



Politecnico
di Torino

Introduction to Web Applications

Authentication

Juan Pablo Sáenz



Goals

- Understand the concept of **sessions** in web applications
- Learn how **Flask-Login** manages user authentication
- Implement **login, logout, and user session persistence**
- **Protect routes** to restrict access to authenticated users
- Handle user loading and session management properly

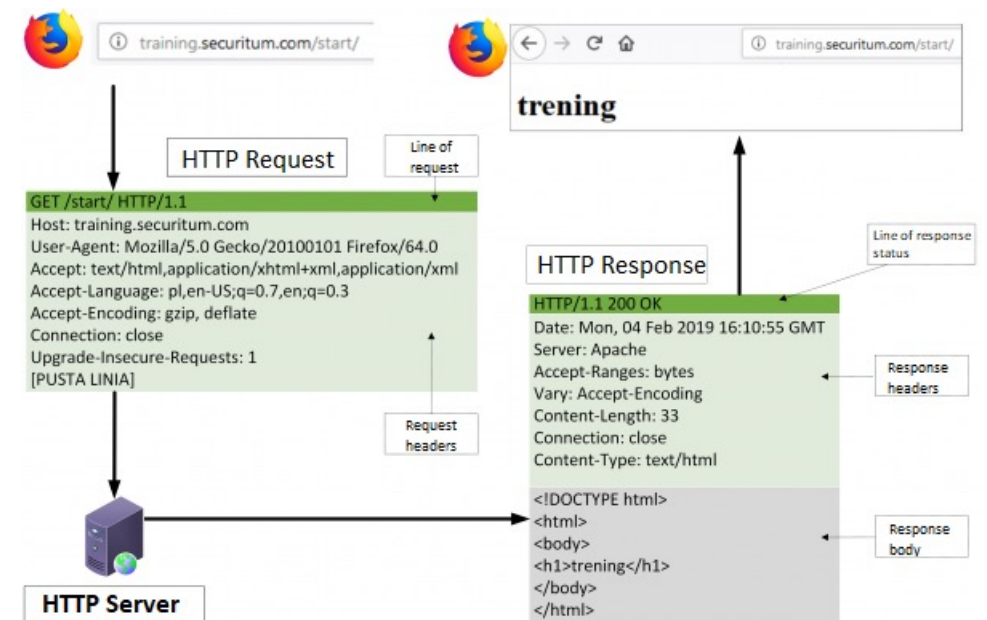
Sessions

HTTP is stateless

- Each request is independent and does not retain information from previous interactions

However, web applications often need to maintain information across multiple requests

- In an online shop, when we add a book to the shopping cart, we expect it to stay there
- As we browse other pages, our shopping cart («state») should be remembered



Sessions

A session is a **temporary and interactive exchange** of data between two or more parties (e.g., devices)

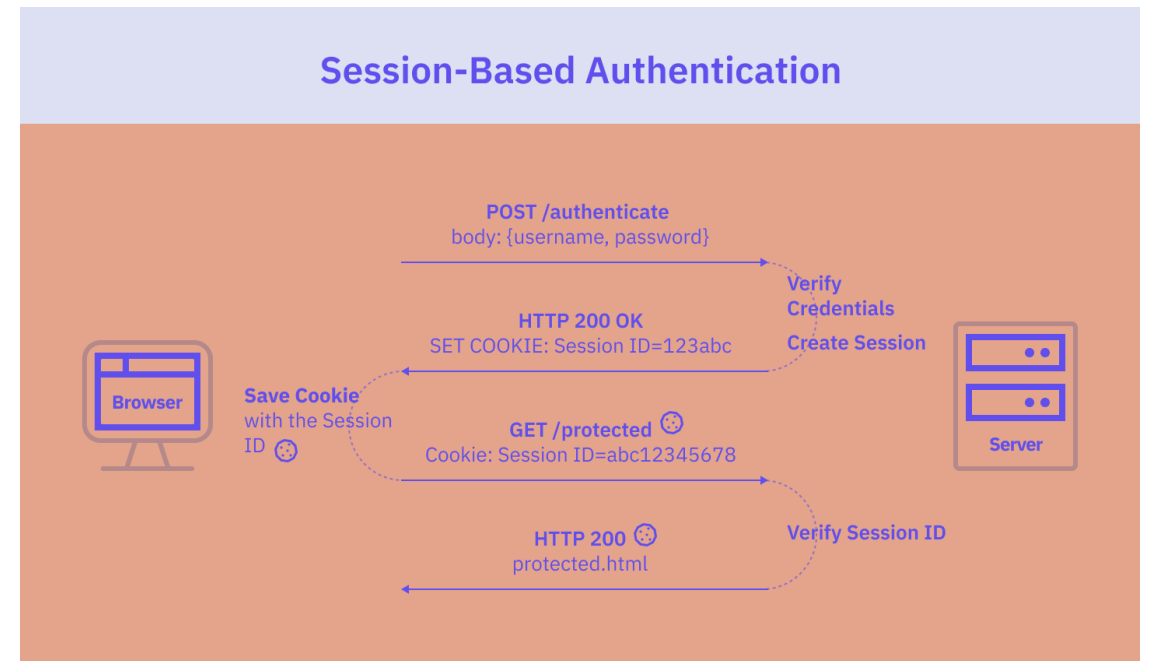
It involves **one or more messages** sent in each direction

Typically, **one party maintains the application state** during the session

A session has a **defined beginning** and **ends** at a later point

Session-Based Authentication

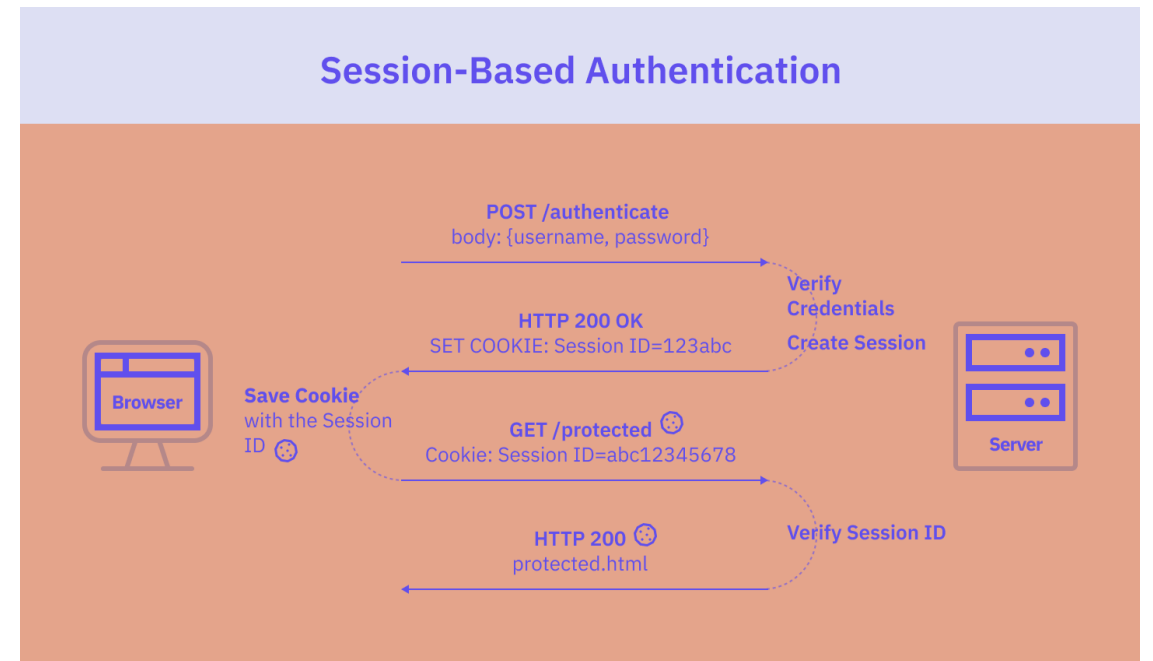
1. The user fills out a **form** with a username and password in the **client** application
2. The **client** validates the input and, if valid, sends it to the **server** through a **POST request**
3. The **server** receives the request, checks if the user exists, and verifies the password using **cryptographic hashes**
4. If the user is not found or the password does not match, the **server** responds with an error message like "Wrong username and/or password"



<https://www.criipto.com/blog/session-token-based-authentication>

Session-Based Authentication

5. If the credentials are correct, the **server** generates a **session ID**
6. The session ID, along with some user information retrieved from the **database**, is stored in the server's session storage
7. The **server** sends back an **HTTP response** containing a **cookie** with the **session ID**
8. The **browser** receives the **cookie**, stores it automatically, and the **web application** handles the response (e.g., displaying a "Welcome!" message)



<https://www.criipto.com/blog/session-token-based-authentication>

Sessions: Session ID

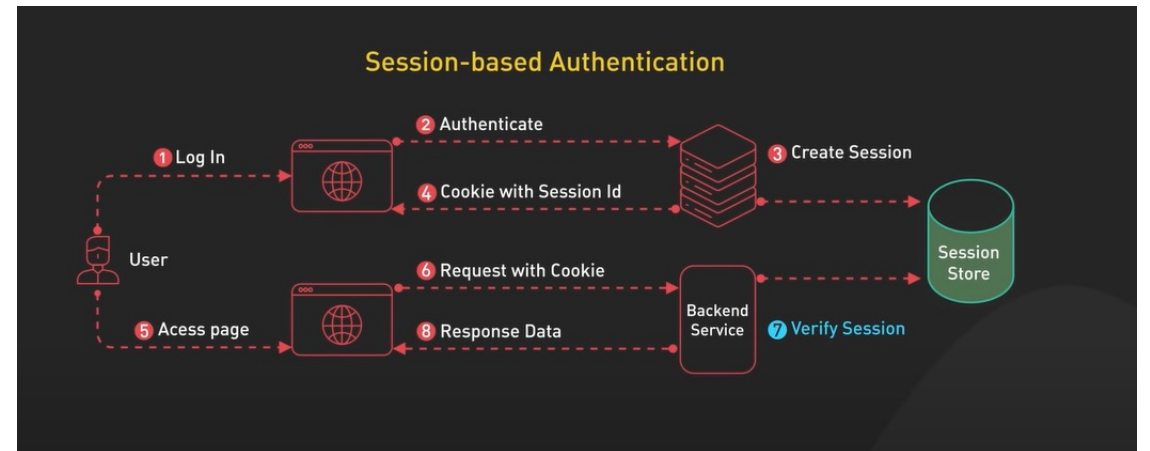
A **unique identifier assigned by the server** to maintain a session with the client

- It allows the server to **recognize the client** across multiple HTTP requests as authenticated

After authentication, the **server** sends a **session ID** to the **client**

The client sends the session ID back to the server with **every request** during the session

- Stored on the client side
- Sent automatically** with each request, typically via **cookies** 🍪



<https://levelup.gitconnected.com/understanding-web-authentication-session-based-authentication-vs-json-web-tokens-jwts-11871084f3ec>

Sessions: Cookies 🍪

A **small piece of information** stored by the **browser** in its internal cookie storage

- It allows the browser to **retain information across different requests** and **sessions** (e.g., **session IDs**, preferences, tracking)

The browser **automatically saves** cookies received from the server

Cookies are **automatically sent** back to the server with every request to the same **domain** and matching **path**

⚠ Sensitive information should never be stored in cookies!



Sessions: Session ID and Cookies 🍪

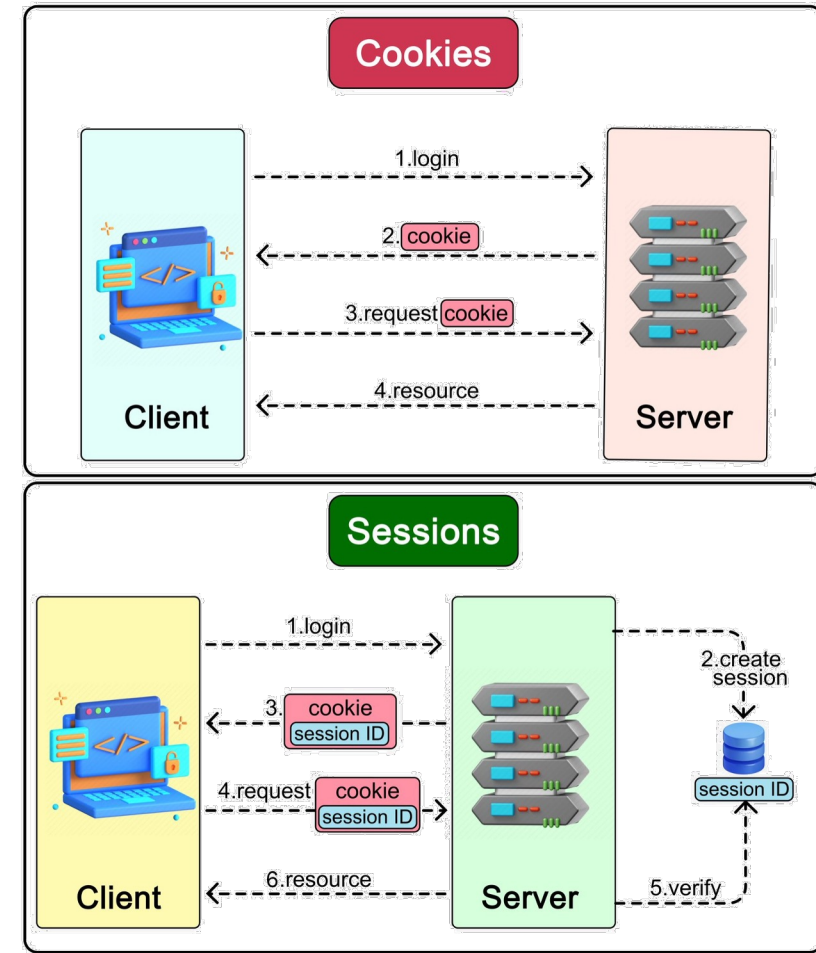
A **Session ID** is just a piece of data

- A **unique identifier** that the server uses to recognize a user across multiple HTTP requests

A **Cookie** is a storage mechanism in the **browser**

- A way to store small pieces of data like the Session ID

The Session ID is typically **stored inside** a cookie



<https://blog.bytebytego.com/p/ep90-how-do-sql-joins-work>

Sessions: Cookie attributes

name (mandatory): the name of the cookie

- Example: SessionID

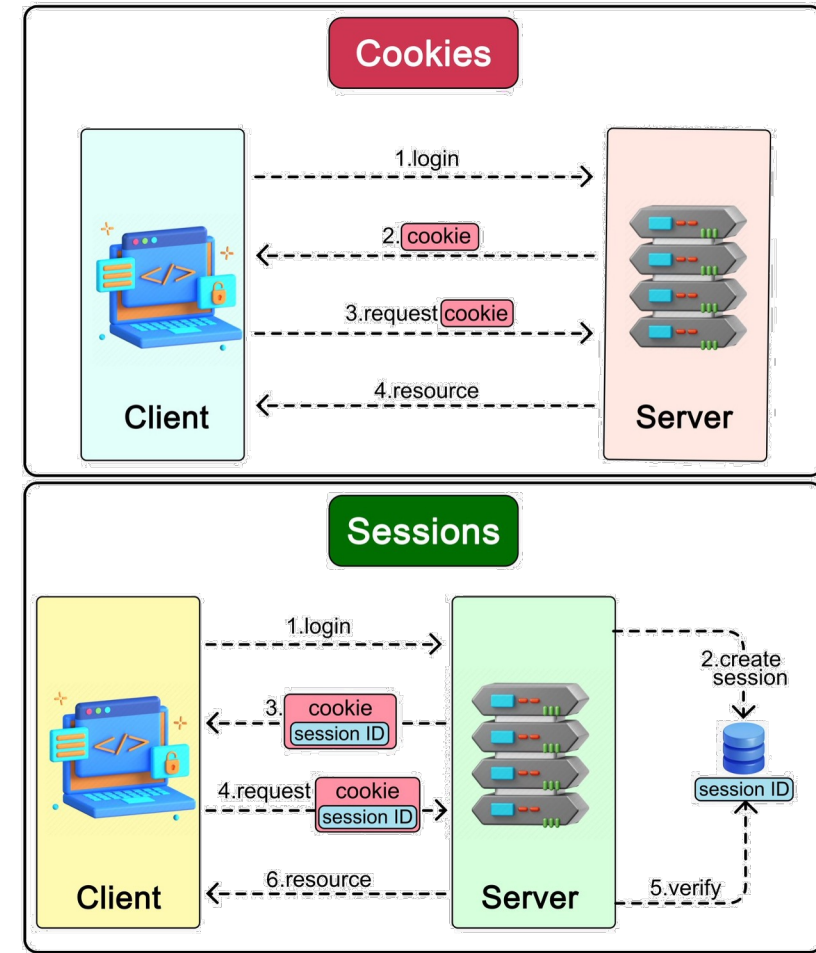
value (mandatory): the value stored in the cookie

- Example: 94\$KKDEC3343KCQ1!

secure: If set, the cookie is sent only over **HTTPS**

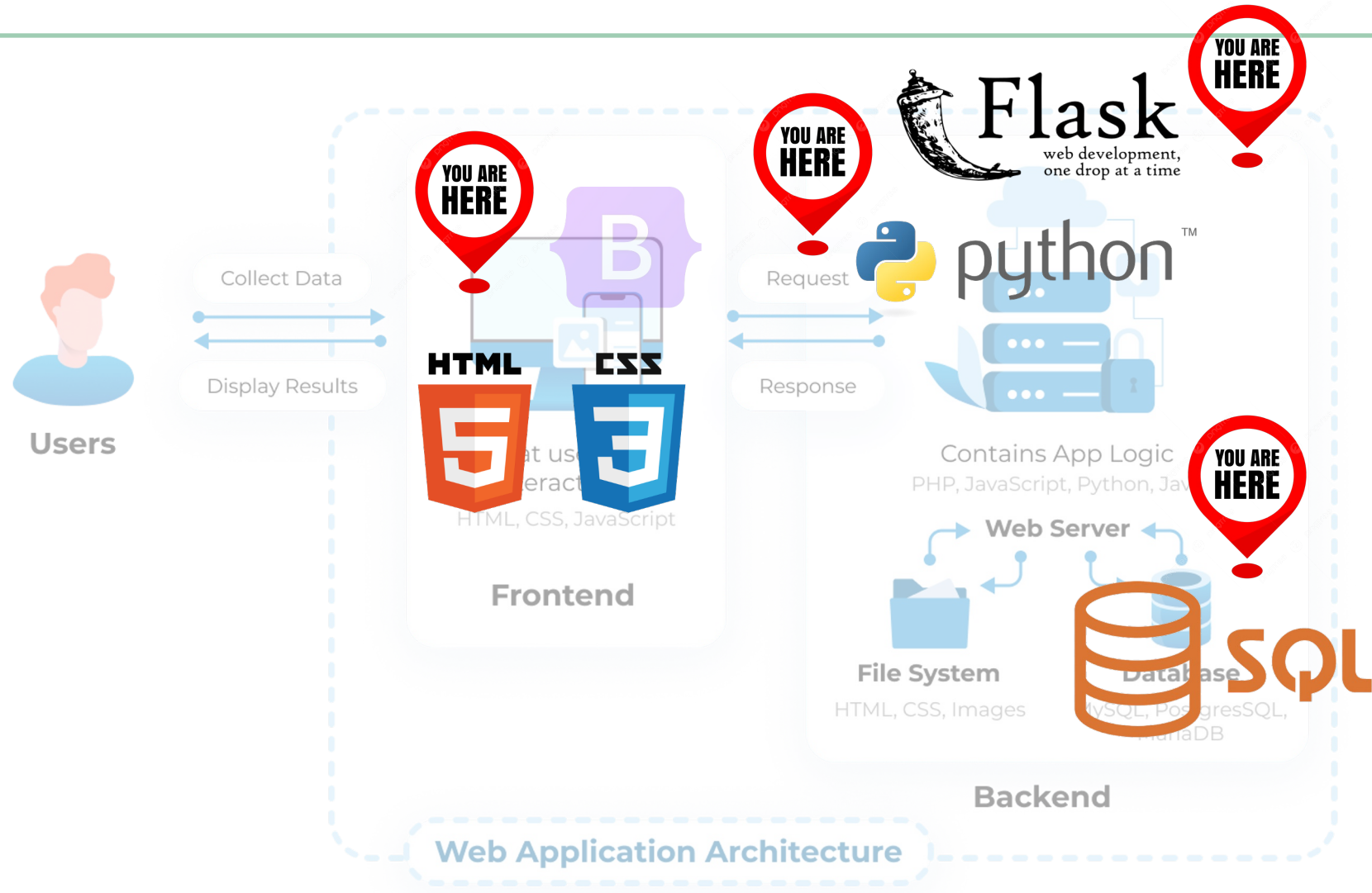
httpOnly: If set, the cookie cannot be accessed via **JavaScript**

expiration date: Specifies when the cookie should expire



<https://blog.bytebytego.com/p/ep90-how-do-sql-joins-work>

📍 Authentication: where are we?



Authentication and Authorization

Authentication: Verifies **who you are** (identity)

- Typically done using credentials (e.g., username and password)
- Enables a personalized user experience

Authorization: Determines **what you are allowed to access**

- Depends on the authenticated identity
- Grants permission to access resources, based on **roles**

👉 Used in conjunction to protect access to a system

Authentication vs authorization

Authentication	VS	Authorization
Verifying a user's identity before giving them permission to access a system, account, or file.	Definition	Verifying a user's access level to a system, account, or file.
To confirm the user's identity and prevent unauthorized access	Purpose	To ensure users can only access resources they are allowed to
Compares user credentials with stored data	Process	Grants or denies access based on roles/permissions
Username/password, OTP, security questions	Methods	Role-Based Access Control (RBAC), permissions



Authentication and Authorization

Developing authentication and authorization mechanisms

- Is **complicated**, **time-consuming**, and **prone to errors**
- May require integration with **third-party systems** (e.g., Google, Facebook login)
- Involves both **client** and **server**
- Requires understanding several new concepts

 **Best Approach:** Follow best practices and standardized processes

Authentication in Flask

Flask-Login is an extension that manages user authentication and session handling in Flask applications

 <https://flask-login.readthedocs.io/en/latest/>

Uses **sessions** to keep users logged in

Handles **login**, **logout**, and '**remember me**' functionality 

Stores the active user's ID in the **Flask session** 

Easily **log users in** and **out** 

Restrict access to views based on login status 

```
pip install flask-login
```

Flask-Login Setting Up

Flask-Login uses a **LoginManager**

- It defines **how to load a user** from an ID
- Where to **redirect** users when they need to **log in**

The **SECRET_KEY** is used to **sign session cookies**, making sure data sent by the client has not been modified

```
from flask import Flask
from flask_login import
LoginManager
```

```
app = Flask(__name__)
app.config["SECRET_KEY"] =
"arbitrary string"
```

```
login_manager = LoginManager()
login_manager.init_app(app)
```

Flask-Login Setting Up

We need to provide Flask-Login, at least, two things:

User model

- Represents what a user is in the app
- You decide what information to store for each user
- Can be based on any database system

user_loader callback

- Tells Flask-Login **how to load a user** from the session

Flask-Login: User model

The User model must implement the following properties for Flask-Login to work

- **is_authenticated**: Returns **True** if the user is logged in
- **is_active**: Returns **True** if the user's account is active (e.g., not suspended or deactivated)
- **is_anonymous**: Returns **True** if the user is not logged in
- **get_id()**: Returns a unique **str** identifier for the user (used by the **user_loader**)

Flask-Login: User model

UserMixin provides default implementations for the methods that Flask-Login requires

We can inherit from **UserMixin**

```
from flask_login import UserMixin

class User(UserMixin):
    def __init__(self, id, name,
surname, email, password):
        self.id = id
        self.name = name
        self.surname = surname
        self.email = email
        self.password = password
```

Flask-Login: user_loader

We need to tell Flask-Login **how to load a user** from a Flask request and from its session

To do this, we define a **user_loader** callback

```
@login_manager.user_loader
def load_user(user_id):
    db_user = dao.get_user_by_id(user_id)
    user = User(id=db_user["id"],
               name=db_user["nome"],
               surname=db_user["cognome"],
               email=db_user["email"],
               password=db_user["password"],)
    return user
```

Flask-Login: login_user()

Logs a user in: we should pass the actual **User** object to this method

Returns **True** if the log in attempt succeeds, and **False** if it fails

```
from flask_login import login_user

@app.route("/login", methods=["POST"])
def login():
    user_form = request.form.to_dict()

    (...)

    new = User(id=user_form["id"],
              name=user_form["nome"],
              surname=user_form["cognome"],
              email=user_form["email"],
              password=user_form["password"],)

    login_user(new)

    return redirect(url_for("profile"))
```

Flask-Login: login_required

Views that require users to be logged in can be decorated with the **login_required** decorator

```
from flask_login import login_required

(...)

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

Flask-Login: `logout_user()`

Logs out a user: any **cookies** for the **session** will be cleaned up

```
from flask_login import logout_user

(...)

@app.route("/logout")
@login_required
def logout():
    logout_user()
    return redirect(url_for('home'))
```

Flask-Login: `current_user`

We can access the logged-in user with the **`current_user`** proxy, which is available in **every template**

```
# app.py
from flask_login import current_user
# For instance, anywhere in the code:
post['id_utente'] = int(current_user.id)
```

```
<!-- templates/home.html -->
{% if current_user.is_authenticated %}
    Hi {{ current_user.name }}!
{% endif %}
```

Let's see it in practice

Storing passwords in the Server

👉 **Never store plain text passwords** (e.g., in the database)

- Always **hash passwords** before storing them
- Hashing is a **one-way function**, ensuring passwords cannot be retrieved from their hash

werkzeug.security is a Python library that we can use



<https://werkzeug.palletsprojects.com/en/stable/utils/>

```
pip install werkzeug
```

```
from werkzeug.security import
generate_password_hash,
check_password_hash

(...)

new_user = {
    "name": name,
    "surname": surname,
    "email": email,
    "password":
    generate_password_hash(password,
    method='scrypt')
}
```

Let's see it in practice



- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
 - **Share** – copy and redistribute the material in any medium or format
 - **Adapt** – remix, transform, and build upon the material
 - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
 - **Attribution** – You must give [appropriate credit](#), provide a link to the license, and [indicate if changes were made](#). You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
 - **NonCommercial** – You may not use the material for [commercial purposes](#).
 - **ShareAlike** – If you remix, transform, or build upon the material, you must distribute your contributions under the [same license](#) as the original.
 - **No additional restrictions** – You may not apply legal terms or [technological measures](#) that legally restrict others from doing anything the license permits.
- <https://creativecommons.org/licenses/by-nc-sa/4.0/>