



THE 21ST EDITION OF THE INTERNATIONAL CONFERENCE EUROPEAN INTEGRATION REALITIES AND PERSPECTIVES

Hierarchical and Non-Relational Data Structures in MongoDB: A Case Study of a Reservation Management System

Costel-Petruț Teșcan¹, Cosmin-Petrică Burlacu², Dragoș-Cătălin Stratulat³, Gabriel-Andrei Nistor⁴, Gabriel-Bogdan Caramaliu⁵, Lucian Andar⁶, Romeo-Angelin Hoge⁷

Abstract: Objectives: This paper presents a reservation management system used to demonstrate hierarchical and non-relational data structures in MongoDB. Prior Work: The study builds on document-oriented data modeling, embedded documents, references, and full-stack web application development. Approach: A case study application was developed with a separated SvelteKit frontend and Express backend. Users select a resource and enter a start and end interval, while the backend validates the request against embedded availability windows stored in MongoDB. Results: The prototype successfully separates frontend and backend responsibilities, stores hierarchical resource availability data, and prevents invalid or overlapping reservations. Implications: The system can support academic demonstrations of NoSQL modeling and practical reservation workflows. Value: The main contribution is a clear implementation that combines user-selected intervals with hierarchical MongoDB documents.

Keywords: Reservation; SvelteKit; Express; TypeScript; MongoDB

JEL Classification: C88; O33; L86

1. Introduction

Modern software systems often need to store flexible, nested and changing data structures. Relational databases remain important, but many application scenarios are naturally represented using documents, arrays and embedded objects. Reservation systems are a useful example because they combine users,

¹ Faculty of Media, Design and Technology, Danubius International University, Romania, Address: Blvd. Galați, nr.3, Galați, Romania, Corresponding author: tescanconstelpetrut@gmail.com.

² Faculty of Media, Design and Technology, Danubius International University, Romania, Address: Blvd. Galați, nr.3, Galați, Romania, E-mail: burlacucosmin28@gmail.com.

³ Faculty of Media, Design and Technology, Danubius International University, Romania, Address: Blvd. Galați, nr.3, Galați, Romania, E-mail: catalin.stratulat@gmail.com.

⁴ Faculty of Media, Design and Technology, Danubius International University, Romania, Address: Blvd. Galați, nr.3, Galați, Romania, E-mail: nistor.gabriel.andrei@gmail.com.

⁵ Faculty of Media, Design and Technology, Danubius International University, Romania, Address: Blvd. Galați, nr.3, Galați, Romania, E-mail: bogdancaramaliu@yahoo.com.

⁶ Faculty of Media, Design and Technology, Danubius International University, Romania, Address: Blvd. Galați, nr.3, Galați, Romania, E-mail: andar73@libero.it.

⁷ Faculty of Media, Design and Technology, Danubius International University, Romania, Address: Blvd. Galați, nr.3, Galați, Romania, E-mail: h.romeoangelin@gmail.com.



resources, availability intervals and confirmed reservations. A non-relational approach can represent these elements without forcing every nested component into a separate table.

MongoDB is a document-oriented database that supports flexible data modeling through embedded documents and references. The official MongoDB documentation describes data modeling as a process in which relationships can be represented either by embedding related data or by storing references, depending on application access patterns (MongoDB, 2026a). Embedded data models are especially useful when related data is frequently accessed together (MongoDB, 2026b).

This paper presents a reservation management system built as a case study for hierarchical and non-relational data structures in MongoDB. The final implementation separates the frontend and backend into two distinct parts. The frontend is implemented with SvelteKit, while the backend is implemented with Express and MongoDB. The SvelteKit routing model is based on filesystem routes in the source directory (Svelte, 2026), while Express provides a minimal web framework for building HTTP APIs in Node.js (Express, 2026).

The main improvement compared with a hardcoded time slot demonstration is that the user can now enter a custom start and end interval in the frontend. The backend receives this interval as a slot object and validates it against availability windows embedded inside MongoDB resource documents. This approach preserves the hierarchical MongoDB model while making the reservation flow more realistic.

2. Related Work

NoSQL databases were introduced to address data storage needs that do not always fit rigid relational schemas. Sadalage and Fowler (2012) describe NoSQL systems as a group of database technologies that emphasize flexible models and scalability. Document databases are particularly suitable when data can be represented as hierarchical objects.

MongoDB supports a document model in which each record is represented as a document. Such documents can contain nested objects and arrays. In this project, the resource document contains an embedded array named `availabilityWindows`. This design follows the idea of embedding related data directly inside the parent document when the data has a strong ownership relationship (MongoDB, 2026b).

The project also uses references. A reservation document references a user through `userId` and a resource through `resourceId`. This follows the mixed modeling approach recommended for cases where some data is embedded and other data remains independent (MongoDB, 2026a).

For the web application layer, the project uses SvelteKit on the frontend and Express on the backend. SvelteKit provides a component-based frontend application, while Express provides a separate HTTP server. This separation makes the architecture clearer because the user interface, server logic and database access are located in different folders and run on different ports.

3. Problem Statement

The initial reservation prototype stored frontend and backend logic inside one full-stack SvelteKit application. Although such an architecture is technically valid, it does not make the separation between frontend and backend immediately visible. For an academic case study, a clearer separation is useful

because the reader can identify where the user interface ends and where the server-side MongoDB logic begins.

Another issue is the use of predefined time slot identifiers. A user can select a fixed slot, but the system does not show the flow where the user provides start and end values from the interface. The improved version solves this by allowing the user to select a custom interval using datetime inputs. The backend then validates the selected interval.

The problem addressed by this paper is therefore twofold: first, to design a clear frontend/backend separation; second, to preserve MongoDB hierarchical modeling while supporting user-selected start and end reservation intervals.

4. Concepts and Terms

The main concepts used in the system are User, Resource, TimeSlot, Reservation and SystemState. A user is a person who can create a reservation. A resource is an item or service that can be reserved. Examples include conference rooms, laboratories, equipment, a bicycle, a PlayStation, a DVD item and a guided building tour.

A TimeSlot represents a start and end interval. In the final model, resource availability is represented by embedded availability windows. A reservation also contains a selected slot, which stores the interval chosen by the user.

SystemState is an in-memory representation of users, resources and reservations loaded from MongoDB. The backend creates this state, applies domain logic, and then saves the updated state back to the database.

5. Solution Approach

The solution is implemented as two separate applications inside the same GitHub repository. The frontend folder contains the SvelteKit application. The backend folder contains the Express server, MongoDB connection, route handlers and domain logic. This structure makes the role of each part explicit.

The frontend runs on `http://localhost:5173`. It displays users, resources, availability windows and existing reservations. The user selects a resource and enters a start and end interval. The frontend sends the following JSON payload to the backend:

```
{
  "userId": "user_1",
  "resourceId": "resource_1",
  "slot": { "start": "2026-05-20T09:30:00.000Z", "end": "2026-05-20T10:00:00.000Z" }
}
```

The backend runs on `http://localhost:3000`. It exposes endpoints for users, resources, reservations and seed data. When a reservation request is received, the backend loads the current state from MongoDB and calls the `createReservation` function.

Table 1. Main Project Layers and Responsibilities

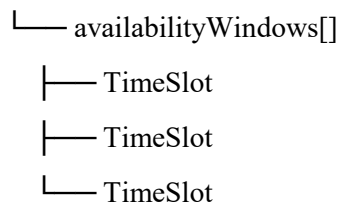
Layer	Folder	Responsibility
Frontend	frontend/	User interface, data display and HTTP requests
Backend	backend/	Validation, API routes and MongoDB access
Database	MongoDB	Users, resources, availability windows and reservations

6. MongoDB Data Model

The MongoDB database is named `reservation_system` and contains three main collections: users, resources and reservations. The most important hierarchical structure is the resources collection. Each resource contains an embedded `availabilityWindows` array.

A resource document has the following conceptual structure:

Resource



The `availabilityWindows` array is embedded because availability belongs directly to a resource. This design allows the backend to check whether a selected interval is inside one of the embedded windows. At the same time, reservations are stored separately because they represent independent events that reference users and resources.

The resources collection includes several sample resource types: rooms, a computer science laboratory, equipment, a bicycle, a Christmas tree, a PlayStation, a DVD item and a guided building tour. The guided building tour demonstrates that the same model can handle scheduled services, not only physical objects.

7. Implementation Using SvelteKit, Express and MongoDB

The frontend contains a SvelteKit page for reservations. The page loads data from the Express backend using HTTP requests. It does not use internal SvelteKit server routes for database access. This design makes the frontend responsibility limited to presentation and interaction.

The backend contains an Express server with routes for GET `/users`, GET `/resources`, GET `/reservations`, POST `/reservations` and POST `/seed`. The MongoDB connection is implemented using the official MongoDB Node.js driver, which supports JavaScript and TypeScript applications (MongoDB, 2026c).

The reservation endpoint receives `userId`, `resourceId` and `slot`. It validates the data by checking that the user and resource exist, that the date interval is valid, that the interval is inside an availability window, and that it does not overlap an existing reservation.

The backend domain logic is separated into TypeScript files: `types.ts`, `maybe.ts`, `validation.ts`, `state.ts` and `reservation.ts`. The `createReservation` function returns either a valid reservation or a list of validation errors. This makes the business logic independent from the Express route handler.

8. Analysis of Results

The final implementation satisfies the architectural requirement of separating frontend and backend. The repository now contains a frontend folder for SvelteKit and a backend folder for Express and MongoDB. This makes the system easier to understand and closer to a typical client-server structure.

The new reservation workflow also solves the problem of hardcoded time slots. The user can choose a start and end interval in the interface. The backend performs validation and decides if the interval is acceptable. This improves realism because the frontend no longer chooses only a fixed slot identifier.

The MongoDB hierarchical model remains visible because resource documents contain embedded availability windows. The selected reservation interval is not arbitrary from the database perspective: it must be inside one of these embedded windows. Therefore, the project combines flexible user input with hierarchical document modeling.

The prototype also prevents overlapping reservations. For example, if a reservation already exists from 12:30 to 13:00, a new request from 12:45 to 13:15 for the same resource is rejected. This demonstrates that application-level validation is necessary when using a document database without relational foreign-key constraints.

9. Discussion

The separated architecture has several advantages. The frontend can be developed and tested independently from the backend. The backend can expose a clear API that could be consumed by another frontend in the future. MongoDB remains responsible for persistent storage, while TypeScript domain logic remains responsible for validation.

The use of availabilityWindows also makes the MongoDB model more flexible than a fixed slot-id approach. A resource may be available for a larger interval, and the user can select a smaller interval inside it. This design is useful for booking systems where resources are available during certain periods but users may reserve only part of that time.

The model also has limitations. The application currently reloads and saves state in a simple way, which is acceptable for a prototype but should be improved for production. In a high-concurrency environment, MongoDB transactions or atomic update strategies would be needed to avoid conflicts between simultaneous reservation requests.

10. Conclusions

This paper presented a reservation management system as a case study for hierarchical and non-relational data structures in MongoDB. The final implementation uses a separated SvelteKit frontend and Express backend. This separation clearly shows the difference between user interface logic and server-side database logic.

The project demonstrates how MongoDB can represent hierarchical data through embedded availability windows inside resource documents. At the same time, the reservations collection uses references to users and resources. This combination of embedded documents and references provides a practical document-oriented model.

The improved version allows users to select custom start and end intervals in the frontend. The backend validates these intervals against MongoDB availability windows and existing reservations. As a result, the system is both more realistic and still suitable for demonstrating hierarchical MongoDB modeling.

11. Future Work

Future work may include a more advanced calendar interface, authentication, role-based permissions, administrative resource management, MongoDB transactions for concurrent reservation handling, and visual reports about resource usage. Another improvement would be to add tests for the domain logic and API endpoints.

References

- Express. (2026). *Express - Node.js web application framework*. from <https://expressjs.com/>
- MongoDB. (2026a). *Data modeling*. MongoDB Documentation. <https://www.mongodb.com/docs/manual/data-modeling/>
- MongoDB. (2026b). *Embedded data models*. MongoDB Documentation. <https://www.mongodb.com/docs/manual/data-modeling/embedding/>
- MongoDB. (2026c). *MongoDB Node.js driver*. MongoDB Documentation. <https://www.mongodb.com/docs/drivers/node/current/>
- Sadalage, P. J., & Fowler, M. (2012). *NoSQL distilled: A brief guide to the emerging world of polyglot persistence*. Addison-Wesley.
- Svelte. (2026). *Routing*. SvelteKit Documentation. <https://svelte.dev/docs/kit/routing>