Last Updated on 2023/02/28

Databases in Applications

Software Architecture

Teacher Version

March 6, 2023

Brae Webb & Evan Hughes

Figure 1: A map of data storage techniques from Designing Data-Intensive Applications [1].

1 This Week

This week our goal is to:

- explore the various techniques developers use to store data; and
- upgrade our todo application to use a local sqlite relational database.

2 Databases and Data Models

Unfortunately, to build interesting software we often need to store and use data. The storage of data introduces a number of challenges when designing, creating, and maintaining our software. However, not all data storage techniques are created equal; the choice of data storage model can have a profound impact on our software's complexity and maintainability. In this practical, we want to take a superficial exploration of our island of data storage models. For a more in-depth treatment of data storage models that is outside the scope of this course, see Chapter 2 of the *Designing Data-Intensive Applications* book [1].

> For the teacher
>
> Discuss the following different storage technologies and mention some use cases of when you would choose each one. Discuss some popular implementations of each.
>
> Aim for no more than 30 minutes of discussion.

2.1 Relational Storage

Relational databases are what you have been exposed to the most in your University career — think MySQL, Postgres, Oracle DB, etc. This type of database is good at modelling the real world which is often a highly connected environment.

Some popular offerings are below:

- MySQL/MariaDB [Amazon RDS / Amazon Aurora].
- Postgres [Amazon RDS / Amazon Aurora].
- SQLite.

The AWS offerings of these services come in two different types, we have the traditional approach of server capacity (x cores, y ram) and we have a server-less approach. The server-less approach is a more dynamic database that can scale to large amounts of load when needed though at a cost per request.

2.1.1 ORM

Object Relational Mapping (ORM) is a fairly common tool for programmers to use to make developing with databases smoother. One fairly prevalent example of this is SQLAlchemy which is a very widely used database abstraction for python. SQLAlchemy allows us to move to a higher level of abstraction than SQL queries and perform database actions using standard python code.

The benefits of ORMs are the ability to model database objects in our existing programming language instead of having large blocks of SQL text within our source code. The disadvantages come in when we need to do specific SQL work or where the abstractions cost is greater than the benefits.

2.2 Wide-Column Storage

> For the teacher
>
> Examples of big apps that depend on this technology is Netflix `https://netflixtechblog.com/netflixs-viewing-data-how-we-know-where-you-are-in-house-of-cards-608dd61077da`.

Wide-Column databases are a form of NoSQL or non-relational data stores. In these data stores the data model design is focused more on having efficient queries at the cost of data duplication. A warning to the reader that these models are not flexible after creation, it is much easier to answer a new use case in a relational model.

- Apache Cassandra [Amazon Keyspaces for Cassandra].
- Apache HBase.

2.3 Key-Value Storage

Key-Value stores are very popular for cache or remote config use cases, some of the most notable are Redis and Memcached. These stores allow efficient lookup of values via keys and are usually stored in-memory.

- Redis [Amazon ElastiCache for Redis].
- Memcached [Amazon ElastiCache for Memcached].
- Amazon DynamoDB.
- Amazon MemoryDB for Redis.

2.4 Time Series Storage

For the teacher

Something to mention here is that relations are usually not utilised between tables in time series databases.

Time series databases are highly focused storage which is tailored to retrieving results by timestamp ranges. Many implementations also take advantage of the data model to allow efficient rollover of data and partitioning. One of the most popular time series databases is Prometheus which is used to store monitoring metrics.

- Amazon Timestream.
- TimescaleDB (Postgres + Addon).
- Prometheus.
- InfluxDB.
- PostgreSQL

2.5 Document Storage

Document databases are a subset of NoSQL databases with a focus on a flexible data model. MongoDB for instance allows the user to store JSON documents and perform queries on those documents. One advantage of document databases is that they match a programmers existing mental model of storing data in formats such as JSON.

- MongoDB.
- Apache CouchDB.
- Amazon DocumentDB.
- Amazon DynamoDB.

2.6 Graph Storage

For the teacher

If you haven't experienced graph databases, a good usecase is "recommendation systems", which use the connected nature of items to figure out what to suggest to a person. Another example is the https://neo4j.com/blog/analyzing-panama-papers-neo4j/ Panama Papers.

Graph Databases are relational storage with a few enhancements to allow fast neighbour look-ups. These databases also allow the implementation of graph algorithms to query data.

- Amazon Neptune.
- Neo4J.
- Janus Graph.

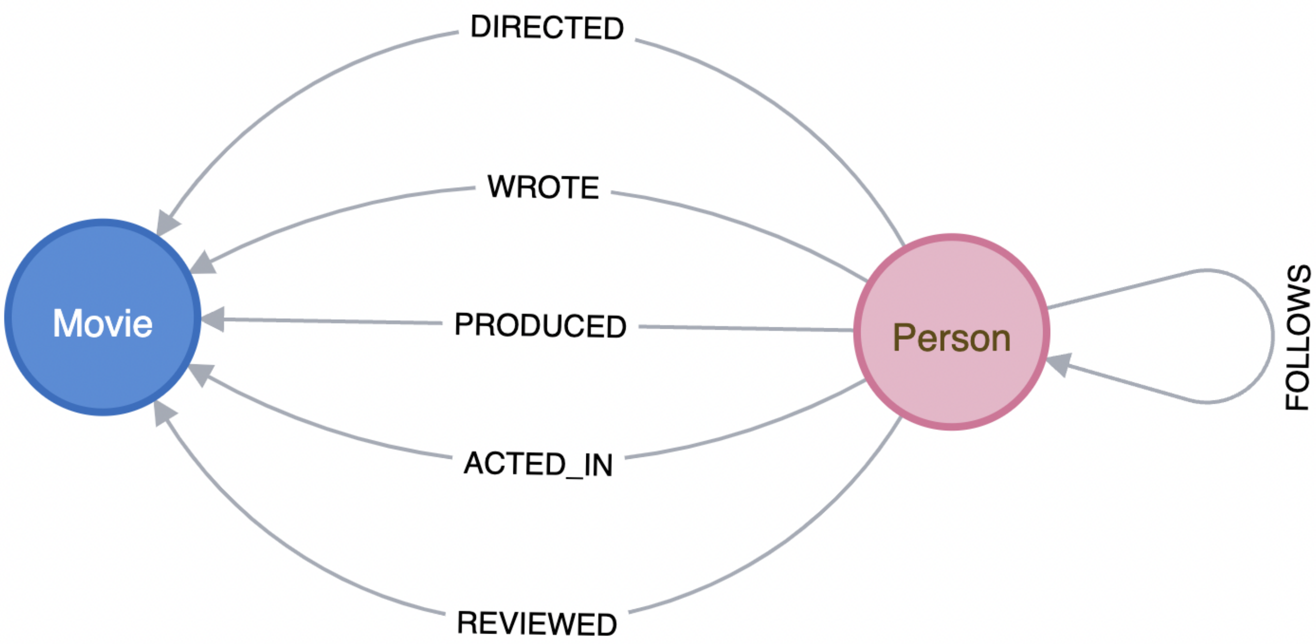

Figure 2: Graph Database Example from the Neo4J documentation.

3 Enhancing the Todo App with Storage

Last week we created a simple web server that can listen on a port and respond to HTTP requests. The endpoints that we created are all stubs the return a hardcoded JSON response. This week we will add a database to our application to support persistent storage.

3.1 Creating a Practical Repository

Navigate to the GitHub Classroom link for this practical provided by your tutor in Slack. As with last week, this will create a new repository for you in the course organisation. You can now clone this repository to your local machine or work directly in the browser with GitHub codespaces. This repository will be populated with our solution to last weeks practical exercise. You may modify this solution or replace it with your own.

3.2 Installing the Database Dependencies

We will be using a Python library called SQLAlchemy to interact with our database. This library abstracts the SQL queries away and allows us to interact with the database using Python objects. We will be using SQLAlchemy using a package called Flask-SQLAlchemy, a wrapper around SQLAlchemy, that is designed to work with Flask our WebServer library.

```
$ pipenv install flask-sqlalchemy
```

3.3 Creating the Database and Models

We will be using a database called SQLite for this practical. SQLite is a file-based database which is easy to setup and use. As the database is isolated to a file, SQLite is a good choice for initial development.

Now that our dependencies are installed, navigate to the cloned practical directory and create a new folder within the todo folder called `models`. Inside this folder create a new file called `todo.py` and a new file called `__init__.py`.

Inside the `__init__.py` file we will add the following code:

```python
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()
```

All this file does is setup a new SQLAlchemy object which we will use to interact with our database. In the `todo.py` file we will add the following code:

```python
import datetime
from . import db

# Inheriting from db.Model tells SQLAlchemy that this class is a model
class Todo(db.Model):
    # Sets the tables name from the default
    __tablename__ = 'todos'

    # This is how we define a column, this is also the primary key
    id = db.Column(db.Integer, primary_key=True)
    # This is a manadatory column of 80 characters
    title = db.Column(db.String(80), nullable=False)
    # This is an optional column of 120 characters
    description = db.Column(db.String(120), nullable=True)
    # This column has a default value of False
    completed = db.Column(db.Boolean, nullable=False, default=False)
    deadline_at = db.Column(db.DateTime, nullable=True)
    # This column has a default value which is a function call
    created_at = db.Column(db.DateTime, nullable=False, default=datetime.datetime.
        utcnow)
    # This column has a default value which is a function call and also updates on
        update
```

```
    updated_at = db.Column(db.DateTime, nullable=False, default=datetime.datetime.
        utcnow, onupdate=datetime.datetime.utcnow)

    # This is a helper method to convert the model to a dictionary
    def to_dict(self):
        return {
            'id': self.id,
            'title': self.title,
            'description': self.description,
            'completed': self.completed,
            'deadline_at': self.deadline_at.isoformat() if self.deadline_at else
                None,
            'created_at': self.created_at.isoformat() if self.created_at else None,
            'updated_at': self.updated_at.isoformat() if self.updated_at else None,
        }

    # String representation of our model
    def __repr__(self):
        return f'<Todo {self.id} {self.title}>'
```

The above code is doing a lot of the heavy lifting for us in our database table generation, have a look at the comments above to see what each line is doing.

3.4 Configuring the Database

Now that we have defined our database schema using an ORM, we need to configure our application to use the database. Open the `todo/__init__.py` file and change the code to the following:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

def create_app():
    app = Flask(__name__)

    app.config['SQLALCHEMY_DATABASE_URI'] = "sqlite:///db.sqlite"

    # Load the models
    from todo.models import db
    from todo.models.todo import Todo
    db.init_app(app)

    # Create the database tables
    with app.app_context():
      db.create_all()
      db.session.commit()

    # Register the blueprints
    from todo.views.routes import api
    app.register_blueprint(api)
```

```
    return app
```

In the above we set a default location, defined as a URI, for our database. As we using SQLite we can set the database to be a file on the file system. In production systems, you would set this to a URI that defines the credentials and hostname of your database server. In latter practicals, we will get experience using more complex URIs.

If we run our application now, we will see that the database file has been created for us. Your project structure should have a least the follow structure (there may be additional files):

```
-- README.md
-- Pipfile
-- Pipfile.lock
-- endpoints.http
-- instance
  | -- db.sqlite
-- todo
  | -- __init__.py
  | -- views
      | -- routes.py
  | -- models
      | -- __init__.py
      | -- todo.py
```

3.5 Inspecting the Database

We can use the `sqlite3` command line tool to inspect the database. Open a terminal and navigate to the root of your project. Then run the following command:

```
$ sqlite3 instance/db.sqlite
```

Info

Most platforms including MacOS, WSL2, and most Linux environments come with SQLite installed. If you get an error running the above command, you may have to install SQLite.

This will open the SQLite command line tool and connect to the database file. We can then run the following command to see the tables in our database:

```
.tables
```

This will show us the tables in our database. We can then run the following command to see the schema of our table:

```
.schema todos
```

This will show us the columns in our table. We can then run the following command to see the data in our table:

```sql
SELECT * FROM todos;
```

This should initially show us no rows. We can then run the following command to exit the SQLite command line tool:

```
.exit
```

You should notice that our table is called todos and not todo because we specified todos with the `__tablename__` attribute.

3.6 Using the Database

Now that we have a database intergrated into our application, we will modify our endpoints to take advantage of it.

Open the `todo/views/routes.py` file and add the following imports to the top of the file:

```python
from flask import Blueprint, jsonify, request
from todo.models import db
from todo.models.todo import Todo
```

Open the `todo/views/routes.py` file and change the `get_todos` endpoint to the following:

```python
@api.route('/todos', methods=['GET'])
def get_todos():
    todos = Todo.query.all()
    result = []
    for todo in todos:
        result.append(todo.to_dict())
    return jsonify(result)
```

This will query the database for all the todos and return them as JSON. We can then change the `get_todo` endpoint to the following:

```python
@api.route('/todos/<int:id>', methods=['GET'])
def get_todo(id):
    todo = Todo.query.get(id)
    if todo is None:
        return jsonify({'error': 'Todo not found'}), 404
    return jsonify(todo.to_dict())
```

Now we have modified these endpoints to use the database — let's test that our application is still functioning correctly. Restart your webserver and navigate to the `/api/v1/todos` endpoint. You should see the following JSON response:

```
[]
```

Of course our API does not have any todo items in it yet. We will now add the ability to insert todo items to our database. Open the `todo/views/routes.py` file and change the `create_todo` endpoint to the following:

```python
@api.route('/todos', methods=['POST'])
def create_todo():
    todo = Todo(
        title=request.json.get('title'),
        description=request.json.get('description'),
        completed=request.json.get('completed', False),
        deadline_at=request.json.get('deadline_at')
    )
    # Adds a new record to the database or will update an existing record
    db.session.add(todo)
    # Commits the changes to the database, this must be called for the changes to
       be saved
    db.session.commit()
    return jsonify(todo.to_dict()), 201
```

This endpoint now allows us create a todo item in the database. Test this endpoint by going to endpoints.http (or your API query tool of choice) and running the POST request. You should see the following response (with different created_at and updated_at values):

```json
{
  "id": 1,
  "title": "Test Todo",
  "description": "This is a test todo",
  "completed": false,
  "deadline_at": null,
  "created_at": "2023-02-27T12:00:00.000000Z",
  "updated_at": "2023-02-27T12:00:00.000000Z"
}
```

Now if we go to our `/api/v1/todos` endpoint we should see the todo item we just created:

```json
[
  {
    "id": 1,
    "title": "Test Todo",
    "description": "This is a test todo",
    "completed": false,
    "deadline_at": null,
    "created_at": "2023-02-27T12:00:00.000000Z",
    "updated_at": "2023-02-27T12:00:00.000000Z"
```

```
  }
]
```

Now let's add the remaining endpoints. Change the `update_todo` endpoint to the following:

```
@api.route('/todos/<int:id>', methods=['PUT'])
def update_todo(id):
    todo = Todo.query.get(id)
    if todo is None:
        return jsonify({'error': 'Todo not found'}), 404
    todo.title = request.json.get('title', todo.title)
    todo.description = request.json.get('description', todo.description)
    todo.completed = request.json.get('completed', todo.completed)
    todo.deadline_at = request.json.get('deadline_at', todo.deadline_at)
    db.session.commit()
    return jsonify(todo.to_dict())
```

This endpoint will update a todo item in the database. Let's test this endpoint by going to our endpoints.http and running the PUT request. You should see the following response:

```
{
  "id": 1,
  "title": "Updated Test Todo",
  "description": "This is an updated test todo",
  "completed": false,
  "deadline_at": null,
  "created_at": "2023-02-27T12:00:00.000000Z",
  "updated_at": "2023-02-27T12:00:00.000000Z"
}
```

To implement delete functionalitty, we will use the HTTP DELETE method and the delete .method of the database session. Open the `todo/views/routes.py` file and change the `delete_todo` endpoint to the following:

```
@api.route('/todos/<int:id>', methods=['DELETE'])
def delete_todo(id):
    todo = Todo.query.get(id)
    if todo is None:
        return jsonify({}), 200

    db.session.delete(todo)
    db.session.commit()
    return jsonify(todo.to_dict()), 200
```

We now have a set of endpoints that can perform the CRUD operations of our API but some functionality is missing. We are gonna add that functionality after setting up some tests to help us do **Test Driven Development**.

4 Testing the API

4.1 Setting up the testing environment

In the project, you will have a `tests` folder. This contains `test_todo.py` which has a range of provided tests for the todo endpoints. However, we need to setup a few components to make it work.

Inside the `tests` folder create a `base.py` file. Inside the `base.py` file add the following code:

```python
from todo import create_app
import unittest


class TodoTest(unittest.TestCase):
    def setUp(self):
        self.app = create_app(config_overrides={
            'SQLALCHEMY_DATABASE_URI': 'sqlite:///:memory:',
            'TESTING': True
        })

        with self.app.app_context():
            from todo.models import db
            db.create_all()
            db.session.commit()

        self.client = self.app.test_client()

    def assertDictSubset(self, expected_subset: dict, whole: dict):
        for key, value in expected_subset.items():
            self.assertEqual(whole[key], value)
```

This base class is what we will use to help setup our tests and provide an assertion helper method. The `setUp` method is called before each test and is used to initialise the in-memory database. The `assertDictSubset` method is a helper method that we will use to compare the todo items we get from the API with the todo items we expect to get from the API.

As you can see we use a slight modification to the `create_app` function. We are passing in a dictionary of config overrides. This allows us to override the config values for the testing environment.

4.2 Prepping the config for testing

Open the `todo/__init__.py` file and adjust the `create_app` function to the following:

```python
def create_app(config_overrides=None):
    app = Flask(__name__)

    app.config['SQLALCHEMY_DATABASE_URI'] = "sqlite:///db.sqlite"
    if config_overrides:
        app.config.update(config_overrides)

    # Load the models
```

```
    from todo.models import db
    from todo.models.todo import Todo
    db.init_app(app)

    # Create the database tables
    with app.app_context():
        db.create_all()
        db.session.commit()

    # Register the blueprints
    from .views.routes import api
    app.register_blueprint(api)

    return app
```

4.3 Writing our first tests

Now we will write our first tests. Open the `tests` folder and create a `test_health.py` file. Add the following code:

```
from tests.base import TodoTest


class TestHealth(TodoTest):
    def test_health(self):
        response = self.client.get('/api/v1/health')
        self.assertEqual(response.status_code, 200)
        self.assertEqual(response.json, {'status': 'ok'})
```

This test will make a GET request to the `/api/v1/health` endpoint and check

- that the response is a 200 status code; and
- that the response is a JSON object with the key `status` and the value `ok`.

To run this test stop your server, if it is running, and run the following command:

```
$ pipenv run python3 -m unittest tests/test_health.py
```

You should see the following output:

```
>> pipenv run python3 -m unittest tests/test_health.py
....
----------------------------------------------------------------------
Ran 1 tests in 0.011s

OK
```

4.4 Test Driven Development

Now you've created a test for the health endpoint and run it. To run the suite of provided tests, use the following command:

```
$ pipenv run python3 -m unittest discover -s tests
```

If you've used only the code we've provided, you should see some of the tests fail.

The tests are based on the specification from week one, we would like you to change your API to make the tests pass.

To check if the request is a JSON request you can use the `request.is_json` method.

If you get stuck feel free to ask your peers / staff for help.

5 Finishing Up

We now have a working API which we can use to create, read, update, and delete todo items. We can also use the API to:

- mark todo items as completed;
- filter todo items by whether they are completed or not; and
- filter todo items by whether they are within the next N days.

Next week we will dockerise our API and use docker-compose to run our API and a database within containers.

References

[1] M. Kleppmann, *Designing Data-Intensive Applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media, Inc., March 2017.