

Web 2 Reloaded

Concepts Théoriques du Backend Moderne

TypeScript + Express + Prisma

Plan de la Présentation

1. Introduction aux Technologies
2. Architecture Backend Moderne
3. TypeScript : Typage Statique
4. Express.js : Framework Web
5. Prisma : ORM Moderne
6. Patterns et Bonnes Pratiques
7. Sécurité et Production

Pourquoi ces Technologies ?

Problèmes du JavaScript Vanilla

- Absence de typage → erreurs à l'exécution
- Code difficile à maintenir
- Refactoring risqué
- Documentation insuffisante

Solutions Modernes

- **TypeScript** : Typage statique
- **Express** : Framework mature et flexible
- **Prisma** : ORM type-safe et moderne

TypeScript : Le JavaScript Typé

Avantages

- 🔍 Détection d'erreurs à la compilation
- 🧠 IntelliSense amélioré dans l'IDE
- 📖 Auto-documentation du code
- 🔄 Refactoring sécurisé
- 👥 Collaboration facilitée en équipe

Concepts Clés

```
// Types primitifs  
let nom: string = "Moustache";  
let age: number = 25;  
let actif: boolean = true;
```

```
// Interfaces
```

Express.js : Le Framework Web

Philosophie

- **Minimaliste** : ne fournit que l'essentiel
- **Middleware-centric** : architecture modulaire
- **Flexible** : s'adapte à tous les besoins
- **Mature** : écosystème riche

Architecture Middleware

```
app.use(express.json());           // Parser JSON
app.use(helmet());                 // Sécurité
app.use(cors());                   // CORS
app.use('/api', routesAPI);        // Routes
app.use(errorHandler);             // Gestion erreurs
```

Middleware : Le Cœur d'Express

Concept

Fonctions qui s'exécutent **séquentiellement** pour chaque requête

Types de Middleware

- Application-level : `app.use()`
- Router-level : `router.use()`
- Error-handling : `(err, req, res, next) => {}`
- Built-in : `express.json()`, `express.static()`
- Third-party : `helmet`, `cors`, `morgan`






Flux d'une Requête

Prisma : L'ORM Moderne

Qu'est-ce qu'un ORM ?

Object-Relational Mapping : Pont entre objets et base de données

Avantages de Prisma

-  **Type-safe** : Pas d'erreurs de requête
-  **Schema-first** : Base de données comme source de vérité
-  **Performance** : Requêtes optimisées
-  **Developer Experience** : Outils excellents
-  **Multi-database** : SQLite, PostgreSQL, MySQL, etc.

Prisma Schema : La Source de Vérité

Structure

```
generator client {
  provider = "prisma-client-js"
}

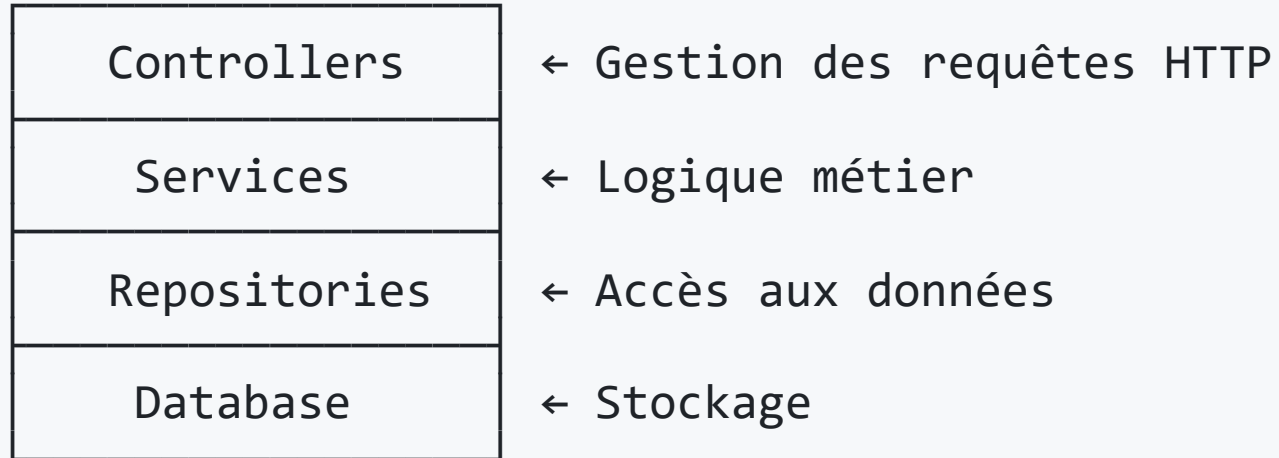
datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}

model User {
  id          Int      @id @default(autoincrement())
  email       String    @unique
  name        String?
  posts       Post[]
  createdAt   DateTime @default(now())
}

model Post {
  id          Int      @id @default(autoincrement())
  title       String
  content     String?
  author      User     @relation(fields: [authorId], references: [id])
  authorId    Int
}
```


Architecture en Couches

Séparation des Responsabilités



Controller Layer

Responsabilités

- Validation des entrées
- Orchestration des services
- Formatage des réponses
- Gestion des codes de statut HTTP

Exemple

```
export class MustacheController {  
  constructor(private mustacheService: MustacheService) {}  
  
  async getAllMustaches(req: Request, res: Response) {  
    try {  
      const mustaches = await this.mustacheService.findAll();  
      res.json({ success: true, data: mustaches });  
    } catch (error) {
```

Service Layer

Responsabilités

- Logique métier complexe
- Validation des règles business
- Orchestration entre différents repositories
- Transformation des données

Exemple

```
export class MustacheService {  
  constructor(private mustacheRepo: MustacheRepository) {}  
  
  async findAll(): Promise<Mustache[]> {  
    const mustaches = await this.mustacheRepo.findMany();  
    return mustaches.map(m => this.formatMustache(m));  
  }  
}
```

Repository Pattern

Abstraction de l'Accès aux Données

```
interface MustacheRepository {  
    findMany(): Promise<Mustache[]>;  
    findById(id: number): Promise<Mustache | null>;  
    create(data: CreateMustacheDto): Promise<Mustache>;  
    update(id: number, data: UpdateMustacheDto): Promise<Mustache>;  
    delete(id: number): Promise<void>;  
}  
  
class PrismaMustacheRepository implements MustacheRepository {  
    constructor(private prisma: PrismaClient) {}  
  
    async findMany(): Promise<Mustache[]> {  
        return this.prisma.mustache.findMany();  
    }  
    // ... autres méthodes  
}
```

Dependency Injection

Principe

Fournir les dépendances depuis l'extérieur plutôt que de les créer dans la classe

Avantages

- **Testabilité** : Mock des dépendances
- **Flexibilité** : Changement d'implémentation
- **Découplage** : Classes indépendantes

Exemple

```
// Sans DI - Couplage fort
class UserService {
    private repo = new UserRepository(); // ❌ Difficile à tester
```

Validation des Données

Pourquoi Valider ?

- **Sécurité** : Prévention des injections
- **Intégrité** : Données cohérentes
- **UX** : Messages d'erreur clairs
- **Maintenance** : Contrats explicites

Outils de Validation

```
import { z } from 'zod';

const MustacheSchema = z.object({
  name: z.string().min(1).max(100),
  url: z.string().url().optional(),
  tags: z.array(z.string()).default([])
});
```

Gestion des Erreurs

Types d'Erreurs

- **Validation** : Données incorrectes (400)
- **Authentication** : Non autorisé (401)
- **Authorization** : Accès interdit (403)
- **Ressource** : Non trouvée (404)
- **Serveur** : Erreur interne (500)

Error Handling Middleware

```
const errorHandler = (err: Error, req: Request, res: Response, next: NextFunction) => {  
  if (err instanceof ValidationError) {  
    return res.status(400).json({ error: err.message });  
  }  
  console.error(err);  
}
```

API RESTful

Principes REST

- **Stateless** : Chaque requête est indépendante
- **Resource-based** : URLs représentent des ressources
- **HTTP Methods** : GET, POST, PUT, DELETE
- **Status Codes** : Communication claire des résultats

Convention de Nommage

GET	/moustaches	# Liste toutes les moustaches
GET	/moustaches/1	# Récupère la moustache #1
POST	/moustaches	# Crée une nouvelle moustache
PUT	/moustaches/1	# Met à jour la moustache #1
DELETE	/moustaches/1	# Supprime la moustache #1

Sécurité Backend

Menaces Communes

- **Injection SQL** : Requêtes malveillantes
- **XSS** : Scripts malveillants côté client
- **CSRF** : Requêtes cross-site forgées
- **DoS** : Dénî de service
- **Data Exposure** : Exposition de données sensibles

Protections

```
app.use(helmet()); // Headers de sécurité
app.use(cors({ origin: allowedOrigins })); // CORS strict
app.use(rateLimit({ windowMs: 15 * 60 * 1000, max: 100 })); // Rate limiting
app.use(express.json({ limit: '10mb' })); // Limite de payload
```

Environment et Configuration

Principe des 12 Factors

Configuration via variables d'environnement

Structure .env

```
# Database
DATABASE_URL="file:./dev.db"

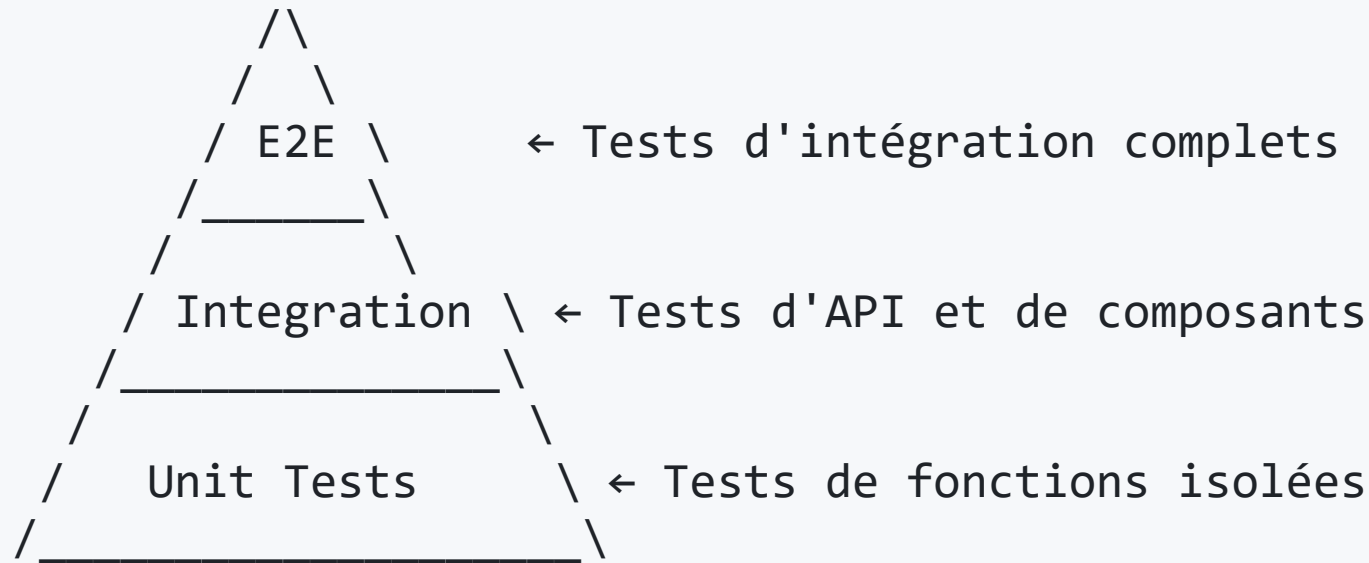
# Server
PORT=3000
NODE_ENV=development

# Security
JWT_SECRET="super-secret-key"
ALLOWED_ORIGINS="http://localhost:3000"

# External APIs
STRIPE_SECRET_KEY="sk test ..."
```

Testing Strategy

Pyramide des Tests



Types de Tests Backend

- **Unit** : Fonctions individuelles
- **Integration** : API endpoints

Tests avec Jest et Supertest

Configuration

```
// jest.config.js
module.exports = {
  preset: 'ts-jest',
  testEnvironment: 'node',
  setupFilesAfterEnv: ['<rootDir>/src/test/setup.ts'],
};
```

Test d'API

```
import request from 'supertest';
import app from '../app';

describe('Mustaches API', () => {
  it('should get all mustaches', async () => {
    const response = await request(app)
      .get('/mustaches')
```

Monitoring et Logging

Observabilité

- **Logs** : Traçabilité des événements
- **Metrics** : Performance et usage
- **Tracing** : Suivi des requêtes

Structured Logging

```
import winston from 'winston';

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.json()
  ),
  transports: [
```

Performance et Optimisation

Stratégies d'Optimisation

- **Database Indexing** : Requêtes rapides
- **Connection Pooling** : Réutilisation des connexions
- **Caching** : Réduction des requêtes
- **Compression** : Réduction de la bande passante
- **Rate Limiting** : Protection contre les abus

Caching avec Redis

```
import Redis from 'redis';

const redis = Redis.createClient();

async function getCachedMustaches() {
  const cached = await redis.get('mustaches');
```

Déploiement et DevOps

Environnements

- **Development** : Machine locale
- **Staging** : Réplique de production
- **Production** : Environnement live

CI/CD Pipeline

```
# .github/workflows/deploy.yml
name: Deploy
on:
  push:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
```

Docker et Containerisation

Avantages

- **Portabilité** : Même environnement partout
- **Isolation** : Pas de conflits de dépendances
- **Scalabilité** : Déploiement facile
- **Reproductibilité** : Builds identiques

Dockerfile

```
FROM node:18-alpine

WORKDIR /app

COPY package*.json ./
RUN npm ci --only=production
COPY
```


Database Migrations

Principe

Évolution contrôlée du schéma de base de données

Avantages avec Prisma

- **Versioning** : Historique des changements
- **Rollback** : Retour en arrière possible
- **Team Sync** : Synchronisation d'équipe
- **Production Safety** : Déploiements sécurisés

Workflow

```
# 1. Modifier le schema.prisma
# 2. Créer la migration
npx prisma migrate dev --name add-user-avatar
```

Patterns Avancés

Repository Pattern

Abstraction de l'accès aux données

Service Layer Pattern

Encapsulation de la logique métier

Dependency Injection

Inversion de contrôle

Observer Pattern

Événements et notifications

Strategy Pattern

Microservices vs Monolithe

Monolithe

- ✓ Simplicité de développement et déploiement
- ✓ Performance : pas de latence réseau
- ✗ Scalabilité limitée
- ✗ Technology Lock-in

Microservices

- ✓ Scalabilité indépendante
- ✓ Technology Diversity
- ✓ Team Independence
- ✗ Complexité opérationnelle
- ✗ Network Latency

GraphQL vs REST

REST

- ✓ Simplicité et maturité
- ✓ Caching HTTP standard
- ✓ Tooling riche
- ✗ Over/Under-fetching

GraphQL

- ✓ Flexible : un seul endpoint
- ✓ Type System fort
- ✓ Real-time avec subscriptions
- ✗ Complexité de cache
- ✗ Learning Curve

Conclusion

Compétences Acquises

-  Architecture backend moderne
-  TypeScript pour la robustesse
-  Express.js pour les APIs
-  Prisma pour les données
-  Patterns et bonnes pratiques
-  Sécurité et performance
-  Testing et qualité
-  Déploiement et monitoring

Prochaines Étapes

Questions & Discussion

Ressources pour Aller Plus Loin

Documentation

- [TypeScript Handbook](#)
- [Express.js Guide](#)
- [Prisma Documentation](#)

Pratique

- Projets personnels
- Contributions open source
- Code reviews en équipe