The Developer Architect

Design like an architect and code like a developer

@codewkarim

Abdelkrim Bourennane

- **Software Engineer & Consultant** @ Cellenza
- Microsoft Certified Trainer

- **Head of Teach** @ SoCode

- Social media :

@codewkarim

/in/abdelkrim-bournane/

@karimkos

What is this talk about

Business Applications

e.g : Banking, E-commerce, Delivery, etc.

Tooling for designing maintainable software

Thinking like an architect

Technology agnostic

You can apply this with any tech stack.

Food For Thought

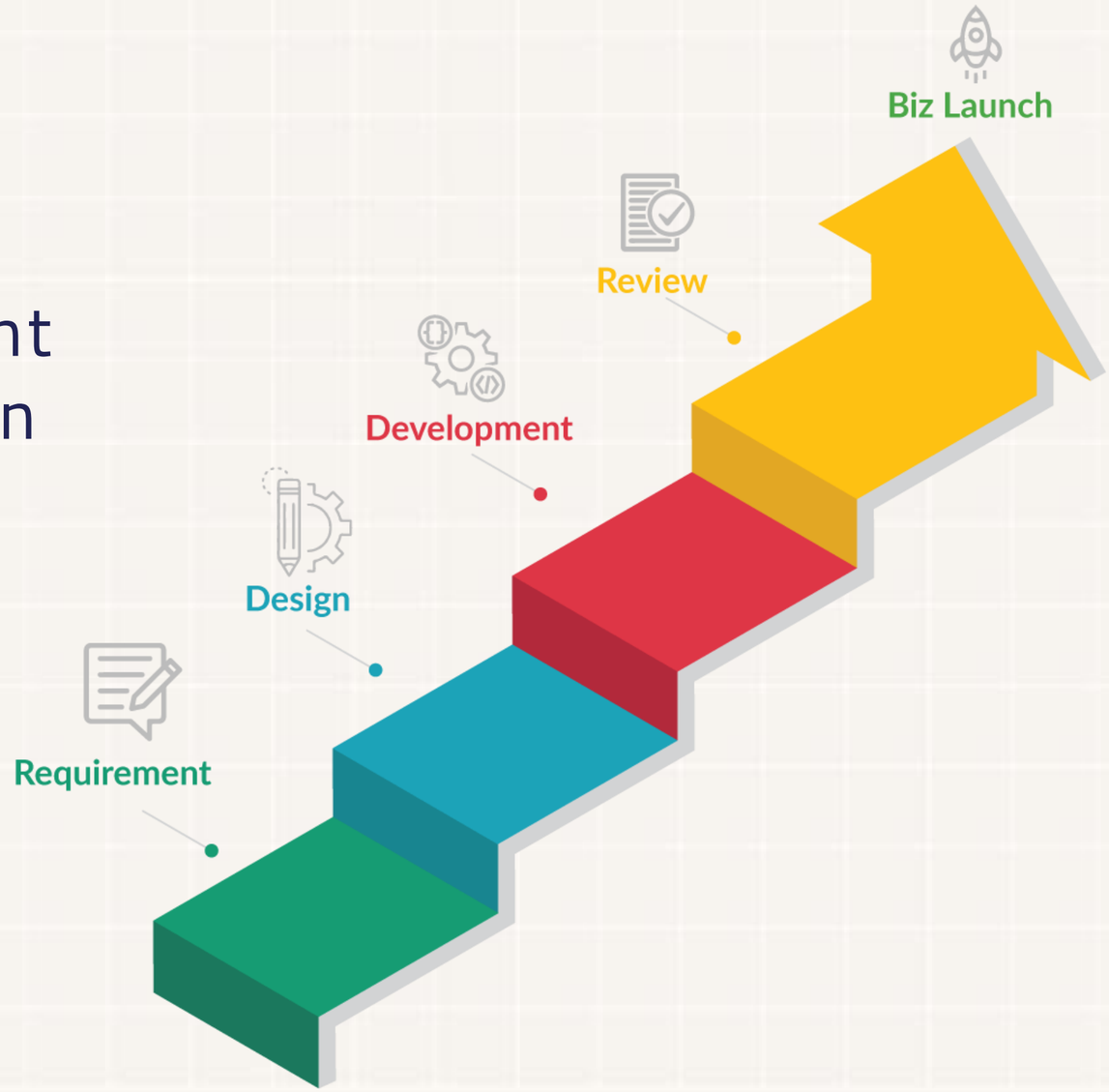
Take this as a guide for your further reading and exploration

01

The History

1. The History ⏳

Big
Upfront
Design

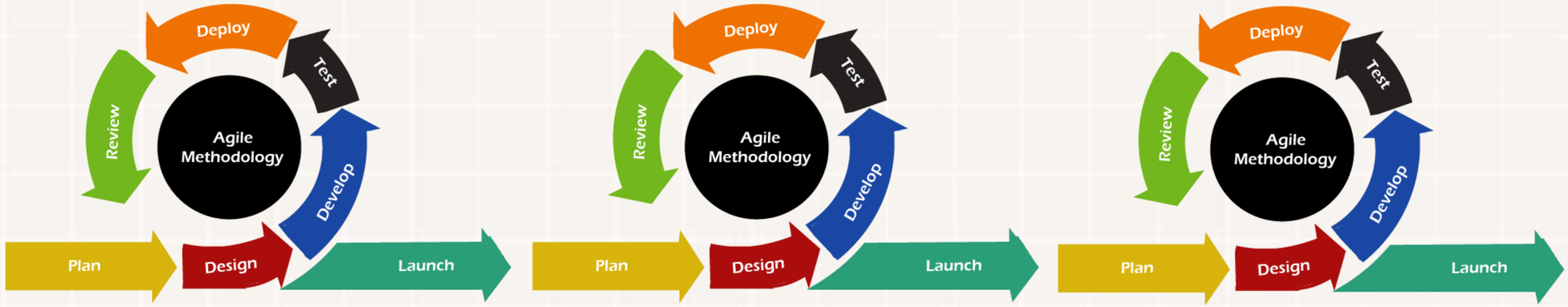


The History ⏳

The Development team

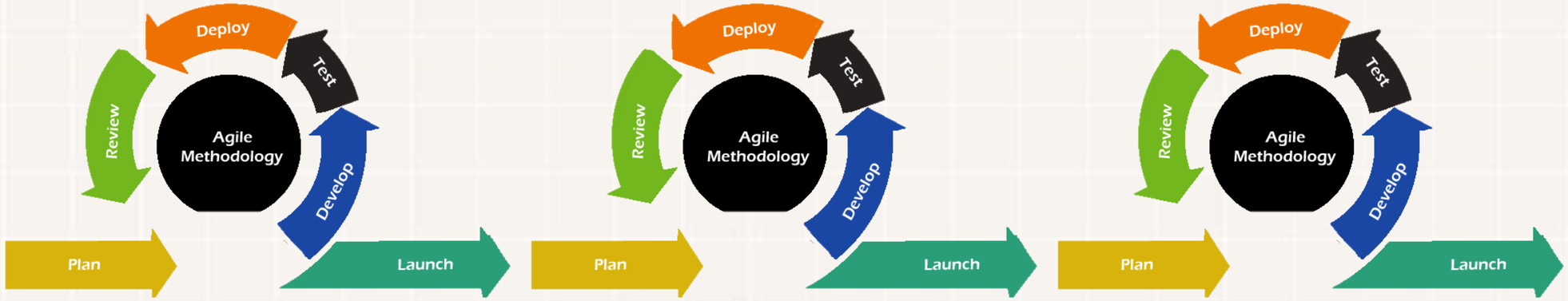
The client

1. The History ⏳



Just in time design

1. The History ⏳



كيما
جات
تجي
Design

1. The History ⏳

The Design

1. The History ⏳

The Result

02

The Solution

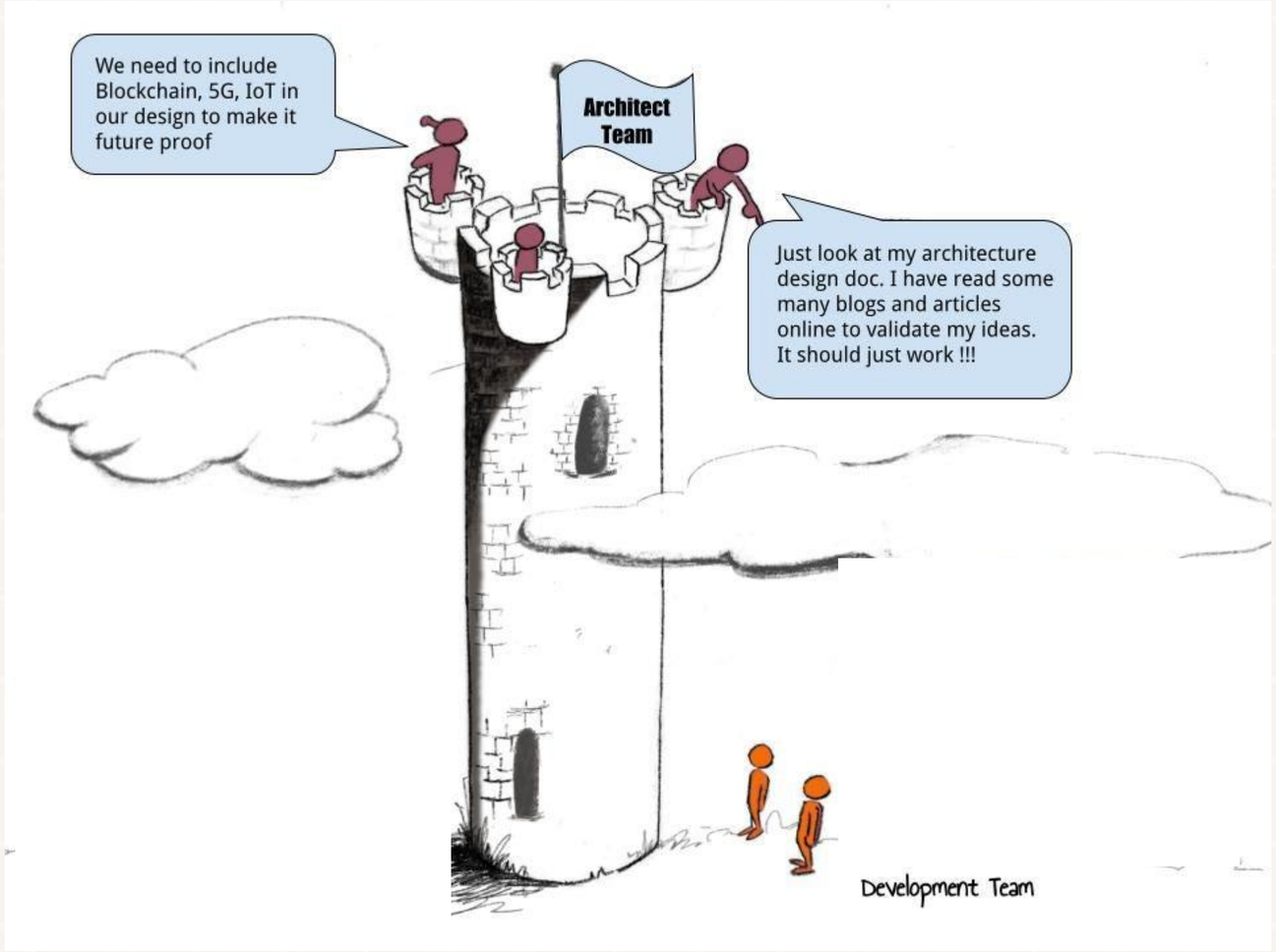
2. The Solution ✅

- Take Design Seriously
- Have Design Skills

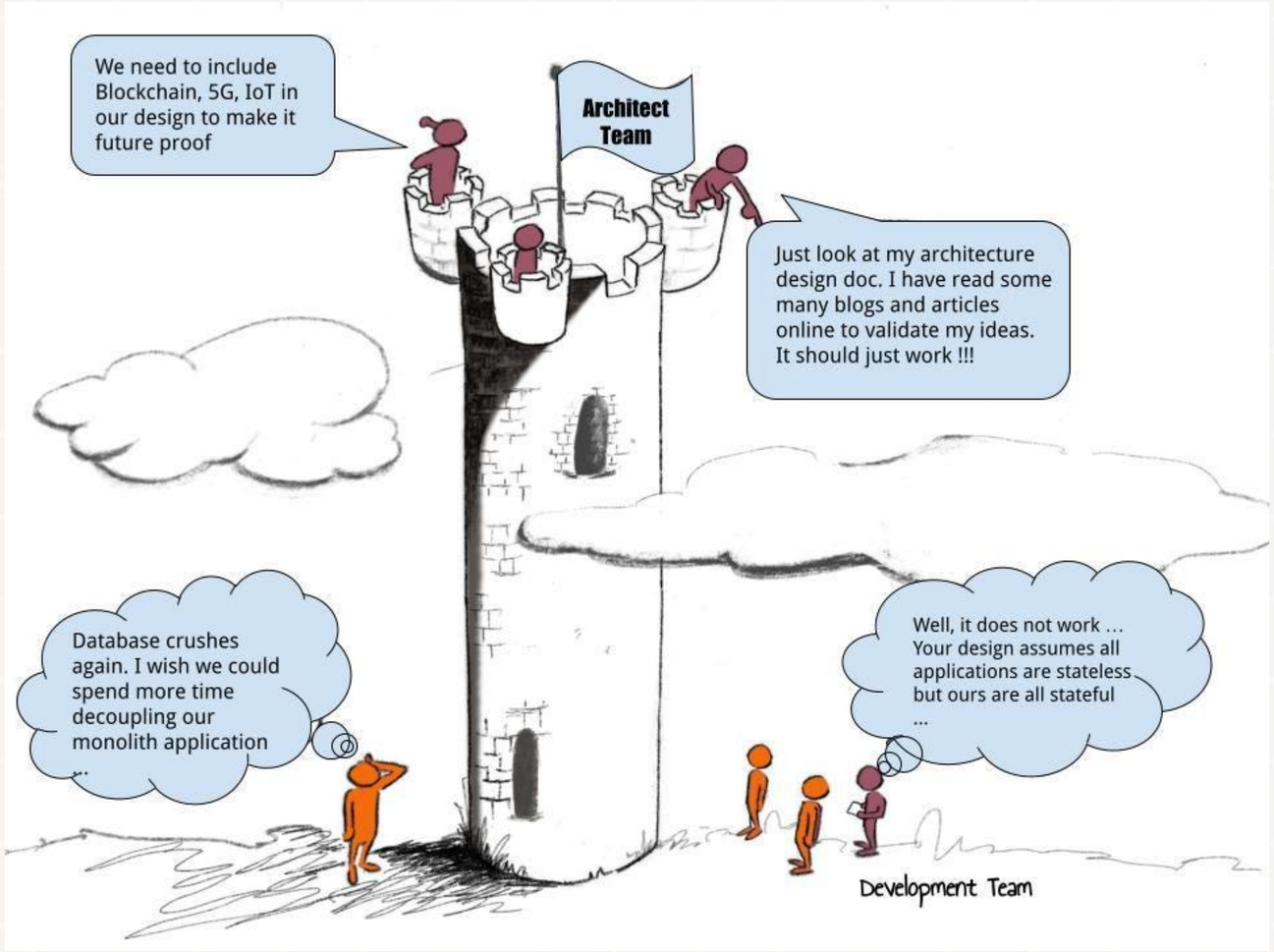
2. The Solution ✅

1. I’m not a software engineer /architect.
2. Not my job to do design.

1. I’m not a software engineer /architect.
2. Not my job to do design.



1. I’m not a software engineer /architect.
2. Not my job to do design.

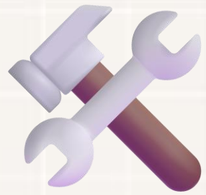


The Developer Architect

Architects should be part of the team, and the development team should think like an architect

03

The Skills



3. The Skills 🛠️

DRY

SOLID

Clean Architecture

CQRS

Dependency Inversion

WET

Single responsibility principle

KISS

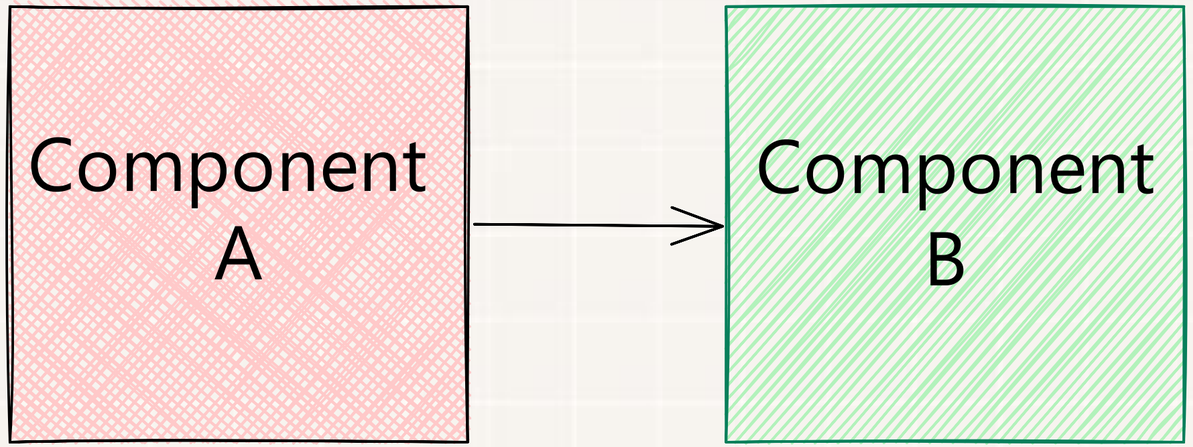
3. The Skills 🛠️

Coupling - الاقتران

Cohesion - التناسق

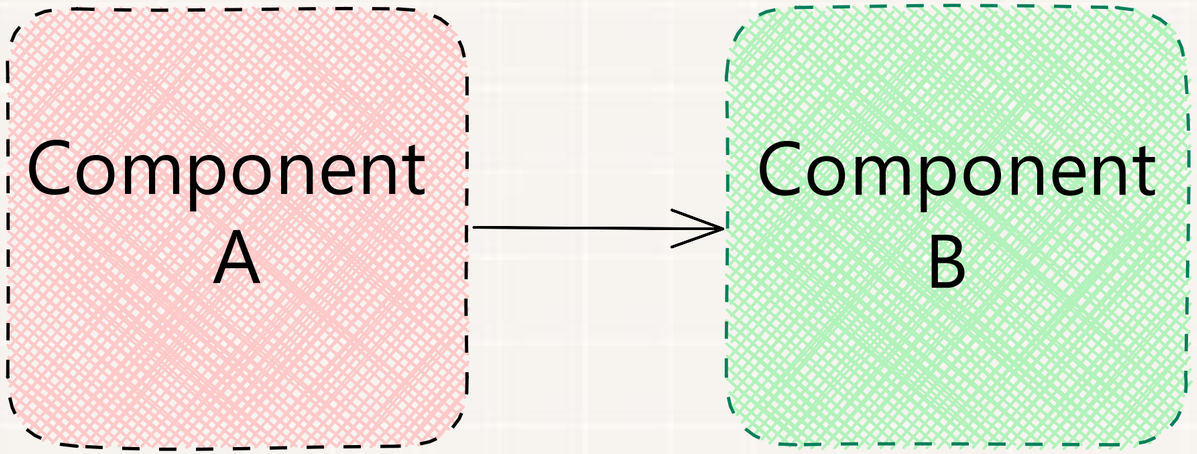
Coupling - الاقتران

A is coupled with B



Coupling - الاقتران

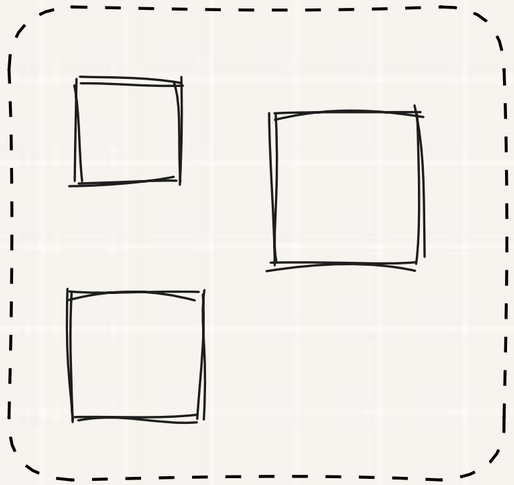
Changes in B require Changes in A



Cohesion - التناسق

We say that a component is cohesive

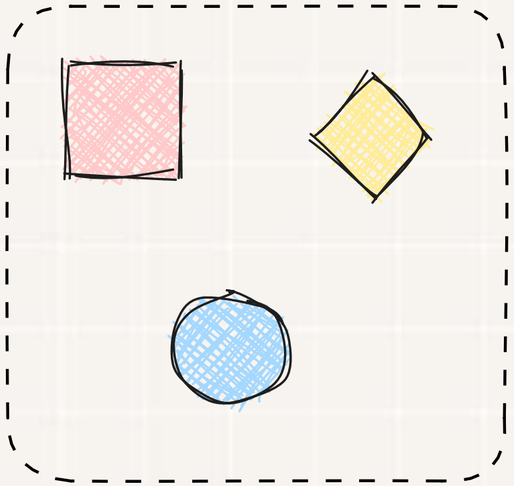
Component
C



Cohesion - التناسق

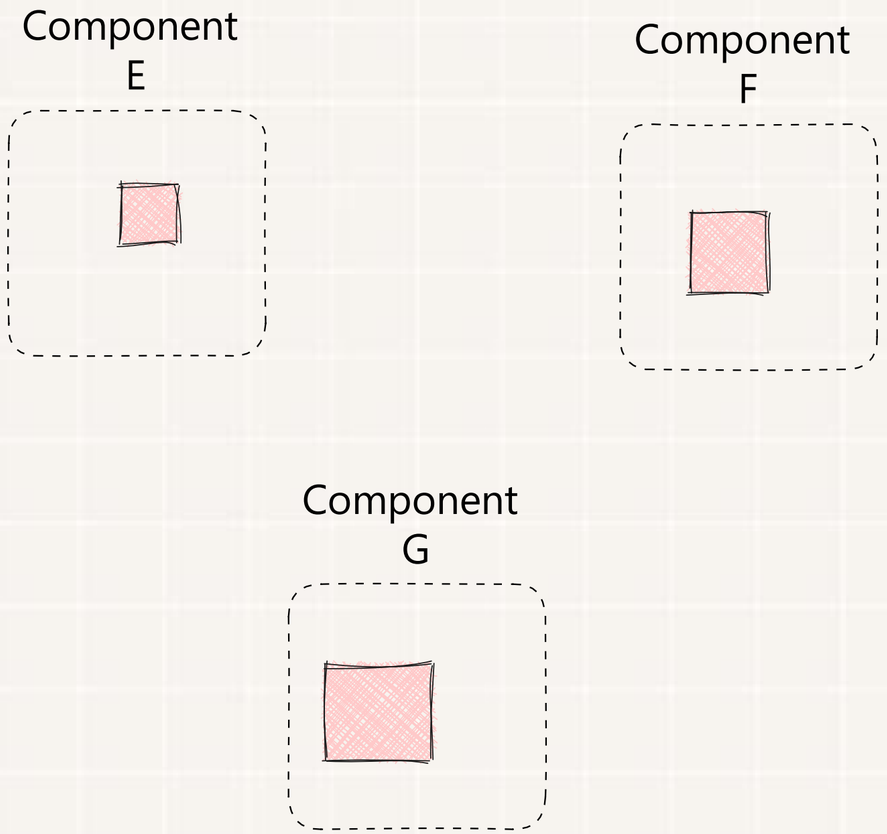
We say that a component is not cohesive

Component
D

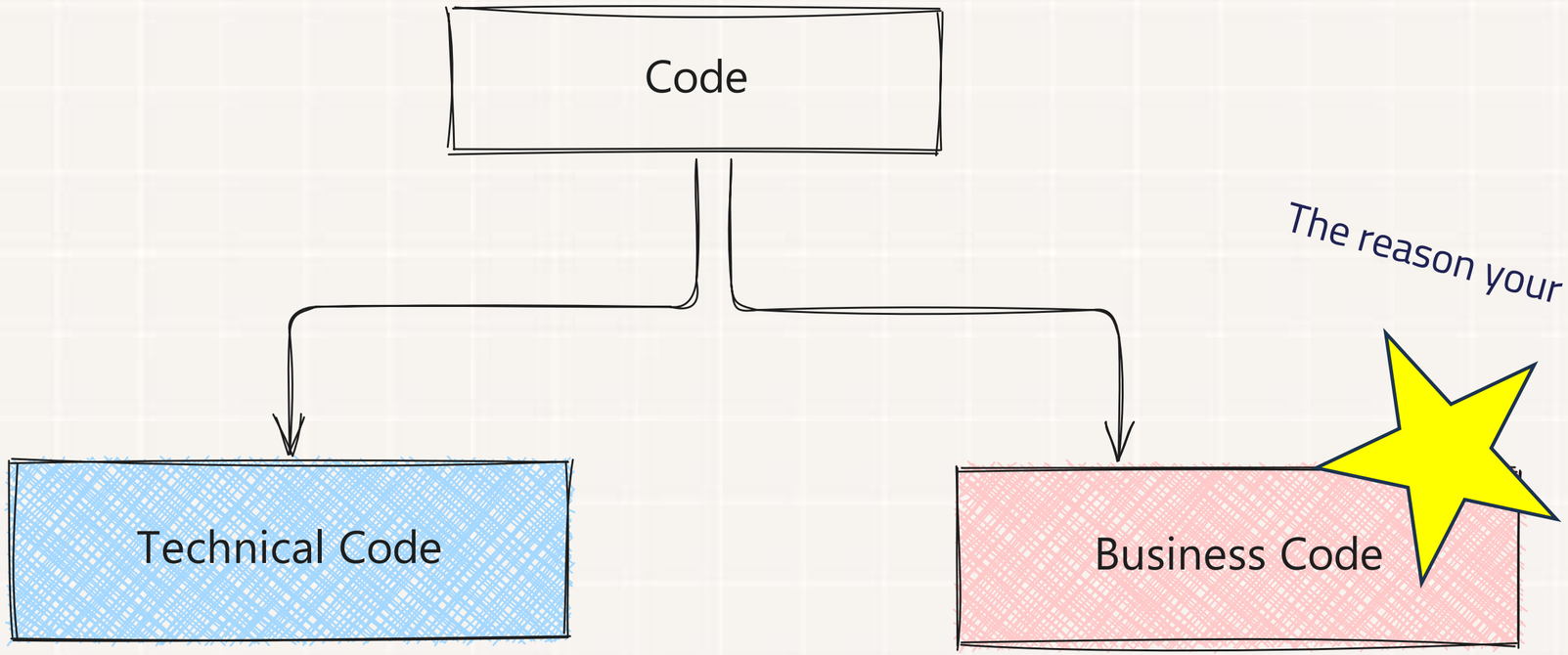


Cohesion - التناسق

We say that a component is not cohesive



Type of codes



1.Read for the database
2.Send a notification
3.Receive and send an HTTP request

1.Increase salary of an employee
2.Pay a product
3.Ship product
4.Process a payment

As a legend once said

```
class CustomerRepository {
    private pool: Pool;

    constructor(pool: Pool) {
        this.pool = pool;
    }

    async getCustomerStatus(customerId: number): Promise<CustomerStatus> {
        const query = `
            WITH customer_stats AS (
                SELECT
                    c.id,
                    c.name,
                    SUM(o.total_amount) as yearly_spent,
                    COUNT(o.id) as order_count,
                    -- Business logic in SQL: Calculate average order value
                    SUM(o.total_amount) / COUNT(o.id) as avg_order_value,
                    -- Business logic: Check if had any returns
                    EXISTS (SELECT 1 FROM returns r WHERE r.customer_id = c.id) as
had_returns,
                    -- Business logic: Calculate days since last order
                    EXTRACT(DAY FROM NOW() - MAX(o.created_at)) as days_since_last_order
                FROM customers c
                LEFT JOIN orders o ON o.customer_id = c.id
                WHERE c.id = $1
                AND o.created_at > NOW() - INTERVAL '1 year'
                GROUP BY c.id, c.name
            )
            SELECT
                id,
                name,
                yearly_spent,
                -- Business logic: Determine customer tier
                CASE
                    WHEN yearly_spent >= 10000 AND order_count >= 24 AND NOT had_returns THEN
'PLATINUM'
                    WHEN yearly_spent >= 5000 AND order_count >= 12 THEN 'GOLD'
                    WHEN yearly_spent >= 1000 AND order_count >= 4 THEN 'SILVER'
                    ELSE 'BRONZE'
                END as tier,
                -- Business logic: Calculate loyalty points
                CASE
                    WHEN days_since_last_order < 30 THEN FLOOR(yearly_spent * 1.2)
                    WHEN days_since_last_order < 90 THEN FLOOR(yearly_spent * 1.0)
                    ELSE FLOOR(yearly_spent * 0.8)
                END as loyalty_points
            FROM customer_stats;
```

1. We see that business code is tangled with SQL
2. We mixed two types of code technical and functional.
3. If the SQL query changes, we are also changing business code!
4. Everything becomes hard, business logic changes, database update and changes, debugging testing, etc.

```
async createOrder(req: express.Request): Promise<Order> {
    // Business validation mixed with HTTP request access
    if (!req.body.items || !req.body.customerId) {
        throw new Error('Invalid request body');
    }

    // Business logic calculating totals from HTTP request
    const totalAmount = req.body.items.reduce((sum: number, item: any) => {
        return sum + (item.price * item.quantity);
    }, 0);

    // Business rules mixed with HTTP data
    let discount = 0;
    if (req.body.promoCode === 'SUMMER2024') {
        discount = totalAmount * 0.1;
    }

    // Payment processing using HTTP request data
    const paymentResult = await this.paymentGateway.processPayment({
        amount: totalAmount - discount,
        paymentMethod: req.body.paymentMethod
    });

    // Saving order using HTTP request data
    const order = await this.orderRepository.save({
        customerId: req.body.customerId,
        items: req.body.items,
        totalAmount,
        discount,
        paymentId: paymentResult.id,
        shippingAddress: req.body.shippingAddress
    });

    return order;
}
```

1. What if we no longer using the body?
2. What if a new of the web framework update drops.
3. Everything becomes hard, business logic changes, web api update and changes, debugging testing, etc.

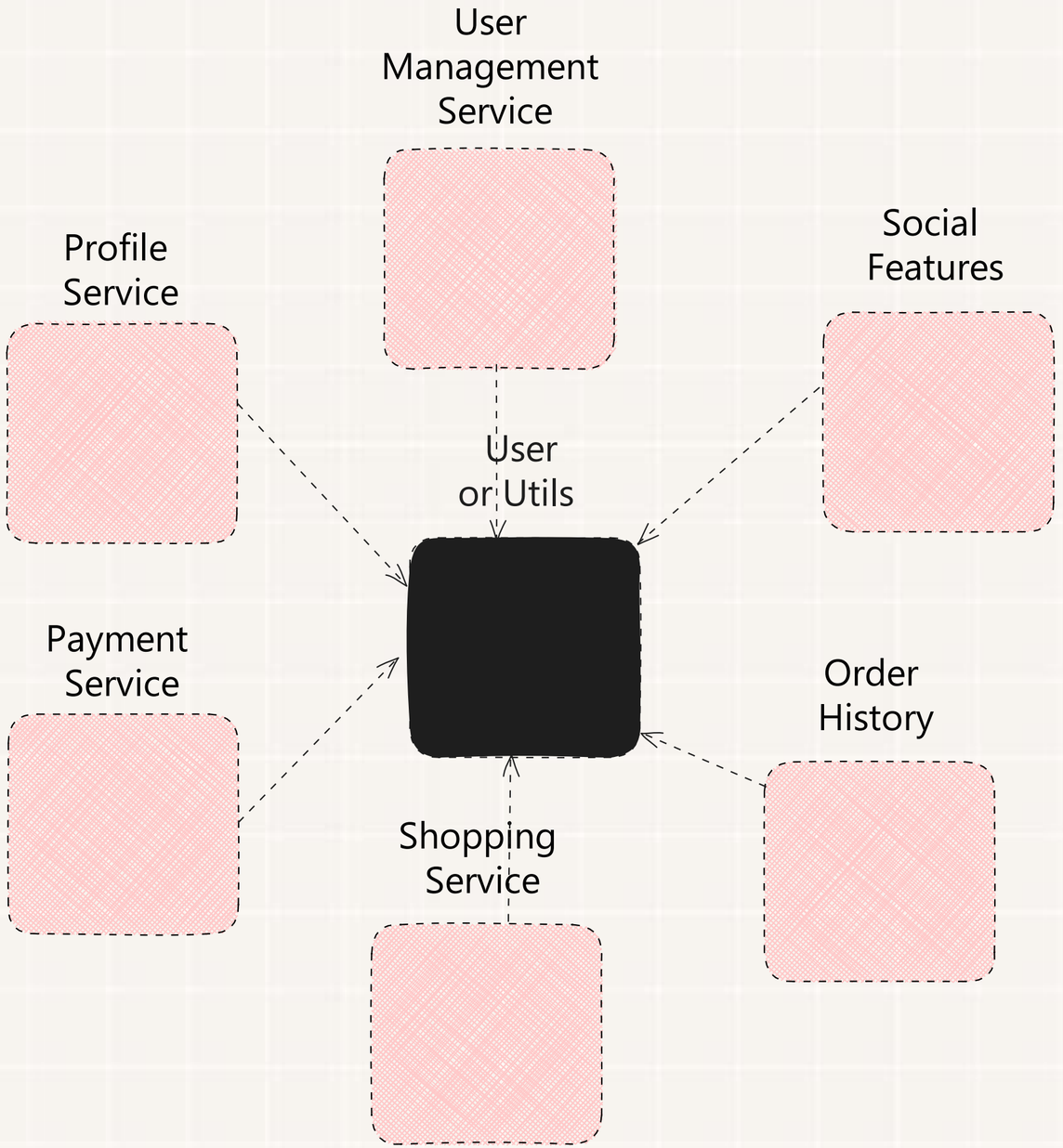
```
class Utils {
    // User authentication methods
    static async validatePassword(password) {...}


    static generateSessionToken(userId) {...}

    // Date formatting methods
    static formatDate(date) {...}

    static getDaysBetweenDates(date1, date2) {...}

    // File handling methods
    static getFileExtension(filename) {...}

    static formatFileSize(bytes) {...}

    // Mathematical calculations
    static calculateAverage(numbers) {...}

    static standardDeviation(numbers) {...}

    // String manipulation
    static capitalizeString(str) {...}

    static generateSlug(text) {...}

    // Network requests
    static async fetchWithTimeout(resource, options = {}) {...}

    // Cache handling
    static setCache(key, value, expirationMinutes = 60) {...}

    static getCache(key) {...}
}
```

1. One Class that contain **unrelated functions**

```
class User {
    // User identity properties
    private id: string;
    private email: string;
    private password: string;

    // Profile information
    private firstName: string;
    private lastName: string;
    private avatar: string;
    private bio: string;

    // Payment information
    private creditCardNumber: string;
    private cvv: string;
    private cardExpiryDate: Date;
    private billingAddress: Address;

    // Shopping preferences
    private favoriteCategories: string[];
    private wishlist: Product[];
    private shoppingCart: CartItem[];

    // Order history
    private orders: Order[];
    private returnedItems: Product[];
    private refundHistory: Refund[];

    // Social features
    private followers: User[];
    private following: User[];
    private posts: Post[];
    private comments: Comment[];

    // Notification preferences
    private emailNotificationsEnabled: boolean;
    private pushNotificationsEnabled: boolean;
    private notificationFrequency: string;

    // Analytics and tracking
```

1. One Class that contain **unrelated functions**

2. Although they are all properties of the User but they are unrelated to each other.


User
Management
Service
Profile
Service
Social
Features
User
or Utils
Payment
Service
Order
History
Shopping
Service

3. The Skills 🛠️

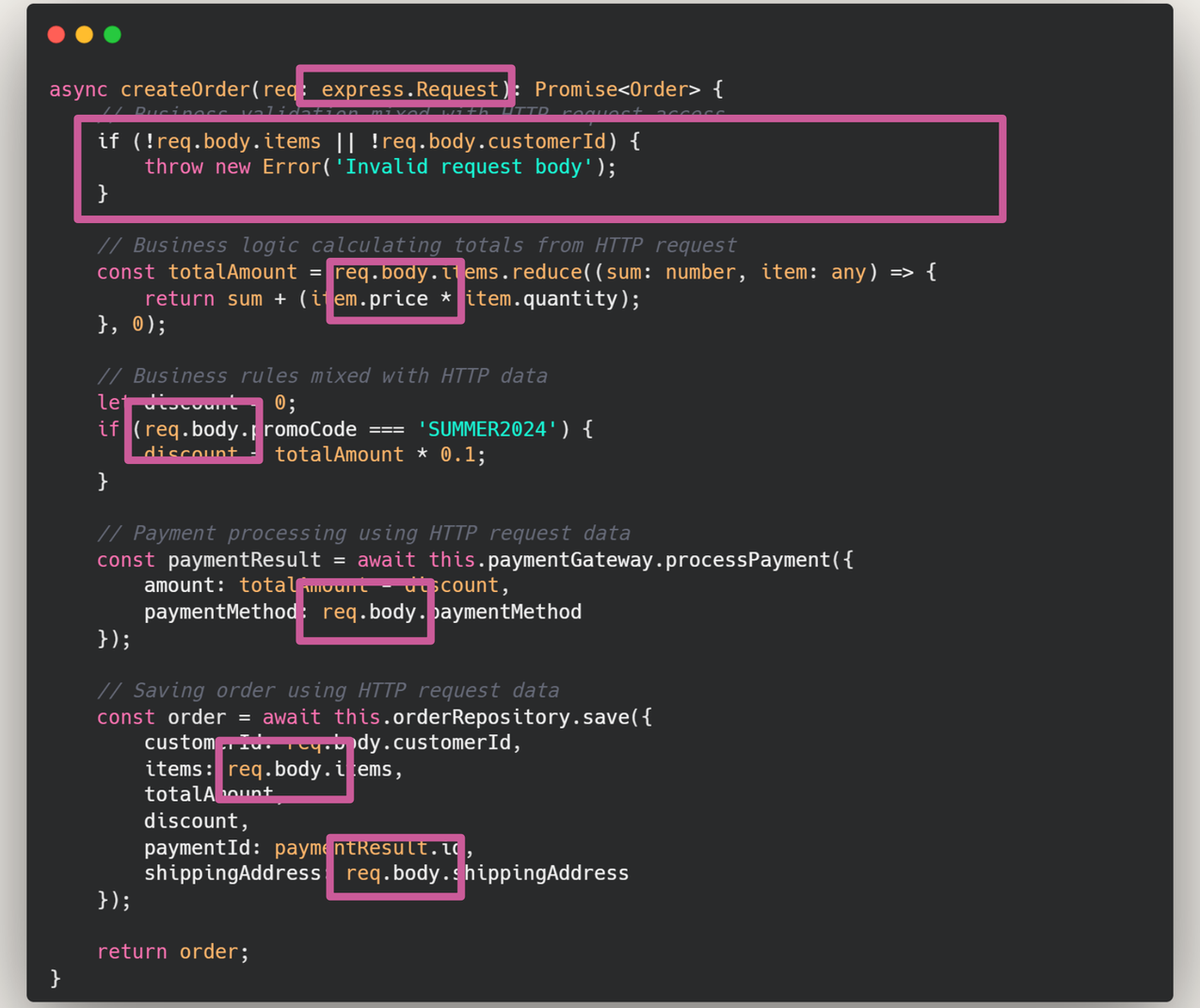
1. Separate technical from business code. (**reduce coupling**)

2. Keep code together if it’s part of the same business domain. (**increase cohesion**)

This should be out of the query and be in separate component that belongs to the business layer (service, or a handler...)

```
class CustomerRepository {
    private pool: Pool;

    constructor(pool: Pool) {
        this.pool = pool;
    }

    async getCustomerStatus(customerId: number): Promise<CustomerStatus> {
        const query = `
            WITH customer_stats AS (
                SELECT
                    c.id,
                    c.name,
                    SUM(o.total_amount) as yearly_spent,
                    COUNT(o.id) as order_count,
                    -- Business logic in SQL: Calculate average order value
                    SUM(o.total_amount) / COUNT(o.id) as avg_order_value,
                    -- Business logic: Check if had any returns
                    EXISTS (SELECT 1 FROM returns r WHERE r.customer_id = c.id) as
had_returns,
                    -- Business logic: Calculate days since last order
                    EXTRACT(DAY FROM NOW() - MAX(o.created_at)) as days_since_last_order
                FROM customers c
                LEFT JOIN orders o ON o.customer_id = c.id
                WHERE c.id = $1
                AND o.created_at > NOW() - INTERVAL '1 year'
                GROUP BY c.id, c.name
            )
            SELECT
                id,
                name,
                yearly_spent,
                -- Business logic: Determine customer tier
                CASE
                    WHEN yearly_spent >= 10000 AND order_count >= 24 AND NOT had_returns THEN
'PLATINUM'
                    WHEN yearly_spent >= 5000 AND order_count >= 12 THEN 'GOLD'
                    WHEN yearly_spent >= 1000 AND order_count >= 4 THEN 'SILVER'
                    ELSE 'BRONZE'
                END as tier,
                -- Business logic: Calculate loyalty points
                CASE
                    WHEN days_since_last_order < 30 THEN FLOOR(yearly_spent * 1.2)
                    WHEN days_since_last_order < 90 THEN FLOOR(yearly_spent * 1.0)
                    ELSE FLOOR(yearly_spent * 0.8)
                END as loyalty_points
            FROM customer_stats;
        `;

        const result = await this.pool.query(query, [customerId]);
        return result.rows[0];
    }
}
```

All this code should stay in the http API layer (controller or your route code)

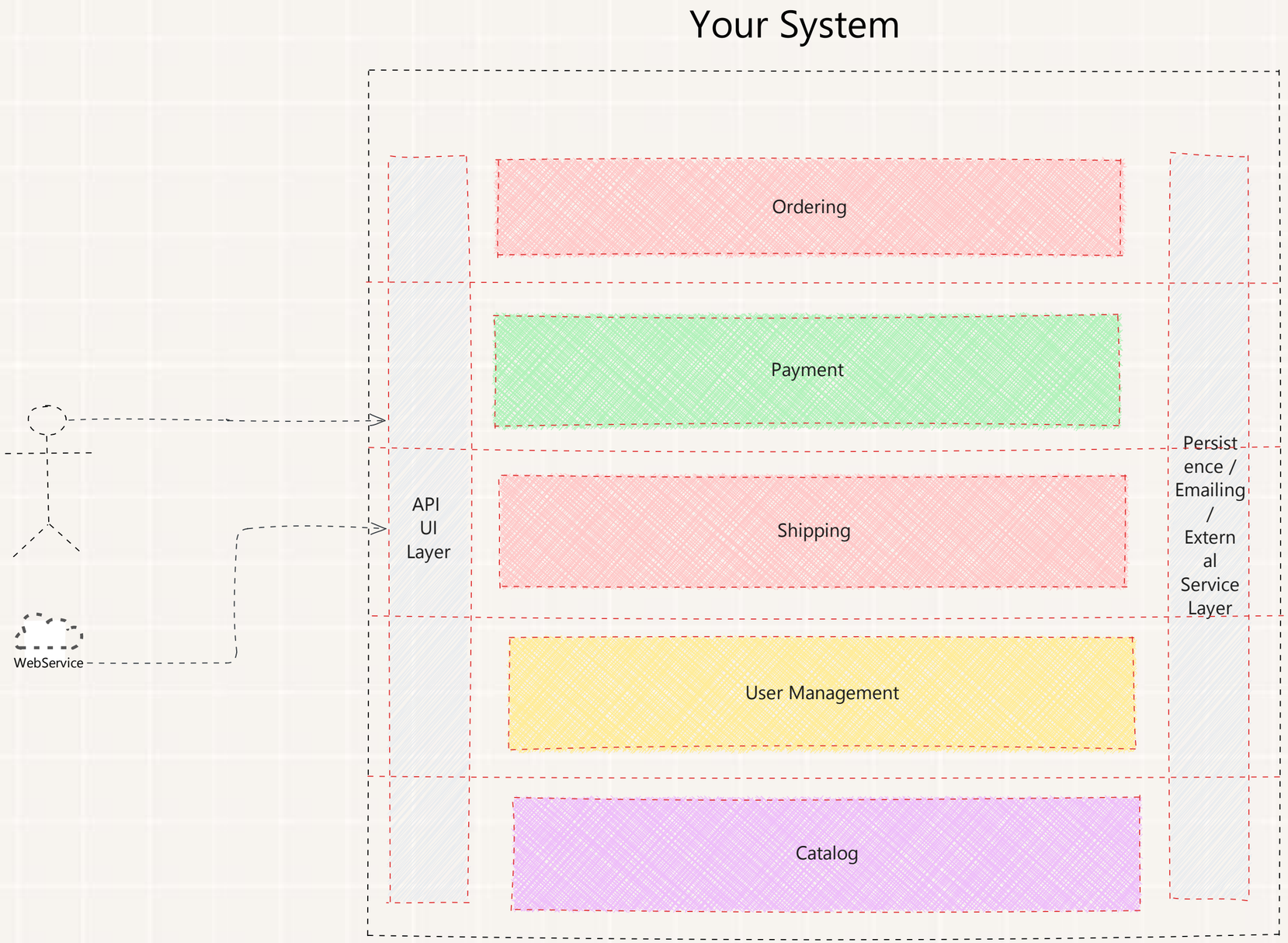


```
async createOrder(req: express.Request): Promise<Order> {
    // Business validation mixed with HTTP request access
    if (!req.body.items || !req.body.customerId) {
        throw new Error('Invalid request body');
    }

    // Business logic calculating totals from HTTP request
    const totalAmount = req.body.items.reduce((sum: number, item: any) => {
        return sum + (item.price * item.quantity);
    }, 0);

    // Business rules mixed with HTTP data
    let discount = 0;
    if (req.body.promoCode === 'SUMMER2024') {
        discount = totalAmount * 0.1;
    }

    // Payment processing using HTTP request data
    const paymentResult = await this.paymentGateway.processPayment({
        amount: totalAmount - discount,
        paymentMethod: req.body.paymentMethod
    });

    // Saving order using HTTP request data
    const order = await this.orderRepository.save({
        customerId: req.body.customerId,
        items: req.body.items,
        totalAmount,
        discount,
        paymentId: paymentResult.id,
        shippingAddress: req.body.shippingAddress
    });

    return order;
}
```

```typescript
// Identity and Authentication
class UserIdentity {
    constructor(
        private id: string,
        private email: string,
        private password: string,
        private lastLoginTime: Date
    ) {}
}

// Profile Information
class UserProfile {
    constructor(
        private userId: string,
        private firstName: string,
        private lastName: string,
        private avatar: string,
        private bio: string
    ) {}

}

// Payment Management
class PaymentProfile {
    private paymentMethods: PaymentMethod[] = [];

}

// Shopping Functionality
class ShoppingProfile {
    constructor(
        private userId: string,
        private wishlist: Product[] = [],
        private shoppingCart: CartItem[] = []
    ) {}
```

```typescript
// Order Management
class OrderHistory {
    constructor(
        private userId: string,
        private orders: Order[] = [],
        private returns: Return[] = []
    ) {}


}

// Social Features
class SocialProfile {
    constructor(
        private userId: string,
        private followers: string[] = [],
        private following: string[] = [],
        private posts: Post[] = []
    ) {}


// Notification Preferences
class NotificationPreferences {
    constructor(
        private userId: string,
        private emailEnabled: boolean = true,
        private pushEnabled: boolean = true,
        private frequency: NotificationFrequency = 'daily'
    ) {}

}

// Analytics Tracking
class UserAnalytics {
    constructor(
        private userId: string,
        private loginHistory: LoginRecord[] = [],
        private deviceInfo: DeviceInfo[] = []
    ) {}

}
```

Be Careful not to break a lot of your code because the cost of change will be also higher


Your System
Ordering
Payment
Shipping
User Management
Catalog
API
UI
Layer
Persist
ence /
Emailing
/
Extern
al
Service
Layer
WebService

04

The real role of the architect

3. The Skills 🛠️

1. Think ahead (the vision)
2. Be the leader not the boss
3. Collect feedback from developers and act accordingly
4. Explain the why and be challenged
5. Be the guarantor of the architecture

05

Conclusion

5. Conclusion

1. Understand and detect coupling
2. Learn the **domain** to know what cohesive code is.
3. Find balance between coupling and cohesion
4. Implement - collect feedback - reiterate.

05 . Conclusion

Thanks

Do you have any questions?

abdelkrim.bournane@gmail.com
abdelkrim.blog

@codewkarim

/in/abdelkrim-bournane/

@karimkos