Architecture Logicielle

1. MVC (Model-View-Controller)

Définition

Le modèle MVC est un patron architectural largement utilisé dans le développement d'applications web. Il se divise en trois composants distincts :

- **Model (Modèle)** : Ce composant gère la logique de données et les règles métier. Il représente l'état de l'application et les opérations qui peuvent être effectuées sur ces données (comme la création, la mise à jour et la suppression).
- **View (Vue)** : La vue est responsable de la présentation des données à l'utilisateur. Elle reçoit les entrées de l'utilisateur et les envoie au contrôleur pour traitement.
- **Controller (Contrôleur)** : Le contrôleur agit comme un intermédiaire entre le modèle et la vue. Il traite les entrées de l'utilisateur, effectue les opérations nécessaires sur le modèle et met à jour la vue en conséquence.

Exemple de Code

JavaScript (Express.js) :

```
// Modèle (model.js)
let users = []; // Un tableau pour stocker les utilisateurs

function addUser(user) {
    users.push(user); // Ajouter un utilisateur
}

function getUsers() {
    return users; // Récupérer la liste des utilisateurs
}

// Contrôleur (controller.js)
const UserModel = require('./model');

function createUser(req, res) {
    UserModel.addUser(req.body); // Ajouter un utilisateur à partir de la requête
    res.status(201).send('User created'); // Répondre avec un message de
confirmation
}

function getUsers(req, res) {
    const users = UserModel.getUsers(); // Récupérer les utilisateurs
    res.status(200).json(users); // Retourner les utilisateurs au format JSON
}

// Vue (app.js avec Express)
const express = require('express');
```

```
const bodyParser = require('body-parser');
const app = express();
app.use(bodyParser.json());

app.post('/users', createUser); // Route pour créer un utilisateur
app.get('/users', getUsers); // Route pour obtenir les utilisateurs

app.listen(3000, () => {
    console.log('Server running on port 3000'); // Indiquer que le serveur est en
fonctionnement
});
```

Java (Spring Boot) :

```
// Modèle (User.java)
public class User {
    private String name; // Nom de l'utilisateur
    private String email; // Email de l'utilisateur
    // Getters et setters pour accéder aux attributs
}

// Repository (UserRepository.java)
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
    // Interface pour gérer les opérations sur les utilisateurs
}

// Contrôleur (UserController.java)
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/users") // Route de base pour les utilisateurs
public class UserController {
    private final UserRepository userRepository; // Référencer le repository

    public UserController(UserRepository userRepository) {
        this.userRepository = userRepository; // Initialiser le repository
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userRepository.save(user); // Ajouter un utilisateur
    }

    @GetMapping
    public List<User> getUsers() {
        return userRepository.findAll(); // Récupérer la liste des utilisateurs
```

```
    }
}
```

2. Microservices

Définition

L'architecture microservices consiste à décomposer une application en plusieurs services autonomes, chacun responsable d'une fonctionnalité ou d'une tâche spécifique. Chaque microservice fonctionne indépendamment et communique avec d'autres microservices via des API RESTful. Cette approche permet une meilleure scalabilité, une facilité de déploiement et une flexibilité dans le choix des technologies utilisées.

Exemple de Code

JavaScript (Express.js pour un service) :

```
const express = require('express');
const app = express();
app.use(express.json());

// Service de gestion des utilisateurs
app.get('/users', (req, res) => {
    res.json([{ id: 1, name: 'John Doe' }]); // Exemple de réponse avec un
utilisateur
});

// Service de gestion des commandes
app.post('/orders', (req, res) => {
    res.status(201).send('Order created'); // Répondre avec un message de
confirmation de création de commande
});

app.listen(3000, () => {
    console.log('User service running on port 3000'); // Indiquer que le service des
utilisateurs est en fonctionnement
});
```

Java (Spring Boot pour un service) :

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.List;

@SpringBootApplication
@RestController
@RequestMapping("/products") // Route de base pour les produits
public class ProductService {
    private final List<Product> products = new ArrayList<>(); // Liste pour stocker
les produits

    public static void main(String[] args) {
        SpringApplication.run(ProductService.class, args); // Démarrer le service
    }

    @PostMapping
    public Product createProduct(@RequestBody Product product) {
        products.add(product); // Ajouter un produit
        return product; // Retourner le produit créé
    }

    @GetMapping
    public List<Product> getProducts() {
        return products; // Récupérer la liste des produits
    }
}

// Modèle (Product.java)
class Product {
    private String name; // Nom du produit
    private double price; // Prix du produit
    // Getters et setters pour accéder aux attributs
}
```

3. Hexagonal Architecture / Architecture en Couches

Définition

L'architecture hexagonale, également connue sous le nom d'architecture en couches, vise à séparer les préoccupations en divisant une application en plusieurs couches. Cela rend le système plus modulaire et facilite les tests unitaires. Les principales couches incluent :

- **Couches externes** : Gèrent les interfaces utilisateur, les API et toute autre interaction avec le monde extérieur.
- **Couches internes** : Contiennent la logique métier et les services de l'application. Ces couches ne dépendent pas des couches externes.

Exemple de Code

JavaScript :

```
// Couche de services (service.js)
function calculateTotalPrice(order) {
    return order.items.reduce((total, item) => total + item.price, 0); // Calculer
le total
}

// Couche de contrôleur (controller.js)
function handleOrder(req, res) {
    const totalPrice = calculateTotalPrice(req.body); // Appeler la fonction du
service
    res.status(200).send(`Total Price: ${totalPrice}`); // Retourner le prix total
}

// Couche d'application (app.js)
const express = require('express');
const app = express();
app.use(express.json());

app.post('/orders', handleOrder); // Route pour gérer les commandes

app.listen(3000, () => {
    console.log('Order service running on port 3000'); // Indiquer que le service
des commandes est en fonctionnement
});
```

Java :

```
// Couche de service (OrderService.java)
public class OrderService {
    public double calculateTotalPrice(Order order) {
        return order.getItems().stream()
                .mapToDouble(Item::getPrice)
                .sum(); // Calculer le prix total des articles
    }
}

// Couche de contrôleur (OrderController.java)
@RestController
@RequestMapping("/orders") // Route de base pour les commandes
public class OrderController {
    private final OrderService orderService; // Référencer le service des commandes

    public OrderController(OrderService orderService) {
        this.orderService = orderService; // Initialiser le service
    }
```

```
    @PostMapping
    public String handleOrder(@RequestBody Order order) {
        double totalPrice = orderService.calculateTotalPrice(order); // Appeler le
service pour calculer le total
        return "Total Price: " + totalPrice; // Retourner le prix total
    }
}

// Modèle (Order.java)
public class Order {
    private List<Item> items; // Liste des articles dans la commande
    // Getters et setters
}

// Modèle (Item.java)
public class Item {
    private String name; // Nom de l'article
    private double price; // Prix de l'article
    // Getters et setters
}
```