

What we're learning

1. Stateful vs Stateless Backends
2. State management in a Backend app
3. Singleton Pattern
4. Pub Subs + Singleton pattern

Stateful vs Stateless Backends

Common interview question

Stateless servers

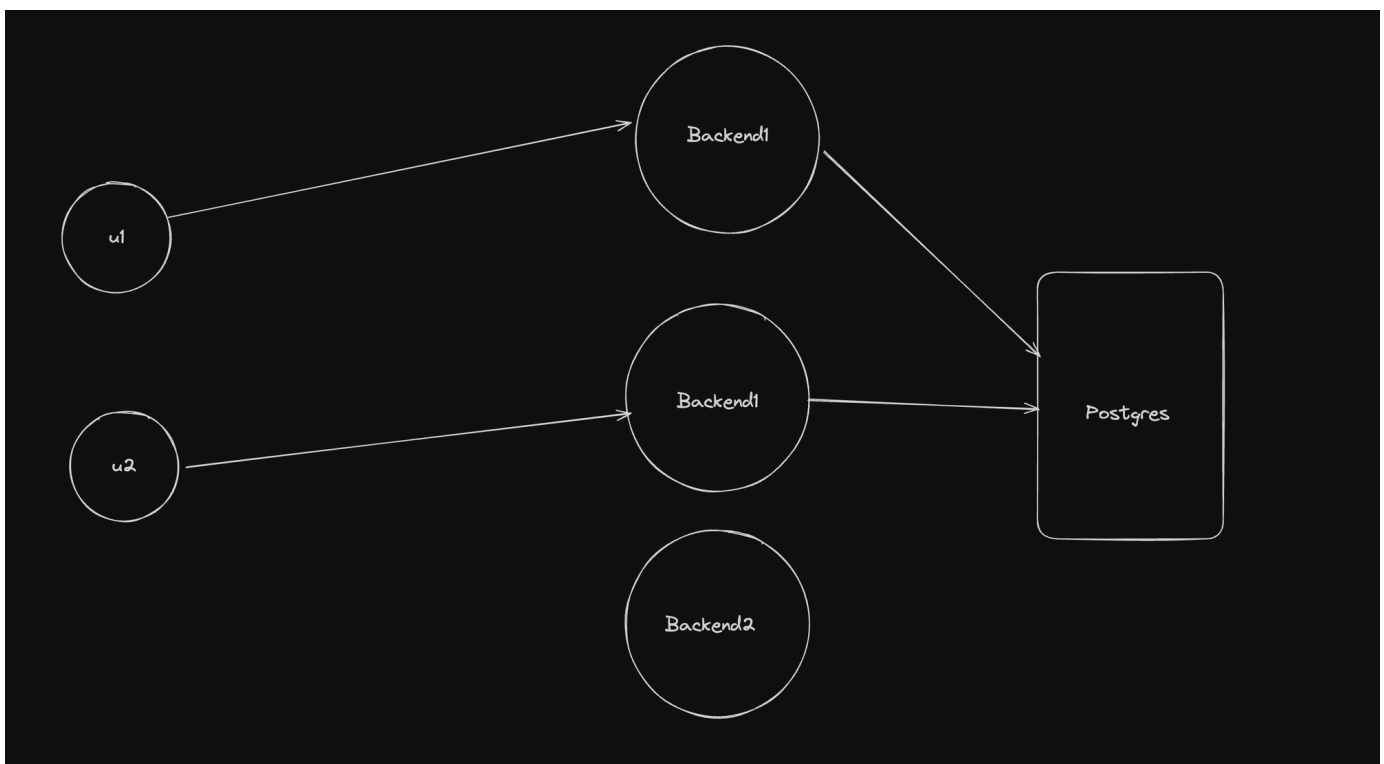
Usually when you write HTTP servers, they don't hold any **state**

This means, they don't have any in memory variables that they use

They usually rely on the **database** for **state**

Advantages

1. Users can connect to a random server, there is no need of stickiness
2. Can autoscale up and down easily and simply decide where to route traffic based on CPU usage.



Stateful servers

A lot of times, you make servers hold **state**

Good examples of this are

1. Creating an **in memory cache** -

<https://github.com/code100x/cms/blob/e905c71eacf9d99f68db802b24b7b3a924ae27f1/src/db/Cache.ts#L3>

2. Storing the **state** of a Game in case of a realtime game -

<https://github.com/code100x/chess/blob/main/apps/ws/src/Game.ts#L41-L47>

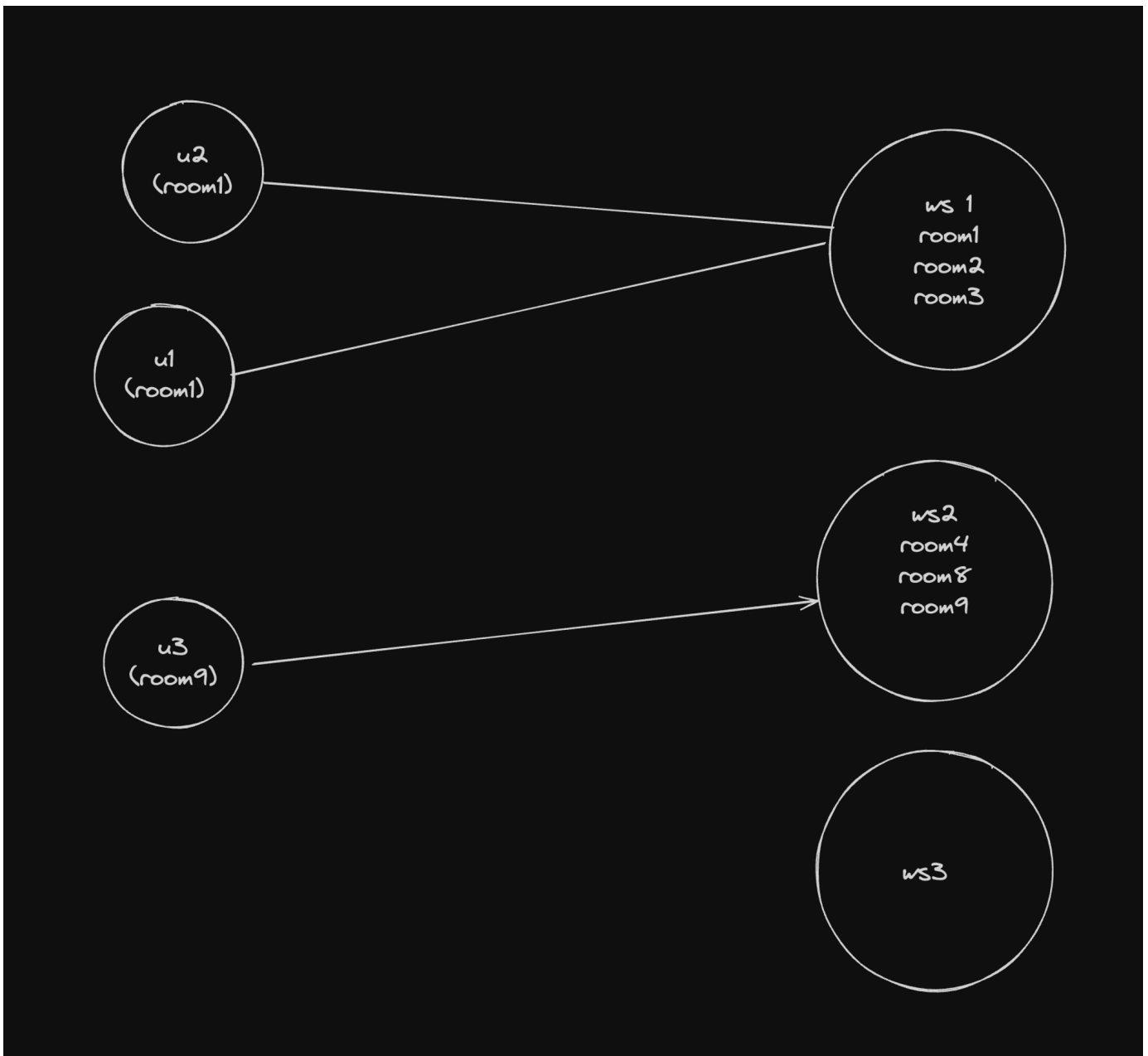
3. Storing a list of 10 most latest chats in memory for a **chat** application

In case **1**, there is no need of **stickiness**

In case of **2** and **3**, there is need of **stickiness**

Stickiness

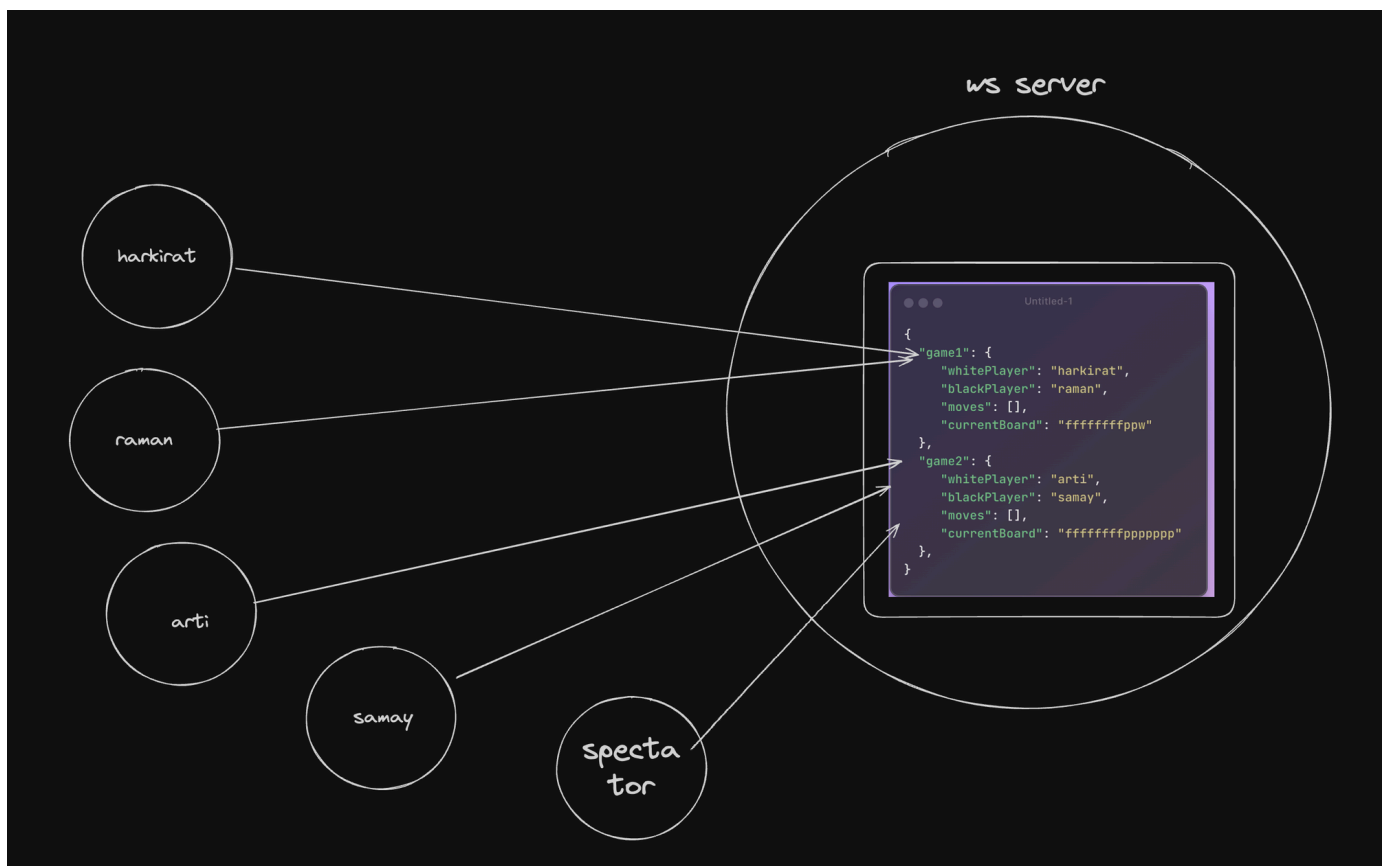
Making sure that the user who is interested in a **specific** room, gets connected to a **specific** server.



Good question to ask at this point is - How to store state in a JS project?

State in JS process

How/where can you store **state** in a Javascript process



This state might be used by multiple files, not just one, so the following approach might not work

Lets try the following -

- index.ts - pushes to games array

```
import { games } from "../store";
import { startLogger } from "../logger";
```

Copy

```
startLogger();

setInterval(() => {
  games.push({
    "whitePlayer": "harkirat",
    "blackPlayer": "jaskirat",
    moves: []
  })
}, 5000)
```

- logger.ts - uses the games array

```
import { games } from "./store";

export function startLogger() {
  setInterval(() => {
    console.log(games);
  }, 4000)
}
```

- store.ts - Exports the game array

```
interface Game {
  whitePlayer: string;
  blackPlayer: string;
  moves: string[];
}

export const games: Game[] = [];
```

This will work, but a lot of times you need to attach functionality to **state** as well.

Let's see how can we create a class called **GameManager** and expose some functions on it that can be called by files using it



There are other ways of storing state in a TS project as well, redux being a popular one. Yes, you can use redux in the backend as well

Classes and Singleton pattern

Let's create a class that

1. Stores games
2. Exposes functions that let you mutate the state

```
interface Game {  
  id: string;  
  whitePlayer: string;  
  blackPlayer: string;  
  moves: string[];  
}  
  
export class GameManager {  
  private games: Game[] = [];  
  
  public addGame(game: Game) {  
    this.games.push(game);  
  }  
  
  public getGames() {  
    return this.games;  
  }  
  
  // e5e7  
  public addMove(gameId: string, move: string) {  
    const game = this.games.find(game => game.id === gameId);  
    if (game) {  
      game.moves.push(move);  
    }  
  }  
  
  public logState() {  
    console.log(this.games);  
  }  
}
```

Copy

Bad approach

Create separate instance of `GameManager` in every file that needs it

▼ GameManager.ts

```
interface Game {
  id: string;
  whitePlayer: string;
  blackPlayer: string;
  moves: string[];
}

export class GameManager {
  private games: Game[] = [];

  public addGame(game: Game) {
    this.games.push(game);
  }

  public getGames() {
    return this.games;
  }

  // e5e7
  public addMove(gameId: string, move: string) {
    const game = this.games.find(game => game.id === gameId);
    if (game) {
      game.moves.push(move);
    }
  }
}
```

[Copy](#)

▼ logger.ts

```
import { GameManager } from "../GameManager";

const gameManager = new GameManager();

export function startLogger() {
  setInterval(() => {
    gameManager.logState();
  }, 4000)
}
```

[Copy](#)

▼ index.ts

```
import { GameManager } from "../GameManager";
import { startLogger } from "../logger";

const gameManager = new GameManager();

startLogger();

setInterval(() => {
  gameManager.addGame({
    id: Math.random().toString(),
    "whitePlayer": "harkirat",
    "blackPlayer": "jaskirat",
    moves: []
  })
}, 5000)
```

Copy

Slightly Better approach

Export a single instance of `gameManager` from `GameManager.ts` and use it everywhere

Even better approach - Singleton Pattern

Completely prevent any developer from ever creating a new instance of the `GameManager` class

Static attributes -

In JavaScript, the keyword `static` is used in classes to declare static methods or static properties. Static methods and properties belong to the class itself, rather than to any specific instance of the class. Here's a breakdown of what this means

▼ Example of a class with static attributes

```
class Example {
  static count = 0;

  constructor() {
    Example.count++; // Increment the static property using the c
  }
}
```

Copy


```
let ex1 = new Example();  
let ex2 = new Example();  
console.log(Example.count); // Outputs: 2
```

Copy

```
interface Game {  
  id: string;  
  whitePlayer: string;  
  blackPlayer: string;  
  moves: string[];  
}  
  
export class GameManager {  
  private static instance: GameManager; // Create a static instance of  
  private games: Game[] = [];  
  
  private constructor() {  
    // Private constructor ensures that a new instance cannot be cre  
  }  
  
  public static getInstance(): GameManager {  
    if (!GameManager.instance) {  
      GameManager.instance = new GameManager();  
    }  
    return GameManager.instance;  
  }  
  // ... other methods  
}  
  
// Usage GameManager.getInstance().addGame()
```

▼ GameManager.ts

Copy

```
interface Game {  
  id: string;  
  whitePlayer: string;  
  blackPlayer: string;  
  moves: string[];  
}  
  
export class GameManager {
```

```
private static instance: GameManager; // Create a static instance
private games: Game[] = [];

private constructor() {
    // Private constructor ensures that a new instance cannot be c
}

public static getInstance(): GameManager {
    if (!GameManager.instance) {
        GameManager.instance = new GameManager();
    }
    return GameManager.instance;
}

public addGame(game: Game) {
    this.games.push(game);
}

public getGames() {
    return this.games;
}

public addMove(gameId: string, move: string) {
    const game = this.games.find(game => game.id === gameId);
    if (game) {
        game.moves.push(move);
    }
}

public logState() {
    console.log(this.games);
}
}
```

▼ logger.ts

```
import { GameManager } from "../GameManager";

export function startLogger() {
    setInterval(() => {
        GameManager.getInstance().logState();
    }, 4000)
}
```

Copy

▼ index.ts

Copy

```
import { GameManager } from "../GameManager";
import { startLogger } from "../logger";

startLogger();

setInterval(() => {
  GameManager.getInstance().addGame({
    id: Math.random().toString(),
    "whitePlayer": "harkirat",
    "blackPlayer": "jaskirat",
    moves: []
  })
}, 5000)
```

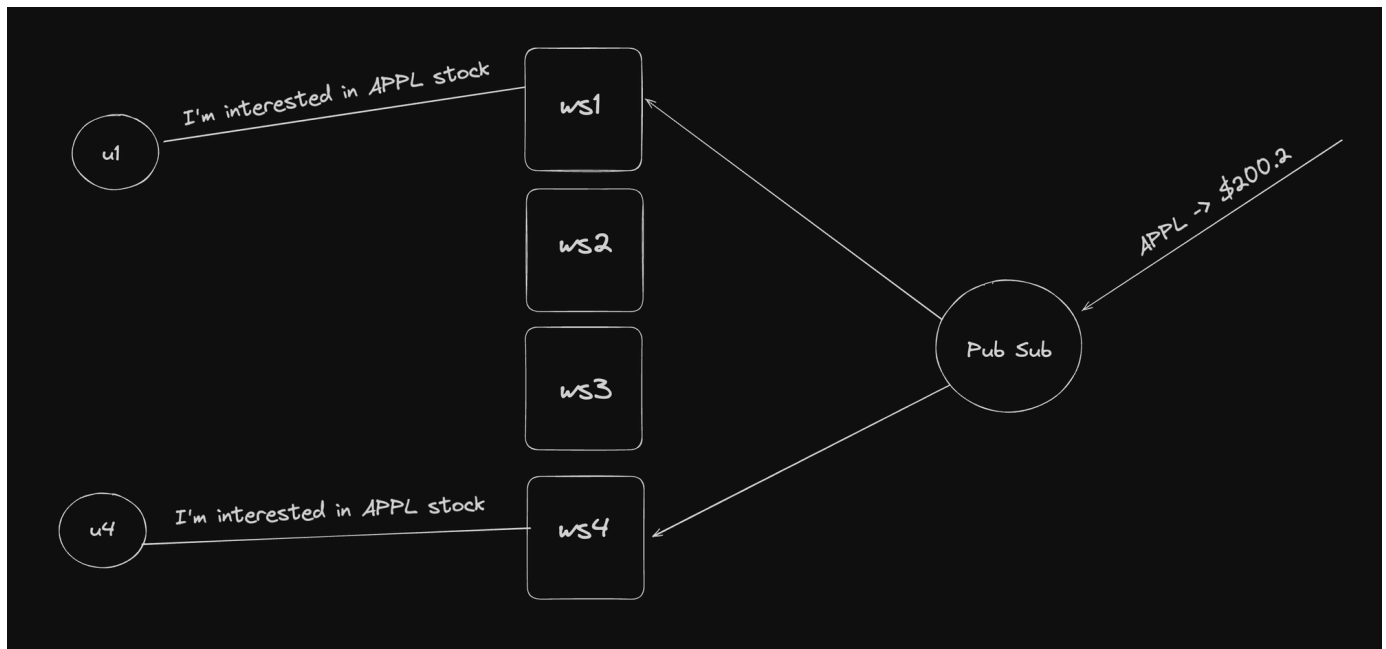
Try creating a new instance of the `GameManager` class. Notice it wont let you.

Pub Sub + Singleton

What if You want to create a system where users can subscribe to the `feed` of stocks (prices)

This application will be used by >1Mn users

How would you build it?



- Create a PubSubManager class (singleton)
- It keeps track of what all stocks are users on **this server** interested in
- It tells the pub sub whenever a **new stock** is added or a stock is removed from the list of interested stocks on that server
- It relays the events to the right sockets whenever an event is received

Pub Sub + Singleton (Implementation)

Starting the pub sub

- Start a pub sub (redis is a decent one)

```
docker run -d -p 6379:6379 redis
```

- Try a simple publish subscribe in two terminals

```
docker exec -it d1da6bcf089f /bin/bash
redis-cli
```

```

+ week-21-2 docker exec
+ week-21-2 docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS                NAMES
d1da6bcf089f   redis     "docker-entrypoint.s..." 32 seconds ago Up 31 seconds    0.0.0.0:6379->6379/tcp loving_sam
t
1f96b18250fb   postgres  "docker-entrypoint.s..." 2 days ago    Up 2 days      0.0.0.0:5432->5432/tcp loving_vau
an
+ week-21-2 docker exec -it d1da6bcf089f /bin/bash
root@d1da6bcf089f:/data# redis-cli
127.0.0.1:6379> SUBSCRIBE APPL
1) "subscribe"
2) "APPL"
3) (integer) 1
1) "message"
2) "APPL"
3) "100"
1) "message"
2) "APPL"
3) "100.1"
1) "message"
2) "APPL"
3) "100.01"
Reading messages... (press Ctrl-C to quit or any key to type command)
Last login: Sun Apr 21 17:01:34 on ttys016
+ ~ docker exec -it d1da6bcf089f /bin/bash
root@d1da6bcf089f:/data# PUBLISH APPL 100
(integer) 1
127.0.0.1:6379> PUBLISH APPL 100.1
(integer) 1
127.0.0.1:6379> PUBLISH APPL 100.01
(integer) 1
127.0.0.1:6379>

```

Creating the PubSubManager

- Init a simple node.js project

```

npm init -y
npx tsc --init
npm install redis

```

▼ Create the Pub Sub Manager

```

// Import the necessary module from the 'redis' package
import { createClient, RedisClientType } from 'redis';

export class PubSubManager {
  private static instance: PubSubManager;
  private redisClient: RedisClientType;
  private subscriptions: Map<string, string[]>;

  // Private constructor to prevent direct construction calls with t
  private constructor() {
    // Create a Redis client and connect to the Redis server
    this.redisClient = createClient();
    this.redisClient.connect();
    this.subscriptions = new Map();
  }

  // The static method that controls the access to the singleton ins
  public static getInstance(): PubSubManager {
    if (!PubSubManager.instance) {
      PubSubManager.instance = new PubSubManager();
    }
    return PubSubManager.instance;
  }

  public userSubscribe(userId: string, stock: string) {

```

```

    if (!this.subscriptions.has(stock)) {
        this.subscriptions.set(stock, []);
    }
    this.subscriptions.get(stock)?.push(userId);

    if (this.subscriptions.get(stock)?.length === 1) {
        this.redisClient.subscribe(stock, (message) => {
            this.handleMessage(stock, message);
        });
        console.log(`Subscribed to Redis channel: ${stock}`);
    }
}

public userUnSubscribe(userId: string, stock: string) {
    this.subscriptions.set(stock, this.subscriptions.get(stock)?.filter(
        (sub) => sub !== userId
    ));

    if (this.subscriptions.get(stock)?.length === 0) {
        this.redisClient.unsubscribe(stock);
        console.log(`UnSubscribed to Redis channel: ${stock}`);
    }
}

// Define the method that will be called when a message is published
private handleMessage(stock: string, message: string) {
    console.log(`Message received on channel ${stock}: ${message}`);
    this.subscriptions.get(stock)?.forEach((sub) => {
        console.log(`Sending message to user: ${sub}`);
    });
}

// Cleanup on instance destruction
public async disconnect() {
    await this.redisClient.quit();
}
}

```

▼ Create a simple index.ts file to simulate users

```

import { PubSubManager } from "../PubSubManager";

setInterval(() => {

```

Copy

```
PubSubManager.getInstance().userSubscribe(Math.random().toString()  
{}, 5000)
```