

WebSocket Explanation

What is a WebSocket?

explain this in details so that I understand it well and do not have doubts about this.

What is a WebSocket?

A **WebSocket** is a communication protocol that provides **full-duplex, real-time communication** between a client (like a browser) and a server over a single, long-lived TCP connection.

In simple words:

WebSocket allows the client and server to send messages to each other at any time without repeatedly creating new HTTP requests.

1 Why Do We Need WebSocket?

Let's first understand the problem with normal HTTP.

◆ Normal HTTP (Request–Response Model)

In HTTP:

- Client sends request
- Server sends response
- Connection closes

If the client wants new data:

- It must send another request

👉 This is inefficient for **real-time applications** like:

- Chat apps
- Live stock market updates
- Multiplayer games
- Live notifications
- Real-time dashboards

Because the client has to keep asking:

```
"Do you have new data?"  
"Do you have new data?"  
"Do you have new data?"
```

This wastes resources.

2 What WebSocket Solves

WebSocket creates:

- ✓ One persistent connection
- ✓ Full-duplex communication
- ✓ Low latency
- ✓ Server can push data anytime

After connection is established:

- Client can send data anytime
 - Server can send data anytime
 - No need to reopen connection
-

3 How WebSocket Works (Step-by-Step)

Step 1: HTTP Handshake

WebSocket starts as an HTTP request.

The client sends:

```
vbnet
```

```
GET /chat HTTP/1.1
```

```
Upgrade: websocket
```

```
Connection: Upgrade
```

The server responds:

```
makefile
```

```
HTTP/1.1 101 Switching Protocols
```

```
Upgrade: websocket
```

```
Connection: Upgrade
```

👉 After this handshake, the connection becomes a WebSocket connection.

Now it is no longer normal HTTP.

Step 2: Persistent TCP Connection

After handshake:

- Connection stays open
 - Data is exchanged using WebSocket frames
 - No more HTTP headers
 - Very lightweight communication
-

4 Full-Duplex Communication (Very Important)

What is Full-Duplex?

It means:

- Client and server can send data at the same time.
- No need to wait for request-response cycle.

Example:

In Chat App:

- User A sends message → Server
- Server instantly pushes message → User B
- User B replies immediately
- Server pushes to User A

All happening in real time.

5 HTTP vs WebSocket (Clear Comparison)

Feature	HTTP	WebSocket
Communication Type	Request–Response	Full-Duplex
Connection	Short-lived	Persistent
Real-time Support	Poor	Excellent
Server Push	Not possible (normally)	Possible
Overhead	High (headers every request)	Low
Use Case	REST APIs	Chat, live updates

6 WebSocket Architecture in System Design

pgsql

```
Client (Browser / Mobile)
  |
  | WebSocket Connection
  |
WebSocket Server
  |
  | (Optional)
  |
Database / Message Broker
```

In large systems:

- WebSocket server maintains many active connections
- Often combined with:
 - Redis Pub/Sub
 - Kafka
 - Message queues

Example:

- One user sends message

- WebSocket server publishes to Redis
 - All connected users receive update
-

7 Real-World Examples

Chat Applications

- WhatsApp Web
- Slack
- Discord

Live Sports Updates

- Cricket score updates
- Football live score

Stock Market

- Real-time price changes

Multiplayer Games

- Position updates
-

8 WebSocket in Backend (Example: Spring Boot)

Since you're learning backend and Spring Boot, this is important for you.

In Spring Boot, we use:

- `@EnableWebSocket`
- `@ServerEndpoint`
- STOMP protocol
- SockJS (fallback option)

WebSocket endpoint example:

```
java
```

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {
```

```
@Override
public void registerStompEndpoints(StompEndpointRegistry registry) {
    registry.addEndpoint("/chat").withSockJS();
}
}
```

9 WebSocket vs Polling vs Long Polling

◆ Polling

Client repeatedly asks server every few seconds.

Bad → Too many requests.

◆ Long Polling

Client sends request

Server holds request until data is available

Then responds

Better than polling but still HTTP-based.

◆ WebSocket

Single connection

Server pushes data instantly

Best for real-time.

10 Important Concepts in WebSocket

1. ws:// and wss://

- ws:// → Non-secure
- wss:// → Secure (like HTTPS)

Always use **wss** in production.

2. Frames

WebSocket sends data in small units called **frames**:

- Text frame
 - Binary frame
 - Ping
 - Pong
 - Close frame
-

3. Connection Lifecycle

1. Handshake
 2. Open
 3. Message exchange
 4. Close
-

Advantages

- ✓ Real-time communication
 - ✓ Low latency
 - ✓ Reduced server load
 - ✓ Efficient network usage
-

Disadvantages

- ✗ More complex than REST
 - ✗ Harder to scale
 - ✗ Requires connection management
 - ✗ Stateful (harder in distributed systems)
-

When NOT to Use WebSocket

Don't use WebSocket if:

- You only need simple CRUD APIs
- No real-time requirement
- Request-response is enough

Example:

- Login API
- Payment API
- Profile update API

Use REST instead.

Final Simple Understanding

If you remember only one thing:





HTTP = Ask and Get


WebSocket = Always Connected & Talk Anytime

Since you're preparing for backend roles, understanding WebSocket helps in:

- System Design Interviews
- Building chat systems
- Notification systems
- Real-time dashboards
- Payment status tracking

If you want next, I can explain:

-  WebSocket scaling problem
-  WebSocket with Redis in distributed systems
-  How WebSocket works internally at TCP level
-  How to implement Chat app using Spring Boot + WebSocket

Just tell me 

2 What WebSocket Solves

WebSocket creates:

- ✓ One persistent connection
- ✓ Full-duplex communication
- ✓ Low latency
- ✓ Server can push data anytime

explain this in details so that i understand it well and do not have doubt in this

Great 👍 let's go **very deep** into this part so you have **zero confusion**.

You asked about this:

WebSocket creates

- ✓ One persistent connection
- ✓ Full-duplex communication
- ✓ Low latency
- ✓ Server can push data anytime

Let's break each one clearly and practically.

1 One Persistent Connection

♦ What happens in HTTP?

In normal HTTP:

1. Client sends request
2. Server sends response
3. Connection closes

If you send 100 requests → 100 connections.

That means:

- TCP handshake happens again and again

- HTTP headers are sent again and again
 - More CPU + memory usage
-

◆ What happens in WebSocket?

In WebSocket:

1. One handshake happens (HTTP → WebSocket upgrade)
2. After that, **connection stays open**
3. It remains open until client or server closes it

So instead of:

```
sql
```

Request → Response → **Close**

Request → Response → **Close**

Request → Response → **Close**

You get:

```
pgsql
```

Open connection

Send → Receive

Send → Receive

Send → Receive

Close (**when** done)

💡 Why is this powerful?

Imagine a chat app:

Without WebSocket:

- Every message = new HTTP request

With WebSocket:

- One connection handles all messages

This reduces:

- Network overhead
 - Server load
 - Connection setup cost
-

2 Full-Duplex Communication

This is VERY IMPORTANT.

◆ What is Full-Duplex?

Full-duplex means:

Both client and server can send data to each other at the same time.

Not one by one.

Not waiting.

Not request-response.

◆ Compare With HTTP

HTTP is:

Half-duplex

Client must send request

Then only server can respond

Server CANNOT send data on its own.

◆ WebSocket Example (Chat)

Imagine:

User A sends message → Server

Server instantly pushes to User B

User B replies → Server


Server pushes to User A


All this happens without waiting for request cycles.

Both sides can speak anytime.

Real-Life Example

Think of:

 Phone Call → Full-duplex

 Email → Request-response

WebSocket = Phone call

HTTP = Email

Low Latency

◆ What is Latency?

Latency = Delay between sending and receiving data.

◆ Why HTTP has more latency?

Because every time:

1. TCP handshake
2. HTTP headers sent
3. Server processes
4. Response
5. Connection close

This takes time.

◆ Why WebSocket is Faster?

After initial handshake:

- No repeated TCP handshake
- No repeated HTTP headers

- Small lightweight frames
- Connection already established

So message delivery is almost instant.

💡 Example: Live Cricket Score

Without WebSocket:

- App checks every 5 seconds → delay possible

With WebSocket:

- As soon as score changes → server pushes instantly

That's low latency.

👉 Server Can Push Data Anytime

This is the biggest difference.

◆ In HTTP

Server cannot send data unless client asks.

It always waits for:

```
arduino
```

```
Client → Request
```

```
Server → Response
```

Server cannot say:

```
"Hey! New notification arrived!"
```

Because connection is closed.

◆ In WebSocket

Since connection is open:

Server can send data anytime.

Example:

User logs in → WebSocket connected

After 10 minutes → someone sends message

Server immediately pushes notification

Client did NOT request it.

Real Example

Bank transaction:

You pay using UPI.

Server processes payment.

Immediately status update appears.

This is server push.

Now Let's Combine All 4 Together

Imagine you are building:

- Chat App
- Trading App
- Gaming Server

What you need:

Requirement	Why WebSocket Helps
Continuous communication	Persistent connection
Both sides speak anytime	Full-duplex
Instant update	Low latency
Notifications	Server push

Deep Internal Understanding

Under the hood:

- WebSocket uses TCP
- Keeps TCP connection alive
- Sends data in small frames
- No HTTP overhead after handshake
- Maintains connection state

Because connection is maintained:

Server keeps track of:

- Which client is connected
- Their session
- Their socket connection

That's why WebSocket is stateful.

Important System Design Insight

Because connection stays open:

- Server memory usage increases
- Scaling becomes complex
- Load balancer must support sticky sessions
- You may need Redis for distributed messaging

This is why WebSocket is powerful but complex.

Simple Final Mental Model

If you remember this, you'll never forget:

HTTP → Knock the door every time

WebSocket → Stay inside the room and talk freely

If you want next level understanding, I can explain:

- 🔥 How WebSocket scales in distributed systems
- 🔥 WebSocket with Redis Pub/Sub
- 🔥 Why load balancer configuration is important
- 🔥 How WhatsApp-like architecture works

Tell me which one you want 🧐

WebSocket vs Polling vs Long Polling

explain this in details so that I understand it well and do not have doubts about this.

Excellent question 🧐

This is **very important for system design interviews** and real-world backend development.

We'll go step-by-step so you have **zero confusion**.

🧠 The Core Problem

We want:

How can the server send data to the client when new data arrives?

Example:

- New chat message
- New stock price
- Live score update
- Payment status update

Now let's compare the 3 approaches.

1 Polling

♦ What is Polling?

Polling means:

Client keeps asking the server again and again at fixed intervals.

Example:

- Every 5 seconds → "Any new data?"
- Every 5 seconds → "Any new data?"
- Every 5 seconds → "Any new data?"

◆ How It Works

pgsql

Client → Request

Server → Response

Connection closes

(wait 5 seconds)

Client → Request

Server → Response

Connection closes

◆ Example (Chat App)

Even if there is no new message:

Client still sends request every 5 seconds.

That means:

- Many useless requests
- Wasted bandwidth
- Increased server load

◆ Advantages

- ✓ Very simple
 - ✓ Easy to implement
 - ✓ Works everywhere
-

◆ Disadvantages

- ✗ High network overhead
 - ✗ Delay depends on polling interval
 - ✗ Not real-time
 - ✗ Waste of server resources
-

◆ Real-World Analogy

Polling is like:

- ☎ Calling your friend every 5 minutes and asking:

"Did you send the file?"

Even if they didn't.

2 Long Polling

Long Polling is an improved version of Polling.

◆ What is Long Polling?

Client sends request

Server **does NOT** respond immediately

Server waits until:

- New data arrives
- OR
- Timeout happens

Then server responds.

After receiving response, client immediately sends another request.

◆ How It Works

arduino

Client → Request

Server → (Waits...)

(New data arrives)

Server → Response

Connection closes

Client immediately sends new request

◆ If No Data?

Server waits for some time (e.g., 30 seconds)

If nothing happens → server sends empty response

Client sends new request again

◆ Why It's Better Than Polling

Because:

- No unnecessary repeated requests
 - Server responds only when data is available
 - Reduced waste compared to polling
-

◆ But Still Has Problems

- ✗ Still HTTP based
- ✗ Connection closes after each response
- ✗ Still repeated TCP handshakes
- ✗ More complex than polling

◆ Real-Life Analogy

Long polling is like:

📞 Calling your friend and saying:

"Stay on call. Tell me when the file is ready."

Friend keeps silent until ready.

Then hangs up.

You call again.

Better than polling — but still repeated calls.

3 WebSocket

Now comes the modern solution.

◆ What is WebSocket?

WebSocket creates:

- ✓ One persistent TCP connection
- ✓ Full-duplex communication
- ✓ Server can push anytime

◆ How It Works

Step 1: HTTP handshake

Step 2: Connection upgrades to WebSocket

Step 3: Connection stays open

```
pgsql
```

```
Open connection
```

```
Client ↔ Server communicate anytime
```

```
Connection stays open
```

No closing after each message.

◆ Example (Chat)

User A sends message

Server instantly pushes to User B

No new request needed.

◆ Advantages

- ✓ True real-time
 - ✓ Very low latency
 - ✓ Low overhead after handshake
 - ✓ Efficient network usage
-

◆ Disadvantages

- ✗ More complex
 - ✗ Harder to scale
 - ✗ Requires connection management
 - ✗ Stateful
-



Clear Comparison Table

Feature	Polling	Long Polling	WebSocket
Protocol	HTTP	HTTP	WebSocket (over TCP)
Connection Type	Short-lived	Long-held but closes	Persistent
Real-time	✗ No	⚠ Almost	✓ Yes
Server Push	✗ No	⚠ Indirect	✓ Yes
Overhead	High	Medium	Low

Feature	Polling	Long Polling	WebSocket
Latency	High	Medium	Very Low
Complexity	Easy	Medium	Complex
Best For	Simple apps	Moderate real-time	High real-time apps

When Should You Use What?

Use Polling When:

- App is simple
- Updates are rare
- Real-time is not critical

Example:

- Admin dashboard refresh
- Checking email occasionally

Use Long Polling When:

- Need near real-time
- Cannot use WebSocket (firewall/proxy issues)
- Moderate traffic

Use WebSocket When:

- Chat apps
- Trading platforms
- Live gaming
- Real-time collaboration
- Live notifications

Deep System Design Insight

In large systems:

WebSocket servers often use:

- Redis Pub/Sub
- Kafka
- Load balancer with sticky sessions

Because:

WebSocket connections are stateful.

If a user connects to Server A, future messages must reach Server A.

That makes scaling more complex than REST APIs.





Final Mental Model (Very Important)

Polling → Keep knocking the door

Long Polling → Knock once, wait, then knock again

WebSocket → Enter the house and talk freely

If you want next level understanding, I can explain:

-  How WhatsApp uses WebSockets
-  How WebSockets scale with Redis
-  Why WebSockets are stateful
-  Interview questions on this topic

Tell me which one you want 🙌

- ◆ WebSocket scaling problem
- ◆ WebSocket with Redis in distributed systems
- ◆ How WebSocket works internally at TCP level

explain this in details so that I understand it well and do not have doubts about this.

Perfect 🙌

Now we are entering **real backend + system design level understanding**.

I'll explain this deeply but clearly so you have **zero confusion**.

We'll cover:

- 1 WebSocket scaling problem
- 2 WebSocket with Redis in distributed systems
- 3 How WebSocket works internally at TCP level

1 🔥 WebSocket Scaling Problem

First Understand the Core Issue

WebSocket is:

- Persistent connection
- Stateful
- Server holds connection in memory

This creates scaling challenges.

♦ Why REST APIs Scale Easily?

In REST:

- Client sends request
- Server responds
- Connection closes
- No memory of client required

This is **stateless**.

Any request can go to:

- Server A
- Server B

- Server C

No problem.

◆ Why WebSocket Is Harder?

In WebSocket:

- Connection stays open
- Server stores active connections
- Each client is attached to a specific server

Example:

```
pgsql
```

```
User A → Connected to Server 1
```

```
User B → Connected to Server 2
```

Now:

User A sends message to User B.

Problem:

- Server 1 doesn't know User B is on Server 2
- Server 2 doesn't know about message from Server 1

This is the scaling problem.

◆ Another Big Issue: Memory Usage

Each WebSocket connection:

- Uses memory
- Uses file descriptor
- Uses TCP socket

If:

- 100,000 users connected
- Each connection consumes memory

Server must handle:

- Large RAM
 - Large concurrent connections
-

◆ Load Balancer Problem

If you use load balancer:

arduino

Client → Load Balancer → Server

Next request might go to another server.

But WebSocket requires:

👉 Sticky sessions

Because once connected:

Client must stay on same server.

Otherwise connection breaks.

2 🔥 WebSocket With Redis in Distributed Systems

This is the real-world solution.

Let's understand properly.

🎯 Problem We Need to Solve

We have:

arduino

Server 1

Server 2

Server 3

Users are connected randomly:

- User A → Server 1
- User B → Server 2
- User C → Server 3

Now:

User A sends message to User C.

How will Server 1 send message to Server 3?

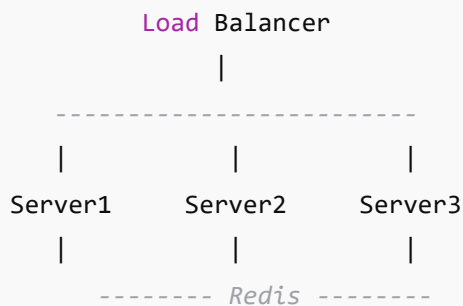
They are separate machines.

💡 Solution: Use Redis Pub/Sub

Redis acts like a **message broker**.

◆ Architecture

pgsql



◆ How It Works Step-by-Step

Step 1:

User A sends message to Server 1.

Step 2:

Server 1 publishes message to Redis channel.

Example:

```
arduino
```

```
Redis.publish("chat-room-1", message)
```

Step 3:

All servers are subscribed to Redis channel.

```
nginx
```

```
Server1 subscribed
```

```
Server2 subscribed
```

```
Server3 subscribed
```

Step 4:

Redis broadcasts message to all servers.

Step 5:

Server 3 checks:

"Is User C connected to me?"

If yes:

→ Send message via WebSocket

Why This Works

Because:

- Servers don't talk directly
- Redis becomes central communication layer
- System becomes horizontally scalable

You can add:

- Server 4
- Server 5
- Server 6

No architecture change needed.

◆ Why Redis?

Because Redis:

- Extremely fast (in-memory)
- Supports Pub/Sub
- Low latency
- Lightweight

Other alternatives:

- Kafka
- RabbitMQ

3 🔥 How WebSocket Works Internally at TCP Level

Now we go deep into networking.

Step 1: TCP Connection

WebSocket uses TCP underneath.

Before WebSocket:

TCP 3-way handshake happens:

```
arduino
```

```
Client → SYN
```

```
Server → SYN-ACK
```

```
Client → ACK
```

Now TCP connection established.

Step 2: HTTP Upgrade Handshake

Client sends:

```
vbnet
```

```
GET /chat HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: random_key
```

Server replies:

```
makefile

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hashed_key
```

Now protocol switches.

Step 3: Frame-Based Communication

After upgrade:

No more HTTP.

Data is sent in **WebSocket frames**.

Each frame contains:

- FIN bit
- Opcode (text/binary/ping/pong)
- Payload length
- Masking key (for client → server)
- Payload data

◆ Important Detail: Masking

Client-to-server messages are masked for security.

Server-to-client messages are not masked.

Step 4: Connection Stays Alive

TCP connection remains open.

WebSocket supports:

- Ping frame
- Pong frame

Used to:

- Check if connection is alive
- Prevent idle timeout

◆ What Happens If Network Breaks?

TCP detects:

- Connection reset
- Timeout

Server removes connection from memory.



Very Important Concept

WebSocket = Stateful over TCP

Server stores:

- Socket reference
- Session info
- User mapping

Example:

```
javascript
```

```
Map<userId, WebSocketSession>
```

That's why:

Scaling requires:

- Sticky sessions
 - Redis
 - Distributed messaging
-

Final Mental Model

REST API

Stateless

Easy to scale

WebSocket

Stateful

Real-time

Harder to scale

Distributed WebSocket

WebSocket + Redis

→ Scalable real-time architecture

Real Interview-Level Summary

WebSocket scaling problems include:

1. Persistent connections consume memory
2. Requires sticky sessions
3. Cross-server communication issue
4. Horizontal scaling complexity

Solution:

Use:

- Load balancer
 - Redis Pub/Sub
 - Shared message broker
 - Proper connection management
-

If you want next level knowledge, I can explain:

- 🔥 WhatsApp-like architecture
- 🔥 Designing a chat system for 10 million users
- 🔥 How to handle 1 million concurrent WebSocket connections
- 🔥 WebSocket vs gRPC streaming

Tell me what you want next 🤖