Okay, a Splatoon 3 Salmon Run scheduling site for your internship application sounds like a fantastic and relevant project! It's visually engaging, has clear functionality, and hits the right notes for demonstrating your skills. Let's outline a plan using popular and reliable tech stacks, keeping it light and mainstream, and focusing on that 6-digit code user identification.

Here's a breakdown of how we can approach this, focusing on simplicity and effectiveness:

**Tech Stack (Mainstream & Reliable - "MERN-ish" but lighter):**

* **Frontend (User Interface):**
  + **React.js:** A very popular and widely used JavaScript library for building user interfaces. It's component-based, making development organized and efficient. It's perfect for creating interactive elements like calendars and forms.
  + **HTML/CSS (and maybe a CSS Framework):** Standard web technologies for structuring and styling your site.
    - **Tailwind CSS (Optional but Recommended):** A utility-first CSS framework that makes styling very fast and consistent. It's extremely popular and will speed up your frontend development considerably. It's also very mainstream.
  + **JavaScript:** For interactivity in your React components.
* **Backend (API and Data Handling):**
  + **Node.js with Express.js:** Node.js is a JavaScript runtime environment, and Express.js is a minimal and flexible Node.js web application framework. It's extremely popular for building backends, especially when paired with React frontends. JavaScript on both frontend and backend simplifies development.
  + **MongoDB (or a simpler alternative for "light"):** A NoSQL database. MongoDB is document-based, flexible, and easy to set up. It's well-suited for storing user data, availability slots, and appointments.
    - **Alternatively (Even Lighter):** For truly "minimal users" and to keep it *extra* light for a demo project, you could even consider:
      * **JSON File Database:** For the absolute simplest backend, you could store your data in JSON files and read/write to them using Node.js. This is *very* basic and might be sufficient for a small demo. However, MongoDB is still relatively light and more scalable in concept.
      * **SQLite:** A file-based SQL database. Still lightweight and very easy to set up. Might be overkill for the simplicity of this project compared to JSON, but more robust if you want to demonstrate SQL knowledge.
* **Deployment (Easy and Free for Demo):**
  + **Frontend:**
    - **Netlify or Vercel:** Both are excellent platforms for deploying static sites and single-page applications (like React apps) for free. They integrate well with Git repositories and make deployment incredibly easy.
  + **Backend:**
    - **Heroku (Free Tier):** A popular Platform-as-a-Service (PaaS) that offers a free tier suitable for demo projects. You can deploy your Node.js/Express backend to Heroku.
    - **Railway (Free Tier - Similar to Heroku):** Another good PaaS option with a free tier that's easy to use.

**Features and Implementation Plan:**

1. **Homepage - Salmon Run Theme:**
   * Visually appealing landing page themed around Splatoon 3 Salmon Run. Use colors, fonts, and imagery from the game.
   * Briefly explain the site's purpose: "Schedule Salmon Run sessions with fellow players!" or something similar.
2. **User Identification (6-Digit Code):**
   * **Code Generation:** When a user first visits the site (or when they want to create availability), generate a random 6-digit code.
   * **Storage (Frontend - Local Storage):** Store this 6-digit code in the user's browser's localStorage. This means the code persists even if they close and reopen the browser (on the same device).
   * **Backend Association (Implicit):** When a user creates availability slots or books an appointment, associate these actions with the 6-digit code. You don't need a separate "user" table. The 6-digit code *is* their identifier.
3. **"Create Availability" Feature:**
   * **Page/Section:** A dedicated area (or page) for creating availability.
   * **Time Slot Selection:**
     + Use a calendar component (React libraries like react-datepicker or react-big-calendar are available, but for simplicity, you could even use basic HTML <input type="date"> and time inputs).
     + Allow users to select a date and start/end times for their Salmon Run session. Consider time intervals (e.g., 30-minute slots).
   * **Stage Selection (Optional but Thematic):** Include a dropdown or selection for Salmon Run stages (Spawning Grounds, Marooner's Bay, etc.). This adds to the Splatoon theme.
   * **"Generate My Code" (If needed):** If the code isn't generated automatically on first visit, have a button to generate the 6-digit code and display it to the user. Instruct them to keep this code safe.
   * **Submit Availability:** Button to save the availability slot. Send a request to your backend API (Node.js/Express) with the date, time, stage (optional), and the user's 6-digit code (from localStorage).
4. **"View Availability" Feature:**
   * **Page/Section:** Display available Salmon Run sessions.
   * **Calendar View (Ideal):** If you use a calendar library, you can display availability slots visually on a calendar.
   * **List View (Simpler):** Alternatively, a list of available slots, grouped by date, time, and maybe stage.
   * **Fetching Availability:** Frontend makes a request to your backend API to get all available slots.
   * **"Book Slot" Button:** For each available slot, have a button to "Book This Slot."
     + When clicked, send a request to your backend API with the slot ID and the user's 6-digit code (from localStorage). Mark the slot as "booked" and associate it with the user who booked it.
5. **"My Schedule" or "My Availability" Feature:**
   * **Page/Section:** Allow users to see their created availability slots and the slots they have booked.
   * **Input for 6-Digit Code:** Have an input field where users can enter their 6-digit code.
   * **Display Slots:**
     + **Created Slots:** Show the availability slots created by this user (using their 6-digit code to filter data from the backend).
     + **Booked Slots:** Show the slots this user has booked.
   * **(Optional) "Cancel Booking" or "Delete Availability":** Allow users to cancel booked slots or delete their own availability slots.

**Data Structure (Example with MongoDB):**

You might have a MongoDB collection called availability\_slots. Each document could look like this:

{

"\_id": ObjectId("..."), // MongoDB automatically generates this

"userId": "123456", // 6-digit user code

"date": "2024-07-28",

"startTime": "19:00", // 24-hour format

"endTime": "20:00",

"stage": "Spawning Grounds", // Optional

"bookedByUserId": null, // Initially null, filled with 6-digit code when booked

"isBooked": false // Boolean flag if booked

}

**Project Steps (Breakdown for Development):**

1. **Setup Development Environment:**
   * Install Node.js, npm (Node Package Manager).
   * Create a React app (using create-react-app or Vite).
   * Set up a Node.js/Express backend project.
   * (If using MongoDB) Install MongoDB locally or use MongoDB Atlas (cloud-based, free tier available).
2. **Backend API Development (Node.js/Express):**
   * Define API endpoints for:
     + Creating availability slots (POST /api/availability).
     + Getting all available slots (GET /api/availability).
     + Booking a slot (POST /api/booking).
     + Getting availability slots for a specific user (GET /api/user/availability/:userId).
   * Implement data storage logic (using MongoDB or your chosen data storage method).
3. **Frontend Structure and Basic UI (React):**
   * Create React components for:
     + Homepage
     + Create Availability Form
     + View Availability Calendar/List
     + My Schedule/Availability Page
   * Set up basic routing (using react-router-dom) to navigate between pages.
4. **Connect Frontend to Backend:**
   * Use fetch or axios in your React components to make API calls to your backend.
   * Handle data fetching and display in the frontend.
5. **Implement 6-Digit Code Logic:**
   * Generate and store the code in localStorage on the frontend.
   * Send the code with API requests to the backend.
   * Use the code to identify users and associate data.
6. **Styling and Theming (CSS/Tailwind CSS):**
   * Apply Splatoon 3 theme to your site's design. Use colors, fonts, and imagery.
   * Style components to be visually appealing and user-friendly.
7. **Deployment:**
   * Deploy your React frontend to Netlify or Vercel.
   * Deploy your Node.js/Express backend to Heroku or Railway.
8. **Testing and Refinement:**
   * Test all features thoroughly.
   * Refine UI/UX based on testing.
   * Add error handling and user feedback (e.g., success messages, error messages).

**Why this is a good approach for your internship application:**

* **Demonstrates Full-Stack Skills (Simplified):** You'll be touching on both frontend and backend development, database interaction, and deployment.
* **Uses Popular Tech:** React, Node.js, Express, MongoDB are all highly sought-after technologies in the industry.
* **Clear and Practical Functionality:** The project has a clear purpose and solves a real (albeit niche) need.
* **Scalable (in Concept):** Even though you're keeping it light, the tech stack is capable of scaling if needed, showing you understand scalable architectures.
* **Visually Engaging (Splatoon Theme):** The theme makes it stand out and shows your creativity and attention to detail.
* **Manages Complexity:** By using the 6-digit code authentication and keeping features focused, you avoid over-engineering and keep the project manageable within an internship application timeframe.

**Key things to emphasize in your application:**

* **Explain your tech stack choices** and why you chose them (mainstream, reliable, good for the project scope).
* **Highlight the 6-digit code user identification** as a conscious decision to keep it light and user-friendly for a demo.
* **Showcase the Splatoon 3 theme** and how it makes the project unique and engaging.
* **Describe the features you implemented** and how they work together.
* **Mention potential future enhancements** (even if you don't implement them) to show you're thinking ahead.

This plan should give you a solid foundation for your Splatoon 3 Salmon Run scheduling site. Good luck with your project and your internship application! Let me know if you have any more questions as you build it.

Hey I was working with another model with this. Here’s our conversation history, I ask you to continue help me with the work.

Okay, I will provide a comprehensive handover summary covering the project idea, all requirements (original and newly clarified), the chosen tech stack, the progress made so far, the current status, and the immediate next steps. This should allow another AI model to pick up exactly where we left off.

**Handover Summary for AI Model Transition - Salmon Run Scheduler Project**

**User Context:** The user is new to coding and building this project as a showcase for an intern application. The goal is to demonstrate understanding of popular, mainstream, and lightweight web development technologies. The user has hit a rate limit and needs another model to continue guiding them step-by-step.

**Project Name/Theme:** Splatoon 3 Salmon Run Themed Availability & Scheduling Website.

**Overall Project Goal:** To create a simple website where the user (Host) can post their availability for playing Salmon Run, and other players (Visitors) can view the Host's availability and book specific time slots to play together.

**Target Users:** Minimal users (primarily the Host and a small group of friends/players).

**Core Functionality & Requirements (Original + Clarified):**

1. **External Salmon Run Schedule Integration:** The website will fetch and display the *official* in-game Salmon Run schedule data from a public external API (user has a potential URL). This provides context for the Host's availability.
2. **Host Availability Management:**
   * The Host (user) needs a way to input their available time slots.
   * This availability should ideally align with the official 40-hour Salmon Run shifts.
   * Bookable slots should be in multiples of 30 minutes.
3. **Availability Display:**
   * Visitors view the official Salmon Run schedule on the website.
   * The Host's available periods/shifts are highlighted or clearly indicated within or alongside the official schedule display.
   * Clicking a highlighted period shows a more detailed breakdown of the Host's *exact bookable time slots* within that window (e.g., specific 30-minute blocks).
4. **Booking Process:**
   * Visitors select a specific, available time slot.
   * A booking form appears where the Visitor enters:
     + Their Nintendo Switch Friend Code.
     + An optional message.
   * Submitting the form creates a booking record in the database.
5. **Persistent Visitor Identification (6-Digit Code):**
   * Upon the *first* successful booking by a Visitor, the system generates a **unique, persistent 6-digit code** for *that visitor*.
   * This code acts as the Visitor's ongoing ID for the website.
   * Visitors are shown this code after their first booking.
   * Visitors can potentially use this code to "login" and **manage their scheduled sessions** (specific management actions need clarification - view details, cancel?).
   * The idea of visitors changing this ID was mentioned, but is of lower priority and might be complex to implement securely for a first version.
6. **Host Contact Information:** The Host's contact details (e.g., Discord) must be displayed on the website.
7. **External Confirmation Step:** The website booking is preliminary. The Visitor *must* take their 6-digit Visitor ID and send it to the Host's contact information (outside the website) to finalize and confirm the session. The website does not handle the final confirmation handshake.
8. **Simplicity and Theme:** The website should be generally simplistic, informational, and incorporate a clear Splatoon 3 Salmon Run theme (fonts, basic imagery) without being overly complex or cluttered.

**Tech Stack:**

* **Frontend:** React with Vite (Build tool). Selected for popularity, mainstream use, modern development experience (Vite speed), component architecture, and suitability for intern projects.
* **Backend:** Node.js with Express.js. Selected for popularity, lightweight nature, and ability to create APIs in JavaScript (full-stack JS consistency).
* **Database:** SQLite. Selected for extreme lightweightness (file-based), ease of setup/management for minimal users, standard SQL, and suitability for small project scale, while still being a widely used technology.
* **Styling:** Tailwind CSS v4 (using the @tailwindcss/vite plugin). Selected for rapid, utility-first styling, lightweight final CSS output, and mainstream popularity.

**Progress To Date:**

1. **Backend Setup:**
   * Node.js, Express.js, and SQLite are installed.
   * A basic Express server (index.js) is set up and listens on port 3001.
   * The SQLite database file (salmon\_run\_scheduler.db) is initialized.
   * The initial database table availability\_slots was created.
   * Basic API endpoints were defined:
     + POST /api/availability (intended for Host to create slots)
     + GET /api/availability/:code (intended for Visitors to view a specific slot - **this endpoint is now less relevant based on the persistent ID requirement and displaying *all* available slots**).
   * **Status:** The backend setup is functional but the existing API endpoints and database schema (availability\_slots) **DO NOT YET reflect the updated requirements** (persistent Visitor ID, bookings table, fetching *available* slots).
2. **Frontend Setup:**
   * A React project has been created using Vite.
   * The basic App component is set up.
   * Component structure folders (CreateAvailability, JoinSession) were created, and basic component files were added (CreateAvailability.jsx, JoinSession.jsx).
   * **Tailwind CSS v4 is successfully installed and configured** using the official @tailwindcss/vite plugin method. This involved resolving an issue where the npx tailwindcss init command failed in v4 (as the CLI executable structure changed). The correct setup was confirmed via the vite.config.js plugin integration and testing styles in the browser after resolving conflicts with default Vite styles. Basic Tailwind classes are applying correctly.
   * **Status:** The frontend development environment is set up, structured with placeholder components, and styled with Tailwind CSS. The components **DO NOT YET contain the logic or forms** for the new requirements (displaying external schedule, listing available slots, booking form, displaying persistent ID, visitor login/management). The CreateAvailability component *was* partially built with a form based on the *old* requirement of it returning a slot code, but this form/logic needs to be updated for the new backend API and Host workflow.

**Current Status & Immediate Next Steps:**

We have just finished confirming the detailed, revised requirements. The immediate next step is to **refactor and expand the Backend (Node.js/Express/SQLite)** to fully support the new requirements, specifically:

1. **Update the Database Schema:** Modify availability\_slots (add is\_booked field) and create a new bookings table (to store slot\_id, booking\_code (the persistent Visitor ID), friend\_code, message, etc.).
2. **Update Existing/Create New API Endpoints:**
   * Adjust POST /api/availability (for Host to create slots - return slot ID, not a code).
   * Create GET /api/availability (to fetch all *available* slots for Visitors).
   * Create POST /api/bookings (to handle Visitor booking - generate 6-digit Visitor ID, save booking, mark slot as booked, return Visitor ID).
   * (Potential Future) Endpoints for Visitors to "login" with their ID and view/manage bookings.
   * (Potential Future) Endpoints for Host to view bookings.

Once the backend is updated and tested (via curl or similar), we will proceed to the Frontend again to build out the UI logic using the new backend APIs.

**Points Requiring Further Clarification from User:**

Before implementing certain parts of the backend/frontend logic, confirmation is still needed on:

1. **External Salmon Run Schedule API URL:** The specific URL for the external game schedule API.
2. **Host Availability Input Method:** Exactly how the Host defines their availability (specific times, selecting shifts, etc.) and the minimum bookable time slot duration (always 30 mins, or flexible?).
3. **Visitor "Manage Scheduled Sessions":** What specific actions should a Visitor be able to perform after logging in with their ID?

**Readiness to Continue:** The project is ready to continue with **Backend Refactoring** based on the clarified requirements.

Contents

**No table of contents entries found.**

# Pulling data from .ink api

This summary provides all the necessary context. You can now ask the user to provide the clarification details, and then proceed with implementing the backend changes as outlined in the "Immediate Next Steps" section. Good luck!

Okay, this is a great idea and definitely aligns better with the theme and player expectations! Integrating the official schedule makes the tool much more useful and intuitive for Salmon Run players.

\*\*Revised Plan:\*\*

1. \*\*Fetch Official Schedule (Frontend):\*\*

\* In `AvailabilityDisplay.jsx` (or a dedicated component), fetch the schedule data from `https://splatoon3.ink/data/schedules.json`.

\* Parse the JSON and extract the relevant Salmon Run shifts (specifically from `data.coopGroupingSchedule.regularSchedules.nodes`). This array contains objects with `startTime`, `endTime`, `setting.coopStage`, `setting.weapons`, etc.

2. \*\*Fetch Host's Availability Slots (Frontend):\*\*

\* Continue fetching the host's 30-minute available slots from our backend (`GET /api/availability`).

3. \*\*Combine and Display (Frontend):\*\*

\* Iterate through the \*\*official shifts\*\* obtained from splatoon3.ink.

\* For each official shift, display its details (Stage, Weapons, Start/End Time).

\* \*\*Highlighting Logic:\*\* Determine if the host has \*any\* availability slots (from our backend) that fall \*within\* the time range of the current official shift.

\* This requires comparing the `startTime`/`endTime` of the official shift with the `start\_time`/`end\_time` of each of the host's available slots.

\* If there's an overlap, apply a visual highlight (e.g., different background color, border) to the displayed official shift card/element.

\* Make the \*highlighted\* official shift elements clickable.

4. \*\*"Zoom In" / Detail View (Frontend):\*\*

\* When a user clicks a \*highlighted\* official shift:

\* \*\*Filter:\*\* Find all the host's available 30-minute slots (from our backend data stored in state) that fall \*specifically within the start/end time of the clicked official shift\*.

\* \*\*Display:\*\* Show these filtered 30-minute slots in a list below the clicked shift, or perhaps in a modal. This list should look very similar to our current slot display (time + Book button).

\* The booking form and success modal logic we just built can be reused here when the user clicks "Book" on one of these detailed 30-minute slots.

\*\*Impact on Existing Code:\*\*

\* \*\*Backend:\*\* \*\*No changes needed!\*\* The backend continues to manage the host's availability as discrete 30-minute slots and handles booking those specific slots. This is good.

\* \*\*Frontend (`CreateAvailability.jsx`):\*\* No changes needed. The host still defines their availability in blocks, and the backend breaks it down.

\* \*\*Frontend (`AvailabilityDisplay.jsx`):\*\* \*\*Significant changes required.\*\*

\* Needs state to store the official schedule data.

\* Needs a new `useEffect` (or combine with the existing one) to fetch the official schedule.

\* Needs new logic to parse the official schedule JSON and extract Salmon Run data.

\* Needs complex rendering logic to:

\* Display official shifts.

\* Compare official shift times with host availability slot times to determine highlighting.

\* Handle clicking on highlighted shifts.

\* Filter and display the relevant host slots when a shift is clicked.

\* The existing slot rendering list (`ul` with `.map`) will be repurposed for the "detail view" when a shift is clicked.

\* The booking form and success modal logic remain largely the same but are triggered from the detail view.

\*\*Reusability:\*\*

You are correct, we can reuse:

\* The backend API entirely.

\* The `CreateAvailability` component.

\* The state variables and logic for handling the booking form (`selectedSlotId`, `friendCode`, `message`, `handleBookingSubmit`, etc.).

\* The success modal component and its logic.

\* The helper functions for formatting time (`formatTime`).

\*\*Work Involved:\*\*

This is definitely more work on the \*\*frontend rendering and state management\*\* side compared to just displaying the flat list of slots. The main challenge is the logic to compare the two data sources (official schedule vs. host slots) and manage the state for showing/hiding the detailed slot view for a selected shift. However, it's a very logical and user-friendly approach for this specific application.

\*\*Let's Start:\*\*

Please provide the picture/screenshot showing how the other website displays the shift data. This will give us a visual target for how to structure the initial display of the official shifts.

Once I see the visual layout, we can start modifying `AvailabilityDisplay.jsx` to:

1. Add state for the official schedule.

2. Add the fetch call for the external API.

3. Start building the JSX to display the official shifts (without highlighting or interaction yet).

# Include ink api

Perfect, that image is exactly what we needed! It gives a clear visual target and shows the key information displayed for each shift:

\* Start/End Time (formatted nicely)

\* Time Remaining / "In X days..." (We might skip this relative time for V1 to keep it simpler)

\* Stage Image & Name

\* Weapon Images

Let's modify `AvailabilityDisplay.jsx` to fetch this external schedule and display it in a similar card-like format. We'll worry about fetching our \*own\* availability data and doing the highlighting/interaction logic \*after\* we get the basic official schedule display working.

\*\*Action Step 1: Add State for Official Schedule\*\*

1. \*\*Open `src/components/AvailabilityDisplay/AvailabilityDisplay.jsx`\*\*.

2. \*\*Add new state variables\*\* to hold the official schedule data and its loading/error status:

```jsx

function AvailabilityDisplay() {

// State for Host's available slots (from our backend)

const [hostSlots, setHostSlots] = useState([]); // Renamed for clarity

const [isLoadingHostSlots, setIsLoadingHostSlots] = useState(true); // Renamed

const [hostSlotsError, setHostSlotsError] = useState(null); // Renamed

// --- Add State for Official Schedule ---

const [officialSchedule, setOfficialSchedule] = useState([]); // Holds the extracted Salmon Run shifts

const [isLoadingSchedule, setIsLoadingSchedule] = useState(true); // Loading state for external API

const [scheduleError, setScheduleError] = useState(null); // Error state for external API

// --- End Add State ---

// State for the booking process (keep these as they are)

const [selectedSlotId, setSelectedSlotId] = useState(null);

const [friendCode, setFriendCode] = useState('');

const [message, setMessage] = useState('');

const [isBookingLoading, setIsBookingLoading] = useState(false);

const [bookingError, setBookingError] = useState(null);

const [lastVisitorCode, setLastVisitorCode] = useState(null);

const [isSuccessModalOpen, setIsSuccessModalOpen] = useState(false);

// Define host info (can be moved later)

const hostContactInfo = "YOUR\_DISCORD\_USERNAME#1234"; // <<< REPLACE

// ... (Helper functions: formatDateHeader, formatTime) ...

```

\* \*\*Renaming:\*\* I renamed the state variables related to \*our backend\* (`availableSlots` -> `hostSlots`, etc.) to avoid confusion with the \*official\* schedule data. Make sure to update the corresponding `set...` function names as well if you rename them.

\*\*Action Step 2: Modify Fetching Logic\*\*

We now need to fetch from \*two\* APIs. We can do this in parallel using `Promise.all`.

1. \*\*Modify `fetchSlots` -> `fetchAllData`:\*\* Rename the main data-fetching function and update it to fetch both sets of data.

```jsx

// --- Fetch BOTH sets of data ---

// Renamed function for clarity

const fetchAllData = useCallback(async () => {

console.log("Fetching all data...");

// Set loading states for both fetches

setIsLoadingHostSlots(true);

setIsLoadingSchedule(true);

setHostSlotsError(null);

setScheduleError(null);

try {

// Use Promise.all to fetch both APIs concurrently

const [hostSlotsResponse, scheduleResponse] = await Promise.all([

fetch('http://localhost:3001/api/availability'), // Our backend

fetch('https://splatoon3.ink/data/schedules.json') // Official schedule API

]);

// --- Process Host Slots Response ---

if (!hostSlotsResponse.ok) {

throw new Error(`Host Slots Error! Status: ${hostSlotsResponse.status}`);

}

const hostData = await hostSlotsResponse.json();

setHostSlots(hostData); // Update state for host slots

// --- Process Official Schedule Response ---

if (!scheduleResponse.ok) {

throw new Error(`Official Schedule Error! Status: ${scheduleResponse.status}`);

}

const scheduleData = await scheduleResponse.json();

// Extract only the relevant Salmon Run shifts

// Adjust path based on the actual JSON structure you provided:

// data.coopGroupingSchedule.regularSchedules.nodes

if (scheduleData.data?.coopGroupingSchedule?.regularSchedules?.nodes) {

setOfficialSchedule(scheduleData.data.coopGroupingSchedule.regularSchedules.nodes);

} else {

console.warn("Could not find Salmon Run schedule in expected path in API response.");

setOfficialSchedule([]); // Set empty if path not found

// Optionally throw an error or set scheduleError here

// throw new Error("Could not parse Salmon Run schedule data.");

}

} catch (err) {

console.error("Error fetching data:", err);

// Distinguish which fetch failed if possible, or set a general error

// For simplicity now, let's potentially set both errors if one fails,

// or check err.message content.

setHostSlotsError(err.message); // May be inaccurate if schedule fetch failed

setScheduleError(err.message); // May be inaccurate if host fetch failed

setHostSlots([]); // Clear data on error

setOfficialSchedule([]); // Clear data on error

} finally {

// Set loading states false for both

setIsLoadingHostSlots(false);

setIsLoadingSchedule(false);

}

}, []); // Empty dependency array remains

// --- useEffect to fetch all data on mount ---

useEffect(() => {

fetchAllData(); // Call the combined fetch function

}, [fetchAllData]); // Update dependency array

// ... (handleBookClick, handleCancelBooking, handleBookingSubmit remain the same for now) ...

```

\* \*\*`Promise.all([...])`\*\*: Takes an array of Promises (our two `fetch` calls) and runs them in parallel. It waits until \*all\* promises in the array have resolved. It then returns an array containing the results (the `Response` objects) in the same order.

\* \*\*Destructuring Assignment:\*\* `const [hostSlotsResponse, scheduleResponse] = await Promise.all(...)` assigns the first resolved response to `hostSlotsResponse` and the second to `scheduleResponse`.

\* \*\*Separate Processing:\*\* We now process each response individually, checking `ok`, parsing JSON, and setting the corresponding state (`hostSlots` or `officialSchedule`).

\* \*\*JSON Path:\*\* \*\*Crucially\*\*, we extract the Salmon Run data using the path `scheduleData.data?.coopGroupingSchedule?.regularSchedules?.nodes`. The `?.` (optional chaining) prevents errors if any part of the path doesn't exist in the response.

\* \*\*Error Handling:\*\* The `catch` block now catches errors from either fetch. The error handling could be more granular (checking which promise rejected), but for now, it sets both error states.

\* \*\*`finally`:\*\* Sets both loading states to false.

\* \*\*`useEffect`:\*\* Updated to call `fetchAllData` and include it in the dependency array.

\*\*Action Step 3: Update Rendering Logic for Official Schedule\*\*

Now, let's change the main rendering part to display the official schedule cards, similar to the image. We'll temporarily ignore the host slots and booking form.

1. \*\*Replace\*\* the entire content \*inside\* the main `return (...)` `div` (after the `<h2>`) with this new structure focused on the official schedule:

```jsx

return (

<div className="bg-gray-700 p-6 rounded-lg shadow-lg w-full">

<h2 className='text-2xl font-semibold mb-4 text-cyan-300'>

Salmon Run Schedule {/\* Updated Title \*/}

</h2>

{/\* --- Loading/Error States for Official Schedule --- \*/}

{isLoadingSchedule && <p className='text-gray-400'>Loading official schedule...</p>}

{scheduleError && <p className='text-red-400'>Error loading official schedule: {scheduleError}</p>}

{/\* --- Display Official Schedule --- \*/}

{!isLoadingSchedule && !scheduleError && officialSchedule.length === 0 && (

<p className='text-gray-400'>No official schedule found.</p>

)}

{!isLoadingSchedule && !scheduleError && officialSchedule.length > 0 && (

<div className="space-y-5"> {/\* Space between shift cards \*/}

{officialSchedule.map((shift, index) => {

// Check if the necessary data exists before trying to render

const stage = shift.setting?.coopStage;

const weapons = shift.setting?.weapons;

const boss = shift.setting?.boss; // King Salmonid

// Basic check to ensure we have the core data needed

if (!stage || !weapons || !shift.startTime || !shift.endTime) {

console.warn("Skipping rendering shift due to missing data:", shift);

return null; // Don't render this card if data is incomplete

}

// Simple way to differentiate current/future - check if shift end time is in the past

const isPast = new Date(shift.endTime) < new Date();

// Check if current time is between start and end

const isCurrent = new Date(shift.startTime) <= new Date() && !isPast;

// TODO: Add logic here later to check if host has availability during this shift

const hostHasAvailability = false; // Placeholder for now

return (

// --- Shift Card ---

// Add border/background later for highlighting based on hostHasAvailability

<div key={shift.startTime} className={`bg-gray-800 p-4 rounded-lg shadow-md border ${hostHasAvailability ? 'border-green-500' : 'border-gray-700'}`}> {/\* Basic card styling \*/}

{/\* Row 1: Time \*/}

<div className="mb-2">

<span className={`font-semibold text-lg ${isCurrent ? 'text-orange-400' : 'text-orange-300'}`}>

{formatDateHeader(shift.startTime)} {/\* Use helper \*/}

</span>

<span className={`font-semibold text-lg ${isCurrent ? 'text-orange-400' : 'text-orange-300'}`}>

{' - '}

</span>

<span className={`font-semibold text-lg ${isCurrent ? 'text-orange-400' : 'text-orange-300'}`}>

{formatDateHeader(shift.endTime)} {/\* Use helper \*/}

</span>

{isCurrent && <span className="ml-3 inline-block bg-red-600 px-2 py-0.5 rounded text-xs font-bold animate-pulse">NOW</span>}

</div>

{/\* Row 2: Stage Info & Weapons \*/}

<div className="flex flex-col sm:flex-row gap-4 items-center">

{/\* Left Side: Stage \*/}

<div className="flex-shrink-0 w-full sm:w-1/3">

<img

src={stage.image?.url}

alt={stage.name}

className="rounded-lg w-full object-cover aspect-video border border-gray-600" // aspect-video for consistent ratio

onError={(e) => { e.target.style.display = 'none'; /\* Hide if image fails \*/ }}

/>

<p className="text-center font-semibold text-cyan-200 mt-1">{stage.name}</p>

</div>

{/\* Right Side: Weapons \*/}

<div className="flex-grow">

<p className="text-sm font-semibold text-gray-400 mb-1">Supplied Weapons:</p>

<div className="flex flex-wrap gap-2 justify-center sm:justify-start">

{weapons.map((w, idx) => (

<img

key={idx} // Using index is okay here as weapon order is fixed per shift

src={w.image?.url}

alt={w.name}

title={w.name} // Tooltip on hover

className="h-10 w-10 object-contain bg-gray-700 p-1 rounded-full border border-gray-600"

onError={(e) => { e.target.style.display = 'none'; }}

/>

))}

</div>

{/\* Optional: King Salmonid \*/}

{boss && (

<p className="text-xs text-center sm:text-left text-gray-500 mt-2">

King Salmonid: {boss.name}

</p>

)}

</div>

</div>

{/\* Placeholder for future detailed slot display \*/}

{/\* { hostHasAvailability && <div className="mt-3 border-t border-gray-600 pt-3"> ... show host slots ... </div> } \*/}

</div>

// --- End Shift Card ---

);

})}

</div>

)}

{/\* --- Booking Form & Success Modal (Keep these, they are hidden until triggered) --- \*/}

{selectedSlotId !== null && ( <div className="mt-6..."> {/\* Booking Form JSX \*/} </div> )}

<Dialog open={isSuccessModalOpen} onClose={...}> {/\* Success Modal JSX \*/} </Dialog>

{/\* --- End Booking Form & Modal --- \*/}

</div>

);

```

\*\*Explanation of Rendering Changes:\*\*

1. \*\*Title Update:\*\* Changed `h2` to "Salmon Run Schedule".

2. \*\*Loading/Error Checks:\*\* Now checks `isLoadingSchedule` and `scheduleError`.

3. \*\*Mapping `officialSchedule`:\*\* We map over the `officialSchedule` state array.

4. \*\*Data Check:\*\* Added checks (`if (!stage || !weapons ...`) to ensure essential data exists for a shift before trying to render it. Logs a warning and returns `null` (renders nothing for that item) if data is missing.

5. \*\*`isCurrent` / `isPast`:\*\* Basic logic to determine if a shift is currently active. Used for styling the time and adding a "NOW" badge.

6. \*\*`hostHasAvailability` Placeholder:\*\* Added a variable `hostHasAvailability` (currently hardcoded to `false`) which we will calculate later. It's used to conditionally apply a green border (`border-green-500`) to the shift card.

7. \*\*Shift Card Structure:\*\*

\* Outer `div` represents the card. Basic background/padding/border.

\* \*\*Time Row:\*\* Displays start/end times using the `formatDateHeader`. Applies different text color if `isCurrent`. Shows a "NOW" badge if current.

\* \*\*Stage/Weapon Row:\*\* Uses flexbox (`flex flex-col sm:flex-row`) to arrange stage info and weapons side-by-side on small screens and up (`sm:`).

\* \*\*Stage Image:\*\* Displays the stage image using `<img>`. Added `aspect-video` for consistent sizing and `onError` to hide the image if the URL fails to load.

\* \*\*Weapon Images:\*\* Maps over the `weapons` array and displays each weapon image (`<img>`) with styling (rounded, background, border). Added `title` attribute for hover tooltips.

\* \*\*King Salmonid:\*\* Optional display if `boss` data exists.

8. \*\*Booking Form/Modal:\*\* Kept the JSX for the booking form and success modal at the end. They are hidden by default (`selectedSlotId === null`, `isSuccessModalOpen === false`) and will only appear when triggered later.

\*\*Step 4: Test\*\*

1. \*\*Replace\*\* the placeholder `hostContactInfo` again if needed.

2. Save the file.

3. Restart frontend/backend servers.

4. Refresh the browser.

\*\*Expected Result:\*\*

You should now see:

\* "Loading official schedule..." briefly.

\* A list of Salmon Run shift "cards", each showing:

\* Formatted Start/End Dates/Times.

\* Stage Image and Name.

\* Weapon Images.

\* King Salmonid Name (optional).

\* A "NOW" badge on the currently active shift (if any).

\* The booking form and success modal should \*not\* be visible initially.

\* The cards will have a gray border for now (since `hostHasAvailability` is false).

This is a big step! Let me know if it renders correctly and displays the official schedule data.

# Deleted code that might be useful during ink api adaptation

            {isLoadingHostSlots && <p className='text-gray-400'>Loading available slots...</p>}

            {error && <p className='text-red-400'>Error loading slots: {error}</p>}

            {!isLoadingHostSlots && !error && availableSlots.length === 0 && (

                <p className='text-gray-400'>No available slots found at the moment.</p>

            )}

            {!isLoadingHostSlots && !error && availableSlots.length > 0 && (

                <div className='space-y-6'> {/\* Add vertical space between date groups \*/}

                    {

                        // --- Grouping Logic using reduce ---

                        Object.entries( // 1. Get[key, value] paris from the grouped object

                            availableSlots.reduce((groups, slot) => {

                                // 2. For each slot, get the date part (YYYY-MM-DD) as the key

                                const dateKey = slot.start\_time.split('T')[0]; // Simple split on 'T'

                                // 3. If this dateKey isn't in groups yet, create an empty array for it

                                if (!groups[dateKey]) {

                                    groups[dateKey] = [];

                                }

                                // 4. Push the current slot into the array for its dateKey

                                groups[dateKey].push(slot);

                                // 5. Return the modified groups object for the next iteration

                                return groups;

                            }, {}) // 6. Start with an empty object {} as the initial value for the groups

                        )

                        // --- End Grouping Logic ---

                        // 7. Map over the [dateKey, slotsArray] pairs

                        .map(([dateKey, slotsInGroup]) => (

                            // 8. For each group, render a section with a key

                            <div key={dateKey}>

                                {/\* 9. Render the Date Header \*/}

                                <h3 className='text-lg font-semibold text-orange-300 mb-2 border-b border-gray-600 pb-1'>

                                    {formatDateHeader(dateKey)} {/\* Use helper function \*/}

                                </h3>

                                {/\* 10. Render the list of slots for \*this\* group \*/}

                                <ul className='space-y-2'>

                                    {slotsInGroup.map((slot) => (

                                        // 11. Render each slot item (similar before)

                                        <li key={slot.id} className='bg-gray-600 p-3 rounded-md flex justify-between items-center'>

                                            <span className='text-white font-medium'>

                                                {/\* 12. Use formatTime for start and end \*/}

                                                {formatTime(slot.start\_time)} - {formatTime(slot.end\_time)}

                                            </span>

                                            <button

                                                onClick={() => handleBookClick(slot.id)}

                                                className='bg-green-500 hover:bg-green-600 text-white font-bold py-1 px-3 rounded text-sm'

                                                disabled={isBookingLoading || selectedSlotId === slot.id}

                                            >

                                                预约

                                            </button>

                                        </li>

                                    ))}

                                </ul>

                            </div>

                        ))

                        // --- End Rendering Logic ---

                    }

                </div>

            )}

# Interesting debug, turned out it was my typo

Thanks for the detailed logs! This tells us a lot:

1. \*\*Multiple `useEffect` Runs:\*\* You noticed `fetchAllData` being called multiple times. This is happening because the dependency array in your `useEffect` hook is `[fetchAllData]`. Since `fetchAllData` is defined \*inside\* the component, it technically gets redefined on every render. Although `useCallback` \*memoizes\* it, there might be subtle interactions (especially during development with React's Strict Mode or Hot Module Replacement) causing the effect to re-run.

2. \*\*Successful Fetch:\*\* The logs clearly show \*both\* fetches eventually succeed ("Host availability fetched", "Ink api SR schedule fetched successfully").

3. \*\*State Updates Correctly:\*\* The \*final\* "Rendering Check" logs show `isLoadingHostSlots: false`, `isLoadingOfficialSchedule: false`, `hostSlotsError: null`, `officialScheduleError: null`, and `officialScheduleExists: true`. This means all the conditions in our main JSX check `{!isLoadingHostSlots && !isLoadingOfficialSchedule && ... && officialSchedule}` \*are\* eventually met.

4. \*\*Data Structure Correct:\*\* The log for `officialSchedule` shows the correct structure `{bannerImage: null, regularSchedules: {nodes: Array(5)}, bigRunSchedules: {nodes: Array(0)}, teamContestSchedules: {nodes: Array(0)}}`.

\*\*Conclusion:\*\* The state \*is\* being updated correctly, and the main condition to render the schedules \*is\* becoming true. The reason you aren't seeing the "Upcoming Shifts" heading and placeholders \*\*must be within the JSX rendering logic itself\*\*.

Let's re-examine the JSX block that renders the schedules:

```jsx

{!isLoadingHostSlots && !isLoadingOfficialSchedule && !hostSlotsError && !officialScheduleError && officialSchedule && (

<div className='space-y-6'>

{/\* --- Section for Regular Schedules --- \*/}

{officialSchedule.regularSchedules?.nodes?.length > 0 && ( // CHECK 1

<div>

<h3 className='text-xl font-semibold text-orange-300 mb-3'>Upcoming Shifts</h3>

<div className='space-y-4'>

{officialSchedule.regularSchedules?.nodes.map(shift => ( // CHECK 2

<p key={shift.startTime}>Regular Shift: {shift.setting.coopStage.name} ({shift.startTime})</p>

))}

</div>

</div>

)}

{/\* --- Section for Big Run --- \*/}

{officialSchedule.bigRunSchedules?.nodes?.length > 0 && ( // CHECK 3

// ... content ...

)}

{/\* --- Section for Team Contest --- \*/}

{officialSchedule.teamContestSchedules?.nodes?.length > 0 && ( // CHECK 4

// ... content ...

)}

{/\* --- If no schedules found at all --- \*/}

{!officialSchedule.regularSchedules?.nodes?.length && // CHECK 5a

!officialSchedule.bigRunSchedules?.nodes?.length && // CHECK 5b

!officialSchedule.teamContestSchedules?.nodes?.length && // CHECK 5c

(<p className='text-gray-400'>No upcoming Salmon Run schedules found in the Ink api data</p>)

}

</div>

)}

```

\*\*Possible Reasons Nothing Shows Up:\*\*

\* \*\*Check 1 Fails:\*\* `officialSchedule.regularSchedules?.nodes?.length > 0` evaluates to `false`. Your log shows `regularSchedules: {nodes: Array(5)}`, so the length \*is\* greater than 0. This check \*should\* pass.

\* \*\*Check 2 Fails (Inside Map):\*\* Is there an error \*inside\* the `.map` call for regular schedules? For example, is `shift.setting.coopStage.name` sometimes missing in the API data for one of the shifts? This could break the rendering of the entire list.

\* \*\*Checks 3 & 4 Fail:\*\* `bigRunSchedules?.nodes?.length > 0` and `teamContestSchedules?.nodes?.length > 0` are correctly evaluating to `false` because the arrays are empty in the current data. This is expected.

\* \*\*Check 5 (No Schedules Message):\*\* The logic here is "if Regular Length is NOT > 0 \*\*AND\*\* BigRun Length is NOT > 0 \*\*AND\*\* TeamContest Length is NOT > 0". Since Regular Length \*is\* > 0, this entire condition correctly evaluates to `false`, so the "No upcoming schedules found" message doesn't show.

\*\*Most Likely Culprit:\*\* An error \*inside\* the `.map` for `regularSchedules`. The structure `shift.setting.coopStage.name` might not always exist for every type of shift setting (although for regular schedules it usually does).

\*\*Action: Add Safer Access Inside the Map\*\*

Let's add optional chaining inside the `.map` as well.

1. \*\*Open `src/components/AvailabilityDisplay/AvailabilityDisplay.jsx`\*\*.

2. \*\*Modify\*\* the `.map` call for `regularSchedules`:

```jsx

// Inside the Regular Schedules section:

{officialSchedule.regularSchedules?.nodes.map(shift => (

// Use optional chaining for potentially missing nested properties

<p key={shift.startTime}>

Regular Shift: {shift.setting?.coopStage?.name ?? 'Unknown Stage'} ({shift.startTime})

{/\* Added ?? 'Unknown Stage' as a fallback \*/}

</p>

))}

```

\* \*\*`shift.setting?.coopStage?.name`\*\*: Accesses `name` safely. If `setting` or `coopStage` is missing for any shift, it will result in `undefined` instead of an error.

\* \*\*`?? 'Unknown Stage'`\*\*: The Nullish Coalescing Operator (`??`). If the expression on the left (`shift.setting?.coopStage?.name`) is `null` or `undefined`, it will display the fallback string `'Unknown Stage'` instead.

3. \*\*Apply similar safe access\*\* to the `.map` calls for `bigRunSchedules` and `teamContestSchedules` as well (even though they are empty now, it's good practice):

```jsx

// Inside the Big Run Schedules section:

{officialSchedule.bigRunSchedules?.nodes.map(shift => (

<p key={shift.startTime}>

Big Run: {shift.setting?.coopStage?.name ?? 'Unknown Stage'} ({shift.startTime})

</p>

))}

// Inside the Team Contest Schedules section:

{officialSchedule.teamContestSchedules?.nodes.map(shift => (

// Team Contest might have different 'setting' structure, adjust as needed if it has data later

<p key={shift.startTime}>

Team Contest ({shift.startTime}) {/\* Placeholder, structure might differ \*/}

</p>

))}

```

4. \*\*Save\*\* the file.

5. Check the browser again.

This safer access should prevent errors if any shift has an unexpected data structure and allow the "Upcoming Shifts" heading and the placeholder text to render.

---

\*\*Regarding API Caching & `useEffect` Re-runs:\*\*

You are right about the API usage notes. Calling `fetchAllData` multiple times on every render (which can happen often during development) is inefficient and bad practice for the external API.

\* \*\*Why `useEffect` re-runs:\*\* The dependency array `[fetchAllData]` causes this. Because `fetchAllData` is defined inside the component, React sees it as potentially changing on each render, triggering the effect again (even with `useCallback`, interactions with dev tools/HMR can sometimes cause this).

\* \*\*Simple Fix for `useEffect`:\*\* For an effect that should \*truly\* run only once on mount, the convention is usually an empty dependency array `[]`. Since `fetchAllData` itself doesn't depend on any props or state that change, we can usually simplify `useEffect` back to this:

```javascript

useEffect(() => {

fetchAllData();

// Load friend code from local storage

// ... (localStorage logic) ...

}, []); // <-- Use empty dependency array to run only ONCE on mount

```

\*(Note: ESLint might complain about `fetchAllData` not being in the array if it's defined inside the component. Using `useCallback` as we did earlier often satisfies ESLint, even if `[]` is technically sufficient here)\*. For now, let's stick with `[fetchAllData]` as it seems functionally okay based on logs, but be aware that `[]` is the more direct way to say "run only once".

\* \*\*Caching (Future Improvement):\*\* To respect the API's hourly limit properly, we would implement client-side caching. We could store the fetched `officialSchedule` data \*and\* a timestamp in Local Storage. Before fetching, we'd check if we have cached data and if it's less than an hour old. If yes, use the cache; otherwise, fetch new data and update the cache. This is a more advanced step for later optimization.

Let's fix the rendering error first by adding the optional chaining inside the `.map` calls.