|  |
| --- |
|  |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ |
| Федеральное государственное бюджетное образовательное учреждение высшего образования  **"МИРЭА - Российский технологический университет"**  **РТУ МИРЭА** |
| Институт информационных технологий (ИТ) |
| Кафедра инструментального и прикладного программного обеспечения |

|  |  |
| --- | --- |
|  | |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ №5** | |
| **по дисциплине** | |
| **«Архитектура клиент-серверных приложений»**  **Тема: «**Введение в многослойные клиент-серверные архитектуры**»** | |
| Выполнил студент группы **ИКБО-20-21** | Сидоров С.Д. |
| Принял преподаватель кафедры ИиППО | Волков М.Ю. |

|  |  |  |
| --- | --- | --- |
| Практические работы выполнены | « » 2023г. |  |
| «Зачтено» | « » 2023г. |  |

Москва 2023

**Теоретическое введение**

В предыдущих практических работах мы неявно познакомились с одной из архитектур распределенных систем: клиент-серверная архитектура.

‎**Тонкие клиенты‎**

‎Тонкий клиент спроектирован так, чтобы основная часть обработки данных происходила на сервере. Тонкий клиент как правило без жесткого диска: действуют как простой терминал к серверу и требует постоянной связи с сервером. ‎

‎‎**Толстые клиенты‎**

‎Толстый клиент выполняет основную часть обработки. У толстых клиентов нет необходимости в непрерывной связи с сервером, поскольку они в основном передают информацию на сервер.

**Когда какой клиент использовать‎**

Тонкие клиенты обеспечивают работу рабочего стола в средах, где конечный пользователь имеет четко определенное и регулярное количество задач, для которых используется система. Тонких клиентов можно найти в медицинских офисах, авиабилетах, школах, правительствах, производственных предприятиях и даже колл-центрах. Наряду с простотой установки, тонкие клиенты также предлагают более низкую общую стоимость владения по сравнению с толстыми клиентами.

Если вашим приложениям требуются мультимедийные компоненты или которые интенсивно используют пропускную способность, стоит присмотреться к разработке толстых клиентов. Одно из самых больших преимуществ толстых клиентов – некоторые операционные системы и программное обеспечение не могут работать на тонких клиентах. Толстые клиенты могут справиться с ними, поскольку у них есть свои собственные ресурсы.

**Разница между «уровнем» и «слоем» в контексте распределённых систем**

«Слой» и «уровень» относятся к функциональной части программного обеспечения, но «уровень» относится к ПО, которое работает в инфраструктуре отдельных частей приложения (на разных физических системах). Например, приложение "Контакты" на телефоне является трехслойным приложением, но одноуровневым приложением, потому что все три слоя работают на одном физическом устройстве – на телефоне. Так, «слои» не могут предложить тех преимуществ, которыми обладают «уровни».

**Трёхуровневая архитектура распределённых систем**

Трёхуровневая архитектура (рисунок 5.1) организует приложения в три логических и физических вычислительных уровня: уровень представления (пользовательский интерфейс), уровень приложений (обработка данных происходит здесь) и уровень данных (хранение и управление данными). Поскольку каждый уровень работает в своей собственной инфраструктуре, он может разрабатываться одновременно отдельной командой разработчиков и может обновляться или масштабироваться по мере необходимости, не влияя на другие уровни, не говоря уже о реплицировании.

В трехуровневом приложении вся связь проходит через уровень приложения. Уровень представления и уровень данных не могут напрямую взаимодействовать друг с другом.

**Уровень презентации**

Это пользовательский интерфейс и слой связи приложения, на котором конечный пользователь взаимодействует с приложением. Его основная цель – отображать информацию для и собирать информацию от пользователя. Уровень презентации может работать в веб-браузере, в качестве настольного приложения или графического пользовательского интерфейса (GUI).

**Уровень приложений**

Уровень приложений, также известный как уровень логики или средний уровень, является сердцем приложения. На этом уровне информация, собранная на уровне представления, обрабатывается с использованием бизнес-логики. Уровень приложений также может добавлять, удалять или изменять данные на уровне данных.

**Уровень данных**

Уровень данных, иногда называемый уровнем базы данных, уровнем доступа к данным или серверной частью, – это место, где хранится и управляется информация, обрабатываемая приложением.

**Зачем нужна трёхуровневая архитектура в распределённых системах**

Как уже отмечалось ранее, главное преимущество трехуровневой архитектуры заключается в ее логическом и физическом разделении функций. Каждый уровень может работать на отдельной операционной системе и серверной платформе. Соответственно, не важно где будут расположены платформы: в облаке, в одном здании, на разных континентах. В любом случае части распределенной системы должны быть соединены надежными и защищенными линиями связи. Что касается скорости передачи данных, то она в значительной степени зависит от важности соединения между двумя частями системы с точки зрения обработки и передачи данных и в меньшей степени от их удаленности. Службы каждого уровня могут быть настроены и оптимизированы без влияния на другие уровни.

**Другие преимущества:**

• более быстрая разработка: отдельный уровень может разрабатываться одновременно разными командами, организация может быстрее выводить приложение на рынок, а программисты могут использовать новейшие и лучшие языки и инструменты в соответствии с каждым уровнем;

• улучшенная масштабируемость: отдельный уровень можно масштабировать независимо от других по мере необходимости;

• повышенная надежность: сбой на одном уровне с меньшей вероятностью повлияет на доступность или производительность других уровней;

• улучшенная безопасность: так как уровень представления и уровень данных не могут взаимодействовать напрямую, хорошо спроектированный уровень приложений может функционировать как своего рода внутренний брандмауэр, предотвращающий инъекции SQL и другие вредоносные эксплойты.

**Постановка задачи**

Поскольку для трёхуровневой архитектуры необходимо физическое разделение подсистем, то предлагается разработать трёхслойное приложение: БД, сервер, приложение. В качестве приложения можно использовать: запросы в postman/insomnia/testmace, простой сайт, десктопное приложение, мобильное приложение. В качестве БД можно использовать: SQLite, PostgreSQL.

Нечетные варианты реализуют синхронное взаимодействие через API. Чётные варианты реализуют несохранное асинхронное взаимодействие через WebSockets.

**Программный код**

В листингах 1 – 3 продемонстрирован программный код части классов серверного приложения на Java Spring. В листингах 4 – 5 продемонстрирован код Docker-compose и Dockerfile необходимые для сборки и работы сервера.

Листинг 1 – ProductController.java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/products")

public class ProductController {

@Autowired

private ProductService productService;

@GetMapping

public ResponseEntity<List<Product>> getProductById() {

return ResponseEntity.ofNullable(productService.getProductAll());

}

@GetMapping("/{id}")

public ResponseEntity<Product> getProductById(@PathVariable Long id) {

Product product = productService.getProductById(id);

return ResponseEntity.ok(product);

}

@PostMapping

public ResponseEntity<Product> createProduct(@RequestBody Product product) {

Product createdProduct = productService.createProduct(product);

return ResponseEntity.status(HttpStatus.CREATED).body(createdProduct);

}

@PutMapping("/{id}")

public ResponseEntity<Product> updateProduct(@PathVariable Long id, @RequestBody Product product) {

Product updatedProduct = productService.updateProduct(id, product);

return ResponseEntity.ok(updatedProduct);

}

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {

productService.deleteProduct(id);

return ResponseEntity.noContent().build();

}

}

Листинг 2 – Product.java

import jakarta.persistence.\*;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import java.math.BigDecimal;

@AllArgsConstructor

@NoArgsConstructor

@Data

@Entity

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Column(name = "price")

private BigDecimal price;

@Column(name = "type")

private String type;

@Column(name = "name")

private String name;

@Column(name = "amount")

private Integer amount;

}

Листинг 3 – ProductServiceS.java

package mirea.ru.prakt6.service;  
  
import mirea.ru.prakt6.model.Product;  
import mirea.ru.prakt6.repos.ProductRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
  
@Service  
public class ProductService {  
  
 @Autowired  
 private ProductRepository productRepository;  
  
 public List<Product> getProductAll(){  
 return productRepository.findAll();  
 }  
  
 public Product getProductById(Long id) {  
 return productRepository.findById(id).orElse(null);  
 }  
  
 public Product createProduct(Product product) {  
 return productRepository.save(product);  
 }  
  
 public Product reduceAmount(Long id, Integer amount){  
 Product exProduct = getProductById(id);  
 if (exProduct.getAmount() < amount){  
 return null;  
 }  
  
 exProduct.setAmount(exProduct.getAmount() - amount);  
 productRepository.save(exProduct);  
 return exProduct;  
 }  
  
 public Product updateProduct(Long id, Product product) {  
 Product existingProduct = getProductById(id);  
 existingProduct.setName(product.getName());  
 existingProduct.setType(product.getType());  
 existingProduct.setPrice(product.getPrice());  
 return (Product) productRepository.save(existingProduct);  
 }  
  
 public void deleteProduct(Long id) {  
 Product existingProduct = getProductById(id);  
 productRepository.delete(existingProduct);  
 }  
}

Листинг 4 – Docker-compose.yml

version: '3'  
services:  
 db:  
 image: postgres  
 restart: always  
 environment:  
 POSTGRES\_USER: root  
 POSTGRES\_PASSWORD: password  
 POSTGRES\_DB: GetContact  
 ports:  
 - "5433:5432"  
 volumes:  
 - ./init.sql:/docker-entrypoint-initdb.d/init.sql  
 app:  
 build:  
 context: .  
 dockerfile: Dockerfile  
 ports:  
 - "8081:8080"  
 depends\_on:  
 - db  
 environment:  
 - SPRING\_DATASOURCE\_URL=jdbc:postgresql://db:5432/prakt6  
 - SPRING\_DATASOURCE\_USERNAME=root  
 - SPRING\_DATASOURCE\_PASSWORD=password

Листинг 5 – dockerfile

FROM gradle:latest AS *builder*COPY --chown=gradle:gradle . /home/gradle/src  
WORKDIR /home/gradle/src  
RUN gradle build  
FROM openjdk:20  
COPY --from=*builder* /home/gradle/src/build/libs/prakt6-0.0.1-SNAPSHOT.jar /app/app.jar  
EXPOSE 8081  
ENTRYPOINT ["java", "-jar", "/app/app.jar"]

**Вывод программы**

Чтобы начать работу приложения необходимо сначала запустить сервер (рис. 1). В качестве приложения используется postman, через него был отправлен тестовый запрос (рис. 2).

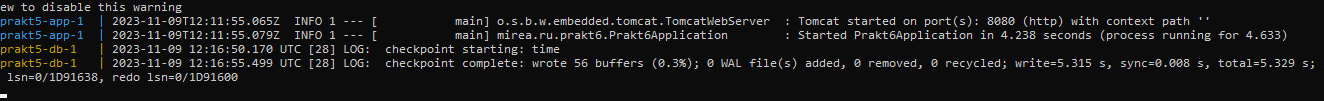


Рисунок 1 – Запуск сервера и структура проекта

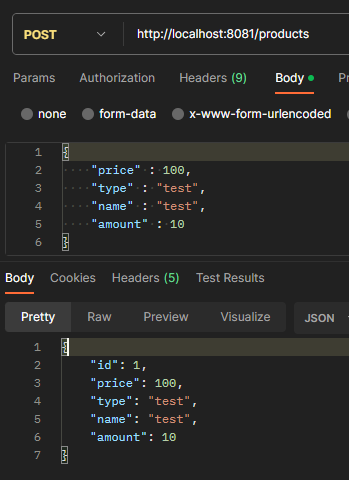


Рисунок 2 – Работа приложения

**Вывод**

В результате выполнения данной практической работы было проведено ознакомление с многослойными клиент-серверными архитектурами, посмотреть разницу между ними, выявить плюсы и минусы.