routers.py

# Routers

> *Resource routing allows you to quickly declare all of the common routes for a given resourceful controller. Instead of declaring separate routes for your index... a resourceful route declares them in a single line of code.*
>
> — *Ruby on Rails Documentation*

Some Web frameworks such as Rails provide functionality for automatically determining how the URLs for an application should be mapped to the logic that deals with handling incoming requests.

REST framework adds support for automatic URL routing to Django, and provides you with a simple, quick and consistent way of wiring your view logic to a set of URLs.

## Usage

Here's an example of a simple URL conf, that uses `SimpleRouter`.

```
from rest_framework import routers

router = routers.SimpleRouter()
router.register(r'users', UserViewSet)
router.register(r'accounts', AccountViewSet)
urlpatterns = router.urls
```

There are two mandatory arguments to the `register()` method:

- `prefix` - The URL prefix to use for this set of routes.
- `viewset` - The viewset class.

Optionally, you may also specify an additional argument:

- `basename` - The base to use for the URL names that are created. If unset the basename will be automatically generated based on the `queryset` attribute of the viewset, if it has one. Note that if the viewset does not include a `queryset` attribute then you must set `basename` when registering the viewset.

The example above would generate the following URL patterns:

- URL pattern: `^users/$` Name: `'user-list'`
- URL pattern: `^users/{pk}/$` Name: `'user-detail'`
- URL pattern: `^accounts/$` Name: `'account-list'`
- URL pattern: `^accounts/{pk}/$` Name: `'account-detail'`

---

**Note**: The `basename` argument is used to specify the initial part of the view name pattern. In the example above, that's the `user` or `account` part.

Typically you won't *need* to specify the `basename` argument, but if you have a viewset where you've defined a custom `get_queryset` method, then the viewset may not have a `.queryset` attribute set. If you try to register that viewset you'll see an error like this:

```
'basename' argument not specified, and could not automatically determine the name from the viewset, as it does no
```

This means you'll need to explicitly set the `basename` argument when registering the viewset, as it could not be automatically determined from the model name.

---

## Using `include` with routers

The `.urls` attribute on a router instance is simply a standard list of URL patterns. There are a number of different styles for how you can include these URLs.

For example, you can append `router.urls` to a list of existing views...

```
router = routers.SimpleRouter()
router.register(r'users', UserViewSet)
router.register(r'accounts', AccountViewSet)

urlpatterns = [
    path('forgot-password/', ForgotPasswordFormView.as_view()),
]

urlpatterns += router.urls
```

Alternatively you can use Django's `include` function, like so...

```
urlpatterns = [
    path('forgot-password', ForgotPasswordFormView.as_view()),
    path('', include(router.urls)),
]
```

You may use `include` with an application namespace:

```
urlpatterns = [
    path('forgot-password/', ForgotPasswordFormView.as_view()),
    path('api/', include((router.urls, 'app_name'))),
]
```

Or both an application and instance namespace:

```
urlpatterns = [
    path('forgot-password/', ForgotPasswordFormView.as_view()),
    path('api/', include((router.urls, 'app_name'), namespace='instance_name')),
]
```

See Django's URL namespaces docs and the `include` API reference for more details.

---

**Note**: If using namespacing with hyperlinked serializers you'll also need to ensure that any `view_name` parameters on the serializers correctly reflect the namespace. In the examples above you'd need to include a parameter such as `view_name='app_name:user-detail'` for serializer fields hyperlinked to the user detail view.

The automatic `view_name` generation uses a pattern like `%(model_name)-detail`. Unless your models names actually clash you may be better off **not** namespacing your Django REST Framework views when using hyperlinked serializers.

---

### Routing for extra actions

A viewset may mark extra actions for routing by decorating a method with the `@action` decorator. These extra actions will be included in the generated routes. For example, given the `set_password` method on the `UserViewSet` class:

```
from myapp.permissions import IsAdminOrIsSelf
from rest_framework.decorators import action

class UserViewSet(ModelViewSet):
    ...

    @action(methods=['post'], detail=True, permission_classes=[IsAdminOrIsSelf])
    def set_password(self, request, pk=None):
        ...
```

The following route would be generated:

- URL pattern: `^users/{pk}/set_password/$`
- URL name: `'user-set-password'`

By default, the URL pattern is based on the method name, and the URL name is the combination of the `ViewSet.basename` and the hyphenated method name. If you don't want to use the defaults for either of these values, you can instead provide the `url_path` and `url_name` arguments to the `@action` decorator.

For example, if you want to change the URL for our custom action to `^users/{pk}/change-password/$`, you could write:

```
from myapp.permissions import IsAdminOrIsSelf
from rest_framework.decorators import action

class UserViewSet(ModelViewSet):
    ...

    @action(methods=['post'], detail=True, permission_classes=[IsAdminOrIsSelf],
            url_path='change-password', url_name='change_password')
    def set_password(self, request, pk=None):
        ...
```

The above example would now generate the following URL pattern:

- URL path: `^users/{pk}/change-password/$`
- URL name: `'user-change_password'`

## API Guide

### SimpleRouter

This router includes routes for the standard set of `list`, `create`, `retrieve`, `update`, `partial_update` and `destroy` actions. The viewset can also mark additional methods to be routed, using the `@action` decorator.

| URL Style | HTTP Method | Action | URL Name |
|---|---|---|---|
| {prefix}/ | GET | list | {basename}-list |
| | POST | create | |
| {prefix}/{url_path}/ | GET, or as specified by \`methods\` argument | \`@action(detail=False)\` decorated method | {basename}-{url_name} |
| {prefix}/{lookup}/ | GET | retrieve | {basename}-detail |
| | PUT | update | |
| | PATCH | partial_update | |
| | DELETE | destroy | |
| {prefix}/{lookup}/{url_path}/ | GET, or as specified by \`methods\` argument | \`@action(detail=True)\` decorated method | {basename}-{url_name} |

By default the URLs created by `SimpleRouter` are appended with a trailing slash. This behavior can be modified by setting the `trailing_slash` argument to `False` when instantiating the router. For example:

```
router = SimpleRouter(trailing_slash=False)
```

Trailing slashes are conventional in Django, but are not used by default in some other frameworks such as Rails. Which style you choose to use is largely a matter of preference, although some javascript frameworks may expect a particular routing style.

The router will match lookup values containing any characters except slashes and period characters. For a more restrictive (or lenient) lookup pattern, set the `lookup_value_regex` attribute on the viewset. For example, you can limit the lookup to valid UUIDs:

```
class MyModelViewSet(mixins.RetrieveModelMixin, viewsets.GenericViewSet):
    lookup_field = 'my_model_id'
    lookup_value_regex = '[0-9a-f]{32}'
```

### DefaultRouter

This router is similar to `SimpleRouter` as above, but additionally includes a default API root view, that returns a response containing hyperlinks to all the list views. It also generates routes for optional `.json` style format suffixes.

| URL Style | HTTP Method | Action | URL Name |
|---|---|---|---|
| [.format] | GET | automatically generated root view | api-root |
| {prefix}/[.format] | GET | list | {basename}-list |
| | POST | create | |
| {prefix}/{url_path}/[.format] | GET, or as specified by \`methods\` argument | \`@action(detail=False)\` decorated method | {basename}-{url_name} |
| {prefix}/{lookup}/[.format] | GET | retrieve | {basename}-detail |
| | PUT | update | |
| | PATCH | partial_update | |
| | DELETE | destroy | |
| {prefix}/{lookup}/{url_path}/[.format] | GET, or as specified by \`methods\` argument | \`@action(detail=True)\` decorated method | {basename}-{url_name} |

As with `SimpleRouter` the trailing slashes on the URL routes can be removed by setting the `trailing_slash` argument to `False` when instantiating the router.

```
router = DefaultRouter(trailing_slash=False)
```

## Custom Routers

Implementing a custom router isn't something you'd need to do very often, but it can be useful if you have specific requirements about how the URLs for your API are structured. Doing so allows you to encapsulate the URL structure in a reusable way that ensures you don't have to write your URL patterns explicitly for each new view.

The simplest way to implement a custom router is to subclass one of the existing router classes. The `.routes` attribute is used to template the URL patterns that will be mapped to each viewset. The `.routes` attribute is a list of `Route` named tuples.

The arguments to the `Route` named tuple are:

**url**: A string representing the URL to be routed. May include the following format strings:

- `{prefix}` - The URL prefix to use for this set of routes.
- `{lookup}` - The lookup field used to match against a single instance.
- `{trailing_slash}` - Either a '/' or an empty string, depending on the `trailing_slash` argument.

**mapping**: A mapping of HTTP method names to the view methods

**name**: The name of the URL as used in `reverse` calls. May include the following format string:

- `{basename}` - The base to use for the URL names that are created.

**initkwargs**: A dictionary of any additional arguments that should be passed when instantiating the view. Note that the `detail`, `basename`, and `suffix` arguments are reserved for viewset introspection and are also used by the browsable API to generate the view name and breadcrumb links.

## Customizing dynamic routes

You can also customize how the `@action` decorator is routed. Include the `DynamicRoute` named tuple in the `.routes` list, setting the `detail` argument as appropriate for the list-based and detail-based routes. In addition to `detail`, the arguments to `DynamicRoute` are:

**url**: A string representing the URL to be routed. May include the same format strings as `Route`, and additionally accepts the `{url_path}` format string.

**name**: The name of the URL as used in `reverse` calls. May include the following format strings:

- `{basename}` - The base to use for the URL names that are created.
- `{url_name}` - The `url_name` provided to the `@action`.

**initkwargs**: A dictionary of any additional arguments that should be passed when instantiating the view.

## Example

The following example will only route to the `list` and `retrieve` actions, and does not use the trailing slash convention.

```
from rest_framework.routers import Route, DynamicRoute, SimpleRouter

class CustomReadOnlyRouter(SimpleRouter):
    """
    A router for read-only APIs, which doesn't use trailing slashes.
    """
    routes = [
        Route(
            url=r'^{prefix}$',
            mapping={'get': 'list'},
            name='{basename}-list',
            detail=False,
            initkwargs={'suffix': 'List'}
        ),
        Route(
            url=r'^{prefix}/{lookup}$',
            mapping={'get': 'retrieve'},
            name='{basename}-detail',
            detail=True,
            initkwargs={'suffix': 'Detail'}
        ),
        DynamicRoute(
            url=r'^{prefix}/{lookup}/{url_path}$',
            name='{basename}-{url_name}',
            detail=True,
            initkwargs={}
        )
    ]
```

Let's take a look at the routes our `CustomReadOnlyRouter` would generate for a simple viewset.

`views.py`:

```
class UserViewSet(viewsets.ReadOnlyModelViewSet):
    """
    A viewset that provides the standard actions
    """
    queryset = User.objects.all()
    serializer_class = UserSerializer
    lookup_field = 'username'

    @action(detail=True)
    def group_names(self, request, pk=None):
        """
        Returns a list of all the group names that the given
        user belongs to.
        """
        user = self.get_object()
        groups = user.groups.all()
        return Response([group.name for group in groups])
```

`urls.py`:

```
router = CustomReadOnlyRouter()
router.register('users', UserViewSet)
urlpatterns = router.urls
```

The following mappings would be generated...

| URL | HTTP Method | Action | URL Name |
|---|---|---|---|
| /users | GET | list | user-list |
| /users/{username} | GET | retrieve | user-detail |
| /users/{username}/group_names | GET | group_names | user-group-names |

For another example of setting the `.routes` attribute, see the source code for the `SimpleRouter` class.

## Advanced custom routers

If you want to provide totally custom behavior, you can override `BaseRouter` and override the `get_urls(self)` method. The method should inspect the registered viewsets and return a list of URL patterns. The registered prefix, viewset and basename tuples may be inspected by accessing the `self.registry` attribute.

You may also want to override the `get_default_basename(self, viewset)` method, or else always explicitly set the `basename` argument when registering your viewsets with the router.

# Third Party Packages

The following third party packages are also available.

## DRF Nested Routers

The drf-nested-routers package provides routers and relationship fields for working with nested resources.

## ModelRouter (wq.db.rest)

The wq.db package provides an advanced ModelRouter class (and singleton instance) that extends `DefaultRouter` with a `register_model()` API. Much like Django's `admin.site.register`, the only required argument to `rest.router.register_model` is a model class. Reasonable defaults for a url prefix, serializer, and viewset will be inferred from the model and global configuration.

```
from wq.db import rest
from myapp.models import MyModel

rest.router.register_model(MyModel)
```

## DRF-extensions

The `DRF-extensions` package provides routers for creating nested viewsets, collection level controllers with customizable endpoint names.