

# Conectando o Flask ao Banco de Dados

Experiência Criativa: Criando  
Soluções Computacionais

Antonio David Viniski

[antonio.david@pucpr.br](mailto:antonio.david@pucpr.br)

# Agenda

- Arquitetura MVC
  - Model (Modelo).
  - View (Visualização).
  - Controller (Controladores).
- Flask SQLAlchemy.
  - Criação de modelos e Tabelas.
  - Criação da estrutura do Banco de Dados.
  - Inserção de dados.
  - Realização de consultas.
  - Atualização de registros.

# Arquitetura MVC

# Arquitetura Modelo, Visão, Controle (MVC)

- O MVC é um padrão de arquitetura de software que separa a sua aplicação em 3 camadas.
- O MVC é utilizado em muitos projetos devido a arquitetura que possui, o que possibilita a divisão do projeto em camadas muito bem definidas.
- Cada uma delas, o **Model**, o **Controller** e a **View**, executa o que lhe é definido e nada mais do que isso.

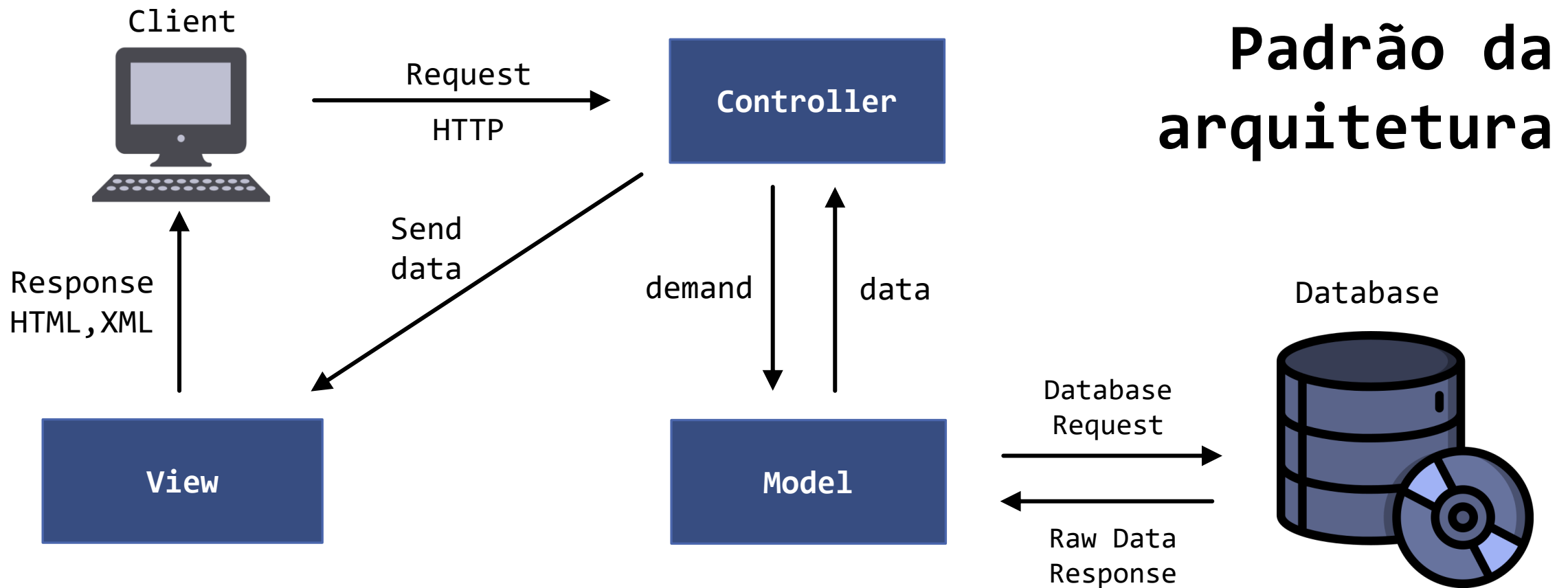
# Arquitetura Modelo, Visão, Controle (MVC)

- A utilização do padrão MVC traz como benefício o isolamento das **regras de negócios** da **lógica de apresentação** (interface com o usuário).
  - possibilita a existência de várias interfaces com o usuário que podem ser modificadas sem a necessidade de alterar as regras de negócios.
  - proporcionando muito mais flexibilidade e oportunidades de reuso das classes.
- Uma das características de um padrão de projeto é poder aplicá-lo em sistemas distintos.
  - O padrão MVC pode ser utilizado em vários tipos de projetos como, por exemplo, desktop, web e mobile.

# Arquitetura Modelo, Visão, Controle (MVC)

- A Camada **View**: É a camada de interação com o usuário. Apenas faz a exibição dos dados.
- A Camada **Model**: É responsável pela leitura e escrita de dados, e também de suas validações.
- A Camada **Controller**: É responsável por receber todas as requisições do usuário. Seus métodos chamados *actions* são responsáveis por uma página, controlando qual *model* usar e qual *view* será mostrado ao usuário.

# Arquitetura Modelo, Visão, Controle (MVC)



# Arquitetura MVC com Flask



# Controllers Flask

- Recebem as requisições da Aplicação.
  - Aplicação *Flask*;
  - Módulos *Blueprint*;
- Redirecionam as *views* como resposta para as requisições.
  - Renderização dos templates;
  - Envio de dados para as *views* (Objetos, listas, dicionários, etc).
- Gerenciam os dados da aplicação que por meio dos *models*.
  - Recebem dos dados do usuário e solicitam o armazenamento do banco;
  - Solicitam a recuperação e manipulação dos dados do banco de dados;



# Views Flask

- Representam os templates HTML (Jinja2).
- Apresentam os dados aos usuários;
- Solicitam entradas do usuário (Formulários);
- Apresentam as opções de navegação da aplicação para o usuário.

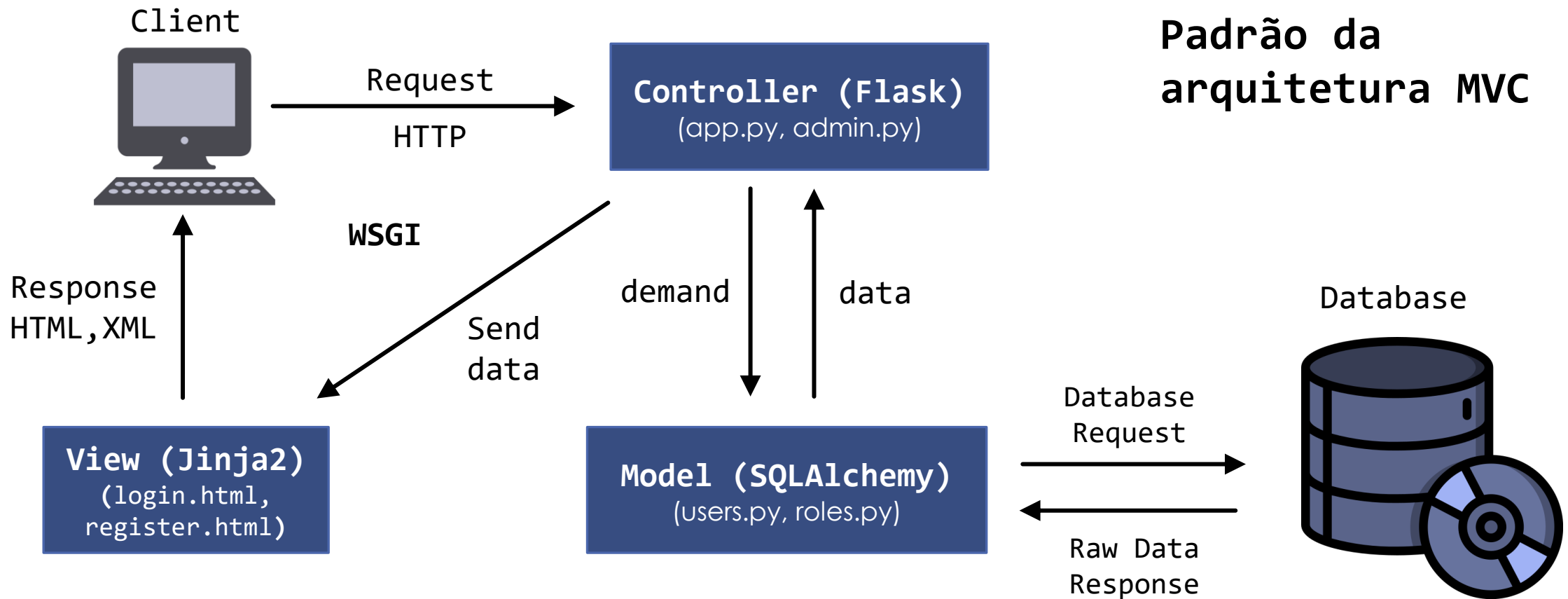


# Models Flask - SQLAlchemy

- Representar a estrutura dos dados que irá trafegar pela aplicação.
- Modelar a forma como os dados estão relacionados.
- Realizar a persistência e recuperação dos dados do banco de dados.



# MVC com Flask



# ORM Flask SQLAlchemy

# ORM (Object-Relational Mapping)

- *Object-Relational Mapping* (ORM), em português, mapeamento objeto-relacional, é uma ferramenta que utiliza mecanismos que possibilitam a manipulação dos objetos por meio do mapeamento entre sistemas orientados a objetos e banco de dados relacionais.
- O uso da técnica de mapeamento objeto-relacional é realizado através de um **mapeador objeto-relacional** que geralmente é a **biblioteca ou framework** que ajuda no mapeamento e uso do banco de dados.
  - ORM é um mecanismo de mapeamento que viabiliza a relação dos objetos com os dados que eles representam.



# ORM (Object-Relational Mapping) II



- **Object**

- São as instâncias de uma classe que receberão e guardarão as informações dos dados do banco de dados relacional.

- **Relational**

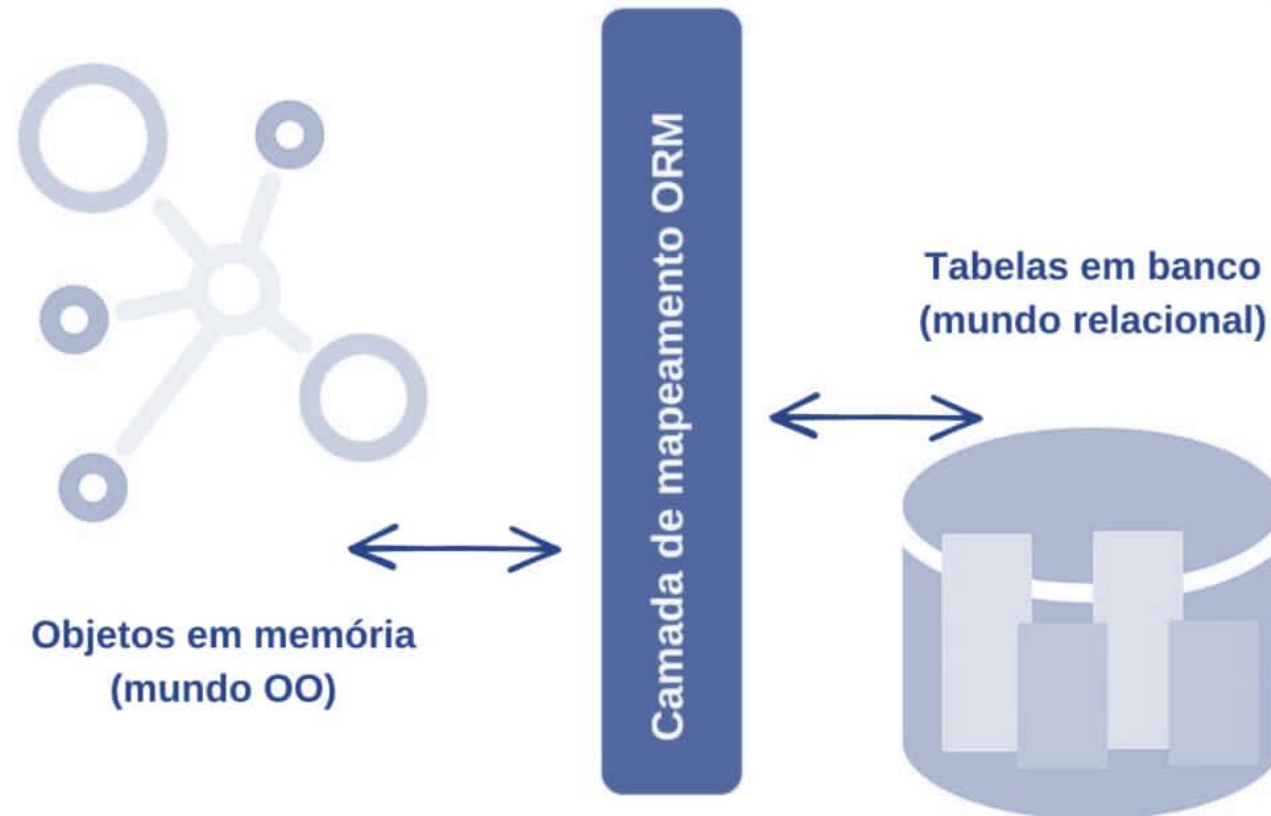
- É o sistema que faz o gerenciamento do banco de dados relacional, armazena e fornece acesso a pontos de dados relacionados entre si.

- **Mapper**

- Mapeador de objetos, que faz a ligação entre os objetos da aplicação e o banco de dados relacional por meio de suas tabelas.

# ORM (Object-Relational Mapping) III

**SQLAlchemy**





# Características de um ORM



- Os ORMs são utilizados para aplicar comandos no banco de dados com subconjuntos da linguagem SQL, como DML, DDL e DQL.
  - Esses subconjuntos geralmente são chamados no sistema de CRUD (Create, Read, Update, Delete)
    - as 4 operações básicas utilizadas em bancos de dados relacionais.
- Os ORMs tornam todo o mapeamento e manipulação desses dados mais simples, aumentando a produtividade e agilidade nas operações CRUD.
- Aplicações que utilizam ORM geralmente não usam SQL de forma direta, pois os métodos criarão todas as operações necessárias diretamente do banco de dados relacional.

# Flask SQLAlchemy - instalação

- Instalando o *flask* utilizando o gerenciador de pacotes **pip**.

- Opção 1: `pip install flask-sqlalchemy`

- Opção 2: `Python -m pip install flask-sqlalchemy`

# Ajustes na Estrutura

- Com base na estrutura disponibilizada no Canvas:
  - Transferir a criação da aplicação flask para o arquivo `app_controller.py` disponível no diretório `controllers`;
    - Criar uma função que seja responsável por criar a aplicação flask, registrar os módulos e definir as páginas associação diretamente a aplicação.
    - A função retorna o objeto Flask (Ex: `app`)

```
✓ RESTAURANT  
  > controllers  
  > models  
  > static  
  > views  
  🐍 app.py
```

# Ajustes na Estrutura

Lembrem-se que precisamos realizar todos os imports necessários: métodos, classes, módulos , etc.

```
#app_controller.py
def create_app() -> Flask:
    app = Flask(__name__,
                 template_folder="./views/",
                 static_folder="./static/",
                 root_path="./")

    app.register_blueprint(base, url_prefix='/base')
    app.register_blueprint(auth, url_prefix='/auth')
    app.register_blueprint(billing, url_prefix='/billing')
    app.register_blueprint(payment, url_prefix='/payment')
    app.register_blueprint(people, url_prefix='/people')
    app.register_blueprint(product, url_prefix='/product')
    app.register_blueprint(ticket, url_prefix='/ticket')
    app.register_blueprint(iot, url_prefix='/iot')

    @app.route('/')
    def index():
        return render_template("home.html")

    return app
```

# Ajustes na Estrutura (app.py)

- Agora nosso arquivo app.py será ajustado para importar a função de criação da aplicação (create\_app) e depois executar.

```
#app.py
from controllers.app_controller import create_app

if __name__ == "__main__":
    app = create_app()
    app.run(debug=True)
```

Criar uma variável que recebe a aplicação **flask** criada pela função **create\_app**, depois executar...

# Criação do objeto SQLAlchemy

- Na pasta **models**, criar um arquivo chamado **db.py**

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()
instance = 'sqlite:///restaurant'
```

```
▼ RESTAURANT
  > controllers
  ▼ models
    | 🐍 db.py
  > static
  > views
  🐍 app.py
```

# Ligando o objeto db (SQLAlchemy) ao flask

- Na arquivo **app\_controller.py**, realizar a importação dos atributos **db** e **instance** do arquivo model/db.py

```
from models.db import db, instance
```

- Na função `create_app()`, após a criação do objeto da aplicação, adicionar as configurações necessárias para vincular a aplicação ao banco de dados

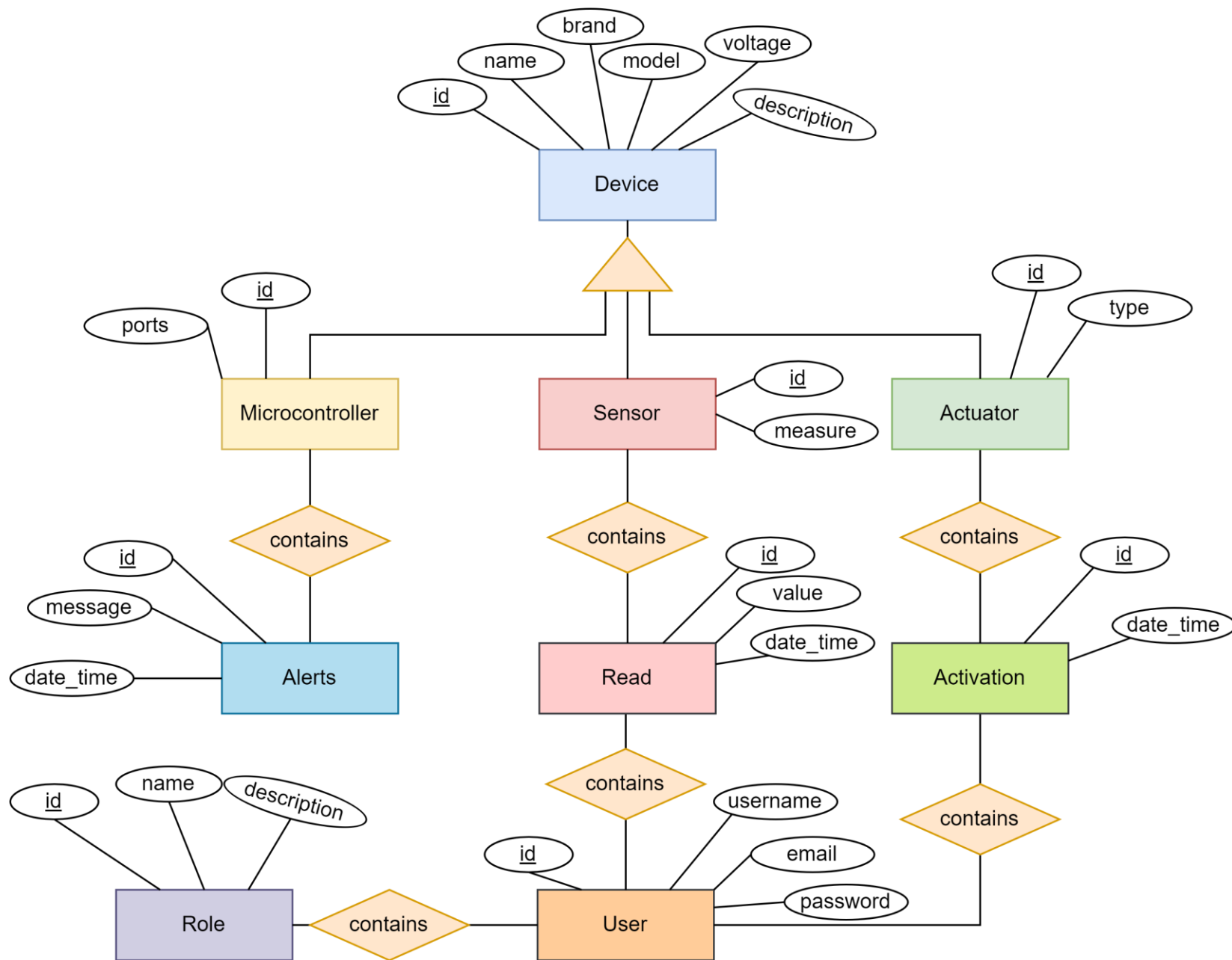
```
app.config['TESTING'] = False
app.config['SECRET_KEY'] = 'generated-secrete-key'
app.config['SQLALCHEMY_DATABASE_URI'] = instance
db.init_app(app)
```

# Alguns tipos de dados SQLAlchemy

SQLAlchemy	Python	SQL
BigInteger	int	BIGINT
Boolean	bool	BOOLEAN ou SMALLINT
Date	datetime.date	DATE (SQLite: STRING)
DateTime	datetime.datetime	DATETIME (SQLite: STRING)
Enum	str	ENUM ou VARCHAR
Float	float ou Decimal	FLOAT ou REAL
Integer	int	INTEGER
Interval	datetime.timedelta	INTERVAL ou DATE from epoch
LargeBinary	byte	BLOB ou BYTEA
Numeric	decimal.Decimal	NUMERIC ou DECIMAL
Unicode	unicode	UNICODE ou VARCHAR
Text	str	CLOB ou TEXT
Time	datetime.time	DATETIME

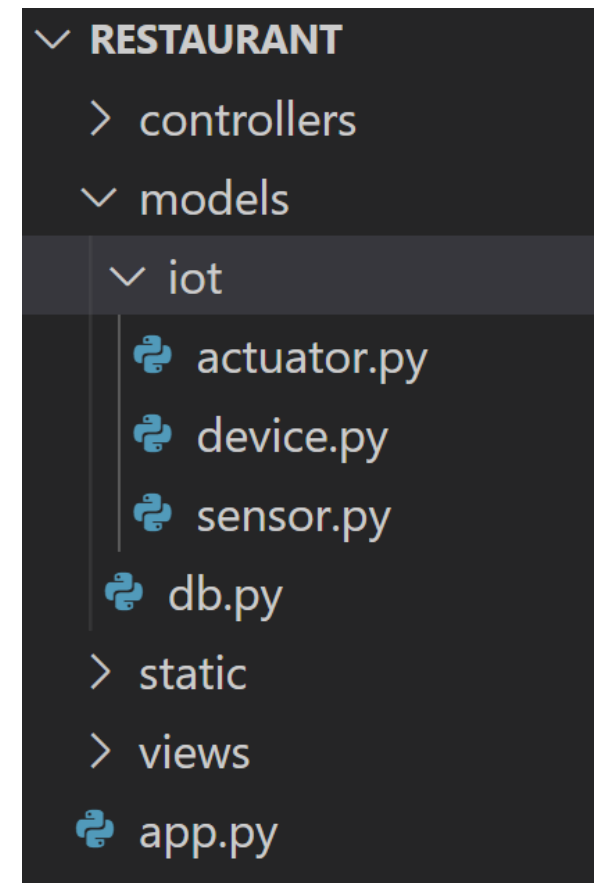


# Modelo Conceitual - IoT



# Estrutura para os modelos de IOT

- Após a ligação da nossa aplicação ao banco de dados, podemos iniciar a criação dos modelos.
  - Dentro do diretório `models/`, adicionar um subdiretório chamado `iot/`
  - Criar os arquivos no diretório: **`devices.py`**, **`sensors.py`** e **`actuators.py`**.
    - Em cada arquivos vamos criar as respectivas classes que serão utilizadas para instanciar os objetos do tipo Device, Sensor e Actuator.



# Criando os modelos

## ○ Criando a classe Device:

```
from models.db import db

class Device(db.Model):
    __tablename__ = 'devices'
    id = db.Column('id', db.Integer, primary_key=True)
    name = db.Column(db.String(50))
    description = db.Column(db.String(512))
    brand = db.Column(db.String(50))
    model = db.Column(db.String(50))
    voltage = db.Column(db.Float)
    is_active = db.Column(db.Boolean, nullable = False, default = False)
```

Especifica o nome da tabela do banco de dados que essa classe representa

Todo modelo no SQLAlchemy precisa ter uma chave primária (id), neste caso estamos criando uma chave primária do tipo Inteiro

Também podemos especificar se o campo aceita valores nulos, bem como definir um valor padrão para ele.

# Criando os modelos

- Para permitir que a nossa instância do **SQLAlchemy** fique acessível de forma mais fácil, vamos criar no diretório **models/** o arquivo **\_\_init\_\_.py** e importar o objeto **db** (disponível em **models/db**) nesse arquivo.

```
from models.db import db
```


Assim, para utilizar a instância do SQLAlchemy nos demais arquivos do projeto, basta importar diretamente do diretório models.


## ✓ RESTAURANT

> controllers

✓ models


> iot

 **\_\_init\_\_.py**

 db.py

> static

> views

 app.py

# Criando Sensor

- Novamente, para que a classe dispositivo esteja mais facilmente acessível, vamos importá-la no arquivo **models/\_\_init\_\_.py**

```
#__init__.py
from models.db import db
from models.iot.device import Device
```

- Considerando que os objetos **sensor** e **atuador** são tipos de **dispositivos**, precisamos especificar a estrutura de especialização e generalização na criação desses objetos e definir como eles estarão interligados.

```
#sensor.py
from models.db import db
from models import Device
```

```
class Sensor(db.Model):
    __tablename__ = 'sensors'
    id = db.Column('id', db.Integer, db.ForeignKey(Device.id), primary_key=True)
    measure = db.Column(db.String(20))
```

o campo id é tanto chave primária quanto estrangeira



# Relacionamentos – ajustando Device

- O ORM permite que possamos acessar todas os objetos relacionados diretamente a um outro tipo de objeto específico. Para isso, precisamos especificar esse relacionamento.
- No exemplo anterior, sabemos que a classe dispositivo pode estar associada a vários sensores, assim, podemos especificar um atributo na classe device que receba todos esses objetos relacionados:

```
from models.db import db

class Device(db.Model):
    :
    sensors = db.relationship('Sensor', backref='devices', lazy=True)
```

# Exercício

- Criar a classe atuador (**Actuator**), a qual, assim como o sensor, é um tipo de dispositivo (**Device**).
  - Campos id (Inteiro), actuator\_type (String)
- Ajustar novamente a classe **Device** para indicar que pode estar associada a vários atuadores.

# Modelo Role

- Uma **Role** indica quais funcionalidades um usuário poderá acessar posteriormente.
  - Contém os atributos ***id***, ***name*** e ***description***.

```
#role.py
from models.db import db

class Role(db.Model):
    __tablename__ = 'roles'
    id = db.Column('id', db.Integer(), primary_key=True)
    name = db.Column(db.String(50))
    description = db.Column(db.String(512))
```

- Uma **Role** pode estar associada a vários usuários.



# Modelo User

- A classe **User** armazena a estrutura de um usuário da aplicação.
- Contém os atributos **id**, **username** e **password**.

```
#role.py
from models.db import db

class User(db.Model):
    __tablename__ = 'users'
    id = db.Column('id', db.Integer(), primary_key=True)
    username = db.Column(db.String(45))
    password = db.Column(db.String())
```

- Um **User** pode estar associado a várias instâncias de **Role**.

# Relacionamento N:N – Modelo UserRoles

- Como é uma tabela gerado por conta de um relacionamento N:N, ela possui como atributo as chaves estrangeiras desse relacionamento.
  - **user\_id, role\_id**

```
#user_roles.py
from models import db, User, Role

class UserRole(db.Model):
    __tablename__ = "user_roles"
    user_id = db.Column(db.Integer(), db.ForeignKey(User.id), primary_key=True)
    role_id = db.Column(db.Integer(), db.ForeignKey(Role.id), primary_key=True)
```

# Relacionamento N:N – otimizando acesso

- O ORM SQLAlchemy permite a recuperação de todas as instâncias associadas a um objeto que participa de um relacionamento N:N.
  - Ou seja, ele permite a seleção de todas as roles de um **User** específico.
  - Da mesma forma, permite a recuperação de todos os usuários associados a uma **Role** específica.
- Para isso, utilizamos o método **db.relationship**:

```
from models import db

class User(db.Model):
    __tablename__ = "users"
    id = db.Column("id", db.Integer(), primary_key=True)
    username = db.Column(db.String(45), nullable=False, unique=True)
    email = db.Column(db.String(30), nullable=False, unique=True)
    password = db.Column(db.String(), nullable=False)

    roles = db.relationship("Role", back_populates="users", secondary="user_roles")
```

## Exercício 2

- Ajustar a classe **Role**, fazendo com que a partir de uma instância dela seja possível acessar todas as instâncias de **User** associadas a ela.

# Relacionamento 1:N – Classe Read

- Podemos ter várias leituras (**Read**) associadas a uma instância da classe **Sensor** específica.
  - Dessa forma, **Read** precisa ter uma chave estrangeira que referencia o atributo chave primária da classe **Sensor**.
- Da mesma forma, uma leitura (**Read**) pode ter sido solicitada por um usuário (**User**) específico
  - Assim, a classe **Read** também precisa ter uma chave estrangeira que faça a associação dela a um usuário (**User**).

```
from models import db, Sensor, User
from datetime import datetime

class Read(db.Model):
    __tablename__ = "reads"
    id = db.Column("id", db.Integer(), primary_key=True)
    user_id = db.Column(db.Integer(), db.ForeignKey(User.id))
    sensor_id = db.Column(db.Integer(), db.ForeignKey(Sensor.id))
    value = db.Column(db.Float())
    read_datetime = db.Column(db.DateTime(), nullable=False, default=datetime.now())
```

# Relacionamento 1:N – otimizando acesso

- Assim como em uma relação N:N, na relação 1:N também podemos recuperar todos os N objetos associados a 1 objeto específico que participa desse relacionamento.
- Para isso, utilizamos também o método **db.relationship**:
- Exemplo:
  - podemos recuperar todas as leituras efetuadas por um sensor específico diretamente do objeto do tipo Sensor:

```
from models import db, Device

class Sensor(db.Model):
    __tablename__ = "sensors"
    id = db.Column("id", db.ForeignKey(Device.id), primary_key=True)
    measure = db.Column(db.String(20), nullable=False)

    reads = db.relationship("Read", backref="sensors", lazy=True)
```

# Exercício 3 – Relacionamento 1:N

- A. Ajustar a classe **User**, adicionando também o atributo **reads** que irá retornar todas as leituras relacionadas a um usuário (**User**) específico.
- B. Criar a classe **Activation**, que está associada a classe **Actuator** e também a classe **User** por meio de relação 1-N.
  - I. Ou seja, uma instância de **Actuator** pode estar relacionada a várias instâncias de **Activation**.
  - II. Uma instância de **Activation** está ligada a uma única instância de **Actuator**.
- C. Ajustar a classe **Actuator** para indicar, por meio do relacionamento (`db.relationship`), que ela pode estar associada a várias instâncias de **Activation**.
- D. Ajustar a classe **User**, adicionando também o atributo **activations** que irá retornar todas as ativações relacionadas a um usuário (**User**) específico.

# Criando o banco de dados

- O SQLAlchemy permite a criação de toda a estrutura do banco de dados caso ela não esteja criada.
- Para isso, vamos criar um diretório na raiz do nosso projeto chamado **utils/**.
  - Nesse diretório, vamos criar o arquivo **create\_db.py** com o código a seguir.


## ✓ RESTAURANT

> controllers


> models

> static

✓ utils

 create\_db.py

> views

 app.py

```
from flask import Flask
from models import *
```

Importando todos os models criados, bem como o objeto SQLAlchemy (db) do diretório models

```
def create_db(app:Flask):
    with app.app_context():
        db.drop_all()
        db.create_all()
```

Recebendo como parâmetro a aplicação Flask

Somente os models que estiverem sido importados no arquivo **models/\_\_init\_\_.py**, serão criados



# Referências

- <https://flask-sqlalchemy.palletsprojects.com/en/3.0.x/>
- <https://jinja.palletsprojects.com/en/3.1.x/>
- <https://flask.palletsprojects.com/en/2.2.x/>