MASYNCTECH
SCHOOL

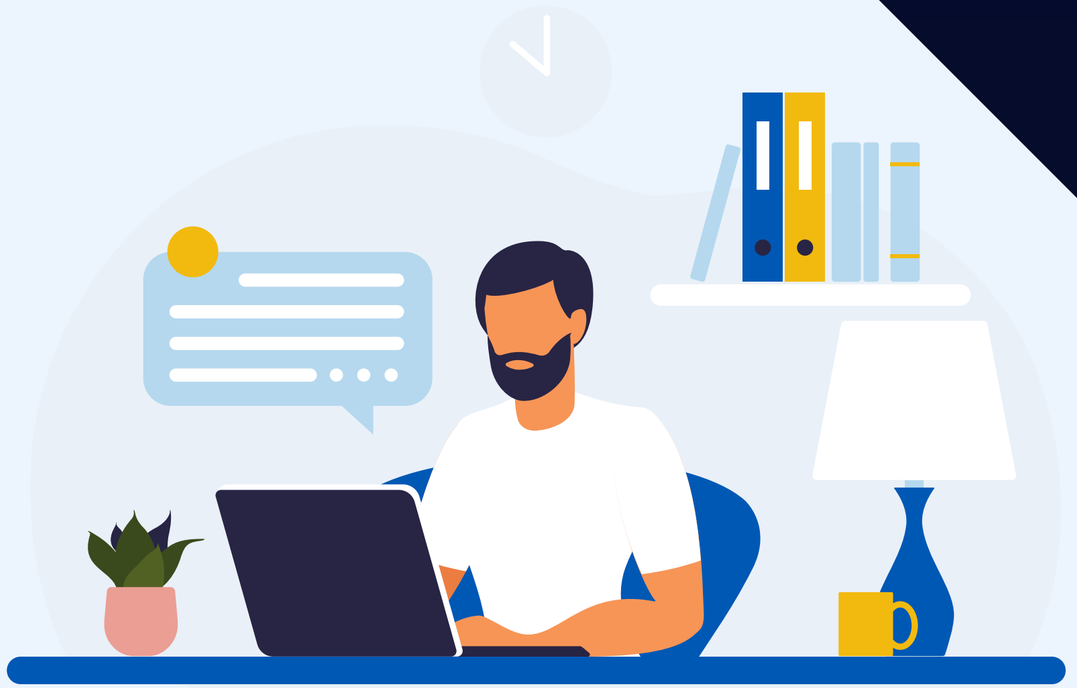
FULLSTACK WEB DEV

mongoose

BACKEND DEVELOPMENT

MASYNCTECH
SCHOOL

WHAT'S MONGOOSE



mongoose

CORE CONCEPTS

MONGOOSE OVERVIEW

MASYNCTECH SCHOOL

High-Level Explanation

- Mongoose: ODM library for MongoDB in Node.js
- Offers schema-based models, validation, queries, hooks, etc.
- Simplifies MongoDB interaction in Node.js apps.

Analogue

- Analogy: Mongoose as a toy box organizer
- Keeps MongoDB data neat and organized
- Ensures data matches your desired structure

Best Practices

- Steps for Mongoose usage:
  - Define schema for collections
  - Employ validators for data integrity
  - Utilize middleware for pre/post-save operations

Deep Dive

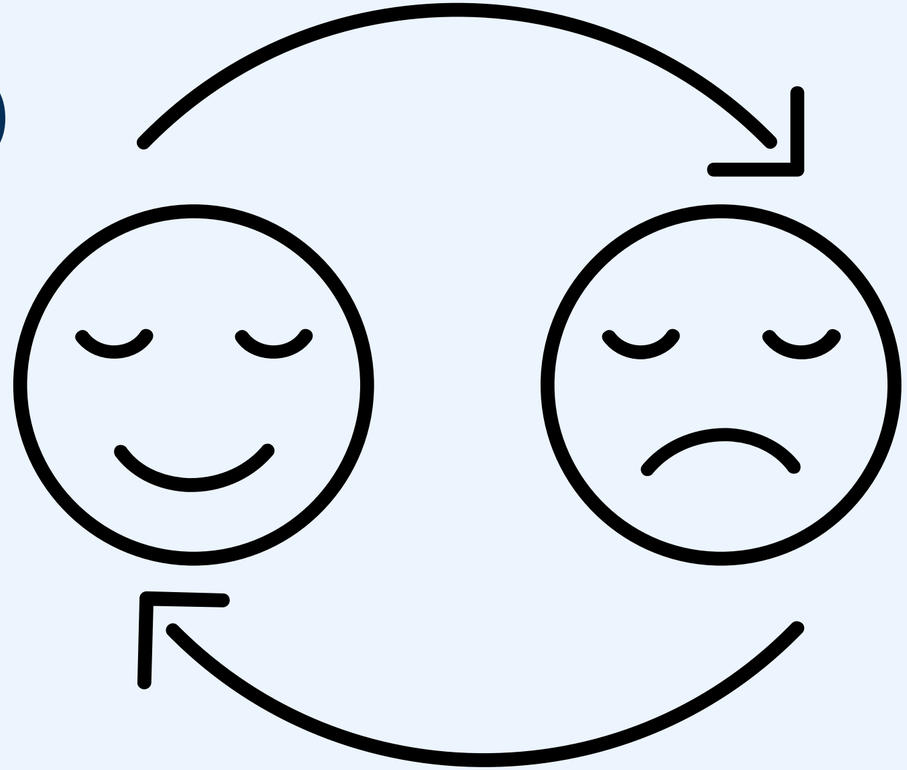
- Mongoose abstracts MongoDB for ease
- Handles validation, casting, queries, and hooks
- Defines document structure with Schemas
- Features: population, virtual properties, middleware, plugins
- Maintains MongoDB's flexibility with added structure

When to use?

- Mongoose benefits for:
  - Complex projects with structured DB needs
  - Ensuring data adheres to rules/schemas
  - Quick prototyping for speed and convenience

MASYNCTECH
SCHOOL

MONGODB VS MONGOOSE



mongoose

CORE CONCEPTS

MONGOOSE VS MONGODB

MASYNCTECH SCHOOL

High-Level Explanation

- Mongoose vs. MongoDB native driver:
  - Mongoose: Higher-level, schema-based, validation
  - MongoDB native driver: Lower-level, direct access

Analogue

- Mongoose: Organized kitchen with helpers for correct data storage.
- MongoDB Native Driver: Basic kitchen without organizers; user responsibility for data handling.

Best Practices

- **Mongoose:**
  - Leverage schemas and built-in validation.

- **MongoDB Native Driver**:
  - Manage data integrity manually, exercise caution in DB operations.

Deep Dive

- Mongoose:
- ODM layer with schemas, rich features.
- Strongly-typed schemas, hooks, virtuals.
- Eases development but adds some abstraction.

- **MongoDB Native Driver:**
  - Direct connection, more control, less abstraction.
  - Developer handles data integrity.
  - Faster for basic CRUD due to less overhead.

When to use?

- **Mongoose:**
  - Complex apps, data validation, ease of use.
  - Speedy development.

- **MongoDB Native Driver:**
  - Simple CRUD, intricate MongoDB operations.

MASYNCTECH
SCHOOL

MONGOOSE OFFICIAL DOCS

mongoose

CORE CONCEPTS

MASYNCTECH
SCHOOL

MONGOOSE

INSTALLATION

mongoose

CORE CONCEPTS

MASYNCTECH
SCHOOL

COONECT TO MONGOOSE

mongoose

CORE CONCEPTS

MASYNCTECH
SCHOOL

SCHEMA OVERVIEW

mongoose

CORE CONCEPTS

SCHEMA OVERVIEW

MASYNCTECH
SCHOOL

High-Level Explanation

- MongoDB schema: Blueprint for collection documents
- Mongoose schema: Defines fields, types, validators, structure

Analogue

- Analogy: MongoDB is like open LEGO pieces.
- Mongoose is the rulebook for building a stable LEGO castle.
- Ensures walls, towers, drawbridge follow the right design.

Best Practices

- Define document shape with Mongoose schema.
- Apply suitable field validation.
- Minimize use of `any` data type.
- Implement pre- and post-save hooks as needed.

Deep Dive

- **Mongoose schemas:**
  - Define field types, default values, validation
  - Create virtual properties for app logic
  - Maintain data integrity in schema-less MongoDB
  - Manage complex relationships, validation, hooks, methods

When to use?

Use Mongoose schemas when:

- Structured data with specific types needed.
- Data validation before database save crucial.
- Pre- or post-save hooks required.
- Handling complex relationships between documents.

MASYNCTECH
SCHOOL

SCHEMA TYPES

mongoose

CORE CONCEPTS

SCHEMA TYPES

MASYNCTECH SCHOOL

High-Level Explanation

- Mongoose schema types:
  - `String`, `Number`, `Date`, `Boolean`, `Array`, `ObjectID`
  - Used to define structure of MongoDB documents.

Analogue

- Analogy: Schema types as LEGO blocks
- `**String**`: Long, flat piece for names, descriptions
- `**Number**`: Small, square block for age, price
- `**Array**`: Tray for multiple blocks of same/different shapes

Best Practices

- Opt for the most restrictive type for data integrity.
- Minimize use of `Mixed` unless compelling need.
- Think about performance implications.
- Use `ObjectID` for document references in relational data.

Deep Dive

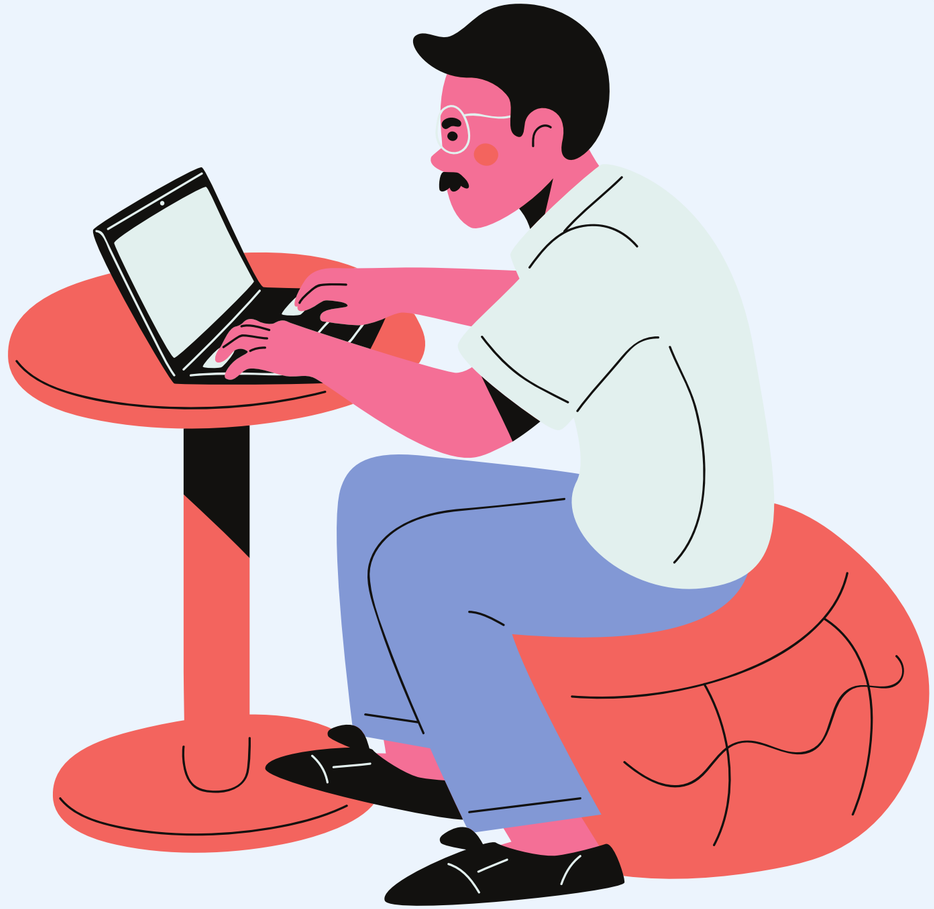
- Mongoose schema types:
  - `**String**`: Textual data, symbols, numbers.
  - `**Number**`: Integers and floats.
  - `**Date**`: Date and optional time.
  - `**Boolean**`: True or false.
  - `**Array**`: Multiple values, often of the same type.
  - `**ObjectID**`: Unique identifier, typically auto-generated.
  - `**Mixed**`: Flexible, holds any data type (use sparingly).
  - Embedded Documents: Nested, complex structures within schemas.

When to use?

Use specific schema types when you:
- Have precise data expectations.
- Need validation or transformation.
- Want data integrity assurance.
- Seek query performance optimization.

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

MODELS

OVERVIEW

mongoose

CORE CONCEPTS

MODELS OVERVIEW

MASYNCTECH
SCHOOL

High-Level Explanation

- Mongoose model: Interface for MongoDB interactions.
- Created by combining schema with CRUD methods.
- Defines fields, types, and how to work with data.

Analogue

- Analogy: Model as a blueprint for houses
- Blueprint (Schema): Defines walls, doors, windows (fields)
- Build houses (documents) based on blueprint
- Find houses with specific features (querying)

Best Practices

- **Single Responsibility:** One model, one concept.
- **Naming**: Singular, capitalized model names.
- M**ethod Attachment**: Attach custom logic via instance or static methods.

Deep Dive

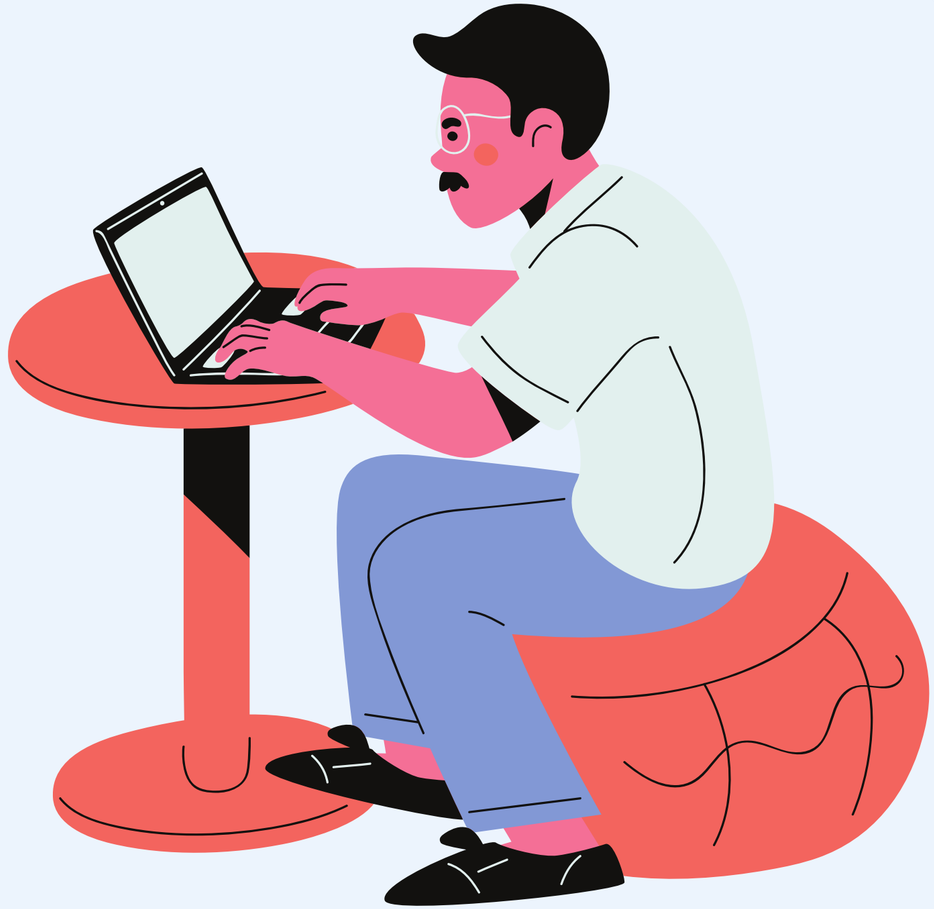
- Models construct MongoDB documents.
- Compiled from a schema.
- Map to MongoDB collections.
- Define document structure.
- Perform CRUD and aggregations.
- Components: Schema, Collection, Document, Query Interface.

When to use?

Use models when you:
- Perform read/write operations in MongoDB.
- Need to enforce structure on collection documents.
- Want to encapsulate data-related business logic (preferably simple logic).

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

MONGOOSE CRUD OPERATIONS

mongoose

MASYNCTECH
SCHOOL

CREATE OPERATIONS (.save())

mongoose

CRUD OPERATIONS

CREATE OPERATIONS USING `.SAVE()

MASYNCTECH SCHOOL

High-Level Explanation

- `.save()` in Mongoose: Create operation, saves new document to DB.

Analogue

- Analogy: `.save()` in Mongoose like 'Save Game' in video games.
- Stores the new 'game state' (document) in MongoDB.

Best Practices

- Always handle promises (`.then/.catch` or `async/await`).
- Validate input before `.save()` to prevent incorrect data.
- Prepare for possible failures like duplicate keys.

Deep Dive

- Instantiate a Mongoose object using a model for an instance.
- Invoke `.save()` on the instance.
- Mongoose handles validation, transformation, MongoDB `insert`.
- Promise resolves on success, rejects on failure.
- Catch errors with `.catch()`.

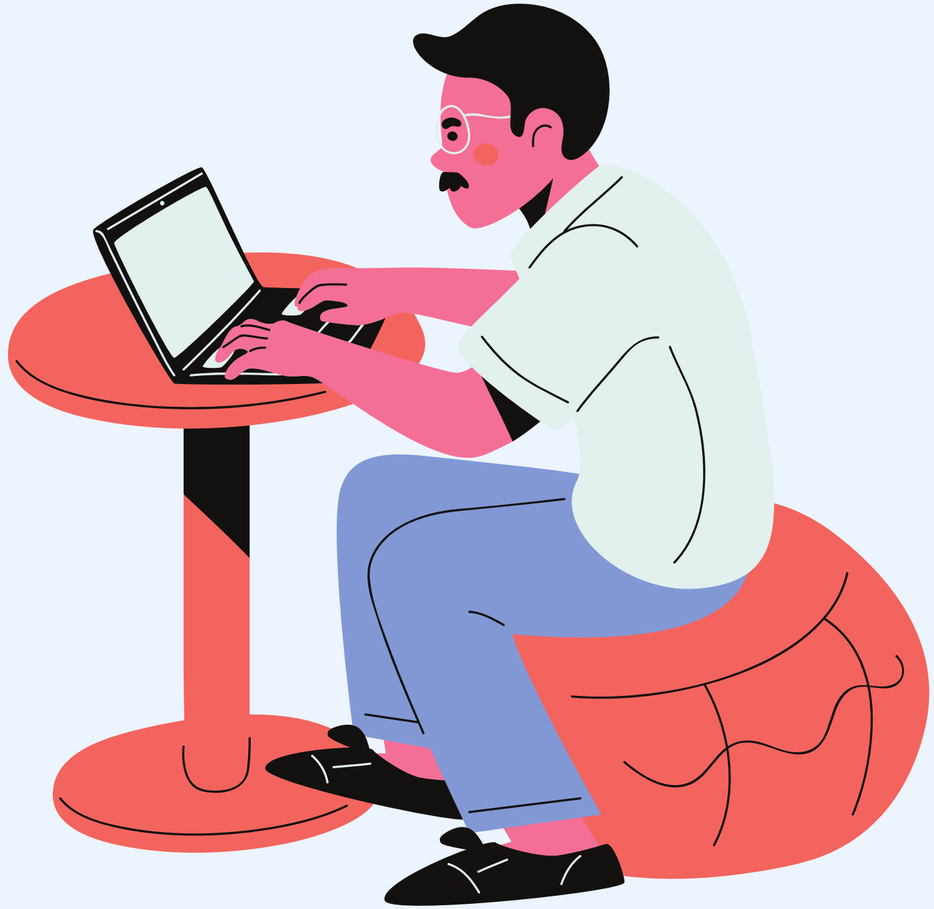
When to use?

Use `.save()` when:

- Inserting a new document into the collection.
- Updating an existing document after making changes.

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

CREATE OPERATIONS (.create())

mongoose

CRUD OPERATIONS

CREATE OPERATIONS USING THE `.CREATE()`

MASYNCTECH SCHOOL

High-Level Explanation

- `**.create()**` in Mongoose: Shortcut for saving one or more documents to MongoDB.
- Combines model instantiation and `.save()` in one step.

Deep Dive

- Traditionally: Instantiate, modify, and `.save()` for a new document.
- `.create()` simplifies, creating and saving in one step.
- Makes code more concise and maintainable.

Analogue

- Analogy: Traditional vs. `.create()` method in video game.
- Traditional: Gather materials, go to site, build.
- `.create()`: Cheat code, instant house construction.

When to use?

Use `.create()` when:

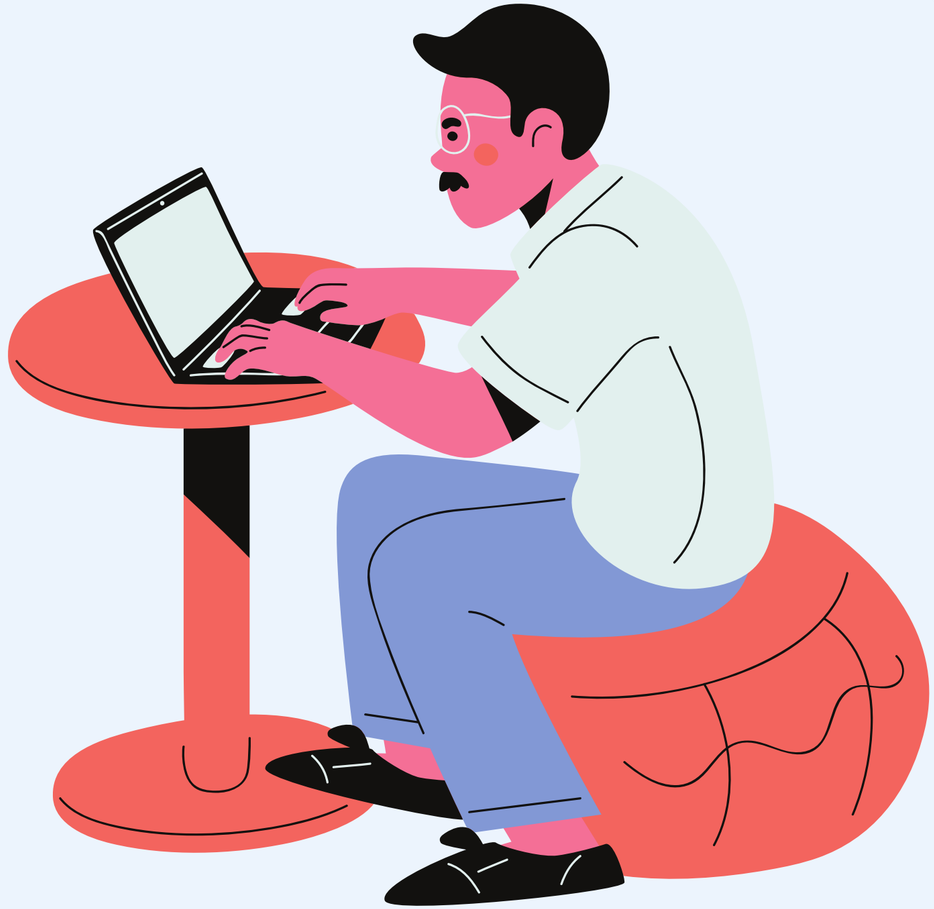
- Swift document insertion without prior manipulation.
- Inserting multiple documents with confidence.
- Aiming for concise and clean code.

Best Practices

- Handle errors with the returned promise (`.then` or `await`).
- Validate data before using `.create()`; schema validations run.
- Caution with multiple document insertions; one failure blocks all.

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

CREATE OPERATIONS (.insertMany())

mongoose

CRUD OPERATIONS

BATCH OPERATIONS USING USING `.INSERTMANY()

MASYNCTECH
SCHOOL

High-Level Explanation

- `.insertMany()` in MongoDB: Insert multiple docs in one operation.
- Useful for batch processing, efficient data insertion.

Deep Dive

- `.insertMany()`: Insert multiple items in collection at once.
- Ideal for seeding databases or efficient bulk inserts.

Analogue

- Analogy: `.insertMany()` like dumping Lego bricks into a big box.
- Saves time and effort by inserting multiple docs at once.

When to use?

Use `.insertMany()` when:

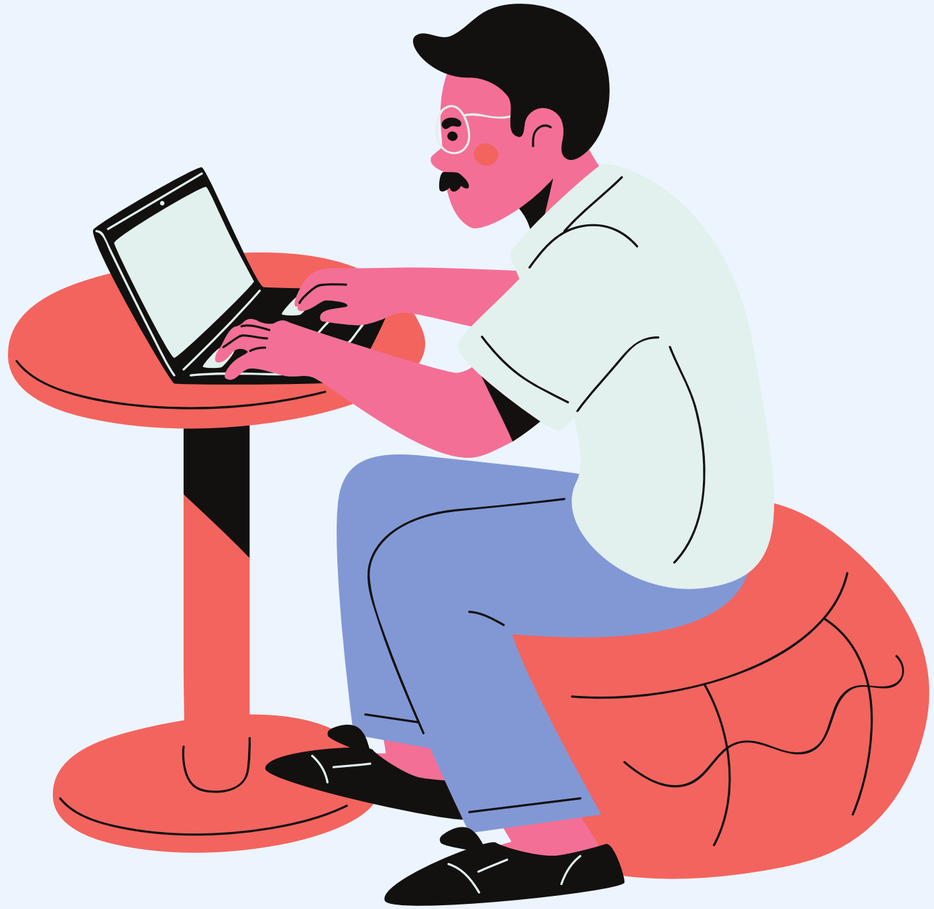
- Seeding a database with initial data.
- Inserting an array of objects into a collection.

Best Practices

- Validate documents against schema.
- Watch document size limit (16MB).

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

CREATE OPERATIONS (.insertMany())

mongoose

CRUD OPERATIONS

BATCH OPERATIONS USING USING `.INSERTMANY()

MASYNCTECH
SCHOOL

High-Level Explanation

- `.insertMany()` in MongoDB: Insert multiple docs in one operation.
- Useful for batch processing, efficient data insertion.

Deep Dive

- `.insertMany()`: Insert multiple items in collection at once.
- Ideal for seeding databases or efficient bulk inserts.

Analogue

- Analogy: `.insertMany()` like dumping Lego bricks into a big box.
- Saves time and effort by inserting multiple docs at once.

When to use?

Use `.insertMany()` when:

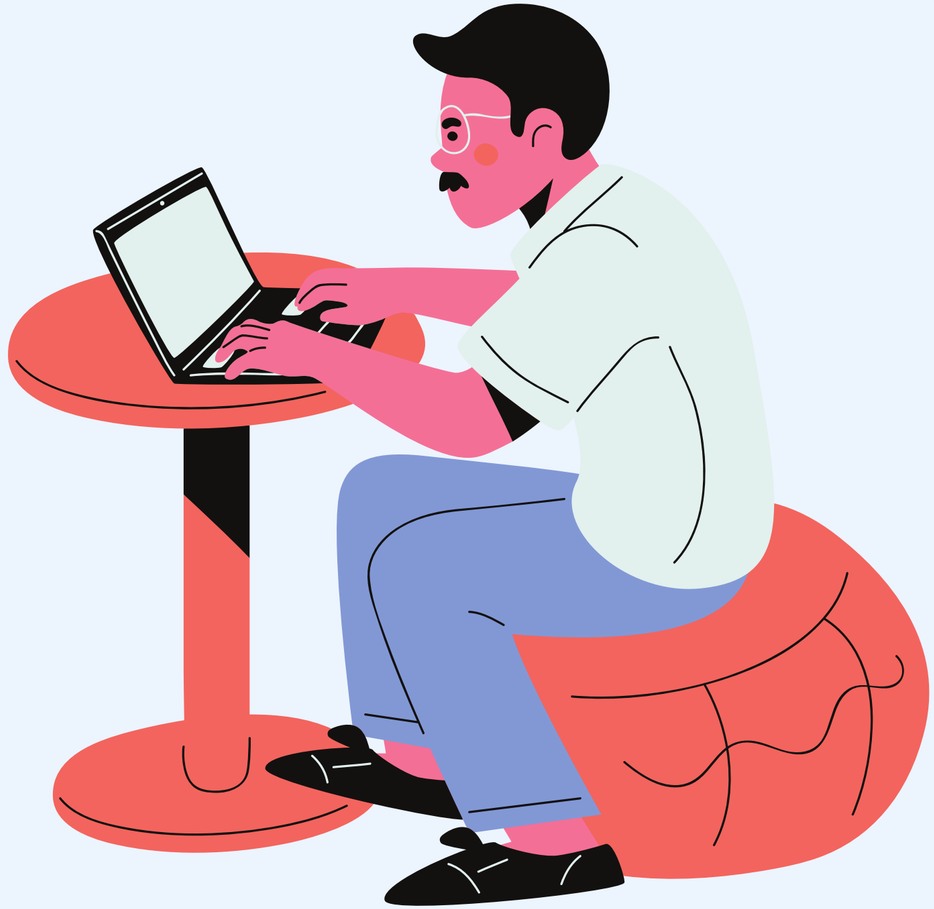
- Seeding a database with initial data.
- Inserting an array of objects into a collection.

Best Practices

- Validate documents against schema.
- Watch document size limit (16MB).

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

READ

OPERATIONS

mongoose

CRUD OPERATIONS

READ OPERATIONS .FIND() AND .FINDONE()

MASYNCTECH SCHOOL

High-Level Explanation

- `.**find**()`: Fetch multiple documents based on query.
- `.**findOne**()`: Retrieve single document based on criteria.

Analogue

- Analogy: `.find()` like scrolling through matches.
- `.findOne()` like picking the top match based on preferences.

Best Practices

- Projections: Limit fields returned, reduce data transfer.
- Options: `sort`, `limit`, `skip` for query fine-tuning.
- Handle errors using try-catch or `.catch()`.

Deep Dive

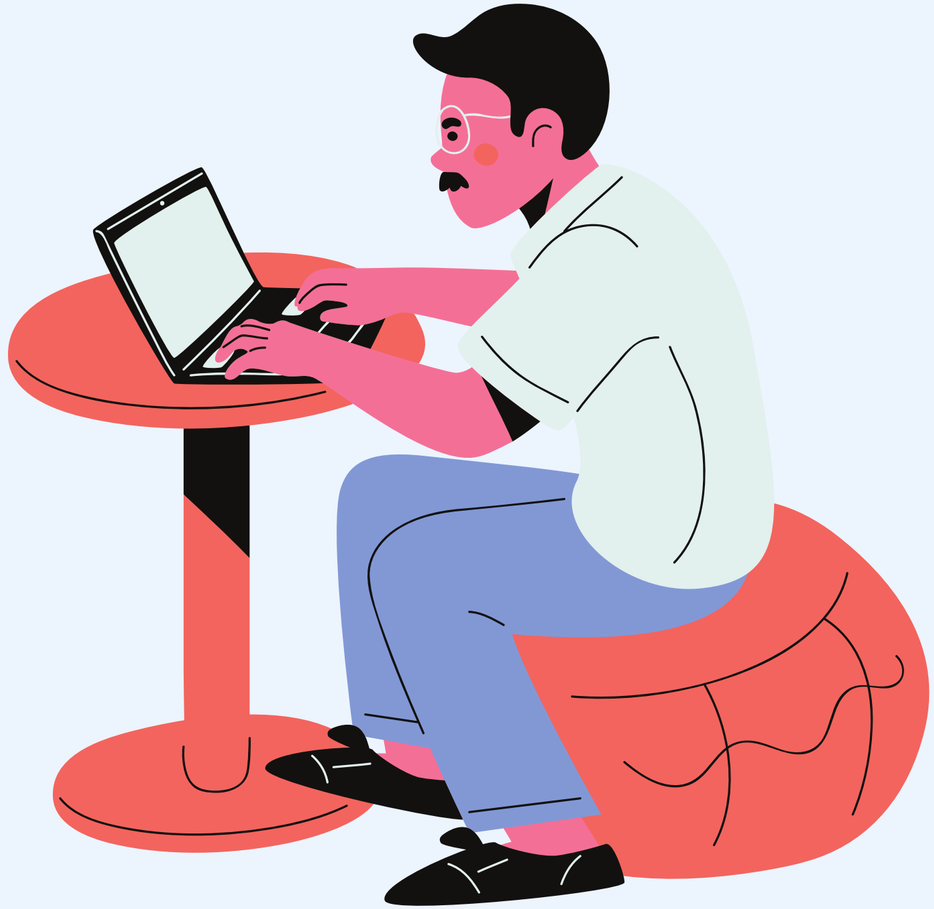
- `.**find**()`: Returns array of matching docs or empty array.
- `.**findOne**()`: Returns first matching doc or `null`.
- Both accept projections, sorting, limiting, and skipping.

When to use?

- `.find()`: For multiple docs, like all products in a category.
- `.findOne()`: For a single doc, like user details by username.

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

READ QUERIES OPERATIONS

mongoose

QUERY OPERATIONS

READ OPERATIONS `.WHERE()`, `.SORT()`, `.LIMIT()`

MASYNCTECH SCHOOL

High-Level Explanation

- `**.where()**`, `**.sort()**`, **and** `**.limit()**` refine query results.
- Typically chained to `.find()` or `.findOne()` queries.

Deep Dive

- `**.where()**`: Filter docs with conditions, logical operators.
- `**.sort()**`: Sort docs based on fields.
- `**.limit()**`: Limit the number of returned docs.

Analogue

- `**.where()**`: "Red" filter
- `**.sort()**`: "Sort by Price: Low to High"
- `**.limit()**`: "Show only the first 5 options"

When to use?

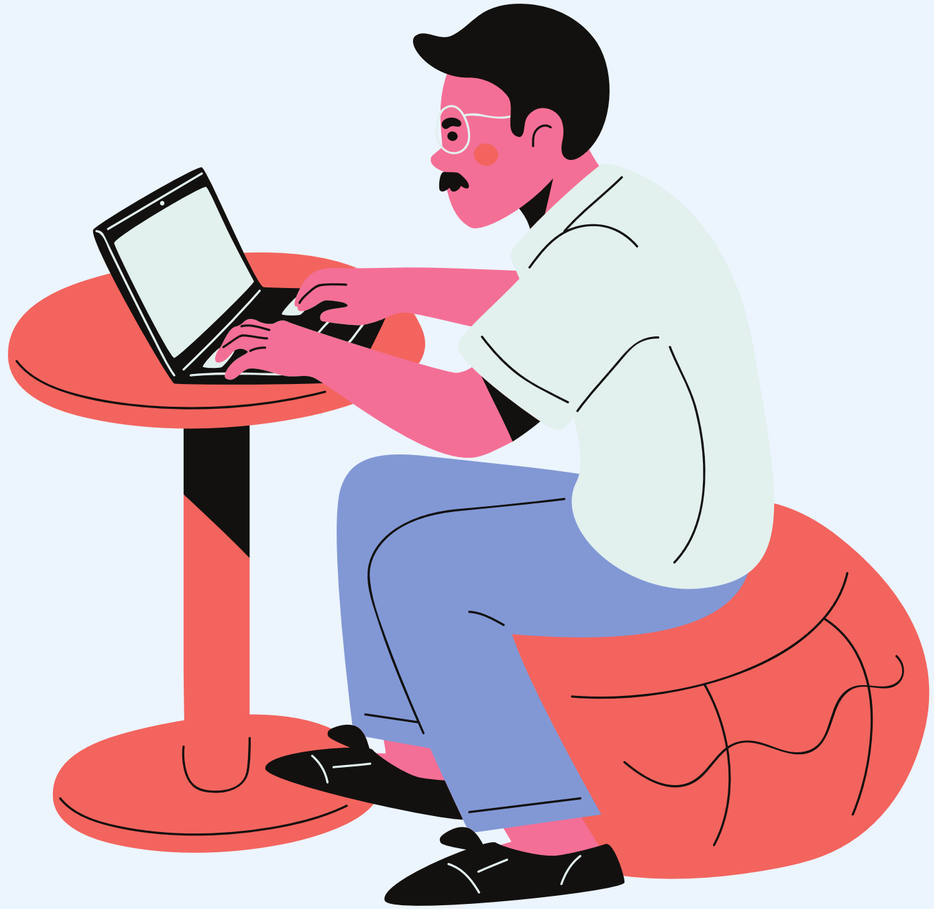
- `**.where**()`: Complex filtering with multiple conditions.
- `**.sort**()`: Organizing data for readability or analysis.
- `**.limit**()`: Pagination or performance optimization by limiting records.

Best Practices

- `**.where**()`: Validate input to prevent injection attacks.
- `**.sort**()`: Let the database handle sorting for efficiency.
- `**.limit**()`: Be cautious with the limit to avoid missing data or overwhelming the system.

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

READ QUERIES OPERATIONS

Filtering Results

mongoose

QUERY OPERATIONS

READ OPERATIONS USING QUERY CONDITIONS

MASYNCTECH SCHOOL

High-Level Explanation

- MongoDB query conditions filter returned documents.
- Operators like `$eq`, `$gt`, `$lt`, `$in`, `$nin` used for criteria.
- Essential for precise search results.

Analogue

- Analogous to music playlist filters
- `$eq`: Genre equals a certain type
- `$gt`: Released after 2010
- `$nin`: Not in disliked genres list
- Filters tailor playlist to specific taste

Best Practices

- Optimize by indexing frequently queried fields.
- Ensure data validation to prevent misuse.
- Enhance readability with helper methods (e.g., `.gte()`, `.lt()`).

Deep Dive

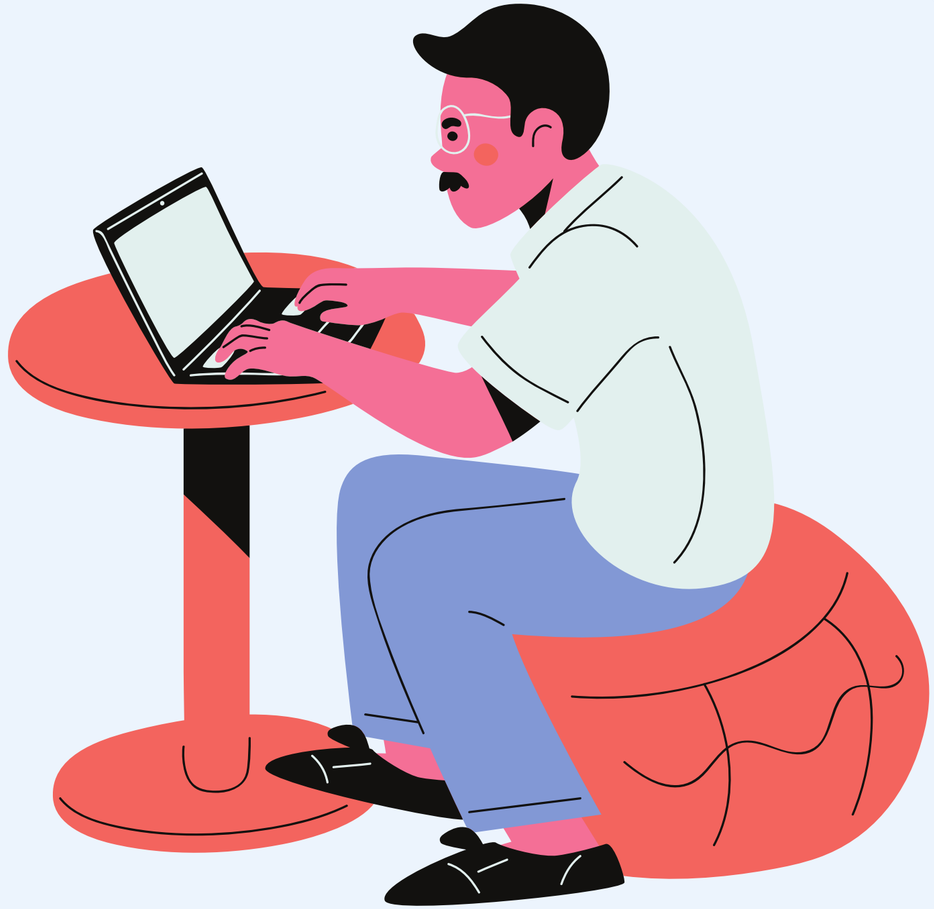
- Query conditions: Key-value pairs
- Key: Field of interest
- Value: Condition (often another key-value pair)
- Examples:
  - `$**eq**`: Equal
  - `$**gt**`: Greater than
  - `$**lt**`: Less than
  - `$**in**`: Matches any in array
  - `$**nin**`: Matches none in array

When to use?

- Query conditions ideal for large datasets
- Valuable in:
  - Data analytics
  - Reporting tools
  - Rapid info retrieval in real-time apps

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

QUERY

PROJECTION

Selecting fields

mongoose

QUERY OPERATIONS

PROJECTIONS: SELECTING SPECIFIC FIELDS

MASYNCTECH SCHOOL

High-Level Explanation

- MongoDB projections: Select specific document fields
- Include/exclude fields, boost query performance
- Minimize data transfer over network

Analogue

- Analogous to phone contact list
- Display only necessary info (name, photo)
- Like MongoDB projections, faster access

Best Practices

- Avoid mixing inclusion/exclusion, except for `_id`
- Use projections carefully for efficiency
- Be mindful of array field results

Deep Dive

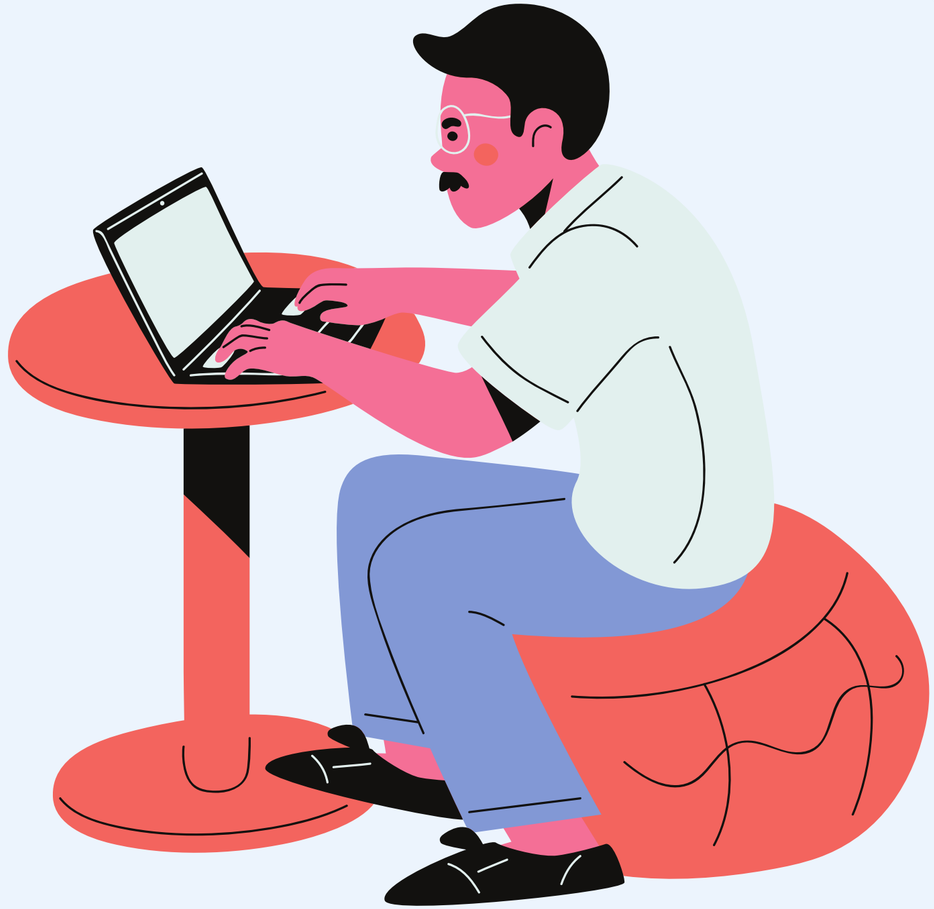
- MongoDB projection in `.find()` and `.findOne()`
- Object specifies inclusion (`1` or `true`) or exclusion (`0` or `false`)
- Exception: `_id` field included by default, can be explicitly excluded.

When to use?

- Reasons to use MongoDB projections:
  - Minimize network data
  - Crucial for data-sensitive mobile apps
  - Tailor fields for specific use cases

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

UPDATE OPERATIONS

Using `.update()` and `.updateOne()`...

mongoose

QUERY OPERATIONS

UPDATE OPERATIONS

MASYNCTECH
SCHOOL

High-Level Explanation

- `**updateOne**`: Update a single document.
- `**updateMany**`: Update multiple documents.
- `**findByIdAndUpdate**`: Find by ID and update a document.
- `**findOneAndUpdate**`: Find one document and update it.

Analogue

- Analogy: Twitter feed as MongoDB collection, tweets as documents
- `**findOneAndUpdate**`: Modify single tweet with conditions
- `**updateMany**`: Change tweets with specific hashtag
- `**findByIdAndUpdate**`: Update tweet with known ID

Best Practices

- Validate updates to avoid unintended changes.
- Use `{ new: true }` option for updated document return.
- Note: Not all update methods trigger Mongoose middleware.

Deep Dive

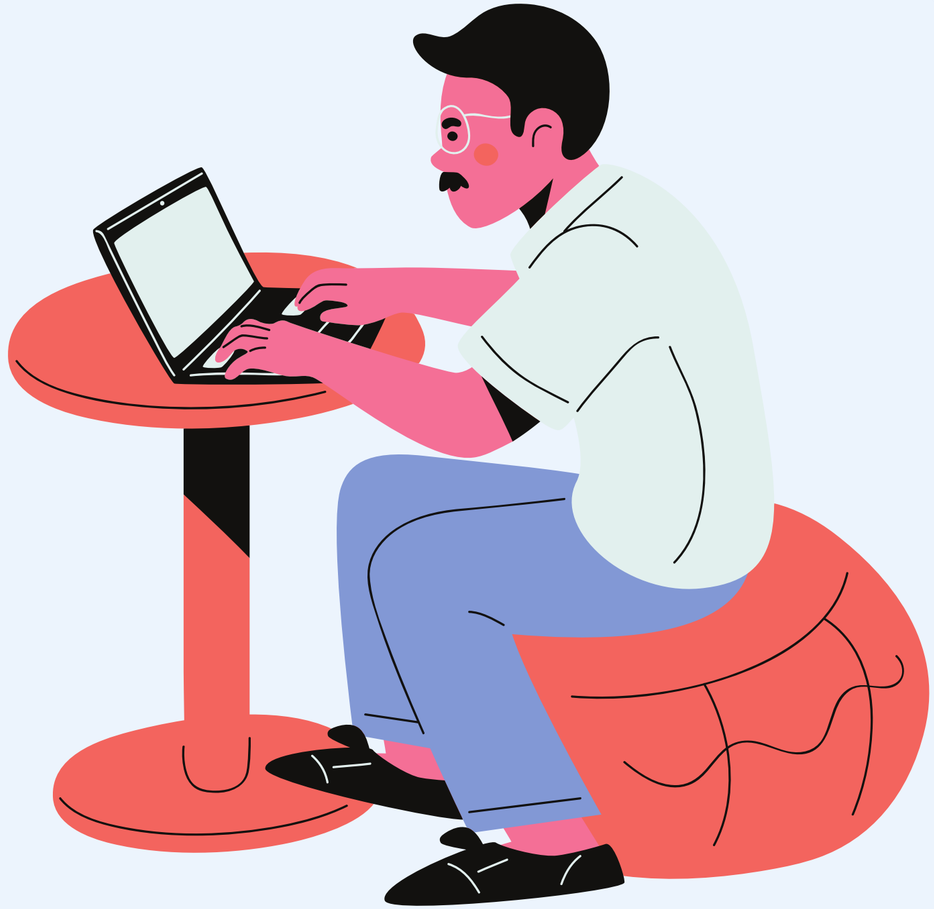
- Use appropriate Mongoose update method for operation scale.
- Define update conditions and changes.
- Framework interacts with MongoDB for updates.
- Account for update options, default values, and middleware.

When to use?

- `**updateOne**`: Single doc, specific criteria.
- `**updateMany**`: Bulk updates, multiple matching criteria.
- `**findByIdAndUpdate**`: Unique ID known.
- `**findOneAndUpdate**`: Update first matching filter.

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

UPDATE

OPERATORS

mongoose

QUERY OPERATIONS

UPDATE OPERATORS

MASYNCTECH
SCHOOL

High-Level Explanation

- Mongoose update operators modify field values in documents.
- Useful for changing values without replacing the entire doc.

Deep Dive

Field Update Operators:

- `$set`: Set field value in document.
- `$unset`: Remove field from document.

Array Update Operators:

- `$addToSet`: Add item if not exists in array.
- `$push`: Add item to array.
- `$pop`: Remove item from array.
- `$pull`: Remove all instances of value from array.
- `$pullAll`: Remove all matching values from array.

Arithmetic Operators:

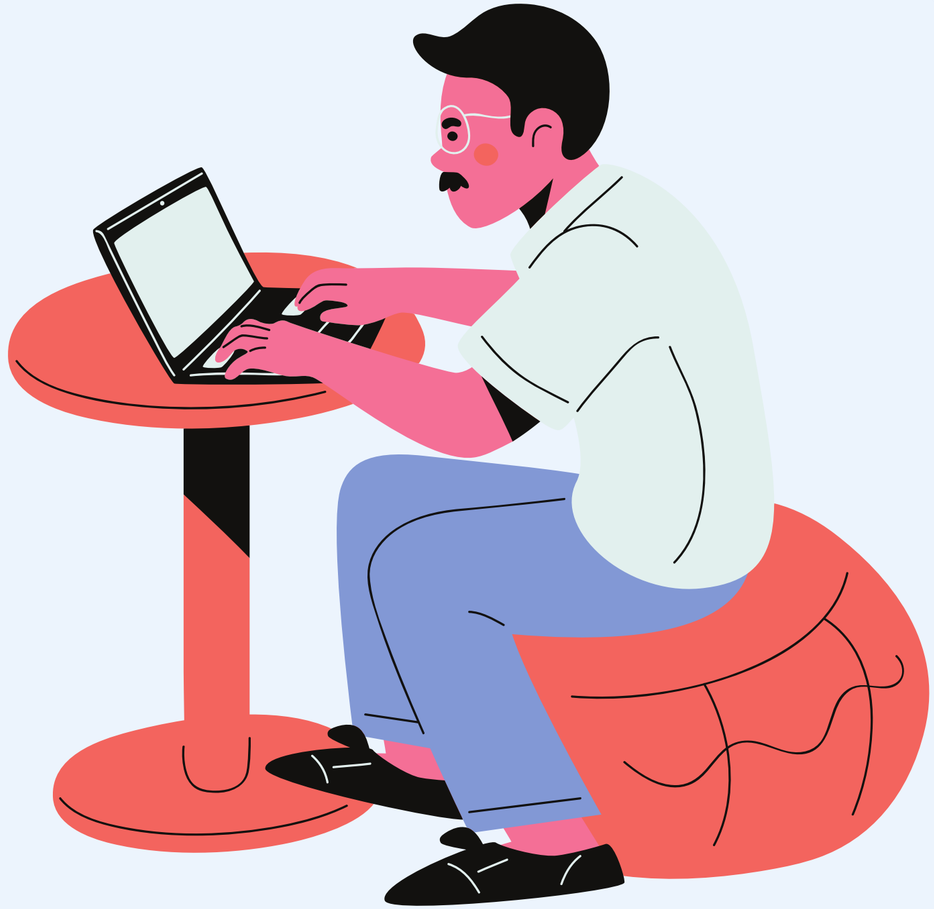
- `$inc`: Increment field's value.
- `$mul`: Multiply field's value.
- `$min`: Update field to minimum of current and given values.
- `$max`: Update field to maximum of current and given values.

Date Operators:

- `$currentDate`: Set field to current date.

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

DELETE

OPERATIONS

Using `.remove()`, `.deleteOne()`, and `.deleteMany()`

mongoose

QUERY OPERATIONS

DELETE OPERATIONS

MASYNCTECH SCHOOL

High-Level Explanation

- `**deleteOne**`: Delete a single document based on criteria.
- `**deleteMany**`: Remove multiple documents matching criteria.
- `**findByIdAndDelete**`: Find by ID and delete a document.
- `**findOneAndDelete**`: Find one document and delete it.

Analogue

- Analogy: Collection as deck of cards
- `**deleteOne**`: Remove first 'Heart'
- `**deleteMany**`: Remove all 'Hearts'
- `**findByIdAndDelete**`: Remove card by ID
- `**findOneAndDelete**`: Remove and show first 'Heart'

Best Practices

- Test deletions on dev database first.
- Use callbacks/promises for error handling or post-deletion actions.
- Be cautious with `deleteMany` in production.

Deep Dive

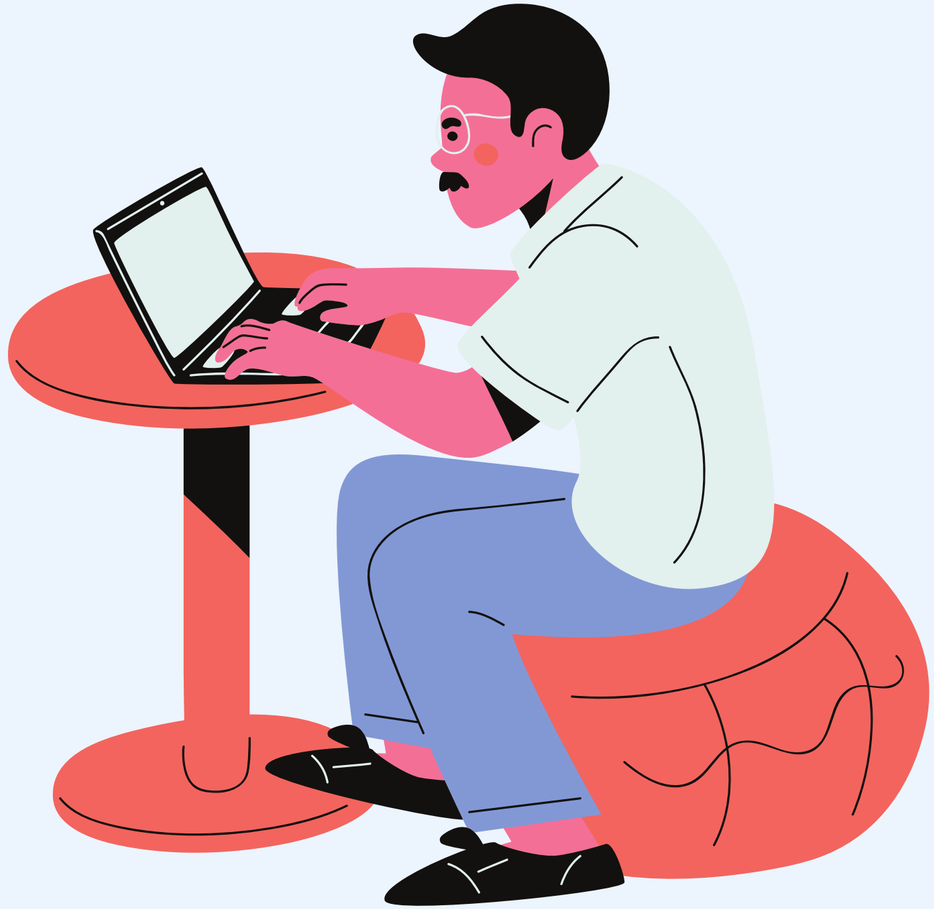
- `**deleteOne**`: Removes first matching document by filter.
- `**deleteMany**`: Deletes all matching documents by filter.
- `**findByIdAndDelete**`: Deletes document by unique `_id`.
- `**findOneAndDelete**`: Deletes and returns first matching doc by query.

When to use?

- `**deleteOne**`: Remove first matching doc.
- `**deleteMany**`: Batch deletion with common attribute.
- `**findByIdAndDelete**`: Delete by known `_id`.
- `**findOneAndDelete**`: Show data of deleted doc.

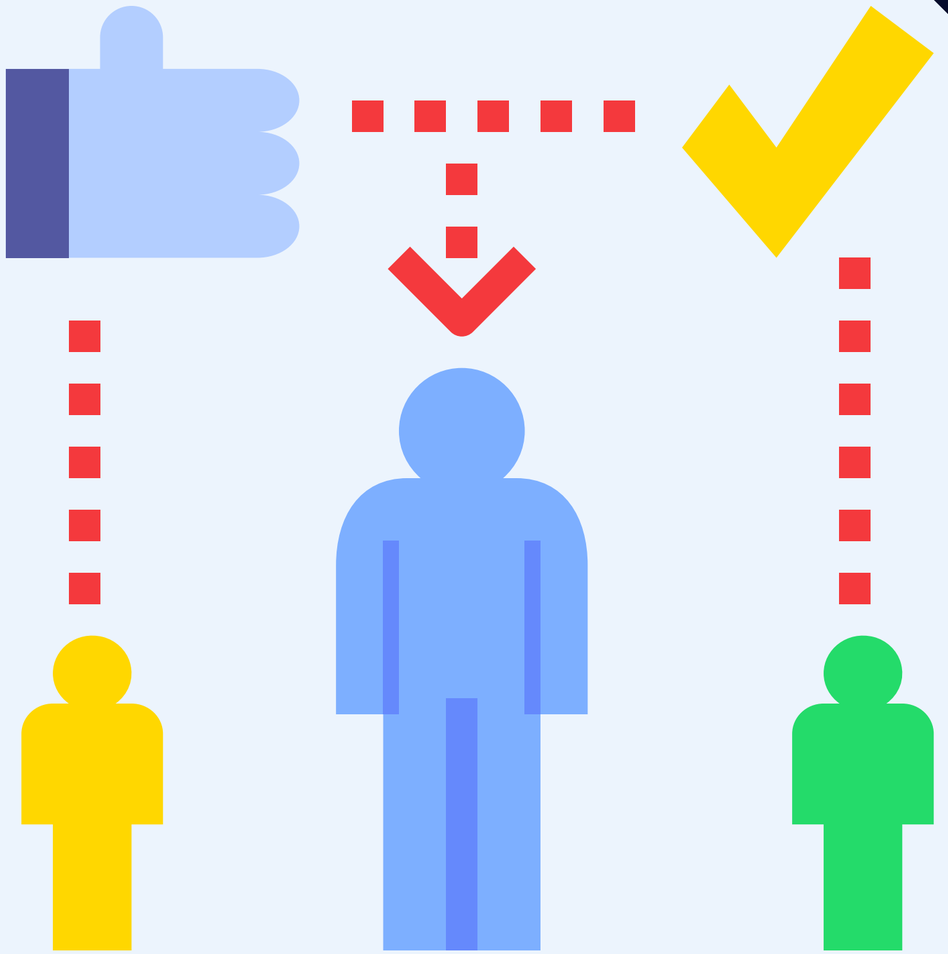
MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

VALIDATIONS

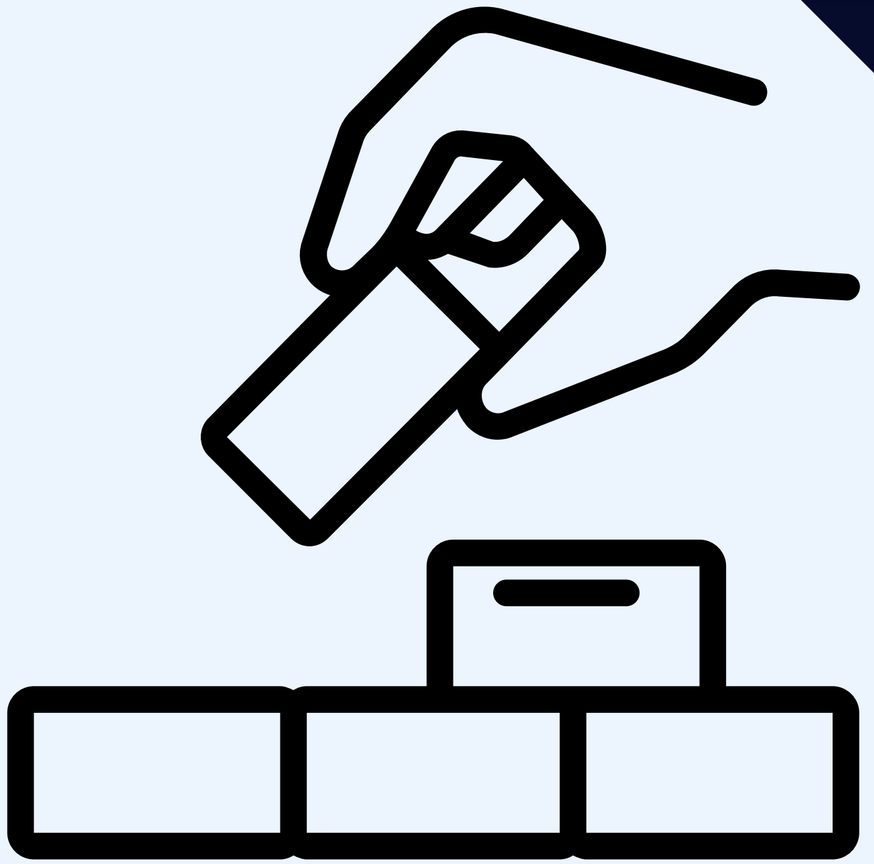


mongoose

MONGOOSE VALIDATION

MASYNCTECH
SCHOOL

BUILT-IN VALIDATORS



mongoose

MONGOOSE VALIDATION

BUILT-IN MONGOOSE VALIDATORS

MASYNCTECH SCHOOL

High-Level Explanation

- Mongoose validators: Enforce data validation on MongoDB docs.
- Predefined checks like `required`, `min`, `max`, `enum`, `unique`, etc.
- Streamline data validation process.

Analogue

- Analogy: Mongoose validators like spell checker.
- Built-in features prevent mistakes.
- E.g., field for email only, Mongoose enforces it.

Best Practices

- Pair with server-side checks for robust validation.
- Don't rely solely on `unique` for critical uniqueness.
- Provide meaningful error messages for client-side feedback.

Deep Dive

Required: Field must exist with a value.
Type: Enforce datatype (String, Number, Date, etc.).
Minlength/Maxlength: Set string length limits.
Validate: Custom async validation.
Min/Max: Enforce numerical range.
Enum: Value must match options in an array.
Unique: Ensure uniqueness within a collection (schema option).
Match: Validate against a regex pattern.

When to use?

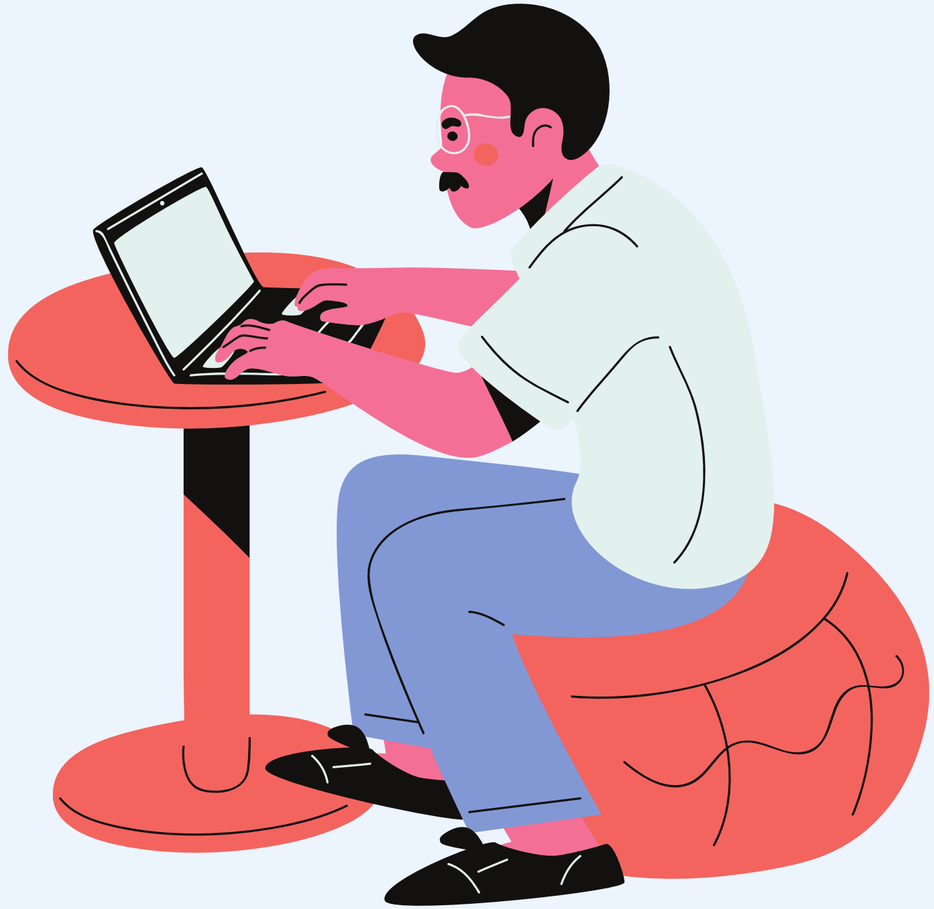
- Use Mongoose built-in validators for:
  - Quick, efficient validation.
  - Standard requirements without custom logic.

Summary

- Mongoose built-in validators: Quick, reliable, standard rules.
- Cover field requirements, types, value constraints.
- Maintain data integrity without extensive custom code.

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

CUSTOM VALIDATION LOGIC

mongoose

MONGOOSE VALIDATION

CUSTOM VALIDATION LOGIC

MASYNCTECH SCHOOL

High-Level Explanation

- Custom validation logic in Mongoose: Define own rules.
- Use `validate` property in schema definition.
- Implement custom function returning `true` or `false` for validation.

Analogue

- Analogy: Hosting a party, acting as a custom validator
- Players pick specific balls based on custom rules
- Ensuring everyone follows the unique criteria

Best Practices

- Handle both synchronous and asynchronous errors.
- Provide clear and meaningful error messages.
- Rigorously test your custom validation logic.
- Keep validation functions as lightweight as possible.

Deep Dive

- Mongoose's `validate` attribute: Custom validation function
- Returns `true` for valid, `false` for invalid
- Supports `Promise` for async validation
- Use for custom criteria beyond built-in validators

When to use?

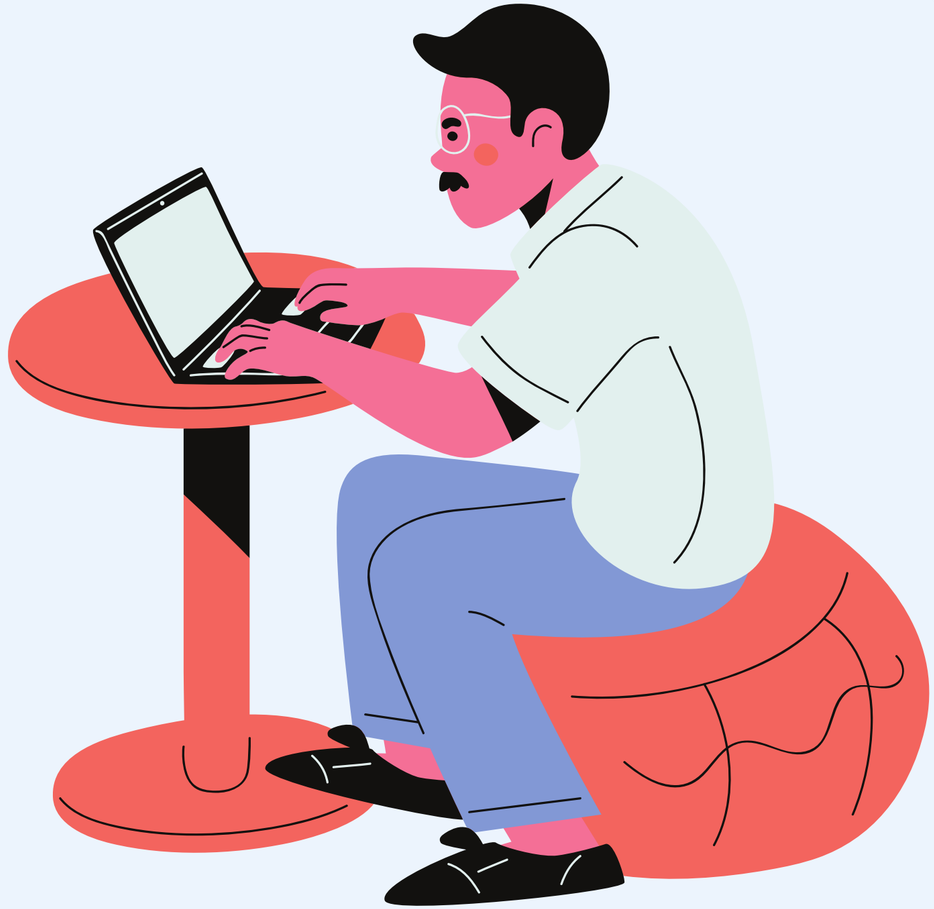
- Built-in validators are inadequate.
- Complex logic spans multiple fields.
- Asynchronous operations, like API calls, are required for validation.

Summary

- Mongoose custom validation: Flexible, beyond built-in validators
- Enforce unique, complex rules, or involve external APIs
- Custom validation for all your specific needs

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

ASYNC CUSTOM VALIDATORS

mongoose

MONGOOSE VALIDATION

ASYNCHRONOUS CUSTOM VALIDATORS

MASYNCTECH SCHOOL

High-Level Explanation

- Asynchronous custom validators in Mongoose: Perform async operations
- Use `validate` with Promise for asynchronous logic
- Ideal for database queries, API calls, async validation

Analogue

- Analogy: Asynchronous validation as party entry
- Bouncer checks list online (async)
- Confirmation depends on list loading

Best Practices

- Handle Promise rejections to prevent unhandled errors.
- Keep operations lightweight to maintain application speed.
- Implement timeouts to prevent indefinite hangs in async operations.

Deep Dive

- Asynchronous custom validator returns a Promise.
- Resolve with `true` for success, `false` for failure.
- Use `async/await` for async operations.
- Validation failure results in rejection and error thrown.

When to use?

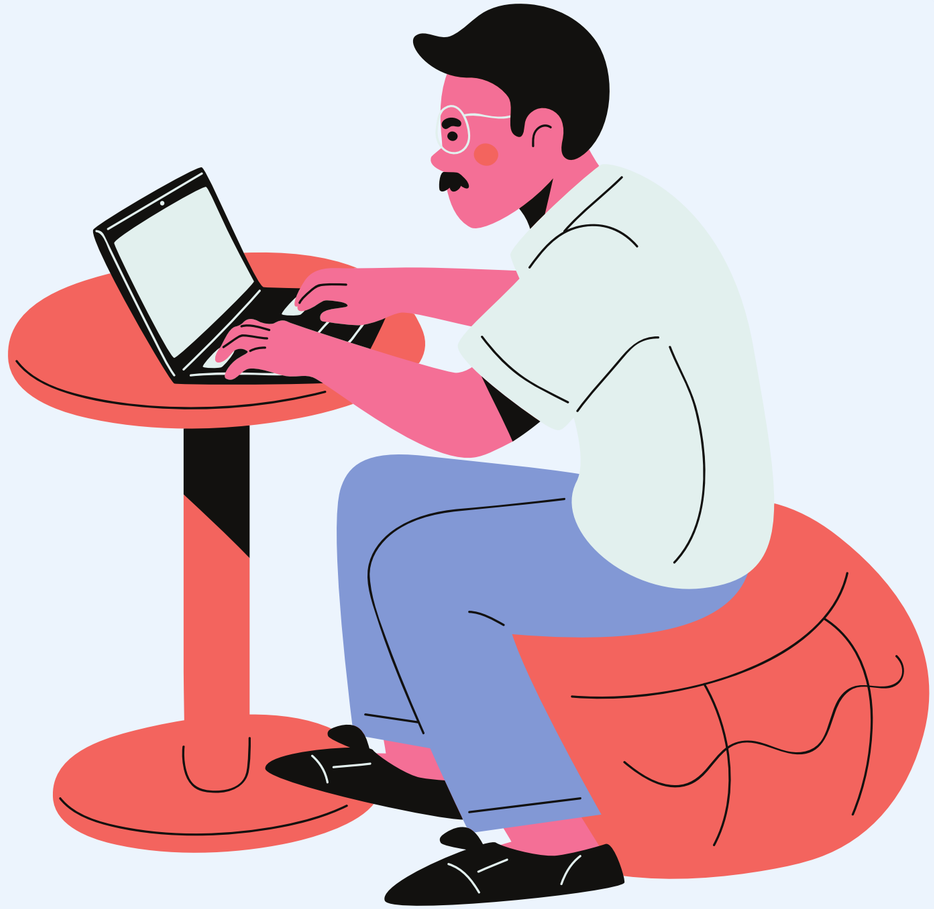
- External database checks for unique values are required.
- Validation relies on external API responses.
- I/O operations are necessary for your validation logic.

Summary

- Asynchronous custom validators: Useful for external/time-consuming logic
- Implemented with Promise in `validator` or `async/await`

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

DATA SANITIZATION TECHNIQUES

mongoose

MONGOOSE VALIDATION

DATA SANITIZATION TECHNIQUES

MASYNCTECH SCHOOL

High-Level Explanation

- Data sanitization: Clean, validate data before processing/saving
- Enforces quality standards, prevents malicious code

Deep Dive

- Data sanitization layers: Client to server-side
- Techniques: Remove special chars, type conversion, allow-lists
- Prevents attacks like SQL injection, XSS, injection
- Mongoose, ExpressJS: Middleware, custom validators, libraries

Analogue

- Analogy: Database as a refrigerator
- Data sanitization as checking and cleaning items
- Ensure only safe, quality data is stored

When to use?

- When handling external or user-generated data.
- For sensitive data with strict formatting (e.g., emails, URLs).
- To protect against malicious activities in database transactions.

Best Practices

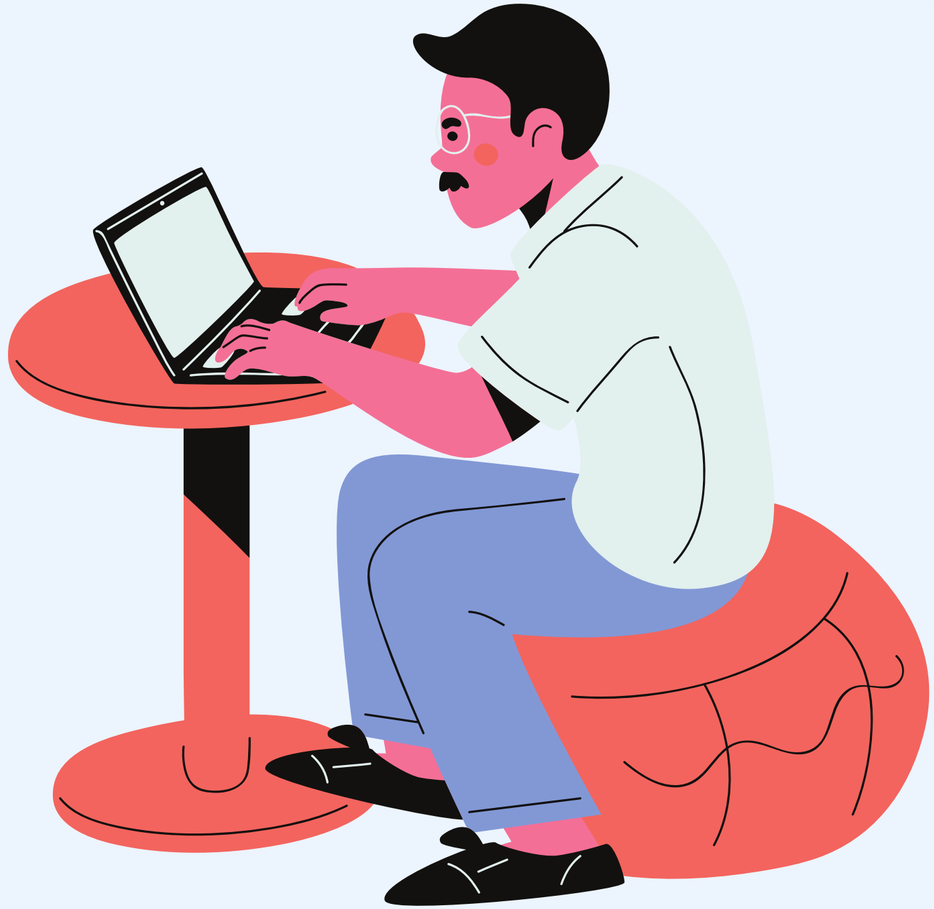
- Never trust user input; sanitize before processing.
- Utilize trusted libraries for common sanitization.
- Validate data types, length, format, and range.
- Use parameterized SQL queries to thwart SQL injection.
- Apply the "Principle of Least Privilege" to limit accepted data.

Summary

- Data sanitization: Clean, validate data for quality, security.
- Essential for application security, data integrity.

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

VALIDATOR LIBRAY

mongoose

MONGOOSE VALIDATION

MASYNCTECH
SCHOOL

THE `SET` & `GET` SCHEMATYPE OPTIONS

mongoose

MONGOOSE VALIDATION

THE `SET` AND `GET` SCHEMATYPE OPTIONS

MASYNCTECH SCHOOL

High-Level Explanation

- `set` option: Transforms data before saving to the database.
- `get` option: Transforms data after retrieval but before use.

Deep Dive

- Setters (`set`): Functions called when setting property values.
- Standardize data formatting (e.g., lowercase, trim).
- Getters (`get`): Functions after retrieving data but before use.
- Modify values (e.g., append domain to username).

Analogue

- Analogy: Digital locker with `set` and `get`
- `set`: Locker organizes items when you put them in.
- `get`: Locker gift-wraps items when you take them out.

When to use?

- Setters: Sanitize, format data before saving to DB.
- Getters: Format, combine fields for app use without altering DB.

Best Practices

- Setters: Avoid complex validation; focus on transformation.
- Note: Setters run each time property is set, not just on save.

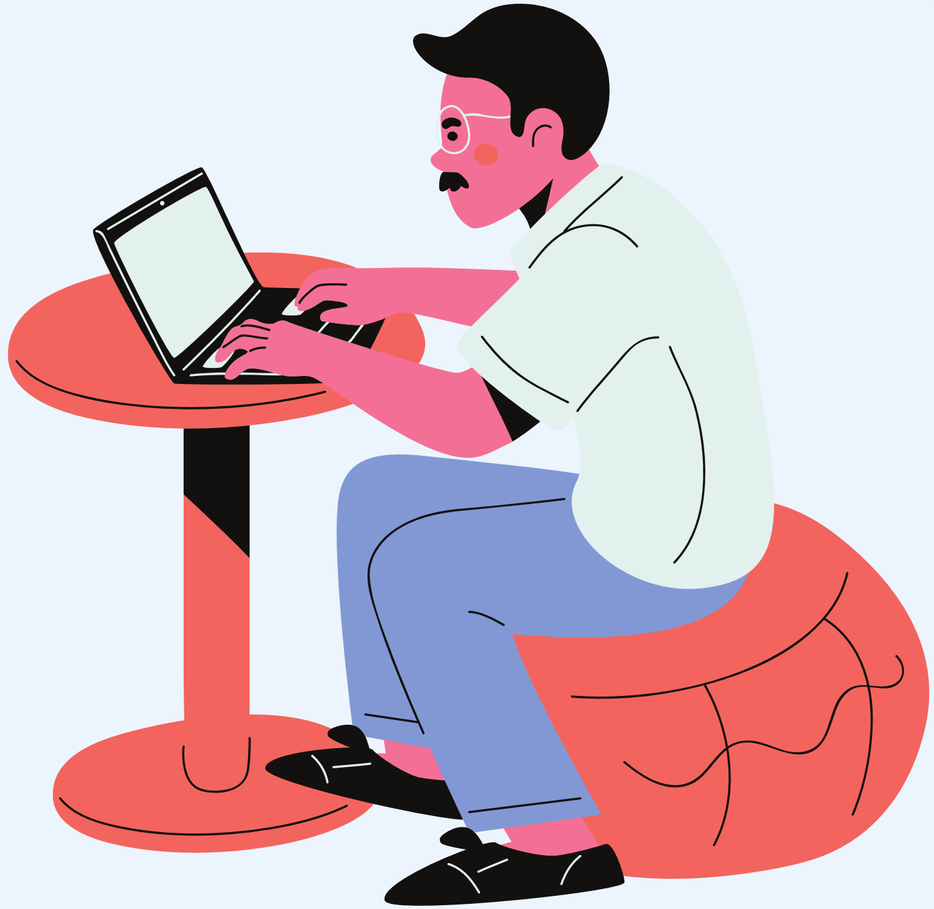
- Getters: Avoid sensitive logic exposed to clients.
- Use sparingly to prevent unexpected behavior, debugging issues.

Summary

- `set` and `get` in Mongoose schemas: Data transformation tools
- Setters prepare, sanitize before storage
- Getters polish, adjust data during retrieval

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

DATA MODELLING

Overview

mongoose

DATA MODELLING OVERVIEW

MASYNCTECH
SCHOOL

High-Level Explanation

- Data modeling: Defines data shape, relationships, integrity.
- Mongoose schemas: Specify MongoDB document structure.

Deep Dive

- Data modeling: High-level design, relationships, integrity, DB-agnostic.
- Mongoose schemas: Specific for MongoDB, define types, validation, structure.

Analogue

- Analogy: Data modeling as house furniture arrangement.
- Schema as furniture instruction manual.
- Defines parts and assembly for specific furniture.

When to use?

- Data modeling: Initial app design, high-level data plan.
- Schemas: Implement high-level design in MongoDB with Mongoose.

Best Practices

- Data modeling: Grasp business requirements thoroughly.
- Schemas: Define types, set up validation for data integrity.

Summary

- Data modeling: Design data structure and relations.
- Schemas in Mongoose: Implement design for MongoDB documents.

MASYNCTECH
SCHOOL

EMBEDDED DOCUMENTS

mongoose

DATA MODELLING

UNDERSTANDING EMBEDDED DOCUMENTS

MASYNCTECH SCHOOL

High-Level Explanation

- Embedded documents: Sub-docs within a parent doc.
- Suitable for nested structures, e.g., address in user profile.

Analogue

- Analogy: Embedded docs as backpack pockets.
- Backpack = Main item (parent doc).
- Pockets = Embedded docs holding specific items.
- Pockets are part of backpack, not separate.

Best Practices

- Monitor document size limit.
- Avoid embedded docs for frequently independent queries.
- Acknowledge complexity in updates to embedded docs within parent.

Deep Dive

- Define embedded documents in Mongoose with separate schema.
- Include them in parent schema as array or nested object.
- Embedded docs have fields, validation, methods.
- No separate database collection; only in parent doc.
- Efficient reads but potential document size limits.

When to use?

- Use embedded docs when:
  - Clear hierarchical data.
  - Frequent retrieval of parent and embedded data.
  - Embedded data within MongoDB's size limit (16MB).

Summary

- Embedded docs in Mongoose: Sub-docs in parent doc.
- Ideal for nested, hierarchical data.
- Limitations include MongoDB's doc size limit.

UNDERSTANDING EMBEDDED DOCUMENTS

MASYNCTECH SCHOOL

parent document schema

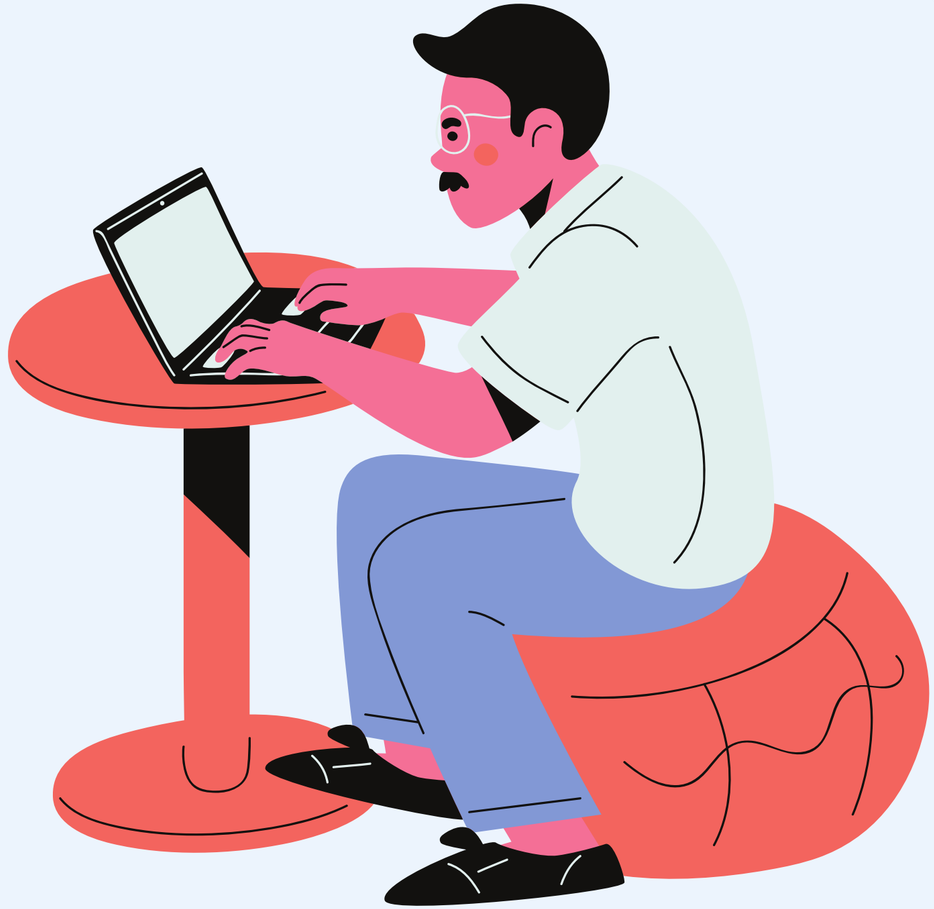
```
const UserSchema = new Schema({
  name: String,
  email: String,
  address: AddressSchema // Embedding AddressSchema
});
```

Embedded document schema

```
const AddressSchema = new Schema({
  street: String,
  city: String,
  state: String,
  zip: Number
});
```

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

ARRAYS OF
SUB-DOCUMENTS

mongoose

DATA MODELLING

WORKING WITH ARRAYS OF SUBDOCUMENTS

MASYNCTECH SCHOOL

High-Level Explanation

- Arrays of subdocuments in Mongoose manage embedded docs.
- Useful for one-to-many relationships within one parent doc.
- Manipulate arrays for efficient data management (add, update, delete subdocs).

Analogue

- Analogy: Embedded docs as backpack pockets.
- Backpack = Main item (parent doc).
- Pockets = Embedded docs holding specific items.
- Pockets are part of backpack, not separate.

Best Practices

- Watch for MongoDB's 16MB doc size limit.
- Large subdoc arrays: Consider pagination or separate collections.
- Use Mongoose validation for subdocs to ensure data integrity.

Deep Dive

- Create arrays of subdocs with `[]` notation in schema.
- Add subdocs using `push` or by assigning a new array.
- Update by finding parent and using array methods or Mongoose subdoc API.
- Delete by retrieving parent and using `splice` or Mongoose methods.
- Query subdoc fields, but performance may differ from standalone docs.

When to use?

- Use arrays of subdocs for:
  - Strongly related nested data.
  - Collections within doc size limits.
  - Operations often need entire sub-array, reducing complex queries.

Summary

- Arrays of subdocs in Mongoose: Manage one-to-many within one doc.
- Great for smaller, closely related data sets.
- Challenges: Complex querying, 16MB doc size limit.
- Awareness of constraints informs design decisions.

WORKING WITH ARRAYS OF SUBDOCUMENTS

MASYNCTECH
SCHOOL

Sub document schema

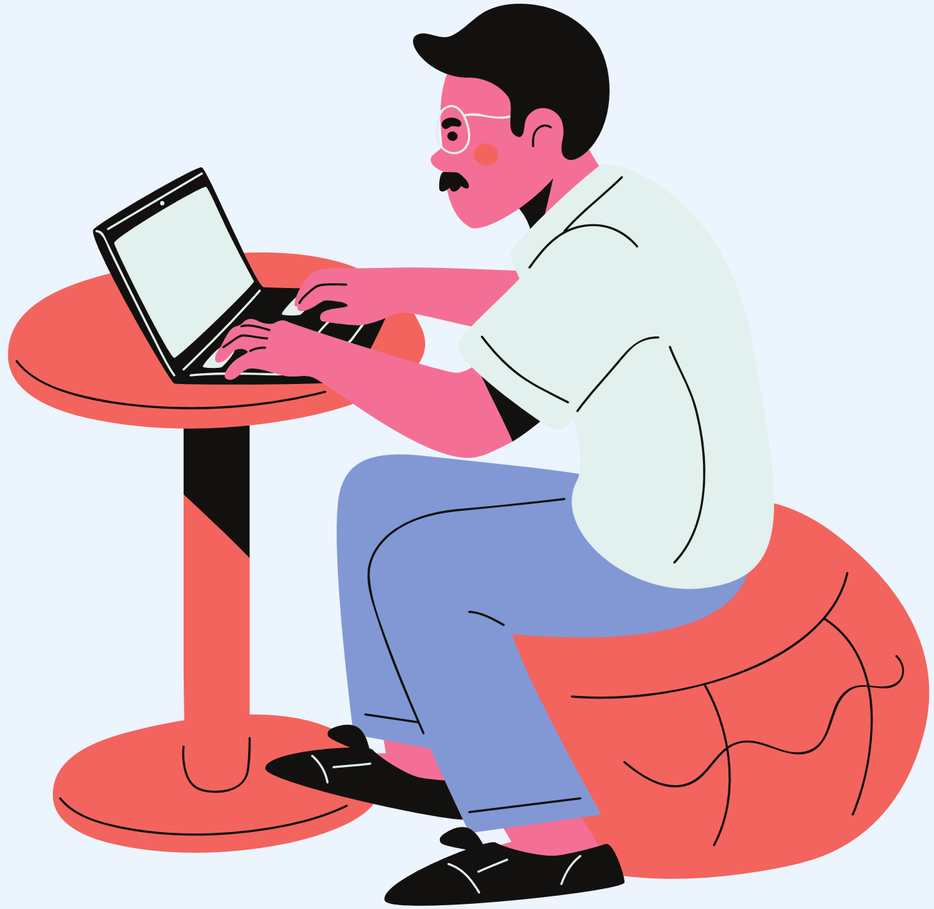
```
const SubdocumentSchema = new mongoose.Schema({
  field1: String,
  field2: Number
});
```

Parent schema

```
const ParentSchema = new mongoose.Schema({
  subdocuments: [SubdocumentSchema]
});
```

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

LIMITATIONS OF EMBEDDED DOCUMENTS

mongoose

DATA MODELLING

LIMITATIONS OF EMBEDDED DOCUMENTS

MASYNCTECH SCHOOL

High-Level Explanation

- Embedded docs in Mongoose: Store nested data in MongoDB doc.
- Manage relationships without joins.
- Limitations: Doc size constraints, query complexity, limited ops support.
- Consider pros and cons for design decisions.

Deep Dive

- MongoDB's 16MB BSON document size limit applies to embedded docs.
- Query complexity increases with nested fields.
- Some operations (e.g., "distinct") less straightforward.
- Redundancy and data integrity challenges.
- ACID transactions less performant on deeply nested embedded docs.

Analogue

- Analogy: Computer's game folders.
- Main folder = "My Games."
- Sub-folders = Embedded docs.
- 16MB limit = Computer's "no more space."
- Complexity = Treasure hunt for specific info in embedded docs.

MASYNCTECH
SCHOOL

DATA

REFERENCING

mongoose

DATA MODELLING

UNDERSTANDING DATA REFERENCING

MASYNCTECH SCHOOL

High-Level Explanation

- Data referencing in Mongoose: Linking docs from different collections.
- Used for many-to-many, one-to-many relationships.
- Avoids size limits, supports complex queries.

Analogue

- Analogy: Toy box (collection) and friends (another collection).
- Referencing: Friend's name on toy list, not on toy (no embedding).
- Query: List helps find who has which toy.

Best Practices

- `.populate()`: Easy data retrieval but can be slow for large datasets.
- For frequent multi-collection references, optimize data architecture (indexing).
- Beware "callback hell" with nested queries for multiple references.

Deep Dive

- Data referencing in Mongoose: Two types - manual and population.
- **Manual referencing**: Store related `_id`, manual queries.
- **Population**: `.populate()` auto-replaces `_id`, simpler retrieval.
- **Normalization**: Data separate in collection, not duplicated.
- Contrast: Denormalized data has subdocs embedded in parent.

When to use?

- Use data referencing when:
  - Referenced data is large (exceeds 16MB limit).
  - Data is shared among multiple docs.
  - Complex queries on referenced data needed.

Summary

- Data referencing in Mongoose: Relate separate docs.
- Manual referencing = query control.
- `.populate()`: Simplifies data retrieval.
- Strategic use enhances database efficiency.

UNDERSTANDING DATA REFERENCING

MASYNCTECH
SCHOOL

Authors document schema

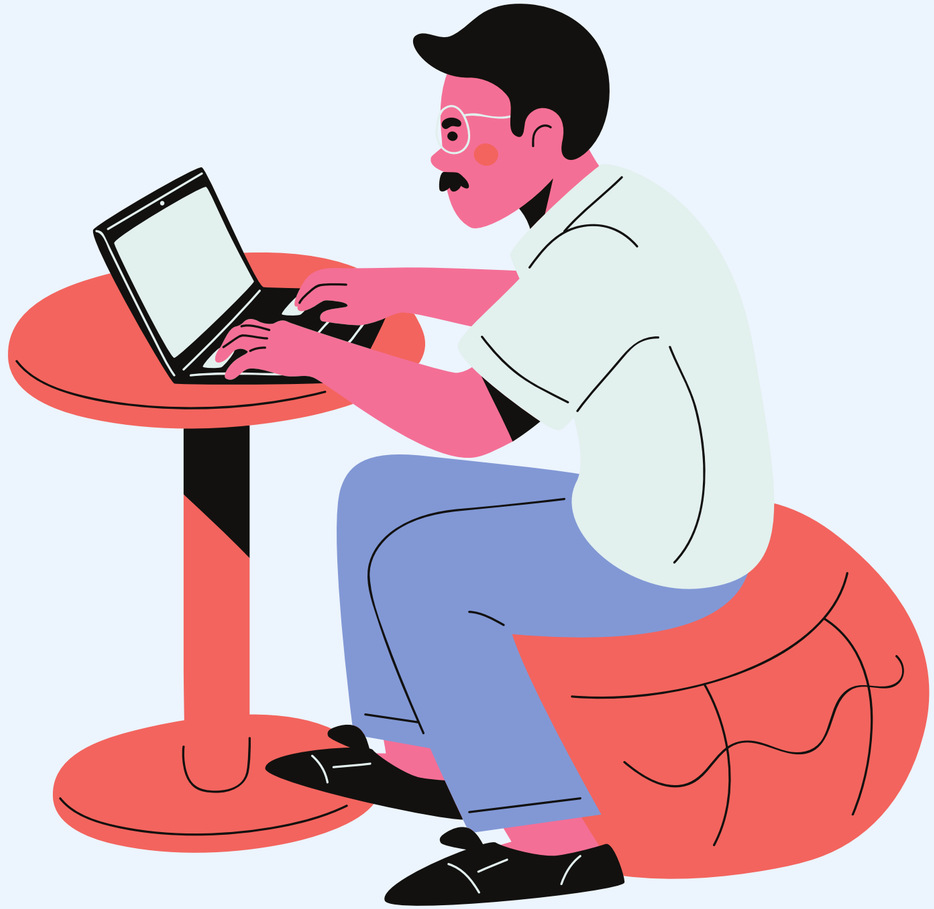
```
const AuthorSchema = new mongoose.Schema({
  name: String,
  birthYear: Number
});
```

Books document schema

```
const BookSchema = new mongoose.Schema({
  title: String,
  author: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Author'
  }
});
```

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

THE `REF` & POPULATE PROPERTIES

mongoose

DATA MODELLING

THE `REF` PROPERTY AND POPULATION

MASYNCTECH
SCHOOL

High-Level Explanation

- `ref` specifies the referenced model in a schema.
- `populate` replaces Object IDs with data from referenced model.
- Similar to SQL JOIN, simplifying data retrieval.

Deep Dive

- `ref` property: Used to specify the referenced model in a schema.
- `populate` method: Replaces Object IDs with data from the referenced model.
- Query-level population: Allows for dynamic data retrieval as needed.

Analogue

- The `ref` property specifies the referenced model for a field.
- `populate()` fetches data from the referenced model, akin to opening a treasure chest to reveal its contents in a game.

When to use?

- Use `ref` in schema to specify referenced model.
- `populate` dynamically replaces Object IDs with data.
- Useful for normalized data, complex updates, and on-demand loading.

Best Practices

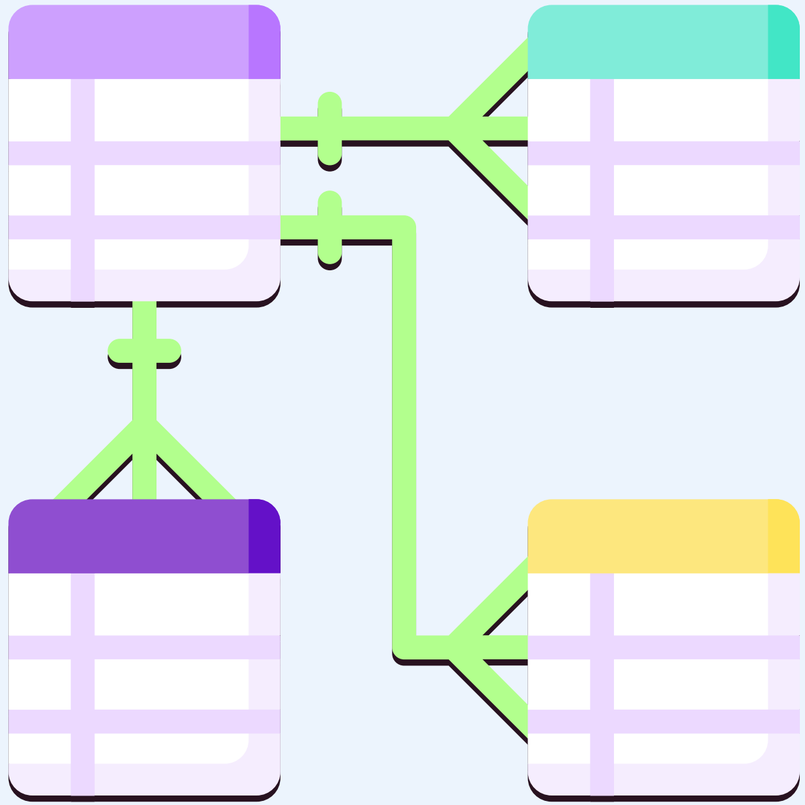
- Use population sparingly for large datasets.
- Selective field population can limit populated fields.

Summary

- `ref` property specifies the referenced model for an Object ID field.
- `populate` uses this reference to replace IDs with actual docs.
- Essential for managing complex data relationships in Mongoose.

MASYNCTECH
SCHOOL

DATA RELATIONSHIP



mongoose

MONGOOSE DATA RELATIONSHIP

MASYNCTECH
SCHOOL

ONE-ONE RELATIONSHIPS

mongoose

MONGOOSE DATA RELATIONSHIP

ONE-TO-ONE RELATIONSHIPS

MASYNCTECH SCHOOL

High-Level Explanation

- One-to-one relationship: Single doc in one collection matches one in another.
- Implement via Embedded Docs or References in Mongoose.

Analogue

- Embedded: "About Me" content directly on profile page.
- Reference-based: "About Me" content on a separate page, linked from the profile.

Best Practices

- Embed data for read-heavy and stable data.
- Use references for write-heavy or dynamic data.
- Base decision on application's data access patterns.

Deep Dive

- Embedded: Store related doc as sub-doc within parent.
- Reference-based: Use ObjectId reference in another collection.
- Choice depends on app requirements: access patterns, change rate, data size.

When to use?

- Frequent access to associated data together.
- True one-to-one connection between documents.
- Data size won't exceed MongoDB's document size limit.

Summary

- One-to-one relationships in MongoDB via embedding or referencing.
- Decision hinges on app needs, like data access and change frequency.

MASYNCTECH SCHOOL

Embedded Approach

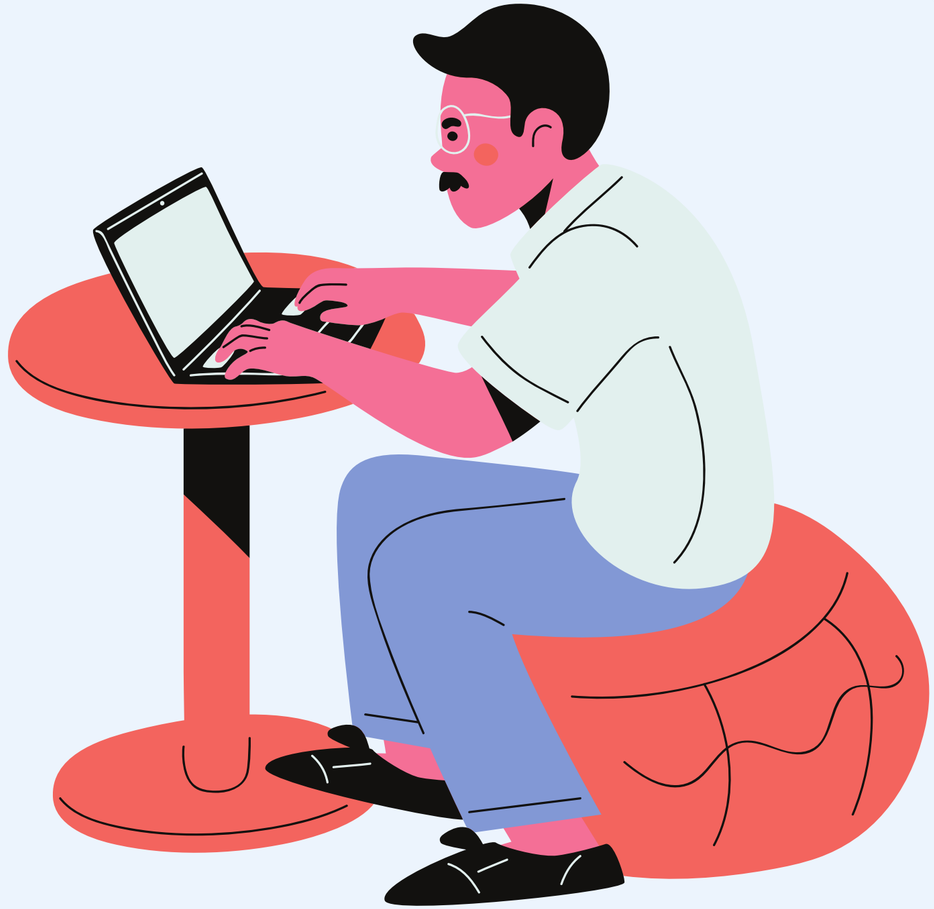
```
Users Collection
└── User Document
    ├── _id: ObjectId("...")
    ├── name: "John Doe"
    ├── email: "john.doe@example.com"
    └── address: Embedded Document
        ├── street: "123 Main St"
        ├── city: "Springfield"
        └── zipCode: "62704"
```

Reference-Based Approach

```
Users Collection                                  Addresses Collection
└── User Document                                  └── Address Document
    ├── _id: ObjectId("...")                           ├── _id: ObjectId("...")
    ├── name: "John Doe"                               ├── street: "123 Main St"
    ├── email: "john.doe@example.com"                  ├── city: "Springfield"
    └── address: ObjectId("...")  --------> └── zipCode: "62704"
```

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

ONE-MANY
MANY-ONE
RELATIONSHIPS

mongoose

MONGOOSE DATA RELATIONSHIP

ONE-TO-MANY AND MANY-TO-ONE RELATIONSHIPS

MASYNCTECH SCHOOL

High-Level Explanation

- One-to-many: One document in Collection A associated with multiple in Collection B.
- Many-to-one: Multiple in Collection A refer to one in Collection B.
- Perspective shift differentiates the two.

Analogue

- Classroom analogy: One-to-many – one teacher, many students.
- Student's perspective: Many students, one teacher – many-to-one.

Best Practices

- Use arrays for one-to-many but watch document size limit.
- Avoid deep nested arrays for simpler queries/updates.
- In many-to-one, maintain referential integrity by verifying existence of referred-to document.

Deep Dive

- **One-to-many:** Use arrays with ObjectIds in source collection.
  - E.g., Blog post with an array of comment references.
- **Many-to-one**: Multiple records refer to one in target collection.
  - E.g., Many comments belong to a single blog post, specified by a field.

When to use?

- **One-to-many:** Good for logical grouping, infrequent access.
  - Data consistency less critical (no MongoDB transactions).
- **Many-to-one**: Useful when frequent access to a single document.
  - Logical to link multiple records to one central record.

Summary

- One-to-many and many-to-one are two perspectives of the same relationship.
- One-to-many: Array of references in 'one' to 'many.'
- Many-to-one: 'Many' pointing back to the 'one.'

MASYNCTECH SCHOOL

One-to-Many Relationship

```
BlogPosts Collection                          Comments Collection
└── BlogPost Document                         └── Comment Document 1
    ├── _id: ObjectId("...")                      ├── _id: ObjectId("...")
    ├── title: "My First Blog"                    ├── text: "Great post!"
    └── comments: Array of ObjectIds  --->        └── blogPostId: ObjectId("...")
                                                  └── Comment Document 2
                                                      ├── _id: ObjectId("...")
                                                      ├── text: "Thanks for sharing!"
                                                      └── blogPostId: ObjectId("...")
```

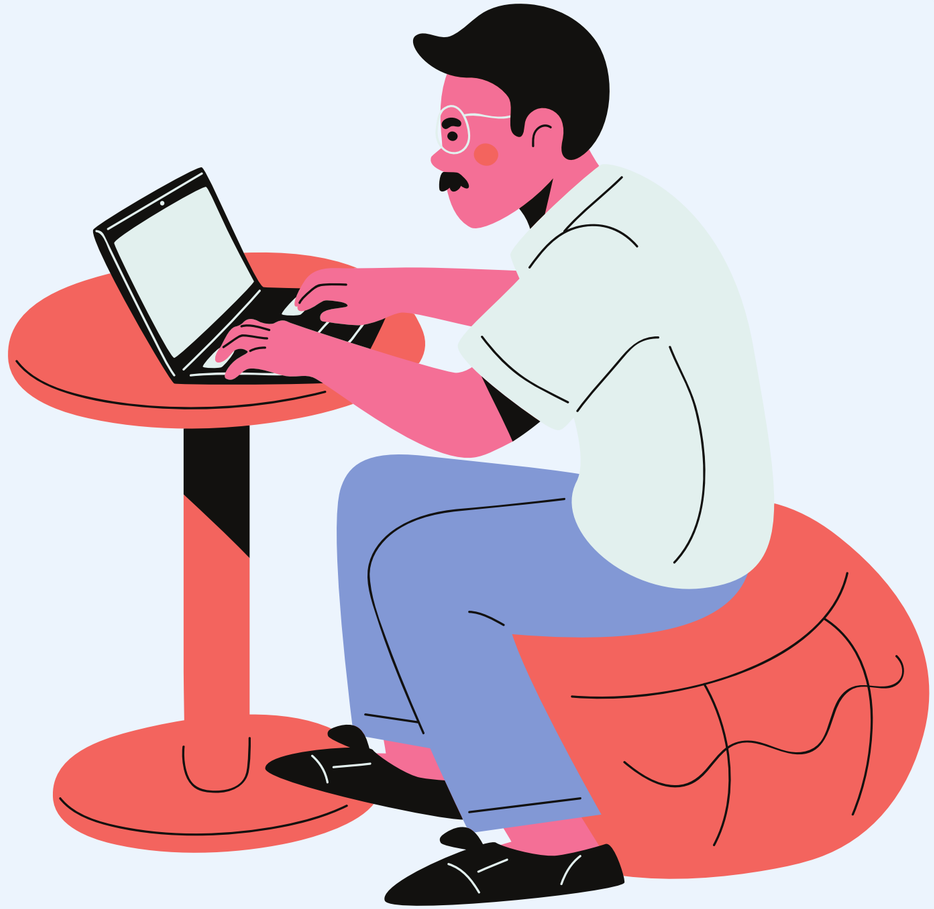
MASYNCTECH
SCHOOL

Many-to-One Relationship

```
Comments Collection                          BlogPosts Collection
└── Comment Document 1                        └── BlogPost Document
    ├── _id: ObjectId("...")                      ├── _id: ObjectId("...")
    ├── text: "Great post!"                       ├── title: "My First Blog"
    └── blogPostId: ObjectId("...")  <---          └── comments: Array of ObjectIds
└── Comment Document 2
    ├── _id: ObjectId("...")
    ├── text: "Thanks for sharing!"
    └── blogPostId: ObjectId("...")
```

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

MANY TO MANY RELATIONSHIPS

mongoose

MONGOOSE DATA RELATIONSHIP

MASYNCTECH
SCHOOL

Many-to-Many Relationship

```
Students Collection                          Courses Collection
└── Student Document 1                       └── Course Document 1
    ├── _id: ObjectId("...")                     ├── _id: ObjectId("...")
    ├── name: "Alice"                            ├── name: "Math 101"
    └── enrolledCourses: Array of ObjectIds ---> enrolledStudents: Array of Obje
         └── ObjectId("...")                          └── ObjectId("...")
                                                      └── ObjectId("...")

└── Student Document 2                       └── Course Document 2
    ├── _id: ObjectId("...")                     ├── _id: ObjectId("...")
    ├── name: "Bob"                              ├── name: "History 101"
    └── enrolledCourses: Array of ObjectIds ---> enrolledStudents: Array of Obje
         └── ObjectId("...")                          └── ObjectId("...")
         └── ObjectId("...")                          └── ObjectId("...")
```

MANY-TO-MANY RELATIONSHIPS

MASYNCTECH
SCHOOL

High-Level Explanation

- Many-to-many relationship: Multiple docs in one collection associate with multiple docs in another.
- Complex compared to one-to-many and many-to-one relations.

Analogue

- Social media platforms like Instagram: Classic many-to-many relationship.
- You follow multiple people, and multiple people follow you.
- Mutual, bidirectional relationship.

Best Practices

- Watch array size due to MongoDB's 16MB limit.
- Complex relationships? Consider a join collection.
- Maintain data consistency and integrity during updates/deletes.

Deep Dive

- Many-to-many relationship in Mongoose: Arrays of ObjectIds in both collections.
- Example: Students with course ObjectIds, Courses with student ObjectIds.
- Allows multiple students in multiple courses.

When to use?

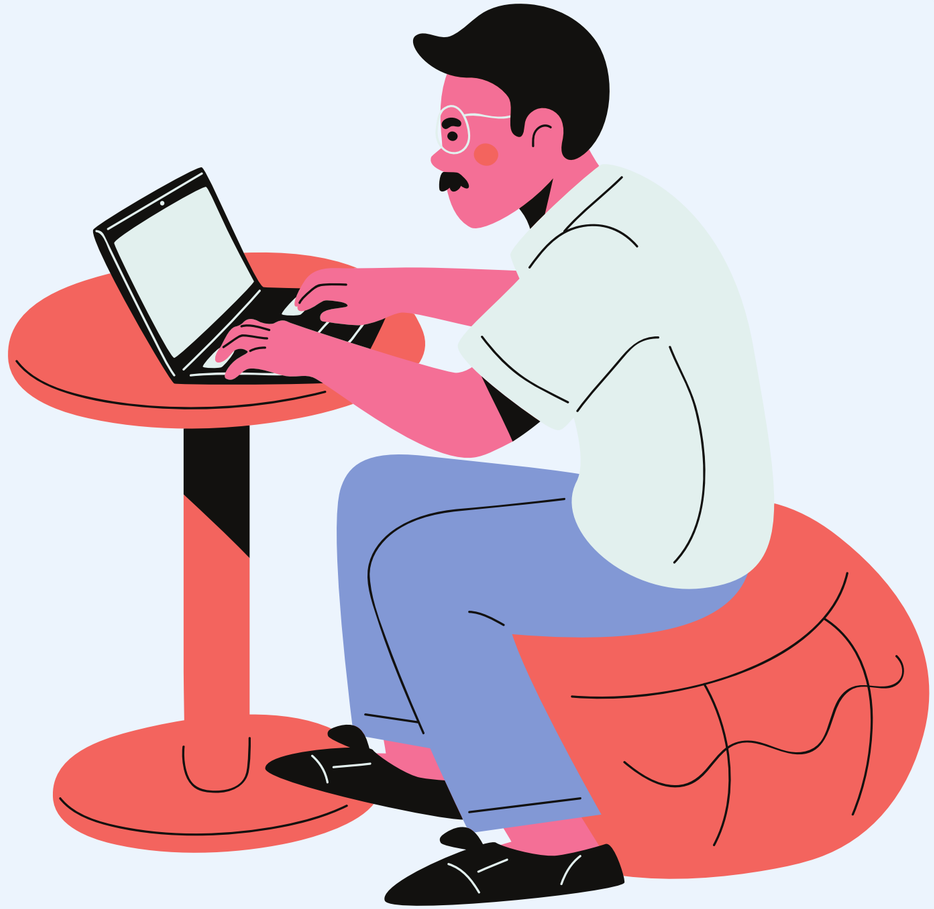
- Use many-to-many relationships when:
  - Complex associations between document sets.
  - Non-hierarchical, non-singular relationships.
  - Frequent need for querying these relationships.

Summary

- Many-to-many relationships: Multiple docs related to multiple docs.
- Implement with arrays of references or separate join collection.

MASYNCTECH
SCHOOL

DEMO TIME



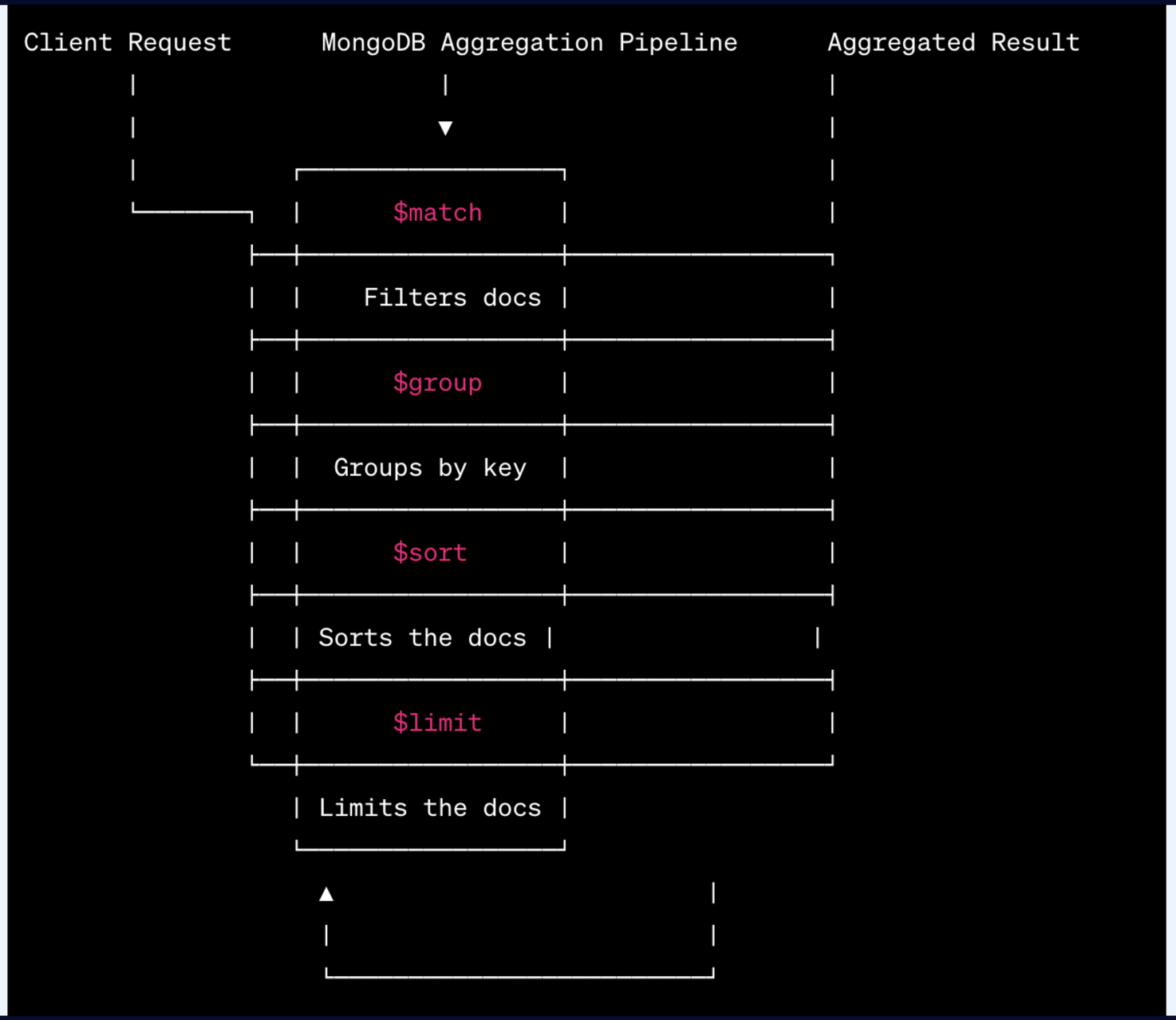
MASYNCTECH
SCHOOL

DATA
AGGREGATION

mongoose

MONGOOSE DATA RELATIONSHIP

DATA AGGREGATION



DATA AGGREGATION

MASYNCTECH SCHOOL

High-Level Explanation

- Data aggregation in MongoDB: Transform data via operations.
- Compute derived data, stats, manipulations.
- Original dataset remains unchanged.

Analogue

- Analogy: Creating a playlist from music collection.
- Aggregation stages:
  1. `$match`: Pick rock songs.
  2. `$group`: Categorize by artist.
  3. `$sort`: Sort by song length.
  4. `$limit`: Select top 10 songs.

Best Practices

- Optimize with indexes for `$match` and `$sort`.
- Control document count per stage for memory.
- Monitor performance for resource-intensive aggregations.

Deep Dive

- MongoDB aggregation pipeline: Complex data transformation.
- Ordered stages process documents to aggregated results.
- Common stages: `$match`, `$group`, `$sort`, `$limit`, etc.
- Mongoose: Use `.aggregate()` method on a model.

When to use?

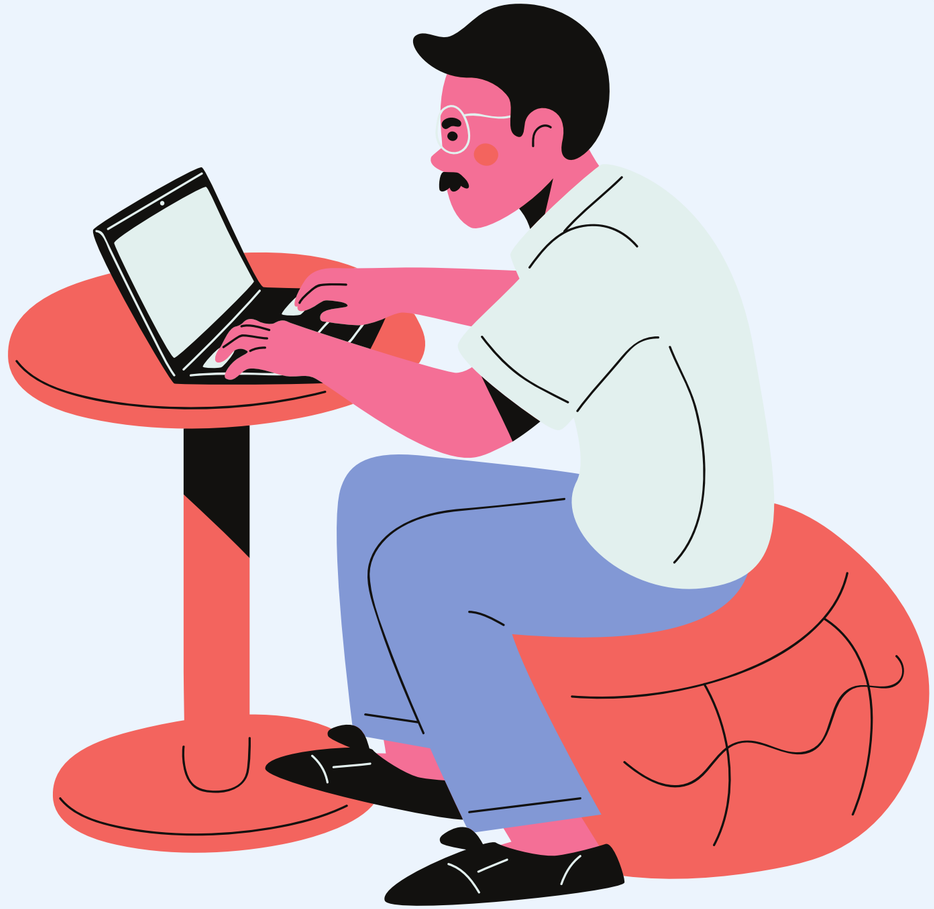
- Use MongoDB aggregation when you:
  - Summarize data for analytics.
  - Transform documents into different forms.
  - Compute derived data or statistics.

Summary

- Data aggregation in MongoDB/Mongoose: Powerful for data manipulation.
- Aggregation pipeline: Sequence of ops for filtering, grouping, sorting, and aggregating.
- Generates aggregated results from documents.

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

MONGOOSE MIDDLEWARES OVERVIEW

mongoose

MONGOOSE MIDDLEWARES

MONGOOSE MIDDLEWARES OVERVIEW

MASYNCTECH SCHOOL

High-Level Explanation

- Mongoose middlewares: Functions in Mongoose document lifecycle.
- Hook into stages like validation, saving, finding.
- Enable custom logic, data modification.

Analogue

- Analogy: Mongoose middlewares as video game checkpoints.
- Check, modify data before proceeding to next stage.
- Ensure conditions are met before advancing.

Best Practices

- Keep them small and focused, performing a single task.
- Handle errors in asynchronous operations.
- Be cautious when modifying queries or data to avoid unintended consequences.

Deep Dive

Mongoose middlewares are categorized into different types:

- **Document Middlewares**: Operate on individual documents (e.g., `save`, `validate`, `remove`).
- **Query Middlewares**: Operate on queries (e.g., `find`, `update`).
- **Aggregate Middlewares**: Work with aggregation pipelines.
- **Model Middlewares**: Apply to model's static functions.

They can be synchronous or asynchronous, registered using `pre` and `post` hooks.

When to use?

- Document middlewares: Single document actions (validation, modification).
- Query middlewares: Query modification, logging.
- Aggregate middlewares: Complex data transformation.
- Model middlewares: Collection-wide actions (e.g., data seeding).

Summary

- Mongoose middlewares: Insert custom logic at data operation stages.
- Apply at document, query, and other levels.
- Essential for effective data management in Mongoose apps.

MASYNCTECH
SCHOOL

DOCUMENT MIDDLEWARES

mongoose

MONGOOSE MIDDLEWARES

DOCUMENT MIDDLEWARES

MASYNCTECH SCHOOL

High-Level Explanation

- Mongoose Document middlewares (hooks): Custom logic pre/post events.
- Hooks: `init`, `validate`, `save`, `remove`.
- Trigger at various stages in document lifecycle.

Analogue

- Analogy: Mongoose Document middlewares as theater crew.
- `init`: Preparing the stage before the show.
- `validate`: Ensuring everything is in place.
- `save`: Managing the action during the performance.
- `remove`: Cleaning up after the show.

Best Practices

- Asynchronous: Employ `next()` for async continuation.
- Error Handling: Manage errors with try-catch or pass to `next()`.
- Immutable Fields: Don't alter immutable fields in `pre('save')` hooks.

Deep Dive

Mongoose Document middlewares provide control and customization of operations:

- `init`: Triggers on document load.
- `validate`: Pre-validation.
- `save`: Pre/post save (before/after).
- `remove`: Pre/post remove (before/after deletion from collection).

When to use?

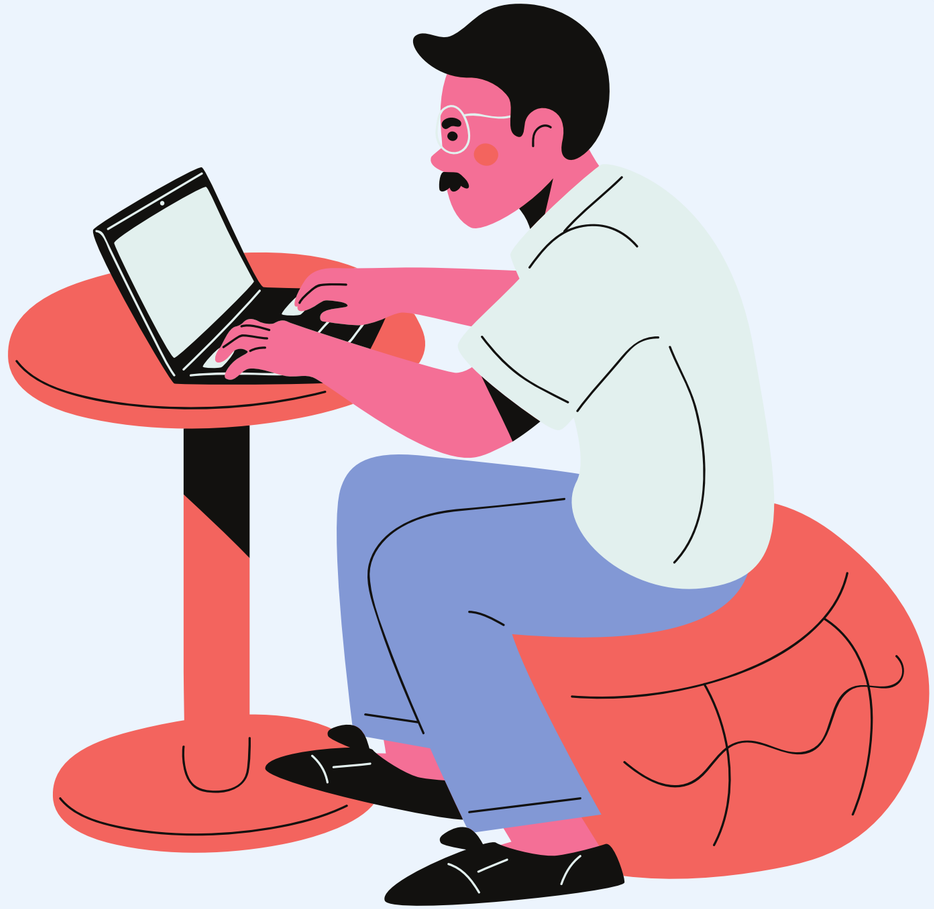
- `init`: Set initial values or conditions.
- `validate`: Enforce complex validation rules.
- `save`: Automate tasks, manipulate data pre/post save.
- `remove`: Perform clean-up, delete associated records.

Summary

- Mongoose Document middlewares: Custom logic at lifecycle stages
- Init, validate, save, remove hooks
- Enhance app robustness and maintainability

MASYNCTECH
SCHOOL

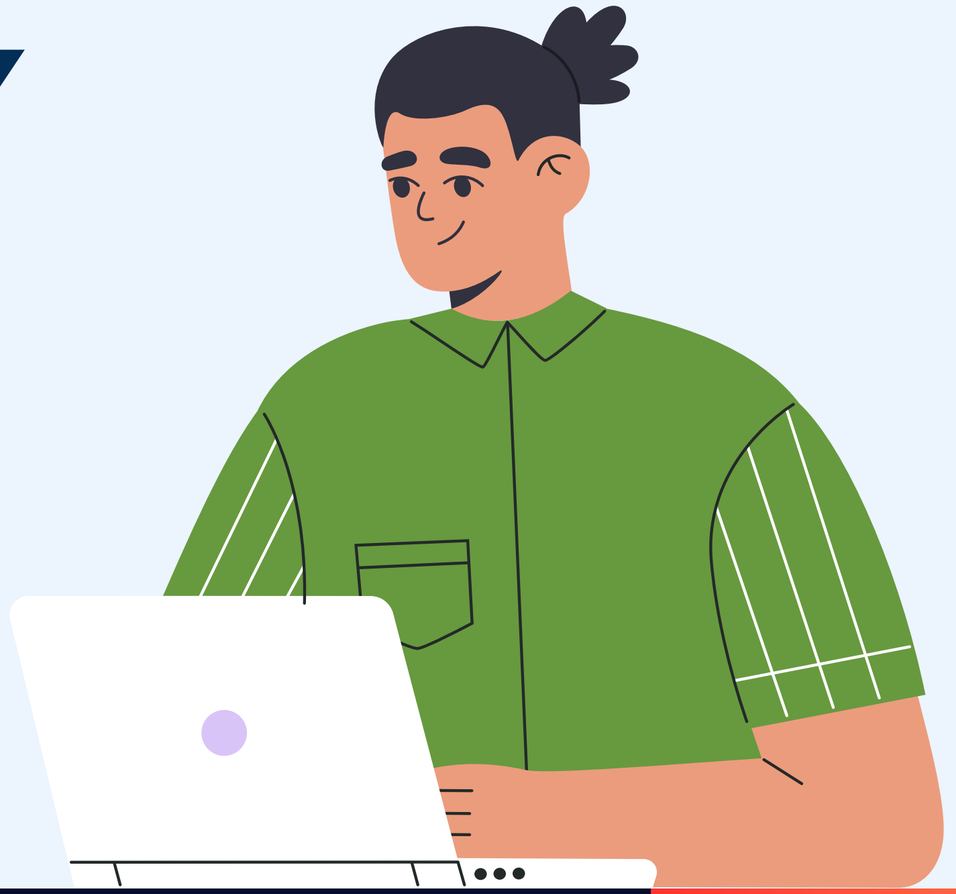
DEMO TIME



MASYNCTECH
SCHOOL

QUERY

MIDDLEWARES



mongoose

MONGOOSE MIDDLEWARES

QUERY MIDDLEWARES

MASYNCTECH SCHOOL

High-Level Explanation

- Mongoose Document middlewares (hooks): Custom logic pre/post events.
- Hooks: `init`, `validate`, `save`, `remove`.
- Trigger at various stages in document lifecycle.

Deep Dive

- Query middleware uses: Data normalization, validation, caching
- Operate on query objects, not documents
- Two types: "pre" (before) and "post" (after)
- Modify query with `pre('find')`, process results with `post('find')`

Analogue

- Analogy: Google Search and Mongoose query middlewares
- Pre-middleware: Filter out specific terms (like Google)
- Post-middleware: Count results for analytics (like Google)
- Mongoose middlewares for database operations

When to use?

Common use cases for Mongoose query middlewares:

1. Data Validation: Ensure queries meet specific criteria.
2. Authorization: Check permissions before executing a query.
3. Caching: Implement caching for query performance.
4. Logging: Track database activities for auditing purposes.

Best Practices

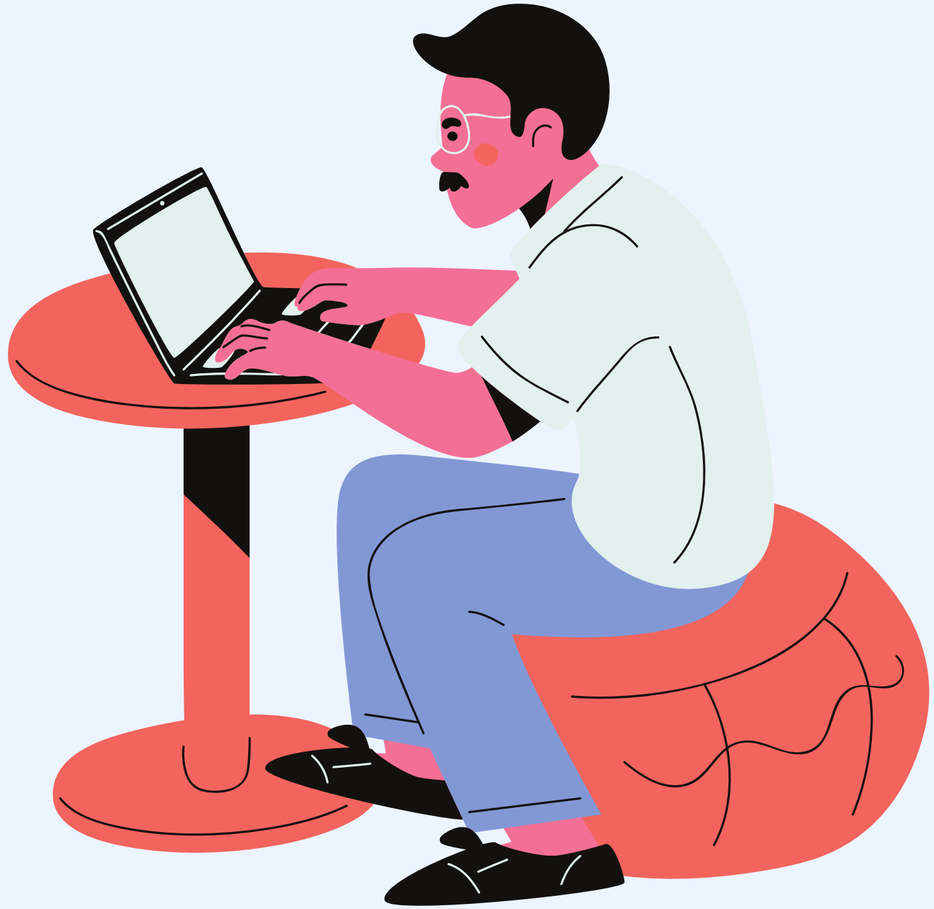
1. Utilize `async/await` for async operations in middlewares.
2. Keep middlewares lightweight; avoid heavy computations for performance.
3. Carefully chain multiple middlewares to maintain code readability.
4. Always call `next()` in pre-middlewares to continue execution.

Summary

- Mongoose query middlewares: Hooks into query lifecycle
- Pre/post-processing for validation, logging, caching
- Powerful for augmenting database operation behavior

MASYNCTECH
SCHOOL

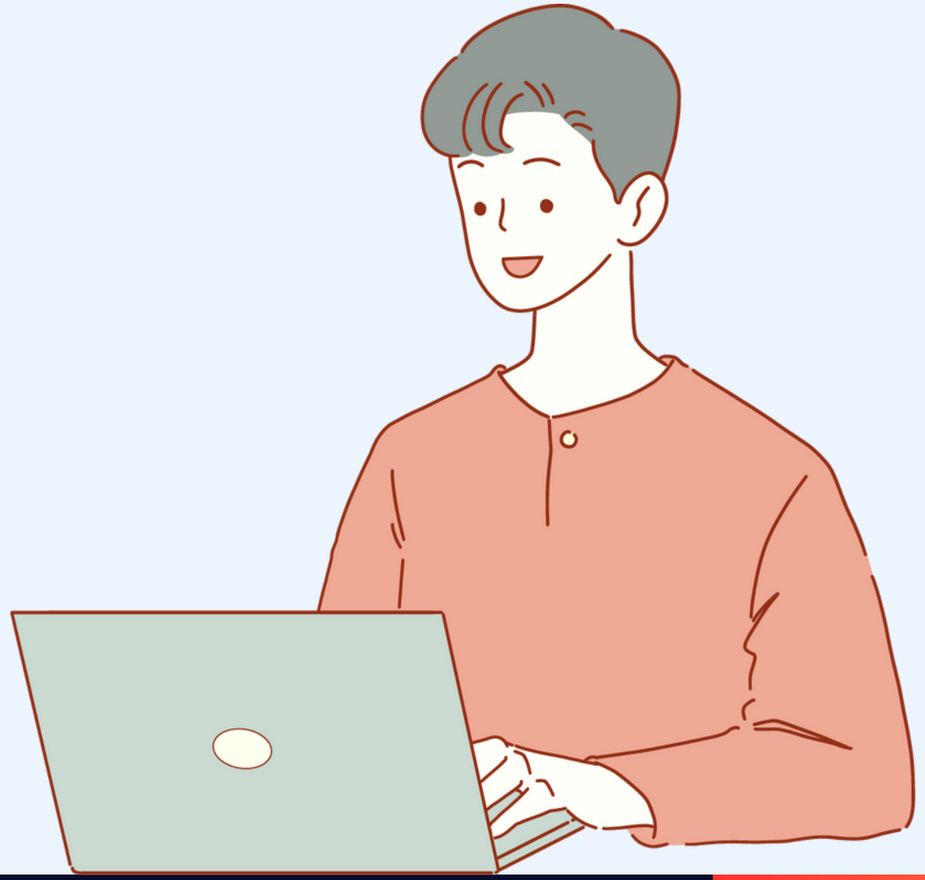
DEMO TIME



MASYNCTECH
SCHOOL

AGGREGATE

MIDDLEWARES



mongoose

MONGOOSE MIDDLEWARES

AGGREGATE MIDDLEWARES

MASYNCTECH SCHOOL

High-Level Explanation

- Mongoose's Aggregate Middlewares: Pre and post hooks
- Intervene in aggregation pipeline execution
- Modify pipeline or process aggregated data

Analogue

- Analogy: Aggregation pipeline as a party playlist
- `pre('aggregate')`: Adjust before the party, add excitement.
- `post('aggregate')`: Review after the party, note dance hits.

Best Practices

- Pre-aggregate: Avoid unintentional pipeline changes.
- Post-aggregate: Watch for increased computational complexity.
- Use both judiciously to maintain code readability.

Deep Dive

- Mongoose schema allows defining middlewares for various operations.
- `pre('aggregate')`: Executed before aggregation, modifies pipeline.
- `post('aggregate')`: Executed after aggregation, post-processing.

When to use?

- Pre-aggregate: Enforce security policies or conditions.
  - e.g., User can only aggregate their data.

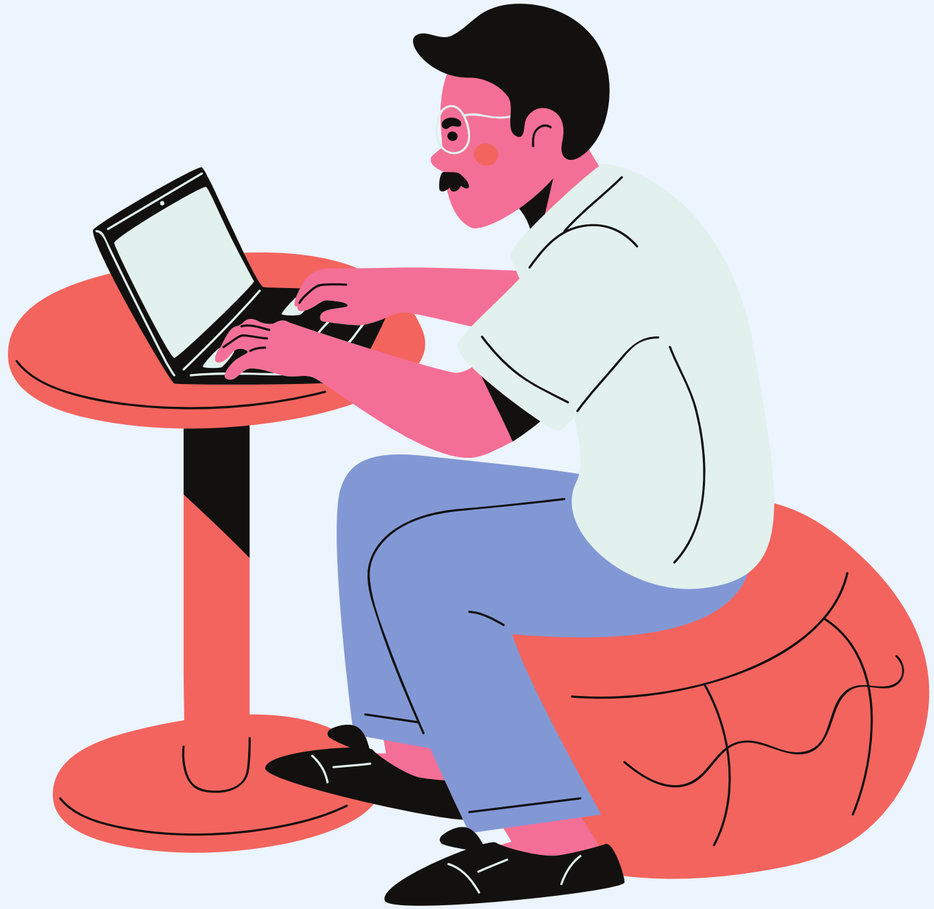
- Post-aggregate: Log for auditing, transform aggregated data.
  - e.g., Activity logging or additional data processing.

Summary

- Mongoose Aggregate Middlewares: Hooks for pre/post aggregation.
- Useful for dynamic pipeline modification, rule enforcement, data processing.
- Use with care for code readability and performance.

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

MODEL
MIDDLEWARES

mongoose

MONGOOSE MIDDLEWARES

MODEL MIDDLEWARES

MASYNCTECH SCHOOL

High-Level Explanation

- Mongoose model middlewares: Pre and post-hooks for model methods.
- **Pre-hooks**: Triggered before method execution, e.g., `pre('save')`.
- **Post-hooks**: Run after method execution, handle operation results, e.g., `post('save')`.

Analogue

- Analogy: Blog post on a website and Mongoose model middlewares
- `pre-save` hook: Spell-check or tag addition before "Publish."
- `post-save` hook: Share on social media after publication.

Best Practices

- **Ordering**: Sequence of hook definitions matters.
- **Asynchronous Operations**: Handle async operations carefully.
- **Error Handling**: Ensure error handling, especially in pre-hooks for data validation.

Deep Dive

- Mongoose schema allows defining middlewares for various operations.
- `pre('aggregate')`: Executed before aggregation, modifies pipeline.
- `post('aggregate')`: Executed after aggregation, post-processing.

When to use?

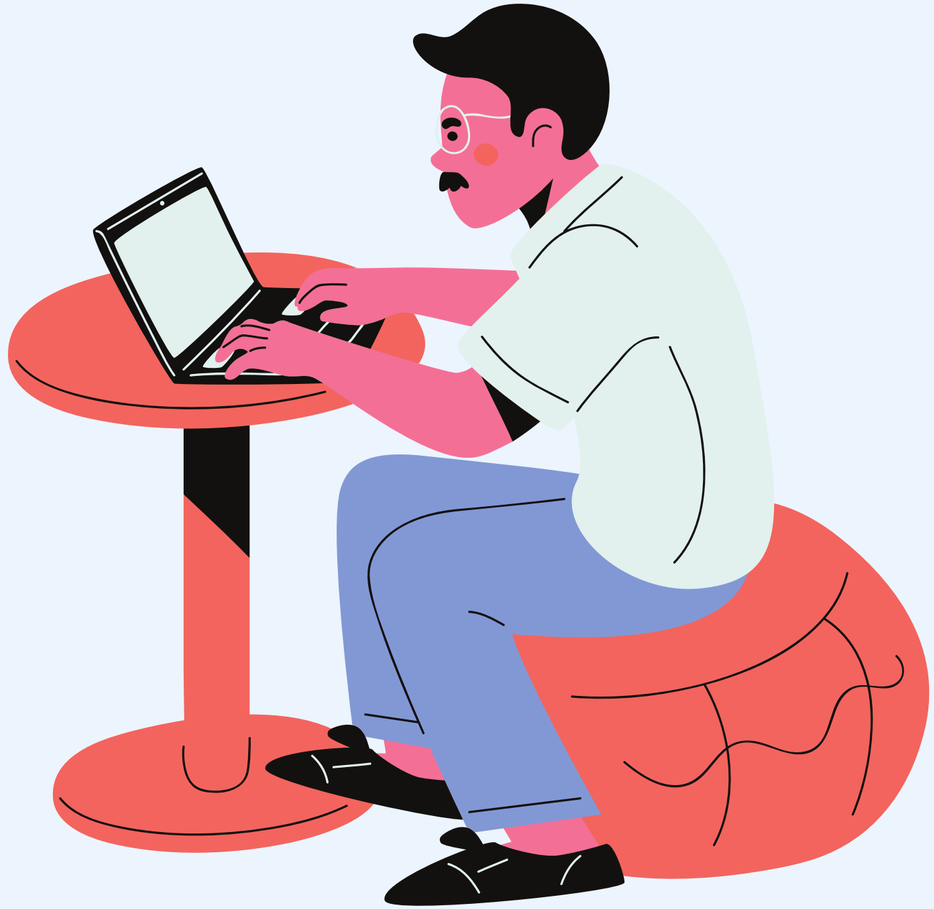
- Pre-hooks: Validation, data transformation, business logic.
- Post-hooks: Notifications, logging, post-operation actions.

Summary

- Mongoose Model Middlewares: Gatekeepers and observers.
- Useful for validation, transformation, logging, notifications.
- Implement with care for a clean, efficient codebase.

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

MIDDLEWARES
STAGES

mongoose

MONGOOSE MIDDLEWARES

MIDDLEWARE STAGES

MASYNCTECH SCHOOL

High-Level Explanation

- `**init**`: Data manipulation after fetch but before use.
- `**validate**`: Last chance data validation before save.
- `**save**`: Pre/post save logic, widely used.
- `**remove**`: Pre-document removal logic, e.g., cascading deletes, notifications.

Analogue

- Analogy for Mongoose middleware stages:
  - `init`: Starting a new game level, stage is set.
  - `validate`: Checking inventory before proceeding.
  - `save`: Reaching a checkpoint, progress is saved.
  - `remove`: Exiting with an option to save friend's scores.

Best Practices

- Asynchrony: Call `next()` after async operations complete.
- Error Handling: Pass errors to `next()` for proper handling.

Deep Dive

- Mongoose schema allows defining middlewares for various operations.
- `pre('aggregate')`: Executed before aggregation, modifies pipeline.
- `post('aggregate')`: Executed after aggregation, post-processing.

When to use?

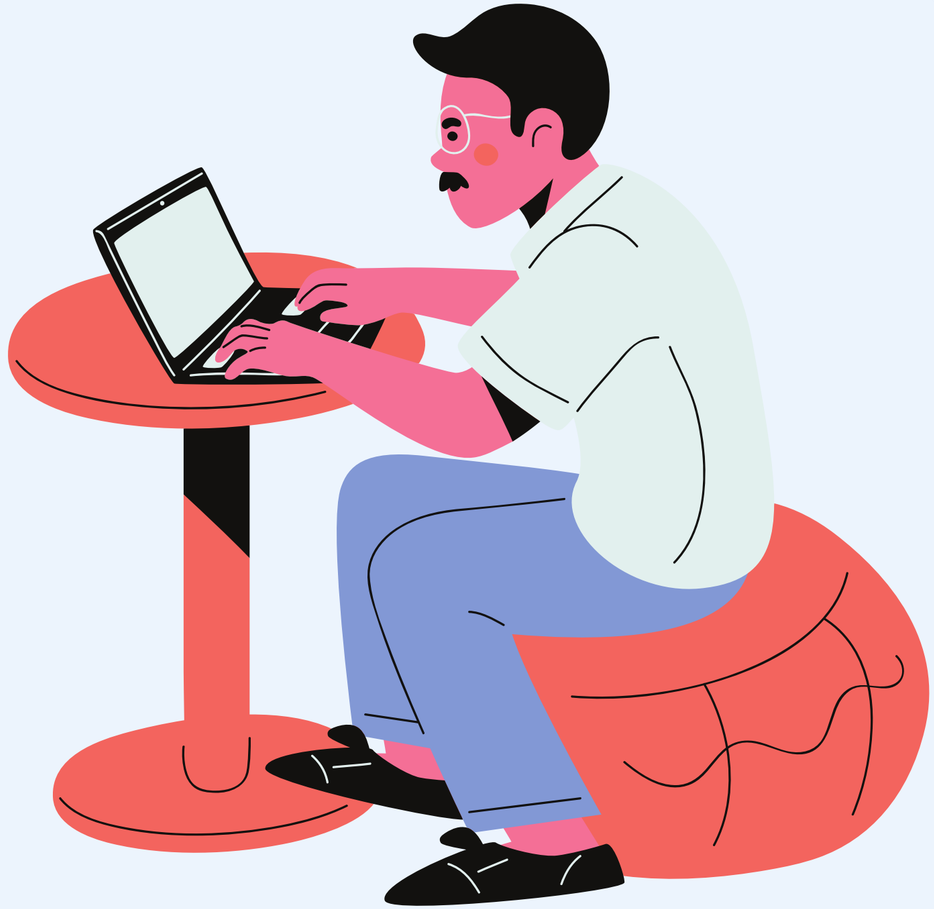
- `init`: Initial state setup, document transformations.
- `validate`: Custom validation rules.
- `save`: Timestamps, encryption, business logic.
- `remove`: Cleanup tasks, associated record deletion.

Summary

- Mongoose middleware stages offer precise document lifecycle control.
- Improve data integrity and application logic.
- Follow best practices: error handling, async operation management.

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

POPULATING REFERENCES AUTOMATICALLY

mongoose

MONGOOSE MIDDLEWARES

POPULATING REFERENCES AUTOMATICALLY

MASYNCTECH
SCHOOL

High-Level Explanation

- Mongoose Query Middleware: Populate references automatically.
- Fill referenced documents in schema before returning data to app.

Deep Dive

- Mongoose Schema: Define fields as ObjectIds referencing other collections.
- Normally, use `.populate()` method to fill references.
- Query Middleware automates this, populating references on query.

Analogue

- Analogy: Auto-complete for search bar.
- Normally, separate button for full details.
- With Query Middleware, related data auto-populated.

When to use?

- Frequent queries needing populated references.
- When client apps expect fully formed objects, server-side population.

Summary

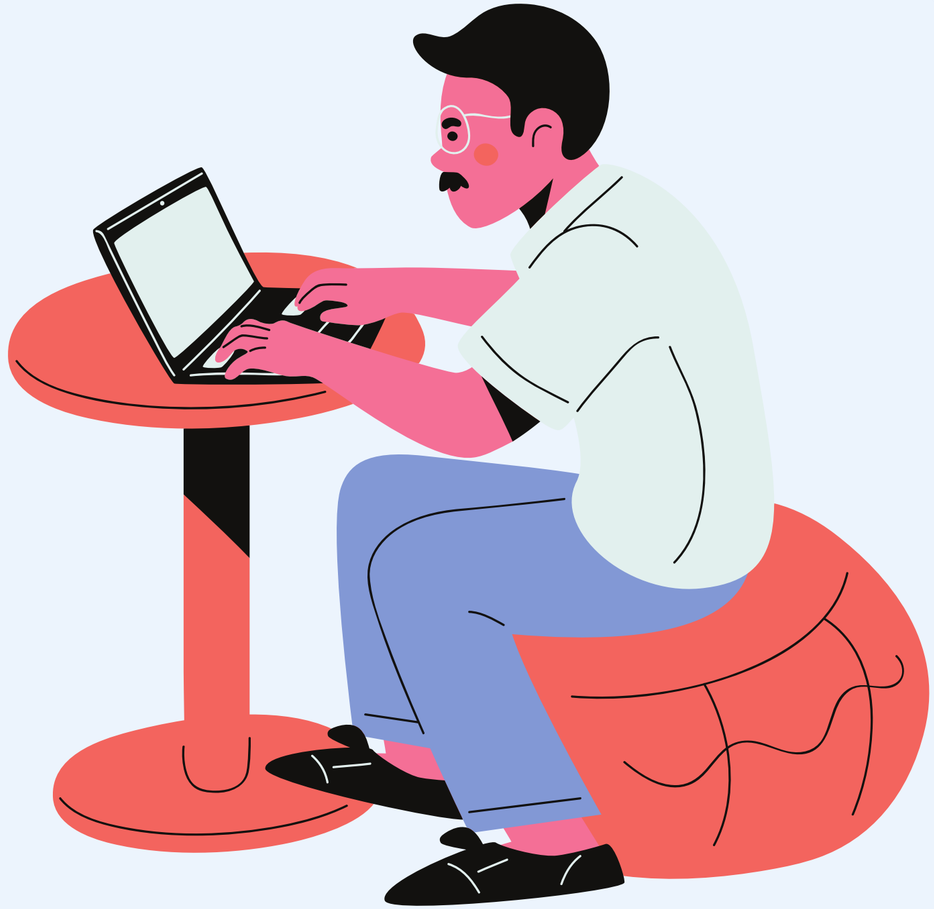
- Auto-populating references: Automatically fill in data.
- Resolves references between collections.
- Simplifies client-side code.
- Query Middleware automates and streamlines this process.

Best Practices

- Watch out for performance impact with extensive auto-population.
- Avoid overuse, especially for occasional reference needs.

MASYNCTECH
SCHOOL

DEMO TIME



MASYNCTECH
SCHOOL

THANK YOU