

Relazione elaborato Programmazione di Reti

A.A. 2022/2023

Traccia 3: Python Web Server

*Marco Antolini
0000977047
marco.antolini6@studio.unibo.it*

Indice

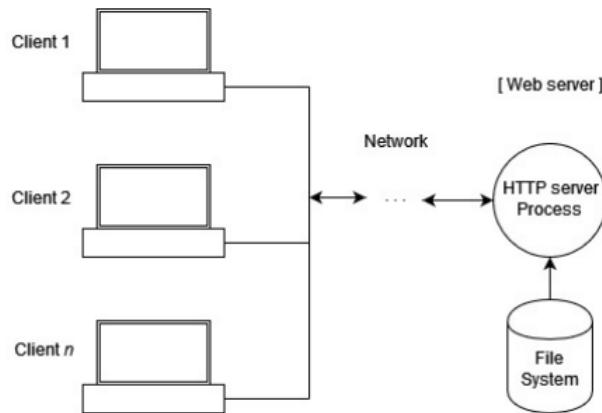
1	Analisi	3
2	Progettazione	3
2.1	Front-end	4
2.1.1	Template HTML	4
2.1.2	CSS	4
2.1.3	JavaScript	5
2.2	Back-end	6
2.2.1	Funzionamento generale	6
2.2.2	Gestione view tramite Flask	7
2.2.3	Gestione database tramite flask_sqlalchemy	8
2.2.4	Gestione autenticazione tramite flask_login e werkzeug.security	9
2.2.5	Gestione errori (status code)	10
3	Esecuzione	11

1 Analisi

Requisiti

Si immagini di dover realizzare un Web Server in Python per una agenzia di viaggi. I requisiti del Web Server sono i seguenti:

- Il web server deve consentire l'accesso a più utenti in contemporanea.
- La pagina iniziale deve consentire di visualizzare la lista dei servizi erogati dall'agenzia di viaggi e per ogni servizio avere un link di riferimento ad una pagina dedicata presente nella stessa working directory della pagina principale.
- Nella pagina principale dovrà anche essere presente un link per il download di un file pdf da parte del browser.
- Come requisito facoltativo si chiede di autenticare gli utenti nella fase iniziale della connessione.
- L'interruzione da tastiera (o da console) dell'esecuzione del web server deve essere opportunamente gestita in modo da liberare la risorsa socket.



2 Progettazione

Per la realizzazione del progetto è stato utilizzato il linguaggio di programmazione Python, in particolare grazie all'aiuto della libreria Flask che fornisce un'interfaccia per la creazione di applicazioni web in Python e comprende numerose funzionalità che facilitano la creazione e la gestione di un web server.

La libreria Flask comprende al suo interno la libreria Jinja2 che permette di creare template HTML che possono essere utilizzati per la creazione di pagine web dinamiche.

Quindi tramite Python ho gestito lo sviluppo del back-end, mentre per il front-end ho utilizzato i template HTML, assieme assieme a CSS puro per la gestione del layout e della grafica e JavaScript per la gestione delle funzionalità dinamiche.

2.1 Front-end

2.1.1 Template HTML

Per quanto riguarda il front-end, come introdotto in precedenza, ho usato la libreria Jinja2 (parte di Flask) che permette di creare dei template HTML, che verranno poi caricati dalle funzioni di view del back-end.

Jinja2 permette di creare template di base che possono essere estesi e personalizzati per ogni pagina web. Ho quindi creato un template base.html che contiene la struttura base di ogni pagina web (come ad esempio il layout, il menu di navigazione, il footer, ecc.) e poi ho creato dei template che estendono il template base e che vengono utilizzati per le varie pagine web.

Come si può notare nell'immagine è possibile utilizzare del codice python all'interno del template HTML racchiuso all'interno di parentesi graffe {}, questo codice viene poi eseguito dal back-end e il risultato viene poi inserito nel template HTML. Inoltre è possibile notare che possono essere usate delle variabili (come ad esempio request.path) all'interno del template se opportunamente passate dal back-end. Il template base al suo interno contiene dei tag block che vengono poi implementati dai template che estendono il template base e vi accedono grazie al nome assegnato al tag block (nel mio esempio title e content).

Come si può notare nell'immagine, il template welcome.html estende il template base.html e quindi può accedere ai tag block del template base. Inoltre è possibile notare che è possibile passare delle variabili al template base tramite il tag extends (nel mio esempio title e content).

```
<!DOCTYPE html>

    <head>
        <meta charset="utf-8" />
        <title>% block title %</title>
        <link rel="stylesheet" href="/static/css/style.css" />
        <link rel="icon" type="image/x-icon" href="https://raw.githubusercontent.com/MarcoAntolini/marco-antolini/main/images/favicon.ico" />
        <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, user-scalable=no" />
        <script src="/static/js/script.js"></script>
    </head>
    <body>
        <header>
            <div class="nav">
                {% for title, url in nav %}
                    <a class="nav-item" href="{{url}}>{{title}}</a>
                {% endfor %}
            </div>
        </header>
        <main>% block content %</main>
        <footer>
            <div>
                <p>Contact us at:</p>
                <ul>
                    <li><a href="https://github.com/MarcoAntolini">github.com/MarcoAntolini</a></li>
                    <li><a href="mailto:marco.antonini@studio.unibo.it">marco.antonini@studio.unibo.it</a></li>
                </ul>
            </div>
            <p>Copyright © 2023. All rights Reserved.</p>
        </footer>
    </body>
</html>
```

Template base.html

```
1  :> extends "base.html" %} {% block title %}Welcome | Fly Away{% endBlock %} {% block content %}
2  <div class="top-log title sub-title">login to access our website</h1>
3  <div class="form">
4      <form method="post" name="form">
5          <input type="text" name="username" id="username" placeholder="username" required autocomplete="off" />
6          <input type="password" name="password" id="password" placeholder="password" required autocomplete="off" />
7          <input type="hidden" name="submit_type" id="submit_type" value="register" />
8          <input type="button" name="submit" value="Register" id="submit" /><br />
9          {% if error %}
10             <div class="error">{{error}}</div>
11         {% endif %}
12         <input type="button" name="class" value="Already registered?" id="switch" class="register" /><br />
13     </form>
14 </div>
15 {% endBlock %}
```

Esempio di template esteso (welcome.html)

2.1.2 CSS

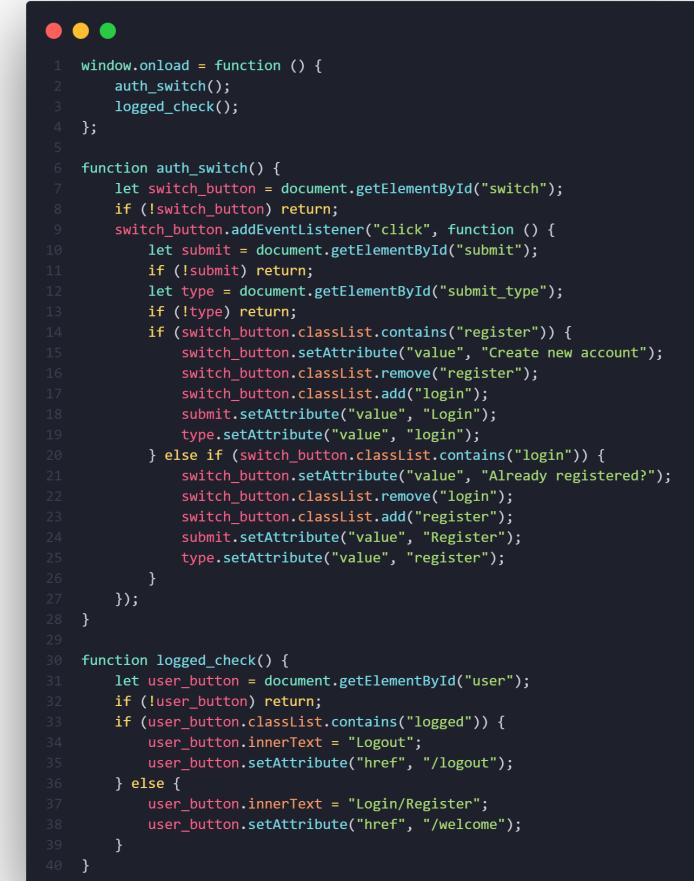
Per quanto riguarda il CSS ho utilizzato solo CSS puro ed esterno (quindi un file style.css che accede ad ogni elemento grazie al suo tag HTML o alla sua classe), senza utilizzare nessuna libreria o framework.

```
/* NORMALIZE */
1  html {
2      font-family: "Arial", sans-serif;
3      line-height: 1.5;
4      margin: 0;
5      padding: 0;
6      overflow-x: hidden;
7  }
8
9  body {
10     font-size: 14px;
11     background-image: linear-gradient(rgba(255, 255, 255, 0.6), rgba(255, 255, 255, 0.6)), url("../images/bg.png");
12     background-size: cover;
13 }
14
15 a {
16     text-decoration: none;
17 }
18 a,
19 a:is(:visited, :hover, :active, :focus) {
20     color: blue;
21 }
```

Esempio di CSS utilizzato

2.1.3 JavaScript

Per quanto riguarda JavaScript, ho implementato solo due funzioni di base per cambiare dinamicamente il contenuto di alcuni elementi quindi non ho avuto bisogno di nessuna libreria o framework.



```
1 window.onload = function () {
2     auth_switch();
3     logged_check();
4 };
5
6 function auth_switch() {
7     let switch_button = document.getElementById("switch");
8     if (!switch_button) return;
9     switch_button.addEventListener("click", function () {
10         let submit = document.getElementById("submit");
11         if (!submit) return;
12         let type = document.getElementById("submit_type");
13         if (!type) return;
14         if (switch_button.classList.contains("register")) {
15             switch_button.setAttribute("value", "Create new account");
16             switch_button.classList.remove("register");
17             switch_button.classList.add("login");
18             submit.setAttribute("value", "Login");
19             type.setAttribute("value", "login");
20         } else if (switch_button.classList.contains("login")) {
21             switch_button.setAttribute("value", "Already registered?");
22             switch_button.classList.remove("login");
23             switch_button.classList.add("register");
24             submit.setAttribute("value", "Register");
25             type.setAttribute("value", "register");
26         }
27     });
28 }
29
30 function logged_check() {
31     let user_button = document.getElementById("user");
32     if (!user_button) return;
33     if (user_button.classList.contains("logged")) {
34         user_button.innerText = "Logout";
35         user_button.setAttribute("href", "/logout");
36     } else {
37         user_button.innerText = "Login/Register";
38         user_button.setAttribute("href", "/welcome");
39     }
40 }
```

Funzioni JavaScript utilizzate

2.2 Back-end

2.2.1 Funzionamento generale

L'applicazione viene eseguita dal file `main.py` che richiama la funzione `create_app()` del file `__init__.py` che crea l'applicazione Flask e la configura insieme al database, al login manager e ai blueprint. Dopodiché viene eseguita la funzione `run()` che avvia il server Flask.

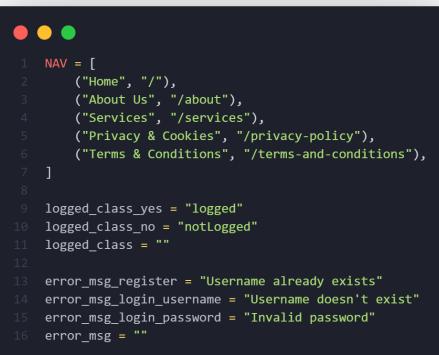
All'interno del file `__init__.py` vengono inoltre registrati i blueprint che permettono di gestire le varie funzionalità dell'applicazione dai file `views.py` e `auth.py`.



```
from website import create_app
app = create_app()
if __name__ == "__main__":
    # app.run(debug=True)
    app.run()
```

`main.py`

Oltre a questi file, sono presenti anche i file `models.py`, che contiene la classe dell'oggetto che andrà inserito nel database, e `utils.py` che contiene alcune variabili a cui hanno accesso le view (sia generali sia quelle riguardanti l'autenticazione) che poi invieranno come parametri ai template HTML.



```
NAV = [
    ("Home", "/"),
    ("About Us", "/about"),
    ("Services", "/services"),
    ("Privacy & Cookies", "/privacy-policy"),
    ("Terms & Conditions", "/terms-and-conditions"),
]
logged_class_yes = "logged"
logged_class_no = "notLogged"
logged_class = ""
error_msg_register = "Username already exists"
error_msg_login_username = "Username doesn't exist"
error_msg_login_password = "Invalid password"
error_msg = ""
```

`util.py`

Infine per quanto riguarda il download del PDF ho utilizzato la funzione `send_file()` di Flask che permette di inviare un file al client.



```
@views.route("/static/report/Report.pdf", methods=["GET"])
def download():
    return send_file("static/report/Report.pdf", as_attachment=True)
```

Funzione per il download del PDF

2.2.2 Gestione view tramite Flask

Come spiegato in precedenza, ho usato la libreria Flask, in particolare grazie ai suoi Blueprint (un meccanismo per separare diverse parti dell'applicazione in moduli riutilizzabili, serve a riorganizzare views e templates in modo da avere un codice più pulito e riutilizzabile, in questo modo è possibile organizzare l'app in diverse sezioni e utilizzare comunque la stessa struttura di routing).

I blueprint vengono creati tramite la funzione Blueprint che prende come parametri il nome del blueprint e il nome del package (nel mio caso `__name__`). Per poi registrare il blueprint all'interno dell'applicazione è necessario utilizzare la funzione register_blueprint che prende come parametri il blueprint da registrare e il prefisso da utilizzare per le views del blueprint. Per quanto riguarda poi l'effettiva creazione delle views e reindirizzamento del web server a tali view, ho utilizzato la funzione route propria delle view e dei blueprint di flask che prende come parametri il percorso della view e il metodo HTTP da utilizzare per accedere alla view e restituisce la funzione render_template che prende come parametri il template da utilizzare e le variabili da passare al template.

```
● ● ●  
1 from .views import views  
2 from .auth import auth  
3 app.register_blueprint(views, url_prefix="/")  
4 app.register_blueprint(auth, url_prefix="/")
```

Registrazione dei blueprint

```
● ● ●  
1 views = Blueprint("views", __name__)
```

Creazione del blueprint per le views

```
● ● ●  
1 auth = Blueprint("auth", __name__)
```

Creazione del blueprint per l'autenticazione

```
● ● ●  
1 @views.route("/services", methods=["GET"])  
2 @login_required  
3 def services():  
4     return render_template("services.html", nav=NAV)  
5  
6  
7 @views.route("/", methods=["GET"])  
8 @views.route("/index", methods=["GET"])  
9 def index():  
10    if current_user.is_authenticated:  
11        logged_class = logged_class_yes  
12    else:  
13        logged_class = logged_class_no  
14    return render_template("index.html", nav=NAV, logged=logged_class)  
15  
16  
17 @views.route("<template>")  
18 def load_template(template):  
19    try:  
20        if not template.endswith(".html"):  
21            template += ".html"  
22        return render_template(template, nav=NAV)  
23    except TemplateNotFound:  
24        return render_template("404.html"), 404  
25    except:  
26        return render_template("500.html"), 500
```

Esempio di implementazione delle view

2.2.3 Gestione database tramite flask_sqlalchemy

Per la gestione del database (che servirà poi per registrare i dati relativi all'autenticazione degli utenti) ho utilizzato la libreria flask_sqlalchemy che permette di gestire il database tramite oggetti. L'immagine a destra mostra il file `models.py` che si occupa della creazione dell'oggetto User che andrà poi inserito all'interno del database.

Per quanto riguarda invece la sua inizializzazione, è gestita all'interno del file `__init__.py` che si occupa di creare il database (se non esiste già), di inizializzare la tabella user e di collegare il database all'app.

Sono riportati qui sotto alcuni esempi di utilizzo del database nell'applicazione: quali l'uso all'interno della funzione `load_user` del LoginManager tramite una query che restituisce l'utente con l'id passato come parametro, l'uso di un'altra query sempre sulla tabella User per verificare se l'utente è già presente nel database, e infine l'effettiva creazione di un nuovo oggetto User e il suo inserimento nel database tramite la funzione `db.session.add` e `db.session.commit`.

```
● ● ●
1 from . import db
2 from flask_login import UserMixin
3
4
5 class User(db.Model, UserMixin):
6     id = db.Column(db.Integer, primary_key=True)
7     username = db.Column(db.String(150), unique=True, nullable=False)
8     password = db.Column(db.String(150), nullable=False)
```

Modello del database

```
● ● ●
1 db = SQLAlchemy()
2 DB_NAME = "database.db"
3
4
5 def create_app():
6     app = Flask(__name__)
7     app.config["SECRET_KEY"] = "secret key"
8     app.config["SQLALCHEMY_DATABASE_URI"] = f"sqlite:///{DB_NAME}"
9
10 db.init_app(app)
11 from .models import User
12 if not path.exists("website/" + DB_NAME):
13     with app.app_context():
14         db.create_all()
```

Inizializzazione del database

```
● ● ●
1 @login_manager.user_loader
2 def load_user(id):
3     return User.query.get(int(id))
```

Query per ottenere l'utente con l'id passato come parametro

```
● ● ●
1 user = User.query.filter_by(username=username).first()
```

Query per verificare se l'utente è già presente nel database

```
● ● ●
1 new_user = User(username=username, password=generate_password_hash(password, method="sha256"))
2 db.session.add(new_user)
3 db.session.commit()
```

Creazione di un nuovo oggetto User e inserimento nel database

2.2.4 Gestione autenticazione tramite flask_login e werkzeug.security

Per quanto riguarda invece la gestione dell'autenticazione, ho utilizzato la libreria di flask `flask_login` che permette di gestire l'autenticazione degli utenti. Per prima cosa è necessario creare un oggetto `LoginManager` (che io ho creato all'interno del file `__init__.py`) che si occupa di gestire l'autenticazione degli utenti. Dopo la sua creazione è possibile impostare alcune configurazioni: quali la `login_view` che indica la view da reindirizzare in caso di utente non autenticato, la `refresh_view` che indica la view da reindirizzare in caso di utente autenticato ma con sessione scaduta, `remember_cookie_duration` che indica la durata della sessione di un utente autenticato (e quindi la durata del cookie di autenticazione), `session_protection` che indica il livello di protezione della sessione (in questo caso impostato a `strong` che indica che la sessione viene protetta da attacchi CSRF) e infine la funzione `load_user` che ho spiegato in precedenza. Inoltre è necessario fare in modo che l'oggetto `User` (che verrà poi utilizzato all'interno del database) estenda l'oggetto `UserMixin` (come si può vedere da un'immagine precedente), sempre facente parte della libreria `flask_login`, che si occupa di gestire le informazioni dell'utente.

```
● ● ●
1 login_manager = LoginManager()
2 login_manager.login_view = "auth.welcome"
3 login_manager.refresh_view = "auth.welcome"
4 login_manager.remember_cookie_duration = timedelta(minutes=30)
5 login_manager.session_protection = "strong"
6 login_manager.init_app(app)
```

Creazione del LoginManager

```
● ● ●
1 @auth.route("/welcome", methods=["GET", "POST"])
2 def welcome():
3     if request.method == "GET":
4         return render_template("welcome.html", nav=NAV)
5     elif request.method == "POST":
6         submit_type = request.form.get("submit_type")
7         if submit_type == "register":
8             username = request.form.get("username")
9             password = request.form.get("password")
10            user = User.query.filter_by(username=username).first()
11            if (user):
12                error_msg = error_msg_register
13                return render_template("welcome.html", nav=NAV, error=error_msg)
14            else:
15                new_user = User(username=username, password=generate_password_hash(password, method="sha256"))
16                db.session.add(new_user)
17                db.session.commit()
18                login_user(new_user, remember=True)
19                logged_class = logged_class_yes
20                return render_template("index.html", nav=NAV, logged=logged_class)
21        elif submit_type == "login":
22            username = request.form.get("username")
23            password = request.form.get("password")
24            user = User.query.filter_by(username=username).first()
25            if (user):
26                if check_password_hash(user.password, password):
27                    login_user(user, remember=True)
28                    logged_class = logged_class_yes
29                    return render_template("index.html", nav=NAV, logged=logged_class)
30                else:
31                    error_msg = error_msg_login_password
32                    return render_template("welcome.html", nav=NAV, error=error_msg)
33            else:
34                error_msg = error_msg_login_username
35                return render_template("welcome.html", nav=NAV, error=error_msg)
36
37
38 @auth.route("/logout", methods=["GET"])
39 @login_required
40 def logout():
41     logout_user()
42     logged_class = logged_class_no
43     return render_template("index.html", nav=NAV, logged=logged_class)
```

Funzioni per l'autenticazione e il logout

Il modulo `flask_login` permette di gestire l'autenticazione degli utenti tramite due funzioni: `login_user` che permette di autenticare un utente e `logout_user` che permette di invalidare la sessione di un utente. Inoltre permette anche di aggiungere un decorator `login_required` che permette di reindirizzare l'utente alla `login_view` impostata in precedenza in caso di utente non autenticato, e di accedere alle informazioni dell'utente tramite la variabile `current_user` che contiene l'oggetto `User` dell'utente autenticato.

Infine per quanto riguarda la gestione della password, ho usato il modulo `werkzeug.security` che permette la gestione di hashing delle password. In particolare ho utilizzato la funzione `generate_password_hash` che permette di generare un hash della password passata come parametro (nello specifico si basa sull'algoritmo PBKDF2 e utilizza la funzione di hash SHA-256) e la funzione `check_password_hash` che permette di verificare se la password passata come parametro corrisponde al hash passato come secondo parametro. In questo modo è possibile salvare nel database l'hash della password invece che la password stessa, in modo da non doverla salvare in chiaro.

2.2.5 Gestione errori (status code)

Durante l'esecuzione dell'applicazione sono tanti gli status code che possono venire restituiti dal server:

- 200 OK: indica che la richiesta è stata elaborata con successo e che il server ha restituito una risposta corretta.
- 302 Found: indica che la risorsa richiesta è temporaneamente disponibile all'indirizzo specificato in un'intestazione di risposta "Location".
- 304 Not Modified: indica che la risorsa richiesta non è stata modificata dall'ultima volta che è stata richiesta e che il client può utilizzare la copia memorizzata nella cache.
- 404 Not Found: indica che la risorsa richiesta non è stata trovata sul server.
- 500 Internal Server Error: indica che si è verificato un errore interno sul server e che la richiesta non può essere elaborata.

Gli unici status code che dovrebbero essere restituiti dal server dovrebbero essere i primi tre, che indicano un esito positivo, tuttavia è possibile che durante l'esecuzione dell'applicazione si verifichino degli errori imprevisti come ad esempio un errore di tipo 404 se si cerca di accedere ad una pagina che non esiste o un errore di tipo 500 se si verifica un errore interno al server.

Per gestire questi errori è stato implementato un sistema di gestione degli errori che permette di ottenere una pagina di errore personalizzata in base al tipo di errore che si è verificato grazie all'uso del decoratore `errorhandler` dei blueprint di Flask (per quanto riguarda le view relative all'autenticazione) e grazie alla classe `TemplateNotFound` di Jinja2 (per quanto riguarda le view generali).

```
1 @auth.errorhandler(404)
2 def page_not_found(e):
3     return render_template("404.html"), 404
4
5
6 @auth.errorhandler(500)
7 def server_not_working(e):
8     return render_template("500.html"), 500
```

Gestione tramite `errorhandler`

```
1 @views.route("<template>")
2 def load_template(template):
3     try:
4         if not template.endswith(".html"):
5             template += ".html"
6         return render_template(template, nav=NAV)
7     except TemplateNotFound:
8         return render_template("404.html"), 404
9     except:
10        return render_template("500.html"), 500
```

Gestione tramite `TemplateNotFound`

3 Esecuzione

Per eseguire il progetto è necessario seguire le istruzioni riportate nel file README.md:

```
# Clone the sources
git clone https://github.com/MarcoAntolini/Fly-Away
cd Fly-Away

# Virtual environment modules installation
'''(Unix based systems)'''
virtualenv venv
source venv/bin/activate
'''(Windows based systems)'''
virtualenv venv
.\venv\Scripts\Activate.ps1

# Install requirements
pip install -r requirements.txt

# Set the FLASK_APP environment variable
'''(Unix/Mac)''' export FLASK_APP=main.py
'''(Windows)''' set FLASK_APP=main.py
'''(PowerShell)''' $env:FLASK_APP = ".\main.py"

# Set up the DEBUG environment
'''(Unix/Mac)''' export FLASK_DEBUG=True
'''(Windows)''' set FLASK_DEBUG=True
'''(PowerShell)''' $env:FLASK_DEBUG = "True"

# Run the application
# --host=0.0.0.0 - expose the app on all network interfaces (default 127.0.0.1)
# --port=5000 - specify the app port (default 5000)
flask run --host=0.0.0.0 --port=5000

# Access the app in browser: http://127.0.0.1:5000/ or http://localhost:5000/
```

All'apertura del browser all'indirizzo `http://localhost:5000/` o `http://127.0.0.1:5000/` (o qualsiasi sia stato impostato) verrà visualizzata la pagina principale del web server:

