Що таке WebSocket?

Websocket — це протокол комп’ютерного зв’язку, або, кажучи простою мовою, «набір правил» для двох комп’ютерів для спілкування один з одним. Він встановлює двонаправлений зв’язок між двома комп’ютерами через сокет TCP. Давайте розберемося, як WebSocket допомагає безперешкодно надавати дані. WebSocket, по суті, встановлює безперервне двостороннє з’єднання між клієнтом (запитує дані) і сервером (надсилає дані). Одним із прикладів є телефонний дзвінок, коли після з’єднання дзвінка забезпечується безперебійний двосторонній зв’язок

Що таке REST API?

REST — це набір обмежень або принципів проектування для створення веб-служб, які використовують протокол HTTP для зв’язку між клієнтом і сервером, проте зв’язок є односпрямованим у будь-який момент часу. Тому REST API надає дані (або будь-який ресурс) лише за запитом. Це схоже на торговельний автомат: вам потрібно запитати правильний шлях для отримання їжі, щоб отримати її.

Для яких цілей використовується Websocket?

Websocket використовується переважно там, де доставка даних є безперервною (хоча не завжди) і відбувається у дві сторони, наприклад, для низької затримки та високочастотних каналів даних forex або додатків чату. Клієнту буде потрібно багато ресурсів запитувати за 100 одиниць інф. щосекунди за допомогою REST API, тоді як клієнт Websocket може отримувати 100 од.інф. на секунду без надто інтенсивних ресурсів.

Для яких цілей використовується REST API?

REST API використовується там, де вимоги до даних не є безперервними та не мають статусу. Наприклад, запитувати дані в реальному часі, коли вони потрібні вашій програмі, або запитувати історичні дані про часовий ряд для відображення діаграми. REST — це концепція, заснована на ресурсах, схожа на наш приклад торгового автомата.

Коли використовувати Websocket, а коли REST?

Websockets проти REST насправді не є «що найкраще?» питання, а скоріше «що доцільніше?». REST більш універсальний і може надати майже всі доступні ресурси, які ми надаємо. Він чудово працює, коли потреба в даних є тимчасовою, однак вона не корисна, коли доставка даних має бути надшвидкою (наприклад, понад 100 запитів на секунду) і безперервною, коли кожен тік має значення (у випадку даних форекс). Веб-сокети ідеально підходять у сценаріях із високим навантаженням.

WebSockets і REST технічно блискучі самі по собі, звідси їхня популярність останнім часом. Відповідний вибір залежить від потреб користувача в даних.

Термін «програмування сокетів» означає написання програм, які виконуються на кількох комп’ютерах, де всі пристрої підключені один до одного за допомогою мережі.

Є два протоколи зв’язку, які ми можемо використовувати для програмування сокетів: протокол дейтаграм користувача (UDP) і протокол керування передачею (TCP).

Основна відмінність між ними полягає в тому, що UDP не використовує з’єднання, тобто немає сеансу між клієнтом і сервером, тоді як TCP орієнтований на з’єднання, тобто спочатку має бути встановлено ексклюзивне з’єднання між клієнтом і сервером для встановлення зв’язку.

**вступ до програмування сокетів у мережах TCP/IP і демонструє, як писати програми клієнт/сервер на Java.**

**2. Налаштування проекту**

Java надає набір класів та інтерфейсів, які піклуються про деталі зв’язку низького рівня між клієнтом і сервером.

Вони здебільшого містяться в пакеті java.net, тому нам потрібно виконати такий імпорт:

***import*** *java.net.\*;*

***import*** *java.io.\*;*

Це буде програма двостороннього зв’язку, де клієнт вітає сервер, а сервер відповідає.

Я створю серверну програму в класі під назвою HelloServer.java з наступним кодом.

Ми включимо основний метод і глобальні змінні, щоб привернути увагу до того, як ми будемо запускати всі сервери. Для решти прикладів у цьому рефераті ми опустимо такий повторюваний код:

***public******class***HelloServer *{*

***private*** *ServerSocket serverSocket;*

***private*** *Socket clientSocket;*

***private*** *PrintWriter out;*

***private*** *BufferedReader in;*

***public******void******start****(****int*** *port) {*

*serverSocket =* ***new******ServerSocket****(port);*

*clientSocket = serverSocket.accept();*

*out =* ***new******PrintWriter****(clientSocket.getOutputStream(), true);*

*in =* ***new******BufferedReader****(****new******InputStreamReader****(clientSocket.getInputStream()));*

***String*** *greeting = in.readLine();*

***if*** *("hello server".equals(greeting)) {*

*out.println("hello client");*

*}*

***else*** *{*

*out.println("unrecognised greeting");*

*}*

*}*

***public******void******stop****() {*

*in.close();*

*out.close();*

*clientSocket.close();*

*serverSocket.close();*

*}*

***public******static******void******main****(String[] args) {*

*GreetServer server=****new******GreetServer****();*

*server.start(6666);*

*}*

*}*

***public******class******GreetClient*** *{*

***private*** *Socket clientSocket;*

***private*** *PrintWriter out;*

***private*** *BufferedReader in;*

***public******void******startConnection****(String ip,* ***int*** *port) {*

*clientSocket =* ***new******Socket****(ip, port);*

*out =* ***new******PrintWriter****(clientSocket.getOutputStream(), true);*

*in =* ***new******BufferedReader****(****new******InputStreamReader****(clientSocket.getInputStream()));*

*}*

***public*** *String* ***sendMessage****(String msg) {*

*out.println(msg);*

***String*** *resp = in.readLine();*

***return*** *resp;*

*}*

***public******void******stopConnection****() {*

*in.close();*

*out.close();*

*clientSocket.close();*

*}*

*}*

Тепер давайте запустимо сервер..

Потім ми надішлемо серверу привітання за допомогою модульного тесту, який підтверджує, що сервер надсилає привітання у відповідь:

*@Test*

*public void givenGreetingClient\_whenServerRespondsWhenStarted\_thenCorrect() {*

*GreetClient client = new GreetClient();*

*client.startConnection("127.0.0.1", 6666);*

*String response = client.sendMessage("hello server");*

*assertEquals("hello client", response);*

*}*

**4. Як працюють сокети**

Ми використаємо наведений вище приклад, щоб покроково пройти різні частини цього розділу.

За визначенням, сокет — це одна кінцева точка двостороннього зв’язку між двома програмами, що працюють на різних комп’ютерах у мережі. Сокет прив’язаний до номера порту, щоб транспортний рівень міг ідентифікувати програму, куди призначено надсилати дані.

**4.1. Сервер**

Зазвичай сервер працює на певному комп’ютері в мережі та має сокет, прив’язаний до певного номера порту. У нашому випадку ми будемо використовувати той самий комп’ютер, що і клієнт, і запустимо сервер на порту 6666:

***ServerSocket*** *serverSocket =* ***new******ServerSocket****(6666);*

Сервер просто чекає, прослуховуючи сокет, щоб клієнт зробив запит на з’єднання. Це відбувається на наступному етапі:

***Socket*** *clientSocket = serverSocket.accept();*

Коли код сервера зустрічає метод accept, він блокується, доки клієнт не зробить запит на підключення до нього.

Якщо все йде добре, сервер приймає підключення. Після прийняття сервер отримує новий сокет, clientSocket, прив’язаний до того самого локального порту, 6666, а також має свою віддалену кінцеву точку, налаштовану на адресу та порт клієнта.

На цьому етапі новий об’єкт Socket встановлює пряме з’єднання сервера з клієнтом. Потім ми можемо отримати доступ до вихідного та вхідного потоків, щоб писати та отримувати повідомлення клієнту та від нього відповідно:

***PrintWriter*** *out =* ***new******PrintWriter****(clientSocket.getOutputStream(), true);* ***BufferedReader*** *in =* ***new******BufferedReader****(****new******InputStreamReader****(clientSocket.getInputStream()));*

Тепер сервер здатний нескінченно обмінюватися повідомленнями з клієнтом, поки сокет не закриється своїми потоками.

Однак у нашому прикладі сервер може лише надіслати відповідь-вітання, перш ніж закрити з’єднання. Це означає, що якби ми знову запустили наш тест, сервер відмовив би у з’єднанні.

Щоб забезпечити безперервність зв’язку, нам доведеться читати вхідний потік усередині циклу while і виходити лише тоді, коли клієнт надсилає запит на завершення. Ми побачимо це в дії на наступному етапі.

Для кожного нового клієнта серверу потрібен новий сокет, повернутий викликом accept. Ми використовуємо serverSocket, щоб продовжувати прослуховувати запити на підключення, одночасно задовольняючи потреби підключених клієнтів. Ми ще не дозволили цього в нашому першому прикладі.

**4.2. Клієнт**

Клієнт повинен знати ім’я хоста або IP-адресу машини, на якій запущено сервер, і номер порту, який сервер прослуховує.

Щоб зробити запит на підключення, клієнт намагається зустрітися з сервером на машині та порту сервера:

***Socket*** *clientSocket =* ***new******Socket****("127.0.0.1", 6666);*

Клієнт також повинен ідентифікувати себе на сервері, тому він прив’язується до локального номера порту, призначеного системою, який він використовуватиме під час цього з’єднання. Ми самі цим не займаємося.

Наведений вище конструктор створює новий сокет лише тоді, коли сервер прийняв з’єднання; інакше ми отримаємо виключення у з’єднанні. Після успішного створення ми можемо отримати з нього вхідні та вихідні потоки для зв’язку з сервером:

***PrintWriter*** *out =* ***new******PrintWriter****(clientSocket.getOutputStream(), true);* ***BufferedReader*** *in =* ***new******BufferedReader****(****new******InputStreamReader****(clientSocket.getInputStream()));*

Вхідний потік клієнта з’єднаний з вихідним потоком сервера, так само як вхідний потік сервера з’єднаний з вихідним потоком клієнта.

**5. Безперервне спілкування**

Наш поточний сервер блокується, доки до нього не підключиться клієнт, а потім знову блокується, щоб прослухати повідомлення від клієнта. Після єдиного повідомлення він закриває з’єднання, оскільки ми не мали справу з безперервністю.

Таким чином, це корисно лише для запитів ping. Але уявіть, що ми хочемо реалізувати сервер чату; Безперервний зв’язок між сервером і клієнтом обов’язково буде потрібен.

Нам доведеться створити цикл while для постійного спостереження за вхідним потоком серверу для вхідних повідомлень.

Отже, давайте створимо новий сервер під назвою EchoServer.java, єдина мета якого — відтворювати будь-які повідомлення, які він отримує від клієнтів:

***public class EchoServer {***

***public void start(int port) {***

***serverSocket = new ServerSocket(port);***

***clientSocket = serverSocket.accept();***

***out = new PrintWriter(clientSocket.getOutputStream(), true);***

***in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));***

***String inputLine;***

***while ((inputLine = in.readLine()) != null) {***

***if (".".equals(inputLine)) {***

***out.println("good bye");***

***break;***

***}***

***out.println(inputLine);***

***}***

***}***

ми додали умову завершення, коли цикл while завершується, коли ми отримуємо символ крапки.

Ми запустимо EchoServer, використовуючи основний метод, так само, як ми робили для GreetServer. Цього разу ми запускаємо його на іншому порту, наприклад 4444, щоб уникнути плутанини.

EchoClient схожий на GreetClient, тому ми можемо дублювати код. Ми розділяємо їх для ясності.

В іншому тестовому класі ми створимо тест, щоб показати, що кілька запитів до EchoServer обслуговуватимуться без закриття сервером сокета. Це вірно, доки ми надсилаємо запити від одного клієнта.

Робота з кількома клієнтами — це інший випадок, який ми побачимо в наступному розділі.

Тепер давайте створимо метод налаштування для встановлення з’єднання з сервером:

*@Before*

*public void setup() {*

*client = new EchoClient();*

*client.startConnection("127.0.0.1", 4444);*

*}*

Ми також створимо метод tearDown, щоб звільнити всі наші ресурси. Це найкраща практика для кожного випадку, коли ми використовуємо мережеві ресурси:

*@After*

*public void tearDown() {*

*client.stopConnection();*

*}*

Потім ми протестуємо наш ехо-сервер за допомогою кількох запитів:

*@Test*

*public void givenClient\_whenServerEchosMessage\_thenCorrect() {*

*String resp1 = client.sendMessage("hello");*

*String resp2 = client.sendMessage("world");*

*String resp3 = client.sendMessage("!");*

*String resp4 = client.sendMessage(".");*

*assertEquals("hello", resp1);*

*assertEquals("world", resp2);*

*assertEquals("!", resp3);*

*assertEquals("good bye", resp4);*

*}*

Це покращення порівняно з початковим прикладом, де ми спілкувалися лише один раз, перш ніж сервер закрив наше з’єднання. Тепер ми надсилаємо сигнал завершення, щоб повідомити серверу про завершення сеансу.

**6. Сервер із кількома клієнтами**

Незважаючи на те, що попередній приклад був кращим у порівнянні з першим, це все одно не чудове рішення. Сервер повинен мати здатність обслуговувати багато клієнтів і багато запитів одночасно.

У цьому розділі ми розглянемо роботу з кількома клієнтами.

Ще одна особливість, яку ми побачимо тут, полягає в тому, що той самий клієнт може від’єднатися та повторно підключитися знову, не отримуючи винятку відмови у з’єднанні чи скидання з’єднання на сервері. Раніше ми не могли цього зробити.

Це означає, що наш сервер стане більш надійним і стійким до кількох запитів від кількох клієнтів.

Ми зробимо це, створивши новий сокет для кожного нового клієнта та обслуговуючи запит цього клієнта в іншому потоці. Кількість клієнтів, які обслуговуються одночасно, дорівнюватиме кількості запущених потоків.

Основний потік буде виконувати цикл while, коли він прослуховуватиме нові підключення.

Тепер давайте подивимося на це в дії. Ми створимо інший сервер під назвою EchoMultiServer.java. Всередині нього ми створимо клас потоку обробника, щоб керувати зв’язками кожного клієнта з його сокетом:

***public class EchoMultiServer {***

***private ServerSocket serverSocket;***

***public void start(int port) {***

***serverSocket = new ServerSocket(port);***

***while (true)***

***new EchoClientHandler(serverSocket.accept()).start();***

***}***

***public void stop() {***

***serverSocket.close();***

***}***

***private static class EchoClientHandler extends Thread {***

***private Socket clientSocket;***

***private PrintWriter out;***

***private BufferedReader in;***

***public EchoClientHandler(Socket socket) {***

***this.clientSocket = socket;***

***}***

***public void run() {***

***out = new PrintWriter(clientSocket.getOutputStream(), true);***

***in = new BufferedReader(***

***new InputStreamReader(clientSocket.getInputStream()));***

***String inputLine;***

***while ((inputLine = in.readLine()) != null) {***

***if (".".equals(inputLine)) {***

***out.println("bye");***

***break;***

***}***

***out.println(inputLine);***

***}***

***in.close();***

***out.close();***

***clientSocket.close();***

***}***

***}***

Зверніть увагу, що тепер ми викликаємо accept всередині циклу while. Кожного разу, коли виконується цикл while, він блокує виклик accept, доки не підключиться новий клієнт. Потім для цього клієнта створюється потік обробника EchoClientHandler.

Те, що відбувається всередині потоку, те саме, що й EchoServer, де ми обробляли лише одного клієнта. EchoMultiServer делегує цю роботу EchoClientHandler, щоб він міг продовжувати прослуховувати більше клієнтів у циклі while.

Ми все одно використовуватимемо EchoClient для тестування сервера. Цього разу ми створимо кілька клієнтів, кожен з яких надсилатиме та отримуватиме кілька повідомлень із сервера.

Давайте запустимо наш сервер, використовуючи його основний метод на порту 5555.

Для ясності ми все одно розмістимо тести в новому пакеті:

*@Test*

*public void givenClient1\_whenServerResponds\_thenCorrect() {*

*EchoClient client1 = new EchoClient();*

*client1.startConnection("127.0.0.1", 5555);*

*String msg1 = client1.sendMessage("hello");*

*String msg2 = client1.sendMessage("world");*

*String terminate = client1.sendMessage(".");*

*assertEquals(msg1, "hello");*

*assertEquals(msg2, "world");*

*assertEquals(terminate, "bye");*

*}*

*@Test*

*public void givenClient2\_whenServerResponds\_thenCorrect() {*

*EchoClient client2 = new EchoClient();*

*client2.startConnection("127.0.0.1", 5555);*

*String msg1 = client2.sendMessage("hello");*

*String msg2 = client2.sendMessage("world");*

*String terminate = client2.sendMessage(".");*

*assertEquals(msg1, "hello");*

*assertEquals(msg2, "world");*

*assertEquals(terminate, "bye");*

*}*

Ми могли б створити скільки завгодно таких тестових випадків, кожен з яких породив би нового клієнта, і сервер обслуговував би їх усіх.

**7. Висновок**

ми зосередилися на вступі до програмування сокетів через TCP/IP і написали просту програму клієнт/сервер на Java.