Chetachi Ezikeuzor  
Jun 1, 2025

Milestone Three: Narrative for Enhanced Artifact

1. Artifact Overview

The artifact I selected for this milestone is Travlr, the full‐stack travel planning application I originally built in early 2025 using Node.js, Express, MongoDB, and Angular. Travlr enables users to register, log in, and manage personalized trip itineraries through an intuitive web interface. While the core functionality was solid, the project’s data‐handling and algorithmic aspects were initially basic—there was no proper pagination, filtering relied on client‐side logic, and database queries fetched entire collections without optimization. Because I wanted to showcase my growing expertise in algorithms and data structures, Travlr was a natural fit: it already had real‐world data flows (trips, costs, user registrations) that would benefit from thoughtful, server‐side algorithmic enhancements.

2. Justification for Inclusion & Skills Demonstrated  
I chose Travlr because it encapsulates several key challenges around data modeling, efficient querying, and server‐side performance. Specifically:

1. Pagination & Sorting Logic  
   Originally, GET /api/trips returned every trip document, forcing the frontend to handle large datasets. By adding server‐side pagination (via .skip()/.limit()), sorting (using indexed fields), and response metadata (page, totalPages, totalTrips), I demonstrated my ability to design scalable data retrieval solutions that avoid large client‐side loops.
2. Use of .lean() for Performance  
   By returning lean JavaScript objects instead of full Mongoose documents, I reduced memory overhead and improved read performance. This change exemplifies applying a “right data structure” mindset: choose the simplest form that meets requirements.
3. Indexing & Query Efficiency  
   I added MongoDB indexes on frequently queried fields (code, name, createdAt, updatedAt), ensuring that lookups for single trips (findOne({ code })) and sorted queries (.sort({ [sortBy]: 1 })) run in logarithmic time rather than scanning the entire collection.
4. Algorithmic Cost Calculation  
   Moving cost computations into a model method (tripSchema.methods.calculateCost) illustrates encapsulating logic close to data, reducing redundant calculations in route handlers and allowing for easy extension (for example, seasonal discounts or currency conversions).
5. Defensive Input Sanitization & Type Checking  
   Although sanitization is less of a pure “algorithm” topic, ensuring that numeric query parameters (page, limit) and string fields (sortBy, tripCode) are validated and coerced involves algorithmic thinking about edge cases and fallback values.

Taken together, these enhancements demonstrate:

* Designing efficient server‐side data retrieval & filtering (Outcome 3).
* Applying appropriate data structures (lean documents, indexed fields) to optimize performance (Outcome 4).
* Balancing trade‐offs—ensuring security via sanitization without sacrificing throughput (Outcome 5).

3. Planned vs. Implemented Enhancements  
During Milestone One, I committed to strengthening Travlr’s data‐handling workflows. Here’s how each planned enhancement aligns with what I completed:

1. Server‐Side Pagination
   * Planned: Implement GET /api/trips?page=:page&limit=:limit.
   * Implemented: I sanitize incoming page/limit via sanitizeQueryParam(), defaulting to page = 1 and limit = 10. Using countDocuments() I calculate totalTrips and use .skip()/.limit() so the backend only returns a single “page” of trips per request. The JSON response now includes { page, totalPages, totalTrips, trips: [ … ] }, giving the frontend exactly what it needs to render pagination controls without loading the entire collection.
2. Sorting via Whitelist
   * Planned: Only allow sorting on certain fields to avoid injection or nonsense sorts.
   * Implemented: I introduced an ALLOWED\_SORT\_FIELDS set—containing only fields that make sense to sort on (e.g. code, name, start, perPerson). If the incoming sortBy param matches one of these fields, I apply it to the query (via .sort({ [sortBy]: 1 })); otherwise, I default back to "name". This prevents malicious or nonsensical sorts, and leverages the indexes I added on those fields.
3. Lean Queries
   * Planned: Return plain JavaScript objects instead of Mongoose documents for read‐only operations.
   * Implemented: All read operations (find(), findOne(), findOneAndUpdate(), findOneAndDelete()) now include .lean(), which eliminates the overhead of Mongoose’s Document methods (getters/setters), drastically reducing memory usage during serialization.
4. Cost Calculation Method
   * Planned: Encapsulate trip‐cost logic within the model so route handlers stay thin.
   * Implemented: I added

tripSchema.methods.calculateCost = function(groupSize) {

return this.perPerson \* groupSize;

};

Then I refactored the GET /api/trips/:tripCode/cost/:groupSize route to fetch the lean document, call calculateCost(), and return the result. By moving business logic into the model, I follow the principle of “store logic next to the data” (object‐oriented design) and make future extensions (like discounts or currency adjustments) trivial.

1. Input Sanitization & Validation
   * Planned: Ensure that tripCode, groupSize, sortBy, page, limit are all sanitized and validated to avoid edge cases and injections.
   * Implemented:
     + sanitizeTripPayload()—field‐by‐field sanitization of POST/PUT payloads (strings are trimmed/escaped, numbers parsed, dates validated, descriptions sanitized for a limited set of HTML tags).
     + sanitizeString() ensures tripCode/sortBy are pure, non‐malicious strings.
     + sanitizeQueryParam() guarantees page/limit are integers ≥ 1.
     + In the cost route, I do parseInt(groupSize, 10) plus isNaN() checks so group sizes below 1 or invalid show 400.

These changes transformed broad, unoptimized data‐fetch functions into precise, index‐driven operations that handle large datasets gracefully—exactly the type of algorithmic thinking (choosing correct data structures + access patterns) I set out to demonstrate.

4. Alignment with Course Outcomes

* Design & Evaluate Computing Solutions (Outcome 3)  
  Creating paginated, sorted queries forced me to weigh trade‐offs (bandwidth vs. responsiveness, page size vs. UI performance) and implement database‐level optimizations.
* Use Well‐Founded Techniques (Outcome 4)  
  Choosing .lean(), whitelist‐based sorting, and the appropriate MongoDB indexes reflects industry best practices. These techniques show I can integrate proper tools to meet performance goals.
* Develop a Security Mindset (Outcome 5)  
  Sanitizing user input—even for numeric parameters—prevents invalid values that might break logic, corrupt data, or open injection vectors. This anticipates adversarial misuse of query strings and payloads.

5. Reflection on the Enhancement Process

1. Underestimating Data Volume  
   Initially, I built Travlr assuming only a handful of trips would exist. Once the dataset grew, fetching all trips at once became untenable. Implementing server‐side pagination forced me to consider how the application would behave with hundreds or thousands of trip documents.
2. Indexes Are Critical but Must Be Thoughtful  
   It’s tempting to index every field, but unnecessary indexes bloat storage and slow down write operations. I had to analyze which fields appear in queries—code for lookups and name/start/perPerson for sorting—before selectively adding indexes.
3. Encapsulation Simplifies Testing & Reuse  
   By moving the cost calculation into a model method (calculateCost), I cleaned up route handlers and made it trivial to write unit tests for that logic. If future requirements change (for example, applying tiered discounts), I can modify one method instead of every route.
4. Choosing Between Client vs. Server Work  
   Although I could have left pagination to the Angular frontend—fetching all trips and slicing arrays—it would have consumed a lot of bandwidth and slowed the initial load. Deciding where computation belongs is a nuanced design choice; in this case, server‐side was clearly the better fit.
5. Testing Edge Cases Is Non‐Negotiable  
   Handling corner cases—page = 0, limit = -5, sortBy = "\_\_proto\_\_", groupSize = 0—forced me to write comprehensive tests for my sanitizers and pagination logic. This underscores that “algorithms” are more than academic—they directly impact security, correctness, and user experience.

6. Final Thoughts

The enhanced version of Travlr has evolved from a simple CRUD prototype into a robust, performance‐optimized application. It now supports thousands of trips without overloading the client, leverages indexes for rapid lookups, and enforces clear data‐structure choices (lean objects, whitelists, model methods). By focusing on server‐side pagination, careful indexing, encapsulated model methods, and defensive sanitization, I have demonstrated practical algorithmic thinking and data‐structure awareness.

This milestone not only meets all requirements of Milestone Three but also deepens my confidence in crafting scalable, secure backends. Travlr is now ready to handle real‐world workloads, showing my ability to translate theoretical principles into production‐ready code. As I prepare to publish my ePortfolio, I can confidently point to Travlr as a concrete example of my growth in algorithms and data structures—an artifact that reflects both where I started and how far I have come.