Multiplayer Langton's Ant – Backend Submission Documentation

Project Overview

This project implements a **multiplayer, real-time Langton's Ant simulation** using WebSockets. Players can:

- Place ants on a shared grid.
- Define custom rules for ant behavior.
- See the grid update in real-time every 250 ms.
- Flip tiles in their own color on the shared grid.

The backend described here is a Node.js WebSocket server responsible for:

- Managing the grid state.
- Tracking player ants and rules.
- Handling client events (placing ants, defining rules, flipping tiles).
- Synchronizing game state to all clients in real-time.

Implementation Details

Language and Framework

- **Node.js** (JavaScript)
- **ws** WebSocket library

Design and Architecture

1️⃣ WebSocket Server

- Runs on port **8080**.
- Uses `ws` to handle connections.
- Each client gets a **unique ID** and **random color** on connect.

Data Structures

- **GRID**: a map of cell positions → { color, ownerId }
    - Only stores *non-white* cells.
- **ANTS**: a map of client IDs → ant objects.
    - Includes position (x, y), direction, color, and custom rules.

Random Color Assignment

On connect, each player receives a random hex color:

```
function randomColor() {
  return `#${Math.floor(Math.random() * 0xffffff).toString(16).padStart(6,
"0")}`;
}
```

Simulation Ticking

- Runs every **250 ms** using `setInterval`.
- For each ant:
    - Reads the grid color at its position.
    - Applies its rules (turning left/right).

    - Flips tiles if on own color.
    - Moves forward one cell.
- After moving all ants, the server broadcasts the updated grid to all clients.

Message Types

Type	Purpose
Welcome	Sent to client on connect with ID and color.
placeAnt	Sets/updates the client's ant on the grid.
flipTile	Flips the color of a tile (if owned by client).
setRules	Sets the ant's custom movement rules.
update	Broadcasted to all clients with full grid state.

Rule Handling

- Rules are per-player and validated.
- Only **white** and the player's own color are valid keys.
- Default behavior if no rule is defined: turn right.

Example ant rule:

```
{
  "white": "R",
  "#aabbcc": "L"
}
```

Collision Handling

- Current implementation:

    - All ants move simultaneously.
    - No advanced collision resolution — overlapping ants can "fight" over tiles.
    - Simplicity was chosen for speed of implementation.

- Future work could include consistent resolution (e.g. priority, randomized resolution, queued moves).

Disconnect Handling

- On disconnect:
    - Ant is removed from the simulation.
    - Player color and ID are forgotten (on reconnect, new color assigned).
- No persistence between sessions.

Technical Choices & Rationale

- **Node.js**: Lightweight, great for fast real-time servers
- **ws**: Minimal WebSocket library with excellent performance
- **Hashmap Grid**: Efficient for sparse grids (only stores occupied cells)
- **Broadcast All State**: Simpler synchronization logic for small-scale play (5–10 players)

Trade-offs & Future Improvements

Trade-offs:

- No persistent storage: grid resets on server restart

- No diff-based updates: always broadcasts entire grid
- No reconnect logic: reconnecting assigns a new clientId and color

Future Improvements:

- Persist grid and ant state to disk or database
- Send only changed cells (diff) to reduce bandwidth
- Use a more robust ID system to support reconnects
- Add authentication and user management

Testing

Manual Testing Performed

- Multiple clients connecting and placing ants
- Rule definition and tile flipping (including *"Highway Test"* example)
- Real-time synchronization of grid updates across all clients
- Graceful handling of client disconnects and removals from state
- **Special Rule Configurations**:
    - Used *known, deterministic rule sets* (e.g., classic Langton's Ant "L-R" patterns) to verify expected emergent behaviors.
    - Ensured that consistent rules produced reproducible movement paths and tile flips.
    - Validated color-based rules by assigning controlled test colors and observing rule branching.

Notes on Test Strategy

- Tested both single-player and multi-client scenarios.
- Verified tile ownership logic (no unauthorized flips of other players' tiles).
- Observed movement over several hundred ticks to confirm stability and correct direction changes.
- Used **simplified test grids** to visualize rule outcomes clearly before scaling up.
-

Edge Cases:

- Invalid JSON messages handled via `try/catch`
- Invalid rule keys filtered before storing

Frontend (React) App

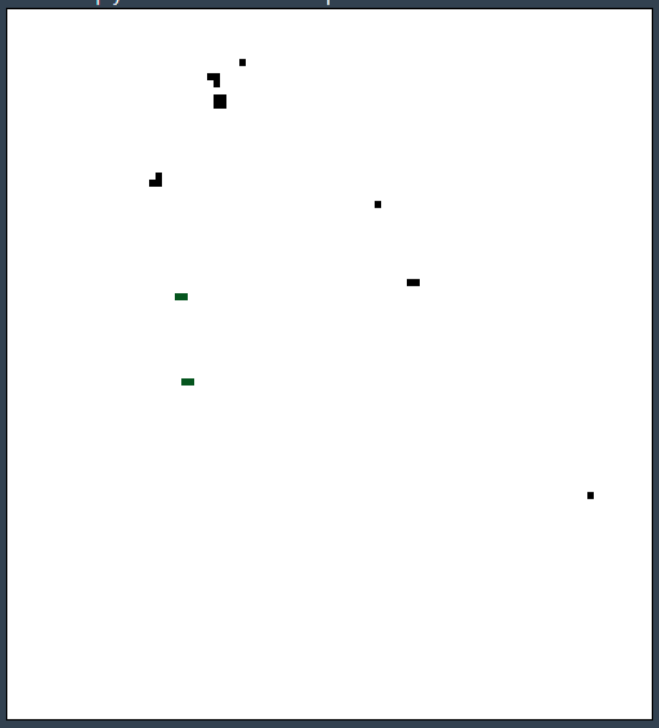
This project also includes a **React + Typescript - based frontend with tailwind styling** designed to interact with the WebSocket server and provide an intuitive user experience.

Langton's Ant Multiplayer

Click to flip. Shift+Click to place your ant.

Your Color:

Click to flip your tile. Shift+Click to place ant.



Define Ant Rules

Select color | R (Right) | Add Rule

Current Rules:

No rules defined.

Send Rules

Features

Canvas Grid Rendering

- Uses HTML `<canvas>` for performance.
- Supports large grids (e.g., 1000x1000 cells).
- Displays ants' traces in unique player colors.
- Zoom and pan optional in future improvements.

The frontend canvas defaults to 500px with 5px cells, showing a 100×100 visible area at a time. This keeps the UI manageable while the server supports unbounded grid sizes (1000×1000+). Users can adjust CELL_SIZE and CANVAS_SIZE in configuration to zoom in/out or show larger areas.

Ant Placement & Tile Flipping

- Click a cell to place your ant.
- Subsequent clicks on your own tiles flip them between white and your color.
- Tiles owned by other players are not interactive.

Real-Time Sync

- Connects to server via WebSockets.
- Receives and renders the full grid state every 250 ms.
- Automatically updates view to show other players' ant movements.

Player Customization UI

- Shows assigned player color on connection.
- Form to define *custom rules*:
    - Example:

        - "White → R (turn right)"
        - "YourColor → L (turn left)"
- Rules are validated and sent to the server.

Status Panel

- Displays:
    - Player ID
    - Assigned color
    - Current rules
- Optional: list of connected players.

WebSocket Integration

- On connect:
    - Receives `welcome` message with `clientId` and assigned color.
    - Stores them in React state.
- Sends messages to server:
    - `placeAnt` with position
    - `flipTile` on click
    - `setRules` on form submission
- Listens for:
    - `update` messages with new grid state
    - Rerenders canvas with received data.

Component Structure (Example)

```
<App>
 ├── <GridCanvas />
 ├── <RuleForm />
 └── <StatusPanel />
```

- **CanvasGrid**:
    - Renders the grid from server data.
    - Handles clicks for placing/flipping.
- **Controls**:
    - Let users define rules.
    - Sends `setRules` to the server.
- **StatusPanel**:
    - Shows player color and current rules.

Technical Stack

- **React** (Vite)
- **WebSocket API** (native browser support)
- **Canvas** for grid rendering

Known Limitations

- No persistent reconnect logic (new color on reconnect).
- Entire grid is redrawn each update.
- Currently designed for ~5–10 players—performance optimization for larger grids (diff updates, viewport-based rendering) is planned but not implemented.

Future Improvements

- Save/load player session with persistent ID and color.
- Grid zoom/pan.
- Smarter diff-only updates to reduce bandwidth.
- Advanced collision resolution UI.

Recent Freelance Project

p1doks.com

I also want to highlight a recent freelance project I delivered: **p1doks.com**.

This is a motorsports-focused e-commerce platform for selling downloadable “Data Packs” with user accounts, subscriptions, and direct purchases.

Below is a breakdown of the technical work I contributed:

I primarily handled **backend architecture, infrastructure, and application logic**.

On the **frontend**, I built the React app’s *coding and data logic* (API integration, state management), but **did not do the design** (this was provided separately).

Frontend

- React (with Tailwind CSS)

- Consumed Directus GraphQL APIs
- Integrated Stripe checkout flows
- Managed authenticated routes with AWS Cognito JWT
- Used CloudFront + S3 for static site hosting

Backend API

- Node.js Express server
- Stripe Webhook handlers (membership subscriptions and individual purchases)
- AWS Cognito JWT authentication integration
- PayPal integration planning (scope included)
- Hosted on EC2 (small instance), managed with PM2

Directus CMS

- Custom setup on EC2 (Docker)
- S3 integration for Data Pack file storage
- PostgreSQL database connection
- Stripe Product integration for subscription management
- Extension management and GraphQL API usage

Infrastructure

- AWS Cognito for user authentication
- AWS CloudFront + S3 for static hosting
- Amazon RDS (PostgreSQL)
- EC2 instances (2x) for backend and CMS

- AWS S3 for file storage and website assets
- S3 access logging
- AWS Lambda for Cognito extension triggers
- AWS SES for email services
- Porkbun for domain management
- Stripe integration with robust webhook management

Highlights

- Designed a fully integrated **Stripe subscription and purchase system** with secure JWT-protected APIs.
- Built and deployed a **Directus-based CMS** with S3 storage for large file assets.
- Managed **AWS infrastructure** including IAM, EC2, S3, RDS, CloudFront.
- Delivered production-ready, maintainable code with clear separation between frontend logic and design.