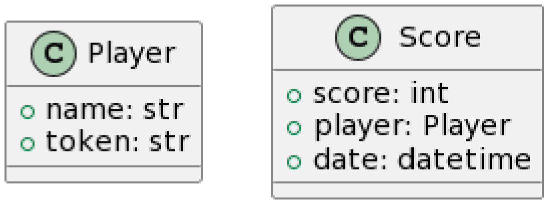
Lab: High scores management app with Flask

In this assignment, we are going to build a web app that stores high scores for a game. The web platform will also have an API that can be queried using JSON. Players will be able to post their score from their game using the API and a secure token. You will need the following libraries (install them with pip):

- `flask`
- `SQLAlchemy`
- `Flask-SQLAlchemy`

Understand the context

The application will work with two entities or classes: `Player` and `Score`. You can find the class diagram below.



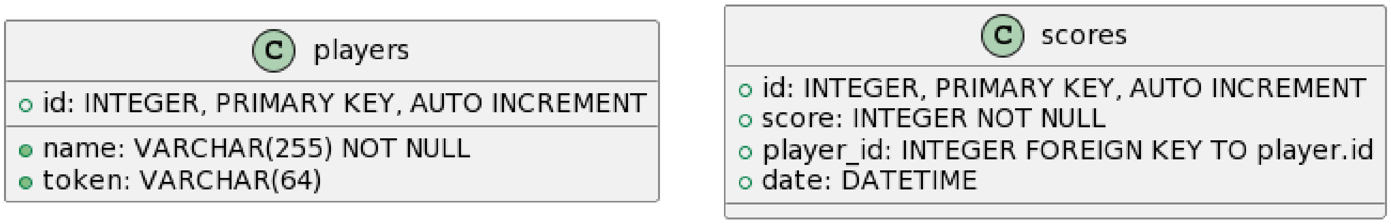
The entities / objects will be saved in a SQL database. There is no way to store Python (or any programming language) objects directly in a database. We need to "translate" the objects and their attributes to SQL tables and fields. Even though we can do that manually, it is much easier to use an **ORM** library (**O**ject **R**elational **M**apping).

In the lab, we will use SQLAlchemy.

> ⚠ SQL Alchemy, like any ORM, has a lot of features and is complex to use. Make sure you use the documentation, and **BE EXTREMELY CAREFUL WITH ONLINE RESOURCES**. A lot of them include syntax and statements for older versions of the library, which are not supported anymore.

Database schema

From the class diagram above, we can build the database schema (or physical data model) that we can use to store our Python entities.


C players
id: INTEGER, PRIMARY KEY, AUTO INCREMENT
name: VARCHAR(255) NOT NULL
token: VARCHAR(64)
C scores
id: INTEGER, PRIMARY KEY, AUTO INCREMENT
score: INTEGER NOT NULL
player_id: INTEGER FOREIGN KEY TO player.id
date: DATETIME

This diagram shows the database structure and how we are going to use foreign keys and references to implement relationships in our objects.

Database implementation in Python

Look at the `models.py` provided. There are two important things to notice:

Python models inherit from another class

All the Python classes inherit from `db.Model`. This is a base class provided by SQL Alchemy that allows us to "register" our class in the database engine. It also allows us to map tables and fields to classes and attributes.

Fields (columns) implementation and relationships

- All database fields present in the database schema are declared on the class as `mapped_column`.
- Each column / mapped column has a type: see `Integer`, `String`, `ForeignKey`, etc.
- And possibly additional options (similar to SQL field options): not null, primary key, index, etc.
- The **relationships** are automatically built by SQL Alchemy using the `relationship` call.
- Make sure you understand the difference between `player_id` (the foreign key in the database) and `player` (the relationship provided by SQL Alchemy).

With the classes above, it is now very easy to manipulate our entities in Python. If `my_score` is an instance of the `Score` class, then:

- `my_score.id` is the primary key (the ID from the DB)
- `my_score.score` is the score value (from the DB)
- `my_score.data` is the date the score was added (from the DB)
- `my_score.player` is the `Player` **OBJECT** that this score is related to (it is pulled from the DB using the foreign key `player_id`)

In the same way, if `tim` is an instance of the Player class, then:

- `tim.id` is the primary key (the ID of the player in the DB)
- `tim.name` is the name of the player (in the database)
- etc.
- `tim.scores` is a list of `Score` instances, which can be automatically built by looking at database records for scores where the `player_id` is equal to `tim.id`

Understand Flask and the starter code

Flask is a web micro-framework that allows Python developers to very quickly create web applications and APIs with minimal code involved. Because it is a micro-framework, it only takes care of the HTTP layer. We will have to setup the DB layer ourselves - with the help of the library `flask-sqlalchemy`.

Basic structure of a Flask application

The minimal version of a Flask application is:

```
from flask import Flask

app = Flask(__name__)

if __name__ == "__main__":
    app.run(debug=True, port=8888)
```

This will run a Flask web application, in debug mode, listening on port 8888. Run it with: `python app.py`.

> ⚠️ By default, the application listens only on the local interfaces, so it is not available to anyone else but you. This also means that you must access it using `localhost` or `127.0.0.1`.

> 💡 Running the app in debug mode sets up live reload: the app will reload itself whenever you make changes to the code. Make sure you reload the web pages in your browser, though...

> ⚠️ Make sure you follow the steps in the lab, unless **you really know what you are doing**. For example, the application could be run with `flask run`. However, in order for it to work well and consistently, you would need a full understanding of Flask, your terminal settings, your Python installation, and your system cofiguration which you are very unlikely to have at this stage. Just don't do it.

Understanding routes and URLs

The `app` variable is a Flask instance and can be used to interact with Web methods and HTTP requests. You can now create "routes" in your application that return data to the web browser. A Flask route is just a Python function, that returns at least one value, typically a string (= the HTML code to be displayed in the browser).

```
@app.route("/")
def home():
    return "HEY THERE"
```

The function above will be called whenever an HTTP request is made on the route `/` (which is the default URL). If you browse to `http://127.0.0.1:8888/` you should see the text above displayed in the browser. Try to change it and refresh the browser.

Using templates

It is very inconvenient to return raw HTML directly from the Python code, and it breaks best practices (separation of code and presentation). It is much better to use *templates*, aka HTML documents whose structure is fully created and complete. We can then just "fill" them with the information we want.

Flask looks for templates in the `templates` folder by default. You can render a template by calling the `render_template` function from Flask, and then return the value to the browser.

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

You can specify where "placeholders" should be in your template, and provide values for the placeholders in the `render_template` function.

```
<h3>My name is {{ name }}</h3>
```

With the HTML template above, `render_template` will replace `{{ name }}` with the value provided as the keyword argument `name=`. For example: `render_template("home.html", name="Tim")`.

`{{ ... }}` will update the template with the result of the Python expression provided. It is usually a variable.

You can also run logic by using `{% ... %}`. For example, the following will loop on the list `my_list`:

```
<ul>
    {% for element in my_list %}
        <li>List element: {{ element }}</p>
    {% endfor %}
</ul>
```

Using blocks and template scaffolding

It is very inconvenient to write a complete HTML file for every view / function we have in Flask. It is very likely that many if not all of our pages will have the same structure, and only specific "areas" (or "blocks") of the page will change.

This can be done by using *template scaffolding*, where you can build templates based on other templates.

Take a look at the template `base.html`. You can see that it contains a basic HTML document, with a specific instruction: `{% block content %}{% endblock %}`. This defines a new *block* that can be overriden by a child template.

Now, take a look at `home.html`. You can see that the template is very short.

1. It *extends* the base template with `{% extends "base.html" %}`.
2. It overrides the `content` block with its own data.

Look at the `app.py` and find the route for `/`. Make changes to the function to use the template. Display the template with your custom values.

> The base template uses the Bootstrap CSS framework.

Understand how Flask connects to the database

Flask uses the database through the SQLAlchemy library. The library provides us with a *database adapter*. The database adapter is instanciated in the `database.py` file, and is available at the module level under the variable `db`. Most of our interaction with the database will go through that variable.

> ⚠️ We need to define the `db` object in a separate Python module because `app` and `models` are very closely related - if we don't, it is vey easy to create a circular dependency in the code.

The `db` adapter will configure itself by using Flask configuration values.

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

This line tells the DB adapter to use the SQLite database in the file `hiscores.db`. We could use a MySQL, PostgreSQL, etc connection string too. We are using SQLite because it is "just a file" even though it has all the features of a full relational database.

Finally, the database adapter is set up when we call `db.init_app(app)`.

Create the tables (first run)

SQLAlchemy mostly takes care of communicating with the database, not setting it up. It expects the tables and columns to be created.

The first time you run the application, you need to create the database and its tables. This is a one-off operation, and is sometimes run by a script. For this lab, I made a dedicated route `/create-tables` in the Flask application that you can visit in your browser. It will create all required tables.

> 💡 If you want to "recreate" the database and start from scratch, just delete the database file and recreate the tables. You can also change the code and call `db.drop_all()` before recreating all tables.

Use the DB adapter and session to read/write records

Once the adapter is setup, all interaction with data goes through a `session`. A session is a virtual concept that represents the connection between the database and a client. Using sessions allows database access to be concurrent and safe, provides rollbacks, etc.

Flask-SQLAlchemy takes care of setting up a new session for each request to your Flask application. You can simply use `db.session` to access the current session object.

Session methods

- `db.session.execute`: executes a query on the database.
- `db.session.add(obj)`: adds an object.
- `db.session.delete(obj)`: deletes an object.

- `db.session.commit()`: commits the changes to the database.

IF YOU DON'T CALL `db.session.commit()`, ANY CHANGES YOU MADE WILL NOT BE SAVED IN THE DATABASE! YOU TYPICALLY NEED TO CALL IT (AT LEAST) ONCE PER REQUEST.

Look at the `create_players` route provided. It seeds the database with users (taken from a text file). For each user, it generates a random secure token and saves all to the database.

Write your own view: the players list

Create a new function for the route `/players`. Write the code that:

- selects all players from the database
- returns them in a table-like format showing their name and token

You will most likely need to create a new child template with your own logic. To execute a query:

1. buid the query with `db.select`, for example: `db.select(Player)`
2. execute the query: `db.session.execute`
3. extract the results

For example:

```
stmt = db.select(Player).where(Player.id < 10).order_by(Player.name)
data = db.session.execute(stmt)
# The results will be a tuple with a single element in it (the player object).
# We use .scalars() to get the actual element in the result.
results = data.scalars()
for elem in results:
    # We access full player objects and their attributes. We can even call their
methods!
    print(elem.name, elem.token)
    print(elem.to_dict())

stmt = db.select(Player.name, Player.token)
results = db.session.execute(stmt)
# We only select specific columns here
# The results will be a list of tuples (NAME, TOKEN)
for elem in results:
    print(elem[0], elem[1]) # We access elements of the tuple = columns
```

Flask-SQLAlchemy query examples

Add links and a new route with a URL parameter

Flask routes can take parameters: values that change depending on the HTTP request.

To add a new route which takes a `player_id` parameter, you can do:

```
@app.route("/player/<int:player_id>")
def player_view(player_id):
    # do something with player_id
```

Note that the function now takes the `player_id` argument, provided by Flask and extracted from the HTTP request. See for example: http://127.0.0.1:8888/player/12345. You can now select the appropriate player object from the database using the ID provided.

Add a new route to regenerate a player's token

Create the route `/player/PLAYER_ID/refresh_token`. When accessed, this route must:

- get the appropriate player from the database
- recreate a new secure token
- save the new token to the database
- redirect the user to, either:
    - the player page (`/player/PLAYER_ID`) if you have one
    - or the homepage, showing all players and their tokens

You can use the Flask function `redirect`: `redirect("/")`. **Don't forget to import it first!**

> Flask-SQLAlchemy has very convenient functions that deal with the logic of returning 404 pages when a record is not found in the database. Take a look at this page - you probably want to use `db.get_or_404` in this situation.

Add a link to the homepage to refresh tokens for any user

In the HTML table of players, add a new column. This column will contain the link that allows visitors to regenerate tokens for any user. You will need to use `<a href=....>` elements. Connect these links with the view you created above.

> ⚠ Do not write links manually! Use `url_for`.

It would be tempting to hardcode the links in the template, using URL `/player/.../refresh_token`. However, if the route changes in Flask (in the `@app` decorator), all the related links will break. Instead, you **SHOULD USE** `url_for` which returns the URL for a specific view with arguments.

```
<a href="{{ url_for('home') }}">Homepage</a>
```

THE ABOVE CREATES A LINK TO THE PAGE DISPLAYED BY THE FLASK ROUTE `def home(...)`. The argument is the name of the function.

You can also use parameters:

```
<a href="{{ url_for('player', player.id) }}">Player {{ player.name }}</a>
```

The above provides the url for the flask route `def player(...)` and builds the URL considering the argument is `player.id`.

Go further: use relationships and build more views

- Create a new route for the URL `/create-scores`.
- In this view, generate random scores by random players at random dates.
- Create a new route for `/player/PLAYER_ID` if you don't have one.
- On that page, show all scores submitted by that user in a table-like format.
- On the homepage, change the table to display:
  - a list of users in the first column
  - a list of all scores posted by that user ordered by date ascending in the second column, separated by `","`

You will need to use SQL `JOIN` statements to achieve this. Be careful about the difference between `INNER` and `OUTER` joins, as well as left/right approach. For example:

```
db.select(Player).where(Player.id == player_id).join(Score)
db.select(Player).where(Player.id == player_id).outerjoin(Score)
```

You will now be able to access `result.scores` as a list of score records linked to the player (note the `S`: we have several scores for one player, but one player for a given score).