CODING NOMADS

18) Going RESTful: Build an API Lesson

Create API Resource Endpoints

8 min to complete · By Brandon Gigous

Contents

- Introduction
- Resource Endpoints
- GET
- POST
- PUT
- More Endpoints
- Paginating JSON Data
- Summary: Create API Resource Endpoints

In this lesson, you'll learn how to make the routes that API clients can use to request, and create, data in your webapp.

Resource Endpoints

Resource endpoints are the *points* of exchange, where your app can receive client requests.

GET

Remember what GET requests are? Of course you do, a GET request is meant to *get* data! One API endpoint you can make is one for getting a particular user:

```
# app/api/users.py

@api.route('/users/<int:id>')
def get_user(id):
    user = User.query.get_or_404(id)
    return jsonify(user.to_json())
```

This route simply takes a user id, gets the associated user, and JSONifies the object. If the user doesn't exist, the API would catch the error and indicate that to the client.

What about multiple objects, like all the compositions in the database? It might be a lot of data, but you could do it like this:

```
# app/api/compositions.py

@api.route('/compositions/')
def get_compositions():
    compositions = Composition.query.all()
    return jsonify({ 'compositions': [composition.to_json()
                                      for composition in composition
```

POST

A POST request to make a new composition through the API can be done like so:

```
# app/api/compositions.py

@api.route('/compositions/', methods=['POST'])
@permission_required(Permission.PUBLISH)
def new_composition():
    composition = Composition.from_json(request.json)
    composition.artist = g.current_user
    db.session.add(composition)
    db.session.commit()
    return jsonify(composition.to_json()), 201, \
        {'Location': url_for('api.get_composition', id=composition.i
```

This route first deserializes the composition data. If all goes well, it then grabs the user that made that request (which was determined in `verify_password`), and adds the composition to the database. It returns 1) the deserialized, then again serialized composition, 2) a 201 status code, and 3) a `Location` header to the url of the new composition.

Of course, your client has to be able to *publish* their composition, so your app must check that they have the appropriate permissions. You can make a custom decorator `permission_required` to do so:

```
# app/api/compositions.py

def permission_required(permission):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if not g.current_user.can(permission):
                return forbidden("Insufficient permissions")
            return f(*args, **kwargs)
        return decorated_function
    return decorator
```

While it's pretty similar than the other `permission_required` decorator you made for your app, this one is special made for your API blueprint.

PUT

Users can also edit their compositions with a PUT request through the API:

```
@api.route('/compositions/<int:id>', methods=['PUT'])
@permission_required(Permission.PUBLISH)
def edit_composition(id):
    composition = Composition.query.get_or_404(id)
    if g.current_user != composition.artist and \
            not g.current_user.can(Permission.ADMIN):
        return forbidden('Insufficient permissions')
    import json
    put_json = json.loads(request.json)
    composition.release_type = put_json.get('release_type', composit
    composition.title = put_json.get('release_type', composition.tit
    composition.description = put_json.get('description', compositio
    db.session.add(composition)
    db.session.commit()
    return jsonify(composition.to_json())
```

This view function first finds the composition and makes sure the client making the request is the same one who made the composition, otherwise that's a no-go for the client. Then, the composition is modified with the new data and readded to the database. The newly-modified composition is returned as JSON.

More Endpoints

The examples above should get you off to a good start with your API, however there are still more make.

Learn by Doing 

Task: Create the API endpoints below.

URL	Method	File	What It Does
`/users/<int:id>/compositions/`	GET	`users.py`	Return all the compositions written by a user
`/users/<int:id>/timeline/`	GET	`users.py`	Return all the compositions followed by a user
`/compositions/<int:id>`	GET	`compositions.py`	Return a composition

- Constant, personal, in-depth support 7 days a week via Discord
- Accountability, feedback, encouragement, and success

Get Mentorship

Paginating JSON Data

Just like you've already done in terms of paginating data within a browser page, you can also paginate your JSON data. Returning a whole table of data can be expensive, so it's better to chunk it up to make it more manageable. However, you won't need a fancy pagination widget, as this can all be taken care of within the route logic.

Let's take the `get_compositions()` view function as an example:

```
@api.route('/compositions/')
def get_compositions():
    page = request.args.get('page', 1, type=int)
    pagination = Composition.query.paginate(
        page,
        per_page=current_app.config['RAGTIME_COMPS_PER_PAGE'],
        error_out=False)
    compositions = pagination.items
    prev = None
    if pagination.has_prev:
        prev = url_for('api.get_compositions', page=page-1)
    next = None
    if pagination.has_next:
        next = url_for('api.get_compositions', page=page+1)
    return jsonify({
```

```
        'compositions': [composition.to_json() for composition in co
        'prev': prev,
        'next': next,
        'count': pagination.total
    })
```

As you can see, a lot of the code is very similar to pagination code you've written before. But this time, URLs are captured for the previous and next chunks of data, respectively. The chunk at the current page, as `compositions`, is then jsonified and returned to the client.

Task: Add pagination to the rest of the routes that return multiple objects.

Hey, great job! If you made it here you're well on your way to becoming a seasoned Flask API developer. The next lesson will guide you through paginating results for large sets of JSON data.

Summary: Create API Resource Endpoints

- Resource endpoints are the *points* of exchange, where your app can receive client requests
- A GET request is meant to *get* data! One API endpoint you can make is one for getting a particular user
- A POST request to make a new composition through the API can be done like so:

```
# app/api/compositions.py

@api.route('/compositions/', methods=['POST'])
@permission_required(Permission.PUBLISH)
def new_composition():
    composition = Composition.from_json(request.json)
    composition.artist = g.current_user
    db.session.add(composition)
    db.session.commit()
```

```
        return jsonify(composition.to_json()), 201, \
            {'Location': url_for('api.get_composition', id=compositi
```

- Users can also edit their compositions with a PUT request through the API
- You can also paginate your JSON data

Previous Next → Video: Using Serialization in API Endpoints