

Streamlining Full-Stack Development with FastAPI and Next.js

Ryan Elian

PyCon APAC 2024

Yogyakarta, Indonesia

26 October 2024



Ryan Elian

- From Jakarta, Indonesia
- Microsoft Certified Solutions Developer
- <https://github.com/ryanelian>
- <https://www.linkedin.com/in/ryan-elian/>
- Currently living and working in Tokyo, Japan
- Software Engineer at HENNGE
- <https://recruit.hennge.com/en/gip/>
- <https://recruit.hennge.com/en/mid-career-ngh/>



Common Pains in Web Application Projects

Frontend-Backend disconnect

API type safety concerns

Frontend state management complexity

Pain #1: Frontend-Backend Disconnect

- Frontend and backend teams often work in silos
- API documentation can be outdated, incomplete, or inaccurate
- Frontend developers may struggle to understand how to use the API correctly, leading to integration problems
- The disconnect between frontend and backend can slow down development velocity, which may cause slower time to market and difficulty to adapt to changing requirements

Pain #2: API type safety concerns

- In dynamically typed languages (like JavaScript), type errors often surface at runtime, leading to unexpected crashes and unpredictable behavior.
- Example: Receiving number from an API endpoint returning a string can cause errors.
- While TypeScript helps with frontend type safety, the standard fetch API can be a weak link.
- fetch often returns any by default, making it easy to miss type errors related to API responses.

Pain #2: API type safety concerns

```
node_modules > .pnpm > @types+node@20.16.15 > node_modules > @types > node > TS globals.d.ts > {} global > ⚡ fetch
 87 declare global {
 423
 424   function fetch([
 425     input: string | URL | globalThis.Request,
 426     init?: RequestInit,
 427   ]: Promise<Response>;
 428
 429   interface Request extends _Request {}
 430   var Request: typeof globalThis extends {
 431     onmessage: any;
 432     Request: infer T;
 433   } ? T
 434   : typeof import("undici-types").Request;
 435
 436   interface ResponseInit extends _ResponseInit {}
 437
 438   interface Response extends _Response {}
 439   var Response: typeof globalThis extends {
 440     onmessage: any;
 441     Response: infer T;
 442   } ? T
 443   : typeof import("undici-types").Response;
 444
```

Pain #2: API type safety – Request Body Type



```
export interface RequestInit {  
    method?: string  
    keepalive?: boolean  
    headers?: HeadersInit  
    body?: BodyInit | null  
    redirect?: RequestRedirect  
    integrity?: string  
    signal?: AbortSignal | null  
    credentials?: RequestCredentials  
    mode?: RequestMode  
    referrer?: string  
    referrerPolicy?: ReferrerPolicy  
    window?: null  
    dispatcher?: Dispatcher  
    duplex?: RequestDuplex  
}
```

```
export type BodyInit =  
    | ArrayBuffer  
    | AsyncIterable<Uint8Array>  
    | Blob  
    | FormData  
    | Iterable<Uint8Array>  
    | NodeJS.ArrayBufferView  
    | URLSearchParams  
    | null  
    | string
```

坑

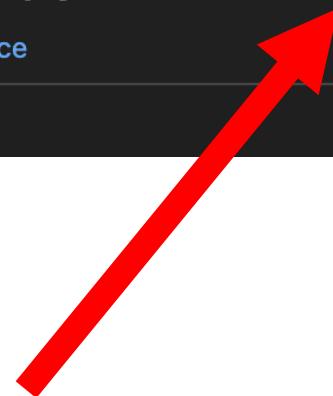
Pain #2: API type safety – Response Body Type



```
const response = await fetch(  
  'https://placebear.com/200/300',  
);  
const data = response.json()
```

(method) Body.json(): Promise<any>
MDN Reference
const response = await response.json()
const data = response.json()

惊





Part 3: Frontend state management complexity

- Modern web apps juggle data from many sources (user input, API calls, real-time updates). Managing this efficiently is key to avoiding a tangled mess of code and unpredictable behavior.
- While essential, fetch lacks state management. It's easy to fall into traps like network waterfalls and redundant requests.
- From React official documentations:
 - <https://react.dev/reference/react/useEffect#fetching-data-with-effects>
 - "Writing fetch calls inside Effects is a popular way to fetch data, especially in fully client-side apps. This is however, a very manual approach and it has significant downsides!"

Today we are going to:

Bootstrap Fast API project from scratch

Bootstrap Next.js project from scratch

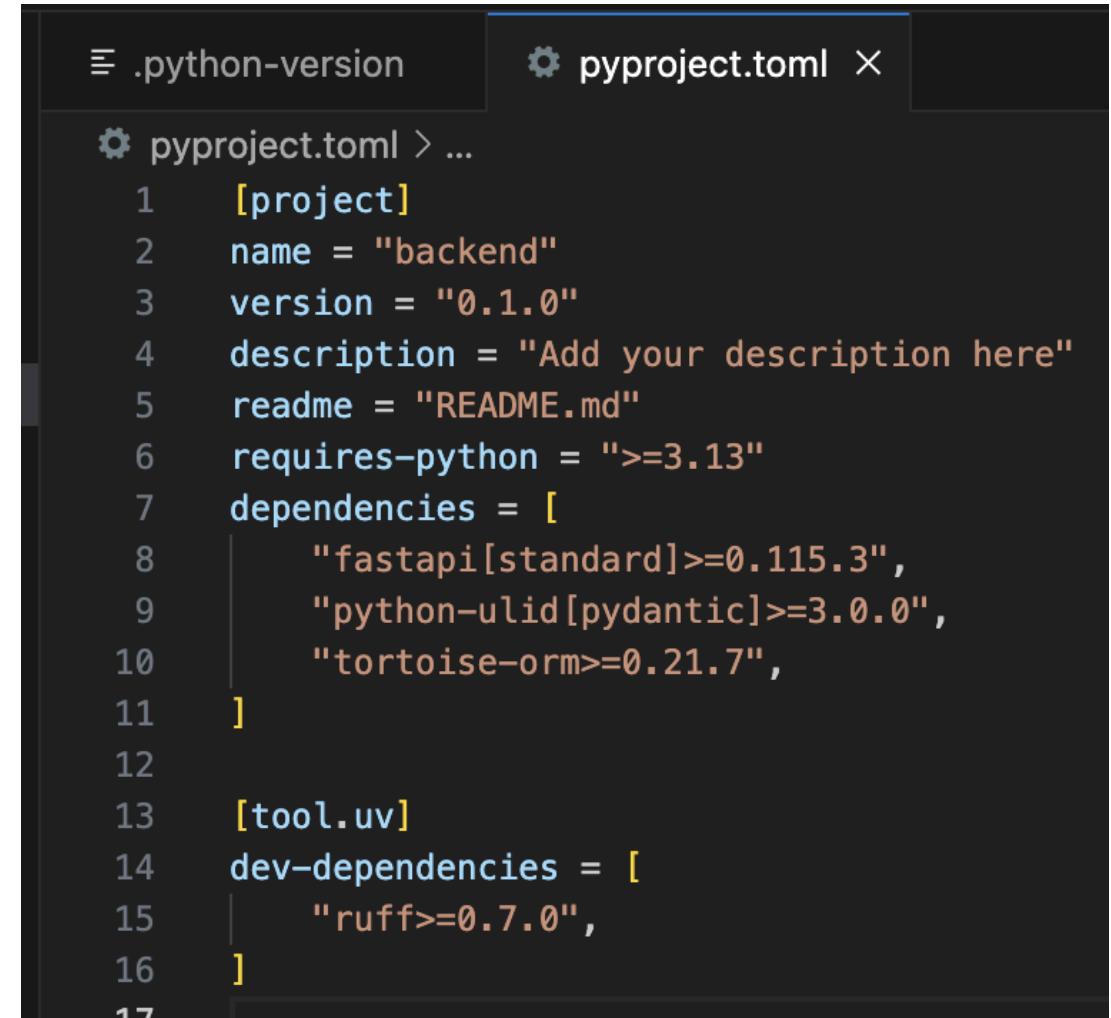
Demonstrate how to integrate both projects

System requirements

- Install uv
 - <https://github.com/astral-sh/uv>
 - An extremely fast Python package and project manager, written in Rust
 - A single tool to replace pip, pip-tools, pipx, poetry, pyenv, virtualenv, and more
- Install Node.js
 - <https://nodejs.org/en>
- Install PNPM
 - <https://pnpm.io/>
 - Fast project and package manager for Node.js

Create new Python / FastAPI Project

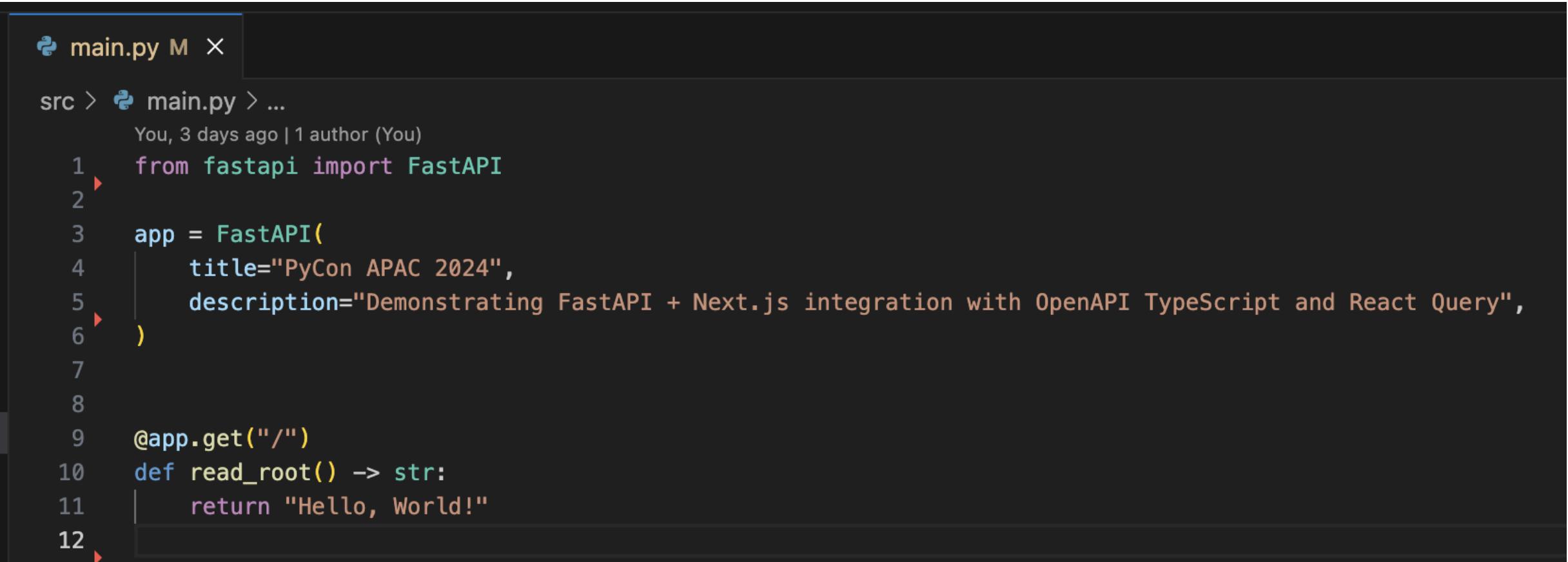
- mkdir ~/code/pycon
- cd ~/code/pycon
- uv init --app backend
- cd backend
- uv sync
- uv add "fastapi[standard]"
- uv add "python-ulid[pydantic]"
- uv add "tortoise-orm"
- code .



The image shows a code editor interface with two tabs: ".python-version" and "pyproject.toml". The "pyproject.toml" tab is active, displaying the following configuration:

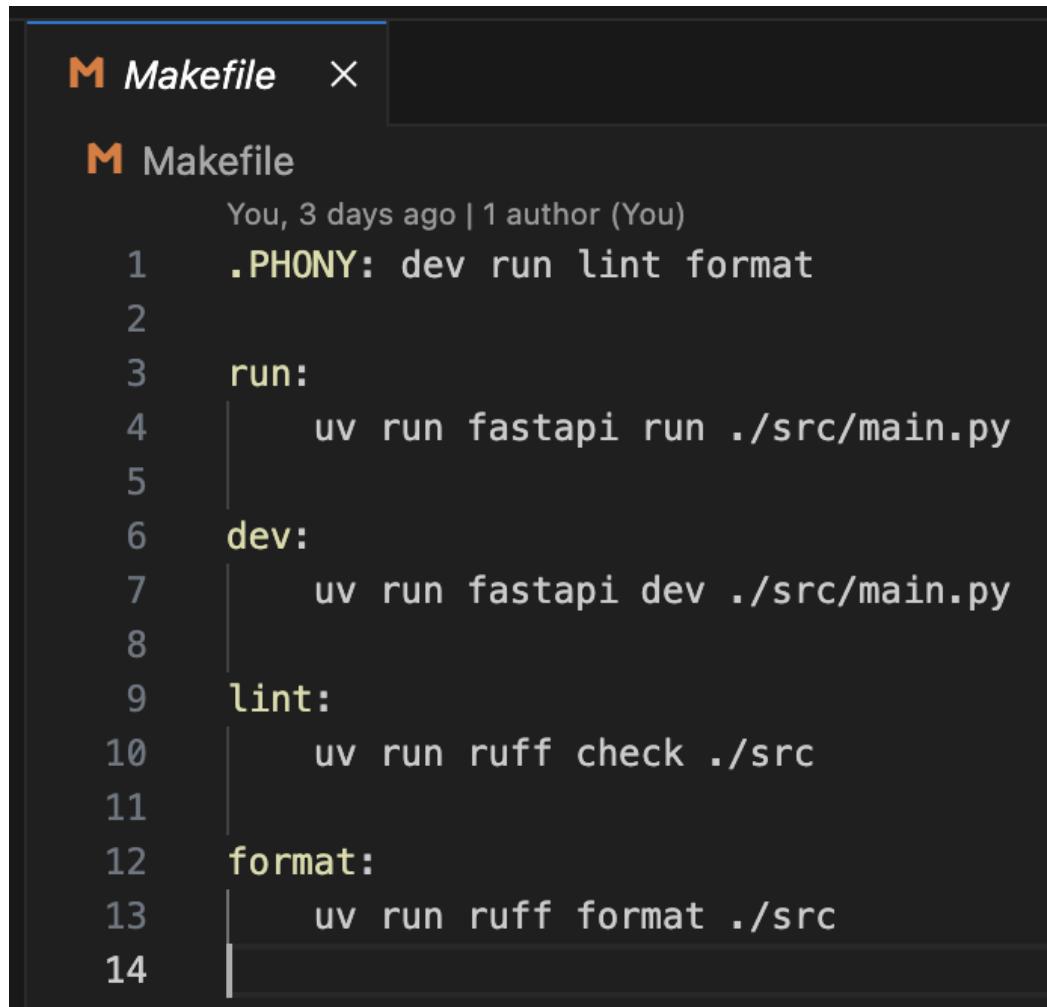
```
1 [project]
2 name = "backend"
3 version = "0.1.0"
4 description = "Add your description here"
5 readme = "README.md"
6 requires-python = ">=3.13"
7 dependencies = [
8     "fastapi[standard]>=0.115.3",
9     "python-ulid[pydantic]>=3.0.0",
10    "tortoise-orm>=0.21.7",
11 ]
12
13 [tool.uv]
14 dev-dependencies = [
15     "ruff>=0.7.0",
16 ]
17
```

Create /src/main.py



```
main.py M X
src > main.py > ...
You, 3 days ago | 1 author (You)
1 from fastapi import FastAPI
2
3 app = FastAPI(
4     title="PyCon APAC 2024",
5     description="Demonstrating FastAPI + Next.js integration with OpenAPI TypeScript and React Query",
6 )
7
8
9 @app.get("/")
10 def read_root() -> str:
11     return "Hello, World!"
```

Scripts to run the project



A screenshot of a code editor showing a Makefile. The file contains the following content:

```
Makefile
Makefile
You, 3 days ago | 1 author (You)
.PHONY: dev run lint format

run:
    uv run fastapi run ./src/main.py

dev:
    uv run fastapi dev ./src/main.py

lint:
    uv run ruff check ./src

format:
    uv run ruff format ./src
```

- If you do not want to use Makefile, just make .sh files
- Run the development server:
 - make dev

http://localhost:8000/docs

PyCon APAC 2024 0.1.0 OAS 3.1

[/openapi.json](#)

Demonstrating FastAPI + Next.js integration with OpenAPI TypeScript and React Query

default

GET / Read Root

Parameters

No parameters

Responses

Code	Description	Links
200	Successful Response	No links

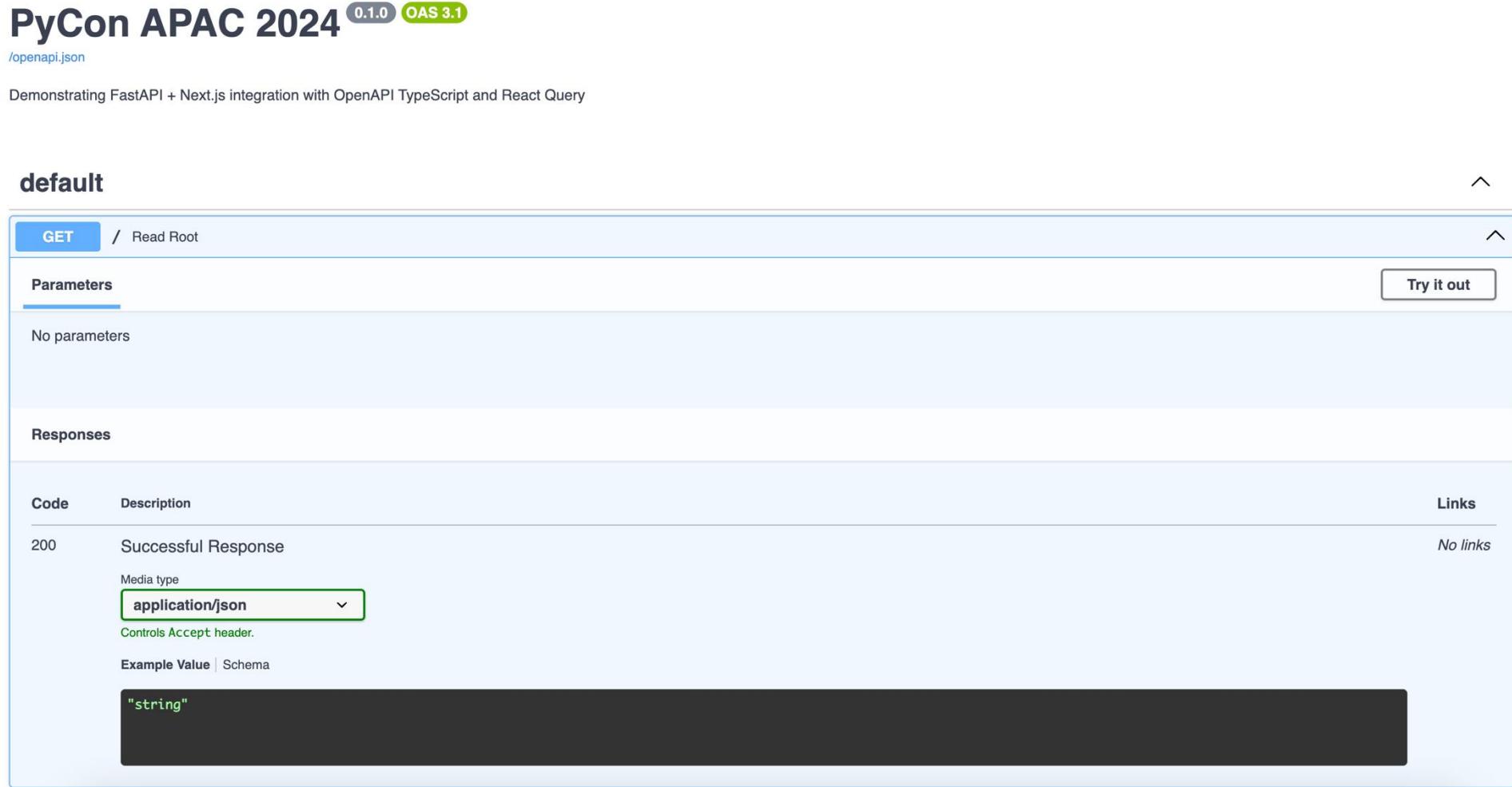
Media type

application/json

Controls Accept header.

[Example Value](#) | [Schema](#)

"string"



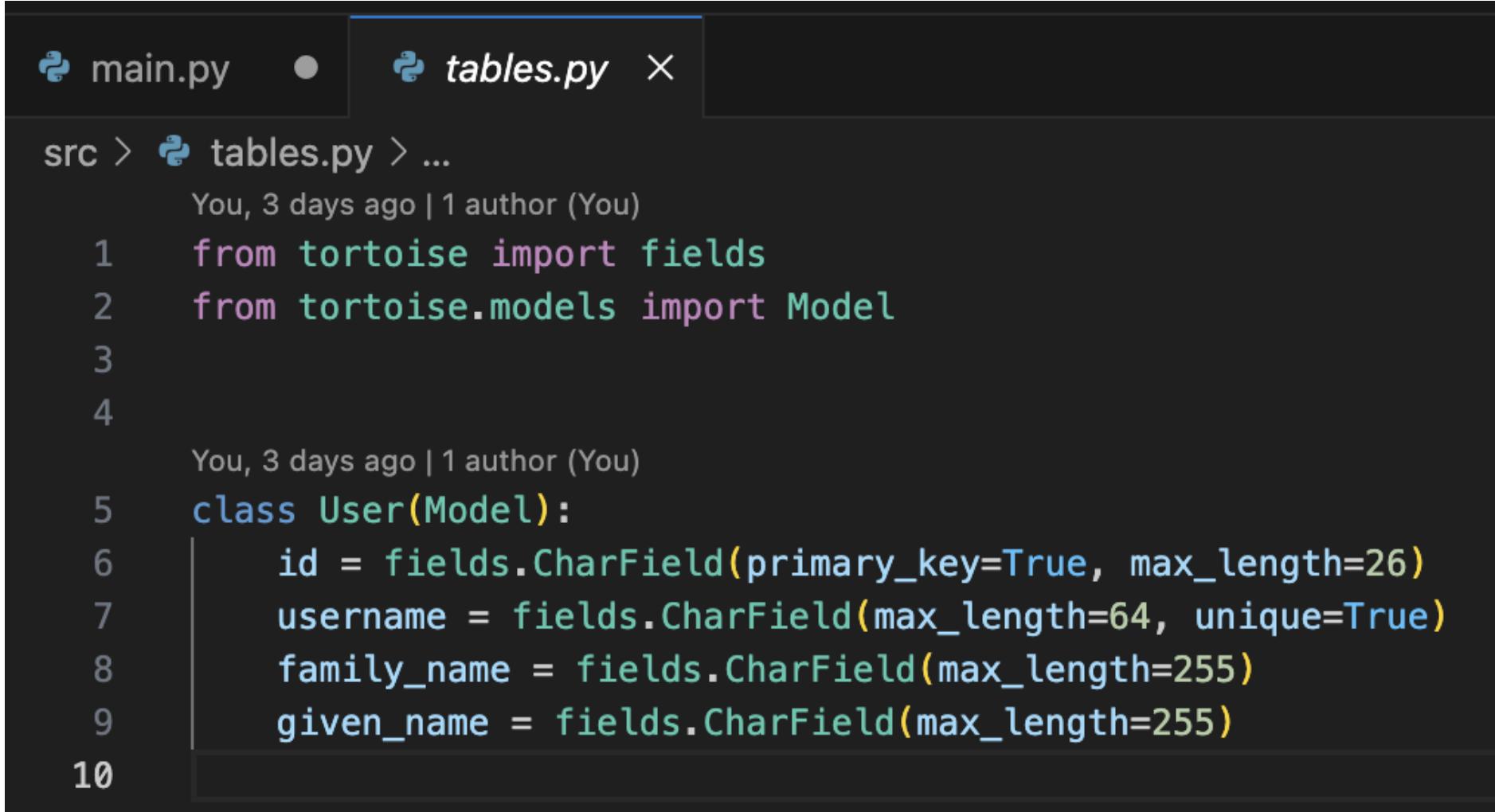
Let's add a database! (SQLite)

```
from tortoise.contrib.fastapi import RegisterTortoise

@asynccontextmanager
async def lifespan(app: FastAPI):
    async with RegisterTortoise(
        app,
        db_url="sqlite://database.db",
        modules={"models": ["tables"]},
        # don't do this in production 🙃 🙃 🙃
        # usually, migration scripts or CLI is used to generate tables
        generate_schemas=True,
        add_exception_handlers=True,
    ):
        yield

app = FastAPI(
    title="PyCon APAC 2024",
    description="Demonstrating FastAPI + Next.js integration with OpenAPI TypeScript and React Query",
    lifespan=lifespan,
)
```

Create tables.py



The screenshot shows a code editor interface with a dark theme. At the top, there are two tabs: 'main.py' (disabled) and 'tables.py' (active). Below the tabs, the file structure is shown as 'src > tables.py > ...'. The commit history for 'tables.py' is visible, with one entry from 'You, 3 days ago | 1 author (You)'. The code itself is a Python class definition for 'User' using the 'tortoise' library:

```
src > tables.py > ...
You, 3 days ago | 1 author (You)
1  from tortoise import fields
2  from tortoise.models import Model
3
4
5  You, 3 days ago | 1 author (You)
6  class User(Model):
7      id = fields.CharField(primary_key=True, max_length=26)
8      username = fields.CharField(max_length=64, unique=True)
9      family_name = fields.CharField(max_length=255)
10     given_name = fields.CharField(max_length=255)
```

Create API: GET /v1/users

```
You, 3 days ago | 1 author (You)
from fastapi import APIRouter
from pydantic import BaseModel
from tables import User

router = APIRouter()

You, 3 days ago | 1 author (You)
class V1UserResponseModel(BaseModel):
    id: str
    username: str
    family_name: str
    given_name: str

# GET /users - Retrieve all users
@router.get("/users")
async def get_users_v1() -> list[V1UserResponseModel]:
    users = await User.all().values("id", "username", "family_name", "given_name")
    return [
        V1UserResponseModel(
            id=user["id"],
            username=user["username"],
            family_name=user["family_name"],
            given_name=user["given_name"],
        )
        for user in users
    ]

```

You, 3 days ago • Uncommitted changes

- DO NOT use your table classes directly as request / response models! (Vulnerable to model injection attack!)
- Register the router in your main.py app instance:

```
from api.v1.users import router as v1_users_router
app.include_router(v1_users_router, prefix="/v1")
```

GET /v1/users Get Users V1

Parameters

No parameters

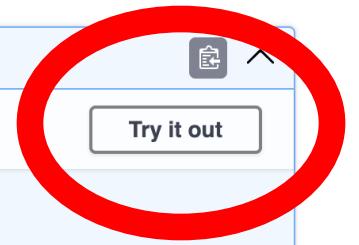
Responses

Curl

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

Request URL

```
http://localhost:8000/v1/users
```



Responses

Code	Description	Links
200	Successful Response	<i>No links</i>

Media type

application/json ▾

Controls Accept header.

[Example Value](#) | [Schema](#)

```
[  
  {  
    "id": "string",  
    "username": "string",  
    "family_name": "string",  
    "given_name": "string"  
  }  
]
```

Try it out!

Code

Details

200

Response body

```
[  
  {  
    "id": "01JANASR7KDFZ3HG0J6MJ3WFM8",  
    "username": "ryanelian",  
    "family_name": "Elian",  
    "given_name": "Ryan"  
  },  
  {  
    "id": "01JANB82QYX0BZNAT6WM203FAD",  
    "username": "jonsnow",  
    "family_name": "Snow",  
    "given_name": "Jon"  
  },  
  {  
    "id": "01JANC52ETZ8QVRAM0CDB3QYCF",  
    "username": "gigachad",  
    "family_name": "Chad",  
    "given_name": "Giga"  
  }]
```



Download

Response headers

```
content-length: 298  
content-type: application/json  
date: Wed, 23 Oct 2024 09:21:23 GMT  
server: uvicorn
```

Create a “Create and Update” Model

```
from pydantic import Field

You, 3 days ago | 1 author (You)
class V1UserCreateAndUpdateRequestModel(BaseModel):
    username: str = Field(
        ...,
        min_length=3,
        max_length=64,
        description="Username should be between 3 and 64 characters",      You, 3 days ago
    )
    family_name: str = Field(
        ..., max_length=255, description="Family name must not exceed 255 characters"
    )
    given_name: str = Field(
        ..., max_length=255, description="Given name must not exceed 255 characters"
    )
```

GET /v1/users/{id}

```
from tortoise.exceptions import DoesNotExist
from responses.NotFound404 import NotFound404Exception, not_found_404_response

# GET /users/{id} - Retrieve a specific user by ID
@router.get(
    "/users/{id}",
    responses={400: not_found_404_response},
)
async def get_user_details_v1(id: str) -> V1UserResponseModel:
    try:
        user = await User.get(id=id).values(
            "id", "username", "family_name", "given_name"
        )
    except DoesNotExist:
        raise NotFound404Exception
    return V1UserResponseModel(
        id=user["id"],
        username=user["username"],
        family_name=user["family_name"],
        given_name=user["given_name"],
    )
```

Documenting 404 Not Found response

```
NotFound404.py ×

src > responses > NotFound404.py > ...
    You, 3 days ago | 1 author (You)
1   from fastapi import HTTPException
2   from pydantic import BaseModel
3
4
    You, 3 days ago | 1 author (You)
5   class NotFound404(BaseModel):
6       detail: str
7
8
9   not_found_404_response = {
10      "model": NotFound404,
11      "description": "Resource not found",
12      "content": {"application/json": {"example": {"detail": "Resource not found"}}},
13  }
14
15
    You, 3 days ago | 1 author (You)
16  class NotFound404Exception(HTTPException):
17      def __init__(self):
18          super().__init__(status_code=404, detail="Resource not found")
19
```

POST /v1/users

```
from ulid import ULID

# POST /users - Create a new user with validation, return True if successful
@router.post(
    "/users",
    status_code=201,
)
async def create_user_v1(body: V1UserCreateAndUpdateRequestModel) -> str:
    user = User(
        id=str(ULID()),
        username=body.username,
        family_name=body.family_name,
        given_name=body.given_name,
    )
    await user.save()
    return user.id
```

PATCH /v1/users/{id}

```
# PATCH /users/{id} – Update certain fields of a user by ID, return True if successful
@router.patch("/users/{id}", responses={400: not_found_404_response})
async def update_user_v1(id: str, body: V1UserCreateAndUpdateRequestModel) -> str:
    try:
        user = await User.get(id=id)
        user.username = body.username
        user.family_name = body.family_name
        user.given_name = body.given_name
        await user.save()
        return user.id
    except DoesNotExist:
        raise NotFound404Exception
```

DELETE /v1/users/{id}

```
# DELETE /users/{id} - Delete a user by ID, return True if successful
@router.delete("/users/{id}", responses={400: not_found_404_response})
async def delete_user_v1(id: str) -> str:
    try:
        user = await User.get(id=id)
        await user.delete()
        return user.id
    except DoesNotExist:
        raise NotFound404Exception
```

Documenting non-OK responses!

400	Resource not found	No links
<p>Media type</p> <p>application/json ▾</p> <p>Example Value Schema</p> <pre>{ "detail": "Resource not found" }</pre>		
422	Validation Error	No links
<p>Media type</p> <p>application/json ▾</p> <p>Example Value Schema</p> <pre>{ "detail": [{ "loc": ["string", 0], "msg": "string", "type": "string" }] }</pre>		

<http://localhost:8000/docs>

GET	/v1/users	Get Users V1	▼
POST	/v1/users	Create User V1	▼
GET	/v1/users/{id}	Get User Details V1	▼
PATCH	/v1/users/{id}	Update User V1	▼
DELETE	/v1/users/{id}	Delete User V1	▼

API Version Management

- **Respect the Contract:** Once your API is out there, changing its interface can break existing applications that rely on it. Think of it as a commitment to your users – don't break it!
- **Give consumers time to adapt:** Increment the version when you make changes and run both the old and new versions concurrently.
- **Deprecate with Grace:** Clearly communicate the deprecation of older versions, providing ample time and migration guides to help consumers transition smoothly.

API Version Management Strategy

- Version number based:
 - /v1/users
 - /v2/users
 - **Pros:** Simple, widely understood, easy to manage for most projects.
- Date based:
 - /v20241025/users
 - /v20241026/users
 - **Pros:** Highly specific, clearly indicates when changes were made, useful for APIs with frequent, incremental updates.
 - **Cons:** Can be verbose, may not be suitable for all types of changes, requires careful date management.

GET /v2/users

Implementing cursor-based pagination

You, 5 days ago | 1 author (You)

```
class V2UserResponseModel(BaseModel):
    id: str
    username: str
    family_name: str
    given_name: str
```

You, 22 minutes ago | 1 author (You)

```
class V2UserListResponseModel(BaseModel):
    items: List[V2UserResponseModel]
    cursor: Union[str, None]
```

```
# GET /v2/users – Cursor-based pagination for retrieving users
@router.get("/users", response_model=V2UserListResponseModel)
async def get_users_v2(
    cursor: Optional[str] = Query(
        None, description="The last user ID from the previous page"
    ), # Cursor for pagination
) -> V2UserListResponseModel:
    # Hardcoded limit
    # limit = 50
    limit = 2

    # Build the query
    query = User.all().order_by("id")

    if cursor:
        # Only get users with IDs greater than the provided cursor
        query = query.filter(id__gt=cursor)

    # Fetch the results with a limit
    users = await query.limit(limit).values(
        "id", "username", "family_name", "given_name"
    )

    # Prepare the response data
    items = [
        V2UserResponseModel(
            id=user["id"],
            username=user["username"],
            family_name=user["family_name"],
            given_name=user["given_name"],
        )
        for user in users
    ]

    # If there are possibly more results, as in we managed to fetch 50 items,
    # return the last item's ID as the cursor to continue search
    cursor = items[-1].id if len(items) == limit else None

    return V2UserListResponseModel(items=items, cursor=cursor)
```

Sample paginated response

GET /v2/users Get Users V2 ^

Parameters Cancel

Name	Description
cursor string (query)	The last user ID from the previous page CURSOR

Code Details

200 Response body

```
{  
  "items": [  
    {  
      "id": "01JANASR7KDFZ3HG0J6MJ3WFM8",  
      "username": "ryanelian",  
      "family_name": "Elian",  
      "given_name": "Ryan"  
    },  
    {  
      "id": "01JANB82QYX0BZNAT6WM203FAD",  
      "username": "jonsnow",  
      "family_name": "Snow",  
      "given_name": "Jon"  
    },  
    {  
      "id": "01JANC52ETZ8QVRAM0CDB3QYCF",  
      "username": "gigachad",  
      "family_name": "Chad",  
      "given_name": "Giga"  
    }  
  ],  
  "cursor": ""  
}
```

Download

<http://localhost:8000/docs>

GET	/v1/users	Get Users V1	▼
POST	/v1/users	Create User V1	▼
GET	/v1/users/{id}	Get User Details V1	▼
PATCH	/v1/users/{id}	Update User V1	▼
DELETE	/v1/users/{id}	Delete User V1	▼
GET	/v2/users	Get Users V2	▼

Common Pains in Web Application Projects

Better communication between backend and frontend team using automatic OpenAPI documentation!

API type safety concerns

Frontend state management complexity

Let's make a Next.js app!

```
cd ~/code/pycon
```



The best way to start a full-stack, typesafe Next.js app

Documentation >

GitHub ↗

npm create t3-app@latest



Create T3 App

- `pnpm create t3-app@latest`

File-Based Routing: Hello World

<https://nextjs.org/docs/app/building-your-application/routing>

The screenshot shows a dark-themed code editor interface. On the left is the Explorer sidebar with a tree view of the project structure:

- FRONTEND
- > .next
- > node_modules
- > public
- < src
 - < app
 - layout.tsx
 - page.tsx

The "page.tsx" file is currently selected and highlighted with a blue bar at the bottom of the sidebar. In the main editor area, there are two tabs: "layout.tsx" and "page.tsx". The "page.tsx" tab is active, showing the following code:

```
src > app > page.tsx > ...
1  export default function HomePage() {
2    return <main>Hello World</main>;
3  }
4
```

At the bottom of the editor, there is a toolbar with icons for file operations like copy, paste, and refresh, along with a status bar showing "PyCon APAC 2024 - Swagger" and "Create T3 App". Below the editor is a browser-like header with navigation buttons (back, forward, home) and a URL field showing "localhost:3000". At the very bottom, there is a navigation bar with links to various services: Graphite Billing, Inbox, TeamSpirit, ChatGPT, Graphite, Gemini, and Spacing.

Hello World

How to communicate with the backend?

The screenshot shows the homepage of the OpenAPI TypeScript project. The page has a dark background with white and light blue text. At the top, there is a navigation bar with links for "OpenAPI TS", "Search", and "Version". Below the header, the title "OpenAPI TypeScript" is displayed in large, bold, blue letters. A subtitle below it reads "Convert OpenAPI 3.0/3.1 schemas to TypeScript types and create type-safe fetching." Two buttons are visible: a blue "Get Started" button and a grey "View on GitHub" button. The main content area features three callout boxes with rounded corners. The first box, titled "Blazing Fast", states that nothing is faster than instant. Static TypeScript types provide zero runtime cost and zero client weight. The second box, titled "Type-safe", explains that using your OpenAPI schema to typecheck your entire codebase with no setup and no tests. The third box, titled "Works anywhere", notes that it runs anywhere TypeScript does, works for any stack and any framework.

OpenAPI TS

Search

Version

Blazing Fast

Type-safe

Works anywhere

Get Started

View on GitHub

Convert OpenAPI 3.0/3.1 schemas to TypeScript types and create type-safe fetching.

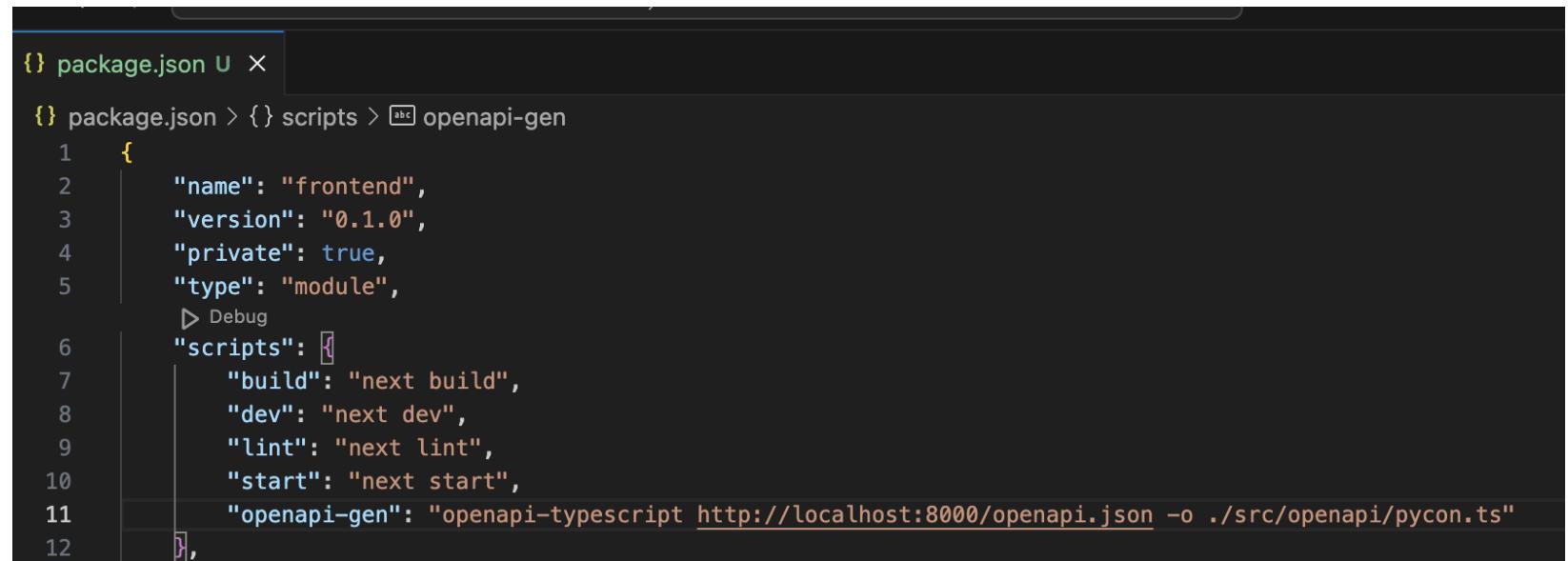
Nothing is faster than instant. Static TypeScript types provide zero runtime cost and zero client weight.

Use your OpenAPI schema to typecheck your entire codebase with no setup and no tests.

Runs anywhere TypeScript does. Works for any stack and any framework.

Generate OpenAPI Client

- pnpm install -E -D openapi-typescript
- pnpm install -E openapi-fetch
- Update package.json:scripts
 - openapi-typescript http://localhost:8000/openapi.json -o ./src/openapi/pycon.ts
- Run script
 - pnpm openapi-gen



```
{} package.json U X
{} package.json > {} scripts > abc openapi-gen
1  {
2    "name": "frontend",
3    "version": "0.1.0",
4    "private": true,
5    "type": "module",
6    "scripts": [
7      "build": "next build",
8      "dev": "next dev",
9      "lint": "next lint",
10     "start": "next start",
11     "openapi-gen": "openapi-typescript http://localhost:8000/openapi.json -o ./src/openapi/pycon.ts"
12   ],
13 }
```

Export a TypeScript OpenAPI Fetch Client

```
TS pyconClient.ts U ×  
src > openapi > TS pyconClient.ts > ...  
1 import createClient from "openapi-fetch";  
2 import type { paths } from "./pycon";  
3  
4 export const pyconClient = createClient<paths>({  
5   baseUrl: "http://localhost:8000/",  
6 });  
7  
8 import type { Middleware } from "openapi-fetch";  
9
```

Modify middleware to throw on error

```
import type { Middleware } from "openapi-fetch";

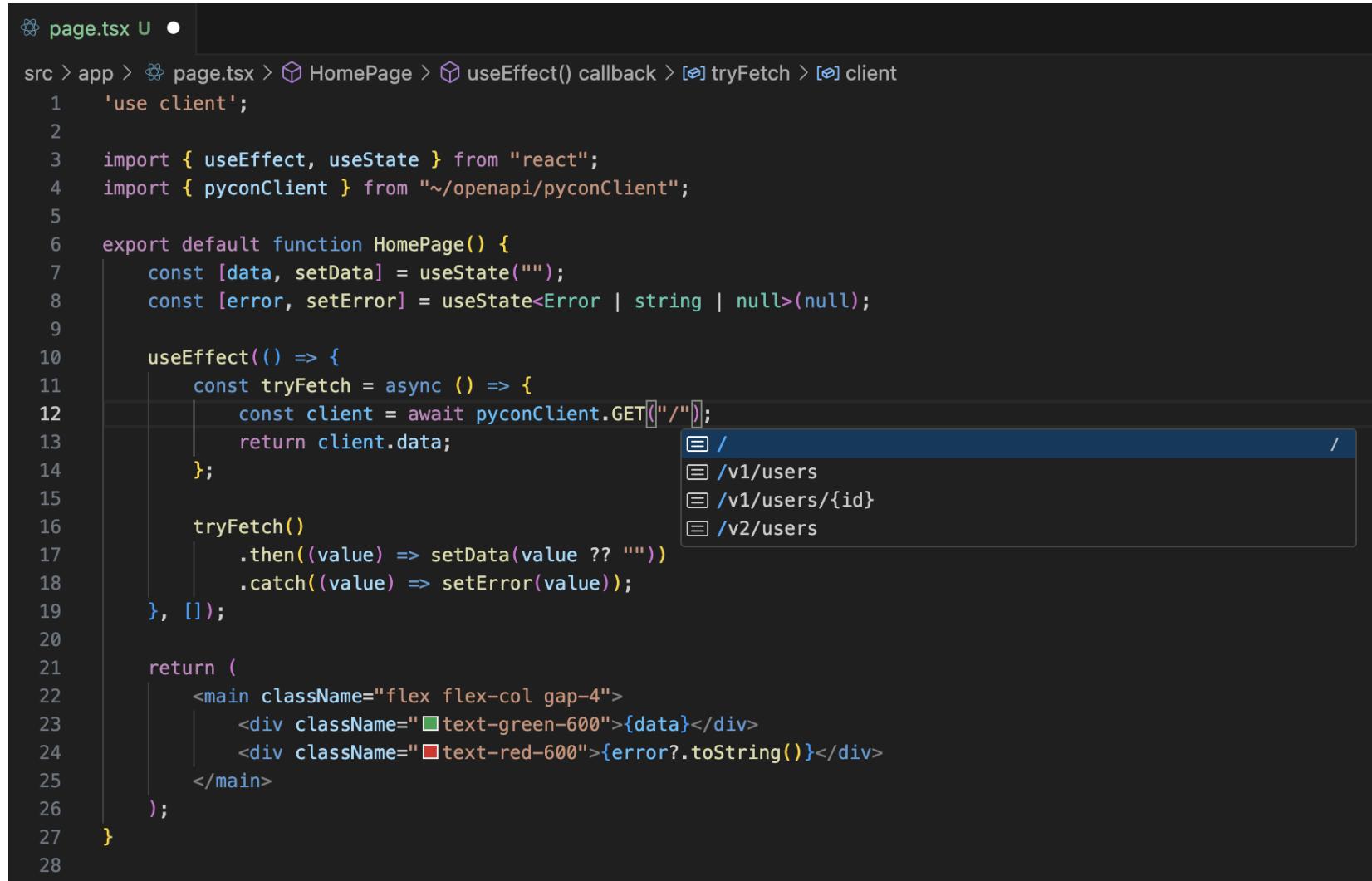
/**
 * Safely tries to get the response body as JSON or text.
 */
async function tryGetResponseBody(response: Response): Promise<unknown> {
    try {
        return response.headers.get("content-type")?.includes("json")
            ? await response.clone().json()
            : await response.clone().text();
    } catch {
        return "(Failed to parse response body)";
    }
}

/**
 * Middleware to throw an error if the response status code is 4xx or 5xx.
 * This is required for react-query to handle errors correctly.
 *
 * @see https://github.com/openapi-ts/openapi-typescript/blob/main/packages/openapi-fetch/examples/react-query/src/lib/api/index.ts
 */
export const throwErrorMiddleware: Middleware = {
    async onResponse({ response }) {
        if (response.status >= 400) {
            const body = await tryGetResponseBody(response);
            throw new DetailedApiError(response.status, response.statusText, body);
        }
        return undefined;
    },
};

class DetailedApiError extends Error {
    constructor(
        public status: number,
        public statusText: string,
        public body: unknown,
    ) {
        super(`HTTP ${status}: ${statusText}`);
    }
}

pyconClient.use(throwErrorMiddleware);
```

Let's use typed API client! 😊



The screenshot shows a code editor with a dark theme. The file is named `page.tsx`. The code is a `HomePage` component using `useState` and `useEffect` hooks. It imports `pyconClient` and defines a `tryFetch` function using `async/await` to call `pyconClient.GET('/')`. The code editor has a tooltip open over the `GET` method, displaying a dropdown menu with options: `/`, `/v1/users`, `/v1/users/{id}`, and `/v2/users`.

```
page.tsx U ●
src > app > page.tsx > HomePage > useEffect() callback > tryFetch > client
1  'use client';
2
3  import { useEffect, useState } from "react";
4  import { pyconClient } from "~/openapi/pyconClient";
5
6  export default function HomePage() {
7      const [data, setData] = useState("");
8      const [error, setError] = useState<Error | string | null>(null);
9
10     useEffect(() => {
11         const tryFetch = async () => {
12             const client = await pyconClient.GET("/");
13             return client.data;
14         };
15
16         tryFetch()
17             .then((value) => setData(value ?? ""))
18             .catch((value) => setError(value));
19     }, []);
20
21     return (
22         <main className="flex flex-col gap-4">
23             <div className="text-green-600">{data}</div>
24             <div className="text-red-600">{error?.toString()}</div>
25         </main>
26     );
27 }
28 }
```

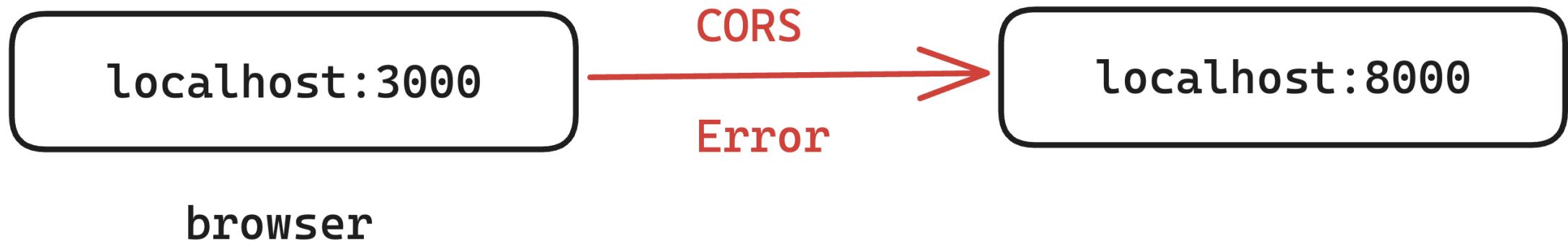
CORS error! 😞



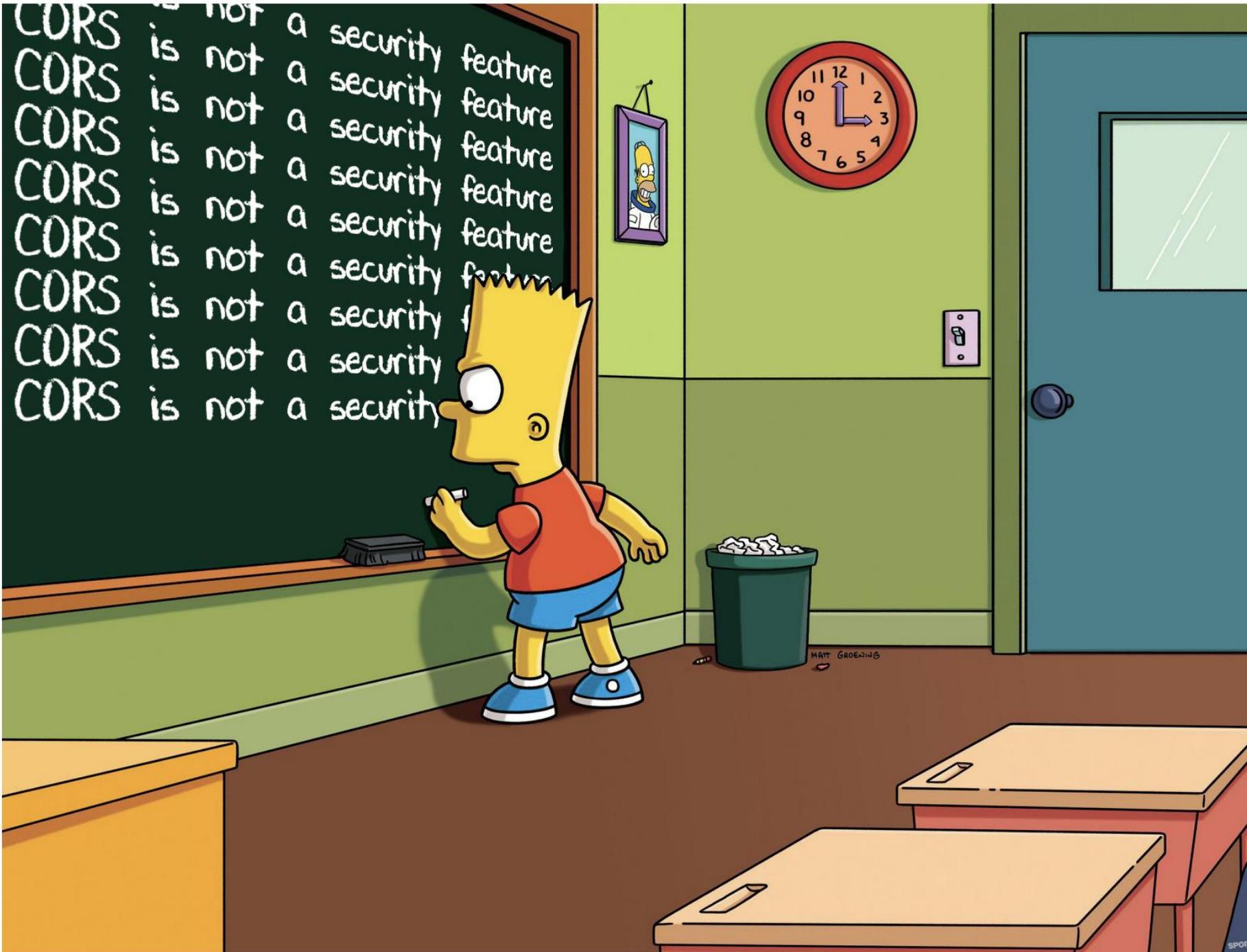
TypeError: Failed to fetch

A screenshot of the Chrome DevTools Network tab. The tab is active and shows several network requests. The configuration section on the left has checkboxes for 'Group similar messages in console' (checked), 'Show CORS errors in console' (checked), 'Preserve log' (unchecked), 'Selected context only' (unchecked), 'Hide network' (unchecked), 'Log XMLHttpRequests' (unchecked), 'Eager evaluation' (checked), 'Autocomplete from history' (checked), and 'Treat code evaluation as user action' (checked). The main pane displays two errors related to CORS policy violations. The first error is 'Access to fetch at 'http://localhost:8000/' from origin 'http://localhost:3000' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.' This is associated with 'localhost:1' and 'page.tsx:12'. The second error is 'GET http://localhost:8000/_net::ERR_FAILED 200 (OK)' followed by another identical CORS violation message, also associated with 'localhost:1' and 'page.tsx:12'.

What happened? What is CORS?



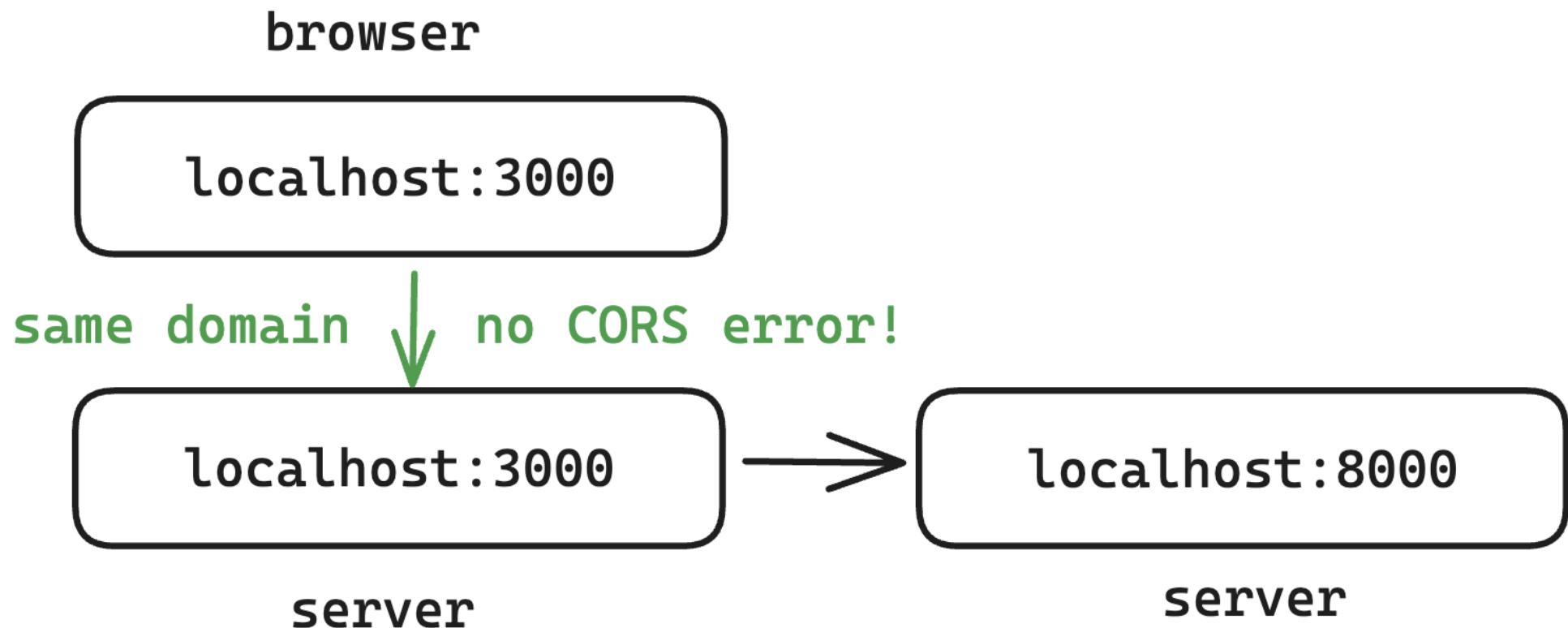
CORS is not a security feature
CORS is not a security



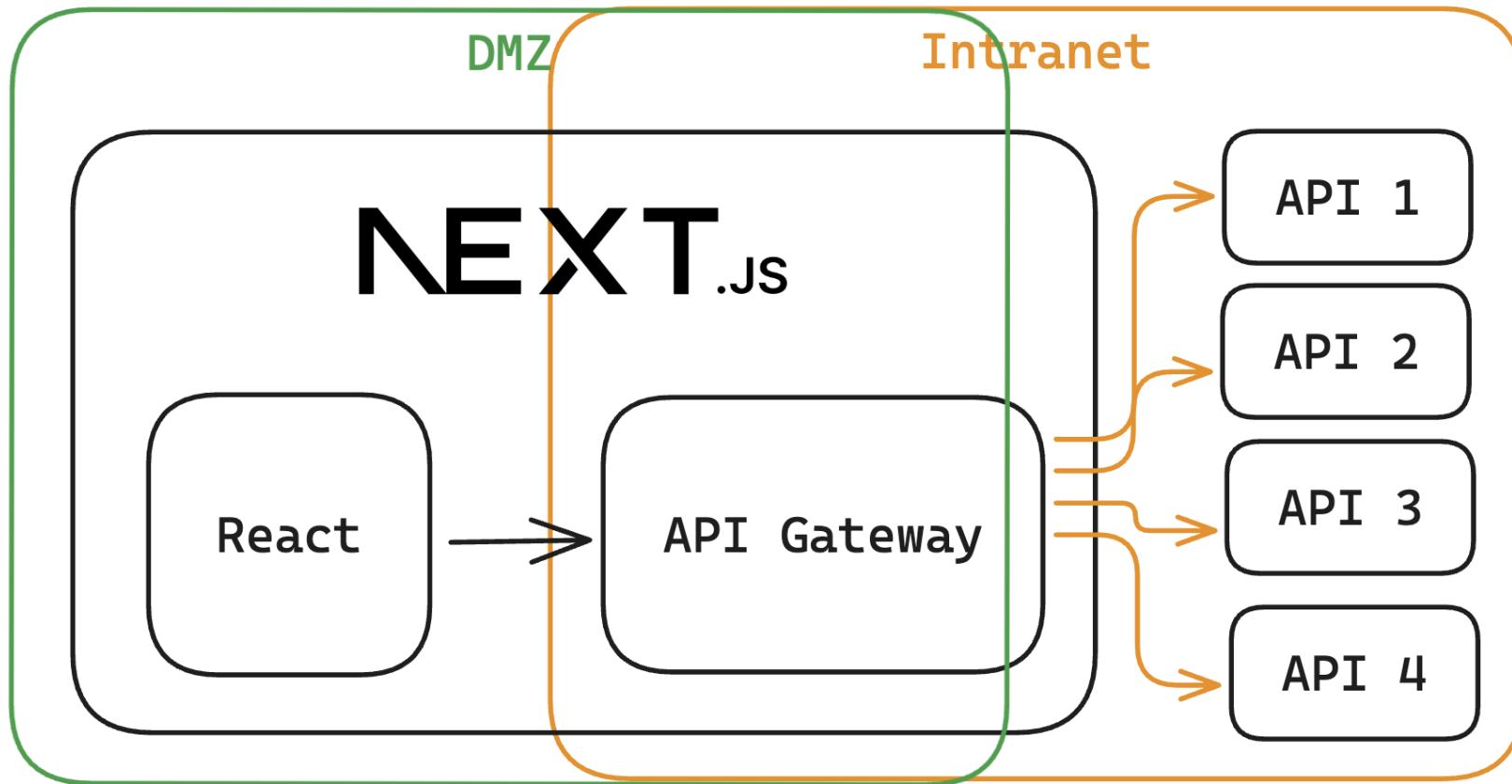
Proxy Path to Backend

```
JS next.config.js U X
JS next.config.js > ...
1  /**
2   * Run `build` or `dev` with `SKIP_ENV_VALIDATION` to skip env validation. This is especially useful
3   * for Docker builds.
4  */
5  await import("./src/env.js");
6
7  /** @type {import("next").NextConfig} */
8  const config = {
9    async rewrites() {
10      return [
11        {
12          source: "/fastapi/:path*",
13          destination: "http://localhost:8000/:path*",
14        },
15      ];
16    },
17  };
18
19  export default config;
20
```

Next.js Server as an API Gateway

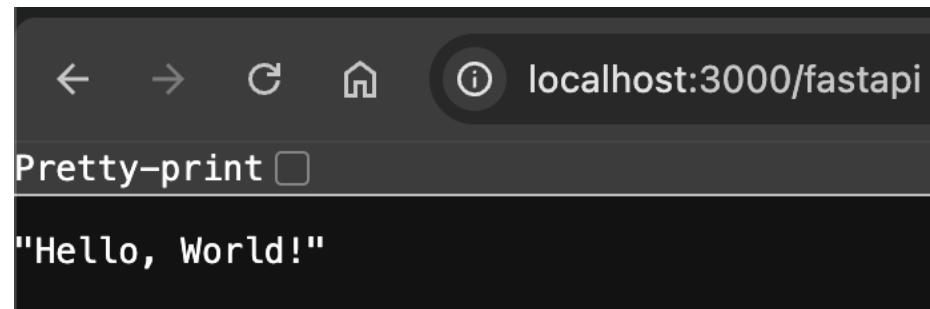


Next.js for Microservices API Management



In Kubernetes, Next.js can be LoadBalancer / NodePort, and your microservices as ClusterIP!

Proxy to Backend (port is 3000, not 8000)



```
src > openapi > TS pyconClient.ts > [o] pyconClient > 🔍 baseUrl
1   import createClient from "openapi-fetch";
2   import type { paths } from "./pycon";
3
4   export const pyconClient = createClient<paths>({
5     baseUrl: "/fastapi/",
6   });
7
```

A screenshot of the Network tab in the Chrome DevTools developer console. The tab bar includes Elements, Console, Network, Sources, Performance, Memory, Application, Security, Lighthouse, Recorder, and Settings. The Network tab is selected. The main pane lists network requests, with the first entry for 'fastapi/' highlighted. The details panel on the right shows the following information for the selected request:

Name	Headers	Preview	Response	Initiator	Timing	Cookies
fastapi/	▼ General					
fastapi/	Request URL:	http://localhost:3000/fastapi				
fastapi	Request Method:	GET				
fastapi	Status Code:	200 OK				
	Remote Address:	[:1]:3000				
	Referrer Policy:	strict-origin-when-cross-origin				

Common Pains in Web Application Projects

Better communication between backend and frontend team using automatic OpenAPI documentation!

OpenAPI TypeScript client allows strongly-typed API data access. Works well with Next.js!

Frontend state management complexity

We don't write tutorial code at work

```
type UserListItem = PythonClientTypes["schemas"]["V1UserResponseModel"];

export default function HomePage() {
  const [data, setData] = useState<UserListItem[]>([]);
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState<unknown>(null);

  useEffect(() => {
    const tryFetch = async () => {
      try {
        setIsLoading(true);
        const response = await pyconClient.GET("/v1/users");

        if (!response.data) {
          throw new Error("No data received");
        }
        setData(response.data);
      } catch (error) {
        setError(error);
      } finally {
        setIsLoading(false);
      }
    };

    tryFetch();
  }, []);
}
```

- Adding isLoading state to prevent one of the two deadliest UX sins: Cumulative Layout Shift
- What's wrong with this code?
 - Fetch call is asynchronous
 - Meaning, it may not finish immediately
 - But if the server re-renders the UI...
 - The client makes second fetch call!
 - There's no guarantee the first fetch call will finish before the second one! (Stale data)

Okay so let's just ignore the first fetch call?

```
useEffect(() => {
  let ignore = false;

  const tryFetch = async () => {
    try {
      setIsLoading(true);
      const response = await pyconClient.GET("/v1/users");

      if (!ignore) {
        return;
      }

      if (!response.data) {
        throw new Error("No data received");
      }

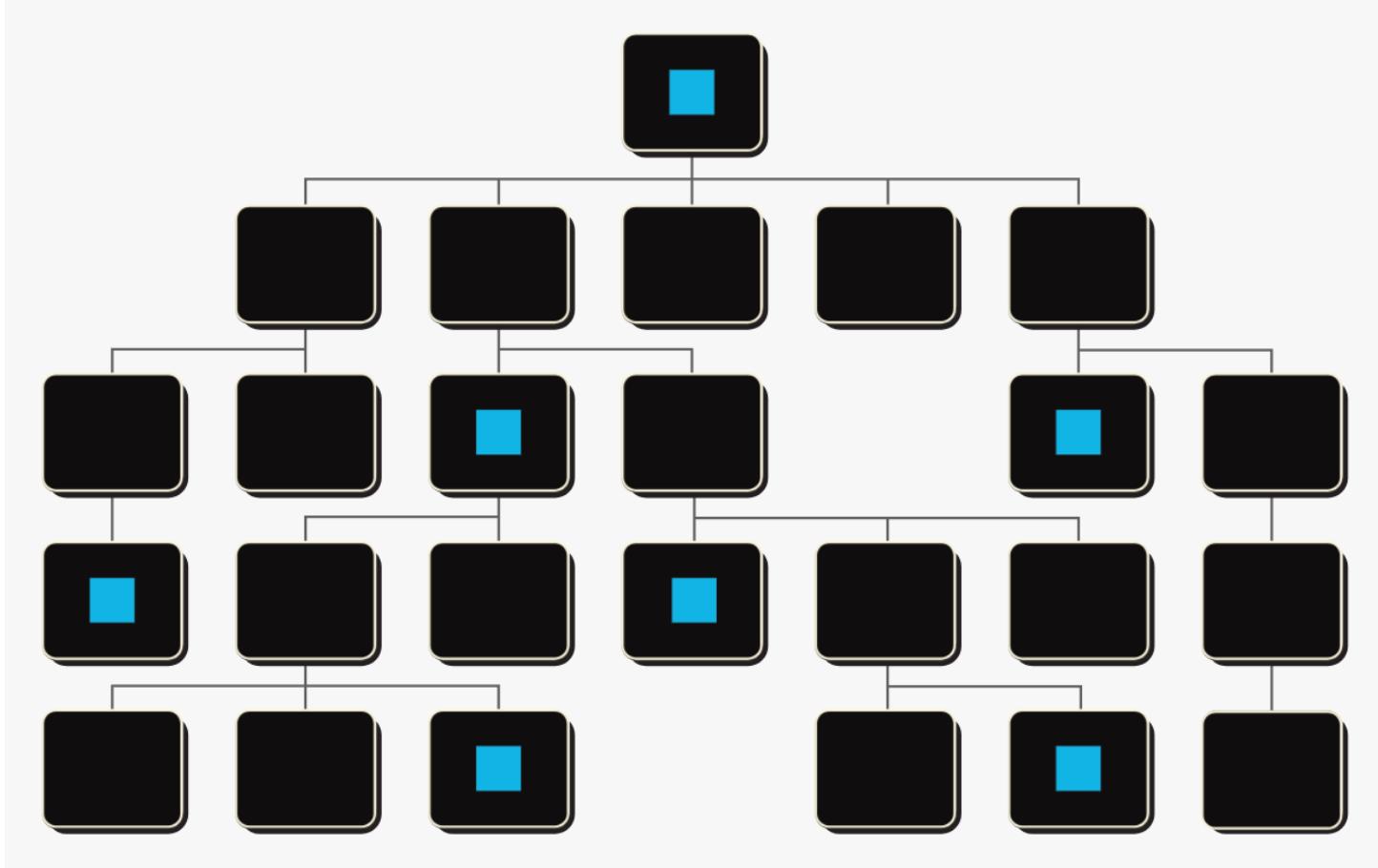
      setData(response.data);
      setIsLoading(false);
    } catch (error) {
      setError(error);
      setIsLoading(false);
    }
  };

  tryFetch();

  return () => {
    ignore = true;
 };
}, []);
```

- Still doesn't solve the problem
- Fetched data is local to the component that fetched it
- What if the component is rendered in two different places?
 - For example, a dropdown

Okay let's just lift the state?



It works! But...

- We have just introduced a small, in-memory cache to the top-level of the application
- Unfortunately, we have just traded our **predictability** problem with **optimization** problem!!!
- When top-level context or drilled props changed, React will re-render the entire application tree
- Even when the change is not related to the specific URL that is being fetched!

Did I say cache?

- There are two hard problems in computer science:
 1. Cache Invalidation
 2. Naming Things
 3. Off-By-One Errors
- <https://martinfowler.com/bliki/TwoHardThings.html>

State Management Strategy

- From React official documentations:
 - <https://react.dev/reference/react/useEffect#fetching-data-with-effects>
 - "Writing fetch calls inside Effects is a popular way to fetch data, especially in fully client-side apps. This is however, a very manual approach and it has significant downsides!"
- When you need to manage states owned by your application (client-states), use Jotai atoms (<https://jotai.org/>)
- When you need to manage states owned by a remote service (server-states), use React Query (<https://tanstack.com/query/v3>)
 - 1 in 5 React applications use React Query

Add React Query to Project

- pnpm install @tanstack/react-query -E
- Modify the root layout.tsx to include React Query Client Provider
 - <https://tanstack.com/query/latest/docs/framework/react/guides/advanced-ssr#initial-setup>

useQuery

- useQuery deduplicates fetch calls, manages cache, reloads when data is stale
- queryKey allows invalidating cache when user data is updated elsewhere
 - <https://tanstack.com/query/latest/docs/framework/react/guides/query-invalidation>
 - <https://tanstack.com/query/latest/docs/framework/react/guides/invalidations-from-mutations>

```
export default function HomePage() {
  const { data, isLoadingError, error, isLoading } = useQuery({
    queryKey: ["users"],
    queryFn: async () => {
      const response = await pyconClient.GET("/v1/users");
      return response.data;
    },
  });

  const users = data ?? [];

  return (
    <main className="flex flex-col gap-4">
      <h1 className="text-4xl">Users</h1>
      {isLoading && <p>Loading...</p>}
      {isLoadingError ? <p>{toErrorMessage(error)}</p> : null}

      {users.map((user) => (
        <div key={user.id} className="flex flex-col gap-1">
          <h2>{user.username}</h2>
          <p>
            {user.family_name} {user.given_name}
          </p>
        </div>
      ))}
    </main>
  );
}
```

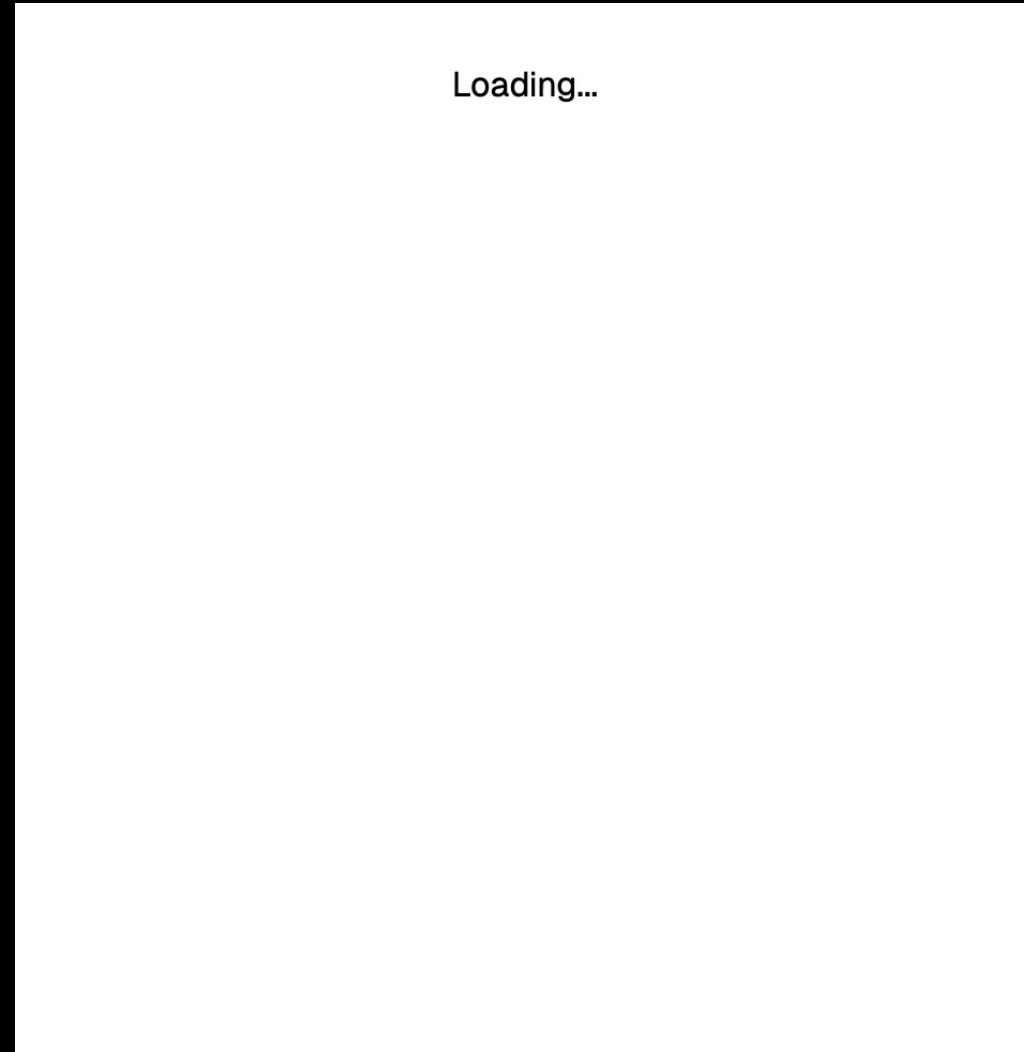
When does stale query get refetched?

- New instances of query mount
- Window is re-focused
- Network is re-connected after disconnection
- Query is configured with auto-refetch interval
- **Makes your frontend application much more sane!**
- <https://tanstack.com/query/latest/docs/framework/react/overview>

Again, we don't write tutorial
code at work

What about Pagination? Search? Filtering? Error Page? Loading Indicator?

Infinite Pagination Example:



Loading...

useInfiniteQuery

- Use previous page cursor to fetch the next page
- When cursor is undefined / null, there are no more pages

```
const fetchPage = async ({  
  pageParam,  
}: {  
  pageParam?: string;  
) => {  
  const response = await pyconClient.GET("/v2/users", {  
    params: {  
      query: {  
        cursor: pageParam,  
      },  
    },  
  });  
  
  return {  
    items: response.data?.items ?? [],  
    cursor: response.data?.cursor,  
  };  
};  
  
export default function HomePage() {  
  const {  
    data,  
    isLoadingError,  
    isLoading,  
    fetchNextPage,  
    hasNextPage,  
    isFetchingNextPage,  
  } = useInfiniteQuery({  
    queryKey: ["users-v2"],  
    queryFn: fetchPage,  
    initialPageParam: undefined,  
    getNextPageParam: (lastPage) => lastPage?.cursor,  
  });
```

useInfiniteQuery (2nd half)

```
if (isLoading) {
  return <div className="flex flex-col items-center">Loading...</div>;
}

if (isLoadingError || !data) {
  return <div>Error Page</div>;
}

return (
  <main className="flex flex-col gap-4 items-center">
    <h1 className="text-4xl">Users</h1>

    {data.pages.map((page) =>
      page?.items.map((user) => (
        <div
          key={user.id}
          className="flex w-96 flex-col gap-2 border border-neutral-600 p-4 rounded-lg bg-white shadow"
        >
          <h2 className="text-2xl font-medium">
            {user.family_name} {user.given_name}
          </h2>
          <p>{user.username}</p>
        </div>
      )));
    }

    <div>
      <DataLoader
        isFetchingNextPage={isFetchingNextPage}
        fetchNextPage={fetchNextPage}
        hasNextPage={hasNextPage}
      />
    </div>
  </main>
);
```

```
function DataLoader({
  isFetchingNextPage,
  fetchNextPage,
  hasNextPage,
}: {
  isFetchingNextPage: boolean;
  fetchNextPage: () => void;
  hasNextPage: boolean;
}) {
  if (isFetchingNextPage) {
    return <span>Loading More...</span>;
  }

  if (hasNextPage) {
    return (
      <button
        type="button"
        onClick={() => {
          fetchNextPage();
        }}
      >
        Load More
      </button>
    );
  }
}

return <span>End of List</span>;
}
```

Solving pains in Web App Projects

Better communication between backend and frontend team using automatic OpenAPI documentation!

OpenAPI TypeScript client allows strongly-typed API data access. Works well with Next.js!

React Query allows management of complex server states, such as API calls



We build
too many
walls and
not enough
bridges.

Isaac Newton



If I have seen
further than
others, it is by
standing upon the
shoulders of
giants.

Isaac Newton

Takeaways

- In software development, we often face challenges where communication gaps exist, particularly between the frontend and backend teams.
- Using tools like **OpenAPI documentation** and **TypeScript OpenAPI clients**, we can build bridges that connect these teams more effectively.
- Sometimes, the solutions to our problems are already out there—just waiting to be used.
- By leveraging powerful tools like **FastAPI**, **Next.js**, and **React Query** we can build upon the foundations laid by others and create truly exceptional web applications in timely manner.

Thank You for Listening!

Q&A

<https://github.com/ryanelian/pycon-apac-2024>

<https://recruit.hennge.com/en/gip/>

<https://recruit.hennge.com/en/mid-career-ngh/>

