ChatGPT

Love & Dice API Integration Mapping

This document describes how the **frontend components** and **backend routes** in the Love & Dice project correspond and interact. It serves as a stitching layer overview for developers to ensure clarity across architecture layers.

Integration Overview

Frontend Component/Context	Backend API Route	Purpose
`chat/ChatWindow.tsx`	`POST /story/continue`	Sends player message, receives story/NPC response
`chat/Message.tsx`	(N/A)	Displays message data in styled bubble format
`character/CharacterSheet.tsx`	`POST /character/create`, `GET /character/:id`	Create and retrieve character data
`npc/NPCProfile.tsx`	`GET /story/npc/:id`	Load NPC data including AP and tags
`npc/NPCActionBar.tsx`	`POST /geist/roll`	Execute Geist Rolls (flirt, joke, question, etc)
`pages/Play.tsx`	Combines `character`, `story`, `geist` routes	Main gameplay UI combining other components
`context/GameContext.tsx`	(Client-side state)	Manages current character, NPC, chat state
`utils/dice.ts`	(Mirrors backend `/utils/dice.ts`)	Roll functions used locally for effects/testing
`image/NPCPortrait.tsx`	`POST /image/generate`	(Optional) Generate NPC image via AI backend

API Interface File (`frontend/api/api.ts`)

All fetch requests from frontend components route through this abstraction layer to:

- Keep component logic clean
- Allow refactoring of endpoints without changing UI files

Example:

```ts
export async function sendStoryMessage(message: string, characterId: string) {
  const res = await fetch("/api/story/continue", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ message, characterId })
  });
  return res.json();
}

export async function rollGeist(type: string, npcId: string) {
  const res = await fetch("/api/geist/roll", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ type, npcId })
  });
  return res.json();
}
```

Flow Example: Player Interacts With NPC

1. **Player types message** in `ChatInput`
2. `ChatWindow` sends message via `sendStoryMessage()`
3. Backend route `/story/continue` processes message, AI reply, roll updates, AP
4. Response returned → `ChatWindow` updates message thread
5. If player clicks a special action ("Flirt") → `rollGeist()`
6. Backend `/geist/roll` computes result → returns success/fail → `NPCProfile` updates

---

Backend Route Summary

- `POST /character/create` — create a new player character
- `GET /character/:id` — retrieve existing character data
- `POST /story/continue` — main chat + narrative continuation
- `GET /story/npc/:id` — retrieve NPC data + AP
- `POST /geist/roll` — run a geist roll action
- `POST /image/generate` — (optional) generate NPC or background images

Suggested Frontend Directory (Partial)

```
frontend/
├── api/
│   └── api.ts
├── chat/
│   ├── ChatWindow.tsx
│   ├── ChatInput.tsx
│   └── Message.tsx
├── character/
│   └── CharacterSheet.tsx
├── npc/
│   ├── NPCProfile.tsx
│   └── NPCActionBar.tsx
├── pages/
│   ├── Play.tsx
│   └── Create.tsx
├── context/
│   └── GameContext.tsx
└── utils/
    └── dice.ts
```

Let me know if you'd like a visual diagram or live interactive whiteboard export (e.g. Mermaid, draw.io schema).