open-archive-collective

abstract :

this paper contains how the project open-archive-collective works and what it is about

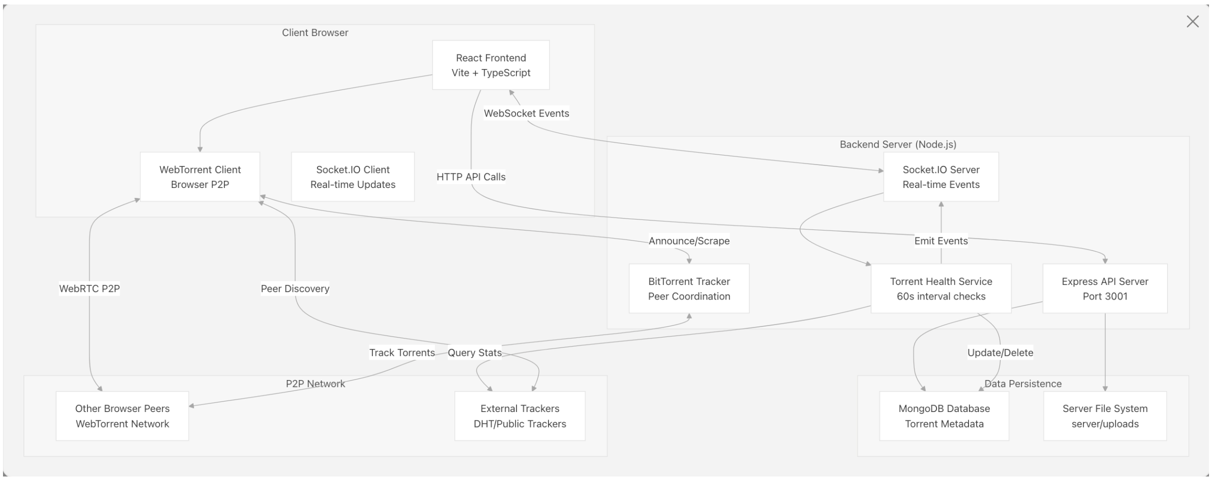
open-archive-collective is a decentralised peer-to-peer file sharing platform that works under the guiding principal that knowledge should be free and available to all

the platform is designed for the liberation of books , research papers , softwares and more and their free distribution

When a user submits a file using the Contribute website, their browser starts a torrent and is the first "First Peer" to seed the file. The file itself is never sent to the backend; just the magnet URI, info hash, and descriptive metadata are. The file can then be found by other users via the Library page, and they can download or stream it straight from peer devices.

system architecture:

The Open Archive Collective uses a hybrid peer-to-peer paradigm in which all file content is delivered exclusively over peer-to-peer connections, with a centralized backend acting as a metadata library. The server stores just cryptographic hashes and metadata, functioning as a "zero-knowledge" directory.



Architecture Principles:

Principle	Implementation	Code Reference
Zero-Knowledge Server	Backend stores only `magnetURI` , `infoHash` , and metadata	server/models/Torrent.js
Client-Side Seeding	Files seeded directly from browser via `client.seed()`	src/pages/Contribute.tsx
Real-Time Monitoring	Background service checks torrent health every 60 seconds	server/index.js29-35
Event Broadcasting	Socket.IO emits `torrent-updated` and `torrent-deleted` events	server/services/torrentHealth.js
Pure P2P Distribution	All file transfers via WebRTC, bypassing server entirely	src/components/TorrentPlayer.tsx

Component Interaction Model

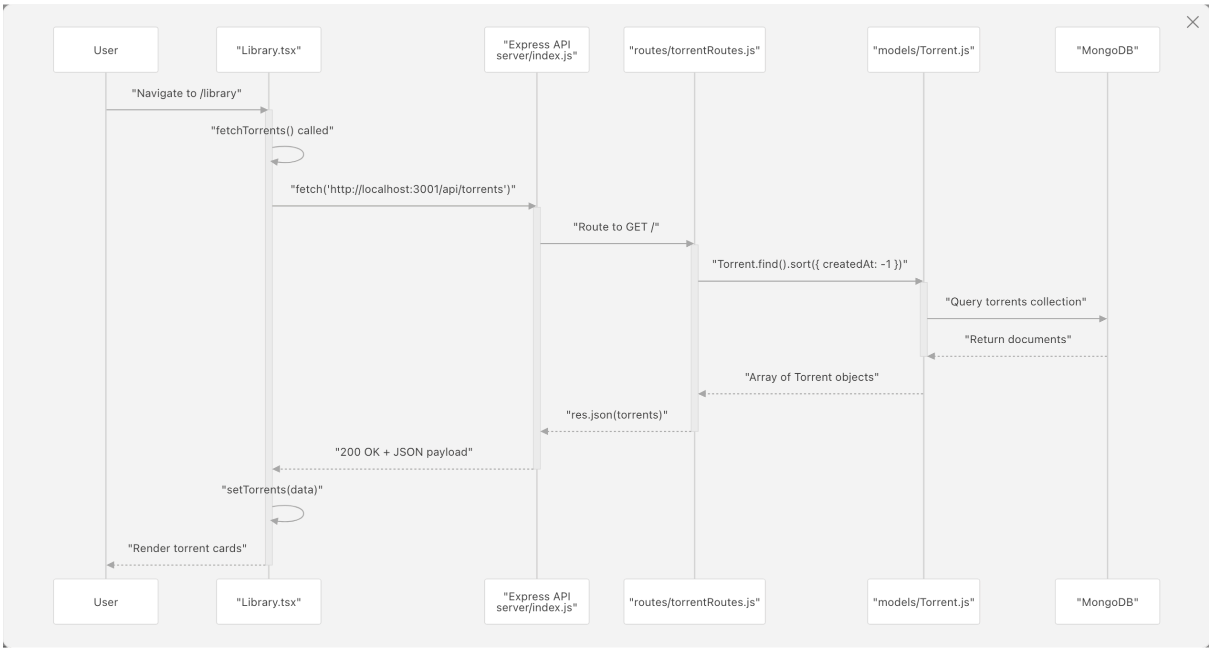
Through well specified interfaces, the system coordinates interactions between frontend elements, backend API routes, Socket.IO events, and the P2P network.

API Endpoint & Event Mapping

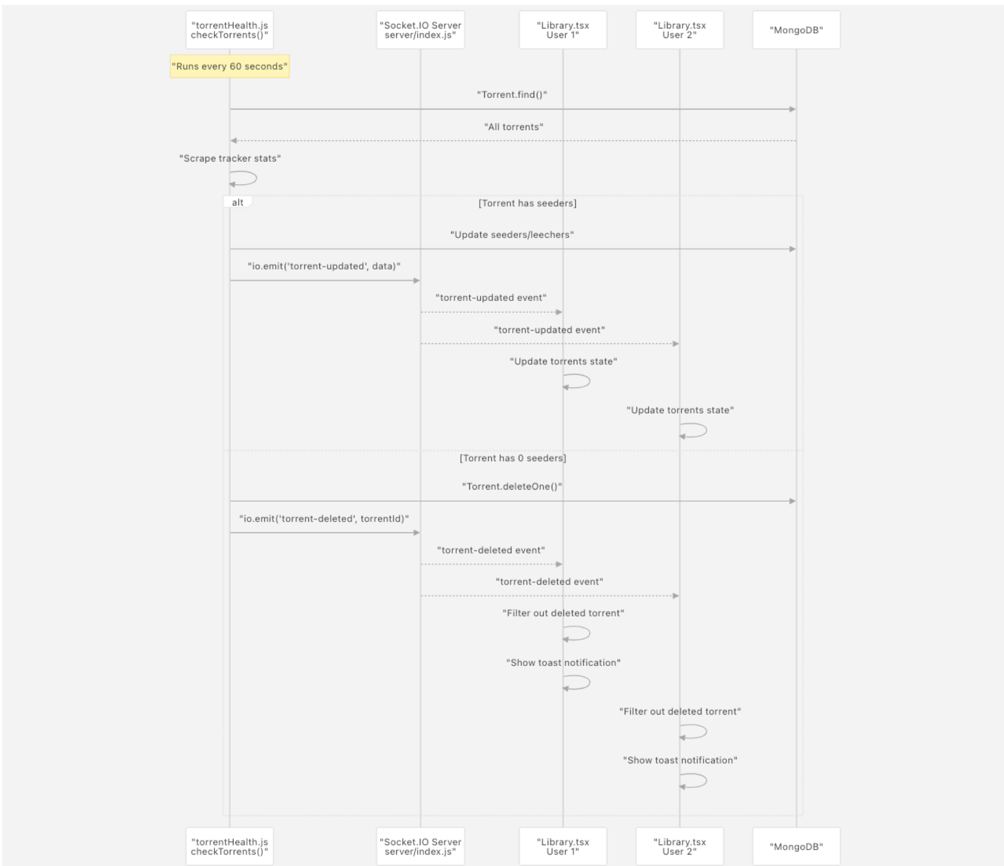
Frontend Component	Backend Route/Event	Data Operation	Purpose
`Library.tsx:fetchTorrents()`	`GET /api/torrents`	`Torrent.find()`	List all torrents
`Contribute.tsx:handleSubmit()`	`POST /api/torrents/upload`	`Torrent.create()`	Register new torrent metadata

Frontend Component	Backend Route/Event	Data Operation	Purpose
`Library.tsx` Socket listener	`torrent-updated` event	N/A (broadcast)	Update seeder/leecher counts in real-time
`Library.tsx` Socket listener	`torrent-deleted` event	N/A (broadcast)	Remove dead torrents from UI
`TorrentPlayer` component	N/A (pure P2P)	`client.add(magnetURI)`	Stream file from peers

Metadata Query Flow (Initial Page Load)



Real-Time Update Flow (Socket.IO Events)



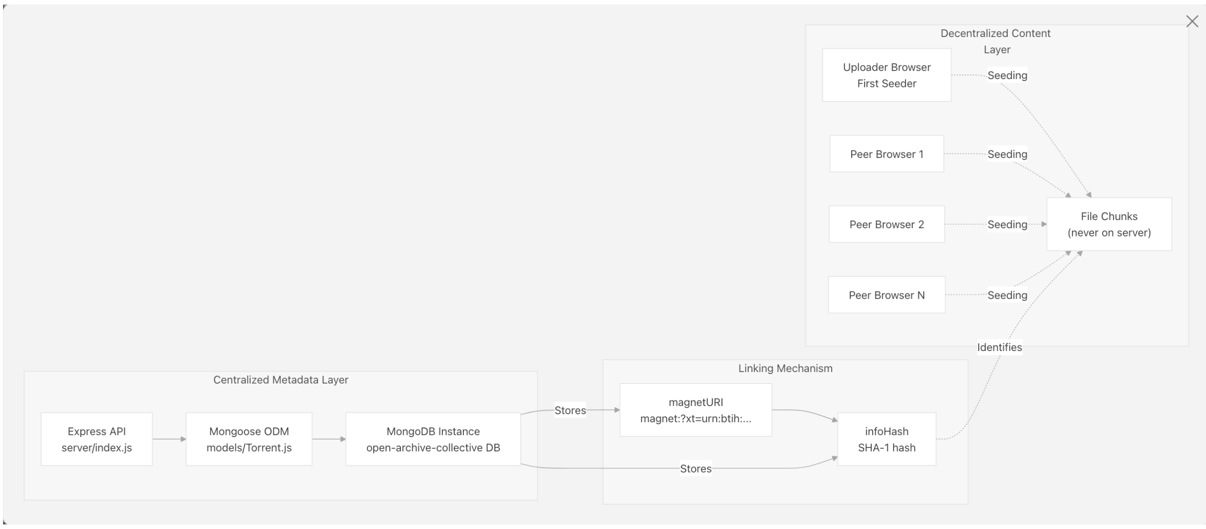
Socket.IO Event Handlers:

Event	Emitted By	Received By	Payload	Action
`torrent-updated`	server/services/torrentHealth.js	src/pages/Library.tsx39-45	`{ _id, seeders, leechers }`	Update torrent stats in state
`torrent-deleted`	server/services/torrentHealth.js	src/pages/Library.tsx47-53	`torrentId` (string)	Remove torrent from state, show toast

Storage Architecture

The system employs a dual-storage approach that distinguishes searchable metadata from distributed content

Dual Storage Pattern



MongoDB Schema Structure

```
Torrent Document:
├── _id: ObjectId (auto-generated)
├── title: String (required)
├── description: String (optional)
├── fileName: String (required)
├── fileSize: Number (required, bytes)
├── magnetURI: String (required)
├── infoHash: String (required, unique index)
├── category: String (default: "General")
├── uploadedBy: String (default: "Anonymous")
└── createdAt: Date (auto-generated timestamp)
```

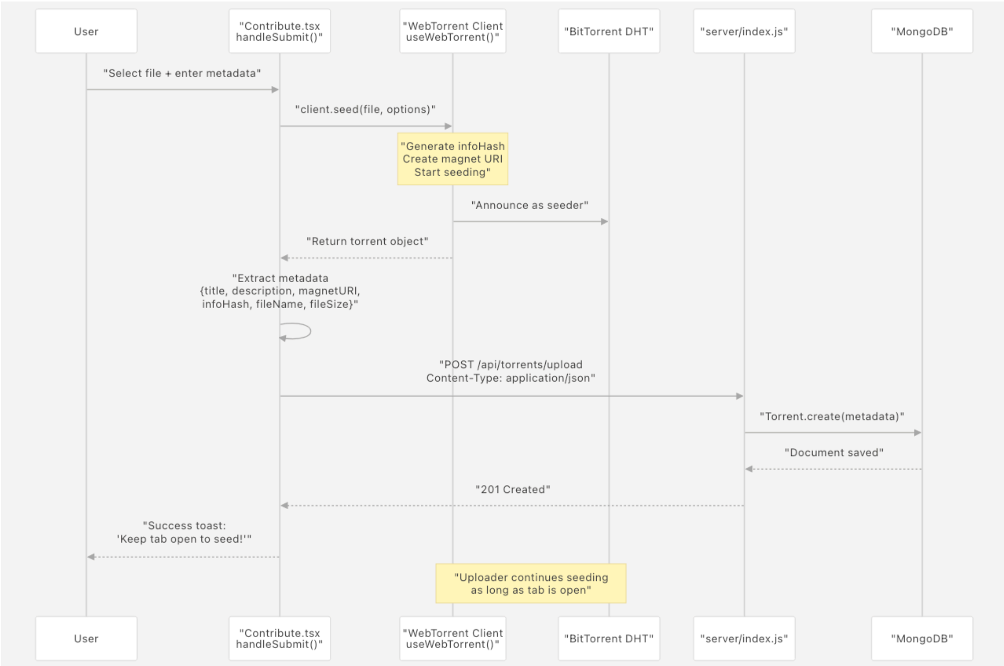
Storage Guarantees:

Layer	Data Stored	Persistence Mechanism	Accessibility
MongoDB	Metadata only (title, description, magnetURI, infoHash)	Disk-backed database	Queryable via API
P2P Network	Actual file content (binary data)	Seeder memory/disk	Accessible via BitTorrent protocol
Browser IndexedDB	WebTorrent cache	Browser storage	Per-user local cache

Important Architectural Choice: There are no uploads or file storage folders in the server directory. Although express.json() middleware is included in the server/index.js10-11 setup, file upload processing is purposefully left out. The zero-knowledge concept is thereby upheld.

Data Flow Patterns

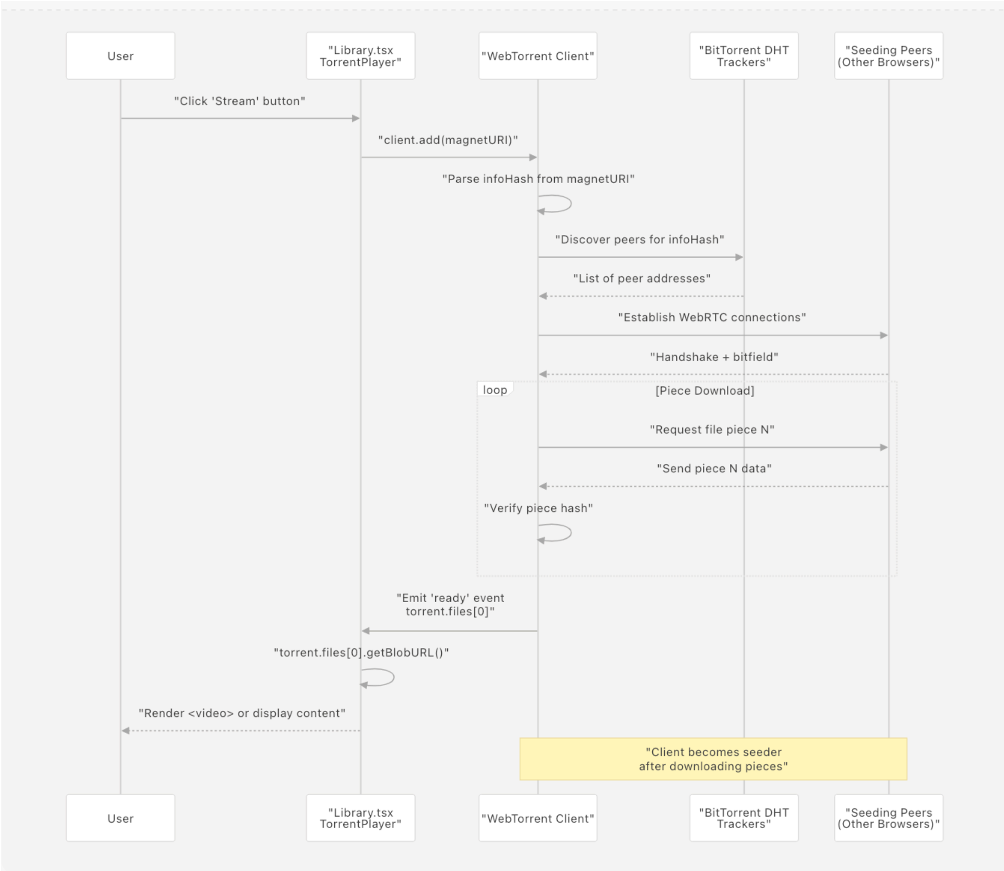
Upload Flow (Contribute Page)



Upload Flow Characteristics:

- The file stays in the browser's memory and is never sent to the server.
- WebTorrent generates the magnetURI client-side.
- Only six metadata fields are sent to the backend.
- Uploader instantly becomes "First Peer"

Download/Stream Flow (Library Page)



Streaming Characteristics:

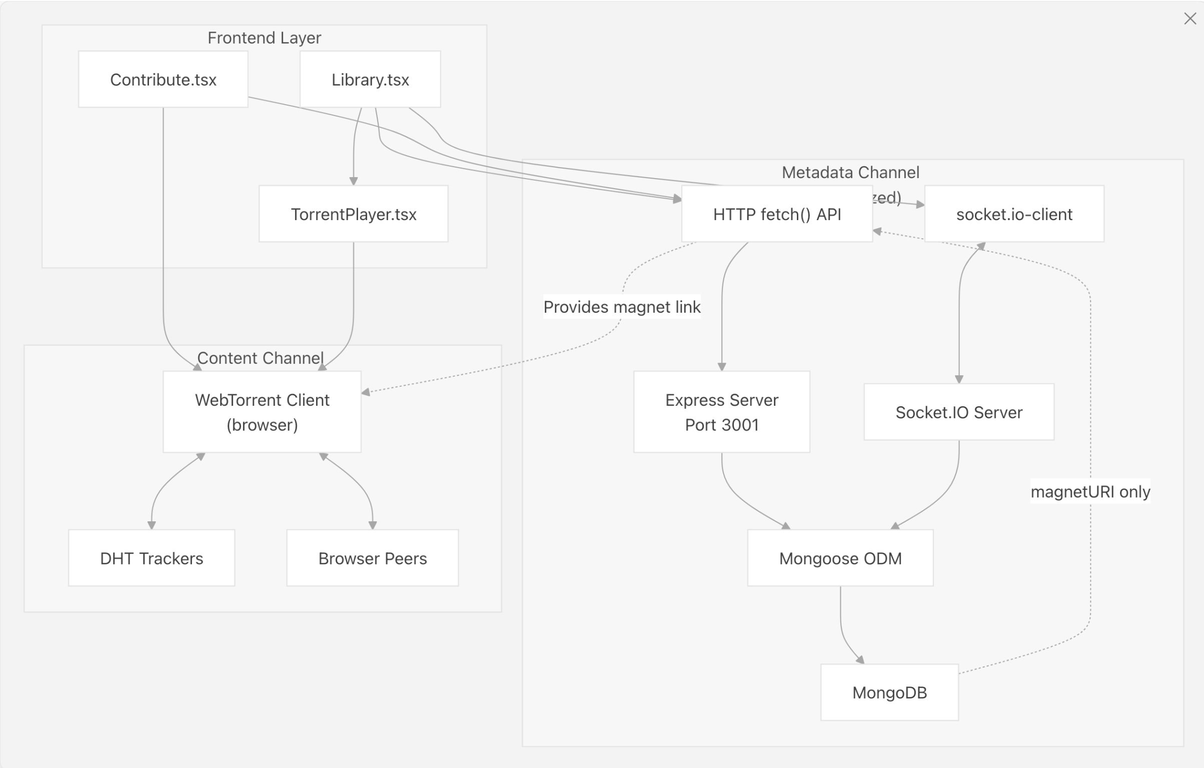
- magnetURI obtained with an API from MongoDB
- DHT-based peer finding without server participation
- Streaming prior to completion is made possible by progressive download.
- The client becomes a seeder automatically ("tit-for-tat").

Network Communication Architecture

Three separate, never-intersecting network channels are used by the system to function:

Channel	Protocol	Port	Purpose	Client Component	Server Component
Metadata API	HTTP/1.1	3001	CRUD operations for torrent metadata	`fetch()` in src/pages/Library.tsx62	Express at server/index.js9-16
Real-Time Events	WebSocket (Socket.IO)	3001	Live torrent health updates	Socket.IO client at src/pages/Library.tsx33	Socket.IO server at server/index.js17-22
File Transfer	WebRTC	Ephemeral	Direct peer-to-peer file data	WebTorrent client in src/components/TorrentPlayer.tsx	Other browser peers (no server)

Dual-Channel Data Flow



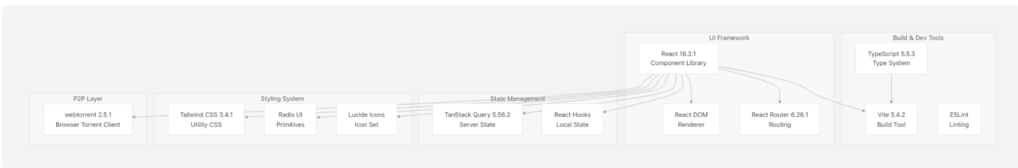
Critical Architectural Separation:

There is no P2P data channel access for the Express server at server/index.js. Only WebRTC connections between browser peers are used to transfer file content. The server's function is restricted to:

1. **Storing metadata** (magnetURI, infoHash, title, description)
2. **Broadcasting health events** (via Socket.IO)
3. **Serving as a searchable directory** (via REST API)

Technology Stack Integration

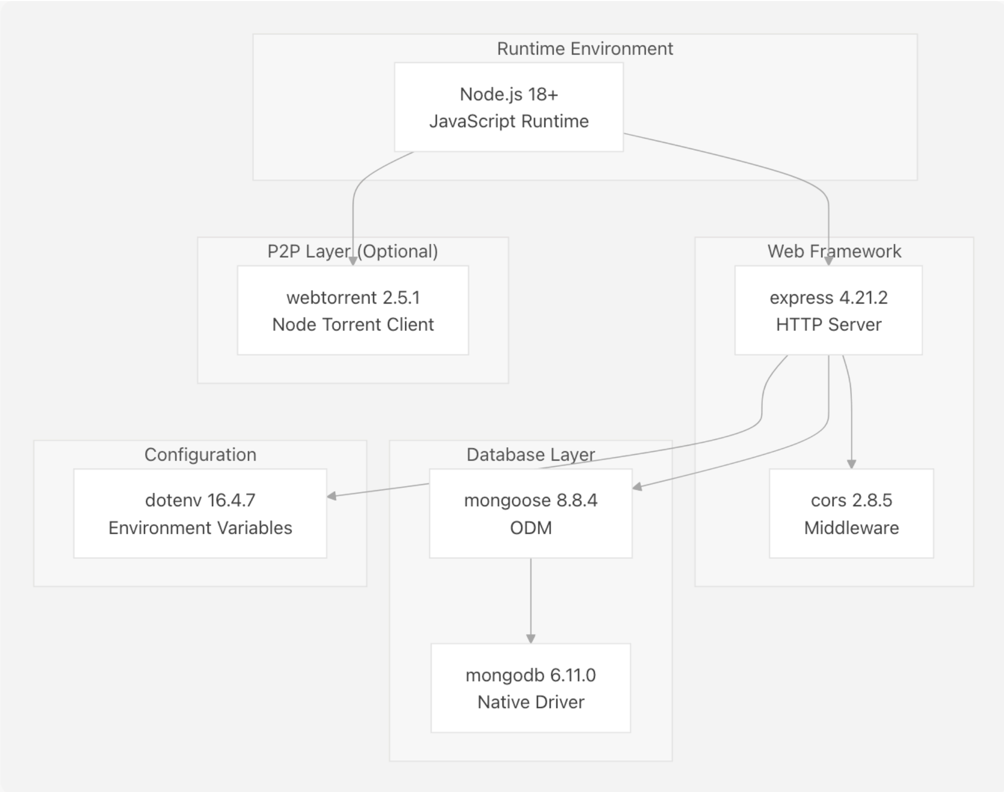
Frontend Stack Composition



Key Dependencies:

- **webtorrent:** Enables browser-based BitTorrent functionality without plugins
- **@tanstack/react-query:** Manages server state synchronization for API calls
- **@radix-ui/*:** Provides accessible UI primitives (Dialog, Input, Button)
- **tailwindcss:** Utility-first CSS framework for rapid styling

Backend Stack Composition



Backend Characteristics:

- **Minimal Dependencies:** Only essential libraries for API and database
- **No File Processing:** Absence of `multer` configuration in active use
- **Mongoose ODM:** Provides schema validation and query building at server/models/Torrent.js
- **Environment-Driven Config:** server/index.js1 loads MongoDB URI from `.env`

Deployment Architecture

Multi-Container Setup

Three services are orchestrated by the system using Docker Compose:

```
docker-compose.yml:
├── frontend (Nginx serving static build)
├── backend (Node.js Express server)
└── mongodb (MongoDB container)
```

Container Communication:

- Frontend → Backend: HTTP requests to `http://backend:5001`
- Backend → MongoDB: Mongoose connection to `mongodb://mongodb:27017`
- Clients → P2P Network: Direct WebRTC connections (bypass containers)

Port Mapping:

Service	Internal Port	Exposed Port	Purpose
Frontend	80	8080	Serve React SPA
Backend	5001	5001	Expose API endpoints
MongoDB	27017	27017	Database access