel.utils.**related_classes**(*instance*)
Return all classes which would be deleted if the passed instance were deleted too by employing the cascade machinery of Django itself. Does **not** return instances, only classes.

Note! When using Django 1.5, autogenerated models (many to many through models) are returned too.

towel.utils.**resolve_args_and_kwargs**(*context*, *args*, *kwargs*)
Resolves arguments and keyword arguments parsed by `parse_args_and_kwargs` using the passed context instance

See `parse_args_and_kwargs` for usage instructions.

towel.utils.**safe_queryset_and**(*head*, **tail*)
Safe AND-ing of querysets. If one of both queries has its DISTINCT flag set, sets distinct on both querysets. Also takes extra care to preserve the result of the following queryset methods:

- `reverse()`
- `transform()`
- `select_related()`
- `prefetch_related()`

towel.utils.**substitute_with**(*to_delete*, *instance*)
Substitute the first argument with the second in all relations, and delete the first argument afterwards.

towel.utils.**tryreverse**(**args*, ***kwargs*)
Calls `django.core.urlresolvers.reverse`, and returns `None` on failure instead of raising an exception.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index