# A Comprehensive Guide to Building a MongoDB ODM with GitHub Copilot

## 1. Introduction to MongoDB ODMs and Project Goals

MongoDB, a prominent NoSQL database, distinguishes itself through its document-based structure, offering a departure from the rigid schema constraints of traditional relational databases. This design allows developers to work with data in a format that closely mirrors how it is represented in their applications, utilizing JSON-like documents for storage and retrieval. The inherent flexibility of MongoDB's schema is particularly advantageous for modern applications that often require evolving data structures to accommodate changing requirements. However, this very flexibility can introduce challenges related to data consistency and structure if not managed effectively at the application level. Therefore, a structured approach to interacting with MongoDB becomes essential.

An Object-Document Mapper (ODM) serves as an intermediary layer, abstracting the complexities of direct database interactions and providing developers with a more intuitive way to manage data. While the term Object-Relational Mapper (ORM) is traditionally associated with relational databases, in the context of MongoDB, the terms ODM and ORM are often used interchangeably due to their similar objective: to facilitate seamless interaction between the application's objects and the database. By mapping application-level objects to MongoDB documents, an ODM simplifies data manipulation, allowing developers to focus on the application's logic rather than the intricacies of database queries.

This project aims to develop a custom MongoDB ODM tailored for Node.js environments, with a specific focus on supporting both TypeScript and JavaScript. A key requirement is adherence to the Airbnb style guide for code formatting, ensuring consistency and readability across the codebase. Furthermore, the goal is to create an npm package, similar in functionality to the popular TypeORM, which provides a robust and developer-friendly interface for working with MongoDB. This endeavor will involve providing clear, step-by-step instructions for each component of the ODM, enabling GitHub Copilot to effectively understand the requirements and generate the desired code. The explicit request for Airbnb style and TypeORM-like functionality underscores the need for a solution that is both familiar and adheres to established coding standards.

## 2. Understanding Core Concepts

### MongoDB's Document Data Model

At the heart of MongoDB lies its document data model, where data is organized into collections of documents. These documents are structured in a manner akin to JSON, consisting of key-value pairs that can represent a wide variety of data types, including nested objects and arrays. One of the significant benefits of this model is the flexible schema, which allows documents within the same collection to have different sets of fields or variations in data types. This adaptability is particularly useful in scenarios where data requirements evolve over time, as it avoids the need for costly and disruptive schema migrations often associated with relational

databases. MongoDB stores these JSON-like documents in a binary format called BSON (Binary JSON), which extends the JSON representation to include additional data types, enhancing efficiency in data processing and storage.
A key advantage of the document model is the ability to embed related data within a single document. For instance, a customer's address or a blog post's comments can be stored directly within the customer or post document, respectively. This approach minimizes the need for joining data across multiple collections, which can improve read performance and simplify data retrieval. Conversely, MongoDB also supports referencing data that resides in separate collections. This is achieved by storing identifiers, typically ObjectId values, in one document that link to related documents in another collection. This method is particularly useful for managing complex relationships, such as one-to-many or many-to-many, and for avoiding data duplication when the same data is referenced by multiple documents. The choice between embedding and referencing is a fundamental aspect of MongoDB schema design and depends heavily on the specific data access patterns and relationships within the application. The document model's inherent structure allows for a closer alignment with how developers naturally conceptualize data in object-oriented applications, potentially reducing the impedance mismatch often encountered with relational databases and the need for complex join operations.

## Object-Document Mapping (ODM)

An Object-Document Mapper (ODM) acts as a bridge between the object-oriented paradigm used in application code and the document-based structure of MongoDB. It facilitates the mapping of application objects to MongoDB documents, allowing developers to interact with their data using familiar object-oriented concepts rather than having to write raw database queries. The benefits of employing an ODM are manifold. Firstly, it abstracts away the underlying query language of MongoDB, providing a more declarative and intuitive API for data access. This abstraction can significantly enhance developer productivity and reduce the learning curve associated with mastering MongoDB's specific query syntax.
Secondly, an ODM manages the serialization and deserialization of data between the application's objects and the database's documents. This process involves converting data from the format used by the application (e.g., JavaScript or TypeScript objects) into the format required by MongoDB (BSON) when writing data, and vice versa when reading data. This automated handling of data transformation can save developers a considerable amount of time and effort. Thirdly, an ODM often provides mechanisms for enforcing schema at the application level. While MongoDB's schema flexibility is a strength, defining and enforcing data structures within the application can help maintain data integrity and consistency. This is particularly important in larger applications where multiple developers are working on the same codebase. Finally, by providing a higher level of abstraction and promoting code reuse through well-defined patterns, an ODM can contribute to improved code maintainability and overall application robustness. While MongoDB drivers offer the fundamental tools for database interaction, ODMs provide a more sophisticated and developer-centric approach to data management.

## Architectural Patterns

When designing an ODM, considering established architectural patterns can lead to a more structured and maintainable solution. Two prominent patterns in the context of ORMs/ODMs are Active Record and Data Mapper.
The Active Record pattern proposes that an entity (in this case, representing a MongoDB

document) should encapsulate both its data and the behavior for persisting that data to the database. In essence, the entity object itself is responsible for performing CRUD operations on its corresponding database record. This pattern is often favored for simpler applications due to its ease of implementation and the tight coupling between the domain model and the persistence layer.
Conversely, the Data Mapper pattern advocates for a clear separation of concerns between the entity and the data persistence logic. In this pattern, separate mapper objects are responsible for transferring data between the entities and the database. The entities themselves are typically plain objects with no database-specific logic. This separation can be particularly beneficial for larger and more complex applications, as it promotes better testability, maintainability, and flexibility in terms of database changes. TypeORM, a popular ORM for Node.js, notably supports both the Active Record and Data Mapper patterns , offering developers the flexibility to choose the pattern that best suits their application's needs. This flexibility is a significant advantage, and the custom ODM being developed should consider offering similar options to its users.

# 3. Drawing Inspiration from Existing Solutions

## Overview of Popular Node.js MongoDB ODMs

The Node.js ecosystem boasts several well-established MongoDB ODMs, each with its own strengths and design philosophies. Examining these existing solutions can provide valuable insights for the development of a custom ODM.
**TypeORM** stands out as a versatile ORM that supports both TypeScript and JavaScript. Its key feature is its ability to work with various database systems, including relational databases and MongoDB. TypeORM offers support for both the Active Record and Data Mapper patterns , providing developers with a choice based on their preference and application requirements. While primarily known for its relational database capabilities, TypeORM also provides basic support for MongoDB, as documented in its official documentation.
**Mongoose** is another highly popular ODM in the Node.js landscape, specifically designed for MongoDB. It employs a schema-based approach, allowing developers to define the structure of their MongoDB documents, including data validation rules and middleware. Mongoose is tightly integrated with the Node.js environment and provides a rich set of features for interacting with MongoDB in a structured and efficient manner.
**Prisma** presents itself as a modern database toolkit and ORM with comprehensive support for TypeScript, JavaScript, and multiple databases, including MongoDB. It emphasizes type-safe database access through an auto-generated query builder, ensuring that database interactions are both efficient and less prone to runtime errors. Prisma follows a schema-first design approach, where the database schema is defined in a declarative language and then used to generate database clients and other artifacts.
Each of these ODMs offers distinct advantages and caters to different development preferences and project needs. TypeORM's flexibility in supporting multiple patterns and languages, along with its existing (though basic) MongoDB support, makes it a compelling model for the custom ODM being planned.

## Focus on TypeORM's MongoDB Implementation

Given the user's explicit mention of creating an ODM similar to TypeORM, it is beneficial to

delve deeper into how TypeORM handles MongoDB integration. While TypeORM's primary focus lies in relational databases, it does offer fundamental support for MongoDB. This support includes specific decorators and classes tailored for MongoDB's document structure.
For instance, when defining entities that map to MongoDB collections, TypeORM utilizes the @ObjectIdColumn decorator to designate the primary key, which corresponds to MongoDB's _id field. This is in contrast to the @PrimaryColumn or @PrimaryGeneratedColumn decorators used for relational database entities. Furthermore, TypeORM allows for the definition of embedded documents by treating them as separate classes and then using them as properties within the main entity, decorated with the @Column() decorator and specifying the type of the embedded document. This approach mirrors MongoDB's ability to nest documents within each other.
For performing database operations, TypeORM provides MongoEntityManager and MongoRepository classes, which extend the base EntityManager and Repository classes with MongoDB-specific functionalities. These classes offer methods for common CRUD operations, such as finding, creating, updating, and deleting documents. Basic entity definition and bootstrapping in TypeORM for MongoDB involve importing the necessary decorators and classes from the typeorm library and then configuring a DataSource with the mongodb type and connection details. TypeORM also supports querying subdocuments and arrays within MongoDB documents using a syntax that aligns with MongoDB's query operators. TypeORM's strategy for integrating with MongoDB involves extending its core ORM concepts with features specific to MongoDB's document-based nature, offering a potential blueprint for the custom ODM.

# 4. Designing the Architecture of the Custom ODM

The architecture of the custom MongoDB ODM should be carefully designed to ensure it is flexible, maintainable, and effectively abstracts the underlying MongoDB interactions. Several core components will be necessary to achieve these goals.

## Core Components

**Entity Definition:** The ODM needs a mechanism for developers to define their data models, which will correspond to MongoDB collections and the documents within them. This can be achieved using TypeScript classes and interfaces. Custom decorators, similar to those used by TypeORM, can provide a clean and declarative way to define entities. For instance, an @Entity() decorator could mark a class as a persistent entity, while @Column() could specify the fields and their data types. Given MongoDB's use of ObjectId as the default primary key, a custom @ObjectIdColumn() decorator would be appropriate for identifying the primary key field within each entity. Entities should ideally be simple Plain Old JavaScript Objects (POJOs) or Plain Old CLR Objects (POCOs) in JavaScript and TypeScript, allowing for maximum flexibility in how they are used throughout the application.
**Connection Management:** A crucial aspect of the ODM is its ability to establish and manage connections to a MongoDB database. This component should leverage the official Node.js driver for MongoDB. It should include functionality to initialize a connection using configuration details such as the MongoDB connection URI, the database name, and potentially other connection options that can be passed directly to the MongoDB driver. To ensure efficient resource utilization and performance, the connection management module should implement

connection pooling, where a pool of pre-established connections is maintained and reused for database operations, rather than creating a new connection for each request. Configuration options for the connection pool, such as the maximum pool size, should be exposed. Furthermore, the connection management component should include robust error handling for initial connection failures and subsequent disconnections.

**CRUD Operations:** The ODM must provide a straightforward and intuitive API for performing basic Create, Read, Update, and Delete (CRUD) operations on MongoDB collections. This can be implemented through classes or methods within a repository layer. The API should support both single document operations (e.g., creating one document, finding one document, updating one document, deleting one document) and bulk operations (e.g., creating multiple documents at once). For read, update, and delete operations, the ODM should provide a mechanism for specifying query criteria to target the desired documents. This could involve accepting query objects that mirror MongoDB's query syntax.

**Relationship Handling:** Given the importance of relationships in data modeling, the ODM should provide support for both embedding and referencing related documents. For embedding, this could be as simple as allowing developers to define properties in their entities whose type is another entity class, decorated with @Column(). For referencing, a property could be defined with the type ObjectId or a string, potentially accompanied by a custom decorator like @Reference() to explicitly indicate a reference to another entity and its corresponding collection. In this initial implementation, the ODM may not automatically handle the population of referenced documents (i.e., fetching the related documents in a single query), but this could be considered for future enhancements. The ODM should aim to support common relationship types such as one-to-one, one-to-many, and many-to-many.

**Querying Capabilities:** The ODM should offer basic querying capabilities to allow developers to retrieve data from MongoDB based on specific criteria. This could involve implementing a find method within the repository that accepts a query object as an argument, where the query object follows a syntax similar to MongoDB's query language. This method would then use the find method of the MongoDB driver's collection object, passing the provided query. The initial querying capabilities should support common query operators such as equality, comparison operators (e.g., $gt, $lt), and logical operators (e.g., $and, $or). While aiming for TypeORM-like functionality, starting with a foundational set of querying features will allow for a more manageable initial development scope.

## TypeScript and JavaScript Interoperability

To cater to a broad range of developers, the custom ODM should seamlessly support both TypeScript and JavaScript. The most effective approach to achieve this is to develop the ODM primarily in TypeScript, leveraging its strong typing capabilities and features. Once the TypeScript code is written, the TypeScript compiler can be used to generate standard JavaScript files (.js) that can be readily used in JavaScript projects. Crucially, the compiler can also generate TypeScript declaration files (.d.ts), which provide type information for the generated JavaScript code. These declaration files are essential for TypeScript users, as they enable type checking and IntelliSense within their development environment when using the ODM. It is also important to ensure that the generated JavaScript code adheres to the Airbnb style guide, maintaining consistency across both language environments. Employing a build process that includes TypeScript compilation and the generation of declaration files is vital for creating a dual-package that serves both TypeScript and JavaScript users effectively.

### Adherence to Airbnb Style

A specific requirement for this project is that all generated code, whether TypeScript or the resulting JavaScript, must strictly adhere to the Airbnb style guide. This style guide is widely adopted in the JavaScript community and promotes code consistency, readability, and maintainability. To enforce this, the project setup should include ESLint, a popular JavaScript linter, configured with the eslint-config-airbnb-typescript plugin. This plugin extends the base Airbnb style guide with rules specific to TypeScript. Additionally, Prettier, an opinionated code formatter, should be integrated to automatically format the code according to the Airbnb conventions. This includes rules such as using single quotes for strings, maintaining consistent indentation, and potentially omitting unnecessary semicolons based on Prettier's configuration. Clear instructions on how to set up and configure these tools within the project will be essential to ensure that GitHub Copilot generates code that aligns with the desired style.

# 5. Detailed Instructions for GitHub Copilot - Setting Up the Project

To begin building the custom MongoDB ODM, the project needs to be properly initialized and configured. The following steps provide detailed instructions for GitHub Copilot to set up the foundational elements of the project.

### Step 1: Initializing the npm Package

Instruction: "Create a package.json file with the following basic information: name (e.g., my-mongo-odm), version (e.g., 0.1.0), description, main entry point (dist/index.js), types entry point (dist/index.d.ts), and author."
Instruction: "Add necessary dependencies to package.json such as the official mongodb driver and development dependencies like typescript, eslint, prettier, and eslint-config-airbnb-typescript."
A well-structured package.json file is fundamental for any npm package. It serves as the manifest for the project, containing essential metadata such as the package's name, version, and description, which are crucial for identification and discoverability within the npm registry. The main field specifies the entry point of the package when it is imported into another project, while the types field points to the main TypeScript declaration file, which is essential for providing type information to TypeScript users. Including an author and license helps in attributing the work and defining the terms under which it can be used. Furthermore, the package.json file manages the project's dependencies, listing both the runtime dependencies (like the official mongodb driver, which enables interaction with MongoDB databases) and the development dependencies (tools used during development, such as typescript for language features, eslint and eslint-config-airbnb-typescript for code linting according to Airbnb style, and prettier for automatic code formatting).

### Step 2: Configuring TypeScript for Dual Output

Instruction: "Create a tsconfig.json file with the following compiler options: target (e.g., ES2020), module (e.g., CommonJS), outDir (dist), declaration (true), emitDeclarationOnly (false), sourceMap (true), esModuleInterop (true), forceConsistentCasingInFileNames (true), strict

(true), skipLibCheck (true), and include and exclude to specify source files."
Instruction: "Explain that declaration: true generates .d.ts files, and outDir specifies the output directory for both JavaScript and declaration files."
Configuring TypeScript correctly is essential for generating both JavaScript and TypeScript declaration files, ensuring compatibility with both language ecosystems. The tsconfig.json file controls how the TypeScript compiler behaves. The target option specifies the ECMAScript version for the output JavaScript code, while module defines the module system to be used (e.g., CommonJS for Node.js environments). The outDir option specifies the directory where the compiled JavaScript files and the generated declaration files will be placed, typically dist. Setting declaration to true instructs the compiler to generate .d.ts files, which provide type information for TypeScript users consuming the package. emitDeclarationOnly: false ensures that both JavaScript and declaration files are output. sourceMap: true generates source map files, which are useful for debugging. esModuleInterop: true and forceConsistentCasingInFileNames: true help in maintaining consistency and avoiding common import-related issues. The strict: true option enables all strict type-checking options, promoting better code quality. skipLibCheck: true can be used to speed up compilation by skipping type checking of declaration files. Finally, include and exclude arrays specify which files should be included in the compilation process and which should be ignored.

## Step 3: Integrating ESLint and Prettier

Instruction: "Install ESLint and Prettier as development dependencies using npm."
Instruction: "Create an ESLint configuration file (.eslintrc.js or .eslintrc.json) and extend the eslint-config-airbnb-typescript configuration. Configure parserOptions.project to point to tsconfig.json."
Instruction: "Create a Prettier configuration file (.prettierrc.js or .prettierrc.json) with Airbnb-compatible formatting rules (e.g., single quotes, trailing commas)."
Instruction: "Add npm scripts to package.json for linting (eslint. --ext.ts,.js), formatting (prettier --write.), and running the TypeScript compiler (tsc)."
Integrating linters and formatters early in the project is crucial for ensuring code quality and consistency, particularly for adhering to the Airbnb style guide. ESLint is a powerful tool for identifying and reporting on patterns found in ECMAScript/JavaScript code, with the goal of making code more consistent and avoiding bugs. By installing ESLint as a development dependency, along with the eslint-config-airbnb-typescript plugin, the project can leverage a pre-configured set of rules that enforce the Airbnb style guide for both TypeScript and JavaScript code. The ESLint configuration file (.eslintrc.js or .eslintrc.json) should extend this configuration and specify the TypeScript parser (@typescript-eslint/parser). The parserOptions.project setting within the ESLint configuration should point to the tsconfig.json file, allowing ESLint to understand the project's type information. Prettier is an opinionated code formatter that automatically reformats code to ensure a consistent style. By installing Prettier as a development dependency and creating a configuration file (.prettierrc.js or .prettierrc.json), the project can define formatting rules that are compatible with the Airbnb style, such as using single quotes and trailing commas. To streamline the development process, npm scripts should be added to the scripts section of the package.json file. These scripts will allow developers to easily run ESLint for linting, Prettier for formatting, and the TypeScript compiler (tsc) for building the project using simple commands in the terminal.

# 6. Detailed Instructions for GitHub Copilot - Implementing Core ODM Functionality

With the project setup complete, the next step is to implement the core functionality of the custom MongoDB ODM. The following instructions guide GitHub Copilot in creating the essential components.

## Step 4: Defining Entities/Models

Instruction: "Create a src/entities directory."
Instruction: "For each entity, create a TypeScript class decorated with a custom @Entity() decorator. Define properties of the entity using @Column() decorator and mark the primary key with a custom @ObjectIdColumn() decorator."
Instruction: "Provide examples of entity definitions for common scenarios, such as a User entity with _id, firstName, and lastName properties."
Instruction: "Explain how to define embedded documents as separate classes and use them as properties in the main entity with the @Column() decorator, specifying the type."
The src/entities directory will house the definitions for the data models that the ODM will manage. For each MongoDB collection that the application needs to interact with, a corresponding TypeScript class should be created within this directory. To provide a clean and declarative way to define these entities, custom decorators can be used, similar to TypeORM. An @Entity() decorator can be used to mark a class as representing a MongoDB document. Within each entity class, properties can be defined using the @Column() decorator, which will map these properties to fields in the MongoDB document. For the primary key field, which in MongoDB is typically named _id and has a type of ObjectId, a custom @ObjectIdColumn() decorator should be used to specifically identify this field.
For common scenarios, an example of a User entity can be provided:

```
// src/entities/user.ts
import { ObjectId } from 'mongodb';
import { Entity, Column, ObjectIdColumn } from '../decorators'; //
Assuming custom decorators

@Entity('users') // Specify the collection name
export class User {
  @ObjectIdColumn()
  _id: ObjectId;

  @Column()
  firstName: string;

  @Column()
  lastName: string;
}
```

To handle embedded documents, separate TypeScript classes can be created to represent the structure of the embedded data. These classes can then be used as types for properties within

the main entity class, decorated with @Column(). For example, if a User entity needs to embed an Address document:

```
// src/entities/address.ts
import { Column } from '../decorators';

export class Address {
  @Column()
  street: string;

  @Column()
  city: string;

  @Column()
  zipCode: string;
}

// src/entities/user.ts
import { ObjectId } from 'mongodb';
import { Entity, Column, ObjectIdColumn } from '../decorators';
import { Address } from './address';

@Entity('users')
export class User {
  @ObjectIdColumn()
  _id: ObjectId;

  @Column()
  firstName: string;

  @Column()
  lastName: string;

  @Column((type) => Address) // Specify the type for the embedded
document
  address: Address;
}
```

This decorator-based approach provides a readable and maintainable way to define the structure of MongoDB documents within the application code.

## Step 5: Implementing Connection Management

Instruction: "Create a src/connection directory and a connection.ts file."
Instruction: "Use the MongoClient class from the mongodb driver to establish a connection to the database."
Instruction: "Implement a function to initialize the connection using connection URI, database name, and options (e.g., from environment variables)."
Instruction: "Implement connection pooling by reusing the MongoClient instance across the

application."
Instruction: "Provide instructions on how to handle initial connection errors and disconnections."
The src/connection directory will contain the logic for managing the connection to the MongoDB database. A connection.ts file within this directory should utilize the MongoClient class from the official mongodb driver to establish a connection. A dedicated function should be implemented to handle the initialization of the connection. This function will take parameters such as the MongoDB connection URI, the database name, and potentially an options object to configure the connection (e.g., connection timeout, pool size). These configuration parameters can be sourced from environment variables for better flexibility and security.
Connection pooling is crucial for the performance and scalability of database-driven applications. The ODM should implement connection pooling by ensuring that a single instance of MongoClient is created and reused across the application. The MongoClient instance manages a pool of connections to the MongoDB server, reducing the overhead of establishing new connections for each database operation.
The connection management module should also include mechanisms for handling potential errors during the initial connection attempt and for managing disconnections that may occur after the connection is established. This can involve using promises with .catch() blocks or try...catch statements to handle connection errors gracefully. Event listeners can be attached to the connection object to monitor for events like 'error', 'connected', and 'disconnected', allowing the application to respond appropriately to changes in the connection state.

## Step 6: Implementing CRUD Operations

Instruction: "Create a src/repository directory and a base repository class."
Instruction: "Implement methods for create, find, findOne, update, and delete operations using the Db object from the MongoDB driver."
Instruction: "Explain how to use insertOne and insertMany for create operations."
Instruction: "Explain how to use find and findOne with query filters for read operations."
Instruction: "Explain how to use updateOne and updateMany with update operators ($set, etc.) for update operations."
Instruction: "Explain how to use deleteOne and deleteMany with delete filters for delete operations."
Instruction: "Provide examples of using these methods in TypeScript and JavaScript."
The src/repository directory will contain the classes responsible for interacting with the MongoDB database. A base repository class can be created to encapsulate the common CRUD operations. This base class will need access to the Db object obtained from the MongoDB connection.
For create operations, the repository should implement methods like create (for inserting a single document) and createMany (for inserting multiple documents). These methods will utilize the insertOne and insertMany methods, respectively, provided by the MongoDB driver's collection object.
For read operations, methods like find (for retrieving multiple documents) and findOne (for retrieving a single document) should be implemented. These methods will use the find and findOne methods of the collection object, allowing for the specification of query filters to retrieve documents based on certain criteria.
For update operations, methods like update (for updating a single document) and updateMany (for updating multiple documents) should be implemented. These methods will use the updateOne and updateMany methods of the collection object, requiring both a filter to identify

the documents to update and an update document that specifies the changes to be made. Update operators like $set can be used to modify specific fields within the documents.
For delete operations, methods like delete (for deleting a single document) and deleteMany (for deleting multiple documents) should be implemented. These methods will use the deleteOne and deleteMany methods of the collection object, requiring a filter to identify the documents to remove.
Examples of how to use these methods in both TypeScript and JavaScript should be provided to illustrate the ODM's API. For instance, in TypeScript:

```
// src/repository/base.repository.ts
import { Db, Collection, ObjectId } from 'mongodb';
import { Entity } from '../entities/base.entity'; // Assuming a base
entity class

export class BaseRepository<T extends Entity> {
  protected readonly collection: Collection<T>;

  constructor(db: Db, collectionName: string) {
    this.collection = db.collection<T>(collectionName);
  }

  async create(entity: Omit<T, '_id'>): Promise<ObjectId> {
    const result = await this.collection.insertOne(entity as T);
    return result.insertedId;
  }

  async findOne(query: Partial<T>): Promise<T | null> {
    return this.collection.findOne(query);
  }

  // Implement other CRUD methods similarly
}

// src/repository/user.repository.ts
import { Db } from 'mongodb';
import { User } from '../entities/user';
import { BaseRepository } from './base.repository';

export class UserRepository extends BaseRepository<User> {
  constructor(db: Db) {
    super(db, 'users');
  }

  // Add user-specific repository methods if needed
}
```

## Step 7: Handling Relationships (Embedding and Referencing)

Instruction: "In the entity definition, for embedding, simply define a property with the type of the embedded document class and use the @Column() decorator."
Instruction: "For referencing, define a property with the type ObjectId or a string, and potentially use a custom decorator (e.g., @Reference()) to indicate a reference to another entity and its collection name."
Instruction: "Explain that the ODM will not automatically handle population of referenced documents in this basic implementation, but this could be a future enhancement."
Instruction: "Provide examples of entities with embedded and referenced relationships."
Handling relationships is a key aspect of data modeling in MongoDB. For embedding related documents, the approach within the ODM should be straightforward. In the entity definition, if a document needs to embed another document, the developer can simply define a property with the type of the embedded document's class and decorate it with the @Column() decorator. For example, as shown in the entity definition section, embedding an Address within a User entity involves defining an address property of type Address and using @Column((type) => Address).
For referencing documents in other collections, the ODM should allow developers to define a property in their entity that will store the ObjectId of the referenced document. This property can be of type ObjectId (imported from the mongodb driver) or a string. To provide more clarity and potentially enable future features like automatic population, a custom decorator, such as @Reference(), could be used to annotate these reference properties. This decorator could take the name of the target entity or collection as an argument. For example:

```
// src/entities/order.ts
import { ObjectId } from 'mongodb';
import { Entity, Column, ObjectIdColumn } from '../decorators';
// Assuming a custom Reference decorator
// import { Reference } from '../decorators';

@Entity('orders')
export class Order {
  @ObjectIdColumn()
  _id: ObjectId;

  @Column()
  orderDate: Date;

  // @Reference('users') // Custom decorator for referencing
  @Column()
  userId: ObjectId; // Storing the ObjectId of the referenced user
}
```

It is important to note that in this basic implementation of the ODM, the automatic population of referenced documents (i.e., automatically fetching the User document when querying an Order) will not be included. This functionality, often found in more advanced ODMs, involves performing additional database queries to retrieve the related data. However, this could be considered as a potential enhancement for future versions of the custom ODM.

## Step 8: Implementing Basic Querying Capabilities

Instruction: "In the repository class, implement a find method that accepts a query object (similar

to MongoDB's query syntax) as an argument."
Instruction: "Use the find method of the MongoDB driver's collection object, passing the query object."
Instruction: "Provide examples of basic queries using equality, comparison operators, and logical operators."
To enable basic querying capabilities, the base repository class (and subsequently, specific entity repositories) should implement a find method. This method will accept a query object as its argument. This query object should be structured in a way that mirrors the query syntax used by MongoDB itself. The find method will then internally use the find method provided by the MongoDB driver's collection object, directly passing the received query object to it. This allows developers familiar with MongoDB's querying to leverage that knowledge when using the ODM. Here's an example of how the find method might be implemented in the BaseRepository:

```
// src/repository/base.repository.ts
import { Db, Collection, ObjectId } from 'mongodb';
import { Entity } from '../entities/base.entity';

export class BaseRepository<T extends Entity> {
  //... (previous methods)

  async find(query: Partial<T>): Promise<T> {
    return this.collection.find(query).toArray();
  }
}
```

Developers can then use this find method in their entity-specific repositories, passing query objects to retrieve data. For instance, in the UserRepository:

```
// src/repository/user.repository.ts
import { Db } from 'mongodb';
import { User } from '../entities/user';
import { BaseRepository } from './base.repository';

export class UserRepository extends BaseRepository<User> {
  constructor(db: Db) {
    super(db, 'users');
  }

  async findByFirstName(firstName: string): Promise<User> {
    return this.find({ firstName }); // Equality query
  }

  async findByAgeGreaterThan(age: number): Promise<User> {
    return this.find({ age: { $gt: age } }); // Comparison operator
($gt - greater than)
  }

  async findActiveUsers(isActive: boolean, role: string):
Promise<User> {
    return this.find({ $and: [{ isActive }, { role }] }); // Logical
```

```
operator ($and)
  }
}
```

This basic querying capability allows for a flexible way to retrieve data based on various conditions, directly leveraging MongoDB's powerful query language.

# 7. Detailed Instructions for GitHub Copilot - Ensuring Language Support and Code Quality

To ensure that the custom MongoDB ODM effectively supports both TypeScript and JavaScript users while maintaining a high standard of code quality, the following steps are crucial.

## Step 9: Generating TypeScript Declaration Files

Instruction: "Ensure that the tsconfig.json file has declaration: true."
Instruction: "Explain that when the TypeScript compiler runs (tsc), it will automatically generate .d.ts files in the dist directory alongside the JavaScript files."
Instruction: "Mention that these files allow TypeScript users to have type checking and IntelliSense when using the ODM."
As previously discussed, setting the declaration option to true in the tsconfig.json file is essential for generating TypeScript declaration files (.d.ts). When the TypeScript compiler (tsc) is executed to build the project, it will not only produce the JavaScript files in the specified outDir (e.g., dist) but also automatically generate corresponding .d.ts files for each TypeScript file in the project. These declaration files serve as type definitions for the JavaScript code. For TypeScript users who install and use the custom ODM in their projects, these .d.ts files will provide valuable type checking during development, allowing the TypeScript compiler to verify the correct usage of the ODM's API. Furthermore, most modern Integrated Development Environments (IDEs) like Visual Studio Code utilize these declaration files to provide IntelliSense features, such as code completion and type information on hover, significantly enhancing the developer experience.

## Step 10: Enforcing Airbnb Style

Instruction: "Remind GitHub Copilot that all generated TypeScript and JavaScript code must strictly follow the Airbnb style guide."
Instruction: "This includes using single quotes, consistent indentation, no unnecessary semicolons (if configured with Prettier), and adhering to other rules defined in the Airbnb style guide."
Instruction: "Emphasize the importance of running the linting and formatting scripts (npm run lint and npm run format) to ensure compliance."
Throughout the code generation process, it is crucial to consistently remind GitHub Copilot that all TypeScript and JavaScript code must strictly adhere to the Airbnb style guide. This includes various stylistic conventions such as the use of single quotes for string literals, maintaining consistent indentation (typically two spaces), and potentially omitting unnecessary semicolons if Prettier is configured to do so. Adherence to the Airbnb style guide ensures a uniform and readable codebase, making it easier for developers to understand and maintain the ODM. To

verify compliance with the style guide, the linting script (npm run lint) should be executed regularly. This script will run ESLint, which is configured with the eslint-config-airbnb-typescript plugin, and report any violations of the Airbnb style rules. Additionally, the formatting script (npm run format) should be used to automatically reformat the code according to the Prettier configuration, which should also be aligned with the Airbnb style. Regularly running these scripts will help maintain a consistent and high-quality codebase.

# 8. Detailed Instructions for GitHub Copilot - Unit Testing

To ensure the reliability and correctness of the custom MongoDB ODM, implementing a comprehensive suite of unit tests is essential. Unit tests verify that individual units of code (e.g., functions, methods, classes) behave as expected in isolation.

## Step 11: Setting Up Unit Tests

Instruction: "Choose a unit testing framework such as Jest or Mocha."
Instruction: "Install the chosen framework and any necessary type definitions (e.g., @types/jest or @types/mocha) as development dependencies."
Instruction: "Create a test directory at the root of the project."
Instruction: "Write unit tests for the core functionalities of the ODM, such as entity definition, connection management, and CRUD operations."
Instruction: "Provide examples of writing tests using the chosen framework's syntax (e.g., describe, it, expect in Jest; describe, it, assert in Mocha with an assertion library like Chai)."
Instruction: "Add an npm script to package.json to run the unit tests (e.g., jest or mocha)."
For setting up unit tests, a suitable JavaScript testing framework needs to be chosen. Two popular options are Jest and Mocha. Jest is known for its "zero-configuration" approach and includes built-in support for assertions, mocking, and coverage reports, making it a comprehensive testing solution. It is particularly popular for testing React applications but works well with Node.js and TypeScript projects. Mocha, on the other hand, is a more flexible and customizable framework that provides the basic testing structure but requires the use of additional libraries for assertions (e.g., Chai), mocking (e.g., Sinon), and coverage reporting. Mocha is often preferred for more complex testing scenarios and offers better control over the testing setup.
Once a framework is chosen, it needs to be installed as a development dependency using npm, along with any necessary type definitions (e.g., @types/jest if Jest is chosen). A test directory should be created at the root of the project to house the unit test files. Within this directory, test files should be written to cover the core functionalities of the ODM, including the correct definition of entities, the establishment and management of database connections, and the proper implementation of CRUD operations.
Examples of writing tests using the chosen framework's syntax should be provided. For instance, if Jest is selected, a test file might look like this:

```
// test/user.test.ts
import { User } from '../src/entities/user';

describe('User Entity', () => {
  it('should create a new User instance', () => {
```

```
    const user = new User();
    expect(user).toBeInstanceOf(User);
  });

  it('should have firstName and lastName properties', () => {
    const user = new User();
    user.firstName = 'John';
    user.lastName = 'Doe';
    expect(user.firstName).toBe('John');
    expect(user.lastName).toBe('Doe');
  });
});
```

Finally, an npm script should be added to the package.json file to easily run the unit tests. For Jest, this might be "test": "jest", and for Mocha, it could be "test": "mocha".
Instruction: "*Table Specification:* Include a table comparing Jest and Mocha to help the user decide which framework to use. The table should include features like built-in assertions, mocking, snapshot testing, ease of configuration, and use cases".

| Feature | Jest | Mocha |
|---|---|---|
| Built-in Assertions | Yes | No (requires assertion library like Chai) |
| Built-in Mocking | Yes | No (requires mocking library like Sinon) |
| Snapshot Testing | Yes | No (requires additional tools) |
| Code Coverage | Yes | No (requires additional tools) |
| Ease of Configuration | Zero configuration, works out of the box | Requires configuration with additional libraries |
| Performance | Can be slower for large test suites | Generally faster |
| Use Cases | Front-end (React), all-in-one solution | Back-end (Node.js), customizable setups |

This table provides a comparison of Jest and Mocha across several key features, helping developers choose the framework that best aligns with their project's requirements and their personal preferences.

# 9. Detailed Instructions for GitHub Copilot - Packaging and Distribution

Once the custom MongoDB ODM is implemented and thoroughly tested, the final step is to prepare it for distribution as an npm package.

## Step 12: Preparing for npm Publication

Instruction: "Ensure that the package.json file contains all necessary metadata, including name, version, description, main, types, author, license, and keywords."
Instruction: "Specify the files to be included in the npm package using the files array in

package.json (e.g., dist)."
Instruction: "Consider adding a .gitignore file to exclude development-related files (e.g., node_modules, test) from the published package."
Instruction: "Explain the process of building the project (npm run build), linting (npm run lint), formatting (npm run format), and testing (npm run test) before publishing."
Instruction: "Provide basic instructions on how to publish the package to npm using the npm publish command."
Before publishing the ODM to npm, it is essential to ensure that the package.json file contains all the necessary metadata. This includes a unique and descriptive name for the package, a semantic version number, a concise description of the package's functionality, the path to the main JavaScript entry point in the main field (typically dist/index.js), the path to the main TypeScript declaration file in the types field (typically dist/index.d.ts), information about the author, the license under which the package is distributed, and relevant keywords that will help users find the package on npm.
The files array in package.json should specify which files and directories should be included when the package is installed by users. Typically, this should include the compiled JavaScript code and the TypeScript declaration files located in the dist directory (e.g., "files": ["dist"]). A .gitignore file should be added to the project to exclude development-related files and directories, such as node_modules, the test directory, and any temporary build artifacts, from being included in the published package.
Before publishing, it is crucial to ensure that the project is built (npm run build), that the code adheres to the Airbnb style guide by running the linter (npm run lint), that the code is properly formatted (npm run format), and that all unit tests pass successfully (npm run test). This ensures that the published package is of high quality and functions as expected. Finally, to publish the package to the npm registry, the developer needs to have an npm account and be logged in via the terminal. The npm publish command, executed from the root directory of the project, will then publish the package for others to use.

# 10. Advanced Considerations and Best Practices

While the preceding instructions cover the fundamental aspects of building a custom MongoDB ODM, there are several advanced considerations and best practices that can further enhance its functionality, performance, and maintainability.

## MongoDB Schema Design

Effective MongoDB schema design is critical for the performance and scalability of applications using the ODM. Developers should carefully consider the application's use cases and data access patterns when designing their schemas. The decision of when to use embedding versus referencing related data is a key aspect of schema design. Embedding is generally favored for one-to-one and one-to-few relationships where the embedded data is frequently accessed with the parent document. Referencing is more suitable for one-to-many and many-to-many relationships, or when the referenced data needs to be accessed independently. Developers should also be mindful of MongoDB's 16MB document size limit and avoid embedding excessive amounts of data that could lead to performance issues or exceeding this limit.

## Connection Pooling and Management

Elaborating on connection pooling, it is essential for optimizing database interactions by reducing the overhead of establishing new connections. The ODM should allow configuration of the connection pool size, including the minimum and maximum number of connections to maintain. Timeouts for idle connections and connection establishment should also be configurable. Implementing strategies for handling connection errors, such as automatic retries with exponential backoff, can improve the resilience of applications using the ODM.

## Error Handling

Robust error handling is crucial for building reliable applications. The ODM should provide clear and informative error messages when database operations fail. Utilizing promises and try...catch blocks for asynchronous operations is a recommended approach for handling errors gracefully. Consider creating custom error classes that extend the built-in Error class to represent specific ODM-related error conditions, such as entity not found or connection failure. This allows for more specific error handling logic within the application code.

# 11. Conclusion

This comprehensive guide provides a detailed roadmap for leveraging GitHub Copilot to build a custom MongoDB Object-Document Mapper (ODM) for Node.js. By following the step-by-step instructions, developers can create an npm package that supports both TypeScript and JavaScript, adheres to the Airbnb style guide for code formatting, and offers core functionalities similar to TypeORM. This includes defining entities, managing database connections, performing CRUD operations, and handling basic relationships through embedding and referencing. The inclusion of unit testing setup and guidance on packaging and distribution ensures that the resulting ODM is reliable and easily shareable. While this guide covers the essential aspects, further exploration of advanced features like automatic population of referenced documents, more sophisticated querying capabilities, and integration with schema validation libraries can lead to an even more powerful and versatile custom ODM tailored to specific project needs.

### Nguồn trích dẫn

1. MongoDB Architecture Guide | KW Foundation, https://kwfoundation.org/wp-content/uploads/2023/08/MongoDB_Architecture_Guide.pdf 2. Data Modeling - Database Manual v8.0 - MongoDB Docs, https://www.mongodb.com/docs/manual/data-modeling/ 3. MongoDB Schema Design Best Practices and Techniques - GeeksforGeeks, https://www.geeksforgeeks.org/mongodb-schema-design-best-practices-and-techniques/ 4. MongoDB Architecture Guide, https://www.mongodb.com/resources/products/fundamentals/mongodb-architecture-guide 5. MongoDB ORMs, ODMs, and Libraries, https://www.mongodb.com/developer/products/mongodb/mongodb-orms-odms-libraries/ 6. ORMs, ODMs, and Libraries - Drivers - MongoDB Docs, https://www.mongodb.com/docs/drivers/odm/ 7. MongoDB Schema Design: Data Modeling Best

Practices,
https://www.mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/ 8. Embedded Data Versus References - Database Manual v8.0 - MongoDB Docs,
https://www.mongodb.com/docs/manual/data-modeling/concepts/embedding-vs-references/ 9. Embedding MongoDB Documents For Ease And Performance,
https://www.mongodb.com/resources/products/fundamentals/embedded-mongodb 10. Model One-to-Many Relationships with Embedded Documents - Database Manual v8.0,
https://www.mongodb.com/docs/manual/tutorial/model-embedded-one-to-many-relationships-between-documents/ 11. MongoDB relationships: embed or reference? - Stack Overflow,
https://stackoverflow.com/questions/5373198/mongodb-relationships-embed-or-reference 12. why the use of an ORM with NoSql (like MongoDB) [closed] - Stack Overflow,
https://stackoverflow.com/questions/8051614/why-the-use-of-an-orm-with-nosql-like-mongodb 13. Understanding Object-Relational Mapping - AltexSoft,
https://www.altexsoft.com/blog/object-relational-mapping/ 14. Active record pattern - Wikipedia,
https://en.wikipedia.org/wiki/Active_record_pattern 15. Pragmatic developer: Learn architecture and design patterns. Using ORM and ActiveRecord,
https://dev.to/vitalykrenel/pragmatic-development-learn-architecture-and-design-patterns-using-orm-and-activerecord-15oc 16. Active Record vs Data Mapper - typeorm - GitBook,
https://orkhan.gitbook.io/typeorm/docs/active-record-data-mapper 17. ORM Patterns: The Trade-Offs of Active Record and Data Mappers for Object-Relational Mapping - Thoughtful Code, https://www.thoughtfulcode.com/orm-active-record-vs-data-mapper/ 18. Patterns Implemented by SQLAlchemy - zzzeek,
https://techspot.zzzeek.org/2012/02/07/patterns-implemented-by-sqlalchemy/ 19. TypeORM v. MikroORM: Object Relational Mappers in Custom Software Development,
https://www.soliantconsulting.com/blog/typeorm-v-mikroorm/ 20. TypeORM - Amazing ORM for TypeScript and JavaScript (ES7, ES6, ES5). Supports MySQL, PostgreSQL, MariaDB, SQLite, MS SQL Server, Oracle, WebSQL databases. Works in NodeJS, Browser, Ionic, Cordova and Electron platforms., https://typeorm.io/ 21. MongoDB - typeorm - GitBook,
https://orkhan.gitbook.io/typeorm/docs/mongodb 22. typeorm/docs/mongodb.md at master - GitHub, https://github.com/typeorm/typeorm/blob/master/docs/mongodb.md 23. Top 11 Node.js ORMs, query builders & database libraries in 2022 - Prisma,
https://www.prisma.io/dataguide/database-tools/top-nodejs-orms-query-builders-and-database-libraries 24. Express Tutorial Part 3: Using a Database (with Mongoose) - Learn web development | MDN,
https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/Express_Nodejs/mongoose 25. Mongoose Schemas Creating a Model | GeeksforGeeks,
https://www.geeksforgeeks.org/mongoose-schemas-creating-a-model/ 26. Mongoose Tutorial - GeeksforGeeks, https://www.geeksforgeeks.org/mongoose-tutorial/ 27. Top 6 ORMs for Modern Node.js App Development - Amplication,
https://amplication.com/blog/top-6-orms-for-modern-nodejs-app-development 28. MongoDB database connector | Prisma Documentation,
https://www.prisma.io/docs/orm/overview/databases/mongodb 29. Start from scratch with Prisma ORM using MongoDB and TypeScript (15 min),
https://www.prisma.io/docs/getting-started/setup-prisma/start-from-scratch/mongodb-typescript-mongodb 30. Build a CRUD API With MongoDB, Typescript, Express, Prisma, and Zod,
https://www.mongodb.com/developer/products/atlas/mongodb-express-prisma-validation/ 31. Working with MongoDB in TypeORM - Tutorialspoint,
https://www.tutorialspoint.com/typeorm/typeorm_working_with_mongodb.htm 32. MongoDB with

TypeORM integration - Typeix, https://typeix.com/documentation/typeorm/mongodb/ 33. How to connect TypeORM with MongoDB? | Step-by-Step Approach - YouTube, https://www.youtube.com/watch?v=OtpPjB6sc5c 34. How do you structure your project by components if you are using mongoose as your orm? : r/node - Reddit, https://www.reddit.com/r/node/comments/9k7a8g/how_do_you_structure_your_project_by_components/ 35. Connecting to MongoDB - Mongoose, https://mongoosejs.com/docs/connections.html 36. How do I manage MongoDB connections in a Node.js web application? - Stack Overflow, https://stackoverflow.com/questions/10656574/how-do-i-manage-mongodb-connections-in-a-node-js-web-application 37. Multiple MongoDB Connections in a Single Application, https://www.mongodb.com/developer/products/atlas/multiple-mongodb-connections-in-a-single-application/ 38. TypeScript Integration With MongoDB Guide, https://www.mongodb.com/resources/products/compatibilities/using-typescript-with-mongodb-tutorial 39. Connect via Drivers - Atlas - MongoDB Docs, https://www.mongodb.com/docs/atlas/driver-connection/ 40. How to Use ORM Connection Pooling Effectively | GeeksforGeeks, https://www.geeksforgeeks.org/how-to-use-orm-connection-pooling-effectively/ 41. Managing Mongodb connections in Java as Object Oriented - Stack Overflow, https://stackoverflow.com/questions/60180119/managing-mongodb-connections-in-java-as-object-oriented 42. Connection Pool Overview - Database Manual v8.0 - MongoDB Docs, https://www.mongodb.com/docs/manual/administration/connection-pool-overview/ 43. MongoDB CRUD Operations, https://www.mongodb.com/resources/products/fundamentals/crud 44. How to Execute Create, Read, Update and Delete Operations in MongoDB, https://delbridge.solutions/mongodb-crud-operations/ 45. CRUD Operations - Node.js Driver - MongoDB, https://www.mongodb.com/docs/drivers/node/current/fundamentals/crud/ 46. MongoDB CRUD Operations | GeeksforGeeks, https://www.geeksforgeeks.org/mongodb-crud-operations/ 47. MongoDB CRUD Operations - Database Manual v8.0, https://www.mongodb.com/docs/manual/crud/ 48. How To Perform CRUD Operations in MongoDB - DigitalOcean, https://www.digitalocean.com/community/tutorials/how-to-perform-crud-operations-in-mongodb 49. MongoDB Relationships Embed or Reference | GeeksforGeeks, https://www.geeksforgeeks.org/mongodb-relationships-embed-or-reference/ 50. Embedded vs. Referenced Documents in MongoDB | GeeksforGeeks, https://www.geeksforgeeks.org/embedded-vs-referenced-documents-in-mongodb/ 51. Many to many relationship and linked table/collection - Working with Data - MongoDB, https://www.mongodb.com/community/forums/t/many-to-many-relationship-and-linked-table-collection/130305 52. How To Create An NPM Package | Total TypeScript, https://www.totaltypescript.com/how-to-create-an-npm-package 53. How to build dual package npm from Typescript - the easiest way - Duy NG, https://tduyng.com/blog/dual-package-typescript/ 54. Documentation - Publishing - TypeScript, https://www.typescriptlang.org/docs/handbook/declaration-files/publishing.html 55. How to build an NPM Package that's both Typescript and Javascript friendly? - Reddit, https://www.reddit.com/r/typescript/comments/k67pd0/how_to_build_an_npm_package_thats_both_typescript/ 56. Documentation - Creating .d.ts Files from .js files - TypeScript, https://www.typescriptlang.org/docs/handbook/declaration-files/dts-from-js.html 57. TSConfig Option: declaration - TypeScript, https://www.typescriptlang.org/tsconfig/declaration.html 58. How to Use ESLint With the Airbnb JavaScript Style Guide - MakeUseOf, https://www.makeuseof.com/eslint-with-airbnb-javascript-style-guide/ 59. Guide to Setting Up

Prettier, Airbnb ESLint, and Husky for Your Next Project, https://dev.to/emmanuelo/guide-to-setting-up-prettier-airbnb-eslint-and-husky-for-your-next-project-17ge 60. eslint-config-airbnb-typescript - NPM, https://www.npmjs.com/package/eslint-config-airbnb-typescript 61. How to Set Up Prettier in Your JavaScript/TypeScript Project? - GeeksforGeeks, https://www.geeksforgeeks.org/how-to-set-up-prettier-in-your-javascript-typescript-project/ 62. Configuration File - Prettier, https://prettier.io/docs/configuration 63. How To Format Code with Prettier in Visual Studio Code | DigitalOcean, https://www.digitalocean.com/community/tutorials/how-to-format-code-with-prettier-in-visual-studio-code 64. Creating our first Node.js package with Typescript - At Moonshot Partners, https://www.moonshot.partners/blog/creating-our-first-node-js-package-with-typescript 65. Install - Prettier, https://prettier.io/docs/install 66. Example how to use TypeORM with MongoDB using TypeScript. - GitHub, https://github.com/typeorm/mongo-typescript-example 67. Using TypeScript with Mongoose, https://mongoosejs.com/docs/typescript.html 68. Handling ORM-Free Data Access Layer in TypeScript With MongoDB | HackerNoon, https://hackernoon.com/handling-orm-free-data-access-layer-in-typescript-with-mongodb 69. eslint-config-airbnb - Yarn 1, https://classic.yarnpkg.com/en/package/eslint-config-airbnb 70. TypeScript Unit Testing 101: A Developer's Guide - Testim, https://www.testim.io/blog/typescript-unit-testing-101/ 71. Unit testing JavaScript and TypeScript - Visual Studio (Windows) | Microsoft Learn, https://learn.microsoft.com/en-us/visualstudio/javascript/unit-testing-javascript-with-visual-studio?view=vs-2022 72. Top 9 JavaScript Testing Frameworks | BrowserStack, https://www.browserstack.com/guide/top-javascript-testing-frameworks 73. TypeScript Unit Testing Frameworks - Capicua, https://www.capicua.com/blog/typescript-unit-testing-frameworks 74. JavaScript unit testing frameworks in 2024: A comparison · Raygun Blog, https://raygun.com/blog/javascript-unit-testing-frameworks/ 75. Jest vs Mocha: Comparing NodeJS Unit Testing Frameworks, https://www.browserstack.com/guide/jest-vs-mocha 76. Using Node.js's test runner, https://nodejs.org/en/learn/test-runner/using-test-runner 77. Guide to Testing Node.js Applications with Jest and TypeScript - YouTube, https://www.youtube.com/watch?v=uzMuyymMtvU 78. Jest · Delightful JavaScript Testing, https://jestjs.io/ 79. Jest vs Mocha: Which One Should You Choose? - GeeksforGeeks, https://www.geeksforgeeks.org/jest-vs-mocha-which-one-should-you-choose/ 80. www.browserstack.com, https://www.browserstack.com/guide/jest-vs-mocha#:~:text=Jest%20vs%20Mocha%3A%20Main%20Differences,-At%20first%20glance&text=The%20most%20basic%20difference%20is,additional%20libraries%20for%20these%20functionalities. 81. Jest vs Mocha: What's the Difference? - Pieces for developers, https://pieces.app/blog/whats-the-difference-between-jest-and-mocha 82. Jest vs. Mocha vs. Jasmine: Which Is Right for You? | Perfecto by Perforce, https://www.perfecto.io/blog/jest-vs-mocha-vs-jasmine 83. Jest vs. Mocha For Unit Testing Comprehensive Guide - Sauce Labs, https://saucelabs.com/resources/blog/jest-vs-mocha 84. 1. To Embed or Reference - MongoDB Applied Design Patterns [Book] - O'Reilly Media, https://www.oreilly.com/library/view/mongodb-applied-design/9781449340056/ch01.html 85. MongoDB schema design, can I embed and reference data at the same time?, https://stackoverflow.com/questions/65534874/mongodb-schema-design-can-i-embed-and-reference-data-at-the-same-time 86. Embed vs. Referencing question - Working with Data - MongoDB Developer Community Forums, https://www.mongodb.com/community/forums/t/embed-vs-referencing-question/118759