# Commentaires Techniques Finaux - Riziky-Boutic

## Analyse Approfondie des Choix Techniques et Implémentations

Cette documentation finale présente une analyse exhaustive des décisions techniques, des patterns d'architecture, et des commentaires explicatifs sur l'ensemble de l'implémentation de la plateforme Riziky-Boutic.

---

## Philosophie Architecture et Design Patterns

### Principes Fondamentaux Appliqués

#### 1. Separation of Concerns (Séparation des Préoccupations)

```
//  PRINCIPE: Chaque module a une responsabilité unique et bien définie

//  MAUVAISE APPROCHE - Tout dans un seul composant
const BadProductPage = () => {
  // Logique d'authentification
  const [user, setUser] = useState(null);
  // Logique de produits
  const [products, setProducts] = useState([]);
  // Logique de panier
  const [cart, setCart] = useState([]);
  // Logique UI
  // ... 200+ lignes de code mélangé
};

//  BONNE APPROCHE - Séparation claire des responsabilités
const ProductPage = () => {
  //  Authentification déléguée au hook spécialisé
  const { user, isAuthenticated } = useAuth();

  //  Gestion des produits déléguée au hook métier
  const { products, loading, fetchProducts } = useProducts();

  //  Gestion du panier déléguée au context global
  const { addToCart } = useStore();

  //  Composant se concentre uniquement sur la présentation
  return (
    <div className="product-page">
      <ProductGrid products={products} loading={loading} />
```

```
      <ProductFilters onFilter={fetchProducts} />
    </div>
  );
};
```

**Commentaire technique :** Cette séparation permet une maintenance facilitée, des tests unitaires ciblés, et une réutilisabilité maximale des composants.

### 2. Single Responsibility Principle (SRP)

```
//  PRINCIPE: Chaque fonction/classe/module n'a qu'une seule raison de changer

//  Service d'authentification - UNIQUEMENT l'authentification
class AuthService {
  async login(credentials) { /* ... */ }      // → Connexion utilisateur
  async logout() { /* ... */ }                // → Déconnexion utilisateur
  async verifyToken(token) { /* ... */ }      // → Vérification de token
  async refreshToken(token) { /* ... */ }     // → Renouvellement de token
}

//  Service de produits - UNIQUEMENT les produits
class ProductService {
  async getAllProducts() { /* ... */ }        // → Récupération des produits
  async getProductById(id) { /* ... */ }      // → Produit spécifique
  async searchProducts(query) { /* ... */ }   // → Recherche de produits
  async updateProduct(id, data) { /* ... */ } // → Mise à jour produit
}

//  Service de validation - UNIQUEMENT la validation
class ValidationService {
  validateEmail(email) { /* ... */ }          // → Validation email
  validatePassword(password) { /* ... */ }    // → Validation mot de passe
  sanitizeInput(input) { /* ... */ }          // → Nettoyage des entrées
}
```

**Commentaire technique :** Cette approche garantit que chaque modification de logique métier n'affecte qu'un seul service, réduisant les risques de régression.

### 3. Dependency Injection Pattern

```
//  PRINCIPE: Injecter les dépendances plutôt que de les créer

//  MAUVAISE APPROCHE - Couplage fort
class OrderService {
  constructor() {
    this.paymentService = new PaymentService(); // Couplage direct
    this.emailService = new EmailService();     // Dépendance hard-codée
```

```
  }
}

//  BONNE APPROCHE - Injection de dépendances
class OrderService {
  constructor(paymentService, emailService, logger) {
    this.paymentService = paymentService;   // → Dépendance injectée
    this.emailService = emailService;       // → Service modulaire
    this.logger = logger;                   // → Logging configurable
  }

  async processOrder(order) {
    this.logger.info(' Traitement de la commande', { orderId: order.id });

    try {
      //  Traitement du paiement via service injecté
      const payment = await this.paymentService.processPayment(order.payment);

      //  Envoi de confirmation via service injecté
      await this.emailService.sendOrderConfirmation(order, payment);

      this.logger.info(' Commande traitée avec succès', { orderId: order.id });
      return { success: true, payment };

    } catch (error) {
      this.logger.error(' Erreur traitement commande', { orderId: order.id, error });
      throw error;
    }
  }
}

//  Factory pour l'injection des dépendances
const createOrderService = () => {
  const paymentService = new StripePaymentService(); // ou PayPalPaymentService
  const emailService = new SendGridEmailService();   // ou MailgunEmailService
  const logger = new WinstonLogger();                // ou ConsoleLogger

  return new OrderService(paymentService, emailService, logger);
};
```

**Commentaire technique :** L'injection de dépendances facilite les tests unitaires (mocking) et permet de changer d'implémentation sans modifier le code métier.

---

## Patterns d'Architecture Avancés

### 1. Repository Pattern (Abstraction des Données)

```typescript
//  PRINCIPE: Abstraire l'accès aux données pour permettre différentes implémentations

//  Interface générique pour l'accès aux données
interface Repository<T> {
  findAll(): Promise<T[]>;
  findById(id: string): Promise<T | null>;
  create(entity: T): Promise<T>;
  update(id: string, entity: Partial<T>): Promise<T>;
  delete(id: string): Promise<boolean>;
}

//  Implémentation pour fichiers JSON (actuelle)
class JsonProductRepository implements Repository<Product> {
  private filePath = './data/products.json';

  async findAll(): Promise<Product[]> {
    console.log(' Lecture des produits depuis JSON');
    const data = await fs.readFile(this.filePath, 'utf8');
    return JSON.parse(data);
  }

  async findById(id: string): Promise<Product | null> {
    console.log(` Recherche produit ID: ${id}`);
    const products = await this.findAll();
    return products.find(p => p.id === id) || null;
  }

  async create(product: Product): Promise<Product> {
    console.log(' Création nouveau produit');
    const products = await this.findAll();
    const newProduct = { ...product, id: generateId(), createdAt: new Date() };
    products.push(newProduct);
    await this.saveProducts(products);
    return newProduct;
  }

  private async saveProducts(products: Product[]): Promise<void> {
    await fs.writeFile(this.filePath, JSON.stringify(products, null, 2));
    console.log(' Produits sauvegardés dans JSON');
  }
}
```

```typescript
// 🔮 Implémentation pour base de données (future)
class DatabaseProductRepository implements Repository<Product> {
  constructor(private database: Database) {}

  async findAll(): Promise<Product[]> {
    console.log('🗄️ Requête base de données: SELECT * FROM products');
    return this.database.query('SELECT * FROM products');
  }

  async findById(id: string): Promise<Product | null> {
    console.log(`🗄️ Requête base de données: SELECT * FROM products WHERE id = ${id}`);
    const result = await this.database.query('SELECT * FROM products WHERE id = ?', [id]);
    return result[0] || null;
  }

  // ... autres méthodes avec implémentation SQL
}

// 🏭 Factory pour basculer entre les implémentations
class RepositoryFactory {
  static createProductRepository(): Repository<Product> {
    if (process.env.DATABASE_TYPE === 'postgres') {
      console.log('🐘 Utilisation du repository PostgreSQL');
      return new DatabaseProductRepository(new PostgresDatabase());
    } else {
      console.log('📄 Utilisation du repository JSON');
      return new JsonProductRepository();
    }
  }
}

// 🎯 Service métier utilisant l'abstraction
class ProductService {
  constructor(private productRepository: Repository<Product>) {
    console.log('🎯 ProductService initialisé avec repository:', productRepository.constructo
  }

  async getAllProducts(): Promise<Product[]> {
    console.log('🎯 Service: Récupération de tous les produits');
    return this.productRepository.findAll();
  }

  async getProductById(id: string): Promise<Product | null> {
    console.log(`🎯 Service: Recherche produit ${id}`);
    return this.productRepository.findById(id);
  }
```

```
}
```

**Commentaire technique :** Le Repository Pattern permet de changer facilement de système de stockage (JSON → PostgreSQL → MongoDB) sans modifier la logique métier.

### 2. Observer Pattern (Événements et Notifications)

```
// 🎯 PRINCIPE: Notifier automatiquement les observateurs lors de changements d'état

// 📋 Interface pour les observateurs
interface Observer<T> {
  update(data: T): void;
}

// 🔔 Classe observable générique
class Observable<T> {
  private observers: Observer<T>[] = [];

  // ➕ Ajouter un observateur
  subscribe(observer: Observer<T>): void {
    this.observers.push(observer);
    console.log(`✅ Nouvel observateur ajouté (total: ${this.observers.length})`);
  }

  // ➖ Supprimer un observateur
  unsubscribe(observer: Observer<T>): void {
    const index = this.observers.indexOf(observer);
    if (index > -1) {
      this.observers.splice(index, 1);
      console.log(`❌ Observateur supprimé (total: ${this.observers.length})`);
    }
  }

  // 📢 Notifier tous les observateurs
  notify(data: T): void {
    console.log(`📢 Notification envoyée à ${this.observers.length} observateurs`);
    this.observers.forEach(observer => {
      try {
        observer.update(data);
      } catch (error) {
        console.error('⚠️ Erreur dans un observateur:', error);
      }
    });
  }
}
```

```typescript
// Implémentation pour le panier d'achat
class CartObservable extends Observable<CartEvent> {
  private cart: CartItem[] = [];

  addItem(product: Product, quantity: number): void {
    console.log(`Ajout au panier: ${product.name} (qty: ${quantity})`);

    const existingItem = this.cart.find(item => item.product.id === product.id);
    if (existingItem) {
      existingItem.quantity += quantity;
    } else {
      this.cart.push({ product, quantity });
    }

    // Notification de l'ajout
    this.notify({
      type: 'ITEM_ADDED',
      item: { product, quantity },
      cart: [...this.cart],
      timestamp: new Date()
    });
  }

  removeItem(productId: string): void {
    console.log(`Suppression du panier: ${productId}`);

    const index = this.cart.findIndex(item => item.product.id === productId);
    if (index > -1) {
      const removedItem = this.cart.splice(index, 1)[0];

      // Notification de la suppression
      this.notify({
        type: 'ITEM_REMOVED',
        item: removedItem,
        cart: [...this.cart],
        timestamp: new Date()
      });
    }
  }
}

// Observateur pour les analytics
class AnalyticsObserver implements Observer<CartEvent> {
  update(event: CartEvent): void {
    console.log('Analytics: Événement panier reçu', event.type);
```

```
    switch (event.type) {
      case 'ITEM_ADDED':
        this.trackProductAddedToCart(event.item.product);
        break;
      case 'ITEM_REMOVED':
        this.trackProductRemovedFromCart(event.item.product);
        break;
    }
  }

  private trackProductAddedToCart(product: Product): void {
    console.log(`  Analytics: Produit ajouté au panier - ${product.name}`);
    // Envoi vers service d'analytics
  }

  private trackProductRemovedFromCart(product: Product): void {
    console.log(`  Analytics: Produit retiré du panier - ${product.name}`);
    // Envoi vers service d'analytics
  }
}

//  Observateur pour les emails marketing
class EmailMarketingObserver implements Observer<CartEvent> {
  update(event: CartEvent): void {
    console.log('  Email Marketing: Événement panier reçu', event.type);

    if (event.type === 'ITEM_ADDED') {
      //  Programmer un email de récupération de panier abandonné
      this.scheduleAbandonedCartEmail(event.cart);
    }
  }

  private scheduleAbandonedCartEmail(cart: CartItem[]): void {
    console.log('  Programmation email panier abandonné dans 24h');
    setTimeout(() => {
      console.log('  Envoi email: "Vous avez oublié des articles dans votre panier"');
      // Logique d'envoi d'email
    }, 24 * 60 * 60 * 1000); // 24 heures
  }
}

//  Utilisation du pattern
const cartObservable = new CartObservable();

//  Inscription des observateurs
```

```
cartObservable.subscribe(new AnalyticsObserver());
cartObservable.subscribe(new EmailMarketingObserver());

//  Utilisation normale du panier - les observateurs sont notifiés automatiquement
cartObservable.addItem(product, 1); // → Déclenche analytics + email marketing
```

**Commentaire technique :** Le pattern Observer découple les événements métier de leurs effets de bord, permettant d'ajouter facilement de nouvelles fonctionnalités sans modifier le code existant.

### 3. Strategy Pattern (Algorithmes Interchangeables)

```
//  PRINCIPE: Encapsuler des algorithmes et les rendre interchangeables

//  Interface pour les stratégies de paiement
interface PaymentStrategy {
  name: string;
  processPayment(amount: number, paymentData: any): Promise<PaymentResult>;
  validatePaymentData(paymentData: any): boolean;
}

//  Stratégie de paiement par carte de crédit
class CreditCardPaymentStrategy implements PaymentStrategy {
  name = 'Carte de Crédit';

  async processPayment(amount: number, paymentData: CreditCardData): Promise<PaymentResult>
    console.log(` Traitement paiement carte: ${amount}€`);

    // Validation des données de carte
    if (!this.validatePaymentData(paymentData)) {
      throw new Error('Données de carte invalides');
    }

    // Simulation traitement Stripe/PayPal
    console.log(` Chiffrement données carte: **** **** **** ${paymentData.number.slice(-4)}
    console.log(` Appel API processeur de paiement`);

    // Simulation délai réseau
    await new Promise(resolve => setTimeout(resolve, 2000));

    const success = Math.random() > 0.1; // 90% de succès

    if (success) {
      console.log(' Paiement carte accepté');
      return {
        success: true,
```

```
        transactionId: `cc_${Date.now()}`,
        method: 'credit_card',
        amount: amount
      };
    } else {
      console.log('  Paiement carte refusé');
      throw new Error('Paiement refusé par la banque');
    }
  }

  validatePaymentData(data: CreditCardData): boolean {
    console.log('  Validation données carte de crédit');
    return !!(data.number && data.expiryDate && data.cvv && data.holderName);
  }
}

//  Stratégie de paiement PayPal
class PayPalPaymentStrategy implements PaymentStrategy {
  name = 'PayPal';

  async processPayment(amount: number, paymentData: PayPalData): Promise<PaymentResult> {
    console.log(`  Traitement paiement PayPal: ${amount}€`);

    if (!this.validatePaymentData(paymentData)) {
      throw new Error('Token PayPal invalide');
    }

    console.log(`  Redirection vers PayPal pour authentification`);
    console.log(`  Appel API PayPal`);

    // Simulation traitement PayPal
    await new Promise(resolve => setTimeout(resolve, 1500));

    console.log('  Paiement PayPal confirmé');
    return {
      success: true,
      transactionId: `pp_${Date.now()}`,
      method: 'paypal',
      amount: amount
    };
  }

  validatePaymentData(data: PayPalData): boolean {
    console.log('  Validation token PayPal');
    return !!(data.token && data.payerId);
  }
```

```
}

// 🏦 Stratégie de paiement par virement bancaire
class BankTransferPaymentStrategy implements PaymentStrategy {
  name = 'Virement Bancaire';

  async processPayment(amount: number, paymentData: BankTransferData): Promise<PaymentResult
    console.log(`🏦 Traitement virement bancaire: ${amount}€`);

    if (!this.validatePaymentData(paymentData)) {
      throw new Error('Données bancaires invalides');
    }

    console.log(`📄 Génération ordre de virement`);
    console.log(`📧 Envoi instructions de paiement par email`);

    // Le virement sera confirmé manuellement plus tard
    return {
      success: true,
      transactionId: `bt_${Date.now()}`,
      method: 'bank_transfer',
      amount: amount,
      status: 'pending' // En attente de confirmation manuelle
    };
  }

  validatePaymentData(data: BankTransferData): boolean {
    console.log('🔍 Validation coordonnées bancaires');
    return !!(data.accountNumber && data.routingNumber);
  }
}

// 🎯 Context utilisant les stratégies
class PaymentProcessor {
  private strategy: PaymentStrategy;

  constructor(strategy: PaymentStrategy) {
    this.strategy = strategy;
    console.log(`🎯 PaymentProcessor configuré avec: ${strategy.name}`);
  }

  // 🔄 Changer de stratégie dynamiquement
  setStrategy(strategy: PaymentStrategy): void {
    this.strategy = strategy;
    console.log(`🔄 Stratégie changée vers: ${strategy.name}`);
  }
```

```
  async processPayment(amount: number, paymentData: any): Promise<PaymentResult> {
    console.log(` Traitement paiement avec stratégie: ${this.strategy.name}`);

    try {
      const result = await this.strategy.processPayment(amount, paymentData);
      console.log(` Paiement réussi: ${result.transactionId}`);
      return result;

    } catch (error) {
      console.error(` Échec paiement avec ${this.strategy.name}:`, error.message);
      throw error;
    }
  }
}

//  Factory pour sélectionner la stratégie appropriée
class PaymentStrategyFactory {
  static createStrategy(paymentMethod: string): PaymentStrategy {
    console.log(` Création stratégie pour: ${paymentMethod}`);

    switch (paymentMethod.toLowerCase()) {
      case 'credit_card':
        return new CreditCardPaymentStrategy();
      case 'paypal':
        return new PayPalPaymentStrategy();
      case 'bank_transfer':
        return new BankTransferPaymentStrategy();
      default:
        console.log(` Méthode inconnue, utilisation carte par défaut`);
        return new CreditCardPaymentStrategy();
    }
  }
}

//  Utilisation du pattern
const processOrder = async (order: Order) => {
  console.log(` Traitement commande ${order.id}`);

  //  Sélection de la stratégie basée sur les préférences utilisateur
  const strategy = PaymentStrategyFactory.createStrategy(order.paymentMethod);
  const processor = new PaymentProcessor(strategy);

  try {
    //  Traitement du paiement avec la stratégie appropriée
    const paymentResult = await processor.processPayment(order.total, order.paymentData);
```

```
    console.log(` Commande ${order.id} payée avec succès`);
    return { success: true, payment: paymentResult };

  } catch (error) {
    console.error(` Échec paiement commande ${order.id}:`, error.message);

    //  Possibilité d'essayer une stratégie alternative
    if (order.paymentMethod === 'credit_card') {
      console.log(` Tentative avec PayPal en fallback`);
      processor.setStrategy(new PayPalPaymentStrategy());
      // Nouveau tentative...
    }

    throw error;
  }
};
```

**Commentaire technique :** Le Strategy Pattern permet d'ajouter facilement de nouvelles méthodes de paiement sans modifier le code existant, et de changer dynamiquement d'algorithme selon le contexte.

---

## Patterns de Sécurité Avancés

### 1. Middleware Chain Pattern (Chaîne de Sécurité)

```
/**
 *  PATTERN: Chaîne de middlewares de sécurité
 * Chaque middleware a une responsabilité spécifique et peut interrompre la chaîne
 */

//  Classe de base pour les middlewares de sécurité
class SecurityMiddleware {
  constructor(next = null) {
    this.next = next; // Référence vers le middleware suivant
  }

  //  Méthode à implémenter par chaque middleware
  async handle(req, res, context) {
    throw new Error('La méthode handle doit être implémentée');
  }

  //  Passer au middleware suivant
  async passToNext(req, res, context) {
    if (this.next) {
```

```
      return await this.next.handle(req, res, context);
    }
    return context; // Fin de chaîne
  }
}

//  Middleware de vérification d'IP bloquée
class BlockedIPMiddleware extends SecurityMiddleware {
  constructor(next) {
    super(next);
    this.blockedIPs = new Set(['192.168.1.100', '10.0.0.5']); // IPs blacklistées
  }

  async handle(req, res, context) {
    const clientIP = req.ip || req.connection.remoteAddress;
    console.log(`  Vérification IP: ${clientIP}`);

    if (this.blockedIPs.has(clientIP)) {
      console.log(`  IP bloquée détectée: ${clientIP}`);
      res.status(403).json({
        error: 'Accès interdit',
        code: 'IP_BLOCKED',
        timestamp: new Date().toISOString()
      });
      return null; // Arrêt de la chaîne
    }

    console.log(`  IP autorisée: ${clientIP}`);
    context.clientIP = clientIP;
    return await this.passToNext(req, res, context);
  }
}

//  Middleware de rate limiting
class RateLimitMiddleware extends SecurityMiddleware {
  constructor(next, maxRequests = 100, windowMs = 15 * 60 * 1000) {
    super(next);
    this.maxRequests = maxRequests;
    this.windowMs = windowMs;
    this.requests = new Map(); // IP -> [timestamps...]
  }

  async handle(req, res, context) {
    const clientIP = context.clientIP;
    const now = Date.now();
```

```
    console.log(`  Vérification rate limit pour: ${clientIP}`);

    //  Nettoyage des anciennes requêtes
    if (!this.requests.has(clientIP)) {
      this.requests.set(clientIP, []);
    }

    const ipRequests = this.requests.get(clientIP);
    const validRequests = ipRequests.filter(timestamp =>
      now - timestamp < this.windowMs
    );

    //  Vérification du dépassement de limite
    if (validRequests.length >= this.maxRequests) {
      console.log(`  Rate limit dépassé pour ${clientIP}: ${validRequests.length}/${this.max

      res.status(429).json({
        error: 'Trop de requêtes',
        code: 'RATE_LIMIT_EXCEEDED',
        retryAfter: Math.ceil(this.windowMs / 1000),
        limit: this.maxRequests,
        remaining: 0,
        resetTime: new Date(now + this.windowMs).toISOString()
      });
      return null; // Arrêt de la chaîne
    }

    //  Enregistrement de la requête
    validRequests.push(now);
    this.requests.set(clientIP, validRequests);

    console.log(`  Rate limit OK: ${validRequests.length}/${this.maxRequests}`);
    context.rateLimitInfo = {
      remaining: this.maxRequests - validRequests.length,
      resetTime: new Date(now + this.windowMs)
    };

    return await this.passToNext(req, res, context);
  }
}

//  Middleware d'authentification JWT
class JWTAuthMiddleware extends SecurityMiddleware {
  constructor(next, secretKey) {
    super(next);
    this.secretKey = secretKey;
```

```
    this.optionalPaths = ['/api/public', '/api/auth/login']; // Chemins sans auth requise
  }

  async handle(req, res, context) {
    const path = req.originalUrl;
    console.log(`  Vérification authentification pour: ${path}`);

    //  Chemins publics - pas d'auth requise
    if (this.optionalPaths.some(publicPath => path.startsWith(publicPath))) {
      console.log(`  Chemin public, authentification optionnelle`);
      return await this.passToNext(req, res, context);
    }

    //  Extraction du token
    const authHeader = req.headers.authorization;
    if (!authHeader || !authHeader.startsWith('Bearer ')) {
      console.log(`  Token manquant pour: ${path}`);
      res.status(401).json({
        error: 'Token d\'authentification requis',
        code: 'MISSING_TOKEN'
      });
      return null;
    }

    const token = authHeader.substring(7); // Enlever "Bearer "

    try {
      //  Vérification et décodage du token
      const decoded = jwt.verify(token, this.secretKey);
      console.log(`  Token valide pour utilisateur: ${decoded.userId}`);

      //  Vérification de l'expiration
      const now = Math.floor(Date.now() / 1000);
      if (decoded.exp && decoded.exp < now) {
        console.log(`  Token expiré pour: ${decoded.userId}`);
        res.status(401).json({
          error: 'Token expiré',
          code: 'TOKEN_EXPIRED'
        });
        return null;
      }

      //  Ajout des infos utilisateur au contexte
      context.user = {
        id: decoded.userId,
        email: decoded.email,
```

```
        role: decoded.role,
        permissions: decoded.permissions || []
      };

      return await this.passToNext(req, res, context);

    } catch (error) {
      console.log(` Token invalide:`, error.message);
      res.status(401).json({
        error: 'Token invalide',
        code: 'INVALID_TOKEN'
      });
      return null;
    }
  }
}

//  Middleware de validation des permissions
class PermissionMiddleware extends SecurityMiddleware {
  constructor(next, requiredPermissions = []) {
    super(next);
    this.requiredPermissions = requiredPermissions;
  }

  async handle(req, res, context) {
    //  Vérification de la présence de l'utilisateur
    if (!context.user) {
      console.log(` Utilisateur non authentifié pour vérification permissions`);
      return await this.passToNext(req, res, context); // Laisser passer, l'auth sera vérif
    }

    const { user } = context;
    const userPermissions = user.permissions || [];
    const userRole = user.role;

    console.log(` Vérification permissions pour ${user.email} (${userRole})`);
    console.log(` Permissions requises:`, this.requiredPermissions);
    console.log(` Permissions utilisateur:`, userPermissions);

    //  Super admin a tous les droits
    if (userRole === 'super_admin') {
      console.log(` Super admin détecté - tous droits accordés`);
      return await this.passToNext(req, res, context);
    }

    //  Vérification des permissions spécifiques
```

```
    const hasPermission = this.requiredPermissions.every(required =>
      userPermissions.includes(required) || userRole === 'admin'
    );

    if (!hasPermission) {
      console.log(` Permissions insuffisantes pour ${user.email}`);
      res.status(403).json({
        error: 'Permissions insuffisantes',
        code: 'INSUFFICIENT_PERMISSIONS',
        required: this.requiredPermissions,
        current: userPermissions
      });
      return null;
    }

    console.log(` Permissions validées pour ${user.email}`);
    return await this.passToNext(req, res, context);
  }
}

//  Middleware de nettoyage des données
class DataSanitizationMiddleware extends SecurityMiddleware {
  async handle(req, res, context) {
    console.log(` Nettoyage des données de la requête`);

    //  Patterns malveillants à détecter
    const maliciousPatterns = [
      /<script[^>]*>.*?<\/script>/gi,          // Scripts JavaScript
      /javascript:/gi,                          // URLs JavaScript
      /on\w+\s*=/gi,                           // Event handlers HTML
      /(union|select|insert|delete|drop)\s+/gi, // SQL Injection
      /\.\.\/|\.\.\\|\.\.%2f|\.\.%5c/gi       // Directory traversal
    ];

    //  Fonction récursive de nettoyage
    const sanitizeObject = (obj, path = '') => {
      if (typeof obj === 'string') {
        //  Détection de contenu malveillant
        for (const pattern of maliciousPatterns) {
          if (pattern.test(obj)) {
            console.log(` Contenu malveillant détecté dans ${path}: ${pattern}`);
            throw new Error(`Contenu malveillant détecté: ${pattern.source}`);
          }
        }

        //  Nettoyage basique
```

```
      return obj
        .trim()                              // Suppression espaces
        .replace(/[<>]/g, '')               // Suppression < et >
        .substring(0, 1000);                // Limitation longueur

    } else if (Array.isArray(obj)) {
      return obj.map((item, index) =>
        sanitizeObject(item, `${path}[${index}]`)
      );

    } else if (obj && typeof obj === 'object') {
      const sanitized = {};
      for (const [key, value] of Object.entries(obj)) {
        sanitized[key] = sanitizeObject(value, `${path}.${key}`);
      }
      return sanitized;
    }

    return obj;
  };

  try {
    //  Nettoyage du body
    if (req.body) {
      req.body = sanitizeObject(req.body, 'body');
      console.log(`  Body nettoyé`);
    }

    //  Nettoyage des query params
    if (req.query) {
      req.query = sanitizeObject(req.query, 'query');
      console.log(`  Query params nettoyés`);
    }

    //  Nettoyage des params d'URL
    if (req.params) {
      req.params = sanitizeObject(req.params, 'params');
      console.log(`  URL params nettoyés`);
    }

    return await this.passToNext(req, res, context);

  } catch (error) {
    console.log(`  Erreur de validation sécurité:`, error.message);
    res.status(400).json({
      error: 'Données invalides détectées',
```

```
        code: 'MALICIOUS_CONTENT',
        details: error.message
      });
      return null;
    }
  }
}

// Builder pour construire la chaîne de sécurité
class SecurityChainBuilder {
  constructor() {
    this.middlewares = [];
  }

  // Ajouter vérification IP
  blockIPs(blockedIPs = []) {
    const middleware = new BlockedIPMiddleware();
    blockedIPs.forEach(ip => middleware.blockedIPs.add(ip));
    this.middlewares.push(middleware);
    return this;
  }

  // Ajouter rate limiting
  rateLimit(maxRequests = 100, windowMs = 15 * 60 * 1000) {
    this.middlewares.push(new RateLimitMiddleware(null, maxRequests, windowMs));
    return this;
  }

  // Ajouter authentification JWT
  requireAuth(secretKey) {
    this.middlewares.push(new JWTAuthMiddleware(null, secretKey));
    return this;
  }

  // Ajouter vérification permissions
  requirePermissions(permissions) {
    this.middlewares.push(new PermissionMiddleware(null, permissions));
    return this;
  }

  // Ajouter nettoyage données
  sanitizeData() {
    this.middlewares.push(new DataSanitizationMiddleware());
    return this;
  }
```

```
  //  Construire la chaîne
  build() {
    //  Chaînage des middlewares
    for (let i = 0; i < this.middlewares.length - 1; i++) {
      this.middlewares[i].next = this.middlewares[i + 1];
    }

    //  Retourner le middleware principal
    return async (req, res, next) => {
      const context = { startTime: Date.now() };

      try {
        const result = await this.middlewares[0].handle(req, res, context);

        if (result !== null) {
          //  Toute la chaîne a réussi
          req.securityContext = result;
          console.log(` Chaîne de sécurité validée en ${Date.now() - context.startTime}ms`)
          next();
        }
        // Si result === null, une réponse a déjà été envoyée par un middleware

      } catch (error) {
        console.error(` Erreur dans la chaîne de sécurité:`, error);
        res.status(500).json({
          error: 'Erreur de sécurité interne',
          code: 'SECURITY_ERROR'
        });
      }
    };
  }
}

//  Utilisation de la chaîne de sécurité
const createSecurityChain = () => {
  return new SecurityChainBuilder()
    .blockIPs(['192.168.1.100'])           // Bloquer IPs suspectes
    .rateLimit(100, 15 * 60 * 1000)        // 100 req/15min
    .sanitizeData()                        // Nettoyer les données
    .requireAuth(process.env.JWT_SECRET)    // Authentification
    .requirePermissions(['read:products'])  // Permissions spécifiques
    .build();
};

//  Application de la chaîne sur les routes
app.use('/api/protected', createSecurityChain());
```

```
app.use('/api/admin',
  new SecurityChainBuilder()
    .blockIPs()
    .rateLimit(20, 15 * 60 * 1000)          // Limite plus stricte pour admin
    .sanitizeData()
    .requireAuth(process.env.JWT_SECRET)
    .requirePermissions(['admin:manage'])
    .build()
);
```

**Commentaire technique :** Cette chaîne de middlewares de sécurité applique le principe de défense en profondeur, où chaque couche ajoute une protection spécifique. L'ordre des middlewares est crucial pour l'efficacité.

Cette documentation finale présente une analyse technique complète des patterns, architectures et choix d'implémentation utilisés dans Riziky-Boutic, avec des commentaires détaillés expliquant les raisons techniques de chaque décision.