Instrukcja laboratoryjna **3**	**Przetwarzanie / Systemy odporne na błędy**
	Temat: Obsługa sieci
	Przygotował: mgr inż. Michał Młodawski

1.Obsługa sieci w Javie

W czystej javie możliwe sposoby obsługi sieci to:

1. Wykorzystanie klas **java.net.Socket** i **java.net.ServerSocket** - jak to mechanizm gniazd, który umożliwia komunikację między procesami w sieci.
2. Wykorzystanie klas **java.net.URL** i **java.net.URLConnection** - te klasy umożliwiają dostęp do plików i zasobów sieciowych, w tym protokołów HTTP, FTP i inne.
3. Wykorzystanie klas **java.net.DatagramSocket** i **java.net.DatagramPacket** - te klasy umożliwiają wysyłanie i odbieranie pakietów danych przy użyciu protokołu UDP (User Datagram Protocol).
4. Wykorzystanie bibliotek zewnętrznych, takich jak Apache HttpClient, OkHttp, czy Retrofit - to popularne biblioteki w Javie, które ułatwiają komunikację z serwerami sieciowymi i zapewniają wiele dodatkowych funkcjonalności, takich jak obsługa uwierzytelniania, wątkowość, obsługa błędów i wiele innych.

2.Obsługa gniazd

Załóżmy, że naszym zadaniem jest napisać aplikację w formie czatu lub grę. Możemy oczywiście zrobić aplikację lokalną działającą na jednym komputerze, ale jeśli chcemy uruchomić swój program na różnych komputerach i wspólnie komunikować się bądź grać? W tym celu musimy stworzyć interakcję między wieloma użytkownikami i można to zrobić stosując obsługę sieci.

Jednym z podstawowych sposób jest użycia gniazda. Jest to interfejs do wysyłania lub odbierania danych między różnymi procesorami, programami itp. w kolejności dwukierunkowej. Gniazdo zapisujemy jako kombinacje naszego adresu komputera w sieci i portu z zakresu od 0 do 65535, choć zalecane jest stosowanie adresów od 1024. Przykładowy adres gniazda może być następujący: **127.0.0.1:2136**.

W celu rozpoczęcia komunikacji, **klient** żąda połączenia z **serwerem** przy użyciu adresu i portu, na którym serwer nasłuchuje przychodzących żądań. Serwer po prostu czeka na nowe żądanie i akceptuje je lub nie. Jeśli połączenie zostało zaakceptowane, serwer tworzy gniazdo do interakcji z klientem. Zarówno klient, jak i serwer mogą przesyłać do siebie dane za pomocą swoich gniazd. Z reguły jeden serwer współdziała z wieloma klientami.

2.1.Gniazda w Javie

Java udostępnia dwie główne klasy do interakcji między programami korzystającymi z gniazd w pakiecie java.net.* :

- **Socket** - reprezentuje jedną stronę połączenia dwukierunkowego, używanego przez klientów i serwery;
- **ServerSocket** - reprezentuje specjalny typ gniazd, które nasłuchują i akceptują połączenia do klientów, używane tylko przez serwery.

Aby rozpocząć interakcję z serwerem potrzebujemy przynajmniej jednego klienta. Najpierw powinniśmy stworzyć gniazdo, określając ścieżkę do serwera.

```
Socket socket = new Socket("127.0.0.1", 2136);
```

Jako przykładu używamy adresu localhost bądź "127.0.0.1". W prawdziwym przypadku nasz serwer jest hostowany na innym komputerze. Tak więc dobrą praktyką jest pobieranie adresu i portu z zewnętrznej konfiguracji lub argumentów wiersza poleceń. Oczywiście należy założyć, że nasza aplikacja będzie miała więcej niż jednego klienta, wtedy z pomocą przyjdzie nam wielowątkowość. Dlatego nasz program będzie oparty właśnie o wątkach.

Do wysyłania i odbierania danych na serwer potrzebujemy strumieni wejściowych i wyjściowych:

```
DataInputStream input = new DataInputStream(socket.getInputStream());
DataOutputStream output  = new DataOutputStream(socket.getOutputStream());
```

- Wywołanie input.readUTF()odbiera komunikat tekstowy od klienta;
- Wywołanie output.writeUTF(message)wysyła do klienta komunikat tekstowy.

Poniżej przedstawiono program źródłowy klienta:

```
package com.example.demo.tcp;

import java.io.*;
import java.net.Socket;

public class ClientTCP extends Thread {
    public void run() {
        // Pobierz adres hosta i numer portu z argumentów
        String host = "127.0.0.1";
        int portNumber = Integer.parseInt("2136");

        // Łączenie z serwerem
        try (Socket socket = new Socket(host, portNumber)) {
            // Tworzenie strumieni wejścia i wyjścia
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
true);
            BufferedReader stdIn = new BufferedReader(new
InputStreamReader(System.in));

            String userInput;
            System.out.println("Connected to the server. Type your message
and press Enter to send. Type 'quit' to exit.");
            while ((userInput = stdIn.readLine()) != null) {
                // Wysyłanie wiadomości do serwera
                out.println(userInput);

                // Odbieranie odpowiedzi od serwera
                String serverResponse = in.readLine();
                System.out.println("Server: " + serverResponse);

                // Zakończ, jeśli użytkownik wprowadził "quit"
                if ("quit".equalsIgnoreCase(userInput)) {
                    break;
                }
            }
            // Zamykanie strumieni
            System.out.println("Closing connection with the server");
            stdIn.close();
            in.close();
            out.close();
        } catch (IOException e) {
            System.out.println("Exception caught when trying to connect to "
+ host + " on port " + portNumber);
            System.out.println(e.getMessage());
        }
    }
}
```

Zasada działania serwera jest podobna do tej z klienta, następuje utworzenie połączenia i jego akceptacja, następnie serwera oczekuje dostarczenia danych i zwraca je tworząc swego

rodzaju „echo”. Poniżej przedstawiono program źródłowy serwera:

```
package com.example.demo.tcp;

import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

public class ServerTCP extends Thread {
    public void run() {
        int portNumber = Integer.parseInt("2136");

        // Utwórz serwer z wykorzystaniem ServerSocket
        try (ServerSocket serverSocket = new ServerSocket(portNumber)) {
            System.out.println("Server is listening on port " + portNumber);

            // Nasłuchuj połączeń od klientów
            while (true) {
                // Akceptuj połączenie od klienta
                Socket clientSocket = serverSocket.accept();
                System.out.println("Client connected");

                // Tworzenie strumieni wejścia i wyjścia
                BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
                PrintWriter out = new
PrintWriter(clientSocket.getOutputStream(), true);

                // Odbierz wiadomość od klienta
                String inputLine;
                while ((inputLine = in.readLine()) != null) {
                    System.out.println("Received from client: " +
inputLine);

                    // Odpowiedz klientowi
                    out.println("Server received: " + inputLine);

                    // Zamknij połączenie, jeśli klient wysłał "quit"
                    if ("quit".equalsIgnoreCase(inputLine)) {
                        break;
                    }
                }

                // Zamknij połączenie z klientem
                System.out.println("Closing connection with client");
                in.close();
                out.close();
                clientSocket.close();
            }
        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen on
port " + portNumber);
            System.out.println(e.getMessage());
        }
    }
}
```

Poniżej przedstawiono implementacje powyższego rozwiązania:

```
package com.example.demo.tcp;

import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SocketApplication implements ApplicationRunner {

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

    @Override
    public void run(ApplicationArguments args) throws Exception {

        ServerTCP server = new ServerTCP();
        server.start();
        Thread.sleep(10);

        ClientTCP client = new ClientTCP();
        client.start();
        Thread.sleep(10);
    }
}
```

Serwer będzie nasłuchiwać na podanym porcie (w tym przypadku 12345) i odbierać wiadomości od klientów. Aby przetestować serwer, możesz użyć klienta napisanego w Javie lub użyć narzędzi takich jak **telnet** lub **nc** (netcat).

Poniżej przedstawiono implementacje powyższego rozwiązania:

```
@SpringBootApplication
public class SocketApplication implements ApplicationRunner {

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

    @Override
    public void run(ApplicationArguments args) throws Exception {

        ServerTCP server = new ServerTCP();
        server.start();
        Thread.sleep(10);

        ClientTCP client = new ClientTCP();
        client.start();
        Thread.sleep(10);
    }
}
```

Klient będzie próbował nawiązać połączenie z serwerem na podanym adresie hosta (w tym przypadku "127.0.0.1") i numerze portu (2136). Gdy połączenie zostanie nawiązane, klient będzie mógł wysyłać wiadomości do serwera. Aby zakończyć połączenie, wprowadź "quit" i naciśnij Enter.

Upewnij się, że serwer jest uruchomiony przed uruchomieniem klienta.

3.Obsługa protokołu UDP

Protokół UDP (User Datagram Protocol) to protokół warstwy transportowej, który umożliwia przesyłanie pakietów danych w sieciach komputerowych. W odróżnieniu od protokołu TCP, który zapewnia niezawodną transmisję danych, protokół UDP jest protokołem bezpołączeniowym, co oznacza, że nie zapewnia on bezpośrednio żadnych gwarancji dotyczących dostarczenia pakietów.

Komunikacja pakietów z wykorzystaniem UDP przebiega w następujący sposób:

1. Nadawca tworzy datagram (pakiet danych) i przekazuje go do gniazda UDP.
2. Gniazdo UDP dodaje nagłówek datagramu, który zawiera informacje o adresie źródłowym i docelowym oraz numerze portu.
3. Datagram jest przesyłany przez sieć w postaci niezależnych pakietów.

4. Odbiorca otrzymuje pakiety i przetwarza je, ignorując duplikaty, niekompletne pakiety lub te, które zawierają błędy.
5. Odbiorca potwierdza otrzymanie pakietu lub wysyła zapytanie o ponowne przesłanie, jeśli pakiet nie został odebrany poprawnie.

Komunikacja pakietów z wykorzystaniem UDP jest szybka i nie wymaga dużych nakładów obliczeniowych. Wadą jest brak mechanizmów kontroli błędów, co może prowadzić do utraty pakietów lub duplikatów. Protokół UDP jest często stosowany w aplikacjach, w których szybkość przesyłu danych jest ważniejsza niż niezawodność, takich jak transmisja strumieniowa, gry online czy aplikacje VoIP (Voice over Internet Protocol).

Poniżej przedstawiam prosty przykład kodu klienta UDP napisanego w języku Java z wykorzystaniem klasy DatagramSocket i DatagramPacket:

```
package com.example.demo.udp;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.*;

public class ClientUDP extends Thread {
    public void run() {
        String host = "127.0.0.1";
        int portNumber = Integer.parseInt("2136");

        // Utwórz klienta z wykorzystaniem DatagramSocket
        try (DatagramSocket clientSocket = new DatagramSocket()) {
            InetAddress serverAddress = InetAddress.getByName(host);
            BufferedReader stdIn = new BufferedReader(new
InputStreamReader(System.in));

            String userInput;
            System.out.println("Connected to the server. Type your message
and press Enter to send. Type 'quit' to exit.");

            while ((userInput = stdIn.readLine()) != null) {
                // Wysyłanie wiadomości do serwera
                byte[] sendData = userInput.getBytes();
                DatagramPacket sendPacket = new DatagramPacket(sendData,
sendData.length, serverAddress, portNumber);
                clientSocket.send(sendPacket);

                // Odbieranie odpowiedzi od serwera
                byte[] receiveData = new byte[1024];
                DatagramPacket receivePacket = new
DatagramPacket(receiveData, receiveData.length);
                clientSocket.receive(receivePacket);

                // Wyświetl odpowiedź serwera
                String serverResponse = new String(receivePacket.getData(),
0, receivePacket.getLength());
```

```
                System.out.println("Server: " + serverResponse);

                // Zakończ, jeśli użytkownik wprowadził "quit"
                if ("quit".equalsIgnoreCase(userInput)) {
                    break;
                }
            }

            // Zamykanie strumienia
            System.out.println("Closing connection with the server");
            stdIn.close();
        } catch (SocketException e) {
            System.out.println("SocketException occurred");
            System.out.println(e.getMessage());
        } catch (UnknownHostException e) {
            System.out.println("UnknownHostException occurred");
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println("IOException occurred");
            System.out.println(e.getMessage());
        }
    }
}
```

Kod klienta tworzy wiadomość, adres i port serwera oraz konwertuje wiadomość na tablicę bajtów. Następnie tworzony jest pakiet z tymi danymi oraz utworzone zostaje gniazdo. Na końcu wysyłany jest pakiet na serwer i gniazdo jest zamykane.

Poniżej przedstawiam prosty przykład kodu serwera UDP napisanego w języku Java z wykorzystaniem klasy DatagramSocket i DatagramPacket:

```
package com.example.demo.udp;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;

public class ServerUDP extends Thread {
    public void run() {

        // Pobierz numer portu z argumentów
        int portNumber = Integer.parseInt("2136");

        // Utwórz serwer z wykorzystaniem DatagramSocket
        try (DatagramSocket serverSocket = new DatagramSocket(portNumber)) {
            System.out.println("Server is listening on port " + portNumber);

            // Przygotuj bufor do odbierania danych
            byte[] receiveBuffer = new byte[1024];

            // Nasłuchuj połączeń od klientów
            while (true) {
                // Odbierz datagram od klienta
```

```
                DatagramPacket receivePacket = new
DatagramPacket(receiveBuffer, receiveBuffer.length);
                serverSocket.receive(receivePacket);

                // Konwersja otrzymanych danych na String
                String receivedMessage = new String(receivePacket.getData(),
0, receivePacket.getLength());
                System.out.println("Received from client: " +
receivedMessage);

                // Przygotuj odpowiedź dla klienta
                byte[] sendBuffer = ("Server received: " +
receivedMessage).getBytes();

                // Wyślij odpowiedź do klienta
                DatagramPacket sendPacket = new DatagramPacket(sendBuffer,
sendBuffer.length, receivePacket.getAddress(), receivePacket.getPort());
                serverSocket.send(sendPacket);
            }
        } catch (SocketException e) {
            System.out.println("Exception caught when trying to listen on
port " + portNumber);
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println("IOException occurred");
            System.out.println(e.getMessage());
        }
    }
}
```

Kod serwera tworzy port na którym będzie działał serwer. Następnie tworzone jest gniazdo i pakiet, które będą służyły do odbierania danych. W pętli while, która działa w nieskończoność, serwer odbiera pakiety, wyświetla otrzymaną wiadomość i wysyła odpowiedź z powrotem do klienta. Gdy odpowiedź zostanie wysłana, pętla rozpoczyna odbieranie kolejnego pakietu.

UDP (User Datagram Protocol) to protokół bezpołączeniowy, który nie gwarantuje dostarczenia pakietu, a także nie zapewnia kolejności ich dostarczenia. Z tego powodu, mogą pojawić się pewne problemy przy używaniu UDP w porównaniu do TCP.

Poniżej opisane są niektóre z typowych problemów związanych z wykorzystaniem UDP wraz z przykładowym kodem w języku Java:

1. Brak potwierdzenia otrzymania pakietu - UDP nie zapewnia potwierdzenia otrzymania pakietu, więc wysyłający nie będzie wiedział, czy pakiet dotarł do odbiorcy. Może to spowodować utratę danych lub powtarzanie transmisji.

Przykładowy kod Java, który wysyła dane UDP i nie oczekuje na potwierdzenie otrzymania:

```
DatagramSocket socket = new DatagramSocket();
byte[] data = "Hello World".getBytes();
DatagramPacket packet = new DatagramPacket(data, data.length,
InetAddress.getByName("localhost"), 1234);
socket.send(packet);
```

Brak zapewnienia kolejności - Ponieważ UDP nie gwarantuje kolejności dostarczenia pakietów, odbiorca może otrzymać pakiety w innej kolejności, niż zostały wysłane. Może to spowodować błędy w przetwarzaniu danych.

Przykładowy kod Java, który odbiera dane UDP bez uwzględnienia kolejności:

```
DatagramSocket socket = new DatagramSocket(1234);
byte[] buffer = new byte[1024];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
socket.receive(packet);
String data = new String(packet.getData(), 0, packet.getLength());
```

Limit rozmiaru pakietu - UDP posiada limit rozmiaru pakietu (ok. 65 507 bajtów), więc w przypadku przesyłania większych danych, konieczne może być podział danych na mniejsze pakiety. Przykładowy kod Java, który wysyła większe dane UDP w mniejszych pakietach:

```
DatagramSocket socket = new DatagramSocket();
byte[] data = "Hello World".getBytes();
int packetSize = 1024;
int numPackets = (int) Math.ceil((double) data.length / packetSize);
for (int i = 0; i < numPackets; i++) {
    int start = i * packetSize;
    int end = Math.min((i + 1) * packetSize, data.length);
    byte[] packetData = Arrays.copyOfRange(data, start, end);
    DatagramPacket packet = new DatagramPacket(packetData,
packetData.length, InetAddress.getByName("localhost"), 1234);
    socket.send(packet);
}
```

Serwer UDP będzie nasłuchiwać na podanym porcie (w tym przypadku 12345) i odbierać datagramy od klientów. Aby przetestować serwer, możesz użyć klienta napisanego w Javie lub użyć narzędzi takich jak **nc** (netcat) z opcją **-u**.

Poniżej przedstawiono implementacje powyższego rozwiązania:

```
@SpringBootApplication
public class SocketApplication implements ApplicationRunner {

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

    @Override
    public void run(ApplicationArguments args) throws Exception {

        ServerUDP server = new ServerUDP();
        server.start();
        Thread.sleep(10);

        ClientUDP client = new ClientUDP();
        client.start();
        Thread.sleep(10);
    }
}
```

Klient będzie próbował nawiązać połączenie z serwerem na podanym adresie hosta (w tym przypadku "127.0.0.1") i numerze portu (2136). Gdy połączenie zostanie nawiązane, klient będzie mógł wysyłać wiadomości do serwera. Aby zakończyć połączenie, wprowadź "quit" i naciśnij Enter.

Upewnij się, że serwer jest uruchomiony przed uruchomieniem klienta.

Podsumowując, UDP jest protokołem bezpołączeniowym, co oznacza, że nie oferuje pewnych funkcjonalności, takich jak potwierdzenie otrzymania pakietu czy zapewnienie kolejności pakietów. W związku z tym, konieczne może być dodatkowe programowanie, aby zabezpieczyć dane i zapewnić poprawne przetwarzanie ich przez aplikację.

4.Obsługa zapytań HTTP z wykorzystaniem HttpURLConnection

W Javie wykonuje się zapytania HTTP w celu komunikacji z serwerami WWW lub innymi aplikacjami sieciowymi, w celu pobierania lub przesyłania danych.

Zapytania HTTP umożliwiają tworzenie aplikacji sieciowych, które mogą pobierać i wysyłać dane za pomocą standardowego protokołu sieciowego HTTP. Wiele aplikacji sieciowych opiera się na architekturze REST, która wykorzystuje zapytania HTTP do udostępniania zasobów i usług w sposób prosty i skalowalny.

Przykłady zastosowań zapytań HTTP w Javie to:

- Pobieranie danych z serwera API za pomocą żądań HTTP GET.
- Przesyłanie danych do serwera API za pomocą żądań HTTP POST, PUT lub PATCH.
- Wykonywanie operacji usuwania danych z serwera API za pomocą żądań HTTP DELETE.
- Pobieranie danych z serwerów WWW za pomocą protokołu HTTP.
- Logowanie do systemów autoryzacji i uwierzytelniania za pomocą żądań HTTP POST.

Do wykonywania zapytań HTTP w Javie można wykorzystać wiele narzędzi i bibliotek, takich jak HttpURLConnection, Apache HttpClient, OkHttp, Retrofit i wiele innych.

Obsługa zapytań HTTP to proces, w którym aplikacja sieciowa (takie jak serwer WWW) odbiera żądanie HTTP od klienta, przetwarza go i zwraca odpowiedź HTTP do klienta.

W obsłudze zapytań HTTP ważną rolę odgrywają nagłówki HTTP, które zawierają informacje o żądaniu lub odpowiedzi, takie jak typ zawartości, długość zawartości, kodowanie i wiele innych. Na przykład nagłówek "Content-Type" mówi, w jakim formacie jest zawartość żądania lub odpowiedzi, a nagłówek "Content-Length" określa długość zawartości.

Podczas obsługi zapytań HTTP aplikacja sieciowa może wykonywać różne zadania, takie jak pobieranie lub przesyłanie danych, uwierzytelnianie użytkowników, wykonywanie zapytań do bazy danych, przetwarzanie i walidacja danych, wykonywanie logiki biznesowej i wiele innych.

Współczesne frameworki webowe, takie jak Spring, zapewniają wiele narzędzi i bibliotek, które ułatwiają obsługę zapytań HTTP i pozwalają programistom szybciej tworzyć aplikacje sieciowe.

4.1.Wykonywanie zapytania pobierającego zasób

W tym przykładzie tworzony jest obiekt URL, zawierający adres URL API oraz parametry, które zostaną przesłane w zapytaniu HTTP GET. Następnie tworzony jest obiekt HttpURLConnection, który umożliwia nawiązanie połączenia i wysłanie żądania GET.

Wartość parametrów jest dodawana do adresu URL za pomocą operatora **?**, a każdy parametr oddzielony jest od innych znakiem **&**.

Odpowiedź z serwera jest pobierana za pomocą BufferedReader, a następnie zapisywana do StringBuffer. Na koniec wyświetlana jest odpowiedź z serwera na standardowym wyjściu.

```
import java.net.*;
import java.io.*;

public class HttpGetWithParamsExample {
    public static void main(String[] args) {
        try {
            String url = "https://example.com/api/users";
            String params = "name=John&age=30";

            URL apiUrl = new URL(url + "?" + params);
            HttpURLConnection con = (HttpURLConnection)
apiUrl.openConnection();
            con.setRequestMethod("GET");

            BufferedReader in = new BufferedReader(new
InputStreamReader(con.getInputStream()));
            String inputLine;
            StringBuffer response = new StringBuffer();

            while ((inputLine = in.readLine()) != null) {
                response.append(inputLine);
            }
            in.close();

            System.out.println(response.toString());
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

4.2.Wykonywanie zapytania dodający zasób

W tym przykładzie tworzony jest obiekt URL zawierający adres URL API. Następnie tworzony jest obiekt HttpURLConnection, który umożliwia nawiązanie połączenia i wysłanie żądania POST.

```
import java.net.*;
import java.io.*;

public class HttpPostWithJsonBodyExample {
    public static void main(String[] args) {
        try {
            String url = "https://example.com/api/users";
            URL apiUrl = new URL(url);

            HttpURLConnection con = (HttpURLConnection)
apiUrl.openConnection();
            con.setRequestMethod("POST");
            con.setRequestProperty("Content-Type", "application/json; utf-
8");
            con.setRequestProperty("Accept", "application/json");
            con.setDoOutput(true);

            String jsonInputString = "{\"name\": \"John\", \"age\": 30}";

            try(OutputStream os = con.getOutputStream()) {
                byte[] input = jsonInputString.getBytes("utf-8");
                os.write(input, 0, input.length);
            }

            BufferedReader in = new BufferedReader(new
InputStreamReader(con.getInputStream()));
            String inputLine;
            StringBuffer response = new StringBuffer();

            while ((inputLine = in.readLine()) != null) {
                response.append(inputLine);
            }
            in.close();

            System.out.println(response.toString());
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Do nagłówka żądania dodawany jest nagłówek Content-Type, wskazujący, że wysyłane dane są w formacie JSON, oraz nagłówek Accept, który określa, że oczekiwana odpowiedź również powinna być w formacie JSON.

Ciało żądania JSON jest zapisywane jako ciąg znaków w zmiennej **jsonInputString**, a następnie przekazywane do strumienia wyjściowego za pomocą metody **setDoOutput**.

Odpowiedź z serwera jest pobierana za pomocą BufferedReader, a następnie zapisywana do StringBuffer. Na końcu wyświetlana jest odpowiedź z serwera na standardowym wyjściu.

Uwaga: Warto zwrócić uwagę, że w tym przykładzie operacje na strumieniach wyjściowych i wejściowych powinny być wykonane w bloku try-with-resources, aby zapewnić ich poprawne zamykanie po zakończeniu operacji.

Content-Type jest to nagłówek HTTP, który określa format (typ MIME) danych, które są przesyłane w ciele żądania lub odpowiedzi HTTP. Content-Type jest ważnym elementem w procesie przetwarzania żądań i odpowiedzi HTTP, ponieważ informuje aplikację, jakie typy danych przetwarza lub oczekuje.

Istnieje wiele różnych wartości Content-Type, w zależności od rodzaju przesyłanych danych. Niektóre z najczęściej stosowanych wartości to:

- application/json - do przesyłania danych w formacie JSON (JavaScript Object Notation)
- application/xml - do przesyłania danych w formacie XML (eXtensible Markup Language)
- text/plain - do przesyłania tekstu w formacie plain text
- multipart/form-data - do przesyłania danych formularza, w tym plików

Stosowanie właściwego Content-Type jest ważne, ponieważ pomaga serwerowi i klientowi zrozumieć, jakie typy danych są przetwarzane, co umożliwia poprawne przetwarzanie danych w żądaniach i odpowiedziach HTTP.

Na przykład, jeśli serwer oczekuje danych w formacie JSON, ale otrzyma żądanie z innym typem MIME, może zwrócić błąd lub nieprawidłowe dane. W przypadku, gdy klient oczekuje odpowiedzi w formacie JSON, ale serwer zwraca dane w innym formacie, klient może nie być w stanie poprawnie przetworzyć odpowiedzi.

Warto zwrócić uwagę, że wartość Content-Type może również zawierać dodatkowe parametry, takie jak kodowanie znaków, które pomagają zdefiniować, jak powinny być przetwarzane dane. Na przykład, wartość Content-Type application/json; charset=utf-8 oznacza, że dane w ciele żądania lub odpowiedzi są w formacie JSON i powinny być kodowane za pomocą UTF-8.

Nagłówek Accept w protokole HTTP jest używany do informowania serwera, jakie typy MIME danych może obsłużyć klient, czyli co klient może akceptować w odpowiedzi. Serwer może odpowiedzieć wybierając jedną z oferowanych wartości, która będzie pasować do wymagań klienta.

Istnieją różne wartości nagłówka Accept, które klient może wysyłać w zależności od potrzeb:

1. */* - oznacza, że klient akceptuje dowolny typ MIME.
2. type/subtype - oznacza, że klient akceptuje określony typ MIME. Na przykład, application/json lub image/png.

3. type/* - oznacza, że klient akceptuje każdy podtyp danego typu MIME. Na przykład, text/* akceptuje text/plain, text/html, itd.
4. type/subtype;q=wartosc - oznacza, że klient akceptuje określony typ MIME i przypisuje mu wartość "q". Wartość "q" określa, jak bardzo dany typ MIME jest preferowany przez klienta, gdzie wartość q=1 oznacza najwyższą preferencję, a q=0 oznacza, że dany typ MIME nie jest akceptowany przez klienta.

Przykłady nagłówka Accept:

- Accept: application/json - oznacza, że klient akceptuje tylko dane w formacie JSON.
- Accept: text/html, application/xhtml+xml, application/xml;q=0.9, */*;q=0.8 - oznacza, że klient akceptuje dane w formacie HTML, XHTML i XML. Wartość "q" jest przypisana tylko dla typu MIME application/xml, co oznacza, że jest ona mniej preferowana niż inne typy MIME.
- Accept: text/*;q=0.3, application/json;q=0.7, application/*;q=0.5, */*;q=0.1 - oznacza, że klient preferuje dane w formacie JSON, a preferencja innych typów MIME jest określana przez wartość "q". Wartość q=0.1 oznacza, że klient najmniej preferuje dane w dowolnym formacie MIME.

4.3.Wykonywanie zapytania aktualizujący zasób

W tym przykładzie tworzymy obiekt HttpURLConnection i ustawiamy metodę żądania jako PUT. Włączamy wysyłanie danych i tworzymy strumień wyjściowy, aby wysłać request body jako ciąg bajtów. W końcu pobieramy kod odpowiedzi z serwera.

Ważne: Przy wysyłaniu danych typu JSON należy ustawić odpowiedni nagłówek Content-Type na "application/json".

```
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class PutRequestExample {

    public static void main(String[] args) throws Exception {
        String url = "http://example.com/api/resource";
        String requestBody = "This is the request body";

        URL obj = new URL(url);
        HttpURLConnection con = (HttpURLConnection) obj.openConnection();

        // Ustawiamy metodę żądania jako PUT
        con.setRequestMethod("PUT");

        // Włączamy wysyłanie danych
        con.setDoOutput(true);

        // Tworzymy strumień wyjściowy do wysyłania danych
        OutputStream os = con.getOutputStream();
```

```
        // Wysyłamy request body jako ciąg bajtów
        os.write(requestBody.getBytes());
        os.flush();
        os.close();

        // Pobieramy kod odpowiedzi
        int responseCode = con.getResponseCode();
        System.out.println("Response Code : " + responseCode);
    }
}
```

4.4.Wykonywanie zapytania usuwający zasób

W tym przykładzie tworzymy obiekt HttpURLConnection i ustawiamy metodę żądania jako DELETE. Następnie pobieramy kod odpowiedzi z serwera.

Ważne: Przed wysłaniem zapytania DELETE należy upewnić się, że zasób, który chcemy usunąć, istnieje i możemy go usunąć.

```
import java.net.HttpURLConnection;
import java.net.URL;

public class DeleteRequestExample {

    public static void main(String[] args) throws Exception {
        String url = "http://example.com/api/resource/123";

        URL obj = new URL(url);
        HttpURLConnection con = (HttpURLConnection) obj.openConnection();

        // Ustawiamy metodę żądania jako DELETE
        con.setRequestMethod("DELETE");

        // Pobieramy kod odpowiedzi
        int responseCode = con.getResponseCode();
        System.out.println("Response Code : " + responseCode);
    }
}
```

5.Obsługa zapytań HTTP z wykorzystaniem OkHttp

OkHttp to biblioteka HTTP dla języka Java, stworzona przez firmę Square, która ułatwia tworzenie i wykonywanie zapytań HTTP w aplikacjach. OkHttp zapewnia łatwą obsługę protokołów HTTP/1.1 i HTTP/2, automatyczną obsługę ciasteczek (cookies), cache'owanie odpowiedzi i wiele innych funkcjonalności.

Główne cechy biblioteki OkHttp to:

- Obsługa protokołów HTTP/1.1 i HTTP/2.
- Automatyczna obsługa ciasteczek (cookies) i przekierowań.
- Cache'owanie odpowiedzi, co może poprawić wydajność i zmniejszyć liczbę zapytań sieciowych.

- Szyfrowanie SSL/TLS zgodnie z najnowszymi standardami.
- Obsługa protokołu WebSocket.
- Dostępna jako biblioteka Maven.
- Wsparcie dla Androida.

OkHttp jest często wybierane przez deweloperów ze względu na swoją łatwość użycia i wydajność. Dodatkowo, biblioteka ta oferuje wiele zaawansowanych funkcjonalności, takich jak obsługa interceptorów, logowanie zapytań i odpowiedzi, obsługa zapytań asynchronicznych i wiele innych.

Aby dodać bibliotekę OkHttp do projektu z wykorzystaniem Maven, należy dodać wpis do pliku **pom.xml**. Oto kroki, które należy wykonać:

1. Otwórz plik **pom.xml** w swoim projekcie.
2. Znajdź sekcję **<dependencies>** i dodaj do niej wpis z informacjami o bibliotece OkHttp:

```
<dependency>
    <groupId>com.squareup.okhttp3</groupId>
    <artifactId>okhttp</artifactId>
    <version>4.9.1</version>
</dependency>
```

3. Zapisz plik **pom.xml**.

Po wykonaniu tych kroków, biblioteka OkHttp zostanie dodana do projektu. Teraz można używać jej w kodzie źródłowym aplikacji.

5.1.Wykonywanie zapytania pobierający zasób

W poniższym przykładzie dodatkowo została użyta metoda **addHeader()** do ustawienia nagłówka **Accept** na wartość **application/json**, co oznacza akceptację odpowiedzi w formacie JSON. Otrzymaną odpowiedź można odczytać za pomocą metody **string()** obiektu **ResponseBody**.

```
import okhttp3.HttpUrl;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;
import okhttp3.MediaType;
import java.io.IOException;

public class OkHttpGetJsonExample {
    public static void main(String[] args) throws IOException {
        OkHttpClient client = new OkHttpClient();

        // Utworzenie obiektu reprezentującego adres URL z parametrami
        HttpUrl.Builder urlBuilder =
```

```
HttpUrl.parse("https://example.com/api/data").newBuilder();
        urlBuilder.addQueryParameter("param1", "value1");
        urlBuilder.addQueryParameter("param2", "value2");
        String url = urlBuilder.build().toString();

        // Utworzenie obiektu reprezentującego żądanie GET z akceptacją typu
JSON
        Request request = new Request.Builder()
                .url(url)
                .addHeader("Accept", "application/json")
                .build();

        // Wykonanie żądania i otrzymanie odpowiedzi
        Response response = client.newCall(request).execute();

        // Wypisanie treści odpowiedzi
        String responseBody = response.body().string();
        System.out.println(responseBody);
    }
}
```

5.2.Wykonywanie zapytania dodający zasób

W poniższym przykładzie użyto metody **post()** do ustawienia metody żądania na **POST** oraz utworzenia obiektu **RequestBody** z treścią żądania w formacie JSON. Utworzony obiekt **RequestBody** został następnie przekazany do metody **post()** obiektu **Request.Builder**. Otrzymaną odpowiedź można odczytać za pomocą metody **string()** obiektu **ResponseBody**.

```
import okhttp3.MediaType;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.RequestBody;
import okhttp3.Response;

import java.io.IOException;

public class OkHttpPostJsonExample {
    public static void main(String[] args) throws IOException {
        OkHttpClient client = new OkHttpClient();

        // Utworzenie treści żądania w formacie JSON
        MediaType mediaType = MediaType.parse("application/json");
        String requestBody = "{\"username\": \"testuser\", \"password\":
\"testpass\"}";

        // Utworzenie obiektu reprezentującego żądanie POST z treścią w
formacie JSON
        Request request = new Request.Builder()
                .url("https://example.com/api/login")
                .post(RequestBody.create(mediaType, requestBody))
                .build();

        // Wykonanie żądania i otrzymanie odpowiedzi
        Response response = client.newCall(request).execute();
```

```
        // Wypisanie treści odpowiedzi
        String responseBody = response.body().string();
        System.out.println(responseBody);
    }
}
```

5.3.Wykonywanie zapytania aktualizujący zasób

W poniższym przykładzie użyto metody **put()** do ustawienia metody żądania na **PUT** oraz utworzenia obiektu **RequestBody** z nową treścią. Utworzony obiekt **RequestBody** został następnie przekazany do metody **put()** obiektu **Request.Builder**. Otrzymaną odpowiedź można odczytać za pomocą metody **string()** obiektu **ResponseBody**.

```
import okhttp3.MediaType;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.RequestBody;
import okhttp3.Response;

import java.io.IOException;

public class OkHttpPutExample {
    public static void main(String[] args) throws IOException {
        OkHttpClient client = new OkHttpClient();

        // Utworzenie treści żądania w formacie tekstowym
        MediaType mediaType = MediaType.parse("text/plain");
        String requestBody = "New content";

        // Utworzenie obiektu reprezentującego żądanie PUT z nową treścią
        Request request = new Request.Builder()
                .url("https://example.com/api/content/123")
                .put(RequestBody.create(mediaType, requestBody))
                .build();

        // Wykonanie żądania i otrzymanie odpowiedzi
        Response response = client.newCall(request).execute();

        // Wypisanie treści odpowiedzi
        String responseBody = response.body().string();
        System.out.println(responseBody);
    }
}
```

5.4.Wykonywanie zapytania usuwający zasób

W poniższym przykładzie użyto metody **delete()** do ustawienia metody żądania na **DELETE**. Otrzymaną odpowiedź można odczytać za pomocą metody **code()** obiektu **Response**, która zwróci status odpowiedzi jako liczbę całkowitą.

```
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;

import java.io.IOException;

public class OkHttpDeleteExample {
    public static void main(String[] args) throws IOException {
        OkHttpClient client = new OkHttpClient();

        // Utworzenie obiektu reprezentującego żądanie DELETE
        Request request = new Request.Builder()
                .url("https://example.com/api/content/123")
                .delete()
                .build();

        // Wykonanie żądania i otrzymanie odpowiedzi
        Response response = client.newCall(request).execute();

        // Wypisanie statusu odpowiedzi
        System.out.println(response.code());
    }
}
```

5.5.Wysyłanie zapytań HTTP z wykorzystaniem RestTemplate

RestTemplate to klient HTTP dostarczany przez Spring Framework, który upraszcza proces wysyłania żądań HTTP. Pozwala na łatwe wykonywanie operacji REST, takich jak GET, POST, PUT i DELETE.

Główne cechy RestTemplate:

- Wsparcie dla różnych metod HTTP
- Automatyczna serializacja/deserializacja obiektów Java do/z JSON lub XML
- Obsługa nagłówków HTTP
- Możliwość konfiguracji timeout'ów, interceptorów itp.

Aby użyć RestTemplate w projekcie Spring Boot, nie trzeba dodawać żadnych dodatkowych zależności, ponieważ jest on częścią spring-boot-starter-web.

W każdym z poniższych przykładów, RestTemplate automatycznie serializuje/deserializuje obiekty do/z formatu JSON (domyślnie). Możesz też łatwo dostosować format odpowiedzi, używając odpowiednich klas HttpMessageConverter.

Wykonywanie zapytania GET

```
import org.springframework.web.client.RestTemplate;

public class test {
    public static void main(String[] args) {

        RestTemplate restTemplate = new RestTemplate();
        String url = "https://api.example.com/resource";

        String result = restTemplate.getForObject(url, String.class);
        System.out.println(result);
    }

}
```

Wykonywanie zapytania POST

```
public class test {
    public static void main(String[] args) {

        RestTemplate restTemplate = new RestTemplate();
        String url = "https://api.example.com/resource";

        String requestBody = "Hello, World!";
        String result = restTemplate.postForObject(url, requestBody,
String.class);
        System.out.println(result);

    }

}
```

Wykonywanie zapytania PUT

```
import org.springframework.web.client.RestTemplate;

public class test {
    public static void main(String[] args) {

        RestTemplate restTemplate = new RestTemplate();
        String url = "https://api.example.com/resource/1";

        String requestBody = "Updated content";
        restTemplate.put(url, requestBody);

    }

}
```

Wykonywanie zapytania DELETE

```
public class test {
    public static void main(String[] args) {

        RestTemplate restTemplate = new RestTemplate();
        String url = "https://api.example.com/resource/1";
        restTemplate.delete(url);

    }

}
```

6.Testowanie zapytań HTTP z wykorzystaniem aplikacji Postman

1. Uruchomienie aplikacji Postman.
2. Wybierz rodzaj żądania HTTP (np. GET, POST, PUT, DELETE) z rozwijanego menu w lewym górnym rogu ekranu.
3. Wpisz adres URL API lub serwera docelowego w pasek adresu w górnym środkowym panelu ekranu.
4. Dodaj parametry żądania HTTP, jeśli są potrzebne. Możesz to zrobić na dwa sposoby:

- Dodaj parametry do adresu URL w pasku adresu.
- Dodaj parametry w zakładce "Params" w dolnym środkowym panelu ekranu.

5. Dodaj nagłówki HTTP, jeśli są potrzebne. Możesz to zrobić w zakładce "Headers" w dolnym środkowym panelu ekranu.
6. Dodaj ciało żądania HTTP, jeśli jest potrzebne. Możesz to zrobić na dwa sposoby:

- Wybierz odpowiedni typ ciała żądania (np. formularz, JSON, XML) w rozwijanym menu pod adresem URL w górnym środkowym panelu ekranu. Następnie wprowadź odpowiednie dane w dolnym środkowym panelu ekranu.
- Wybierz zakładkę "Body" w dolnym środkowym panelu ekranu, a następnie wybierz odpowiedni typ ciała żądania. Następnie wprowadź odpowiednie dane w dolnym środkowym panelu ekranu.

7. Kliknij przycisk "Send" w prawym górnym rogu ekranu, aby wysłać żądanie HTTP.
8. Odczekaj na odpowiedź. Odpowiedź zostanie wyświetlona w dolnym panelu ekranu.
9. Sprawdź kod odpowiedzi HTTP i treść odpowiedzi, aby upewnić się, że żądanie zostało poprawnie przetworzone.
10. Możesz zapisać żądanie HTTP i odpowiedź, klikając przycisk "Save" w prawym górnym rogu ekranu i wybierając odpowiednią opcję z menu.

W aplikacji Postman możemy użyć zmiennych środowiskowych (environment variables) do przechowywania wartości, które mogą być łatwo konfigurowane i zmieniane. Oto jak to zrobić:

1. Otwórz Postman i przejdź do żądania, które chcesz skonfigurować.
2. W prawym górnym rogu okna Postman znajdziesz przycisk "Environment" (prawy górny róg pod przyciskami zamykającymi aplikację). Kliknij na przycisk "Manage Environments" (ikona tabeli z piktogramem oka) obok rozwijanego menu.
3. Kliknij przycisk "Add" w prawym dolnym rogu okna "Manage Environments", aby utworzyć nową wartość globalną lub lokalną (Add umieszczone z prawej górnej strony).
4. Wprowadź nazwę dla środowiska, np. "My API Environment".
5. W tabeli poniżej wprowadź "Key" i "Value" dla zmiennej, której wartość chcesz skonfigurować. Na przykład:
    - Key: **api_base_url**
    - Value: **https://example.com/api**
6. Kliknij przycisk "Add" obok pola "Key" i "Value", a następnie "Save" w prawym dolnym rogu, aby zapisać środowisko.
7. Zamknij okno "Manage Environments" i wybierz nowo utworzone środowisko z rozwijanego menu "Environment" w prawym górnym rogu Postmana.
8. W żądaniu, które chcesz skonfigurować, użyj zmiennej w nawiasach podwójnych i nawiasach klamrowych, np. **{{api_base_url}}**. Wartość zmiennej zostanie automatycznie podstawiona w miejscu użycia. Przykład:

```
GET {{api_base_url}}/endpoint
```

Teraz, gdy chcesz zmienić wartość zmiennej, wystarczy zmienić wartość zmiennej w środowisku, a zmiana zostanie zastosowana we wszystkich żądaniach używających tej zmiennej.

Aby zarządzać różnymi konfiguracjami, możesz utworzyć więcej środowisk, na przykład dla środowisk testowych i produkcyjnych. Wówczas wystarczy wybrać odpowiednie środowisko z rozwijanego menu, aby zmienić wartości zmiennych dla wszystkich żądań.

7.Logowanie zdarzeń w zapytaniach HTTP

Logowanie zdarzeń HTTP jest ważne, ponieważ umożliwia monitorowanie i analizowanie aktywności użytkowników w aplikacji webowej oraz identyfikowanie problemów z wydajnością i bezpieczeństwem.

Oto kilka powodów, dla których warto logować zdarzenia HTTP:

1. Monitorowanie aktywności użytkowników Logowanie zdarzeń HTTP umożliwia monitorowanie aktywności użytkowników w aplikacji webowej, takich jak żądania HTTP, odpowiedzi, czas odpowiedzi, ilość przesłanych danych itp. Te informacje są przydatne, gdyż pozwalają na analizę, jak użytkownicy korzystają z aplikacji, co pozwala na ich lepsze zrozumienie i dostosowanie interfejsu użytkownika do ich potrzeb.
2. Identyfikacja problemów z wydajnością Logowanie zdarzeń HTTP pozwala na identyfikowanie problemów z wydajnością aplikacji webowej, takich jak długi czas odpowiedzi, duże obciążenie serwera, błędy w przetwarzaniu żądań itp. Dzięki temu można szybciej reagować na problemy z wydajnością i podejmować odpowiednie działania, aby poprawić jej jakość.
3. Bezpieczeństwo aplikacji Logowanie zdarzeń HTTP jest również ważne w kontekście bezpieczeństwa aplikacji webowej. Dzięki logowaniu można monitorować zdarzenia związane z próbami ataków, takimi jak próby logowania z nieprawidłowymi danymi, próby wstrzykiwania SQL czy ataki DDoS. Pozwala to na szybsze reagowanie na zagrożenia i podjęcie odpowiednich działań, aby zabezpieczyć aplikację przed atakami.
4. Audytowanie aktywności Logowanie zdarzeń HTTP umożliwia również audytowanie aktywności w aplikacji webowej, co jest szczególnie ważne w przypadku aplikacji, które obsługują poufne dane lub transakcje finansowe. Logowanie takich informacji pozwala na śledzenie aktywności użytkowników i zapobieganie nieprawidłowym lub nieautoryzowanym działaniom.

Podsumowując, logowanie zdarzeń HTTP jest ważne dla monitorowania i analizowania aktywności użytkowników w aplikacji webowej oraz identyfikowania problemów z wydajnością i bezpieczeństwem. Dzięki logowaniu można szybciej reagować na problemy i podejmować odpowiednie działania, aby poprawić jakość i bezpieczeństwo aplikacji.

Filtry zapytań (Request Filters) w Spring Boot to mechanizm umożliwiający przechwytywanie, modyfikowanie i analizowanie żądań HTTP przed ich przetworzeniem przez kontrolery. Filtry są często stosowane w celu implementacji aspektów wspólnych dla wielu żądań, takich jak uwierzytelnianie, autoryzacja, rejestracja i obsługa błędów.

Filtry w Spring Boot są zazwyczaj implementowane jako klasy, które implementują interfejs **javax.servlet.Filter**. Interfejs ten definiuje trzy metody: **init()**, **doFilter()** i **destroy()**.

1. **init(FilterConfig filterConfig)**: Ta metoda jest wywoływana przez kontener serwletów podczas inicjalizacji filtra. Można ją wykorzystać do konfiguracji filtra oraz wykonania zadań inicjalizacji.
2. **doFilter(ServletRequest request, ServletResponse response, FilterChain chain)**: Jest to główna metoda filtra, w której możemy przechwycić, analizować i modyfikować żądania oraz odpowiedzi HTTP. W tej metodzie możemy również zdecydować, czy przekazać żądanie dalej do następnego filtra lub kontrolera, używając metody **chain.doFilter(request, response)**.
3. **destroy()**: Ta metoda jest wywoływana przez kontener serwletów podczas zamykania filtra. Można ją wykorzystać do zwolnienia zasobów i zakończenia zadań związanych z filtrem.

W Spring Boot, aby zarejestrować filtr jako komponent, wystarczy dodać adnotację **@Component** do klasy filtra. Spring Boot automatycznie zarządza tą klasą jako komponentem i używa jej do przetwarzania żądań HTTP.

Filtry w Spring Boot są wywoływane w określonej kolejności, zgodnie z ich priorytetem. Kolejność filtra można kontrolować, dodając adnotację **@Order** lub implementując interfejs **Ordered**. Wartości liczbowe używane w adnotacji **@Order** określają priorytet filtra; im mniejsza wartość, tym wyższy priorytet i wcześniejsze wywołanie filtra.

Przykładem zastosowania filtra może być rejestrowanie informacji o żądaniach HTTP, w jaki sposób zostało to przedstawione w poprzednich odpowiedziach. Filtr ten przechwytuje żądania HTTP, rejestruje informacje, takie jak metoda HTTP, adres URL, nagłówki żądania, parametry zapytania, adres IP i czas zapytania, a następnie przekazuje żądanie dalej do następnego filtra lub kontrolera.

Aby pobrać i zarejestrować informacje o żądaniach HTTP w aplikacji Spring Boot, można zastosować filtr lub aspekt. Poniżej przedstawię sposób użycia filtru do przechwytywania informacji o żądaniach HTTP.

1. Utwórz klasę filtru, która implementuje interfejs **Filter**:

```
@Component
public class MyFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        logRequestInfo(httpRequest);
        chain.doFilter(request, response);
    }

    private void logRequestInfo(HttpServletRequest httpRequest) {
        String method = httpRequest.getMethod();
        String requestURI = httpRequest.getRequestURI();
        String queryString = httpRequest.getQueryString();
        String ipAddress = getClientIpAddress(httpRequest);
        long timestamp = System.currentTimeMillis();

        System.out.println("Metoda HTTP: " + method);
        System.out.println("Adres URL żądania: " + requestURI);
        System.out.println("Parametry zapytania: " + queryString);
        System.out.println("Adres IP: " + ipAddress);
        System.out.println("Czas zapytania: " + timestamp);

        Enumeration<String> headerNames = httpRequest.getHeaderNames();
        System.out.println("Nagłówki żądania:");
        while (headerNames.hasMoreElements()) {
            String headerName = headerNames.nextElement();
            String headerValue = httpRequest.getHeader(headerName);
            System.out.println(headerName + ": " + headerValue);
        }
    }

    private String getClientIpAddress(HttpServletRequest request) {
        String[] headers = {
                "X-Forwarded-For",
                "Proxy-Client-IP",
                "WL-Proxy-Client-IP",
                "HTTP_CLIENT_IP",
                "HTTP_X_FORWARDED_FOR"
        };

        String ipAddress = null;

        for (String header : headers) {
            ipAddress = request.getHeader(header);
            if (ipAddress != null && ipAddress.length() != 0 &&
!"unknown".equalsIgnoreCase(ipAddress)) {
                break;
```

```
            }
        }

        if (ipAddress == null || ipAddress.length() == 0 ||
"unknown".equalsIgnoreCase(ipAddress)) {
            ipAddress = request.getRemoteAddr();
        }

        return ipAddress;
    }

    @Override
    public void destroy() {
    }
}
```

2. W powyższym kodzie, klasa **LoggingFilter** implementuje interfejs **Filter** i używa metody **doFilter()** do przechwytywania żądań HTTP. Metoda **logRequestInfo()** pobiera i rejestruje informacje o żądaniu, takie jak metoda HTTP, adres URL, parametry zapytania, adres IP oraz czas żądania.
3. Dodaj adnotację **@Component**, aby Spring automatycznie zarządzał tą klasą jako komponentem. Aplikacja Spring Boot będzie używać tego filtru do przechwytywania żądań HTTP.

Teraz, gdy uruchomisz swoją aplikację Spring Boot, informacje o żądaniach HTTP zostaną zarejestrowane w konsoli. Możesz również zastosować własny logger (np. używając biblioteki log4j lub SLF4J) zamiast **System.out.println()**, aby rejestrować informacje w bardziej elastyczny sposób, np. do plików logów.

8.Wprowadzenie do architektury REST

Należy pamiętać, że aplikacje klienckie i serwerowe muszą być w stanie rozwijać się oddzielnie, bez żadnej zależności od siebie. Klient powinien znać tylko identyfikatory zasobów. Tak długo, jak utrzymywany jest czysty interfejs między klientem a serwerem, powinno się to odbywać w sposób naturalny i bezproblemowy.

Serwery i klienci mogą być również wymienne i rozwijane niezależnie od siebie, tak długo, jak długo interfejs między nimi nie będzie zmieniany. Przy rozpoczynaniu każdego nowego projektu programista powinien zapewniać jasny podział zgodny z architekturą klient-serwer. Oznacza to, że projekty nie mogą być już rozwijane jako samodzielne jednostki. Na przykład, jeśli tworzymy projekt, w ramach którego aplikacja uzyskuje dostęp do bazy danych, musimy zadbać o to, aby aplikacja uzyskiwała dostęp do bazy danych za pomocą interfejsu zapewniającego podział zgodny na warstwę logiki biznesowej i prezentacji bądź zgodny z architekturą klient-serwer.

8.1.Nazewnictwo zasobów

Na początek dowiedzmy się czym jest zasób. Zasób to obiekt posiadający typ, powiązany z nim dane, relacje z innymi zasobami oraz zestaw metod, które na nim operują. Jest podobny do obiektu w obiektowym języku programowania, z tą ważną różnicą, że dla zasobu zdefiniowanych jest tylko kilka metod odpowiadających standardowym metodom HTTP, a są nimi: GET, POST, PUT i DELETE, podczas gdy instancja obiektu ma zazwyczaj wiele metod. Zasób jest więc obiektem, który reprezentuje jakąś "rzecz". W języku angielskim wszystkie "rzeczy" są reprezentowane są za pomocą rzeczownika. Te same zasady języka angielskiego powinny być stosowane do nazewnictwa zasobów w interfejsie API RESTful. Co więcej, dla każdego zasobu interfejs API powinien umożliwiać możliwość pobrania z zasobu albo kolekcji obiektów, albo pojedynczego obiektu. W nazwach zasobów nie należy używać czasowników. To, co wiele osób robi źle, to próba zakodowania operacji, którą próbują wykonać w nazwie swojego URI. Na przykład, aby uzyskać listę wszystkich studentów początkowy programista stworzy zasób:

/getAllInformationAboutStudents

Jest to kompletnie niezrozumienie idei RESTful API, która zakłada maksymalizację dostępnych zasobów przy jak najmniejszej ilości URI.

Jeśli budujemy interfejs REST API do zarządzania informacjami o studentach, to jest dość oczywistym jest, że będziemy miał do czynienia z obiektami zawierającymi informacje o studentach.

Dlatego naturalnie utworzyłbyś zasób o nazwie "students" w następujący sposób:

/students

Gdy klient wywoła ten zasób w Twoim API za pomocą metody GET HTTP, powinien spodziewać się że w odpowiedzi otrzyma zbiór wszystkich informacji o studentach. Jeśli klient wie, o jakim studencie chce uzyskać informacje, powinien mieć możliwość pobrania informacji tego pojedynczego studenta, wysyłając żądanie GET HTTP do zasobu w następującym formacie:

/students/404

Tutaj używamy tego samego zasobu wysokopoziomowego z dodaniem identyfikatora studenta, o którym chcemy uzyskać informacje. Zasób ten zasadniczo odpowie albo informacji o studencie, którego szukamy, lub ze statusem **HTTP 404 NOT FOUND**, jeśli API nie może znaleźć studenta o podanym identyfikatorze.

Wiemy już, jak nazwać nasz zasób, abyśmy mogli łatwo pobrać albo kolekcję studentów lub pojedynczego studenta, ale co z tworzeniem nowych studentów, aktualizacją istniejących studentów lub usuwaniem informacji o studentach. Poniższa tabela pokazuje nam, jak powinniśmy wykonać te inne operacje i przedstawia podstawowe operacje, które należy umieć wykonać na zasobach przy użyciu różnych metod HTTP, które są dla nas dostępne. Możemy wykonać całkiem sporo operacji na naszym obiekcie klienta przy minimalnej ilości tras URL zdefiniowanych w naszym API.

Zasób	**GET** czytaj	**POST** stwórz	**PUT** zaktualizuj	**DELETE** usuń
/students	Zwraca kolekcję studentów	Stwórz nowego studenta	Aktualizuje wszystkich studentów	Usuwa wszystkich studentów
/students/404	Zwraca konkretnego studenta	Metoda niedozwolona, błąd: 405	Aktualizuje konkretnego studenta	Usuwa konkretnego studenta

8.2.Bezstanowość aplikacji

Zgodnie z architekturą REST serwer nie przechowuje żadnego stanu sesji klienta po stronie serwera. To ograniczenie nazywane jest bezstanowością. Każde żądanie od klienta do serwera musi zawierać wszystkie informacje niezbędne do zrozumienia i nie może korzystać z żadnego kontekstu przechowywanego na serwerze. Stan sesji jest przechowywany w całości po stronie klienta. Dlatego to klient jest odpowiedzialny za przechowywanie i obsługę wszystkich informacji związanymi ze stanem aplikacji.

Oznacza to również, że klient jest odpowiedzialny za przesyłanie wszelkich informacji o stanie do serwera gdy tylko są one potrzebne. Na serwerze nie powinno istnieć żadne powiązanie sesji.

Bezstanowość oznacza także, że każde żądanie HTTP odbywa się w całkowitym odizolowaniu. Gdy klient zgłasza żądanie HTTP, zawiera ono wszystkie informacje niezbędne do tego, aby serwer mógł je zrealizować. Serwer nigdy nie opiera się na informacjach z poprzednich żądań. Gdyby te informacje były ważne, klient wysłałby je ponownie w tym żądaniu.

Aby umożliwić dostęp do tych bezstanowych interfejsów API, konieczne jest, aby serwery również zawierały wszystkie informacje, których klient może potrzebować do utworzenia stanu po swojej stronie.

Stan aplikacji i stan zasobów to dwie zupełnie różne rzeczy i nie powinny być mylone ze sobą.

Stan aplikacji to dane po stronie serwera, które serwery przechowują w celu identyfikacji przychodzących żądań klientów, szczegółów ich poprzednich interakcji oraz informacji o bieżącym kontekście.

Stan zasobu to bieżący stan zasobu na serwerze w dowolnym momencie i nie ma on nic wspólnego z interakcją między klientem, a serwerem. Jest to, co użytkownik otrzymuje jako odpowiedź z API. Bezstanowość REST oznacza brak zależności od stanu aplikacji.

Posiadanie bezstanowych interfejsów API REST ma kilka bardzo widocznych zalet:

1. Bezstanowość pomaga w skalowaniu interfejsów API do milionów współbieżnych użytkowników poprzez wdrażanie wielu instancji. Każda instancja API może obsługiwać dowolne żądanie w dowolnym czasie, ponieważ nie ma zależności związanych z sesją.

2. Bezstanowość sprawia, że interfejsy API REST są mniej złożone - poprzez usunięcie całej logiki synchronizacji stanu po stronie serwera.

3. Bezstanowy interfejs API jest również łatwy do buforowania. Konkretny klient może zdecydować, czy chce buforować wynik żądania HTTP, patrząc tylko na to jedno żądanie. Nie ma w tym przypadku niepewności, że stan z poprzedniego żądania może wpłynąć na możliwość buforowania bieżącej odpowiedzi. Poprawia to wydajność aplikacji.

4. Serwer nigdy nie traci orientacji, gdzie w aplikacji znajduje się każdy klient, ponieważ klient wysyła wszystkie niezbędne informacje z każdym żądaniem.

8.3.Metody HTTP

Poniżej opisano różne metody HTTP i ich zastosowanie w ramach API.

Metoda	**Zakres**	**Semantyka**
GET	Kolekcja	Uzyskaj wszystkie zasoby w kolekcji.
GET	Zasób	Uzyskaj pojedynczy zasób.
POST	Kolekcja	Tworzenie nowego zasobu w kolekcji.
PUT	Kolekcja	Zaktualizuj całą kolekcję zasobów.
PUT	Zasób	Aktualizuj pojedynczy zasób.
PATCH	Zasób	Aktualizuj jedną lub więcej wartości w pojedynczym zasobie.
DELETE	Kolekcja	Usuwa wszystkie zasoby w kolekcji. Nie należy używać treści wiadomości.
DELETE	Zasób	Usuwa pojedynczy zasób. Nie należy używać treści wiadomości.
HEAD	Kolekcja	Uzyskaj wszystkie zasoby z kolekcji, ale tylko nagłówki HTTP, bez treści
HEAD	Zasób	Uzyskaj pojedynczy zasobu, ale tylko nagłówki HTTP, bez treści
OPTIONS	Obojętne	Zwróć dostępne metody HTTP i inne opcje.

Nie używamy **requestBody** w operacjach DELETE. Należy tak zaprojektować swoją operację DELETE, aby nie zależała od **requestBody**. Zgodnie z Open API Specyfikacja **requestBody** powinna być ignorowana przez konsumentów. Rozwija to specyfikacja HTML/1.1, która wyjaśnia, że wysłanie zawartości w komunikacie żądania DELETE może spowodować, że niektóre istniejące implementacje metod mogą odrzucić żądanie.

8.4.Aktualizacja zasobów

Jeśli chodzi o aktualizację zasobów, istnieje wiele opcji. Najprostszą z nich jest użycie **HTTP PUT**, który ma tę zaletę, że jest idempotentny i **RESTful** (interakcja odbywa się poprzez reprezentacje zasobów).

PUT jest szczególnie przydatny w przypadku dużych aktualizacji, ale częściowe aktualizacje mogą zużywać więcej zasobów, niż oczekiwano, ponieważ **PUT** zmusza użytkownika do pobrania zasobu, zanim będzie mógł go zaktualizować (dokument HTTP RFC określa, że **PUT** musi przyjmować pełną reprezentację nowego zasobu jako podmiot żądania).

Rozwiązaniem tego problemu jest wyeksponowanie tych właściwości zasobu, o których wiesz, że mogą się zmienić, jako pod-zasobów i wysyłanie tam żądań PUT.

Na przykład, dla studenta zmieniającego swój adres email, użylibyśmy po prostu polecenia:

PUT /students/{userId}/email

Zamiast aktualizować cały zasób użytkownika w ten sposób:

PUT /students/{userId}

Użycie **PATCH** ma pewne wady, z których główną jest narzut zarówno po stronie klienta i serwera. Jednak **PATCH** znacznie poprawia komfort pracy programisty, jednak jest on wykonywany wolnej niż PUT, dlatego jeżeli zasoby nie muszą być często aktualizowane, bądź rozmiar przesyłanych danych jest znaczący lepiej użyć **PATCH**, zamiast **PUT**. Jednak należy pamiętać, że użycie **PATCH** jest zalecane tylko dla obiektów, które istnieją.

8.5.Kody statusu HTTP

W standardzie HTTP zdefiniowanych jest ponad 70 kodów stanu. Bardzo niewiele interfejsów API będzie musiało wykorzystywać wszystkie kody statusów, jednak my jako programiści API będziemy często korzystać z nich i warto wiedzieć o najważniejszych:

Status kodu	**Status wiadomości**	**Opis działania**
200	OK	Standardowa odpowiedź dla udanych żądań HTTP. Rzeczywista odpowiedź zależy od użytej metody żądania. W przypadku żądania GET, odpowiedź będzie zawierać encję odpowiadającą żądanemu zasobu. W przypadku żądania POST odpowiedź będzie zawierała encję opisującą lub zawierającą wynik działania.
201	Created	Żądanie zostało zrealizowane, co spowodowało utworzenie nowego zasobu. Odpowiedź może zawierać encję odpowiadającą utworzonemu zasobowi, ale dobrą praktyką jest również umieszczenie URI (lokalizacja) utworzonego zasobu w nagłówku http.
202	Accepted	Zapytanie został przyjęty do realizacji, ale realizacja nie została zakończona. Żądanie może, ale nie musi być ostatecznie rozpatrzone i może zostać odrzucone podczas przetwarzania.
204	No Content	Serwer pomyślnie przetworzył żądanie i nie zwraca żadnej zawartości.
301	Moved Permanently	Wszystkie żądania należy kierować do podanego identyfikatora URI.

304	Not Modified	Wskazuje, że zasób nie został zmodyfikowany od wersji podanej w nagłówkach żądania. W takim przypadku nie ma potrzeby retransmitowania zasobu, ponieważ klient nadal posiada poprzednio pobraną kopię.
308	Permanent Redirect	To żądanie i wszystkie przyszłe żądania należy powtórzyć, używając innego URI. 307 i 308 zachowują się podobnie do 302 i 301, ale nie pozwalają na zmianę metody HTTP. Tak więc, na przykład, przesyłanie formularza do zasobu, który został na stałe przekierowany, może być kontynuowane bezproblemowo.
400	Bad Request	Serwer nie może lub nie chce przetworzyć żądania z powodu błąd klienta (np. nieprawidłowa składnia żądania, zbyt duży rozmiar, nieprawidłowe wiadomości żądania lub kierowanie żądania).
401	Unauthorized	Podobne do 403 Zabronione, ale przeznaczone do stosowania, gdy wymagane jest uwierzytelnienie, które nie powiodło się lub nie zostało jeszcze rozpoczęte.
403	Forbidden	Żądanie było prawidłowe, ale serwer odmawia podjęcia działania. Użytkownik może nie mieć odpowiednich uprawnień do danego zasobu.
404	Not Found	Żądany zasób nie został znaleziony, ale może być dostępny w przyszłości. Kolejne żądania klienta są dopuszczalne.
408	Request Timeout	Serwer przestał czekać na żądanie. Zgodnie ze specyfikacją protokołu HTTP specyfikacji: "Klient nie zgłosił żądania w czasie w którym serwer był przygotowany do oczekiwania. Klient może powtórzyć żądanie bez modyfikacji w dowolnym późniejszym czasie."
409	Conflict	Wskazuje, że żądanie nie mogło zostać przetworzone z powodu konfliktu w bieżącym stanie zasobu, np. konflikt edycji pomiędzy wieloma jednoczesnymi aktualizacjami.
415	Unsupported Media Type	Serwer oczekiwał innej zwartości niż dostał np. zostało przesłane żądanie z wiadomością typu JSON, a oczekiwał XML
422	Unprocessable	Kod stanu 422 (Unprocessable Entity) oznacza, że serwer rozumie typ zawartości żądanej encji (dlatego

	Entity	kod statusu 415 (Unsupported Media Type) jest niewłaściwy), a także składnię elementu żądania jest poprawna (dlatego kod statusu 400 (Bad Request) jest nieodpowiedni), ale nie był w stanie przetworzyć zawartych instrukcji. Na przykład, ten stan błędu może wystąpić, jeśli treść żądania XML zawiera dobrze sformowany (tzn. poprawny składniowo), ale semantycznie błędne instrukcje XML.
500	Internal Server Error	Twórcy interfejsów API powinni unikać tego błędu. Jeśli błąd wystąpi w globalnym bloku catch, należy zarejestrować ślad (stracktrace), a nie zwracać go jako odpowiedź.
503	Service Unavailable	Serwer nie może obsłużyć żądania (ponieważ jest przeciążony lub wyłączony z powodu konserwacji). Na ogół jest to stan tymczasowy.
504	Gateway Timeout	Serwer działał jako brama lub proxy i nie otrzymywał terminowej odpowiedzi od serwera nadrzędnego.

9.Implementacja RESTful API w Spring

RESTful API (ang. Representational State Transfer) to styl architektury oprogramowania oparty na protokole HTTP, który pozwala na tworzenie usług sieciowych o różnych zasobach, manipulowanych za pomocą standardowych metod HTTP (GET, POST, PUT, DELETE, itp.). API oparte na REST jest skalowalne, wydajne i łatwe w utrzymaniu, dzięki czemu jest popularne wśród deweloperów.

Aby zaimplementować RESTful API w Spring Boot, wykonaj następujące kroki:

1. Utwórz nowy projekt Spring Boot za pomocą narzędzia Spring Initializr lub innego narzędzia, takiego jak IntelliJ IDEA lub Eclipse. Wybierz odpowiednie zależności, takie jak "Web" (Spring Web), które są niezbędne do utworzenia aplikacji internetowej.
2. Po utworzeniu projektu otwórz klasę główną, która zawiera metodę **main** i adnotację **@SpringBootApplication**. Ta klasa służy jako punkt wejścia do aplikacji.
3. Utwórz nową klasę kontrolera, która będzie obsługiwać żądania HTTP. Dodaj adnotację **@RestController** do klasy kontrolera, aby wskazać, że będzie ona obsługiwać żądania HTTP w stylu REST

```java
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MyRestController {

}
```

4. Zdefiniuj metody kontrolera, które będą obsługiwać różne żądania HTTP. Użyj adnotacji, takich jak **@GetMapping**, **@PostMapping**, **@PutMapping** i **@DeleteMapping**, do mapowania metod na odpowiednie żądania HTTP. Adnotacje te przyjmują argumenty, takie jak ścieżki URI, które mają być obsługiwane.

Przykład prostego RESTful API do zarządzania danymi użytkowników:

```java
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/users")
public class UserController {

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    @GetMapping("/{id}")
    public User getUserById(@PathVariable Long id) {
        return userService.getUserById(id);
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.createUser(user);
    }

    @PutMapping("/{id}")
    public User updateUser(@PathVariable Long id, @RequestBody User user) {
        return userService.updateUser(id, user);
    }

    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        userService.deleteUser(id);
    }
}
```

W powyższym przykładzie:

- Adnotacja **@RequestMapping("/users")** wskazuje, że kontroler będzie obsługiwać żądania związane z zasobem **/users**.
- Metoda **getAllUsers()** jest mapowana na żądania GET na **/users**, zwracając listę wszystkich użytkowników.
- Metoda **getUserById()** jest mapowana na żądania GET na **/users/{id}**, zwracając użytkownika o określonym ID.
- Metoda **createUser()** jest mapowana na żądania POST na **/users**, tworząc nowego użytkownika.
- Metoda **updateUser()** jest mapowana na żądania PUT na **/users/{id}**, aktualizując istniejącego użytkownika o określonym ID.
- Metoda **deleteUser()** jest mapowana na żądania DELETE na **/users/{id}**, usuwając użytkownika o określonym ID.

5. Utwórz serwis i repozytorium (jeśli potrzebujesz) do obsługi logiki biznesowej i dostępu do danych.
6. Uruchom aplikację Spring Boot, korzystając z metody **main** w klasie z adnotacją **@SpringBootApplication**. Aplikacja powinna uruchomić wbudowany serwer Tomcat

Aby otrzymywać dane z zapytań, można wykorzystać różne mechanizmy i biblioteki obsługujące różne protokoły komunikacji, takie jak HTTP, WebSocket czy gRPC. W kontekście Spring Boot, najczęściej używa się jego wbudowanego wsparcia dla usług RESTful, które komunikują się za pomocą protokołu HTTP.

Oto niektóre dane, które można otrzymać z zapytań w Spring Boot:

1. Parametry zapytania (Query Parameters) - parametry przekazywane w adresie URL, np. **example.com/api/users?name=John**. Parametry te można odczytać za pomocą adnotacji **@RequestParam** w kontrolerze.
2. Zmienne ścieżki (Path Variables) - zmienne części ścieżki URL, np. **example.com/api/users/{userId}**. Można je odczytać za pomocą adnotacji **@PathVariable**.
3. Nagłówki (Headers) - informacje o zapytaniu zawarte w nagłówkach HTTP, takie jak "Content-Type" czy "Authorization". Aby odczytać nagłówki, można użyć adnotacji **@RequestHeader**.
4. Ciało zapytania (Request Body) - dane przekazywane w ciele zapytania, zwłaszcza w przypadku zapytań typu POST, PUT i PATCH. Można je odczytać za pomocą adnotacji **@RequestBody**, która automatycznie zamienia dane na obiekty Java.
5. Informacje o żądaniu (Request Information) - szczegóły na temat zapytania, takie jak metoda HTTP (GET, POST, PUT, DELETE), adres URL, czy informacje o kliencie. Można je odczytać przekazując obiekt **HttpServletRequest** jako argument metody kontrolera.
6. Cookies - małe fragmenty danych przechowywane przez przeglądarkę i wysyłane z każdym zapytaniem do serwera. Można je odczytać za pomocą adnotacji **@CookieValue**.

7. Informacje o autentykacji i autoryzacji - jeśli aplikacja korzysta z zabezpieczeń Spring Security, można uzyskać informacje o bieżącym użytkowniku, jego rolach i uprawnieniach, korzystając z obiektów **Authentication** oraz **Principal** przekazywanych do kontrolera.
8. Atrybuty sesji - jeśli aplikacja używa sesji HTTP, można uzyskać dostęp do atrybutów sesji za pomocą obiektu **HttpSession**. Można przechowywać i odczytywać dane specyficzne dla sesji użytkownika, takie jak informacje o koszyku zakupowym czy preferencje.
9. Informacje o środowisku - dzięki obiektowi **Environment** można uzyskać informacje na temat środowiska uruchomieniowego, takie jak zmienne środowiskowe, wartości właściwości konfiguracyjnych czy profilu aktywnego.
10. Informacje o punkcie końcowym - w przypadku korzystania z interfejsów sieciowych (API) opartych o standard OpenAPI lub Swagger, można uzyskać informacje o punkcie końcowym, takie jak opis, parametry, czy odpowiedzi za pomocą adnotacji **@Api*** (dla Swagger 2) lub **@Operation**, **@Parameter** (dla OpenAPI 3).
11. Dostosowane atrybuty żądania - jeśli używasz filtrów lub aspektów, możesz dodać niestandardowe atrybuty do żądania, które następnie można odczytać w kontrolerze za pomocą obiektu **HttpServletRequest**.
12. Własne parametry - jeśli potrzebujesz przekazać specyficzne dane, które nie są obsługiwane przez standardowe adnotacje, możesz zdefiniować własne adnotacje, argumenty rozwiązujące (ang. argument resolvers) oraz przekazywać te dane do kontrolera.
13. Wiadomości internacjonalizacji (i18n) - za pomocą klasy **MessageSource** można odczytać zlokalizowane wiadomości na podstawie kluczy, języka oraz regionu użytkownika.

W przypadku korzystania z Spring Boot, dane te można otrzymać w kontrolerach, które obsługują żądania HTTP. Odpowiednie adnotacje i obiekty umożliwiają łatwe odczytanie tych danych, a następnie przetworzenie ich w logice aplikacji.

ResponseEntity to klasa używana w Spring Framework (w szczególności w module Spring Web MVC), która pozwala na reprezentowanie odpowiedzi HTTP z dodatkowymi informacjami, takimi jak kod statusu, nagłówki i treść odpowiedzi. Dzięki temu umożliwia pełną kontrolę nad odpowiedzią, która zostanie wysłana do klienta.

Za pomocą **ResponseEntity** możemy określić:

1. Kod statusu HTTP: ResponseEntity pozwala na wybór kodu statusu HTTP, który chcemy zwrócić, np. **HttpStatus.OK**, **HttpStatus.BAD_REQUEST**, **HttpStatus.INTERNAL_SERVER_ERROR** itp. Kody te reprezentują różne statusy odpowiedzi, które są zrozumiałe dla klienta HTTP.
2. Nagłówki: ResponseEntity pozwala na ustawienie nagłówków odpowiedzi, które będą zwrócone wraz z odpowiedzią. Nagłówki to metadane przesyłane z odpowiedzią, takie jak format danych, długość treści, informacje o cachowaniu itp.
3. Treść odpowiedzi: ResponseEntity umożliwia również definiowanie treści odpowiedzi, która zostanie zwrócona do klienta. Może to być JSON, XML, tekst lub inny format danych, który chcemy przekazać.

Kiedy tworzymy instancję **ResponseEntity**, możemy określić typ danych (generyczny), który zostanie zwrócony jako treść odpowiedzi. Na przykład, **ResponseEntity<String>** oznacza, że treścią odpowiedzi będzie tekst.

Aby utworzyć obiekt **ResponseEntity**, można użyć jednego z poniższych podejść:

1. Konstruktor: używając konstruktorów klasy ResponseEntity, np. **new ResponseEntity<>(body, status)**.
2. Metody statyczne: ResponseEntity dostarcza zestawu metod statycznych, takich jak **ok()**, **badRequest()**, **status()**, itp. Służą one do tworzenia odpowiedzi o określonym statusie HTTP.

Przykład użycia **ResponseEntity**:

```
@GetMapping("/example")
public ResponseEntity<String> exampleEndpoint() {
    String responseBody = "Hello, World!";
    HttpHeaders headers = new HttpHeaders();
    headers.add("Custom-Header", "custom_value");

    return new ResponseEntity<>(responseBody, headers, HttpStatus.OK);
}
```

W powyższym przykładzie, metoda **exampleEndpoint()** zwraca obiekt **ResponseEntity<String>** z treścią "Hello, World!", nagłówkiem "Custom-Header" o wartości "custom_value" i kodem statusu HTTP **200 OK**.

9.1.Implementacja obsługi wybranych zasobów

Aby zwrócić obiekt jako plik JSON, możemy użyć **ResponseEntity** wraz z adnotacją **@RestController**, która automatycznie przekształca obiekty Java w JSON. W poniższym przykładzie zdefiniujemy klasę **Person** oraz kontroler z endpointem, który zwraca obiekt **Person** jako JSON.

1. Utwórz klasę **Person**:

```java
package com.example.demo.model;

public class Person {
    private String firstName;
    private String lastName;
    private int age;

    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    // Getters and setters
    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

2. Utwórz kontroler z endpointem, który zwraca obiekt **Person** jako JSON:

```java
package com.example.demo.controller;

import com.example.demo.model.Person;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class PersonController {

    @GetMapping("/person")
    public ResponseEntity<Person> getPerson() {
        Person person = new Person("John", "Doe", 30);

        // Jeśli chcesz dodać nagłówki, można to zrobić tutaj
        // HttpHeaders headers = new HttpHeaders();
        // headers.add("Custom-Header", "custom_value");

        // Zwraca obiekt Person jako JSON z kodem statusu 200 OK
        return new ResponseEntity<>(person, HttpStatus.OK);
    }
}
```

Teraz, gdy wyślesz żądanie GET do endpointu **/person**, otrzymasz odpowiedź w formacie JSON:

```json
{
    "firstName": "John",
    "lastName": "Doe",
    "age": 30
}
```

Spring Boot automatycznie przekształca obiekt **Person** w JSON dzięki adnotacji **@RestController** i konwersji Jacksona.

Aby zaimplementować endpoint typu POST obsługujący przesyłanie plików oraz nagłówków w Spring Boot, wykonaj następujące kroki:

1. Utwórz kontroler w odpowiedniej lokalizacji pakietu, np. **com.example.demo.controller**.

```
package com.example.demo.controller;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MultipartFile;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;

@RestController
public class FileUploadController {

    private final String UPLOAD_DIRECTORY = "uploads";

    @PostMapping("/upload")
    public ResponseEntity<String> uploadFile(
            @RequestParam("file") MultipartFile file,
            @RequestHeader("custom-header") String customHeaderValue) throws
IOException {

        // Przetwarzanie nagłówka
        System.out.println("Custom header value: " + customHeaderValue);

        // Przetwarzanie pliku
        if (file.isEmpty()) {
            return new ResponseEntity<>("File is empty",
HttpStatus.BAD_REQUEST);
        }

        try {
            Path targetLocation = Paths.get(UPLOAD_DIRECTORY)
                    .resolve(file.getOriginalFilename());
            Files.copy(file.getInputStream(), targetLocation,
StandardCopyOption.REPLACE_EXISTING);
        } catch (IOException e) {
            return new ResponseEntity<>("File upload failed",
HttpStatus.INTERNAL_SERVER_ERROR);
        }

        return new ResponseEntity<>("File uploaded successfully: " +
file.getOriginalFilename(), HttpStatus.OK);
```

```
    }
}
```

Aby zaimplementować obsługę plików zgodnie z architekturą RESTful w Spring Boot, należy utworzyć kontroler RESTful z endpointami dla różnych operacji CRUD (Create, Read, Update, Delete). W poniższym przykładzie pokażemy, jak zaimplementować obsługę plików dla operacji dodawania, pobierania, aktualizowania i usuwania plików.

```
package com.example.demo.controller;

import org.springframework.core.io.Resource;
import org.springframework.core.io.UrlResource;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import java.io.IOException;
import java.net.MalformedURLException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;

@RestController
@RequestMapping("/files")
public class FileController {

    private final String UPLOAD_DIRECTORY = "uploads";

    @PostMapping
    public ResponseEntity<String> uploadFile(@RequestParam("file")
MultipartFile file) throws IOException {
        if (file.isEmpty()) {
            return new ResponseEntity<>("File is empty",
HttpStatus.BAD_REQUEST);
        }

        Path targetLocation =
Paths.get(UPLOAD_DIRECTORY).resolve(file.getOriginalFilename());
        Files.copy(file.getInputStream(), targetLocation,
StandardCopyOption.REPLACE_EXISTING);

        String fileDownloadUri =
ServletUriComponentsBuilder.fromCurrentContextPath()
                .path("/files/")
                .path(file.getOriginalFilename())
                .toUriString();

        return new ResponseEntity<>("File uploaded successfully: " +
fileDownloadUri, HttpStatus.CREATED);
    }
```

```
    @GetMapping("/{filename}")
    public ResponseEntity<Resource> downloadFile(@PathVariable String
filename) {
        Path filePath =
Paths.get(UPLOAD_DIRECTORY).resolve(filename).normalize();

        Resource resource;
        try {
            resource = new UrlResource(filePath.toUri());
        } catch (MalformedURLException e) {
            return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
        }

        if (!resource.exists()) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }

        HttpHeaders headers = new HttpHeaders();
        headers.add(HttpHeaders.CONTENT_DISPOSITION, "attachment;
filename=\"" + resource.getFilename() + "\"");

        return new ResponseEntity<>(resource, headers, HttpStatus.OK);
    }

    @PutMapping("/{filename}")
    public ResponseEntity<String> updateFile(
            @PathVariable String filename,
            @RequestParam("file") MultipartFile updatedFile) throws
IOException {

        if (updatedFile.isEmpty()) {
            return new ResponseEntity<>("File is empty",
HttpStatus.BAD_REQUEST);
        }

        Path targetLocation = Paths.get(UPLOAD_DIRECTORY).resolve(filename);
        Files.copy(updatedFile.getInputStream(), targetLocation,
StandardCopyOption.REPLACE_EXISTING);

        return new ResponseEntity<>("File updated successfully: " + filename,
HttpStatus.OK);
    }

    @DeleteMapping("/{filename}")
public ResponseEntity<String> deleteFile(@PathVariable String filename) {
    Path filePath =
Paths.get(UPLOAD_DIRECTORY).resolve(filename).normalize();

    if (!Files.exists(filePath)) {
        return new ResponseEntity<>("File not found", HttpStatus.NOT_FOUND);
    }

    try {
        Files.delete(filePath);
    } catch (IOException e) {
```

```
        return new ResponseEntity<>("File deletion failed",
HttpStatus.INTERNAL_SERVER_ERROR);
    }

    return new ResponseEntity<>("File deleted successfully: " + filename,
HttpStatus.NO_CONTENT);
}
}
```

W powyższym kontrolerze **FileController**:

- Utworzono kontroler z adnotacją **@RestController** i mapowaniem na **/files**.
- Zdefiniowano metody dla operacji CRUD:
    - **uploadFile()**: Dodawanie plików - obsługuje żądania POST na **/files**.
    - **downloadFile()**: Pobieranie plików - obsługuje żądania GET na **/files/{filename}**.
    - **updateFile()**: Aktualizowanie plików - obsługuje żądania PUT na **/files/{filename}**.
    - **deleteFile()**: Usuwanie plików - obsługuje żądania DELETE na **/files/{filename}**.

Teraz masz kontroler obsługujący pliki zgodnie z architekturą RESTful w Spring Boot. Aby przetestować te endpointy, możesz użyć narzędzi takich jak Postman, Insomnia lub cURL. Pamiętaj, że przed przetestowaniem należy utworzyć katalog **uploads** w głównym katalogu projektu, ponieważ pliki będą tam przechowywane.

Aby zezwolić tylko na przesyłanie plików zip w Spring Boot, można dodać walidację formatu pliku w metodzie kontrolera obsługującej żądanie przesyłania pliku. W przypadku używania **MultipartFile**, można sprawdzić rozszerzenie pliku przed przetworzeniem go. Oto jak to zrobić:

1. Utwórz prostą metodę pomocniczą do sprawdzenia rozszerzenia pliku:

```
private boolean isZipFile(String filename) {
    return filename != null && filename.toLowerCase().endsWith(".zip");
}
```

2. Zastosuj walidację rozszerzenia pliku w kontrolerze obsługującym przesyłanie plików:

```
@PostMapping
public ResponseEntity<String> uploadFile(@RequestParam("file") MultipartFile
file) throws IOException {
    String filename = file.getOriginalFilename();

    if (file.isEmpty()) {
        return new ResponseEntity<>("File is empty", HttpStatus.BAD_REQUEST);
    }

    if (!isZipFile(filename)) {
        return new ResponseEntity<>("Only ZIP files are allowed",
HttpStatus.BAD_REQUEST);
```

```
    }

    Path targetLocation = Paths.get(UPLOAD_DIRECTORY).resolve(filename);
    Files.copy(file.getInputStream(), targetLocation,
StandardCopyOption.REPLACE_EXISTING);

    String fileDownloadUri =
ServletUriComponentsBuilder.fromCurrentContextPath()
            .path("/files/")
            .path(filename)
            .toUriString();

    return new ResponseEntity<>("File uploaded successfully: " +
fileDownloadUri, HttpStatus.CREATED);
}
```

W powyższym kodzie, metoda **uploadFile()** sprawdza, czy przesłany plik jest pusty, a następnie sprawdza, czy ma rozszerzenie **.zip**, korzystając z metody **isZipFile()**. Jeśli plik nie jest w formacie zip, żądanie zostaje odrzucone z odpowiedzią o kodzie **400 Bad Request**. W przeciwnym razie, plik zostaje przetworzony i zapisany na serwerze.

Teraz Twój kontroler akceptuje tylko pliki zip do przesyłania.

Aby zabezpieczyć się przed fałszywymi plikami zip, można użyć biblioteki do sprawdzania zawartości pliku, zamiast polegać wyłącznie na rozszerzeniu pliku. Jedną z takich bibliotek jest Apache Commons Compress. Oto jak użyć jej do sprawdzania, czy plik jest prawdziwym plikiem zip:

1. Dodaj zależność **commons-compress** do pliku **pom.xml** lub **build.gradle**.

pom.xml:

```
<dependencies>
    <!-- ... -->
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-compress</artifactId>
        <version>1.21</version>
    </dependency>
    <!-- ... -->
</dependencies>
```

2. Utwórz metodę pomocniczą, która sprawdza, czy plik jest prawdziwym plikiem zip:

```
import org.apache.commons.compress.archivers.zip.ZipArchiveInputStream;

private boolean isTrueZipFile(MultipartFile file) {
    try (ZipArchiveInputStream zipInput = new
ZipArchiveInputStream(file.getInputStream())) {
        // Spróbuj odczytać pierwszy wpis z archiwum zip
        // Jeśli się powiedzie, plik jest prawdziwym plikiem zip
        return zipInput.getNextZipEntry() != null;
    } catch (IOException e) {
```

```
        // Jeśli wystąpi wyjątek, plik nie jest prawdziwym plikiem zip
        return false;
    }
}
```

Teraz Twój kontroler sprawdza, czy przesłany plik jest prawdziwym plikiem zip, zanim przetworzy go i zapisze na serwerze. Jeśli plik nie jest prawdziwym plikiem zip, żądanie zostaje odrzucone z odpowiedzią o kodzie **400 Bad Request**.

10. Obsługa sieci w Pythonie

Python oferuje różne sposoby obsługi sieci:

1. Moduł socket - umożliwia komunikację między procesami w sieci.
2. Moduł urllib - zapewnia dostęp do plików i zasobów sieciowych, w tym protokołów HTTP, FTP i inne.
3. Moduł socketserver - ułatwia tworzenie serwerów sieciowych.
4. Biblioteki zewnętrzne, takie jak requests, aiohttp czy httpx - popularne biblioteki ułatwiające komunikację z serwerami sieciowymi.

10.1.Obsługa gniazd

Należy pamiętać, że przykłady poniżej nie posiadają solidnej obsługi błędów (np. co jeśli port serwera jest już zajęty?). W kodzie produkcyjnym warto dodać sprawdzanie i obsługę wyjątków, aby kod był bardziej odporny.

Przykład implementacji serwera i klienta TCP w Pythonie:

```
# Serwer TCP
import socket


def start_server():
    host = '127.0.0.1'
    port = 12345

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((host, port))
    server_socket.listen(1)

    print(f"Serwer nasłuchuje na {host}:{port}")

    while True:
        client_socket, addr = server_socket.accept()
        print(f"Połączenie od {addr}")

        data = client_socket.recv(1024).decode('utf-8')
        print(f"Otrzymano: {data}")

        response = f"Serwer odebrał: {data}"
        client_socket.send(response.encode('utf-8'))
```

```
        client_socket.close()


# Klient TCP
def start_client():
    host = '127.0.0.1'
    port = 12345

    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((host, port))

    message = "Witaj, serwerze!"
    client_socket.send(message.encode('utf-8'))

    response = client_socket.recv(1024).decode('utf-8')
    print(f"Odpowiedź serwera: {response}")

    client_socket.close()


if __name__ == "__main__":
    import threading

    server_thread = threading.Thread(target=start_server)
    server_thread.start()

    # Daj serwerowi czas na uruchomienie
    import time

    time.sleep(1)

    start_client()
```

Kod jest podzielony na dwie główne części:

- Funkcja start_server(): Definiuje zachowanie serwera.
- Funkcja start_client(): Definiuje zachowanie klienta.

Blok if __name__ == "__main__": organizuje wykonanie zarówno serwera, jak i klienta za pomocą wątków, zapewniając ich jednoczesne działanie.

Serwer (start_server)

1. **Konfiguracja:**
    - Bindowanie do localhost (127.0.0.1) na porcie 12345.
    - Rozpoczyna nasłuchiwanie połączeń przychodzących, pozwalając na maksymalnie jedno połączenie w kolejce.
2. **Pętla Obsługi Połączeń (while True)**
    - Akceptuje połączenie klienta, uzyskując client_socket do komunikacji i adres klienta (addr).
    - Odbiera do 1024 bajtów danych od klienta, dekoduje je z UTF-8.
    - Wyświetla odebrane dane.
    - Tworzy wiadomość z odpowiedzią.
    - Wysyła odpowiedź z powrotem do klienta.

- Zamyka połączenie z klientem.

Klient (start_client)

1. **Konfiguracja:**
   - Łączy się z serwerem na localhost na porcie 12345.
2. **Komunikacja:**
   - Wysyła wiadomość powitalną ("Witaj, serwerze!") do serwera.
   - Odbiera odpowiedź serwera (do 1024 bajtów), dekoduje ją z UTF-8.
   - Wyświetla odpowiedź serwera.
   - Zamyka połączenie.

Współbieżność z Wątkami

- Tworzony jest wątek dla funkcji start_server, umożliwiając serwerowi działanie w tle.
- Wprowadzane jest krótkie opóźnienie (1 sekunda), aby dać serwerowi czas na uruchomienie przed próbą połączenia klienta.
- Funkcja start_client jest następnie wykonywana w głównym wątku.

Kluczowe Punkty

- **TCP:** Ten kod demonstruje podstawowy model żądanie-odpowiedź komunikacji TCP.
- **Wątki:** Wątki umożliwiają serwerowi i klientowi jednoczesne działanie, co jest niezbędne w rzeczywistych aplikacjach sieciowych, gdzie wielu klientów może łączyć się z serwerem.

10.2.Obsługa protokołu UDP

Przykład implementacji klienta i serwera UDP w Pythonie:

```
# Serwer UDP
import socket


def start_udp_server():
    host = '127.0.0.1'
    port = 12345

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_socket.bind((host, port))

    print(f"Serwer UDP nasłuchuje na {host}:{port}")

    while True:
        data, addr = server_socket.recvfrom(1024)
        print(f"Otrzymano od {addr}: {data.decode('utf-8')}")

        response = f"Serwer UDP odebrał: {data.decode('utf-8')}"
        server_socket.sendto(response.encode('utf-8'), addr)
```

```
# Klient UDP
def start_udp_client():
    host = '127.0.0.1'
    port = 12345

    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    message = "Witaj, serwerze UDP!"
    client_socket.sendto(message.encode('utf-8'), (host, port))

    data, _ = client_socket.recvfrom(1024)
    print(f"Odpowiedź serwera UDP: {data.decode('utf-8')}")

    client_socket.close()


if __name__ == "__main__":
    import threading

    server_thread = threading.Thread(target=start_udp_server)
    server_thread.start()

    # Daj serwerowi czas na uruchomienie
    import time

    time.sleep(1)

    start_udp_client()
```

Podobnie jak poprzedni przykład, ten kod również składa się z dwóch głównych części:

- Funkcja start_udp_server(): Definiuje zachowanie serwera UDP.
- Funkcja start_udp_client(): Definiuje zachowanie klienta UDP.

Blok if __name__ == "__main__": zarządza jednoczesnym wykonaniem serwera i klienta za pomocą wątków.

Serwer UDP (start_udp_server)

1. **Konfiguracja:**
    - Tworzy gniazdo (socket) typu SOCK_DGRAM, co oznacza, że będzie używać protokołu UDP.
    - Binduje gniazdo do adresu localhost (127.0.0.1) na porcie 12345.
    - Wyświetla komunikat informujący, że serwer nasłuchuje.
2. **Pętla Obsługi Danych (while True)**
    - Odbiera datagram (do 1024 bajtów) wraz z adresem nadawcy (addr) za pomocą recvfrom.
    - Wyświetla odebrane dane i adres nadawcy.
    - Tworzy wiadomość z odpowiedzią.
    - Wysyła odpowiedź z powrotem do nadawcy (addr) za pomocą sendto.

Klient UDP (start_udp_client)

1. **Konfiguracja:**
    - Tworzy gniazdo typu SOCK_DGRAM dla UDP.
2. **Komunikacja:**
    - Tworzy wiadomość do wysłania.
    - Wysyła wiadomość do serwera na localhost (127.0.0.1), port 12345 za pomocą sendto.
    - Odbiera odpowiedź od serwera (do 1024 bajtów) wraz z adresem nadawcy (który jest ignorowany w tym przypadku za pomocą _).
    - Wyświetla odpowiedź serwera.
    - Zamyka gniazdo.

Współbieżność z Wątkami

- Podobnie jak poprzednio, wątek jest tworzony dla funkcji serwera (start_udp_server), umożliwiając jego działanie w tle.
- Wprowadza się krótkie opóźnienie, aby dać serwerowi czas na uruchomienie.
- Funkcja klienta (start_udp_client) jest wykonywana w głównym wątku.

Kluczowe Różnice między UDP i TCP

- **Brak Połączenia:** UDP jest protokołem bezpołączeniowym, co oznacza, że nie ma ustanawiania formalnego połączenia między klientem a serwerem. Każdy datagram jest wysyłany niezależnie.
- **Brak Gwarancji Dostarczenia:** UDP nie gwarantuje, że datagramy dotrą do celu, ani że dotrą w takiej samej kolejności, w jakiej zostały wysłane.
- **Mniejsza Narzut:** UDP ma mniejszy narzut niż TCP, ponieważ nie ma potrzeby utrzymywania stanu połączenia.

10.3.Obsługa zapytań HTTP

Kod poniżej demonstruje podstawowe operacje CRUD (Create, Read, Update, Delete) na hipotetycznym API RESTful przy użyciu biblioteki requests. Wysyła różne typy żądań HTTP do serwera API i wyświetla odpowiedzi.

Importy:

- requests: Importuje bibliotekę requests, która upraszcza wysyłanie żądań HTTP i obsługę odpowiedzi.

Żądania HTTP:

- **GET:**
    - Wysyła żądanie GET do https://api.example.com/users, aby pobrać listę użytkowników.
    - Wyświetla kod statusu odpowiedzi (np. 200 oznacza sukces) oraz zawartość odpowiedzi w formacie JSON.

- **POST:**
  - Tworzy słownik data z danymi nowego użytkownika (imię i email).
  - Wysyła żądanie POST do https://api.example.com/users wraz z danymi w formacie JSON, aby utworzyć nowego użytkownika.
  - Wyświetla kod statusu i zawartość odpowiedzi (może to być np. identyfikator nowo utworzonego użytkownika).
- **PUT:**
  - Ustawia user_id na 1.
  - Tworzy słownik data z zaktualizowanymi danymi użytkownika.
  - Wysyła żądanie PUT do https://api.example.com/users/1 (zastępując 1 wartością user_id), aby zaktualizować dane użytkownika o podanym identyfikatorze.
  - Wyświetla kod statusu i zawartość odpowiedzi (może to być np. zaktualizowane dane użytkownika).
- **DELETE:**
  - Ustawia user_id na 1.
  - Wysyła żądanie DELETE do https://api.example.com/users/1, aby usunąć użytkownika o podanym identyfikatorze.
  - Wyświetla kod statusu odpowiedzi (np. 204 oznacza brak zawartości, co jest typową odpowiedzią na udane żądanie DELETE).

```python
import requests

# GET request
response = requests.get('https://api.example.com/users')
print(f"Status: {response.status_code}")
print(f"Zawartość: {response.json()}")

# POST request
data = {'name': 'Michał Młodawski', 'email': 'michal@mlodaw.ski'}
response = requests.post('https://api.example.com/users', json=data)
print(f"Status: {response.status_code}")
print(f"Zawartość: {response.json()}")

# PUT request
user_id = 1
data = {'name': 'Paulina Kowalska', 'email': 'pkowalska@tu.kielce.pl'}
response = requests.put(f'https://api.example.com/users/{user_id}',
json=data)
print(f"Status: {response.status_code}")
print(f"Zawartość: {response.json()}")

# DELETE request
user_id = 1
response = requests.delete(f'https://api.example.com/users/{user_id}')
print(f"Status: {response.status_code}")
```

Ten kod ilustruje, jak w prosty sposób komunikować się z API RESTful za pomocą biblioteki requests. Pokazuje, jak wysyłać różne typy żądań HTTP (GET, POST, PUT, DELETE) wraz z danymi i jak obsługiwać odpowiedzi serwera. Jest to przydatne narzędzie do interakcji z różnymi usługami internetowymi, które udostępniają swoje API.

10.4.Wprowadzenie do architektury REST

REST (Representational State Transfer) to styl architektury dla rozproszonych systemów hipermedialnych. Główne zasady REST:

1. Jednolity interfejs
2. Bezstanowość
3. Buforowanie
4. System warstwowy
5. Kod na żądanie (opcjonalnie)
6. Architektura klient-serwer

Zasoby w REST są identyfikowane przez URI, a operacje na zasobach są wykonywane za pomocą metod HTTP:

- GET: Pobieranie zasobu
- POST: Tworzenie nowego zasobu
- PUT: Aktualizacja istniejącego zasobu
- DELETE: Usuwanie zasobu

6. Implementacja RESTful API w Pythonie

Kod poniżej implementuje proste API RESTful do zarządzania użytkownikami, wykorzystując Flask oraz przykładową bazę danych w postaci listy. Zapewnia podstawowe operacje CRUD (Create, Read, Update, Delete) na zasobach użytkowników.

Importy

- Flask, jsonify, request: Importuje niezbędne elementy z biblioteki Flask do tworzenia aplikacji internetowych, formatowania danych do JSON oraz obsługi żądań HTTP.

Przykładowa Baza Danych

- users: Lista słowników reprezentująca przykładową bazę danych użytkowników, zawierającą identyfikator, imię i adres e-mail każdego użytkownika.

Trasy (Routes) i Funkcje Obsługi

- /users (GET): Zwraca listę wszystkich użytkowników w formacie JSON.
- /users/<int:user_id> (GET): Zwraca dane konkretnego użytkownika na podstawie jego identyfikatora (user_id). Jeśli użytkownik nie istnieje, zwraca błąd 404.
- /users (POST): Tworzy nowego użytkownika na podstawie danych JSON przesłanych w ciele żądania. Przypisuje nowy identyfikator (największy istniejący + 1) i dodaje użytkownika do listy users. Zwraca dane nowego użytkownika z kodem statusu 201 (Created).

- /users/<int:user_id> (PUT): Aktualizuje dane istniejącego użytkownika na podstawie jego identyfikatora i danych JSON przesłanych w ciele żądania. Jeśli użytkownik nie istnieje, zwraca błąd 404.
- /users/<int:user_id> (DELETE): Usuwa użytkownika na podstawie jego identyfikatora. Zwraca pusty ciąg z kodem statusu 204 (No Content).

Uruchamianie Aplikacji

- if __name__ == '__main__':: Zapewnia, że kod w tym bloku jest wykonywany tylko wtedy, gdy skrypt jest uruchamiany bezpośrednio.
- app.run(debug=True): Uruchamia aplikację Flask w trybie debugowania, co umożliwia automatyczne przeładowanie serwera po zmianach w kodzie oraz dostarcza szczegółowe informacje o błędach.

```python
from flask import Flask, jsonify, request

app = Flask(__name__)

# Przykładowa baza danych
users = [
    {"id": 1, "name": "Michal", "email": "michal@mlodaw.ski"},
    {"id": 2, "name": "Paulina", "email": "pkowalska@tu.kielce.pl"}
]

@app.route('/users', methods=['GET'])
def get_users():
    return jsonify(users)

@app.route('/users/<int:user_id>', methods=['GET'])
def get_user(user_id):
    user = next((user for user in users if user['id'] == user_id), None)
    if user:
        return jsonify(user)
    return jsonify({"error": "User not found"}), 404

@app.route('/users', methods=['POST'])
def create_user():
    new_user = request.json
    new_user['id'] = max(user['id'] for user in users) + 1
    users.append(new_user)
    return jsonify(new_user), 201

@app.route('/users/<int:user_id>', methods=['PUT'])
def update_user(user_id):
    user = next((user for user in users if user['id'] == user_id), None)
    if user:
        user.update(request.json)
        return jsonify(user)
    return jsonify({"error": "User not found"}), 404

@app.route('/users/<int:user_id>', methods=['DELETE'])
def delete_user(user_id):
    global users
    users = [user for user in users if user['id'] != user_id]
    return '', 204
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

8. Logowanie zdarzeń w zapytaniach HTTP

Przykład logowania zdarzeń HTTP przy użyciu dekoratora w Flask:

```
from flask import Flask, request
import logging
from functools import wraps
app = Flask(__name__)

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def log_request(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        logger.info(f"Request: {request.method} {request.url}")
        logger.info(f"Headers: {request.headers}")
        logger.info(f"Body: {request.get_data(as_text=True)}")
        response = f(*args, **kwargs)
        logger.info(f"Response: {response}")
        return response
    return decorated_function

@app.route('/example', methods=['GET', 'POST'])
@log_request
def example():
    if request.method == 'GET':
        return "This is a GET request"
    elif request.method == 'POST':
        return "This is a POST request"

if __name__ == '__main__':
    app.run(debug=True)
```

Kod powyżej implementuje prostą aplikację Flask, która loguje przychodzące żądania HTTP i ich odpowiedzi. Wykorzystuje dekorator log_request do przechwytywania szczegółów żądania i odpowiedzi oraz rejestrowania ich za pomocą modułu logging.

Importy

- Flask, request: Importuje niezbędne komponenty z biblioteki Flask do tworzenia aplikacji internetowej i obsługi żądań HTTP.
- logging: Importuje moduł do rejestrowania zdarzeń i informacji diagnostycznych.
- wraps: Importuje funkcję wraps z modułu functools, która jest używana do zachowania metadanych oryginalnej funkcji podczas dekorowania.

Konfiguracja Logowania

- logging.basicConfig(level=logging.INFO): Konfiguruje podstawowy poziom logowania na INFO. Oznacza to, że będą rejestrowane wszystkie komunikaty o poziomie INFO lub wyższym (np. WARNING, ERROR).
- logger = logging.getLogger(__name__): Tworzy obiekt loggera powiązany z nazwą bieżącego modułu (__name__).

Dekorator log_request

- @wraps(f): Dekorator wraps zapewnia, że dekorowana funkcja zachowuje swoją oryginalną nazwę, docstring i inne atrybuty.
- decorated_function(*args, **kwargs): Ta funkcja wewnętrzna jest wywoływana za każdym razem, gdy dekorowana funkcja jest wywoływana.
    - Rejestruje metodę HTTP, URL, nagłówki i treść żądania.
    - Wywołuje oryginalną funkcję (f(*args, **kwargs)) i przechowuje jej odpowiedź.
    - Rejestruje odpowiedź.
    - Zwraca odpowiedź z oryginalnej funkcji.

Trasa (Route) /example

- @app.route('/example', methods=['GET', 'POST']): Definiuje trasę /example, która obsługuje zarówno żądania GET, jak i POST.
- @log_request: Dekoruje funkcję example, aby automatycznie rejestrować żądania i odpowiedzi dla tej trasy.

Funkcja example

- Sprawdza metodę HTTP żądania (request.method).
- Zwraca odpowiedni komunikat w zależności od metody (GET lub POST).

Uruchamianie Aplikacji

- if __name__ == '__main__':: Zapewnia, że kod w tym bloku jest wykonywany tylko wtedy, gdy skrypt jest uruchamiany bezpośrednio, a nie importowany jako moduł.
- app.run(debug=True): Uruchamia aplikację Flask w trybie debugowania. Tryb debugowania zapewnia szczegółowe informacje o błędach i automatycznie przeładowuje serwer po zmianach w kodzie.