FULL-STACK WEB APPLICATION PROJECT REPORT

Project Title: Civil Aviation

DECEMBER 3, 2024

888 – PEAKY BLINDER
Xuan Tuan Minh Nguyen – Ba Viet Anh (Henry) Nguyen – Trong Dat Hoang
Tutor: Ningran Li

Introduction

This project report outlines the system architecture and implementation of a the AviAI website application that provides real-time predictions using a machine learning model, especially Random Forest. The report demonstrates the integration of AviAI's user-friendly front-end. In addition, by using FastAPI back-end, the website performs data transfer and user can easily access prediction results and they can send their inputs through a simple form and receive predictions swiftly. The report also outlines the connection between the front-end and back-end creates a great experience, demonstrating that data can be transfer smoothly, ensuring efficiency and clarity in every interaction.

In addition, this report highlights the AI model integration, deployment considerations, and future improvements to enhance functionality and user experience.

Finally, this report points out how the solution addresses a real-world issue by providing user-friendly input and visual responses.

System Architecture

The architecture of the system is designed to efficiently connect the front-end, back-end, and AI model components, ensuring a smooth user experience and efficient data processing. In this Assignment, the system architecture that our group decided to stick on with is:

Client Layer (Frontend)

- **NextJS SPA Architecture**
  - Implements **server-side rendering** (SSR) for improved performance and **SEO**.
  - Utilizes **next-safe-action** for secure server actions.
  - Takes advantage of Next.js 14 features like **App Router** and **Server Components.**
  - **Zod** for effortlessly validating the user input.
- **UI/UX Implementation**
  - Combines **shadcn/ui**, **aceternity-ui**, and **magic-ui** for cohesive component design.
  - Implements responsive design using **TailwindCSS**.
  - Uses **framer-motion** through UI libraries for smooth animations.

Server Layer (Backend)

- **API Architecture**

  - **FastAPI** implementation with **RESTful** endpoints
  - **JWT** authentication middleware for secure route protection
  - Integration with **SerpAPI** for real-time flight data fetching
- **Database Integration**
  - **Prisma ORM** for type-safe database operations
  - Connection pooling for efficient **PostgreSQL** database management

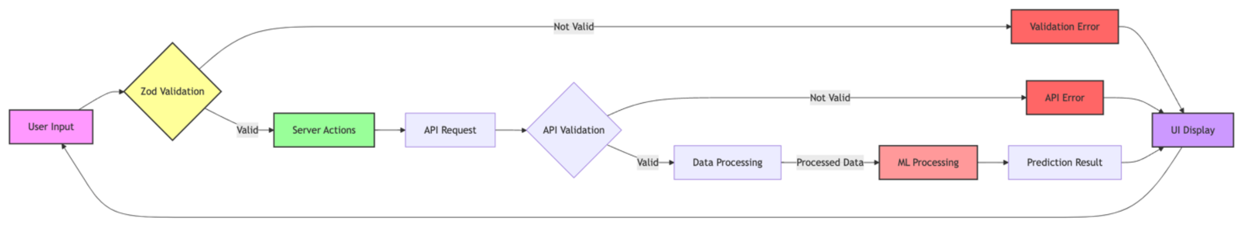
AI/ML Integration

- **Data Processing Pipeline**
  - Using **pandas** for data cleaning and preprocessing.
- **Machine Learning Implementation**
  - Using previously trained model and data pipeline to enhance the prediction quality and better visualization.

Integration & DevOps

- **Version Control & Deployment**
  - GitHub-based development workflow
  - Branch protection rules and code review processes
  - CI/CD pipeline implementation
- **Back-end and Front-end communication**
  - RESTful API contracts between frontend and backend
  - Type safety across the stack using TypeScript and Python type hints.

This architecture ensures that the codebase is scalable and maintainable, safe data handling and authentication, effortless real-time data processing, smooth user experience with modern user interfaces and reliable machine learning model deployment.



Front-End Implementation

Key Features: User Input and Validation

One of the front-end key features is User Input Form, which is a simple and intuitive form where users can input the required data to achieve expected result.

In addition, the User Input data Validation is implemented, which reacts with User Input Form to ensure that ensures that users provide all necessary inputs in the correct format, reducing errors and improving data quality.

Technically, when the user fills in the input, the **zod library,** which is well-known as a modern data validation library, is also used for making sure that all the user fields are corrected to the defined requirements.

For example, fields in webpages are required to follow specific formats (e.g., dates, or destination) that are defined in a schema. If the user fails to follow the rule, then the error will be displayed on the screen.

Key Features: User Input data interact with back-end.

The front-end uses HTTP requests to send the user input data to the back end.

The front-end collects user input, such as departure city, destination city, and booking date, through a simple form with validation. This data is sent to the back end via a POST request in JSON format. The FastAPI server processes the data, feeds it into the AI model, and returns predictions. The results are then displayed on the front-end through dynamic visualizations, providing users with quick and clear insights.

Key Features: User obtained prediction.

After the back end processes the input with the AI model, it sends back the prediction results. Once predictions are retrieved, the results are displayed using clear and interactive visualizations line chart, bar chart to visualize Price trend, Price distribution and Seasonality Analysis.

Back-End Implementation

FastAPI server setup

This project employed the modern and efficient FastAPI. The server is designed to handle incoming requests from the front-end, process user input data, run the AI model, and. then return the predictions to user (Lautaro 2024).

In this project, our group has decided to implement a clean architecure pattern for our back-end section to create a robust and easy-to-scale back-end code. We defined all endpoints in the routers folder, used for exposing API endpoints. For all the controllers we use for processing data and retrieving results, we have left them in the controllers' file. In addition, the input and output model will be in model's folder and all the side functions, such as logging, exceptions, middleware's, authentication, machine learning and data processing will be placed in utils folder. This structure ensures the scalability and the robustness of the API when it comes to further development.

API endpoint documentation

This section outlines the implementation of various HTTP request methods in our flight booking system's API architecture.

Authentication

To access protected routes, a JWT token must be obtained via the Sign Up or Sign In routes. Which is specify as:

- **POST /api/v1/auth/signUp**: Used for creating a user and saved it to the PostgreSQL database. Once the mutation process is completed, the user information with JWT access and refresh token is provided. If the information is not correctly inputted, then a 422-error message will be thrown.
- **POST /api/v1/auth/signIn**: Used for looking up the user information in the PostgreSQL database. If the user is not found on the database, a 409-conflict error will be thrown, and 422 error will also be thrown if the input body does not match. Once the process is finished, then user information with JWT access and refresh token will be provided.

Once obtained the JWT access token, simply include the token in the `Authorization` header as `Bearer <token>` and save the refresh token for obtain a new access token if expired.

Flight Prices

These endpoints specify for providing real-time flight prices data obtained from SerpAPI and Google Flights. These are all protected routes, thus a JWT token must be obtained via the Sign Up or Sign In routes. The flight prices routes contain:

- **GET /api/v1/flight-prices/destinations**: Retrieve a full list of available airport destinations.
- **GET /api/v1/flight-prices/destinations/search?q=region_name**: Search for airport destinations based on the query q. For example: search?q=Frank will return Frankfurt Airport. If there are no destinations, then it returns an empty list.
- **POST /api/v1/flight-prices**: Retrive flight prices based on the given requestbody.
    - Request Body:

- **trip_type**: Type of the trip, which only allows either "oneway" or "round".
- **departure_id**: The IATA code of the departure airport, which could be obtained via GET /api/v1/flight-prices/destinations/search?q=region_name.
- **arrival_id**: The IATA code of the arrival airport, which could be obtained via GET /api/v1/flight-prices/destinations/search?q=region_name.
- **outbound_date**: The departure date in format of YYYY-MM-DD
- **currency**: The currency, which only allows EUR, AUD and USD
- **adults**: Number of adults in string, default is "1".
- **children**: Number of children in string, default is "0".
- **infants_on_lap**: Number of infants on lap in string, default is "0".

- **GET /api/v1/flight-prices/airlines**: Retrieve a list of all airlines.

Flight Prediction

This endpoint is used for providing real-time flight prices prediction using pre-trained models. These are all protected routes, thus a JWT token must be obtained via the Sign Up or Sign In routes. The flight prediction routes contain:

- **POST /api/v1/prediction**: Generate flight price predictions.
  - Request Body:
    - **departure_date**: The departure date in format of YYYY-MM-DD
    - **departure_time**: The departure time in format of HH:MM: SS
    - **arrival_date**: The arrival date in format of YYYY-MM-DD
    - **arrival_time**: The arrival time in format of HH:MM: SS
    - **airline**: The list of airlines, can only be used from **GET /api/v1/flight-prices/airlines**
    - **departure_city**: The IATA code + name of the departure airport, which could be obtained via GET /api/v1/flight-prices/destinations/search?q=region_name.
    - **arrival_city**: The IATA code + name of the arrival airport, which could be obtained via GET /api/v1/flight-prices/destinations/search?q=region_name.
    - **stops:** The number of stops, which are limited to "direct", "1" and "2".

Chart Data

This endpoint is used for providing real-time flight prices prediction using pre-trained models. These are all protected routes, thus a JWT token must be obtained via the Sign Up or Sign In routes. The flight prices routes contain:

- **GET /api/v1/prediction/charts/data**: Fetch data from the dataset used for training the model for chart visualizations.

User Management

These endpoints are used for providing queries and mutations towards user information. These are all protected routes, thus a JWT token must be obtained via the Sign Up or Sign In routes. The flight prices routes contain:

- **GET /api/v1/user**: Retrieve a list of all users.
- **POST /api/v1/user**: Create a new user.
- **GET /api/v1/user/{item_id}**: Retrieve a user by their ID.
- **PUT /api/v1/user/{item_id}**: Update an existing user by their ID.
- **DELETE /api/v1/user/{item_id}**: Delete a user by their ID.

Please refer to localhost:8000/docs to check the endpoint.

AI Model Integration

Recap the previous AI model

In Assignment 2, we evaluated three machine learning models for flight price prediction: Random Forest Regressor, Gradient Boosting Regressor, and Random Forest Classifier. After comprehensive testing including 5-fold cross-validation and hypertunning parameter, the Random Forest Regressor emerged as the optimal choice. This selection was supported by its superior performance in handling non-linear relationships within flight pricing data and its ability to process complex feature interactions (Breiman, 2001; Li & Li, 2019). The model demonstrated robust predictive capabilities, explaining 82.68% of the variance in flight prices on the test set, while achieving a training score of 0.9139.

The Random Forest Regressor's effectiveness in flight price prediction stems from its ensemble learning approach, which combines multiple decision trees to enhance accuracy while mitigating overfitting risks (Ho, 1995). This characteristic is particularly valuable in the aviation industry, where pricing is influenced by numerous dynamic factors such as seasonality, demand fluctuations, and route popularity (Belobaba et al., 2015; Piga & Bachis, 2007).

Random Forest Regressor model is integrated into the web application

The integration of the AI model into the web application involved several key steps and modifications to ensure seamless deployment and operation:

1. Model Serialization:
    a. The trained Random Forest Regressor model was serialized using pickle, enabling efficient storage and quick loading during web operations.
    b. This approach significantly reduces deployment complexity while maintaining model integrity.
2. Data Processing Pipeline:

   a. A comprehensive data processing system was implemented to handle incoming user input from the front end.
   b. Alongside the model, key preprocessing components were also serialized using pickle:
      i. The feature scaler (StandardScaler) used during training
      ii. The categorical encoder used for processing categorical variables
      iii. The feature selection configuration that determines the final set of predictive features
   c. This preservation of preprocessing components ensures consistency between training and deployment environments, maintaining the model's predictive accuracy.
   d. The system includes cleaning routines and data transformation processes that mirror the training pipeline:
      i. Numerical features are scaled using the preserved scaler
      ii. Categorical variables (such as departure cities and airlines) are encoded using the saved encoder
      iii. Features are selected based on the preserved feature selection configuration
   e. This synchronized approach guarantees that new data undergoes identical transformations to those applied during model training.
3. Server-Side Integration:
   a. The model was integrated into the server architecture, where it receives processed input data.
   b. After prediction processing, the server returns the results to the frontend for user display.
   c. This architecture ensures efficient handling of multiple concurrent requests while maintaining prediction accuracy.

The integration maintains the model's predictive power while adapting it for real-time web-based predictions, creating a practical tool for flight price estimation. By preserving and utilizing the exact preprocessing components from the training phase, we ensure consistency and reliability in the prediction pipeline.

Enhancements

Our website has several enhancements to improve quality and the user experience:

+ All the buttons and icons are responsive and clickable to enhance user experience. Each button has its own meaning to really improve user experience and also performance of the website.

+ If the customer does not sign in or sign up to the website, they will not be eligible to use any functions like get or predict flight price then they will be redirected to the page error page that require to sign in or sign up to continue using the website. The reason we do that is that we want to increase the sign-up rate of the website, which will help our website gain more users in the future.

+ Enhanced Visualizations: implement 3 types of charts by using Plotly.js with interactive features and react-plotly-js for easy integration with React, which are Price Distribution Chart (bar chart), Price Trend Analysis Chart (line chart) and Seasonal Analysis chart (combination of area chart and line chart). Refer to Appendix 1.

+ Auto Suggestions Input: Provided advanced text field input with comprehensive validation and dynamic user feedback. All our customer input is automatically completed, and we recommend input for customers. For example, when a customer types 'D' in 'From' field, our system will recommend 'Dortmund', 'Dresden' or 'Dusseldorf'. Refer to Appendix 2 for more information.

+ UX Enhancements: With the powerful combination of Server-Side Rendering (SSR), Increment Site Generation (ISR), Static Site Generation (SSG), Client-Side Rendering (CSR) and the help of caching strategies that NextJS has provided. Our website has a blazingly fast First Contentful Paint (FCP) score that is less than 500ms, thus enhancing the overall User Experience. In addition, the Loading Skeleton component, the effortless validation process using Zod, has also played a vital role in optimizing the webpage performance for better user experience. Refer to Appendix 3.

+ Error handling: With the use of Zod validation library for front-end and Server Actions and pydantic for back-end, it is easier to provide error handling in both client and server-side validation. Thus, enhancing the development experience and seamless user experience. Refer to Appendix 4 and 5 for more information.

+ Server Action: In this project, we have also adopted server actions from NextJS, which allows us to write server-side code with fully type-safe. Although we have implemented a separate back-end service, Server Actions help us to easily working with forms, validations and error handling and tasks that need authentication, thus reducing the amount of work to working with fetching data and allow effortless coding experience. Refer to Appendix 5 and 6.

+ JSON Web Token: To not let user illegally accessing our services, which are flight prices and flight price predictions, we have implemented a simple Json Web Token (JWT) strategy, which is "a security guy" that protects the data from being illegally taken. This will check if the user has a legal access token, which could be obtained by logging in with your username and password. If the token is legal then allow the user to access our services, otherwise it will not allow that user to access. This implementation ensures the integrity of the API and protects data from being stolen from hackers. Refer to Appendix 7 and 8.

+ Rate Limiter: The rate limiter is a middleware that only allows users to access database for several attempts at several times, once the user exceeds the limit then the server will deny all requests from the user. This feature creates an indispensable protection for back-end to not getting DDOS attack. Refer to Appendix 9 and 10

Please refer to the appendix for the screenshot of code and evidence of enhancements.

Conclusion

The web application effectively resolved the challenge of predicting and categorizing flight prices in the German air travel market, offering users insights for better travel planning. By implementing machine learning models, the application provided price predictions and categorized flights into budget ranges. Additionally, AviAI web application implements clear visualizations then crucial factors affecting prices, making it easier for users to understand what affects flight costs and helping them make booking choices.

Bibliography

[1] Lautaro,S., 2024, 'FastAPI: SOLID Principles and Design Patterns', *Medium*, 15 September. Available at: https://medium.com/@lautisuarez081/fastapi-best-practices-and-design-patterns-building-quality-python-apis-31774ff3c28a

[2] Breiman, L. ,2001, ‘Random Forests’, *Machine Learning*, 45(1), 5–32.

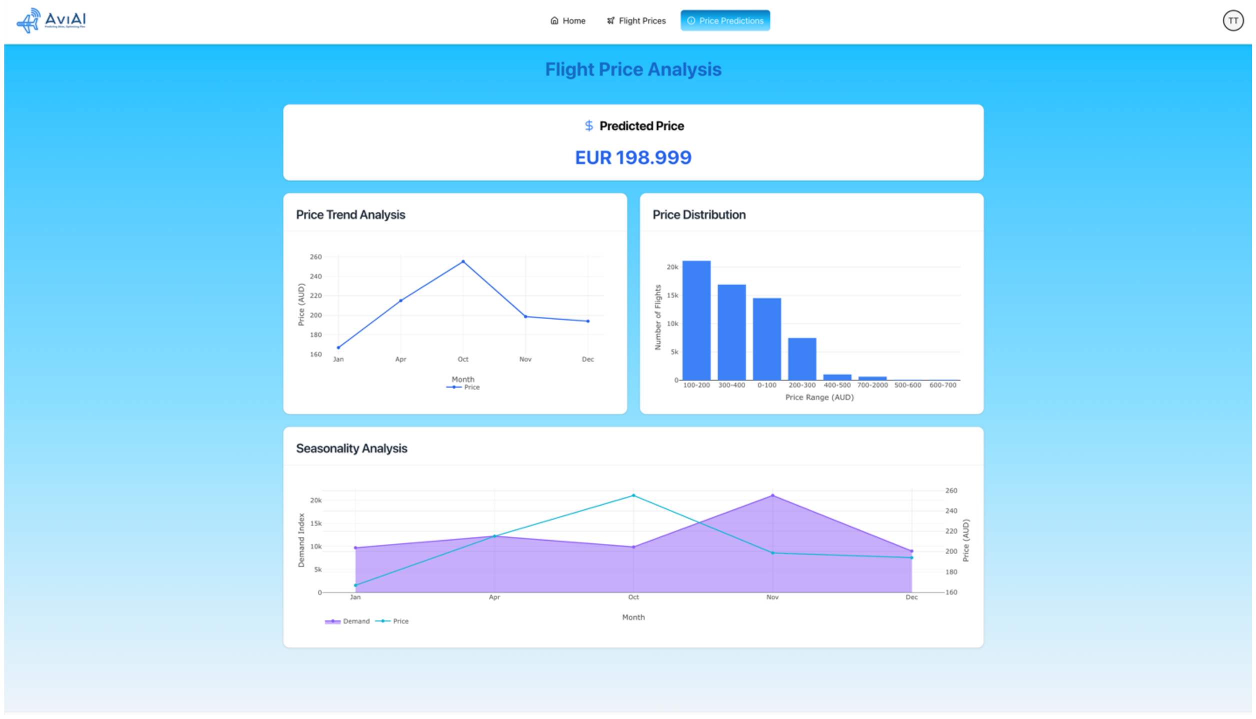
[3] Li, S., & Li, X. ,2019,’Flight Ticket Price Prediction and Analysis Based on Machine Learning’, In *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA),* pp. 76–80, IEEE.

[4] Ho, T. K., 1995, ‘Random Decision Forests’, In *Proceedings of the Third International Conference on Document Analysis and Recognition,* Vol. 1, pp. 278–282, IEEE.

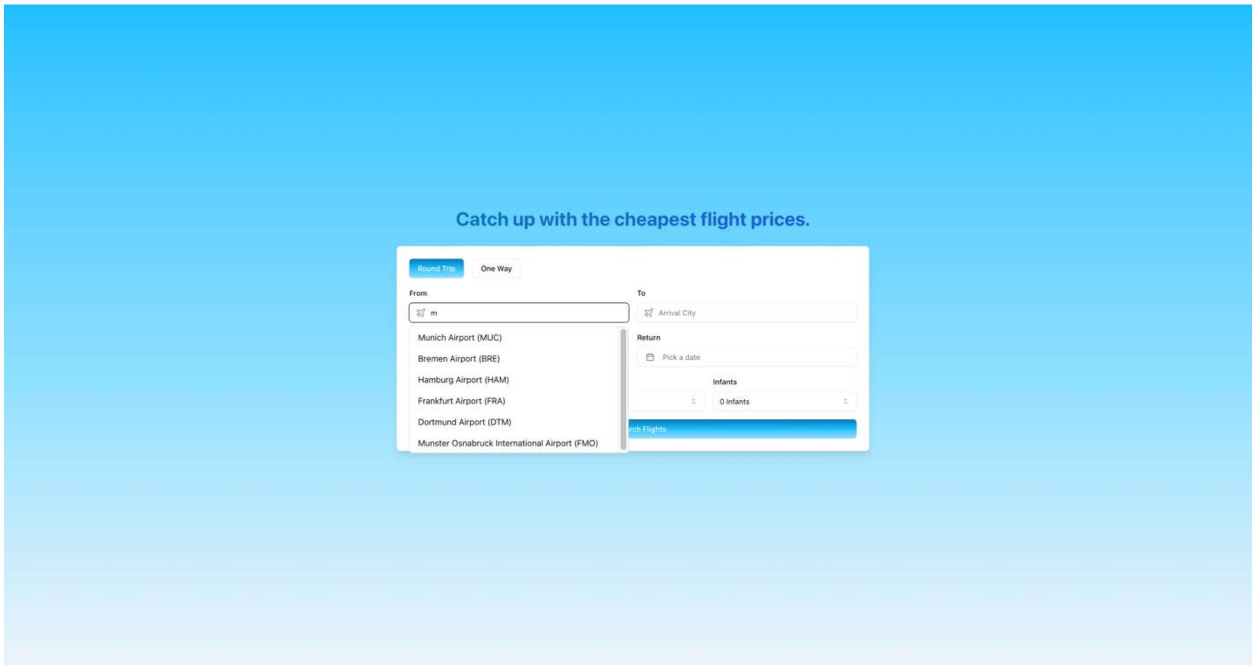
[5] Belobaba, P., Odoni, A., & Barnhart, C., Eds 2015, ‘*The Global Airline Industry* (2nd ed.)’, John Wiley & Sons.

[6] Piga, C. A., & Bachis, E., 2007, ‘Pricing strategies by European traditional and low-cost airlines: Or, when is it the best time to book online?’, In *Advances in Airline Economics,* Vol. 2, pp. 319–344, Emerald Group Publishing Limited.

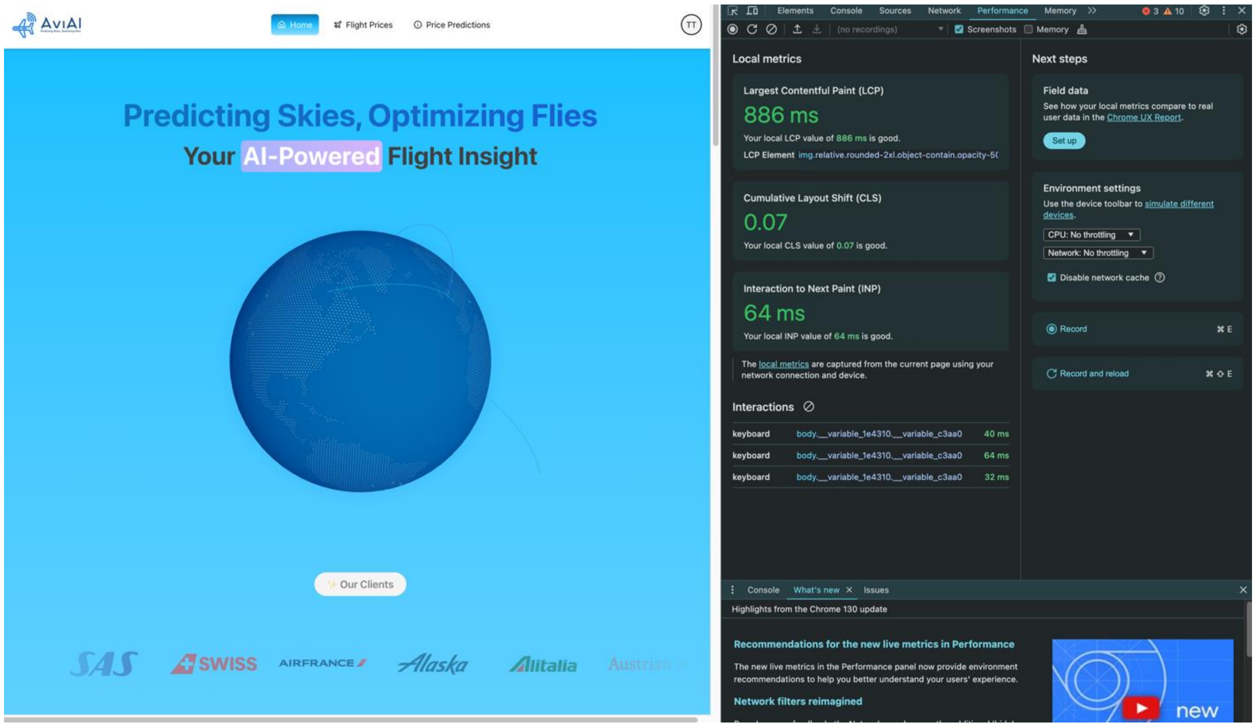
Appendix



Appendix 1: Three types of charts



Appendix 2: Auto Suggestion Feature



Local metrics

Largest Contentful Paint (LCP)
886 ms
Your local LCP value of 886 ms is good.
LCP Element img.relative.rounded-2xl.object-contain.opacity-5(

Cumulative Layout Shift (CLS)
0.07
Your local CLS value of 0.07 is good.

Interaction to Next Paint (INP)
64 ms
Your local INP value of 64 ms is good.

The local metrics are captured from the current page using your network connection and device.

Interactions

keyboard	body.__variable_1e4310.__variable_c3aa0	40 ms
keyboard	body.__variable_1e4310.__variable_c3aa0	64 ms
keyboard	body.__variable_1e4310.__variable_c3aa0	32 ms

Next steps

Field data
See how your local metrics compare to real user data in the Chrome UX Report.

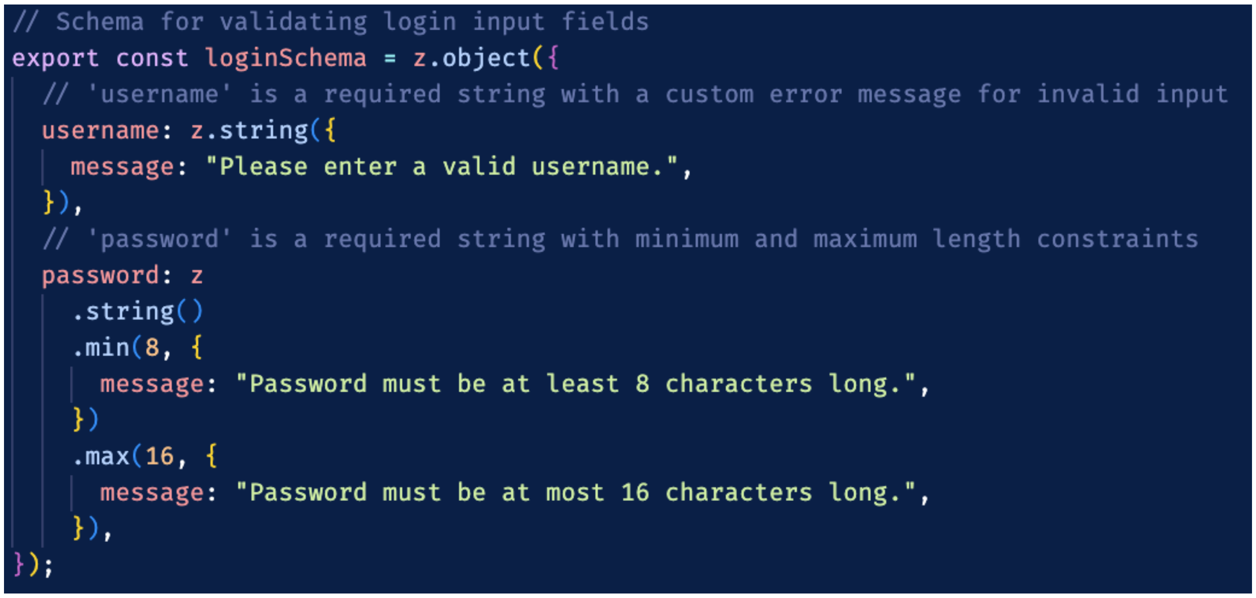
Environment settings
Use the device toolbar to simulate different devices.
CPU: No throttling
Network: No throttling
Disable network cache

Highlights from the Chrome 130 update

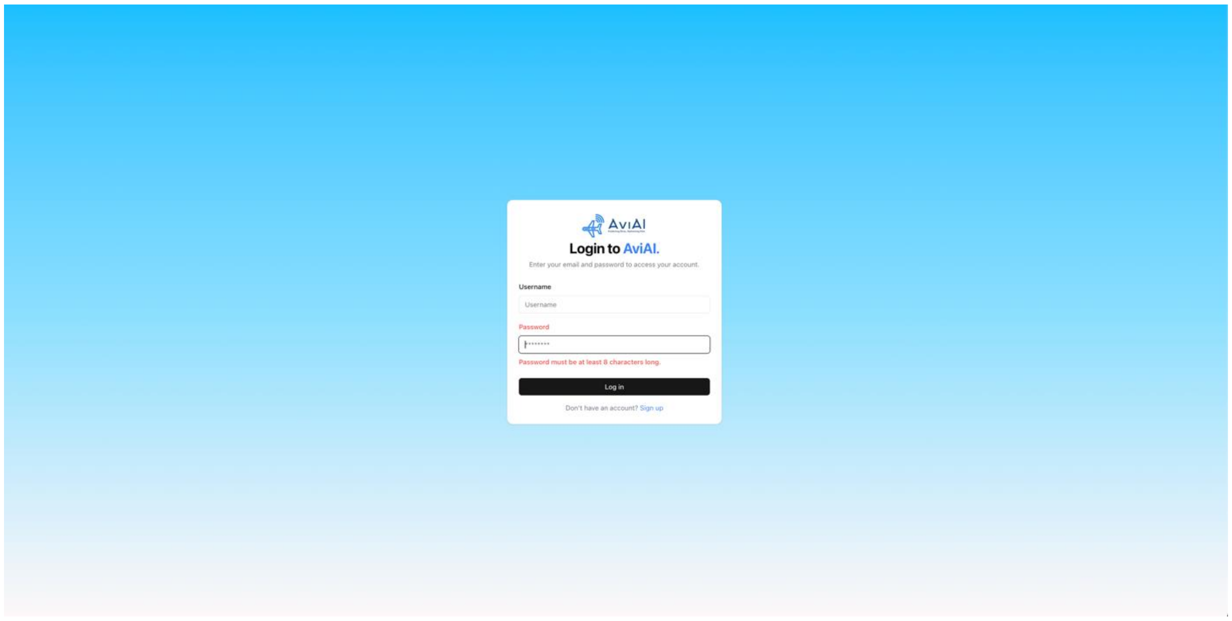
Recommendations for the new live metrics in Performance
The new live metrics in the Performance panel now provide environment recommendations to help you better understand your users' experience.

Network filters reimagined

Appendix 3: Largest Contentful Paint (LCP) in seconds


```
// Schema for validating login input fields
export const loginSchema = z.object({
  // 'username' is a required string with a custom error message for invalid input
  username: z.string({
    message: "Please enter a valid username.",
  }),
  // 'password' is a required string with minimum and maximum length constraints
  password: z
    .string()
    .min(8, {
      message: "Password must be at least 8 characters long.",
    })
    .max(16, {
      message: "Password must be at most 16 characters long.",
    }),
});
```

Appendix 4: Zod schema for validating input fields



Login to AviAI.

Enter your email and password to access your account.

Username

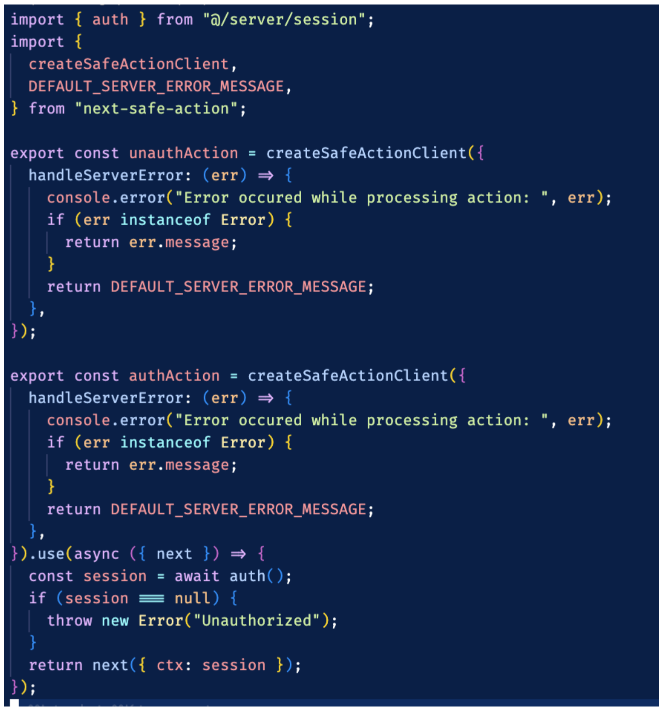
Username

Password

Password must be at least 8 characters long.

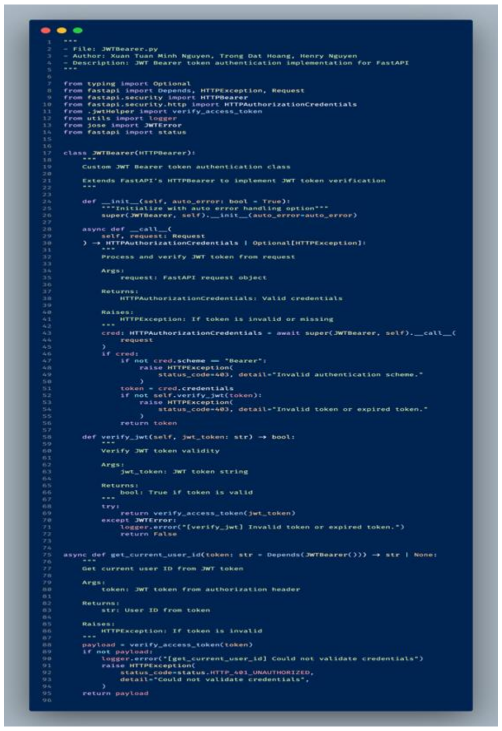
Don't have an account? Sign up

Appendix 5: Zod validators in action



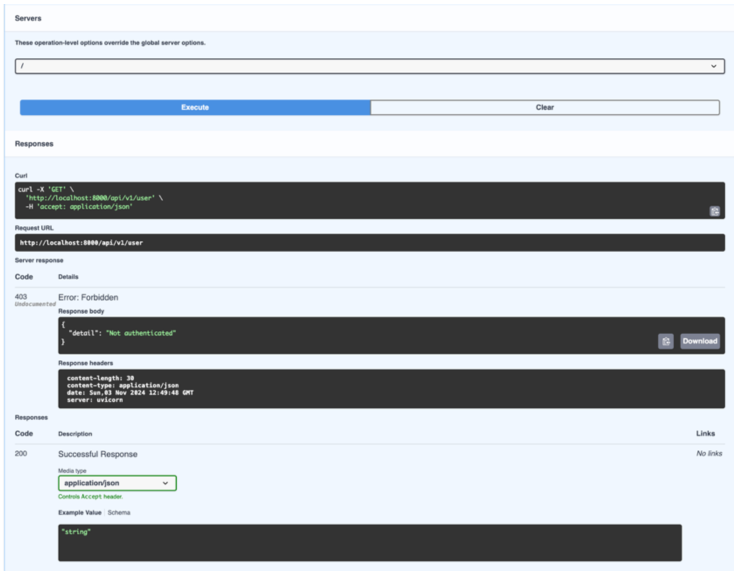
```
import { auth } from "@/server/session";
import {
  createSafeActionClient,
  DEFAULT_SERVER_ERROR_MESSAGE,
} from "next-safe-action";

export const unauthAction = createSafeActionClient({
  handleServerError: (err) => {
    console.error("Error occured while processing action: ", err);
    if (err instanceof Error) {
      return err.message;
    }
    return DEFAULT_SERVER_ERROR_MESSAGE;
  },
});

export const authAction = createSafeActionClient({
  handleServerError: (err) => {
    console.error("Error occured while processing action: ", err);
    if (err instanceof Error) {
      return err.message;
    }
    return DEFAULT_SERVER_ERROR_MESSAGE;
  },
}).use(async ({ next }) => {
  const session = await auth();
  if (session === null) {
    throw new Error("Unauthorized");
  }
  return next({ ctx: session });
});
```

Appendix 6: Applying next-safe-action to safely interact with Server Actions



```python
"""
- File: JWTBearer.py
- Author: Xuan Tuan Minh Nguyen, Trong Dat Hoang, Henry Nguyen
- Description: JWT Bearer token authentication implementation for FastAPI
"""

from typing import Optional
from fastapi import Depends, HTTPException, Request
from fastapi.security import HTTPBearer
from fastapi.security.http import HTTPAuthorizationCredentials
from .jwtHelper import verify_access_token
from utils import logger
from jose import JWTError
from fastapi import status


class JWTBearer(HTTPBearer):
    """
    Custom JWT Bearer token authentication class

    Extends FastAPI's HTTPBearer to implement JWT token verification
    """

    def __init__(self, auto_error: bool = True):
        """Initialize with auto error handling option"""
        super(JWTBearer, self).__init__(auto_error=auto_error)

    async def __call__(
        self, request: Request
    ) → HTTPAuthorizationCredentials | Optional[HTTPException]:
        """
        Process and verify JWT token from request

        Args:
            request: FastAPI request object

        Returns:
            HTTPAuthorizationCredentials: Valid credentials

        Raises:
            HTTPException: If token is invalid or missing
        """
        cred: HTTPAuthorizationCredentials = await super(JWTBearer, self).__call__(
            request
        )
        if cred:
            if not cred.scheme == "Bearer":
                raise HTTPException(
                    status_code=403, detail="Invalid authentication scheme."
                )
            token = cred.credentials
            if not self.verify_jwt(token):
                raise HTTPException(
                    status_code=403, detail="Invalid token or expired token."
                )
            return token

    def verify_jwt(self, jwt_token: str) → bool:
        """
        Verify JWT token validity

        Args:
            jwt_token: JWT token string

        Returns:
            bool: True if token is valid
        """
        try:
            return verify_access_token(jwt_token)
        except JWTError:
            logger.error("[verify_jwt] Invalid token or expired token.")
            return False


async def get_current_user_id(token: str = Depends(JWTBearer())) → str | None:
    """
    Get current user ID from JWT token

    Args:
        token: JWT token from authorization header

    Returns:
        str: User ID from token

    Raises:
        HTTPException: If token is invalid
    """
    payload = verify_access_token(token)
    if not payload:
        logger.error("[get_current_user_id] Could not validate credentials")
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Could not validate credentials",
        )
    return payload

```

Appendix 7: Code used for check JWT token in server



Servers

These operation-level options override the global server options.

/

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:8000/api/v1/user' \
  -H 'accept: application/json'
```

Request URL

```
http://localhost:8000/api/v1/user
```

Server response

Code	Details
403 *Undocumented*	Error: Forbidden

Response body

```
{
  "detail": "Not authenticated"
}
```

Response headers

```
content-length: 30
content-type: application/json
date: Sun,03 Nov 2024 12:49:48 GMT
server: uvicorn
```

Responses

Code	Description	Links
200	Successful Response	*No links*

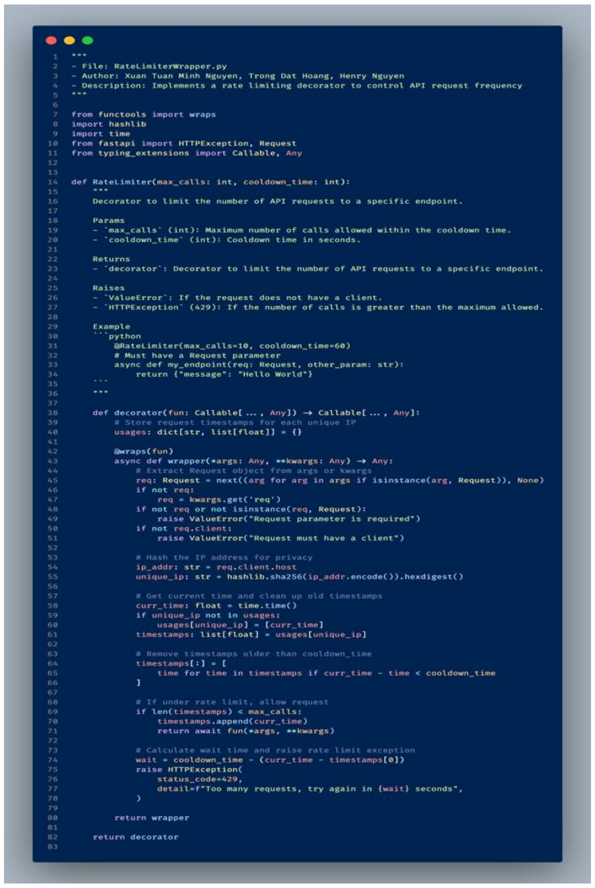
Media type: application/json

Controls Accept header.

Example Value | Schema

```
"string"
```

Appendix 8: Not authorized response if no JWT token is provided



~~~python
"""
- File: RateLimiterWrapper.py
- Author: Xuan Tuan Minh Nguyen, Trong Dat Hoang, Henry Nguyen
- Description: Implements a rate limiting decorator to control API request frequency
"""

from functools import wraps
import hashlib
import time
from fastapi import HTTPException, Request
from typing_extensions import Callable, Any


def RateLimiter(max_calls: int, cooldown_time: int):
    """
    Decorator to limit the number of API requests to a specific endpoint.

    Params
    - `max_calls` (int): Maximum number of calls allowed within the cooldown time.
    - `cooldown_time` (int): Cooldown time in seconds.

    Returns
    - `decorator`: Decorator to limit the number of API requests to a specific endpoint.

    Raises
    - `ValueError`: If the request does not have a client.
    - `HTTPException` (429): If the number of calls is greater than the maximum allowed.

    Example
    ```python
      @RateLimiter(max_calls=10, cooldown_time=60)
      # Must have a Request parameter
      async def my_endpoint(req: Request, other_param: str):
          return {"message": "Hello World"}
    ~~~
    """

    def decorator(fun: Callable[... , Any]) → Callable[... , Any]:
        # Store request timestamps for each unique IP
        usages: dict[str, list[float]] = {}

        @wraps(fun)
        async def wrapper(*args: Any, **kwargs: Any) → Any:
            # Extract Request object from args or kwargs
            req: Request = next((arg for arg in args if isinstance(arg, Request)), None)
            if not req:
                req = kwargs.get('req')
            if not req or not isinstance(req, Request):
                raise ValueError("Request parameter is required")
            if not req.client:
                raise ValueError("Request must have a client")

            # Hash the IP address for privacy
            ip_addr: str = req.client.host
            unique_ip: str = hashlib.sha256(ip_addr.encode()).hexdigest()

            # Get current time and clean up old timestamps
            curr_time: float = time.time()
            if unique_ip not in usages:
                usages[unique_ip] = [curr_time]
            timestamps: list[float] = usages[unique_ip]

            # Remove timestamps older than cooldown_time
            timestamps[:] = [
                time for time in timestamps if curr_time - time < cooldown_time
            ]

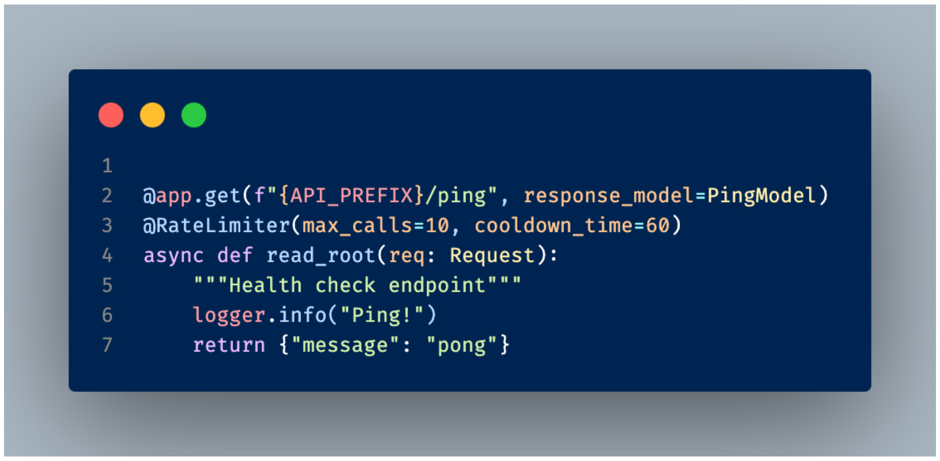
            # If under rate limit, allow request
            if len(timestamps) < max_calls:
                timestamps.append(curr_time)
                return await fun(*args, **kwargs)

            # Calculate wait time and raise rate limit exception
            wait = cooldown_time - (curr_time - timestamps[0])
            raise HTTPException(
                status_code=429,
                detail=f"Too many requests, try again in {wait} seconds",
)

        return wrapper

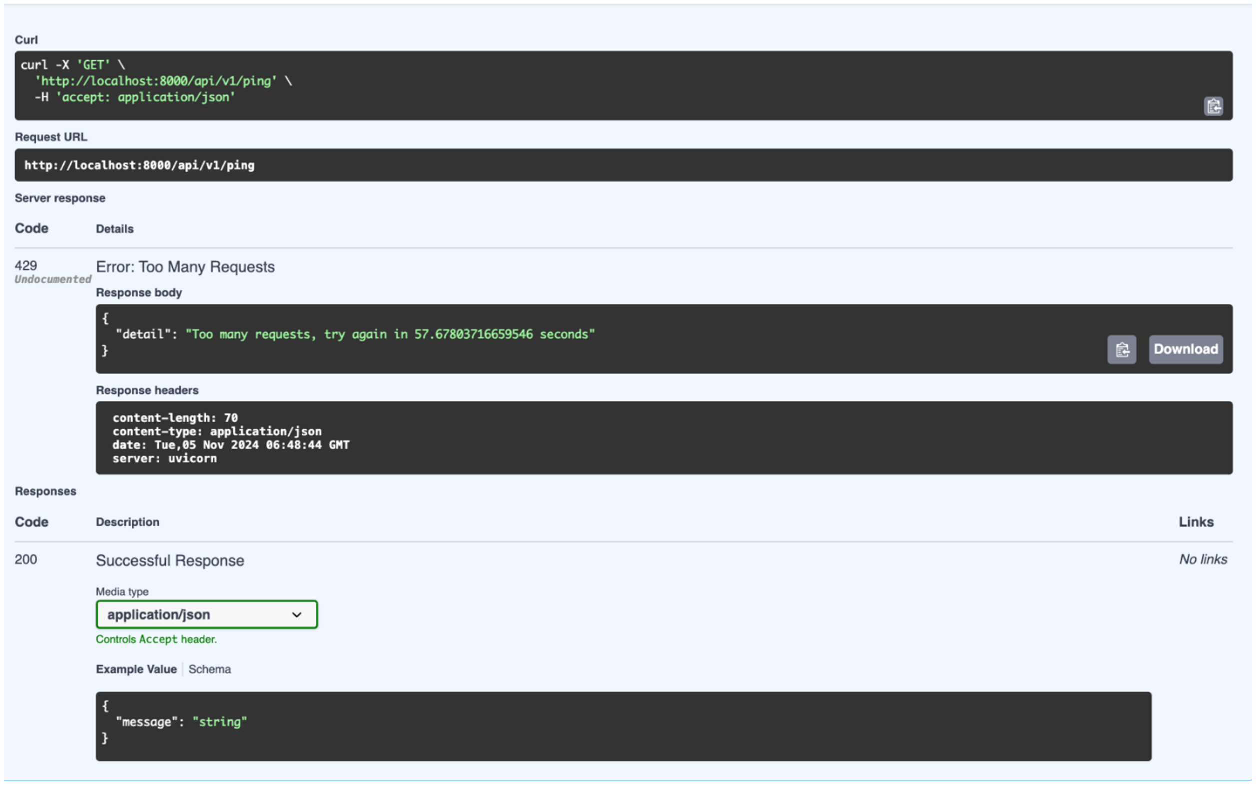
    return decorator

Appendix 9: Rate Limiter Wrapper



```python

@app.get(f"{API_PREFIX}/ping", response_model=PingModel)
@RateLimiter(max_calls=10, cooldown_time=60)
async def read_root(req: Request):
    """Health check endpoint"""
    logger.info("Ping!")
    return {"message": "pong"}
```

Appendix 10: Rate Limiter usage, allows users to access 10 times and ban for 60 seconds



Curl

```
curl -X 'GET' \
  'http://localhost:8000/api/v1/ping' \
  -H 'accept: application/json'
```

Request URL

```
http://localhost:8000/api/v1/ping
```

Server response

Code | Details

429 *Undocumented* Error: Too Many Requests

Response body

```
{
  "detail": "Too many requests, try again in 57.67803716659546 seconds"
}
```

Response headers

```
 content-length: 70
 content-type: application/json
 date: Tue,05 Nov 2024 06:48:44 GMT
 server: uvicorn
```

Responses

Code | Description | Links

200 Successful Response *No links*

Media type: application/json

Controls Accept header.

Example Value | Schema

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

Appendix 11: Rate Limiter in action