MS CHARAC DOCUMENTATION

Release 1.0

Khadija Yasmine Mzabi

Jun 19, 2024

CONTENTS:

CHAPTER

ONE

MS-CHARAC-BACK

1.1 app package

1.1.1 Subpackages

app.migrations package

Submodules

app.migrations.0001_initial module

`class app.migrations.0001_initial.Migration(name, app_label)`

Bases: `Migration`

This migration script initializes the database schema for the application. It creates tables and relationships for various models including boards, connectors, folders, interfaces, I/O types, links, resource names, contacts, files, and associations.

Models and Fields:

- boards: Represents a board with a primary key field *board*.
- connecteur: Represents a connector with fields *dimension_row* and *dimension_column*.
- connector: Represents a connector with fields *name*, *row_dim*, and *column_dim*.
- Folder: Represents a folder with a field *folder_name*.
- interfaces: Represents an interface with a primary key field *interface*.
- io_type: Represents an I/O type with fields *io_type_name* and *io_type_symbol*.
- link: Represents a link with fields *first_interface*, *second_interface*, and *nom_link*.
- resource_name: Represents a resource name with a primary key field *name*.
- contacts: Represents a contact with fields *signal_name*, *num_row*, *num_column*, and a foreign key to *connecteur*.
- File: Represents a file with fields *file_name*, *data*, and a foreign key to *Folder*.
- interfaceConnector: Represents a connection between an interface and a connector with foreign keys to *interfaces* and *connector*.
- connecteur: Adds a foreign key to *interfaces*.

- association: Represents an association with fields *nombre_resource* and foreign keys to *interfaces* and *resource_name*.
- IO_list: Represents an I/O list with fields *resource_name*, *signal_name*, *nature*, *from_to*, *source_destination_board*, *description*, *connector*, and a foreign key to *association*.
- resources_categories: Represents resource categories with fields *category*, *count*, and a many-to-many relationship to *io_type*.
- modele_io_mapping: Represents a mapping of I/O models with fields *modele_name*, and many-to-many relationships to *interfaces*, *io_type*, *link*, *resource_name*, and *resources_categories*.
- assigned_resources: Represents assigned resources with fields *signal_name*, *board_internal_mapping*, *commentaire*, *io_index*, and foreign keys to *io_type* and *resources_categories*.
- wirings: Represents wirings with fields *first_signal*, *second_signal*, and a foreign key to *link*.

Dependencies:
This migration has no dependencies.

Operations:

- migrations.CreateModel: Creates each model and its fields as described above.

Commands to Generate and Apply Migration:

1. Generate the migration script: `` `bash python manage.py makemigrations app ` ``
2. Apply the migration to update the database schema: `` `bash python manage.py migrate ` ``

Possible Errors:

- ImportError: If Django is not installed or not available on the PYTHONPATH.
- OperationalError: If there is an issue with the database connection or schema.
- FieldError: If there are issues with field definitions, such as invalid field types or relationships.
- ProgrammingError: If there are issues with the SQL commands generated by the migration.
- IntegrityError: If there are issues with data integrity, such as violating unique constraints.

Notes:

- The *on_delete* parameter in foreign keys is used to handle deletion behavior, such as *CASCADE* or *SET_NULL*.
- The *related_name* parameter in foreign keys and many-to-many fields is used to specify the name of the reverse relation from the target model back to this one.

`dependencies = []`

`initial = True`

```
operations = [<CreateModel name='boards', fields=[('board',
<django.db.models.fields.TextField>)]>, <CreateModel name='connecteur',
fields=[('id', <django.db.models.fields.BigAutoField>), ('dimension_row',
<django.db.models.fields.IntegerField>), ('dimension_column',
<django.db.models.fields.IntegerField>)]>, <CreateModel name='connector',
fields=[('id', <django.db.models.fields.BigAutoField>), ('name',
<django.db.models.fields.CharField>), ('row_dim',
<django.db.models.fields.IntegerField>), ('column_dim',
<django.db.models.fields.IntegerField>)]>, <CreateModel name='Folder',
fields=[('id', <django.db.models.fields.BigAutoField>), ('folder_name',
<django.db.models.fields.CharField>)]>, <CreateModel name='interfaces',
fields=[('interface', <django.db.models.fields.TextField>)]>, <CreateModel
name='io_type', fields=[('io_type_name', <django.db.models.fields.CharField>),
('io_type_symbol', <django.db.models.fields.CharField>)]>, <CreateModel name='link',
fields=[('first_interface', <django.db.models.fields.CharField>),
('second_interface', <django.db.models.fields.CharField>), ('nom_link',
<django.db.models.fields.CharField>)]>, <CreateModel name='resource_name',
fields=[('name', <django.db.models.fields.TextField>)]>, <CreateModel
name='contacts', fields=[('id', <django.db.models.fields.BigAutoField>),
('signal_name', <django.db.models.fields.CharField>), ('num_row',
<django.db.models.fields.IntegerField>), ('num_column',
<django.db.models.fields.IntegerField>), ('connecteur',
<django.db.models.fields.related.ForeignKey>)]>, <CreateModel name='File',
fields=[('id', <django.db.models.fields.BigAutoField>), ('file_name',
<django.db.models.fields.CharField>), ('data',
<django.db.models.fields.BinaryField>), ('folder_name',
<django.db.models.fields.related.ForeignKey>)]>, <CreateModel
name='interfaceConnector', fields=[('id', <django.db.models.fields.BigAutoField>),
('connecteur', <django.db.models.fields.related.ForeignKey>), ('interface',
<django.db.models.fields.related.ForeignKey>)]>, <AddField model_name='connecteur',
name='interface', field=<django.db.models.fields.related.ForeignKey>>, <CreateModel
name='association', fields=[('id', <django.db.models.fields.BigAutoField>),
('nombre_resource', <django.db.models.fields.IntegerField>), ('interface',
<django.db.models.fields.related.ForeignKey>), ('resource',
<django.db.models.fields.related.ForeignKey>)]>, <CreateModel name='IO_list',
fields=[('id', <django.db.models.fields.BigAutoField>), ('resource_name',
<django.db.models.fields.IntegerField>), ('signal_name',
<django.db.models.fields.TextField>), ('nature',
<django.db.models.fields.CharField>), ('from_to',
<django.db.models.fields.CharField>), ('source_destination_board',
<django.db.models.fields.CharField>), ('description',
<django.db.models.fields.TextField>), ('connector',
<django.db.models.fields.CharField>), ('interface_resource',
<django.db.models.fields.related.ForeignKey>)]>, <CreateModel
name='resources_categories', fields=[('category',
<django.db.models.fields.CharField>), ('count',
<django.db.models.fields.IntegerField>), ('io_type',
<django.db.models.fields.related.ManyToManyField>)]>, <CreateModel
name='modele_io_mapping', fields=[('modele_name',
<django.db.models.fields.CharField>), ('interfaces',
<django.db.models.fields.related.ManyToManyField>), ('iotypes',
<django.db.models.fields.related.ManyToManyField>), ('links',
<django.db.models.fields.related.ManyToManyField>), ('resource_names',
<django.db.models.fields.related.ManyToManyField>), ('resources',
<django.db.models.fields.related.ManyToManyField>)]>, <CreateModel
name='assigned_resources', fields=[('id', <django.db.models.fields.BigAutoField>),
('signal_name', <django.db.models.fields.CharField>), ('board_internal_mapping',
<django.db.models.fields.CharField>), ('commentaire',
<django.db.models.fields.TextField>), ('io_index',
<django.db.models.fields.CharField>), ('io_type',
```

Module contents

1.1.2 Submodules

1.1.3 app.admin module

This module configures the Django admin interface for the application. It imports and registers various models with the Django admin to enable CRUD (Create, Read, Update, Delete) operations via the admin panel. Each model registration enhances the admin interface by allowing administrators to manage database records for the application directly through a web interface.

Usage:
This module is automatically used by Django's admin app to determine which models are available in the admin interface and how they are displayed. No further steps are required beyond importing this module in your admin.py file.

Modifications:
You can customize the behavior of models in the admin by creating ModelAdmin classes and registering them with their respective models. See Django's documentation on ModelAdmin for more details: https://docs.djangoproject.com/en/stable/ref/contrib/admin/#django.contrib.admin.ModelAdmin

Models Registered:

- Folder: Represents a folder in a file system (potentially virtual).
- File: Represents a file contained within a Folder.
- resources_categories: Categorizes various resources, potentially linking to io_types.
- io_type: Defines types of I/O operations or resources.
- assigned_resources: Links resources to specific tasks or uses in the system.
- resource_name: Names a particular resource, giving it a unique identifier.
- association: Represents a many-to-many relationship between resources and other elements in the system.
- connecteur: Represents a physical or logical connector within the system.
- interfaces: Defines various interfaces available within the system.
- IO_list: Lists I/O operations or tasks.
- contacts: Represents contact points or nodes in connecteurs.
- link: Represents a link or connection between two interfaces.
- wirings: Details the wiring configurations used within the system.
- modele_io_mapping: Maps I/O models to various components or configurations within the system.

Examples:
To add a model to the admin interface, simply import the model and use the *admin.site.register()* function: ```python from django.contrib import admin from .models import MyModel

admin.site.register(MyModel) ```

Notes:

- Ensure that each model has a unique representation in the admin; otherwise, it can be confusing to differentiate between similar records.

- Use *list_display*, *search_fields*, and other *ModelAdmin* options to enhance admin interface usability.

Warnings:

- Improper configuration of the admin can lead to security risks, e.g., exposing sensitive data or allowing unintended operations. Always restrict admin access using appropriate permissions.

1.1.4 app.apps module

class app.apps.**AppConfig**(*app_name*, *app_module*)

Bases: `AppConfig`

This class represents the configuration for the Django application named 'app'. It is responsible for setting the application's default auto field type and importing signal handlers when the application is ready.

Attributes:

default_auto_field (str): Specifies the type of auto-created primary key field to use by default. name (str): The name of the application.

Methods:

ready(): Imports the application's signal handlers.

default_auto_field = 'django.db.models.BigAutoField'

name = 'app'

ready()

This method is called when the application is ready. It imports the signal handlers defined in the 'app.signals' module.

Usage:

This method is automatically called by Django when the application is initialized. No direct call to this method is necessary.

Notes:

Ensure that the 'app.signals' module exists and contains the necessary signal handlers. Improper configuration or missing signal handlers can lead to application errors.

Examples:

To define a signal handler, create a 'signals.py' file in your 'app' directory and define your signals as follows:

```python
from django.db.models.signals import post_save
from django.dispatch import receiver
from .models import MyModel

@receiver(post_save, sender=MyModel)
def my_signal_handler(sender, instance, created, **kwargs):
    if created:
        # Handle the signal
        pass
```

This signal handler will be imported and connected when the application is ready.

1.1.5 app.manage module

Django's command-line utility for administrative tasks.

`app.manage.main()`

Run administrative tasks.

This function sets the *DJANGO_SETTINGS_MODULE* environment variable to specify the settings module for the Django project. It then attempts to import and execute Django's command-line utility for administrative tasks.

Environment Variables:
DJANGO_SETTINGS_MODULE (str): Specifies the settings module to be used by Django. It is set to 'mapping.settings' by default.

Exceptions:
ImportError: Raised if Django is not installed or not available on the PYTHONPATH environment variable.

Examples:
To execute a Django management command, run this script from the command line:

```
$ python3 ./manage.py <command>
```

Replace <*command*> with the desired administrative command, such as *runserver* or *migrate*.

Raises:
ImportError: If Django is not installed or the *DJANGO_SETTINGS_MODULE* is not set correctly.

Notes:

- This function is typically called automatically when the script is executed directly.
- It ensures that the Django environment is properly set up before running administrative tasks.

1.1.6 app.models module

class `app.models.File`(**args*, ***kwargs*)

Bases: `Model`

Represents a file contained within a Folder.

Attributes:
file_name (str): The name of the file.

folder_name (ForeignKey): A reference to the folder containing the file.

data (BinaryField): The binary data of the file.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

data

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

file_name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

folder_name

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

folder_name_id

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

class app.models.**Folder**(**args*, ***kwargs*)

Bases: `Model`

Represents a folder in a file system, potentially virtual.

Attributes:

folder_name (str): The name of the folder.

exception **DoesNotExist**

Bases: `ObjectDoesNotExist`

exception **MultipleObjectsReturned**

Bases: `MultipleObjectsReturned`

files

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

folder_name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

class `app.models.`**`IO_list`**(**args*, ***kwargs*)

Bases: `Model`

Represents a list of I/O operations or tasks.

Attributes:

resource_name (int): The name of the resource.

signal_name (str): The name of the signal.

nature (str): The nature of the I/O operation.

from_to (str): Indicates the source and destination of the I/O operation.

source_destination_board (str): The board involved in the I/O operation.

description (TextField): A description of the I/O operation.

connector (str): The connector associated with the I/O operation.

interface_resource (ForeignKey): A reference to the association model.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

connector

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

description

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

from_to

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

interface_resource

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOne-ToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

interface_resource_id

nature

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

`objects = <django.db.models.manager.Manager object>`

`resource_name`

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

`signal_name`

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

`source_destination_board`

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

`class app.models.assigned_resources`(**args*, ***kwargs*)

Bases: `Model`

Links resources to specific tasks or uses in the system.

Attributes:

signal_name (str): The name of the signal.

board_internal_mapping (str): The internal mapping of the board.

commentaire (TextField): Additional comments or notes.

io_type (ForeignKey): A reference to the io_type model.

io_index (str): The index of the I/O.

category (ForeignKey): A reference to the resources_categories model.

`exception DoesNotExist`

Bases: `ObjectDoesNotExist`

`exception MultipleObjectsReturned`

Bases: `MultipleObjectsReturned`

`board_internal_mapping`

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

`category`

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```python
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

`category_id`

`commentaire`

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

io_index

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

io_type

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

io_type_id

objects = <django.db.models.manager.Manager object>

signal_name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

class app.models.**association**(**args*, ***kwargs*)

Bases: `Model`

Represents a many-to-many relationship between resources and interfaces in the system.

Attributes:

resource (ForeignKey): A reference to the resource_name model.

interface (ForeignKey): A reference to the interfaces model.

nombre_resource (int): The number of resources associated with the interface.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

IO_list

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

interface

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

interface_id

nombre_resource

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

resource

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

resource_id

class app.models.**boards**(**args*, ***kwargs*)

Bases: `Model`

Represents a board in the system.

Attributes:

board (str): The name of the board.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

board

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

class app.models.**connecteur**(*id*, *dimension_row*, *dimension_column*, *interface*)

Bases: `Model`

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

contacts

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

dimension_column

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

dimension_row

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

interface

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

interface_id

objects = <django.db.models.manager.Manager object>

class app.models.**connector**(**args*, ***kwargs*)

Bases: `Model`

Represents a connector in the system.

Attributes:

name (str): The name of the connector.

row_dim (int): The row dimension of the connector.

column_dim (int): The column dimension of the connector.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

column_dim

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

interfaces

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

row_dim

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

class app.models.**contacts**(**args*, ***kwargs*)

Bases: `Model`

Represents contact points or nodes in connectors.

Attributes:

connecteur (ForeignKey): A reference to the connecteur model.

signal_name (str): The name of the signal.

num_row (int): The row number of the contact point.

num_column (int): The column number of the contact point.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

connecteur

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

connecteur_id

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

num_column

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

num_row

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

signal_name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

class app.models.**interfaceConnector**(**args*, ***kwargs*)

Bases: `Model`

Represents the connection between an interface and a connector in the system.

Attributes:

interface (ForeignKey): A reference to the interface model.

connecteur (ForeignKey): A reference to the connector model.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

connecteur

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

connecteur_id

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

interface

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

interface_id

objects = <django.db.models.manager.Manager object>

class app.models.**interfaces**(**args*, ***kwargs*)

Bases: `Model`

Represents an interface in the system.

Attributes:

interface (str): The name of the interface.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

connected

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

connectors

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

interface

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

interfaces

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

modele_io_mapping_set

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

objects = <django.db.models.manager.Manager object>

class app.models.**io_type**(*io_type_name*, *io_type_symbol*)

Bases: `Model`

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

assigned_resources_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

io_type_name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

io_type_symbol

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

modele_io_mapping_set

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

objects = <django.db.models.manager.Manager object>

resources_categories

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

class `app.models.`**link**(**args*, ***kwargs*)

Bases: `Model`

Represents a link or connection between two interfaces.

Attributes:

first_interface (str): The name of the first interface in the link.

second_interface (str): The name of the second interface in the link.

nom_link (str): The name of the link (primary key).

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

first_interface

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

modele_io_mapping_set

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

nom_link

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

second_interface

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

wirings

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

class app.models.**modele_io_mapping**(**args*, ***kwargs*)

Bases: `Model`

Maps I/O models to various components or configurations within the system.

Attributes:

modele_name (str): The name of the I/O model.

resources (ManyToManyField): A many-to-many relationship with the resources_categories model.

iotypes (ManyToManyField): A many-to-many relationship with the io_type model.

resource_names (ManyToManyField): A many-to-many relationship with the resource_name model.

interfaces (ManyToManyField): A many-to-many relationship with the interfaces model.

links (ManyToManyField): A many-to-many relationship with the link model.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

interfaces

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

iotypes

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

links

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

modele_name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

resource_names

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

resources

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

class `app.models.`**resource_name**(*name*)

Bases: `Model`

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

modele_io_mapping_set

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

name

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

resource

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

class app.models.**resources_categories**(**args*, ***kwargs*)

Bases: `Model`

Categorizes various resources, potentially linking to io_types.

Attributes:

category (str): The name of the resource category.

count (int, optional): The count of resources in this category.

io_type (ManyToManyField): A many-to-many relationship with the io_type model.

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

assigned_resources_set

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`category`

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

`count`

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

`io_type`

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`modele_io_mapping_set`

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`Pizza.toppings` and `Topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`objects = <django.db.models.manager.Manager object>`

class `app.models.`**`wirings`**`(*args, **kwargs)`

Bases: `Model`

Represents the wiring configuration used within the system.

Attributes:

link_type (ForeignKey): A reference to the link model.

first_signal (TextField): The first signal in the wiring configuration.

second_signal (TextField): The second signal in the wiring configuration.

`exception DoesNotExist`

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

first_signal

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

link_type

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

link_type_id

objects = <django.db.models.manager.Manager object>

second_signal

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

1.1.7 app.signals module

app.signals.**create_assigned_resources**(*sender*, *instance*, *created*, ***kwargs*)

Signal receiver that creates or updates assigned resources when a resources_categories instance is saved.

This function is triggered after a resources_categories instance is saved. It handles both the creation of new assigned resources when a new category is created and the update of existing assigned resources when a category's count is modified.

Args:

sender (Model): The model class that sent the signal. instance (resources_categories): The instance of the model that was saved. created (bool): A boolean indicating whether a new record was created. **kwargs: Additional keyword arguments.

Examples:

When a new resources_categories instance is created:

```
category = resources_categories.objects.create(category='NewCategory', count=5)
# This will automatically create 5 assigned_resources instances for the new␣
↪category.
```

When an existing resources_categories instance is updated:

```
category = resources_categories.objects.get(category='NewCategory')
category.count = 10
category.save()
```

(continues on next page)

(continued from previous page)

```
# This will automatically update the assigned_resources instances to match the␣
↪new count.
```

Notes:

- The *create_assigned_resources* function uses Django's *post_save* signal to automatically manage the creation and updating of *assigned_resources* instances. This ensures that the number of *assigned_resources* matches the *count* attribute of the *resources_categories* instance.
- If the *resources_categories* instance is newly created, the function creates the specified number of *assigned_resources* instances.
- If the *resources_categories* instance is updated, the function checks whether the number of existing *assigned_resources* instances matches the *count* attribute. If not, it deletes the existing instances and creates new ones to match the updated count.
- This automation helps maintain data consistency and integrity by ensuring that the number of *assigned_resources* is always in sync with the *count* attribute of the *resources_categories* instance.

1.1.8 app.tests module

1.1.9 app.urls module

URL Configuration for the Application.

This module defines the URL patterns for the application. Each URL pattern is associated with a specific view function that handles the corresponding HTTP request. The URLs cover various functionalities including retrieving data, updating resources, and managing associations.

Usage:
These URL patterns are automatically used by Django to route HTTP requests to the appropriate view functions. Each pattern is mapped to a specific endpoint and a view function.

Examples:
To add a new URL pattern, import the view function and add a path entry to the urlpatterns list:

```
from . import views
urlpatterns.append(path('new_endpoint/', views.new_view, name='new_view'))
```

Attributes:
urlpatterns (list): A list of URL patterns to be matched against incoming requests.

Notes:

- Ensure that the view functions referenced in the URL patterns are defined in the views module.
- URL patterns can include parameters to capture parts of the URL and pass them to the view function.

URL Patterns:

- '' : Home page, handled by *views.home*.
- 'get_iotypes/' : Retrieve I/O types, handled by *views.get_iotypes*.
- 'get_resources/' : Retrieve resources, handled by *views.get_resources*.
- 'get_signals/' : Retrieve signals, handled by *views.get_signals*.
- 'get_names/' : Retrieve resource names, handled by *views.get_names*.

- ‘get_boards/’ : Retrieve boards, handled by *views.get_boards*.
- ‘get_infos_boards/’ : Retrieve board information, handled by *views.get_infos_boards*.
- ‘get_interfaces/’ : Retrieve interfaces, handled by *views.get_interfaces*.
- ‘get_infos_interfaces/’ : Retrieve interface information, handled by *views.get_infos_interfaces*.
- ‘get_iosignals/’ : Retrieve I/O signals, handled by *views.get_iosignals*.
- ‘get_associations/’ : Retrieve associations, handled by *views.get_associations*.
- ‘get_connecteur/’ : Retrieve connectors, handled by *views.get_connecteur*.
- ‘update_io/’ : Update I/O details, handled by *views.update_io*.
- ‘delete_iotype/’ : Delete an I/O type, handled by *views.delete_iotype*.
- ‘update_resource/’ : Update a resource, handled by *views.update_resource*.
- ‘delete_ressourceCategory/’ : Delete a resource category, handled by *views.delete_ressourceCategory*.
- ‘update_signal/’ : Update a signal, handled by *views.update_signal*.
- ‘delete_signal/’ : Delete a signal, handled by *views.delete_signal*.
- ‘update_name/’ : Update a resource name, handled by *views.update_name*.
- ‘delete_name/’ : Delete a resource name, handled by *views.delete_name*.
- ‘update_board/’ : Update a board, handled by *views.update_board*.
- ‘delete_board/’ : Delete a board, handled by *views.delete_board*.
- ‘update_interface/’ : Update an interface, handled by *views.update_interface*.
- ‘delete_interface/’ : Delete an interface, handled by *views.delete_interface*.
- ‘associate/’ : Create an association, handled by *views.associate*.
- ‘delete_association/’ : Delete an association, handled by *views.delete_association*.
- ‘update_iosignal/’ : Update an I/O signal, handled by *views.update_iosignal*.
- ‘delete_iosignal/’ : Delete an I/O signal, handled by *views.delete_iosignal*.
- ‘associate_connecteur/’ : Associate a connector, handled by *views.associate_connecteur*.
- ‘get_connector/’ : Retrieve a connector, handled by *views.get_connector*.
- ‘update_connector/’ : Update a connector, handled by *views.update_connector*.
- ‘delete_connector/’ : Delete a connector, handled by *views.delete_connector*.
- ‘update_interface_connector/’ : Update an interface-connector relationship, handled by *views.update_interface_connector*.
- ‘delete_interface_connector/’ : Delete an interface-connector relationship, handled by *views.delete_interface_connector*.

1.1.10 app.views module

`app.views.associate(request)`

Add an association between an interface and a resource type based on the provided data.

This view function handles the creation of an *association* object. It processes the incoming JSON data to create a new association between an interface and a resource type and returns a JSON response indicating success.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success of the operation.

Examples:
Example JSON request body for adding an association:

```
{
    "interface": "Interface1",
    "resource type": "ResourceType1",
    "count": 20
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "We associated an interface with a resource type."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It retrieves the *interfaces* and *resource_name* objects based on the provided data.
- It creates a new *association* object with the specified interface, resource type, and count.
- The function always returns a success message on completion.

`app.views.associate_connecteur(request)`

Associate a connector (connecteur) with an interface based on the provided data.

This view function handles the creation of a *connecteur* object and associates it with an *interfaces* object. It processes the incoming JSON data to create the connector and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation.

Examples:
Example JSON request body for associating a connecteur:

```
{
    "interface": "Interface1",
    "row": 10,
```

(continues on next page)

(continued from previous page)

```
    "column": 5
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Successfully associated an interface with a connecteur."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It checks for the presence of required fields in the received data.
- It retrieves the *interfaces* object by the provided interface name.
- It creates a new *connecteur* object with the specified dimensions and associates it with the interface.
- The function returns a success message on successful association.
- On error, it returns a JSON response with an appropriate error message and status code.

app.views.**delete_association**(*request*)

Delete an association based on the provided data.

This view function handles the deletion of an *association* object. It processes the incoming JSON data to identify the association to delete and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation.

Examples:
Example JSON request body for deleting an association:

```
{
    "association": "Association1"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Association deleted successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It fetches the *association* object by the provided association identifier and deletes it.
- On error, it returns a JSON response with an appropriate error message and status code.

app.views.**delete_board**(*request*)

Delete a board based on the provided data.

This view function handles the deletion of a *boards* object. It processes the incoming JSON data to identify the board to delete and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation.

Examples:
Example JSON request body for deleting a board:

```
{
    "board": "Board1"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Board deleted successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It fetches the *boards* object by the provided board name and deletes it.
- On error, it returns a JSON response with an appropriate error message and status code.

app.views.**delete_connecteur**(*request*)

Delete a connecteur based on the provided data.

This view function handles the deletion of a *connecteur* object. It processes the incoming JSON data to identify the connecteur to delete and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation.

Examples:
Example JSON request body for deleting a connecteur:

```
{
    "connecteur": "Connecteur1"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Connecteur deleted successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It fetches the *connecteur* object by the provided connecteur name and deletes it.
- The function returns a success message on successful deletion.
- On error, it returns a JSON response with an appropriate error message and status code.

app.views.**delete_connector**(*request*)

Delete a connector based on the provided data.

This view function handles the deletion of a *connector* object. It processes the incoming JSON data to identify the connector to delete and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation.

Examples:
Example JSON request body for deleting a connector:

```
{
    "name": "Connector1"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Connector deleted successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It fetches the *connector* object by the provided name and deletes it.
- On error, it returns a JSON response with the error message and a 404 status code if the connector is not found.

app.views.**delete_interface**(*request*)

Delete an interface based on the provided data.

This view function handles the deletion of an *interfaces* object. It processes the incoming JSON data to identify the interface to delete and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation.

Examples:
Example JSON request body for deleting an interface:

```
{
    "interface": "Interface1"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Interface deleted successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It fetches the *interfaces* object by the provided interface name and deletes it.
- On error, it returns a JSON response with an appropriate error message and status code.

`app.views.`**`delete_interface_connector`**`(`*request*`)`

Delete an interface-connector association based on the provided data.

This view function handles the deletion of an *interfaceConnector* object. It processes the incoming JSON data to identify the interface-connector association to delete and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation.

Examples:
Example JSON request body for deleting an interface-connector association:

```
{
    "interface_connector_id": 1
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "InterfaceConnector deleted successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It fetches the *interfaceConnector* object by the provided ID and deletes it.
- The function returns a success message on successful deletion.
- On error, it returns a JSON response with an appropriate error message and status code.

`app.views.`**`delete_iosignal`**`(`*request*`)`

Delete an I/O signal based on the provided data.

This view function handles the deletion of an *assigned_resources* object. It processes the incoming JSON data to identify the signal to delete and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation.

Examples:
Example JSON request body for deleting an I/O signal:

```
{
    "signal": "Signal1"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Signal deleted successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It fetches the *assigned_resources* object by the provided signal name and deletes it.
- On error, it returns a JSON response with an appropriate error message and status code.

app.views.**delete_iotype**(*request*)

Delete an I/O type based on the provided data.

This view function handles the deletion of an *io_type* object. It processes the incoming JSON data to identify the I/O type to delete and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation.

Examples:
Example JSON request body for deleting an I/O type:

```
{
    "ioType": "IOType1"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "IO type deleted successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It fetches the *io_type* object by the provided I/O type name and deletes it.
- On error, it returns a JSON response with an appropriate error message and status code.

`app.views.`**`delete_name`**`(`*request*`)`

Delete a resource name based on the provided data.

This view function handles the deletion of a *resource_name* object. It processes the incoming JSON data to identify the resource name to delete and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation.

Examples:
Example JSON request body for deleting a resource name:

```
{
    "resource type": "ResourceType1"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Resource name deleted successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It fetches the *resource_name* object by the provided resource type and deletes it.
- On error, it returns a JSON response with an appropriate error message and status code.

`app.views.`**`delete_ressourceCategory`**`(`*request*`)`

Delete a resource category based on the provided data.

This view function handles the deletion of a *resources_categories* object. It processes the incoming JSON data to identify the resource category to delete, performs the deletion, and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation. On success, it includes a success message.

Examples:
Example JSON request body:

```
{
    "resourcename": "Resource1"
}
```

Example JSON response on success:

```
{
    "success": True,
```

(continues on next page)

(continued from previous page)

```
    "message": "Resource category deleted successfully"
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It fetches the *resources_categories* object by the provided category name.
- The *delete* method is called on the fetched object, which will cascade and delete related entries if the ForeignKey is set with *on_delete=models.CASCADE*.
- On error, it returns a JSON response with the error message and a 500 status code.

`app.views.delete_signal(request)`

Delete a signal based on the provided data.

This view function handles the deletion of an *assigned_resources* object. It processes the incoming JSON data to identify the signal to delete and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation.

Examples:
Example JSON request body for deleting a signal:

```
{
    "signal": "Signal1"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Signal deleted successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It fetches the *assigned_resources* object by the provided signal name and deletes it.
- On error, it returns a JSON response with an appropriate error message and status code.

`app.views.get_associations(request)`

Retrieve all associations between interfaces and resource types from the database and return them as a JSON response.

This view function fetches all *association* objects from the database, constructs a list of dictionaries representing each association, and returns the list as a JSON response. Each dictionary contains details about the interface, resource type, and count of resources.

Args:
request (HttpRequest): The HTTP request object.

Returns:
JsonResponse: A JSON response containing a list of dictionaries, each representing an association.

Examples:
Example JSON response:

```
[
    {
        "interface": "Interface1",
        "resource type": "ResourceType1",
        "count": 20
    },
    {
        "interface": "Interface2",
        "resource type": "ResourceType2",
        "count": 15
    }
]
```

Notes:

- This view is typically used to provide data for an ag-Grid component in a web application.
- The *safe* parameter in *JsonResponse* is set to *False* to allow serializing a list of dictionaries.

app.views.**get_boards**(*request*)

Retrieve all boards from the database and return them as a JSON response.

This view function fetches all *boards* objects from the database, constructs a list of dictionaries representing each board, and returns the list as a JSON response. Each dictionary contains the board name.

Args:
request (HttpRequest): The HTTP request object.

Returns:
JsonResponse: A JSON response containing a list of dictionaries, each representing a board.

Examples:
Example JSON response:

```
[
    {
        "board": "Board1"
    },
    {
        "board": "Board2"
    }
]
```

Notes:

- This view is typically used to provide data for an ag-Grid component in a web application.
- The *safe* parameter in *JsonResponse* is set to *False* to allow serializing a list of dictionaries.

app.views.**get_connecteur**(*request*)

Retrieve connectors associated with a specified interface and return them as a JSON response.

This view function fetches *connecteur* objects associated with a specified interface from the database, constructs a list of dictionaries representing each connector, and returns the list as a JSON response. Each dictionary contains the row and column dimensions of the connector.

Args:
request (HttpRequest): The HTTP request object.

Returns:
JsonResponse: A JSON response containing a list of dictionaries, each representing a connector.

Examples:
Example JSON request body:

```
{
    "interface": "Interface1"
}
```

Example JSON response:

```
[
    {
        "row": 10,
        "column": 5
    },
    {
        "row": 8,
        "column": 4
    }
]
```

Notes:

- This view expects a POST request with a JSON body containing the *interface* field.
- The *safe* parameter in *JsonResponse* is set to *False* to allow serializing a list of dictionaries.

`app.views.get_connector(request)`

Retrieve connector details based on a specified ID or return all connectors if no ID is specified.

This view function fetches a *connector* object from the database based on the provided ID and returns its details as a JSON response. If no ID is specified, it returns a list of all connectors. Each dictionary in the response contains the name, row dimension, and column dimension of the connector.

Args:
request (HttpRequest): The HTTP request object.

Returns:
JsonResponse: A JSON response containing the details of a specific connector or a list of all connectors.

Examples:
Example JSON request with ID:

```
GET /get_connector?id=1 HTTP/1.1
```

Example JSON response for a specific connector:

```
{
    "success": True,
    "connector": {
        "name": "Connector1",
        "row_dim": 10,
        "column_dim": 5
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Example JSON response for all connectors:

```
{
    "success": True,
    "connectors": [
        {
            "name": "Connector1",
            "row_dim": 10,
            "column_dim": 5
        },
        {
            "name": "Connector2",
            "row_dim": 8,
            "column_dim": 4
        }
    ]
}
```

Notes:

- This view expects a GET request with an optional *id* query parameter.
- If the *id* parameter is provided, the view returns the details of the specified connector.
- If no *id* parameter is provided, the view returns a list of all connectors.
- On error, the view returns a JSON response with an appropriate error message and status code.

app.views.**get_infos_boards**(*request*)

Retrieve board information to facilitate the selection of interfaces and return it as a JSON response.

This view function fetches all *boards* objects from the database, constructs two lists of dictionaries representing the boards, and returns them in a JSON response. Each list represents boards for different selection purposes.

Args:
request (HttpRequest): The HTTP request object.

Returns:
JsonResponse: A JSON response containing two lists of dictionaries, each representing a board for different purposes.

Examples:
Example JSON response:

```
{
    "liste board 1": [
        {"Board 1": "Board1"},
        {"Board 1": "Board2"}
    ],
    "liste board 2": [
        {"Board 2": "Board1"},
        {"Board 2": "Board2"}
    ]
]
```

Notes:

- This view is designed to facilitate the selection of interfaces by providing board information in two separate lists.
- The *safe* parameter in *JsonResponse* is set to *False* to allow serializing a dictionary containing lists.

`app.views.`**`get_infos_interfaces`**`(`*request*`)`

Retrieve interfaces and resource names from the database and return them as a JSON response.

This view function fetches all *interfaces* and *resource_name* objects from the database, constructs lists of dictionaries representing each interface and resource name, and returns them in a JSON response.

Args:
request (HttpRequest): The HTTP request object.

Returns:
JsonResponse: A JSON response containing two lists of dictionaries, one for interfaces and one for resource names.

Examples:
Example JSON response:

```
{
    "liste noms": [
        {"resource type": "ResourceType1"},
        {"resource type": "ResourceType2"}
    ],
    "liste interfaces": [
        {"interface": "Interface1"},
        {"interface": "Interface2"}
    ]
]
```

Notes:

- This view is typically used to provide data for an ag-Grid component in a web application.
- The *safe* parameter in *JsonResponse* is set to *False* to allow serializing a dictionary containing lists.

`app.views.`**`get_interfaces`**`(`*request*`)`

Retrieve all interfaces from the database and return them as a JSON response.

This view function fetches all *interfaces* objects from the database, constructs a list of dictionaries representing each interface, and returns the list as a JSON response. Each dictionary contains details about the boards, interface name, and connector name.

Args:
request (HttpRequest): The HTTP request object.

Returns:
JsonResponse: A JSON response containing a list of dictionaries, each representing an interface.

Examples:
Example JSON response:

```
[
    {
        "board1": "Board1",
        "board2": "Board2",
```

(continues on next page)

(continued from previous page)

```
        "interfaceName": "Board1_Board2",
        "connectorName": "Connector1"
    },
    {
        "board1": "Board3",
        "board2": "Board4",
        "interfaceName": "Board3_Board4",
        "connectorName": "Connector2"
    }
]
```

Notes:

- This view is typically used to provide data for an ag-Grid component in a web application.
- The *safe* parameter in *JsonResponse* is set to *False* to allow serializing a list of dictionaries.

app.views.**get_io_res**(*resource*)

Retrieve the I/O types associated with a given resource.

This function fetches all *io_type* objects associated with the provided resource, formats them into a list of their names, and returns the list.

Args:
resource (resources_categories): The resource category for which to retrieve associated I/O types.

Returns:
list: A list of I/O type names associated with the provided resource.

Examples:
Assuming *resource* is an instance of *resources_categories*:

```
resource = resources_categories.objects.get(category='Resource1')
io_types = get_io_res(resource)
print(io_types)  # Output: ['IOType1', 'IOType2']
```

Notes:

- The function uses a reverse relationship lookup to find *io_type* objects associated with the given resource.
- The returned list contains only the names of the I/O types.

app.views.**get_iosignals**(*request*)

Retrieve all I/O signals from the database and return them as a JSON response.

This view function fetches all *assigned_resources* objects from the database, constructs a list of dictionaries representing each I/O signal, and returns the list as a JSON response. Each dictionary contains details about the interface, resource name, signal number, signal name, from/to information, source destination board, description, and nature.

Args:
request (HttpRequest): The HTTP request object.

Returns:
JsonResponse: A JSON response containing a list of dictionaries, each representing an I/O signal.

Examples:
Example JSON response:

```
[
    {
        "interface": "Interface1",
        "resource name": "Resource1",
        "signal number": 1,
        "signal name": "Signal1",
        "from/to": "From1/To1",
        "source destination board": "Board1",
        "description": "Description1",
        "nature": "Nature1"
    },
    {
        "interface": "Interface2",
        "resource name": "Resource2",
        "signal number": 2,
        "signal name": "Signal2",
        "from/to": "From2/To2",
        "source destination board": "Board2",
        "description": "Description2",
        "nature": "Nature2"
    }
]
```

Notes:

- This view is typically used to provide data for an ag-Grid component in a web application.
- The *safe* parameter in *JsonResponse* is set to *False* to allow serializing a list of dictionaries.

`app.views.`**`get_iotypes`**`(`*request*`)`

Retrieve all I/O types from the database and return them as a JSON response.

This view function fetches all *io_type* objects from the database and constructs a list of dictionaries representing each I/O type. Each dictionary contains the keys 'ioType' and 'symbol' corresponding to the I/O type's name and symbol, respectively. The resulting list is returned as a JSON response.

Args:
request (HttpRequest): The HTTP request object.

Returns:
JsonResponse: A JSON response containing a list of dictionaries, each representing an I/O type.

Examples:
Example JSON response:

```
[
    {"ioType": "Type1", "symbol": "T1"},
    {"ioType": "Type2", "symbol": "T2"}
]
```

Notes:

- This view is typically used to provide data for an ag-Grid component in a web application.
- The *safe* parameter in *JsonResponse* is set to *False* to allow serializing a list of dictionaries.

`app.views.`**`get_names`**`(`*request*`)`

Retrieve all resource names from the database and return them as a JSON response.

This view function fetches all *resource_name* objects from the database, constructs a list of dictionaries representing each resource name, and returns the list as a JSON response. Each dictionary contains the resource type.

Args:
request (HttpRequest): The HTTP request object.

Returns:
JsonResponse: A JSON response containing a list of dictionaries, each representing a resource name.

Examples:
Example JSON response:

```
[
    {
        "resource type": "ResourceType1"
    },
    {
        "resource type": "ResourceType2"
    }
]
```

Notes:

- This view is typically used to provide data for an ag-Grid component in a web application.
- The *safe* parameter in *JsonResponse* is set to *False* to allow serializing a list of dictionaries.

app.views.**get_resources**(*request*)

Retrieve all resource categories from the database and return them as a JSON response.

This view function fetches all *resources_categories* objects from the database, constructs a list of dictionaries representing each resource category, and returns the list as a JSON response. Each dictionary contains the resource name, count, and a list of associated I/O types.

Args:
request (HttpRequest): The HTTP request object.

Returns:
JsonResponse: A JSON response containing a list of dictionaries, each representing a resource category.

Examples:
Example JSON response:

```
[
    {
        "resourcename": "Resource1",
        "count": 10,
        "io type": ["IOType1", "IOType2"]
    },
    {
        "resourcename": "Resource2",
        "count": 5,
        "io type": ["IOType3"]
    }
]
```

Notes:

- This view is typically used to provide data for an ag-Grid component in a web application.

- The *safe* parameter in *JsonResponse* is set to *False* to allow serializing a list of dictionaries.

`app.views.`**`get_signals`**`(`*request*`)`

Retrieve all signals from the database and return them as a JSON response.

This view function fetches all *assigned_resources* objects from the database, constructs a list of dictionaries representing each signal, and returns the list as a JSON response. Each dictionary contains details about the signal, including the resource, signal name, board internal mapping, comment, I/O type, and I/O index.

Args:
request (HttpRequest): The HTTP request object.

Returns:
JsonResponse: A JSON response containing a list of dictionaries, each representing a signal.

Examples:
Example JSON response:

```
[
    {
        "resource": "Resource1",
        "signal name": "Signal1",
        "board internal mapping": "Mapping1",
        "commentaire": "Comment1",
        "io type": "IOType1",
        "io index": "1"
    },
    {
        "resource": "Resource2",
        "signal name": "Signal2",
        "board internal mapping": "Mapping2",
        "commentaire": "Comment2",
        "io type": "IOType2",
        "io index": "2"
    }
]
```

Notes:

- This view is typically used to provide data for an ag-Grid component in a web application.
- The *safe* parameter in *JsonResponse* is set to *False* to allow serializing a list of dictionaries.

`app.views.`**`home`**`(`*request*`)`

Handle requests to the home page.

This view function processes both GET and POST requests to the home page. It handles the creation of folders and files based on the submitted form data and retrieves all folders, I/O types, and resource categories to be displayed on the page.

Args:
request (HttpRequest): The HTTP request object.

Returns:
HttpResponse: The HTTP response object with the rendered home page.

POST Handling:

- If the request method is POST, it checks the form_name to determine the type of form submitted.
- **For 'file_form':**

    - Retrieves or creates the specified folder.
    - Processes the uploaded files and saves them to the folder.

- **For 'folder_form':**
    - Retrieves the folder name from the request.
    - Checks if a folder with the same name already exists.
    - Creates a new folder if it doesn't exist or handles the case where it already exists.
    - Redirects to the home page to avoid form resubmission on page refresh.

Examples:

- **Submitting a 'file_form' with selected files:**

```
data = {
    'form_name': 'file_form',
    'selected-folder': 'Documents',
    'files': [<file1>, <file2>]
}
response = client.post('/', data)
```

- **Submitting a 'folder_form' to create a new folder:**

```
data = {
    'form_name': 'folder_form',
    'folder_name': 'NewFolder'
}
response = client.post('/', data)
```

Notes:

- The view handles file uploads and folder creation.
- It prevents folder name duplication by checking if a folder with the same name already exists.
- Redirects after folder creation to avoid duplicate form submissions on page refresh.

Returns:
HttpResponse: The rendered home page with context data including folders, I/O types, and resource categories.

`app.views.`**`update_board`**`(`*request*`)`

Add or update a board based on the provided data.

This view function handles the creation or updating of a *boards* object. It processes the incoming JSON data to either create a new board or update an existing one and returns a JSON response indicating success.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success of the operation.

Examples:
Example JSON request body for adding or updating a board:

```
{
    "board": "Board1"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Board added successfully."
}
```

Example JSON response on update:

```
{
    "success": True,
    "message": "Board updated successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It uses *update_or_create* to ensure the board exists or creates a new one if it does not.
- The function returns a message indicating whether the board was created or updated.

`app.views.`**`update_connector`**`(`*`request`*`)`

Create or update a connector based on the provided data.

This view function handles the creation or updating of a *connector* object. It processes the incoming JSON data, updates or creates the connector, and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation. On success, it includes the details of the created or updated connector.

Examples:
Example JSON request body for creating or updating a connector:

```
{
    "name": "Connector1",
    "row_dim": 10,
    "column_dim": 5
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Connector created successfully.",
    "connector": {
        "name": "Connector1",
        "row_dim": 10,
        "column_dim": 5
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It uses *update_or_create* to ensure the connector exists or creates a new one if it does not.
- On error, it returns a JSON response with the error message and a 500 status code.

app.views.**update_interface**(*request*)

Add or update an interface based on the provided data.

This view function handles the creation or updating of an *interfaces* object. It processes the incoming JSON data to either create a new interface or update an existing one and returns a JSON response indicating success.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success of the operation.

Examples:
Example JSON request body for adding or updating an interface:

```
{
    "board1": "Board1",
    "board2": "Board2",
    "connector": "Connector1"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Interface updated successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It uses *get_or_create* to ensure the boards exist or creates new ones if they do not.
- It uses *update_or_create* to ensure the interface exists or creates a new one if it does not.
- The function always returns a success message on completion.

app.views.**update_interface_connector**(*request*)

Update an interface-connector association based on the provided data.

This view function handles the updating of an *interfaceConnector* object. It processes the incoming JSON data to update the interface and connector associations and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation.

Examples:

Example JSON request body for updating an interface-connector association:

```
{
    "interface_connector_id": 1,
    "interface_id": 2,
    "connector_id": 3
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "InterfaceConnector updated successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It fetches the *interfaceConnector* object by the provided ID and updates its attributes.
- The function returns a success message on successful update.
- On error, it returns a JSON response with an appropriate error message and status code.

app.views.**update_io**(*request*)

Create or update an I/O type based on the provided data.

This view function handles the creation or updating of an *io_type* object. It processes the incoming JSON data to either create a new I/O type or update an existing one and returns a JSON response indicating success or failure.

Args:

request (HttpRequest): The HTTP request object containing the JSON data.

Returns:

JsonResponse: A JSON response indicating the success or failure of the operation. On success, it includes the details of the created or updated I/O type.

Examples:

Example JSON request body for creating an I/O type:

```
{
    "ioType": "IOType1",
    "symbol": "Symbol1"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "IO type created successfully.",
    "io": {
        "ioType": "IOType1",
        "symbol": "Symbol1"
    }
}
```

Example JSON request body for updating an I/O type:

```
{
    "ioType": "IOType1",
    "symbol": "NewSymbol"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "IO type updated successfully.",
    "io": {
        "ioType": "IOType1",
        "symbol": "NewSymbol"
    }
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It uses *get_or_create* to ensure the I/O type exists or creates a new one if it does not.
- On error, it returns a JSON response with an appropriate error message and status code.

app.views.**update_iosignal**(*request*)

Update an I/O signal based on the provided data.

This view function handles the updating of an *assigned_resources* object. It processes the incoming JSON data to update the signal's attributes and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation.

Examples:
Example JSON request body for updating an I/O signal:

```
{
    "interface": "Interface1",
    "resource name": "Resource1",
    "signal number": 1,
    "signal name": "NewSignalName",
    "from/to": "From1/To1",
    "source destination board": "Board1",
    "description": "NewDescription",
    "nature": "NewNature"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Signal updated successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It retrieves the *interfaces* and *resource_name* objects based on the provided data.
- It updates the attributes of the *assigned_resources* object and saves it.
- The function always returns a success message on completion.

app.views.**update_name**(*request*)

Add a new resource name based on the provided data.

This view function handles the creation of a *resource_name* object. It processes the incoming JSON data to create a new resource name and returns a JSON response indicating success.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success of the operation.

Examples:
Example JSON request body for adding a new resource name:

```
{
    "resource type": "ResourceType1"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Resource name added successfully."
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It creates a new *resource_name* object with the provided resource type.
- The function always returns a success message on completion.

app.views.**update_resource**(*request*)

Update or create a resource category based on the provided data.

This view function handles the updating or creation of a *resources_categories* object. It processes the incoming JSON data, updates or creates the resource category, associates the specified I/O types, and returns a JSON response indicating success or failure.

Args:
request (HttpRequest): The HTTP request object containing the JSON data.

Returns:
JsonResponse: A JSON response indicating the success or failure of the operation. On success, it includes the updated resource details.

Examples:
Example JSON request body:

```
{
    "resourcename": "Resource1",
```

(continues on next page)

(continued from previous page)

```
    "count": 10,
    "ioTypes": ["IOType1", "IOType2"]
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Resource name updated successfully",
    "resource": {
        "resourcename": "Resource1",
        "count": 10,
        "ioTypes": ["IOType1", "IOType2"]
    }
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It uses *get_or_create* to ensure the resource category exists.
- The function associates the specified I/O types with the resource category.
- It returns a detailed JSON response including the updated resource information on success.
- On error, it returns a JSON response with the error message and a 500 status code.

app.views.**update_signal**(*request*)

Update a signal based on the provided data.

This view function handles the updating of an *assigned_resources* object. It processes the incoming JSON data to update the signal's attributes and returns a JSON response indicating success or failure.

Args:

request (HttpRequest): The HTTP request object containing the JSON data.

Returns:

JsonResponse: A JSON response indicating the success or failure of the operation. On success, it includes the details of the updated signal.

Examples:

Example JSON request body for updating a signal:

```
{
    "resource": "Resource1",
    "signal number": 1,
    "signal name": "NewSignalName",
    "board internal mapping": "NewMapping",
    "commentaire": "NewComment",
    "ioType": "NewIOType",
    "resourceFPGA": "NewResourceFPGA"
}
```

Example JSON response on success:

```
{
    "success": True,
    "message": "Signal updated successfully.",
    "signal": {
        "resource": "Resource1",
        "signal name": "NewSignalName",
        "board internal mapping": "NewMapping",
        "io type": "NewIOType",
        "io index": "NewIOIndex"
    }
}
```

Notes:

- The function uses *json.loads* to parse the incoming JSON data.
- It retrieves the corresponding *resources_categories* object based on the provided resource name.
- It retrieves the *assigned_resources* object using the signal number and category.
- The signal's attributes are updated with the provided data, and the signal is saved.
- The function returns a detailed JSON response including the updated signal information on success.
- On error, it returns a JSON response with an appropriate error message and status code.

1.1.11 Module contents

1.2 generate_documentation package

1.2.1 Submodules

1.2.2 generate_documentation.adjust_paths_rst module

This script corrects the module paths in the *modules.rst* file generated by Sphinx.

It ensures that all module paths in the *modules.rst* file are prefixed with the project root module name (*MS-CHARAC-BACK.*). This is necessary for Sphinx to correctly locate and document the modules within the project.

The script performs the following steps: 1. Reads the content of the *modules.rst* file. 2. Checks each line for the .. *automodule::* directive. 3. Adds the project root prefix (*MS-CHARAC-BACK.*) to the module name if it is not already present. 4. Writes the corrected content back to the *modules.rst* file.

Usage:
Simply run the script in an environment where the *BASE_DIR* environment variable is set to the root directory of your project.

Examples:
Assuming the *BASE_DIR* environment variable is set, run the script as follows:

`` `python python correct_modules_rst.py ` ``

Notes:

- Ensure that the *BASE_DIR* environment variable is correctly set to the root directory of your project before running this script.
- This script assumes that the *modules.rst* file is located in the *docs* directory under the project root.

Possible Errors:

- FileNotFoundError: If the *modules.rst* file is not found at the specified path.
- IOError: If there are issues with reading from or writing to the *modules.rst* file.

1.2.3 generate_documentation.modify_conf module

This script modifies the Sphinx *conf.py* and *index.rst* files to correctly configure the documentation build process for a Django project.

It ensures that: 1. The necessary paths and settings for Django are added. 2. Required Sphinx extensions are included. 3. Unnecessary TODO settings are removed. 4. The HTML theme is updated. 5. The *modules* directive is added to *index.rst* for module documentation.

Steps Performed:

1. Read and modify *conf.py* to: - Add the Django setup and *sys.path* modifications. - Ensure necessary Sphinx extensions are included. - Update *exclude_patterns* to exclude build artifacts. - Change the HTML theme to *sphinx_rtd_theme*. - Remove *todo_include_todos* if present.
2. Read and modify *index.rst* to: - Add the *modules* directive under the *:caption: Contents:* section.

Usage:
Simply run the script in an environment where the *BASE_DIR* environment variable is set to the root directory of your project.

Examples:
Assuming the *BASE_DIR* environment variable is set, run the script as follows:

`` `python python modify_sphinx_files.py ` ``

Possible Errors:

- FileNotFoundError: If the *conf.py* or *index.rst* file is not found at the specified path.
- IOError: If there are issues with reading from or writing to the *conf.py* or *index.rst* file.

Notes:

- Ensure that the *BASE_DIR* environment variable is correctly set to the root directory of your project before running this script.
- This script assumes that the *conf.py* and *index.rst* files are located in the *docs* directory under the project root.
- It also ensures a brief wait time using *time.sleep(2)* after making modifications.

1.2.4 Module contents

1.3 mapping package

1.3.1 Submodules

1.3.2 mapping.asgi module

ASGI config for mapping project.

It exposes the ASGI callable as a module-level variable named `application`.

For more information on this file, see https://docs.djangoproject.com/en/3.2/howto/deployment/asgi/

1.3.3 mapping.settings module

Django settings for mapping project.

Generated by ‘django-admin startproject’ using Django 3.2.20.

For more information on this file, see https://docs.djangoproject.com/en/3.2/topics/settings/

For the full list of settings and their values, see https://docs.djangoproject.com/en/3.2/ref/settings/

1.3.4 mapping.urls module

mapping URL Configuration

The *urlpatterns* list routes URLs to views. For more information please see:
https://docs.djangoproject.com/en/3.2/topics/http/urls/

Examples: Function views

1. Add an import: from my_app import views
2. Add a URL to urlpatterns: path(‘’, views.home, name=’home’)

Class-based views

1. Add an import: from other_app.views import Home
2. Add a URL to urlpatterns: path(‘’, Home.as_view(), name=’home’)

Including another URLconf

1. Import the include() function: from django.urls import include, path
2. Add a URL to urlpatterns: path(‘blog/’, include(‘blog.urls’))

1.3.5 mapping.wsgi module

WSGI config for mapping project.

It exposes the WSGI callable as a module-level variable named `application`.

For more information on this file, see https://docs.djangoproject.com/en/3.2/howto/deployment/wsgi/

1.3.6 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX