readme.md for @api.global/typedrequest

A TypeScript library for making **fully typed request/response cycles** across any transport — HTTP, WebSockets, broadcast channels, or custom protocols. Define your API contract once as a TypeScript interface, then use it on both client and server with compile-time safety, automatic routing, middleware chains, virtual streams for real-time data, and built-in traffic monitoring hooks.

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.

Install

```
pnpm install @api.global/typedrequest
```

You'll also want the interfaces package for defining stream types:

```
pnpm install @api.global/typedrequest-interfaces
```

Usage

All examples use ESM imports and TypeScript.

```
import {
  TypedRequest,
  TypedHandler,
```

```
  TypedRouter,
  TypedTarget,
  VirtualStream,
  TypedResponseError,
} from '@api.global/typedrequest';
```

□□ Define Your API Contract

Every request/response pair is described by a simple interface extending `ITypedRequest`:

```
interface IGetUser {
  method: 'getUser';
  request: { userId: string };
  response: { username: string; email: string };
}

interface IAddNumbers {
  method: 'add';
  request: { a: number; b: number };
  response: { result: number };
}
```

The `method` field acts as a discriminator — it's what the router uses to match requests to handlers.

□□ Making Typed Requests (Client Side)

`TypedRequest` fires a typed request against an HTTP endpoint or a `TypedTarget`:

```
// Against an HTTP endpoint
const getUser = new TypedRequest<IGetUser>('https://api.example.com/rpc', 'getUser');
const user = await getUser.fire({ userId: 'user-123' });
console.log(user.username); // fully typed!

// With response caching
const cachedUser = await getUser.fire({ userId: 'user-123' }, true);
```

□□ Handling Requests (Server Side)

`TypedHandler` processes a specific method and returns a typed response:

```
const addHandler = new TypedHandler<IAddNumbers>('add', async (req) => {
  return { result: req.a + req.b };
});
```

The second argument to the handler function is an optional `TypedTools` instance that gives access to guard validation and transport-layer context:

```
const secureHandler = new TypedHandler<IGetUser>('getUser', async (req, tools) => {
  // Access transport-layer context (e.g., authenticated user info)
  const peer = tools.localData.peer;

  // Validate with guards
  await tools.passGuards([myAuthGuard], req);

  return { username: 'Alice', email: 'alice@example.com' };
});
```

□□ Routing Requests

`TypedRouter` dispatches incoming requests to the correct handler based on the `method` field:

```
const router = new TypedRouter();
router.addTypedHandler(addHandler);
router.addTypedHandler(secureHandler);

// Route an incoming request object
const response = await router.routeAndAddResponse(incomingTypedRequest);
```

Routers are **composable** — you can nest them to build modular API architectures:

```
const coreRouter = new TypedRouter();
const authRouter = new TypedRouter();

// Each sub-router manages its own handlers
coreRouter.addTypedHandler(addHandler);
authRouter.addTypedHandler(secureHandler);

// Link them together — requests flow through the entire chain
```

```
const mainRouter = new TypedRouter();
mainRouter.addTypedRouter(coreRouter);
mainRouter.addTypedRouter(authRouter);
```

□□ Middleware

Add middleware functions to a `TypedRouter` that run **before** any handler on that router executes. Middleware is great for authentication, logging, rate limiting, or input validation:

```
const router = new TypedRouter();

// Add authentication middleware
router.addMiddleware(async (typedRequest) => {
  const token = typedRequest.localData?.authToken;
  if (!token || !isValidToken(token)) {
    throw new TypedResponseError('Unauthorized', { reason: 'invalid_token' });
  }
});

// Add logging middleware
router.addMiddleware(async (typedRequest) => {
  console.log(`Processing ${typedRequest.method}`);
});

// Handlers are only reached if all middleware passes
router.addTypedHandler(secureHandler);
```

Middleware functions receive the full `ITypedRequest` object and run in the order they were added. Throw a `TypedResponseError` from any middleware to reject the request before it reaches the handler.

□□ Custom Targets with TypedTarget

For non-HTTP transports (WebSockets, broadcast channels, IPC), use `TypedTarget` with a custom post function:

```
// Synchronous target (post returns the response directly)
const target = new TypedTarget({
  postMethod: async (payload) => {
```

```
    // Send via your custom transport and return the response
    return await myWebSocket.sendAndWait(payload);
  },
});

const request = new TypedRequest<IGetUser>(target, 'getUser');
const user = await request.fire({ userId: 'user-123' });
```

For **async targets** where the response arrives separately (e.g., WebSocket push), pair a `TypedTarget` with a `TypedRouter`:

```
const router = new TypedRouter();

const asyncTarget = new TypedTarget({
  postMethodWithTypedRouter: async (payload) => {
    // Fire-and-forget — response will arrive via router
    mySocket.send(JSON.stringify(payload));
  },
  typedRouterRef: router,
});

// When the response arrives later, route it back:
mySocket.onMessage((data) => {
  router.routeAndAddResponse(JSON.parse(data));
});
```

□□ Virtual Streams

`VirtualStream` enables **bidirectional binary streaming** over any transport that supports typed requests. Data is automatically chunked with backpressure control:

```
import type { IVirtualStream } from '@api.global/typedrequest-interfaces';

// Define a streaming endpoint
interface IFileUpload {
  method: 'uploadFile';
  request: { filename: string; dataStream: IVirtualStream };
  response: { bytesReceived: number };
}
```

```
// Client side: create and send a stream
const uploadStream = new VirtualStream<Uint8Array>();
const upload = new TypedRequest<IFileUpload>('https://api.example.com/rpc', 'uploadFile');

const responsePromise = upload.fire({
  filename: 'data.bin',
  dataStream: uploadStream,
});

// Send data through the stream
await uploadStream.sendData(new TextEncoder().encode('Hello, World!'));
await uploadStream.close();

const result = await responsePromise;
console.log(result.bytesReceived);
```

```
// Server side: receive and process the stream
const uploadHandler = new TypedHandler<IFileUpload>('uploadFile', async (req) => {
  let total = 0;
  // Read chunks from the stream
  const chunk = await req.dataStream.fetchData();
  total += chunk.byteLength;

  return { bytesReceived: total };
});
```

VirtualStreams also integrate with the Web Streams API:

```
// Pipe a ReadableStream into a VirtualStream
await virtualStream.readFromWebstream(readableStream);

// Pipe a VirtualStream into a WritableStream
await virtualStream.writeToWebstream(writableStream);
```

⚠ Error Handling

Throw `TypedResponseError` inside handlers to send structured errors back to the caller:

```
const handler = new TypedHandler<IGetUser>('getUser', async (req) => {
  const user = await db.findUser(req.userId);
  if (!user) {
    throw new TypedResponseError('User not found', { userId: req.userId });
  }
  return { username: user.name, email: user.email };
});
```

On the client side, `TypedResponseError` is thrown when the server responds with an error:

```
try {
  await getUser.fire({ userId: 'nonexistent' });
} catch (err) {
  if (err instanceof TypedResponseError) {
    console.error(err.errorText); // 'User not found'
    console.error(err.errorData); // { userId: 'nonexistent' }
  }
}
```

□□ Traffic Monitoring Hooks

Monitor all TypedRequest traffic with global or per-router hooks:

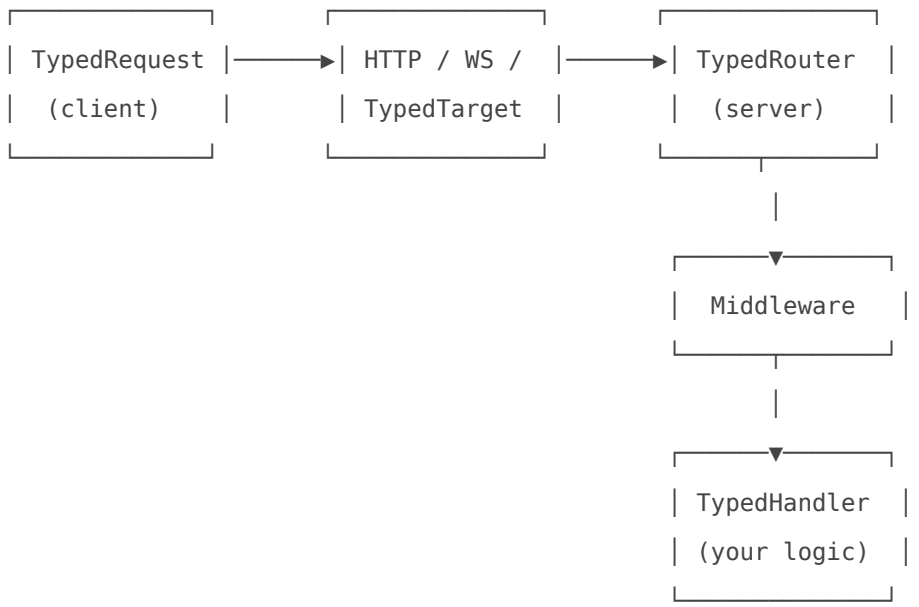
```
// Global hooks — apply to ALL routers and requests across bundles
TypedRouter.setGlobalHooks({
  onOutgoingRequest: (entry) => {
    console.log(`→ ${entry.method} [${entry.correlationId}]`);
  },
  onIncomingResponse: (entry) => {
    console.log(`← ${entry.method} took ${entry.durationMs}ms`);
  },
  onIncomingRequest: (entry) => {
    console.log(`⇒ handling ${entry.method}`);
  },
  onOutgoingResponse: (entry) => {
    if (entry.error) console.error(`⇐ ${entry.method} error: ${entry.error}`);
  },
});
```

```
// Per-router hooks
router.setHooks({
  onIncomingRequest: (entry) => metrics.trackRequest(entry),
});

// Skip hooks for internal requests (e.g., health checks)
const internalReq = new TypedRequest<IHealthCheck>(target, 'healthCheck');
internalReq.skipHooks = true;
```

Global hooks are shared across all bundles via `globalThis`, making them ideal for application-wide observability.

□□ Architecture Overview



Component	Role
TypedRequest	Fires typed requests against a URL or TypedTarget
TypedTarget	Abstracts the transport layer (HTTP, WebSocket, custom)
TypedRouter	Routes incoming requests to the correct handler; composable via `addTypedRouter()`
TypedHandler	Processes a single method and returns a typed response
Middleware	Pre-handler functions for auth, validation, logging — throw `TypedResponseError` to reject
VirtualStream	Bidirectional binary streaming with backpressure over any supported transport

Component	Role
TypedResponseError	Structured error propagation across the wire
TypedTools	Guard validation and transport-layer context available inside handlers

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 10:48:07 UTC by foss.global Team
Updated 2026-03-28 12:13:16 UTC by foss.global Team