RESTful Services

We are sending/receiving data between the client and server

- REST stands for *Representational State Transfer*

We are sending objects back and forth that "represent" our domain objects (customers, invoices, dragons, etc.)

We send all necessary data back and forth each request/response rather than keeping the object stored entirely on the server

Classic Web Services vs RESTful APIs

Web Services	REST
XML for payload data	Can use plain-text, XML, JSON (commonly), etc.
Uses SOAP (Simple Object Access Protocol)	No specific protocol
Invokes methods remotely	Access data via URIs
Has formal standards: SOAP, WSDL, WS-Security, etc.	Solutions tend to be simpler to learn, build, and execute
XML is difficult to work with within browsers	JSON is easy to use within browsers
Has error handling capabilities built into SOAP	Errors are managed through HTTP status codes

RESTful Resources

- A *resource* is a core or fundamental component in a RESTful application

- A resource is an object being served (returned)
  - It usually derives from the business domain
  - Typically, operations will be performed on this entity

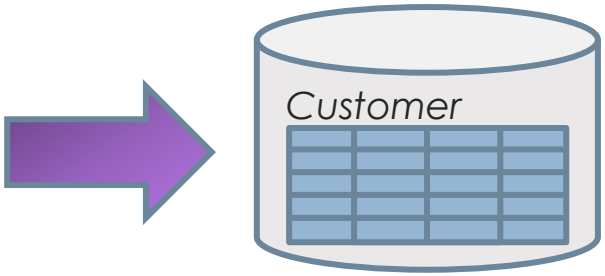
- Resource examples:
  - Invoices
  - Patients
  - Customers
  - Documents

A resource can be a collection (like *accounts*) or singular (like *account*) and may contain sub-resources (like *address*)

Key Components of a RESTful Service

- URIs to map to resources

http://server.com/api/customers

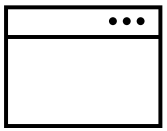


- Manipulation of resources via HTTP methods

GET
POST
PUT
DELETE

http://server.com/api/customers

- Transfer of state (stateless)

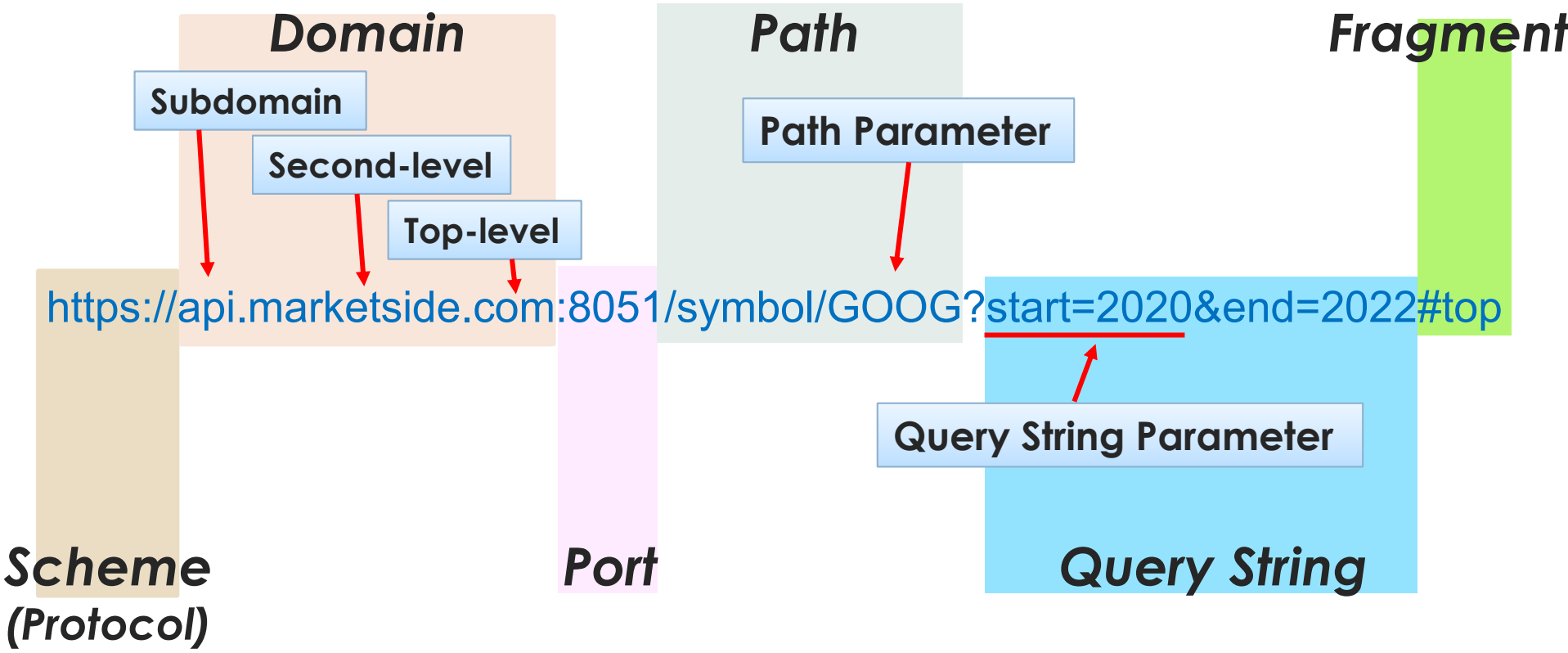


```
{
  customer_id: 121,
  name: Blackburn Industries,
  …
}
```

- Cacheable

```
Expires: Fri, 20 May 2022 19:20:49 GMT
```

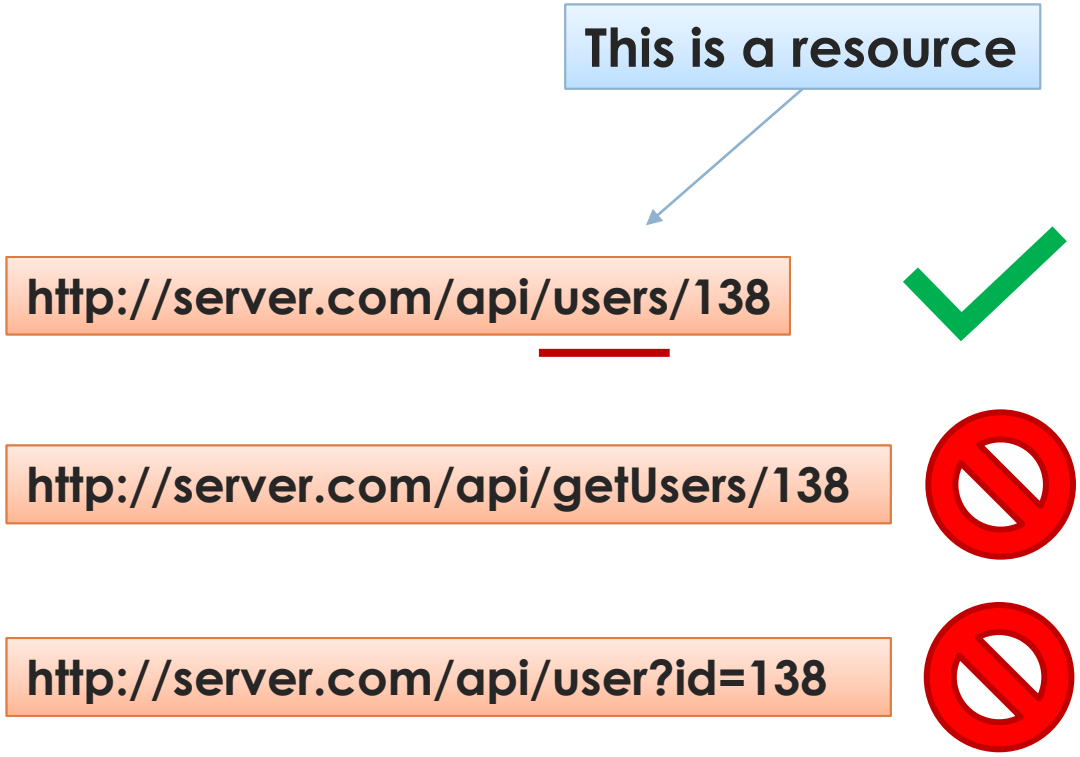
Parts of a URL



It will be important to distinguish between a *path* parameter and a *query string* parameter

RESTful Design Principles

- Design the API around resources (nouns)

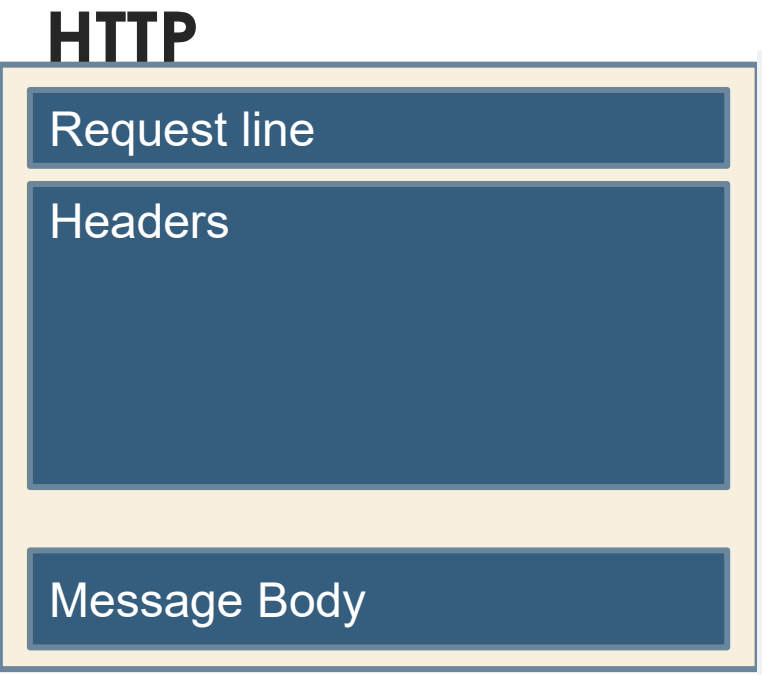


RESTful Design Principles *(continued)*

- Use HTTP methods to define operations

GET /api/invoices	Retrieves all invoices
GET /api/invoices/2952	*Reads* a specific invoice
POST /api/invoices	*Creates* a new invoice
PUT /api/invoices/2952	*Updates* a specific invoice
PATCH /api/invoices/2952	*Partially Updates* a specific invoice
DELETE /api/invoices/2952	*Deletes* a specific invoice

HTTP/HTTPS



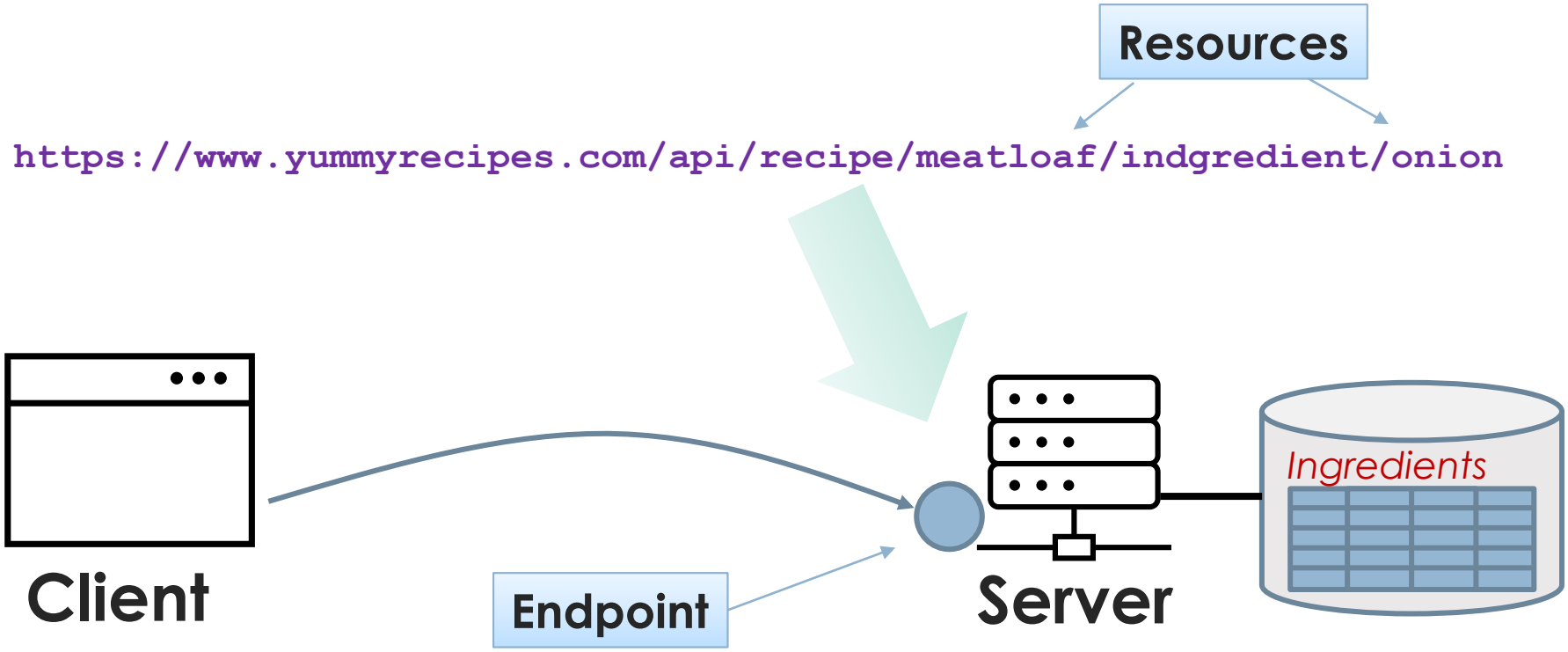

```
POST /process/state HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64)
Host: www.sample.com
Content-Type: application/x-www-form-urlencoded
Content-Length: length
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

state=Colorado&capital=Denver
```

POST, PUT, PATCH, and DELETE will place parameters to be sent in the HTTP message body

Endpoints vs Resources

- Endpoints are *locations* (addresses), often exposing services or resources

- Resources are objects that are returned from a URL



RESTful Python API Frameworks

- Numerous tools exist for developing RESTful applications

Server-side Tools

Django REST

Flask

Flask-RESTful

FastAPI

Falcon

Bottle

Client-side Tools

Requests

Faster-than-requests

PycURL

Introducing Flask

- Flask is one of the top Python tools for building RESTful applications

- Since it is a third-party tool, it must be installed first
  - Use any of these (note: Flask will already be installed for you)

(the 3.10 reference should match your Python version)

pip install Flask
pip3 install Flask
pip3.10 install Flask
python -m pip install Flask

https://flask.palletsprojects.com

Flask Routing Styles

- Flask supports several mapping styles

```
@app.route('/api/state/<st_name>/capital/<cap_name>')

def do_stuff(st_name, cap_name):
    ...
```

```
@app.route('/api/email/<int:id>')
def do_stuff(id):
    ...
```

This type casts the value after /email/ encountered in the URL to an int type

Bringing In the Database

Flask SQLAlchemy

- A commonly used Python tool for interacting with a database is called *SQLAlchemy*

https://flask-sqlalchemy.palletsprojects.com/en/2.x/

- SQLAlchemy performs *automatic Python object-to-relational database mapping* (ORM)

- SQLAlchemy is a third-party tool and must be installed

pip install sqlalchemy

- Flask-SQLAlchemy is a plugin that integrates the two frameworks

pip install flask-sqlalchemy

  - Flask plugins must be installed normally
  - *Our setup already includes both of these*

Configuring SQLAlchemy with Flask

- Three steps to get SQLAlchemy up and running
  1. Configure the Flask-SQLAlchemy plugin (shown below)
  2. Define models to interact with the database
  3. Invoke model methods to perform database operations

- Step 1. Configure the plugin by defining how to connect to the database

```
from flask import Flask, request
from flask_restx import Resource, Api
from flask_sqlalchemy import SQLAlchemy

student_files_dir = Path(__file__).parents[1]
db_file = student_files_dir / 'data/course_data.db'

app = Flask(__name__)
api = Api(app, prefix='/api')

app.config['SQLALCHEMY_DATABASE_URI'] =
                                    'sqlite:///' + str(db_file)

db = SQLAlchemy(app)
```

This is the location of our database

Defining and Using a Model

- Step 2. Create a model class to map to the database

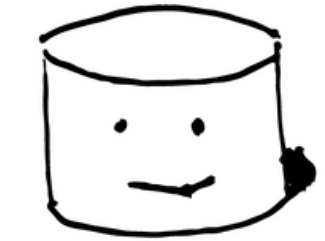
```
app = Flask(__name__)
api = Api(app, prefix='/api')

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///' + str(db_file)
db = SQLAlchemy(app)

class CelebrityModel(db.Model):
    __tablename__ = 'celebrity'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100))
    pay = db.Column(db.Float)
    year = db.Column(db.String(15))
    category = db.Column(db.String(50))

    def __init__(self, name, pay, year, category):
        self.name = name
        self.pay = pay
        self.year = year
        self.category = category

    def __str__(self):
        return f'{self.year} {self.name} {self.pay} {self.category}'
```

The model inherits from db.Model

Define the fields and types to map to the table

The __init__ and __str__ are optional but can be helpful

Flask-Marshmallow



- What is Flask-marshmallow?
  - Flask-marshmallow is a helpful tool for converting objects from Python to JSON or JSON to Python
  - Ideal for returning a Flask database object as JSON from an API

Marshmallow homepage: https://marshmallow.readthedocs.io/en/stable/
Flask-Marshmallow (plugin) home: https://flask-marshmallow.readthedocs.io/en/latest/

- To use Marshmallow:
  1. Configure the plugin within Flask
  2. Define and instantiate a "schema" (the fields to be serialized)
  3. Use the schema to return from API methods (as needed)

Installing Flask-Marshmallow

- Flask-marshmallow is third-party module and must be installed using the appropriate statement

```
pip install flask-marshmallow
pip3 install flask-marshmallow
pip3.10 install flask-marshmallow
python -m pip install flask-marshmallow
```

- When using SQLAlchemy objects with Marshmallow, another plugin, Marshmallow-SQLAlchemy, should also be installed

```
pip install marshmallow-sqlalchemy
pip3 install marshmallow-sqlalchemy
pip3.10 install marshmallow-sqlalchemy
python -m pip install marshmallow-sqlalchemy
```

- *Neither of these need to be installed for our environment*

Integrating Flask-Marshmallow Is Easy

```
from flask import Flask, request
from flask_marshmallow import Marshmallow

app = Flask(__name__)
ma = Marshmallow(app)

class CelebrityModel(db.Model):
    ...
class Celebrities(Resource):
    ...
class Celebrity(Resource):
    ...

class CelebritySchema(ma.Schema):
    class Meta:
        fields = ('id', 'name', 'year', 'pay', 'category')

celebrity_schema = CelebritySchema()
celebrities_schema = CelebritySchema(many=True)
app.run(host='localhost', port=8051)
```

Step 1. Configure the plugin

Step 2. Define and instantiate schema

POST Again with Flask-Marshmallow

```
class Celebrities(Resource):
    def get(self):
        return {'celebrities': ''}

    def post(self):
        name = request.form.get('name')
        year = request.form.get('year')
        category = request.form.get('category')
        pay = float(request.form.get('pay'))

        new_celeb = CelebrityModel(name, pay, year, category)
        db.session.add(new_celeb)
        db.session.commit()


        return celebrity_schema.jsonify(new_celeb)
```

celebrity_schema.jsonify(obj) - returns JSON

celebrity_schema.dump(obj) -- returns a dict

Step 3. The Marshmallow schema definition will now convert your (SQLAlchemy) objects to JSON

Test the POST operation again (this time with Marshmallow included) either using Postman or by running *15_testing_flask_sqlalchemy_marshmallow.py*

GET and GET-all with Flask-SQLAlchemy

```
class Celebrities(Resource):
    def get(self):
        all_celebs = CelebrityModel.query.all()
        return {'results': celebrities_schema.dump(all_celebs)}

    def post(self):
        (unchanged from previous)

class Celebrity(Resource):
    def get(self, name):
        celeb = CelebrityModel.query.get(id)
        return celebrity_schema.jsonify(celeb)

    def delete(self, name):
        return {'action': 'delete'}

    def put(self, name):
        return {'action': 'put'}
```

Performs equivalent to a SELECT * FROM celebrity

Query the database and find the record that matches the specified id primary key

Test the two GET operations either using Postman or by running *17_testing_flask_sqlalchemy_get.py*

API PUT / DELETE with Flask-SQLAlchemy

```
class Celebrity(Resource):
    def get(self, id):
        (as shown previously)

    def delete(self, id):
        celeb = CelebrityModel.query.get(id)
        db.session.delete(celeb)
        db.session.commit()
        return celebrity_schema.jsonify(celeb)

    def put(self, id):
        year = request.form.get('year')
        category = request.form.get('category')
        pay = float(request.form.get('pay'))

        celeb = CelebrityModel.query.get(id)
        celeb.year = year
        celeb.category = category
        celeb.pay = pay

        db.session.commit()
        return celebrity_schema.jsonify(celeb)
```

Using the object id, lookup the object in the database, then use the object to perform a delete

Get the submitted data, instantiate a model, make any changes (e.g., celeb.year = year)

Test the two GET operations either using Postman or by running *19_testing_flask_sqlalchemy_put.py*

Implementing PATCH

- Our PUT operation performs a full update (as much as we are allowed) on the Celebrity object
- But what if we only wanted to update the category and not the pay or year attributes?
- Use PATCH for partial updates

```
def patch(self, id):
    celeb = CelebrityModel.query.get(id)

    if 'year' in request.form:
        celeb.year = request.form.get('year')
    if 'category' in request.form:
        celeb.category = request.form.get('category')
    if 'pay' in request.form:
        celeb.pay = float(request.form.get('pay'))

    db.session.commit()


    return celebrity_schema.jsonify(celeb)
```