Taming the backends

By Pawel Maczewski (PAMK)

```
struct Response: Codable {
  let name: String
  let occupation: String
}
```

We have an app

But we need a backend

→ Let's imagine a dating app tinder.

→ Meets a movie rating system IMDb.

The backend

→ What for?

→ How to get it?

→ How to talk to it?

What for?

A few examples:

→ analytics

→ users management

→ content management

→ updates

→ *what else?*

There are ready solutions!

- → Realm
- → Firebase
- → Parse [1]

[1] *parse is dead, baby, parse is dead)* ⚰️

But we might not want to use them

Why?

- → Privacy concerns 💀
- → Cost 💵
- → Free 🐷[2]

[2] If you're not the customer, you're the product

What are the options?

- multiple
- J2EE
- Azure
- RoR
- Django ❤
- Server-side Swift

Use anything acceptable and what you're happy to work with.

Especially if you got the freedom to choose.

Talking to the backend

There are 2 main ways to interact with the backend.

→ REST
  Representational State Transfer

→ RPC
  Remote Procedure Call

REST

- → operates on resources, each one identified by URL
- → must be stateless
- → hypermedia links in responses
- → nesting problem
- → Examples:
  - → *return a list of all movies*
  - → *create a new movie review*

GraphQL (RPC)

- → instead of resources, operates on a set of abstract procedures on a server
- → client defines in what resources they're interested in

An example?

- → Our client app - IMDb + tinder
- → Our client application needs to fetch the top movies, their rating, the name and picture of some most important people on the cast (and their individual ratings).
- → And then swipe the actors *left* or *right* if they like or not.

How are the resources defined?

```
Person:
  - name
  - picture
  - individual movie rating

Movie:
  - title
  - rating
  - director (-> Person)
  - cast: list of -> Person

Likes:
  - -> Person they liked
```

What operations need to be done?

→ REST

1. Get top movies (`GET /movies.json`)
2. For a particular movie ID get its details (`GET /movies/tt0068646.json`)
3. Get its cast and for each one of them fetch the person details (`GET /person/nm0449984.json`)
4. According to user's pick on the person's photo, create a `Like` resource (`POST /likes.json`)

→ GraphQL

1. Get the top movies, their title and rating and their cast - for each of cast members, include name, picture and individual rating
   `GET /graphql?query={ movie { title, average_rating, cast { name, picture, rating } } }`
2. According to user's pick - create a `Like` or not
   `GET /graphql?mutation={ like { actor_id: nm0449984 } }`

Challenges with REST

- → What to nest into which response?
  - → Needs dialog between server and client developer
  - → *Too much* - heavy JOINs on backend side and heavy response
  - → *Too little* - will require several requests to fetch the needed information

"Universal" strategy

Return entire object, with every nested resource as a hypermedia link to fetch it if needed.

Cost: Some fields that aren't necessary are returned and multiple calls need to be made.

Example:

```
{
    "movie" : {
        "id" : "tt0080684",
        "title" : "Star Wars: Episode V - The Empire Strikes Back",
        "number_of_ratings" : 1071160,
        "avarage_rating" : 8.7,
        "director" : {
            "links" : {
                "self" : "/directors/nm0449984"
            }
        },
        "cast" : [
            { "links" : { "self" : "/movies/tt0080684/actors/nm012131" }},
        ]
    },
    "links" : {
        "rating" : "/movies/tt0080684/rating"
    }
}
```

Challenges with REST

- → How to interpret different paths of links to resources?
  - → `/movies/tt0080684/directors/nm0449984` might return Irvin Kershner
  - → but
  - → `/directors/nm0449984` should too.
  - → should they differ?

Now let's get to the example code

- → We won't recreate tinder. 😢
- → Instead, let's simple do app analytics
- → We'll do `REST`(-ish)
- → We're be using `Django`
  - → because it comes with some pretty nice admin panel (plus some profiling tools) and REST framework is easy to install
  - → There will be < 200 lines of code we need to write
- → And `SwiftUI`
  - → because it's the new hotness
  - → 130 lines of playground code (including empty and dumb lines)

Structure of the Django application

There's a definition of `Models`:

```
class Client(models.Model):
    client_platform = models.CharField(max_length=255, db_index=True)
    client_version = models.CharField(max_length=255, db_index=True)
    client_hash = models.CharField(max_length=2500, primary_key=True, default=random_hash, editable=False)

class LogEvent(models.Model):
    id = models.AutoField(primary_key=True)
    event_name = models.CharField(db_index=True, max_length=255)
    event_time = models.DateTimeField(db_index=True, auto_now_add=True)
    client = models.ForeignKey(Client, on_delete=models.PROTECT)

class EventParam(models.Model):
    id = models.AutoField(primary_key=True)
    param_name = models.CharField(max_length=255, db_index=True)
    param_value = models.CharField(max_length=1400)
    event = models.ForeignKey(LogEvent, on_delete=models.CASCADE, related_name="params")
```

Definition of API views:

```
class ClientSerializer(serializers.ModelSerializer):
    class Meta:
        model = Client
        fields = ['client_platform', 'client_version', 'client_hash']


class ClientHashSerializer(serializers.ModelSerializer):
    class Meta:
        model = Client
        fields = ['client_hash']


class EventParamSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = EventParam
        fields = ['param_name', 'param_value']


class LogEventSerializer(serializers.ModelSerializer):
    params = EventParamSerializer(many=True)

    class Meta:
        model = LogEvent
        fields = ['event_name', 'event_time', 'params', 'client']

    def create(self, validated_data):
        params = [EventParam(**item) for item in validated_data["params"]]
        del (validated_data["params"])
        event = LogEvent(**validated_data)
        event.save()
        for param in params:
            param.event = event
            param.save()
        return event


class LogEventViewSet(viewsets.ModelViewSet):
    queryset = LogEvent.objects.all()
    serializer_class = LogEventSerializer


class ClientViewSet(viewsets.ModelViewSet):
    queryset = Client.objects.all()
    serializer_class = ClientSerializer
```

And routing definitions:

```
from rest_framework import routers

from django.conf.urls import url, include

from .apis import LogEventViewSet, ClientViewSet

router = routers.DefaultRouter()
router.register(r'events', LogEventViewSet)
router.register(r'clients', ClientViewSet)

urlpatterns = [
    url(r'^', include(router.urls))
]
```

That's all for the *custom* code to have REST API running

You can find the code here: [Project on Github]

Questions?