Architecture retenue (routes / controllers / services)

Structure :

```
src/
 ├── prisma.ts
 ├── routes/
 │     └── user.routes.ts
 ├── controllers/
 │     └── user.controller.ts
 └── services/
       └── user.service.ts
```

---

1. Service (logique métier + DB)

Rôle :

1. Parler à Prisma
2. Contenir la logique métier
3. Ne PAS connaître Fastify

```
import { prisma } from "../prisma"

export async function createUser(data: {
  name: string
  email: string
}) {
  return prisma.user.create({ data })
}

export async function getUsers() {
  return prisma.user.findMany()
}
```

---

2. Controller (interface HTTP)

Rôle :

1. Lire request
2. Appeler le service
3. Gérer la réponse HTTP

```
import { FastifyRequest, FastifyReply } from "fastify"
import * as userService from "../services/user.service"

export async function createUser(
  request: FastifyRequest,
```

```
  reply: FastifyReply
) {
  const user = await userService.createUser(
    request.body as any
  )

  return reply.status(201).send(user)
}

export async function getUsers(
  request: FastifyRequest,
  reply: FastifyReply
) {
  const users = await userService.getUsers()
  return reply.send(users)
}
```

---

3. Routes (déclaration des endpoints)

Rôle :

1. Définir les URLs
2. Relier route → controller

```
import { FastifyInstance } from "fastify"
import * as userController from "../controllers/user.controller"

export async function userRoutes(app: FastifyInstance) {

  app.post("/users", userController.createUser)

  app.get("/users", userController.getUsers)

}
```

---

4. prisma.ts (instance unique)

```
import { PrismaClient } from "@prisma/client"

export const prisma = new PrismaClient()
```

---

5. Rappel des responsabilités

Routes :

```
→ mapping URL
```

Controllers :

```
→ gestion HTTP
```

Services :

```
→ logique métier + DB
```

Prisma :

```
→ accès base de données
```

Factorisation

Services

Pour le CRUD :

```
class BaseService<T> {
  constructor(private model) {}

  findAll() {
    return this.model.findMany()
  }

  findById(id: string) {
    return this.model.findUnique({ where: { id } })
  }

  create(data: any) {
    return this.model.create({ data })
  }

  update(id: string, data: any) {
    return this.model.update({
      where: { id },
      data
    })
  }

  delete(id: string) {
    return this.model.delete({
      where: { id }
    })
  }
}
```

Spécialisation par ressource

```
class EventService extends BaseService<Event> {
  constructor() {
    super(prisma.event)
  }

  async joinEvent(eventId: string, userId: string) {
    // logique métier spécifique
  }
}
```

Controller

Faire un helper :

```
const handle = (fn) => async (req, res) => {
  try {
    const result = await fn(req)
    res.send(result)
  } catch (e) {
    res.status(500).send({ error: e.message })
  }
}
```

Pour éviter les try/catch partout

```
const getEvents = handle(async (req) => {
  return eventService.findAll()
})
```

Conclusion

- Pas de classes
- Fonctions simples
- Séparation claire
- Architecture maintenable
- Production ready

Suite >>