LECTURE 23

Object Relational Mapping (ORM / ODM)

Trading raw driver calls for objects and methods — Mongoose, schemas, and the queries that write themselves.

CSC336 Web Technologies

OVERVIEW

In This Lecture

1. What "object-relational mapping" means and why ORMs/ODMs exist
2. Defining models, schemas, fields, and data types with Mongoose
3. CRUD through an ORM/ODM: queries, filtering, projection, pagination
4. Associations: population in Mongoose, joins/associations in SQL ORMs
5. Migrations, seeding, and the trade-offs of using an ORM/ODM

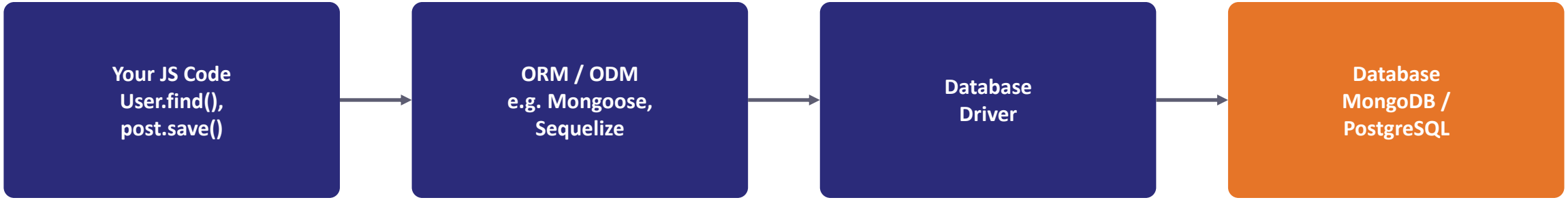
THE BIG IDEA

What Is an ORM/ODM?

1. ORM (Object-Relational Mapping): interact with a relational database using objects and methods instead of raw SQL
2. ODM (Object-Document Mapper): the same idea for document databases like MongoDB
3. Write User.find(...) instead of hand-rolled driver calls
4. Mongoose is the most widely used ODM for MongoDB in Node.js; Sequelize and Prisma are popular SQL ORMs

THE BIG IDEA

Where the ORM/ODM Sits



The ORM/ODM sits between your application code and the raw database driver, translating object method calls into the queries the driver actually sends.

THE BIG IDEA

Why Bother With the Extra Layer?

LESS CODE, MORE STRUCTURE

- Common operations (find, save, update, delete) become one-line calls
- Data validation built into the model definition, checked automatically
- A schema structure even for MongoDB, which enforces none itself

RELATIONSHIPS & PORTABILITY

- Relationships are easier to define and query (population, associations)
- A consistent API means switching databases needs smaller code changes

TERMINOLOGY

"ORM" vs. "ODM"

NOTE

"Object-relational mapping" comes from the relational-database world. When people say "ORM" loosely today, they often mean any object-mapping tool, including document-database ODMs like Mongoose. This course uses "ORM/ODM" to be precise, but expect to see "ORM" used for both.

Schema In, Model Out

1. A schema describes what fields a document should have and what type each field is
2. You compile a schema into a model — the object you actually use to query and save data
3. npm install mongoose to get started

Connecting Mongoose

```
const mongoose = require("mongoose");
require("dotenv").config();

mongoose.connect(process.env.MONGO_URI)
  .then(() => console.log("Connected to MongoDB via Mongoose"))
  .catch((err) => console.error("Connection error:", err));
```

Defining a Schema and Model

```
const { Schema, model } = require("mongoose");

const userSchema = new Schema({
  name: { type: String, required: true, trim: true },
  email: { type: String, required: true, unique: true, lowercase: true },
  age: { type: Number, min: 0, max: 120 },
  role: { type: String, enum: ["student", "instructor", "admin"], default: "student" },
  createdAt: { type: Date, default: Date.now },
});

const User = model("User", userSchema);
```

What Each Field Defines

1. type — the data type (String, Number, Boolean, Date, Array, Schema.Types.ObjectId, ...)
2. Validation rules — required, min/max, enum (a fixed list of allowed values), or a custom validate function
3. Defaults — a value used automatically if none is provided

Validation Runs Automatically

TIP

Mongoose checks these rules every time you save() or create() a document, and throws a ValidationError if something doesn't match — free validation, no if statements needed for every field.

COMPARISON

The Same Idea in Sequelize (SQL ORM)

```
const { DataTypes } = require("sequelize");

const User = sequelize.define("User", {
  name: { type: DataTypes.STRING, allowNull: false },
  email: { type: DataTypes.STRING, allowNull: false, unique: true },
  age: { type: DataTypes.INTEGER },
});
```

The concepts map closely: a Mongoose "schema" is roughly Sequelize's "model definition" — both compile down to an object you use to run queries.

Creating a Document

```
const newUser = await User.create({
  name: "Bilal",
  email: "bilal@email.com",
  age: 20,
});
```

Queries and Filtering

```
// Find all users matching a filter
const students = await User.find({ role: "student" });

// Find one user
const user = await User.findOne({ email: "bilal@email.com" });

// Find by ID (Mongoose's built-in shortcut for _id)
const oneUser = await User.findById("64f1a2b3c4d5e6f7a8b9c0d1");

// Filtering with operators
const adults = await User.find({ age: { $gte: 18 } });
```

Projection: Choosing Fields

```
// Only return name and email
const emailsOnly = await User.find({}, "name email");
// or: User.find().select("name email")
```

Projection asks the database to return only specific fields — good for performance, and for avoiding sending sensitive fields (like a hashed password) to the client by accident.

Pagination

```
async function getUsersPage(pageNumber, pageSize = 10) {
  const skip = (pageNumber - 1) * pageSize;
  const users = await User.find()
    .sort({ createdAt: -1 })
    .skip(skip)
    .limit(pageSize);
  const total = await User.countDocuments();
  return { users, total, page: pageNumber, totalPages: Math.ceil(total / pageSize) };
}
```

.skip() jumps over already-seen documents; .limit() caps how many come back — together they fetch page 2, page 3, and so on.

Updating Documents

```
// Update one document and get the updated version back
const updated = await User.findByIdAndUpdate(
  "64f1a2b3c4d5e6f7a8b9c0d1",
  { age: 23 },
  { new: true, runValidators: true }
);

// Update many at once
await User.updateMany({ role: "student" }, { $set: { active: true } });
```

Validators Are Opt-In on Update

WARNING

By default, findByIdAndUpdate does NOT run your schema's validation rules. Always pass { runValidators: true } if you want Mongoose to check required or enum fields during an update, not just during creation.

Deleting Documents

```
await User.findByIdAndDelete("64f1a2b3c4d5e6f7a8b9c0d1");
await User.deleteMany({ role: "guest" });
```

Population in Mongoose

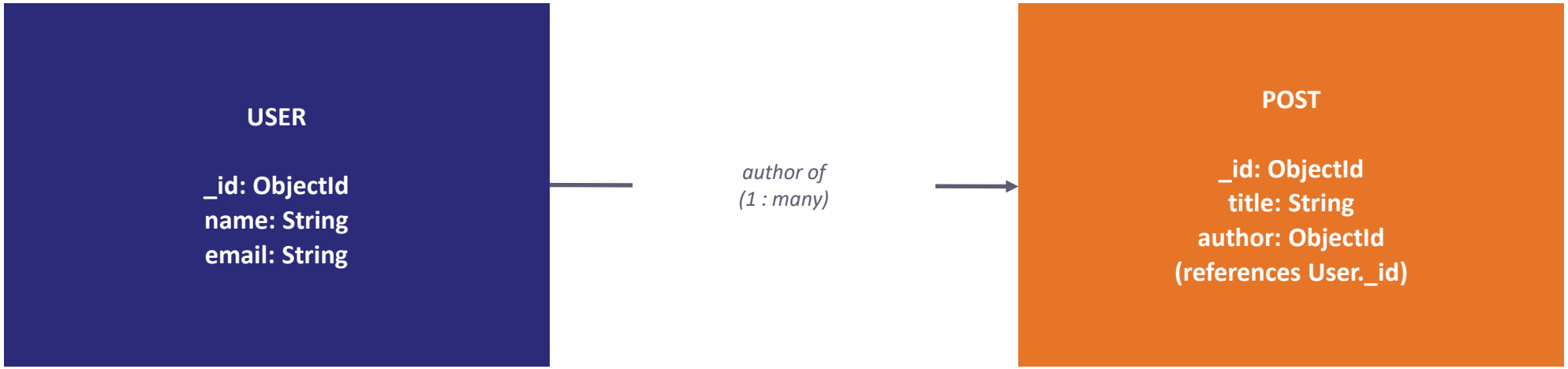
1. MongoDB documents can embed related data directly — but a Post shouldn't embed a full copy of its author
2. Instead, reference another document by its _id, then populate that reference to pull in the full related document
3. Conceptually similar to a SQL JOIN, though implemented as a separate query behind the scenes

Referencing and Populating

```
const postSchema = new Schema({
  title: { type: String, required: true },
  content: String,
  author: { type: Schema.Types.ObjectId, ref: "User" },
});
const Post = model("Post", postSchema);

// Fetch the post WITH the full author document
const post = await Post.findOne({ title: "Learning Mongoose" })
  .populate("author");
console.log(post.author.name); // "Bilal"
```

A User Referenced by Many Posts



post.author stores just a User's ObjectId. .populate("author") runs a second query and swaps that ID for the full User document.

The SQL Equivalent: Associations

NOTE

In a SQL ORM like Sequelize, the equivalent idea is called an association (User.hasMany(Post), Post.belongsTo(User)), and behind the scenes it generates actual SQL JOIN queries.

Migrations

1. A migration is a versioned, scripted change to your database's structure (adding a column, renaming a table)
2. Migrations can be applied — and ideally reversed — so every environment stays in sync over time
3. Central to SQL ORMs like Sequelize; MongoDB's flexible documents mean Mongoose doesn't require them formally
4. Larger MongoDB projects often still use a dedicated tool (like migrate-mongo) for one-off schema-evolution scripts

Running a Sequelize Migration

```
npx sequelize-cli migration:generate --name add-age-to-users
npx sequelize-cli db:migrate
```

Seeding: Loading Sample Data

```
// seed.js
async function seed() {
  await mongoose.connect(process.env.MONGO_URI);
  await User.deleteMany({}); // clear existing data
  await User.insertMany([
    { name: "Ayesha", email: "ayesha@email.com", role: "admin" },
    { name: "Bilal", email: "bilal@email.com", role: "student" },
  ]);
  console.log("Database seeded!");
  await mongoose.disconnect();
}
seed();
```

TRADE-OFFS

Convenience vs. Control

Benefit	Cost
Less boilerplate, faster development	An extra layer to learn, on top of the database itself
Built-in validation and structure	Generated queries can be less efficient for complex cases
Easier to reason about relationships	"Magic" behavior can hide what's really happening
Consistent patterns across a codebase	Sometimes you still need raw queries the ORM doesn't expose

TRADE-OFFS

You Don't Have to Pick One Approach

TIP

Most ORMs/ODMs, including Mongoose, let you drop to raw queries when you need to (.aggregate() for MongoDB's pipeline, for example). Use the ORM for everyday CRUD and drop to raw queries for the rare complex case.

KEY TAKEAWAYS

Lecture 23 in Six Points

1. An ORM (relational) or ODM (document, e.g. Mongoose) maps your database's data onto objects and methods, cutting repetitive query-writing.
2. Mongoose schemas define fields, data types, validation rules, and defaults; a model compiled from a schema is what you actually query with.
3. CRUD through Mongoose uses create, find, findByIdAndUpdate, and findByIdAndDelete, plus filtering, projection, and pagination.
4. Population (.populate()) resolves a referenced ObjectId into the full related document — Mongoose's answer to SQL joins.
5. Migrations version your database structure over time (central to SQL ORMs like Sequelize); seeding loads initial or sample data.
6. ORMs/ODMs trade some control and query efficiency for convenience, validation, and consistency — and you can still write raw queries when needed.