readme.md for @api.global/typedsocket

A TypeScript library for creating typed WebSocket connections with bi-directional communication support. Extends `@api.global/typedrequest` to bring type-safe request/response patterns to WebSocket connections.

Issue Reporting and Security

For reporting bugs, issues, or security vulnerabilities, please visit community.foss.global/. This is the central community hub for all issue reporting. Developers who sign and comply with our contribution agreement and go through identification can also get a code.foss.global/ account to submit Pull Requests directly.

Features

- **Full Type Safety** - Leverages TypeScript for compile-time checking of all request/response payloads
- **Bi-directional Communication** - Both server and client can initiate requests
- **Auto-reconnect** - Client automatically reconnects on connection loss
- **Connection Tagging** - Tag and filter connections for targeted messaging
- **Browser Compatible** - Works in both Node.js and browser environments
- **SmartServe Integration** - Native support for SmartServe's WebSocket handling

Install

```
npm install @api.global/typedsocket
```

Or with pnpm:

```
pnpm add @api.global/typedsocket
```

Usage

Prerequisites

- TypeScript project setup
- Basic understanding of async/await patterns
- Familiarity with `@api.global/typedrequest` concepts

Define Your Request Interface

First, define the typed request interface that both client and server will use:

```
import * as typedrequestInterfaces from '@api.global/typedrequest-interfaces';

interface IGreetingRequest extends typedrequestInterfaces.implementsTR<
  typedrequestInterfaces.ITypedRequest,
  IGreetingRequest
> {
  method: 'greet';
  request: {
    name: string;
  };
  response: {
    message: string;
  };
}
```

Server Setup

Create a WebSocket server that handles typed requests:

```
import { TypedSocket } from '@api.global/typedsocket';
import * as typedrequest from '@api.global/typedrequest';

// Create the router and add handlers
const typedRouter = new typedrequest.TypedRouter();
```

```
typedRouter.addTypedHandler<IGreetingRequest>(
  new typedrequest.TypedHandler('greet', async (requestData) => {
    return {
      message: `Hello, ${requestData.name}! □□,
    };
  })
);

// Start the TypedSocket server (defaults to port 3000)
const server = await TypedSocket.createServer(typedRouter);
```

Integration with SmartServe

For SmartServe-based applications, use `fromSmartServe()` for native integration:

```
import { TypedSocket } from '@api.global/typedsocket';
import { SmartServe } from '@push.rocks/smartserve';
import * as typedrequest from '@api.global/typedrequest';

const typedRouter = new typedrequest.TypedRouter();

// Add handlers for client-to-server requests
typedRouter.addTypedHandler<IGreetingRequest>(
  new typedrequest.TypedHandler('greet', async (requestData) => {
    return { message: `Hello, ${requestData.name}!` };
  })
);

// Create SmartServe with typedRouter in websocket options
const smartServe = new SmartServe({
  port: 3000,
  websocket: {
    typedRouter,
    onConnectionOpen: (peer) => {
      // Tag connections for later filtering
      peer.tags.add('client');
    }
  }
});
```

```
await smartServe.start();

// Create TypedSocket bound to SmartServe
const typedSocket = TypedSocket.fromSmartServe(smartServe, typedRouter);

// Push notifications to tagged clients
const clients = await typedSocket.findAllTargetConnectionsByTag('client');
for (const client of clients) {
  const request = typedSocket.createTypedRequest<INotifyRequest>('notify', client);
  await request.fire({ message: 'Hello from server!' });
}
```

> **Note:** When using SmartServe, the WebSocket transport is managed by SmartServe. TypedSocket acts as a convenience layer for finding connections and sending server-initiated requests.

Client Setup

Connect to the WebSocket server from a client:

```
import { TypedSocket } from '@api.global/typedsocket';
import * as typedrequest from '@api.global/typedrequest';

// Create a router for handling server-initiated requests (if needed)
const clientRouter = new typedrequest.TypedRouter();

// Connect to the server
const client = await TypedSocket.createClient(
  clientRouter,
  'http://localhost:3000'
);
```

Using Window Location (Browser)

In browser environments, you can automatically use the current page's origin:

```
const client = await TypedSocket.createClient(
  clientRouter,
```

```
  TypedSocket.useWindowLocationOriginUrl()
);
```

Sending Requests

Client to Server

```
const request = client.createTypedRequest<IGreetingRequest>('greet');
const response = await request.fire({
  name: 'World',
});

console.log(response.message); // "Hello, World! □□
```

Server to Client

The server can also initiate requests to connected clients:

```
// When only one client is connected, it's automatically selected
const request = server.createTypedRequest<IGreetingRequest>('greet');
const response = await request.fire({
  name: 'Client',
});

// For multiple clients, specify the target connection
const connection = await server.findTargetConnection(async (conn) => {
  // Your filter logic here
  return true;
});
const targetedRequest = server.createTypedRequest<IGreetingRequest>('greet', connection);
```

Connection Tagging

Tag connections for organized, targeted communication:

```
// Client side: add a tag
interface IUserTag extends typedrequestInterfaces.ITag {
  name: 'userRole';
```

```
  payload: 'admin' | 'user' | 'guest';
}

client.addTag<IUserTag>('userRole', 'admin');
```

```
// Server side: find connections by tag
const adminConnections = await server.findAllTargetConnectionsByTag<IUserTag>(
  'userRole',
  'admin'
);

// Send to all admins
for (const conn of adminConnections) {
  const request = server.createTypedRequest<INotificationRequest>('notify', conn);
  await request.fire({ message: 'Admin notification' });
}

// Find a single connection
const firstAdmin = await server.findTargetConnectionByTag<IUserTag>('userRole', 'admin');
```

Event Handling

Subscribe to connection status events:

```
client.eventSubject.subscribe((status) => {
  console.log('Connection status:', status);
});

server.eventSubject.subscribe((status) => {
  console.log('Server connection event:', status);
});
```

Cleanup

Properly close connections when done:

```
// Client
await client.stop();
```

```
// Server
await server.stop();
```

API Reference

TypedSocket

Static Methods

Method	Description
`createServer(router)`	Creates a WebSocket server (defaults to port 3000).
`createClient(router, serverUrl, alias?)`	Creates a WebSocket client that connects to the specified server URL.
`fromSmartServe(smartServe, router)`	Creates a TypedSocket bound to an existing SmartServe instance.
`useWindowLocationOriginUrl()`	Returns the current window location origin (browser only).

Instance Properties

Property	Description
`side`	Whether this instance is a `'server'` or `'client'`.
`typedrouter`	The TypedRouter instance handling requests.
`eventSubject`	RxJS Subject for connection status events.

Instance Methods

Method	Description
`createTypedRequest(method, targetConnection?)`	Creates a typed request for the specified method.
`addTag(name, payload)`	Adds a tag to the client connection (client-side only).
`findAllTargetConnections(filterFn)`	Finds all connections matching the filter (server-side only).
`findTargetConnection(filterFn)`	Finds the first connection matching the filter (server-side only).
`findAllTargetConnectionsByTag(key, payload?)`	Finds all connections with the specified tag.

Method	Description
`findTargetConnectionByTag(key, payload?)`	Finds the first connection with the specified tag.
`stop()`	Closes the WebSocket connection.

License and Legal Information

This repository contains open-source code licensed under the MIT License. A copy of the license can be found in the LICENSE file.

Please note: The MIT License does not grant permission to use the trade names, trademarks, service marks, or product names of the project, except as required for reasonable and customary use in describing the origin of the work and reproducing the content of the NOTICE file.

Trademarks

This project is owned and maintained by Task Venture Capital GmbH. The names and logos associated with Task Venture Capital GmbH and any related products or services are trademarks of Task Venture Capital GmbH or third parties, and are not included within the scope of the MIT license granted herein.

Use of these trademarks must comply with Task Venture Capital GmbH's Trademark Guidelines or the guidelines of the respective third-party owners, and any usage must be approved in writing. Third-party trademarks used herein are the property of their respective owners and used only in a descriptive manner, e.g. for an implementation of an API or similar.

Company Information

Task Venture Capital GmbH Registered at District Court Bremen HRB 35230 HB, Germany

For any legal inquiries or further information, please contact us via email at hello@task.vc.

By using this repository, you acknowledge that you have read this section, agree to comply with its terms, and understand that the licensing of the code does not imply endorsement by Task Venture Capital GmbH of any derivative works.

Revision #7
Created 2026-03-28 10:48:09 UTC by foss.global Team
Updated 2026-03-28 12:13:18 UTC by foss.global Team