Goal

The goal is to write a backend endpoint storing time sequences, modeled as ShortSequence and LongSequence, in a SQL database.

Models

The ShortSequence model is :

```
- id integer not null primary key
- tikee_id integer not null foreign key
- name character varying not null
- description text
- start timestamp without time zone not null
- interval integer not null default 10
- duration integer not null default 86400
- upload_to_cloud boolean not null default true
- image_format character varying not null default jpeg
- keep_local_copy boolean not null default false
- sequence_id bigint not null
- shooting_status character varying
- nb_images_on_sd integer not null default 0
- nb_images_uploaded integer not null default 0
```

- A short sequence begins at `start` date, lasts `duration` seconds and repeats every `interval` seconds.

The LongSequence model is :

```
- id integer not null primary key
- tikee_id integer not null foreign key
- name character varying not null
- description text
- start timestamp without time zone not null
- end timestamp without time zone not null
- upload_to_cloud boolean not null default true
- image_format character varying not null default jpeg
- keep_local_copy boolean not null default false
- infinite_duration boolean not null default false
- sequence_id bigint not null
- shooting_status character varying
- nb_images_on_sd integer not null default 0
- nb_images_uploaded integer not null default 0
```

- A long sequence begins at `start` date and ends at `end` date.
- if `infinite_duration`is `true`, the sequence never ends.

Specifications

We need a CRUD endpoint for the short and long sequences.

Short and long sequences listing

- The backend receives a GET request to the url `/short_sequences/#tikee_id` or `/long_sequences/#tikee_id`.
- `ShortSequence` or `LongSequence` models, with a matching `tikee_id`, must be read from a SQL database
- the returned payload is a Json object containing a list of sequences for the requested tikee (structure to be defined).

Short and long sequences reading

- The backend receives a GET request to the url `/short_sequences/#id` or `/long_sequences/#id`.
- A `ShortSequence` or `LongSequence` model, with a matching `id`, must be read from a SQL database
- the returned payload is a Json object containing a full sequence model.

Short and long sequences creation

- The backend receives a POST request to the url `/short_sequences` or `/long_sequences`.
- The payload is a Json object containing the necessary information (except for `created_at` and `updated_at` attributes, which will have to be handled on the backend side)
- A `ShortSequence` or `LongSequence` model must be stored in a SQL database

Short and long sequences updating

- The backend receives a PUT request to the url `/short_sequences/#id` or `/long_sequences/#id`.
- The payload is a Json object containing the necessary information.
- A `ShortSequence` or `LongSequence` model must be updated in a SQL database

Short and long sequences deleting

- The backend receives a DELETE request to the url `/short_sequences/#id` or `/long_sequences/#id`.
- The `ShortSequence` or `LongSequence` model must be deleted from the SQL database

Error handling

- In case of an error, creation and updating endpoints should respond with a Json object with the following structure, containing a list of validation errors :

```
{
  errors : [
    {
      code: ...,
      title: ...
    },
    {
```

```
      code: ...,
      title: ...
    }
    ...
  ]
}
```

Mandatory business rules (must be implemented)

- Business rule 1 : input fields should be sensible (for instance, non negative `duration`)
- Business rule 2 : two short sequences must not overlap
- Business rule 3 : two long sequences must not overlap

Optional business rules (should be implemented)

- Business rule 3, idempotency :
  - for every requests, the frontend sends a `X-Request-Id` HTTP header, containing a unique ID.
  - Each endpoint should ignore a request for which an identical `X-Request-Id` has been received within the last 24 hours.
- Business rule 4, rate limiting :
  - If the request rate (requests received per minute) exceeds the value of the parameter `max_requests_per_minute`, the endpoints should return a HTTP 429 response.
  - Rate limiting algorithm is not imposed.

What to do ?

- Implement the endpoints
- Implement the mandatory business rules (for business rule 1, only a few validations)
- Implement the error handling
- Implement some tests
- Propose a solution for (or implement, if you feel like doing it !) the optional business rules

Preferably, the endpoints should be using the Rails framework, but you may chose any framework you feel comfortable with...

Note: The purpose of this test is to discuss the code and the choices you made.

Good luck 😉