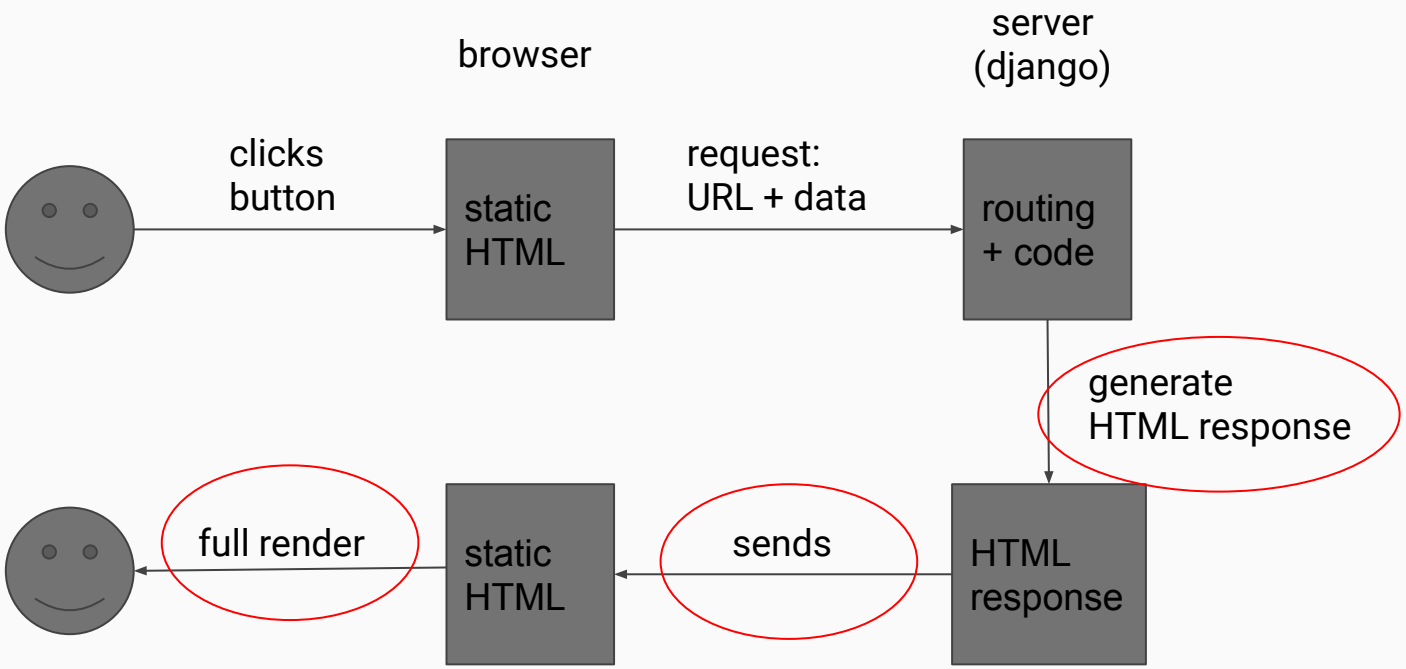
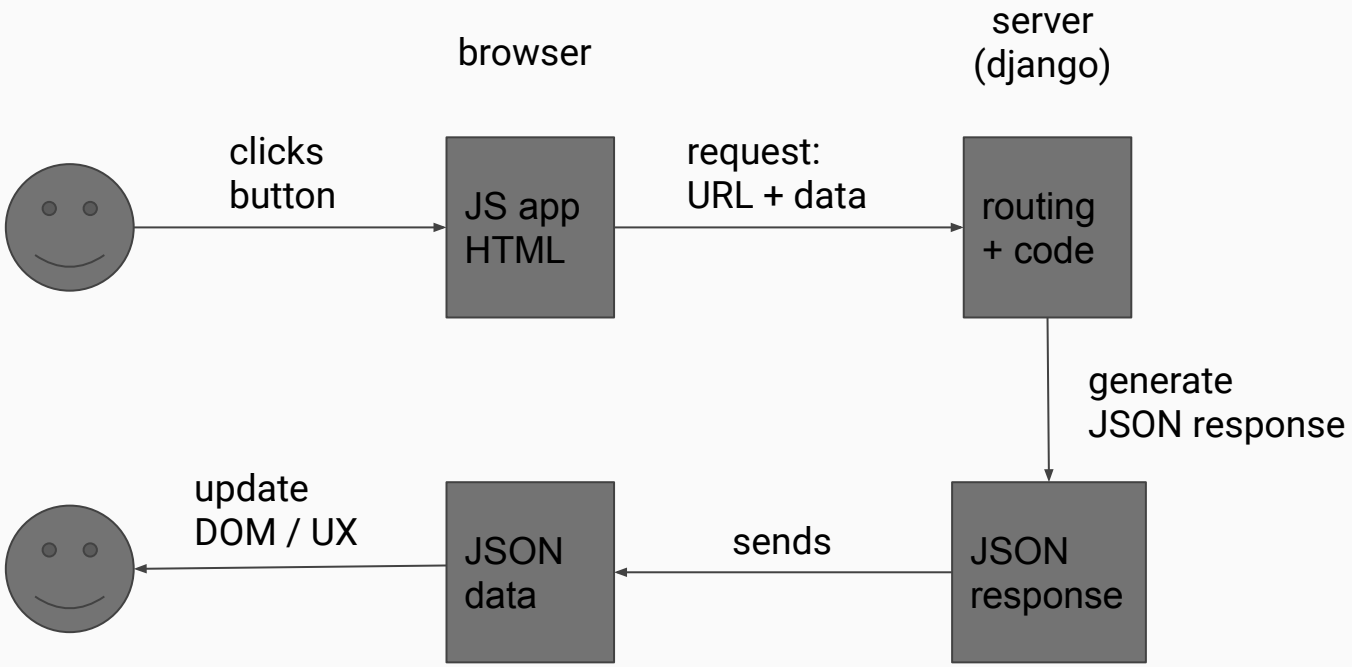
Cooking Recipes API with Django Rest Framework

Israel Saeta Pérez

Static web app



Dynamic Single Page Application



Static vs. SPA webapps

	Static	Single Page Application
Interaction	Every interaction full new HTML document	Page changes without reloading
HTML rendering	Server	Client (Javascript)
Server input	multipart/form-data or form-urlencoded	JSON/XML
Server output	HTML	JSON/XML

What our API will look like (REST)

URL	Method	Action
/recipes/	GET	List/search all recipes
/recipes/	POST	Create recipe
/recipes/:id/	GET	Retrieve recipe
/recipes/:id/	PUT	Replace recipe
/recipes/:id/	PATCH	Update recipe
/recipes/:id/	DELETE	Delete recipe

Example JSON response when GET /recipes/1/

```
{
	“id”: 1,
	“name”: “Pizza”
	“description”: “Put it in the oven”,
	“ingredients”: [{“name”: “dough”}, {“name”: “cheese”}, {“name”: “tomato”}]
}
```

Example recipe creation

```
POST /recipes/
{
      “name”: “Pizza”
      “description”: “Put it in the oven”,
      “ingredients”: [{“name”: “dough”}, {“name”: “cheese”}, {“name”: “tomato”}]
}

Response:
{
      “id”: 1,
      “name”: “Pizza”
      “description”: “Put it in the oven”,
      “ingredients”: [{“name”: “dough”}, {“name”: “cheese”}, {“name”: “tomato”}]
}
```

Example recipe list

```
GET /recipes/
[
    {
      “name”: “Pizza”
      “description”: “Put it in the oven”,
      “ingredients”: [{“name”: “dough”}, {“name”: “cheese”}, {“name”: “tomato”}]
    }
]
```

```
Add search view by name substring:
GET /recipes/?name=Pi
[
    {
      “name”: “Pizza”
      “description”: “Put it in the oven”,
      “ingredients”: [{“name”: “dough”}, {“name”: “cheese”}, {“name”: “tomato”}]
    }
]
```

Example recipe edit

```
PATCH /recipes/1/
    {
      “name”: “Pizza”
      “description”: “Put it in the oven”,
      “ingredients”: [{“name”: “casa-tarradellas”}]
    }
```

Should delete the previous existing ingredients and put “casa-tarradellas” as only ingredient for recipe.

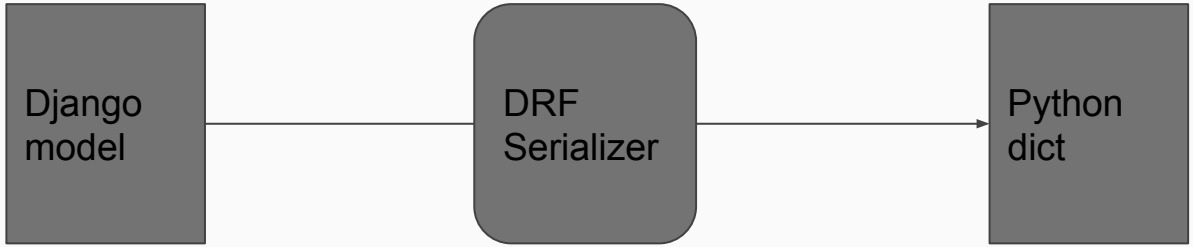
```
Response:
{
      “id”: 1,
      “name”: “Pizza”
      “description”: “Put it in the oven”,
      “ingredients”: [{“name”: “casa-tarradellas”}]
}
```

https://github.com/dukebody/recipes-drf-pycones19
and create a recipe with ingredients

Each step finished state is a branch (stepX)

Tutorial follows same structure as DRF official one

DRF model serializers




```
# serializers.py

from rest_framework import serializers
from myapp.models import Snippet


class SnippetSerializer(serializers.ModelSerializer):
    class Meta:
        model = Snippet
        fields = ['id', 'title', 'code', 'linenos', 'language', 'style']
```

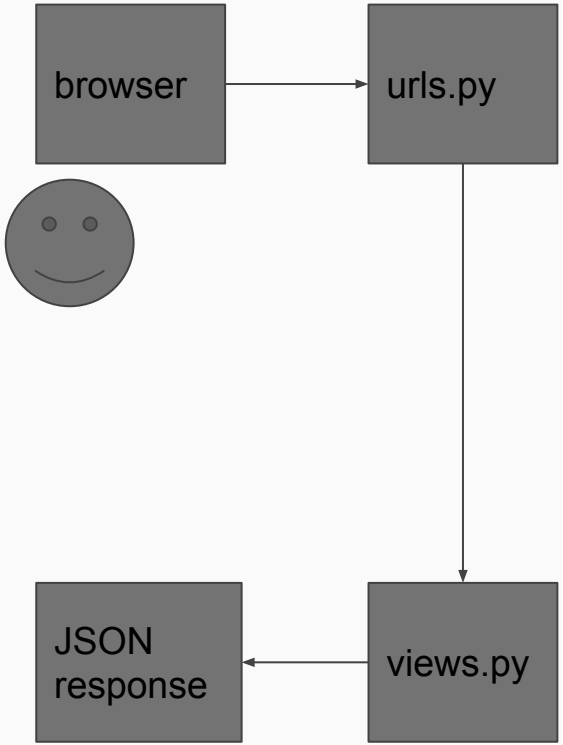
Exercise - Step 1: Create basic RecipeSerializer

- Create a **RecipeSerializer(ModelSerializer) class** that includes the Recipe fields “id”, “name” and “description” into the file tutorial/recipes/serializers.py

How to verify:

```
$ python manage.py shell

from tutorial.recipes.models import Recipe
from tutorial.recipes.serializers import RecipeSerializer
recipe = Recipe.objects.first()
serializer = RecipeSerializer(recipe)
serializer.data

# {'id': 1, 'name': 'Pizza', 'description': 'Put the pizza in the oven'}
```

Routing and view




```
# urls.py
from django.urls import path
from snippets import views

urlpatterns = [
    path('snippets/<int:pk>/',views.snippet_detail),
]
```

```
# views.py
from django.http import HttpResponse, JsonResponse
from rest_framework.parsers import JSONParser
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer

def snippet_detail(request, pk):
    try:
        snippet = Snippet.objects.get(pk=pk)
    except Snippet.DoesNotExist:
        return HttpResponse(status=404)

    serializer = SnippetSerializer(snippet)
    return JsonResponse(serializer.data)
```

Exercise - Step 2: Create detail view and route

- Create a **recipe_detail view function** in tutorial/recipes/views.py that renders the representation of the recipe with the given id
- Route /recipes/:id/ to the previous view adding a url pattern in tutorial/recipes/urls.py

How to verify:

1. python manage.py runserver
2. Visit /recipes/:id/ (where id is a valid recipe id) in your browser. You should see the recipe representation in JSON.

List view

```
# views.py
def snippet_list(request):
    """
    List all code snippets.
    """
    snippets = Snippet.objects.all()
    serializer = SnippetSerializer(snippets, many=True)
    return JsonResponse(serializer.data, safe=False)


# urls.py
from django.urls import path
from snippets import views

urlpatterns = [
    path('snippets/', views.snippet_list),
    path('snippets/<int:pk>/', views.snippet_detail),
]
```

Exercise - Step 3: List view and route

- Create a **recipe_list view function** in tutorial/recipes/views.py that renders the representation of all existing recipes
- Route /recipes/ to the previous view adding a url pattern in tutorial/recipes/urls.py

How to verify:

1. python manage.py runserver
2. Visit /recipes/ in your browser. You should see a list of all the recipes in JSON.

Create view: POST /snippet/ + JSON data

```
from django.http import HttpResponse, JsonResponse
from django.views.decorators.csrf import csrf_exempt
from rest_framework.parsers import JSONParser
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer

@csrf_exempt  # bypass csrf checks on POST
def snippet_list(request):
    """
    List all code snippets, or create a new snippet.
    """
    if request.method == 'GET':
        snippets = Snippet.objects.all()
        serializer = SnippetSerializer(snippets, many=True)
        return JsonResponse(serializer.data, safe=False)

    elif request.method == 'POST':
        data = JSONParser().parse(request)
        serializer = SnippetSerializer(data=data)
        if serializer.is_valid():
            serializer.save()
            return JsonResponse(serializer.data, status=201)
        return JsonResponse(serializer.errors, status=400)
```

Exercise - Step 4: Create recipe view

- Update the **recipe_list view function** in tutorial/recipes/views.py to allow creating new recipes.

How to verify:

1. python manage.py runserver
2. Using an app to interact with web APIs like postman do the following:

```
POST /recipes/
{
      "name": "Pizza",
      "description": "Put it in the oven"
}
```

3. You should see a HTTP 201 return code and the JSON representation of the new recipe.
4. If you visit the Django admin (http://localhost/admin/) you should see the new recipe.

Update and delete views

```
@csrf_exempt
def snippet_detail(request, pk):
    try:
        snippet = Snippet.objects.get(pk=pk)
    except Snippet.DoesNotExist:
        return HttpResponse(status=404)

    if request.method == 'GET':
        serializer = SnippetSerializer(snippet)
        return JsonResponse(serializer.data)

    elif request.method == 'PUT':
        data = JSONParser().parse(request)
        serializer = SnippetSerializer(snippet, data=data)
        if serializer.is_valid():
            serializer.save()
            return JsonResponse(serializer.data)
        return JsonResponse(serializer.errors, status=400)

    elif request.method == 'DELETE':
        snippet.delete()
        return HttpResponse(status=204)
```

Exercise - Step 5: Update/delete views

- Update the **recipe_detail view function** in tutorial/recipes/views.py to allow updating/deleting recipes.

How to verify:

1. python manage.py runserver
2. Using an app like postman do the following:

```
PUT /recipes/:id/
{
      “name”: “Pasta”
      “description”: “Boil it”,
}
```

3. You should see a HTTP 200 return code and the JSON representation of the updated recipe.
4. If you visit the Django admin (http://localhost/admin/) you should see the recipe has new data.
5. Same with DELETE method. You should receive a HTTP 204 and the recipe should disappear from the admin.

Using class-based views

- Using classes instead of functions enables easier reutilization of code in OOP
- Subclass from *rest_framework.views.APIView*, which automatically handle transforming *request.data* JSON to Python dict, make view CSRF exempt and **shows cool API view in browser**
- Return *rest_framework.response.Response* which handles transforming response to JSON or other formats automatically
- **if/elifs are transformed into methods**
- Requires small changes to url patterns
- Add ‘rest_framework’ to settings.INSTALLED_APPS to be able to see API view pages from browser:

```
INSTALLED_APPS = [
    ...
    'rest_framework',
    'tutorial.recipes'
]
```

List/create view as class-based view

```
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from django.http import Http404
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status

class SnippetList(APIView):
    """
    List all snippets, or create a new snippet.
    """
    def get(self, request):
        snippets = Snippet.objects.all()
        serializer = SnippetSerializer(snippets, many=True)
        return Response(serializer.data)

    def post(self, request, format=None):
        serializer = SnippetSerializer(data=request.data)    # no manual JSON parsing
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Retrieve/update/delete view as class-based view

```
class SnippetDetail(APIView):
    """
    Retrieve, update or delete a snippet instance.
    """
    def get_object(self, pk):
        try:
            return Snippet.objects.get(pk=pk)
        except Snippet.DoesNotExist:
            raise Http404

    def get(self, request, pk):
        snippet = self.get_object(pk)
        serializer = SnippetSerializer(snippet)
        return Response(serializer.data)

    def put(self, request, pk):
        snippet = self.get_object(pk)
        serializer = SnippetSerializer(snippet, data=request.data)  # no manual JSON parsing
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    def delete(self, request, pk, format=None):
        snippet = self.get_object(pk)
        snippet.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

Changes to urls.py for class-based views

```
# urls.py

from django.urls import path
from rest_framework.urlpatterns import format_suffix_patterns
from snippets import views

urlpatterns = [
    path('snippets/', views.SnippetList.as_view()),
    path('snippets/<int:pk>/', views.SnippetDetail.as_view()),
]
```

Exercise - Step 6: Use class-based views

1. Transform **recipe_list function** into **RecipeList(APIView) class** in views.py
2. Transform **recipe_detail function** into **RecipeDetail(APIView) class** in views.py
3. Update urls.py to work with new class-based views

How to verify:
Check that Create, Retrieve, Update and Delete work as before using *postman*.

Generic class-based views

- DRF equivalent to Django Generic Views for CRUD operations

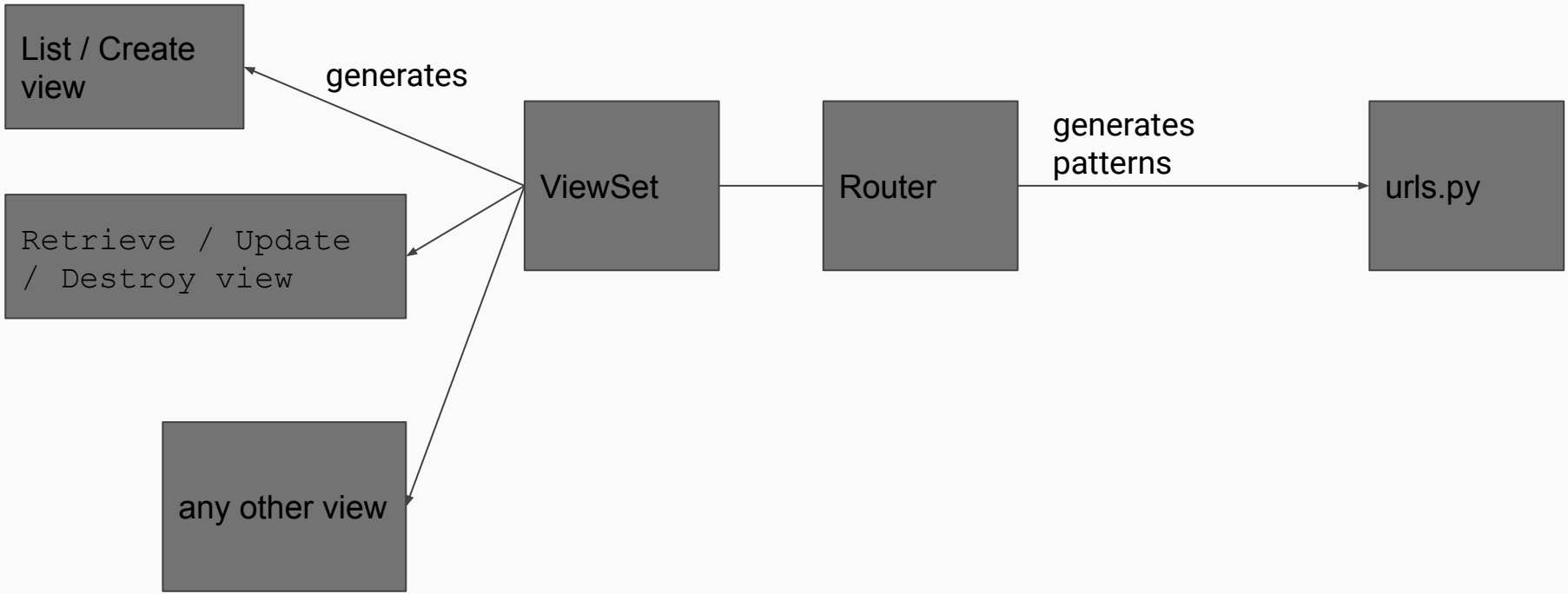
```
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import generics


class SnippetList(generics.ListCreateAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer


class SnippetDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer
```

Exercise - Step 7: Use DRF Generic Views

1. Make *RecipeDetail* inherit from *generics.RetrieveUpdateDestroyAPIView*
2. Make *RecipeList* inherit from *generics.ListCreateAPIView*

How to verify:
Check that Create, Retrieve, Update and Delete work as before using *postman*.

ViewSets and Routers



ViewSets and Routers

```
# views.py
from rest_framework import viewsets

class SnippetViewSet(viewsets.ModelViewSet):
    """
    This viewset automatically provides CRUD actions.
    """
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer



# urls.py
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from snippets import views

# Create a router and register our viewsets with it.
router = DefaultRouter()
router.register(r'snippets', views.SnippetViewSet)

# The API URLs are now determined automatically by the router.
urlpatterns = [
    path('', include(router.urls)),
]
```

Exercise - Step 8: Use ViewSets and Routers

1. Transform **RecipeList** and **RecipeDetail views** into a **ViewSet**.
2. Declare and use a router with the previous ViewSet in urls.py

How to verify:
Check that Create, Retrieve, Update and Delete work as before using *postman*.

Search view - GET /snippets/?code=Pi

```
class SnippetViewSet(viewsets.ModelViewSet):
    """
    This viewset automatically provides CRUD actions.
    """
    queryset = Snippet.objects.all()  # necessary for router
    serializer_class = SnippetSerializer

    def get_queryset(self):
        queryset = Snippet.objects.all()
        code_pattern = self.request.query_params.get('code')
        if code_pattern:
            queryset = queryset.filter(code__icontains=code_pattern)
        return queryset
```

Exercise - Step 9: Allow searching recipes by name

1. Add a *RecipeViewSet.get_queryset* method to filter recipes by name

How to verify:

```
GET /recipes/?name=Pi
[
    {
    “name”: “Pizza”
    “description”: “Put it in the oven”
    }
]
```

Nested serializers

```
# serializers.py

from rest_framework import serializers
from myapp.models import Snippet


class SnippetSerializer(serializers.ModelSerializer):
    tags = TagSerializer(many=True)  # looks up snippet.tags field

    class Meta:
        model = Snippet
        fields = ['id', 'title', 'code', 'linenos', 'language',
'style',
                  'tags']

class TagSerializer(serializers.ModelSerializer):
    class Meta:
        model = Tag
        fields = ['name']
```

Exercise - Step 10: Add “ingredients” nested field to the recipe representation

1. Create a **IngredientSerializer class** showing the ingredient name in serializers.py
2. Add the ingredients representation to RecipeSerializer in serializers.py (new “field”).
   Hint: use **IngredientSerializer(..., required=False)**

How to verify:

```
GET /recipes/1/
{
      “id”: 1,
      “name”: “Pizza”
      “description”: “Put it in the oven”,
      “ingredients”: [{“name”: “dough”}, {“name”: “cheese”}, {“name”: “tomato”}]
}
```

Exercise - Step 11: Allow creating and updating recipes specifying ingredients

1. Define a **RecipeSerializer.create** method to create ingredients based on the “ingredients” data sent when creating a recipe.
2. Define a **RecipeSerializer.update** method to update the ingredients of a recipe. Can delete existing ingredients and create new.

How to verify:

```
POST /recipes/
{
	“name”: “Pizza”
	“description”: “Put it in the oven”,
	“ingredients”: [{“name”: “dough”}, {“name”: “cheese”}, {“name”: “tomato”}]
}

Response:
{
	“id”: 1,
	“name”: “Pizza”
	“description”: “Put it in the oven”,
	“ingredients”: [{“name”: “dough”}, {“name”: “cheese”}, {“name”: “tomato”}]
}
```

Exercise 11 - Allow creating and updating recipes specifying ingredients

1. Define a **RecipeSerializer.create** method to create ingredients based on the “ingredients” data sent when creating a recipe.
2. Define a **RecipeSerializer.update** method to update the ingredients of a recipe. Can delete existing ingredients and create new.

How to verify:

```
PUT /recipes/1/
    {
      “name”: “Bad pizza”
      “description”: “Microwave it”,
      “ingredients”: [{“name”: “casa-tarradellas”}]
    }

Response:
{
      “id”: 1,
      “name”: “Bad pizza”
      “description”: “Microwave it”,
      “ingredients”: [{“name”: “casa-tarradellas”}]
}
```

Exercise 11 - Hints

```
class RecipeSerializer(serializers.ModelSerializer):
    # ...
    def create(self, validated_data):
        # Create a recipe using only “name” and “description” fields in
        # validated_data. Use recipe =Recipe.objects.create(...).

        # for each ingredient in the list validated_data[‘ingredients’]
        # create a new ingredient. UseIngredient.objects.create(recipe=recipe, ...).

        return recipe

     def update(self, instance, validated_data):
        # if there is an “ingredients” field invalidated_data
            # delete all ingredients for the recipeinstance. Use instance.ingredients.delete().
            # create a new ingredient for each ingredient name in the validated_data[“ingredients”] field

        instance = super(RecipeSerializer, self).update(instance, validated_data)
        return instance
```

Extra for advanced track

- Add a field “user” to the Recipe model
- In views, show and allow updating only own Recipes.
- Hint: Update *RecipeViewSet.get_queryset* method.