DA: Development Environment (Backend)

Applications for mobile devices - Theory - Unit 5

Didac Florensa Cazorla

Any: 2021-2022
Curs: 102386
Institut: University of Lleida (Campus Igualada)
Titulació: Bachelor's degree in Digital Interaction and Computing Techniques (GTIDIC)

Agenda

Introduction

Development environment

Docker

Database Service

Homework

Introduction

Warm-up

- Dr. Jordi Mateo Fornés
- **Office**:
  - Office A.12 (Campus Igualada)
  - Office 3.08 (EPS Lleida)
- **Email**: jordi.mateo@udl.cat
- **Twitter**: https://twitter.com/MatForJordi
- **Github**: https://github/JordiMateoUdL

- Ph.D Dídac Florensa Cazorla
- **Office**:
  - Office A.12 (Campus Igualada)
  - Office 3.08 (EPS Lleida)
- **Email**: didac.florensa@udl.cat

Development environment

Frontend and backend

In software engineering, the terms **frontend** and **backend** refer to the separation of concerns between the presentation layer (frontend) and the data access layer or the physical infrastructure or hardware.

Exemple

In your project, the **frontend** is your Android application, while the **backend** will be a set of services that assist the backend, such as the operations with the database.

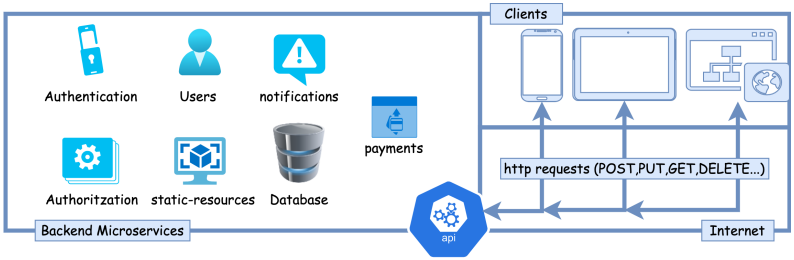


Figure 1: Overview of the frontend/backend in Android Context

DAMCore (Starting Kit)

DAM-ProjectCore] is a GitHub repository created and maintained by the teachers to provide students with a backend starting kit. It contains an actual implementation of the most basic services.

Service

- **Database Server**: MySQL Server.
- **Adminer**: Web App management tool for the MySQL database.
- **Falcon API**: Python 3.7 Restful API developed with Falcon framework.
- **Static File Service**: Nginx Server for managing static resources.

Organization

- **Project**:
  - Android folder (Clone of your repo)
  - DamCore folder (Fork and clone DAM-ProjectCore)
  - static folder (Create this folder (it is required)

Caution. Do not rename the folder **DAM-ProjectCore** when you clone it. If you do, you will need to update *docker-compose file*.

DAMCore (Starting Kit)

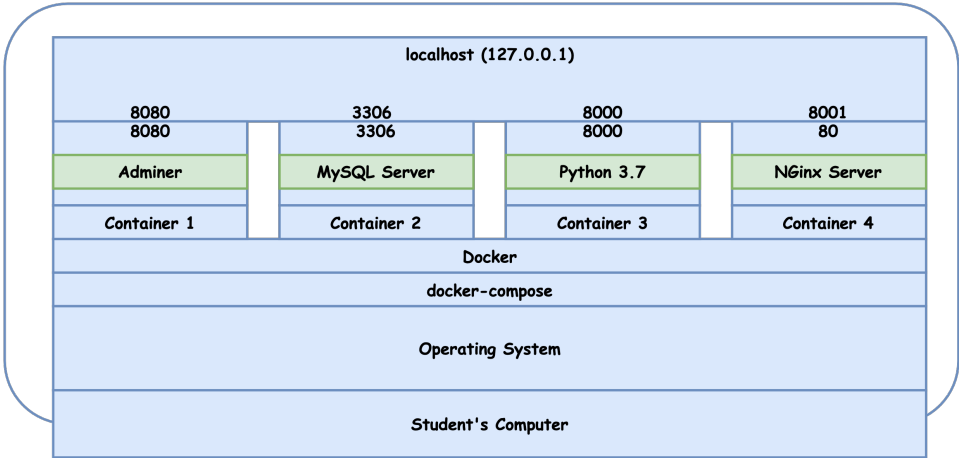


Figure 2: Starting Kit

Repository: DAM-ProjectCore

Docker

What is Docker?

Docker is an open-source containerization platform. It enables developers to package applications into containers—standardized executable components combining application source code with the operating system (OS) libraries and dependencies required to run that code in any environment.

What is a container

- A container represents an isolated process in the userspace.
- Each container can target a single application and install only the software and libraries needed to run that application.
- Resources are allocated to containers that represent a set of processes, files, and partitions.
- All containers share the same *Operating System* (host).

Is a container a Virtual Machine?

Containers and VMs are very similar resource virtualization technologies. **Virtualization** is the process in which a system singular resource like RAM, CPU, Disk, or Networking can be 'virtualized' and represented as multiple resources. The key differentiator between containers and VMs is that VMs virtualize an **entire machine down to the hardware layers** and *containers* only virtualize *software layers* above the **operating system level**.

Which are the main differences between VM and containers?

VMs	Containers
Heavyweight	Lightweight
Limited performance	Native performance
Each VM runs in its OS	All containers share the host OS
Hardware-level virtualization	OS virtualization
Startup time in minutes	Startup time in milliseconds
Allocates required memory	Requires less memory space
Fully isolated and hence more secure	process-level isolation, possibly less safe

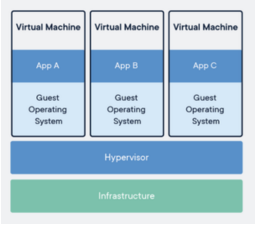


Figure 3: Virtual Machine (schema)

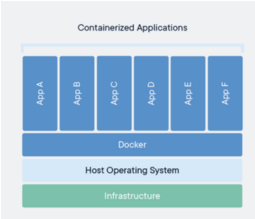
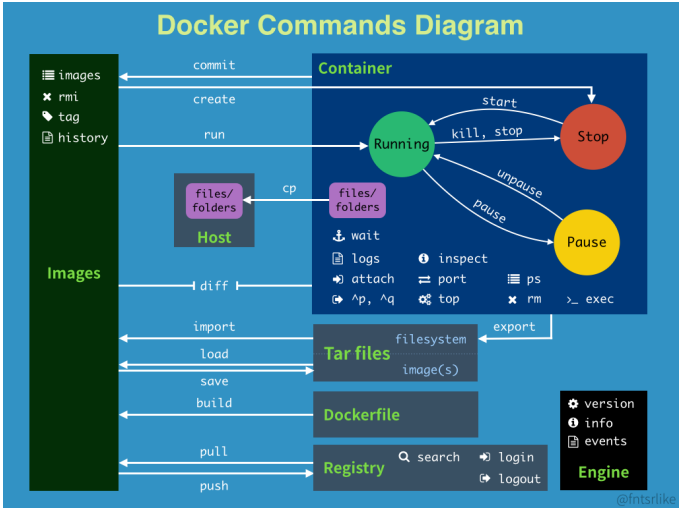
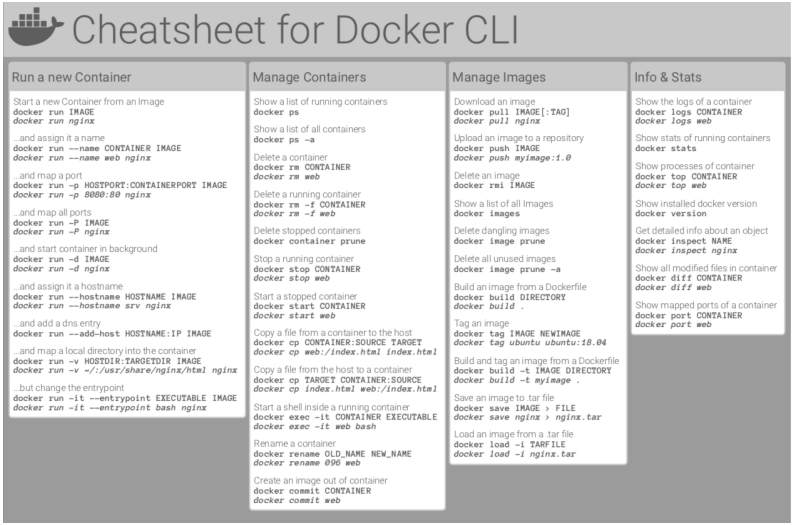


Figure 4: Container – docker (schema)

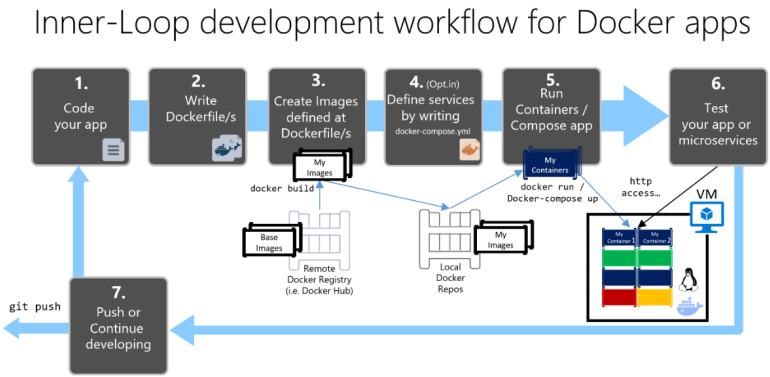
How do Docker works?



Cheatsheet


Cheatsheet for Docker CLI
Run a new Container
Start a new Container from an Image
docker run IMAGE
docker run nginx
...and assign it a name
docker run --name CONTAINER IMAGE
docker run --name web nginx
...and map a port
docker run -p HOSTPORT:CONTAINERPORT IMAGE
docker run -p 8080:80 nginx
...and map all ports
docker run -P IMAGE
docker run -P nginx
...and start container in background
docker run -d IMAGE
docker run -d nginx
...and assign it a hostname
docker run --hostname HOSTNAME IMAGE
docker run --hostname srv nginx
...and add a dns entry
docker run --add-host HOSTNAME:IP IMAGE
...and map a local directory into the container
docker run -v HOSTDIR:TARGETDIR IMAGE
docker run -v ~/:/usr/share/nginx/html nginx
...but change the entrypoint
docker run -it --entrypoint EXECUTABLE IMAGE
docker run -it --entrypoint bash nginx
Manage Containers
Show a list of running containers
docker ps
Show a list of all containers
docker ps -a
Delete a container
docker rm CONTAINER
docker rm web
Delete a running container
docker rm -f CONTAINER
docker rm -f web
Delete stopped containers
docker container prune
Stop a running container
docker stop CONTAINER
docker stop web
Start a stopped container
docker start CONTAINER
docker start web
Copy a file from a container to the host
docker cp CONTAINER:SOURCE TARGET
docker cp web:/index.html index.html
Copy a file from the host to a container
docker cp TARGET CONTAINER:SOURCE
docker cp index.html web:/index.html
Start a shell inside a running container
docker exec -it CONTAINER EXECUTABLE
docker exec -it web bash
Rename a container
docker rename OLD_NAME NEW_NAME
docker rename 096 web
Create an image out of container
docker commit CONTAINER
docker commit web
Manage Images
Download an image
docker pull IMAGE[:TAG]
docker pull nginx
Upload an image to a repository
docker push IMAGE
docker push myimage:1.0
Delete an image
docker rmi IMAGE
Show a list of all Images
docker images
Delete dangling images
docker image prune
Delete all unused images
docker image prune -a
Build an image from a Dockerfile
docker build DIRECTORY
docker build .
Tag an image
docker tag IMAGE NEWIMAGE
docker tag ubuntu ubuntu:18.04
Build and tag an image from a Dockerfile
docker build -t IMAGE DIRECTORY
docker build -t myimage .
Save an image to .tar file
docker save IMAGE > FILE
docker save nginx > nginx.tar
Load an image from a .tar file
docker load -i TARFILE
docker load -i nginx.tar
Info & Stats
Show the logs of a container
docker logs CONTAINER
docker logs web
Show stats of running containers
docker stats
Show processes of container
docker top CONTAINER
docker top web
Show installed docker version
docker version
Get detailed info about an object
docker inspect NAME
docker inspect nginx
Show all modified files in container
docker diff CONTAINER
docker diff web
Show mapped ports of a container
docker port CONTAINER
docker port web

Workflow



1. We will use images from the offical marketplace(MySql, Nginx, Python3.7 and Adminer).
2. We will create a custom docker file image for our Python App.
3. We will orchestrate all the services using the docker-compose tool.

How to create a dockerfile?

Dockerfile

```
FROM python:3.7
COPY start.sh /scripts/start.sh
RUN chmod +x /scripts/start.sh
RUN mkdir /logs
WORKDIR /app
EXPOSE 8000
ENTRYPOINT ["/scripts/start.sh"]
```

File: DAM-ProjectCore/docker/backend/Dockerfile

Script

```
#!/bin/bash
pip install --upgrade pip
pip install -r /app/requirements.txt
export PYTHONPATH=$PYTHONPATH:/app
python /app/dev/reset_database.py
gunicorn -b [::]:8000 app:app --reload
```

File: DAM-ProjectCore/docker/backend/start.sh

How to create a docker-compose?

```yaml
version: '3.7'
services:
  backend:
    build: "./backend"
    ports:
      - "8000:8000"
    environment:
      - DAMCore_DB_HOST=mysql_db_container
    volumes:
      - ../../DAM-ProjectCore/.:/app
      - ../../static/.:/static
    tty: true
    links:
      - adminer_container
      - mysql_db_container
    depends_on:
      - mysql_db_container
```

File: DAM-ProjectCore/docker/docker-compose.yml

How to create a docker-compose? (cont.)

```yaml
  static-file-server:
    image: nginx:latest
    ports:
      - 8001:80
    volumes:
      - ../../static/.:/usr/share/nginx/html/static
  mysql_db_container:
    image: mysql:latest
    command: --default-authentication-plugin=mysql_native_password
    environment:
      MYSQL_ROOT_PASSWORD: root1234
      MYSQL_DATABASE: dev-test
      MYSQL_USER: dev-user
      MYSQL_PASSWORD: 1234
    ports:
      - 3306:3306
    cap_add:
      - SYS_NICE  # CAP_SYS_NIC
  adminer_container:
      image: adminer:latest
      environment:
        ADMINER_DEFAULT_SERVER: mysql_db_container
      ports:
        - 8080:8080
```

File: DAM-ProjectCore/docker/docker-compose.yml

Docker-Compose (1)

```
# Starts existing containers for a service.
docker-compose start
# Stops running containers without removing them.
docker-compose stop
# Pauses all the running containers of a service.
docker-compose pause
# Unpauses paused containers of a service.
docker-compose unpause
# Lists containers.
docker-compose ps
```

Docker-Compose (2)

```
# Builds, (re)creates, starts, and attaches to containers for a service.
docker-compose up (-d) (--build)
# Stops containers and removes containers, networks, volumes, and images created by up.
docker-compose down
# Logs
docker-compose logs
```

Help me...! Crazy Alert! I have broked something, and it used to work, but I did not work any more... Instead of restarting your computer... do:

```
docker system prune -a
```

Database Service

Database Server

MySql Server

The MySQL server provides a database management system with querying and connectivity capabilities and the ability to have excellent data structure and integration with many different platforms. It can handle large databases reliably and quickly in high-demanding production environments.

How to connect to a MySQL DB (python)

```
from sqlalchemy import create_engine
from sqlalchemy.orm import scoped_session, sessionmaker
from settings import DB_USERNAME, DB_PASSWORD, DB_HOST, DB_PORT, DB_NAME, DB_ENCODING
DB_ENGINE = create_engine(
    "mysql+pymysql://{}:{}@{}:{}/{}?charset=utf8".format(DB_USERNAME, DB_PASSWORD, DB_HOST, DB_PORT, DB_NAME),
    encoding=DB_ENCODING, echo=False, pool_recycle=3600)
DB_SESSION_FACTORY = sessionmaker(bind=DB_ENGINE)
DB_SCOPED_SESSION_FACTORY = scoped_session(DB_SESSION_FACTORY)
def create_db_session():
    return DB_SESSION_FACTORY()
```

File: DAM-ProjectCore/db/__init__.py /

Which is the schema provided (Starting Kit)?

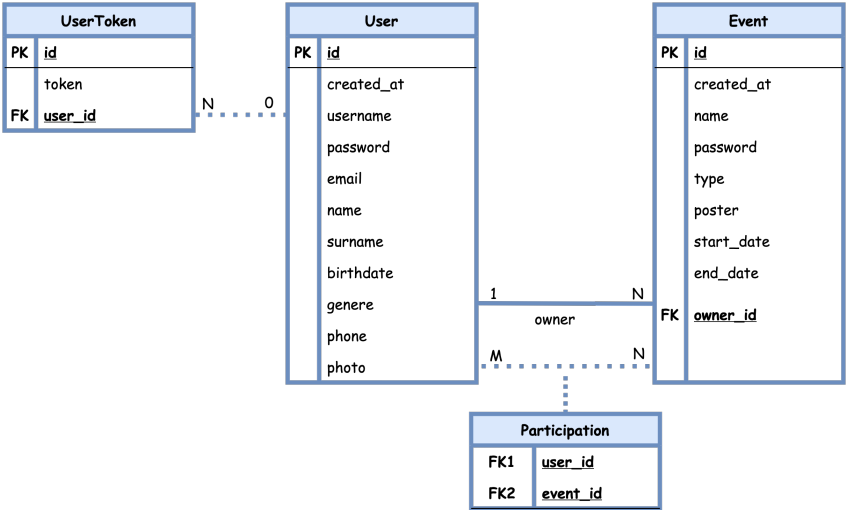


Figure 5: Intial Schema of the DB provided

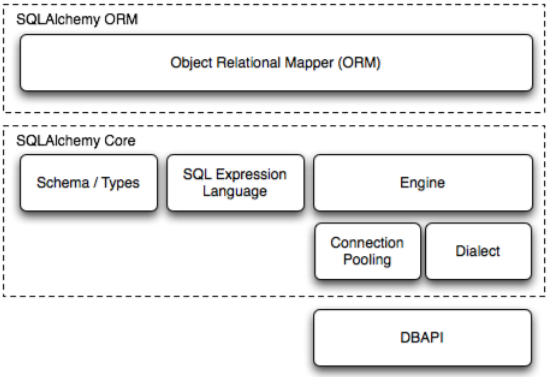
SqlAlchemy

SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.

- SQL databases behave less like object collections.
- SQLAlchemy is most famous for its object-relational mapper (ORM).
- The library automates redundant tasks while the developer controls how the database is organized and how SQL is constructed.
- The main goal of SQLAlchemy is to provide an object-oriented framework to manage SQL databases.

It provides a full suite of well known enterprise-level persistence patterns, designed for efficient and high-performing database access, adapted into a simple Pythonic domain language.

How SqlAlchemy work?



Developing the user model using SqlAlchemy

- **User Class** extends *SQLAlchemyBase* and *JSONModel*. All the models you implement require this action.
- ***tablename*** represents the name of the table in the database.
- *Column* represents an attribute.
- Types needs to be imported: *from sqlalchemy import Column, Date, DateTime, Enum, ForeignKey, Integer, Unicode,UnicodeText, Table*
- *nullable* indicates is attribute can be **NULL**.
- *unquie* indicates if attribute must be unique.

```
class User(SQLAlchemyBase, JSONModel):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    created_at = Column(DateTime,
        default=datetime.datetime.now,
        nullable=False)
    username = Column(Unicode(50),
        nullable=False, unique=True)
    password = Column(UnicodeText, nullable=False)
    email = Column(Unicode(255), nullable=False)
    name = Column(Unicode(50), nullable=False)
    surname = Column(Unicode(50), nullable=False)
    birthdate = Column(Date)
    genere = Column(Enum(GenereEnum),
        nullable=False)
    phone = Column(Unicode(50))
    photo = Column(Unicode(255))
```

File: DAM-ProjectCore/db/models.py/

Why an enum for the genre?

The properties of an enumeration help define an immutable, related set of constant values that may or may not have a semantic meaning.

```
class GenereEnum(enum.Enum):
    male = "M"
    female = "F"
```

What is JsonModel Class?

It is a helper class to *translate/map* and object to a **key:value** JSON dictionary. This will be super helpful to communicate information to the Android client.

```python
class JSONModel(object):
    __metaclass__ = abc.ABCMeta
    def _create_json_model(self, **attributes):
        final_model = dict()
        try:
            for current_key in attributes.keys():
                aux_attribute = getattr(self, attributes[current_key])
                if isinstance(aux_attribute, JSONModel) and aux_attribute is not None:
                    final_model[current_key] = aux_attribute.json_model
                elif isinstance(aux_attribute, datetime.datetime):
                    final_model[current_key] = aux_attribute.strftime(DATETIME_DEFAULT_FORMAT)
                elif isinstance(aux_attribute, datetime.date):
                    final_model[current_key] = aux_attribute.strftime(DATE_DEFAULT_FORMAT)
                elif isinstance(aux_attribute, datetime.time):
                    final_model[current_key] = aux_attribute.strftime(TIME_DEFAULT_FORMAT)
                else:
                    final_model[current_key] = aux_attribute
            return final_model
        except KeyError as e:
            raise falcon.HTTPInternalServerError(description=str(e))

    @abc.abstractmethod
    def json_model(self):
        pass
    def to_json_model(self, **attributes):
        return self._create_json_model(**attributes)
```

File: DAM-ProjectCore/db/json_model.py/

User model class (Methods)

```python
@hybrid_property
def photo_url(self):
    return _generate_media_url(self, "photo")

@hybrid_property
def photo_path(self):
    return _generate_media_path(self, "photo")

@hybrid_method
def set_password(self, password_string):
    self.password = pbkdf2_sha256.hash(password_string)

@hybrid_method
def check_password(self, password_string):
    return pbkdf2_sha256.verify(password_string, self.password)
```

User model class (Methods)

```
@hybrid_property
    def json_model(self):
        return {
            "created_at": self.created_at.strftime(
                settings.DATETIME_DEFAULT_FORMAT),
            "username": self.username,
            "email": self.email,
            "name": self.name,
            "surname": self.surname,
            "birthdate": self.birthdate.strftime(
                settings.DATE_DEFAULT_FORMAT) if
                self.birthdate is not None else self.birthdate,
            "genere": self.genere.value,
            "phone": self.phone,
            "photo": self.photo_url
        }
```

Basic Relationship Patterns: **One to Many**

```python
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
```

Information extracted from SqlAlchemy Documentation, check it **SqlAlchemy Docs**.

Basic Relationship Patterns: **Many to One**

```python
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", back_populates="parents")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parents = relationship("Parent", back_populates="child")
```

Information extracted from SqlAlchemy Documentation, check it **SqlAlchemy Docs**.

Basic Relationship Patterns: **One to One**

```python
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child = relationship("Child", uselist=False,
    back_populates="parent")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
    parent = relationship("Parent", back_populates="child")
```

Information extracted from SqlAlchemy Documentation, check it **SqlAlchemy Docs**.

Basic Relationship Patterns: **Many to many**

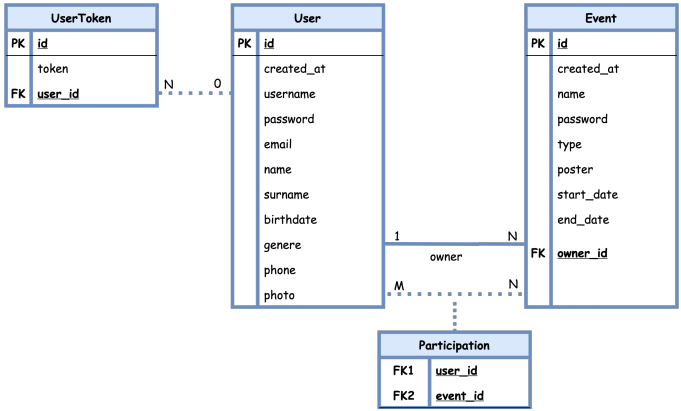
```
association_table = Table('association', Base.metadata,
    Column('left_id', Integer, ForeignKey('left.id')),
    Column('right_id', Integer, ForeignKey('right.id'))
)

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Child",
                    secondary=association_table)

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

HandsOn(DamCore - Events) - Goals

- Create a new data model, an **event**.
- Store them in the database.
- Create the following relations:
  - Only **one user** can create a signle **event**. But this user can create **1 or more** events. *(Relation One to Many)*.
  - A user can be enrolled with **0 or N** events, and also, an event can have **0 or M** registered users. *(Relation Many to Many)*.



HandsOn(DamCore - Events) - Model 1

- We are going to define three types of events using an enum.

```
class EventTypeEnum(enum.Enum):
    hackathon = "H"
    lanparty = "LP"
    livecoding = "LC"
```

File: DAM-ProjectCore/db/models.py

HandsOn(DamCore - Events) - Model 2

- Basic features of the events:

```
    id = Column(Integer, primary_key=True)
    created_at = Column(DateTime, default=datetime.datetime.now,
    nullable=False)
    name = Column(Unicode(255), nullable=False)
    description = Column(UnicodeText)
    type = Column(Enum(EventTypeEnum))
    start_date = Column(DateTime, nullable=False)
    finish_date = Column(DateTime, nullable=False)
```

File: DAM-ProjectCore/db/models.py

HandsOn(DamCore - Events) - Relation (One to Many)

```python
class Event(SQLAlchemyBase, JSONModel):
    __tablename__ = "events"
    ...
    owner_id = Column(Integer,
      ForeignKey("users.id"), nullable=False)
    owner = relationship("User", back_populates="events_owner")
    ...
```

```python
class User(SQLAlchemyBase, JSONModel):
    __tablename__ = "users"
    ...
    events_owner = relationship("Event", back_populates="owner")
    ...
```

File: DAM-ProjectCore/db/models.py

HandsOn(DamCore - Events) - Relation (One to Many)

back_populates, backref

To tell *Sqlalchemy* that two fields are related. Use **back_populates** to define the relationships on every class (cleaner option). If not, you can also use **backref**.

back_populates has the same meaning as backref, except that the complementing relationship property is not created automatically. So using **back_populates** makes the model code more explicit, with no *hidden/implicit* properties.

HandsOn(DamCore - Events) - Relation (One to Many)

DELETE issue

- Now, to delete a user, first, I need all the events owned by this user or change ownership. Because, if I try to delete, I will see:

```
Cannot delete or update a parent row: a foreign key constraint
fails (`dev-test`.`events`, CONSTRAINT `events_ibfk_1`
FOREIGN KEY (`owner_id`) REFERENCES `users` (`id`))
12:55:24 Ordre SQL
```

The **delete cascade** indicates that when a "parent" object is marked for deletion, its related "child" objects should also be marked for deletion. Delete cascade is often used in conjunction with **delete-orphan** cascade, which will emit a *DELETE* for the related row if the "child" object is disassociated from the parent.

The **combination** of **delete and delete-orphan cascade** *covers* both situations where *SQLAlchemy* has to *decide between* settings a foreign key column to *NULL* versus *deleting the row entirely*.

HandsOn(DamCore - Events) - Relation (One to Many)

```python
class Event(SQLAlchemyBase, JSONModel):
    __tablename__ = "events"
    owner_id = Column(Integer,
      ForeignKey("users.id", onupdate="CASCADE",
      ondelete="CASCADE"), nullable=False)
    owner = relationship("User", back_populates="events_owner")
```

```python
class User(SQLAlchemyBase, JSONModel):
    __tablename__ = "users"
    events_owner = relationship("Event", back_populates="owner",
    cascade="all, delete-orphan")
```

File: DAM-ProjectCore/db/models.py

HandsOn(DamCore - Events) - Relation (Many to Many) - Association

```
EventParticipantsAssociation =
  Table("event_participants_association",
  SQLAlchemyBase.metadata,
      Column( "event_id", Integer,
        ForeignKey("events.id", onupdate="CASCADE",
        ondelete="CASCADE"),nullable=False),
      Column("user_id", Integer,
        ForeignKey("users.id", onupdate="CASCADE",
        ondelete="CASCADE"),nullable=False),
    )
```

File: DAM-ProjectCore/db/models.py

HandsOn(DamCore - Events) - Relation (Many to Many)

```python
class Event(SQLAlchemyBase, JSONModel):
    __tablename__ = "events"
    registered = relationship("User",
    secondary=EventParticipantsAssociation,
    back_populates="events_enrolled")
```

```python
class User(SQLAlchemyBase, JSONModel):
    __tablename__ = "users"
    events_enrolled = relationship("Event",
    back_populates="registered")
```

File: DAM-ProjectCore/db/models.py

HandsOn(DamCore - Events) - Reset database script

reset_database.py

This repository provides a tool to init quick restore your database. If you see, this file defines some initial instances of different classes (Users, Events and Tokens).

```
day_period = datetime.timedelta(days=1)
event_hackatoon = Event(
  created_at=datetime.datetime.now(),
  name="event1",
  description="description 1",
  type=EventTypeEnum.hackathon,
  start_date=datetime.datetime.now() + day_period,
  finish_date=datetime.datetime.now() + (day_period * 2),
  owner_id = 0,
  registered=[user_1, user_2]
)
```

File: DAM-ProjectCore/dev/reset_database.py/

Homework

Task A: Install all the required tools

- **Docker**:
  - Windows:
    - https://docs.docker.com/docker-for-windows/install/
  - Linux:
    - https://computingforgeeks.com/install-docker-ce-on-linux-systems/
  - Mac:
    - https://docs.docker.com/docker-for-mac/install/
- **Compose**: https://docs.docker.com/compose/install/
- **Postman:** https://www.postman.com/downloads/

Task B: Deploy the services

1. Fork and clone the project (DamCore).
2. Create the static folder.
3. Go to the docker folder.
4. Run in the terminal “‘docker-compose up -d’”.
5. Check the logs “`docker-compose logs`”.
6. Navigate in your web browser to http:/127.0.0.1:8080.
7. Log into adminer (User:dev-user,Password:1234,DB:dev-test).

Task C: Play and Adapt

1. Create your users, one for each team member and another for the teacher. Update *reset database.pt*.
2. Use adminer to create/delete and update new users and see the relations.
3. Start and stop de service and see what happens to the information in the database.
4. How I can activate or deactivate this feature?

Task D: Check these video tutorials

⇒ Docker - Pelado Nerd - **DOCKER 2021 - De NOVATO a PRO!**.

⇒ Tutorial Book - Jordi Mateo **Tutorial. Book. Part 1** - Only the first 23 minutes.

That's all

QUESTIONS?

About me

www — jordimateofornes.com
github — github.com/JordiMateo
twitter — @MatForJordi
gdc — Distributed computation group