Databases in Applications

Software Architecture

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].

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

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

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

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.
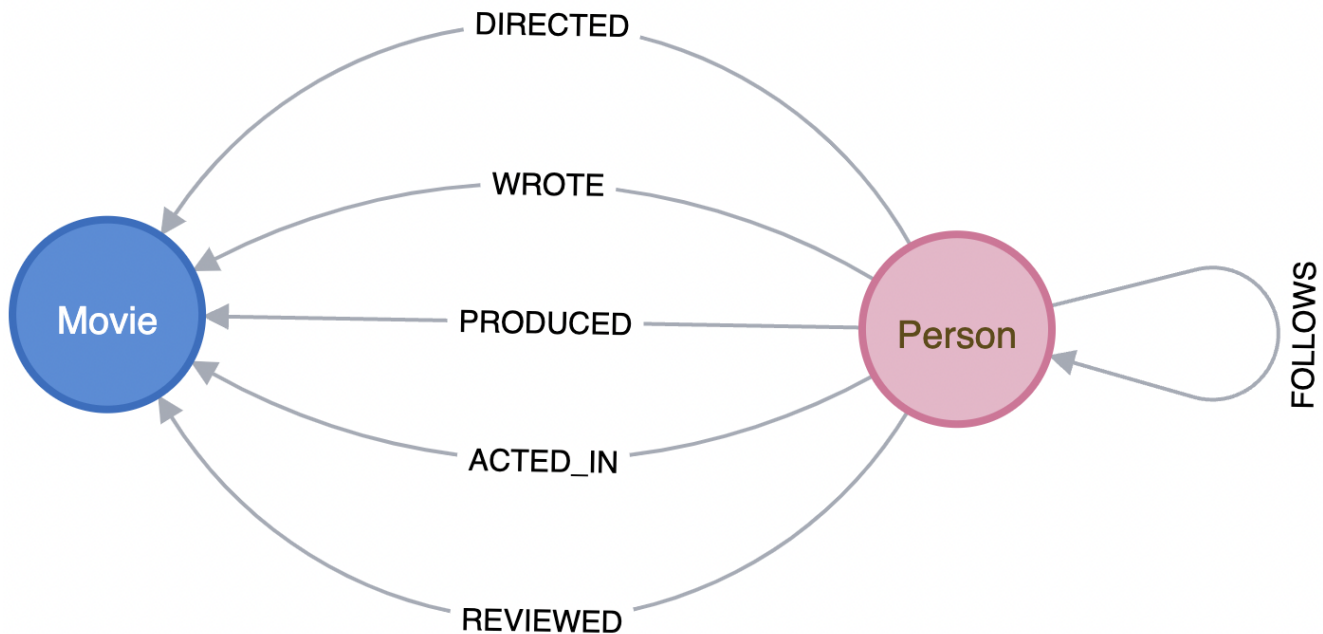
- Amazon Neptune.

- 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 simple JSON response. This week we will be adding a database to our application to store the todo items.

3.1 Creating a Practical Repository

Navigate to the GitHub Classroom link provided by your tutor. You should see a list of practicals, click on the week one practical. 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.

> **Info**
>
> If using Github CodeSpaces the environment will be pre-installed with the libraries you used last week. You will still need to follow on with the instructions below to install the database libraries.

3.2 Installing the Database Dependencies

We will be using a python library called SQLAlchemy to interact with our database. This library abstracts the low level SQL queries away from us and allows us to interact with the database using python objects. We will be using SQLAlchemy using a packaged called Flask-SQLAlchemy which is 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 very easy to setup and use. It is also a good choice for development as it can be blown away by simply deleting a file or by residing in memory.

    Now that our dependencies are installed, lets start by navigating to our new cloned practical folder and creating 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 models import db

# Inheriting from db.Model tells SQLAlchemy that this class is a model
class Todo(db.Model):

    # 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 table, 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
    db.init_app(app)

    # Create the database tables
    db.create_all()

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

    return app
```

In the above we set a default location for our database. Cause we using sqlite we can set the database to be a file on the file system. In most production cases you will be setting this to a URI that defines the credentials and hostname of your database server. In latter practicals we will be exploring this more but for now we will be using a file based database.

If we run our application now, we will see that the database file has been created for us.

TODO: Add a screenshot of the database file being created.

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 db.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 columns in 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:

```
SELECT * FROM todos;
```

This will show us the data in our table. We can then run the following command to exit the SQLite command line tool:

```
.exit
```

You may have noticed that our table is called todos and not todo. This is because SQLAlchemy will automatically pluralise the name of the table for us. This is a common behaviour you will see many database libraries.

3.6   Using the Database

Now that we have a database intergrated into our application, lets 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 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 make_response(jsonify({'error': 'Todo not found'}, 404)
    return make_response(jsonify(todo.to_dict()))
```

So we have upgraded these endpoints to use the database. Lets test that our application is still functioning correctly. Restart your webserver and navigate to the `/todos` endpoint. You should see the following JSON response:

```
[]
```

Of course our API does not have any todo items in it yet. Lets enable the ability to add some 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 make_response(jsonify(todo.to_dict()), 201)
```

This endpoint now lets us create a todo item in the database. Lets test this endpoint by going to our endpoints.http and running the POST request. You should see the following response:

```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. You should see the following response:

```
[
  {
    "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 lets add the remaining endpoints. Change the `update_todo` endpoint to the following:

```python
@api.route('/todos/<int:id>', methods=['PUT'])
def update_todo(id):
    todo = Todo.query.get(id)
    if todo is None:
        return make_response(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 make_response(jsonify(todo.to_dict()))
```

This endpoint will update a todo item in the database. Lets 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 do the delete method we use the delete method on the database session instead of the add method. 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 make_response(jsonify(), 200)

    db.session.delete(todo)
    db.session.commit()
    return make_response(jsonify({'message': 'Todo deleted'}), 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 writing some tests to help us do **Test Driven Development**.

4   Testing the API

4.1   Setting up the testing environment

In the root folder of this practical make a `tests` folder. Inside the `tests` folder create a `base.py` file. Inside the `base.py` file add the following code:

```
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 assertTodoPartialEqual(self, todo: dict, expected: dict):
        for key, value in expected.items():
            self.assertEqual(todo[key], value)
```

This base class is what we will use to help setup our tests and provide some helper methods. The `setUp` method is called before each test and is used to get the in-memory database initialised. The `assertTodoPartialEqual` 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 have 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.

10

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
    db.init_app(app)

    # Create the database tables
    db.create_all()

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

    return app
```

4.3   Writing our first tests

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

```python
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 `/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 discover -s tests
```

You should see the following output:

```
$ pipenv run python3 -m unittest discover -s tests
....
----------------------------------------------------------------------
Ran 1 tests in 0.011s

OK
```

One endpoint down, lets write a test for post todo endpoint. Open the `tests` folder and create a `test_todos.py` file. Inside the `test_todos.py` file add the following code:

```
from tests.base import TodoTest


class TestTodos(TodoTest):
    def test_create_todo(self):
        response = self.client.post('/api/v1/todos', json={
            'title': 'Test Todo',
            'description': 'Test Description',
        })
        self.assertEqual(response.status_code, 201)
        self.assertTodoPartialEqual(response.json, {
            'title': 'Test Todo',
            'description': 'Test Description',
        })
```

Now when we run our tests we should see the following output:

```
$ pipenv run python3 -m unittest discover -s tests
....
----------------------------------------------------------------------
Ran 2 tests in 0.012s

OK
```

4.4   Test Driven Development

We have provided a selection of tests in the github repository which you can now copy into your project. These tests are made from the specification, we would like you to change your API to make the tests pass. To give you a hand here are some hints:

To check if the request is a JSON request you can use the following code:

```
if not request.is_json:
    return make_response(jsonify({'error': 'invalid content type'}), 400)
```

If you get stuck feel free to have a look at the solution in the github repository or 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

- filter todo items by whether they are due before or after a particular date

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

References

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