dynamo Documentation

Release 0.25

Juergen Schackmann

June 16, 2016

Contents

Dynamo let users and admins create and mantain their Django **DYNA** mic **MO** dels dynamically at runtime.

CHAPTER 1

Contents

1.1 Overview

1.1.1 Why would you need a dynamic model?

Dynamic models are beneficial for applications that need data structures, which are only known at runtime, but not when the application is coded. Or when existing models need to be extended at runtime by additional fields. Typical use cases are:

1. *CMS:* In content management systems, users often need to maintain content that is unique for their specific website. The required data structures to store and maintain this content is therefore not known to the developers beforehand.
2. *Web Shop:* The owner of a web shop has highly customized products, with very special product attributes. The shop developers want the web shop owner to define these attributes herself.
3. *Survey:* If you have an application to create and maintain online surveys, you do neither know the questions nor the possible answers at runtime, but let your users define these, as they implement their surveys.

Dynamo supports the three of these use cases - and many more!

1.1.2 How does Dynamo work?

Dynamo lets you define the meta data for your models their fields. This metadata definition is stored in "real" Django models. The defined model is then created at runtime. And of course, you can also modify the models later on, e.g. adding, renaming or deleting fields; or changing model attributes. It will also automatically manage your admin and app cache for the dynamic models. The meta data maintenance can be done via the Django Admin or via the provided API.

1.1.3 What else is there?

There are various approaches and implemenations available for Django developers:

- The most straight forward approach is to use the Django internals and its DB API to create and maintain models at runtime. Numerous authors have elaborated this option in the Django Wiki . Michael Hall has created an app following this approach; he has also called in dynamo, I hope this does not cause too much confusion.
- Entity-Attribute-Value / **EAV** Model is the traditioal computer science approach to tackle this kind of problem, and there are also django implementations for that available like django-eav or eav-django.

- Finally, Will Hardy has introduced a South -based concept, that he has presented and discussed at the DjangoCon Europe 2011 . Following this concept, he has implemented dynamic-models

The South based approach seems to be the cleanest and clearly follows the DRY approach: all database handling, maintenance and transactions are left to the excellent South API.

1.1.4 Who else gets credits for Dynamo?

Dynamo is inspired by the excellent work of Will Hardy's dynamic-models and this Django Wiki Article. It also re-uses parts of their concepts and coding. Furthermore, South is used to maintain the Dyanmo related database objects.

1.1.5 Under which license is Dynamo available?

Dynamo is available under the BSD license.

1.1.6 How do I get suport?

If you have any questions, issues or would like to contribute, please let us know at the Dynamo Google Group

1.2 Installation and Setup

1.2.1 Dependencies

- south

Developers might also need Sphinx to maintain and update the docs.

1.2.2 Installation

Install into your python path using pip or easy_install:

```
pip install django-dynamo
```

If you are feeling adventurous you can get the lates code from Bitbucket

1.2.3 Configuration

Now, you just need to add 'dynamo' to your installed apps:

```
INSTALLED_APPS=(
 # other apps
        "dynamo",
)
```

And you are all set and ready to go!

1.2.4 Sample Project

Dynamo also comes with a built-in working sample project, that you can easily use to play around. To setup this project, download the source from Bitbucket into your target directory and run:

```
bootstrap.py
```

This will create a virtualenv, install all dependencies, and create a customized manage script in your root (**Attention**: this manage script is just a helper that calls the "normal" manage.py command). So to start the project, just do:

```
manage syncdb
```

to create and sync the database. And then run:

```
manage runserver
```

to start the server. Now you can open the browser, log into the admin and create your MetaModels and MetaFields.

1.3 The Basics

To create a dynamic model at runtime, the meta information needs to be defined. This meta information is stored in two models: MetaModel and MetaField. Both of these models are available in the Dynamo Admin section. As soon as any instance of these is saved, numerous background activities are triggered to ensure that a Django model and its underlying database table(s) exists.

1.3.1 Creating a Model

The following fields are available in the MetaModel model:

- *name*: the name of your dynamic model; this field is required.
- *description*: an optional description of your dynamic model (this is only for information, but not used for model creation)
- *app*: the application /application label that is used for the creation of the model; default: as specified in settings.
- *admin*: a flag to indicate whether the admin page should be created and maintained as well for the specified model; default as specified in settings.

1.3.2 Creating a Field

For each model, you can define as many fields as you like to in the MetaFields model. The following fields are available in the MetaField model, all of which are described in more detail in the Django Docs

- name
- verbose_name
- meta_model
- type
- related_model
- description
- order

- unique_together
- unique
- help
- choices
- default
- required

Please be aware of the following.

- *type*: type describes the field type as defined in the 'Django Docs <https://docs.djangoproject.com/en/1.3/ref/models/fields/#field-types>'_
- *related_model*: this field can and should only be populated, when type is a relationship field
- *unique_together*: is a flag that represents the Meta Unique_Together Option
- *required*: is a flag that indicates, whether this field can be left empty; reflects the blank and null option of the Django fields.

1.3.3 Background Processes

With each "Save" or "Delete" of a MetaModel instance or MetaField instance, the following chain of activities is performed with regard to the underlying dynamic model:

1. Check if relevant attributes have changed that would change the Dynamic Model (via a hash value). If no change is detected, the process stops here.
2. (Re-)generate Dynamic Model based on the new information.
3. Update the database based to the latest changes of the Dynamic Model (e.g. add another table column).
4. Update Model Cache with the regenerated Dynamic Model.
5. Update Admin page ofr the regenerated Dynamic Model.
6. Trigger Dynamo Signals .

1.4 Settings

The following settings are available to customize the Dynamo behavior.

- **DYNAMO_DELETE_COLUMNS**: Flag to define, whether database columns (including content) are deleted after the field has been deleted in the MetaField definition; default: True.
- **DYNAMO_DELETE_TABLES**: Flag to define, whether database tables (including content) are deleted after the model has been deleted in the MetaModel definition; default: True.
- **DYNAMO_DEFAULT_APP**: Default app to be used when model is generated; default: dynamo.
- **DYNAMO_DEFAULT_MODULE**: Default module to be used when model is generated; default: dynamo.models.
- **DYNAMO_FIELD_TYPES**: list of availabe Dynamo field types; default: all Django Field Types (as of Django 1.3). If you want to make any customized fields available, you would need to add them here!
- **STANDARD_FIELD_TYPES**: list of Dynamo standard field types; default: all Django Field Types (as of Django 1.3) except for relationship ones.

- **INTEGER_FIELD_TYPES**: list of integer field types, that control how the choices tuple is generated; default: all Django integer field types.
- **STRING_FIELD_TYPES**: list of string field types, that define how the "require" field controls the blank and null option; default:all Django string field types.
- **RELATION_FIELD_TYPES**: list of field types that require an entry in related_model; default: all Django relation field types.

1.5 Signals

Dynamo provides numerous Django signals to let developers hook into the process.

- **pre_model_creation**: triggers before a model is created; providing_args=['new_model']
- **post_model_creation**: triggers after a model is created: providing_args=['new_model']
- **pre_model_update**: triggers before a model is updated; providing_args= ['old_model','new_model']
- **post_model_update**: triggers after a model has been updated; providing_args=['old_model','new_model']
- **pre_model_delete**: triggers before a model is deleted; providing_args=['old_model']
- **post_model_delete**: triggers after a model has been deleted; providing_args=['old_model']
- **pre_field_creation** triggers before a field is created; providing_args=['new_field']
- **post_field_creation**: triggers after a field has been created; providing_args=['new_field']
- **pre_field_update**: triggers before a field is updated; providing_args=['old_field','new_field']
- **post_field_update**: triggers after a field has been updated; providing_args=['old_field','new_field']
- **pre_field_delete**: triggers before a field is deleted; providing_args=['old_field']
- **post_field_delete**: triggers after a field has been deleted; providing_args=['old_field']

1.6 Signal Handlers

Dynamo has attached numerous handlers to the Django model signals. If you are sure that you know what you are doing, you can disconnect these and connect your own.

1. **when_classes_prepared**: Runs the given function as soon as the model dependencies are available. This is used to build dyanmic models classes on startup, that are already existent in the database.
2. **field_pre_save: A signal handler to run any MetaField pre save activities and trigger the built-in pre_save signals**

    (a) Detect renamed fields and store the old field name for migration

    (b) Detect if the field is just created and store this information.

    (c) Detect if the field is just updated and store this information

    (d) Trigger the pre creation signal

    (e) Trigger the pre update signal
3. **field_post_save: A signal handler to run any MetaField post save activities and trigger the built-in post_save signals:**

    (a) Detect renamed fields and run migration

   (b) Create new fields, if necessary
   (c) Trigger post creation signal
   (d) Trigger post update signal
4. **field_pre_delete: A signal handler to run any MetaField pre delete activities and trigger the built-in pre delete signals:**
   (a) Trigger the pre delete signal
5. **field_post_delete: A signal handler to run any MetaField pre save activities and trigger the built-in pre_save signals:**
   (a) Update the database with regard to the deleted field
   (b) Trigger the post delete signal
6. **model_pre_save: A signal handler to run any MetaModel pre save activities and trigger the built-in pre save signals:**
   (a) Detect if the model is just created and store this information.
   (b) Detect if the model is just updated and store this information
   (c) Trigger the pre creation signal
   (d) Trigger the pre update signal
7. **model_post_save: A signal handler to run any MetaModel pre save activities and trigger the built-in pre save signals:**
   (a) Detect if the model has been changed and appy these changes to the db
   (b) Trigger the pre creation signal
   (c) Trigger the pre update signal
8. **model_pre_delete: A signal handler to run any MetaModel pre delete activities and trigger the built-in pre delete signals**
   (a) Delete the table in db, if settings require us to do so
   (b) Trigger the pre delete signal
9. **model_post_delete: A signal handler to run any MetaModel post delete activities and trigger the built-in pre delete signals**
   (a) Trigger the post delete signal

1.7 API Helpers

Please first implement me and then document me!

1.8 Watchouts

Dynamo is still in alpha state and there are still some serious watchouts, that you should have in mind, when using it:

- **Be aware of the User**: With Dynamo you are handing over a powerful tool to the user and there is a realistic chance that he breaks your overall data model.

- **Multiple Users changing the same model**: If multiple users are working on and changing the same model, this might create quite some trouble, and currently this is neither prevented nor somenow handled by Dynamo.
- Sophisticated Model Migrations
- Timing of Admin Updates

CHAPTER 2

Indices and tables

- genindex
- modindex
- search