JSON Serialization for Your Flask API

10 min to complete · By Brandon Gigous

Contents

In this lesson you'll learn about how to convert your app's data to and from JSON in order for API clients to interact with your API.

Serializing Resources

When parts of the web and its users talk to one another, data is exchanged through HTTP requests as JSON. That means in order for your app to send data to your clients through the API, it must take what's internally stored in your app's database and convert it into JSON. This process is called JSON *serialization*, and the reverse is *deserialization*, where your app receives a JSON dictionary and converts it to data your database understands. Without serialization, your API wouldn't be an API!

JSON Serialization

So you have your models in the database, and you need to serialize them in order to send them over to clients. How do you do that? By adding a function to each model of course!

Before getting into that, let's make clear what you'll allow clients of your API to interact with. In particular, the underlying database models you'll want to serialize are:

- `User`
- `Composition`

The tables these models represent the bulk of the content your clients might care about.

What did this function look like? Well, a JSON object is simply a dictionary containing fields which contain values. A Python dictionary would get you most of the way to JSON, then that dictionary can easily be converted to JSON using built in libraries. So you can simply make a `to_json()` function for each objects of one of these models. Here's one for the `User` model for example:

```
class User(UserMixin, db.Model):
    # ...
    def to_json(self):
        json_user = {
            'url': url_for('api.get_user', id=self.id),
            'username': self.username,
            'last_seen': self.last_seen,
            'compositions_url': url_for('api.get_user_compositions',
            'followed_compositions_url': url_for('api.get_user_follo
            'composition_count': self.compositions.count()
        }
        return json_user
```

A dictionary like this gives a client enough about a user without, of course, giving away sensitive information. Each of the `*url` fields (`url`, `compositions_url`, and `followed_compositions_url`) must return a URL for that particular resource. So, `url` would be a link to a user profile, like `http://yourapp.com/user/tom` for user Tom. And yes, the routes in the first argument of `url_for` will be defined by you a bit later! For now you don't have to worry about it.

Some data returned in a JSON object like this can be arbitrary, meaning they don't have to directly map to something you have in the database. For example,

`composition_count` represents how many compositions a user has.

Learn by Doing 

Task: Define the `to_json()` method for the `Composition` model. Refer to the tables below when creating the JSON data for each.

Composition:

Key	Value
`url`	URL of composition. Use `api.get_composition` as the route
`release_type`	Release type of the composition
`title`	Title of the composition
`description`	Description (plaintext)
`description_html`	Description (HTML)
`timestamp`	Time composition was posted
`artist_url`	URL of user who posted composition. Use `api.get_user` as the route

Comment:

Key	Value
`url`	URL of comment. Use `api.get_comment` as the route
`composition_url`	URL of composition. Use `api.get_composition` as the route
`body`	Body (plaintext)
`body_html`	Body (HTML)

Key	Value
`timestamp`	Time comment was written
`user_url`	URL of user who wrote the comment. Use `api.get_user` as the route

JSON Deserialization

You can also give your clients the capability to create compositions through your API. This is similar to a user posting a composition or via the forms in your webapp, but an API client would send the relevant data through JSON instead.

When your app receives such JSON data, like for a composition, your app needs to check that the data is valid before adding to the database. You can write a static method `from_json()` for a `Composition` like so, which would take in a JSON dictionary and return a `Composition`:

```
from app.exceptions import ValidationError

class Composition(db.Model):
    # ...
    @staticmethod
    def from_json(json_composition):
        release_type = json_composition.get('release_type')
        title = json_composition.get('title')
        description = json_composition.get('description')
        if release_type is None:
            raise ValidationError("Composition must have a release t
        if title is None:
            raise ValidationError("Composition must have a title")
        if description is None:
            raise ValidationError("Composition must have a descripti
        return Composition(release_type=release_type,
                           title=title,
                           description=description)
```

What's `ValidationError`? Well, if the data received by your app is invalid, the `from_json()` method needs to be able to indicate that it can't handle the invalid data properly. So, you can define a new exception for cases like these in `app/exceptions.py`:

```
class ValidationError(ValueError):
    pass
```

However, if you keep the code like this and a client *does* send invalid data, your server would error out with a `ValidationError` resulting in a 500. To more gracefully handle it, and indicate to the client that they sent bad data, you create an error handler to for your API:

```
from ..exceptions import ValidationError

@api.errorhandler(ValidationError)
def validation_error(e):
    return bad_request(e.args[0])
```

Nicely done! Now data will be able to flow smoothly both in and out of your app. Next is to make the endpoints where the data can be found by clients.

Summary: JSON Serialization and Deserialization

- To send data to your clients through the API, it must take what's internally stored in your app's database and convert it into JSON. This process is called *serialization*, and the reverse is *deserialization*, where your app receives a JSON dictionary and converts it to data your database understands. Without serialization, your API wouldn't be an API!
- A JSON object is simply a dictionary containing fields which contain values. A Python dictionary would get you most of the way to JSON, then that dictionary can easily be converted to JSON using built in libraries. So you can simply make a `to_json()` function for each object.

- To deserialize: When your app receives such JSON data, like for a composition, your app needs to check that the data is valid before adding to the database. You can write a static method `from_json()` for a `Composition`, which would take in a JSON dictionary and return a `Composition`

Previous Next → Video: Serialize API Resources