Full-Stack Developer Assessment: Log Ingestion and Querying System

Part I: The Project Brief - A Log Ingestion and Querying System

Section 1.1: Introduction and Core Objective

This document outlines the requirements for a full-stack take-home assignment designed to assess proficiency in Node.js, React, and general software engineering principles. The project is scoped to be complete within a single day (approximately 6-8 hours of focused work) by a developer with around three years of full-stack experience.

The core objective is to build a self-contained "Log Ingestion and Querying System." This application will simulate a real-world developer tool used for monitoring and debugging applications. Such tools are fundamental in modern software development for providing visibility into system behavior, performance, and errors.[1] This task provides a practical context that is more indicative of real-world engineering challenges than abstract exercises or overly simplified applications like to-do lists.[3]

The system will consist of two primary components:

1. **A Log Ingestor:** A backend service built with Node.js that exposes an API endpoint to accept and store log data.
2. **A Log Query Interface:** A frontend application built with React that allows a user to search, filter, and view the logs that have been ingested.

This assignment is designed to evaluate a candidate's ability to build a complete, functional feature from front to back, touching upon API design, data handling, state management, and user interface development. It provides an opportunity to showcase knowledge of common building blocks in web applications, such as API integration, data manipulation, and creating a responsive user experience.[3]

Section 1.2: Functional Requirements (User Stories)

The application's functionality should be guided by the following user stories. These stories define the minimum viable product and represent the core features that must be implemented for a successful submission.

Log Ingestion

- **As a system,** I can send a POST request containing a single log entry in a predefined JSON format to a specific API endpoint, so that the log is durably stored for later analysis.

Log Viewing and Filtering

- **As a developer,** I can view all ingested logs in a clean, reverse-chronological list (most recent first) to get an immediate overview of recent activity.
- **As a developer,** I can perform a case-insensitive, full-text search across the message field of all logs to quickly find entries containing specific keywords or error messages.
- **As a developer,** I can filter the logs by their level (e.g., 'error', 'info', 'warning') using a dropdown or multi-select component to isolate logs of a certain severity.
- **As a developer,** I can filter logs to show only those that fall within a specific timestamp range by selecting a start and end date/time, enabling me to investigate incidents that occurred during a specific period.
- **As a developer,** I can filter logs by their resourceId by typing into a text field, allowing me to see all logs originating from a particular server, service, or application instance.
- **As a developer,** I can combine multiple filters simultaneously (e.g., filtering for 'error' level logs containing the word "database" within the last 24 hours from "server-1234") to perform highly specific and targeted searches.

Section 1.3: User Interface (UI) Guidance and Inspiration

While a pixel-perfect design is not required, the usability and clarity of the interface are important. The goal is to create a functional and intuitive tool for a developer audience. Instead of a rigid wireframe, this section provides guidance on the key components and suggests sources of inspiration, allowing for a degree of creative interpretation. This approach assesses the ability to translate functional requirements into a practical UI, a common task for full-stack developers.[6]

Key UI Components

1. **Filter Bar:** A prominent and persistent section, likely at the top of the page, should house all filtering controls. This area should be logically organized and clearly labeled. It must contain:

   - A text input field for the full-text search on the message field.
   - A dropdown or multi-select component for the level filter.
   - A text input field for the resourceId filter.
   - Two date/time input fields to define the timestamp range (start and end).
   - A "Search" or "Apply Filters" button, or preferably, the log view should update dynamically as filter values change.
2. **Log Results View:** The main area of the page should display the filtered log entries. The presentation should be clean and easy to scan. A tabular or list-based layout is recommended. Each log entry displayed should be clearly distinguishable from the next.
3. **Visual Cues for Log Levels:** To enhance readability and quick diagnosis, log entries should be visually distinct based on their level. For example, 'error' logs could have a red left border or a light red background, 'warning' logs could use yellow, and 'info' logs could use blue or grey. This demonstrates attention to user experience details.[6]

UI/UX Inspiration and Best Practices

Candidates are encouraged to draw inspiration from established, professional-grade logging and monitoring tools. The aim is to replicate their clarity and functionality, not their entire feature set.

- **Professional Tooling as a Benchmark:** Interfaces from tools like Grafana Loki, Datadog, or Splunk serve as excellent examples of effective log query UIs. Notice how they handle high-density information, provide powerful filtering mechanisms, and use color to convey meaning.[7] Dribbble also hosts many inspirational designs for log dashboards and developer tools that showcase clean layouts and modern aesthetics.[8] Designs such as "Prefect Logs," which clearly delineates log levels, and "Infrastructure change logs," which presents a clean dashboard, are particularly relevant.[8]
- **Interactive Filtering:** The filtering experience should be fluid. As a user modifies a filter (e.g., types in the search box or selects a new level), the log results should update dynamically without requiring a full page reload. This demonstrates a solid understanding of React's state management and event handling capabilities.[9] A more experienced developer might consider performance optimizations, such as debouncing the text input to prevent excessive API calls while the user is typing.

- **Clarity and Simplicity:** The design should prioritize simplicity and avoid clutter. A clean, minimalistic layout allows the log data itself to be the focus. The user should be able to understand how to use the interface immediately without instruction.[6]

The design of the UI is an opportunity to demonstrate a practical "product sense." A candidate who not only implements the required functionality but also presents it in a clean, logical, and user-friendly manner will be viewed more favorably. This simulates a real-world scenario where developers must often build functional interfaces based on specifications rather than pixel-perfect mockups.

Part II: Technical Specifications and Implementation Guidance

This section provides the technical constraints and specific implementation details necessary to complete the project. Adherence to these specifications is a primary evaluation criterion.

Section 2.1: Backend API Specification (Node.js & Express)

The backend server must be implemented using Node.js and the Express.js framework.

Data Persistence Mandate

A critical requirement of this assignment is the method of data persistence. **The application must use a single JSON file as its database.** The use of any external database systems (e.g., MongoDB, PostgreSQL, SQLite) or cloud-based storage services (e.g., Firebase, DynamoDB) is explicitly disallowed.

This constraint is intentional. It forces the implementation of all querying, filtering, and sorting logic directly within the Node.js application layer. This provides a direct and unambiguous signal of the candidate's proficiency with core JavaScript data manipulation techniques (e.g., Array.prototype methods like filter, map, sort) and their ability to handle data in memory. This approach simplifies the project's setup dependencies, allowing the candidate to focus their limited time on the application logic itself, which is the primary subject of evaluation.[11]

To facilitate this, candidates may use a library like node-json-db [14] for convenience in reading from and writing to the JSON file, or they may implement their own file system logic using Node's built-in

fs module.[15] The choice of implementation should be documented in the

README.md file.

Table 1: Log Data Schema

All log entries, both in the request body for ingestion and in the response body for queries, must adhere to the following JSON schema. This schema is based on common structured logging practices, where logs are treated as data with consistent key-value pairs rather than unstructured strings.[1]

Field	Type	Description	Example	Required
level	String	The severity level of the log. Must be one of: error, warn, info, debug.	"error"	Yes
message	String	The primary log message. This field will be the target of full-text searches.	"Failed to connect to database."	Yes
resourceId	String	The identifier of the resource that generated the log (e.g., a server name, container ID).	"server-1234"	Yes
timestamp	String	The ISO 8601 formatted timestamp indicating when the log event occurred.	"2023-09-15T08:00:00Z"	Yes
traceId	String	A unique	"abc-xyz-123"	Yes

		identifier used to correlate logs from a single request or transaction across multiple services.		
spanId	String	A unique identifier for a specific operation or "span" within a trace.	"span-456"	Yes
commit	String	The git commit hash of the code version that generated the log.	"5e5342f"	Yes
metadata	Object	A nested JSON object containing additional, unstructured context relevant to the log entry.	{"parentResourceId": "server-5678"}	Yes

Table 2: REST API Endpoint Specification

The backend server must implement the following RESTful API endpoints. The design adheres to standard REST conventions, using HTTP methods to denote actions and endpoint paths to represent resources.[17] A well-defined API contract is fundamental to successful full-stack development, and this table serves as that contract.

Method	Path	Description	Query Parameters / Request Body	Success Response	Error Response
POST	/logs	Ingests a single log entry and	**Body:** A single JSON object that	**Status:** 201 Created.	**Status:** 400 Bad Request if the

		persists it to the JSON file database.	strictly conforms to the Log Data Schema defined in Table 1.	**Body:** The log object that was successfully created and stored.	request body is missing, malformed, or fails schema validation. **Status:** 500 Internal Server Error for any other server-side failure during processing or persistence.
GET	/logs	Retrieves a list of log entries. This endpoint must support filtering based on the provided query parameters. All filters should be combinable (i.e., work together using an AND logic).	**Query Params (all optional):**level (string)message (string, for full-text search)resourceId (string)timestamp_start (ISO 8601 string)timestamp_end (ISO 8601 string)traceId (string)spanId (string)commit (string)	**Status:** 200 OK. **Body:** A JSON array of log objects that match all applied filter criteria. The array should be sorted in reverse chronological order by timestamp. If no logs match, an empty array `` should be returned.	**Status:** 500 Internal Server Error for any server-side failure during data retrieval or filtering.

Section 2.2: Frontend Application Specification (React)

The frontend client application must be built using React. Bootstrapping the project with a standard tool like create-react-app or Vite is recommended for a quick setup.[9]

Core Requirements

- **Component Architecture:** The application should be built using functional components and leverage React Hooks (primarily useState and useEffect) for state management and side effects.[9]

- **State Management:** There is no requirement to use a global state management library such as Redux or MobX. Component-level state managed with useState or state shared via the Context API is sufficient for the scope of this project.[9] However, candidates are free to use a more advanced library if they believe it is justified and should explain their reasoning in the README.md.
- **API Integration:** All communication with the backend API should be handled cleanly. The native fetch API or a library like axios are both acceptable choices.[9] API call logic should be well-organized, likely within useEffect hooks or custom hooks.
- **Responsiveness:** The UI should be designed for a standard desktop browser view. While a fully mobile-responsive design is not required, the layout should not break on typical desktop screen sizes.

The structure of the frontend application should reflect a clear separation of concerns. For example, UI components, API service functions, and custom hooks should be organized into logical directories. This demonstrates an understanding of maintainable and scalable frontend architecture.

Part III: Evaluation Criteria and Submission Guidelines

This section details how the submission will be evaluated and provides instructions on how to submit the completed project. Transparency in the evaluation process is key, so these criteria should be reviewed carefully.

Section 3.1: Core Competencies Assessed

The assignment is designed to measure a range of competencies that are critical for a successful full-stack developer:

- **Backend Logic:** The correctness and efficiency of the API implementation. This includes proper request handling, robust data filtering that works with combined criteria, and strict adherence to the specified data schema.
- **Frontend Logic:** The correct use of React principles, including component composition, state management, and handling of side effects for API integration. The implementation of all required UI functionality (filtering, searching, displaying

data) will be verified.

- **Code Quality & Best Practices:** The overall quality of the codebase. This is a broad category that includes readability, consistency in style, logical project structure (both frontend and backend), the appropriate use of comments, and effective error handling (e.g., try-catch blocks, sending appropriate HTTP error codes).
- **Problem Solving:** The candidate's approach to translating the project requirements into a functional application. This is assessed by observing the design choices made in the code.
- **Documentation:** The clarity, completeness, and professionalism of the README.md file. Good documentation is a critical, and often overlooked, engineering skill.

Section 3.2: Detailed Evaluation Rubric

To ensure a fair, consistent, and objective evaluation process, all submissions will be assessed against the following rubric.

Table 3: Evaluation Rubric

Criterion	Does Not Meet Expectations (0-1)	Meets Expectations (2-3)	Exceeds Expectations (4-5)
Backend API Implementation	API endpoints are missing, return incorrect status codes, or do not handle requests as specified. Filtering logic is non-functional, incorrect for multiple combined filters, or highly inefficient.	All API endpoints are implemented correctly according to the specification. Filtering works reliably for all specified fields, including when they are combined.	The API is robust, includes server-side validation for the POST request body, and provides meaningful error messages. The filtering logic is implemented in a clean, efficient, and easily extensible manner.
Frontend UI/UX and Functionality	The UI is missing key features (e.g., one or more filters), is non-functional, or is extremely difficult to	All required UI components are present and fully functional. Filtering and searching trigger	The UI is polished, intuitive, and responsive. State management is efficient and well-

	use. State management is buggy, leading to an inconsistent UI.	the correct API calls and the results are displayed correctly. The application feels stable.	organized. Thoughtful UX improvements are present, such as loading indicators during API calls, debouncing on the text search input, or a "clear filters" button.
Code Quality and Structure	Code is disorganized, difficult to read, and inconsistent in style. There is no clear project structure, with logic mixed inappropriately across files.	The code is generally clean, well-organized into logical components/modules , and readable. It follows standard naming conventions and formatting for the respective languages.	The code is exceptionally clean, well-structured, and demonstrates a strong command of software design principles. Components/module s are well-composed, reusable, and have a clear purpose.
Data Persistence Logic	The application fails to use the specified JSON file for storage, or the implementation is fundamentally broken, leading to data loss or file corruption on writes.	The application correctly uses a single JSON file for persistence. Data is successfully read on server start and written to upon ingestion. The filtering logic correctly reads from this data source.	The data persistence logic is implemented robustly. For example, it might include safeguards against race conditions during writes or handle potential file read/write errors gracefully (e.g., creating the file if it doesn't exist).
Documentation (README.md)	The README.md is missing, or it lacks the essential instructions needed to set up and run the project, forcing the evaluator to guess or	The README.md includes clear, correct, and easy-to-follow instructions on how to install dependencies and run both the frontend	The README.md is comprehensive. In addition to setup instructions, it explains key design decisions, trade-offs considered (e.g., why

	inspect package.json.	and backend servers.	a certain library was or was not used), and any assumptions made during development.

Section 3.3: Optional Bonus Challenges

For candidates who complete the core requirements with time to spare, the following optional challenges provide an opportunity to showcase additional skills. These are **not required**, and a submission will not be penalized for their absence.

1. **Real-Time Log Ingestion:** Implement WebSockets [3] on the server and client. When a new log is ingested via the
   POST /logs endpoint, the server should push the new log data to all connected clients, causing the UI to update in real-time without requiring a refresh.
2. **Basic Analytics View:** Add a small dashboard component to the UI that displays a simple chart (e.g., using a library like Chart.js or Recharts) showing the count of logs by level over the currently filtered time range.
3. **Containerization:** Provide a Dockerfile and a docker-compose.yml file that allows the entire full-stack application (backend and frontend) to be built and run with a single docker-compose up command.
4. **Unit Testing:** Add a few meaningful unit tests for a critical piece of the application's logic. An excellent candidate for this would be the backend filtering function, with tests covering various combinations of filters.

Section 3.4: Submission Instructions

Please follow these instructions carefully to ensure your submission can be reviewed efficiently.

1. The entire project, including both the frontend and backend code, must be contained within a single Git repository.
2. Submit a link to this repository (e.g., on GitHub, GitLab). Please ensure the repository is public or that you have provided access to the evaluators.
3. The root of the repository must contain a README.md file.
4. The README.md file is a critical part of your submission and must include:
   - A brief overview of the project and your approach.
   - Clear, step-by-step instructions on how to install all necessary dependencies

for both the frontend and backend applications.

- Unambiguous instructions on how to run both the frontend and backend servers.
- A brief explanation of any significant design decisions, trade-offs, or assumptions you made. For example, explain your choice of libraries or your approach to the file-based persistence.
- (If applicable) Instructions on how to run any tests or bonus features you implemented.