

# Criando Rotas CRUD para Gerenciamento de Tarefas em FastAPI

| <https://fastapidozero.dunossauro.com/09/>

## Objetivos da Aula

- Criação das rotas para as operações CRUD das tarefas
- Fazer com só o usuário dono da tarefa possa acessar e modificar suas tarefas
- Escrita e execução dos testes para cada operação das tarefas

# Como funciona um todo list?

| [https://selenium.dunossauro.com/todo\\_list.html](https://selenium.dunossauro.com/todo_list.html)

## Vamos por partes :)

1. Um novo router para tarefas
2. Uma nova tabela no banco
3. Novos schemas para tarefas
4. Novos endpoints para tarefas

# O primeiro endpoint

| Parte 1

Vamos criar um novo router para os todos em `fast_zero/routers/todos.py`

```
from fastapi import APIRouter

router = APIRouter(prefix='/todos', tags=['todos'])
```

E linkar ele:

```
from fast_zero.routers import auth, todos, users
from fast_zero.schemas import Message

app = FastAPI()

app.include_router(users.router)
app.include_router(auth.router)
app.include_router(todos.router)
```

# Nosso primeiro endpoint

Um endpoint de criação de todos

```
@router.post('/', response_model=???)
def create_todo(todo: ???):
    return ???
```

## Mas e os schemas?

```
# schemas.py
from fast_zero.models import TodoState
# ...
class TodoSchema(BaseModel):
    title: str
    description: str
    state: TodoState

class TodoPublic(TodoSchema):
    id: int
```

```
# models.py
from enum import Enum
# ...
class TodoState(str, Enum):
    draft = 'draft'
    todo = 'todo'
    doing = 'doing'
    done = 'done'
    trash = 'trash'
```



# De volta ao endpoint

```
from typing import Annotated

from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session

from fast_zero.database import get_session
from fast_zero.models import User
from fast_zero.schemas import TodoPublic, TodoSchema
from fast_zero.security import get_current_user

router = APIRouter(prefix='/todos', tags=['todos'])

Session = Annotated[Session, Depends(get_session)]
CurrentUser = Annotated[User, Depends(get_current_user)]

@router.post('/', response_model=TodoPublic)
def create_todo(
    todo: TodoSchema,
    user: CurrentUser,
    session: Session,
):
    return todo
```

## Parte 2

| A tabela dos todos

# A tabela e seu relacionamento

```
from sqlalchemy import ForeignKey, func
# ...
@table_registry.mapped_as_dataclass
class Todo:
    __tablename__ = 'todos'

    id: Mapped[int] = mapped_column(init=False, primary_key=True)
    title: Mapped[str]
    description: Mapped[str]
    state: Mapped[TodoState]

    # Toda tarefa pertence a alguém
    user_id: Mapped[int] = mapped_column(ForeignKey('users.id'))
```

# Juntando o endpoint com o banco de dados

```
from fast_zero.models import Todo, User
# ...
@router.post('/', response_model=TodoPublic)
def create_todo(
    todo: TodoSchema,
    user: CurrentUser,
    session: Session,
):
    db_todo = Todo(
        title=todo.title,
        description=todo.description,
        state=todo.state,
        user_id=user.id,
    )
    session.add(db_todo)
    session.commit()
    session.refresh(db_todo)

    return db_todo
```

# Testes

```
def test_create_todo(client, token):
    response = client.post(
        '/todos/',
        headers={'Authorization': f'Bearer {token}'},
        json={
            'title': 'Test todo',
            'description': 'Test todo description',
            'state': 'draft',
        },
    )
    assert response.json() == {
        'id': 1,
        'title': 'Test todo',
        'description': 'Test todo description',
        'state': 'draft',
    }
```

**Funciona?**

| <http://127.0.0.1:8000/docs>

## Nem tudo são flores

```
sqlalchemy.exc.OperationalError: (sqlite3.OperationalError) no such table: todos  
[SQL: INSERT INTO todos (title, description, state, user_id) VALUES (?, ?, ?, ?)]  
[parameters: ('string', 'string', 'draft', 8)]  
(Background on this error at: https://sqlalche.me/e/20/e3q8)
```

## Executando a migração

```
alembic revision --autogenerate -m "create todos table"  
alembic upgrade head
```



**Funciona?**

| <http://127.0.0.1:8000/docs>

# Parte 3

| O endpoint de GET

## A querystring

Como agora temos vários parâmetros de query como `title`, `description` e `state`, podemos criar um modelo como esse:

```
# fast_zero/schemas.py
class FilterTodo(FilterPage):
    title: str | None = None
    description: str | None = None
    state: TodoState | None = None
```

Uma coisa interessante de observar nesse modelo é que ele usa `FilterPage` como base, para que além dos campos propostos, tenhamos o `limit` e `offset` também.

## Endpoint de GET

```
from fastapi import APIRouter, Depends
from sqlalchemy import select
from fast_zero.schemas import TodoList, TodoPublic, TodoSchema
# ...
@router.get('/', response_model=???)
def list_todos(
    session: Session,
    user: CurrentUser,
    todo_filter: Annotated[FilterTodo, Query()],
): ...
```

Vamos olhar o swagger

| [http://127.0.0.1:8000/docs#/todos/list\\_todos\\_todos\\_\\_get](http://127.0.0.1:8000/docs#/todos/list_todos_todos__get)

## Pegando os valores da query

```
@router.get('/', response_model=TodoList) # implementar
def list_todos(...):
    query = select(Todo).where(Todo.user_id == user.id)

    if title: # o título contém
        query = query.filter(Todo.title.contains(title))

    if description: # a descrição contém
        query = query.filter(Todo.description.contains(description))

    if state: # o estado é igual
        query = query.filter(Todo.state == state)

    todos = session.scalars(query.offset(offset).limit(limit)).all()

    return {'todos': todos}
```

## O schema

```
class TodoList(BaseModel):  
    todos: list[TodoPublic]
```

## Testes para o GET

- Devemos testar os parâmetros dos paths
- Pra validar os filtros, precisamos de N 'todos'
- Factory-boy vai nos ajudar aqui!



# O factory

```
# tests/test_todos.py
import factory.fuzzy
from fast_zero.models import Todo, TodoState
# ...
class TodoFactory(factory.Factory):
    class Meta:
        model = Todo

    title = factory.Faker('text')
    description = factory.Faker('text')
    state = factory.fuzzy.FuzzyChoice(TodoState)
    user_id = 1
```

- **FuzzyChoice**: Fuzzy é uma forma de escolher randomicamente algo, no caso um TodoState

## O primeiro teste

```
def test_list_todos_should_return_5_todos(session, client, user, token):
    expected_todos = 5
    session.bulk_save_objects(TodoFactory.create_batch(5, user_id=user.id))
    session.commit()

    response = client.get(
        '/todos/', # sem query
        headers={'Authorization': f'Bearer {token}'},
    )

    assert len(response.json()['todos']) == expected_todos
```

## offset e limit

```
def test_list_todos_pagination_should_return_2_todos(
    session, user, client, token
):
    expected_todos = 2
    session.bulk_save_objects(TodoFactory.create_batch(5, user_id=user.id))
    session.commit()

    response = client.get(
        '/todos/?offset=1&limit=2',
        headers={'Authorization': f'Bearer {token}'},
    )

    assert len(response.json()['todos']) == expected_todos
```

## Por título

```
def test_list_todos_filter_title_should_return_5_todos(
    session, user, client, token
):
    expected_todos = 5
    session.bulk_save_objects(
        TodoFactory.create_batch(5, user_id=user.id, title='Test todo 1')
    )
    session.commit()

    response = client.get(
        '/todos/?title=Test todo 1',
        headers={'Authorization': f'Bearer {token}'},
    )

    assert len(response.json()['todos']) == expected_todos
```

## Filtro por descrição

```
def test_list_todos_filter_description_should_return_5_todos(
    session, user, client, token
):
    expected_todos = 5
    session.bulk_save_objects(
        TodoFactory.create_batch(5, user_id=user.id, description='description')
    )
    session.commit()

    response = client.get(
        '/todos/?description=desc',
        headers={'Authorization': f'Bearer {token}'},
    )

    assert len(response.json()['todos']) == expected_todos
```

## Filtro por estado

```
def test_list_todos_filter_state_should_return_5_todos(
    session, user, client, token
):
    expected_todos = 5
    session.bulk_save_objects(
        TodoFactory.create_batch(5, user_id=user.id, state=TodoState.draft)
    )
    session.commit()

    response = client.get(
        '/todos/?state=draft',
        headers={'Authorization': f'Bearer {token}'},
    )

    assert len(response.json()['todos']) == expected_todos
```

# Parte 4

| O delete

## O Delete

```
@router.delete('/{todo_id}', response_model=Message)
def delete_todo(todo_id: int, session: Session, user: CurrentUser):
    todo = session.scalar(
        select(Todo).where(Todo.user_id == user.id, Todo.id == todo_id)
    )

    if not todo:
        raise HTTPException(
            status_code=HTTPStatus.NOT_FOUND, detail='Task not found.'
        )

    session.delete(todo)
    session.commit()

    return {'message': 'Task has been deleted successfully.'}
```



## Testando o delete

```
def test_delete_todo(session, client, user, token):
    todo = TodoFactory(user_id=user.id)
    session.add(todo)
    session.commit()

    response = client.delete(
        f'/todos/{todo.id}', headers={'Authorization': f'Bearer {token}'}
    )

    assert response.status_code == HTTPStatus.OK
    assert response.json() == {'message': 'Task has been deleted successfully.'}
```

```
def test_delete_todo_error(client, token):
    response = client.delete(
        f'/todos/{10}', headers={'Authorization': f'Bearer {token}'}
    )

    assert response.status_code == HTTPStatus.NOT_FOUND
    assert response.json() == {'detail': 'Task not found.'}
```

# Parte 5

| O endpoint de alteração via Patch

## O Patch

O `Patch`, diferente do verbo `PUT` permite que somente os dados a serem alterados sejam enviados. Para isso precisamos de um novo schema:

```
class TodoUpdate(BaseModel):  
    title: str | None = None  
    description: str | None = None  
    state: TodoState | None = None
```

## O endpoint

```
@router.patch('/{todo_id}', response_model=TodoPublic)
def patch_todo(
    todo_id: int, session: Session, user: CurrentUser, todo: TodoUpdate
):
    db_todo = session.scalar(
        select(Todo).where(Todo.user_id == user.id, Todo.id == todo_id)
    )

    if not db_todo:
        raise HTTPException(
            status_code=HTTPStatus.NOT_FOUND, detail='Task not found.'
        )

    for key, value in todo.model_dump(exclude_unset=True).items():
        setattr(db_todo, key, value)

    session.add(db_todo)
    session.commit()
    session.refresh(db_todo)

    return db_todo
```

## Testando o patch

```
def test_patch_todo(session, client, user, token):
    todo = TodoFactory(user_id=user.id)

    session.add(todo)
    session.commit()

    response = client.patch(
        f'/todos/{todo.id}',
        json={'title': 'teste!'},
        headers={'Authorization': f'Bearer {token}'},
    )
    assert response.status_code == HTTPStatus.OK
    assert response.json()['title'] == 'teste!'
```

```
def test_patch_todo_error(client, token):
    response = client.patch(
        '/todos/10',
        json={},
        headers={'Authorization': f'Bearer {token}'},
    )
    assert response.status_code == HTTPStatus.NOT_FOUND
    assert response.json() == {'detail': 'Task not found.'}
```

# Exercícios

1. Adicione os campos `created_at` e `updated_at` na tabela `Todo`
  - Eles devem ser `init=False`
  - Deve usar `func.now()` para criação
  - O campo `updated_at` deve ter `onupdate`
2. Criar uma migração para que os novos campos sejam versionados e também aplicar a migração

# Exercícios

3. Adicionar os campos `created_at` e `updated_at` no schema de saída dos endpoints. Para que esse valores sejam retornados na API. Essa alteração deve ser refletida nos testes também!
4. Crie um teste para o endpoint de busca (GET) que valide todos os campos contidos no `Todo` de resposta. Até o momento, todas as validações foram feitas pelo tamanho do resultado de todos.

# Quiz

Não esqueça de responder ao quiz dessa aula



# Commit

```
git add .  
git commit -m "Implementado os endpoints de tarefas"
```