django-river Documentation

Release 0.10.0

Ahmet DAL

Jan 05, 2019

Contents

River is an open source and always free workflow framework for `Django` which support on the fly changes instead of hardcoding states, transitions and authorization rules.

The main goal of developing this framework is **to be able to edit any workflow item on the fly.** This means that all the elements in a workflow like states, transitions or authorizations rules are editable at any time so that no changes requires a re-deploying of your application anymore.

CHAPTER 1

Getting Started

You can easily get started with `django-river` by following *Getting Started*.

CHAPTER 2

Contents

2.1 Getting Started

1. Install and enable it

```
pip install django-river
```

```
INSTALLED_APPS=[
...
river
...
]
```

3. Create your first state machine in your model

```
from django.db import models
from river.models.fields.state import StateField

class MyModel(models.Model):
    my_state_field = StateField()
```

3. Create your states as one of them will be your initial state on the admin page (Look at *State Administration*.)
4. Create your transition approval metadata with your model (`MyModel` - `my_state_field`) information and authorization rules along with their priority on the admin page (Look at *Transition Approval Meta Administration*.)
5. Enjoy your `django-river` journey.

```
my_model=MyModel.objects.get(....)

my_model.river.my_state_field.approve(as_user=transactioner_user)
my_model.river.my_state_field.approve(as_user=transactioner_user,next_
↪state=State.objects.get(label='re-opened'))
```

(continues on next page)

(continued from previous page)

```
# and much more. Check the documentation
```

Note: Whenever a model object is saved, it's state field will be initialized with the state is given at step-3 above by `django-river`.

Note: Make sure that there is only one initial state defined in your workflow, so that `django-river` can pick that one automatically when a model object is created. All other workflow items will be managed by `django-river` after object creations.

2.2 Overview

Main goal of developing this framework is **to be able to edit any workflow item on the fly.** This means, all elements in workflow like states, transitions, user authorizations(permission), group authorization are editable. To do this, all data about the workflow item is persisted into DB. **Hence, they can be changed without touching the code and re-deploying your application.**

There is ordering aprovments for a transition functionality in `django-river`. It also provides skipping specific transition of a specific objects.

Playground: There is a fake jira example repository as a playground of django-river. https://github.com/javrasya/fakejira

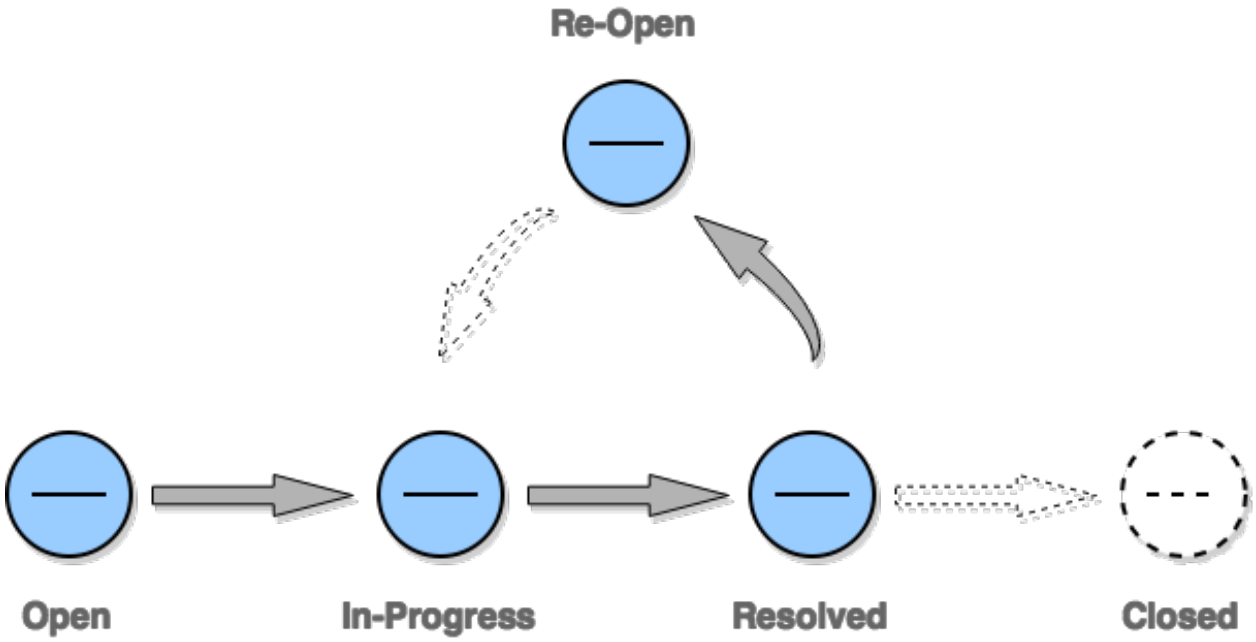
2.2.1 Requirements

- Python (`2.7`, `3.4`, `3.5`, `3.6`)
- Django (`1.7`, `1.8`, `1.9`, `1.10`, `1.11`, `2.0`)
- Django 2.0 is supported with `Python3.5` and `Python3.6`
- Django 1.7 is not for `Python3.5`

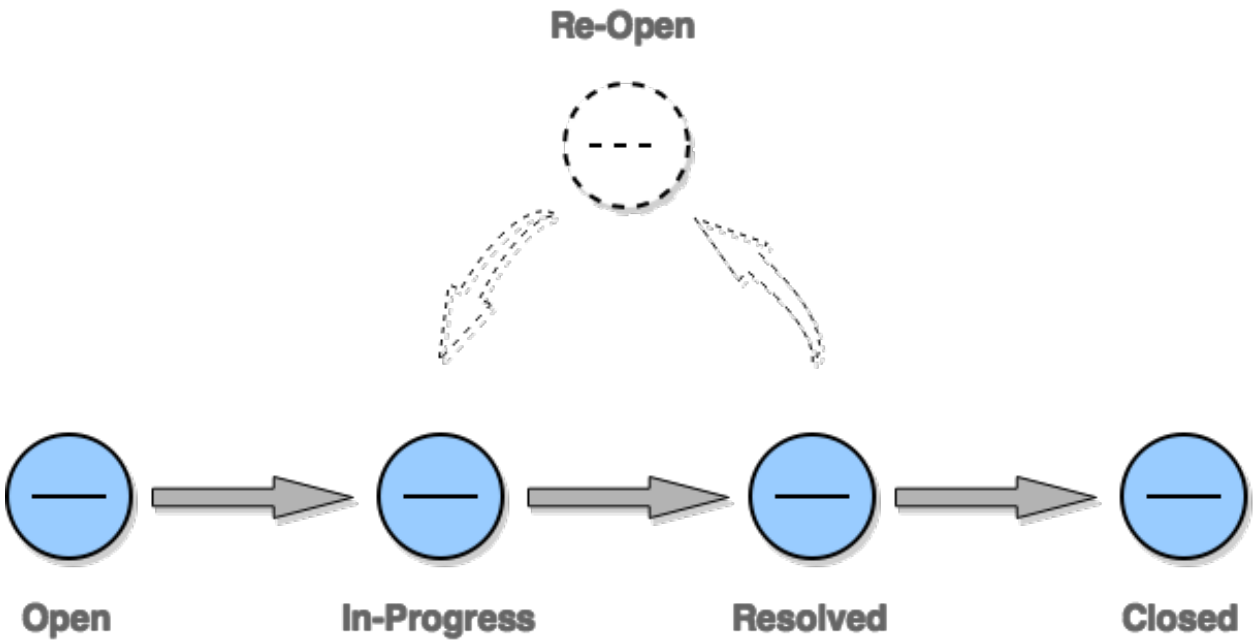
2.2.2 Example Scenarios

Simple Issue Tracking System

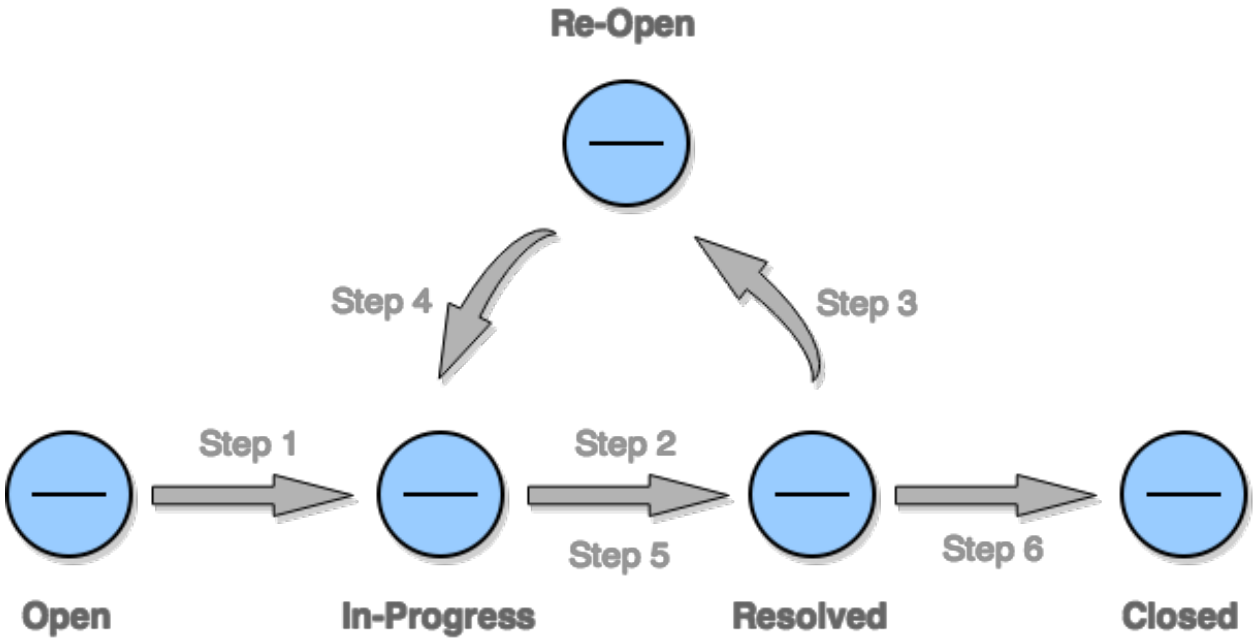
Re-Open case



Closed without Re-Open case



Closed with Re-Open case



2.3 Administration

Since `django-river` keeps all the data needed in a database, those should be pre-created before your first model object is created. Otherwise **your app will crash first time you create a model object.** Here are all needed models that you need to provide. `django-river` will register an Administration for those model for you. All you need to do is to provide them by using their Django admin pages.

2.3.1 State Administration

Field	Default	Optional	Format	Description
label	NaN	False	String (w+)	A name for the state
description	Empty string	True	String (w+)	A description for the state

2.3.2 Transition Approval Meta Administration

Field	Default	Optional	Format	Description
workflow		False False	Choice of Strings	Your model class along with the field that you want to use this transition approval meta for. `django-river` will list all the possible model and fields you can pick on the admin page
source_state		False	State	Source state of the transition
destination_state		False	State	Destination state of the transition
permissions	Empty List	True	List<Permission>	List of permissions which will be authorized to approve this transition
groups	Empty List	True	List<UserGroup>	List of use groups which will be authorized to approve this transition
priority	0	False	Nummber	The priority of the transition approval. Since there can be more than one transition approval to make that transition which means that some users should approve before some other users can approve the same transition. The closer to zero, the more priort the transition

2.4 API Guide

2.4.1 Class API

This page will be covering the class level API. It is all the function that you can access through your model class like in the example below;

```
>>> MyModel.river.my_state_field.<function>(*args)
```

get_on_approval_objects

This is the function that helps you to fetch all model objects waitig for a users approval.

```
>>> my_model_objects = MyModel.river.my_state_field.get_on_approval_objects(as_
↪user=team_leader)
```

	Type	Default	Optional	Format	Description
as_user	input	NaN	False	Django User	A user to find all the model objects waiting for a user's approvals
	Output			List<MyModel>	List of available my model objects

initial_state

This is a property that is the initial state in the workflow

```
>>> State.objects.get(label="open") = MyModel.river.my_state_field.initial_state
```

Type	Format	Description
Output	State	The initial state in the workflow

final_states

This is a property that is the list of final state in the workflow

```
>>> State.objects.filter(Q(label="closed") | Q(label="cancelled")) = MyModel.river.my_
↪state_field.final_states
```

Type	Format	Description
Output	List<State>	List of the final states in the workflow

2.4.2 Instance API

This page will be covering the instance level API. It is all the function that you can access through your model object like in the example below;

```
my_model=MyModel.objects.get(....)

my_model.river.my_state_field.<function>(*args)
```

approve

This is the function that helps you to approve next approval of the object easily. `django-river` will handle all the availibility and the authorization issues.

```
>>> my_model.river.my_state_field.approve(as_user=team_leader)
>>> my_model.river.my_state_field.approve(as_user=team_leader, next_state=State.
↪objects.get(name='re_opened_state'))
```

	Type	Default	Optional	Format	Description
as_user	input	NaN	False	Django User	A user to make the transaction. `django-river` will check if this user is authorized to make next action by looking at this user's permissions and user groups.
next_state	input	NaN	True/False	State	This parameter is redundant as long as there is only one next state from the current state. But if there is multiple possible next state in place, `django-river` is naturally is unable know which one is actually supposed to be picked. If the given next state is not a valid next state a *RiverException* will be thrown.
god_mod	input	False	True	Boolean	Authorization will be skipped if this is *True*

get_available_approvals

This is the function that helps you to fetch all available approvals waiting for a spesific user according to given source and destination states. If the source state is not provided, `django-river` will pick the current objects source state.

```
>>> transition_approvals = my_model.river.my_state_field.get_available_approvals(as_
↪user=manager)
>>> transition_approvals = my_model.river.my_state_field.get_available_approvals(as_
↪user=manager, source_state=State.objects.get(name='in_progress'))
>>> transition_approvals = my_model.river.my_state_field.get_available_approvals(
        as_user=manager,
        source_state=State.objects.get(name='in_progress'),
        destination_state=State.objects.get(name='resolved'),
    )
```

	Type	Default	Optional	Format	Description
as_user	input	NaN	False	Django User	A user to find all the approvals by user's permissions and groups
source_state	input	Current Object's Source State	True	State	A base state to find all available approvals comes after. Default is current object's source state
destination_state	input	NaN	True	State	A spesific destination state to fetch all available state. If it is not provided, the approvals will be found for all available destination states
god_mod	input	False	True	Boolean	Authorization will be skipped if this is *True*
	Output			List<TransitionApproval>	List of available transition approvals

recent_approval

This is a property that the transition approval which has recently been approved for the model object.

```
>>> transition_approval = my_model.river.my_state_field.last_approval
```

Type	Format	Description
Output	TransitionApproval	Last approved transition approval for the model object

next_approvals

This is a property that the list of transition approvals as a next step.

```
>>> transition_approvals = my_model.river.my_state_field.next_approvals
```

Type	Format	Description
Output	List<TransitionApproval>	List of transition approvals comes after last approved transition approval

on_initial_state

This is a property that indicates if object is on initial state.

```
>>> True == my_model.river.my_state_field.on_initial_state
```

Type	Format	Description
Output	Boolean	True if object is on initial state

on_final_state

This is a property that indicates if object is on final state.

```
>>> True == my_model.river.my_state_field.on_final_state
```

Type	Format	Description
Output	Boolean	True if object is on final state which also means that the workflow is complete

hook_pre_transition

This is a function that helps you to hook pre-transtion. For more detail please look at *Transition Callback Function*.

```
def my_callback(workflow_obj, field_name, transition_approval=None, source_
↪state=None, destination_state=None):
    pass
```

```
>>> my_model.river.my_state_field.hook_pre_transition(my_callback)
>>> my_model.river.my_state_field.hook_pre_transition(my_callback, source_
↪state=in_progress_state, destination_state=resolved_state)
```

	Type	Default	Optional	Format	Description
callback	input	NaN	False	Callable	A callback function for `django-river` to call when the given transition happens
source_state	input	NaN	True	State	Spesific source state for the hook
destination_state	input	NaN	True	State	Spesific destination state for the hook

hook_post_transition

This is a function that helps you to hook post-transtion. For more detail please look at *Transition Callback Function*.

```
def my_callback(workflow_obj, field_name, transition_approval=None):
    pass
```

```
>>> my_model.river.my_state_field.hook_post_transition(my_callback)
>>> my_model.river.my_state_field.hook_post_transition(my_callback, source_
↪state=in_progress_state, destination_state=resolved_state)
```

	Type	Default	Optional	Format	Description
callback	input	NaN	False	Callable	A callback function for `django-river` to call when the given transition happens
source_state	input	NaN	True	State	Spesific source state for the hook
destination_state	input	NaN	True	State	Spesific destination state for the hook

hook_pre_complete

This is a function that helps you to hook pre-complete. For more detail please look at *On Complete Callback Function*.

```
def my_callback(workflow_obj, field_name):
    pass
```

```
>>> my_model.river.my_state_field.hook_pre_complete(my_callback)
```

	Type	Default	Optional	Format	Description
callback	input	NaN	False	Callable	A callback function for `django-river` to call when the given transition happens

hook_post_complete

This is a function that helps you to hook post-complete. For more detail please look at *On Complete Callback Function*.

```
def my_callback(workflow_obj, field_name):
    pass
```

```
>>> my_model.river.my_state_field.hook_post_complete(my_callback)
```

	Type	Default	Optional	Format	Description
callback	input	NaN	False	Callable	A callback function for `django-river` to call when the given transition happens

2.5 Hooking Guide

Hooking is one of the powerful side of `django-river`. It is basically allowing you to have some callback functions due to some circumtances like transitions or workflow completions.

2.5.1 How Should Callback Functions Look Like

You can register your callback function for some circumtances via `django-river` hooking feature. It can either be when a spesific transition happend or when a workflow is complete for an object. Your callback functions should look like how `django-river` wants them to be.

If you want to see how you can register your callback, you can take a look at either *hook_pre_transition* or *hook_post_transition*.

Transition Callback Function

```
def my_callback_function(workflow_object, field_name, transition_
↪approval=None):
    print(f"A transition happened: {transition_approval.source_state} ->
↪{transition_approval.destination_state} by user {transition_approval.
↪transactioner}")
```

Paramters	Type	Format	Description
workflow_object	args	YourModel	The instance of your workflow model that has just transited
field_name	args	String	State field name of that model object in the workflow that the transition happened in
transition_approval	kwargs	TransitionApproval	The transition approval has just been approved right before this transition happened. You can access the transactioner user, source and destination states and many more within transition approval. For more detail look at TransitionApproval model source

On Complete Callback Function

```
def my_callback_function(workflow_object, field_name):
    print(f"The workflow is completed for workflow object {workflow_object}␣
↪and field {field_name}")
```

Paramters	Type	Format	Description
workflow_object	args	YourModel	The instance of your workflow model whose workflow has just been completed
field_name	args	String	State field name of that model object in the workflow that is completed

2.5.2 Hooking Backends

Hooking needs to have a mechanism to manage all the registered callbacks even in multi-process situations properly. This is a part where you can use different types of hooking backends either the built-in ones or the ones that you may want to come up with on your own.

DatabaseHookingBackend

This is the default hooking backend that keeps all the registered in the persistent databsase. Django applicaitons are usually running as multi-process on production and consequently `django-river` will call your callback functions multiple times in different processes with `MemoryHookingBackend`. You may want to have your callback functions being called only at once and this is what `DatabaseHookingBackend` is for.

```
.
RIVER_HOOKING_BACKEND = {
     'backend':'river.hooking.backends.memory.DatabaseHookingBackend',
     'config' : {}
}
.
```

MemoryHookingBackend

This is the hooking backend that keeps all the registered callbacks in memory and it is for development and test purposes. Since it is in memory and not shared, it is not good for multi-process situation. When your object is saved in one instance where you register your callback , but finalized in the workflow in another, your callback completion callback function will never be invoked for instance.

2.6 Change Logs

2.6.1 1.0.0 (Development)

`django-river` is finally having it's first major version bump. In this version, all code and the APIs are revisited and are much easier to understand how it works and much easier to use it now. In some places even more performant. There are also more documentation with this version. Stay tuned :-)

- **Improvement** - Support `Django2.1`
- **Improvement** - Support multiple state fields in a model
- **Improvement** - Make the API very easy and useful by accessing everyhing via model objects and model classes
- **Improvement** - Simplify the concepts
- **Improvement** - Migrate ProceedingMeta and Transition into TransitionApprovalMeta for simplification
- **Improvement** - Rename Proceeding as TransitionApproval
- **Improvement** - Document transition and on-complete hooks
- **Improvement** - Document transition and on-complete hooks
- **Improvement** - Imrove documents in general
- **Improvement** - Minor improvements on admin pages
- **Improvement** - Some performance improvements

2.6.2 0.10.0 (Stable)

- # 39 - **Improvement** - Django has dropped support for pypy-3. So, It should be dropped from django itself too.
- **Remove** - `pypy` support has been dropped
- **Remove** - `Python3.3` support has been dropped
- **Improvement** - `Django2.0` support with `Python3.5` and `Python3.6` is provided

2.6.3 0.9.0

- # 30 - **Bug** - Missing migration file which is `0007` because of `Python2.7` can not detect it.
- # 31 - **Improvement** - unicode issue for Python3.
- # 33 - **Bug** - Automatically injecting workflow manager was causing the models not have default `objects` one. So, automatic injection support has been dropped. If anyone want to use it, it can be used explicitly.
- # 35 - **Bug** - This is huge change in django-river. Multiple state field each model support is dropped completely and so many APIs have been changed. Check documentations and apply changes.

2.6.4 0.8.2

- **Bug** - Features providing multiple state field in a model was causing a problem. When there are multiple state field, injected attributes in model class are owerriten. This feature is also unpractical. So, it is dropped to fix the bug.
- **Improvement** - Initial video tutorial which is Simple jira example is added into the documentations. Also repository link of fakejira project which is created in the video tutorial is added into the docs.
- **Improvement** - No proceeding meta parent input is required by user. It is set automatically by django-river now. The field is removed from ProceedingMeta admin interface too.

2.6.5 0.8.1

- **Bug** - ProceedingMeta form was causing a problem on migrations. Accessing content type before migrations was the problem. This is fixed by defining choices in init function instead of in field

2.6.6 0.8.0

- **Deprecation** - ProceedingTrack is removed. ProceedingTracks were being used to keep any transaction track to handle even circular one. This was a workaround. So, it can be handled with Proceeding now by cloning them if there is circle. ProceedingTracks was just causing confusion. To fix this, ProceedingTrack model and its functions are removed from django-river.
- **Improvement** - Circular scenario test is added.
- **Improvement** - Admins of the workflow components such as State, Transition and ProceedingMeta are registered automatically now. Issue #14 is fixed.

2.6.7 0.7.0

- **Improvement** - Python version 3.5 support is added. (not for Django1.7)
- **Improvement** - Django version 1.9 support is added. (not for Python3.3 and PyPy3)

2.6.8 0.6.2

- **Bug** - Migration `0002` and `0003` were not working properly for postgresql (maybe oracle). For these databases, data can not be fixed. Because, django migrates each in a transactional block and schema migration and data migration can not be done in a transactional block. To fix this, data fixing and schema fixing are seperated.
- **Improvement** - Timeline section is added into documentation.
- **Improvement** - State slug field is set as slug version of its label if it is not given on saving.

2.6.9 0.6.1

- **Bug** - After `content_type` and `field` are moved into `ProceedingMeta` model from `Transition` model in version `0.6.0`, finding initial and final states was failing. This is fixed.
- **Bug** - `0002` migrations was trying to set default slug field of State model. There was a unique problem. It is fixed. `0002` can be migrated now.
- **Improvement** - The way of finding initial and final states is changed. ProceedingMeta now has parent-child tree structure to present state machine. This tree structure is used to define the way. This requires to migrate `0003`. This migration will build the tree of your existed ProceedingMeta data.

2.6.10 0.6.0

- **Improvement** - `content_type` and `field` are moved into `ProceedingMeta` model from `Transition` model. This requires to migrate `0002`. This migrations will move value of the fields from `Transition` to `ProceedingMeta`.
- **Improvement** - Slug field is added in `State`. It is unique field to describe state. This requires to migrate `0002`. This migration will set the field as slug version of `label` field value. (Re Opened -> re-opened)

- **Improvement** - `State` model now has `natural_key` as `slug` field.
- **Improvement** - `Transition` model now has `natural_key` as (`source_state_slug` , `destination_state_slug`) fields
- **Improvement** - `ProceedingMeta` model now has `natural_key` as (`content_type`, `field`, `transition`, `order`) fields
- **Improvement** - Changelog is added into documentation.

2.6.11 0.5.3

- **Bug** - Authorization was not working properly when the user has irrelevant permissions and groups. This is fixed.
- **Improvement** - User permissions are now retreived from registered authentication backends instead of `user.user_permissions`

2.6.12 0.5.2

- **Improvement** - Removed unnecessary models.
- **Improvement** - Migrations are added
- **Bug** - `content_type__0002` migrations cause failing for `django1.7`. Dependency is removed
- **Bug** - `DatabaseHandlerBacked` was trying to access database on django setup. This cause `no table in db` error for some django commands. This was happening; because there is no db created before some commands are executed; like `makemigrations`, `migrate`.

2.6.13 0.5.1

- **Improvement** - Example scenario diagrams are added into documentation.
- **Bug** - Migrations was failing because of injected `ProceedingTrack` relation. Relation is not injected anymore. But property `proceeing_track` remains. It still returns current one.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search