REST and JSON APIs

Download Demo Code <../flask-rest-json-api-demo.zip>

Goals

- Review GET vs POST
- Review other HTTP verbs (PUT, PATCH, DELETE)
- Describe what REST is
- Build and Test JSON APIs

Revewing HTTP Verbs

GET and POST

GET

- Remains in history, can be cached/bookmarked
- Data sent in URL, in query string
- Repeatable

POST

- Doesn't remain in history, is not cached/bookmarked
- Data sent in body of the request
- Not repeatable

When to use ***GET*** or ***POST***?

- Searching / Filtering? ***GET***
- Sending an email? ***POST***
- Updating a user? ***POST*** ?

PUT / PATCH / DELETE

PUT
: Update entire resource

PATCH
: Update *part of* resource *(patch it up)*

DELETE
: Delete resource

Requesting With Methods

HTTP Verb	Forms / Links	AJAX	Server-side

HTTP Verb	Forms / Links	AJAX	Server-side
GET	✓	✓	✓
POST	✓	✓	✓
PUT / PATCH	✗	✓	✓
DELETE	✗	✓	✓

Safety & Idempotence

A **safe** operation is one that does not change the data requested.

An **idempotent** operation can be performed many times (with same data) with the result of all calls being the same as if it was done once.

- Idempotence refers to side-effects not all-effects or responses.
- Example: In arithmetic, calculating absolute value

Which Methods Are Safe / Idempotent?

HTTP Verb	Safe?	Idempotent?
GET	✓	✓
POST	✗	✗
PUT / PATCH	✗	✓
DELETE	✗	✓

Why do we care about this?

- Better describe the routes that we create
- Build standards around how we define routes
- Core part of the REST standard!

Introduction to REST

Imagine you're a developer

- Hopefully this should not be imagination!

- Your task: create route for an API that will update a user!
  - ***POST /users/[id]/update***?
  - ***POST /users/[id]/change***?
  - ***PATCH /users/[id]***?
- With this much flexibility, it's very helpful to standardize!

REST

- Architectural style defining constraints for creating web services
  - Includes things like: client-server model, statelessness and cacheability
- APIs that adhere to these constraints are called *RESTful APIs*

RESTful APIs

- Usually have base url
  - eg *http://api.site.com/* or *http://site.com/api/*
- Have a **resource** after the base url
  - eg *http://api.com/books* or *http://site.com/api/books*
- Use standard HTTP verbs (GET, POST, PUT/PATCH, DELETE)
- Structure routes in a standardized way *("RESTful routing")*

Resource

- An object with a type, associated data, relationships to other resources
- A set of methods that operate on it
- Analogous to instance/methods in OO
  - HTTP verbs describe methods on resource
  - ***DELETE /cats/fluffy*** is same idea as ***fluffy.delete()***

Not every route in a RESTful API will neccessary be around resources. For example, you may have routes to initially authenticate with the API that aren't using a resource in the URL.

RESTful routes

RESTful routes for a resource called *snacks*:

HTTP Verb	Route	Meaning
GET	/snacks	Get all snacks
GET	/snacks/*[id]*	Get snack
POST	/snacks	Create snack
PUT / PATCH	/snacks/*[id]*	Update snack
DELETE	/snacks/*[id]*	Delete snack

But what about X, Y or Z?

No! Make sure you follow these naming conventions!

RESTful Route Responses

Not entirely standardized — but these are common:

GET /snacks
Returns 200 OK, with JSON describing ***snacks***

GET /snacks/*[id]*
Returns 200 OK, with JSON describing single ***snack***

POST /snacks
Returns 201 CREATED, with JSON describing new ***snack***

PUT or PATCH /snacks/*[id]*
Returns 200 OK, with JSON describing updated ***snack***

DELETE
Returns 200 OK, with JSON describing success

HTTP Verb	Route	Meaning	Status	Response JSON
GET	/snacks	Get all	200	`{"snacks": [{id, name, cals}, ...]}`
GET	/snacks/*[id]*	Get	200	`{"snack": {id, name, cals}}`
POST	/snacks	Create	201	`{"snack": {id, name, cals}}`
PUT / PATCH	/snacks/*[id]*	Update	200	`{"snack": {id, name, cals}}`
DELETE	/snacks/*[id]*	Delete	200	`{"deleted": snack-id}`

Examples of RESTful routing:

- Stripe: https://stripe.com/docs/api?lang=curl#charges <https://stripe.com/docs/api?lang=curl#charges>
- Github: https://developer.github.com/v3/repos/ <https://developer.github.com/v3/repos/>
- Yelp: https://www.yelp.com/developers/documentation/v3/event <https://www.yelp.com/developers/documentation/v3/event>
- Spotify: https://developer.spotify.com/documentation/web-api/reference/playlists/ <https://developer.spotify.com/documentation/web-api/reference/playlists/>

Nested Routes

HTTP Verb	Route	Response
GET	/businesses	Get info about all businesses
GET	/businesses/*[biz-id]*	Get info about business
POST	/businesses	Create busines

HTTP Verb	Route	Response
PUT / PATCH	/businesses/*[biz-id]*	Update business
DELETE	/businesses/*[biz-id]*	Delete business
GET	/businesses/*[biz-id]*/reviews	Display all reviews for business
GET	/businesses/*[biz-id]*/reviews/*[rev-id]*	Display review for business
POST	/businesses/*[biz-id]*/reviews	Create review for business
PUT / PATCH	/businesses/*[biz-id]*/reviews/*[rev-id]*	Update review for business
DELETE	/businesses/*[biz-id]*/reviews/*[rev-id]*	Delete review for business

RESTful APIs With Flask

- Can still use Flask and Flask-SQLAlchemy
- Will respond with JSON, not HTML
  - Won't typically use Jinja to make JSON, just ***jsonify*** in route
  - Can't redirect — return JSON of answer

Flask jsonify

`jsonify(thing)`
Returns JSON of thing (usually dict, but could be list)

`jsonify(name="Jane", age=21)`
Returns JSON like `{"name": "Jane", "age": 21}`

Limitations of JSON / jsonify

- JSON can only represent dictionaries, lists, and primitive types
  - Cannot represent things like SQLAlchemy model instances
- Python can't just "turn your objects into JSON"
  - Requires a process called *serialization*

Serialization

You can turn your instances into dictionaries or lists:

demo/app.py

```
def serialize_dessert(dessert):
    """Serialize a dessert SQLAlchemy obj to dictionary."""

    return {
        "id": dessert.id,
```

```
        "name": dessert.name,
        "calories": dessert.calories,
    }
```

Example: RESTful Routes Returning JSON

demo/app.py

```
@app.route("/desserts")
def list_all_desserts():
    """Return JSON {'desserts': [{id, name, calories}, ...]}"""

    desserts = Dessert.query.all()
    serialized = [serialize_dessert(d) for d in desserts]

    return jsonify(desserts=serialized)
```

demo/app.py

```
@app.route("/desserts/<dessert_id>")
def list_single_dessert(dessert_id):
    """Return JSON {'dessert': {id, name, calories}}"""

    dessert = Dessert.query.get(dessert_id)
    serialized = serialize_dessert(dessert)

    return jsonify(dessert=serialized)
```

Sending Data to a Flask JSON API

- For Insomnia, choose JSON as the request type.
- For cURL, set the *Content-Type* header:

```
$ curl localhost:5000/api/desserts \
  -H "Content-Type: application/json" \
  -d '{"name":"chocolate bar","calories": 200}'
```

  (Makes a POST to /api/desserts, passing in that JSON data)

- For AJAX using Axios, sending JSON is the default

Receiving Data in a Flask JSON API

If request is made with *Content-Type: application/json*

- it won't be in ***request.args*** or ***request.form***
- will be inside of ***request.json***!

Example: RESTful Route Receiving and Returning JSON

demo/app.py

```
@app.route("/desserts", methods=["POST"])
def create_dessert():
    """Create dessert from form data & return it.

    Returns JSON {'dessert': {id, name, calories}}
    """

    name = request.json["name"]
    calories = request.json["calories"]

    new_dessert = Dessert(name=name, calories=calories)

    db.session.add(new_dessert)
    db.session.commit()

    serialized = serialize_dessert(new_dessert)

    # Return w/status code 201 --- return tuple (json, status)
    return ( jsonify(dessert=serialized), 201 )
```

Testing our API

- We will be testing the JSON response, not HTML
  - In particular, we'll look at ***response.json***, not ***response.data***
  - We'll also send data via a ***json*** named argument, not a ***data*** named argument
- This makes things even easier! We're just testing data, not presentation
- Can experiement before/while writing tests with Insomnia or curl

Example Tests

demo/tests.py

```
def test_all_desserts(self):
    with app.test_client() as client:
        resp = client.get("/desserts")
        self.assertEqual(resp.status_code, 200)

        self.assertEqual(
            resp.json,
            {'desserts': [{
                'id': self.dessert_id,
                'name': 'TestCake',
                'calories': 10
            }]})
```

demo/tests.py

```
def test_create_dessert(self):
    with app.test_client() as client:
        resp = client.post(
            "/desserts", json={
                "name": "TestCake2",
                "calories": 20,
            })
        self.assertEqual(resp.status_code, 201)

        # don't know what ID it will be, so test then remove
        self.assertIsInstance(resp.json['dessert']['id'], int)
        del resp.json['dessert']['id']

        self.assertEqual(
            resp.json,
            {"dessert": {'name': 'TestCake2', 'calories': 20}})

        self.assertEqual(Dessert.query.count(), 2)
```

Wrap-Up

Wrap Up

- RESTful APIs have standards around routes & methods
- These are used for *API* applications, not HTML-returning applications