CSCI4230U - Movie Website

Movie Website for Final Group Project of CSCI 4230U Advanced Web Development Fall 2025.

Group Members

- Hamza Hassan (100788913)
- Abdullah Mohammed (100784442)

Overview

The platform allows users to browse movies, view showtimes, and book tickets, while administrators can manage the movie catalogue and oversee bookings.

The application is built using **Flask**, and **SQLAlchemy** for the backend, and **HTML**, **CSS**, and **JS** for the frontend. Users authenticate using secure password hashing with salts and a server-side pepper, and JWT tokens manage authorization for protected endpoints. Admin-only functionality ensures that only authenticated administrators can create, edit, and remove movies from the system.

The site integrates with the **OMDb API** (https://www.omdbapi.com/) to fetch movie details and posters dynamically, **Ollama** for an AI chatbot and a **Stripe** placeholder for future payment integration.

The goal of the project was to design a clean, maintainable, and scalable movie booking system demonstrating the backend knowledge and skills learned in the course.

Additionally, since the presentation, we have added an AI chatbot that you can ask questions about the movies being shown. Admin can search/add/remove movies to the database, which the homepage uses to show movie listings. Finally admin can remove users and modify/remove bookings.

Prequisites

Before running or developing the application, ensure the following are installed:

Required Software

- **Python 3.10+**
- **Ollama** (for chatbot)
- **pip** (Python package manager)
- **Virtual environment support** (`venv`)

- **Git**
- **Docker (for containerized deployment)**

Python Dependencies

All required Python packages are listed in `requirements.txt`.
You can install them using:

```
pip install -r requirements.txt
```

Environment Variables

The project relies on database credentials, API keys, and secret keys stored in `.env`.

A template is provided at `.env.example`.

How to Run

1. Create and activate a virtual environment

```
python -m .venv venv
source .venv/bin/activate      # macOS / Linux
.venv\Scripts\activate         # Windows
```

2. Install the dependencies

```
pip install -r requirements.txt
```

3. Create your environment file

```
cp .env.example .env
```

Fill in:

```
SECRET_KEY=...
JWT_SECRET_KEY=...
PEPPER=...
DATABASE_URL=sqlite:///database.db
STRIPE_SECRET_KEY=
TICKET_PRICE_CENTS=1300
```

4. Run Ollama in a separate terminal

This is for the chatbot to work.

```
ollama pull gemma3:1b
ollama serve
```

5. Seed the database

This creates:

- An admin user
- Initial set of movies

```
python seed.py
```

6. Start the server

```
python app.py
```

Your app is now running at: **http://localhost:5000**

7. Run with Docker instead (recommended)

Docker does all the above steps for you.

```
docker build -t movie-website .

docker run --env-file .env -p 5000:5000 -v
"$(pwd)/instance:/app/instance" movie-website
```

Your app is now running at: **http://localhost:5000**

8. Run tests

From the project directory and while in a virtual environment, run:

```
pytest -q
```

Notes when running:

- Admin username: admin
  Admin password: Admin123!
- The website is also deployed: https://www.moviewebsite.ca/

    - Unfortunately, we could not get Ollama working with the deployment site, but the chatbot does work when run locally.

Software Stack

Backend

- **Flask** (routing, server-side rendering)
- **SQLAlchemy** ORM for database modelling
- **Marshmallow** for validation schemas
- **JWT** for secure authentication
- **bcrypt** for password hashing

Frontend

- **HTML5** + Jinja templates
- **CSS**
- **JavaScript**

Database

- **SQLite**

APIs

- **OMDb API** for movie data and posters
- **Ollama** for AI chatbot
- **Stripe** for payments

Testing

- **Pytest** for unit testing
- **SeleniumBase** for e2e testing

Documentation of the API endpoints

GET /register

This endpoint just renders the register HTML page.

POST /register

Body:

```
{
  "user": "string",
  "password": "string",
  "role": "string"
}
```

The role part is optional, if you do not include it, the default will be just the ‘user’ role.

The endpoint will return either 2 things:

- **409** if the username already exists
- **201** with JSON output:
  `{"message": "User registered successfully"}`
- indicating a successful account creation

GET /login

This endpoint will render the login page and clear any JWT tokens saved (if any).

POST /login

Body:

```
{
  "username": "string",
  "password": "string"
}
```

If successful it will return:

```
{
  "message": "Successful login",
  "token": "<jwt_token>"
}
```

Otherwise it will return:

- **404**: User not found
- **401**: Invalid Password

GET /api/bookings

User will need a valid admin token to access this endpoint.

Will return JSON dictionary that looks like:

```
{
  "Bookings": [
    {
      "id": "int",
      "movie_title": "string",
      "show_date": "string",
      "showtime": "string",
      "quantity": "int",
      "booked_by": "string",
      "created_at": "string"
    }
  ]
}
```

It will return all bookings created by all users.

POST /api/bookings

This creates a booking. This endpoint requires a valid token.

Body:

```
{
  "movie_title": "string",
  "show_date": "string",
  "showtime": "string",
  "quantity": "int"
}
```

Response:

- **201** returns booking payload
- **400** for validation errors
- **500** on DB error

POST /api/bookings/checkout

User will need a token for this endpoint.

Body:
Same booking fields as `POST /api/bookings`.

Returns:

- Creates Stripe Checkout Session; returns:
  `{"checkout_url": "string"}`
- **400** validation errors
- **401** unauthenticated
- **500** if Stripe not configured/fails

PUT /api/bookings/<booking_id>

User will need a valid admin token.

Body JSON: any of date, showtime, quantity:

```
{
    "time": "string",

    "available": "int",

    "quantity": "string"

  }
```

Updates booking and preserves the original booked_by.

Returns:

- updated booking payload
- **400** validation errors
- **403** unauthorized
- **404** not found
- **500** DB error

DELETE /api/bookings/<id>

User will need a valid token.

Returns:

```
  {

    "message": "Booking cancelled successfully",

    "booking_id": "int"

  }
```

or

- **401** unauthenticated
- **403** unauthorized
- **404** not found
- **500** on DB error

GET /api/users

Will need a valid admin token.

Returns:

```
{"users": [ { "id": int, "username": "string", "role": "string" } ]}
```

or **403** if not admin.

DELETE /api/users/<user_id>

Will need valid admin token. Deletes user and their bookings.

Returns:

```
{
  "message": "User deleted successfully",
  "user_id": int,
  "deleted_bookings": int
}
```

or

- **403** unauthorized users
- **404** not found
- **500** on DB error

GET /checkout/success

If session_id provided, fetches Stripe session, saves booking from metadata, renders HTML success page.

GET /checkout/cancel

Renders HTML cancel page.

POST /webhook/stripe

Stripe webhook receiver. Expects `STRIPE_WEBHOOK_SECRET`.
On checkout, returns empty **200/400/500**.

GET /home

Renders the main home page of the website. (A valid session token is required.)

Returns:
HTML page showing a list of all movies currently stored in the database.

POST /api/chat

This endpoint powers messaging with the AI chatbot on the homepage. (A valid session token is required.)

Body JSON:

```
{
  "message": "string"
}
```

Returns:

```
{
  "reply": "string"
}
```

Errors:

- **400** — Missing JSON body
- **400** — Empty message provided

GET /movie/<movie_id>

Displays the full movie detail page. (A valid session token is required.)

- Retrieves the movie from the local database by IMDb ID
- Queries OMDB API to fetch full details
- Combines DB + OMDB data into a single object

Returns:

- HTML page `movie.html` with full movie details
- **404** if the movie does not exist

GET /booking/<movie_id>

Loads the booking page for the selected movie. (A valid session token is required.)

- Pulls movie from the database
- Displays available showtimes
- Renders the booking form

Returns:

- HTML page `booking.html` (empty booking form)
- Injects today, showtimes, and the movie data

GET /booking/edit/<booking_id>

Loads the booking edit page. (A valid session token is required.)
Only admins may edit any booking; otherwise, the user is redirected away.

Returns:

- HTML page `booking.html`
- Pre-filled booking data including date, showtime, and quantity
- Movie information

Errors:

- **404** — Booking not found
- Redirect if unauthorized

GET /my-bookings

Shows all bookings made by the logged-in user. (A valid session token is required.)

- Fetches all bookings for the current user
- Sorts them newest-first
- Builds a JSON-friendly booking payload for frontend use

Returns:

- HTML page `bookings.html`

GET /admin

Admin dashboard page. (A valid session token is required.)
Only admin users can access this page; otherwise, the user is redirected away.

Returns:
HTML dashboard showing users and bookings.

GET /admin/manage-movies

Movie management panel for admins. (A valid session token is required.)
Only admin users can access this page.

Returns:

- HTML page listing all movies in the system
- Injects the current week date for expiration scheduling

GET /admin/search-movies?q=string

Provides OMDB search results for admins when adding movies. (A valid session token is required.)
Only admin users can access this page.

q (required query): The movie search string

Returns:

```
{
  "results": [
    {
      "Title": "...",
      "Year": "...",
      "Poster": "...",
      "imdbID": "..."
    }
  ]
}
```

Errors:

- **403** — Unauthorized (not admin)

- **400** — Missing search query

POST /admin/add-movie

Adds a new movie to the database. (A valid session token is required.)
Only admin users can access this page.

Body:

```
{
  "imdb_id": "string",
  "title": "string",
  "year": "string",
  "poster": "string",
  "expiration": "YYYY-MM-DD (optional)"
}
```

Returns:

- `{"message": "Movie added"}` on success
- If the movie already exists: `{"message": "Movie already added"}`

Errors:

- **403** — Unauthorized user

POST /admin/remove-movie

Removes a movie from the database. (A valid session token is required.)
Only admin users can access this page.

Body:

- `{`
- `  "imdb_id": "string"`
- `}`

Returns:

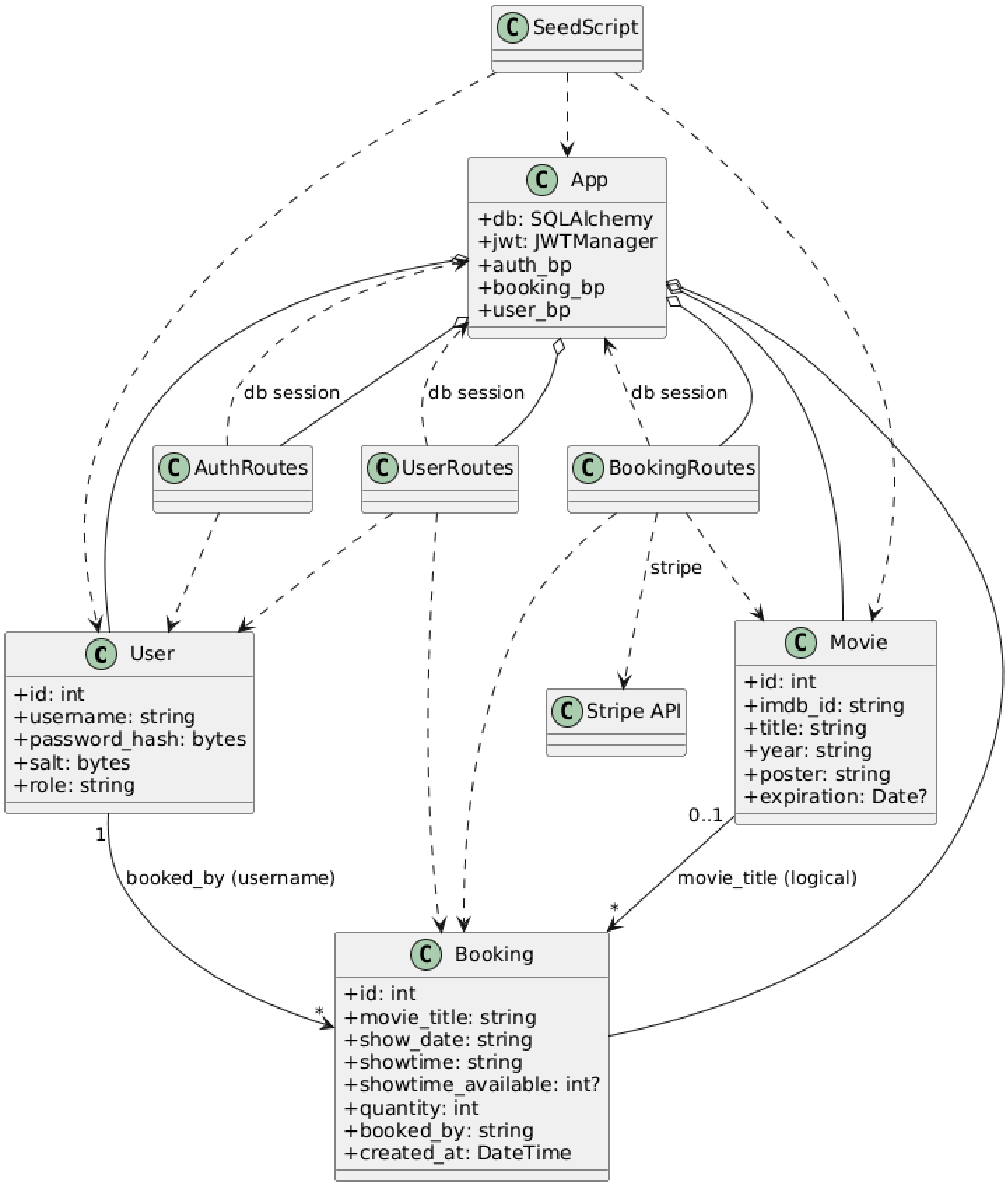
- `{"message": "Movie removed"}`

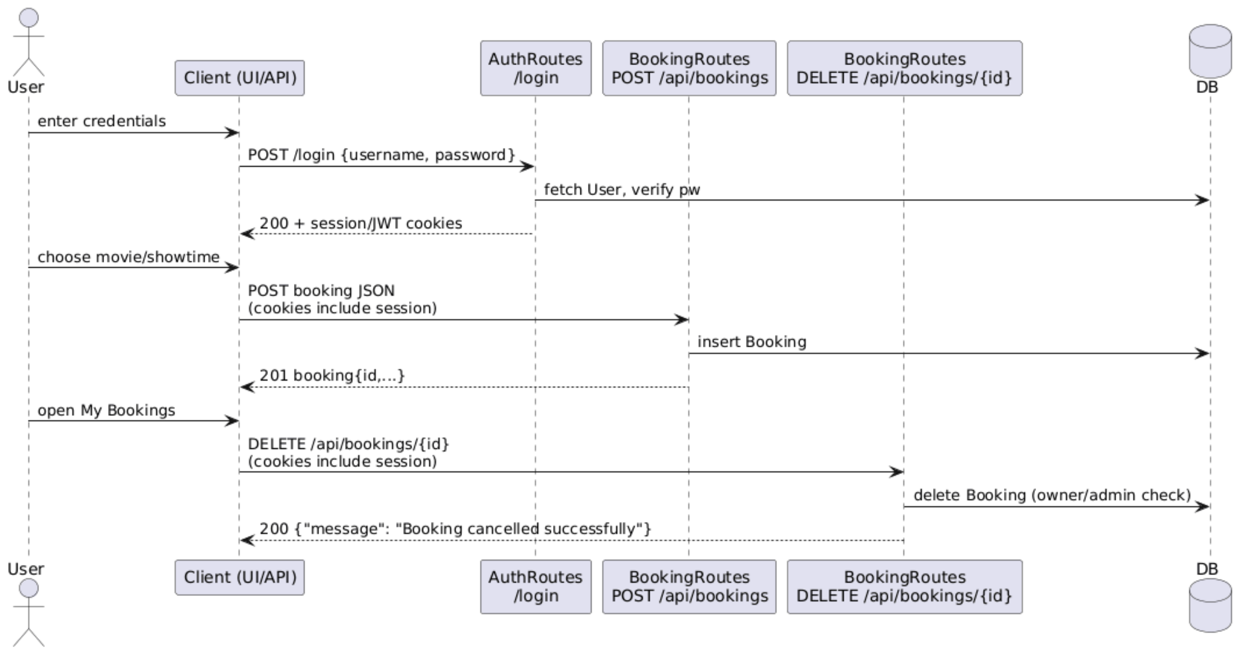
Errors:

- **403** — Unauthorized user

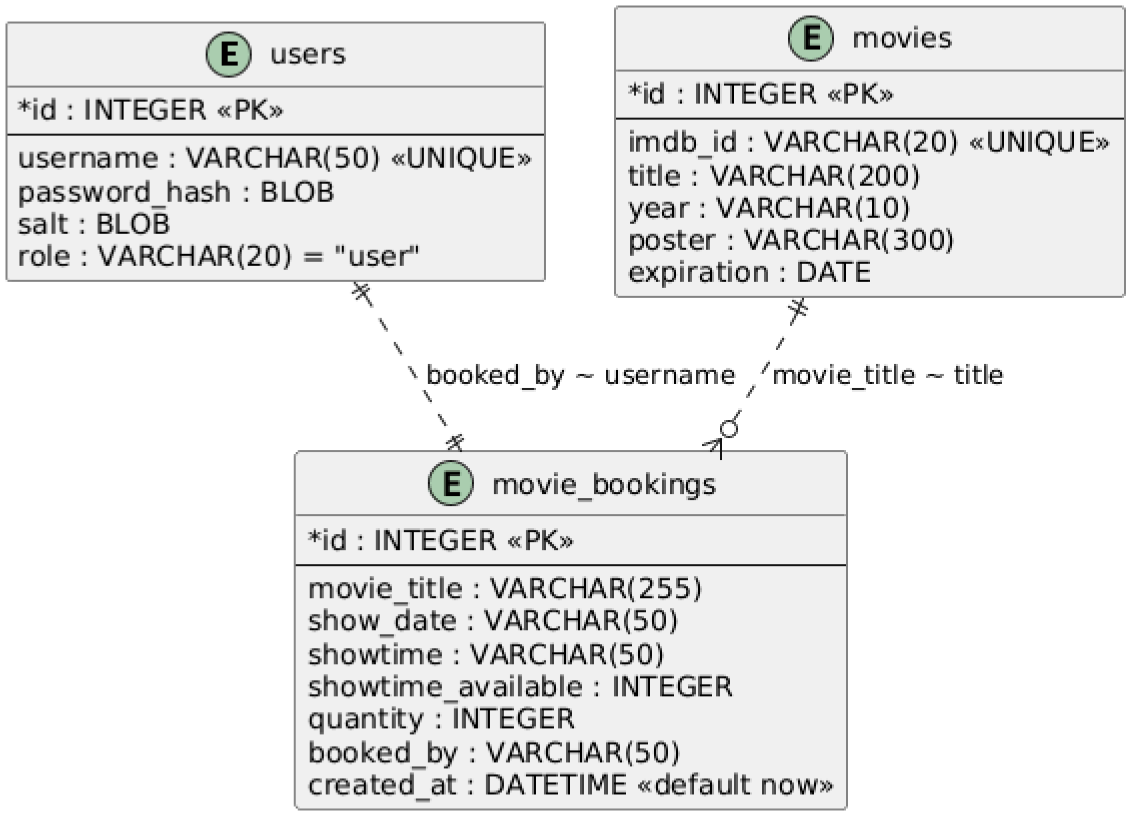
Diagram and explanation of the chosen architecture (UML package and class diagrams)

Our chosen architecture for this project was MVC. Our models is the database. Our View is the HTML with Jinja. And the controller is the flask backend.


SeedScript
App
+db: SQLAlchemy
+jwt: JWTManager
+auth_bp
+booking_bp
+user_bp
db session
db session
db session
AuthRoutes
UserRoutes
BookingRoutes
stripe
User
+id: int
+username: string
+password_hash: bytes
+salt: bytes
+role: string
Stripe API
Movie
+id: int
+imdb_id: string
+title: string
+year: string
+poster: string
+expiration: Date?
0..1
1
booked_by (username)
movie_title (logical)
*
*
Booking
+id: int
+movie_title: string
+show_date: string
+showtime: string
+showtime_available: int?
+quantity: int
+booked_by: string
+created_at: DateTime


User
Client (UI/API)
AuthRoutes /login
BookingRoutes POST /api/bookings
BookingRoutes DELETE /api/bookings/{id}
DB
enter credentials
POST /login {username, password}
fetch User, verify pw
200 + session/JWT cookies
choose movie/showtime
POST booking JSON (cookies include session)
insert Booking
201 booking{id,...}
open My Bookings
DELETE /api/bookings/{id} (cookies include session)
delete Booking (owner/admin check)
200 {"message": "Booking cancelled successfully"}

Database schema design



Deployment to a cloud platform or a general-purpose server & Documentation of the deployment process

The project includes:

1. Local Deployment via Python

The app can be deployed on any machine that has Python installed.
This method is used during development and the demonstration.

Steps:

```
python -m .venv venv
source .venv/bin/activate
pip install -r requirements.txt
python app.py
```

2. Local Deployment via Docker

Docker provides a clean and isolated environment that ensures consistent deployment regardless of the host machine.

Build the image:

```
docker build -t movie-website .
```

Run the container:

```
docker run -p 5000:5000 --env-file .env -v
"$(pwd)/instance:/app/instance" movie-website
```

This containerized deployment simulates how the application would run in a production environment, even though the project is not hosted online.

1. Deployment on Amazon EC2

These are the steps that I did to deploy our website using Amazon EC2

1. Creating an EC2 instance

Went to AWS website and created an EC2 instance. Once I did that, I got a ssh file that I downloaded

2. SSH into my EC2 instance

I used this ssh file to get into the EC2 instance in my terminal using the command:

```
ssh -i kpName.pem ubuntu@PUBLIC_IP
```

3. Install Docker + Git + Nginx

In the EC2 instance terminal I installed Docker, Git, and nginx using the command:

```
sudo apt install -y docker.io nginx git
```

4. Cloned our repo onto EC2

I then pulled our git repo onto the EC2 terminal using:

```
Git pull https://github.com/HamzaHassan02/CSCI4230U-Movie_Website.git
```

5. Built the Docker image

Then we built our docker image and ran it using these commands:

```
docker build -t movie-website .
docker run --env-file .env -p 5000:5000 -v \
"$(pwd)/instance:/app/instance" movie-website
```

6. Configure Nginx to proxy to Flask
7. Point the Domain to EC2 instance

I went to the DNS settings of the website that I bought the domain off of(squarespace domains) and pointed it’s dns to our instance of EC2

8. Install HTTPS

I used certbot to get a certificate for this website and made it secure (HTTPS)

9. Set up automatic deployment (GitHub Actions)

Added a github action script that would deploy the website automatically once you pushed to the main branch

CI/CD with GitHub Actions

The repository includes multiple GitHub Actions workflow located in:

```
.github/workflows/
```

There are a couple workflows:

Dependabot.yml

This will open Pull Requests for outdated dependencies

Codeql-analysis.yml

This will find security vulnerabilities in this app.

run_pytest.yml

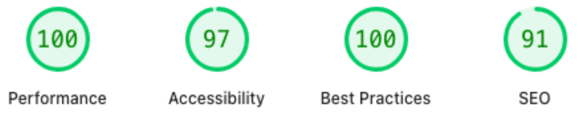
This will run just the unit tests of this app.

Seleniumbase_features.yml

This will run the end to end tests of this app.

This provides continuous integration support without deploying to production, and ensures code.

Analysis of application performance



100

Performance

Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 ■ 50–89 ● 90–100

METRICS Expand view

● First Contentful Paint 0.2 s	● Largest Contentful Paint 0.5 s
● Total Blocking Time 0 ms	● Cumulative Layout Shift 0
● Speed Index 0.6 s	

Captured at Dec 2, 2025, 2:41 PM EST
Initial page load
Emulated Desktop with Lighthouse 13.0.1
Custom throttling
Single page session
Using HeadlessChromium 137.0.7151.119 with lr

97

Accessibility

These checks highlight opportunities to improve the accessibility of your web app. Automatic detection can only detect a subset of issues and does not guarantee the accessibility of your web app, so manual testing is also encouraged.

100

Best Practices

TRUST AND SAFETY

○ Ensure CSP is effective against XSS attacks

○ Use a strong HSTS policy

○ Ensure proper origin isolation with COOP

○ Mitigate clickjacking with XFO or CSP

○ Mitigate DOM-based XSS with Trusted Types

Mobile Desktop

SEO

These checks ensure that your page is following basic search engine optimization advice. There are many additional factors Lighthouse does not score here that may affect your search ranking, including performance on Core Web Vitals. Learn more about Google Search Essentials.

CONTENT BEST PRACTICES

▲ Document does not have a meta description

Format your HTML in a way that enables crawlers to better understand your app's content.

ADDITIONAL ITEMS TO MANUALLY CHECK (1) Show

Run these additional validators on your site to check additional SEO best practices.

PASSED AUDITS (7) Show

NOT APPLICABLE (2) Show