Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar

Chapter 7

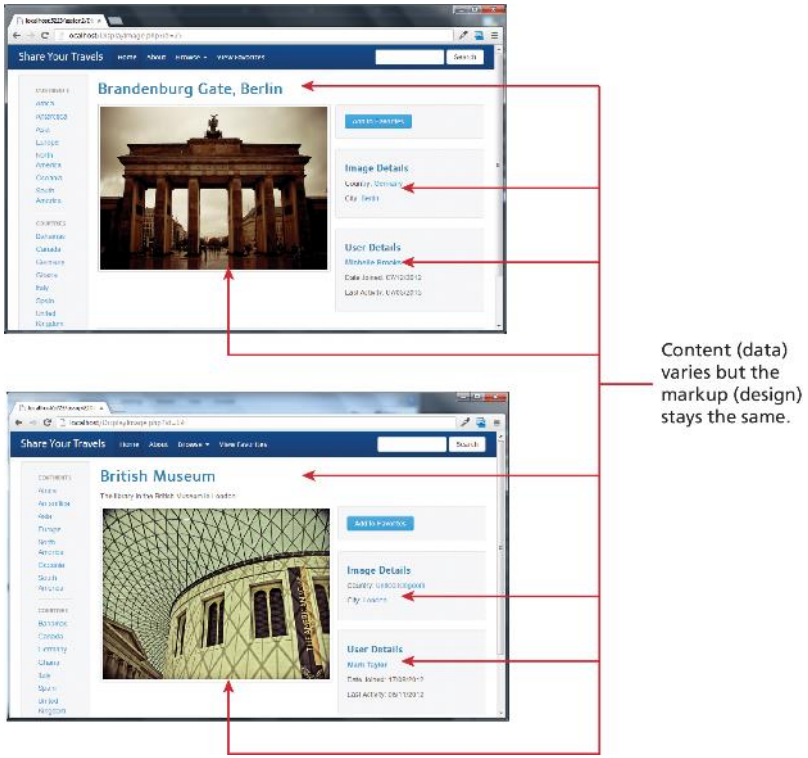
Working with Databases

Part 1

The Role of Databases in Web Development

Databases provide a way to implement an important software design principle: separate static (doesn't change) content from dynamic content.

On the web the visual appearance (i.e., the HTML and CSS) is static, while the data content is dynamic (changes).



Databases and Web Development

To work with data, we can use different **relational** DBMS like **SQLite** or **MySQL**, **PostgreSQL, Oracle Database**, **IBM DB2**, and **Microsoft SQL Server**.

In addition to relational database systems, there are **non-relational** models for database systems that will be used in this course. These systems are usually categorized with the term **NoSQL** and includes systems such as **Cassandra, Firebase** and **MongoDB.**

NoSQL Databases

NoSQL (which stands for Not-only-SQL) is a category of database software that describes a style of databases that doesn't use the relational table model of normal SQL databases.

NoSQL databases rely on a different set of ideas for data modeling that put **fast retrieval** ahead of other considerations like **consistency**.

Systems like DynamoDB, Firebase, and MongoDB now power thousands of sites including household names like eBay, Forbes, mckinsey, ericsson, and others.

Why (and Why Not) Choose NoSQL?

NoSQL systems handle huge datasets better than relational systems. But they aren't the best answer for all scenarios. SQL databases use schemas for a very good reason: they ensure **data consistency and data integrity**.

Definitions:

Data consistency: The guarantee that database constraints are not violated when executing transactions.

Data integrity: The guarantee of all data constraints (primary and foreign keys, data types, etc…).

Key-Value Stores

Key-value stores alone are quite straightforward in that every value, whether an integer, string, or other data structure, has an associated key (i.e., they are analogous to Maps)

Here, every value has a key. This allows fast retrieval through means such as a hash function, and precludes the need for indexes on multiple fields as is the case with SQL.

Examples: DBM, Berkeley DB.

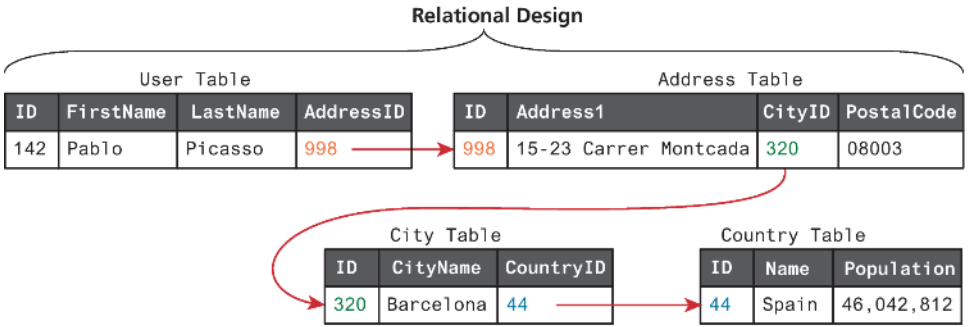
Key	Value
Customer.Name	"Randy"
Price	200.00
ShippingAddress	"4825 Mount Royal Gate SW"
Countries	"Canada","France","Germany","United States"

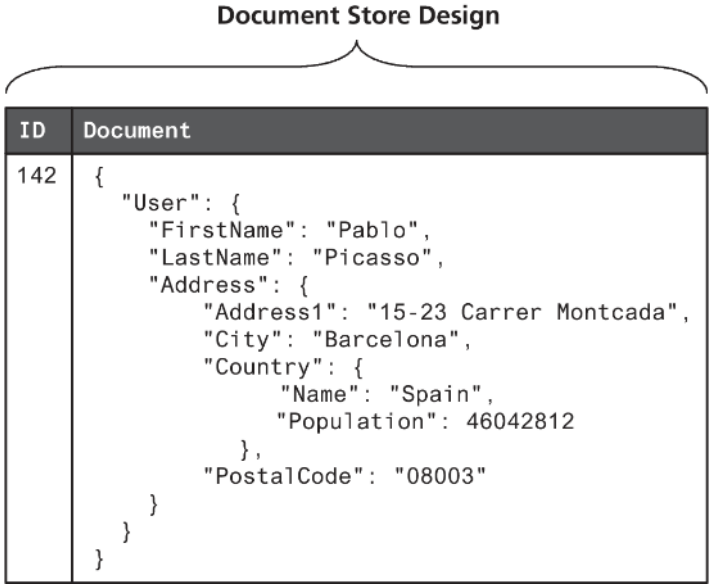
Document Store

Document Stores (also called document-oriented databases) associate keys with values, but unlike key-value stores, they call that value a document.

- A document can be a binary file like a .doc or .pdf or a semi-structured XML or JSON document.
- Most NoSQL systems are of this type. MongoDB, AWS DynamoDB, Google FireBase, and Cloud Datastore are popular examples.

Relational data versus document store data

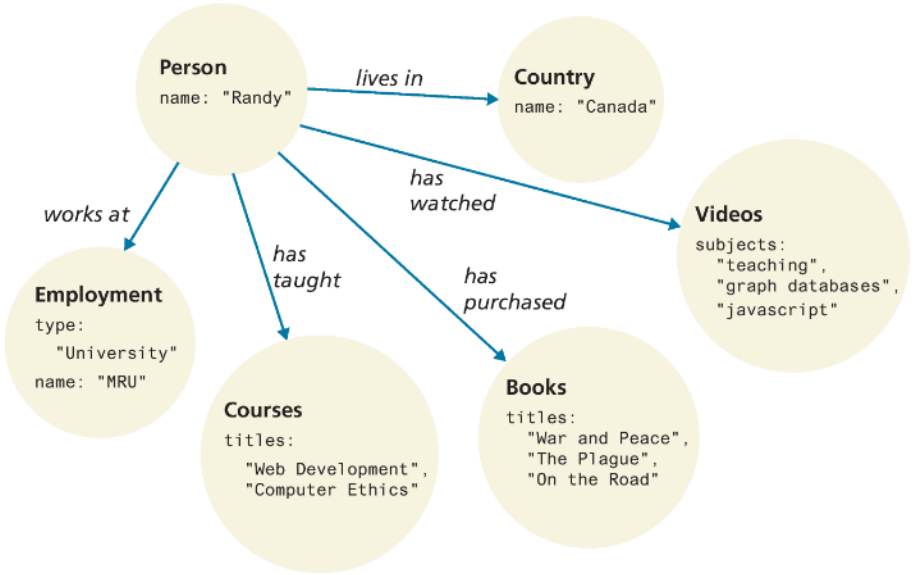




Graph Stores

In a **Graph Store** system (often simply called graph databases), data is represented as a network or graph of entities and their relationships.

Some examples of graph databases include Neo4j, OrientDB, and RedisGraph.



Working with MongoDB in Node

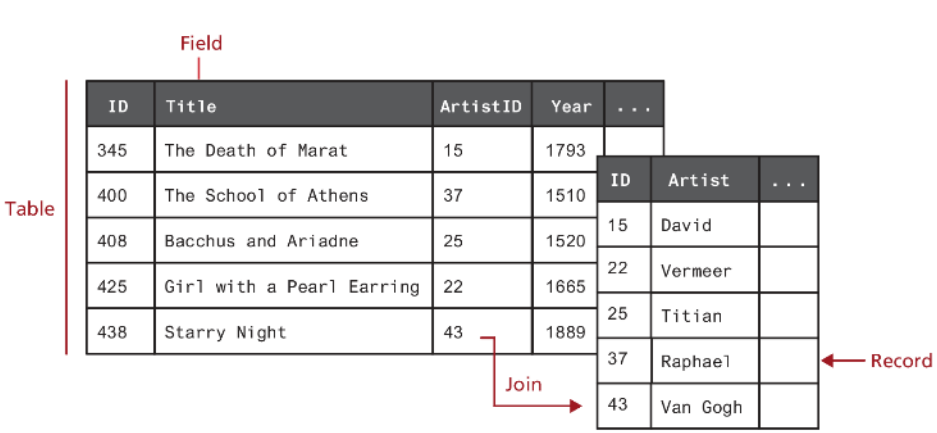
MongoDB is an open-source, NoSQL, document-oriented database. It can be used with any backend technology (JEE, PHP, etc.), it is much more commonly used with Node

You simply package your data as a JSON object, give it to MongoDB, and it stores this object or document as a binary JavaScript object (BSON).

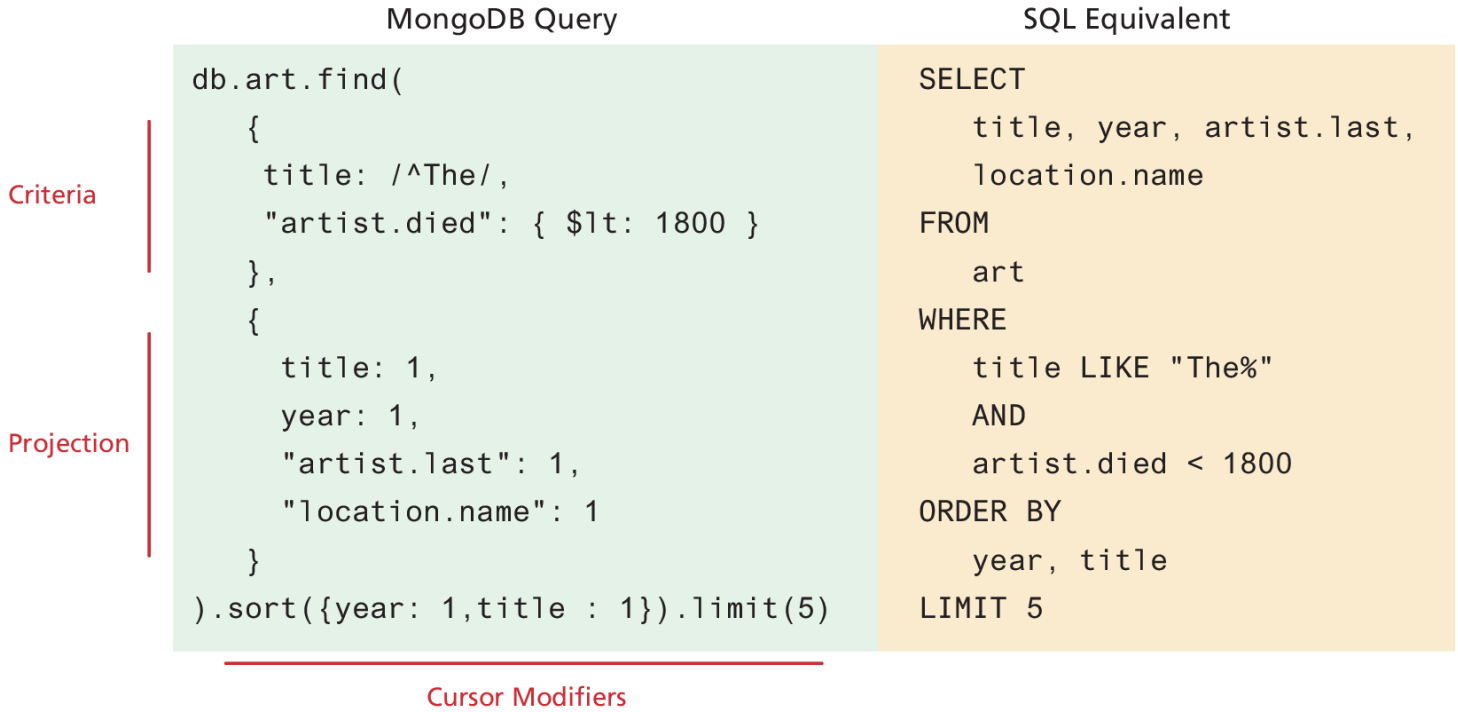
MongoDB does not support: ~~transactions (from 4.0),~~ joins (nested documents instead).

The ability to run on multiple servers means MongoDB can handle large datasets: **replication => redundancy + high availability**

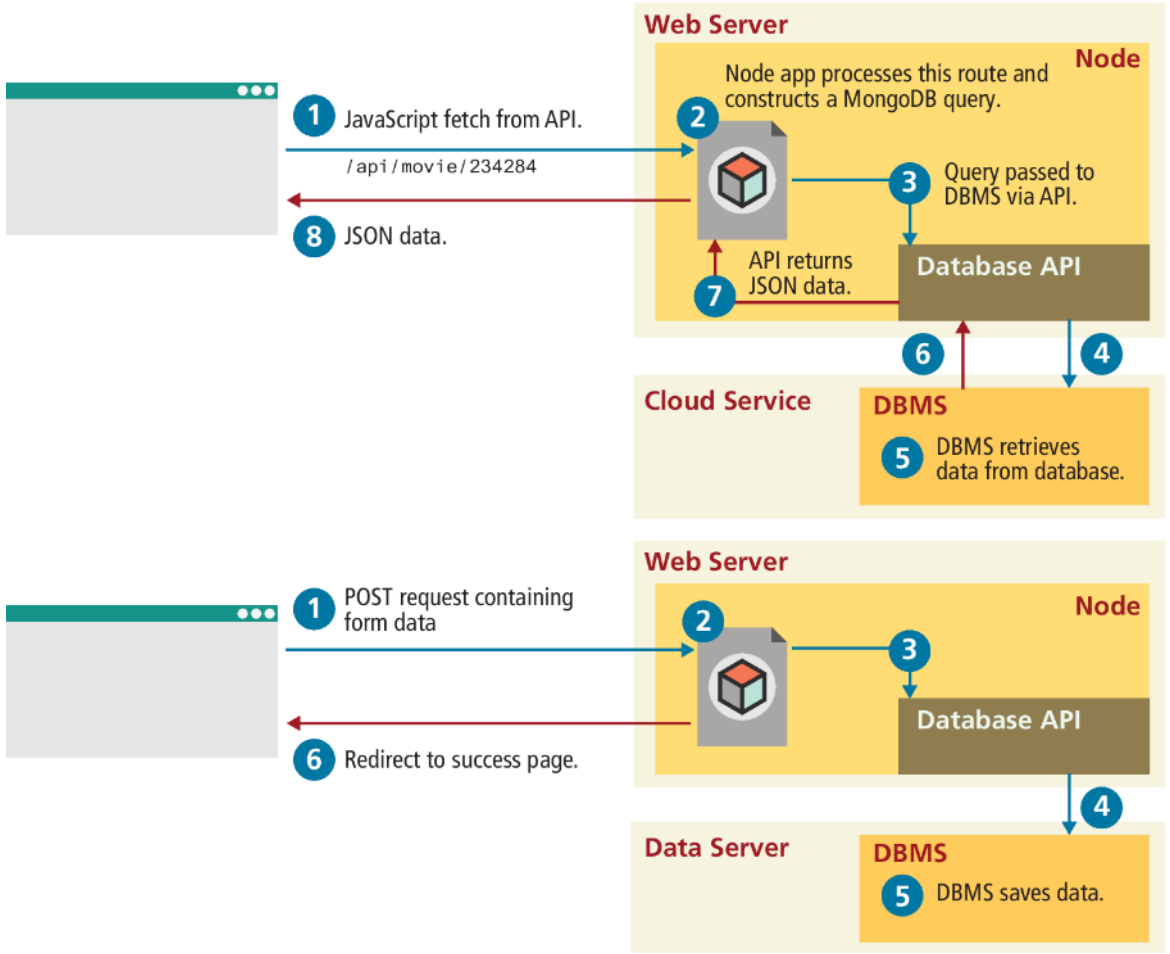
Comparing relational databases to the MongoDB data model



Comparing a MongoDB query to an SQL query



How websites use databases



Designing Data Access

Database details such as connection strings and table and field names are examples of externalities. These details tend to change over the life of a web application.

Initially, the database for our website might be a MongoDB database on our development machine; later it might change to the cloud (Atlas) and even later, to our own server on a cloud server (on digital ocean for example). Ideally, with each change in our database infrastructure, we would have to change very little in our code base.

One simple step might be to extract the connection string into separate configuration file (.env file for example, also add .env to .gitignore for security):

MONGO_URI=mongodb+srv://se371:se371pwd@se371.p9faam0.mongodb.net/?retryWrites=true&w=majority

Connect to the database

```
// First we need to import mongoose which is a module that makes it easy
// to communicate with a MongoDB server
const mongoose = require('mongoose');

// Secondly, we load .env configuration into process.env object using the dotenv external module
require('dotenv').config();

// Connect to database
mongoose.connect(process.env.MONGO_URI);
```

Defining a Schema using mongoose

In employee.js file, we define a schema for an Employee object:

```
const mongoose = require('mongoose');

let employeeSchema = new mongoose.Schema({
        name: { type: String, required: true},
        age: { type: Number},
        positions: { type: [String]}    // an array of strings.
});

// Now export the schema to be used by the app. 'Employee' changed to
// plural automatically will be the name of the collection in the database
// man -> men, person -> people, etc......
module.exports = mongoose.model('Employee', employeeSchema);
```

We can import the model into our app.js:

```
const Employee = require('./schema/employee');
```

Source code: chapter07/01_connect_mongo/app01.js

Saving a new element in the DB

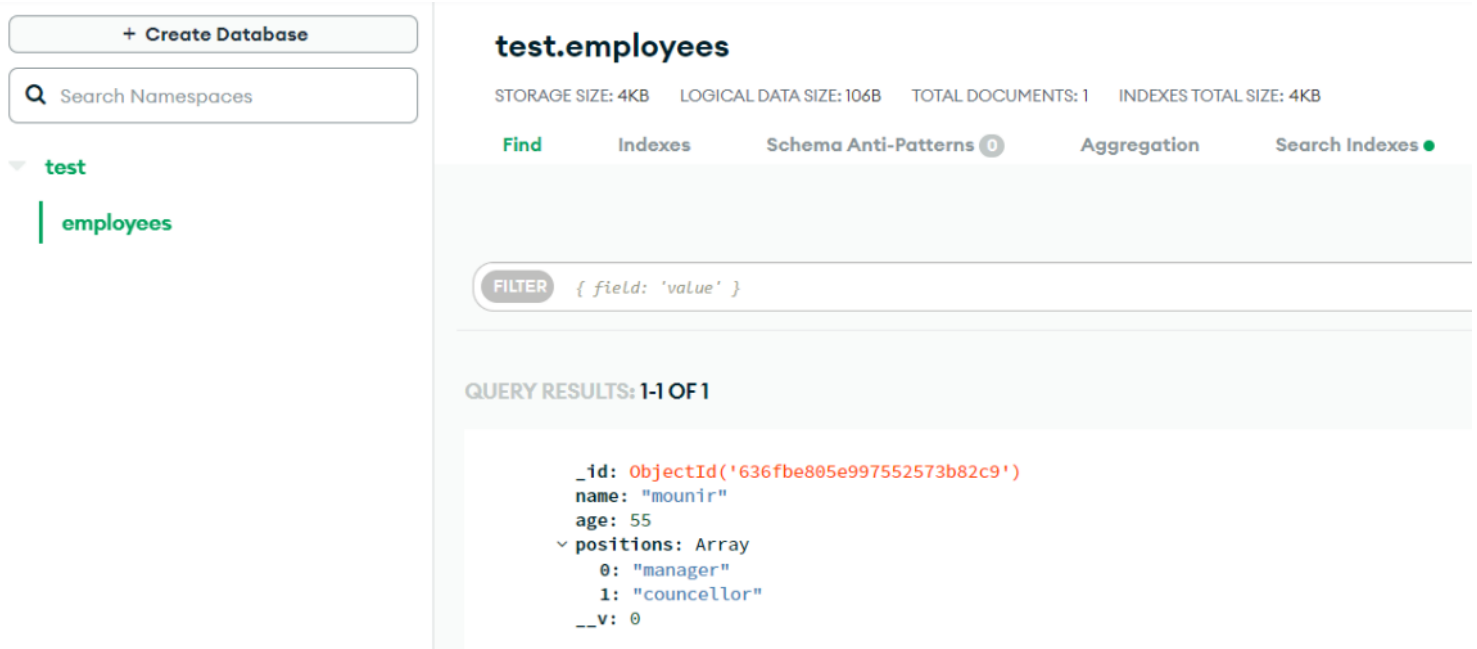

```
app.post('/v1/employees/name/:name/age/:age/positions/:positions',
        (request, response) => {
              let name = request.params.name;
              let age = request.params.age;
              let positions = request.params.positions.split(";");
              console.log(positions);
              let emp = new Employee({name: name, age: age, positions: positions});
              emp.save()
                 .then( (result) => {
                      console.log('Data saved...');
                      // Just to see in the browser that save worked correctly
                      response.send(result);
                  })
                  .catch( (err) => console.log(err));
         }
    );
```


Source code: chapter07/01_connect_mongo/app02.js

Calling the add-employee api endpoint

Then, we can call the API endpoint from the browser (change it to app.get to test):

http://localhost:3000/v1/employees/name/mounir/age/55/positions/manager;councellor



Response we get on the browser:

{"name":"mounir","age":55,"positions":["manager","councellor"],"_id":"636fbe805e997552573b82c9","__v":0}

Retrieving data form the DB

```
// Retrieve all employees from DB
app.get('/v1/employees', (request, response) => {
  Employee.find()
    .then( (result) => {
      // Send result to client
      response.send(result);
    })
    .catch( (err) => {
      console.log(err)
    });
 });
```

- Received Result in the browser, notice that we receive an array of JSON objects:

[{"_id":"636fbe805e997552573b82c9","name":"mounir","age":55,"positions":["manager","councellor"],"__v":0}]

The __v field was added by mongoose and denotes the version of the document.

Source code: chapter07/01_connect_mongo/app02.js

Search for employees by id

```
app.get('/v1/employees/id/:id', (request, response) => {
     Employee.findById(request.params.id)
           .then( (result) => {
               response.send(result);
           })
           .catch( (err) => {
               console.log(err);
           });
   });
```

Then, from the browser we can send this request:

```
http://localhost:3000/find-employee/id/636fbe805e997552573b82c9
```

Source code: chapter07/01_connect_mongo/app02.js

Search by a given criteria

```
app.get('/v1/employees/position/:position', (request, response) => {
  Employee.find({ positions: request.params.position })
    .then( (result) => {
      response.send(result);
    })
    .catch( (err) => {
      console.log(err);
    });
 });
```

Request:

http://localhost:3000/v1/employees/position/software%20engineer

[{"_id":"636fe3c1f64e839426b13390","name":"ali","age":32,"positions":["software engineer"],"__v":0},{"_id":"636fe3dbf64e839426b13392","name":"nawef","age":28,"positions":["software engineer"],"__v":0},{"_id":"636fe3eff64e839426b13394","name":"tahar","age":30,"positions":["software engineer"],"__v":0}]

Sort the result by another criteria

```
app.get('/v2/employees/position/:position', (request, response) => {
  Employee.find({ positions: request.params.position })
    .sort( { age: -1 } )      // -1 for descending order
    .then( (result) => {
      response.send(result);
    })
    .catch( (err) => {
      console.log(err);
    });
 });
```

Request:
http://localhost:3000/v2/employees/position/software%20engineer

[{"_id":"636fe3c1f64e839426b13390","name":"ali","age":32,"positions":["software engineer"],"__v":0}
,{"_id":"636fe3eff64e839426b13394","name":"tahar","age":30,"positions":["software
engineer"],"__v":0},{"_id":"636fe3dbf64e839426b13392","name":"nawef","age":28,"positions":["software engineer"],"__v":0}]

Use projections on the result

Doing a projection means selecting the properties that you want to send to the client. For example, some properties should stay secret:

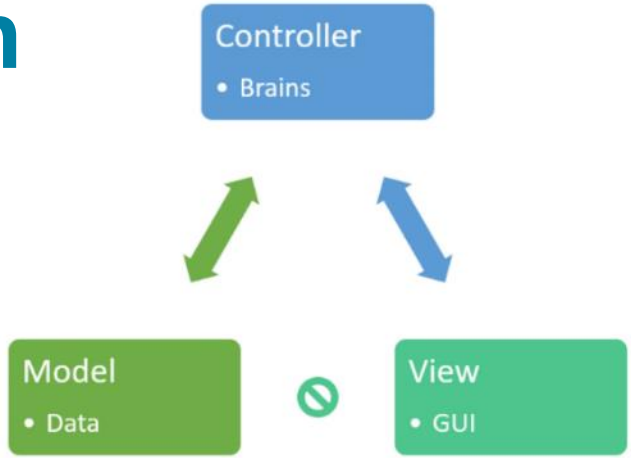
```
app.get('/v3/employees/position/:position', (request, response) => {
  Employee.find({ positions: request.params.position })
    .sort( { age: -1 } )
    .select( 'name positions' ) // select only name and positions properties
    .then( (result) => {
      response.send(result);
    })
    .catch( (err) => {
      console.log(err);
    });
 });
```

Note: the _id is still included in the result.

Complex server application

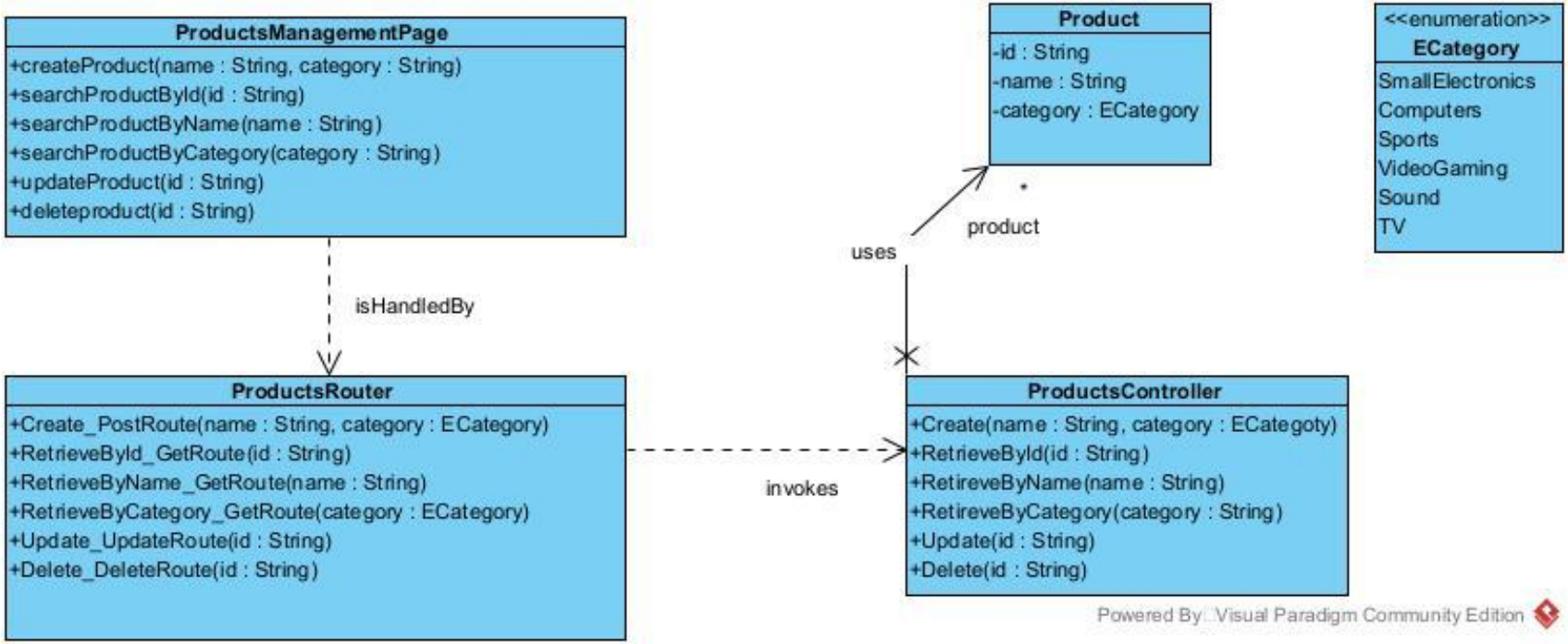
- The few API endpoints that we developed in the previous slides start to give us a complex file that will become difficult to maintain and debug.
- The solution to this problem is to apply a very popular pattern that is recognized to be the best solution to this issue: the MVC architectural pattern.

MVC architectural pattern



- In the MVC pattern, the architecture of the code is decomposed as follows:
  - The model refers to the parts of your application that store (database) and manipulate the data (save, update, delete, retrieve operations).
  - The View is the code responsible for presenting the data to the user (web pages, mobile interface, GUI).
  - The Controller is responsible to link the model and the view. It selects the view that will present the data. It select the model operation to execute and eventually returns the results to the view (here the controller is the express router + business logic).
- MVC is an application of (probably) the most important principle in software engineering: Separation of Concerns. Which has many advantages such as: code is more maintainable, easier to provide different views for the same data (add a mobile app to an existing web app, etc.)

Example of MVC



The controller routes redesign

- First, we redesign our routes to use /v1/employees for any request related to employees.
- Then we externalize routes related to employees in an external file: ./routes/employeesRoutes.js
- This allows us to have a modular controller defined in more than one file with each file handling routes related to a particular subject.

```
const express = require('express');
require('dotenv').config();

const mongoose = require('mongoose');

// Import employee routes
const employeeRouter = require('./routes/employeeRoutes');

// express app
const app = express();

// Connect to database then start server
…… (same)

// Employee Routes
app.use('/v1/employees', employeeRouter);

// Handle wrong requests
app.use((request, response) => {
  response.status(404).send('<h1>Error 404: Resource not found.</h1>');
})
```

Source code: chapter07/02_applying_MVC

The controller routes redesign (2)

- Now we can separate the routing code from the controller business logic.
- We place our controller business logic in: /controllers/employeeController.js

Here we have:
employeeRoutes.js
That contains all the routes related to employees

```
const express = require('express');
const employeeController = require('../controller/employeeController');

const router = express.Router();

router.post('/name/:name/age/:age/positions/:positions',
          employeeController.add_employee);

router.get(‘/', employeeController.all_employees);

router.get('/id/:id', employeeController.find_employee_byID);

router.get('/position/:position',
employeeController.find_employees_byPosition);

module.exports = router;
```

The controller routes redesign (3)

We place our controller business logic in:
/controllers/employeeController.js

```
const Employee = require('../schema/employee');

const add_employee = (request, response) => {
  let name = request.params.name;
  let age = request.params.age;
  let positions = request.params.positions.split(";");

  let emp = new Employee({name: name, age: age, positions: positions});
  emp.save()
      ……
}

const all_employees = (request, response) => {
  …….
}

const find_employee_byID = (request, response) => {
  …….
}


const find_employees_byPosition = (request, response) => {
  ……..
}

 module.exports = {add_employee, all_employees, find_employee_byID,
find_employees_byPosition};
```

Mongoose data validation

```
// In employeeRoutes.js, let's add this route
router.get('/add-employee/age/:age/positions/:positions',
           employeeController.add_employee);


// In employee.js. The model, we have:
let employeeSchema = new mongoose.Schema({
           name: { type: String, required: true},
           age: { type: Number},
           positions: { type: [String]}
});



 // If we send this request:
http://localhost:3000/employees/add-employee/age/45/positions/Tester



We get an error:
                    Error: Employee validation failed: name: Path `name` is required.
```

Case insensitive find

```
const find_employees_byPosition_caseInsensitive = (request, response) => {
  Employee.find({ positions: { $regex : new RegExp(request.params.position, "i")
} })
  .sort({ age : -1})
  .select('name positions')
    .then( (result) => {
      response.send(result);
    })
    .catch( (err) => {
      console.log(err);
    });
}
```

// the “i“ argument in the RegExp constructor is for case incencitive pattern matching.

http://localhost:3000/employees/find-employee/position_incencitive/Software%20engineer

[{"_id":"636fe3a9f64e839426b1338e","name":"ali","positions":["software engineer"]},
{"_id":"636fe3c1f64e839426b13390","name":"ali","positions":["software engineer"]},
{"_id":"636fe3eff64e839426b13394","name":"tahar","positions":["software engineer"]},
{"_id":"636fe3dbf64e839426b13392","name":"nawef","positions":["software engineer"]}]