# □□□□□□□□□□ की खोज

यह ब्लॉग पोस्ट □□□□□□□□-4□ की सहायता से तैयार किया गया है।

---

**परिचय**

नमस्ते, मैं ली झीवेई हूँ। □□□□□□□□□□□ प्लेटफॉर्म के संस्थापक और □□□ के रूप में, और पूर्व □□□□□□□□□□ इंजीनियर के रूप में, मेरे पास □□□□□□□□□□ पर व्यापक अनुभव है, खासकर □□ □□□ के विकास प्रक्रिया में।

□□□□□□□□□□ **का महत्व**

□□□□□□□□□□ एक प्रोटोकॉल है जो एकल □□□ कनेक्शन पर पूर्ण डुप्लेक्स संचार चैनल प्रदान करता है। यह आधुनिक एप्लिकेशन में व्यापक रूप से उपयोग किया जाता है जहां वास्तविक समय में इंटरैक्शन की आवश्यकता होती है, जैसे कि तात्कालिक संदेश, वास्तविक समय टिप्पणियाँ, मल्टीप्लेयर गेम्स, सहयोगी संपादन और वास्तविक समय स्टॉक मूल्य।

□□□□□□□□□□ **का आधुनिक उपयोग**

□□□□□□□□□□ का उपयोग निम्नलिखित क्षेत्रों में व्यापक रूप से किया जाता है: - **तत्काल संदेश (□□)** - **रियल-टाइम टिप्पणियाँ** - **मल्टीप्लेयर गेम्स** - **सहयोगी संपादन** - **रियल-टाइम स्टॉक कीमतें**

□□□□□□□□□□ **का विकास**

**पोलिंग (□□□□□□□□):** क्लाइंट सर्वर से अपडेट प्राप्त करने के लिए बार-बार अनुरोध करता है। **लॉन्ग पोलिंग (□□□□ □□□□□□□□):** सर्वर अनुरोध को तब तक खुला रखता है जब तक नई जानकारी उपलब्ध न हो। □□□□ **द्विदिश कनेक्शन (□□□□ □□□□□□□□□□□□□□ □□□□□□□□□□□):** भेजने और प्राप्त करने के लिए कई कनेक्शन की आवश्यकता होती है, और प्रत्येक अनुरोध में □□□□ हेडर शामिल होता है। **एकल □□□ कनेक्शन (□□□□□□□□□□):** □□□□ द्विदिश कनेक्शन की सीमाओं को दूर करता है, जिससे उच्च वास्तविक समय क्षमता और कम विलंबता प्रदान की जाती है।

□□□ **पर** □□□□□□□□□□ **को लागू करना**

□□□□□□□□□□ एक प्रोटोकॉल है जो क्लाइंट और सर्वर के बीच द्वि-दिशात्मक संचार को सक्षम बनाता है। □□□ ऐप्स में □□□□□□□□□□ का उपयोग करने के लिए, आप `URLSessionWebSocketTask` का उपयोग कर सकते हैं, जो □□□ 13 और बाद के संस्करणों में उपलब्ध है।

**1. □□□□□□□□□□ कनेक्शन स्थापित करना** सबसे पहले, आपको □□□□□□□□□□ कनेक्शन स्थापित करने की आवश्यकता है। यहां एक उदाहरण दिया गया है:

```
import Foundation

let url = URL(string: "wss://your.websocket.server")!
let webSocketTask = URLSession.shared.webSocketTask(with: url)
webSocketTask.resume()
```

**2. संदेश भेजना और प्राप्त करना** □□□□□□□□□□ कनेक्शन स्थापित होने के बाद, आप संदेश भेज और प्राप्त कर सकते हैं।

**संदेश भेजना:**

```
let message = URLSessionWebSocketTask.Message.string("Hello, WebSocket!")
webSocketTask.send(message) { error in
    if let error = error {
        print("WebSocket couldn't send message because: \(error)")
    }
}
```

**संदेश प्राप्त करना:**

```
webSocketTask.receive { result in
    switch result {
    case .failure(let error):
        print("WebSocket couldn't receive message because: \(error)")
    case .success(let message):
        switch message {
        case .string(let text):
            print("Received text: \(text)")
        case .data(let data):
            print("Received data: \(data)")
        @unknown default:
            fatalError()
        }
    }
}
```

**3. कनेक्शन बंद करना** जब आप □□□□□□□□□□ कनेक्शन का उपयोग करना समाप्त कर लें, तो इसे बंद करना महत्वपूर्ण है।

```
webSocketTask.cancel(with: .goingAway, reason: nil)
```

**4. पिंग और पोंग** □□□□□□□□□□ कनेक्शन को सक्रिय रखने के लिए, आप पिंग और पोंग संदेश भेज सकते हैं।

**पिंग भेजना:**

```
webSocketTask.sendPing { error in
    if let error = error {
        print("Ping failed: \(error)")
    }
}
```

**पोंग प्राप्त करना:**

पोंग संदेश स्वचालित रूप से प्राप्त होते हैं जब सर्वर पिंग का जवाब देता है।

**5. त्रुटि प्रबंधन** □□□□□□□□□□ कनेक्शन के दौरान होने वाली त्रुटियों को संभालना महत्वपूर्ण है। आप `URLSessionWebSocketTask` के `receive` और `send` मेथड्स में त्रुटि हैंडलिंग को शामिल कर सकते हैं।

```
webSocketTask.receive { result in
    switch result {
    case .failure(let error):
        print("WebSocket error: \(error)")
    case .success(let message):
        // Handle message
    }
}
```

**6. पूर्ण उदाहरण** यहां एक पूर्ण उदाहरण दिया गया है जो □□□□□□□□□□ कनेक्शन स्थापित करता है, संदेश भेजता और प्राप्त करता है, और कनेक्शन को बंद करता है:

```
import Foundation

let url = URL(string: "wss://your.websocket.server")!
let webSocketTask = URLSession.shared.webSocketTask(with: url)
webSocketTask.resume()
```

```
// Send a message
let message = URLSessionWebSocketTask.Message.string("Hello, WebSocket!")
webSocketTask.send(message) { error in
    if let error = error {
        print("WebSocket couldn't send message because: \(error)")
    }
}

// Receive a message
webSocketTask.receive { result in
    switch result {
    case .failure(let error):
        print("WebSocket couldn't receive message because: \(error)")
    case .success(let message):
        switch message {
        case .string(let text):
            print("Received text: \(text)")
        case .data(let data):
            print("Received data: \(data)")
        @unknown default:
            fatalError()
        }
    }
}

// Close the connection
webSocketTask.cancel(with: .goingAway, reason: nil)
```

यह उदाहरण □□□ ऐप में □□□□□□□□□□ का उपयोग करने के लिए एक बुनियादी संरचना प्रदान करता है। आप अपनी आवश्यकताओं के अनुसार इसे और विकसित कर सकते हैं।

**लोकप्रिय** □□□ □□□□□□□□□□ **लाइब्रेरीज़:** - □□□□□□□□□□□□□□ **(□□□□□□□□□□-□, 4910 स्टार्स)** - □□□□□□□□□□□ **(□□□□□□, 1714 स्टार्स)** - □□□□□□□□□□□□□□□□ **(□□□□□□, 435 स्टार्स)**

□□□□□□□□□□□□ **का उपयोग**

1. **आरंभीकरण और कनेक्शन:**

```
SRWebSocket *webSocket = [[SRWebSocket alloc] initWithURLRequest:[NSURLRequest requestWithURL:[NSUR
webSocket.delegate = self;
[webSocket open];
```

2. **संदेश भेजें:**

```
[webSocket send:@"Hello, World!"];
```

3. **संदेश प्राप्त करें:** आने वाले संदेशों और घटनाओं को संभालने के लिए `SRWebSocketDelegate` विधियों को लागू करें।
4. **त्रुटि प्रबंधन और घटना सूचना:** त्रुटियों को उचित तरीके से संभालें और उपयोगकर्ताओं को कनेक्शन समस्याओं के बारे में सूचित करें।

□□□□□□□□□□ **प्रोटोकॉल का विस्तृत विवरण**

□□□□□□□□□□ एक कंप्यूटर संचार प्रोटोकॉल है जो एक ही □□□ कनेक्शन पर पूर्ण-डुप्लेक्स संचार प्रदान करता है। यह □□□□ प्रोटोकॉल के ऊपर बनाया गया है और इसे मुख्य रूप से वेब ब्राउज़र और सर्वर के बीच रीयल-टाइम संचार के लिए डिज़ाइन किया गया है। □□□□□□□□□□ प्रोटोकॉल को □□□ 6455 द्वारा परिभाषित किया गया है।

□□□□□□□□□□ **प्रोटोकॉल के मुख्य विशेषताएं:**

1. **पूर्ण-डुप्लेक्स संचार**: □□□□□□□□□□ क्लाइंट और सर्वर दोनों एक ही समय पर डेटा भेज और प्राप्त कर सकते हैं।
2. **कम ओवरहेड**: □□□□ की तुलना में □□□□□□□□□□ में डेटा ट्रांसमिशन का ओवरहेड कम होता है।
3. **रीयल-टाइम संचार**: □□□□□□□□□□ रीयल-टाइम एप्लिकेशन जैसे चैट, गेमिंग, और लाइव डेटा स्ट्रीमिंग के लिए आदर्श है।

□□□□□□□□□□ **हैंडशेक:** □□□□□□□□□□ कनेक्शन स्थापित करने के लिए, क्लाइंट और सर्वर के बीच एक हैंडशेक प्रक्रिया होती है। यह प्रक्रिया □□□□ प्रोटोकॉल का उपयोग करती है।

1. **क्लाइंट अनुरोध**: क्लाइंट एक □□□□ अनुरोध भेजता है जिसमें `Upgrade: websocket` और `Connection: Upgrade` हेडर शामिल होते हैं।
2. **सर्वर प्रतिक्रिया**: सर्वर अनुरोध को स्वीकार करता है और एक □□□□ प्रतिक्रिया भेजता है जिसमें `101 Switching Protocols` स्टेटस कोड होता है।
3. **कनेक्शन स्थापित**: एक बार हैंडशेक पूरा हो जाने के बाद, क्लाइंट और सर्वर के बीच □□□□□□□□□□ कनेक्शन स्थापित हो जाता है।

□□□□□□□□□□ **फ्रेम:** □□□□□□□□□□ डेटा को फ्रेम के रूप में ट्रांसमिट करता है। प्रत्येक फ्रेम में निम्नलिखित भाग होते हैं:

- □ □□□ **बिट**: यह इंगित करता है कि यह अंतिम फ्रेम है या नहीं।
- □ □□□□□□: यह फ्रेम के प्रकार को निर्दिष्ट करता है (उदाहरण के लिए, टेक्स्ट, बाइनरी, कनेक्शन बंद करना)।
- □ □□□□ **बिट**: यह इंगित करता है कि डेटा मास्क किया गया है या नहीं।
- □ □□□□□□□ **लंबाई**: यह डेटा की लंबाई को निर्दिष्ट करता है।
- □ □□□□□□□ **डेटा**: वास्तविक डेटा जो ट्रांसमिट किया जा रहा है।

□□□□□□□□□□ **उपयोग के उदाहरण:**

```
//        JavaScript
const socket = new WebSocket('ws://example.com/socket');

socket.onopen = function(event) {
    console.log('WebSocket        ');
    socket.send('Hello Server!');
};

socket.onmessage = function(event) {
    console.log('          : ' + event.data);
};

socket.onclose = function(event) {
    console.log('WebSocket      ');
};

#       Python      (websockets             )
import asyncio
import websockets

async def echo(websocket, path):
    async for message in websocket:
        await websocket.send(message)

start_server = websockets.serve(echo, "localhost", 8765)
```

```
asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

□□□□□□□□□□ प्रोटोकॉल का उपयोग करके, डेवलपर्स अधिक कुशल और रीयल-टाइम संचार सुविधाओं वाले एप्लिकेशन बना सकते हैं।

□□□□□□□□□□ □□□ पर चलता है और इसमें कई सुधार शामिल हैं: - **सुरक्षा मॉडल:** ब्राउज़र-आधारित स्रोत सुरक्षा सत्यापन मॉडल को जोड़ा गया है। - **पता और प्रोटोकॉल नामकरण:** एकल पोर्ट पर कई सेवाओं और एकल □□ पते पर कई डोमेन नामों का समर्थन करता है। - **फ्रेम मैकेनिज्म:** □□ पैकेट के समान फ्रेम मैकेनिज्म के माध्यम से □□□ को बढ़ाया गया है, जिसकी कोई लंबाई सीमा नहीं है। - **बंद करने की प्रक्रिया:** कनेक्शन को साफ तरीके से बंद करने की सुनिश्चितता प्रदान करता है।

### □□□□□□□□□□ प्रोटोकॉल का मूल

□□□□□□□□□□ प्रोटोकॉल एक ऐसा प्रोटोकॉल है जो क्लाइंट और सर्वर के बीच द्विदिश संचार (□□□□□□□□□□□□□□□ □□□□□□□□□□-□□□□) को सक्षम बनाता है। यह □□□□ प्रोटोकॉल पर आधारित है, लेकिन इसे इस तरह से डिज़ाइन किया गया है कि यह कनेक्शन को लंबे समय तक खुला रख सकता है और डेटा को कम ओवरहेड के साथ ट्रांसमिट कर सकता है। □□□□□□□□□□ प्रोटोकॉल के कुछ मुख्य तत्व निम्नलिखित हैं:

1. **हैंडशेक (□□□□□□□□□□):** □□□□□□□□□□ कनेक्शन की शुरुआत एक □□□□ हैंडशेक से होती है। क्लाइंट एक □□□□ अनुरोध (□□□□□□□□) भेजता है जिसमें `Upgrade: websocket` हेडर शामिल होता है। यदि सर्वर □□□□□□□□□□ कनेक्शन को स्वीकार करता है, तो यह एक □□□□ प्रतिक्रिया (□□□□□□□□□) भेजता है जिसमें `101 Switching Protocols` स्टेटस कोड होता है।

2. **फ्रेमिंग (□□□□□□□□):** □□□□□□□□□□ डेटा को फ्रेम्स (□□□□□□□) के रूप में ट्रांसमिट करता है। प्रत्येक फ्रेम में एक छोटा हेडर होता है जो डेटा के प्रकार और लंबाई को निर्दिष्ट करता है। यह फ्रेमिंग मैकेनिज्म □□□□□□□□□□ को डेटा को कुशलतापूर्वक ट्रांसमिट करने में सक्षम बनाता है।

3. **द्विदिश संचार (□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□):** एक बार कनेक्शन स्थापित हो जाने के बाद, क्लाइंट और सर्वर दोनों किसी भी समय डेटा भेज और प्राप्त कर सकते हैं। यह □□□□ के अनुरोध-प्रतिक्रिया मॉडल से अलग है, जहां क्लाइंट को हमेशा अनुरोध शुरू करना पड़ता है।

4. **कनेक्शन क्लोजिंग (□□□□□□□□□□□ □□□□□□□□):** □□□□□□□□□□ कनेक्शन को बंद करने के लिए, क्लाइंट या सर्वर एक क्लोज फ्रेम (□□□□□ □□□□□) भेज सकते हैं। यह फ्रेम कनेक्शन को सुरक्षित रूप से बंद करने के लिए उपयोग किया जाता है।

5. **सुरक्षा (□□□□□□□□□):** □□□□□□□□□□ प्रोटोकॉल में सुरक्षा के लिए □□□/□□□ का उपयोग किया जा सकता है, जिसे □□□-□□□□□□ □□□□□□□ (□□□) कहा जाता है। यह डेटा को एन्क्रिप्ट करके सुरक्षित संचार सुनिश्चित करता है।

□□□□□□□□□□ प्रोटोकॉल का उपयोग वास्तविक समय (□□□□-□□□□) एप्लिकेशन जैसे चैट एप्लिकेशन, ऑनलाइन गेमिंग, और वित्तीय ट्रेडिंग प्लेटफॉर्म में व्यापक रूप से किया जाता है। यह प्रोटोकॉल कम लेटेंसी और उच्च प्रदर्शन वाले संचार को सक्षम बनाता है।

**1. हैंडशेक:** □□□□□□□□□□ हैंडशेक □□□□ अपग्रेड मैकेनिज्म का उपयोग करता है: - **क्लाइंट अनुरोध:** `http GET /chat HTTP/1.1 Host: server.example.com Upgrade: websocket Connection:`

Upgrade Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ== Origin: http://example.com Sec-WebSocket-Protocol: chat, superchat Sec-WebSocket-Version: 13

- **सर्वर प्रतिक्रिया:** http HTTP/1.1 101 Switching Protocols Upgrade: websocket Connection: Upgrade Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo= Sec-WebSocket-Prot chat

**2. डेटा ट्रांसमिशन:** □□□□□□□□□□ फ्रेम में □□□-8 टेक्स्ट, बाइनरी डेटा और कंट्रोल फ्रेम, जैसे कि बंद करना, पिंग और पोंग, शामिल हो सकते हैं।

**3. सुरक्षा:** ब्राउज़र स्वचालित रूप से `Origin` हेडर जोड़ता है, जिसे अन्य क्लाइंट द्वारा नकली नहीं बनाया जा सकता है।

□□□□□□□□□□ □□□

- □□-□□□: `ws://host:port/path?query`
- □□□-□□□: `wss://host:port/path?query`

□□□□□□□□□□ **फ्रेम प्रोटोकॉल**

**फ्रेम संरचना:** - □□□ **(1 बिट):** यह संदेश का अंतिम टुकड़ा है यह दर्शाता है। - □□□**1,** □□□**2,** □□□**3 (प्रत्येक 1 बिट):** भविष्य में उपयोग के लिए आरक्षित। - □□□□□□□ **(4 बिट):** पेलोड डेटा को कैसे पार्स करना है यह परिभाषित करता है। - `0x0`: जारी फ्रेम - `0x1`: टेक्स्ट फ्रेम - `0x2`: बाइनरी फ्रेम - `0x8`: कनेक्शन बंद - `0x9`: पिंग - `0xA`: पोंग - □□□□ **(1 बिट):** यह दर्शाता है कि पेलोड डेटा मास्क किया गया है या नहीं। - **पेलोड लंबाई (7 बिट):** पेलोड डेटा की लंबाई।

**मास्किंग कुंजी (**□□□□□□□□ □□□**):** यह मध्यस्थ हमलों (□□□-□□-□□□-□□□□□□□ □□□□□□□□) को रोकने के लिए क्लाइंट के फ्रेम को मास्क करने के लिए उपयोग की जाती है।

**हैंडशेक बंद करना**

**बंद फ्रेम (**□□□□□ □□□□□**):** - इसमें बंद होने के कारण को दर्शाने वाला बॉडी शामिल हो सकता है। - दोनों पक्षों को बंद फ्रेम भेजना और उसका जवाब देना आवश्यक है।

**उदाहरण**

**उदाहरण 1: बिना मास्क वाला सिंगल फ्रेम टेक्स्ट संदेश**

```
0x81 0x05 0x48 0x65 0x6c 0x6c 0x6f
```

इसमें “□□□□□” शामिल है।

**उदाहरण 2: सिंगल फ्रेम मास्क किए गए टेक्स्ट संदेश**

```
0x81 0x85 0x37 0xfa 0x21 0x3d 0x7f 0x9f 0x4d 0x51 0x58
```

इसमें “□□□□□” शामिल है, जो मास्क कुंजी के साथ है।

**उदाहरण 3: शार्डेड अनमास्क टेक्स्ट संदेश**

```
0x01 0x03 0x48 0x65 0x6c
0x80 0x02 0x6c 0x6f
```

शार्ड में “□□□” और “□□” दो फ्रेम शामिल हैं।

**उन्नत विषय**

**मास्किंग और अनमास्किंग:** - मास्किंग का उपयोग मध्यम व्यक्ति हमलों को रोकने के लिए किया जाता है। - क्लाइंट से आने वाले प्रत्येक फ्रेम को मास्क किया जाना चाहिए। - प्रत्येक फ्रेम के लिए मास्क कुंजी यादृच्छिक रूप से चुनी जाती है।

**शार्डिंग (□□□□□□□□□):** - अज्ञात लंबाई के डेटा को भेजने के लिए उपयोग किया जाता है। - शार्डेड संदेश □□□ 0 वाले फ्रेम से शुरू होता है और □□□ 1 वाले फ्रेम पर समाप्त होता है।

**कंट्रोल फ्रेम:** - कंट्रोल फ्रेम (जैसे कि बंद करना, पिंग और पोंग) के लिए विशिष्ट ऑपकोड होते हैं। - ये फ्रेम □□□□□□□□□□ कनेक्शन की स्थिति को प्रबंधित करने के लिए उपयोग किए जाते हैं।

**विस्तारणीयता (□□□□□□□□□□□□)**

**एक्सटेंशन डेटा को मैसेज बॉडी के एप्लिकेशन डेटा से पहले रखा जा सकता है:** - प्रत्येक फ्रेम को नियंत्रित करने के लिए रिजर्व्ड बिट्स का उपयोग किया जा सकता है। - भविष्य में परिभाषित करने के लिए कुछ ऑपकोड रिजर्व किए जा सकते हैं। - यदि अधिक ऑपकोड की आवश्यकता हो, तो रिजर्व्ड बिट्स का उपयोग किया जा सकता है।

**भेजें:** - कनेक्शन □□□□ स्थिति में होना चाहिए। - डेटा को फ्रेम में एनकैप्सुलेट किया जाता है, और डेटा बड़ा होने पर इसे टुकड़ों में भेजने का विकल्प चुना जा सकता है। - पहले फ्रेम का मान सही होना चाहिए, जो प्राप्तकर्ता को डेटा प्रकार (टेक्स्ट या बाइनरी) बताता है। - अंतिम फ्रेम का □□□ 1 पर सेट होना चाहिए।

**हैंडशेक बंद करना:** - दोनों पक्ष बंद करने का फ्रेम भेज सकते हैं। - बंद करने का फ्रेम भेजने के बाद, कोई और डेटा नहीं भेजा जाता है। - बंद करने का फ्रेम प्राप्त करने के बाद, उसके बाद प्राप्त किसी भी डेटा को छोड़ दिया जाता है।

**कनेक्शन बंद करना:** - □□□□□□□□□□ कनेक्शन को बंद करना, जो नीचे के □□□ कनेक्शन को बंद करने के बराबर है। - बंद करने वाला फ्रेम भेजने या प्राप्त करने के बाद, □□□□□□□□□□ कनेक्शन की स्थिति बंद होने की प्रक्रिया में होती है। - जब नीचे का □□□ कनेक्शन बंद हो जाता है, तो □□□□□□□□□□ कनेक्शन की स्थिति बंद हो जाती है।

**संदर्भ सामग्री**

- □□□□□□□□□□ □□□ : □□□6455
- □□□□□□□□□□□□□ □□□□□□□□□□□□□
- □□□□□□□□□□□□□: □□□□□□□ □□

**आभार**

सभी का धन्यवाद। यदि आपके पास और प्रश्न या चर्चा हैं, तो कृपया □□□□□□ या □□□□□ पर मुझसे संपर्क करने में संकोच न करें।