For Frontend:

Interactive Event Seating Map — Front-End Take-Home Task
==

OVERVIEW
You will build a small React + TypeScript application that renders an interactive seating map for an event. Each seat should be clickable, show its details, and allow the user to pick up to 8 seats. The core solution should be completable in roughly 3 hours; polish and stretch goals are optional.

DELIVERABLES
- A Git repository (GitHub link or zipped archive) that runs with
  `pnpm install && pnpm dev`
- Source code in TypeScript with `strict` mode enabled
- A short README (2–3 paragraphs) explaining:
  - architecture choices and trade-offs
  - any incomplete features / TODOs
  - how to run tests (if included)

DATA (place in `public/venue.json`)
```json
{
  "venueId": "arena-01",
  "name": "Metropolis Arena",
  "map": { "width": 1024, "height": 768 },
  "sections": [
    {
      "id": "A",
      "label": "Lower Bowl A",
      "transform": { "x": 0, "y": 0, "scale": 1 },
      "rows": [
        {
          "index": 1,
          "seats": [
            {
              "id": "A-1-01",
              "col": 1,
              "x": 50,
              "y": 40,
              "priceTier": 1,
              "status": "available"
            },
            {
              "id": "A-1-02",
              "col": 2,
              "x": 80,
              "y": 40,
              "priceTier": 1,
              "status": "reserved"
            }
          ]
        }
      ]
    }
  ]
}
```
Explanation
- `map.width / map.height` define an SVG (or Canvas) drawing area in pixels.
- Seat coordinates are absolute; no layout algorithm is required.
- `status` is one of
  `available | reserved | sold | held`.

REQUIREMENTS
1. Load `venue.json` and render every seat in its correct position.
2. Keep rendering smooth (≈ 60 fps) for large arenas (≈ 15 000 seats).
3. A seat can be selected via mouse click **and** keyboard.
4. Display seat details (section, row, seat, price, status) on click or focus.
5. Allow selecting up to 8 seats; show a live summary with subtotal.
6. Persist the current selection after page reload (e.g. `localStorage`).
7. Provide basic accessibility:
   - `aria-label` on interactive elements
   - focus outline and keyboard navigation.
8. The UI must work on desktop and mobile viewport sizes.

TECH CONSTRAINTS
- React >= 18 and TypeScript (`"strict": true` in `tsconfig.json`).
- Use any additional libraries you feel are appropriate, but document why.
- Project scaffold: either Vite + React or Next.js 14 (app directory).
- ESLint / Prettier configuration is encouraged (no specific rules required).

OPTIONAL STRETCH GOALS
- Live seat-status updates over WebSocket; animate changes.
- Heat-map toggle that colours seats by price tier.
- "Find N adjacent seats" helper button.
- Pinch-zoom + pan for mobile (touch gestures).
- Dark-mode toggle that meets WCAG 2.1 AA contrast ratios.
- End-to-end tests with Playwright or Cypress.

SUBMISSION
- Push to a public or private GitHub repository and share the link, **or**
  send a zipped archive of the repository.
- Ensure `pnpm install && pnpm dev` starts the app without additional steps.

EVALUATION CRITERIA (internal rubric, shared for transparency)
- Correctness: seats render, can be selected, summary updates.
- Code quality: idiomatic React, clean TypeScript types, modular structure.
- Performance: smooth interaction with 15 000 seats on a mid-range laptop.
- Accessibility & UX: keyboard support, focus management, clear status.
- Tests & documentation: unit/integration tests, concise README.
- Discussion: ability to explain decisions and trade-offs in the follow-up
  interview.

GOOD LUCK!
We're excited to review your solution and discuss it with you. If any blocking
question arises during implementation, feel free to email us.

For Backend:
Expert-Level Express.js Assignment: User Data API with Advanced Caching, Rate Limiting, and Asynchronous Processing

Objective:

Develop a highly efficient Express.js API that serves user data with advanced caching strategies, rate limiting, and asynchronous processing to handle high traffic and improve performance.

Requirements:

1. **Setup an Express.js Server**:
    - Initialize a new Express.js application with appropriate middleware (e.g., body-parser, cors).
    - Use TypeScript for type safety and better maintainability.
2. **Advanced In-Memory Cache**:
    - Implement an in-memory cache using a Least Recently Used (LRU) cache strategy.
    - The cache should store user data for 60 seconds before invalidating it.
    - Maintain cache statistics, including cache hits, misses, and current cache size.
    - Implement a mechanism to automatically clear stale cache entries using a background task.
3. **Define API Endpoints**:
    - **GET /users/:id**:

        - Retrieve user data by ID.
        - If the data is available in the cache, return it immediately.
        - If not, simulate a database call by returning a mock user object after a delay of 200ms. The mock user object should contain at least **id**, **name**, and **email**.
        - If the requested user ID does not exist, return a 404 status code with a meaningful error message.
4. **Performance Optimization**:
    - Ensure that the cache is updated with the new user data only if the data is not already cached.
    - Implement a mechanism to handle concurrent requests for the same user ID efficiently. If one request is fetching data, subsequent requests for the same ID should wait for the first request to complete and then return the cached result.
5. **Rate Limiting**:
    - Implement a sophisticated rate-limiting strategy that allows for burst traffic handling. Allow a maximum of 10 requests per minute, with a burst capacity of 5 requests in a 10-second window.
    - Return a 429 status code with a meaningful message when the rate limit is exceeded.
6. **Asynchronous Processing**:
    - Introduce an asynchronous processing mechanism for simulating the database call. Use a queue (e.g., Bull or a simple array-based queue) to manage requests that need to fetch data from the "database".
    - Ensure that the API can handle multiple simultaneous requests without blocking.
7. **Bonus Features**:
    - **Manual Cache Management**: Implement a **DELETE /cache** endpoint to clear the entire cache.
    - **Cache Status**: Implement a **GET /cache-status** endpoint that returns the current cache size, number of cache hits, misses, and the average response time for requests.
    - **User Creation Endpoint**: Implement a **POST /users** endpoint to simulate creating a new user. The new user should be added to the mock data and subsequently cached.
8. **Testing**:
    - Use Postman or any API testing tool to test the endpoints.
    - Measure response times for the first request and subsequent requests to observe the caching effect.
    - Simulate high traffic to test the rate limiting and asynchronous processing functionality.

Mock User Data:

```
javascript
const mockUsers = {
  1: { id: 1, name: "John Doe", email: "john@example.com" },
  2: { id: 2, name: "Jane Smith", email: "jane@example.com" },
  3: { id: 3, name: "Alice Johnson", email: "alice@example.com" }
};
```

Submission:

- Submit your code in a GitHub repository.
- Include a README file with instructions on how to run your application and test the API.

- Provide a brief explanation of your caching strategy, rate limiting implementation, and how you handled asynchronous processing.

Evaluation Criteria:

- Code quality, structure, and adherence to TypeScript best practices
- Correctness and efficiency of the caching, rate limiting, and asynchronous processing implementations
- Performance optimizations and handling of concurrent requests
- Error handling and meaningful responses
- Clarity and completeness of documentation

Additional Challenge:

- Consider implementing a monitoring solution (e.g., Prometheus or a simple logging mechanism) to track API performance metrics over time, including response times, error rates, and cache performance.

Good luck!