Streamlining Full-Stack Development with FastAPI and Next.js

Ryan Elian

PyCon APAC 2024

Yogyakarta, Indonesia

26 October 2024

HENNGE

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- API documentation can be outdated, incomplete, or inaccurate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 😱

```ts
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
}
```

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

Pain #2: API type safety – Response Body Type 😱

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

```
const response = await
const data = response.json()
```

(method) Body.json(): Promise<any>

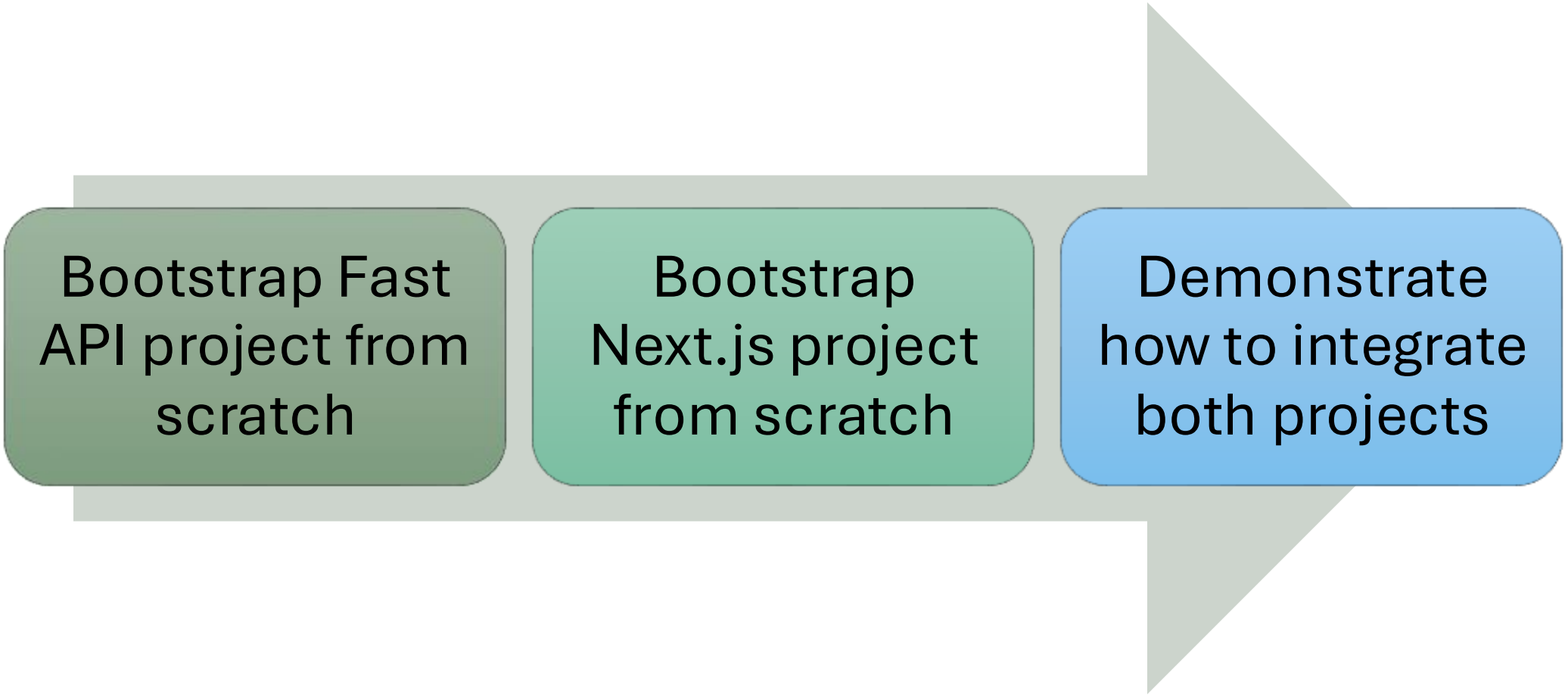
MDN Reference


FETCH
TYPESCRIPT
imgflip.com

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:

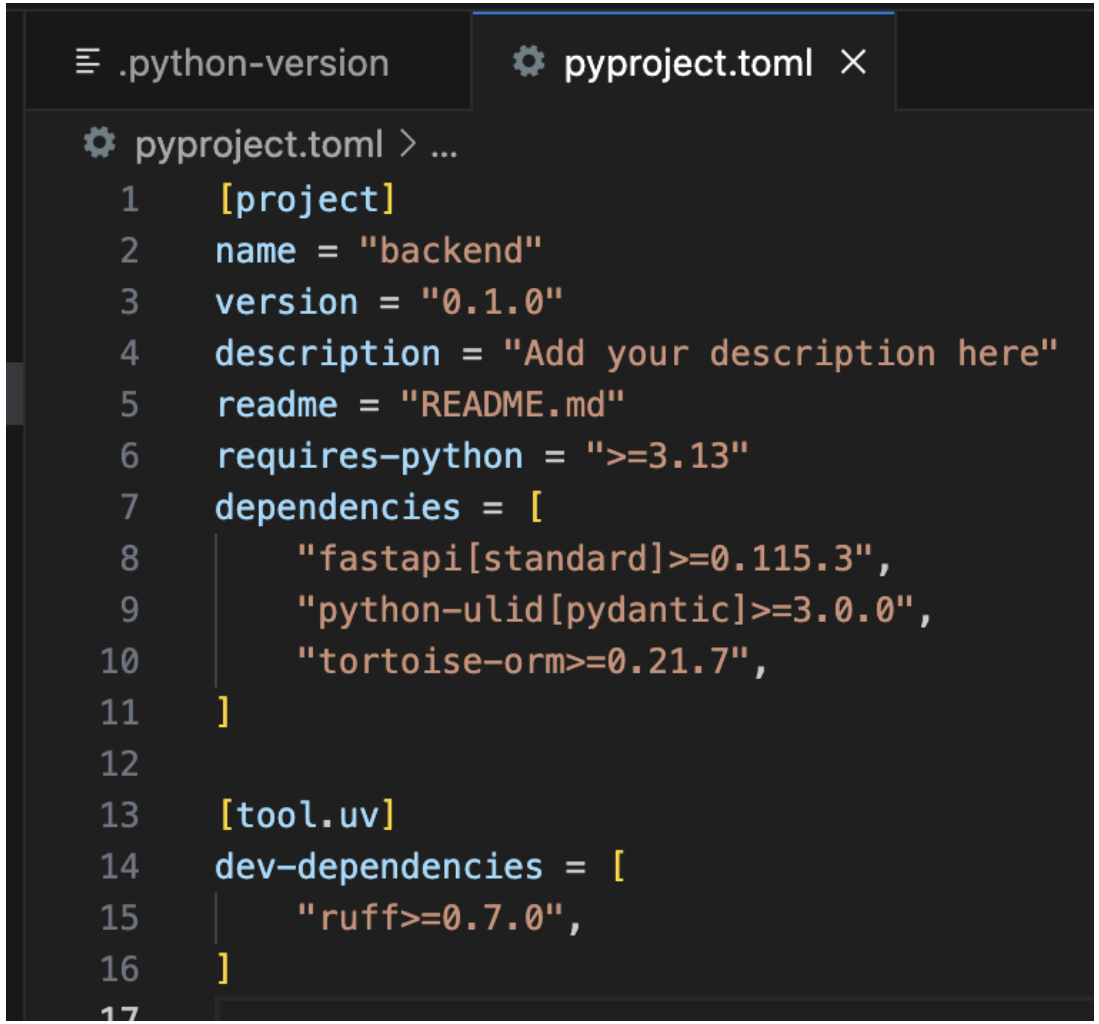


System requirements

- 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 .`



Create /src/main.py

main.py M

src > main.py > ...

You, 3 days ago | 1 author (You)

```python
from fastapi import FastAPI

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


@app.get("/")
def read_root() -> str:
    return "Hello, World!"

```

Scripts to run the project

Makefile

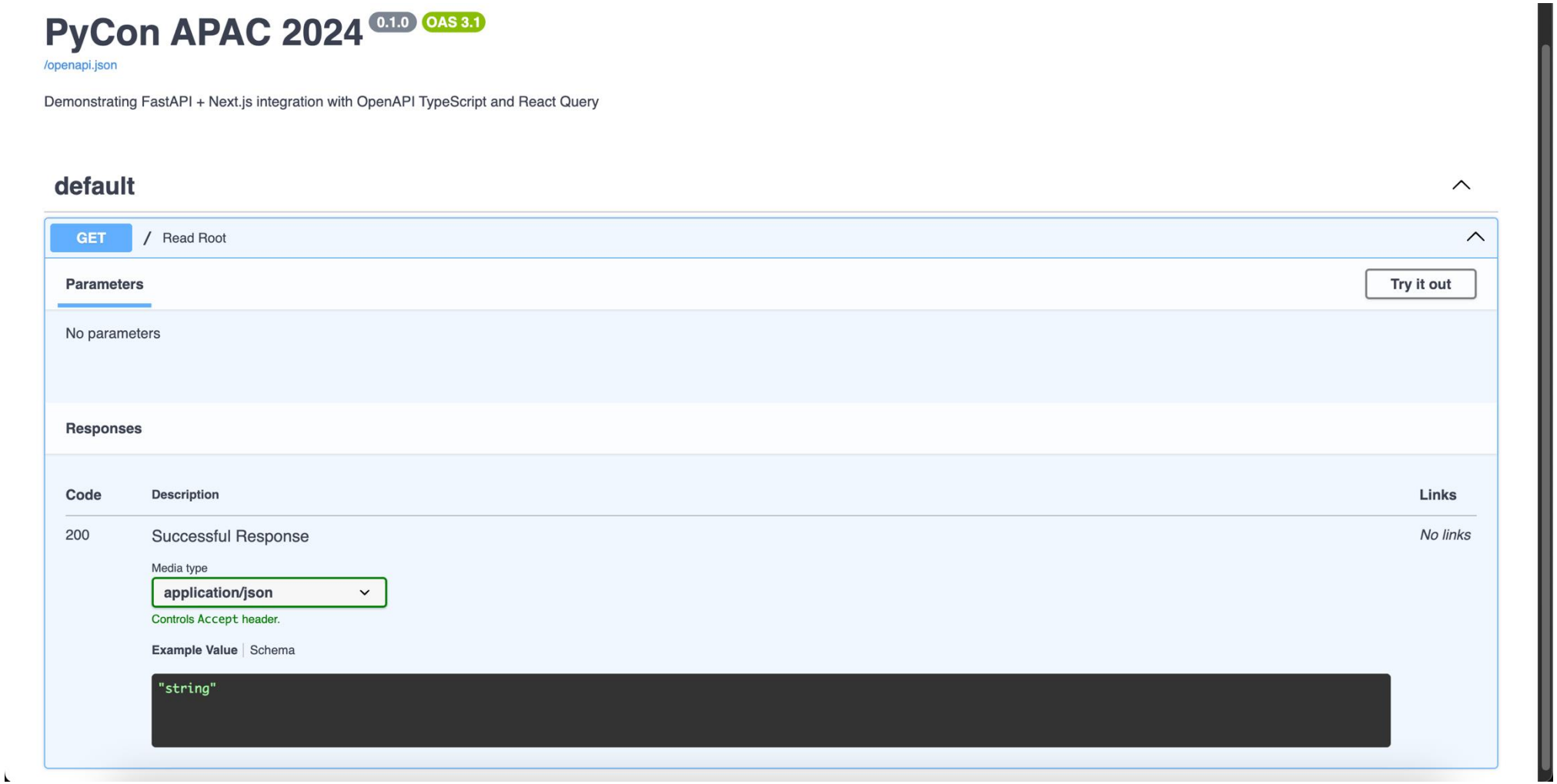
Makefile

You, 3 days ago | 1 author (You)

```
1   .PHONY: dev run lint format
2
3   run:
4       uv run fastapi run ./src/main.py
5
6   dev:
7       uv run fastapi dev ./src/main.py
8
9   lint:
10      uv run ruff check ./src
11
12  format:
13      uv run ruff format ./src
14
```

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

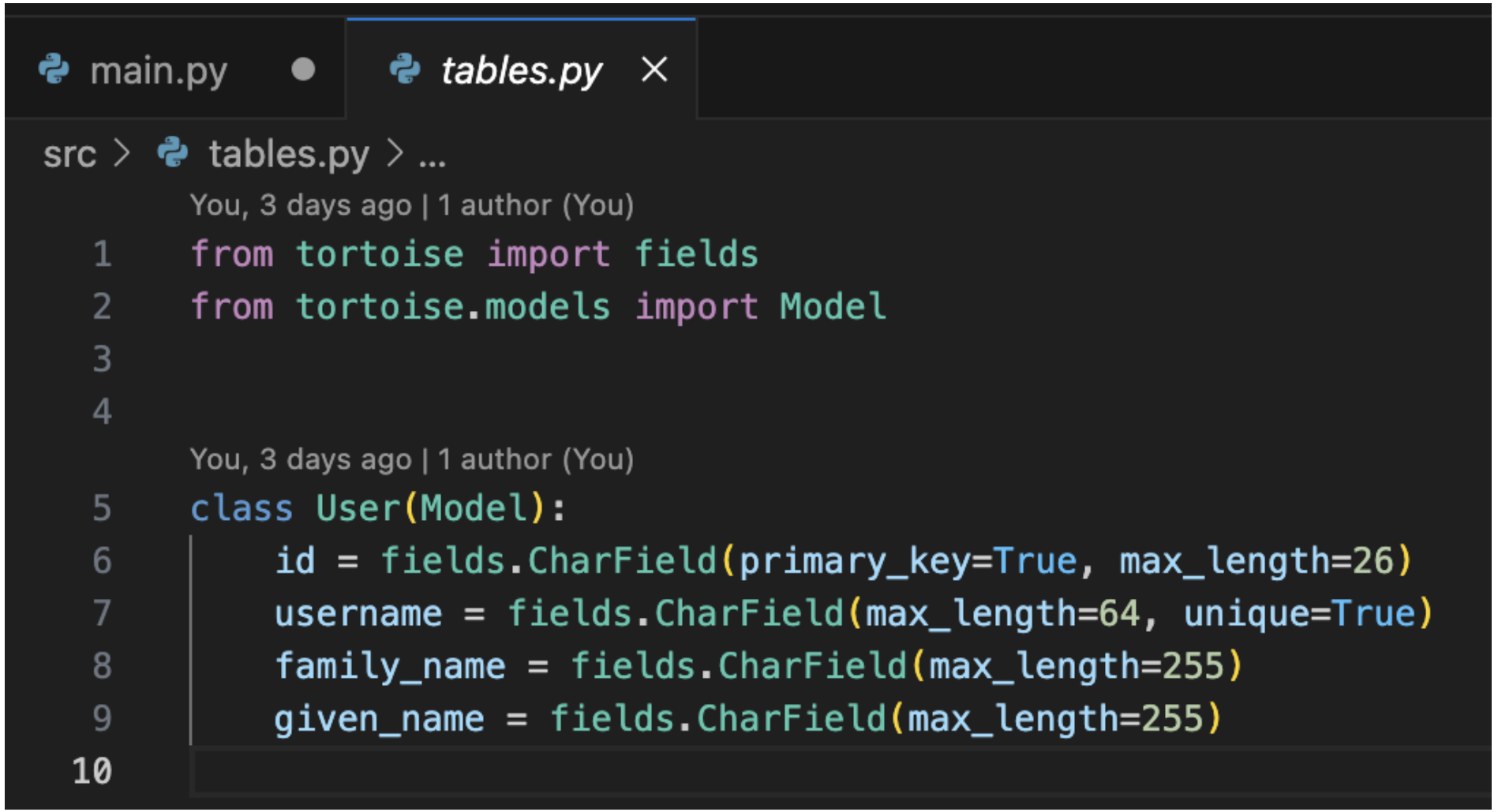
http://localhost:8000/docs



Let’s add a database! (SQLite)

```python
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



Create API: GET /v1/users

```python
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:

```python
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

Try it out

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	*No links*

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

```python
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 ag
    )
    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}

```python
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"
        )      You, 3 days ago • Develop FastAPI
        return V1UserResponseModel(
            id=user["id"],
            username=user["username"],
            family_name=user["family_name"],
            given_name=user["given_name"],
        )
    except DoesNotExist:
        raise NotFound404Exception
```

Documenting 404 Not Found response

NotFound404.py

src > responses > NotFound404.py > ...

```python
# You, 3 days ago | 1 author (You)
from fastapi import HTTPException
from pydantic import BaseModel


# You, 3 days ago | 1 author (You)
class NotFound404(BaseModel):
    detail: str


not_found_404_response = {
    "model": NotFound404,
    "description": "Resource not found",
    "content": {"application/json": {"example": {"detail": "Resource not found"}}},
}


# You, 3 days ago | 1 author (You)
class NotFound404Exception(HTTPException):
    def __init__(self):
        super().__init__(status_code=404, detail="Resource not found")

```

POST /v1/users

```python
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}

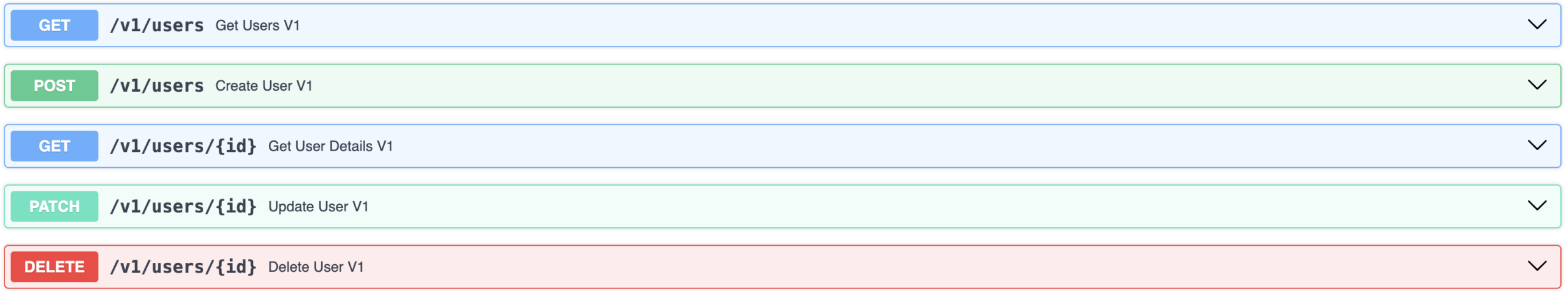
```python
# PATCH /users/{id} - Update certain fields of a user by ID, return True if successful
@router.patch("/users/{id}", responses={400: not_found_404_response})
async def update_user_v1(id: str, body: V1UserCreateAndUpdateRequestModel) -> str:
    try:
        user = await User.get(id=id)
    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

```python
# 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!

http://localhost:8000/docs



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)

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

You, 22 minutes ago | 1 author (You)

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

```python
# 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

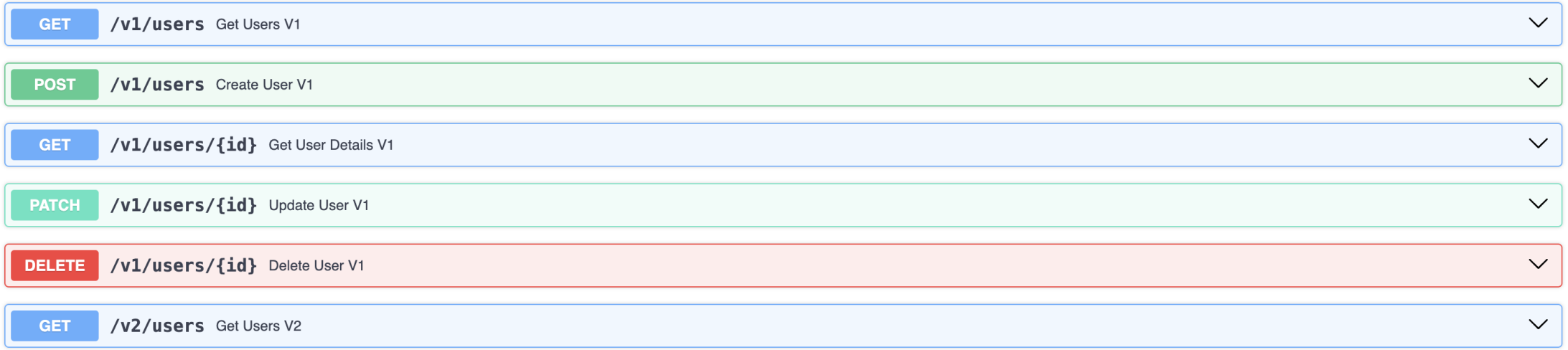
Name	Description
cursor string (query)	The last user ID from the previous page cursor

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



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

Docs FAQ

25K

Search /

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

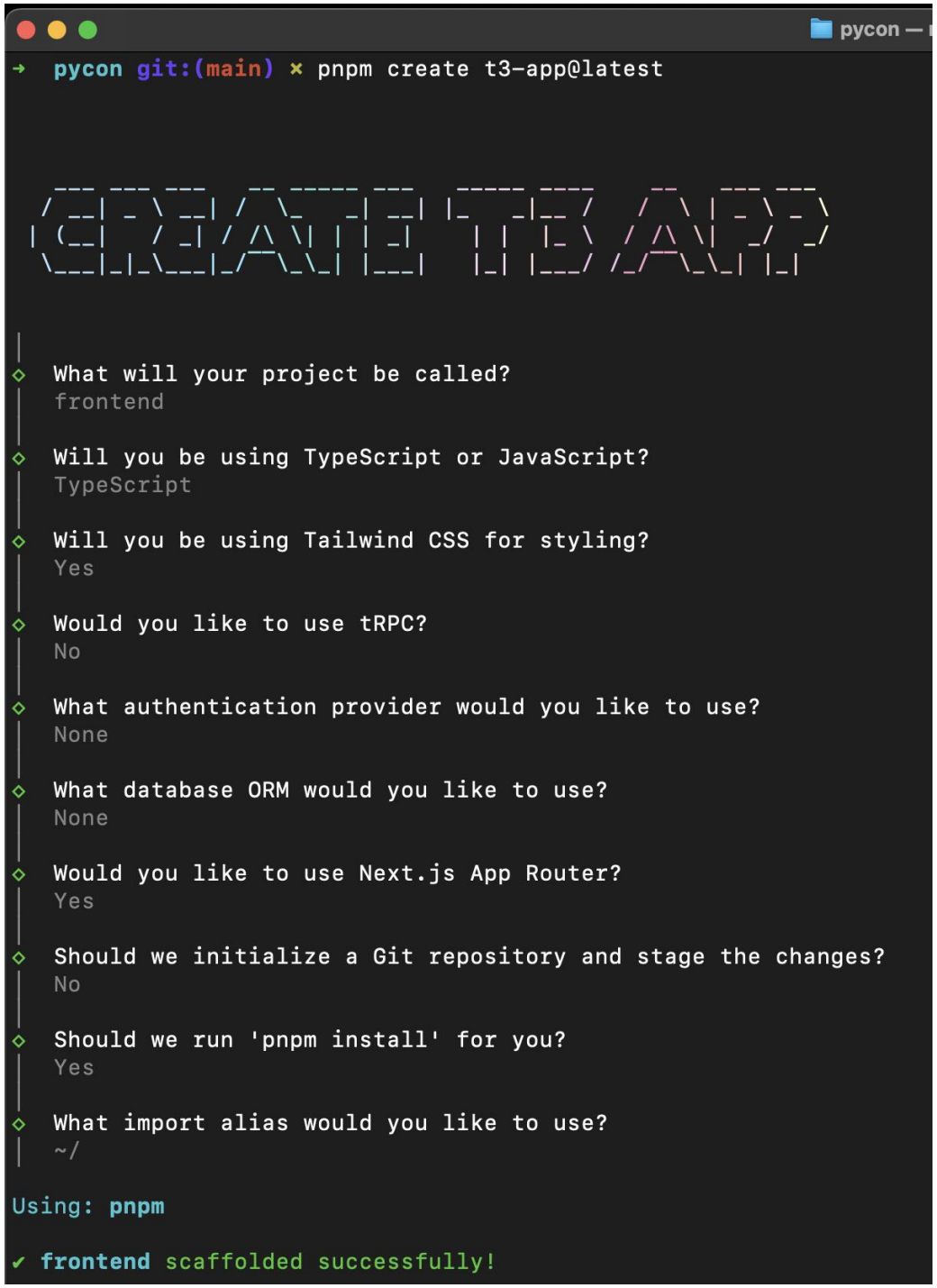
Documentation >

GitHub

```
npm create t3-app@latest
```

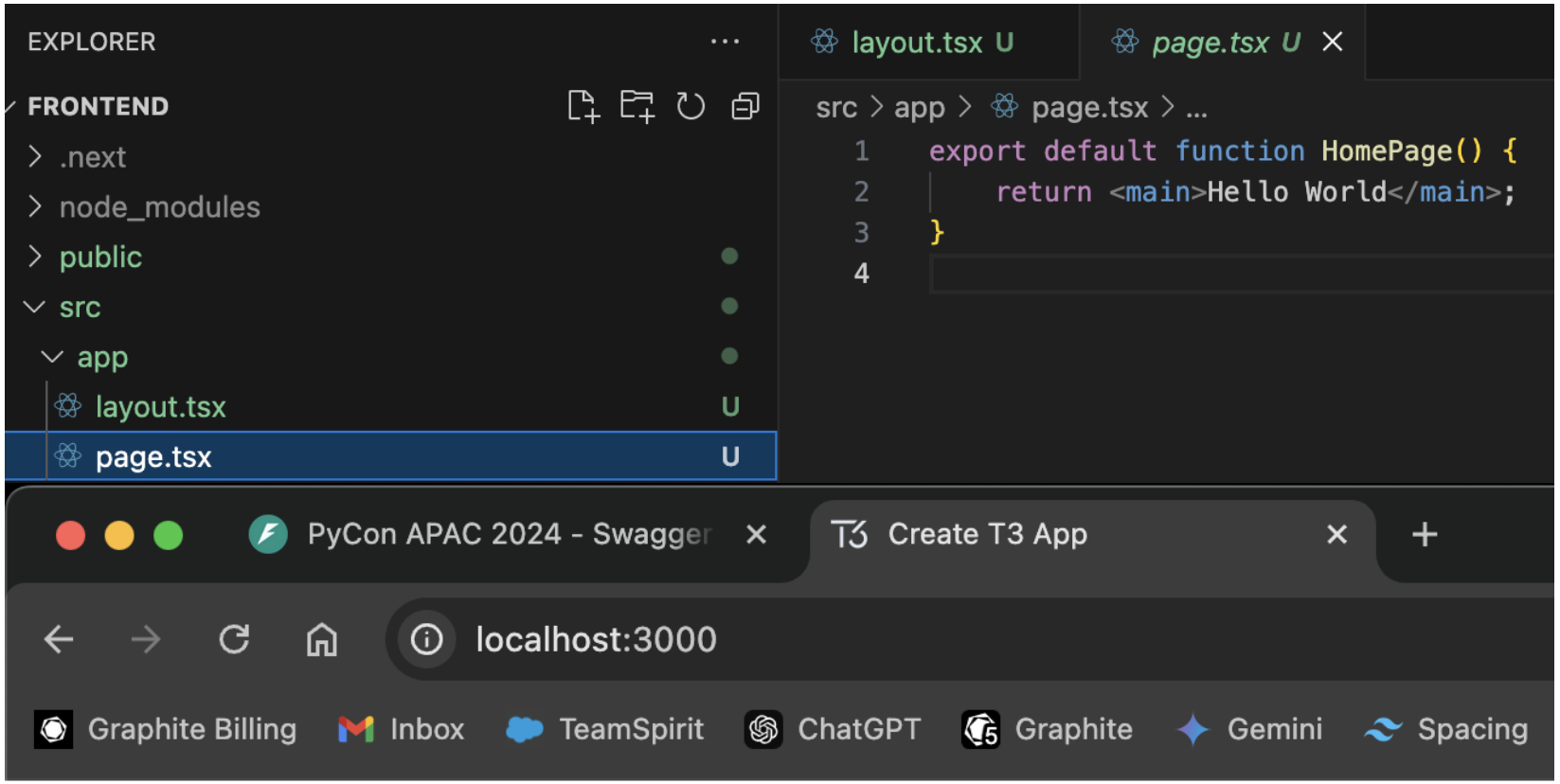
Create T3 App

- pnpm create t3-app@latest



File-Based Routing: Hello World

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

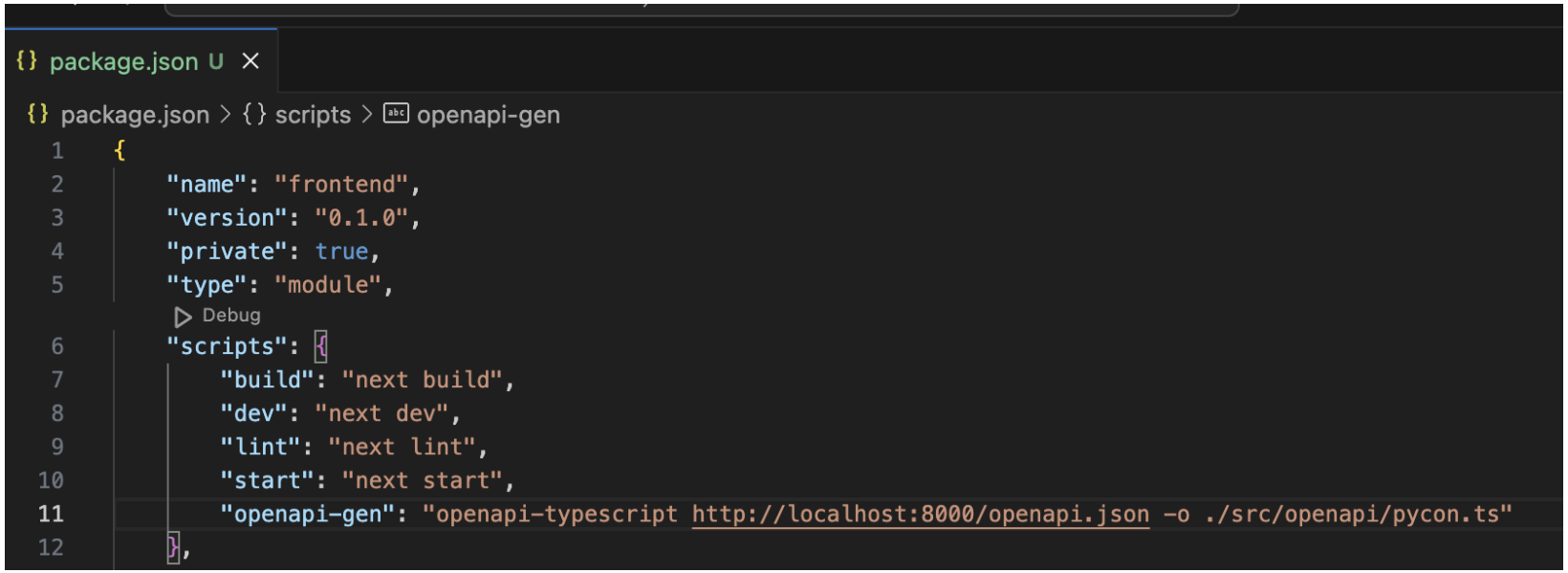


Hello World

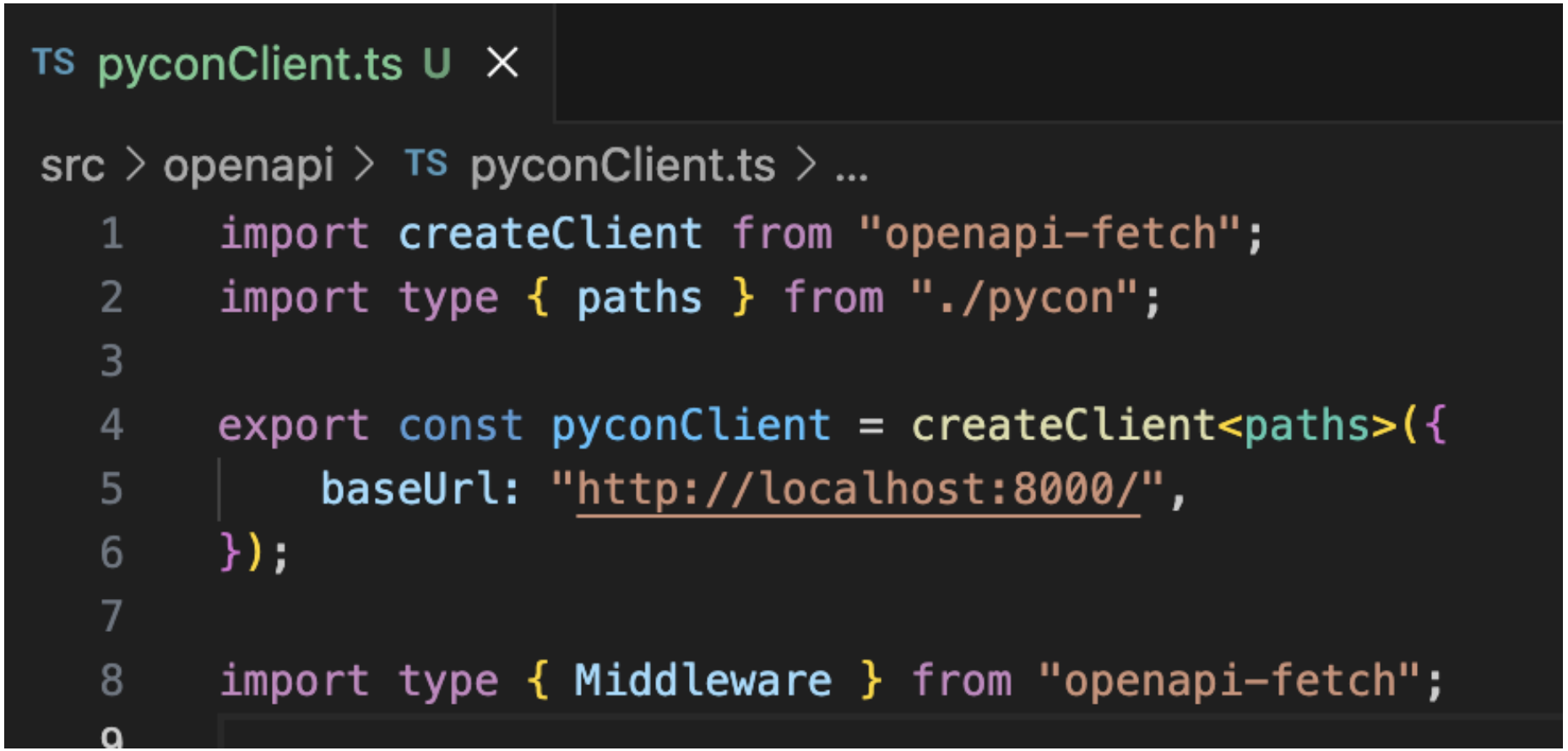
How to communicate with the backend?

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



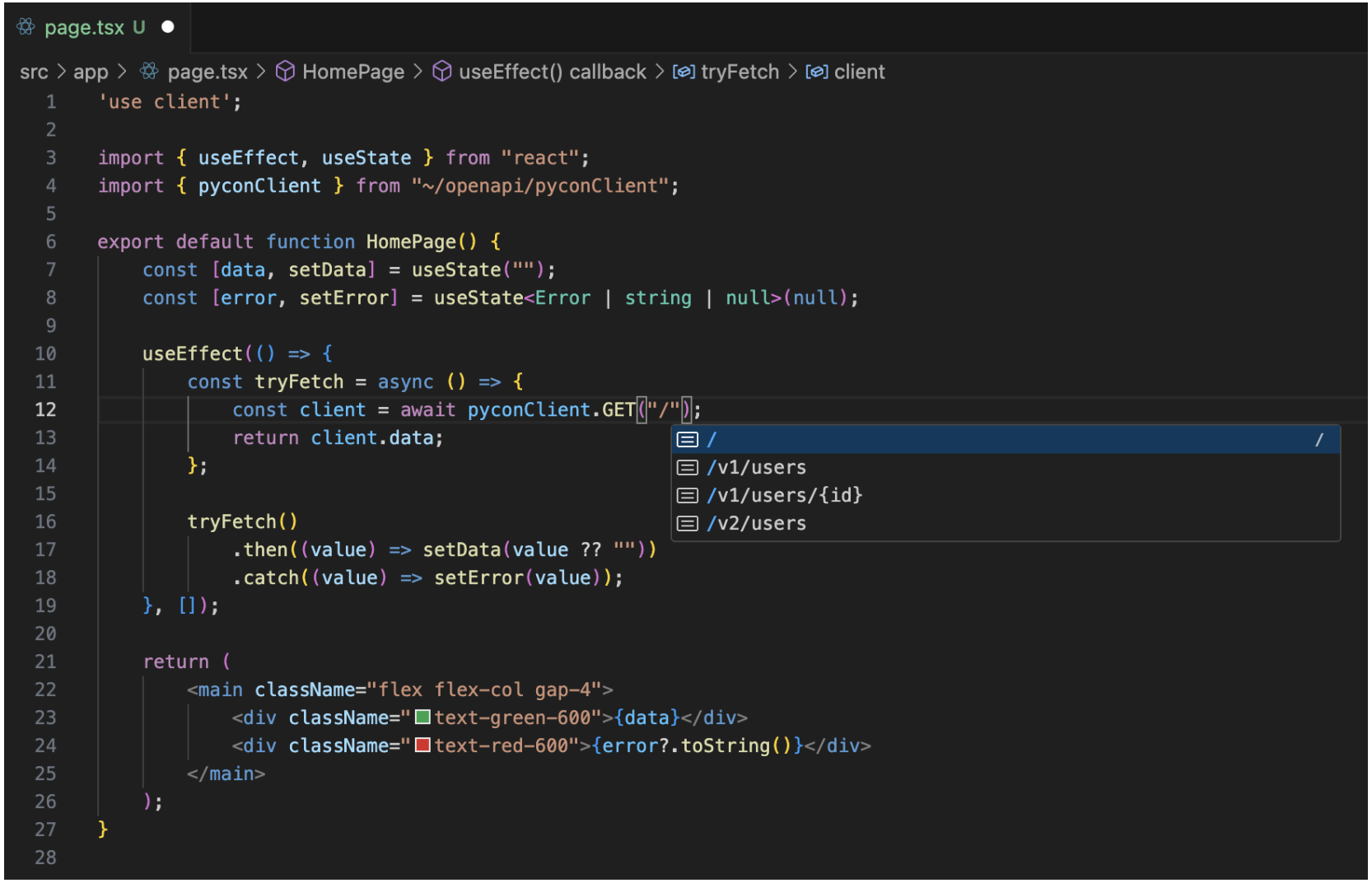
Export a TypeScript OpenAPI Fetch Client



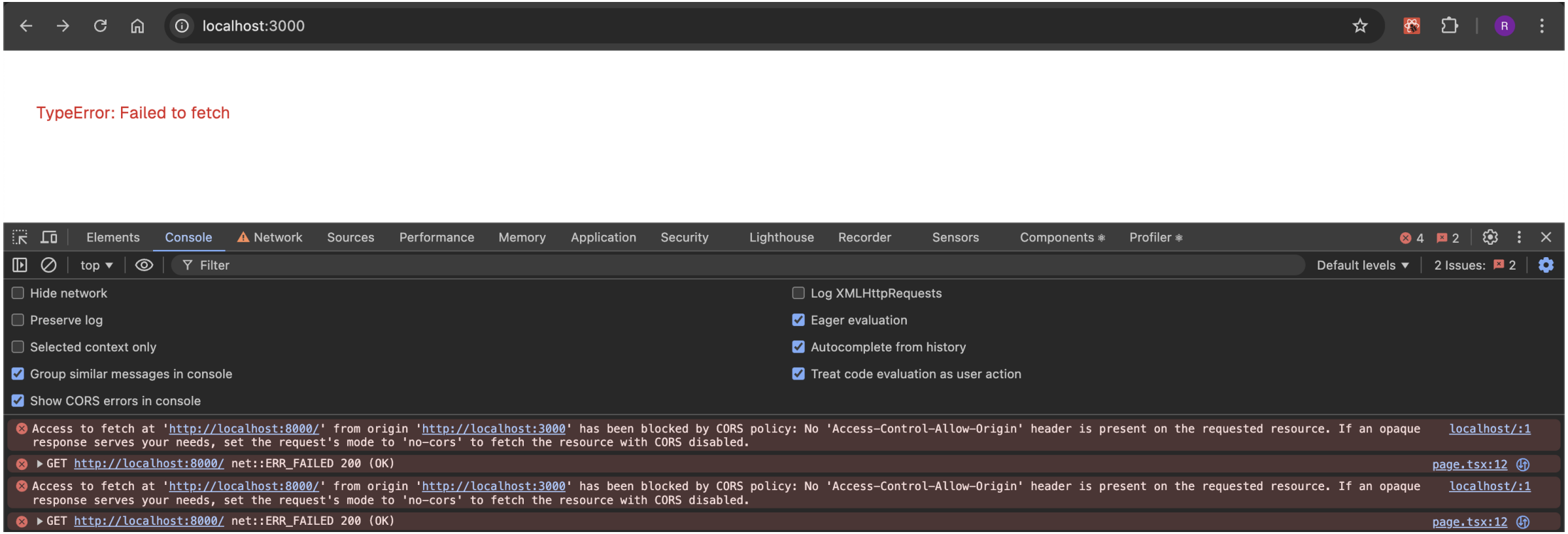
Modify middleware to throw on error

```ts
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 throwOnErrorMiddleware: 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(throwOnErrorMiddleware);
```

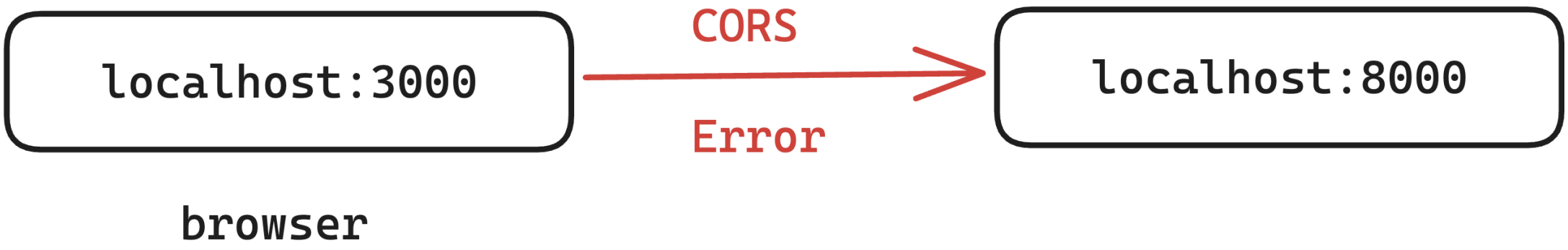
Let’s use typed API client! ☺

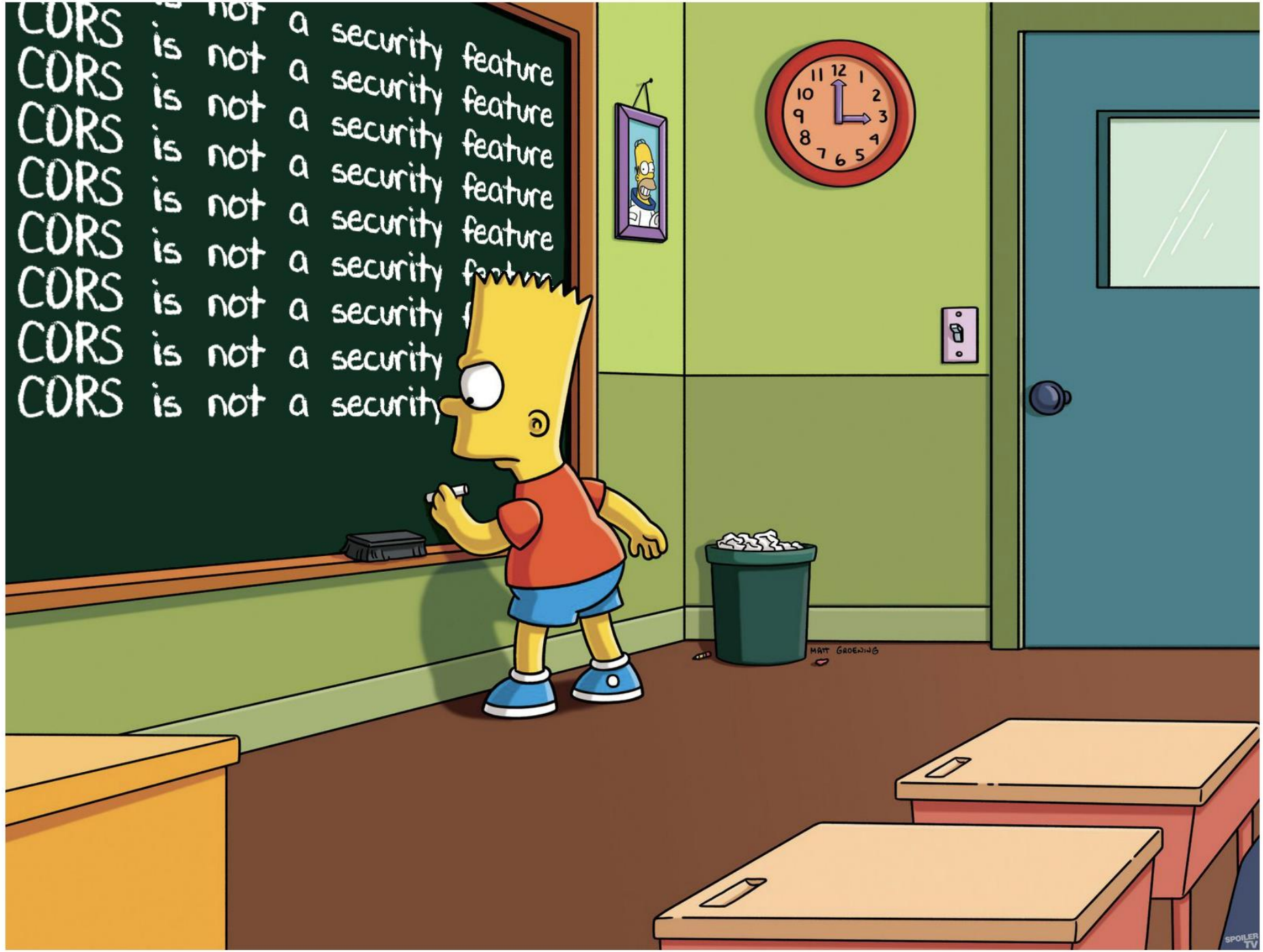


CORS error! ☹


localhost:3000
TypeError: Failed to fetch
Elements
Console
Network
Sources
Performance
Memory
Application
Security
Lighthouse
Recorder
Sensors
Components
Profiler
top
Filter
Default levels
2 Issues: 2
Hide network
Preserve log
Selected context only
Group similar messages in console
Show CORS errors in console
Log XMLHttpRequests
Eager evaluation
Autocomplete from history
Treat code evaluation as user action
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.
localhost/:1
GET http://localhost:8000/ net::ERR_FAILED 200 (OK)
page.tsx:12
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.
localhost/:1
GET http://localhost:8000/ net::ERR_FAILED 200 (OK)
page.tsx:12

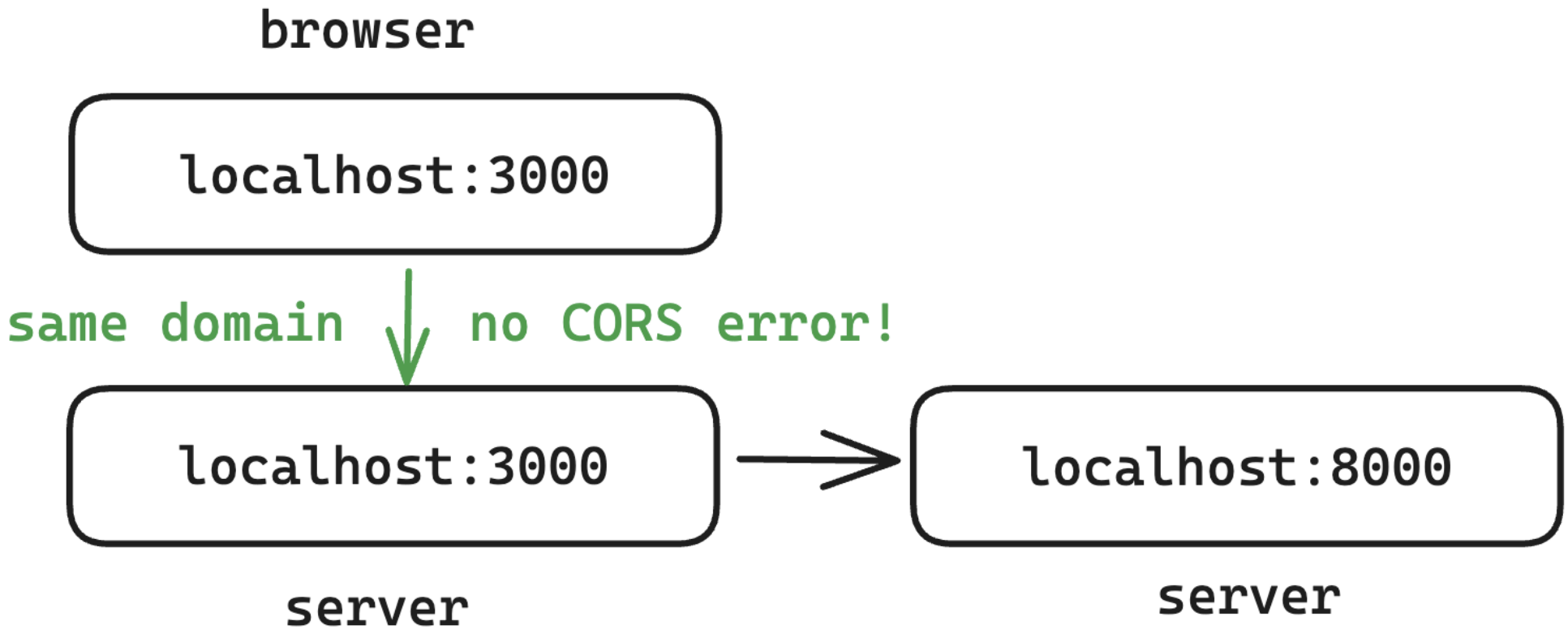
What happened? What is CORS?



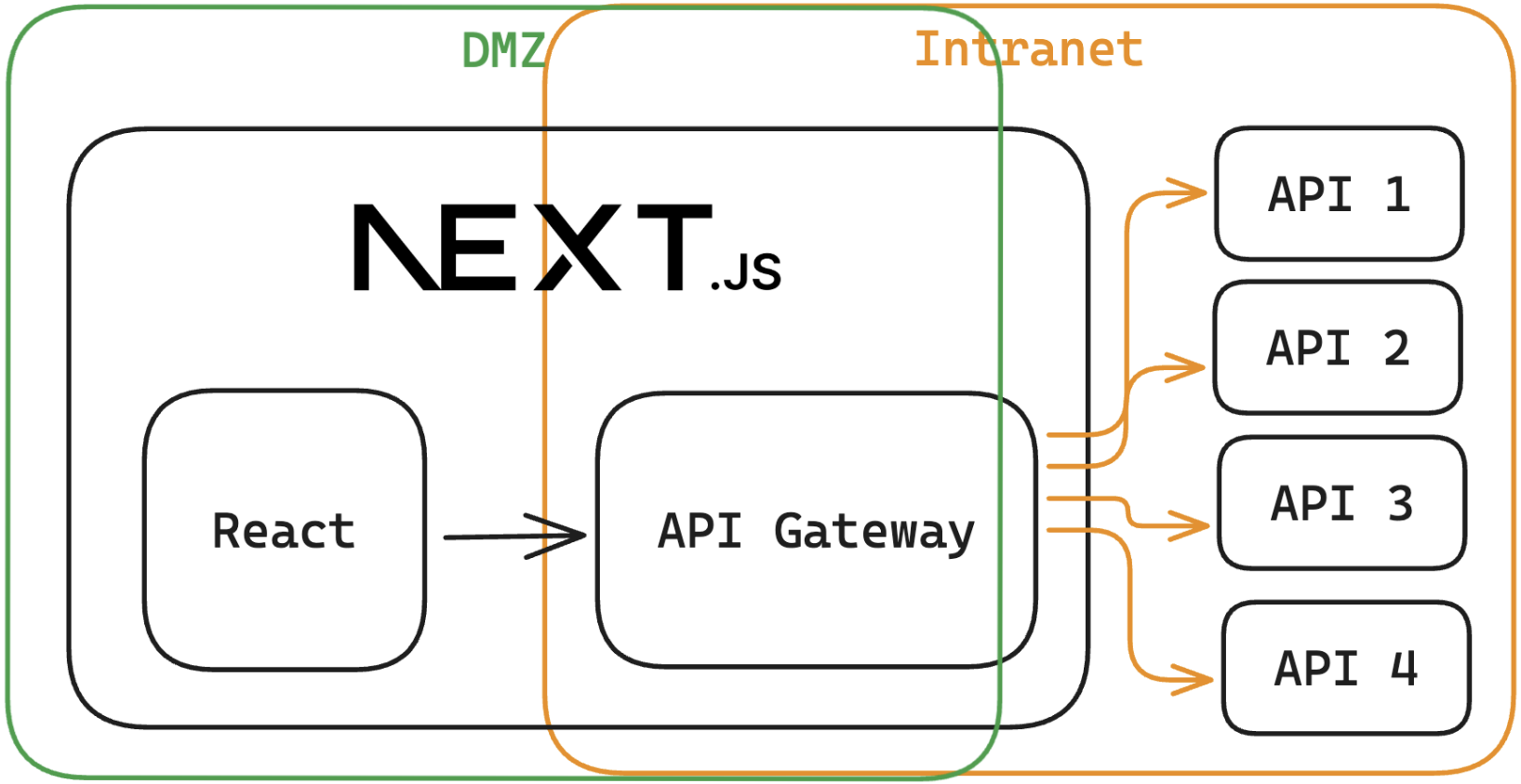

CORS is not a security feature
CORS is not a security feature
CORS is not a security feature
CORS is not a security feature
CORS is not a security feature
CORS is not a security
CORS is not a security
CORS is not a security
MATT GROENING
SPOILER TV

Proxy Path to Backend

Next.js Server as an API Gateway

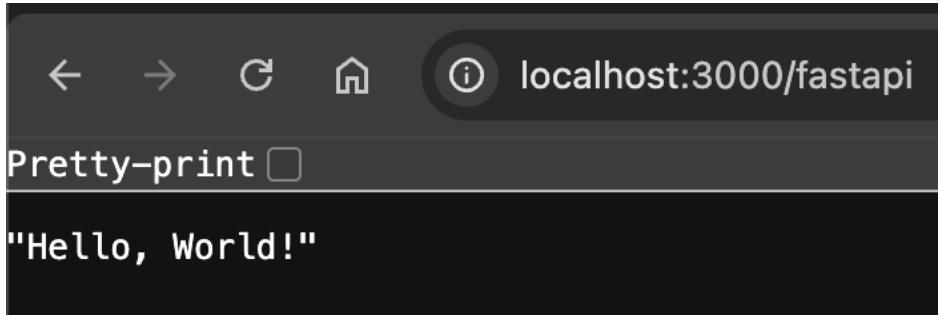


Next.js for Microservices API Management



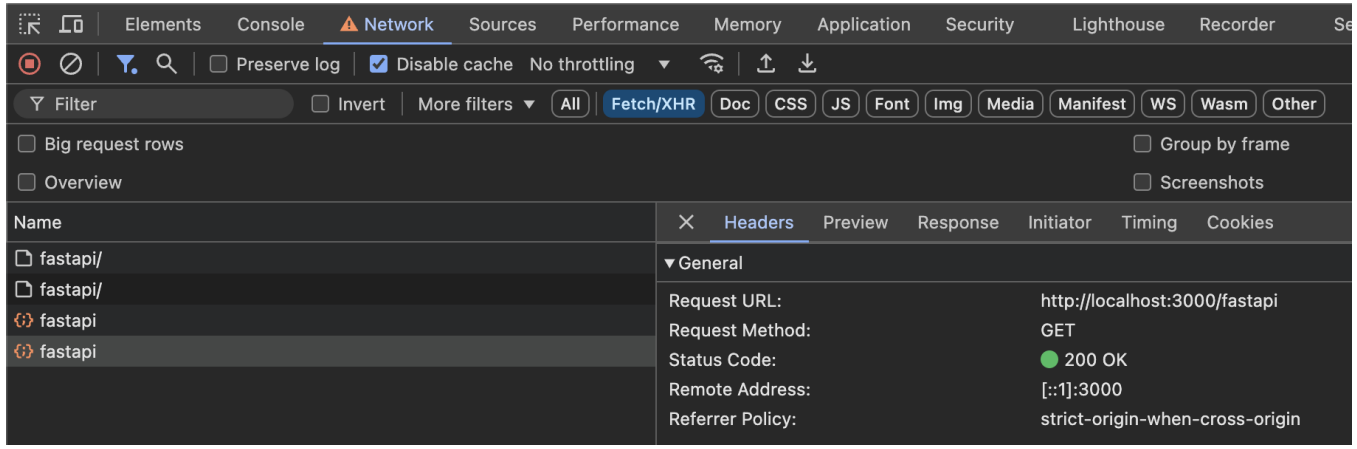
In Kubernetes, Next.js can be LoadBalancer / NodePort, and your microservices as ClusterIP!

Proxy to Backend (port is 3000, not 8000)



Hello, World!

```
src > openapi > TS pyconClient.ts > 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
```



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

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

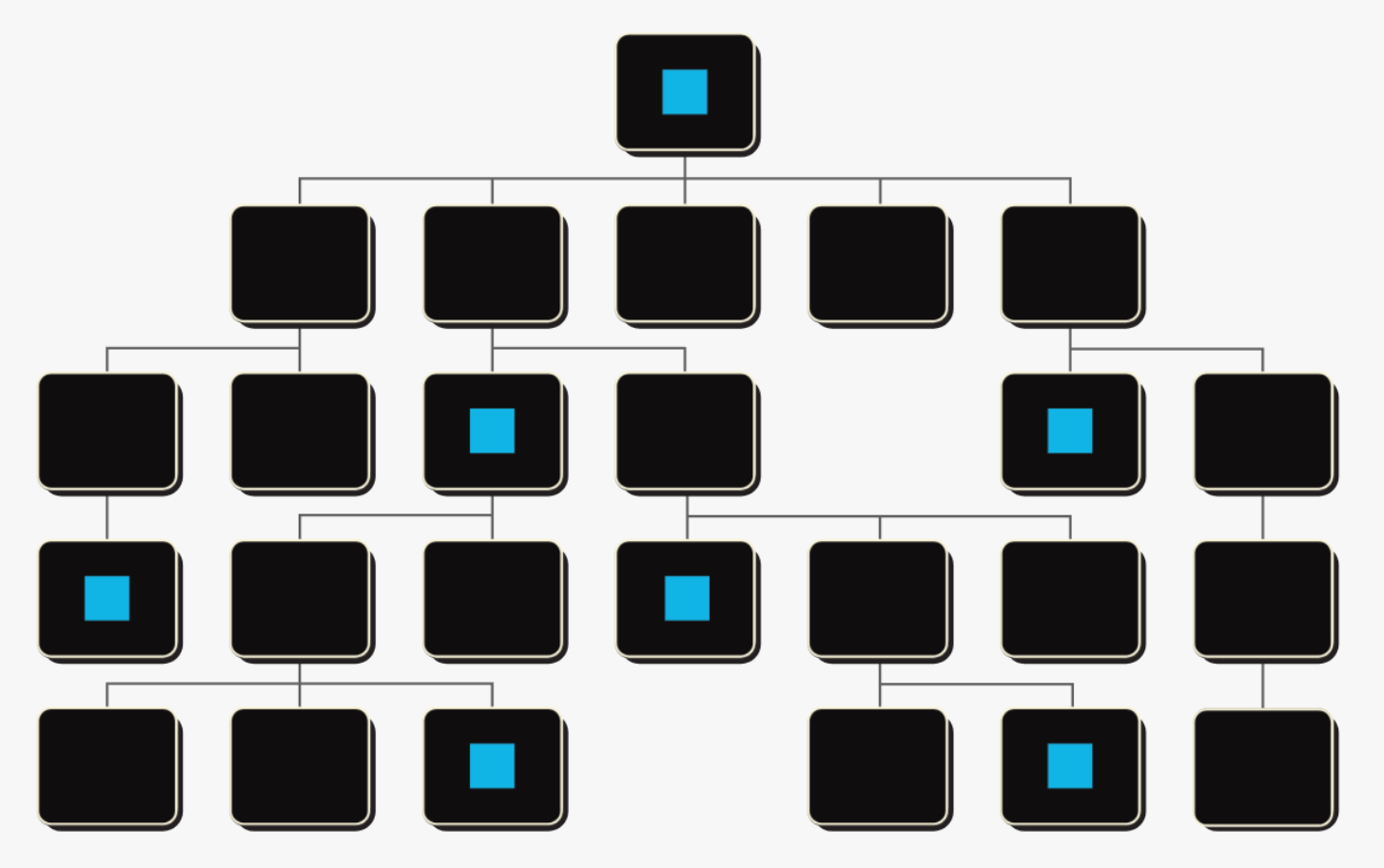
- Adding isLoading state to prevent one of the two deadliest UX sins: Cumulative Layout Shift
- What’s wrong with this code?
  - Fetch call is asynchronous
  - Meaning, it may not finish immediately
  - But if the server re-renders the UI…
  - The client makes second fetch call!
  - There’s no guarantee 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 **predictibility** 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

```tsx
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>
    );
}
```

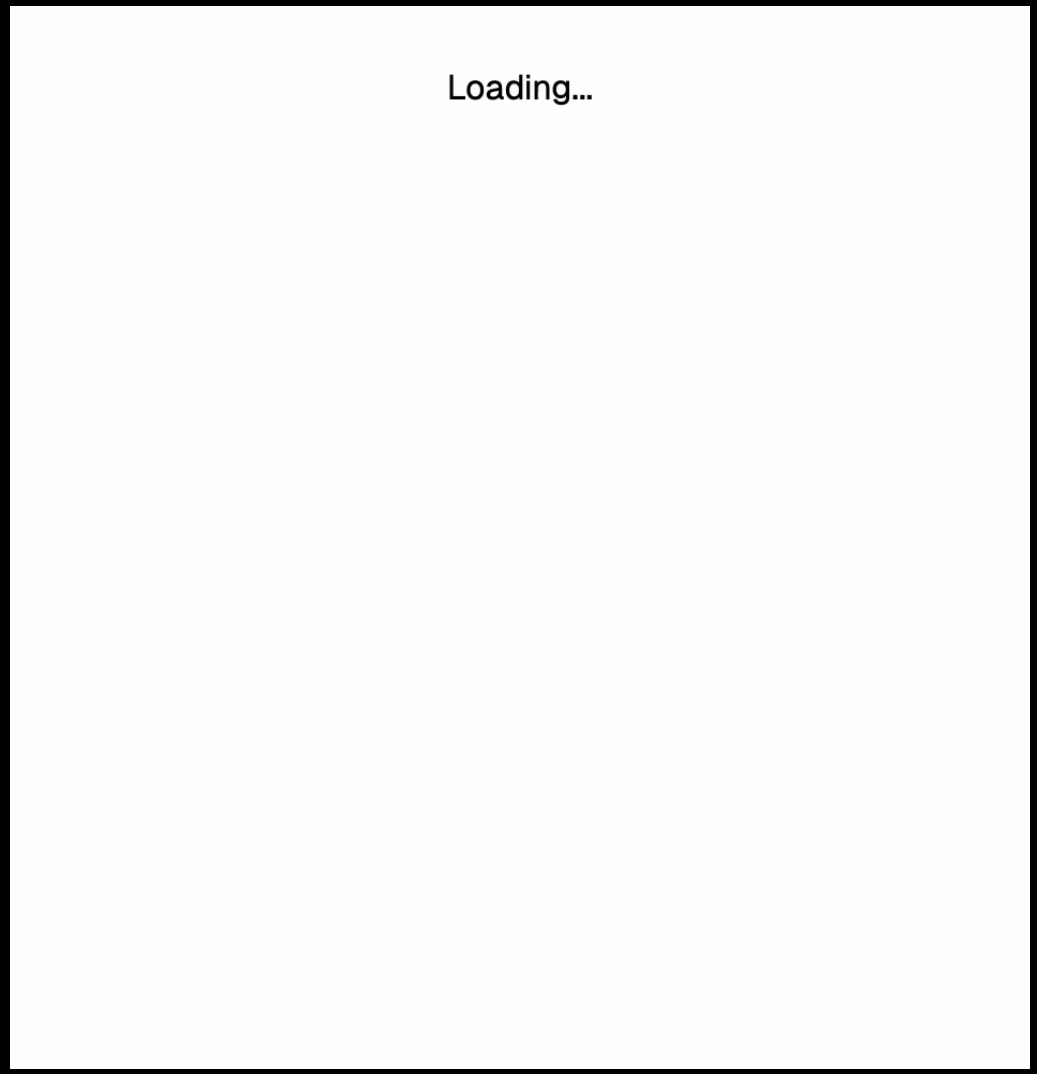
When does stale query get refetched?

- 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:



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)

```tsx
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>
);
```

```tsx
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/