Tipos de Requisição REST em Java Comparação entre Java 8, 11 e 17

Este artigo apresenta as boas práticas e implementações performáticas de requisições REST em Java, comparando as versões 8, 11 e 17.

Também destacaremos as chamadas POST, PUT, GET e DELETE, abordando as diferentes formas de autenticação, como Basic Auth, JWT e autenticação via usuário e senha.

1. Java 8: Uso de HttpURLConnection

No Java 8, a abordagem padrão para requisições HTTP era o **HttpURLConnection**.

Embora funcional, ela gera muito código boilerplate, o que dificulta a manutenção e pode comprometer a performance, especialmente em requisições complexas ou assíncronas.

GET Request com HttpURLConnection (Basic Auth):

```
URL url = new URL("https://example.com/api/resource");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setRequestMethod("GET");
connection.setRequestProperty("Authorization", "Basic " +
Base64.getEncoder().encodeToString(("user:password").getBytes()));

int responseCode = connection.getResponseCode();
if (responseCode == HttpURLConnection.HTTP_OK) {
    BufferedReader in = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
    String inputLine;
    StringBuffer content = new StringBuffer();
    while ((inputLine = in.readLine()) != null) {
        content.append(inputLine);
    }
    in.close();
}
```

POST Request:

```
URL url = new URL("https://example.com/api/resource");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setRequestMethod("POST");
connection.setDoOutput(true);
connection.setRequestProperty("Content-Type", "application/json");

String jsonInputString = "{\"name\": \"John\", \"age\": 30}";
try(OutputStream os = connection.getOutputStream()) {
    byte[] input = jsonInputString.getBytes("utf-8");
    os.write(input, 0, input.length);
}
int responseCode = connection.getResponseCode();
```

PUT Request:

```
URL url = new URL("https://example.com/api/resource/1");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setRequestMethod("PUT");
connection.setDoOutput(true);
connection.setRequestProperty("Content-Type", "application/json");

String jsonInputString = "{\"name\": \"John Updated\", \"age\": 31}";
try(OutputStream os = connection.getOutputStream()) {
    byte[] input = jsonInputString.getBytes("utf-8");
    os.write(input, 0, input.length);
}
int responseCode = connection.getResponseCode();
```

DELETE Request

```
URL url = new URL("https://example.com/api/resource/1");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setRequestMethod("DELETE");
int responseCode = connection.getResponseCode();
```

Referências:

- Documentação oficial HttpURLConnection

2. Java 11: Uso de HttpClient

O HttpClient, introduzido no Java 11, melhora a experiência de desenvolvimento, com suporte nativo a operações síncronas e assíncronas, reduzindo significativamente a quantidade de código necessário.

GET Request (Basic Auth):

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://example.com/api/resource"))
    .header("Authorization", "Basic " +
Base64.getEncoder().encodeToString(("user:password").getBytes()))
    .GET()
    .build();

HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
```

POST Request:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://example.com/api/resource"))
```

```
    .header("Content-Type", "application/json")
    .POST(HttpRequest.BodyPublishers.ofString("{\"name\": \"John\", \"age\": 30}"))
    .build();
HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
```

PUT Request:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://example.com/api/resource/1"))
    .header("Content-Type", "application/json")
    .PUT(HttpRequest.BodyPublishers.ofString("{\"name\": \"John Updated\", \"age\": 31}"))
    .build();
HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
```

DELETE Request:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://example.com/api/resource/1"))
    .DELETE()
    .build();

HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
```

Referências:

- Download Java 11 (Oracle)
- Documentação HttpClient Java 11

3. Java 17: Uso de HttpClient com Melhorias

4.

O Java 17 mantém o HttpClient como padrão para requisições REST e traz melhorias no gerenciamento de memória, performance de garbage collection, e suporte a HTTP/2, tornando-o ideal para sistemas de alto desempenho.

GET Request (JWT Auth):

```
HttpClient client = HttpClient.newBuilder()
    .version(HttpClient.Version.HTTP_2)
    .build();

HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://example.com/api/resource"))
    .header("Authorization", "Bearer " + jwtToken)
    .GET()
    .build();

HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
```

POST Request:

```
HttpClient client = HttpClient.newBuilder().build();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://example.com/api/resource"))
    .header("Content-Type", "application/json")
    .header("Authorization", "Bearer " + jwtToken)
    .POST(HttpRequest.BodyPublishers.ofString("{\"name\": \"John\", \"age\": 30}"))
    .build();

HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
```

PUT Request:

```
HttpClient client = HttpClient.newBuilder().build();

HttpRequest request = HttpRequest.newBuilder()

    .uri(URI.create("https://example.com/api/resource/1"))

    .header("Content-Type", "application/json")

    .header("Authorization", "Bearer " + jwtToken)

    .PUT(HttpRequest.BodyPublishers.ofString("{\"name\": \"John Updated\", \"age\": 31}"))

    .build();


HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());

System.out.println(response.body());
```

DELETE Request:

```
HttpClient client = HttpClient.newBuilder().build();

HttpRequest request = HttpRequest.newBuilder()

    .uri(URI.create("https://example.com/api/resource/1"))

    .header("Authorization", "Bearer " + jwtToken)

    .DELETE()

    .build();


HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());

System.out.println(response.body());
```

Referências:

- Download Java 17 (Oracle)
- Documentação HttpClient Java 17

Boas Práticas

- Use **HttpClient** para Melhor Performance: A partir do Java 11, o HttpClient oferece suporte a requisições assíncronas e maior simplicidade de código.
- Timeouts: Configure timeouts adequados para evitar ociosidade em operações de I/O.

```
HttpClient client = HttpClient.newBuilder()
    .connectTimeout(Duration.ofSeconds(10))
    .build();
```

- Requisições Assíncronas: Utilize sendAsync() para operações não bloqueantes em aplicações de alta escala.
- HTTPS e JWT para Segurança: Sempre utilize HTTPS e tokens JWT para proteger credenciais e dados sensíveis.
- Conexões Persistentes: Configure conexões persistentes para reduzir a sobrecarga de criação de novas conexões TCP.

Dependências e Links para Download

JWT: Para manipulação de tokens JWT em Java, recomenda-se a biblioteca Java JWT (Auth0).

Maven

xml

```
<dependency>
  <groupId>com.auth0</groupId>
  <artifactId>java-jwt</artifactId>
  <version>3.18.1</version>
</dependency>
```

Gradle

```
implementation 'com.auth0:java-jwt:3.18.1'
```

Conclusão

A evolução do Java trouxe melhorias substanciais para a implementação de requisições REST, com a introdução do HttpClient a partir do Java 11, eliminando a necessidade de bibliotecas externas e tornando o código mais performático e conciso.

O Java 8, utilizando HttpURLConnection, apresenta uma solução funcional, porém verbosa e menos eficiente em termos de otimização de I/O e suporte a operações assíncronas.

No Java 11 e 17, o HttpClient introduz capacidades nativas como suporte ao HTTP/2, operações assíncronas por meio de CompletableFuture, e uma API de uso mais intuitiva e flexível, garantindo uma experiência de desenvolvimento muito mais ágil e performática, principalmente para aplicações de alta escala.

O suporte a POST, PUT, GET e DELETE com autenticação JWT, Basic Auth e autenticação via usuário e senha tornam-se significativamente mais simples e com menor propensão a falhas de código.

Com a introdução de melhorias contínuas no gerenciamento de recursos e garbage collection no Java 17, as operações de rede podem ser executadas com maior eficiência de uso de memória e tempo de resposta, consolidando essa versão como uma escolha robusta para sistemas distribuídos e de alta demanda de performance.

Além disso, boas práticas como a implementação de timeouts apropriados, o uso de HTTPS e tokens de segurança (JWT), e a utilização de conexões persistentes contribuem diretamente para a escalabilidade e segurança das aplicações REST. Portanto, adotar essas práticas não apenas aprimora a segurança, mas também reduz significativamente o tempo de latência e o overhead da aplicação.

A transição do Java 8 para o Java 11 ou 17, além de ser recomendada por motivos de suporte de longo prazo (LTS), oferece ganhos claros em termos de produtividade e eficiência.

Para desenvolvedores que lidam com serviços REST, é crucial adaptar-se a essas inovações para garantir a melhor performance e manutenibilidade de seus sistemas.

EducaCiência FastCode para a comunidade