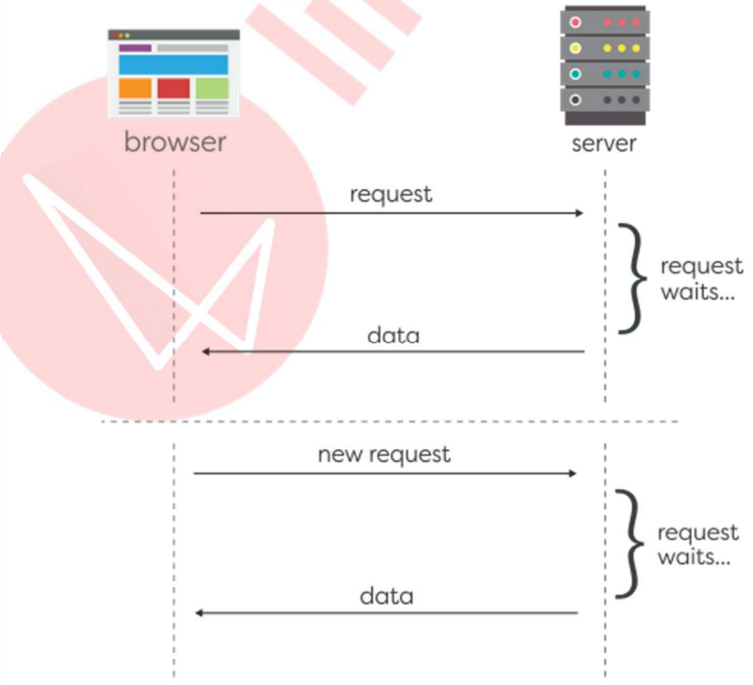
Long polling, WebSocket

U prethodnim lekcijama ilustrovano je nekoliko različitih načina za postizanje komunikacije između klijenata i servera na webu. Takvi načini su podrazumevali upotrebu XMLHttpRequest i Fetch API-a. Svi primeri koje smo videli u prethodnim lekcijama imali su jednu zajedničku osobinu – svi su se odvijali po jasno utvrđenim pravilima *request–response* HTTP modela. To praktično znači da je komunikacija između klijenata i servera svaki put započinjala upućivanjem zahteva sa klijentske strane i da je svaki klijentski zahtev bio propraćen tačno jednim serverskim odgovorom. Iako je za većinu scenarija rada sa podacima takav model u potpunosti odgovarajući, postoje situacije koje iziskuju i nešto drugačije modele komunikacije između klijenata i servera. Na primer, zamislite situaciju u kojoj bi server imao obavezu da samostalno u određenom trenutku dostavi podatke klijentima. Tako nešto ne bi bilo moguće korišćenjem pristupa koji su ilustrovani u lekcijama za nama. Takav i slični scenariji rada sa podacima zahtevaju nešto drugačiji model komunikacije, odnosno postojanje perzistentne konekcije između klijenata i servera. Različiti pristupi za postizane perzistentne konekcije između klijenata i servera biće obrađeni u lekciji pred vama.

Long polling

Najjednostavniji način za postizanje perzistentne konekcije između servera i klijenta jeste realizovanje takozvanog *long polling* modela. Za long polling se kaže da je najjednostavniji zato što se realizuje korišćenjem protokola i funkcionalnosti koje smo već upoznali u prethodnom lekcijama (HTTP protokol i XMLHttpRequest ili Fetch API za upućivanje zahteva).

Osnovni način funkcionisanja long polling modela ilustrovan je slikom 10.1.



Slika 10.1. Kako funkcioniše long polling?

Na slici 10.1. može se videti osnovni način funkcionisanja long polling načina komunikacije između servera i klijenta. Kao i kod tradicionalnog načina komunikacije, klijent inicira takvu komunikaciju slanjem zahteva. Ipak, HTTP server ne odgovora odmah, već čeka da dođe do pojave novih ili promene postojećih podataka za koje je klijent zainteresovan. Tek kada se ispune uslovi, server odgovara i klijentu isporučuje tražene podatke. Kada dobije podatke, klijent odmah iznova inicira komunikaciju sa serverom, upućivanjem novog HTTP zahteva. Proces se tako ponavlja ukrug, sve dok je klijent zainteresovan za kontinuirano dobijanje podataka od servera.

Kako se long polling realizuje u praksi?

Long polling je način komunikacije između klijenta i servera na webu. Stoga svi učesnici u komunikaciji moraju da poštuju identična pravila. Drugim rečima, long polling nije nešto što se može realizovati isključivo na klijentu, već i serverska, backend logika mora biti formulisana na poseban način kako bi bila u stanju da omogući takav vid komunikacije. Mi se u ovom kursu ne bavimo backend programiranjem, stoga neće ni biti razmatrani načini na koje se long polling realizuje na serveru. Načelno funkcionisanje backend logike se može naslutiti sa prikazanog dijagrama. Sa druge strane, ono što nas posebno zanima jeste način na koji se long polling realizuje na klijentu.

Long polling se na klijentu može realizovati korišćenjem već prikazanih Web API-a koji omogućavaju slanje HTTP zahteva. Zapravo, long polling model komunikacije nije ništa drugo do korišćenje već viđenih pristupa na nešto drugačiji način.

Evo kako može da izgleda osnovna klijentska logika koja upućuje long polling zahteve:

```
async function subscribe() {
    let response = await fetch("subscribe.php");

    if (response.status == 502) {
        // reconnect when connection timeout occur
        await subscribe();
    } else if (response.status != 200) {
        // reconnect when error occur
        await subscribe();
    } else {
        // get data
        let message = await response.text();
        // do something with data
        // call subscribe() again
        await subscribe();
    }
}

subscribe();
```

Kod ilustruje maksimalno uprošćenu logiku za slanje long polling zahteva. Možete da vidite da slanje long polling zahteva funkcioniše na principu rekurzije, odnosno funkcije koja poziva samu sebe. Slanje HTTP zahteva obavlja se korišćenjem async funkcije `subscribe()`. Zahtev se upućuje na relativnu URL adresu `subscribe.php`. Zatim se obavlja ispitivanje statusne linije i zaglavlja odgovora. Ukoliko je zahtev uspešno obrađen, konzumira se rezultat i obavlja ponovno pozivanje funkcije `subscribe()`. Funkcija `subscribe()` iznova se poziva i ukoliko dođe do neke greške prilikom obrade zahteva ili server otkaže konekciju usled predugačkog čekanja na isporuku odgovora. Na taj način se osigurava kontinuirana povezanost klijenta i servera. Pošto poziva samu sebe, za funkciju `subscribe()` se kaže da je rekurzivna.

Primer – Dobijanje podataka o broju lajkova korišćenjem long polling modela komunikacije

Na osnovu koda koji je upravo prikazan, sada će biti konstruisan realan primer long polling komunikacije, kojim će klijentska logika kontinuirano dobijati podatke o broju korisnika koji su lajkovali (*like*) jednu sliku. Tako se simulira jedna funkcionalnost koja se danas veoma često sreće na društvenim mrežama.

Kod kompletnog primera izgledaće ovako:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <title>Long Polling</title>
        <style>
            #no-of-likes {
                font-family: sans-serif;
                font-size: 32px;
                vertical-align: bottom;
                font-weight: bold;
            }
        </style>
    </head>
    <body>
        <div>
            <img src="photo-from-vacation.png" alt="me on vacation">
        </div>
        <div>
            <img src="likes.png" alt="like icon"> <span id="no-of-
likes">0</span>
        </div>
        <script>
            var likes = 0;
            async function subscribe() {
                var url = new
URL("http://examples.wizardbit.com/ajax/likes.php");
                url.searchParams.append("likes", likes);
                let response = await fetch(url);
                if (response.status == 502) {
                    // reconnect when connection timeout occur
                    await subscribe();
                } else if (response.status != 200) {
                    // reconnect when error occur
                    await subscribe();
                } else {
                    // get data
                    let data = await response.text();
                    likes = parseInt(data);
                    let likesContainer = document.getElementById('no-of-
likes');
                    likesContainer.innerHTML = likes;
                    // call subscribe() again
                    await subscribe();
```

```
                }
            }
            subscribe();
        </script>
    </body>
</html>
```

Prikazanim primerom, korišćenjem long polling modela komunikacije unutar web pregledača ažuriraju se podaci o broju lajkova jedne slike. HTML dokument stvara efekat prikazan animacijom 10.1.

Animacija 10.1. Efekat primera

Za realizaciju prikazanog primera korišćen je Fetch API. Metoda za upućivanje zahteva nazvana je `subscribe()` i njom se obavlja slanje GET HTTP zahteva na definisanu URL putanju.

Odmah na početku bitno je primetiti da se putanja ovoga puta ne kreira kao u dosadašnjim primerima – korišćenjem `string` vrednosti. Putanja je u primeru kreirana korišćenjem konstruktorske funkcije `URL()` čime je dobijen jedan objekat kojim se URL predstavlja u objektnom obliku. To je učinjeno kako bi unutar URL adrese na elegantan način mogli da budu definisani *query string* parametri:

```
var url = new URL("http://examples.wizardbit.com/ajax/likes.php");
url.searchParams.append("likes", likes);
```

Na ovaj način se serveru zajedno sa svakim zahtevom šalje i vrednost koja predstavlja broj lajkova koji frontend trenutno ima. Na osnovu takve vrednosti, backend logika utvrđuje da li frontend ima najažurnije podatke. Ukoliko su podaci ažurni, backend će čekati, odnosno neće uzvraćati HTTP odgovorom, sve dok ne dobije nove podatke o broju lajkova slike. Onog trenutka kada backend dobije nove podatke o broju lajkova, takvu vrednost će isporučiti web pregledaču kroz HTTP odgovor.

Dobijeni podaci se na frontendu obrađuju, što podrazumeva njihov ispis unutar odgovarajućeg HTML elementa, a odmah nakon toga se serveru upućuje novi HTTP zahtev. Unutar query stringa se smešta nova vrednost ukupnog broja lajkova, a HTTP server ponavlja već opisanu logiku (čeka dok ne dobije nove podatke, a kada ih dobije, isporučuje ih klijentu).

Opisana logika se kontinuirano ponavlja, čime se postiže neka vrsta perzistentne konekcije između servera i klijenta.

Upravo prikazani long polling model komunikacije može biti prihvatljiv kada se podaci koji se dostavljaju klijentu ne menjaju suviše često. S obzirom na to da se long polling zasniva na tradicionalnom Request/Response HTTP modelu, veoma česta promena podataka stvorila bi ogroman broj HTTP zahteva i odgovora, što iz ugla performansi klijentske, a prevashodno serverske logike, može stvoriti problem. U takvim situacijama, za postizanje perzistentne komunikacije između klijenata i servera moguće je koristiti još jedan, daleko napredniji pristup.

Pitanje

Long polling se zasniva na:

a) HTTP protokolu
b) LNGPL protokolu
c) WSS protokolu
d) PHP jeziku

Objašnjenje:

Long polling se realizuje korišćenjem HTTP protokola i XMLHttpRequest ili Fetch API-a za upućivanje zahteva.

Šta je WebSocket?

WebSocket je naziv koji se odnosi na dva pojma:

- WebSocket je protokol standardizovan od strane Internet Engineering Task Force (IETF) organizacije; reč je o protokolu koji definiše pravila dvosmerne komunikacije između klijenata i servera;
- WebSocket je takođe i jedan od Web API-a koje je moguće koristiti prilikom kreiranja frontend logike web aplikacija.

Za početak, biće iznete osnovne osobine WebSocket protokola, a nakon toga će biti demonstrirani pristupi za njegovo upošljavanje korišćenjem WebSocket API-a.

WebSocket protokol

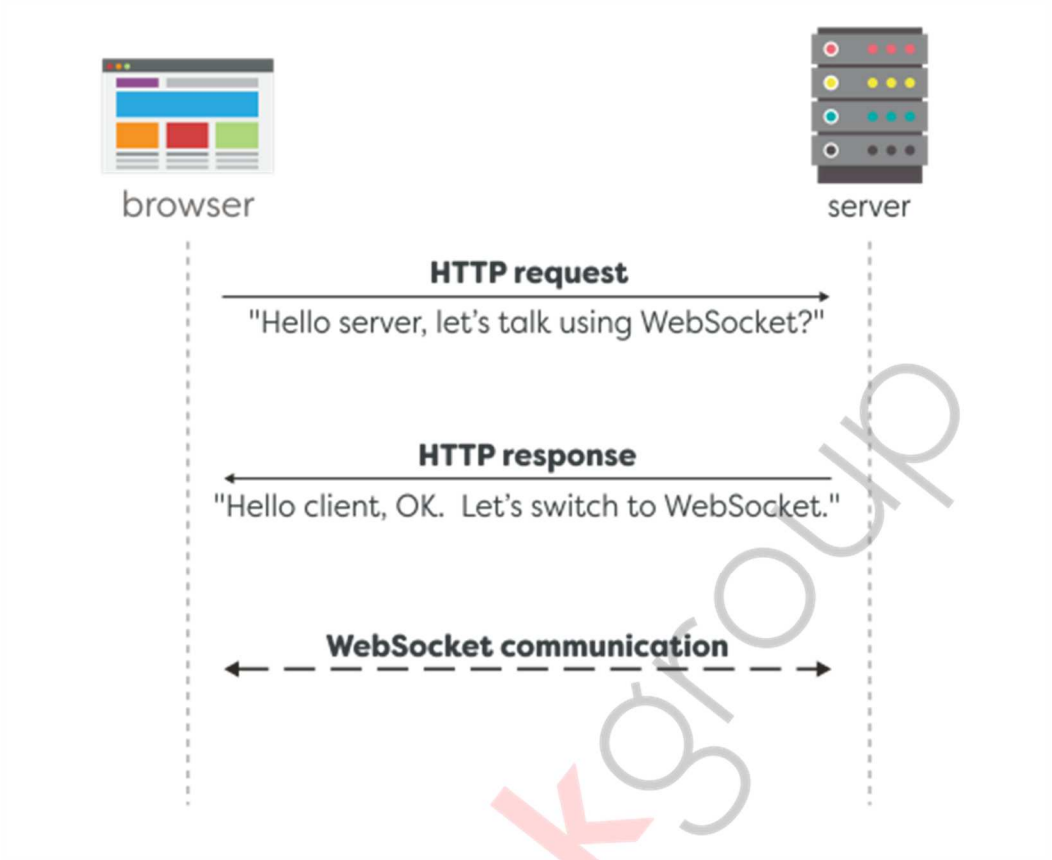
Pričom o WebSocketu prvi put izlazimo iz okvira HTTP-a, standardnog protokola na kome se zasniva kompletan web. U prethodnim lekcijama ste mogli da uvidite osnovne osobine HTTP protokola:

- HTTP protokol specijalno je dizajniran kako bi klijenti mogli da zatraže i dobiju resurse.
- HTTP obrazac komunikacije isključivo se zasniva na klijentski iniciranoj razmeni zahteva i odgovora.
- HTTP je takozvani **stateless** protokol, što praktično znači da HTTP server svaki zahtev interpretira kao potpuno izolovanu celinu; onoga trenutka kada isporuči odgovor klijentu, HTTP server podrazumevano u potpunosti *zaboravlja* na klijenta sa kojim komunicira; komunikacija između servera i klijenta drugačije se naziva sesija, pa se za HTTP može reči da je reč o protokolu kod koga HTTP server ne čuva nikakve informacije o jednoj sesiji.
- HTTP protokol omogućava takozvanu **half-duplex** komunikaciju; to praktično znači da komunikacija između klijenta i servera nije simultana – u jednom trenutku komunikacija može biti samo jednosmerna, odnosno dok jedna strana govori, druga mora da sluša.
- HTTP protokolom moguće je samo simulirati perzistentnu konekciju između servera i klijenta; to ste uostalom mogli i da vidite u prvom delu ove lekcije – long polling model komunikacije samo simulira kontinuiranu povezanost između klijenta i servera, dok se zapravo konekcija prekida i ponovno uspostavlja nakon svakog isporučenog HTTP odgovora.

Baš kao i HTTP, WebSocket u pozadini koristi TCP za ostvarivanje konekcije se serverom. Takođe, WebSocket koristi iste portove kao i HTTP: 80 i 443. Ipak, iako oba protokola u pozadini koriste TCP, samo WebSocket omogućava u potpunosti dvostranu komunikaciju korišćenjem jedne TCP konekcije. Zbog toga se kaže da WebSocket omogućava **full-duplex** komunikaciju u kojoj i server i klijent mogu simultano da razgovaraju.

Kako funkcioniše WebSocket protokol?

WebSocket konekcija između servera i klijenta na webu inicira se korišćenjem HTTP protokola. Pre uspostavljanja WebSocket konekcije, klijent serveru upućuje zahtev da nastavak komunikacije obavljaju korišćenjem WebSocket protokola. Ukoliko server prihvati takav zahtev, nastavak njihove međusobne komunikacije obavlja se poštovanjem pravila WebSocketa. Takav postupak ilustrovan je slikom 10.2.



Slika 10.2. Način iniciranja WebSocket komunikacije

Ukoliko sever prihvati zahtev klijenta da se nastavak komunikacije obavlja upotrebom WebSocket protokola, protokol prethodno ostvarene TCP konekcije se menja iz HTTP-a u WebSocket. Stoga bilo koja razmena podataka do koje dođe nakon tog trenutka biva obavljena isključivo poštovanjem WebSocket protokola. HTTP se ne koristi sve dok se WebSocket konekcija ne zatvori.

U prethodnim redovima je rečeno da iniciranje WebSocket komunikacije podrazumeva razmenu po jednog HTTP zahteva i HTTP odgovora između klijenta i servera. Takva razmena zahteva i odgovora, koja prethodi početku stvarne komunikacije, u telekomunikacijama se drugačije naziva rukovanje (*handshake*). Bitno je znati da je format klijentskog zahteva i serverskog odgovora koji prethode WebSocket komunikaciji precizno utvrđen. Tako osnovni oblik klijentskog zahteva kojim se inicira WebSocket komunikacija izgleda ovako:

```
GET /flight-data
Host: mysite.com
Origin: https://www.mysite.com
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
Sec-WebSocket-Version: 13
```

Prikazane linije ilustruju deo HTTP zahteva, odnosno njegovu statusnu liniju i zaglavlja. Zaglavlja imaju sledeće značenje:

- `Origin` – origin klijentske stranice (o origin pojmu je bilo reči u uvodnoj lekciji ovoga modula); ipak, bitno je znati da se CORS ne primenjuje na WebSocket konekcije; stoga je origin zaglavlje ne serveru moguće koristiti samo za međusobno razlikovanje klijenata;
- `Connection` – korišćenjem ovog zaglavlja sa vrednošću `upgrade` stavlja se do znanja serveru da klijent želi da promeni protokol koji se koristi za komunikaciju;
- `Upgrade` – zaglavlje kojim se definiše protokol koji klijent želi da koristi za nastavak komunikacije; definisana vrednost `websocket` znači da klijent želi da pređe na WebSocket protokol;
- `Sec-WebSocket-Key` – nasumični ključ koji web pregledač generiše iz sigurnosnih razloga;
- `Sec-WebSocket-Version` – verzija WebSocket protokola.

Ukoliko server prihvati da se nastavak komunikacije obavlja upotrebom WebSocketa, neophodno je da klijentu uputi HTTP odgovor sa sledećim zahtevom:

```
101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=
```

Iz prikazanog se može videti da prelazak na WebSocket komunikaciju server potvrđuje emitovanjem odgovora sa statusnom linijom `101 Switching Protocols`. Pored već viđenih zaglavlja `Upgrade` i `Connection`, koja imaju identično značenje kao i u okviru zahteva, HTTP odgovor poseduje još jedno, specifično zaglavlje: `Sec-WebSocket-Accept`. Vrednost ovog zaglavlja web pregledač može da koristi za proveru autentičnosti odgovora. Naime, vrednost ovog zaglavlja nastaje kodiranjem vrednosti koju je klijent dostavio serveru korišćenjem `Sec-WebSocket-Key` zaglavlja. Tako su `Sec-WebSocket-Key` i `Sec-WebSocket-Accept` osnovni mehanizam koji web pregledač i server koriste za autentifikaciju rukovanja koje prethodi početku WebSocket komunikacije.

Nakon razmene upravo ilustrovanog HTTP zahteva i odgovora, nastavak komunikacije se obavlja korišćenjem WebSocket protokola.

WebSocket API

Nakon teoretskog izlaganja o WebSocket protokolu, došao je trenutak da vidimo na koji način se takav protokol može iskoristiti za komunikaciju klijentske JavaScript logike i servera, prilikom frontend razvoja. Bitno je znati da se do sada prikazani pristupi, odnosno XMLHttpRequest i Fetch API, iako namenjeni za upućivanje HTTP zahteva, ne mogu koristiti za iniciranje WebSocket konekcije. Naime, JavaScript jeziku nije dozvoljeno da postavlja specifična HTTP zaglavlja kojima se formira zahtev za iniciranje WebSocket komunikacije.

Kreiranje WebSocket konekcije

Za upošljavanje WebSocket protokola na klijentskom delu web aplikacija programerima je na raspolaganju WebSocket API. Otvaranje WebSocket konekcije postiže se kreiranjem jednog WebSocket objekta:

```
let socket = new WebSocket("wss://echo.websocket.org");
```

Prikazanom naredbom, odnosno kreiranjem `WebSocket` objekta, automatski se inicira WebSocket komunikacija između klijenta i servera čija je adresa definisana kao parametar. To praktično znači da na ovaj način web pregledač automatski upućuje serveru uvodni HTTP zahtev, kojim od njega traži da pređu na komunikaciju korišćenjem WebSocket protokola.

Veoma je bitno da primetite da se URL adresa na koju zahtev upućuje ne formira na do sada viđene načine. Ona ovoga puta započinje odrednicom `wss://`. To je oznaka koja se u URL adresama koristi da označi WebSocket protokol. Baš kao i kod HTTP protokola, i WebSocket poseduje dve varijante – standardni WebSocket i njegovu sigurniju varijantu – WebSocket koji koristi TLS.

WebSocket događaji

Nakon kreiranja WebSocket objekta, programeru su na raspolaganju četiri događaja, koje može koristiti za dobijanje dojava o značajnim pojavama tokom komunikacije:

- `open` – emituje se kada se konekcija ostvari, odnosno kada server prihvati konekciju;
- `message` – emituje se kada dođe do prijema poruke od servera;
- `error` – emituje se kada dođe do greške u komunikaciji;
- `close` – emituje se kada se konekcija zatvori.

Pretplata na navedene događaje može da izgleda ovako:

```
let webSocket = new WebSocket("wss://echo.websocket.org");


webSocket.onopen = function (e) {
      // connection oppened
};

webSocket.onmessage = function (event) {
      console.log("Message received: " + event.data);
};

webSocket.onclose = function (event) {
if (event.wasClean) {
      console.log("Connection closed cleanly: " + event.code + " " +
event.reason);
} else {
      console.log("Connection died");
}
};

webSocket.onerror = function (error) {
console.log(error.message);
};
```

Već je rečeno da se prvom naredbom inicira WebSocket konekcija, tako što klijent serveru upućuje specijalan HTTP zahtev za ažuriranje načina komunikacije, tj. prelazak na WebSocket protokol. Ukoliko tokom procesa konekcije dođe do greške, emituje se `error` događaj, pa se samim tim aktivira i `onerror` funkcija za obradu događaja. Do aktiviranja `onerror` funkcije dolazi i prilikom pojave bilo koje greške tokom komunikacije.

Pojava greške tokom konektovanja ili tokom komunikacije rezultuje zatvaranjem konekcije, što je propraćeno aktiviranjem `close` događaja `onclose` funkcije. Bitno je razumeti da se `onclose` funkcija aktivira nakon `onerror` funkcije. Funkciji `onclose` isporučuje se objekat tipa **CloseEvent**. Unutar njega su upakovane informacije o razlozima za zatvaranje konekcije:

- `wasClean` – svojstvo `boolean` tipa koje govori da li je konekcija zatvorena poštovanjem pravila WebSocket protokola; naime, baš kao što postoje jasno utvrđena pravila za započinjanje WebSocket komunikacije, takav protokol definiše i poseban format serverske poruke kojom se klijentu govori da je konekcija zatvorena kao posledica normalnog prekida komunikacije; tada ovo svojstvo ima vrednost `true`; sa druge strane, ukoliko se konekcija nepredviđeno ili nasilno zatvori, web pregledač od servera ne dobija upravo spomenuti specifični format odgovora, pa ovo svojstvo dobija vrednost `false`;
    - `code` – svojstvo koje sadrži statusni kod koji ukazuje na razlog zatvaranja konekcije:
    - 1000 – Normal Closure
    - 1001 – Going Away
    - 1002 – Protocol Error
    - 1003 – Unsupported Data
    - 1006 – Abnormal Closure
    - 1009 – Message Too Big
    - 1013 – Try Again Later
    - 1014 – Bad Gateway
- `reason` – svojstvo koje sadrži poruku sa razlogom zatvaranje konekcije; vrednost svojstva je `string` tipa.

Slanje podataka serveru

Nakon uspešnog ostvarivanja konekcije, klijent serveru može da pošalje podatke korišćenjem metode `send()`:

```
webSocket.onopen = function (e) {
      webSocket.send("Hello from WebSocket client!");
};
```

U prikazanom isečku koda, poziv metode `send()` je smeštan unutar `onopen` funkcije, koja se aktivira kada se WebSocket konekcija uspešno ostvari. Ovo je veoma važan detalj, zato što pozivanje metode `send()` izvan ove funkcije ne garantuje da će takva poruka zaista i biti dostavljena serveru. Stoga je neophodno čekati da se konekcija ostvari, pa tek onda inicirati slanje poruke serveru, korišćenjem metode `send()`.

Prijem podataka od servera

Prijem podataka od servera obrađuje se korišćenjem `message` događaja, čija pretplata se može obaviti korišćenjem `onmessage` svojstva:

```
webSocket.onmessage = function (event) {
      console.log("Message received: " + event.data);
};
```

Funkcija za obradu `message` događaja dobija jedan parametar `MessageEvent` tipa. Unutar njegovog svojstva `data` upakovani su podaci koje je server poslao klijentu. Podaci se isporučuju kao UTF-8 tekst.

Zatvaranje konekcije

Iniciranje zatvaranja WebSocket konekcije na klijentu se može postići pozivanjem metode `close()`:

```
webSocket.close(1000, "Work complete");
```

Metodi `close()` se prosleđuju dva parametra. Prvi parametar predstavlja statusni kod sa razlogom za prekidanje konekcije. Nešto ranije su navedeni najčešće korišćeni takvi kodovi i njihovo značenje. Na primer, kod 1000 iz primera predstavlja normalno zatvaranje konekcije (Normal Closure). Drugi parametar odnosi se na razlog (`reason`) zatvaranja konekcije. Reč je o tekstualnoj vrednosti kojom se serveru detaljnije govori zbog čega klijent zatvara konekciju.

Praćenje stanja WebSocket konekcije

WebSocket API omogućava praćenje stanja u kome se nalazi konekcija korišćenjem svojstva `readyState`. Praktično, pored već prikazanih događaja koji se emituju u odgovarajućem trenutku životnog toka WebSocket konekcije, jedini način na koji klijent može da dobije status konekcije izvan funkcija za obradu takvih događaja jeste upotreba `readyState` svojstva.

Svojstvo `readyState` može imati sledeće vrednosti:

- `0` ili konstanta `CONNECTING` – proces uspostavljanja konekcije je u toku, ali konekcija još nije ostvarena;
- `1` ili konstanta `OPEN` – konekcija je ostvarena, odnosno otvorena;
- `2` ili konstanta `CLOSING` – konekcija se zatvara;
- `3` ili konstanta `CLOSED` – konekcija je zatvorena.

U narednim primerima biće praktično iskorišćeni svi upravo prikazani pristupi za ostvarivanje komunikacije između klijenta i servera korišćenjem WebSocket protokola.

Primer 1 – Testiranje funkcionisanja WebSocket protokola korišćenjem echo servera

Prvi realan primer korišćenja WebSocket protokola ilustrovaće ostvarivanje komunikacije sa WebSocket test serverom koji je javno dostupan na sledećoj URL adresi:

wss://javascript.info/article/websocket/demo/hello

To je echo server koji javno hostuje kompanija Kaazing. To praktično znači da će sve ono što se uputi serveru biti vraćeno nazad klijentu kao njegov odgovor. Kompletna klijentska logika izgleda ovako:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <title>WebSocket Echo Demo</title>
        <style>
            div,
            input,
            textarea {
                width: 360px;
            }
            button {
                margin-top: 4px;
            }
            #status-container {
                margin-top: 16px;
            }
        </style>
    </head>
    <body>
        <div>
            <input id="message" autocomplete="off">
        </div>
        <div>
            <button id="connect">Connect</button>
            <button id="disconnect">Disconnect</button>
            <button id="send">Send</button>
        </div>
        <div id="status-container">
            <textarea name="status" id="status" cols="30" rows="10"
readonly></textarea>
        </div>
        <script>
            let sendButton = document.getElementById("send");
            let connectButton = document.getElementById("connect");
            let disconnectButton = document.getElementById("disconnect");
            let messageTextArea = document.getElementById("message");
            let statusTextArea = document.getElementById("status");
            let webSocket;
            connectButton.addEventListener("click", function () {
                webSocket = new WebSocket("wss://echo.websocket.org");
                webSocket.onopen = function (e) {
```

```
                    statusTextArea.value += "Connection established!\n";
                };
                webSocket.onmessage = function (event) {
                    statusTextArea.value += "Message received: " +
event.data + "\n";
                };
                webSocket.onclose = function (event) {
                    statusTextArea.value += "Connection closed: " +
event.code + " " + event.reason + "\n";
                };
                webSocket.onerror = function (error) {
                    statusTextArea.value += "Error : " + error.message +
"\n";
                };
            });
            disconnectButton.addEventListener("click", function () {
                if (webSocket !== undefined) {
                    webSocket.close(1000, "Work complete");
                }
            });
            sendButton.addEventListener("click", function () {
                if (messageTextArea.value === '')
                    return;
                if (webSocket === undefined || webSocket.readyState !==
WebSocket.OPEN) {
                    statusTextArea.value += "Please open the connection
first.\n";
                    return;
                }
                webSocket.send(messageTextArea.value);
                statusTextArea.value += "Message sent: " +
messageTextArea.value + "\n";
                messageTextArea.value = '';
            });
        </script>
    </body>
</html>
```

Efekat prikazanog primera ilustrovan je animacijom 10.2.

Animacija 10.2. Efekat komunikacije sa echo serverom korišćenjem WebSocket protokola

Za realizaciju prikazanog primera iskorišćeni su pristupi koji su ilustrovani u dosadašnjem toku ove lekcije. Konekcija se ostvaruje klikom na dugme *Connect*. Nakon uspostavljanja konekcije, unutar `textarea` elementa se ispisuje odgovarajuća poruka. Tek nakon ostvarivanja konekcije, moguće je uputiti poruku serveru, kucanjem poruke unutar `input` elementa i klikom na dugme *Send*. Poruka koju klijent pošalje odmah se emituje nazad od strane servera i ispisuje unutar `textarea` elementa. Zatvaranje konekcije se obavlja klikom na *Disconnect* dugme.

Primer 2 – Kreiranje chat aplikacije

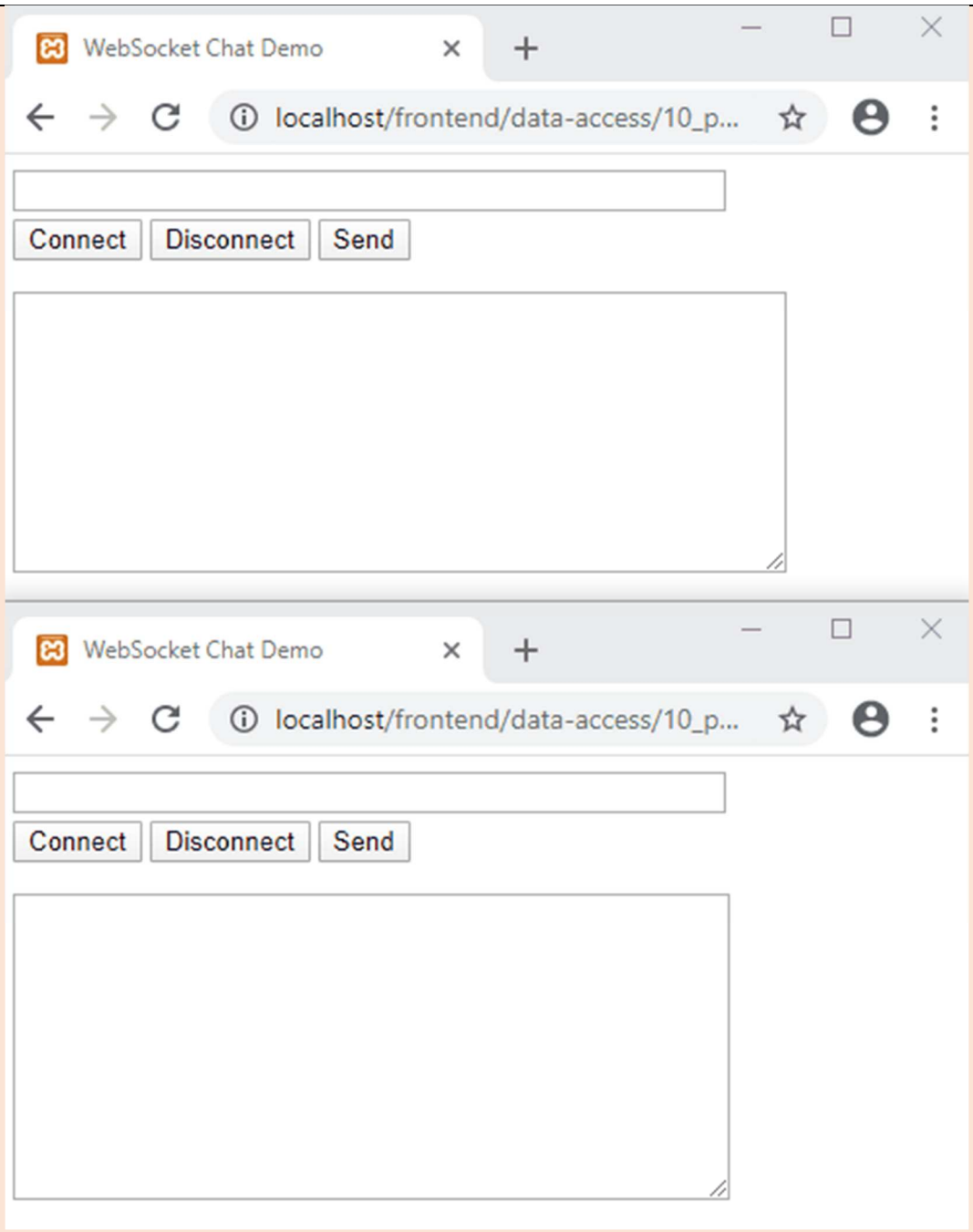
Korišćenjem identične logike, moguće je realizovati još jedan primer. Reč je o primeru chat aplikacije koja omogućava da se poruka jednog klijenta uputi nekom drugom klijentu. Naravno, komunikacija između klijenata će se obavljati posredstvom WebSocket servera. Tako će primer ilustrovati jednu od najznačajnijih mogućnosti WebSocket protokola. Reč je o mogućnosti servera da samostalno uputi poruku klijentu, bez potrebe da klijent to od njega eksplicitno zatraži. Ukoliko se sećate, tako nešto nije moguće uraditi korišćenjem HTTP protokola.

Kao što je rečeno, kod za realizaciju primera identičan je kodu upravo prikazanog primera za upošljavanje echo servera. Jedina razlika će biti URL adresa na kojoj se nalazi WebSocket server:

```
webSocket = new
WebSocket("wss://connect.websocket.in/v3/1?apiKey=3stnCQnpqYQwDpS72574wxyeuQ
ahcnvkYyOmtsr7pQMgQgjLcvH96xW5KsIU");
```

Reč je o naredbi koju je potrebno da izmenite unutar funkcije za obradu događaja klika na dugme *Connect*.

Efekat Chat aplikacije realizovane korišćenjem WebSocket protokola ilustruje animacija 10.3.



Animacija 10.3. Chat sistem kreiran korišćenjem WebSocket protokola

Možete da primetite da testiranje primera podrazumeva korišćenje dva web pregledača. Naime, vi možete otvoriti i veći broj web pregledača, zato što primer omogućava komunikaciju proizvoljnog broja klijenata. Drugim rečima, poruka koja se serveru upućuje od jednog klijenta preusmerava se svim ostalim konektovanim klijentima, isključujući klijenta koji je takvu poruku uputio.

Rezime

- Najjednostavniji način za postizanje perzistentne konekcije između servera i klijenta jeste realizovanje takozvanog long polling modela.
- Long polling se realizuje korišćenjem HTTP protokola i XMLHttpRequest ili Fetch API-a za upućivanje zahteva.
- Kod long polling modela, klijent inicira komunikaciju slanjem zahteva, ali HTTP server ne odgovora odmah, već čeka da dođe do pojave novih ili promene postojećih podataka za koje je klijent zainteresovan; nakon dostave podataka, klijent odmah upućuje novi zahtev i operacije se ponavljaju sve dok je klijent zainteresovan za dobijanje kontinuiranih podataka.
- Long polling model komunikacije može biti prihvatljiv kada se podaci koji se dostavljaju klijentu ne menjaju suviše često.
- WebSocket je protokol koji definiše pravila dvosmerne komunikacije između klijenata i servera.
- WebSocket je takođe i jedan od Web API-a koje je moguće koristiti prilikom kreiranja frontend logike web aplikacija.
- HTTP obrazac komunikacije isključivo se zasniva na klijentski iniciranoj razmeni zahteva i odgovora; HTTP protokol omogućava takozvanu half-duplex komunikaciju.
- WebSocket omogućava full-duplex komunikaciju u kojoj i server i klijent mogu simultano da razgovaraju.
- Otvaranje WebSocket konekcije postiže se kreiranjem jednog WebSocket objekta.