Programmable Web Project

Exercise 1

Introduction to Web Development

Iván Sánchez Milara
Marta Cortés
Mika Oja

OULUN YLIOPISTO

Learning outcomes (I)

- ~~Students understand what a Web API is and learn different Web API architectures.~~
- Students understand the concept of hypermedia and how it can be used to build Web APIs.
- Students are able to design and implement a Web API following REST architectural style principles using existing web frameworks.

Iván Sánchez Milara
Marta Cortés
Mika Oja

Learning outcomes (II)

- Students are able to write unit and functional tests to inspect their APIS.
- Students are able to document their Web APIs using adequate software tools.
- Students are able to implement simple software applications that make use of the APIs.

Iván Sánchez Milara
Marta Cortés
Mika Oja

WEB FRAMEWORKS

Iván Sánchez Milara
Marta Cortés
Mika Oja

Web Frameworks

- Help us to focus in the ”**Business Logic”** of applications
  - Hides HTTP protocol
  - Serialization + parsing of resources representation
  - Security
  - Authentication and authorization

Iván Sánchez Milara
Marta Cortés
Mika Oja

Flask

- Flask is a micro web development framework for Python
  - Simple but extensible core
    - Hooks and signals
  - Support for ORM, Validation, Open authentication through extensions
  - Permits the creation of web applications with no configuration or setup.
    - A complete application may run in one python module.

```
from flask import Flask
app = Flask(__name__)
@app.route("/hello/<name>")
def index(name):
    return “Hello {}“.format(name), 200
```

```
>> flask.run()
```

Flask

`flask.request` is a globally accessible variable which contains HTTP request information.

ATTRIBUTE / METHOD	DESCRIPTION
`.data`	A string containing the request boty
`.args`	A dictionary with the URL query parameters
`.headers`	A dictionary with HTTP request headers.
`.remote_addr`	The ip address of the incoming request
`.json`	Parses the incoming JSON request into python dictionary. This is None if the data is not correctly well-formed, is not JSON, or does not have the Content-Type: application/json

DATABASES AND ORM

Iván Sánchez Milara
Marta Cortés
Mika Oja

Relational Database

- A database:
  - is a data structure
  - stores organized information.
  - can be easily accessed, managed and updated.

- Relational database:
  - data is organized in tables
  - tables are related among each other.

- The structure of a database (tables, fields, relationships...) is called the **database schema**.

Basic Vocabulary in RBD (I)

- Data is stored in tables (or relations)
  - A table is formed by:
    - rows (or records, or tuples)
    - columns (or attributes)
- **Relational Model** implies each row in a table must be unique
- **Primary Keys (PK)** guarantee that Uniqueness:
  - Fields defined as PK are set to be unique and identify each row.
  - Each table can have one PK
- **Foreign Key:** referential constraint between two tables
  - For each record in the **child** table there must be a unique record that matches the key in the **parent** table

Example 1: PK for *users*

id	firstname	lastname	mobile	residence	nickname
1	Jane	Thomas	334455	New York	lucy
2	John	Smith	223355	London	smith

- Possible candidate keys:
- i) id
- ii) (lastname + firstname)
- iii) nickname
- iv) (id, nickname)
- v)(residence + mobile).

Example 2: FK

messages Table

id	title	body	user_id
1	aaa	abc	2

[Child table]

users Table

id	firstname	lastname	nickname
1	Jane	Thomas	lucy
2	John	Smith	smith

[Parent table]

```
Foreign Key (user_id) REFERENCES users.id
```

For each row in *messages* table, it must exist a row in *users* table where

messages.user_id = users.id

Iván Sánchez Milara
Marta Cortés
Mika Oja

Types of relations
One-to-many

many-to-one:
many (pets) to
one (person)

one-to-many:
one (person) to
many (pets)

PET			
id	name	species	owner
1	wanwan	doge	1
2	nyanyan	neko	1

PERSON		
id	name	job
1	winnie	the pooh
2	eeyore	donkey

values in this column
are limited to values
in person.id

Types of relations
One-to-one

one-to-one:
one (image) to
one (fileinfo)

IMAGE			
id	handle	alt	file
1	doge	this dog	1
2	neko	this cat	2

FILEINFO		
id	filename	size
1	dog.png	4096
2	cat.png	2048

values in this column are limited to values in fileinfo.id and are unique

Iván Sánchez Milara
Marta Cortés
Mika Oja

OULUN YLIOPISTO

Types of relations Many-to-Many

ENROLLMENT

class_id	student_id
1	1
1	2

these columns form the PK together; each pair of values is unique

values in this column are limited to values in class.id

values in this column are limited to values in student.id

CLASS

id	name	code
1	internet 101	1337
2	donkeys 101	4451

many-to-many: many (classes) to many (students)

STUDENT

id	name	major
1	winnie	gastronomy
2	eeyore	apathy

Basic Vocabulary in RDB (ii)

- Foreign key **ON DELETE** and **ON UPDATE** clauses are used to configure actions that take place when
  - deleting rows from the parent table
  - modifying the parent key values of existing rows
- E.g. in an *SQLite* database:
  - **SET NULL**: when a parent key is deleted or modified, the child key columns of all rows in the child table that mapped to the parent key are set to contain SQL NULL values.
  - **CASCADE**: A "CASCADE" action propagates the delete or update operation on the parent key to each dependent child key

SQLite

- Lightweight database implementation that support SQL
  - Database is stored in one file
- Only supports five **data types**:
  - null, integer, real, text and blob
  - Type affinity
- **ALTER:** SQLite allows just a subset for ALTER TABLE
  - You can rename a table or add a new column to an existing table (with no constraints).
  - You cannot add or remove constraints after creating the table
- **GRANT and REVOKE** are not supported:
  - SQLite databases are files
  - Thus, file access permission should be used instead
- **PRAGMA command**: SQL extension specific to SQLite
  - PRAGMA FK
- **Foreing Keys** support implicitly
  - you need to execute **PRAGMA foreign_keys = ON** always before any statement or transaction.

ORM with SQLAlchemy

- **Object relational mapping**
- Abstraction layer of a database using models
  - Properties -> database attributes
  - Methods -> SQL operations

- Initialization

```
app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///test.db"
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
db = SQLAlchemy(app)
```

- Model generation

```
class Measurement(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    sensor = db.Column(db.String(20), nullable=False)
    value = db.Column(db.Float, nullable=False)
    time = db.Column(db.DateTime, nullable=False)
```

Iván Sánchez Milara
Marta Cortés
Mika Oja

ORM with SQLAlchemy

- Schema generation

```
db.create_all()
```

- Add objects

```
db.session.add(meas)
db.session.commit()
```

- Retrieving objects

```
measurements = Measurement.query.all()
meas = Measurement.query.first()
measurements = measurement.query.filter_by(sensor="d").all()
meas2 = measurement.query.filter(Measurement.value>100).first()
```

- Removing and modifying objects

```
db.session.delete(meas2)
meas.sensor = ‘donkeysensor1’
db.session.commit()
```

ORM with SQLAlchemy

- One-to-many relations: sensor has multiple measurements

```
class Sensor(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(32), nullable=False, unique=True)
    model = db.Column(db.String(128), nullable=False)
    measurements = db.relationship("Measurement", back_populates="sensor")

class Measurement(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    sensor_id = db.Column(db.Integer, db.ForeignKey("sensor.id", 
                                                       onDelete="CASCADE"))
    value = db.Column(db.Float, nullable=False)
    sensor = db.relationship("Sensor", back_populates="measurements")
```

- DELETE on CASCADE must be also informed to SQLAlchemy

```
 measurements = db.relationship("Measurement", cascade="all, delete-orphan
", back_populates="sensor")
```

ORM with SQLAlchemy

- One-to-one relation: each location can hold only one sensor

```
class Location(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    latitude = db.Column(db.Float, nullable=True)
    longitude = db.Column(db.Float, nullable=True)
    altitude = db.Column(db.Float, nullable=True)
    description=db.Column(db.String(256), nullable=True)
    sensor = db.relationship("Sensor", back_populates="location", uselist=
False)

class Sensor(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(32), nullable=False, unique=True)
    model = db.Column(db.String(128), nullable=False)
    location_id = db.Column(db.Integer, db.ForeignKey("location.id"), on_d
elete="SET NULL“, unique=True)
    location = db.relationship("Location", back_populates="sensor")
```

ORM with SQLAlchemy

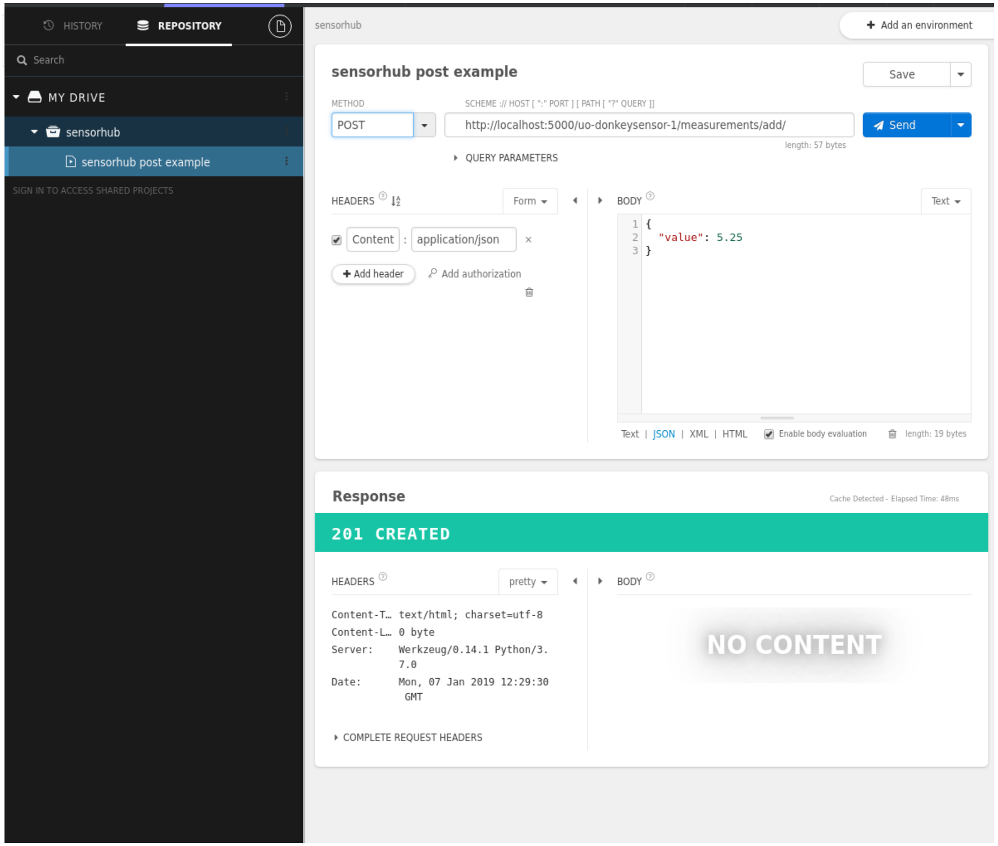
- Many-to-Many relation: one sensor may be in multiple deployments and one deployment has multiple sensors

```
class Deployment(db.Model):
          id = db.Column(db.Integer, primary_key=True)
          start = db.Column(db.DateTime, nullable=False)
          end = db.Column(db.DateTime, nullable=False)
          name = db.Column(db.String(128), nullable=False)
          sensors = db.relationship("Sensor", secondary=deployments, back_populates="deploym
ents")


class Sensor(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(32), nullable=False, unique=True)
    model = db.Column(db.String(128), nullable=False)
    location_id = db.Column(db.Integer, db.ForeignKey("location.id"), on_delete="SET NULL",
unique=True)
    deployments = db.relationship("Deployment", secondary=deployments, back_populates="senso
rs")



deployments = db.Table("deployments",
          db.Column("deployment_id", db.Integer, db.ForeignKey("deployment.id"), primary_key
=True),
          db.Column("sensor_id", db.Integer, db.ForeignKey("sensor.id"), primary_key=True))
```

TESTING

Iván Sánchez Milara
Marta Cortés
Mika Oja

Talend API



Iván Sánchez Milara
Marta Cortés
Mika Oja

Unit Testing

- Unit testing is a process for which small pieces of codes are tested isolating them from the rest of the code.

- **Purpose:** ensure that individual components of the program behave as they are expected to.

- **What to test in databases:**
  - Instances can be created, retrieved, modified and deleted
  - foreign key relationships are created correctly
  - foreign key relationships works as expected (integrity, ondelete, oncascade)
  - Uniqueness, nullability and restrictions works

Testing with pytest

```
import pytest

@pytest.fixture
def my_fixture():
    # do preparations here; eg. Create and populate the database
    yield db
    # teardown: clean the database and resources

def test_something(my_fixture):
    # do some testing
    # you can use the database object defined in my fixture
```

```
assert db_measurement.sensor == db_sensor
```

```
with pytest.raises(IntegrityError):
        db_handle.session.commit()
```

```
>> pytest test.py
```

Iván Sánchez Milara
Marta Cortés
Mika Oja

Environmental Variables

DEBUGGING MODE

- In Linux:

```
export FLASK_ENV=development
```

- In Windows command prompt:

```
set FLASK_ENV=development
```

- You can also use environment variables to run your app from a Python file that is not named app.py:

```
export FLASK_APP=sensorhub.py
```

Iván Sánchez Milara
Marta Cortés
Mika Oja