Intro
<AppWeb/>
2022

Authentication

Authentication in Flask with Flask-Login

Juan Pablo Sáenz

Politecnico di Torino

Outline

- The need for authentication
- HTTP sessions
- Authentication in Flask

Authentication vs. Authorization

Authentication

- Verify you are who you say you are (identity)
- Typically done with credentials
  - e.g., username, password
- Allows a personalized user experience

Authorization

- Decide if you have permission to access a resource
- Granted authorization rights depends on the identity
  - as established during authentication

Often used in conjunction to protect access to a system

Authentication and Authorization

- Developing authentication and authorization mechanisms
  - is complicated
  - is time-consuming
  - is prone to errors
  - may require interacting with third-party systems (login with Google, Facebook, …)
  - …

Authentication and Authorization

- Involve both client and server
  - and requires to understand several new concepts
- Better if you rely upon
  - best practices and “standardized” processes
  - **advice by security experts**!

Layers of Authorization

Who	What	How	When
User	Login / Logout / Navigate pages		
Flask App	Is the user logged? Remember user information	**Flask-Login**	Set at login Destroyed at logout Queried during navigation
Browser	Remembers navigation session	Session Cookie (stores session ID)	Received at login, in HTTP Response Re-sent to server at every HTTP Request
Server	Remember session data	Session storage (creates session ID, remembers associated data: username, group, level, …)	Created at login Destroyed at logout Retrieved at every HTTP Request
Route (HTTP API)	Check authorization Execute API	Verify session validity	At every (non-public) HTTP Request
Route (Login)	Perform authentication	Check user/pass If ok, create session information	At Login time
Route (Logout)	Forget authentication	Destroy session information	At Logout request
Database (at Login)	Validates user information	Queries & password encryption	At Login time
Database (HTTP API)	Retrieves user information	Queries from session information	At every HTTP Request

Giving memory to HTTP

COOKIES AND SESSIONS

Sessions

- **HTTP is stateless**
  - each request is independent and must be self-contained
- A web application may need to keep some information between different interactions
- For example:
  - in an on-line shop, we put a book in a shopping cart
  - we do not want our book to disappear when we go to another page to buy something else!
  - we want our "state" to be remembered while we navigate through the website

Sessions

- A **session** is temporary and interactive data interchanged between two or more parties (e.g., devices)
- It involves one or more messages in each direction
- Often, one of the parties keeps the state of the application
- It is established at a certain point it time and ended at some later point

Session ID

- Basic mechanism to maintain session
- Upon authentication, the client receives from the server a session ID
- The session ID allows the server to recognize subsequent HTTP requests as *authenticated*
- Such an information
  - must be stored on the client side
  - must be sent by the client at every request which is part of the session
  - must not be sensitive!
- Typically stored in and sent as **cookies**

Cookie

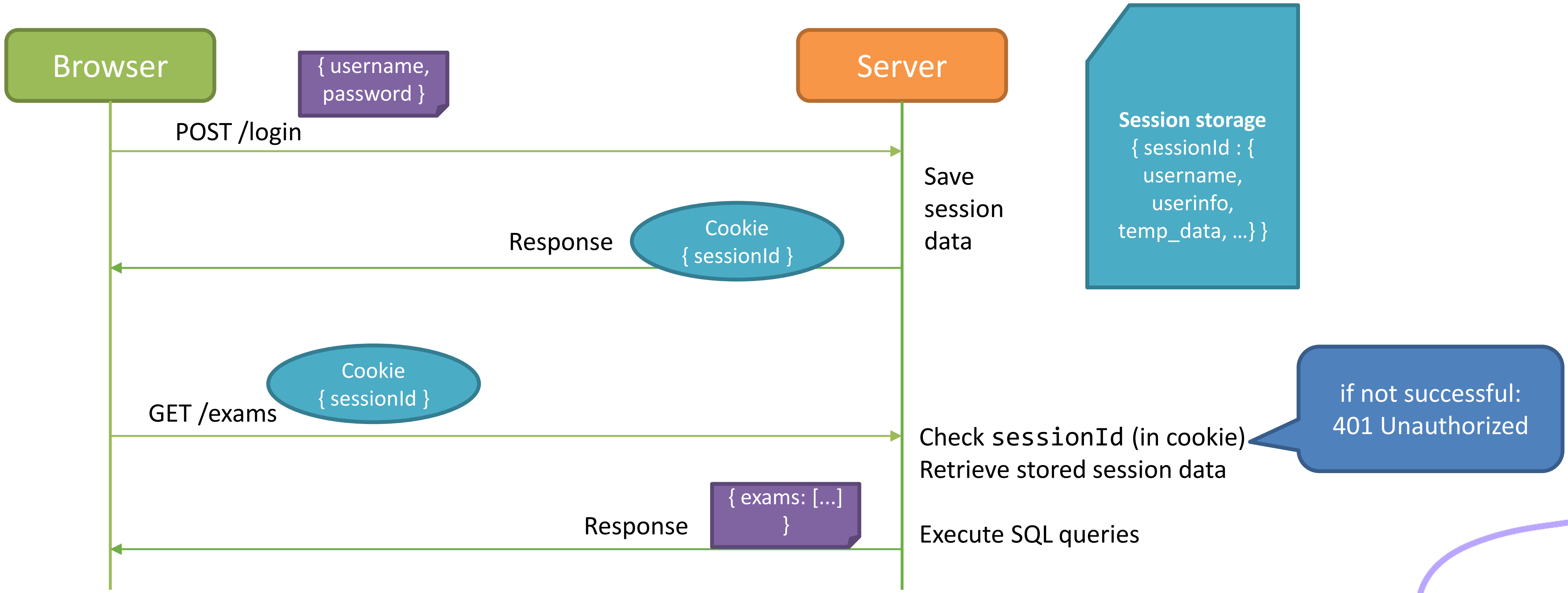
- A small portion of information stored in the browser (in its cookie storage)
- Automatically handled by browsers
- Automatically sent by the browser to servers when performing a request to the same **domain** and **path**
  - options are available to send them in other cases
- Keep in mind that sensitive information should **NEVER** be stored in a cookie!

Cookie

- Some relevant attributes, typically set by the server:
  - **name**, the name of the cookie [mandatory]
    - Example: `SessionID`
  - **value**, the value contained in the cookie [mandatory]
    - Example: `94$KKDEC3343KCQ1!`
  - **secure**, *if set*, the cookie will be sent to the server over HTTPS, only
  - **httpOnly,** *if set*, the cookie will be inaccessible to JavaScript code running in the browser
  - **expiration date**

Session-based Auth

- The user state is stored on the server
  - in a storage or, for development only, in memory



A Note About Security…

- **Always** use HTTPS and "secure" cookies (at least in production)
  - use "httpOnly" cookies
- **Never** store sensitive information into cookies
- Rely on **best practices** and avoid to *re-invent the wheel* for auth
- Web applications can be exposed to several "basic" attacks
  - *CSRF* (Cross-Site Request Forgery), a user is tricked by an attacker into submitting a request that they did not intend
  - *XSS* (Cross-Site Scripting), attackers inject malicious JS code into web pages
  - Most of these can be prevented with a proper usage of frameworks, best practices, and dedicated libraries

Base Login Flow

1. A user fills out a form in the client with a unique user identifier and a password
2. Data is validated and, if ok, is sent to the server, with a POST API
3. The server receives the request and checks whether the user is already registered, and the password matches
   Password comparison exploits cryptographic hashes
4. If not, it sends back a response to the client
   "Wrong username and/or password"

Base Login Flow

5. If username and password are correct, the server generates a session id
6. The server stores the session id (together with some user info retrieved by the database) in its "server session storage"
7. The server replies to the login HTTP request by creating and sending a cookie
8. The browser receives the response with the cookie
   - the cookie is automatically stored by the browser
   - the response is handled by the web application (e.g., to say "Welcome!")

Relying on Flask-Login

AUTHENTICATION IN FLASK

Authentication with Flask-Login

- Flask-Login provides user session management for Flask
  - **Flask-Login**, https://flask-login.readthedocs.io/en/latest/
  - install with: `pip install flask-login`
- It handles the common tasks of logging in, logging out, and remembering your users' sessions over extended periods of time
  - store the active user's ID in the Flask Session, and let you easily log them in and out
  - restrict views to logged-in (or logged-out) users
  - handle the normally-tricky "remember me" functionality

Flask-Login: Configuration

- Flask-Login works via a login manager
- It contains the code that lets your application and Flask-Login work together
  - how to load a user from an ID
  - where to send users when they need to log in
- Once the actual application object has been created:
  - **`login_manager.init_app(app)`**

Flask-Login: Configuration

- Flask-Login uses sessions for authentication
- We must set the **SECRET_KEY** on our application, otherwise Flask will give us an error message

```
from flask import Flask
from flask_login import LoginManager

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret'

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

Flask-Login: Requirements

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

- A **User model** with a set of properties and methods implemented in it
  - it represents what it means for the app to have a user
  - we can decide which information we want to store per user
  - the User model can be based on any database system
- A **`user_loader`** callback
  - specify how to load a user from a Flask request and from its session

Flask-Login: User model

- Flask-Login requires a User model with the following properties:
  - **`is_authenticated`**: should return **True** if the user is authenticated
  - **`is_active`**: should return True if this is an active user.
    In addition to being authenticated, they also have activated their account, not been suspended, or any condition our application has for rejecting an account
  - **`is_anonymous`**: should return True if this is an anonymous user
  - **`get_id()`**: this method must return a str that uniquely identifies this user, and can be used to load the user from the **`user_loader`** callback

Flask-Login: User model

- **`UserMixin`** provides default implementation for the methods that Flask-Login expects user objects to have
- Therefore, 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 need to define our **User model** and 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

(…)

new = User(id=user['id'], name=user['nome'],
surname=user['cognome'],
email=user['email'],
password=user['password'])

login_user(new)

return redirect(url_for('profile'))
```

Flask-Login: `login_required`

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

```
from flask_login import login_required

(…)

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

Flask-Login: `logout_user()`

- When the user is ready to log out
- Any cookies for the session will be cleaned up

```python
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

```
from flask_login import current_user

(…)

{% if current_user.is_authenticated %}
  Hi {{ current_user.name }}!
{% endif %}
```

Storing Passwords in the Server

- **Never** store plain text passwords in the server (e.g., in the database)
- **Always** perform hashing of the password
  - so that nobody can retrieve your password, knowing its hash
  - as hashing is a one-way function

Storing Passwords in the Server

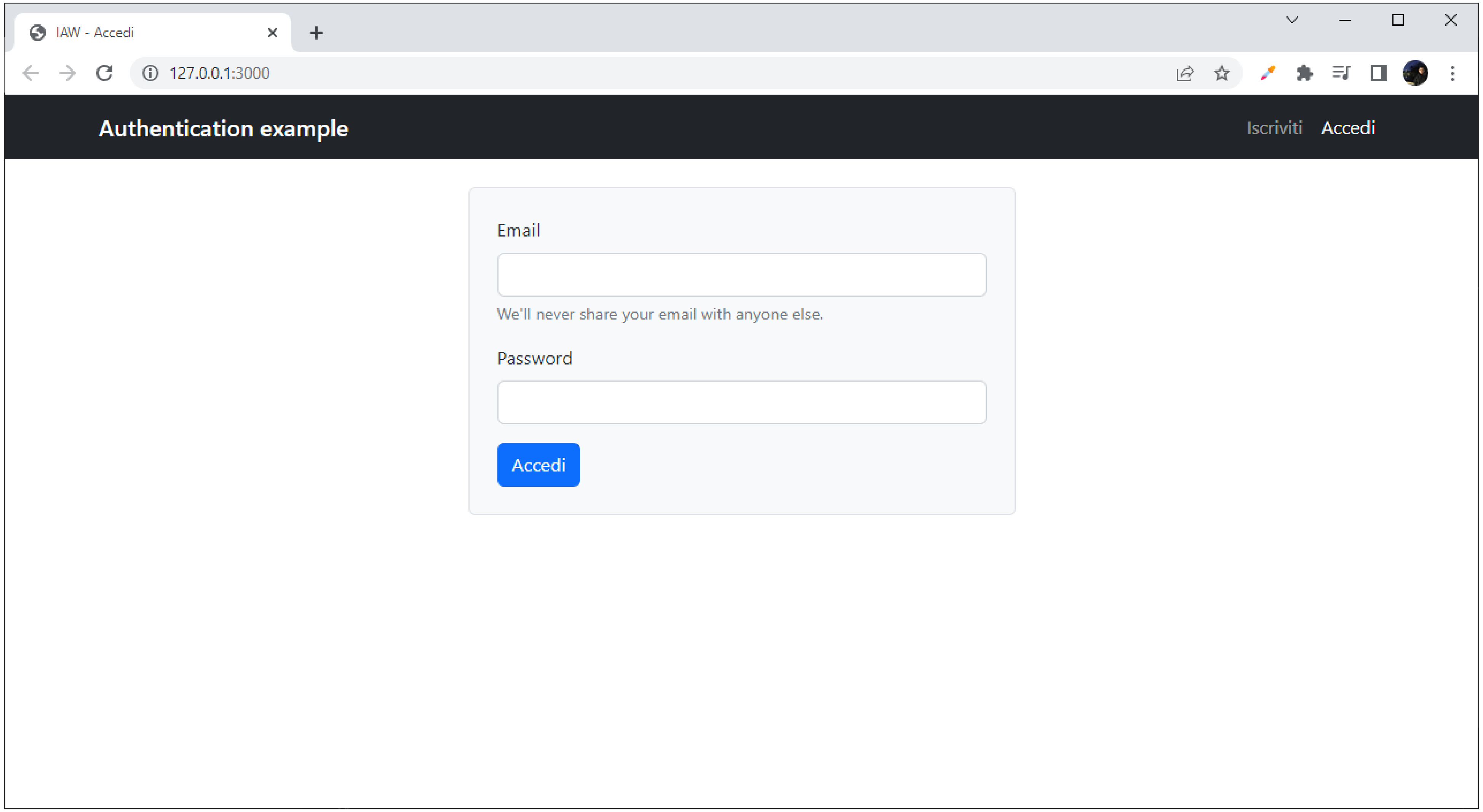
- **`werkzeug.security`** is a Python library that we can use
  - e.g., password -> d72c87d0f077c7766f2985dfab30e8955c373a13a1e93d315203939f542ff86e
- **`pip install werkzeug`**

```
from werkzeug.security import
generate_password_hash, check_password_hash

(…)

new_user = {
            "name": name,
            "surname": surname,
            "email": email,
            "password":
generate_password_hash(password,
method='sha256')
        }
```

LET'S GET TO WORK


IAW - Accedi

127.0.0.1:3000

Authentication example

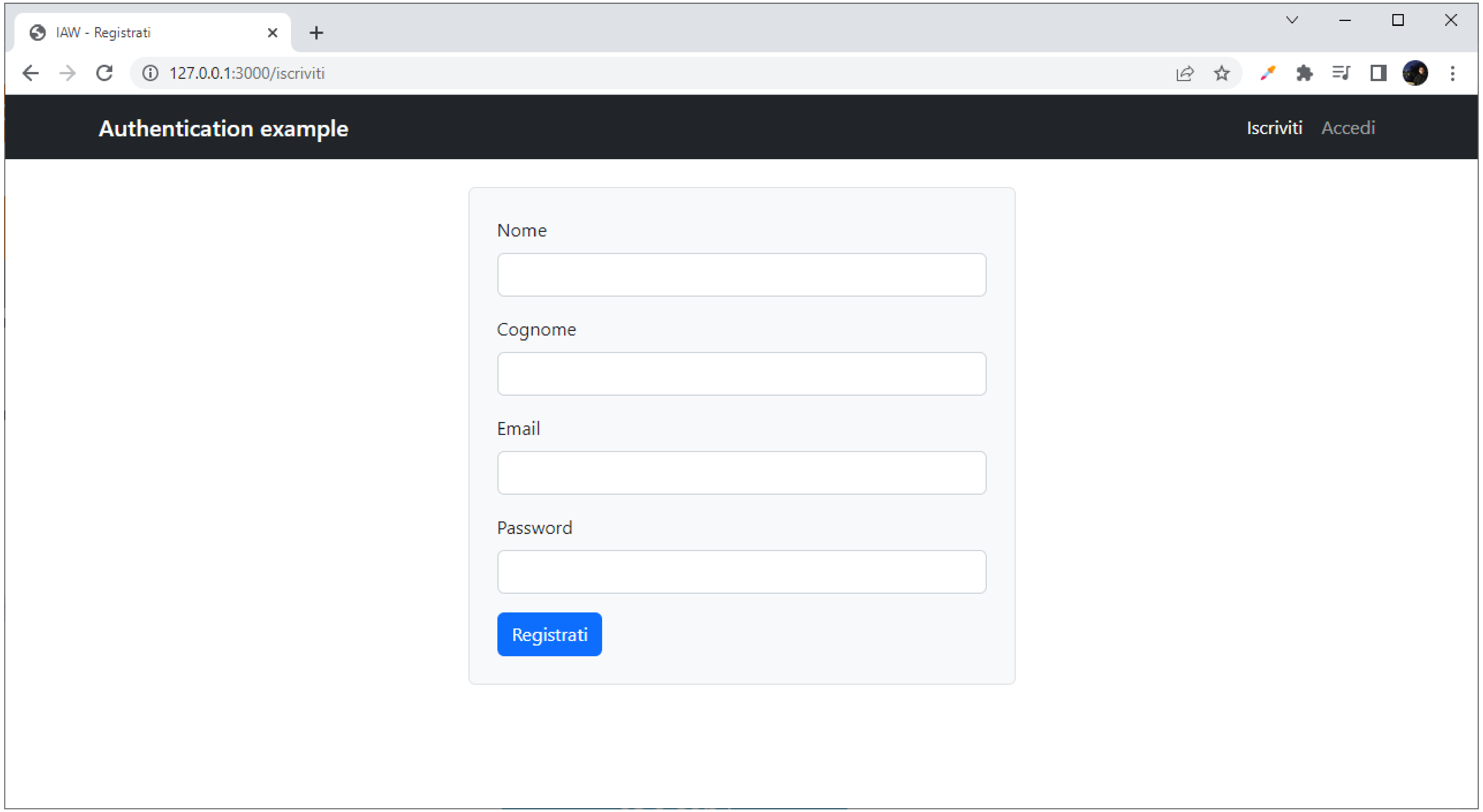
Iscriviti Accedi

Email

We'll never share your email with anyone else.

Password

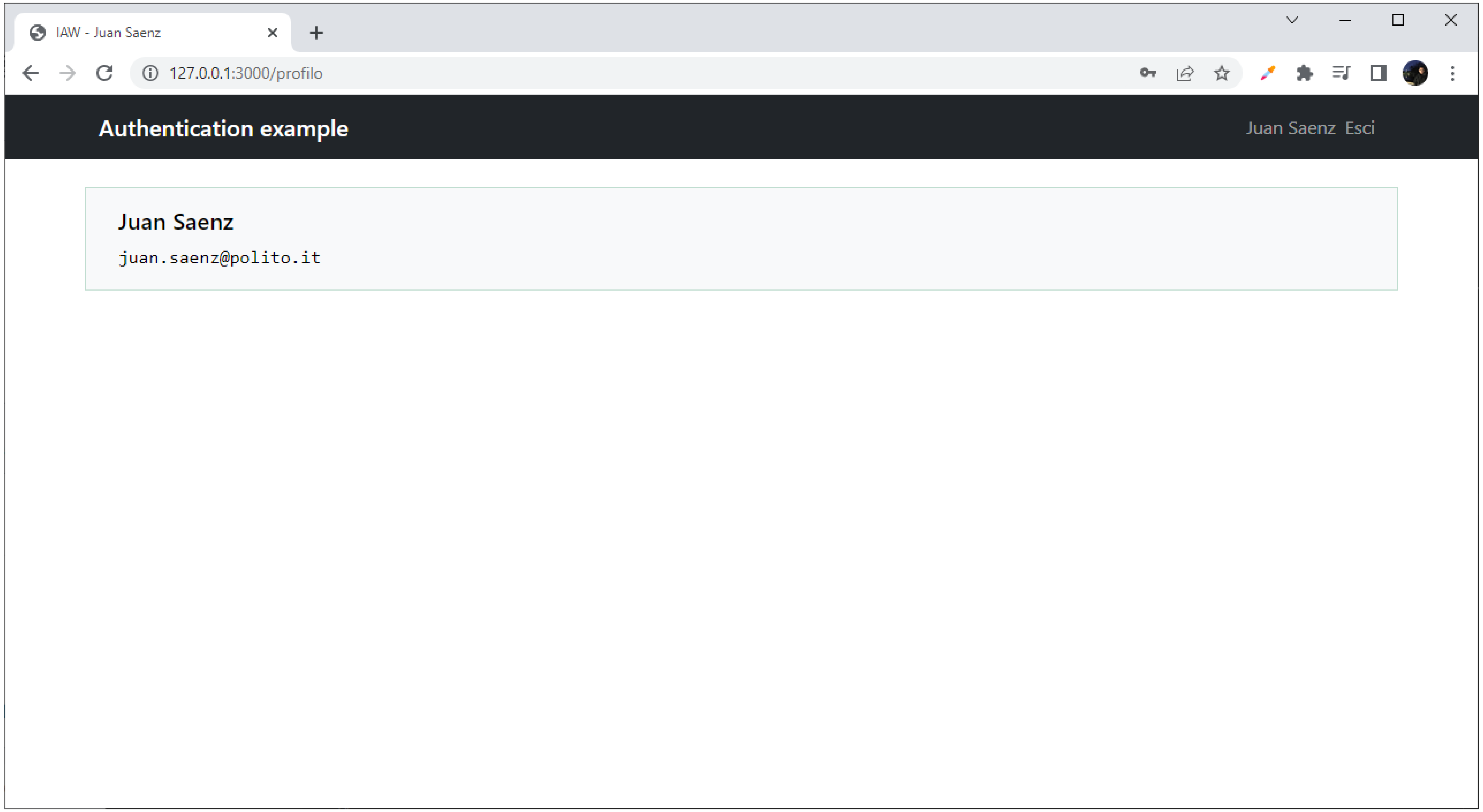
Accedi


IAW - Registrati

127.0.0.1:3000/iscriviti

Authentication example

Nome
Cognome
Email
Password



Juan Saenz

juan.saenz@polito.it

License

- 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/