kii Documentation

Release 0.1

Eliot Berriot

January 15, 2015

Contents

Warning: kii is a side project I started for fun. It's still in early developpment and should **not** be considered as production ready. The API may and will probably be subject to many changes before the first official release.

Kii (Keep It Integrated) is a web content management framework designed around the concept of stream. A stream can be seen as a classic social network wall were users can publish any type of content. The projects aims to provide the components needed for building and managing such streams and therefore can be seen as a foundation for more specific apps, like a blog engine, a code snippets collection or a bookmark management system.

CHAPTER 1

End-users will love

- Having a single place for everything
- Clean, responsive, content-focused UI
- Markdown support for content, including syntax highlighting
- Built-in comments
- Fine-grained permissions
- Atom feeds

CHAPTER 2

Developpers will love

- focusing on writing features instead of boilerplate code

CHAPTER 3

Sysadmins will love

- Compatibility with most famous database engines
- Painless installation and updates

CHAPTER 4

Contents

4.1 Philosophy

4.1.1 Stream and items

Kii is written with the notion of stream in mind. It's dead-simple: a *stream* belongs to a user, who can publish *items* in it. An item could be, for example, a blog entry, a picture, a link to a web page, or any type of content provided by third-party apps.

All items share a common behaviour:

- they support comments
- they get a unique permalink, which allow easy sharing over the web
- they support fine-grained permission. You can mark each of them as public, restricted to a set of users or groups or even entirely private if you want to
- they have a status, published or draft

Of course, this is just the common part. Derived content-types may offer additional behaviour, like syntax highlighting for code snippets.

Because of the common behaviour of items, kii offers various filtering options. While a stream index page will display every items, no matter what their type is, it's also possible to display only a specific content type. The same rule applies for feeds: one could subscribe to your entire stream, or only to your snippets.

4.1.2 Freedom

The project is released under the MIT license, which means you are free to download, modify, and distribute it.

4.2 Glossary

instance A dedicated kii installation.

app A python package that extends kii with a given set of features (e.g. a blog app). This term may also refer to kii's built-in apps, or to a regular django app. In fact, a kii app is also a django app.

stream A feed that is owned by a user and contains an unlimited amount of items. If it helps you, you can picture a stream as a facebook wall: a place you own, where you publish different types of things like statuses, pictures or videos.

steam item A record, stored in a stream (it may also be referred to as an item). All stream items share a common behaviour (they have a title, a content area that accepts markdown, a publication date, a permission system...). In kii's (and django's) terminology, this common behaviour is called a model, and the records are model instances. Stream items are not limited to this common behaviour though, and may be extended via the creation of child models.

For exemple, blog entries and code snippets are two different and possible child models of a stream item. A blog entry may have a slug, while a code snippet may have syntax-highlighting and a language attribute, but both need the common behaviour of the stream item model (publication date, permissions and so on).

theme todo

4.3 Installation

Create a new user for your kii instance:

```
sudo adduser kii --disabled-login
sudo su - kii

cd ~
```

Create a virtualenvironment:

```
pip install --user virtualenv
virtualenv ./kii-virtualenv
```

Activate it:

```
source ./kii-virtualenv/bin/activate
```

Then install required packages :

```
pip install kii
```

Note: PyPi package is not available at the moment

Now, you can create a new kii instance using a template:

```
django-admin.py startproject kii_instance --template=https://code.eliotberriot.com/kii/kii-instance-t

cd kii_instance
```

It should create a `kii_instance` directory, with almost everything set up to make put your new kii instance online.

You can now adapt the settings under `kii_instance/settings.py` to suit your needs. For exemple, you will need to set your database credentials, your installed apps, etc. The settings file is heavily commented so I won't detail everything here.

When you're done, you can initialize the database with:

```
python manage.py syncdb

# create an admin account
python manage.py createsuperuser
```

And collect static files:

```
sudo mkdir /var/www/kii/static -p
sudo chown kii:www-data /var/www/kii/static -R
sudo chmod 770 /var/www/kii/static

python manage.py collectstatic
```

4.3.1 Server deployment

Nginx is the recommanded web server for deploying kii, however, you can totally use kii behind Apache2.

The following setup uses Supervisor and Gunicorn for easier maintenance of kii.

First, install dependencies:

```
pip install gunicorn
sudo apt-get install supervisor nginx
```

Then, check gunicorn is correctly working. You will have to make change to this file if you did not followed exactly the install procedure (different username, path, etc.):

```
chmod +x ./gunicorn_start.sh
./gunicorn_start.sh
```

If Gunicorn works, you can now set up Nginx:

```
sudo cp kii_instance/conf/nginx.conf /etc/nginx/sites-enabled/kii

# Especially, in Nginx conf, you will have to set a correct server name.
sudo nano /etc/nginx/sites-enabled/kii

sudo service nginx reload
```

For easier management of your kii instance, you'll probably want to use a process manager. Your kii instance is bundled with a sample supervisor file you can use to start and stop your instance. Also, your instance will be started on boot, which is nice:

```
sudo cp kii_instance/conf/supervisor.conf /etc/supervisor/conf.d/kii.conf

# you can edit it, eventually, but defaults should be fine
sudo nano /etc/supervisor/conf.d/kii.conf

# launch the server
sudo supervisorctl update
sudo supervisorctl start kii
```

4.3.2 Install kii apps

As Kii is not bundled with any app, you have to install these separately. Each app may have custom installation instruction, but most of the process will remain identical:

1. Download the app python packages (via `pip install`)
2. Open `kii_instance/settings.py`
3. Add the app name under `INSTALLED_APPS`
4. (optional) in the same file, add custom settings required by the app, if any
5. Update the database

The Snippets app is a good place to understand the procedure. Step 1:

```
pip install kii_snippets
```

Step 2, 3 and 4:

```
nano kii_instance/settings.py
```

The `INSTALLED_APPS` line should look like this:

```
INSTALLED_APPS += (
    # insert kii apps here
    # ...
    "kii_snippets.apps.App",
)
```

Step 5:

```
python manage.py syncdb
```

4.4 Architecture

Here is a quick overview of kii architecture, so you can quickly understand what are kii built-in components and their role.

Kii functionalities are splitted accross several Python packages, called *apps*. They are indeed regular django apps.

4.4.1 Kii apps references

api

Provides base API views and logic.

Note: This app is not written yet.

app

Provides app-related features, such as base model and view, a registry and a base class for kii apps, autoregistration of kii apps urls, and menu management.

base_models

Provides many model mixins, base templates, views and forms that are used accross all model-related kii apps (stream, particularly).

It's a core part of kii.

classify

Implements stream items tagging.

Note: The foundations are here, but this app is not fully written. User interface, form and filtering are still to be done.

discussion

Implements comments, trackbacks, pingbacks and webmentions for stream items.

Note: Only comment-related logic has been written for now.

glue

A glue app that sticks all kii apps together. Especially, this app provides a default settings file, some base templates, and most of the default theme static files.

hook

This app is in charge of providing hook-logic, so third-party apps can extend or override the default behaviour of kii apps.

permission

Implements all permission-related stuff on models, views and forms.

stream

A big one, because it implements the core design concept of kii: stream and items. This app relies heavily on most of other core apps, and also provides needed forms, views, urls and templates.

tests

This package is used as a repository for testing code and data. It also provides test urls and settings.

theme

This app should implements theme related logic (like template loaders), but the "how" part is not really clear in my head at the moment.

user

User related-stuff, such as login, registration, forgotten-password, etc.

Note: Except for the login part, almost averything needs to be done here.

utils

A collection of utility classes and functions that does not fit in other apps.

4.5 Apps

4.5.1 app

Overview

This app provides app-related features, such as base model and view, a registry and a base class for kii apps, autoregistration of kii apps urls, and menu management.

Core

This module provides two core components for kii:

- **`apps`, an instance of `AppManager`, used especialy for** automatic URL inclusion of kii apps.
- `App`, a class that extends django regular `django.apps.AppConfig` and you should use for building your own kii apps.

class `kii.app.core.`**`App`**(*app_name*, *app_module*)
A base class for all kii apps

`api_urls` = None
See `urls` for more details: a string containing the path to an URLconf that contains API urls (if any). This URLconf will be included under the `kii:api` namespace.

`get_url_prefix`()
Return a prefix for used for URLconf inclusion, such as `r'^myapp/'`. It must be a raw string, starting with ^ and ending with a trailing slash.

By default, uses the `label` as the prefix.

`index`

Returns A URL pointing to the app index

`installed`

Returns a boolean that indicates if the app is marked as installed by django

`kii_app` = True
wether the app should be considered as a kii app or not.

`menu` = None
If you attach a `MenuNode` instance here, the corresponding menu will be automatically built and included in templates.

`App.ready()` is a good place for registering your menu.

`public_models`()
return a list of models from this app that can be created by any user TODO: is this used ?

`urlconf`(*target*)

Parameters **target** (*str*) – The targeted path

Returns The full path to the targeted URLconf, for further inclusion in a URL pattern

`urls` = None
A string containing the path to the app URLconf (if any). This URLconf will then be automatically registered under the `kii` namespace with `AppManager.get_apps_urls()`.

You can use an absolute path, such as `your_app.subpackage.urls` or a relative path, like `.urls`. In the last case, a full path will be built using `name`.

class `kii.app.core.`**`AppManager`**

Provide a cleaner API to `django.apps.apps`

`all``()`

Returns An iterable containing all the registered app configs

`filter``(`***kwargs*`)`

Returns an iterable of installed apps that match given filters

`get``(`*app_label*`)`

Parameters **app_label** (*str*) –

Returns A `django.apps.AppConfig` instance corresponding to the given app_abel.

`get_apps_urls``(`*urlconf='urls'*`)`

Gather all URLs for kii apps so they can easily be included in a URLconf.

Returns A list of django url patterns

`kii_apps``()`

Returns An iterable containing all registered django apps marked that are also kii apps

Menu

class `kii.app.menu.`**`MenuNode`**`(`***kwargs*`)`

Describe a menu element, and may be attached to an `App` instance (via the `menu` attribute) for automatic inclusion in templates.

`__init__``(`***kwargs*`)`

Parameters

- **route** (*str*) – Either a relative URL, absolute URL or a django URL name, such as `kii:myapp:index`. Defaults to `#`.
- **reverse** (*bool*) – Wether the given route should be reversed using django's `reverse()` or returned 'as is'. Defaults to `True`.
- **parent** – TODO, seems useless
- **weight** (*int*) – Indicate the importance of the node. Higher is more important, default to `0`.
- **require_authentication** (*bool*) – Used to determine if the node should be shown to unauthenticated users. Defaults to `True`.
- **reverse_kwargs** (*list*) – A list of strings that the route will accept when reversing. Defaults to `[]`
- **children** (*list*) – A list of children `MenuNode` instances that will be considered as submenus of this instance. Defaults to `[]`.
- **icon** – TODO, seems useless.

`__weakref__`

list of weak references to the object (if defined)

`add``(`*item*`)`

Add a new node to the instance children and sort them by weight.

Parameters **item** – A menu node instance

`url`(***kwargs*)

Parameters **kwargs** – a dictionary of values that will be used for reversing, if the corresponding key is present in `self.reverse_kwargs`

Returns The target URL of the node, after reversing (if needed)

Models

class `kii.app.models.`**`AppModel`**(**args*, ***kwargs*)

A base class for all apps related models. It implements URL reversing for model instances, so one can do:

```
instance = MyModel.objects.get(pk=42)
assert instance.reverse_update() == "/kii/myapp/mymodel/42/update"
```

classmethod **`class_reverse`**(*suffix*)

Same as `reverse` but callable from class instead of instances. :return: a reversed URL for the model

`get_absolute_url`()

Returns The absolute URL of the instance, which is equal to `self.reverse_detail()` by default

`public_model` **= False**

: If True, any authenticated user will be able to create isntances of this model TODO : is it useful ?

`reverse`(*suffix*, ***kwargs*)

Get a model-instance relative URL, such as a detail, delete or update URL. You can override per-suffix URLs by defining `reverse_<suffix>` methods on the model class.

Parameters

- **suffix** (*str*) – a string that will be used to find the corresponding reverse method on the model class (if any)
- **kwargs** (*dict*) – optional URL kwargs that will be passed to the reverse function

Returns a reversed URL

`reverse_delete`(***kwargs*)

Returns The delete URL of the instance

`reverse_detail`(***kwargs*)

Returns The detail URL of the instance

`reverse_update`(***kwargs*)

Returns The update URL of the instance

`url_namespace`(***kwargs*)

Parameters **user_area** (*bool*) – whether the URL namespace should include the username part

Returns a string representing the URL namespace of the model, such as `kii:myapp:mymodel:`

Template tags

`kii.app.templatetags.app_tags.`**`model_url`**`(`*model*, *suffix*, ***kwargs*`)`

Return a model-related URL. Usage:

```
{% load app_tags %}

{% model_url my_object "delete" as url %}
{% if url %}
    {{ url }}
{% endif %}
```

Will ouptut something like `/myapp/mymodel/12/delete` if the URL exists.

Model can be an instance or a `Model` subclass.

This tag will fail silently if the URL is not foun and return an empty string.

`kii.app.templatetags.app_tags.`**`node_url`**`(`*node*, ***kwargs*`)`

Typical usage (look at `kii/glue/templates/default/glue/menu_node.html` for a real example):

```
{% load app_tags %}

{% with node=app.menu %}

    {% node_url node as url %}
    <a href="{{ url }}">My app root menu node</a>

    {% for child_node in node.children %}

        {% node_url child_node as child_url %}
        <a href="{{ child_url }}">A child node</a>

    {% endfor %}
{% endwith %}
```

Parameters

- **node** – a `kii.app.menu.MenuNode` instance
- **kwargs** – kwargs that will be passed to the `reverse()` function

Returns the target URL of the menu node

Views

class `kii.app.views.`**`AppMixin`**

Add extra method and context to all apps views

`dispatch``(`*request*, **args*, ***kwargs*`)`

Set up some hooks before calling the actual dispatch method

`get_breadcrumbs``()`

Returns a list of breadcrumb elements , such as `(('Delete' '/delete'), ('My model', None), ('My app', None))` for use in templates. The first item of each tuple is the title of the element, the second is the URL. URL can be None.

`get_context_data``(`***kwargs*`)`

Add the current app, the page title, the breadcrumbs and the root URL of kii to context

get_page_title()
Override this method if you want to return a custom title for the page. :return: A page title, as a string

page_title = ''
a page title that will be display in templates and `<title>` tags

pre_dispatch(*request*, **args*, ***kwargs*)
Called just after `setup()`. If you return anything but `None`, the view will stop and return the returned value.

This method is convenient for checking a permission, for example.

Returns None

setup(*request*, **args*, ***kwargs*)
Called at the beginning of `dispatch()`, you can extend this method if you need to set some variables for later use

4.5.2 base_models

Overview

Provides many model mixins, base templates, views and forms that are used accross all model-related kii apps (stream, particularly).

It's a core part of kii.

Filtersets

class `kii.base_models.filterset.`**BaseFilterSet**(*data=None*, *queryset=None*, *prefix=None*, *strict=None*)
A base class for more advanced FilterSets.

FilterSets are used to parse querystring from a URL and transpose it into a corresponding queryset. For example, a call on a List view with the URL `/myapp/mymodel/list?status=draft` would be, under the hood, converted into `MyModel.objects.filter(status="draft")`.

Please report to django_filters documentation for in-depth explanation.

Models

Model-related features of kii are splitted accross several mixins, each implementing a single or a couple of fields. This allow us to compose our final models with great flexibility:

```
from kii.base_models import models

class TitleAndStatusModel(models.TitleMixin, models.StatusMixin):
    pass

class ContentAndOwnerModel(models.ContentMixin, models.OwnerMixin):
    pass
```

All these mixins inherit from `BaseMixin`, which provide useful methods and attributes.

class `kii.base_models.models.`**BaseInheritModel**(**args*, ***kwargs*)
Base class for inheriting model (see below)

class `kii.base_models.models.`**`BaseMixin`**`(*args, **kwargs)`

All kii models should inherit from this one, because it provides important features.

classmethod **`class_name`**`()`

Returns The model class name as a string

classmethod **`get_template_names`**`(suffix)`

`base_models.views.ModelTemplateMixin` uses this method to automatically discover model templates.

Considering the following model:

```
# myapp/models.py
from kii.base_models.models import BaseMixin

class MyModel(BaseMixin):
    pass
```

This method will return a list of possible templates for the given suffix, following the inheritance tree:

```
>>> from myapp.models import MyModel
>>> MyModel.get_template_names('detail')
['myapp/mymodel/detail.html', 'base_models/basemixin/detail.html']
>>> MyModel.get_template_names('update')
['myapp/mymodel/update.html', 'base_models/basemixin/update.html']
```

Returns A list of templates name corresponding to the given suffix

(detail, list, etc.).

Will include templates from parent classes, if any

`get_title``()`

Returns a string representing the object, for usage in templates

`meta``()`

Use this method to access model metadata in templates (underscored attributes are forbidden in django templates)

`new`

Shortcut to check if instance is already saved or not

`send``(signal, instance, **kwargs)`

Send a model signal with `self` as the instance.

Other apps can then subscribe to signals in order to execute some arbitrary code at key moments:

```
from kii.base_models.models import StatusMixin
from kii.hook.signals import InstanceSignal

# our signal
instance_published = InstanceSignal()

# our model
class MyModel(StatusMixin):

    def mark_as_published(self, **kwargs):
        self.status = "pub"
        self.save()
        self.send(instance_published)
```

```
# our hooked function
def print_something(**kwargs):
    instance = kwargs.get('instance')
    print('Instance {0} was marked as published'.format(
        instance.pk))

# binding of our function to our signal
instance_published.connect(print_something)
```

If you create some model instances of the previously defined model, you would get this kind of output:

```
>>> from myapp.models import MyModel
>>> instance = MyModel(status="draft")
>>> instance.save()
>>> instance.mark_as_published()
Instance 1 was marked as published
```

Parameters **signal** – a single instance as returned by

`kii.hook.signals.IntanceSignal()` :param instance: TODO: useless, should be deleted

class `kii.base_models.models.`**`ContentMixin`**(**args*, ***kwargs*)
A mixin with a `content` field that accepts different markup (defaults to markdown)

class `kii.base_models.models.`**`OwnerMixin`**(**args*, ***kwargs*)
A mixin for model instance that have an owner

`owned_by`(*user*)

Returns a boolean indicating if the instance owner

is the given user

class `kii.base_models.models.`**`StatusMixin`**(**args*, ***kwargs*)
A mixin with a status and a publication_date field

`publication_date` **= None**
: a datetime field which is set automatically when status is marked as `pub` for the first time

`save`(**args*, ***kwargs*)
Set publication_date to now if status is set to published

`status` **= None**
a choice field that defaults to `pub`

class `kii.base_models.models.`**`TimestampMixin`**(**args*, ***kwargs*)
A mixin with two datetime-fields that are automatically set

`created` **= None**
the creation datetime of the instance, set on the first save

`last_modified` **= None**
the last modification datetime of the instance updated on each save

class `kii.base_models.models.`**`TitleMixin`**(**args*, ***kwargs*)
An abstract base class for models with a title

`kii.base_models.models.`**`get_inherit_model`**(*local_field*, *target*, *target_class*, *target_related_name*, *target_field=None*)
Return a model that will allow field synchronisation between an instance field and a target field The returned model will have boolean field *inherit_<local_field>*.

When this field is set to True on a model instance, the value of local_field will be automatically fetched from target.target_field. Inherit from must be a string pointing to a ForeignKey field on instance, with a field named target_field, from which the value will be inherited

This method will also return a dict of signals you have to register manually in order to enable field synchronization

Warning: field synchronization won't work if you use the returned model in a abstract base class

Template tags

`kii.base_models.templatetags.base_models.`**`list_item_template`**(*item*)
Find the list_item template that should render a given `kii.base_models.models.BaseMixin` instance

Views

class `kii.base_models.views.`**`Create`**(***kwargs*)
A generic view for creating a new instance of a model

class `kii.base_models.views.`**`Delete`**(***kwargs*)
A generic view for deleting an existing instance of a model

class `kii.base_models.views.`**`Detail`**(***kwargs*)
A generic view for detailing an existing instance of a model

class `kii.base_models.views.`**`List`**(***kwargs*)
A generic view for listing many instances of a model

`filterset` **= None**
:the filterset will be built automatically by the view from `filterset_class`, and used for advanced queryset filtering via GET parameters

`get_filterset_kwargs`()

Returns required arguments for building the filterset

`get_queryset`()
Filter the queryset using GET parameters, if needed

class `kii.base_models.views.`**`ModelFormMixin`**
Implements some common modelform view logic

classmethod **`as_view`**(**args*, ***kwargs*)
Deduce model from form class if needed

class `kii.base_models.views.`**`ModelTemplateMixin`**
Implements some convenient logic for:

•deducing the correct template to use, from `model` and

`action` - building the page title

`action` **= None**
: a string representing an action performed by the view, like update, delete, detail, list...

`get_breadcrumbs`()
Prepend action and model label to the page title

`get_template_names`()

Returns an iterable of possible template names, deduced from `action` and `model`

class `kii.base_models.views.`**`OwnerMixinCreate`**`(`*`**kwargs`*`)`
A create view that set the newly created instance `owner` attribute to `request.user`

class `kii.base_models.views.`**`PermissionMixin`**
A mixin that implements permission checks on single object related views.

No check will be performed if `required_permission` is set to `None`

`has_required_permission``(`*`request, *args, **kwargs`*`)`

Returns A boolean indicating if the request user can access

the view

`permission_denied``(`*`request`*`)`

Returns a response for the case where user has not the required

permission

`required_permission` **= None**
: a string indicating what kind of permission is required for displaying the view.

The actuel check will be done via `has_required_permission()`.

class `kii.base_models.views.`**`RequireAuthenticationMixin`**
Force user authentication before calling `dispatch()`

class `kii.base_models.views.`**`RequireOwnerMixin`**
Permission mixin that checks the requested object is owned by `request.user` before granting access

class `kii.base_models.views.`**`SingleObjectPermissionMixin`**
Implements basic pluggable permission logic on single object views

class `kii.base_models.views.`**`Update`**`(`*`**kwargs`*`)`
A generic view for updating an existing instance of a model

4.5.3 classify

Models

class `kii.classify.models.`**`Tag`**`(`*`*args, **kwargs`*`)`
Hierarchical model that can be attached to a `kii.stream.models.StreamItem` instance.

`get_absolute_url``(`*`*moreargs, **morekwargs`*`)`

Returns The absolute URL of the instance, which is equal to `self.reverse_detail()` by default

class `kii.classify.models.`**`TagStreamItem`**`(`*`*args, **kwargs`*`)`
Many to many relationship between StreamItem and Tag

4.5.4 discussion

Overview

Provides discussion-related features on stream items, such as comment management.

Example of implementation

Here is a quick example of discussion integration on your own models.

Models

```
# myapp/models.py

from django.db import models
from kii.discussion.model import CommentMixin, DiscussionMixin


class BlogEntry(DiscussionMixin):
    title = models.CharField(max_length=255)
    content = models.TextField()


class BlogEntryComment(CommentMixin):
    subject = models.ForeignKey(BlogEntry)
```

After a `python manage.py syncdb`, you should be able to run the following in a shell:

```
>>> from myapp import models
>>> from django.contrib.auth import get_user_model
>>> blog_entry = models.BlogEntry(title="Hello world", content="Yolo!", discussion_open=True)
>>> blog_entry.save()
>>> comment_user = get_user_model().objects.get(username="jeanmichel)
>>> comment = models.BlogEntryComment(user=comment_user, subject=blog_entry, content="Nice post dude
>>> comment.save()
>>> assert blog_entry.comments.all().first() == comment
```

Form

```
# myapp/forms.py

from kii.discussion.forms import CommentForm
from . import models


class BlogEntryCommentForm(CommentForm):
    class Meta(CommentForm.Meta):
        model = models.BlogEntryComment
```

View

```
# myapp/views.py

from django.views.generic import DetailView
from kii.discussion import views

from . import models, forms

class BlogEntryDetail(views.CommentFormMixin, DetailView):

    model =  models.BlogEntry
    comment_form_class = forms.BlogEntryCommentForm
```

URLs

```
# myapp/urls.py

from django.conf.urls import patterns, url
from . import views, forms
from kii.discussion.views import CommentCreate


urlpatterns = patterns('',
    url(r'^(?P<pk>\d+)$', views.BlogEntryDetail.as_view(), name='entry_detail'),
    url(r'^(?P<pk>\d+)/comments/add$', CommentCreate.as_view(
        form_class=forms.BlogEntryCommentForm), name='comment_create'),
)
```

Template

```
# myapp/templates/myapp/blog_entry_detail.html

<h1>{{ object.title }}</h1>
{{ object.content }}

{% if object.discussion_open %}
    <ul class="comments">
        {% for comment in object.comments.public %}
            <li>
                <h2>{{ comment.profile.username }}</h2>
                <p>{{ comment.created|timesince }} ago</p>
                {{ comment.content }}
            </li>
        {% endfor %}
    </ul>
    <h1>Publish a comment</h1>
    <form action="{% url 'comment_create' pk=object.pk %}" method="POST">
        {% csrf_token %}
        {{ comment_form.as_p }}
        <input type="submit" value="OK" />
    </form>
{% else %}
    Discussion is closed for this entry.
{% endif %}
```

Models

class `kii.discussion.models.`**`AnonymousCommenterProfile`**`(`**args*, ***kwargs*`)`

Store informations about anonymous user who leave comments.

class `kii.discussion.models.`**`CommentMixin`**`(`**args*, ***kwargs*`)`

A base class for comment models.

Must be linked to either an authenticated user via the `user` attribute or an anonymous user via `user_profile`.

Subclasses MUST implement a `subject` ForeignKey field to the model that should accept comments.

`profile` **= None**

: reference to a `ProfileWrapper` instance. Will be set automatically on init

user
relationship to an authenticated user

user_profile
relationship to an anonymous user

class `kii.discussion.models.`**DiscussionMixin**(**args*, ***kwargs*)
A mixin for models that accept comments

discussion_open = None
whether the model instance is open to discussion or not

class `kii.discussion.models.`**ProfileWrapper**(*instance*)
Utility class to access comment user data the same way for anonymous and authenticated users.

4.6 Credits

Kii relies on different third-party components, which are listed below.

4.6.1 Server-side

Programming language

Kii is written in Python, and should work under both Python 2 and 3. The test suite is run under Python 2.7 and 3.4. While Python's standard library is used in many place, kii requires some additional packages:

- Markdown, for well... Markdown syntax formatting
- Six, for Python 2 and 3 compatibility

Framework

Kii runs on top of Django, the web framework for perfectionnists with deadlines. This choice has been made because, by nature, Django is very extensible, and many great django apps are useful in the scope of the project, such as:

- Polymorphic, used for convenient inheritance of stream items behaviour, at a database level
- Guardian, for per-object permissions
- MPTT, for tags hierarchy
- REST framework, for the REST API
- Filter, for complex filtering of stream items via URL parameters

Testing

Testing is done via Nose and Tox.

4.6.2 Client-side

- Foundation and Foundation icon fonts, for the default theme
- AngularJS, for comments administration interface

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*