```
class User(Document):
    email: EmailStr
    password: str
    events: Optional[List[Link[Event]]]

    class Settings:
        name = "users"

    class Config:
        schema_extra = {
            "example": {
                "email": "fastapi@packt.com",
                "password": "strong!!!",
                "events": [],
            }
        }

class UserSignIn(BaseModel):
    email: EmailStr
    password: str
```

6. Now that we have defined the documents, let's update the `document_models` field in `connection.py`:

```
from models.users import User
from models.events import Event

async def initialize_database(self):
        client = AsyncIOMotorClient(self.DATABASE_URL)
        await init_beanie(
        database=client.get_default_database(),
                        document_models=[Event, User])
```

7. Lastly, let's create an environment file, `.env`, and add the database URL to finalize the database initialization stage:

```
(venv)$ touch .env
(venv)$ echo DATABASE_URL=mongodb://localhost:27017/
planner >> .env
```

Now that we have successfully added the blocks of code to initialize the database, let's proceed to implement the methods for CRUD operations.

CRUD operations

In `connection.py`, create a new `Database` class that takes a model as an argument during initialization:

```
from pydantic import BaseSettings, BaseModel
from typing import Any, List, Optional

class Database:
    def __init__(self, model):
        self.model = model
```

The model passed during initialization is either the `Event` or `User` document model class.

Create

Let's create a method under the `Database` class to add a record to the database collection:

```
    async def save(self, document) -> None:
        await document.create()
            return
```

In this code block, we have defined the `save` method to take the document, which will be an instance of the document passed to the `Database` instance at the point of instantiation.

Read

Let's create the methods to retrieve a database record or all the records present in the database collection:

```
    async def get(self, id: PydanticObjectId) -> Any:
        doc = await self.model.get(id)
        if doc:
            return doc
```

```
        return False

    async def get_all(self) -> List[Any]:
        docs = await self.model.find_all().to_list()
        return docs
```

The first method, `get()`, takes an ID as the method argument and returns a corresponding record from the database, while the `get_all()` method takes no argument and returns a list of all the records present in the database.

Update

Let's create the method to handle the process of updating an existing record:

```
    async def update(self, id: PydanticObjectId, body:
    BaseModel) -> Any:
        doc_id = id
        des_body = body.dict()
        des_body = {k:v for k,v in des_body.items() if v is
        not None}
        update_query = {"$set": {
            field: value for field, value in
            des_body.items()
        }}

        doc = await self.get(doc_id)
        if not doc:
            return False
        await doc.update(update_query)
        return doc
```

In this code block, the `update` method takes an ID and the Pydantic schema responsible, which will contain the fields updated from the `PUT` request sent by the client. The updated request body is first parsed into a dictionary and then filtered to remove `None` values. Once this has been done, it is then inserted into an update query, which is finally executed by Beanie's `update()` method.

Delete

Lastly, let's create a method to delete a record from the database:

```
    async def delete(self, id: PydanticObjectId) -> bool:
        doc = await self.get(id)
        if not doc:
            return False
        await doc.delete()
        return True
```

In this code block, the method checks whether such a record exists before proceeding to delete it from the database.

Now that we have populated our database file with the necessary methods needed to carry out CRUD operations, let's update the routes as well.

routes/events.py

Let's start by updating the imports and creating a `database` instance:

```
from beanie import PydanticObjectId
from fastapi import APIRouter, HTTPException, status
from database.connection import Database

from models.events import Event
from typing import List
event_database = Database(Event)
```

With the imports and database instance in place, let's update all the routes. Start by updating the `GET` routes:

```
@event_router.get("/", response_model=List[Event])
async def retrieve_all_events() -> List[Event]:
    events = await event_database.get_all()
    return events

@event_router.get("/{id}", response_model=Event)
async def retrieve_event(id: PydanticObjectId) -> Event:
    event = await event_database.get(id)
    if not event:
```

```
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Event with supplied ID does not exist"
        )
    return event
```

In the `GET` routes, we are invoking the methods we defined in the database file earlier. Let's update the `POST` routes:

```
@event_router.post("/new")
async def create_event(body: Event) -> dict:
    await event_database.save(body)
    return {
        "message": "Event created successfully"
    }
```

Let's create the `UPDATE` route:

```
@event_router.put("/{id}", response_model=Event)
async def update_event(id: PydanticObjectId, body: EventUpdate)
-> Event:
    updated_event = await event_database.update(id, body)
    if not updated_event:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Event with supplied ID does not exist"
        )
    return updated_event
```

Lastly, let's update the `DELETE` route:

```
@event_router.delete("/{id}")
async def delete_event(id: PydanticObjectId) -> dict:
    event = await event_database.delete(id)
    if not event:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Event with supplied ID does not exist"
        )
```