HTTP methods and Flask models

Maher Khan

Recitation 8
Date: 2nd November, 2018
slides and codes: https://github.com/maher460/Pitt-CS1520-Recitations
Office hours today from 6:00 PM to 8:00 PM

Table of Contents

HTTP Methods

HTTP Methods

- Most frequently used:
  – GET – retrieve information
  – HEAD – query if there is information to retrieve
  – POST – add or modify a resource's information
  (if the resource identity is not known)
  – PUT – add or replace a resource's information
  (if the resource identity is known (add) or can be predicted
  (replace))
  – DELETE – delete information

HTTP Methods

When choosing which HTTP methods to implement, consider:
– Safety
– Idempotence
– And when/how to use each

- HTML 4 & 5 require only 2: GET and POST
- Therefore, simple HTML form submission only uses GET or POST
- The full set of HTTP methods are available (e.g. by JavaScript), however.

Safe

• A safe request is one that does not change the state of the resource on the server
  – Except for trivial changes such as logging, caching, or incrementing a visit counter.

• E.g. GET should be safe
  – GET should only retrieve information from a source
  – GET should not update that resource

• GET could be used to update a resource, but as the HTTP protocol is defined, it should not be used in that way.

Idempotent

- Multiple identical requests are the same as a single request
  - Analogy in math: taking the absolute value of a number.

- Nice to know on a potentially unreliable Internet:
  - If the first request didn’t work, just try again.
  - If the Google search never returned, just reload

Idempotent Methods

- Safe methods are idempotent
  - Because you are not changing the resource
  - Therefore GET and HEAD are idempotent
- PUT and DELETE are also idempotent
  - PUT adds or wholly replaces a resource
    - Do this a second time identically, you still have the same results.
    - PUT requires knowing how the server will identify the resource
  - DELETE removes a resource
    - DELETE a resource for the second time does not change its state.

POST

- POST is neither safe nor idempotent
  - POST changes the resource
  - If you POST twice, then the results can be different after each change.

- POST should be reserved for modifying or adding to (not replacing), an existing resource.

Summary

Method	Purpose	Safe	Idempotent
GET	Retrieve a resource	Yes	Yes
PUT	Insert or replace a resource	No	Yes
DELETE	Remove a resource	No	Yes
HEAD	Get only header info of a resource	Yes	Yes
POST	Append to or modify a resource	No	No

Flask Models

Flask Models

Simple example:

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)

    def __repr__(self):
        return '<User %r>' % self.username
```

- The baseclass for all your models is called `db.Model`.
- Use **`Column`** to define a column.
- Primary keys are marked with `primary_key=True`
- The types of the column are the first argument to **`Column`**.

Column Types

`Integer`	an integer
`String(size)`	a string with a maximum length (optional in some databases, e.g. PostgreSQL)
`Text`	some longer unicode text
`DateTime`	date and time expressed as Python `datetime` object.
`Float`	stores floating point values
`Boolean`	stores a boolean value
`PickleType`	stores a pickled Python object
`LargeBinary`	stores large arbitrary binary data

One-to-Many

In the following example, a Person object can have multiple Address objects, but any Address object belongs to exactly one Person object.

Example:

```
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    addresses = db.relationship('Address', backref='person', lazy=True)
```

```
class Address(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), nullable=False)
    person_id = db.Column(db.Integer, db.ForeignKey('person.id'),
        nullable=False)
```

- **The “Many” part of the relationship (Address) is going to store the id of the “One” part of the relationship (Person) as a foreign key.**
- Relationships are expressed with the **`relationship()`** function. However the foreign key has to be separately declared with the **`ForeignKey`** class
- **Use** `uselist=False` in the relationship function for one-to-one relationship
- `backref` is a simple way to also declare a new property on the `Address` class. You can then also use `my_address.person` to get to the person at that address. `lazy` defines when SQLAlchemy will load the data from the database

lazy

- ‘select’ / True
- ‘joined’ / False
- ‘subquery’
- ‘dynamic’

lazy

- `'select'` / `True` (which is the default, but explicit is better than implicit) means that SQLAlchemy will load the data as necessary in one go using a standard select statement.

- `'joined'` / `False` tells SQLAlchemy to load the relationship in the same query as the parent using a `JOIN` statement.

- `'subquery'` works like `'joined'` but instead SQLAlchemy will use a subquery.

- `'dynamic'` is special and can be useful if you have many items and always want to apply additional SQL filters to them. Instead of loading the items SQLAlchemy will return another query object which you can further refine before loading the items.

Many-to-many

If you want to use many-to-many relationships you will need to define a helper table that is used for the relationship. In the following example, notice that we set both the columns of the helper table as primary keys, and thus they both act as a single composite key for each entry.

```
tags = db.Table('tags',
    db.Column('tag_id', db.Integer, db.ForeignKey('tag.id'), primary_key=True),
    db.Column('page_id', db.Integer, db.ForeignKey('page.id'), primary_key=True)
)

class Page(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    tags = db.relationship('Tag', secondary=tags, lazy='subquery',
        backref=db.backref('pages', lazy=True))

class Tag(db.Model):
    id = db.Column(db.Integer, primary_key=True)
```

- Here we configured `Page.tags` to be loaded immediately after loading a Page, but using a separate query. This always results in two queries when retrieving a Page, but when querying for multiple pages you will not get additional queries.
- The list of pages for a tag on the other hand is something that's rarely needed. For example, you won't need that list when retrieving the tags for a specific page. Therefore, the backref is set to be lazy-loaded so that accessing it for the first time will trigger a query to get the list of pages for that tag.
- If you need to apply further query options on that list, you could either switch to the `'dynamic'` strategy

Thanks!

Questions?