Gather Clone - Architecture & Implementation Guide

Project Overview

A real-time multiplayer browser application that recreates the core experience of Gather.town. Built with Phaser.js for 2D rendering and Socket.io for WebSocket communication, the app enables users to navigate a top-down 2D environment with custom avatars, see other players moving in real-time, and interact through proximity-based voice/video chat powered by WebRTC.

Tech Stack:

- Frontend: Phaser.js (2D game engine), Socket.io-client, WebRTC
- Backend: Node.js, Express, Socket.io
- Database: PostgreSQL (future)
- Map Editor: Tiled Map Editor

Current Constraints:

- Single map
- 14 users maximum
- Authentication system already implemented

Architecture Recommendation: Don't Build Database Yet

Why Skip Database Initially?

Current State:

- ✅ Frontend auth working
- ✅ Single map
- ✅ 14 users max
- ❌ No real-time multiplayer yet
- ❌ No game rendering yet

The Problem with Database First: Building a database schema now would be premature optimization. You don't yet know:

- What data actually needs to persist vs live in memory
- How positions will update (60 times/second - too fast for DB)

- What user data is critical vs temporary
- How rooms/maps will scale

Key Principle: Databases are for persistence, not real-time game state.

Phase 1: Get Multiplayer Working (Week 1-2)

Step 1: Basic Game Rendering ✓ Priority 1

Goal: See your character move on the map

Tasks:

- Load Tiled map in Phaser
- Render player sprite at spawn point
- WASD/Arrow key movement
- Collision with boundaries layer
- Camera follows player

Deliverable: Single player game works

Step 2: In-Memory Multiplayer ✓ Critical Path

Goal: See other players in real-time

Tasks:

- Socket.io server tracks players in memory
- Broadcast positions to all connected clients
- Render remote players as sprites
- Test with 2+ browser tabs

Server Implementation (In-Memory Only):

javascript

```
// Server-side (NO DATABASE YET)
const activePlayers = {}; // Just store in RAM

socket.on('join', (data) => {
  activePlayers[socket.id] = {
    id: socket.id,
    username: data.username,
    x: data.x,
    y: data.y,
    connectedAt: Date.now()
  };
});
```

Deliverable: 2+ tabs see each other move

Step 3: Smooth Multiplayer

Goal: Movement feels native, not janky

Tasks:

- Client-side prediction (move immediately)
- Interpolation for remote players
- Throttle position updates (20-30/sec)
- Handle disconnections gracefully

Deliverable: Feels smooth with 5+ players

Phase 2: Persistence Layer (Week 3-4)

Step 4: User Persistence

Goal: Remember users across sessions

Tasks:

- Store user profiles (username, avatar, preferences)
- Link auth system to game profiles
- Save last known position (for respawn)

Database Schema (Simple Start):

```sql
-- PostgreSQL example
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  auth_id VARCHAR(255) UNIQUE NOT NULL,  -- From your auth system
  username VARCHAR(50) NOT NULL,
  avatar_url VARCHAR(255),
  last_x INT DEFAULT 100,
  last_y INT DEFAULT 100,
  last_map VARCHAR(50) DEFAULT 'main',
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);

-- Don't store real-time positions in DB!
-- Those stay in memory and update 30 times/second
```

Step 5: Session Management

Goal: Reconnect users smoothly

Tasks:

- Save session on join (user_id → socket_id mapping)
- Restore position on reconnect
- Handle page refresh gracefully

Phase 3: State Persistence (Week 5+)

Step 6: World State (Only if needed)

Goal: Remember world changes

Tasks:

- Interactive objects state (is TV on/off?)
- Shared content (whiteboard drawings)
- Room configurations

Schema:

```sql
CREATE TABLE world_objects (
  id UUID PRIMARY KEY,
  map_name VARCHAR(50) NOT NULL,
  object_type VARCHAR(50) NOT NULL,  -- 'tv', 'whiteboard', 'portal'
  x INT NOT NULL,
  y INT NOT NULL,
  state JSONB,  -- Flexible object-specific data
  updated_at TIMESTAMP DEFAULT NOW()
);
```

What NOT to Store in Database

Never store in DB:

- ❌ Real-time positions (updates 30-60 times/second)
- ❌ Active player list (changes constantly)
- ❌ Current room occupancy (volatile data)
- ❌ WebRTC connection state (peer-to-peer)

Store these in memory only:

```javascript
// Server RAM (Redis optional for scaling)
const gameState = {
  activePlayers: {},      // socket_id → player data
  rooms: {                // room_name → player list
    'main': { players: [], objects: {} }
  }
};
```

Recommended Architecture

```
┌─────────────────────────────────────────┐
│  Frontend (Browser)                     │
│  ├── Auth System (existing)             │
│  ├── Game Client (Phaser + Socket.io)   │
│  └── WebRTC (future)                    │
└─────────────────────────────────────────┘
```

```
          │
          ▼ (WebSocket)
┌─────────────────────────────────────────┐
│ Backend (Node.js)                       │
│  ├─ Express Server                      │
│  ├─ Socket.io (real-time game state)    │
│  │   └─ In-Memory Player Positions      │
│  └─ REST API (persistence)              │
│      └─ Database (user profiles, world) │
└─────────────────────────────────────────┘
          │
          ▼
┌─────────────────────────────────────────┐
│ Database (PostgreSQL/MongoDB)           │
│  ├─ Users (profiles, preferences)       │
│  ├─ World Objects (persistent state)    │
│  └─ Analytics (optional)                │
└─────────────────────────────────────────┘
```

Next 7 Days - Concrete Checklist

Day 1-2: Render Game

- □ Load your Tiled map in Phaser
- □ Show player sprite at spawn point
- □ WASD movement working
- □ Collision detection working
- □ **Deliverable:** Single player game works

Day 3-4: Add Multiplayer

- □ Socket.io server running
- □ Server tracks players in `activePlayers` object (memory only)
- □ Broadcast positions to all clients
- □ Render remote players
- □ **Deliverable:** 2+ tabs see each other move

Day 5-6: Smooth It Out

- □ Add interpolation for remote players
- □ Throttle position updates
- □ Add player name labels
- □ Handle disconnects

□ **Deliverable:** Feels smooth with 5+ players

Day 7: Plan Persistence

□ Design DB schema (use template above)
□ Decide: PostgreSQL vs MongoDB
□ Set up ORM (Prisma/TypeORM) or driver
□ **Don't implement yet** - just plan

Database Decision Tree

Use PostgreSQL if:

- You want structured user data
- Need relational queries (user → rooms → objects)
- Want strong consistency
- **Recommended for this project**

Use MongoDB if:

- Flexible object schemas (JSONB-like everywhere)
- Simpler setup for prototyping
- Don't need complex joins

Use Redis (later) if:

- Scaling beyond 1 server
- Need session storage across servers
- Want pub/sub for multi-server Socket.io

Minimal DB Schema (Start Here)

```sql
```

```
-- Just this for now:
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  auth_id VARCHAR(255) UNIQUE NOT NULL,
  username VARCHAR(50) NOT NULL,
  created_at TIMESTAMP DEFAULT NOW()
);

-- Add more columns ONLY when you need them
-- Don't over-engineer early
```

When to Actually Build the Database

Build DB when:

- ✅ Multiplayer works smoothly
- ✅ You want to remember users between sessions
- ✅ You need to save world state

You'll know it's time when you think: *"I wish the server remembered who this user was"*

Implementation Timeline

Week 1-2: Core Multiplayer

- Build game rendering
- Implement in-memory multiplayer
- Add interpolation and smoothing
- **No database yet**

Week 3: User Persistence

- Add PostgreSQL database
- Create users table
- Link auth system to game profiles
- Save/restore last position

Week 4+: World Persistence

- Add world_objects table (if needed)
- Implement interactive objects
- Save shared state

Key Principles

1. **Build the game first, database second**
    - You learn what data matters by building
    - Database schemas are hard to change
    - In-memory is faster for prototyping
2. **Separate real-time from persistent data**
    - Real-time: positions, active players (memory)
    - Persistent: user profiles, world state (database)
3. **Start simple, add complexity when needed**
    - 14 users doesn't need complex persistence yet
    - Add features when you feel the pain of not having them
4. **Test multiplayer early and often**
    - Open 2+ browser tabs every day
    - Test on different devices
    - Get feedback from real users ASAP

Project Structure (Modular)

```
gather-clone/
├── public/
│   ├── index.html
│   ├── js/
│   │   ├── main.js           # Game initialization
│   │   ├── scenes/
│   │   │   ├── BootScene.js      # Asset loading
│   │   │   └── GameScene.js      # Main game scene
│   │   ├── entities/
│   │   │   └── Player.js         # Player class
│   │   ├── managers/
│   │   │   ├── MapManager.js      # Map handling
│   │   │   ├── PlayerManager.js   # Remote players
│   │   │   └── InputManager.js    # Input handling
│   │   ├── network/
│   │   │   └── SocketManager.js   # Socket.io wrapper
│   │   └── utils/
│   │       └── Constants.js       # Configuration
│   └── assets/
│       ├── maps/
│       │   ├── map.json
│       │   └── tileset.png
│       └── sprites/
│           └── player.png
├── server/
│   ├── server.js                  # Entry point
│   ├── game/
│   │   └── GameState.js           # In-memory state
│   ├── network/
│   │   └── SocketHandler.js       # Socket events
│   └── db/
│       ├── connection.js          # DB setup (Week 3+)
│       └── models/
│           └── User.js            # User model (Week 3+)
└── package.json
```

Summary: Your Action Plan

✅ **NOW (Week 1-2):** Get game rendering + in-memory multiplayer working

⏸️ **LATER (Week 3):** Add database for user profiles only

⏸️ **MUCH LATER (Week 4+):** Add world state persistence if needed

---

Resources

Documentation:

- Phaser 3: https://photonstorm.github.io/phaser3-docs/
- Socket.io: https://socket.io/docs/v4/
- Tiled: https://doc.mapeditor.org/
- PostgreSQL: https://www.postgresql.org/docs/

Key Learning Goals:

1. Understand client-side prediction and reconciliation
2. Learn state synchronization patterns
3. Master WebSocket bi-directional communication
4. Know when to use memory vs database

Document Version: 1.0 Last Updated: December 2024