Task Management System

Release 10/08/2024

Lajward

Oct 08, 2024

CONTENTS

Oct 08, 2024

Welcome to the Task Management System documentation! This documentation provides an overview of the system's functionality and guides you through its features. Add your content using `reStructuredText` syntax. See the reStructuredText documentation for details.

CHAPTER

ONE

AUTHENTICATE PACKAGE

1.1 Subpackages

1.1.1 authenticate.migrations package

Module contents

1.2 Submodules

1.3 authenticate.admin module

1.4 authenticate.apps module

class authenticate.apps.**AuthenticateConfig**(*app_name*, *app_module*)

Bases: AppConfig

__init__(*app_name*, *app_module*)

property _is_default_auto_field_overridden

_path_from_module(*module*)

Attempt to determine app's filesystem path from its module.

classmethod create(*entry*)

Factory that creates an app config from an entry in INSTALLED_APPS.

default_auto_field = 'django.db.models.BigAutoField'

get_model(*model_name*, *require_ready=True*)

Return the model with the given case-insensitive model_name.

Raise LookupError if no model exists with this name.

get_models(*include_auto_created=False*, *include_swapped=False*)

Return an iterable of models.

By default, the following models aren't included:

- auto-created models for many-to-many relations without an explicit intermediate table,
- models that have been swapped out.

Set the corresponding keyword argument to True to include such models. Keyword arguments aren't documented; they're a private API.

import_models()

name = 'authenticate'

ready()

Override this method in subclasses to run code when Django starts.

1.5 authenticate.models module

1.6 authenticate.urls module

This module defines the URL patterns for the authentication views.

GET /login

Maps to the *login* view. Handles user authentication via POST request and renders the login page on GET.

Returns:

- On successful login, redirects to the home page.
- On failure, renders the login form with an error message.

GET /logout/

Maps to the *logout* view. Logs the user out and redirects to the login page.

1.7 authenticate.views module

authenticate.views.**login**(*request*)

Handle user login.

If the request method is POST, it attempts to authenticate the user with the provided username and password. On successful login, the user is redirected to the home page ('tasks'). If authentication fails, an error message is displayed.

Args:

request (HttpRequest): The request object containing metadata about the request.

Returns:

HttpResponse: Renders the login page if the request is GET or if login fails. Redirect: Redirects to the home page on successful login.

authenticate.views.**logout**(*request*)

Handle user logout.

Logs the user out and redirects to the login page.

Args:

request (HttpRequest): The request object containing metadata about the request.

Returns:

Redirect: Redirects to the login page after logout.

1.8 Module contents

CHAPTER

TWO

TASK_MANAGEMENT PACKAGE

The *task_management* package serves as the main entry point for the task management project. It includes all the configurations, routing, and settings necessary for the application to function properly.

2.1 Submodules

2.2 task_management.asgi module

This module handles ASGI configurations for the Django application, enabling support for asynchronous features like WebSockets. ASGI config for task_management 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/4.2/howto/deployment/asgi/

2.3 task_management.consumers module

Contains WebSocket consumers that handle real-time communication for the application, such as notifying users about task updates.

class task_management.consumers.**TaskConsumer**(**args*, ***kwargs*)

Bases: `AsyncWebsocketConsumer`

__init__(**args*, ***kwargs*)

_sync = False

async accept(*subprotocol=None*, *headers=None*)

Accepts an incoming socket

classmethod as_asgi(***initkwargs*)

Return an ASGI v3 single callable that instantiates a consumer instance per scope. Similar in purpose to Django's as_view().

initkwargs will be used to instantiate the consumer instance.

channel_layer_alias = 'default'

async close(*code=None*, *reason=None*)

Closes the WebSocket from the server end

async connect()

async disconnect(*close_code*)

Called when a WebSocket connection is closed.

async dispatch(*message*)

Works out what to do with a message.

groups = None

async receive(*text_data*)

Called with a decoded WebSocket frame.

async send(*text_data=None*, *bytes_data=None*, *close=False*)

Sends a reply back down the WebSocket

async task_message(*event*)

async websocket_connect(*message*)

Called when a WebSocket connection is opened.

async websocket_disconnect(*message*)

Called when a WebSocket connection is closed. Base level so you don't need to call super() all the time.

async websocket_receive(*message*)

Called when a WebSocket frame is received. Decodes it and passes it to receive().

2.4 task_management.routing module

Defines the routing for WebSocket connections, mapping channels to consumers.

2.5 task_management.settings module

This module contains all the project settings for Django, including database configurations, middleware settings, installed applications, and more. Django settings for task_management project.

Generated by 'django-admin startproject' using Django 4.2.11.

For more information on this file, see https://docs.djangoproject.com/en/4.2/topics/settings/

For the full list of settings and their values, see https://docs.djangoproject.com/en/4.2/ref/settings/

2.6 task_management.urls module

This module handles URL routing for the task management project.

The *urlpatterns* list routes URLs to the appropriate views within the application. Below are the main routes configured:

- **Admin Interface**: Accessible at */admin/*, providing admin functionalities for managing the application.
- **Root URL**: Redirects to the login page, accessible at */*, which is set up using *RedirectView*.
- **Authentication URLs**: Routes for authentication views, such as login and logout, handled in the *authenticate* app.

- **Tasks URLs**: Routes for task-related functionalities, such as creating, updating, and deleting tasks, managed in the *tasks* app.
- **OpenAI URLs**: Routes for integrating OpenAI functionalities, handled in the *open_ai* app.

URL configuration for task_management project.

The *urlpatterns* list routes URLs to views. For more information please see:
https://docs.djangoproject.com/en/4.2/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’))

2.7 task_management.wsgi module

This module provides the entry point for WSGI-compatible web servers to serve your project. It is typically used for deployment. WSGI config for task_management 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/4.2/howto/deployment/wsgi/

2.8 Module contents

CHAPTER

THREE

OPEN_AI PACKAGE

The *open_ai* package handles interactions with the OpenAI API and provides views for processing user inputs.

3.1 Subpackages

3.1.1 open_ai.migrations package

Module contents

3.2 Submodules

3.3 open_ai.admin module

This module is responsible for registering the OpenAI models and any other related components in the Django admin interface.

3.4 open_ai.apps module

Contains the configuration for the OpenAI app, including app-specific settings.

class open_ai.apps.**OpenAiConfig**(*app_name*, *app_module*)

Bases: AppConfig

__init__(*app_name*, *app_module*)

property _is_default_auto_field_overridden

_path_from_module(*module*)

Attempt to determine app's filesystem path from its module.

classmethod create(*entry*)

Factory that creates an app config from an entry in INSTALLED_APPS.

default_auto_field = 'django.db.models.BigAutoField'

get_model(*model_name*, *require_ready=True*)

Return the model with the given case-insensitive model_name.

Raise LookupError if no model exists with this name.

get_models(*include_auto_created=False*, *include_swapped=False*)

Return an iterable of models.

By default, the following models aren't included:

- auto-created models for many-to-many relations without an explicit intermediate table,
- models that have been swapped out.

Set the corresponding keyword argument to True to include such models. Keyword arguments aren't documented; they're a private API.

import_models()

name = 'open_ai'

ready()

Override this method in subclasses to run code when Django starts.

3.5 open_ai.ollama_phi module

This module defines the function to interact with the Ollama Phi3 model. It includes methods for sending user input and receiving responses.

open_ai.ollama_phi.**get_phi3_response**(*user_input*)

Interacts with the phi3 model hosted by Ollama to generate a response based on user input.

Args:
user_input (str): The input text from the user.

Returns:
str: The formatted response from the phi3 model. If there is an error, returns an error message.

3.6 open_ai.urls module

This module defines the URL patterns for the OpenAI app views.

GET /chatphi/

GET Maps to the `chatphi` view.
This view handles user input and returns a response from the Ollama Phi3 model.

Returns:

- Renders the *ollama_phi.htm* template with the model's response if the request method is POST.
- Renders the same template for GET requests without a user input.

open_ai.urls.**urlpatterns = [<URLPattern 'chatphi/' [name='chatphi']>]**

URL patterns for the open_ai app.

Routes:

- /chatphi/ : Maps to the *chatphi* view which handles user interactions with the phi3 model.

3.7 open_ai.views module

This module contains views for processing user input and interacting with the OpenAI API.

GET /chatphi/

GET Renders the chat interface for interacting with the Ollama Phi model.

Returns:

- Renders the *ollama_phi.htm* template with the model's response if a POST request is made with user input.
- Renders the same template without a response for GET requests.

open_ai.views.**chatphi**(*request*)

Handles user input for the phi3 model and displays the response.

If the request method is POST, it extracts user input from the request, calls the *get_phi3_response* function to communicate with the phi3 model, and renders the *ollama_phi.htm* template with the model's response.

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

Returns:
HttpResponse: Renders the *ollama_phi.htm* template with the model's response and user input.

3.8 Module contents

CHAPTER

FOUR

TASKS PACKAGE

The *tasks* package handles all functionalities related to task management within the application. This includes task creation, updating, deletion, and representation.

4.1 Subpackages

4.1.1 tasks.migrations package

Submodules

tasks.migrations.0001_initial module

class tasks.migrations.0001_initial.**Migration**(*name*, *app_label*)

Bases: Migration

__init__(*name*, *app_label*)

apply(*project_state*, *schema_editor*, *collect_sql=False*)

Take a project_state representing all migrations prior to this one and a schema_editor for a live database and apply the migration in a forwards order.

Return the resulting project state for efficient reuse by following Migrations.

atomic = True

dependencies = [('auth', '__first__')]

initial = True

mutate_state(*project_state*, *preserve=True*)

Take a ProjectState and return a new one with the migration's operations applied to it. Preserve the original object state by default and return a mutated state from a copy.

operations = [<CreateModel name='Task', fields=[('id',
<django.db.models.fields.BigAutoField>), ('title',
<django.db.models.fields.CharField>), ('description',
<django.db.models.fields.TextField>), ('due_date',
<django.db.models.fields.DateTimeField>), ('status',
<django.db.models.fields.CharField>), ('created_at',
<django.db.models.fields.DateTimeField>), ('assigned_to',
<django.db.models.fields.related.ForeignKey>)]>]

replaces = []

run_before = []

suggest_name()

Suggest a name for the operations this migration might represent. Names are not guaranteed to be unique, but put some effort into the fallback name to avoid VCS conflicts if possible.

unapply(*project_state*, *schema_editor*, *collect_sql=False*)

Take a project_state representing all migrations prior to this one and a schema_editor for a live database and apply the migration in a reverse order.

The backwards migration process consists of two phases:

1. The intermediate states from right before the first until right after the last operation inside this migration are preserved.
2. The operations are applied in reverse order using the states recorded in step 1.

tasks.migrations.0002_alter_task_due_date module

class tasks.migrations.0002_alter_task_due_date.**Migration**(*name*, *app_label*)

Bases: Migration

__init__(*name*, *app_label*)

apply(*project_state*, *schema_editor*, *collect_sql=False*)

Take a project_state representing all migrations prior to this one and a schema_editor for a live database and apply the migration in a forwards order.

Return the resulting project state for efficient reuse by following Migrations.

atomic = True

dependencies = [('tasks', '0001_initial')]

initial = None

mutate_state(*project_state*, *preserve=True*)

Take a ProjectState and return a new one with the migration's operations applied to it. Preserve the original object state by default and return a mutated state from a copy.

operations = [<AlterField model_name='task', name='due_date',
field=<django.db.models.fields.DateTimeField>>]

replaces = []

run_before = []

suggest_name()

Suggest a name for the operations this migration might represent. Names are not guaranteed to be unique, but put some effort into the fallback name to avoid VCS conflicts if possible.

unapply(*project_state*, *schema_editor*, *collect_sql=False*)

Take a project_state representing all migrations prior to this one and a schema_editor for a live database and apply the migration in a reverse order.

The backwards migration process consists of two phases:

1. The intermediate states from right before the first until right after the last operation inside this migration are preserved.
2. The operations are applied in reverse order using the states recorded in step 1.

Module contents

4.2 Submodules

4.3 tasks.admin module

This module contains the admin configuration for managing tasks within the Django admin interface.

4.4 tasks.apps module

This module contains the application configuration for the tasks app.

class tasks.apps.**TasksConfig**(*app_name*, *app_module*)

Bases: AppConfig

__init__(*app_name*, *app_module*)

property _is_default_auto_field_overridden

_path_from_module(*module*)

Attempt to determine app's filesystem path from its module.

classmethod create(*entry*)

Factory that creates an app config from an entry in INSTALLED_APPS.

default_auto_field = 'django.db.models.BigAutoField'

get_model(*model_name*, *require_ready=True*)

Return the model with the given case-insensitive model_name.

Raise LookupError if no model exists with this name.

get_models(*include_auto_created=False*, *include_swapped=False*)

Return an iterable of models.

By default, the following models aren't included:

- auto-created models for many-to-many relations without an explicit intermediate table,
- models that have been swapped out.

Set the corresponding keyword argument to True to include such models. Keyword arguments aren't documented; they're a private API.

import_models()

name = 'tasks'

ready()

Override this method in subclasses to run code when Django starts.

4.5 tasks.models module

This module defines the *Task* model which represents tasks within the application.

class tasks.models.**Task**(**args*, ***kwargs*)

Bases: Model

Represents a task in the task management system.

Attributes:

title (str): The title of the task. description (str, optional): A detailed description of the task. due_date (datetime, optional): The date and time by which the task should be completed. assigned_to (User): The user to whom the task is assigned. status (str): The current status of the task (Todo, Doing, Review, Done). created_at (datetime): The date and time when the task was created.

exception DoesNotExist

Bases: ObjectDoesNotExist

__init__(**args*, ***kwargs*)

add_note()

Exception.add_note(note) – add a note to the exception

args

silent_variable_failure = True

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception MultipleObjectsReturned

Bases: MultipleObjectsReturned

__init__(**args*, ***kwargs*)

add_note()

Exception.add_note(note) – add a note to the exception

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

__init__(**args*, ***kwargs*)

classmethod _check_column_name_clashes()

classmethod _check_constraints(*databases*)

classmethod _check_db_table_comment(*databases*)

classmethod _check_default_pk()

classmethod _check_field_name_clashes()

Forbid field shadowing in multi-table inheritance.

classmethod _check_fields(***kwargs*)

Perform all field checks.

classmethod _check_id_field()

Check if *id* field is a primary key.

classmethod _check_indexes(*databases*)

Check fields, names, and conditions of indexes.

classmethod _check_local_fields(*fields*, *option*)

classmethod _check_long_column_names(*databases*)

Check that any auto-generated column names are shorter than the limits for each database in which the model will be created.

classmethod _check_m2m_through_same_relationship()

Check if no relationship model is used by more than one m2m field.

classmethod _check_managers(*kwargs*)**

Perform all manager checks.

classmethod _check_model()

classmethod _check_model_name_db_lookup_clashes()

classmethod _check_ordering()

Check "ordering" option – is it a list of strings and do all fields exist?

classmethod _check_property_name_related_field_accessor_clashes()

classmethod _check_single_primary_key()

classmethod _check_swappable()

Check if the swapped model exists.

classmethod _check_unique_together()

Check the value of "unique_together" option.

_do_insert(*manager*, *using*, *fields*, *returning_fields*, *raw*)

Do an INSERT. If returning_fields is defined then this method should return the newly created data for the model.

_do_update(*base_qs*, *using*, *pk_val*, *values*, *update_fields*, *forced_update*)

Try to update the model. Return True if the model was updated (if an update query was done and a matching row was found in the DB).

_get_FIELD_display(*field*)

classmethod _get_expr_references(*expr*)

_get_field_expression_map(*meta*, *exclude=None*)

_get_next_or_previous_by_FIELD(*field*, *is_next*, ***kwargs*)

_get_next_or_previous_in_order(*is_next*)

_get_pk_val(*meta=None*)

_get_unique_checks(*exclude=None*, *include_meta_constraints=False*)

Return a list of checks to perform. Since validate_unique() could be called from a ModelForm, some fields may have been excluded; we can't perform a unique check on a model that is missing fields involved in that check. Fields that did not validate should also be excluded, but they need to be passed in via the exclude argument.

_meta = <Options for Task>

_parse_params(**args*, *method_name*, ***kwargs*)

_perform_date_checks(*date_checks*)

_perform_unique_checks(*unique_checks*)

_prepare_related_fields_for_save(*operation_name*, *fields=None*)

_save_parents(*cls*, *using*, *update_fields*, *force_insert*, *updated_parents=None*)

Save all the parents of cls using values from self.

_save_table(*raw=False*, *cls=None*, *force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)

Do the heavy-lifting involved in saving. Update or insert the data for a single table.

_set_pk_val(*value*)

classmethod _validate_force_insert(*force_insert*)

async adelete(*using=None*, *keep_parents=False*)

async arefresh_from_db(*using=None*, *fields=None*, *from_queryset=None*)

async asave(**args*, *force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)

assigned_to

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.

assigned_to_id

classmethod check(***kwargs*)

clean()

Hook for doing any extra model-wide validation after clean() has been called on every field by self.clean_fields. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field defined by NON_FIELD_ERRORS.

clean_fields(*exclude=None*)

Clean all fields and raise a ValidationError containing a dict of all validation errors if any occur.

created_at

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

date_error_message(*lookup_type*, *field_name*, *unique_for*)

delete(*using=None*, *keep_parents=False*)

description

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

due_date

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

classmethod from_db(*db*, *field_names*, *values*)

full_clean(*exclude=None*, *validate_unique=True*, *validate_constraints=True*)

Call clean_fields(), clean(), validate_unique(), and validate_constraints() on the model. Raise a ValidationError for any errors that occur.

get_constraints()

get_deferred_fields()

Return a set containing names of deferred fields for this instance.

get_next_by_created_at(*, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=True*, ***kwargs*)

get_previous_by_created_at(*, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=False*, ***kwargs*)

get_status_display(*, *field=<django.db.models.fields.CharField: status>*)

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>

property pk

prepare_database_save(*field*)

refresh_from_db(*using=None*, *fields=None*, *from_queryset=None*)

Reload field values from the database.

By default, the reloading happens from the database this instance was loaded from, or by the read router if this instance wasn't loaded from any database. The using parameter will override the default.

Fields can be used to specify which fields to reload. The fields should be an iterable of field attnames. If fields is None, then all non-deferred fields are reloaded.

When accessing deferred fields of an instance, the deferred loading of the field will call this method.

save(**args*, *force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)

Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

save_base(*raw=False*, *force_insert=False*, *force_update=False*, *using=None*, *update_fields=None*)

Handle the parts of saving which should be done only once per save, yet need to be done in raw saves, too. This includes some sanity checks and signal sending.

The 'raw' argument is telling save_base not to save any parent models and not to do any changes to the values before save. This is used by fixture loading.

serializable_value(*field_name*)

Return the value of the field name for this instance. If the field is a foreign key, return the id value instead of the object. If there's no Field object with this name on the model, return the model attribute's value.

Used to serialize a field's value (in the serializer, or form output, for example). Normally, you would just access the attribute directly and not use this method.

status

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

title

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

unique_error_message(*model_class*, *unique_check*)

validate_constraints(*exclude=None*)

validate_unique(*exclude=None*)

Check unique constraints on the model and raise ValidationError if any failed.

4.6 tasks.serializers module

This module contains serializers for converting *Task* model instances to JSON format and vice versa.

class tasks.serializers.**TaskSerializer**(**args*, ***kwargs*)

Bases: ModelSerializer

Serializer for the Task model.

This serializer is responsible for converting Task instances to JSON format and validating incoming data for Task creation and updates.

Meta:
model (Task): The Task model to serialize. fields (list): List of fields to include in the serialization.

class Meta

Bases: object

fields = ['id', 'title', 'description', 'due_date', 'assigned_to', 'status', 'created_at']

model

alias of *Task*

__init__(*instance=None*, *data=<class 'rest_framework.fields.empty'>*, ***kwargs*)

_creation_counter = 3

_declared_fields = {}

_get_model_fields(*field_names*, *declared_fields*, *extra_kwargs*)

Returns all the model fields that are being mapped to by fields on the serializer class. Returned as a dict of 'model field name' -> 'model field'. Used internally by *get_uniqueness_field_options*.

_read_only_defaults()

property _readable_fields

property _writable_fields

bind(*field_name*, *parent*)

Initializes the field name and parent for the field instance. Called when a field is added to the parent serializer instance.

build_field(*field_name*, *info*, *model_class*, *nested_depth*)

Return a two tuple of (cls, kwargs) to build a serializer field with.

build_nested_field(*field_name*, *relation_info*, *nested_depth*)

Create nested fields for forward and reverse relationships.

build_property_field(*field_name*, *model_class*)

Create a read only field for model methods and properties.

build_relational_field(*field_name*, *relation_info*)

Create fields for forward and reverse relationships.

build_standard_field(*field_name*, *model_field*)

Create regular model fields.

build_unknown_field(*field_name*, *model_class*)

Raise an error on any unknown fields.

build_url_field(*field_name*, *model_class*)

Create a field representing the object's own URL.

property context

Returns the context as passed to the root serializer on initialization.

create(*validated_data*)

We have a bit of extra checking around this in order to provide descriptive messages when something goes wrong, but this method is essentially just:

return ExampleModel.objects.create(**validated_data)

If there are many to many fields present on the instance then they cannot be set until the model is instantiated, in which case the implementation is like so:

example_relationship = validated_data.pop('example_relationship') instance = ExampleModel.objects.create(**validated_data) instance.example_relationship = example_relationship return instance

The default implementation also does not handle nested relationships. If you want to support writable nested relationships you'll need to write an explicit *.create()* method.

property data

default_empty_html

alias of `empty`

default_error_messages = {'invalid': 'Invalid data. Expected a dictionary, but got {datatype}.'}

default_validators = []

property errors

fail(*key*, ***kwargs*)

A helper method that simply raises a validation error.

fields

A dictionary of {field_name: field_instance}.

get_attribute(*instance*)

Given the *outgoing* object instance, return the primitive value that should be used for this field.

get_default()

Return the default value to use when validating data if no input is provided for this field.

If a default has not been set for this field then this will simply raise *SkipField*, indicating that no value should be set in the validated data for this field.

get_default_field_names(*declared_fields*, *model_info*)

Return the default list of field names that will be used if the *Meta.fields* option is not specified.

get_extra_kwargs()

Return a dictionary mapping field names to a dictionary of additional keyword arguments.

get_field_names(*declared_fields*, *info*)

Returns the list of all field names that should be created when instantiating this serializer class. This is based on the default set of fields, but also takes into account the *Meta.fields* or *Meta.exclude* options if they have been specified.

get_fields()

Return the dict of field names -> field instances that should be used for *self.fields* when instantiating the serializer.

get_initial()

Return a value to use when the field is being returned as a primitive value, without any object instance.

get_unique_for_date_validators()

Determine a default set of validators for the following constraints:

- unique_for_date
- unique_for_month
- unique_for_year

get_unique_together_constraints(*model*)

Returns iterator of (fields, queryset), each entry describes an unique together constraint on *fields* in *queryset*.

get_unique_together_validators()

Determine a default set of validators for any unique_together constraints.

get_uniqueness_extra_kwargs(*field_names*, *declared_fields*, *extra_kwargs*)

Return any additional field options that need to be included as a result of uniqueness constraints on the model. This is returned as a two-tuple of:

('dict of updated extra kwargs', 'mapping of hidden fields')

get_validators()

Determine the set of validators to use when instantiating serializer.

get_value(*dictionary*)

Given the *incoming* primitive data, return the value for this field that should be validated and transformed to a native value.

include_extra_kwargs(*kwargs*, *extra_kwargs*)

Include any 'extra_kwargs' that have been included for this field, possibly removing any incompatible existing keyword arguments.

initial = None

is_valid(*, *raise_exception=False*)

classmethod many_init(**args*, ***kwargs*)

This method implements the creation of a *ListSerializer* parent class when *many=True* is used. You can customize it if you need to control which keyword arguments are passed to the parent, and which are passed to the child.

Note that we're over-cautious in passing most arguments to both parent and child classes in order to try to cover the general case. If you're overriding this method you'll probably want something much simpler, eg:

@classmethod def many_init(cls, *args, **kwargs):

kwargs['child'] = cls() return CustomListSerializer(*args, **kwargs)

property root

Returns the top-level serializer for this field.

run_validation(*data=<class 'rest_framework.fields.empty'>*)

We override the default *run_validation*, because the validation performed by validators and the *.validate()* method should be coerced into an error dictionary with a 'non_fields_error' key.

run_validators(*value*)

Add read_only fields with defaults to value before running validators.

save(***kwargs*)

serializer_choice_field

alias of `ChoiceField`

```
serializer_field_mapping = {<class
'django.contrib.postgres.fields.array.ArrayField'>:  <class
'rest_framework.fields.ListField'>, <class
'django.contrib.postgres.fields.hstore.HStoreField'>:  <class
'rest_framework.fields.HStoreField'>, <class
'django.contrib.postgres.fields.jsonb.JSONField'>:  <class
'rest_framework.fields.JSONField'>, <class 'django.db.models.fields.AutoField'>:
<class 'rest_framework.fields.IntegerField'>, <class
'django.db.models.fields.BigIntegerField'>:  <class
'rest_framework.fields.IntegerField'>, <class
'django.db.models.fields.BooleanField'>:  <class
'rest_framework.fields.BooleanField'>, <class 'django.db.models.fields.CharField'>:
<class 'rest_framework.fields.CharField'>, <class
'django.db.models.fields.CommaSeparatedIntegerField'>:  <class
'rest_framework.fields.CharField'>, <class 'django.db.models.fields.DateField'>:
<class 'rest_framework.fields.DateField'>, <class
'django.db.models.fields.DateTimeField'>:  <class
'rest_framework.fields.DateTimeField'>, <class
'django.db.models.fields.DecimalField'>:  <class
'rest_framework.fields.DecimalField'>, <class
'django.db.models.fields.DurationField'>:  <class
'rest_framework.fields.DurationField'>, <class
'django.db.models.fields.EmailField'>:  <class 'rest_framework.fields.EmailField'>,
<class 'django.db.models.fields.Field'>:  <class
'rest_framework.fields.ModelField'>, <class
'django.db.models.fields.FilePathField'>:  <class
'rest_framework.fields.FilePathField'>, <class
'django.db.models.fields.FloatField'>:  <class 'rest_framework.fields.FloatField'>,
<class 'django.db.models.fields.GenericIPAddressField'>:  <class
'rest_framework.fields.IPAddressField'>, <class
'django.db.models.fields.IntegerField'>:  <class
'rest_framework.fields.IntegerField'>, <class
'django.db.models.fields.NullBooleanField'>:  <class
'rest_framework.fields.BooleanField'>, <class
'django.db.models.fields.PositiveIntegerField'>:  <class
'rest_framework.fields.IntegerField'>, <class
'django.db.models.fields.PositiveSmallIntegerField'>:  <class
'rest_framework.fields.IntegerField'>, <class 'django.db.models.fields.SlugField'>:
<class 'rest_framework.fields.SlugField'>, <class
'django.db.models.fields.SmallIntegerField'>:  <class
'rest_framework.fields.IntegerField'>, <class 'django.db.models.fields.TextField'>:
<class 'rest_framework.fields.CharField'>, <class
'django.db.models.fields.TimeField'>:  <class 'rest_framework.fields.TimeField'>,
<class 'django.db.models.fields.URLField'>:  <class
'rest_framework.fields.URLField'>, <class 'django.db.models.fields.UUIDField'>:
<class 'rest_framework.fields.UUIDField'>, <class
'django.db.models.fields.files.FileField'>:  <class
'rest_framework.fields.FileField'>, <class
'django.db.models.fields.files.ImageField'>:  <class
'rest_framework.fields.ImageField'>, <class
'django.db.models.fields.json.JSONField'>:  <class
'rest_framework.fields.JSONField'>}

serializer_related_field
```

alias of PrimaryKeyRelatedField

serializer_related_to_field

alias of SlugRelatedField

serializer_url_field

alias of HyperlinkedIdentityField

set_value(*dictionary*, *keys*, *value*)

Similar to Python's built in *dictionary[key] = value*, but takes a list of nested keys instead of a single key.

set_value({'a': 1}, [], {'b': 2}) -> {'a': 1, 'b': 2} set_value({'a': 1}, ['x'], 2) -> {'a': 1, 'x': 2} set_value({'a': 1}, ['x', 'y'], 2) -> {'a': 1, 'x': {'y': 2}}

to_internal_value(*data*)

Dict of native values <- Dict of primitive datatypes.

to_representation(*instance*)

Object instance -> Dict of primitive datatypes.

update(*instance*, *validated_data*)

url_field_name = None

validate(*attrs*)

validate_empty_values(*data*)

Validate empty values, and either:

- Raise *ValidationError*, indicating invalid data.
- Raise *SkipField*, indicating that the field should be ignored.
- Return (True, data), indicating an empty value that should be returned without any further validation being applied.
- Return (False, data), indicating a non-empty value, that should have validation applied as normal.

property validated_data

property validators

4.7 tasks.urls module

This module defines the URL patterns for the tasks app.

The *urlpatterns* list routes URLs to views. Below are the main routes configured:

- **/**: Maps to the *index* view which lists all tasks.
- **/create_task/**: Maps to the *create_task* view for creating new tasks.
- **/update_task/<int:id>**: Maps to the *update_task* view for editing existing tasks.
- **/delete_task/<int:id>**: Maps to the *delete_task* view for removing tasks.

tasks.urls.**urlpatterns = [<URLPattern '' [name='tasks']>, <URLPattern 'create_task/' [name='create_task']>, <URLPattern 'update_task/<int:id>' [name='update_task']>, <URLPattern 'delete_task/<int:id>' [name='delete_task']>]**

URL patterns for the tasks app.

Routes:

- / : Maps to the *index* view which lists all tasks.
- /create_task/ : Maps to the *create_task* view for creating new tasks.
- /update_task/<int:id> : Maps to the *update_task* view for editing existing tasks.
- /delete_task/<int:id> : Maps to the *delete_task* view for removing tasks.

4.8 tasks.views module

This module contains the view functions for handling requests related to tasks.

tasks.views.**create_task**(*request*)

Handles the creation of new tasks.

If the request method is POST, validates and saves the new task. It also invalidates the cache and notifies WebSocket clients about the new task.

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

Returns:

HttpResponse: Redirects to the tasks index page on success, or re-renders
the create task form with error messages on failure.

tasks.views.**delete_task**(*request*, *id*)

Handles the deletion of a task.

Checks if the user has permission to delete the task. If the task is found, it deletes the task and invalidates the cache. If the task does not exist, or the user lacks permissions, it returns an appropriate error message.

Args:
request (HttpRequest): The HTTP request object. id (int): The ID of the task to be deleted.

Returns:

HttpResponse: Redirects to the tasks index page after deletion or
shows an error message if deletion fails.

tasks.views.**index**(*request*)

Displays the list of tasks.

If the request method is GET, retrieves tasks from the cache or database and renders the index.html template with the tasks data.

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

Returns:
HttpResponse: Renders the index.html template with the tasks.

tasks.views.**update_task**(*request*, *id*)

Handles the update of an existing task.

Retrieves the task by ID and renders the update form. If the request method is POST, validates and saves the updated task data. Informs WebSocket clients about the task update.

Args:
request (HttpRequest): The HTTP request object. id (int): The ID of the task to be updated.

Returns:

HttpResponse: Redirects to the tasks index page on successful update,
or re-renders the update form with error messages on failure.

4.9 Module contents

CHAPTER

FIVE

TEMPLATES

This section includes the HTML templates used in the authenticate app.

5.1 create.html

The *create.html* template is used for creating a new object.

5.2 update.html

The *update.html* template is used for updating an object.

5.3 index.html

The *index.html* template is the main index page.

5.4 login.html

The *login.html* template is used for user login.

5.5 ollama_phi.html

The *ollama_phi.html* template is used for displaying the Ollama Phi content.

this is in case we have multiple modules in our code

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

/Maps to the `chatphi` view. This view handles user input and returns a response from the Ollama Phi3 model.

GET Maps to the `chatphi` view. This view handles user input and returns a response from the Ollama Phi3 model., 12

/Renders the chat interface for interacting with the Ollama Phi model.

GET Renders the chat interface for interacting with the Ollama Phi model., 13

/chatphi

GET /chatphi/, 13

/login

GET /login, 4

/logout

GET /logout/, 4

PYTHON MODULE INDEX

a

o

t

INDEX

Symbols

A

B

C

L

M

N

O

P

R

W