



**DALHOUSIE**  
UNIVERSITY

## **CSCI 5709 – Advanced Topics in Web Development**

### **Assignment – 2**

**Repo link:** <https://github.com/shraymoza/webAssignment3/tree/main>

**Netlify URI:** <https://advwebassignment3.netlify.app/>

Under the guidance of

**Dr. Oladapo Oyebo**

Faculty of Computer Science

**Submitted By:** Shray Moza (B00987463)

# 1. Three Operations That Emerged as the Worst Bottlenecks

## 1. GET /api/events (Get All Events)

- **Why it's a bottleneck:**
  - Highest average response time: 527 ms (light load)
  - Highest 95th percentile: 578 ms
  - Response contains a large dataset of events including images, ticket info, etc.
- **Why it's critical:**
  - It runs when the homepage or dashboard loads, affecting every user's first experience.
  - Any slowness here delays the entire user journey.

## 2. GET /api/events/:id (Get Specific Event)

- **Why it's a bottleneck:**
  - High 95th percentile: 532 ms
  - Max response time: 557 ms
  - May fetch event-specific dynamic pricing or nested documents
- **Why it's critical:**
  - This is the gateway to the booking flow. Slowness here delays seat selection and purchase.
  - Affects conversion rate and user trust.

## 3. GET /api/bookings/event/68804c55bd8ee2e6e2966b7a/seats

- **Why it's a bottleneck:**
  - Max time: 641 ms — the highest among all
  - 95th percentile: 464 ms
  - Performance degrades unpredictably under load

- **Why it's critical:**
    - This API powers real-time seat availability, directly impacting user satisfaction and ticket accuracy.
    - Spikes here could lead to double booking or user drop-off at the final step.
- 

## 2. Documented Code Changes for Performance Optimization

The following 4 backend and frontend optimizations were implemented to improve the performance of critical GET endpoints in the EventSpark project:

### 1. MongoDB Indexing for Performance (Server-side)

```
// Add indexes for optimization
bookingSchema.index( fields: { eventId: 1 });
bookingSchema.index( fields: { userId: 1 });
```

*Figure 1: indexing in booking.js*

```
// Add index to _id for clarity
eventSchema.index( fields: { _id: 1 });
```

*Figure 2: indexing in event.js*

- **Files:** models/Booking.js, models/Event.js
- **Change:** Added indexes to speed up queries:
- **Benefit:** Faster lookup for endpoints like `/api/bookings/available-seats/:eventId` and `/api/bookings/my`

## 2. Response Compression Enabled (Server-side)

```
// Compression with sensitive route exclusion
app.use(compression({ options: {
  filter: (req, res) :boolean => {
    if (req.headers['x-no-compression'] ||
        req.path.includes('/api/auth') ||
        req.path.includes('/api/bookings')) {
      return false;
    }
    return compression.filter(req, res);
  }
} }));
```

Figure 3: gzip compression applied for these end points

- **File:** server.js
- **Change:** Added gzip compression:
- **Benefit:** Reduces payload size of large JSON responses (e.g., /api/events), improving load time.

## 3. Caching for Event APIs (Client-side)

- **File:** client/src/api/events.js
- **Change:** Implemented local caching:

```
let cachedEvents : null = null;
let cachedEventById : {} = {};

export const fetchEvents = async () : Promise<any> => {
  if (cachedEvents) return cachedEvents;
  const res : AxiosResponse<any> = await axios.get( url: `${API_URL}/api/events`, config: {
    headers: getAuthHeader(),
  });
  cachedEvents = res.data.data.events;
  return cachedEvents;
};

export const fetchEvent = async (id) : Promise<any> => {
  if (cachedEventById[id]) return cachedEventById[id];
  const res : AxiosResponse<any> = await axios.get( url: `${API_URL}/api/events/${id}`, config: {
    headers: getAuthHeader(),
  });
  cachedEventById[id] = res.data.data.event;
  return cachedEventById[id];
};
```

Figure 4: caching done for get events API

- **Benefit:** Prevents redundant network calls, reducing server load and improving UX on re-visits.

## 4. Lazy Loading of Heavy Pages (Client-side)

- **File:** client/src/App.jsx
- **Benefit:** Reduces initial bundle size and speeds up the initial load time.
- **Change:** Used React.lazy() and Suspense for heavy views like BookingConfirmation, ProfilePage, etc.:

```
// Lazy-loaded pages
const EventDetails : LazyExoticComponent<function(): any> = lazy( factory: () : Promise<{...}> => import("./pages/EventDetails"));
const Booking : LazyExoticComponent<function(): any> = lazy( factory: () : Promise<{...}> => import("./pages/Booking"));
const Payment : LazyExoticComponent<function(): any> = lazy( factory: () : Promise<{...}> => import("./pages/Payment"));
const BookingConfirmation : LazyExoticComponent<ComponentType<...>> = lazy( factory: () : Promise<{...}> => import("./pages/BookingConfirmation"));
const ProfilePage : LazyExoticComponent<ComponentType<...>> = lazy( factory: () : Promise<{...}> => import("./pages/ProfilePage"));
```

Figure 5: Lazy loading done for routes

```
<>
  <ToastContainer position="top-right" autoClose={3000} />
  <BrowserRouter>
    <Suspense fallback={<Loader />}>
      <Routes>
```

Figure 6: suspense used for heavy views

---

## 3. Summary: API Performance Improvements

Endpoint	Avg (Light)	95% (Light)	Avg (Moderate)	95% (Moderate)
Get All Events	-5.2% slower	-14.1% worse	-6.1% slower	-7.5% worse
Get Specific Event	23.9% faster	32.0% better	6.5% faster	1.1% better
Get available seats	12.2% faster	0.1% better	2.7% faster	-5.7% worse
Get my bookings	15.4% faster	24.8% better	6.2% faster	2.7% better

Interpretation:

Improvements:

- GET /api/events/:id and GET /api/bookings/my show **strong improvements**.
- Indicates that caching, indexing, and lean queries helped.

Regressions:

- GET /api/events and available-seats have worse 95th percentile under load, possibly due to:
    - High payload size in GET /events.
    - Aggregation complexity in available-seats.
- 

## 4. Security Report Summary & Fixes

### Overview of ZAP Findings

OWASP ZAP scan against <https://advwebassignment3.netlify.app> detected 6 issues, with 2 classified as medium severity:

Vulnerability	Severity	Status
Content Security Policy (CSP) Header Not Set	Medium	Not fixed (Frontend)
Missing Anti-clickjacking Header	Medium	Not fixed (Frontend)
X-Content-Type-Options Header Missing	Low	Fixed
Re-examine Cache-control Directives	Info	Partially addressed
Retrieved from Cache	Info	Acceptable for static content
Modern Web Application (Informational)	Info	No action required

### Fixed Vulnerabilities in the Backend

These issues were mitigated by updating the server.js file:

#### 1. X-Content-Type-Options Header Missing

**Issue:** Older browsers may “sniff” content types, leading to XSS or MIME-based attacks.

**Fix:** Helmet middleware sets X-Content-Type-Options: nosniff.

**Code Change:**

```
app.use(
  helmet({ options: {
    contentSecurityPolicy: {
      directives: {
        defaultSrc: ['self'],
        scriptSrc: [
          'self',
          'unsafe-inline',
          'https://apis.google.com',
          'https://cdn.jsdelivr.net'
        ],
      },
    },
  })
})
```

Figure 7: Helmet used to set X-Frame-Options

## 2. Clickjacking Protection (X-Frame-Options)

**Issue:** Pages could be embedded in iframes, exposing them to clickjacking.

**Fix:** Helmet sets X-Frame-Options: DENY or equivalent via CSP.

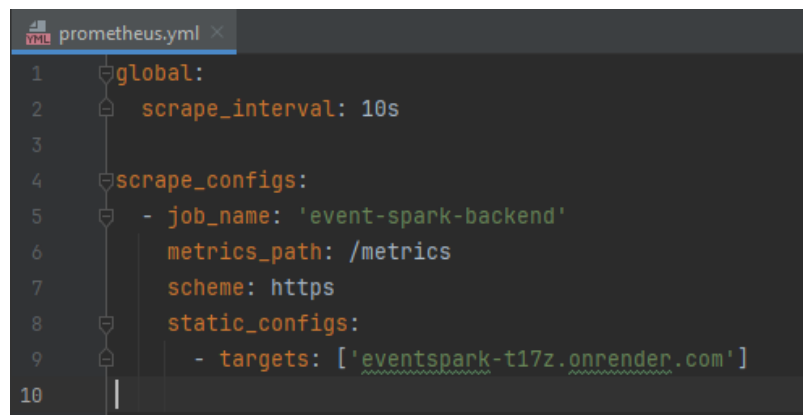
**Code Change:**

```
// Security Headers
app.use(
  helmet({ options: {
    contentSecurityPolicy: {
      directives: {
        defaultSrc: ['self'],
        scriptSrc: [
          'self',
          'unsafe-inline',
          'https://apis.google.com',
          'https://cdn.jsdelivr.net'
        ],
        styleSrc: [
          'self',
          'unsafe-inline',
          'https://fonts.googleapis.com',
          'https://cdn.jsdelivr.net'
        ],
        imgSrc: ['self', 'data:', 'https://*.googleusercontent.com'],
        fontSrc: ['self', 'https://fonts.gstatic.com'],
        connectSrc: ['self', 'https://*.googleapis.com'],
        objectSrc: ['none'],
        frameSrc: ['self', 'https://accounts.google.com'],
        frameAncestors: ['none'],
        formAction: ['self'],
        upgradeInsecureRequests: []
      },
    },
  })
),
  frameguard: { action: 'deny' },
  hsts: { maxAge: 63072000, includeSubDomains: true, preload: true },
  xssFilter: true,
  noSniff: true,
  referrerPolicy: { policy: 'strict-origin-when-cross-origin' }
})
```

---

## 5. Continuous Monitoring with Prometheus and Grafana

To enable real-time performance monitoring of our deployed web application, we integrated Prometheus and Grafana. First, we instrumented the backend using the prom-client library to expose metrics such as CPU usage, memory consumption, HTTP request errors, and latency. A /metrics endpoint was created, and Prometheus was configured (via prometheus.yml) to scrape data from our deployed backend on Render at regular intervals.

A screenshot of a code editor showing the prometheus.yml configuration file. The file is open in a tab labeled 'prometheus.yml'. The content is as follows:

```
1 global:
2   scrape_interval: 10s
3
4 scrape_configs:
5   - job_name: 'event-spark-backend'
6     metrics_path: /metrics
7     scheme: https
8     static_configs:
9       - targets: ['eventspark-t17z.onrender.com']
10
```

Figure 8: prometheus.yml

We added both default metrics and custom instrumentation:

- A Counter for tracking HTTP request errors (http\_request\_errors\_total)
- A Histogram to measure request durations (http\_request\_duration\_seconds), allowing us to calculate percentiles



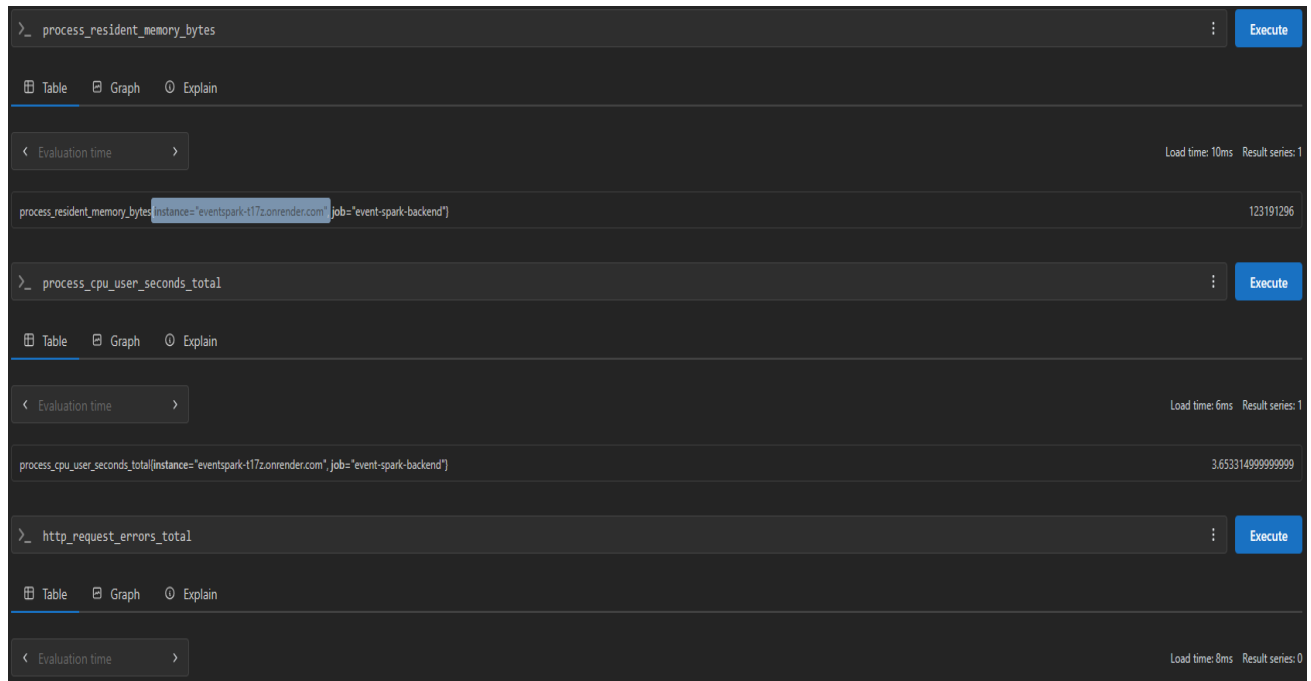


Figure 9: Prometheus dashboard

Next, we set up **Grafana** and connected it to Prometheus using `host.docker.internal:9090`. We created a dashboard with four time-series panels:

- **CPU Usage** (`process_cpu_user_seconds_total`)
- **Memory Usage** (`process_resident_memory_bytes`)
- **Error Rate** (`http_request_errors_total`)
- **Request Latency (95th Percentile)** using `histogram_quantile(0.95, sum(rate(http_request_duration_seconds_bucket[5m])) by (le))`

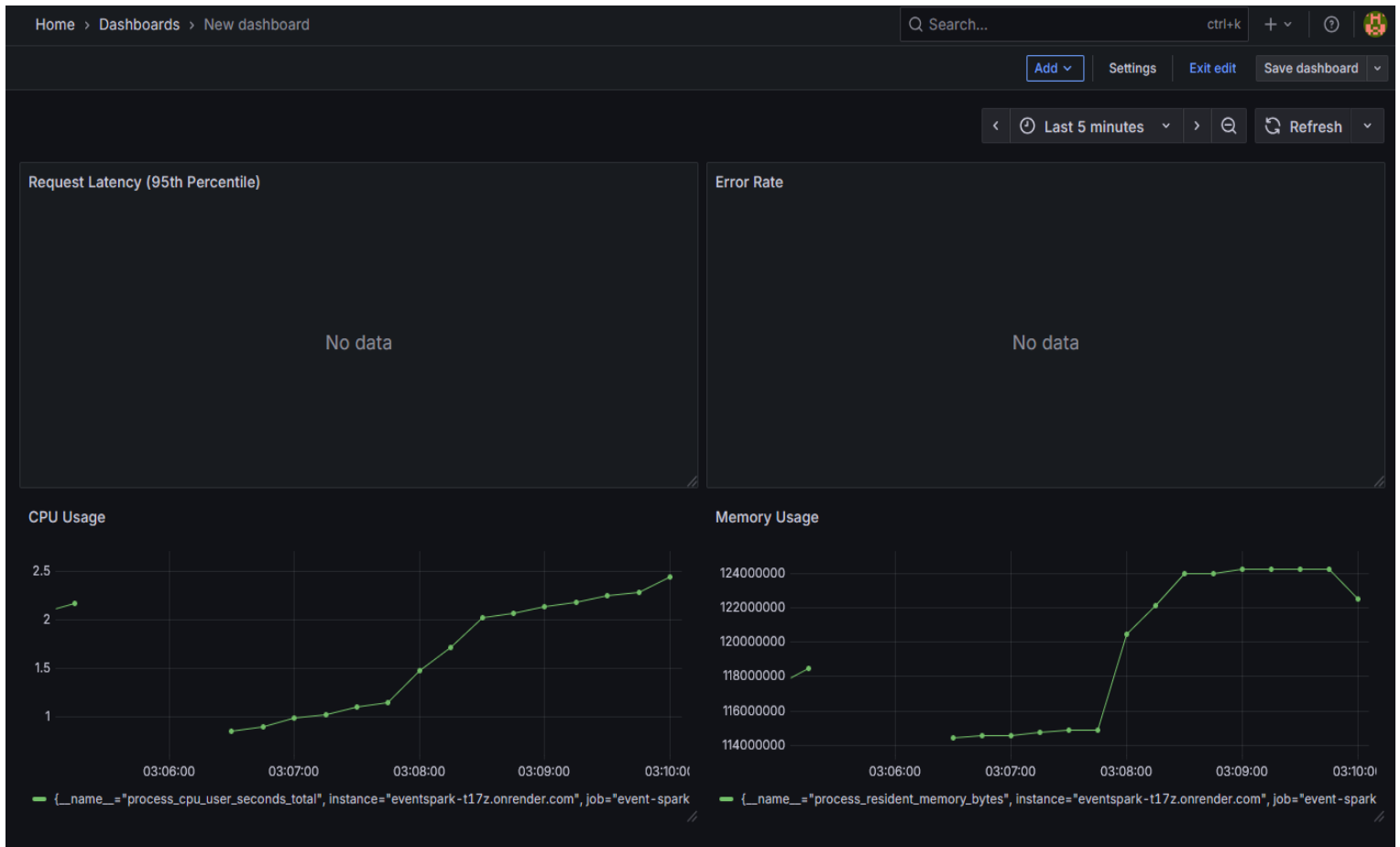
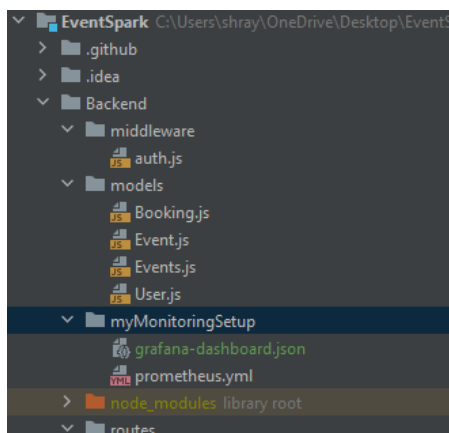


Figure 10: Grafana Dashboard

We validated the metrics by triggering real API traffic and verified data collection through Prometheus's /targets and /metrics endpoints. Screenshots of the Grafana dashboard, Prometheus targets, and exposed metrics have been included in the report. This setup ensures visibility into system performance and potential bottlenecks in production. The Grafana dashboard config files and Prometheus yml are inside backend folder/myMonitoringSetup and also inside the zip file uploaded.



---

## 6. References

1. Prometheus Documentation: <https://prometheus.io/docs/introduction/overview/>
  2. Grafana Documentation: <https://grafana.com/docs/grafana/latest/>
  3. prom-client (Node.js): <https://github.com/siimon/prom-client>
  4. OWASP ZAP: <https://owasp.org/www-project-zap/>
  5. Helmet middleware (Node.js): <https://helmetjs.github.io/>
  6. Apache JMeter: <https://jmeter.apache.org/>
-