How to Implement Authorization into a FastAPI Application

Learn how to implement Authorization in FastAPI applications with Permit.io, a permission management system. Follow a step-by-step guide using RBAC and ABAC.

By Gabriel L. Manor

7 min. read · View original

FastAPI experienced exponential growth lately among Python application developers. It is not only the fastest web framework out there, but also the clean design that allows developers to extend functionality for needs such as authentication and authorization. While authentication (verifying who the users are) is usually a simple task that can be solved with existing plugins, authorization (checking what users can do and see after login) is still a grey area for many developers.

In this article, we will go step by step in building an authorization layer into a FastAPI application. We will start with the basic principles of building it in FastAPI and continue with fully functional scalable authorization implementation. By the end of the article, you'll have the knowledge you need to implement authorization in your FastAPI application. Let's dive in!

The Demo Application

One simple way to demonstrate the various levels of authorization granularity is a simple to-do application. Let's take a look at the following code that declares a simple todo application with FastAPI.

```
@app.get("/tasks")
async def get_tasks():
    return tasks

@app.post("/tasks", dependencies=[Depends(authenticate)])
async def create_task(task: Task):
    tasks.append(task)
    return task

@app.put("/tasks/{task_id}", dependencies=[Depends(authenticate)])
async def update_task(task_id: int, task: Task):
    tasks[task_id - 1] = task
    return task

@app.delete("/tasks/{task_id}", dependencies=[Depends(authenticate)])
async def delete_task(task_id: int):
    task = tasks[task_id - 1]
    tasks.remove(task)
    return task
```

We also created a mock authentication function so we can make sure all the relevant endpoints are protected and only users with verified identities can access them.

```
def authenticate(request: Request, token: str = Depends(token_auth_scheme)):
    return verifyToken(token)
```

With this code in mind, let's continue to design the permission model in our application.

Designing the Authorization Model(s)

Looking at our logic, we can see that there are four endpoints that expose the following operations:

- Get all tasks
- Create a new task
- Update an existing task
- Delete an existing task

Thinking of the permissions required for each operation, we can produce the following table:

Operation	Permission
Get all tasks	Allowed by everyone
Create a new task	Allowed only by admin
Update an existing task	Allowed by all authenticated users
Delete an existing task	Allowed only by admin if the task is marked as done

If you are familiar with permission models, you can see that we need to support two types of permission models:

As we have all the code needed to run the logic and authenticate users, let's continue with the idea of implementing those two permission models into our FastAPI application.

The Authorization Anti-Pattern

One approach for implementing authorization in FastAPI is mixing the policies and permissions with the application logic. For example, we can see developers that create middleware for the endpoint that checks for the relevant permissions in imperative code statements. Here's an example of such permissions check for the delete endpoint in our demo application:

```
def allowed_to_delete_task(task_id: int, user: User):
    task = tasks[task_id - 1]
    return task.done or user.is_admin
```

As you can see, this code is fairly simple and successfully implements the permissions defined in the previous section. Although this code is simple, and this overall approach is common and easy to implement, it has some major drawbacks:

- When we need to perform changes in the model, for example, to allow users to perform operations on their own tasks, we need to change the code in the application logic.
- As we create a logic that is specific to the endpoint, we need to dirt our code with multiple authorization functions.
- If something is changed in the application itself, for example, the task object, we need to rethink the code and change it accordingly.

Let’s examine a better way to implement authorization in FastAPI.

The Authorization Service

The main idea of an authorization service is to decouple the policy and permissions from the application logic. If you think of the previous anti-pattern, we can see that the code we wrote explicitly declares the policy we have for the particular operation. In the authorization service approach, we want to create a generic enforcement point in the application that outsources the policy declaration to an external service.

By using such an authorization service, we can simplify the implementation of our API authorization endpoint to the following code:

```
def authorize(request: Request, token: str = Depends(token_auth_scheme)):
    action = request.method.lower()
    resource = request.path_params if action == "get" else await request.json()

    return authorize_request(token, action, resource)
```

As you can see, the code is very generic and can be used by any application for any endpoint. Not only that, assuming that we have a clean implementation of the authorize_request function, we can seamlessly use it even inside an endpoint function. If, for example, the endpoint scope doesn't have enough information to authorize the request, we can call the authorize_request function everywhere else and get the authorization result. Let's continue by thinking of the right way to implement the authorize_request function.

Configure our Permissions in Permit.io

As you might think of building this authorization service by yourself, there is a simpler way to do it - Using an authorization-as-a-service provider, Permit.io. In the following steps, we will configure all the required permissions for our demo application in a few easy steps.

1. Create a new (and free!) account in Permit.io.
2. Configure our application roles:
3. Configure our application permissions:

As you might notice, at this step, we configured only the RBAC permissions of our application, and every admin user can delete any task. Later in this article, we will demonstrate how to scale the permission model of this app with our requirements without changing any application code. Let's continue with adding the authorize_request abstract function in the form of permit.check in our FastAPI application.

Implementing Permit.io in FastAPI

To make everything simpler, we already created a FastAPI application that is ready to use. Let's continue by doing it interactively in your local Python environment.

First, let's clone the application to your local environment:

```
git clone git@github.com:permitio/permit-fastapi-example.git
```

In the cloned repositories, you'll notice the following files:

Looking at the top of the app.py file, you'll notice the following code:

```
permit = Permit(
    pdp= os.getenv("permit_pdp_url"),
    token= os.getenv("permit_sdk_key")
)

async def authorize(request: Request, token: str = Depends(token_auth_scheme), body={}):
    resource_name = request.url.path.strip('/').split('/')[0]
    method = request.method.lower()
    resource = await request.json() if method in ["post", "put"] else body
    user = token.credentials

    allowed = await permit.check(user, method, {
        "type": resource_name,
        "attributes": resource
    })

    if not allowed:
        raise HTTPException(status_code=403, detail="Not authorized")
```

As we described before, this is the code that is responsible for the authorization, and it only enforces the decision of the authorization service.

To communicate with Permit.io's decision API, we will have to configure the API key in our application. In Permit.io admin app, go to Settings and copy the API Key of your environment to the .env file as the `permit_sdk_key` variable.

Let's run the application and see how it works:

```
uvicorn main:app --reload
```

At this point, you can start test our authorization, do it by run the following commands in a different terminal:

```
curl -X GET http://localhost:8000/tasks
```

As you can see in the terminal output, here are all the mocked tasks that we created on the application startup.

If we try now to create a new task, we'll get an error:

```
curl -X POST http://localhost:8000/tasks \
    -d '{"title": "New Task", "checked": "true"}' \
    -H 'accept: application/json' \
    -H 'Content-Type: application/json'
```

This error, is because the authentication phase, as we didn't provide any token. In our application, we mock the token as the email of the user, let's try to create a new task with the admin user:

```
curl -X POST http://localhost:8000/tasks \
    -d '{"title": "New Task", "checked": "true"}' \
    -H "Authorization: Bearer admin@permit-todo.app" \
    -H 'accept: application/json' \
    -H 'Content-Type: application/json'
```

As you can see, we got a new task created.

What if we will try to run the same request but with a regular user?

```
curl -X POST http://localhost:8000/tasks \
    -d '{"title": "New Task", "checked": "false"}' \
    -H "Authorization: Bearer user@permit-todo.app" \
    -H 'accept: application/json' \
    -H 'Content-Type: application/json'
```

We will get an unauthorized response

```
{
  "detail": "Not authorized"
}
```

Let's try now to call the update endpoint, but now with the regular user:

```
curl -X PUT http://localhost:8000/tasks/4 \
    -d '{"title": "New Task", "checked": "false"}' \
    -H "Authorization: Bearer user@permit-todo.app" \
    -H 'accept: application/json' \
    -H 'Content-Type: application/json'
```

All our permissions work as expected; regular users are allowed to update tasks.

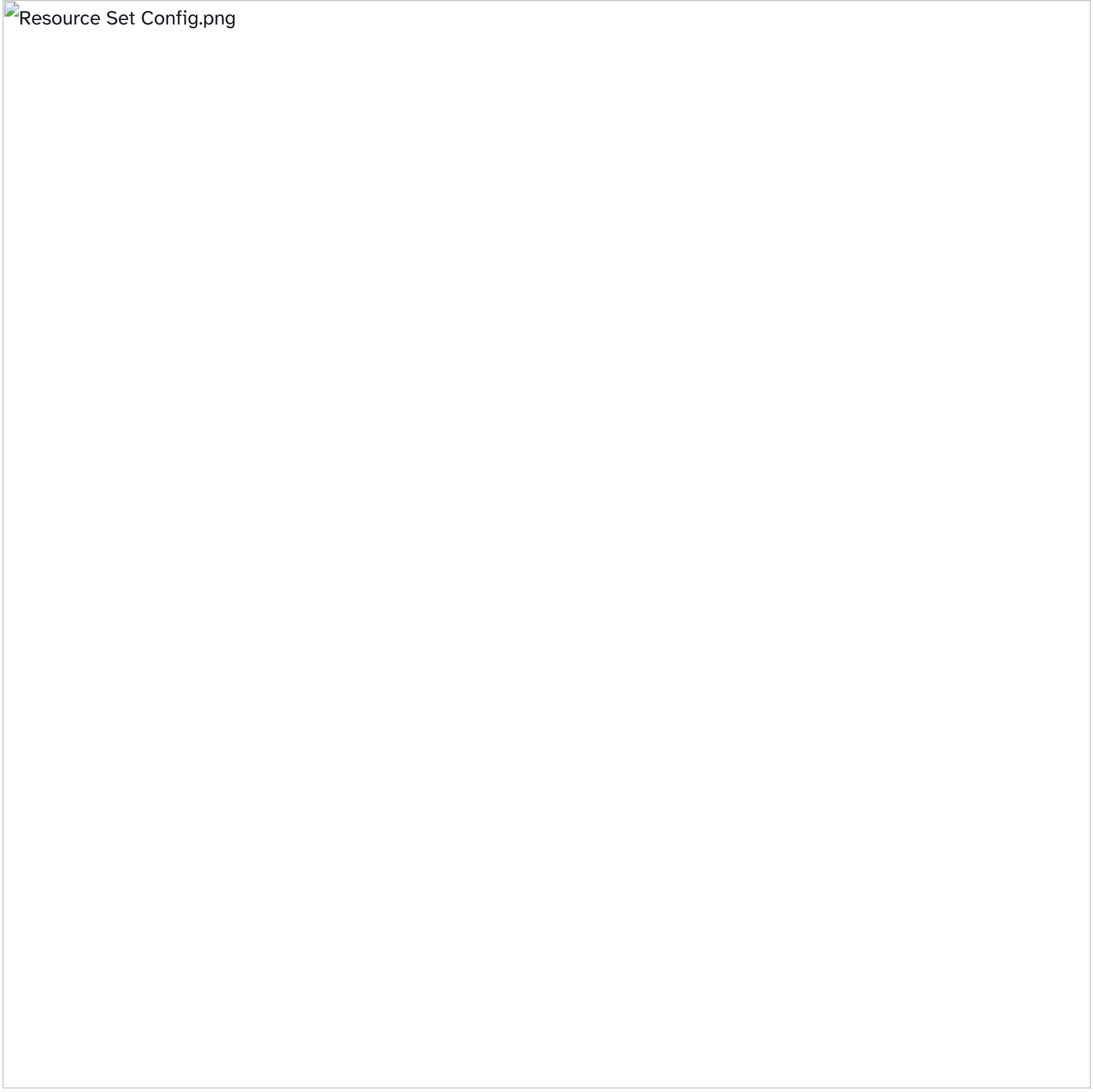
Change Permissions with No Code Changes

The biggest benefit of implementing authorization using this method is the ability to change the policy without changing the application code. We are not only able to change the roles or redefine them, but we could also add support to new permission models. For example, as we described before, our delete endpoint authorization has some more requirements than a simple RBAC check. We want to verify users can delete only their own tasks after they marked them as done.

Let's add this configuration in the Permit app:

1. On the `Policy` page, go to the `ABAC Rules` tab, enable the `ABAC Options` switch, and click `Create New` in the `ABAC Resource Sets` section

2. In the Resource Set dialog, fill the following details:



In this simple configuration, we added a new set of tasks that are owned by the user and marked as done. Let's continue by allowing our users to delete only this resource set by checking the relevant checkboxes on the `Policy` page.

Note: To evaluate the ABAC policy in Permit.io, you should run the PDP service locally. Running the decision engine locally will also help you to make decisions in better performance.

Let's try now to delete a non-checked task:

```
curl -X DELETE http://localhost:8000/tasks/4 \
    -H "Authorization: Bearer admin@permit-todo.app"
```

As you can see, we got an error, as we just configured users are not allowed to delete tasks that are not makred as done.

Trying to delete a task that is owned by the user and marked as done, succeed!

```
curl -X DELETE http://localhost:8000/tasks/4 \
    -H "Authorization: Bearer admin@permit-todo.app"
```

Now, think of the time and effort you'll need to implement this change in your application with the traditional approach. Cool, right?

Conclusion

In this tutorial, we learned how to implement authorization in the FastAPI application using Permit.io. We learned how to implement the authorization service and how to configure it to support our application permissions. We also learned how to use the authorization service in our application and how to change the permissions without changing the code.

A topic we haven't covered much in this article, is the synchronization of your data and authentication provider to Permit.io. To enrich your knowledge of these topics, we invite you to visit Permit.io’s docs to learn more about it.

We also invite you to join our authorization Slack community to discuss ideas and get advice for the right model for your application.