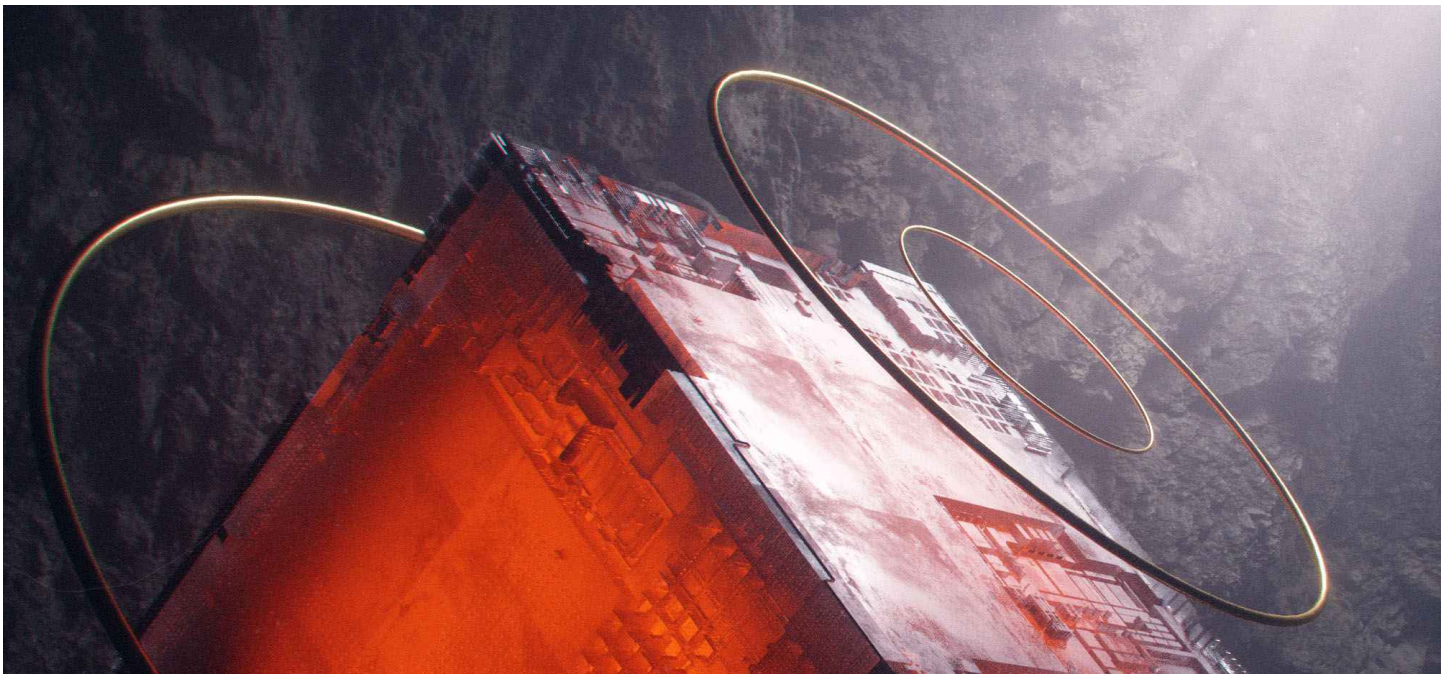


Train-ticket

TODO-List

- [x] ~~统计对于UI来说的API有哪些 - due:4.4~~
- [x] ~~理解API互相之间的联系，对应UI的什么操作 - due:4.7~~
- [x] ~~设计并生成符合逻辑的请求序列。要求可以随机生成，可以组合。这部分需要先写一个大致的技术文档来说明你将会怎么实现(为了减少后期大改的风险) —— 技术文档 - due:4.14~~
- [ ] 设计并生成符合逻辑的请求序列。要求可以随机生成，可以组合。这部分需要先写一个大致的技术文档来说明你将会怎么实现(为了减少后期大改的风险) —— 实现 - due:4.14

1. Start

1.1 Initialization

a. Environment

i. Intellij IDEA - 2023.3.6

ii. Maven - 3.9.6

iii. Dependency(依赖)：

b. 一般依赖直接使用Maven导入；

c. Lombok可以使用Maven导入；如果因为版本原因，可以引入jar包到本地。

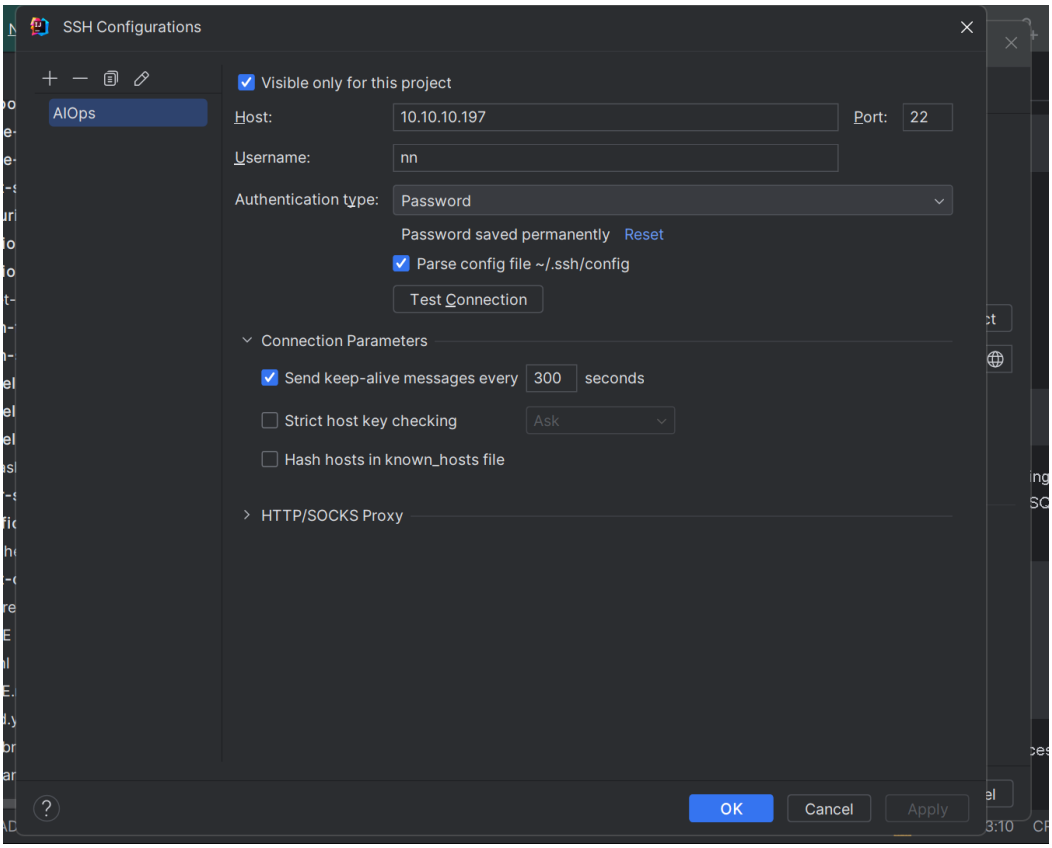
d. 核心环境为：（本地不需要安装） 使用虚拟机

   i. Minikube (https://kubernetes.io/zh-cn/docs/tutorials/configuration/configure-java-microservice/configure-java-microservice/)

   ii. Helm

   iii. Skaffold

e. Deployment & SSH



1.2 Run

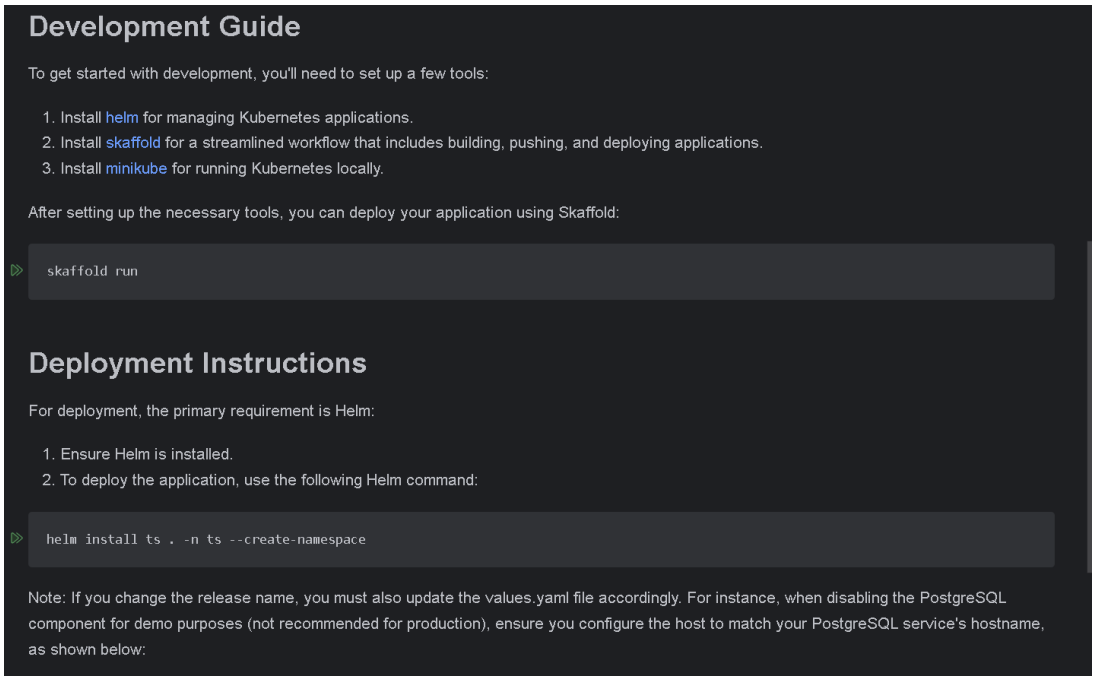
a. In terminal, run command “mvn clean package -DskipTests=true”

b. And, run command “mvn spring-boot:run” .

c. To deploy, run command “skaffold run --kubeconfig ~/.kube/config --default-repo 192.168.1.105/library” in the corresponding catalog, we can deploy the server on the k8s.

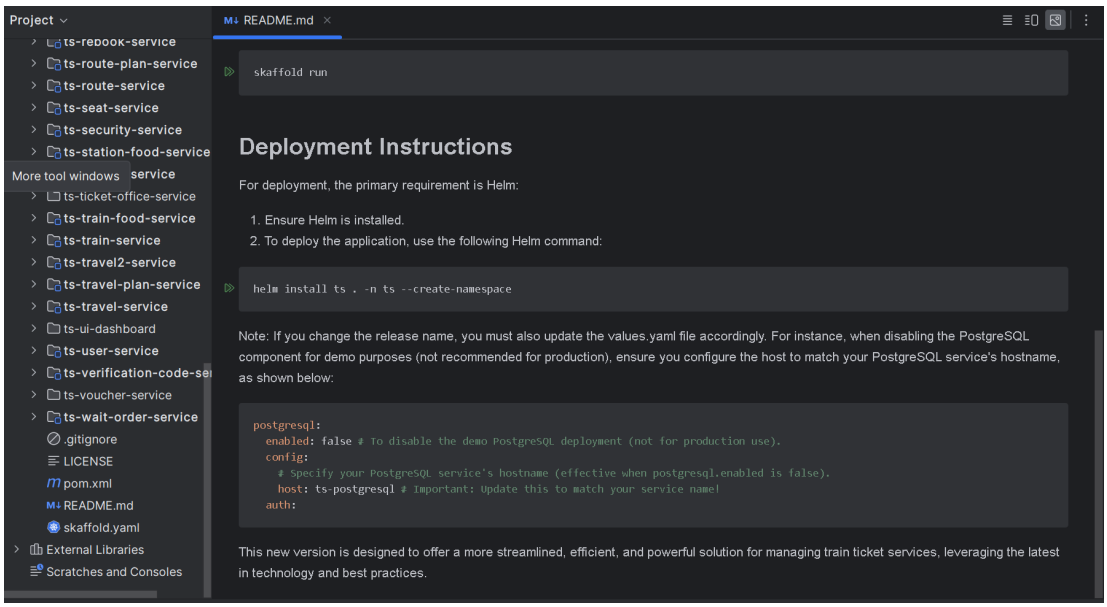
d. Development Guide:

   i. After setting up the necessary tools, you can deploy your application using Skaffold: "skaffold run"



ii. For deployment, the primary requirement is Helm:

1. Ensure Helm is installed.
2. To deploy the application, use the following Helm command: "helm install ts . -n ts --create-namespace"



iii. Note: If you change the release name, you must also update the values.yaml file accordingly. For instance, when disabling the PostgreSQL component for demo purposes (not recommended for production), ensure you configure the host to match your PostgreSQL service's hostname, as shown below:

```
postgresql:
  enabled: false # To disable the demo PostgreSQL deployment (not for production use).
  config:
```

```
4     # Specify your PostgreSQL service's hostname (effective when
  postgresql.enabled is false).
5     host: ts-postgresql # Important: Update this to match your service
  name!
6   auth:
```

1.3 API for UI

1.3.1 Basic knowledge

i. 在软件开发中，UI（User Interface）指的是用户界面，是用户与计算机交互的接口。UI设计包括了用户可以看到、感知和操作的各种元素，如图形、按钮、文本框等，目的是提供一个直观、易用的界面，使用户能够轻松地与应用程序进行交互。

ii. API（Application Programming Interface）指的是应用程序接口，是一组定义了软件组件之间交互规范的接口。API定义了如何与软件组件进行通信和交互，通常由一系列的函数、方法、协议和工具组成，开发人员可以使用这些接口来构建应用程序、实现功能或者访问特定的服务。

iii. 对于UI来说，API通常用于与后端服务进行交互，从而获取数据、执行操作等。一般来说，与UI相关的API可以分为以下几类：

1. 用户认证和登录功能：
    - Java后端可以使用Spring Security等框架来实现用户认证和授权功能。
    - 在IntelliJ IDEA中，您可以创建一个基于Spring Boot的项目，并使用Spring Security配置用户认证。然后，编写前端页面来实现用户登录界面，使用AJAX请求将用户输入的用户名和密码发送到后端进行认证。

2. 数据获取API：用于从后端服务获取数据，如获取用户信息、获取产品列表等。
    - Java后端可以使用Spring MVC或Spring Boot等框架来提供RESTful API，从数据库中获取数据并返回给前端。
    - 在IntelliJ IDEA中，您可以创建一个Spring Boot项目，并编写Controller来处理前端的请求。然后，编写前端页面来展示数据，使用JavaScript或者Vue.js等前端框架来处理数据展示逻辑。

3. 数据提交API：用于向后端服务提交数据，如用户提交表单、上传文件等。

4. 认证和授权API：用于用户登录、权限验证等身份认证和授权操作。

5. 业务逻辑API：用于执行特定的业务逻辑操作，如购买商品、创建订单等。

6. 通知和推送API：用于向用户发送通知消息或推送信息。

7. 文件操作API：用于文件的上传、下载和管理等操作。
    - Java后端可以使用Spring MVC或Spring Boot提供的MultipartResolver来处理文件上传，并提供文件下载的接口。
    - 在IntelliJ IDEA中，您可以创建一个Spring Boot项目，并编写Controller来处理文件上传和下载的请求。然后，编写前端页面来实现文件上传和下载的功能，可以使用HTML表单和JavaScript来实现文件上传，使用超链接或者JavaScript来实现文件下载。

8. 地理位置和地图API：用于获取地理位置信息、显示地图等操作。
    - Java后端可以使用地图API（如百度地图API、高德地图API等）来获取地理位置信息，并将地图展示在前端页面上。
    - 在IntelliJ IDEA中，您可以创建一个Spring Boot项目，并编写Controller来处理获取地理位置信息的请求。然后，编写前端页面来展示地图，可以使用JavaScript和地图API来实现地图展示功能。

9. 支付和金融交易API：用于处理支付操作和金融交易。

iv. 这些API通常由后端服务提供，并通过网络协议（如HTTP）暴露给UI层。UI层通过调用这些API来实现与后端服务的交互，从而实现用户界面上的各种功能和操作。

1.3.2 API互相之间的联系 & 对应的UI操作

i. 用户认证API：

- 联系：用户认证API通常与用户信息API、权限管理API等关联，用于验证用户的身份，并控制其访问权限。
- UI操作：UI可能包括登录、注册、忘记密码等操作，用户通过输入用户名和密码进行身份验证。

ii. 数据获取和提交API：

- 联系：数据获取API通常与数据提交API相关联，用于从后端获取数据并将用户的修改提交回后端。
- UI操作：UI可能包括展示数据列表、表单填写、数据编辑等操作，用户可以通过UI界面浏览数据、填写表单并提交数据。

iii. 支付和订单API：

- 联系：支付API通常与订单API相关联，用于处理用户的支付请求并更新订单状态。
- UI操作：UI可能包括选择商品、填写订单信息、选择支付方式等操作，用户通过UI界面完成购买并进行支付。

iv. 通知和推送API：

- 联系：通知和推送API通常与用户认证API相关联，用于向特定用户发送消息或推送通知。
- UI操作：UI可能包括消息列表、消息详情等操作，用户通过UI界面查看接收到的通知和消息。

v. 地理位置和地图API：

- 联系：地理位置和地图API通常与数据获取API相关联，用于获取地理位置信息并在地图上展示。
- UI操作：UI可能包括地图展示、地点搜索、定位信息等操作，用户通过UI界面查看地图并与地图交互。

vi. 文件操作API：

- 联系：文件操作API通常与数据提交API相关联，用于上传、下载和管理文件。
- UI操作：UI可能包括文件上传、文件下载、文件列表等操作，用户通过UI界面进行文件操作。

2. 根据API模拟真实请求 & 设计并生成符合逻辑的请求序列

——设计并生成符合逻辑的请求序列。要求可以随机生成，可以组合

2.1.1 Design

a. 定义模拟的API接口

- 我们需要定义一些模拟的API接口，这些接口将用于生成请求序列。每个接口应包含URL、请求方法、参数等信息。可以使用Java类或接口来表示API接口。

b. 设计请求序列生成逻辑

- 根据业务逻辑和需求，组合不同的API接口，并生成符合逻辑的请求序列

c. 随机生成请求

- 我们将使用Java的随机数生成器来随机选择API接口，并生成请求序列。通过随机生成的方式，我们可以实现请求的多样性和随机性。

d. 组合请求序列

- 为了进一步增加请求序列的多样性，我们还可以将多个请求组合起来，形成更复杂的请求序列。例如，可以先发送获取用户信息的请求，然后根据用户信息再发送其他相关的请求。

e. 代码示例：

```java
import java.util.Random;

public class ApiSimulation {

    private static final Random random = new Random();

    public static void main(String[] args) {
        int n = 100000; // 要生成的请求数量
        for (int i = 0; i < n; i++) {
            generateRandomRequest();
        }
    }

    private static void generateRandomRequest() {
        int requestType = random.nextInt(5); // 随机选择请求类型
        switch (requestType) {
            case 0:
                // 发送获取用户信息的请求...
                break;
            case 1:
                // 发送更新用户信息的请求...
                break;
            case 2:
                // 发送获取商品信息的请求...
                break;
            case 3:
                // 发送创建订单的请求...
```

```
                break;
            case 4:
                // 发送获取订单信息的请求...
                break;
        }
    }
}

```

2.1.2 获取API

a. 当使用Spring Boot时，可以使用 `RestTemplate` 来发送HTTP请求，并结合Spring的依赖注入和AOP等特性，实现更加方便的API模拟和请求序列生成。

i. 首先，需要在 `pom.xml` 文件中添加 `spring-boot-starter-web` 依赖，以便使用Spring Boot的Web功能和 `RestTemplate`；（或者使用Spring Initializer生成spring boot项目时选择web依赖项）

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

ii. 然后，编写一个Spring Boot的Service类来实现API的模拟和请求序列生成逻辑。在这个示例中，我们定义了一个 `ApiSimulationService` 类，其中包含了四个方法分别用于模拟发送HTTP GET、POST、PUT和DELETE请求。这些方法通过 `RestTemplate` 来发送HTTP请求，并返回或处理响应数据。

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class ApiSimulationService {

    @Autowired
    private RestTemplate restTemplate;

    // 模拟发送HTTP GET请求
```

```
    public String simulateGetRequest(String apiUrl) {
        return restTemplate.getForObject(apiUrl, String.class);
    }

    // 模拟发送HTTP POST请求
    public String simulatePostRequest(String apiUrl, Object requestBody)
{
        return restTemplate.postForObject(apiUrl, requestBody,
String.class);
    }

    // 模拟发送HTTP PUT请求
    public void simulatePutRequest(String apiUrl, Object requestBody) {
        restTemplate.put(apiUrl, requestBody);
    }

    // 模拟发送HTTP DELETE请求
    public void simulateDeleteRequest(String apiUrl) {
        restTemplate.delete(apiUrl);
    }
}
```

iii. 最后，我们在Spring Boot应用程序中调用这些方法来模拟API的请求序列：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application implements CommandLineRunner {

    @Autowired
    private ApiSimulationService apiSimulationService;

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        // 模拟发送GET请求
        String getResponse =
apiSimulationService.simulateGetRequest("https://api.example.com/data");
```

```
        System.out.println("GET Response: " + getResponse);

        // 模拟发送POST请求
        String postResponse = apiSimulationService.simulatePostRequest("https://api.example.com/create", new Object());
        System.out.println("POST Response: " + postResponse);

        // 模拟发送PUT请求
        apiSimulationService.simulatePutRequest("https://api.example.com/update", new Object());

        // 模拟发送DELETE请求
        apiSimulationService.simulateDeleteRequest("https://api.example.com/delete");
    }
}
```

iv. 对于模拟成千上万次（n = 10^5）的请求，可以使用Java的多线程来并发地发送请求。以下是使用Java的Executor框架来实现并发请求：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import java.util.concurrent.Executor;

@SpringBootApplication
@EnableAsync
public class Application implements CommandLineRunner {

    @Autowired
    private ApiSimulationService apiSimulationService;

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
```

```
    }

    @Override
    public void run(String... args) throws Exception {
        int n = 100000; // 要发送的请求数量
        Executor executor = createExecutor();
        for (int i = 0; i < n; i++) {
            executor.execute(() -> {
                // 模拟发送GET请求
                String getResponse =
apiSimulationService.simulateGetRequest("https://api.example.com/data");
                System.out.println("GET Response: " + getResponse);

                // 模拟发送POST请求
                String postResponse =
apiSimulationService.simulatePostRequest("https://api.example.com/create"
, new Object());
                System.out.println("POST Response: " + postResponse);

                // 模拟发送PUT请求

apiSimulationService.simulatePutRequest("https://api.example.com/update",
 new Object());

                // 模拟发送DELETE请求

apiSimulationService.simulateDeleteRequest("https://api.example.com/delet
e");
            });
        }
    }

    private Executor createExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(20); // 设置核心线程池大小
        executor.setMaxPoolSize(100); // 设置最大线程池大小
        executor.setQueueCapacity(500); // 设置队列容量
        executor.setThreadNamePrefix("ApiRequestThread-"); // 设置线程名称
前缀
        executor.initialize();
        return executor;
    }
}

```

2.1.3 根据API模拟真实请求（设计并生成符合逻辑的请求序列。要求可以随机生成，可以组合）

a. 为了模拟真实请求并生成符合逻辑的请求序列，您可以首先定义一些模拟的API接口，然后根据业务逻辑和需求设计和组合这些API，最后通过随机生成的方式生成请求序列。以下是一个简单的示例：

b. 假设我们有以下几个模拟的API接口：

   i. 获取用户信息的API：GET /api/user/{userId}

   ii. 更新用户信息的API：PUT /api/user/{userId}

   iii. 获取商品信息的API：GET /api/product/{productId}

   iv. 创建订单的API：POST /api/order

   v. 获取订单信息的API：GET /api/order/{orderId}

   现在，我们要设计一个随机生成的请求序列，例如：

   i. 随机选择一个用户，然后发送获取用户信息的请求。

   ii. 随机选择一个商品，然后发送获取商品信息的请求。

   iii. 使用同一个用户和商品，发送创建订单的请求。

   iv. 使用同一个用户，更新用户信息的请求。

   v. 随机选择一个订单，然后发送获取订单信息的请求。

c. 以下是一个简单的示例(n = 10)，用于模拟生成符合逻辑的请求序列：

```java
import java.util.Random;

public class ApiSimulation {

    private static final Random random = new Random();

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) { // 假设生成10个请求
            int requestType = random.nextInt(5); // 随机选择请求类型
            switch (requestType) {
                case 0:
                    getUserInfoRequest();
                    break;
                case 1:
```

```
                    updateUserInfoRequest();
                    break;
                case 2:
                    getProductInfoRequest();
                    break;
                case 3:
                    createOrderRequest();
                    break;
                case 4:
                    getOrderInfoRequest();
                    break;
            }
        }
    }

    private static void getUserInfoRequest() {
        int userId = random.nextInt(1000); // 假设用户ID范围在0到999之间
        System.out.println("GET /api/user/" + userId);
        // 发送获取用户信息的请求...
    }

    private static void updateUserInfoRequest() {
        int userId = random.nextInt(1000); // 假设用户ID范围在0到999之间
        System.out.println("PUT /api/user/" + userId);
        // 发送更新用户信息的请求...
    }

    private static void getProductInfoRequest() {
        int productId = random.nextInt(1000); // 假设商品ID范围在0到999之间
        System.out.println("GET /api/product/" + productId);
        // 发送获取商品信息的请求...
    }

    private static void createOrderRequest() {
        int userId = random.nextInt(1000); // 假设用户ID范围在0到999之间
        int productId = random.nextInt(1000); // 假设商品ID范围在0到999之间
        System.out.println("POST /api/order (User ID: " + userId + ",
Product ID: " + productId + ")");
        // 发送创建订单的请求...
    }

    private static void getOrderInfoRequest() {
        int orderId = random.nextInt(1000); // 假设订单ID范围在0到999之间
        System.out.println("GET /api/order/" + orderId);
        // 发送获取订单信息的请求...
    }
}
```

```
61
```

d. 生成n个请求（n = 10^5）

- 我们将生成请求的逻辑放在了一个循环中，并将循环次数设置为n，即要生成的请求数量。每次循环会调用 `generateRandomRequest()` 方法来生成一个随机的请求。您可以根据需要调整循环次数和请求的生成逻辑。

```java
import java.util.Random;

public class ApiSimulation {

    private static final Random random = new Random();

    public static void main(String[] args) {
        int n = 100000; // 要生成的请求数量
        for (int i = 0; i < n; i++) {
            generateRandomRequest();
        }
    }

    private static void generateRandomRequest() {
        int requestType = random.nextInt(5); // 随机选择请求类型
        switch (requestType) {
            case 0:
                getUserInfoRequest();
                break;
            case 1:
                updateUserInfoRequest();
                break;
            case 2:
                getProductInfoRequest();
                break;
            case 3:
                createOrderRequest();
                break;
            case 4:
                getOrderInfoRequest();
                break;
        }
    }

    private static void getUserInfoRequest() {
        int userId = random.nextInt(1000); // 假设用户ID范围在0到999之间
```

```
        System.out.println("GET /api/user/" + userId);
        // 发送获取用户信息的请求...
    }

    private static void updateUserInfoRequest() {
        int userId = random.nextInt(1000); // 假设用户ID范围在0到999之间
        System.out.println("PUT /api/user/" + userId);
        // 发送更新用户信息的请求...
    }

    private static void getProductInfoRequest() {
        int productId = random.nextInt(1000); // 假设商品ID范围在0到999之间
        System.out.println("GET /api/product/" + productId);
        // 发送获取商品信息的请求...
    }

    private static void createOrderRequest() {
        int userId = random.nextInt(1000); // 假设用户ID范围在0到999之间
        int productId = random.nextInt(1000); // 假设商品ID范围在0到999之间
        System.out.println("POST /api/order (User ID: " + userId + ",
Product ID: " + productId + ")");
        // 发送创建订单的请求...
    }

    private static void getOrderInfoRequest() {
        int orderId = random.nextInt(1000); // 假设订单ID范围在0到999之间
        System.out.println("GET /api/order/" + orderId);
        // 发送获取订单信息的请求...
    }
}

```