RESTful Application Programming Interfaces

Brian Dickinson

University of Rochester

Fall 2019
REST is an architectural style for building APIs based on HTTP with the following properties.

- **Client-Server** - There is a clear distinction between client and server. In fact, client and server are often developed by different teams or even different companies.

- **Stateless** - Each client request must contain all relevant information; the server must not rely on any state information about the client. Likewise, each server response should be understandable without any additional context.

- **Cache** - The server should explicitly mark each response as cacheable or noncacheable to allow clients and intermediaries to optimize requests.

- **Uniform Interface** - There should be a precise, well-defined system for addressing and accessing resources.
What is REST? II

- **Layering** - Servers and clients should be agnostic to intermediaries such as proxies, caches, and gateways that are required for performance and reliability.

- **Code-on-Demand** - Clients may be able to download and execute code from the server (JavaScript).
REST is built around the concept of resources. Resources can be
- Objects
- Object Collections
- Functions

Resources are addressed using Unique Resource Identifiers (URIs)
Unique Resource Identifiers

- Uniform Resource Identifiers must uniquely identify a single resource.
- In the context of HTTP, the most logical choice of URI is the Uniform Resource Locator (URL).
- The query string should not be needed to uniquely identify a resource. It should only be used as a filter on a collection or to provide parameters to a function.

protocol address path query string
https://www.example.com/path?arg1=val1&arg2=val2
Client applications want to interact with resources in multiple ways. This is done with HTTP request methods.

<table>
<thead>
<tr>
<th>HTTP Verb</th>
<th>CRUD</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>Create</td>
</tr>
<tr>
<td>GET</td>
<td>Read</td>
</tr>
<tr>
<td>PUT</td>
<td>Update/Replace</td>
</tr>
<tr>
<td>PATCH</td>
<td>Update/Modify</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete</td>
</tr>
</tbody>
</table>

Not every method should be supported for every resource. If a resource does not support a requested method it should return a 405 (Method Not Allowed) response code.

---

1See page 201 in your text and the REST API tutorial.
In addition to returning the requested resource (more on that in a minute), you must also return an appropriate HTTP response code.

<table>
<thead>
<tr>
<th>HTTP Response</th>
<th>Code</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Informational</td>
<td>100</td>
<td>Continue</td>
</tr>
<tr>
<td>Success</td>
<td>200</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>201</td>
<td>Created</td>
</tr>
<tr>
<td></td>
<td>204</td>
<td>No Content</td>
</tr>
<tr>
<td>Redirection</td>
<td>301</td>
<td>Moved Permanently</td>
</tr>
<tr>
<td></td>
<td>302</td>
<td>Found</td>
</tr>
<tr>
<td></td>
<td>303</td>
<td>See Other</td>
</tr>
<tr>
<td>Client Error</td>
<td>400</td>
<td>Bad Request</td>
</tr>
<tr>
<td></td>
<td>401</td>
<td>Unauthorized</td>
</tr>
<tr>
<td></td>
<td>403</td>
<td>Forbidden</td>
</tr>
<tr>
<td></td>
<td>404</td>
<td>Not Found</td>
</tr>
<tr>
<td>Server Error</td>
<td>500</td>
<td>Internal Server Error</td>
</tr>
<tr>
<td></td>
<td>501</td>
<td>Not Implemented</td>
</tr>
</tbody>
</table>
Up to this point we have used HTML in our response bodies. In this case XML and JSON are better suited.

- Both are human readable and hierarchical
- Both are widely supported with parsers
- JSON is more modern and less verbose than XML
- JSON is usually preferred for non-legacy systems
Extensible Markup Language (XML)

- markup language much like HTML
- designed as a format for storing and transporting data
- it stores data in a way that many different languages can read making it useful for passing complex data between diverse applications
- W3C recommendation as of 1998
<?xml version="1.0" encoding="UTF-8"?>
<breakfast-menu>
 <food>
   <name>Belgian Waffles</name>
   <price>$5.95</price>
   <description>two of our famous Belgian waffles with plenty of real maple syrup</description>
   <calories>650</calories>
 </food>
 <food>
   <name>Strawberry Belgian Waffles</name>
   <price>$7.95</price>
   <description>light Belgian waffles covered with strawberries and whipped cream</description>
   <calories>900</calories>
 </food>
 <food>
   <name>French Toast</name>
   <price>$4.50</price>
   <description>thick slices made from our homemade sourdough bread</description>
   <calories>600</calories>
 </food>
 <food>
   <name>Homestyle Breakfast</name>
   <price>$6.95</price>
   <description>two eggs, bacon or sausage, toast, and our ever-popular hash browns</description>
   <calories>950</calories>
 </food>
</breakfast-menu>
JavaScript Object Notation (JSON)

- data interchange format for storing or transporting data
- lighter than XML with similar functionality
- strings written in the native JavaScript format for declaring objects
- works well with JavaScript since it doesn’t require complex parsing

// deserialize a JSON string into a JavaScript object
var obj = JSON.parse('{{"Hello": 0, "World":1}}');

// serialize a JavaScript object into a JSON string
var str = JSON.stringify(obj);
{  "foods":[  {  "name": "Belgian Waffles",  "price": 5.95,  "description": "two of our famous Belgian waffles with plenty of real maple syrup",  "calories": 650  },  {  "name": "Strawberry Belgian Waffles",  "price": 7.95,  "description": "light Belgian waffles covered with strawberries and whipped cream",  "calories": 900  },  {  "name": "French Toast",  "price": 4.50,  "description": "thick slices made from our homemade sourdough bread",  "calories": 600  },  {  "name": "Homestyle Breakfast",  "price": 6.95,  "description": "two eggs, bacon or sausage, toast, and our ever-popular hash browns",  "calories": 950  }  ]  }
}
Resource Serialization

The simplest way to handle serialization is for each model to support “to_json” and “from_json” methods.

- JSON support is limited to basic Python types such as lists, dict, strings, and numbers
- Not every attribute of an object needs to exist in the JSON representation
- It can be helpful to include extra attributes in the JSON that are not in the model
- These serializations will make up the response bodies of calls to your API
Resource Endpoint Examples

```python
from flask import jsonify

@app.route("/api/v1/posts/")
def get_posts():
    posts = Post.query.all()
    return jsonify({"posts": [post.to_json() for post in posts]})

@app.route("/api/v1/post/<int:id>")
def get_post(pid):
    post = Post.query.get_or_404(pid)
    return jsonify(post.to_json())
```
Using Marshmallow Schemas

- Marshmallow is a framework for serializing and deserializing between Python objects and JSON.
- You should consider using marshmallow and/or marshmallow-sqlalchemy to define serialization schemas for your database Models.
- This will involve more initial effort in understanding marshmallow, but with a small investment is much easier and safer than writing custom to_json and from_json methods.
from marshmallow import Schema, fields
from marshmallow_sqlalchemy import ModelSchema

# define an Author database model as we have before
class Author(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    first = db.Column(db.String(80))
    last = db.Column(db.String(80))

# solution with no read only or dynamically created fields
class AuthorSchema(ModelSchema):
    class Meta:
        model = Author

# solution with read only id and dynamic formatted name
class AuthorSchema(Schema):
    id = fields.Int(dump_only=True)
    first = fields.Str()
    last = fields.Str()
    fmtname = fields.Method("fmtname", dump_only=True)
author = Author(first="J. R. R.", last="Tolkein")
author_schema = AuthorSchema()
book = Book(title="The Hobbit", author=author)
session.add(author)
session.add(book)
session.commit()

serialized = author_schema.dump(author)
print(serialized)
# {"books":[1], "id":1, "first":"J. R. R.", "last":"Tolkein"}

deserialized = author_schema.load(serialized, session=session)
print(deserialized == author)
# True
There are a number of additional attributes you can set besides model which will allow you to exercise finer control over serialization. Specifically look into the following

- only
- exclude
- load_only
- dump_only
RESTful APIs must be **Stateless**.

- Our previous auth system used cookies.
- Cookies are a stateless gray area, but should be avoided.
- Our new system will allow Basic HTTP authentication OR a session cookie to acquire an API token.
- API tokens will be required to access API endpoints.
HTTP Authentication and Tokens

- Basic HTTP Authentication encodes the username and password in the HTTP header “Authentication”
- Just like on the login page this means that this **MUST** be done over HTTPS.
- Credentials are separated by a colon “:” and encoded with a URL safe variant of Base64

```python
import base64
import requests

creds = "Aladdin:OpenSesame"
b64creds = base64.urlsafe_b64encode(creds.encode("utf-8"))
url = "https://www.example.com/api/v1/login"
token = requests.get(url, headers={"Authorization": b64creds})
```
import base64

@app.route("/api/v1/login")
def api_login():
    if current_user.is_authenticated:
        return get_token(current_user.id), 200
    if "Authorization" in request.headers:
        try:
            b64creds = request.headers["Authorization"]
            creds = base64.urlsafe_b64decode(b64creds)
            email, password = creds.split(":")
        except Exception:
            return "Invalid Authorization Header", 400
        user = User.query.filter_by(email=email).first()
        if user is not None and user.verify_password(password):
            return get_token(user.id), 200
    return "Authentication Failed", 403
from itsdangerous import TimedJSONWebSignatureSerializer

def get_token(uid):
    secret = app.config["SECRET_KEY"]
    s = TimedJSONWebSignatureSerializer(secret,
        expires_in=3600)
    token = s.dumps({"id": uid})
    return token

def verify_token(token):
    secret = app.config["SECRET_KEY"]
    s = TimedJSONWebSignatureSerializer(secret)
    try:
        uid = s.loads(token).get("id")
    except Exception:
        return None
    return uid
Further Study

- **Versioning** - When the server and client are developed separately, you may need to serve multiple versions of an API at the same time. This is usually done by including the API version in the URL. See pages 202-204 in your text for more information.

- **Pagination** - Some collections will grow too large to be returned in a single response. Your textbook provides a good example for handling this on page 216.
Summary

- RESTful APIs are stateless and serve resources to a client over HTTP.
- Resources are uniquely identified by their URLs and different HTTP request methods.
- Each client request must include all necessary information to identify a resource, indicate the desired action, and provide authentication.
- Each server response should be understandable without any additional context, and should provide URLs necessary to acquire more related information.
- The requested resources are serialized into JSON (or XML) and returned to the client.
- HTTP response codes are used to indicate the results of the requested action.
- Because RESTful APIs must be stateless, we allow for authentication using Basic HTTP Authentication.
Miguel Grinberg.
Application programming interfaces (apis).

Steven Loria.
Marshmallow.


J. Reschke. The 'basic' http authentication scheme. RFC 7617, GreenBytes, September 2015.
Armin Ronacher.
It’s dangerous.
https://pythonhosted.org/itsdangerous/.

Various.
Basic access authentication.