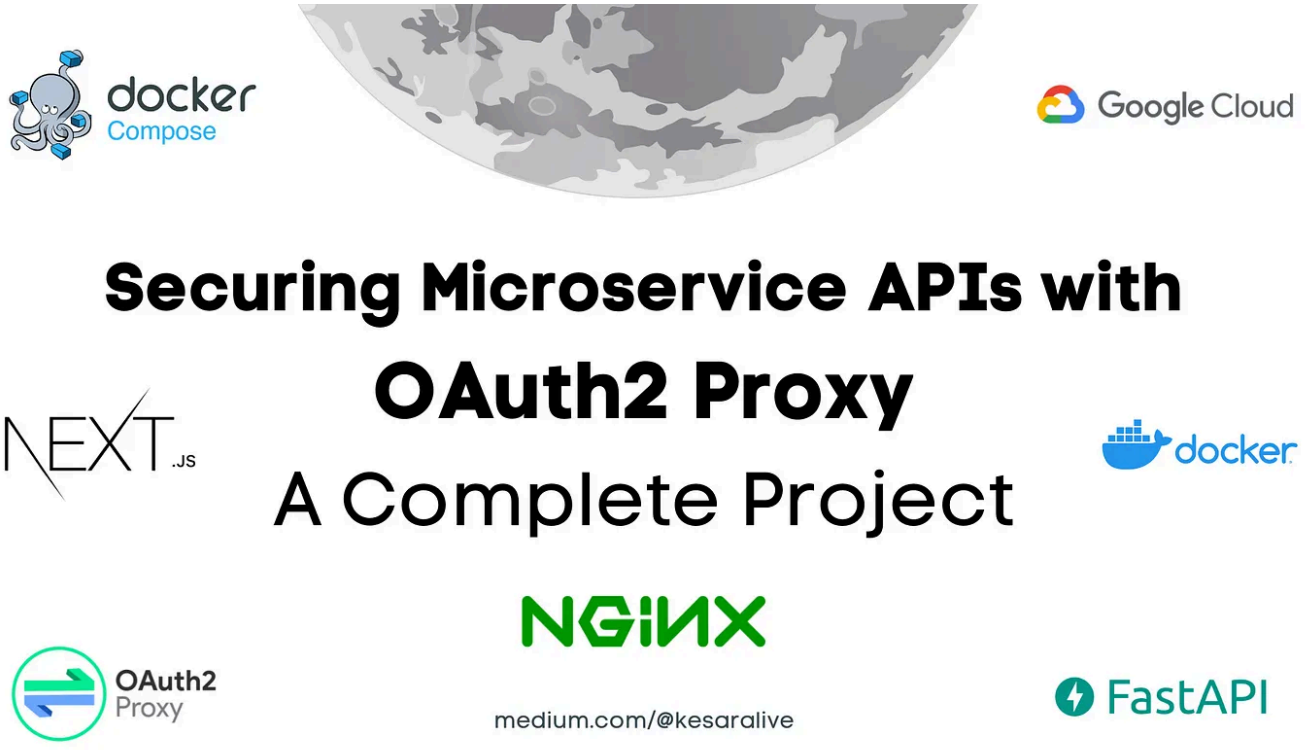
Securing Microservice APIs with OAuth2 Proxy: A Complete Project

Kesara Karannagoda · Follow

Published in DevOps.dev · 25 min read · Apr 16, 2024



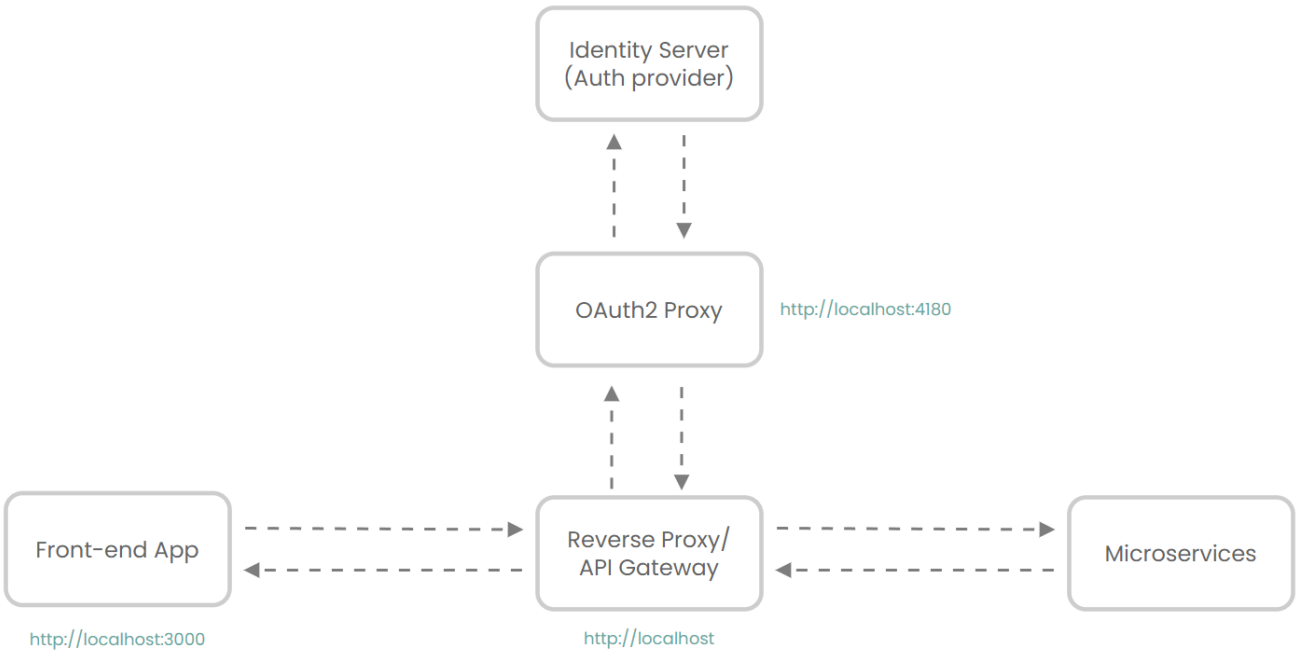
Overview

In this hands-on project, we will discuss how to build & secure microservice APIs using OAuth2 Proxy behind a reverse proxy. We will implement a microservice API, a frontend application, a reverse proxy, and OAuth2 Proxy. This setup will add an additional layer of security to our microservices, ensuring that only authenticated and authorized users can access our APIs.

Introduction

We're going to implement the following architecture on your local machine to learn how to accomplish this task.

> ***Note:*** *If you have a strong understanding of APIs and have your own service ready, you can skip the first two parts of this project and proceed directly to the third part.*



Project Architecture

As I mentioned in the overview section,

1. First, we'll establish a microservice API using FastAPI to test this architecture. (I'll provide a pre-coded solution for you.)
2. Next, we'll configure a frontend application using Next.js to interact with our FastAPI service.
3. Following that, we'll implement a reverse proxy between the frontend app and the microservice we've created using Nginx.
4. Lastly, we'll set up OAuth2 Proxy to secure the FastAPI service we've developed.

The Goal

> The goal of this project is to understand the workings of a modern, secure, and scalable web application architecture and to gain a thorough understanding of the OAuth2 authentication flow.

Prerequisites

A good internet connection.

You need to have docker installed in your computer. If not you can follow the official documentation to download docker.
https://docs.docker.com/engine/install/

It's nice to have a solid grasp of Docker and Docker networking, as I'll be providing you with a pre-coded API service and a frontend application to work with. However, it's **not mandatory**, as I'll explain what would be necessary for this project.

If you need any help while working on this project, you can refer to my Docker networking project to understand concepts like ‘Host Network’ and ‘Bridge Network’. Feel free to leave a comment below if you have any questions.

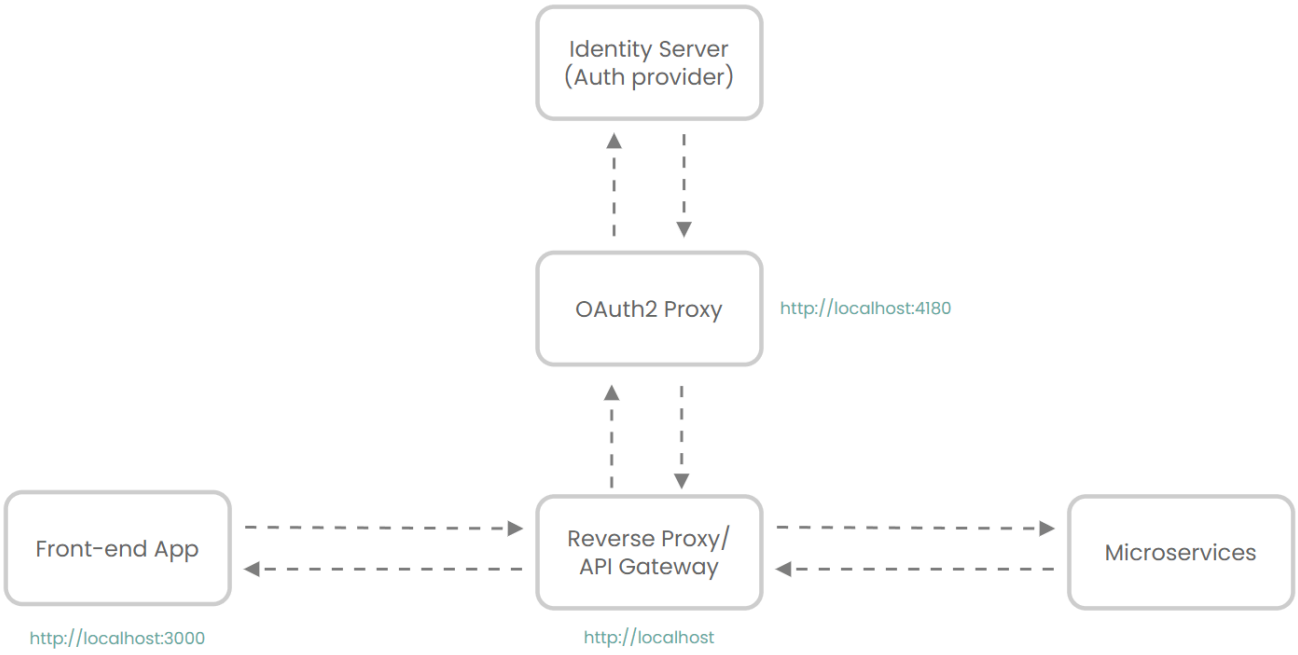
Docker networking: https://medium.com/@kesaralive/diving-deeper-into-docker-networking-with-docker-compose-737e3b8a3c8c

With these in place, you should be ready to start working on the project.

Before Starting,

This will be a long project with many steps to complete. If you find anything difficult to understand, please leave a comment in the article. I’ll do my best to help you out. If you get stuck at any point, don’t hesitate to reach out to me. I’m here to help. ✌️

Project Architecture



Project Architecture

OAuth2 Proxy

OAuth2 Proxy is a reverse proxy and static file server that provides authentication using OAuth2. You'll configure OAuth2 Proxy to secure access to your FastAPI service. This will add an extra layer of security by requiring users to authenticate before accessing the API.

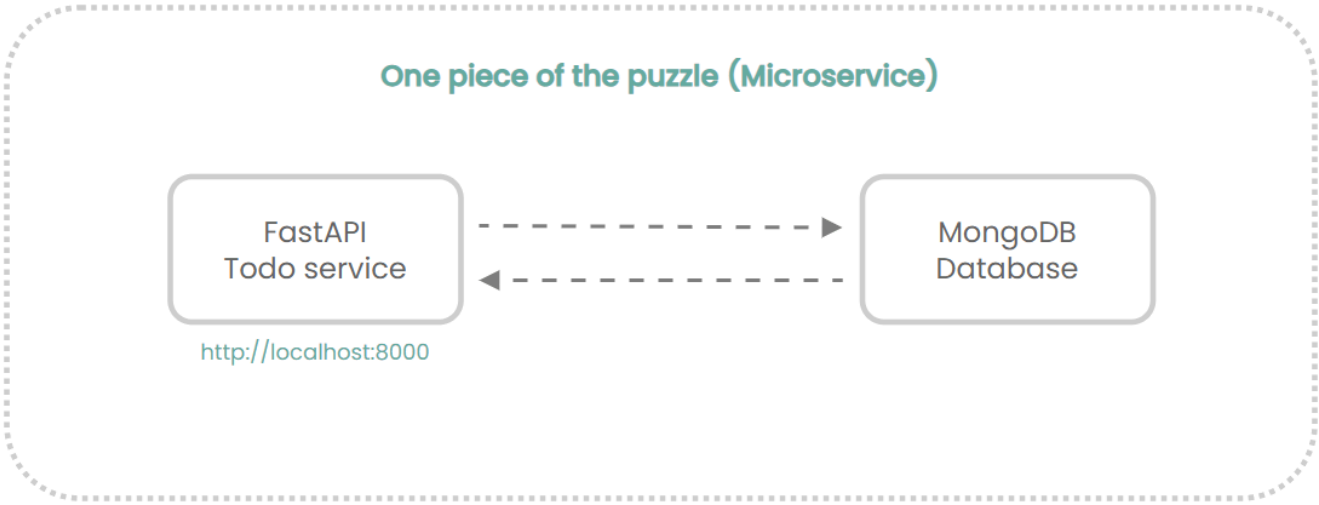
Identity Server

You'll configure OAuth2 Proxy to use Google OAuth 2.0 as the Auth Provider. This means that when a user tries to access your FastAPI service, they will be redirected to Google's authentication service to log in. Once authenticated, Google will issue an access token that allows the user to access your API.

Microservices

Microservices are small, autonomous services that work together. They're like individual pieces of a puzzle that work together to create a bigger picture. Each piece (Microservice) has its own job to do and can be

developed, deployed, and scaled independently, making the overall system more flexible and easier to manage.



A microservice (todo service)

Today, we're going to create one piece of a puzzle (a microservice) to understand how we can secure this puzzle using OAuth2 Proxy. We'll create a very simple Todo service API using FastAPI and MongoDB to represent a piece of this puzzle.

Reverse Proxy/ API Gateway

A reverse proxy/ API gateway acts as an intermediary between cleints and servers. It receives requests from clients and forwards them to appropriate backend servers. Reverse proxies can also perform additional functions like load balancing, caching, SSL termination, and request routing.

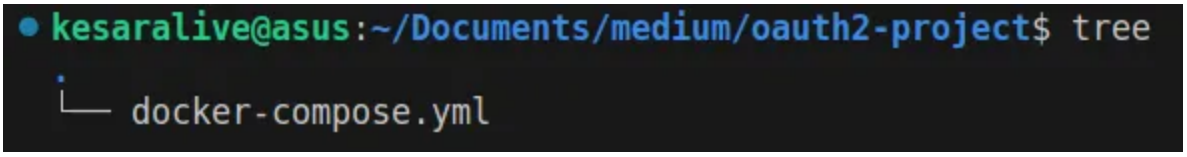
We're going to use Nginx as our reverse proxy/ API gateway for this project. Nginx is known for its high performance, stability, and low resource consumption. It's commonly used as a reverse proxy and API gateway due to its ability to handle a large number of concurrent connections efficiently.

Front-end App / Client

We'll use a Todo client application implemented using Next.js for this project. It's a simple one-page frontend application that can be used to create, read, delete, and update todos.

FastAPI Todo-service — Step 01

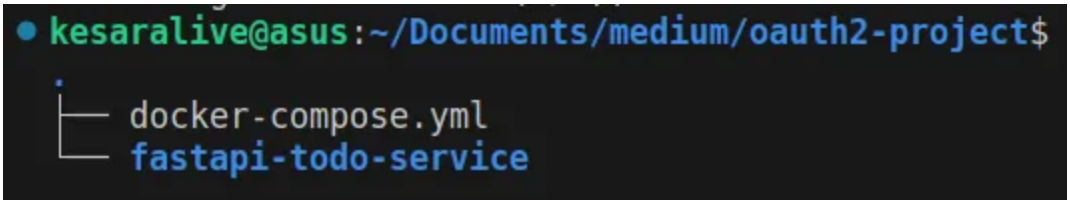
Create a new folder named `oauth2-project` and an `docker-compose.yml` file inside it. I'm going to use the name `oauth2-project` folder in the rest of the project. It's better if you can use the same name as mine.



after creating 'docker-compose.yml'

Then clone the **fastapi-todo-service** into that folder using following command:

```
git clone https://github.com/kesaralive-medium/fastapi-todo-service.git
```



After cloning the fastapi-todo-service

Then update the `docker-compose.yml` file with below code snippet.

```
version: "3.8"
services:
  mongodb:
    image: mongo:latest
    container_name: mongodb
    volumes:
      - mongodb_data:/data/db
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: password

  fastapi-todo-service:
    container_name: todo-api
    build:
      context: ./fastapi-todo-service
      dockerfile: Dockerfile
    depends_on:
      - mongodb
    ports:
      - 8000:8000
    environment:
      ATLAS_URI: mongodb://root:password@mongodb:27017
      DB_NAME: todoapi

volumes:
  mongodb_data:
    driver: local
```

In the above `docker-compose.yml` configuration, I've included a MongoDB container to provide the database service to our FastAPI service. If you have a MongoDB account and a free tier cluster, you can use your MongoDB Atlas URI and a preferred database name as you wish. (In that case you can remove the **mongodb** service from `docker-compose.yml` configuration.)

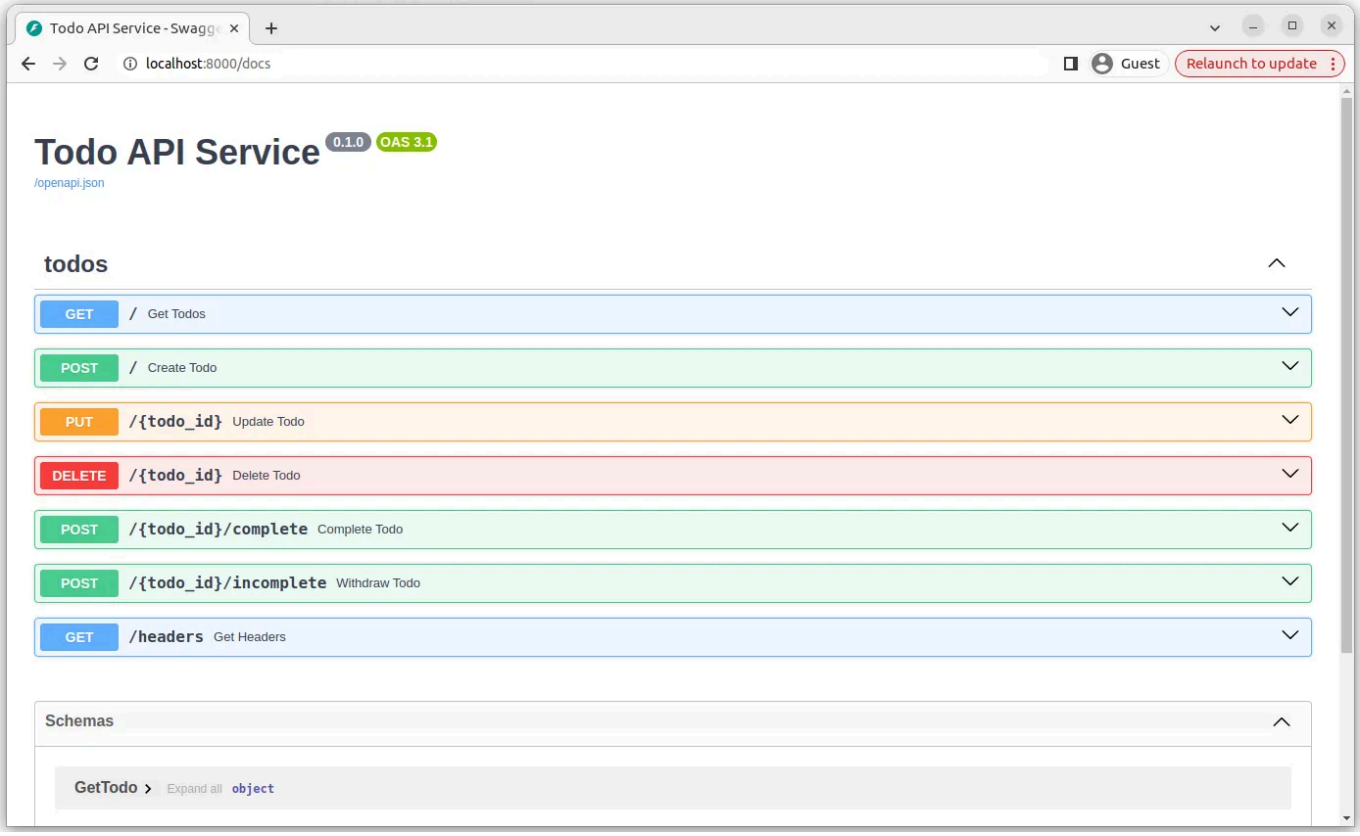
If you're going to use your mongodb cloud atlas uri `docker-compose.yml` will look like this,

```
version: "3.8"
services:
  fastapi-todo-service:
    container_name: todo-api
    build:
      context: ./fastapi-todo-service
      dockerfile: Dockerfile
    ports:
      - 8000:8000
    environment:
      ATLAS_URI: <your-mongodb-atlas-uri>
      DB_NAME: todoapi
```

Once you've selected your way, you can run the following command to up and run the todo-service.

```
docker compose up -d
```

If you've done everything correctly so far, you should be able to see the docs page of the FastAPI Todo service by accessing the http://localhost:8000/docs URL from your browser.

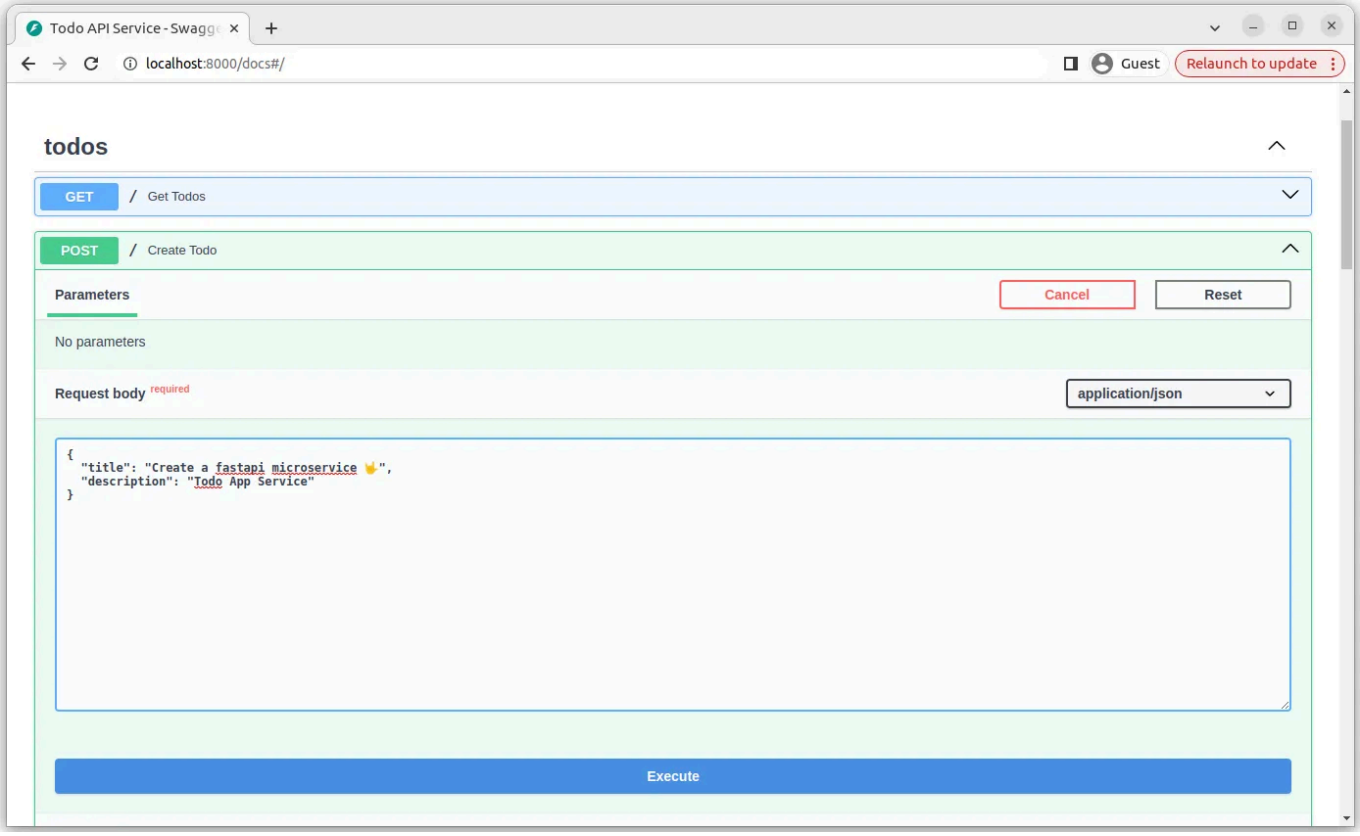


fastapi-todo-service

You will see 6 API endpoints related to the Todo service and one API endpoint to get the headers of our request. (This particular endpoint will help us understand the behavior of our Auth flow later.)

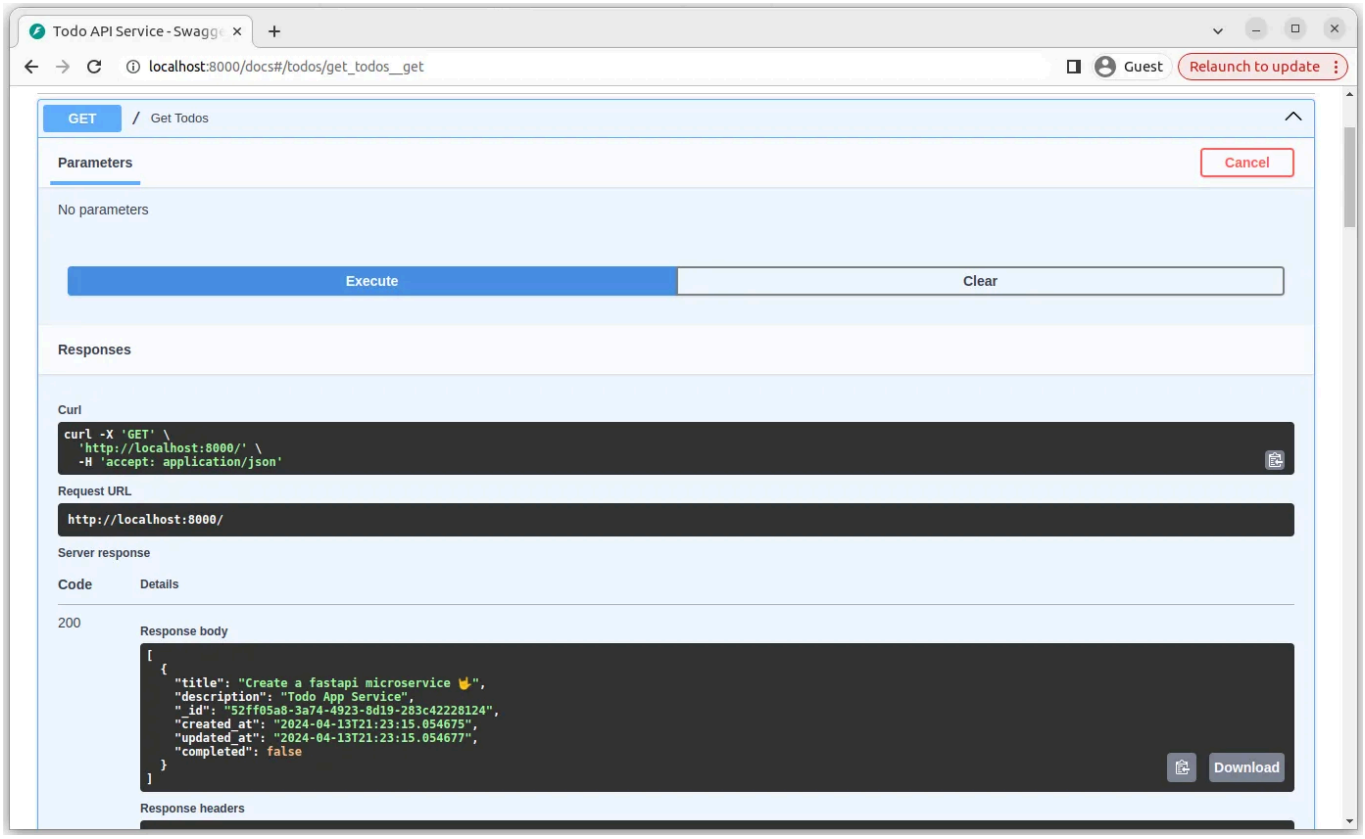
You can try to create a todo using the `POST / Create Todo` API endpoint and retrieve it using the `GET / Get Todos` API endpoint to make sure everything is working fine.

You can simply press the "**Try it out**" button, enter your todo in the request body, and then **Execute** the request.



creating the todo request POST /

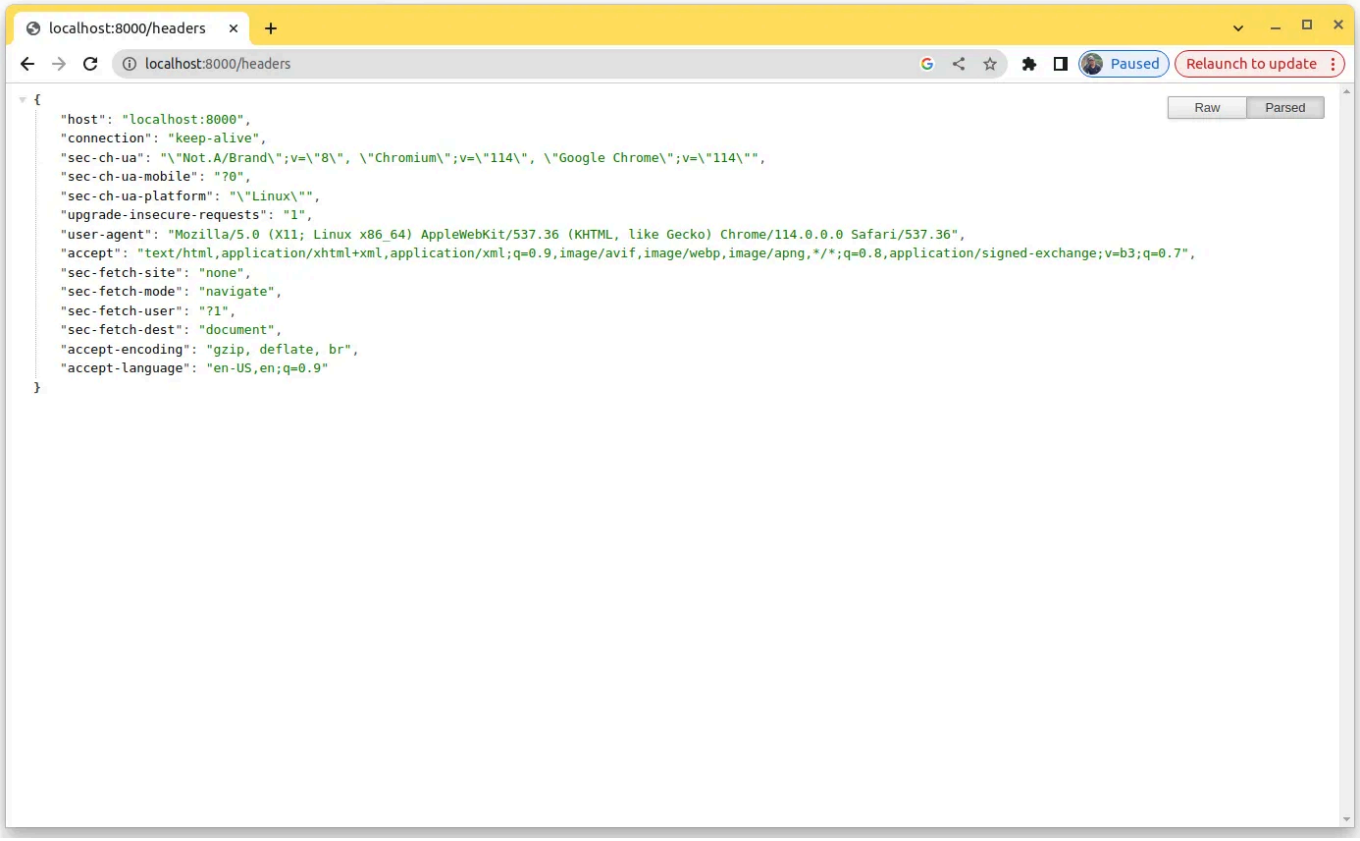
If you've set up your database URI correctly, you should be able to retrieve the created todo using the `GET / Get Todos` endpoint. The response body will be shown below after executing the request, as shown in the image below.



retrieving todos using GET /

Try to access the `/headers` endpoint from your browser.

http://localhost:8000/headers

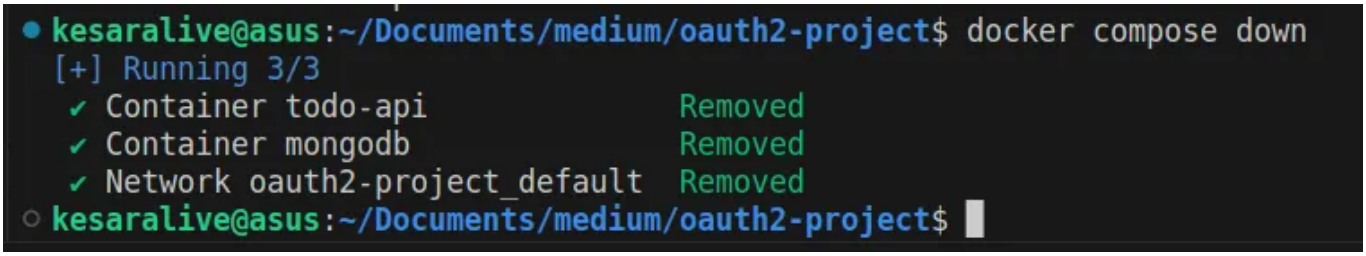


request headers at the fastapi-todo-service end.

Keep the above step in mind; we're going to compare these headers every time we integrate something into this project.

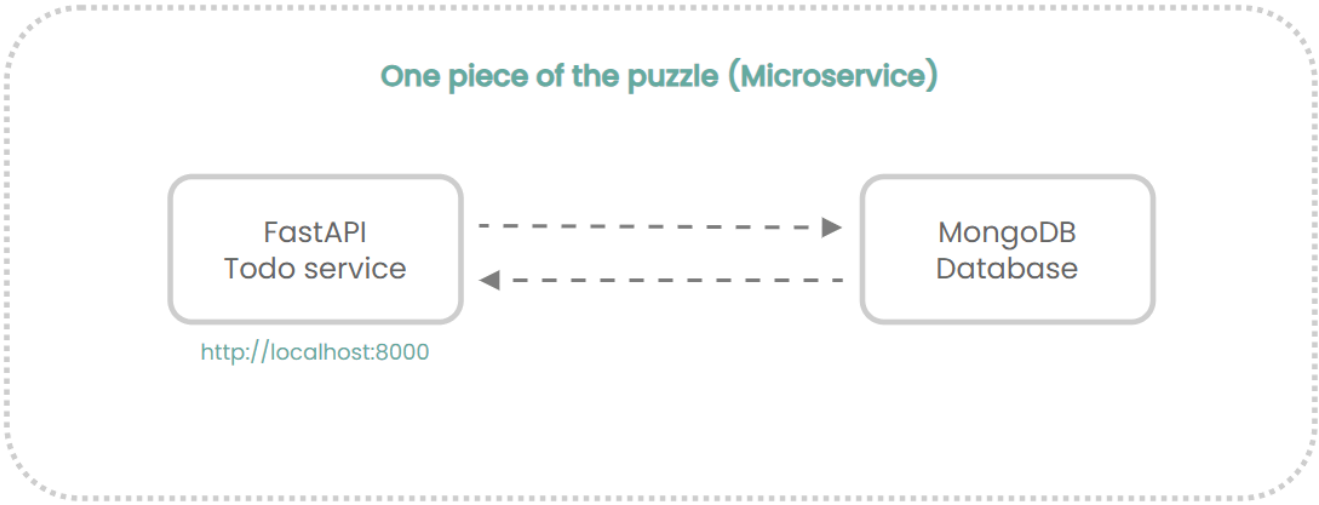
You can use the following command to stop the running containers.

```
docker compose down
```



docker compose down

If you have followed the instructions correctly, you have completed the following part of our project architecture.



fastapi-todo-service (Microservice)

Before proceeding to the next step, I'll share some important details about Docker Compose with you:

- **`docker compose up`** — Use this command to up and run all the containers defined in the `docker-compose.yml` file.
- **`docker compose up -d`** — This command starts the containers in the background (detached mode).
- **`docker compose down`** — Stops and removes all containers, networks, and volumes created by `docker compose up`.
- **`dockse compose build <SERVICE>`** —Builds or rebuilds a specific service defined in the `docker-compose.yml` file. Replace `<SERVICE>` with the name of the service you want to build. **If you've done some changes in any service after using `docker compose up`, you must run this command to reflect those changes in the new build.**

> You should run all of the above commands only inside the project directory.

Some helpful docker-compose.yml specification attributes details

In the `docker-compose.yml` file of this project, there are three main sections, and under the `services` section, there will be around 4-5 services. Here are some important attributes you should understand:

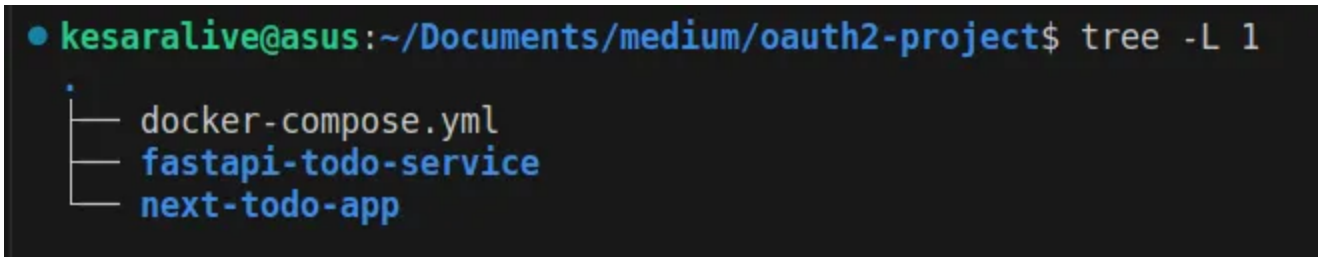
- **`build`** : Specifies how to build the Docker image for the service.
- **`depends_on`** : Specifies that the service depends on another service or services. However, it's important to note that this does not wait for the other service to be "ready" before starting the current service. It simply starts the other service before the current service.
- **`environment`** : This attribute is used to set environment variables required by the service.

Front-end App/ Client — Step 02

Now, let's integrate the front-end application into this setup by cloning the `next-todo-app` repository into the project folder using the following command.

```
git clone https://github.com/kesaralive-medium/next-todo-app.git
```

After cloning the `next-todo-app`, your directory should resemble the image below.



After cloning the next-todo-app

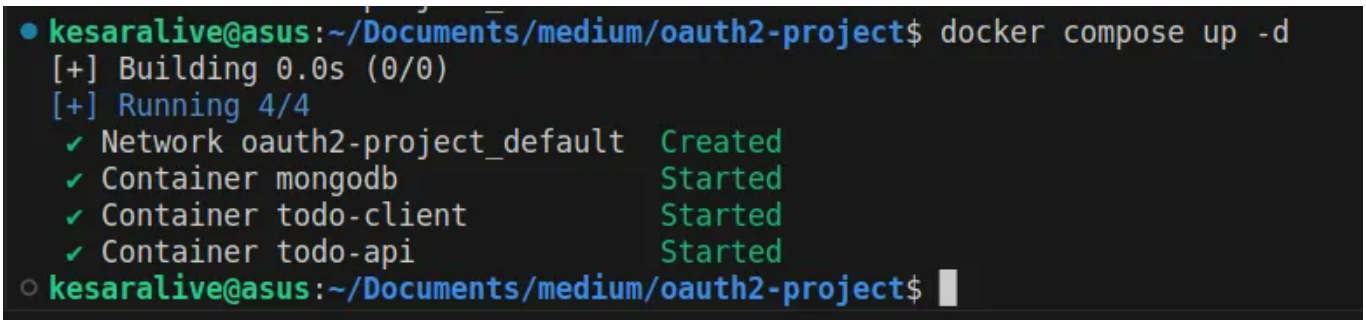
Now, let's add the frontend configuration to the `docker-compose.yml` file.

```
version: "3.8"
services:
  mongodb:
    image: mongo:latest
    container_name: mongodb
    volumes:
      - mongodb_data:/data/db
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: password

  fastapi-todo-service:
    container_name: todo-api
    build:
      context: ./fastapi-todo-service
      dockerfile: Dockerfile
    depends_on:
      - mongodb
    ports:
      - 8000:8000
    environment:
      ATLAS_URI: mongodb://root:password@mongodb:27017
      DB_NAME: todoapi

  next-todo-app:
    container_name: todo-client
    build:
```

```
      context: ./next-todo-app
      dockerfile: Dockerfile
    ports:
      - 3000:3000


volumes:
  mongodb_data:
    driver: local
```

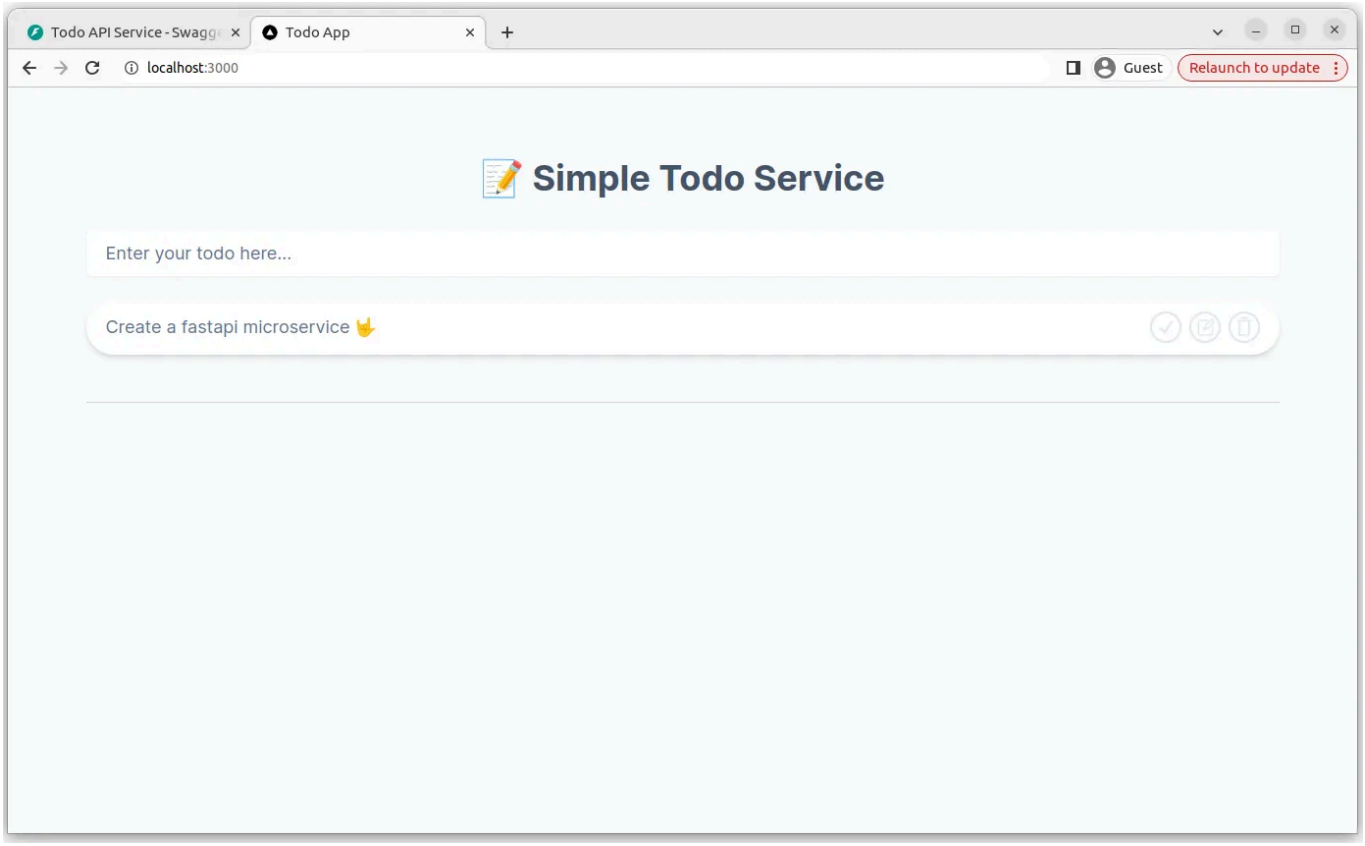
Again, run the following command to start and run all three containers.

```
docker compose up -d
```



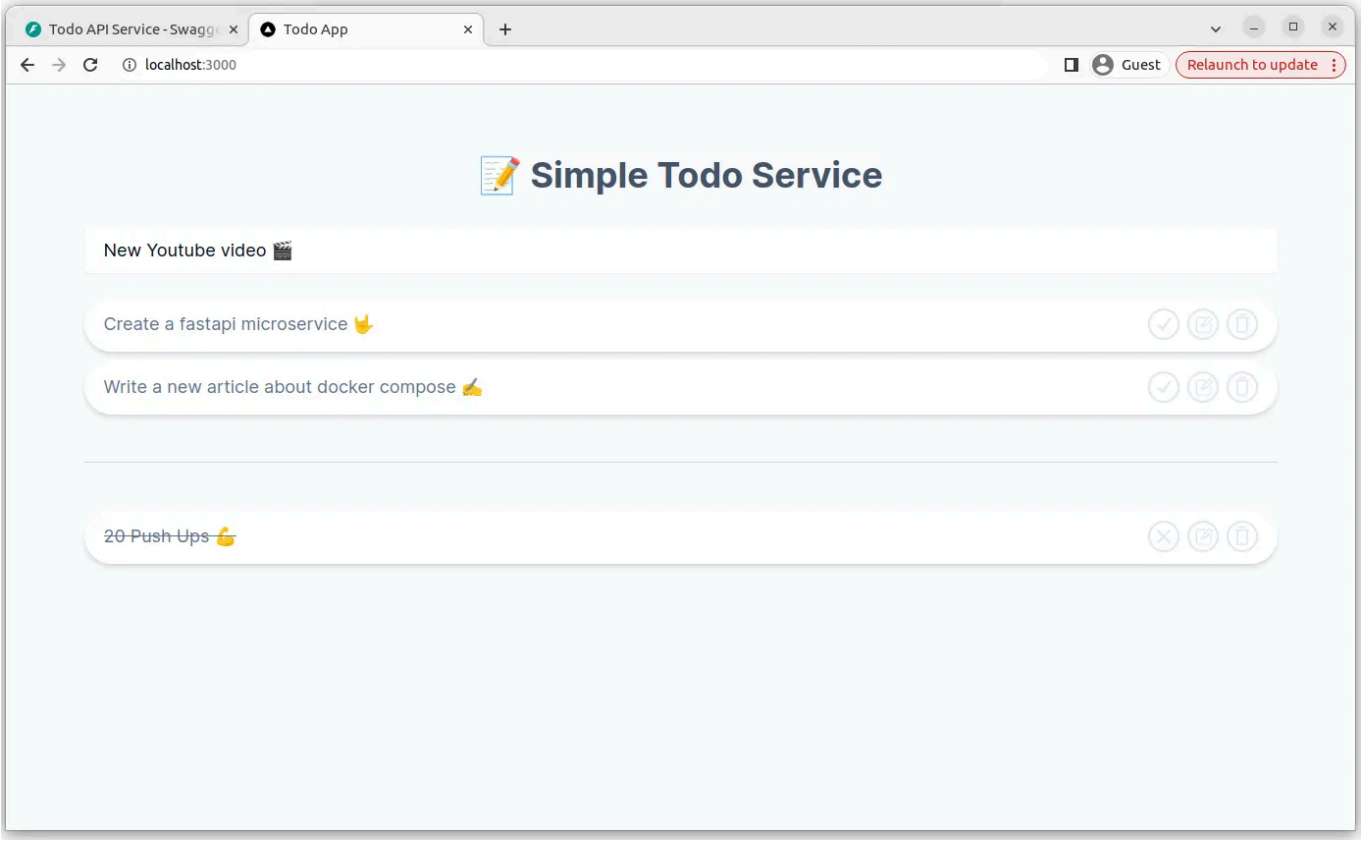
docker compose up -d (detached mode)

If you've integrated the todo-client correctly, you should be able to see the todo frontend application at http://localhost:3000.



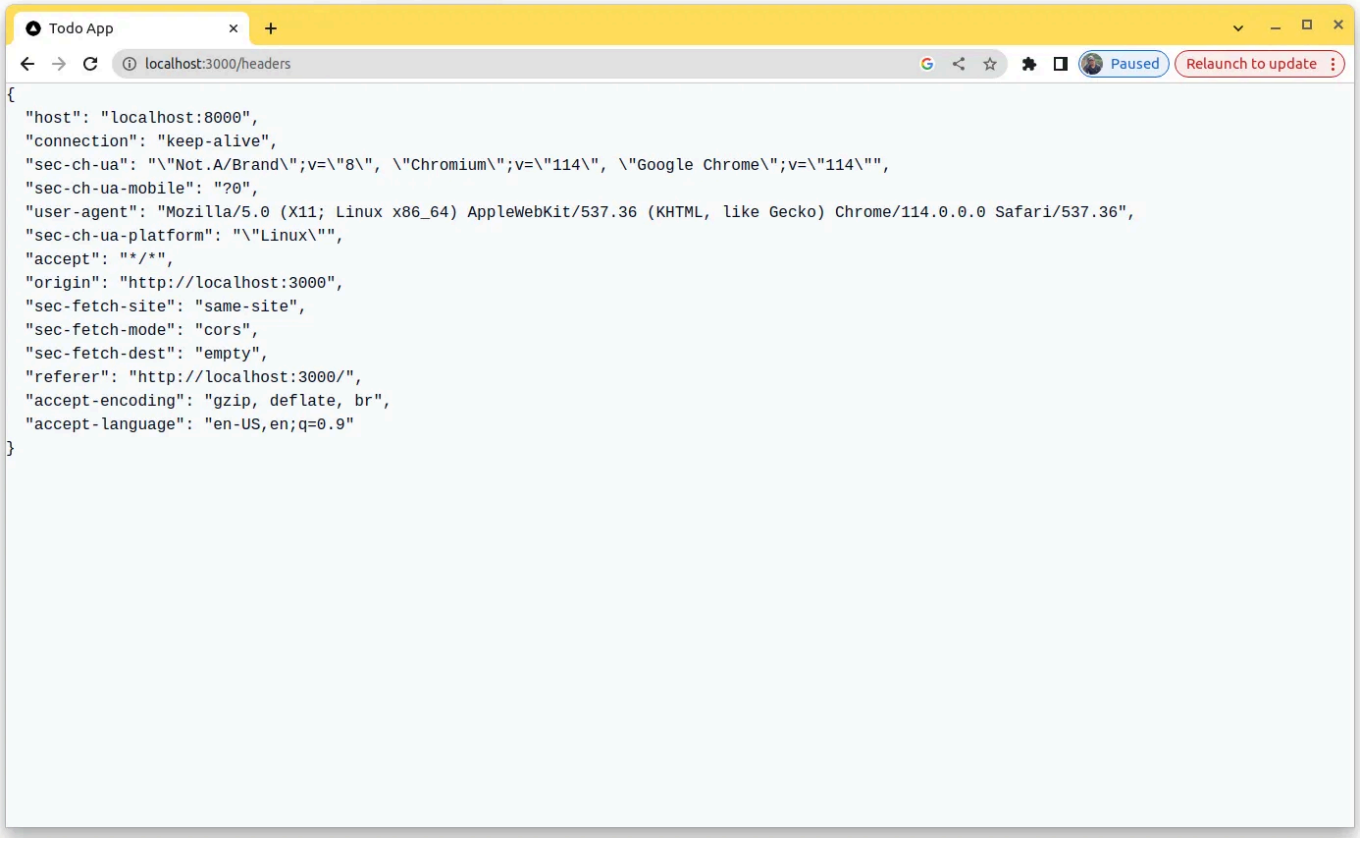
simple todo service frontend (next-todo-app)

You will also see the previously created todo in the todo client application. Try creating, updating, and deleting todos. (Press '**Enter**' to add todos after typing.)



simple todo service frontend (next-todo-app)

Now, try accessing the http://localhost:3000/headers page. You will notice that the headers output is different from when we accessed it from the FastAPI Todo service itself.



/headers from todo-client

You will see some newly added values and updated values there.

```
"accept": "*/*"
"origin": "http://localhost:3000"
"sec-fetch-site": "same-site"
"sec-fetch-mode": "cors"
"sec-fetch-dest": "empty"
"referer": "http://localhost:3000/"
```

The headers mentioned above are part of the CORS (Cross-Origin Resource Sharing) mechanism in web browsers. Since I've already granted access to any origin at the FastAPI Todo service end, our todo-client can access the todo-service using CORS. You'll be able to see the code regarding that in the `/fastapi-todo-service/app/main.py` file.

```
...

origins = [
    "*"
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

...
```

- **“accept”: “*/*”** — Indicates that the client accepts any media type.
- **“origin”: “http://localhost:3000"** — Specifies the origin of the request, which is the domain where the todo-client is hosted.
- **“sec-fetch-site”: “same-site”** — Indicates that the request is being made from the same site as the todo-client.
- **“sec-fetch-mode”: “cors”** — Specifies that the request is a cross-origin request and should be handled using CORS.
- **“sec-fetch-dest”: “empty”** — Indicates that the request’s destination is not a specific resource type.
- **“referer”: “http://localhost:3000/"** — Specifies the address of the previous web page from which the current request originated.

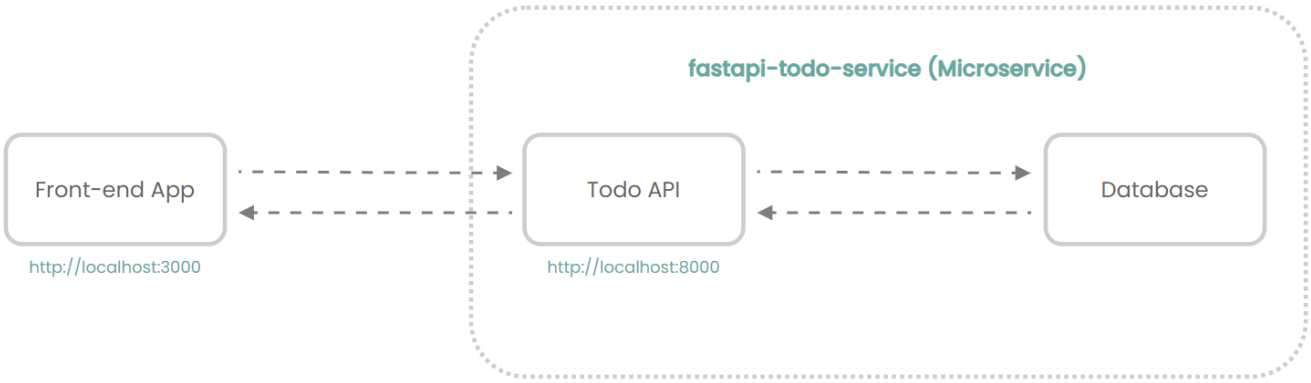
MDN web docs is the best place to learn more about the **HTTP protocol and HTTP headers**.

Accept - HTTP | MDN

The Accept request HTTP header indicates which content types, expressed as MIME types, the client is able to...

developer.mozilla.org

If you have followed the instructions correctly, you have completed the following part of our project architecture.

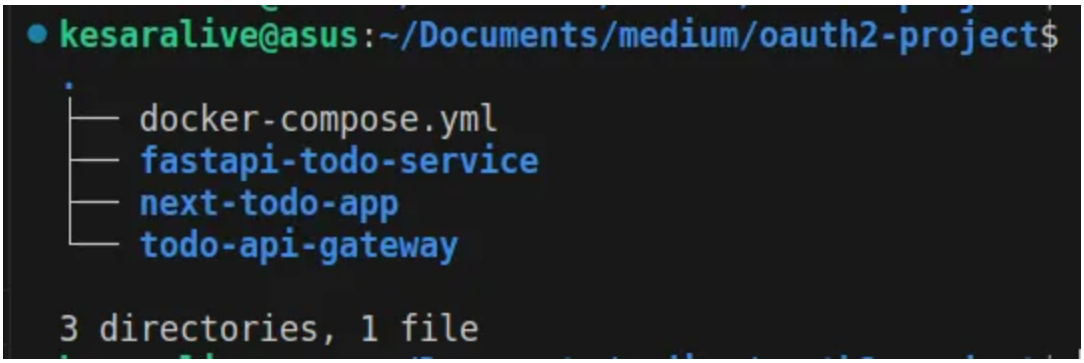


todo-client integration

Reverse Proxy/ API Gateway — Step 03

Now that we have a **todo-client**, **todo-api-service**, and a **database** up and running, let's integrate Nginx as a reverse proxy between the todo-client and the FastAPI todo-service.

Create a folder named `todo-api-gateway` inside your project folder. Your directory structure should look like this.



after creating todo-api-gateway folder

After that, create a **`Dockerfile`** and an **`nginx.conf`** file inside the **`todo-api-gateway`** folder, as shown in the below snippet.

```
.
├── docker-compose.yml
├── fastapi-todo-service
├── next-todo-app
└── todo-api-gateway
    ├── Dockerfile
    └── nginx.conf
```

Dockerfile

```
FROM nginx:latest

COPY nginx.conf /etc/nginx/nginx.conf

EXPOSE 80
EXPOSE 443

CMD ["nginx", "-g", "daemon off;"]
```

This Dockerfile is used to build a custom Nginx image for our project. We use **`nginx:latest`** as the base image for this custom image. Then, we update the default **`nginx.conf`** with our configuration file.

nginx.conf

```
events {
    worker_connections 1024;
}

http {
    server {
        listen 80;
        server_name localhost;

        location / {
            proxy_pass http://todo-api:8000;
        }
    }
}
```

This `nginx.conf` configuration sets up Nginx as a reverse proxy to forward requests from port 80 to the `todo-api` service running on port 8000.

- `events` : This block defines parameters that affect the Nginx event loop processing, such as the maximum number of connections per worker process.
- `http` : This block contains configuration for the HTTP protocol.
- `server` : This block defines a virtual server for handling HTTP requests.
- `listen 80` : Configures the server to listen on port 80 for incoming HTTP requests.

- `server_name localhost;` : Sets the server name to `localhost`
- `location /` : Defines how Nginx should process requests for the root URL. When you access http://localhost/, you will be accessing the logic inside the curly braces `{}` in this location block.
- `proxy_pass http://todo-api:8000;` : Forwards requests to `http://todo-api:8000` , which is the hostname of the `todo-api` service running on port 8000.

Now let's add this nginx reverse-proxy setup to our `docker-compose.yml` configuration.

```
version: "3.8"
services:
  mongodb:
    image: mongo:latest
    container_name: mongodb
    volumes:
      - mongodb_data:/data/db
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: password

  fastapi-todo-service:
    container_name: todo-api
    build:
      context: ./fastapi-todo-service
      dockerfile: Dockerfile
    depends_on:
      - mongodb
    ports:
      - 8000:8000
    environment:
      ATLAS_URI: mongodb://root:password@mongodb:27017
      DB_NAME: todoapi

  next-todo-app:
    container_name: todo-client
```

```
    build:
      context: ./next-todo-app
      dockerfile: Dockerfile
    ports:
      - 3000:3000

  api-gateway:
    container_name: nginx
    build:
      context: ./todo-api-gateway
      dockerfile: Dockerfile
    depends_on:
      - fastapi-todo-service
    ports:
      - 80:80

volumes:
  mongodb_data:
    driver: local
```

Now, you can use the `docker compose down` command to stop all the running containers, and the `docker compose up -d` command to run the newly added API gateway in the project.

```
docker compose down
```

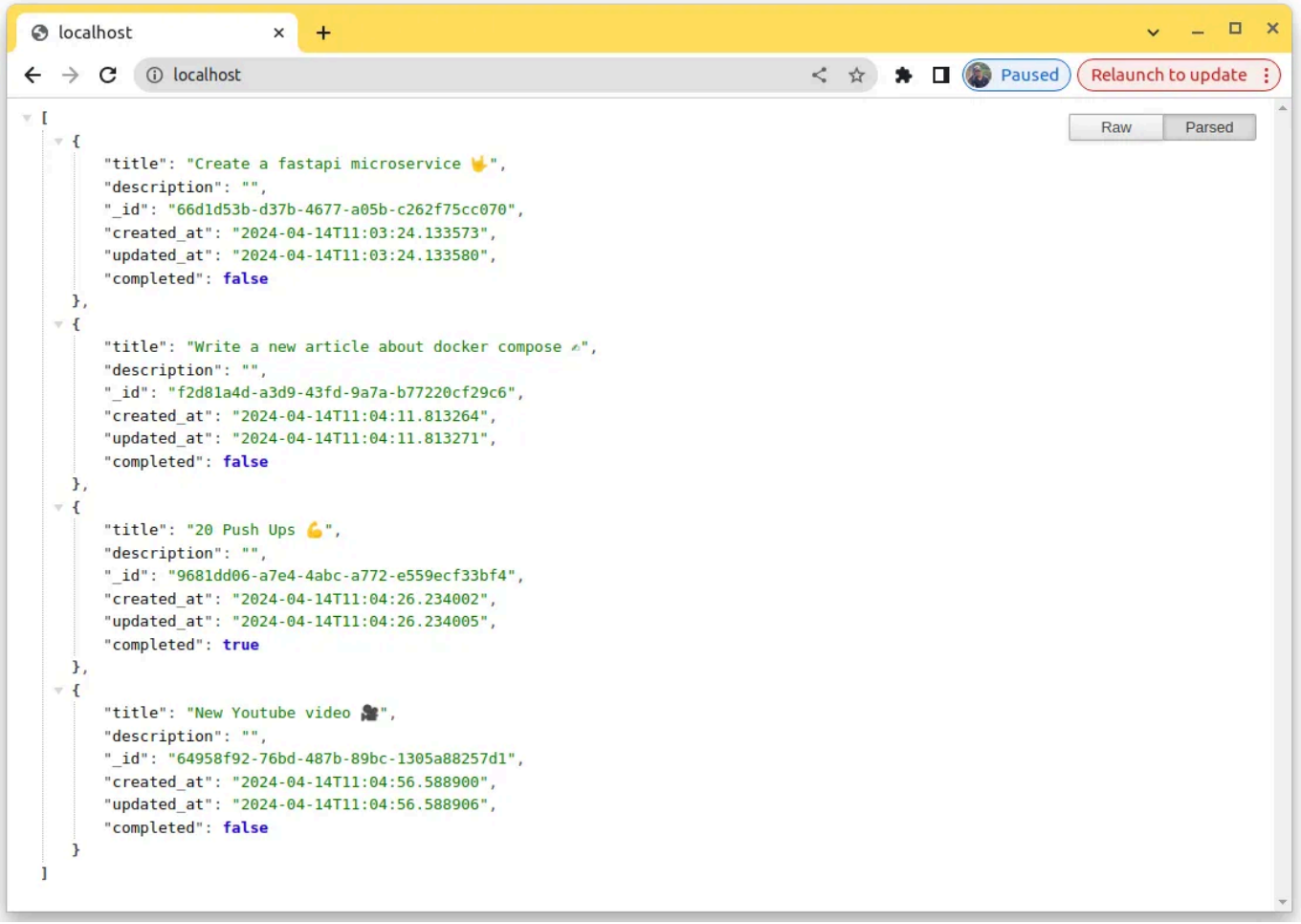
```
docker compose up -d
```

If you enter the following command, you will see four running containers.

```
docker ps
```

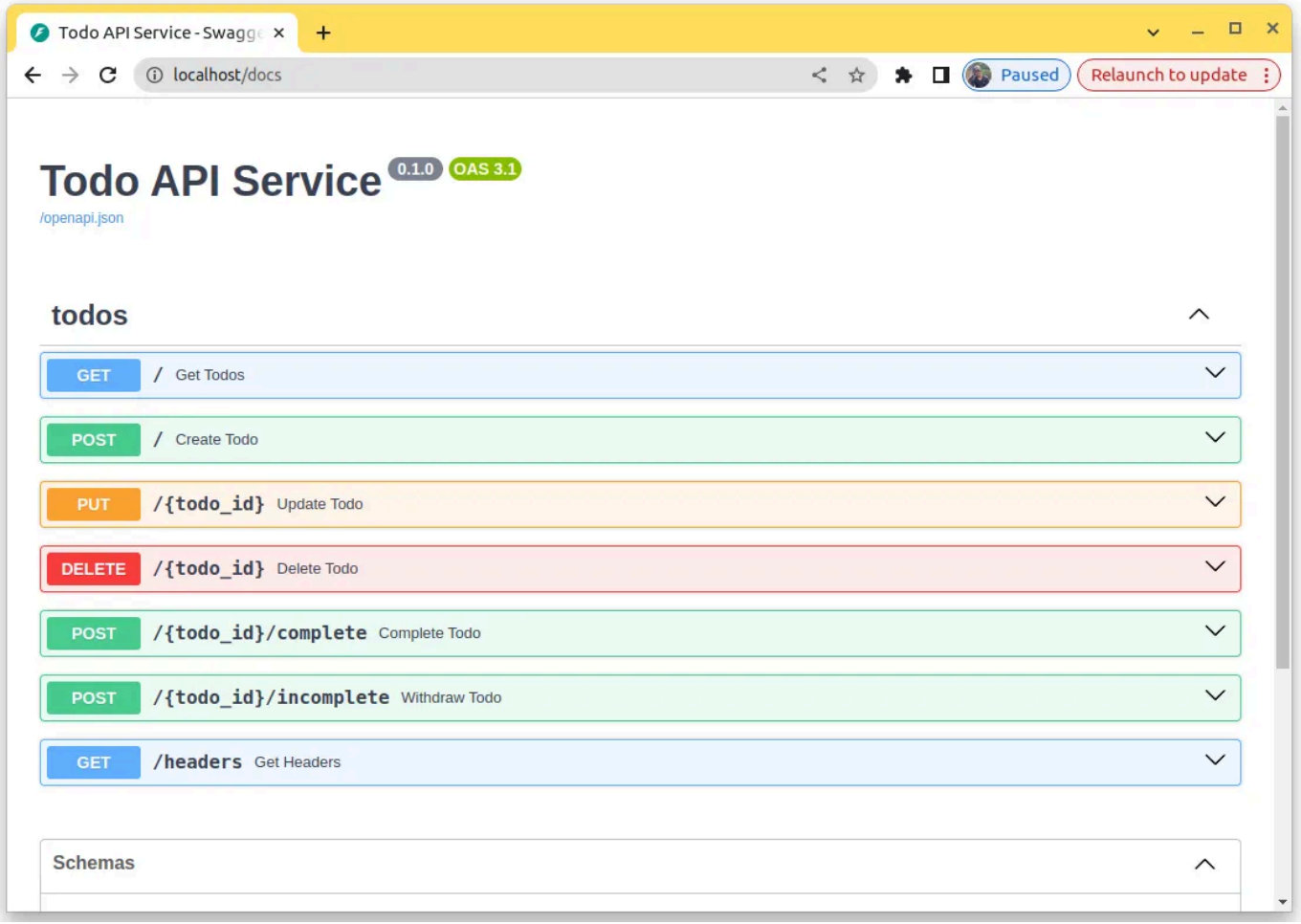
docker ps

In the `docker-compose.yml` file, we've mapped the Nginx service to the host machine's port 80. Therefore, now you should be able to access it at http://localhost/.



nginx reverse-proxy (http://localhost/)

If you have previously added todos, you will be able to see them as well. Similar to before when we accessed http://localhost:8000/docs, now you should be able to access the API documentation at http://localhost/docs.



accessing the todo-api through reverse proxy

Since we can now access the todo-api through the Nginx reverse proxy, let's update our todo-client to send requests using the reverse proxy.

Navigate to the file `next-todo-app/actions/todo.ts` and update the **`API_URL`** to `http://localhost/` . Please note that the **trailing slash** is important, so keep that in mind.

next-todo-app/actions/todo.ts

```
...

const API_URL = "http://localhost/"
```

```
...
```

Then, go to the file `next-todo-app/app/headers/page.tsx` and update the fetch URL inside the `useEffect` hook as well.

next-todo-app/app/headers/page.tsx

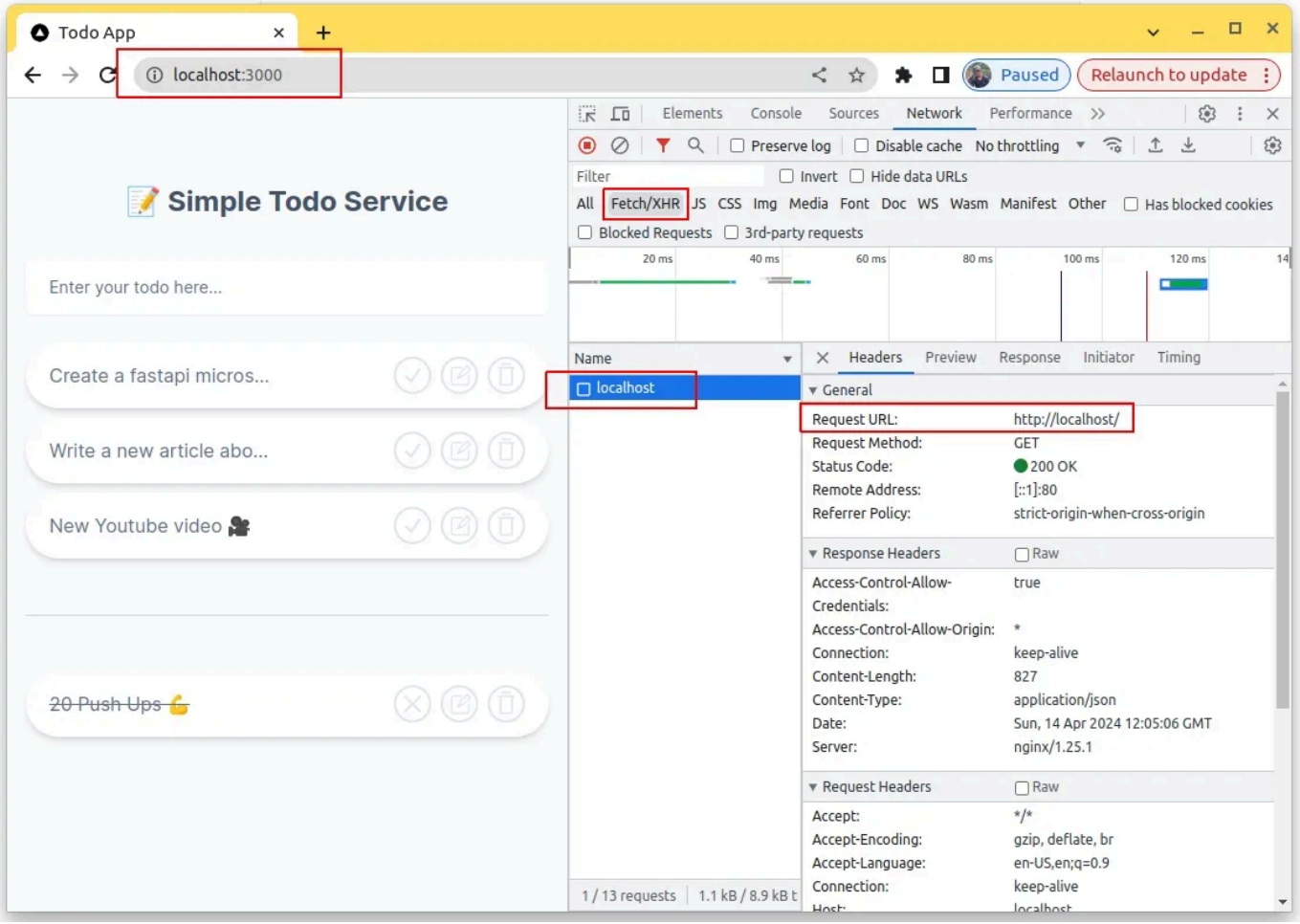
```
...
const response = await fetch('http://localhost/headers')
...
```

Since we've updated 2 files in the todo-client application, we now need to rebuild it in the Docker Compose. To do that, you can run the following command inside the project folder:

```
docker compose build next-todo-app
```

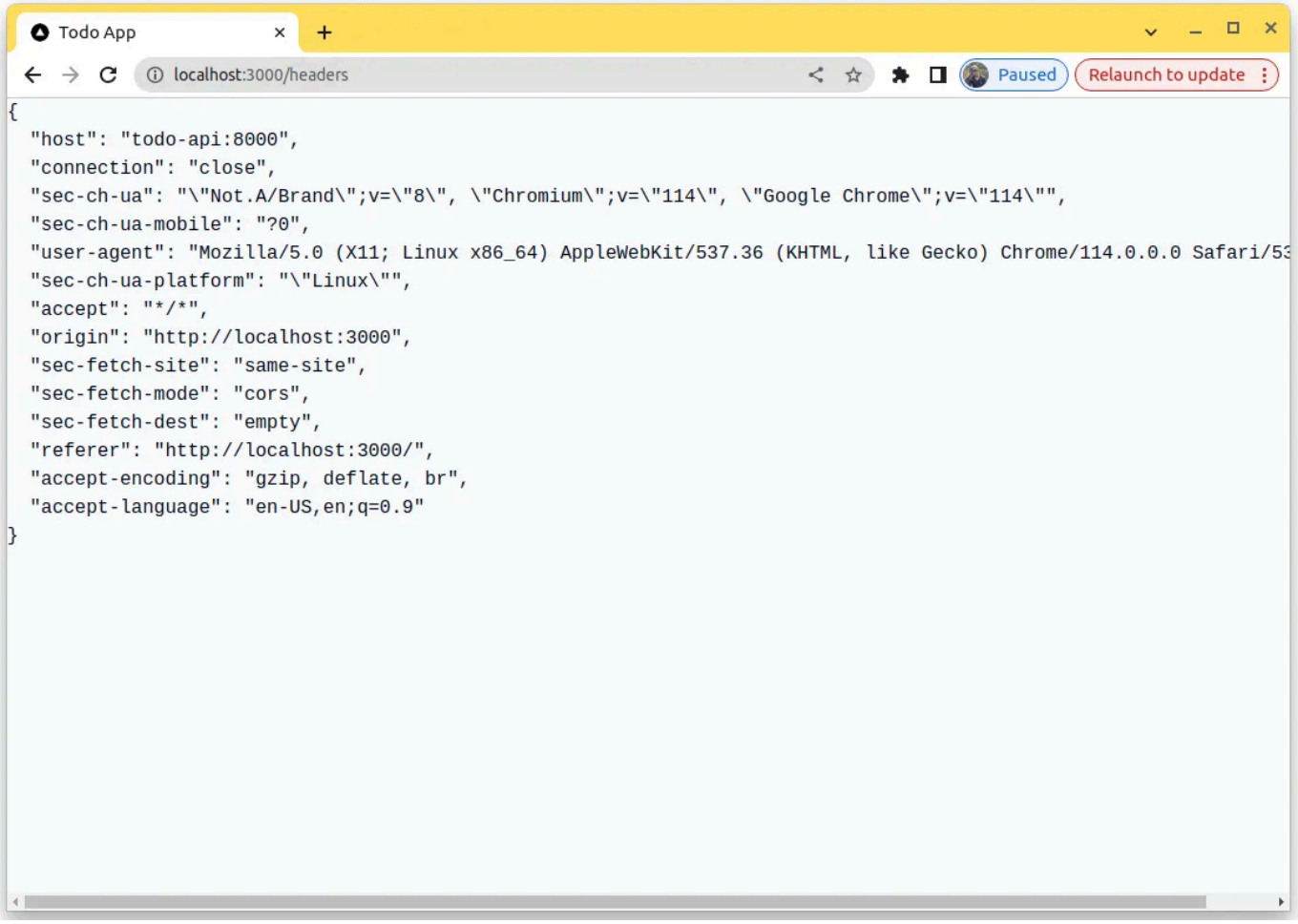
This command will rebuild the `next-todo-app` service in the Docker Compose, incorporating the changes you made to the files.

```
docker compose up -d
```



requests through nginx reverse-proxy

Now, Let's check requests headers by accessing the http://localhost:3000/headers page.



returned request headers

You will see some headers are updated there.

```
"host": "todo-api:8000"
"connection": "close"
```

- `"host": "todo-api:8000"` — indicates the hostname and port number of the server to which the request is being sent (`todo-api` on port 8000). This information is exposed to the client because we forwarded the client requests directly to the todo-api server without any additional configurations.

“Revealing internal infrastructure details to the client is not ideal.”
In the next step of this project, we will change this behavior.

- `“connection”: “close”` — indicates that the connection will be closed after the current transaction is completed. This means that the client or server intends to close the TCP connection after sending or receiving the current HTTP message.
  In an Nginx reverse proxy, this practice can help manage server resources by limiting the number of open connections. It is a part of the connection management strategy to control the lifecycle of connections between clients and servers, helping in the optimization of resource usage and performance improvement.

If you have followed the instructions correctly, you have completed the following part of our project architecture.

after integrating the frontend app

Network Isolation — Step 04

In the current setup, you should be able to access both the Nginx reverse proxy service and the todo-api service. However, when using a reverse proxy, there’s no point in giving direct access to the todo-api. First, let’s see how our current setup works by inspecting the `docker-compose.yml` file.

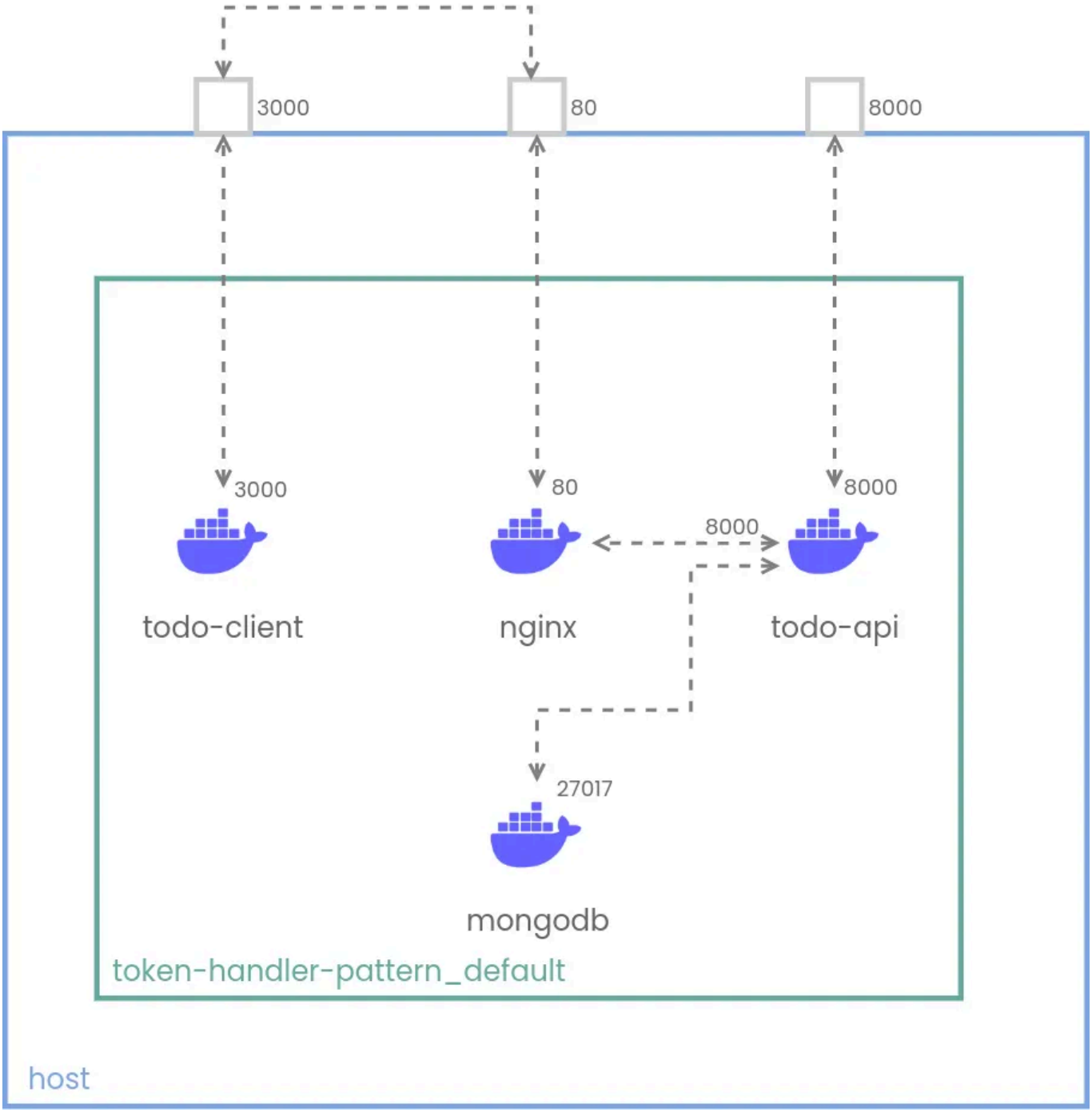
```
...

services:
  mongodb:
    image: mongo:latest
    container_name: mongodb
    volumes:
      - mongodb_data:/data/db
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: password

  fastapi-todo-service:
    container_name: todo-api
    build:
      context: ./fastapi-todo-service
      dockerfile: Dockerfile
    depends_on:
      - mongodb
    ports:
      - 8000:8000
    environment:
      ATLAS_URI: mongodb://root:password@mongodb:27017
      DB_NAME: todoapi

  next-todo-app:
    container_name: todo-client
    build:
      context: ./next-todo-app
      dockerfile: Dockerfile
    ports:
      - 3000:3000

  api-gateway:
    container_name: nginx
    build:
      context: ./todo-api-gateway
      dockerfile: Dockerfile
    depends_on:
      - fastapi-todo-service
    ports:
      - 80:80

...
```

Currently, we have 4 services up and running, and all the services are running on the default bridge network created by Docker Compose when we first ran the setup. You can verify this by using the following command.

```
docker network ls
```

You'll see a bridge network named `token-handler-pattern_default`. This name may vary depending on your project folder name. If you want to understand docker networking deeply, refer my previous article about docker networking.

However, if we visualize the current setup in a diagram, it would look like the image below.



all the containers in token-handler-pattern_default bridge network

The current setup includes the following mappings and configurations:

- The `todo-client` container's port 3000 is mapped to the host's port 3000.
- The `nginx` container's port 80 is mapped to the host's port 80.
- The `todo-api` container's port 8000 is mapped to the host's port 8000.

- The **`todo-api`** container accesses the **`mongodb`** container's port 27017.
- The **`todo-client`** accesses **`nginx`** through the host network using the host's port 80.
- **`nginx`** proxies requests to the **`todo-api`** container using port 8000.

To fix this, we need to remove the unnecessary port mappings and isolate the `nginx`, `todo-api`, and `mongodb` containers to the `token-handler-pattern_default` network. Here's how you can do it:

```
version: "3.8"
services:
  mongodb:
    image: mongo:latest
    container_name: mongodb
    volumes:
      - mongodb_data:/data/db
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: password
    networks:
      - microservices

  fastapi-todo-service:
    container_name: todo-api
    build:
      context: ./fastapi-todo-service
      dockerfile: Dockerfile
    depends_on:
      - mongodb
    environment:
      ATLAS_URI: mongodb://root:password@mongodb:27017
      DB_NAME: todoapi
    networks:
      - microservices

  next-todo-app:
    container_name: todo-client
    build:
      context: ./next-todo-app
```

```
      dockerfile: Dockerfile
    ports:
      - 3000:3000
    networks:
      - client-host

  api-gateway:
    container_name: nginx
    build:
      context: ./todo-api-gateway
      dockerfile: Dockerfile
    depends_on:
      - fastapi-todo-service
    ports:
      - 80:80
    networks:
      - microservices

networks:
  microservices:
    driver: bridge
  client-host:
    driver: bridge

volumes:
  mongodb_data:
    driver: local
```

Updates

I've set up two networks: one for hosting microservices (microservices network) and another for hosting the frontend (client-host).

```
...

networks:
  microservices:
    driver: bridge
  client-host:
    driver: bridge
```

```
...
```

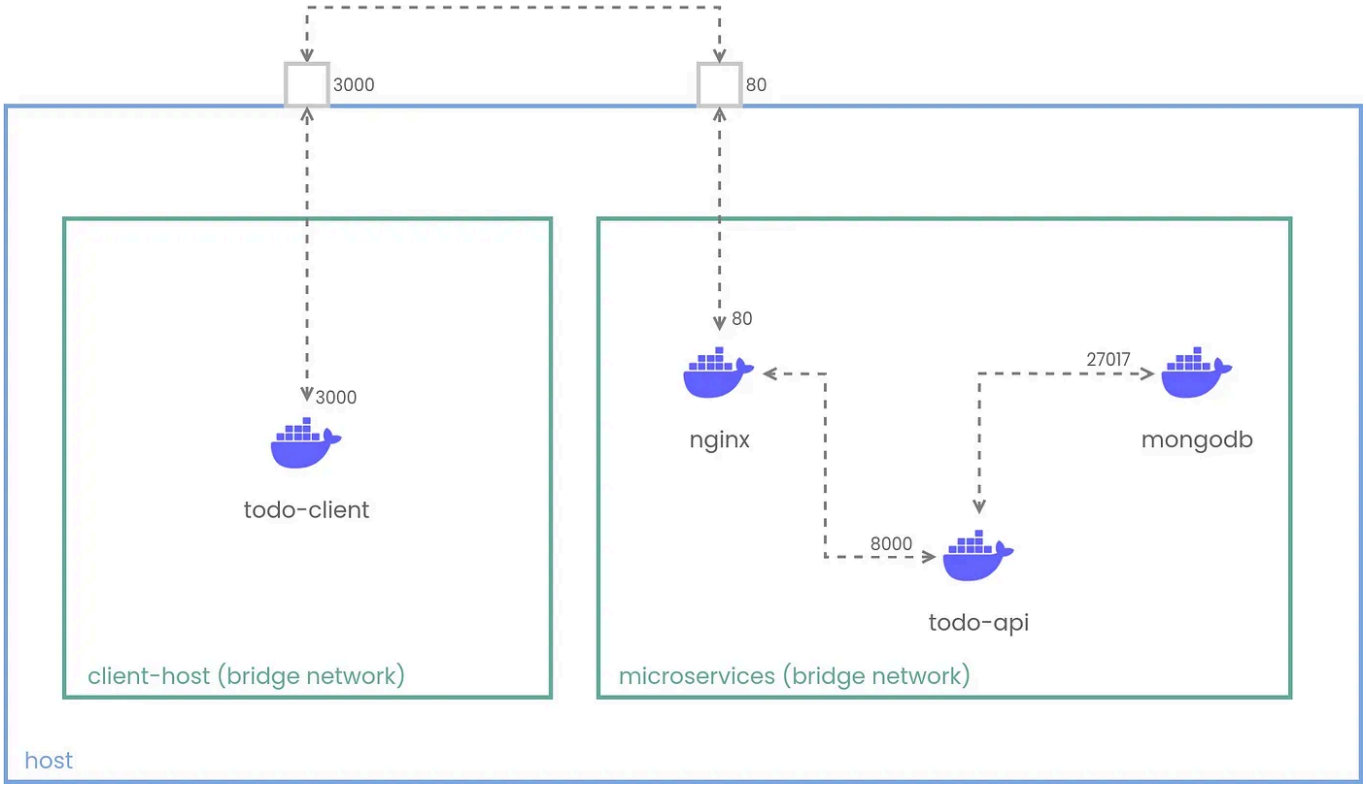
In this setup, the microservices network represents a place to host all your microservices, such as Azure AKS, Azure Container Apps, Google Cloud Run, GKE, or EKS.

The client-host network, on the other hand, represents a different place to host your frontend, like Vercel, Hostinger, or AWS Amplify.

These two networks don't have direct connectivity with each other.

- Removed the unnecessory port mapping of **`todo-api`** container's port 8000 to the host's port 8000.
- Added all the containers to the microservices network except for the `todo-client` .
- `todo-client` container is added to the `client-host` network and its port 3000 is mapped to the host's port 3000.

Diagram



separated bridge networks (user defined)

Now, run the following command to restart the containers and observe the changes.

```
docker compose up -d
```

You can check these two networks by running the following commands. You'll see two newly created networks, each prefixed by the project folder name.

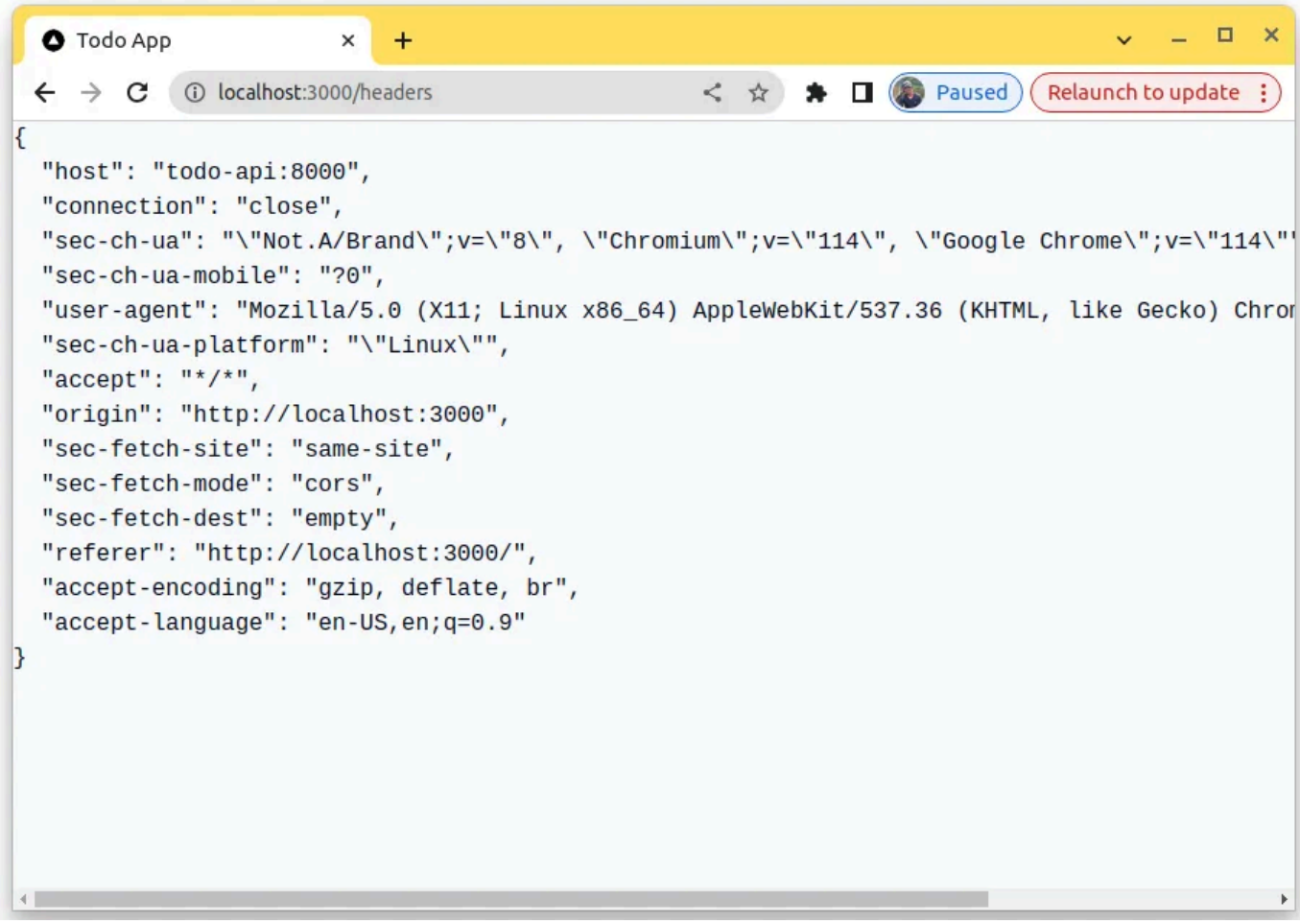
For example, if your project folder is named "oauth2-project", the networks might be named " `oauth2-project_microservices` " and " `oauth2-project_client-host` ".

```
docker network ls
```

You can inspect the network using the following command. This will allow you to see the network configuration for the containers mentioned above.

```
docker network inspect <<network-name>>
```

If you access http://localhost:3000/headers, you'll still see the host header as "todo-api:8000". Let's fix that now.



You can fix this by setting the Host header to the domain of the Nginx API gateway. Add a `proxy_set_header` directive in the Nginx configuration, like in the snippet below:

todo-api-gateway/nginx.conf

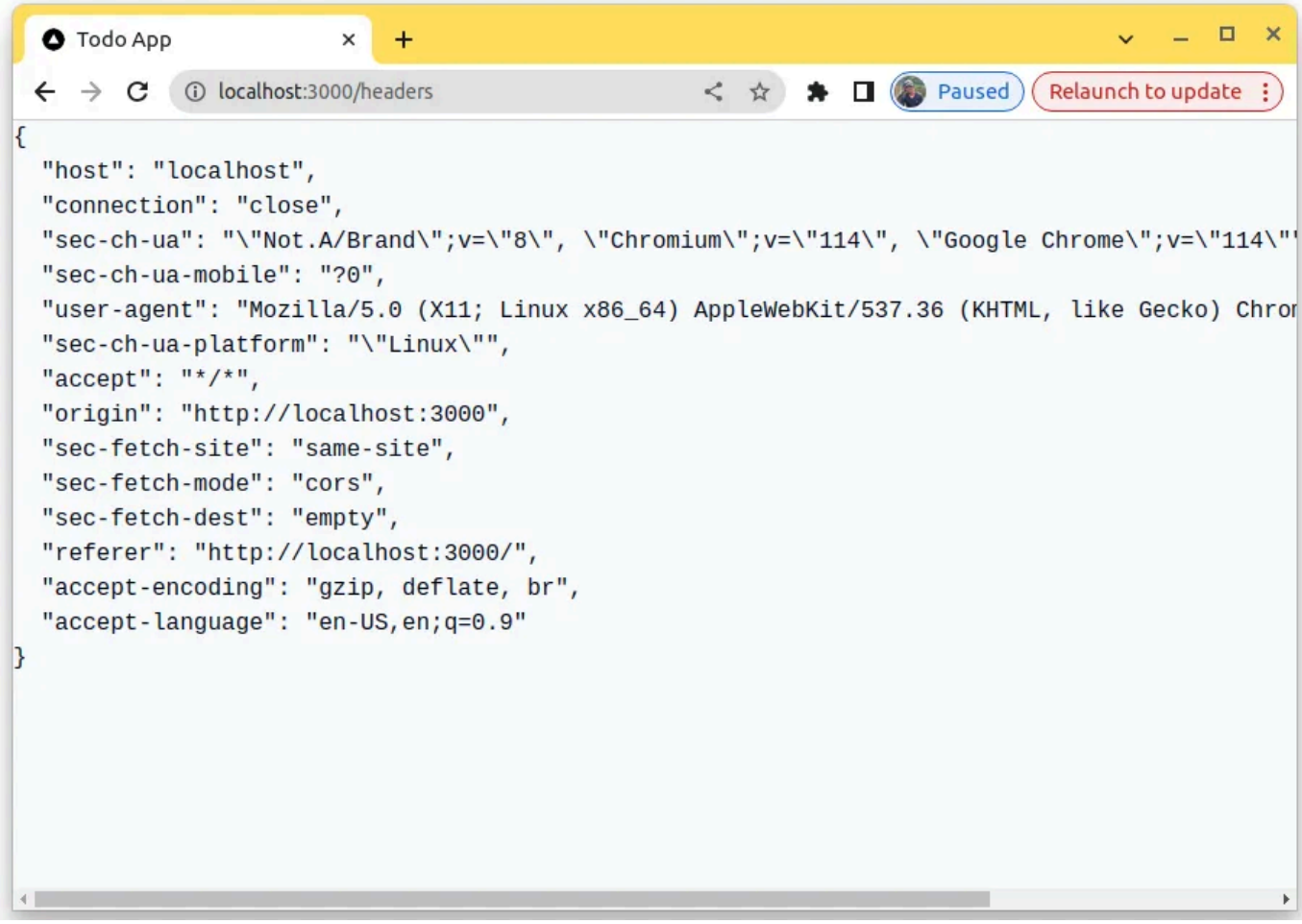
```
events {
    worker_connections 1024;
}

http {
    server {
        listen 80;
        server_name localhost;

        location / {
            proxy_set_header Host       $host;
            proxy_pass http://todo-api:8000;

        }
    }
}
```

To rebuild the API gateway image and restart the container, you can use the following commands:

```
docker compose build api-gateway
```

```
docker compose up -d
```

If you access http://localhost:3000/headers now, you will see that the host has been changed to “localhost” and doesn’t give any information about the underlying architecture.



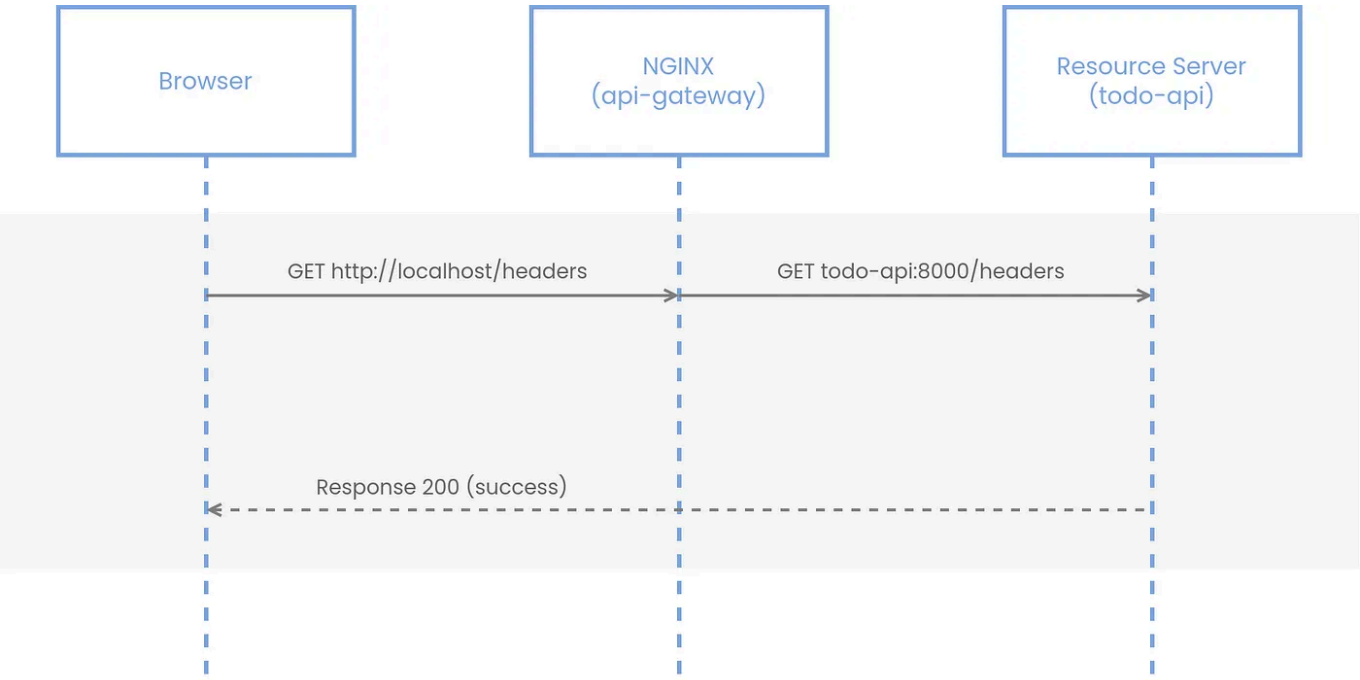
updated returned request headers

OK. now we have the setup to implement OAuth2 proxy to secure our microservice API. (fastapi-todo-service)

Securing APIs (OAuth2 Proxy setup)— part 05

Current Setup

First, let's understand the current setup using the following diagram.



sequence diagram for the /headers request

1. User initiates a request to the `/headers` endpoint.
2. NGINX (API Gateway) proxies the request to `todo-api:8000/headers` as configured.
3. The resource server (`todo-api`) returns the response to the user's browser.

Our Goal

What we're aiming to do is implement an authentication mechanism at the API gateway to inspect incoming requests and determine whether to forward

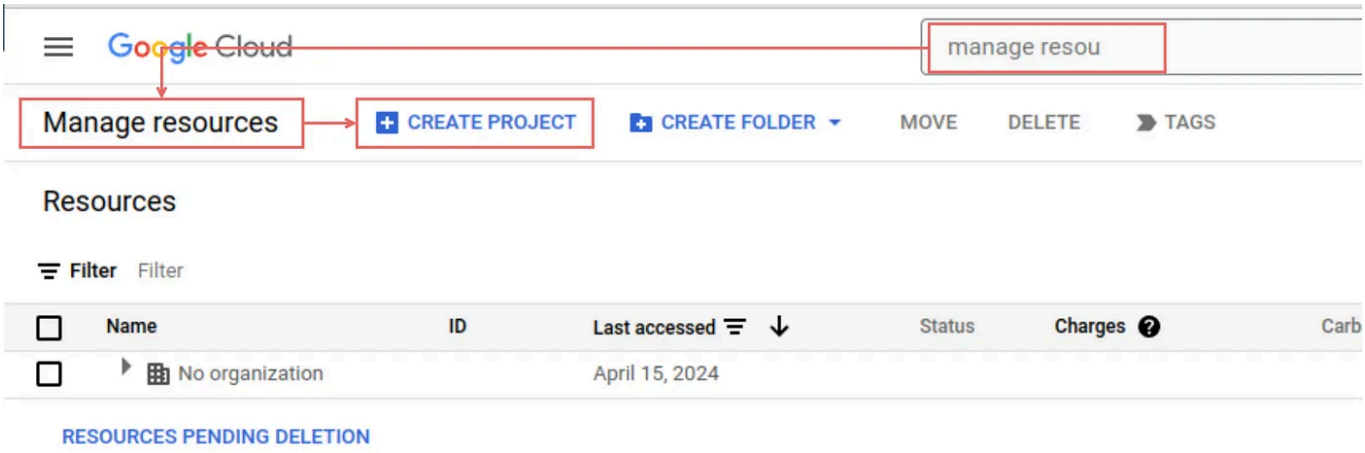
the request to the resource server or not.

This is where we need OAuth2 Proxy. Let's configure OAuth2 Proxy for our current setup and discuss what happens when we try to access our `todo-api` service.

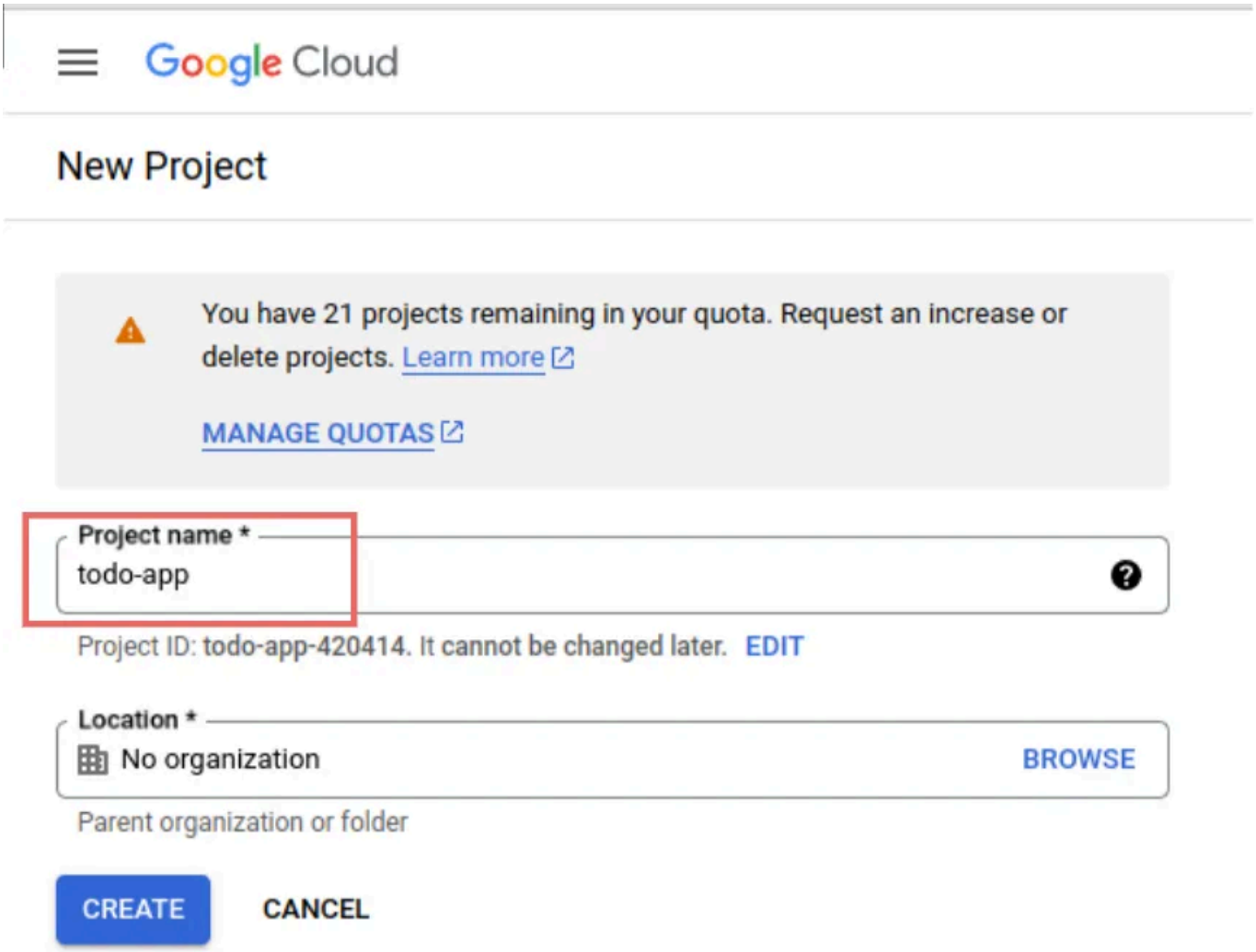
Google OAuth 2.0 setup

In order to setup OAuth2 proxy we need to have an auth provider, In this project we're going to use Google OAuth2.0 as our provider.

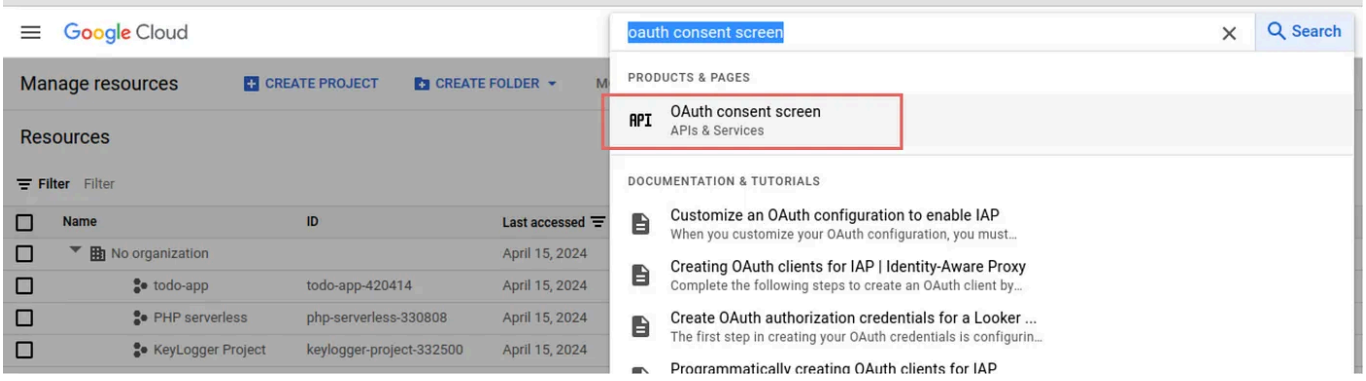
1. Go to your Google Cloud Console and search for "Manage Resources."
2. Click the "Create Project" button to create a new project.



3. Provide a project name and click "Create."

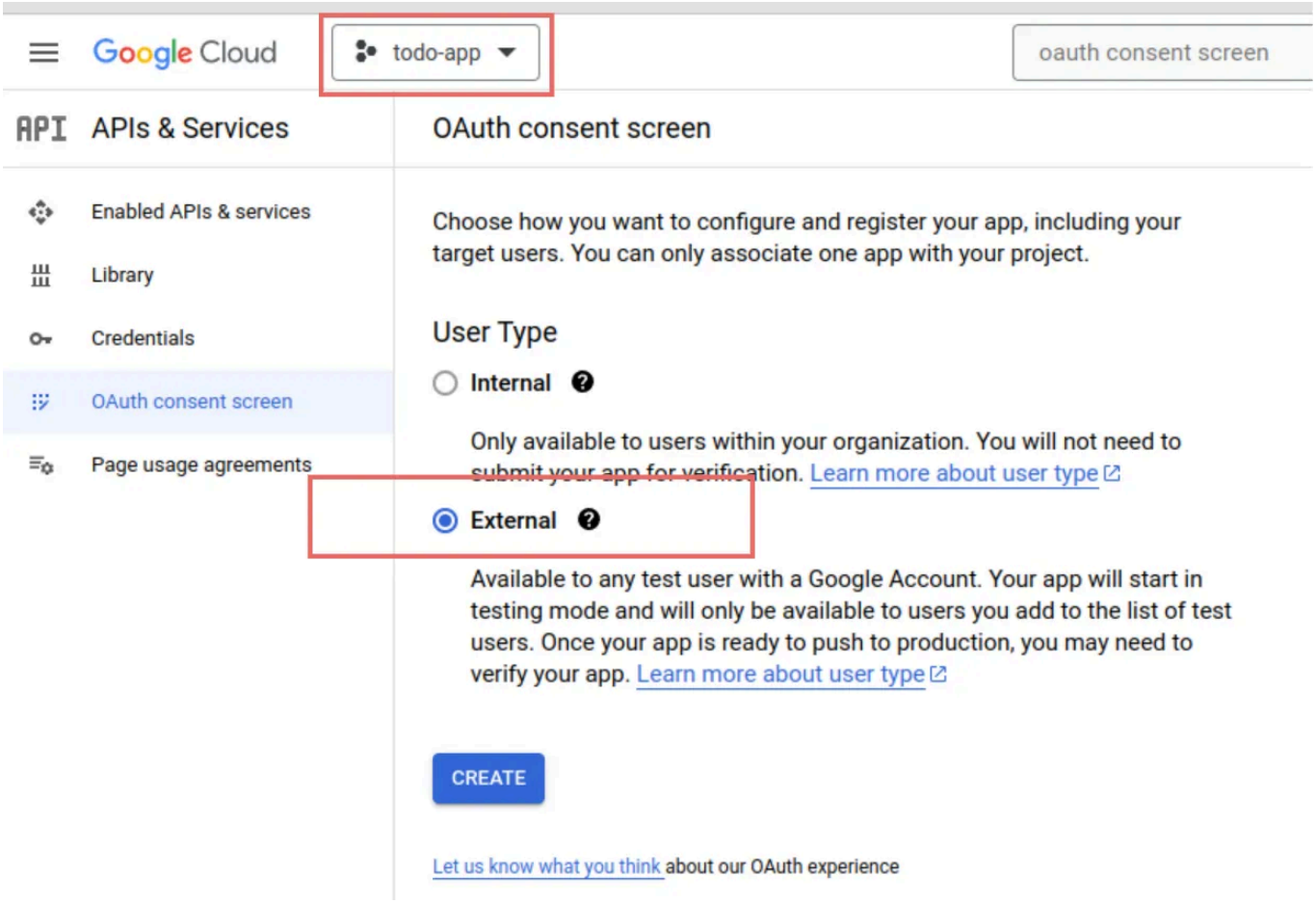


4. Navigate to the OAuth consent screen by searching for it in the Google Cloud Console.

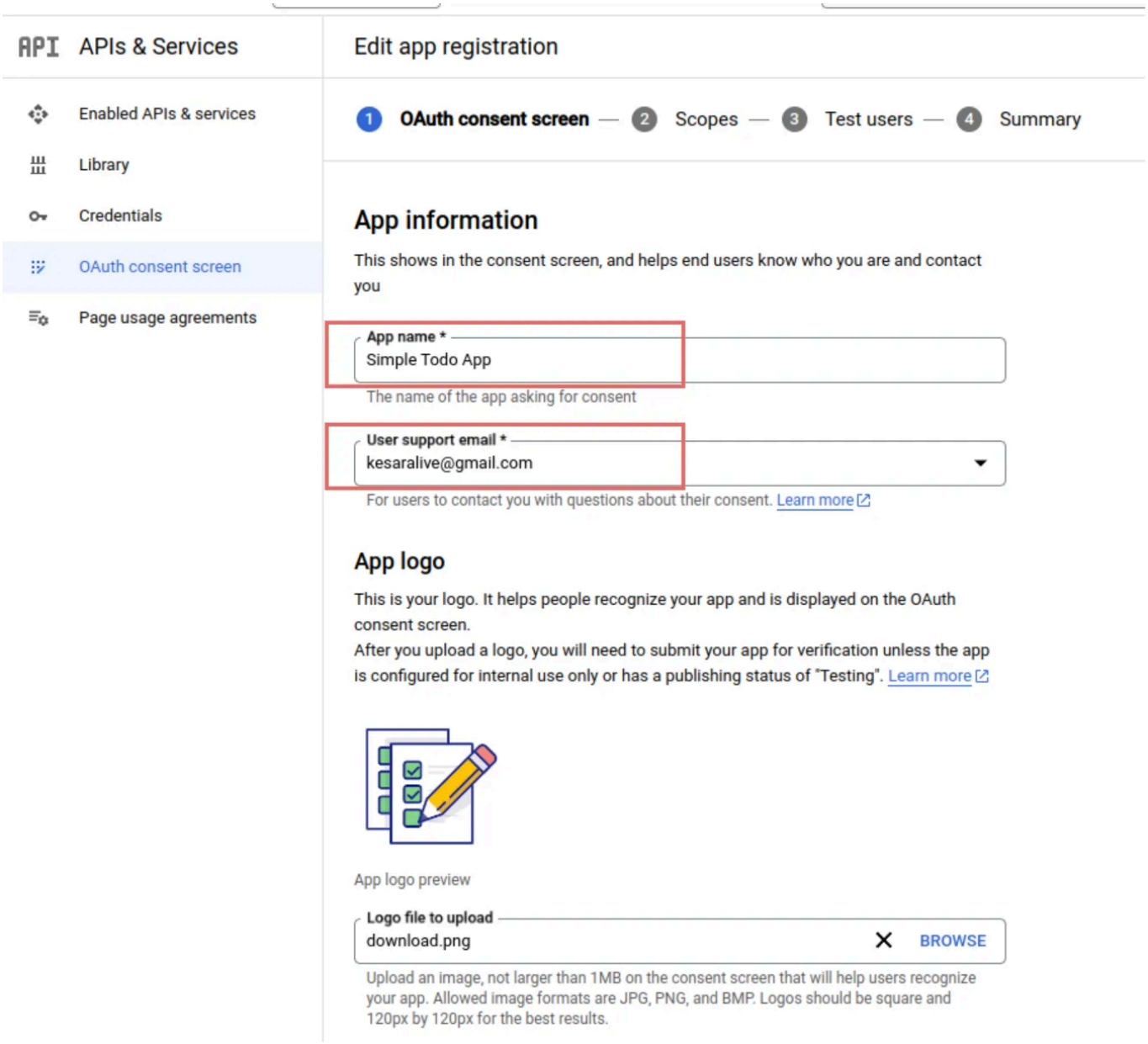


5. Make sure you are in the correct project, ‘todo-app.’

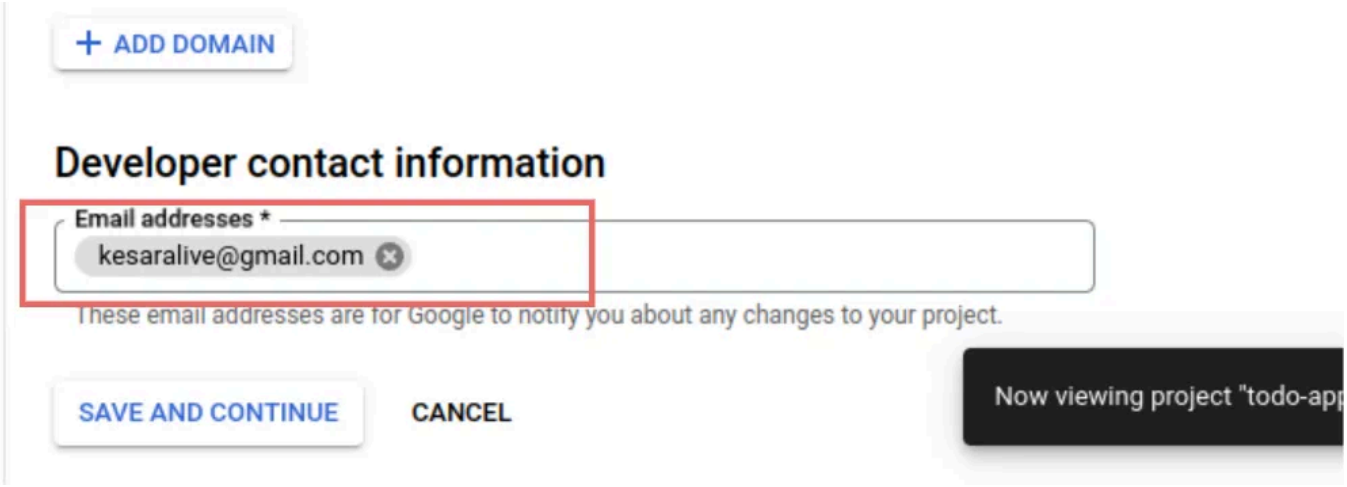
6. Select ‘External’ and click ‘Create.’



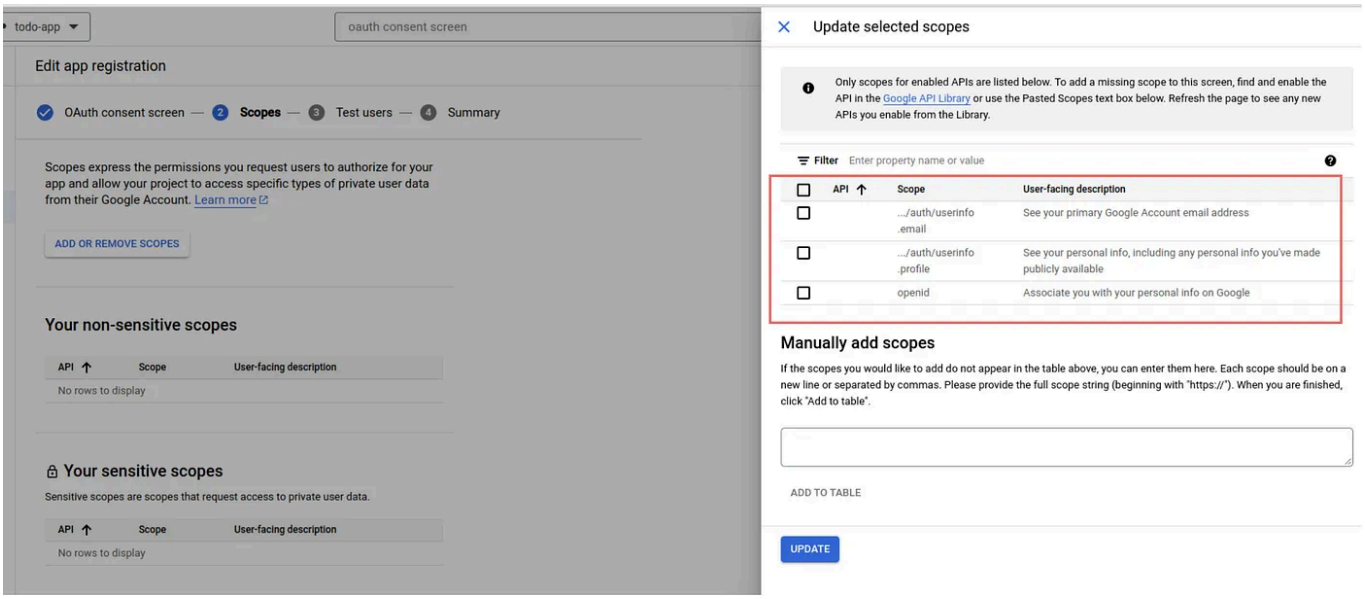
7. Complete all the required details in the form. You can skip non-required ones.



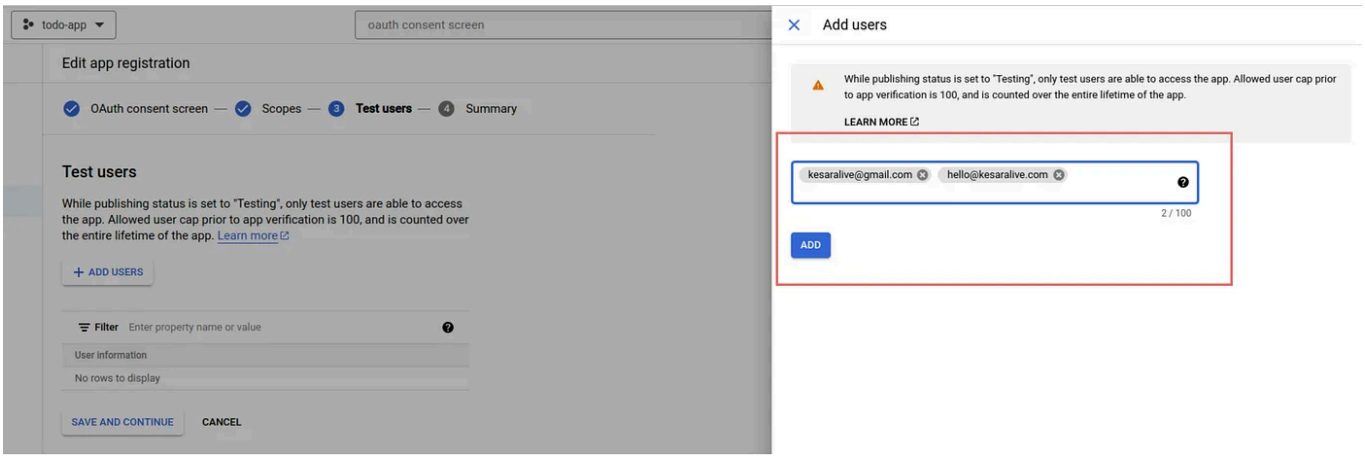
8. Save and continue after entering your email address for the developer contact information.



7. Select all three scopes in the right-hand panel and click ‘Update.’

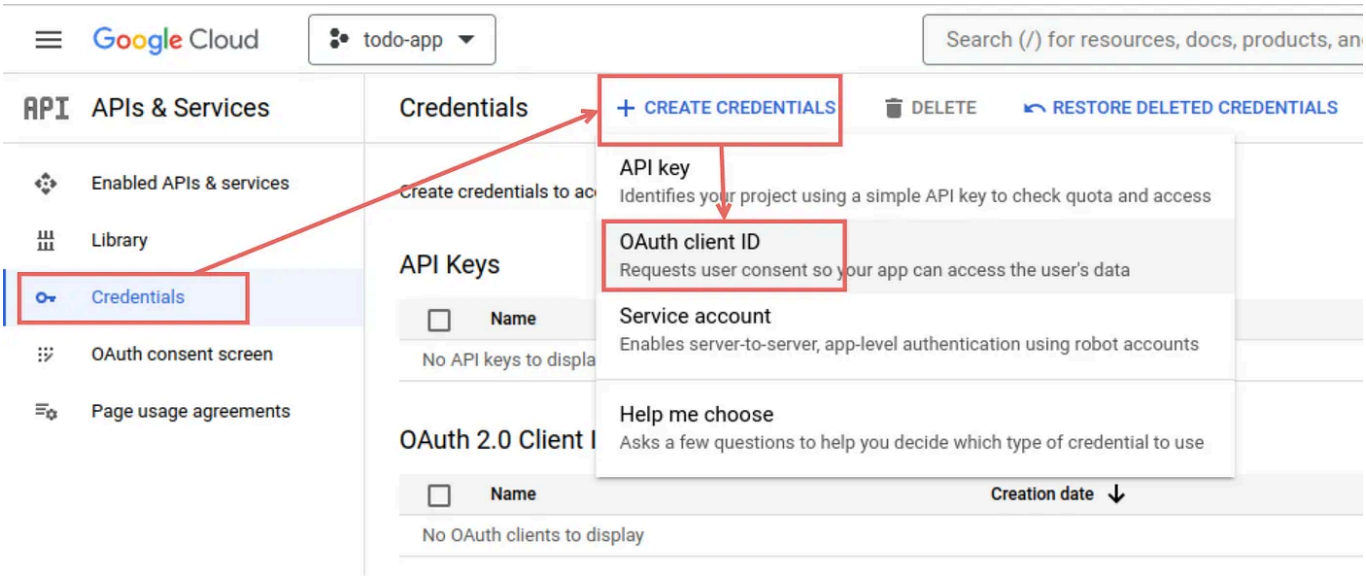


8. Next, enter two or more Gmail accounts to test the setup later.



9. Click ‘Add’ and then go back to the dashboard.

10. Click the ‘Credentials’ button in the left-side menu and create an OAuth client ID.

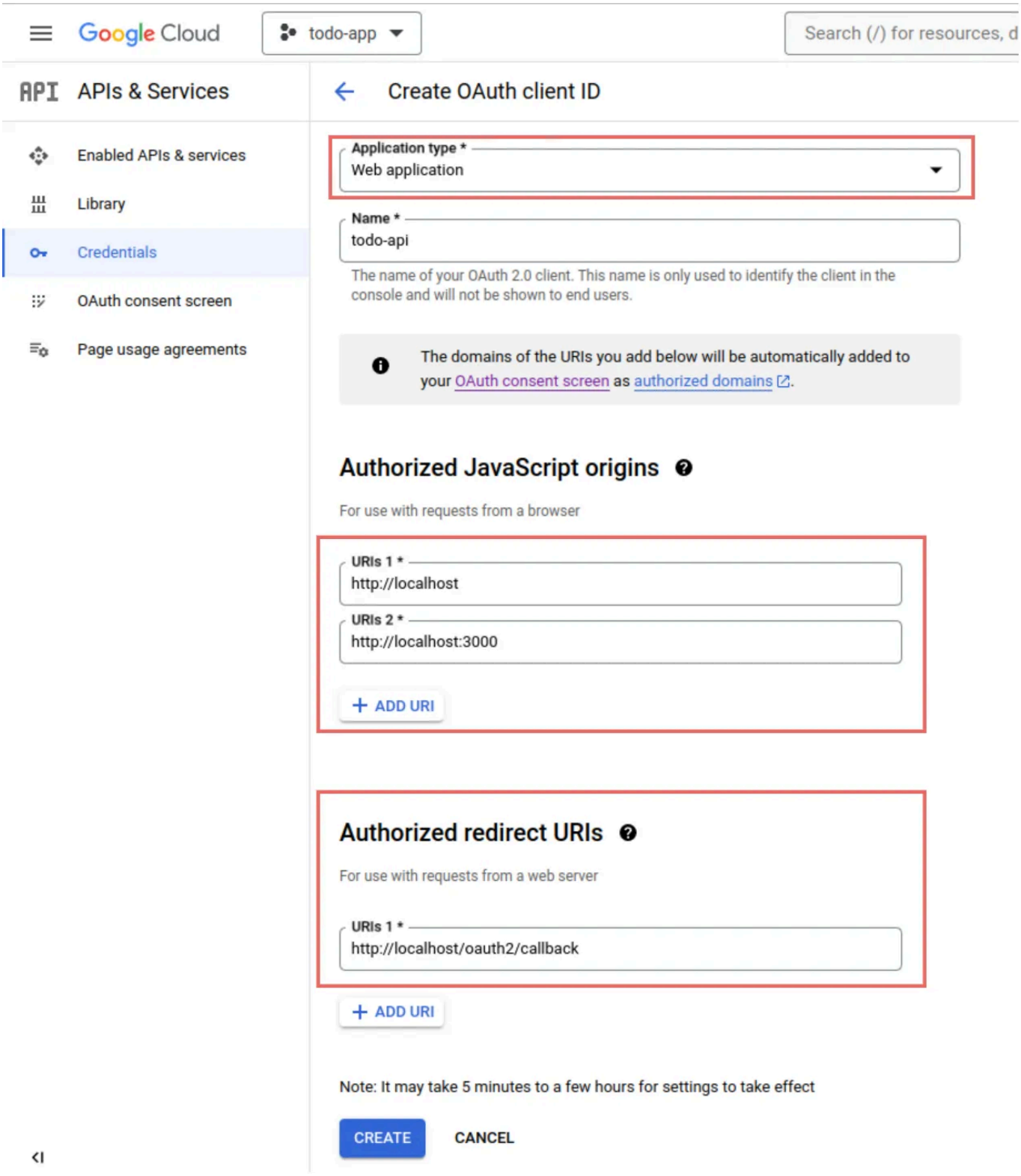


11. Select the application type as ‘Web application’ and give it a name to identify it.

12. Add ‘http://localhost' and ‘http://localhost:3000' as JavaScript origins.

13. Add ‘http://localhost/oauth2/callback' as the redirect URL. Since our NGINX OAuth2 Proxy URL is ‘http://localhost/', the callback URL of the OAuth2 Proxy will be that.

14. Create the credential, and it will show you the client ID and client secret. Save it for later usage.



OAuth2 Proxy setup

First, stop all running containers by running the following command.

```
docker compose down
```

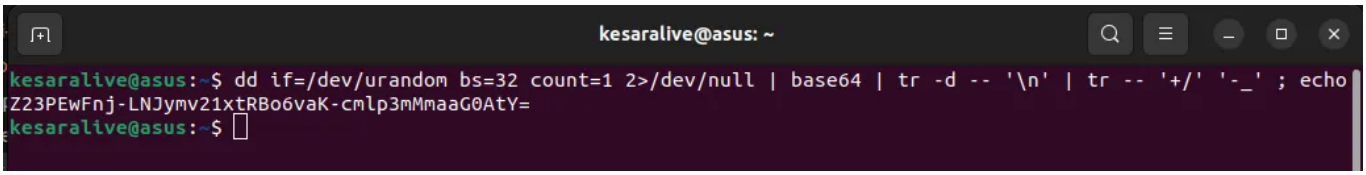
Let's add the OAuth2 Proxy to the `docker-compose.yml` file. Remember, correct indentation is crucial in YAML, so make sure to avoid indentation errors.

```
  oauth2-proxy:
    container_name: oauth2-proxy
    image: quay.io/oauth2-proxy/oauth2-proxy:latest
    environment:
      - OAUTH2_PROXY_CLIENT_ID=<your-client-id>
      - OAUTH2_PROXY_CLIENT_SECRET=<your-client-secret>
      - OAUTH2_PROXY_COOKIE_SECRET=<cookie-secret>
      - OAUTH2_PROXY_EMAIL_DOMAINS=*
      - OAUTH2_PROXY_REVERSE_PROXY=true
      - OAUTH2_PROXY_REDIRECT_URL=http://localhost/oauth2/callback
      - OAUTH2_PROXY_COOKIE_SECURE=false
      - OAUTH2_PROXY_UPSTREAM=http://todo-api:8000
      - OAUTH2_PROXY_HTTP_ADDRESS=http://0.0.0.0:4180
      - OAUTH2_PROXY_SET_AUTHORIZATION_HEADER=true
      - OAUTH2_PROXY_SET_XAUTHREQUEST=true
      - OAUTH2_PROXY_WHITELIST_DOMAINS=.localhost:3000
    command:
      - --http-address=0.0.0.0:4180
      - --upstream=http://todo-api:8000
      # - --skip-provider-button=true
    networks:
      - microservices
```

1. Update the `OAUTH2_PROXY_CLIENT_ID` and `OAUTH2_PROXY_CLIENT_SECRET` with your Google OAuth credentials.
2. To generate a cookie secret,

Linux terminal

```
dd if=/dev/urandom bs=32 count=1 2>/dev/null | base64 | tr -d -- '\n' | tr -- '+
```



generating a cookie secret

COPY the output value for your cookie secret.

Windows Powershell

```
# Add System.Web assembly to session, just in case
Add-Type -AssemblyName System.Web
[Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes([System.Web.Secu
```

Follow the documentation for more details.

Overview | OAuth2 Proxy

oauth2-proxy can be configured via command line options, environment variables or config file (in decreasing order of...

oauth2-proxy.github.io

Before running the services, you need to update your NGINX configuration as well.

nginx.conf

```
events {
    worker_connections 1024;
}

http {

    server {
    listen       80;
    server_name  localhost;

        location / {
            proxy_set_header Host       $host;
            proxy_set_header X-Real-IP  $remote_addr;

            proxy_pass http://oauth2-proxy:4180/;
        }
    }

    server {
        listen  80;
        server_name localhost;

        auth_request /internal-auth/oauth2/auth;

        error_page 401 = http://localhost/oauth2/start?rd=$scheme://$host$reques

        location / {
            proxy_pass http://todo-api:8000;
        }


        location /internal-auth/ {
            internal;

            proxy_set_header Host       $host;
            proxy_set_header X-Real-IP  $remote_addr;
            proxy_set_header X-Forwarded-Uri $request_uri;
            proxy_set_header Content-Length   "";
```

```
            proxy_pass_request_body           off;
            proxy_pass http://oauth2-proxy:4180/;
        }
    }
}
```

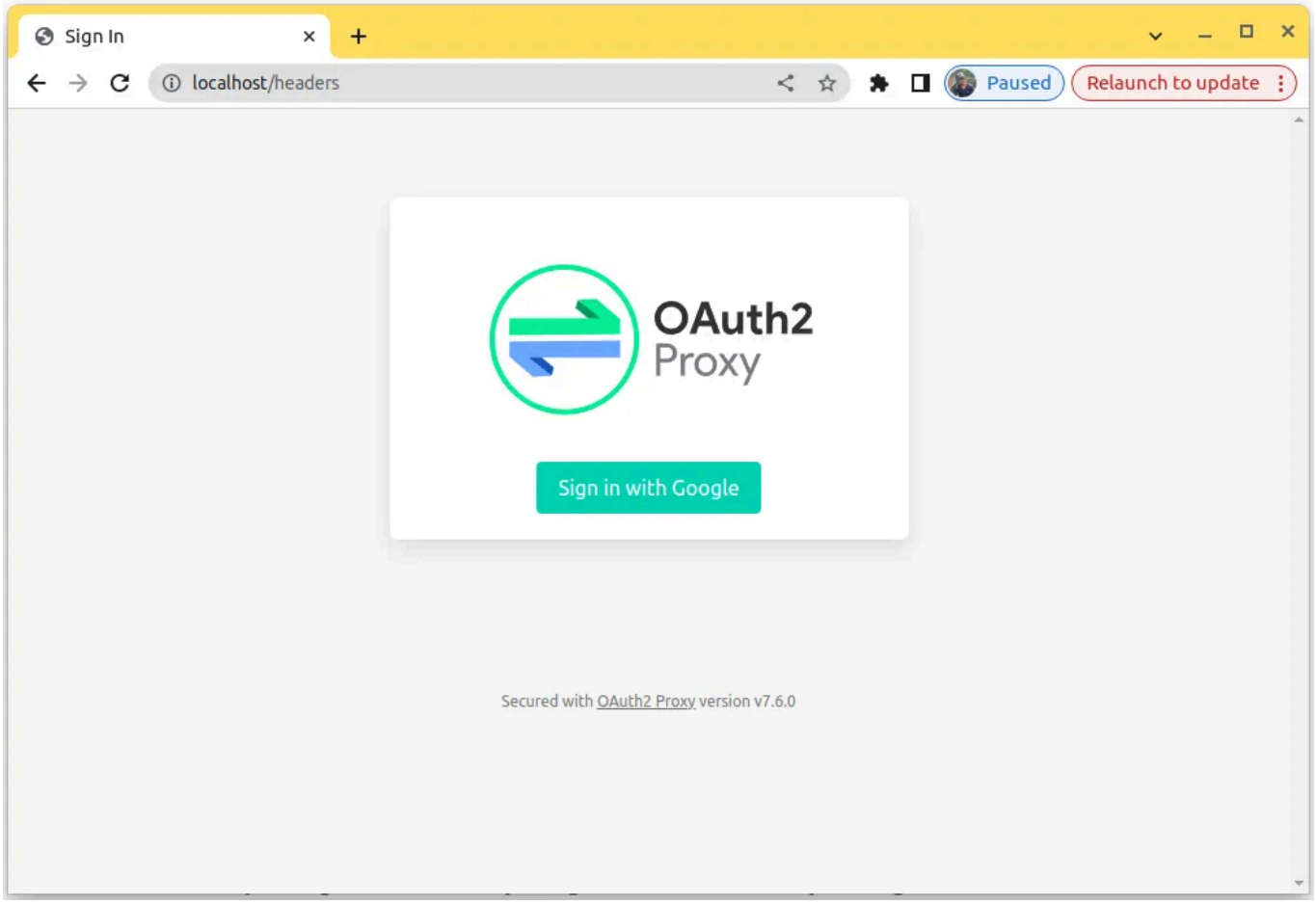
Now rebuild the NGINX API gateway service using the following command.

```
docker compose build api-gateway
```

Let's run all the containers again to see the effect.

```
docker compose up -d
```

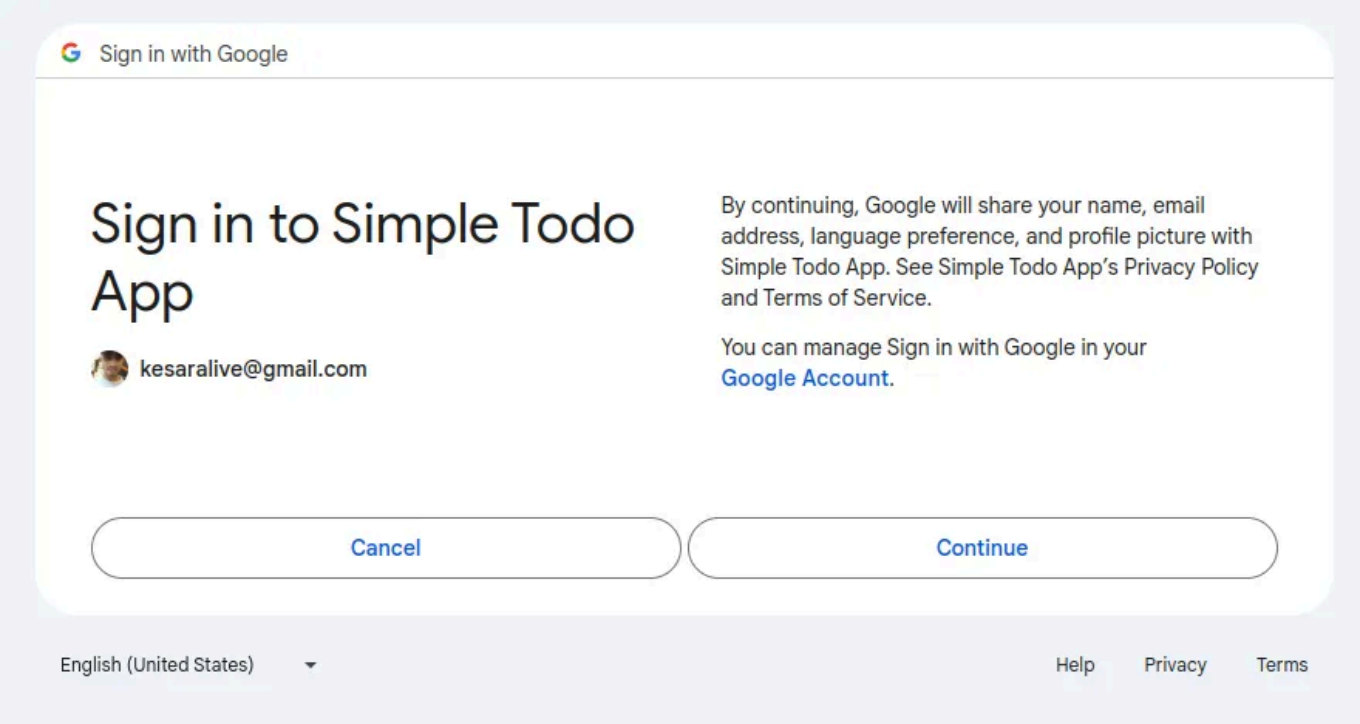
If you've followed all the instructions correctly, when you access `http://localhost/headers`, you should see the OAuth2 Proxy "Sign in with Google" button, similar to the image below.



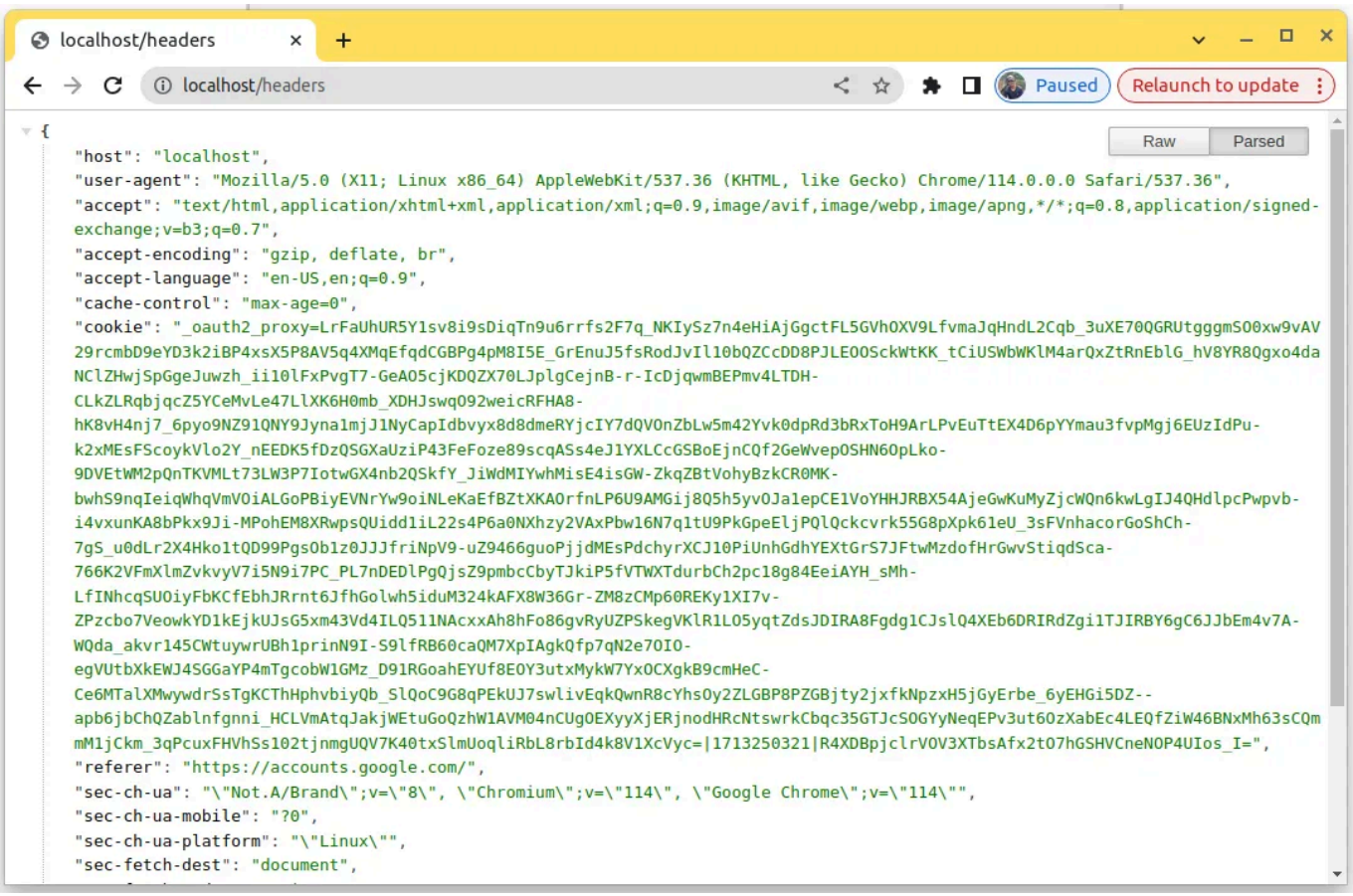
oauth2 proxy login portal

Try to log in with one of your Gmail accounts that you added as a test user when you were creating the Google OAuth 2.0 consent screen.

You will be asked to continue. Click the “Continue” button.

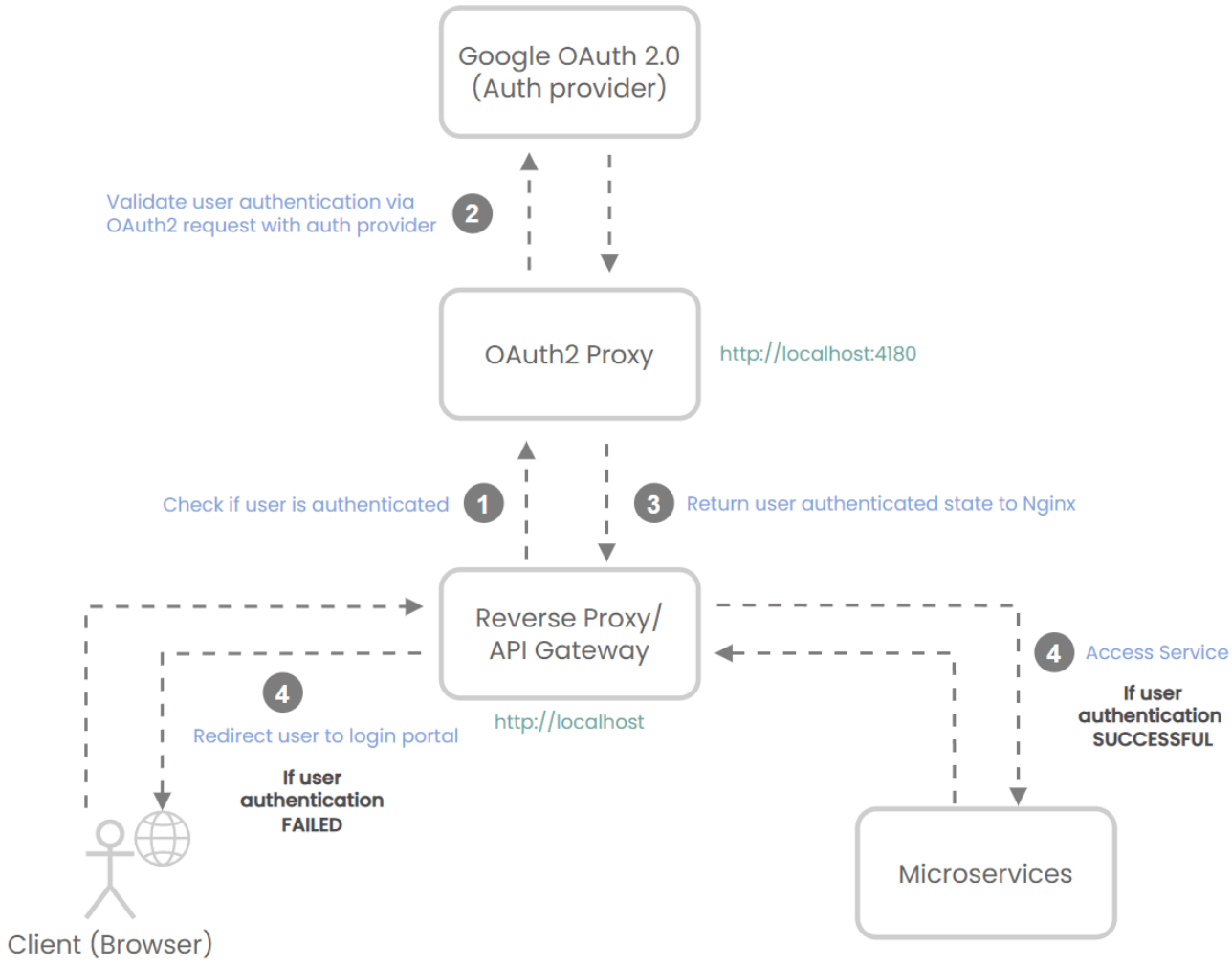


Now, you will see the headers page with a large cookie header.



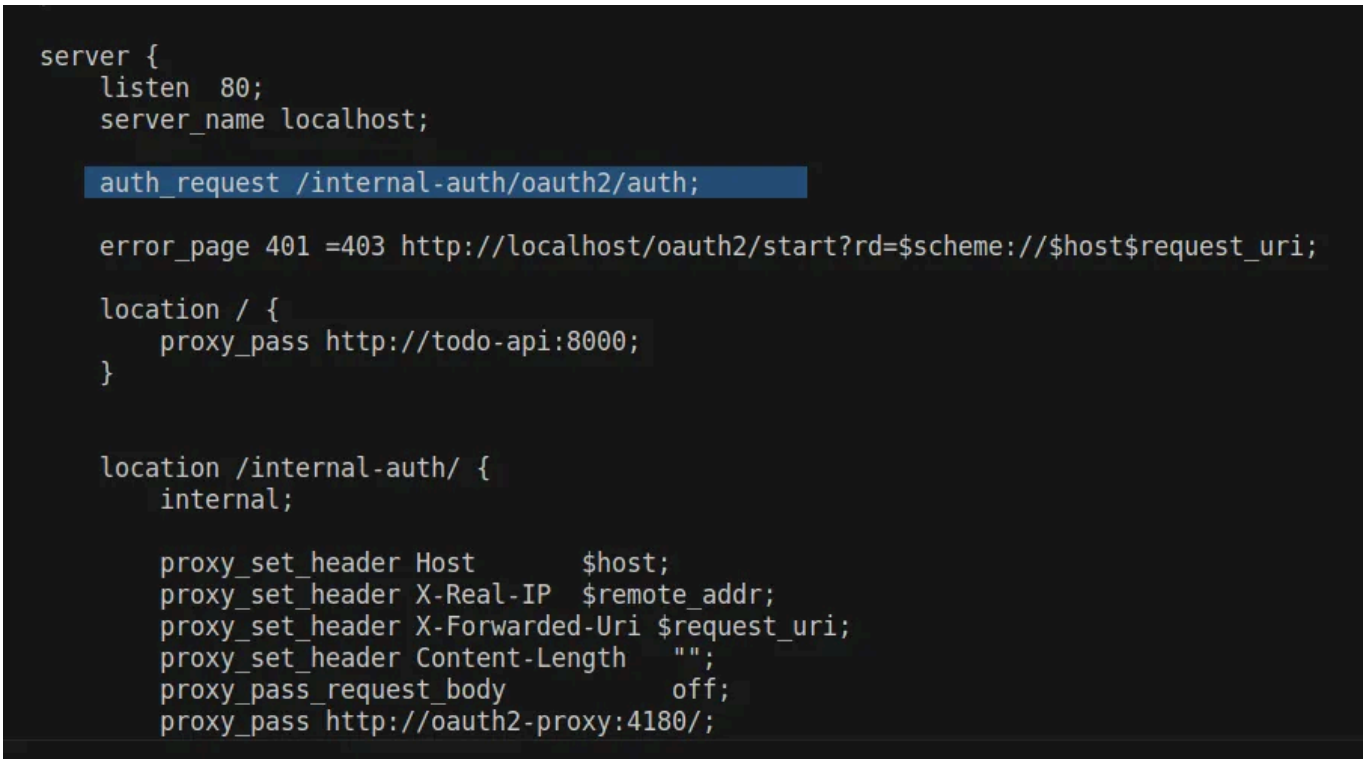
returned request headers with the _oauth2_proxy cookie

Let's discuss what's happening here.



auth flow simplified

1. When a user requests access to a service, it first goes through the NGINX reverse proxy, which checks if the user is authenticated. If you check the `nginx.conf`, you'll see that before proxying to the `todo-api` service, we've added an `auth_request` to verify whether the user is authenticated or not.



```
server {
    listen  80;
    server_name localhost;

    auth_request /internal-auth/oauth2/auth;

    error_page 401 =403 http://localhost/oauth2/start?rd=$scheme://$host$request_uri;

    location / {
        proxy_pass http://todo-api:8000;
    }


    location /internal-auth/ {
        internal;

        proxy_set_header Host       $host;
        proxy_set_header X-Real-IP  $remote_addr;
        proxy_set_header X-Forwarded-Uri $request_uri;
        proxy_set_header Content-Length   "";
        proxy_pass_request_body           off;
        proxy_pass http://oauth2-proxy:4180/;
```

auth_request directive in the nginx.conf file

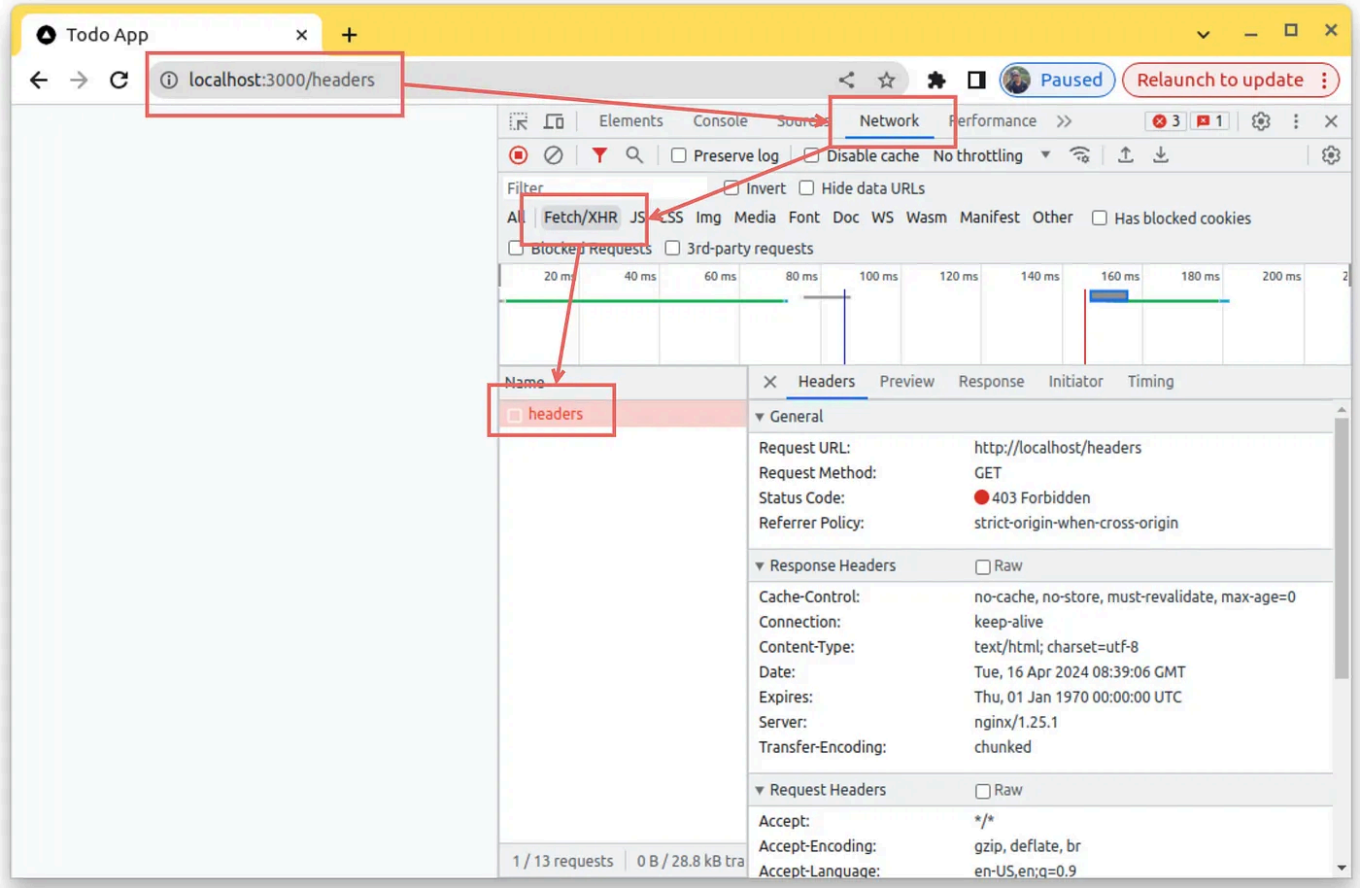
2. Next, OAuth2 Proxy will validate the user authentication via an OAuth2 request with the auth provider as configured. If you look at the `docker-compose.yml`, you'll see all the settings that you've configured there.

You can find all the configuration options in the documentation.
https://oauth2-proxy.github.io/oauth2-proxy/configuration/overview/#command-line-options

3. Then, OAuth2 Proxy will return the authenticated state to NGINX.

4. Depending on the authenticated state (if successful), the user will be allowed to access the service. Otherwise, they will be redirected to the login portal.

If you try to access http://localhost:3000/headers now, you'll see a `403 Forbidden` status for our request. Let's fix that.



Before moving to the frontend, let's ensure that users are redirected to the Google login if they try to directly access our API URLs through NGINX. To do this, uncomment the comment and add the `– skip-provider-button=true` option under the command in the `docker-compose.yml` file.

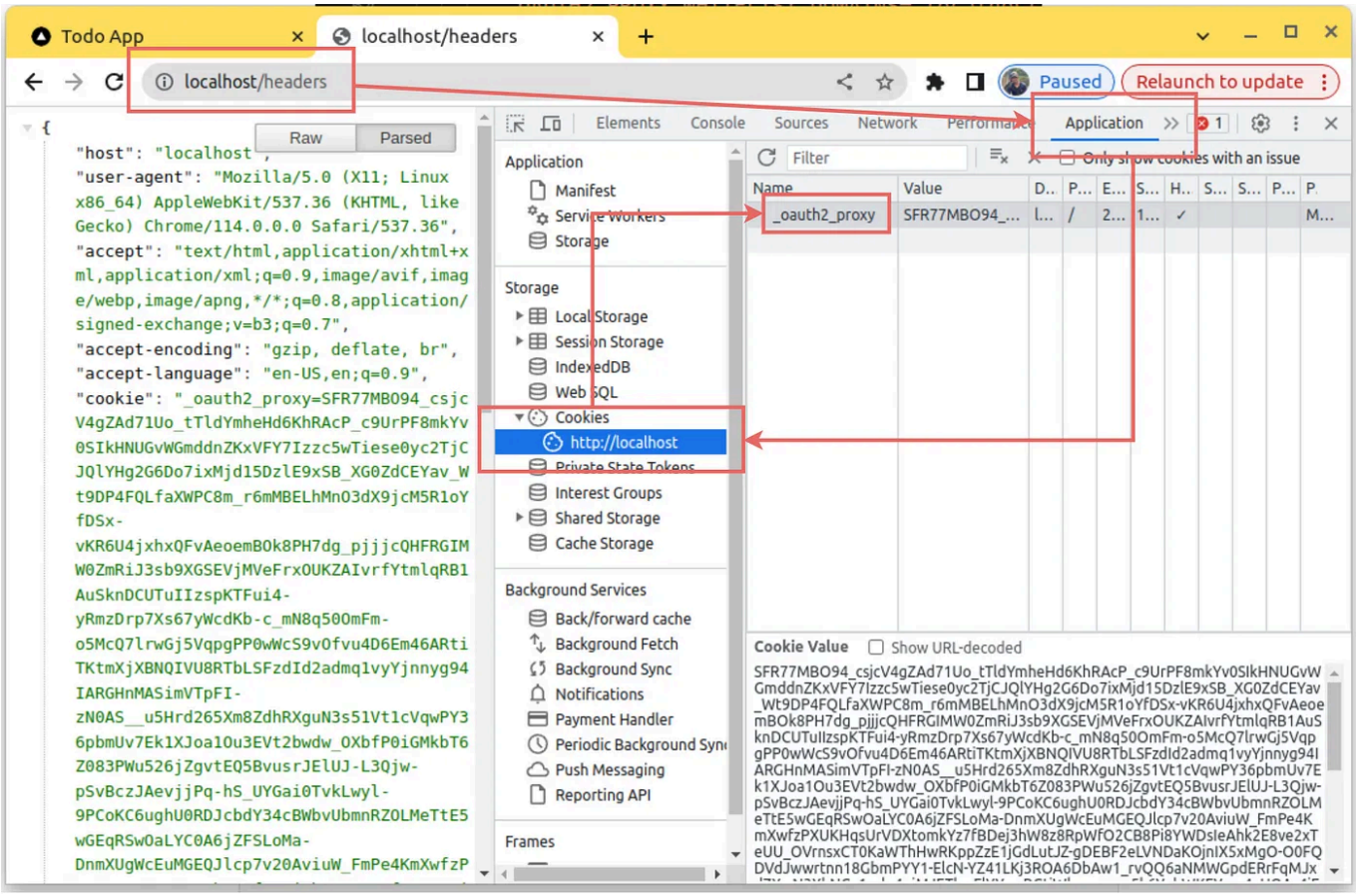
```
64      - OAUTH2_PROXY_WHITELIST_DOMAINS=.localhost
65    command:
66      - --http-address=0.0.0.0:4180
67      - --upstream=http://todo-api:8000
68      # - --skip-provider-button=true
69    networks:
70      - microservices
```

uncommenting the — skip-provider-button=true

Rerun the command `docker compose up -d` to apply the updates made to the `docker-compose.yml` file.

```
docker compose up -d
```

Now, if you try to access `http://localhost/headers`, you'll still be able to see the headers because you've already authenticated and the `_auth2_proxy` cookie is stored in your browser.



You can right-click and manually delete the cookie or use the `http://localhost/oauth2/sign_out` endpoint to clear the cookie.

Now, if you try to access `http://localhost/` again, you will be redirected to the Google login without showing the provider button.

Okay, now let's fix our frontend to work as before.

Fix Todo Client

In order to fetch data correctly from our secured API, we need to pass the `_oauth2_proxy` cookie with our request. We can achieve that by setting **`credentials: "include"`** in the fetch request.

To learn more about the Fetch API, refer to the MDN Web Docs documentation.

Request: credentials property - Web APIs | MDN

The credentials read-only property of the Request interface indicates whether the user agent should send or receive...

developer.mozilla.org

next-todo-app/app/headers/page.tsx

```
...

const response = await fetch('http://localhost/headers', {
    credentials: "include"
})
```

```
...
```

next-todo-app/actions/todo.ts (update 6 todo fetch requests)

```
...

// create a todo
const response = await fetch(API_URL, {
    method: 'post',
    credentials: 'include',
    headers: {
        'Content-Type': 'application/json',
    },
    body: JSON.stringify(data)
})

...
```

```
...

// get todos
const response = await fetch(API_URL, {
    credentials: 'include'
})

...
```

```
...

// delete todo
const response = await fetch(`${API_URL}${id}`, {
    method: 'delete',
    credentials: 'include'
})
```

```
...
```

```
...

// complete todos
const response = await fetch(`${API_URL}${id}/complete`, {
    method: 'post',
    credentials: 'include'
})

...
```

```
...

// withdraw todos
const response = await fetch(`${API_URL}${id}/incomplete`, {
    method: 'post',
    credentials: 'include'
})

...
```

```
...

// update todos
const response = await fetch(`${API_URL}${id}`, {
    method: 'put',
    credentials: 'include',
    headers: {
        'Content-Type': 'application/json',
    },
    body: JSON.stringify(data)
})

...
```

Now, rebuild the `todo-client` service in Docker Compose and rerun the service again to see the updated changes.

```
docker compose build next-todo-app
```

```
docker compose up -d
```

Now, if you access http://localhost:3000/headers or http://localhost:3000/ , and you have a valid `_oauth2_proxy` cookie stored in your browser, you will be able to see todos rendered in the `todo-client` application as you saw before implementing the authentication. (**But still you won't be able to create todos or delete them. We'll fix that later in the project.**)

How can we login from the Next.js todo-client?

In the current setup, to set the `_oauth2_proxy` cookie, we have to access `http://localhost/` .

However, in a real scenario, redirecting users out of your client application like this may not be ideal.

In OAuth2 Proxy, to start the authentication flow, they've given us the `/oauth2/start` endpoint. Invoking that with the `rd` (redirect) URL parameter will help us achieve this task.

Next.js middleware

We can create a file named `middleware.ts` inside the `/next-todo-app/` folder and add the following code snippet to achieve that.

/next-todo-app/middleware.ts

```
import {NextRequest} from "next/server";

export async function middleware(request: NextRequest) {
    const oauthCookie = request.cookies.get('_oauth2_proxy')

    console.log('Current path', request.url)


    if (!oauthCookie && request.nextUrl.pathname !== '/login') {
        return Response.redirect(new URL('/login', request.url));
    }
}

export const config = {
    matcher: ['/', '/headers', '/login']
}
```

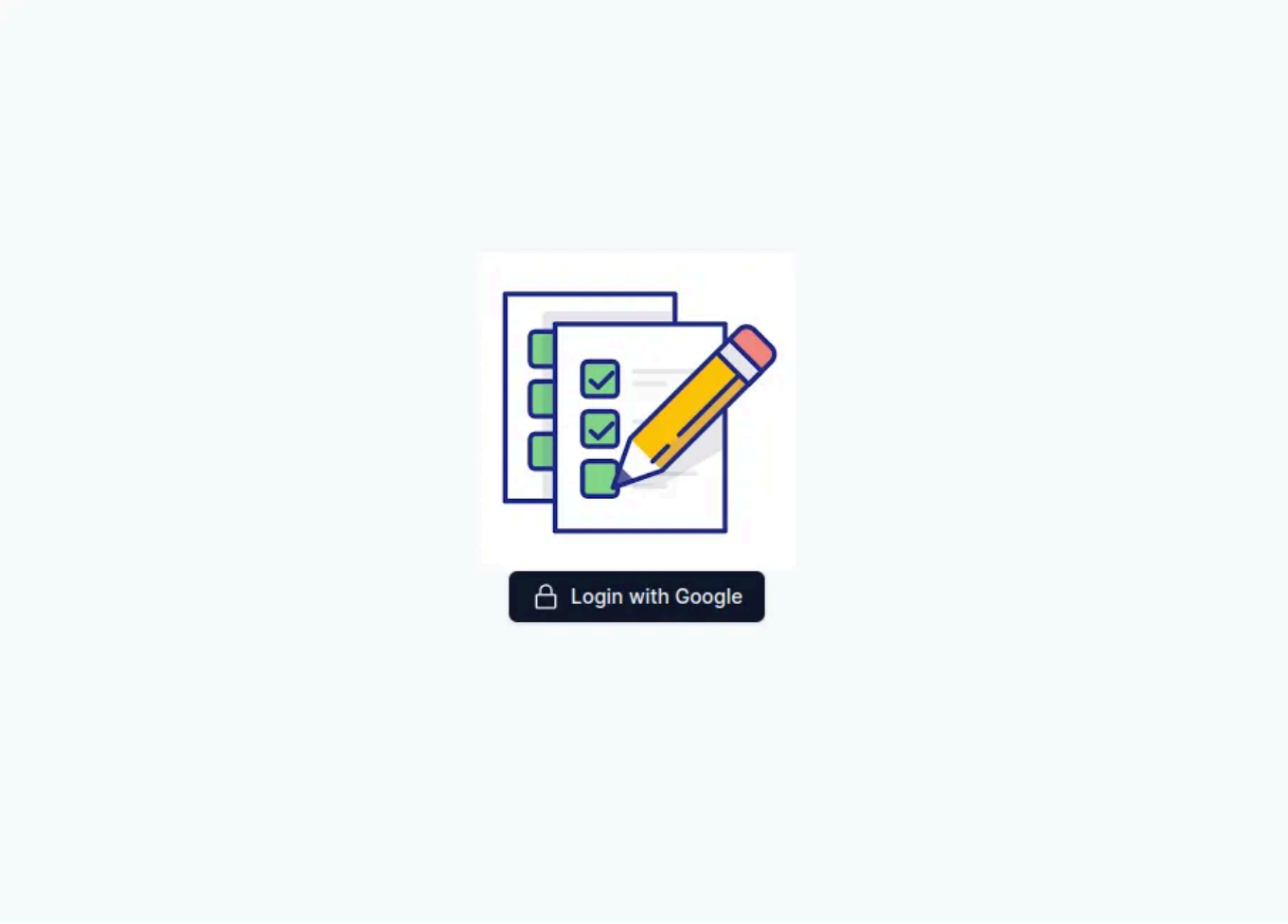
Next Middleware allows you to run code before a request is completed. Then, based on the incoming request, you can modify the response by rewriting, redirecting, modifying the request or response headers, or responding directly.

Routing: Middleware | Next.js

Learn how to use Middleware to run code before a request is completed.

nextjs.org

I've already added a simple login screen with a `Login with Google` button. From `middleware.ts` we will be redirected to that page if we don't have the oauth2 cookie stored in the browser. we can start the auth flow from there.



simple login page for the todo-client

If you want to have a simple logout button, add the following code snippet in your `next-todo-app/app/page.tsx` file.

next-todo-app/app/page.tsx

```
...
```

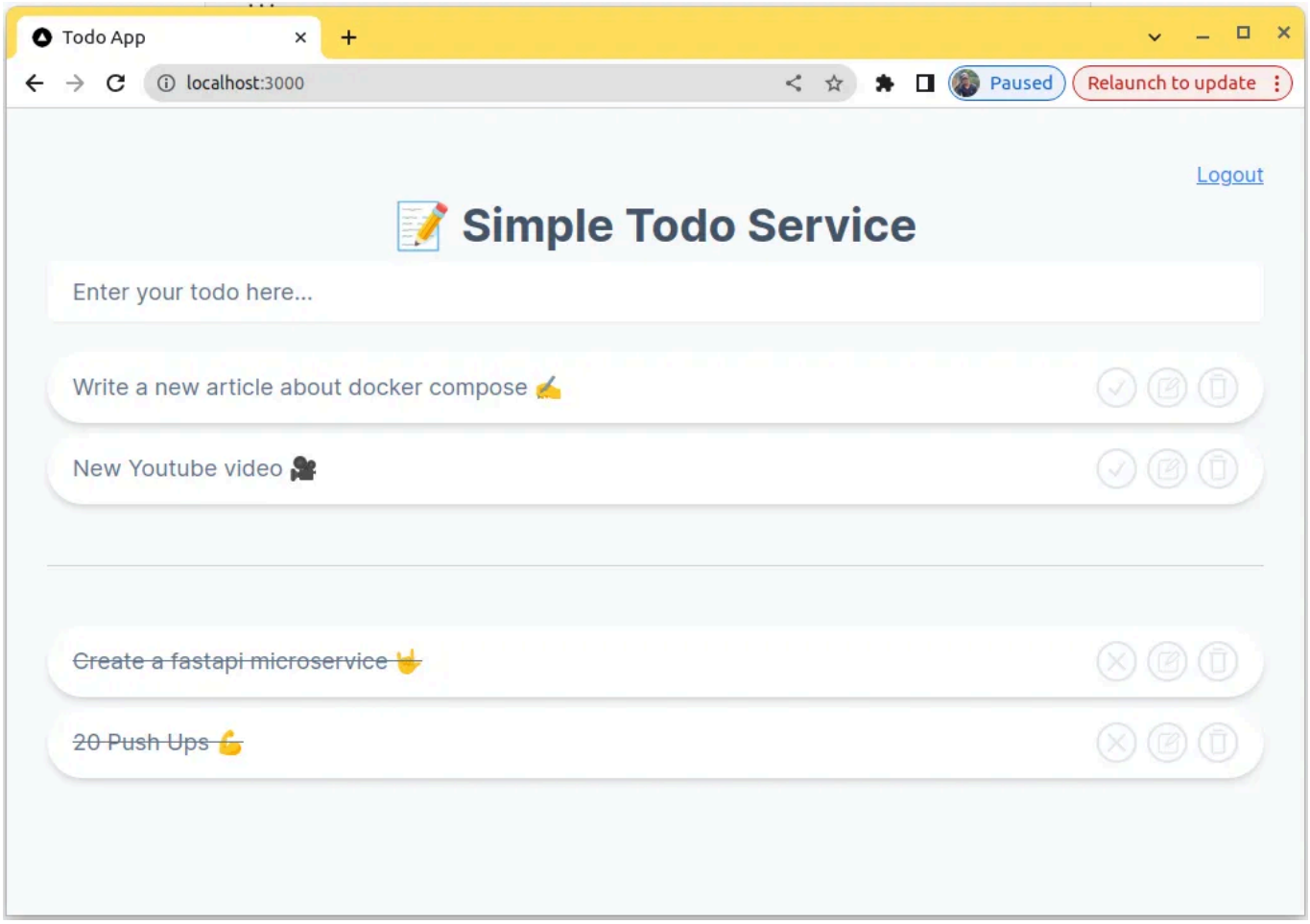
```
<main className="mx-auto max-w-7xl px-4 sm:px-6 lg:px-8 py-10 bg-slate-50 space-
    {/* add logout button here */}
    <div className="flex justify-end">
        <a href="http://localhost/oauth2/sign_out?rd=http%3A%2F%2Flocalhost%3A30
            className="text-blue-500 underline transition cursor-pointer">Logout
    </div>
    {/* logout button end */}

...
```

Now, rebuild the `todo-client` service in Docker Compose and rerun the service again to see the updated changes.

```
docker compose build next-todo-app
```

```
docker compose up -d
```

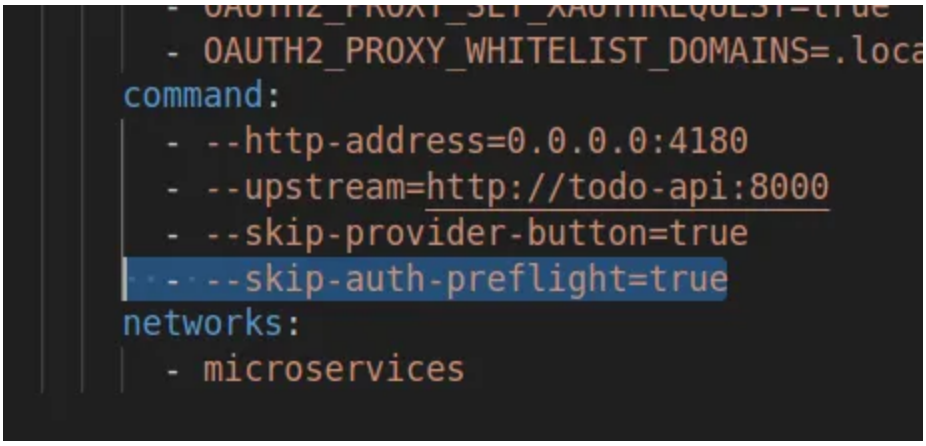
Now, If you access http://localhost:3000 with `_oauth2_proxy` cookie stored in the browser. You'll be able to see your previously entered todos with the newly added logout button.



If you try to interact with the app, you'll notice that you cannot create new todos. This is because OAuth2 Proxy is trying to authenticate preflight requests. You can change that behavior by adding **`– skip-auth-preflight=true`** in your `docker-compose.yml` file.

```
...

      - OAUTH2_PROXY_SET_XAUTHREQUEST=true
      - OAUTH2_PROXY_WHITELIST_DOMAINS=.localhost:3000
    command:
      - --http-address=0.0.0.0:4180
      - --upstream=http://todo-api:8000
      - --skip-provider-button=true
      - --skip-auth-preflight=true
    networks:
      - microservices
```

```
...
```

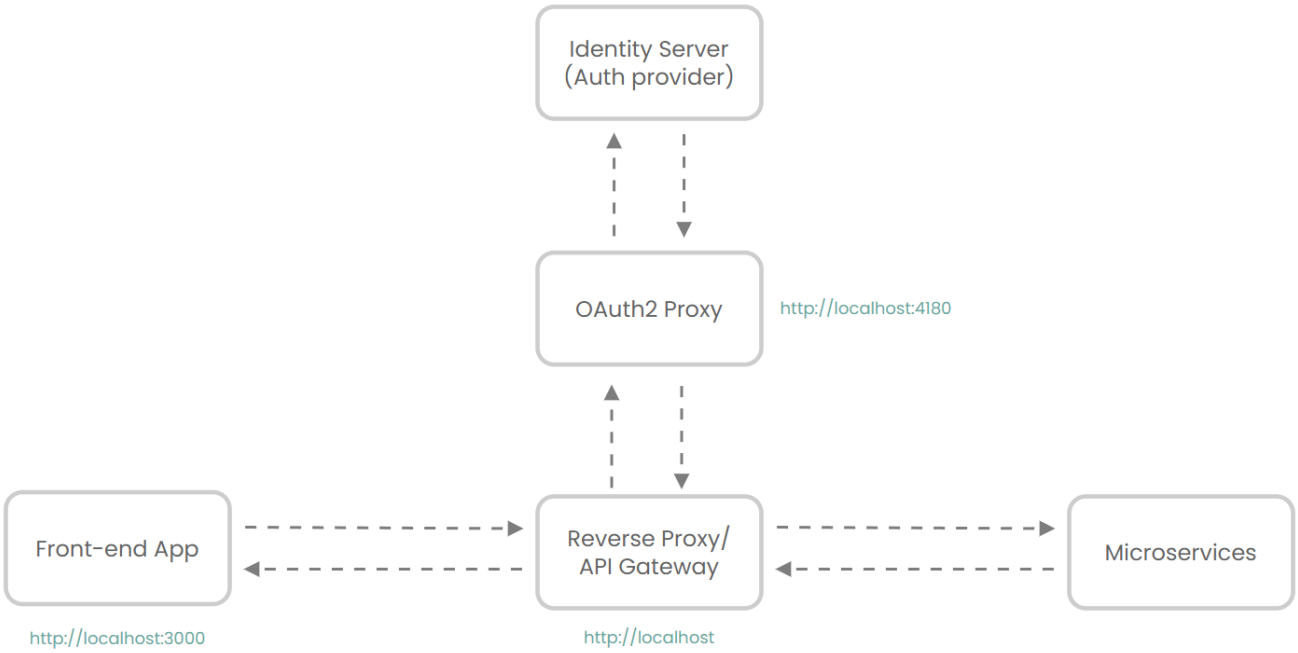


— skip-auth-preflight=true for not to authenticate preflight requests

Rerun all the services to see the effect.

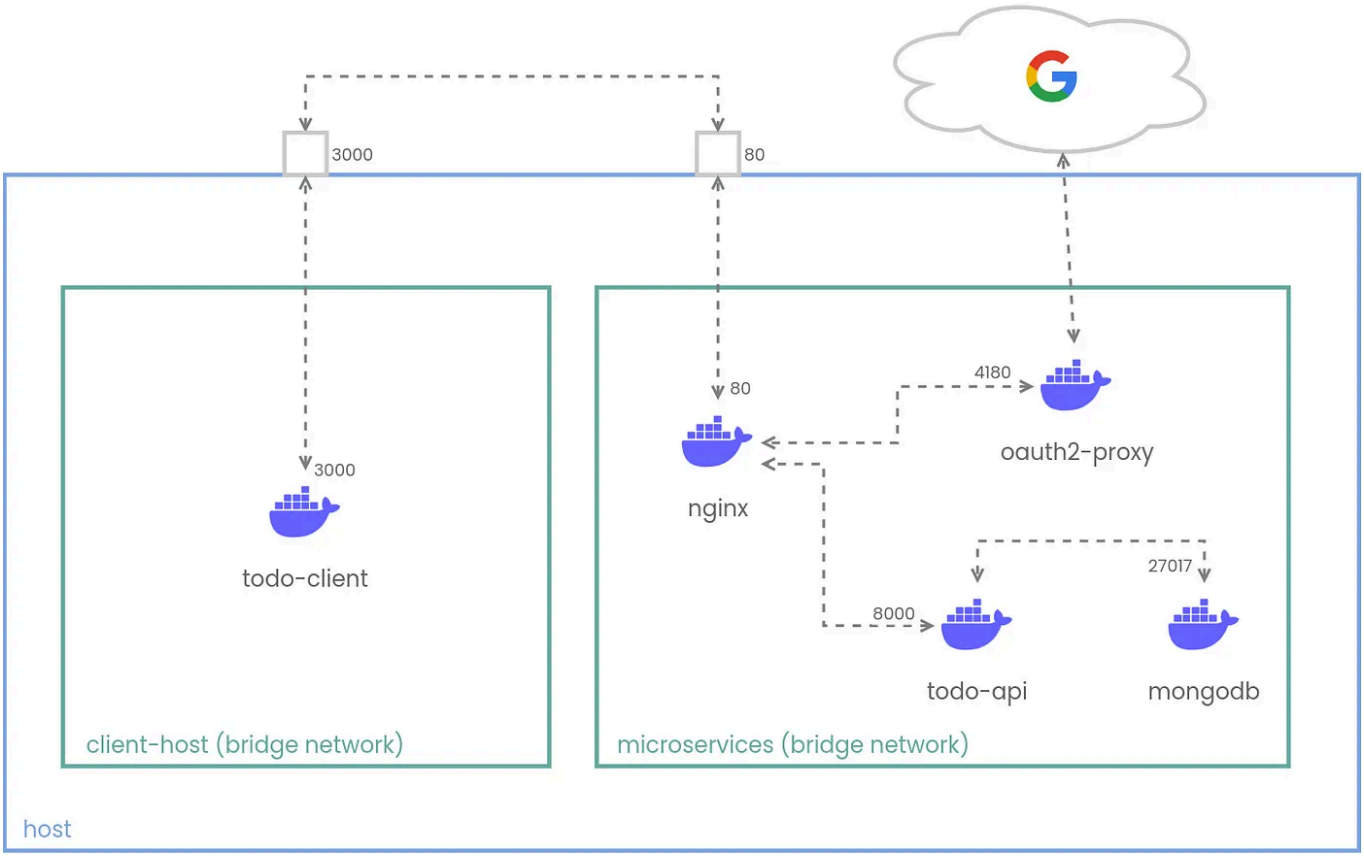
```
docker compose up -d
```

If you have followed all the instructions correctly, you should now be able to log in and log out of your todo application using the todo-client app. You have also completed the following project architecture:



project architecture

Let's have a look at the docker container diagram with network boundaries too.



final container diagram

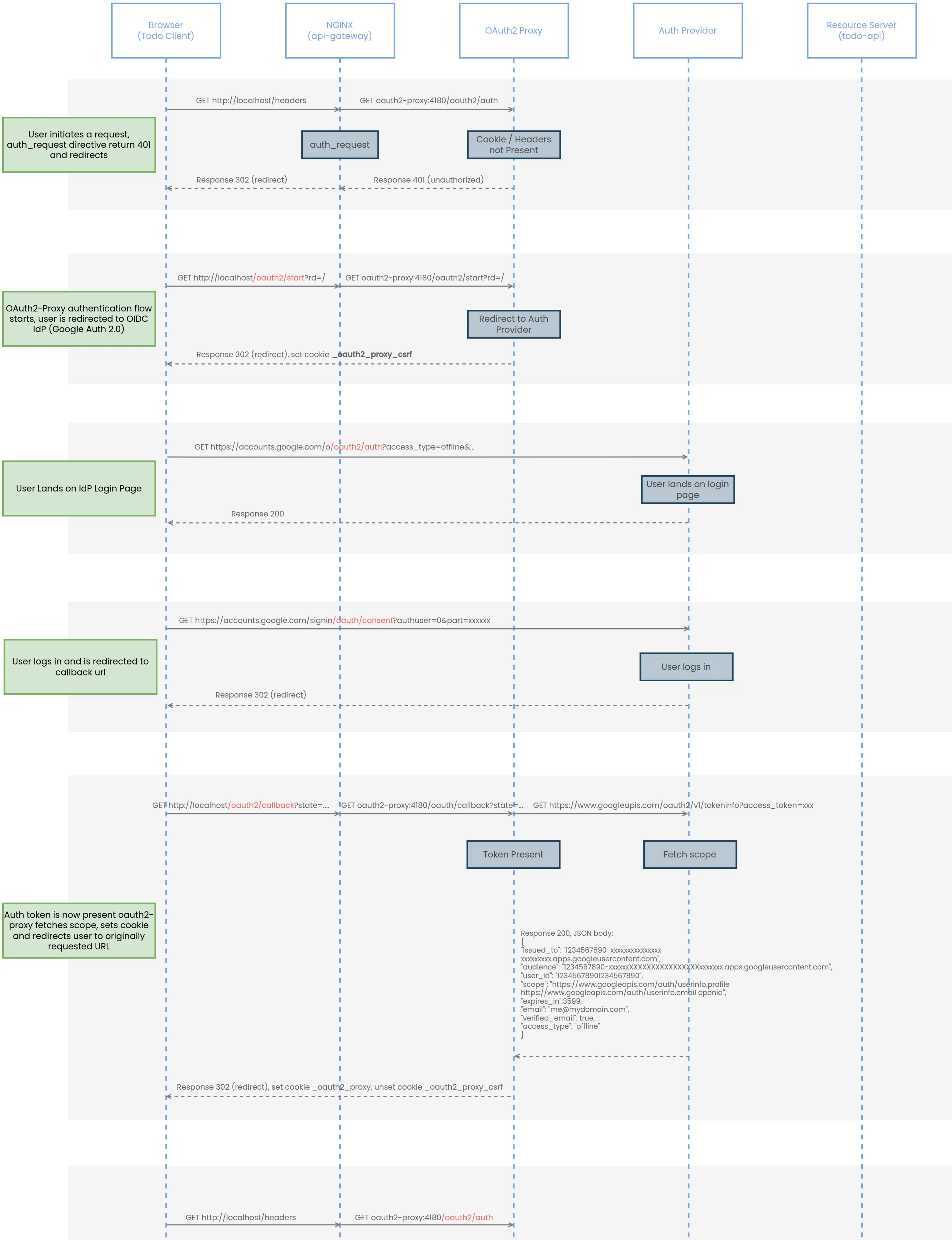
Auth flow — Step 06

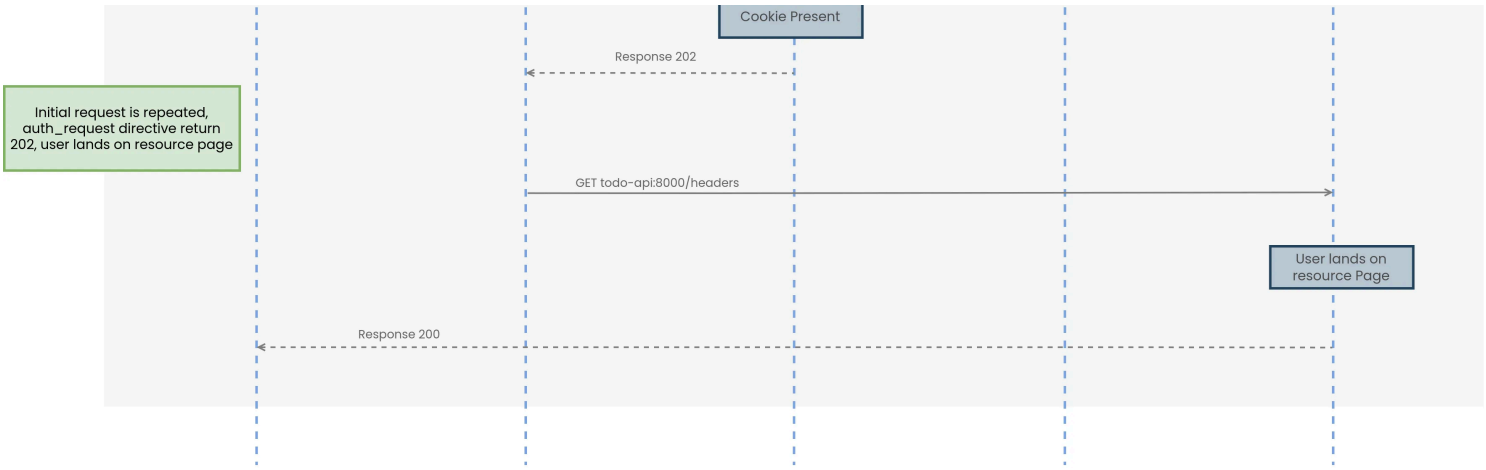
In the diagram below, we explain how a protected API route request is handled through the OAuth2 proxy flow. The user starts by requesting the `/headers` API endpoint. Below diagram is inspired by the 1438th issue in the OAuth2-proxy GitHub repository.

Document/sketch OAuth2-Proxy authentication flow · Issue #1438 · oauth2-proxy/oauth2-proxy

My colleagues and I are currently trying to form a detailed view of how the OAuth2-Proxy authentication flow works, and...

github.com


Browser
(Todo Client)
NGINX
(api-gateway)
OAuth2 Proxy
Auth Provider
Resource Server
(todo-api)
User initiates a request, auth_request directive return 401 and redirects
GET http://localhost/headers
GET oauth2-proxy:4180/oauth2/auth
auth_request
Cookie / Headers not Present
Response 302 (redirect)
Response 401 (unauthorized)
OAuth2-Proxy authentication flow starts, user is redirected to OIDC IdP (Google Auth 2.0)
GET http://localhost/oauth2/start?rd=/
GET oauth2-proxy:4180/oauth2/start?rd=/
Redirect to Auth Provider
Response 302 (redirect), set cookie _oauth2_proxy_csrf
User Lands on IdP Login Page
GET https://accounts.google.com/o/oauth2/auth?access_type=offline&...
User lands on login page
Response 200
User logs in and is redirected to callback url
GET https://accounts.google.com/signin/oauth/consent?authuser=0&part=xxxxxx
User logs in
Response 302 (redirect)
Auth token is now present oauth2-proxy fetches scope, sets cookie and redirects user to originally requested URL
GET http://localhost/oauth2/callback?state=....
GET oauth2-proxy:4180/oauth/callback?state=...
GET https://www.googleapis.com/oauth2/v1/tokeninfo?access_token=xxx
Token Present
Fetch scope
Response 200, JSON body:
{
"issued_to": "1234567890-xxxxxxxxxxxxxxx
xxxxxxxxx.apps.googleusercontent.com",
"audience": "1234567890-xxxxxxXXXXXXXXXXXXXXXXxxxxxxx.apps.googleusercontent.com",
"user_id": "12345678901234567890",
"scope": "https://www.googleapis.com/auth/userinfo.profile
https://www.googleapis.com/auth/userinfo.email openid",
"expires_in":3599,
"email": "me@mydomain.com",
"verified_email": true,
"access_type": "offline"
}
Response 302 (redirect), set cookie _oauth2_proxy, unset cookie _oauth2_proxy_csrf
GET http://localhost/headers
GET oauth2-proxy:4180/oauth2/auth



OAuth2 proxy AUTH FLOW

Congratulations. ☕

Congratulations! If you've made it this far, well done. While I couldn't cover everything I had hoped to in this article, I promise to provide an extension to it. In the next part, I plan to discuss:

- How to handle sign-in, sign-out, and protected routes in the Next.js middleware.
- How to leverage our current setup in the Next.js front-end.
- How to handle authorization in the FastAPI back-end.
- CORS and preflight requests.
- Cross-Site Request Forgery (CSRF) and CSRF cookies.
- OAuth2 proxy refresh cookies, and more options.
- How to add more services to our current setup?

Following that, I plan to continue writing about how to host this kind of architecture in managed services and Kubernetes in the near future.

If you've enjoyed this project, don't forget to give it a 👋. Your support gives me more energy and enthusiasm to write more projects like this. Thank you!

Happy Coding!!! 🐧

Microservices Docker Nginx Nextjs Oauth2

Written by Kesara Karannagoda

Follow

92 Followers · Writer for DevOps.dev

Founder and CEO at Bitzquad | BSc (Hons) Information Systems | Freelance Technical Consultant

More from Kesara Karannagoda and DevOps.dev

Kesara Karannagoda

OAUTH2 Proxy Authentication flow —Part 2

This article explains how the OAuth2 Proxy authentication flow works and explores...

Jul 26 8 1

Obafemi in DevOps.dev

12 Bash Scripts Every DevOps Engineer Should Automate

Let's dive right into it.

Oct 23 346 8

Usama Malik in DevOps.dev

10 Essential SSH Server Security Tips & Best Practices

This article will outline 10 essential SSH server security tips and best practices that will...

Sep 3 284 4

Kesara Karannagoda

Hosting LiveKit Server on Google Cloud: A Step-by-Step Guide 🚀

In a recent project, I had to use Livekit, but I had trouble finding up to date resources on...

Jul 31, 2023 83

See all from Kesara Karannagoda

See all from DevOps.dev

Recommended from Medium

Harendra

How I Am Using a Lifetime 100% Free Server

Get a server with 24 GB RAM + 4 CPU + 200 GB Storage + Always Free

Oct 26 1.5K 21

Saeed Zarinfam in ITNEXT

Reasons behind the recent changes in JetBrains products...

VS Code is getting popular and powerful, and Fleet is getting late!

Oct 28 545 12

Lists

Coding & Development
11 stories · 883 saves

Natural Language Processing
1792 stories · 1400 saves

Obafemi in DevOps.dev

12 Bash Scripts Every DevOps Engineer Should Automate

Let’s dive right into it.

Oct 23 346 8

Dipanshu in AWS in Plain English

Docker pros are shrinking images by 99%: The hidden techniques yo...

Unlock the secrets to lightning-fast deployments and slashed costs—before yo...

Sep 18 2.9K 12

Cleopatra Douglas in Javarevisited

Re-write This Java If-else Code Block Or I’ll Reject Your Pull...

Functional Interfaces to the rescue!

Oct 23 765 23

Chris St. John

UUID Alternatives for Cloud Apps

When UUIDs are not the best solution...

Oct 16 141 13

See more recommendations