Laboratorio 2

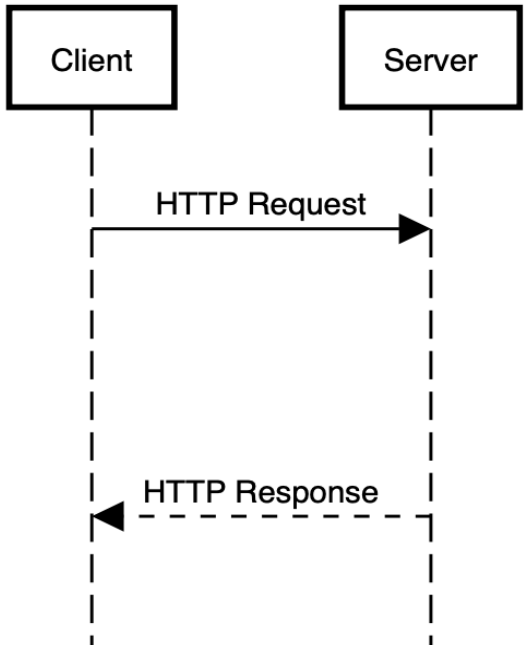
Recap

- **Node.js**: Runtime environment per JavaScript
  - Creazione di moduli
  - Gestore di pacchetti
- **Protocollo HTTP**
  - Server HTTP
- **Express**

Problema

Le interazioni con protocollo HTTP viste finora comprendono un **client** che inizia una richiesta verso il **server**, che risponde con le informazioni necessarie.

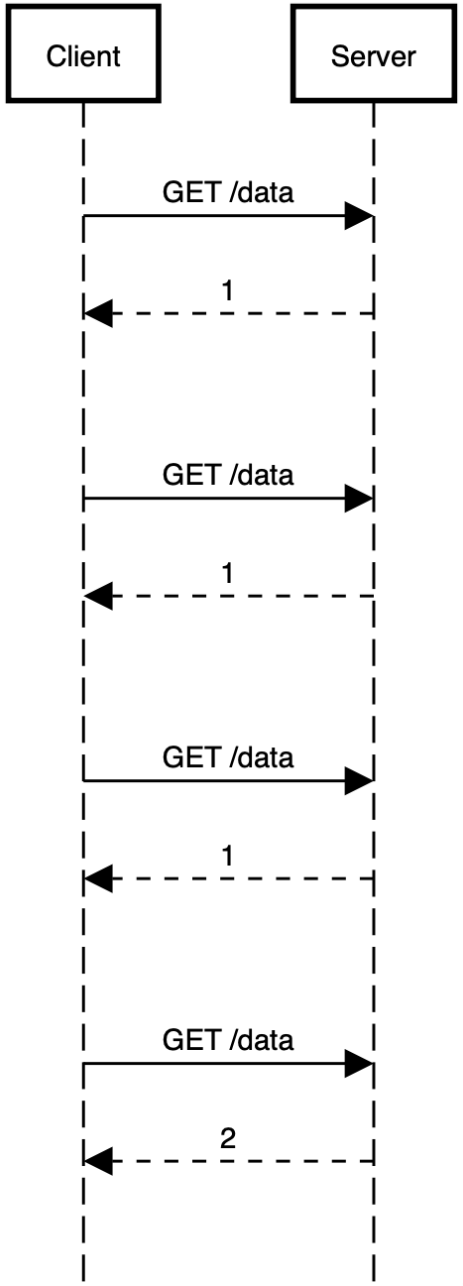
Come si possono ricevere, dal server, nuovi dati appena disponibili?



Polling (esempio 0)

Il **Polling** è un processo in cui un dispositivo viene *controllato ripetutamente* per verificarne la disponibilità e, in caso contrario, si torna ad un'altra attività.

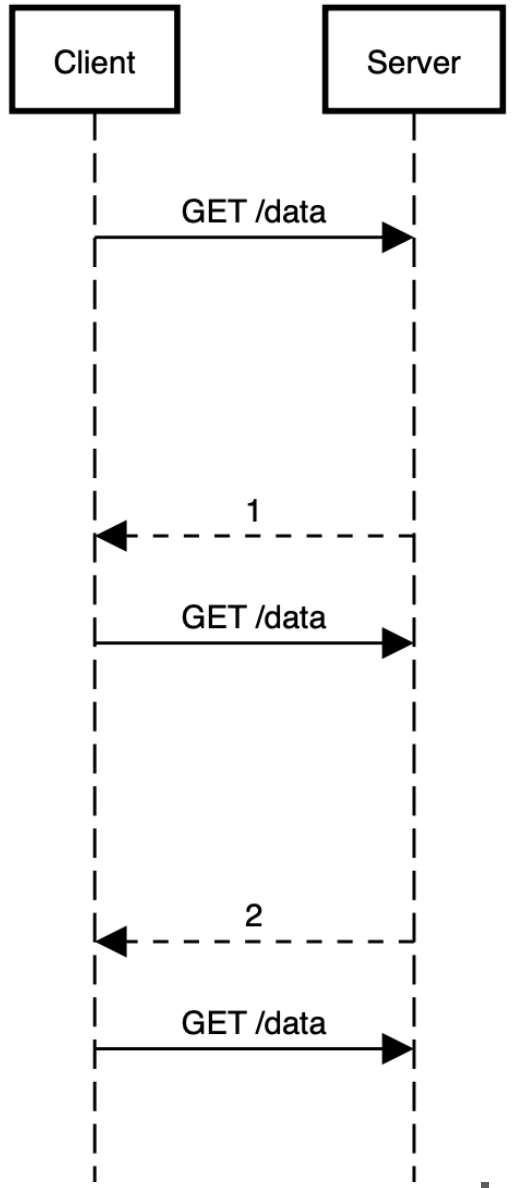
Nel contesto di **HTTP**, il polling si riferisce ad una tecnica in cui un client invia periodicamente richieste al server per verificare se ci sono aggiornamenti o nuovi dati disponibili.



Long Polling (esempio 1)

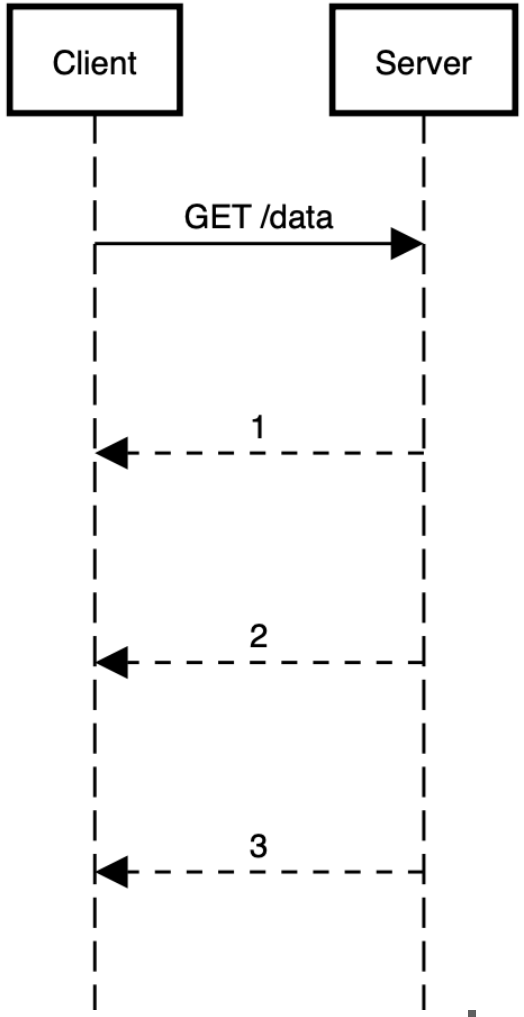
Il **Long Polling** è una tecnica che permette di ridurre il numero di richieste fatte dal client per ottenere nuovi dati dal server.

1. Il client inizia una richiesta verso il server.
2. Il server non risponde finché nuovi dati sono presenti
3. Il client inizia una nuova richiesta



Server Sent Events (esempio 2)

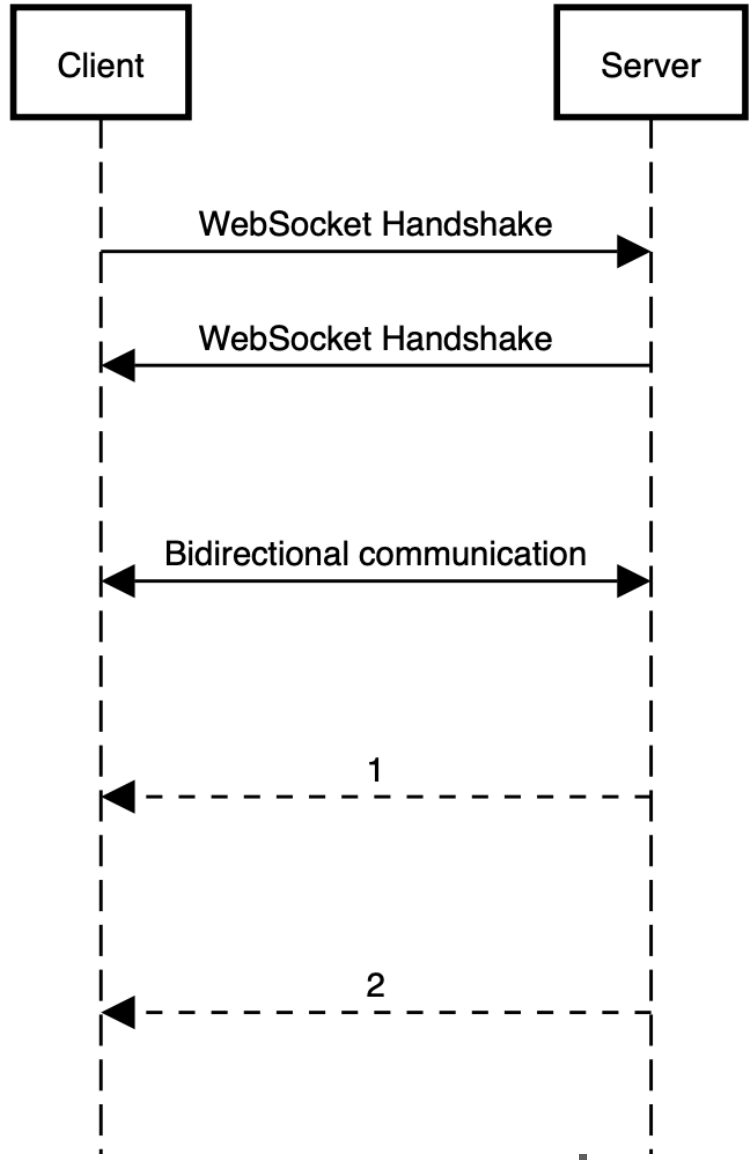
I Server-Sent Events (**SSE**) permettono al server di inviare aggiornamenti automatici al client tramite una connessione HTTP *unidirezionale*, ideale per notifiche in tempo reale senza bisogno di richieste continue dal client.



WebSocket (esempio 3)

Le **WebSocket** forniscono una connessione *bidirezionale* persistente tra client e server, consentendo comunicazioni in tempo reale con bassa latenza, ideale per applicazioni interattive come chat o giochi online.

Tramite il protocollo HTTP è possibile instaurare la socket per la comunicazione.



Riassunto

Tecnica	Tipo di Connessione	Direzione della Comunicazione	Uso tipico
Polling	Richieste HTTP	Request-response	Aggiornamenti poco frequenti
Long Polling	Richieste HTTP prolungate	Request-response	Aggiornamenti in tempo reale, ma non frequenti
Server-Sent Events	Connessione HTTP persistente	Unidirezionale (server-client)	Aggiornamenti in tempo reale, streaming di dati leggeri
WebSocket	Connessione bidirezionale persistente	Bidirezionale	Applicazioni in tempo reale (chat, giochi, etc.)

Socket.io

- **Socket.io**: libreria per la comunicazione a bassa latenza, bidirezionale, e ad eventi tra client e server.

Suggerimento: ricarica automatica

Durante la fase di sviluppo, capita spesso di dover terminare il processo in esecuzione e rieseguirlo per testare le modifiche apportate.

Esisto strumenti che aiutano i programmatori ad eseguire riavvii automatici alla modifica dei file contenuti in una directory.

Nodemon è uno di essi ed aiuta a sviluppare applicazioni basate su Node.js riavviando l'istanza quando vengono rilevate modifiche ai file presenti nella directory.

Installazione globale

```
npm install -g nodemon
nodemon <nome-file>
```

Installazione locale (dev)

```
npm install --save-dev nodemon
npx nodemon <nome-file>
```

Esempio 4: Socket.io

- Creare una **chat** in cui sia possibile scambiare messaggi tra tutti gli utenti collegati.

Esempio 4: Socket.io

- Creare un nuovo progetto, in una nuova cartella, con il comando
  - `npm init`
- Aggiungere le dipendenze a express e a socket.io

```
npm install express
npm install socket.io
```

- Creare l'entry point della vostra applicazione (app.js o index.js)

Esempio 4: Socket.io

1. Lato server, aggiungere le dipendenze a express, socket.io e http
2. Creare i rispettivi server
3. Gestire la richiesta alla root, restituendo l'index.html fornito
4. Avviare il server

```
// 1.
const express = require('express');
const { Server } = require('socket.io');
const { createServer } = require('http');

// 2.
const app = express();
const server = createServer(app);
const io = new Server(server);
```

```
// 3.
app.get('/', (req, res) => {
    res.sendFile(join(__dirname, 'public', 'index.html'));
});

// 4.
server.listen(3000, () => {
    console.log('server running at http://localhost:3000');
});
```

Esempio 4: Socket.io

- Ogni volta che un utente si connette (un client chiama la funzione `io()`) viene generato un evento **connection**. Quando un utente si disconnette, viene generato un evento **disconnect**. È possibile gestirli nel seguente modo

```
io.on('connection', (socket)=>{
        console.log('a user connected');
        socket.on('disconnect', ()=>{
                console.log('user disconnected');
        });
});
```

Esempio 4: Socket.io

- Lato client, nel tag script creare la socket con il comando
  - **const *socket*** = io();
  - Lanciando il server e visitando la root, deve essere visualizzato il messaggio di connessione.
- Alla chiusura della tab, invece, deve essere visualizzato il messaggio di disconnessione.

Esempio 4: Socket.io

- Lato client, cliccando sul bottone di submit, il messaggio deve essere inviato al server.
-

```
$('form').submit((e)=>{
e.preventDefault();
      socket.emit('chat message', $('#m').val());
      $('#m').val('');
      return false;
    });
```

Esempio 4: Socket.io

- Lato server, intercettare l'evento "chat message" (all'interno della callback di connessione)
- Propagare il messaggio agli altri client.
-

```
socket.on('chat message', (msg)=>{
  io.emit('chat message', msg);
});
```

Esempio 4: Socket.io

- Lato client, intercettare l'evento "chat message" e aggiungere il messaggio alla lista dei messaggi

```
socket.on('chat message', (msg)=>{
  $('#messages').append($('<li>').text(msg));
});
```

- Verificare il funzionamento.

Esercizio 1: Socket.io

- Implementare le seguenti funzionalità
  - Mandare messaggio in broadcast quando qualcuno si connette/disconnette
  - Mantenere la storia dei messaggi, servita a `GET /messages`
  - Aggiungere i nickname
  - Visualizzare "utente sta scrivendo.. "
  - Visualizzare lista delle persone online

Esercizio 2

- Creazione un webserver con Express che esponga delle API RESTful per gestire un set di film.
- Operazioni consentite dalle API:
  - Ottenere informazioni su tutti i film disponibili
  - Ottenere informazioni su un film particolare
  - Aggiungere un film alla collezione
  - Modificare un film
  - Cancellare un film

Esercizio 2

- Nessun database per la persistenza
- Il file **JSON** fornito contiene due film che saranno la base del set.
- Gestione dei dati con un **array**.
- Il file `movies.json` contiene i film già presenti nel nostra collezione
- Al riavvio del server verrà riavviato le modifiche effettuate andranno perse.

Esercizio 2

- Gestire solo i casi in cui per lettura, modifica e cancellazione viene specificato un id esistente.
- Si assume che i dati inseriti siano sempre corretti.
- Testare le API con **Postman** o altri tool simili

Attenzione!!! Se è non già installato, va installato!
https://www.getpostman.com/downloads/

Esercizio 2

- Spostarsi dentro la cartella `exercise-02` con lo scheletro dell'esercizio e installare le dipendenze
  - `npm install`

Esercizio 2: buone prassi

La struttura, dentro `src`, contiene due sottocartelle

- `routes`: dovrà contenere un file js che gestisce tutte le rotte dell'applicazione
- `controllers`: dovrà contenere un file js che gestisce la logica di ogni chiamata (potete inserire il file `movies.json` qui).

Questo approccio permette di separare la gestione delle rotte con la logica applicativa

Flusso di chiamate

- `index.js` utilizza il router definito in routes

```
const myRouter = require('./src/routes/myRouter');
app.use('/example', myRouter);
```

- Il router (`src/routes/myRouter`) utilizza la logica definita nei controller

```
const myController = require('../controllers/myController');
router.route('/').get(myController.helloWorld);
```

- Il controller (`src/controllers/myController`) implementa la logica

```
exports.helloWorld = (req, res) => { res.send('Hello World!'); }
```

Esempio rotta

- File moviesRoutes.js

```
router.route('/movies’)
    .get( (req,res) => {
       moviesController.readAllMovies(req,res)
     });
```

Esempio controller

- File moviesControllers.js

```
const data = require('./movies.json’);
exports.readAllMovies = (req, res)=> {
      res.json(data);
};
```

Read movie

- Implementare
  - `readMovie = (req, res)=> { }`
- Gestire la rotta
- router.route('/:id')
  .get((req,res)=>{
  movieControllers.readMovie(req,res);})

Create movie

- Implementare
  - `createMovie = (req, res)=> { }`
- Gestire la rotta
- .post((req,res)=>{ movieControllers.createMovie(req,res);})

Update movie

- Implementare
  - `updateMovie = (req, res)=> { }`
- Gestire la rotta
- .put((req,res)=>{
  movieControllers.updateMovie(req,res);})

Delete movie

- Implementare
  - `deleteMovie = (req, res)=> { }`
- Gestire la rotta
- .delete((req,res)=>{
  movieControllers.deleteMovie(req,res);})

Routes: risultato finale

```
const express = require('express');
const movieControllers = require('../controllers/moviesControllers')

const router = express.Router();

router.get('/',(req,res,next)=>{
   movieControllers.readAllMovies(req,res);
})
   .post((req,res)=>{
      movieControllers.createMovie(req,res);})

router.route('/:id')
   .get((req,res)=>{
   movieControllers.readMovie(req,res);})
   .delete((req,res)=>{
      movieControllers.deleteMovie(req,res);})
   .put((req,res)=>{
      movieControllers.updateMovie(req,res);})

module.exports = router;
```

Route

*"You can create chainable route handlers for a route path by using app.**route**().*
Because the path is specified at a single location, creating modular routes is helpful, as is reducing redundancy and typos."

ExpressJS.com

Link utili

- http://expressjs.com/
- https://socket.io/