COLLEGE CODE: 8203

COLLEGE: AVC COLLEGE OF ENGINEERING

DEPARTMENT: INFORMATION TECHNOLOGY

STUDENT NM-ID: 4D2198FA90720AD8F834BBD4C69121C9

ROLL NO: 23IT100

DATE:03-10-2025

Completed the project named as Phase 4

TECHNOLOGY PROJECT NAME: Admin dashboard with charts

SUBMITTED BY,

NAME: Sivarajaganapathi S

MOBILE NO: 7845102808

Phase 4-Enhancements & Deployment

1.Additional Features

This page outlines the new functionalities and data interaction tools required to elevate the admin dashboard from a basic viewer to a powerful administrative tool.

1.1 Dynamic Data Filtering Controls

The primary enhancement is implementing interactive controls that allow administrators to manipulate the data displayed in the charts. This ensures the dashboard is useful for granular analysis, not just high-level overviews.

- **Date Range Selector:** Integrate a custom date picker component (e.g., using a library like **React Date Picker**). This allows the admin to select a Start Date and End Date. The selected range is sent as parameters in the secured API request.
- **Categorical Filter Dropdowns:** Add dropdowns or checkboxes to filter metrics by attributes defined in the MongoDB schema, such as Region, Product Category, or User Role.
- **Backend Requirement:** The Express API must be enhanced to accept these filter parameters and correctly incorporate them into the MongoDB **Aggregation Pipeline** ($match stage) to return the subset of data requested.

1.2 Chart Interactivity and Customization

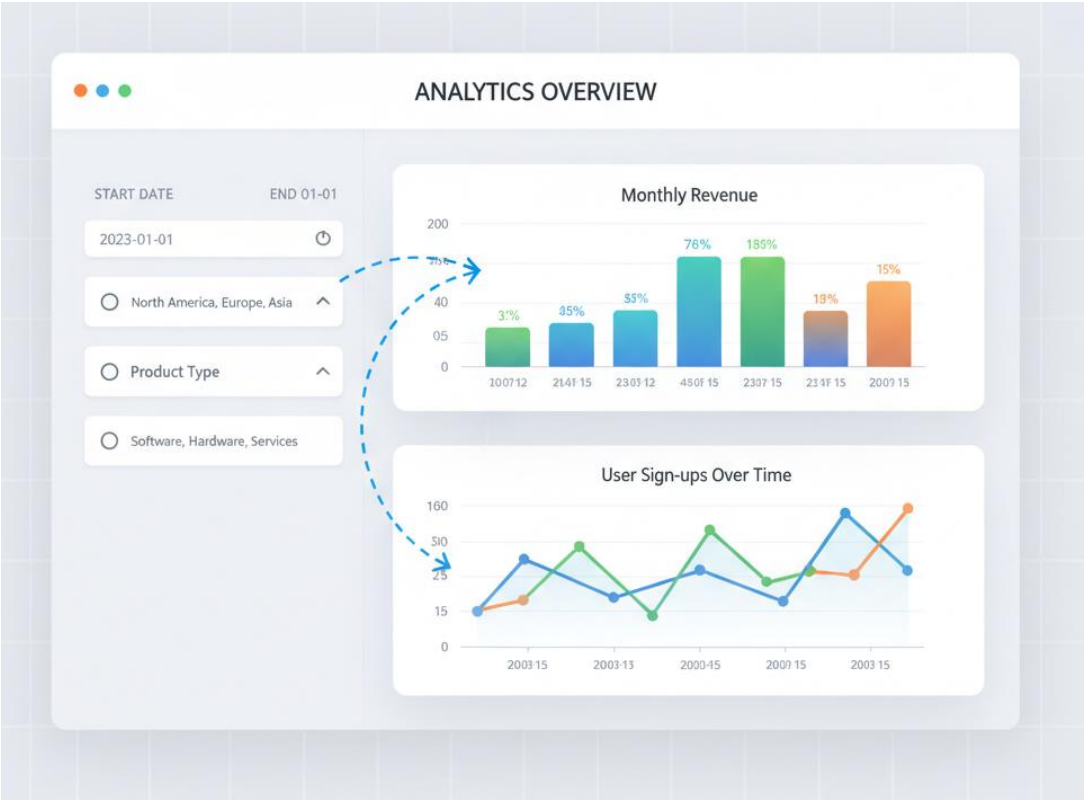
Improvements that allow the administrator to better visualize and interpret the existing data.

- **Metric Toggling:** Implement simple state management to allow an administrator to toggle between displaying Sales Revenue and User Signups on the same line graph without changing the page. This involves updating the datasets property passed to Chart.js.
- **Data Drill-Down:** For Bar or Pie Charts showing grouped data (e.g., Sales by Region), implement a click handler that, when a segment is clicked, triggers a detailed view or a new chart showing the data broken down further (e.g., Sales by City within that Region).

1.3 Data Export Utility

Providing administrators the capability to use the displayed data outside the dashboard.

- **CSV Export Button:** Add an **Export to CSV** button near each primary chart. This function should take the current data object (already filtered and transformed for the chart) and use a client-side library (like react-csv) to generate and download the file. This is crucial for offline reporting.



2.UI/UX Improvements

This section focuses on polishing the user interface for professional use, emphasizing responsiveness, feedback, and accessibility standards.

2.1 Responsive Design and Adaptive Charts

The dashboard must perform flawlessly across all devices.

- **Mobile-First Adaptation:** Ensure the main Dashboard Layout (navigation, header, and chart grids) collapses appropriately for mobile screens.
- **Chart Resizing:** Configure **Chart.js options** to ensure charts maintain their aspect ratio and legibility when the container size changes. This often involves setting maintainAspectRatio: false and using responsive CSS units.

2.2 User Feedback and Status Messaging

Providing immediate and clear feedback is essential for a good user experience.

- **Skeleton Loaders:** When the component mounts or a filter is applied, display skeleton screens over the chart areas. This acknowledges the user's action and masks API latency, improving the perceived performance.
- **Toast Notification System:** Implement a non-intrusive notification library (like React-Toastify) to handle:

- **Success:** "Metrics data updated successfully."
- **Error:** "API Connection Failed. Please check network."
- **Warning:** "Session expiring soon."

2.3 Theme and Aesthetics

- **Dark Mode Implementation:** Integrate a **Context API** in React to manage the application's theme state. This allows the administrator to toggle to a Dark Mode, reducing eye strain, which is important for systems used extensively. This requires styling all core components (charts, buttons, backgrounds) for both themes.

2.4 Accessibility (A11Y)

Ensuring the dashboard is usable by individuals with disabilities is a professional requirement.

- **ARIA Attributes:** Add appropriate ARIA attributes to interactive elements (buttons, links, form inputs) to aid screen reader navigation.
- **Keyboard Navigation:** Confirm that all form fields and dashboard links are reachable and operable using only the Tab key.

3.API Enhancements

This details the critical steps taken on the Node.js/Express backend to ensure the API is fast, scalable, and efficient when handling data requests from MongoDB.

3.1 MongoDB Aggregation Optimization

This is the most critical area for performance, as it shifts heavy computation from the Express server to the MongoDB database.

- **Targeted Aggregation:** Refine all metric endpoints to use the **MongoDB Aggregation Framework**. Instead of using simple find() queries, use a pipeline starting with $match (for filtering), followed by $group (for calculating sums/averages), and finally $sort (for chronological output).
- **Projection Optimization:** Use the $project stage in the pipeline to explicitly select *only* the necessary fields for the chart. This reduces network load and processing time.

3.2 Server-Side Caching Strategy

Implementing caching reduces redundant database queries for data that doesn't change instantly.

- **Caching Middleware:** Use a package like **node-cache** on the Express server.
- **Implementation:** Cache the JSON response for key metric endpoints (/api/admin/metrics) for a short duration (e.g., 5 minutes). When a request comes in, the server checks the cache; if fresh, it serves the cached data instantly.

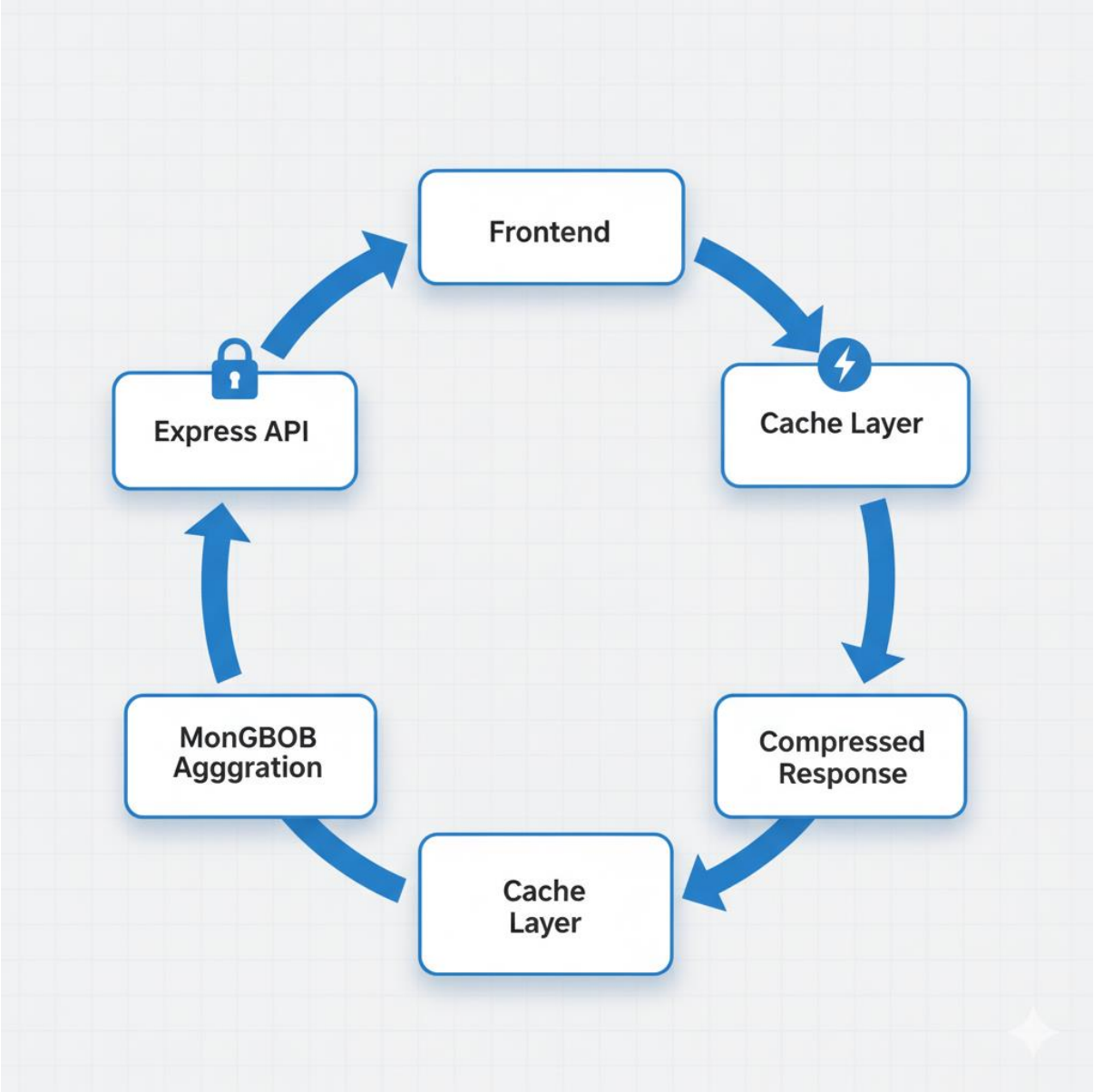
3.3 Response Compression

- **Gzip Compression:** Integrate the compression middleware into Express. This automatically compresses the JSON response data before sending it over the network, drastically reducing the data transfer size and improving load times, especially for users with slower connections.

3.4 API Security Hardening

Beyond basic JWT, these steps lock down the server environment.

- **Strict CORS Policy:** Update the Cross-Origin Resource Sharing (CORS) configuration to **only allow requests** from the known, deployed frontend domain (e.g., https://admin-dashboard.netlify.app), preventing malicious external domains from accessing the API.
- **Rate Limiting Implementation:** Use express-rate-limit on all exposed routes, specifically applying a stricter limit (e.g., 5 attempts per minute) to the /api/auth/login endpoint to effectively block brute-force attacks.



4.Performance & Security Checks

This section focuses on the final audit and deployment preparation, ensuring the application meets production security and scalability standards, often required by environments like **IBM Cloud**.

4.1 Security Audits and Vulnerability Mitigation

A final systematic review of security posture.

- **Input Sanitization and Validation:** Verify that every user input, including URL query parameters for filtering, is rigorously validated on the backend. Employ libraries like **Joi** to define strict schemas for expected data types and reject any input that deviates, mitigating SQL/NoSQL injection risks.

- **Dependency Auditing:** Run npm audit –production and address all high-severity vulnerabilities in dependencies like jsonwebtoken, bcrypt, and Express packages.
- **Security Headers (Helmet):** Install and configure the **Helmet** middleware in Express. Helmet sets critical HTTP headers like Strict-Transport-Security (HSTS) and X-Content-Type-Options to protect against common web vulnerabilities like XSS and clickjacking.

4.2 Database Indexing Confirmation

Performance hinges on fast database lookups.

- **Indexing Verification:** Confirm that permanent indexes are applied in the MongoDB Atlas instance on all frequently queried fields:
    - **Metrics Collection:** Index on date and category fields.
    - **Users Collection:** Index on username (unique index).

4.3 JWT and Session Management Review

- **Token Expiration:** Ensure the JWT lifespan is set appropriately (e.g., 1 hour) and that the frontend gracefully handles an expired token by automatically logging the user out and redirecting to the login page.
- **Secret Protection:** Verify that the JWT_SECRET is a complex, long string and is only stored as an environment variable in the backend's deployment environment.

4.4 Environmental Variable Management

- **Segregation:** Ensure a clear separation between development (.env file) and production (cloud platform settings) environment variables. Production credentials must never be committed to the Git repository.

5. Testing of Enhancements

This section details the critical testing protocols required to ensure all enhancements are stable and the entire application flow is correct.

5.1 Unit and Component Testing

Verifying the smallest parts of the application function correctly in isolation.

- **JWT Logic:** Unit tests for the jsonwebtoken generation and verification functions to ensure correct payload encoding and signature validation.
- **Data Transformation:** Unit tests for the React utility functions that transform the raw JSON API response into the labels and datasets format required by Chart.js.

5.2 Integration Testing (API/DB Flow)

Testing the interconnectedness of the core technologies.

- **Secured Route Test (Supertest):** Use a tool like **Supertest** within a Jest environment to simulate HTTP requests to the Express API. Test cases must include:
    1. Requesting /api/admin/metrics **without** a token (→ Expect 401).
    2. Requesting /api/admin/metrics **with** a valid token (→ Expect 200 and correct data schema).
    3. Requesting /api/admin/metrics **with** a date filter parameter (→ Expect 200 and data matching the filter).

5.3 End-to-End (E2E) Testing

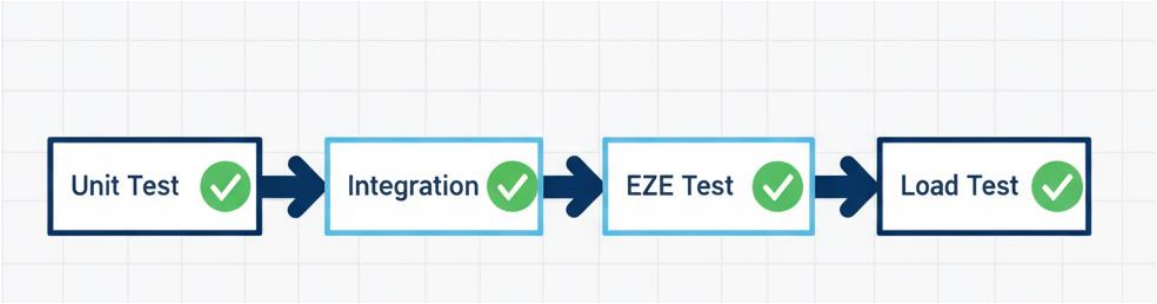
Simulating the full administrator user journey from start to finish.

- **Tool:** Use **Cypress** or **Playwright**.
- **Test Scenarios:**
    1. Successful login and redirection to the dashboard.
    2. Applying a date filter and verifying the chart updates visually.
    3. Attempting an invalid login and verifying the error notification appears.
    4. Testing the UI responsiveness by changing the viewport size during the test run.

5.4 Performance and Load Testing

Assessing the application's stability under realistic usage conditions.

- **Load Simulation:** Utilize tools like Loader.io or JMeter to simulate 10-20 concurrent administrators hitting the /api/admin/metrics endpoint.
- **Metrics:** Measure the API Latency (response time) and Database CPU/Memory utilization. The goal is to ensure the response time remains below 500ms even under load.



6.Deployment(Netlify, Vercel, or Cloud Platform)

This final page details the necessary steps for transitioning the application into a live production environment and establishing a reliable maintenance pipeline.

6.1 Split Deployment Architecture

The project uses a specialized approach for each component to maximize performance and security.

- **Frontend (React) Deployment:** Use **Netlify or Vercel**. These platforms offer global Content Delivery Networks (CDNs) for static assets, resulting in faster load times worldwide.
    - **Configuration:** The production build must be pointed to the live backend URL using a platform environment variable (REACT_APP_API_URL).
- **Backend (Node/Express) Deployment:** Use a robust PaaS (Platform as a Service) like **Render** or a dedicated cloud service like **IBM Cloud** or **AWS EC2/Elastic Beanstalk**. This provides a reliable, scalable environment for the server-side logic and database connections.
- **Database (MongoDB):** Continue using **MongoDB Atlas** as the fully managed service for the production database.

6.2 Continuous Integration/Continuous Deployment (CI/CD)

Automating the process from code commit to live deployment is crucial for maintenance.

- **Pipeline Setup:** Configure **GitHub Actions** or the native CI/CD tools of the chosen platform (Netlify/Render).
- **Workflow:**
    1. **Code Push:** Code is pushed to the main branch of the Git repository.
    2. **Test Run:** Automated unit and integration tests are executed. **Deployment fails if tests fail.**
    3. **Backend Deployment:** The Express application is rebuilt and deployed to the cloud platform (e.g., Render/IBM Cloud).

4. **Frontend Deployment:** The React application is built and deployed to Netlify/Vercel.

6.3 Post-Deployment Configuration and Monitoring

- **Domain and SSL:** Secure a custom domain and ensure both the frontend and backend are accessible exclusively via **HTTPS** (SSL/TLS encryption).
- **Logging and Monitoring:** Implement a production logging tool (e.g., **Winston** for Express) to track errors, API request times, and authentication failures. Configure cloud platform monitoring tools to alert the team if server CPU, memory, or network traffic spikes unexpectedly.
- **Final Access Restriction:** As a final security measure, ensure that the MongoDB Atlas database is configured to **only allow connections from the deployed backend server's IP address** (IP Whitelisting), blocking direct access from any other external source.