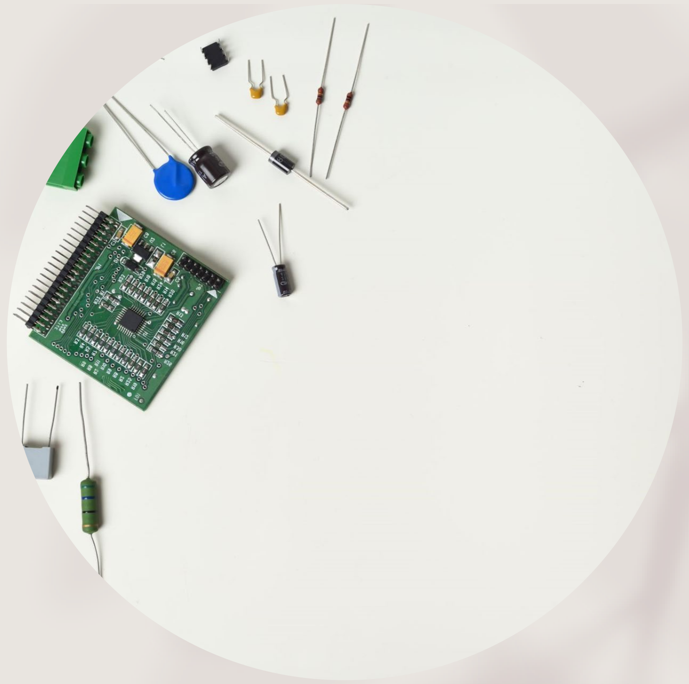
MVC Lecture slides

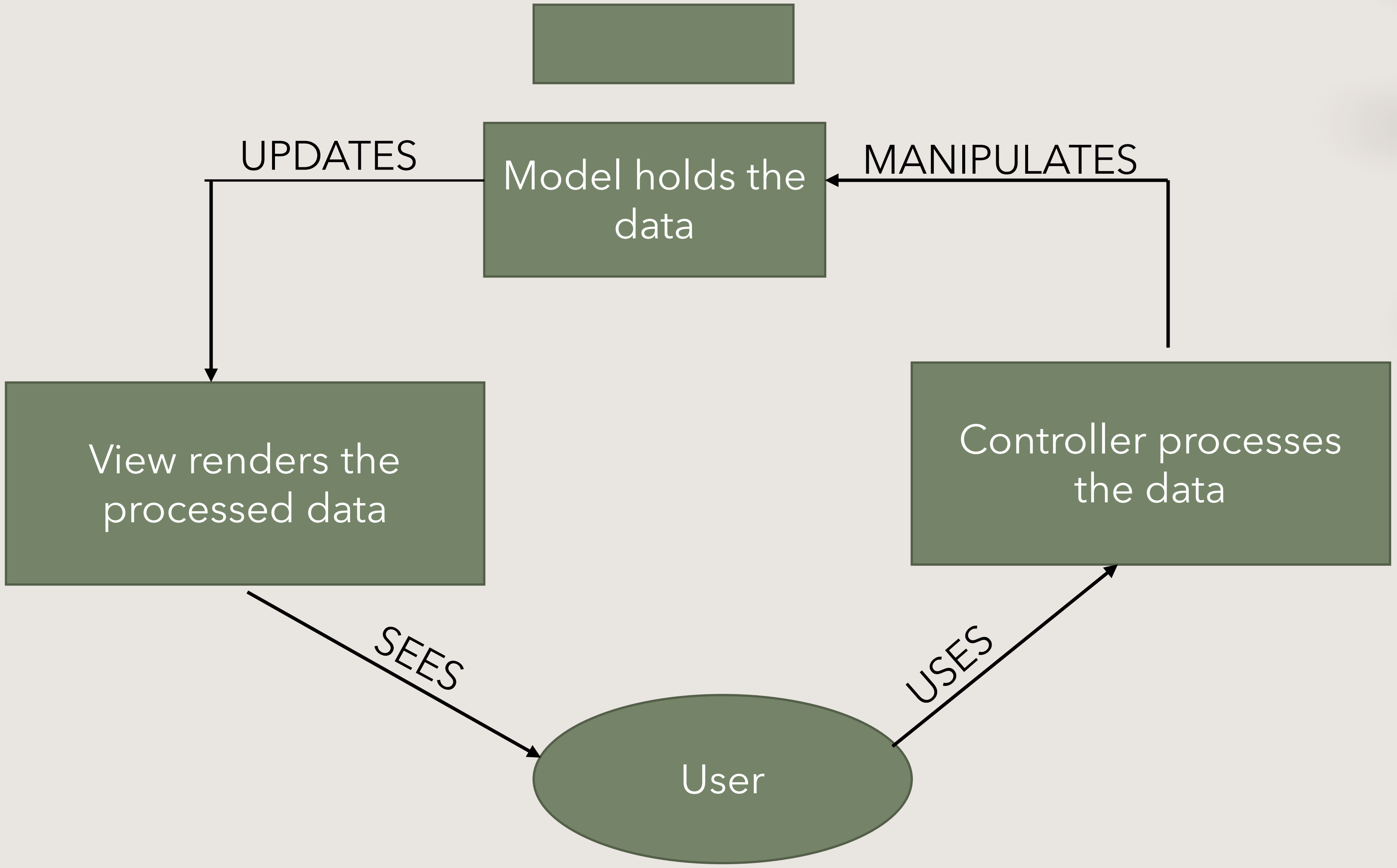
- Meera Ramesh

MVC
Model-view-controller

Model-View-Controller is a pattern used to separate all the components namely, View, Model and Controller as the name suggests. It helps to achieve separation of concerns and abstraction. It is done to separate the way information is represented internally and the way information is presented to the user. Traditionally used for GUI it is adapted for Web Applications. It is used by most of the modern languages and frameworks. Some popular languages that have MVC frameworks are Elixir, Javascript, C#, Python, Ruby, Swift.

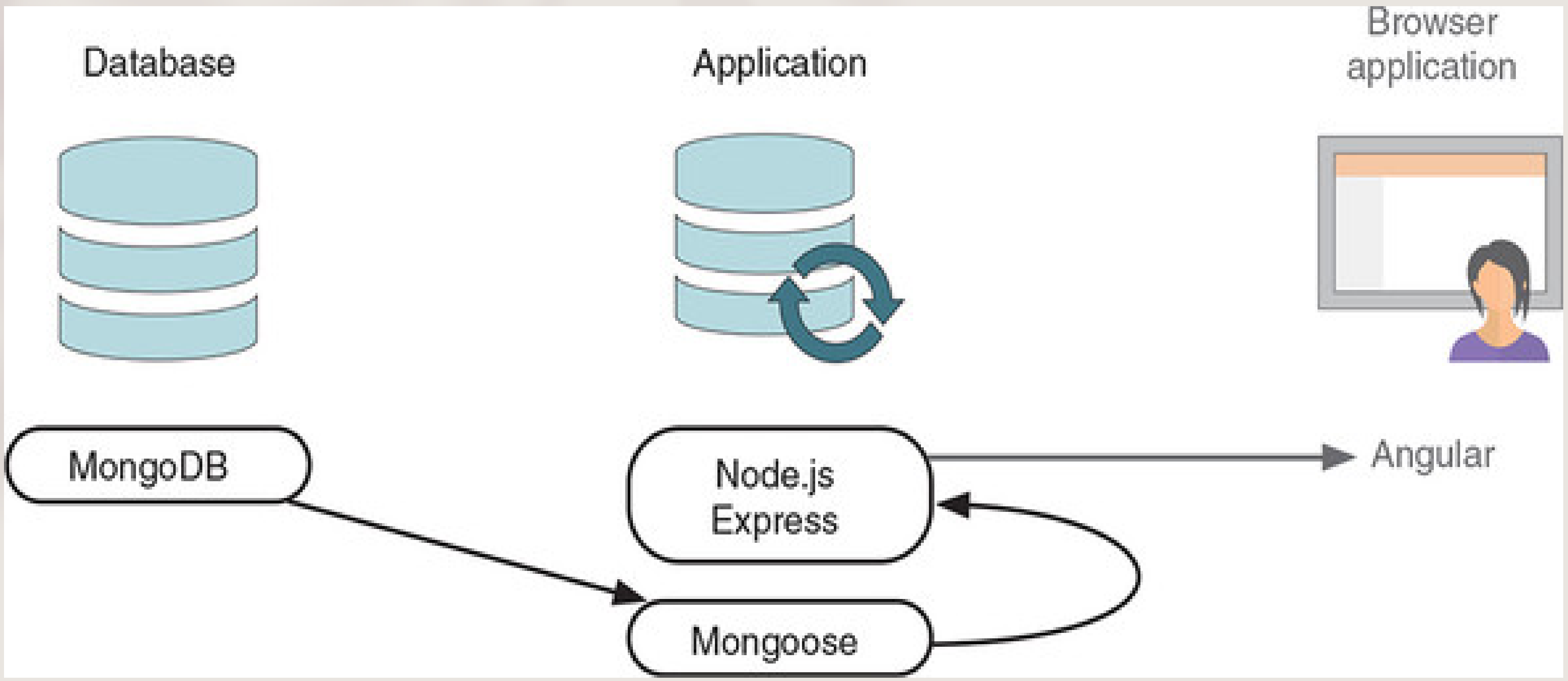


MVC Data Flow Visual Representation


Model holds the data
UPDATES
MANIPULATES
View renders the processed data
Controller processes the data
SEES
USES
User

Model Responsibilities and data

The data interactions in the MEAN stack and where Mongoose fits in. The Node/ Express application interacts with MongoDB through Mongoose; Node and Express can also talk to Javascript/Angular



Model Responsibilities and data

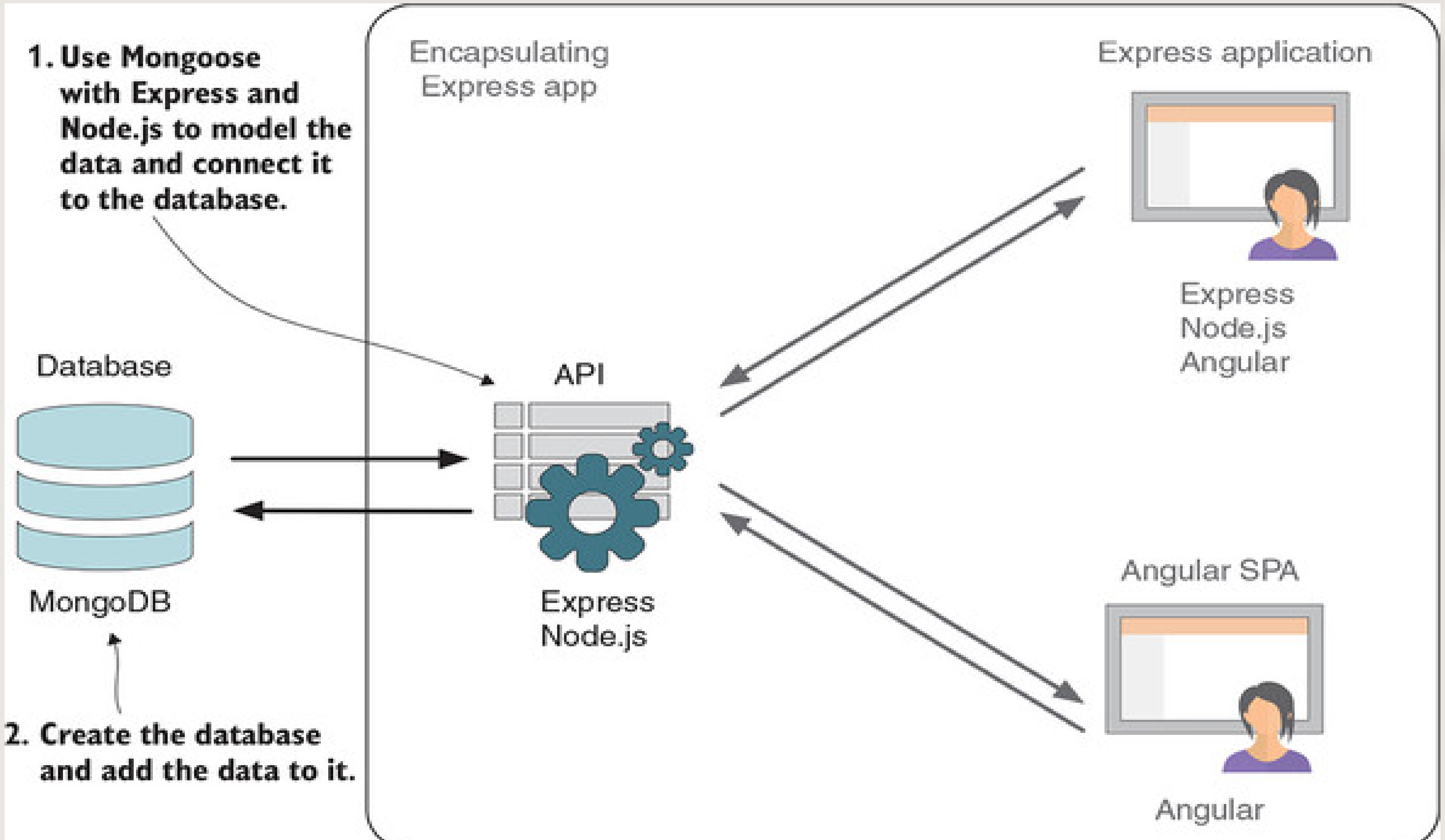
- Model is the central component of MVC. It is the application’s dynamic data structure as it manages the rules, data and logic of the application.
- Model represents the state of the application and operations that should be performed by it. The implementation logic is encapsulated in this component .
- Model receives user input from the controller
- Model schema shows how to create schemas in MongoDB and easily bind them to your database. It defines the validation rules that are applied on the client and server.
- Model contains all data-related logic like schemas and interfaces of a project, the databases and their fields

```
const mongoose = require('mongoose')
const TodoSchema = new mongoose.Schema({
 todo: {
  type: String,
  required: true,
 },
 completed: {
  type: Boolean,
  required: true,
 }
})
module.exports = mongoose.model('Todo',
        TodoSchema)
```

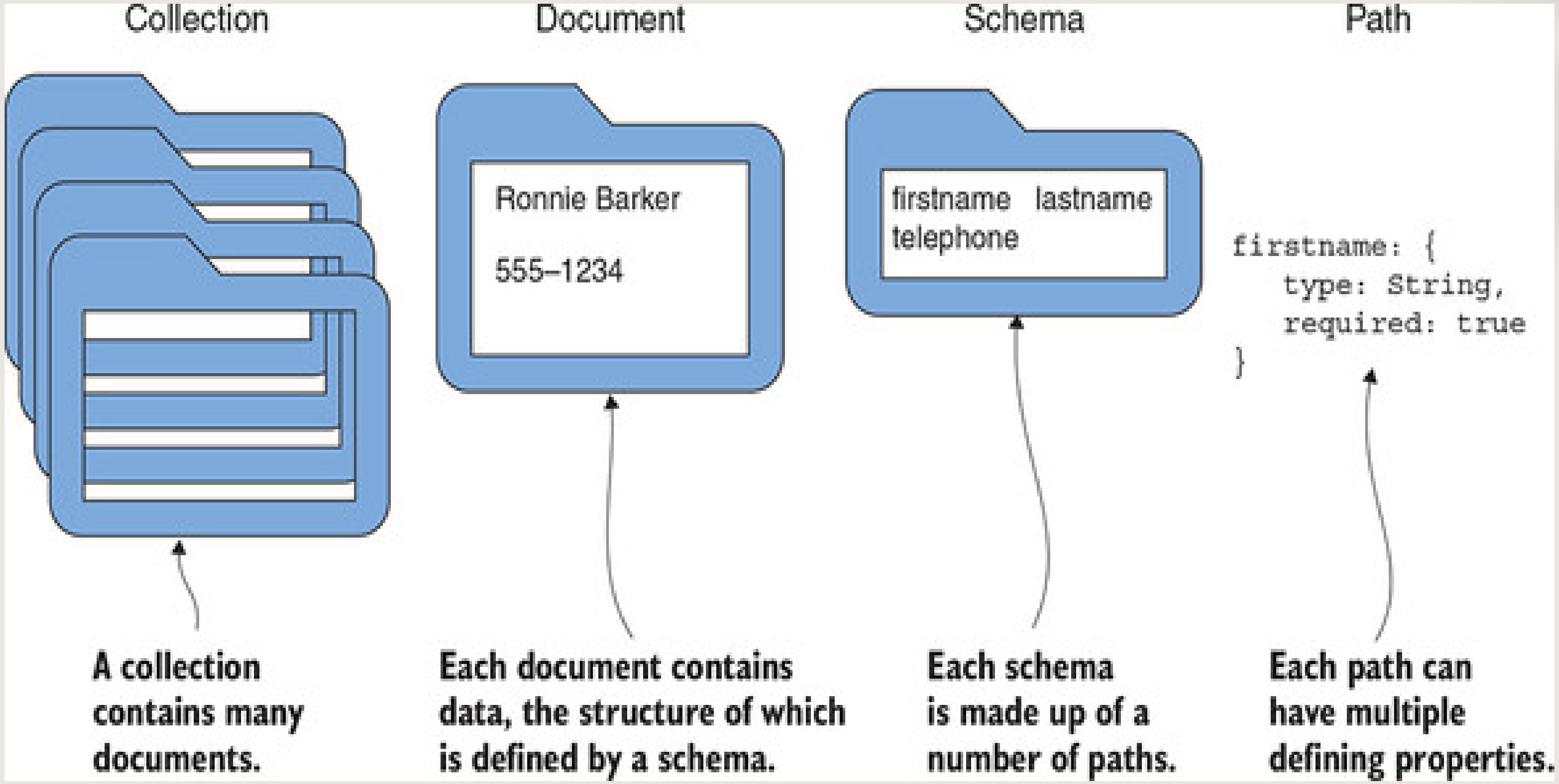
MongoDB and mongoose

- MongoDB is a general purpose , document based built for modern application developers that offers flexibility needed while querying and indexing . MongoDB is a NOSQL database and stores data in a JSON-like documents meaning fields can vary from document to document and data structure can be changed over time. This document model maps to the objects in your application code making data easy to work with.
- Mongoose is an Object Data Modeling(ODM) library for data management that works with MongoDB and Node.js.
- ODM is a data model that combines the features of OOP languages and Relational Data Model. It provides the data structures and operations of RDM along with the features of OOP such as data abstraction, reusability of data structure ad code .
- Mongoose includes the ability to add validation to your data definitions, meaning that you don’t have to write validation code in every place in your application where you send data back to the database.

Viewing the MongoDB database and using Mongoose inside Express to model the data and manage the connection to the database


1. Use Mongoose with Express and Node.js to model the data and connect it to the database.
Encapsulating Express app
Express application
Express
Node.js
Angular
Database
API
MongoDB
Express
Node.js
Angular SPA
Angular
2. Create the database and add the data to it.

- Relationships among collections, documents, schemas, and paths in MongoDB and Mongoose

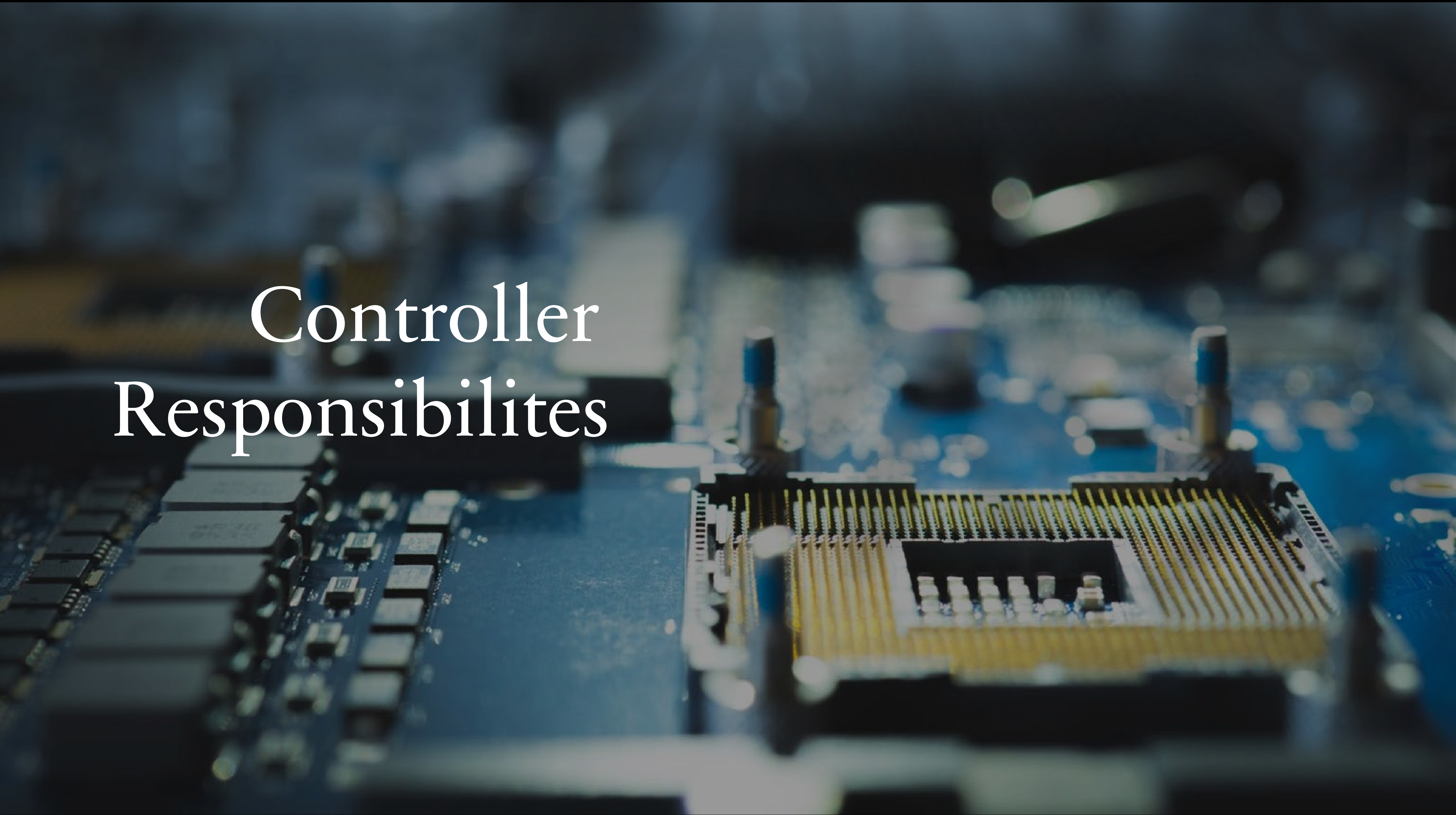

Collection
Document
Schema
Path
Ronnie Barker
555–1234
firstname lastname
telephone
A collection contains many documents.
Each document contains data, the structure of which is defined by a schema.
Each schema is made up of a number of paths.
Each path can have multiple defining properties.

```
firstname: {
   type: String,
   required: true
}
```

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

const connectDB = async () => {

 try {

   const conn = await
          mongoose.connect(process.env.DB_STRING,
          {

    useNewUrlParser: true,

    useUnifiedTopology: true,

    useFindAndModify: false,

   })

   console.log(`MongoDB Connected:
          ${conn.connection.host}`)

 } catch (err) {

   console.error(err)

   process.exit(1)

 }

}

module.exports = connectDB
```

Establishing MongoDB Mongoose Connection

- We use them together by installing MongoDB Atlas and then Mongoose in the terminal. Mongoose library comes with may built-in helper functions that allows us to quickly and easily perform basic CRUD operations.
- To connect to the database , we have to include modules that exist in separate files by using the require function . This function read a JS file , executes it and then proceeds to return the exports object. To ensure , we are properly linked up to the database we are including the success and error messages We have our schema defined in one of the slides above.
- Mongoose.connect returns a promise to fetch data from the mongo cluster. We want the mongo queries to be carried out, get returned result, handle it and return the Json when it is ready.



Controller Responsibilites

Anatomy of an Express Application

A typical structure of an Express server file will most likely contain the following parts:

Dependencies
Importing the dependencies such as the express itself. These dependencies are installed using npm.

Instantiations
These are the statements to create an object. To use express, we have to instantiate the app variable from it.

Configurations
These statements are the custom application based settings that are defined after the instantiations or defined in a separate file (more on this when discuss the project structure) and required in our main server file.

Middleware
These functions determine the flow of request-response cycle. They are executed after every incoming request. We can also define custom middleware functions. We have section on them below.

Routes
They are the endpoints defined in our server that helps to perform operations for a particular client request.

Controller Responsibilities

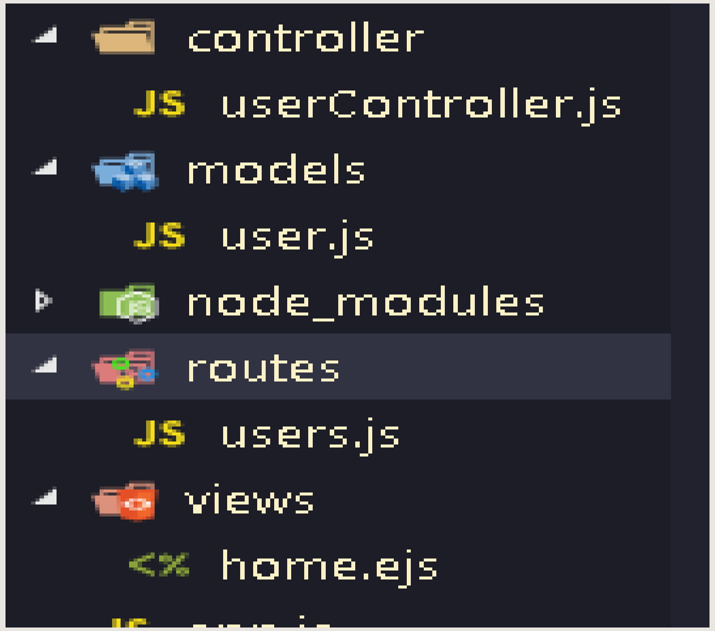
- Controllers are components that handle user interaction while processing all the logic and incoming requests, manipulates data using model and select a view to render the final output file .
- Controller is the initial entry point and is responsible for selecting which model types to work with and which view to render which in turn controls how the app responds to a request.
- It processes all the logic and incoming requests, manipulates data using Model. It renders the final output file interacting with views.

Controllers Repsonsibilities

- Controller is used to define and group a set of action which is a method o a controller which handles requests. They logically group similar actions together, which allows common sets of rules such as routing ,caching and authorization to be applied collectively.
- Requests are mapped to actions through routing. Controllers are responsible for the initial processing o the request and instantiation of the model. It delegates to services handling the responsibilities .It returns either the proper view and its associated view data or the result of the API call.
- Simply route requests to controller actions, implemented as normal programming language methods. Data from the request path, query string, and request body are automatically bound to method parameters.

Project Files and their contents

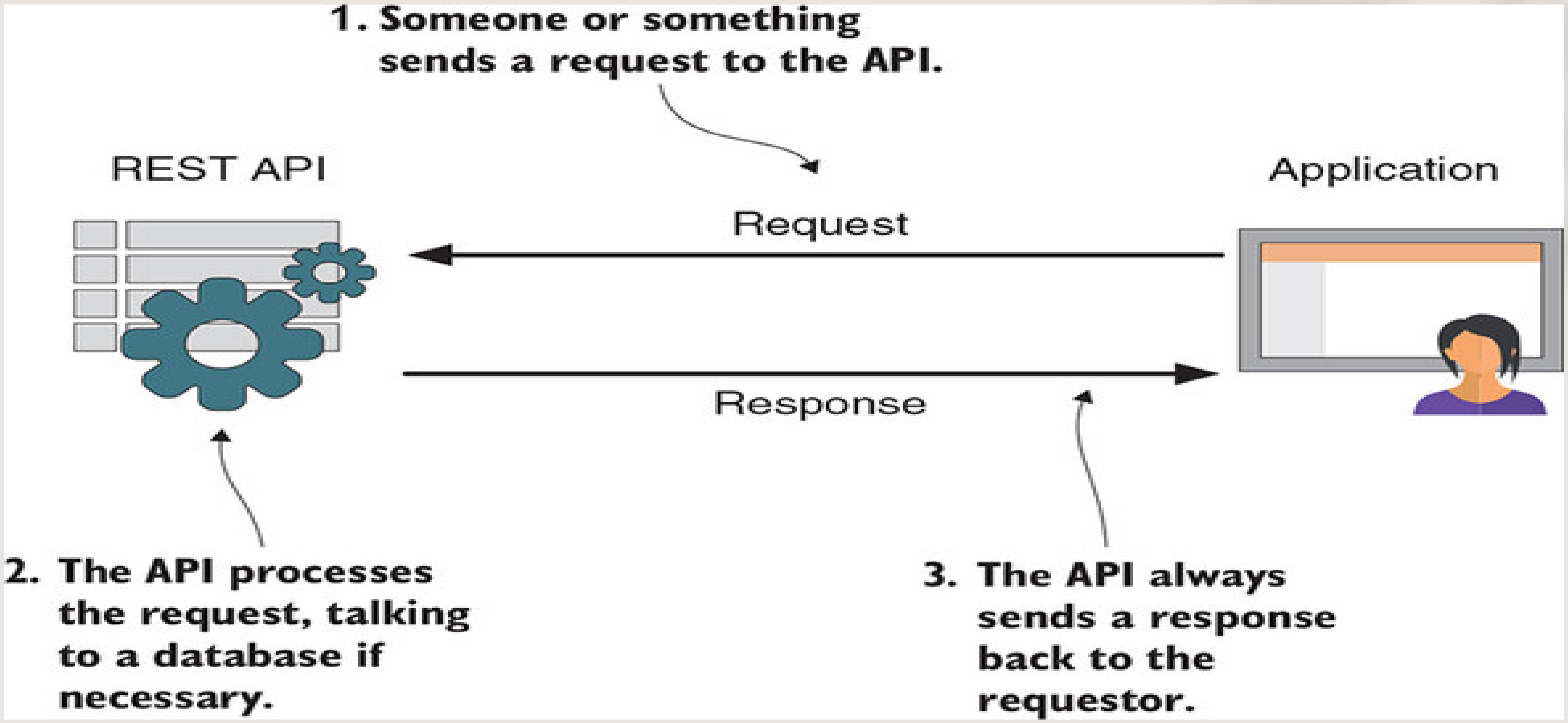
Set up MVC pattern in project



```
◢ controller
    JS userController.js
◢ models
    JS user.js
▷ node_modules
◢ routes
    JS users.js
◢ views
    <% home.ejs
```

- Models folder will contain the entire collection which will be used in our project.
- Views folder will contain pages that will contain a mix of HTML and model data.
- The controller folder will contain the business logic which will be executed on the request.
- Routes will contain the routes of the application.

- A REST API takes incoming HTTP requests, does some processing and returns HTTP responses

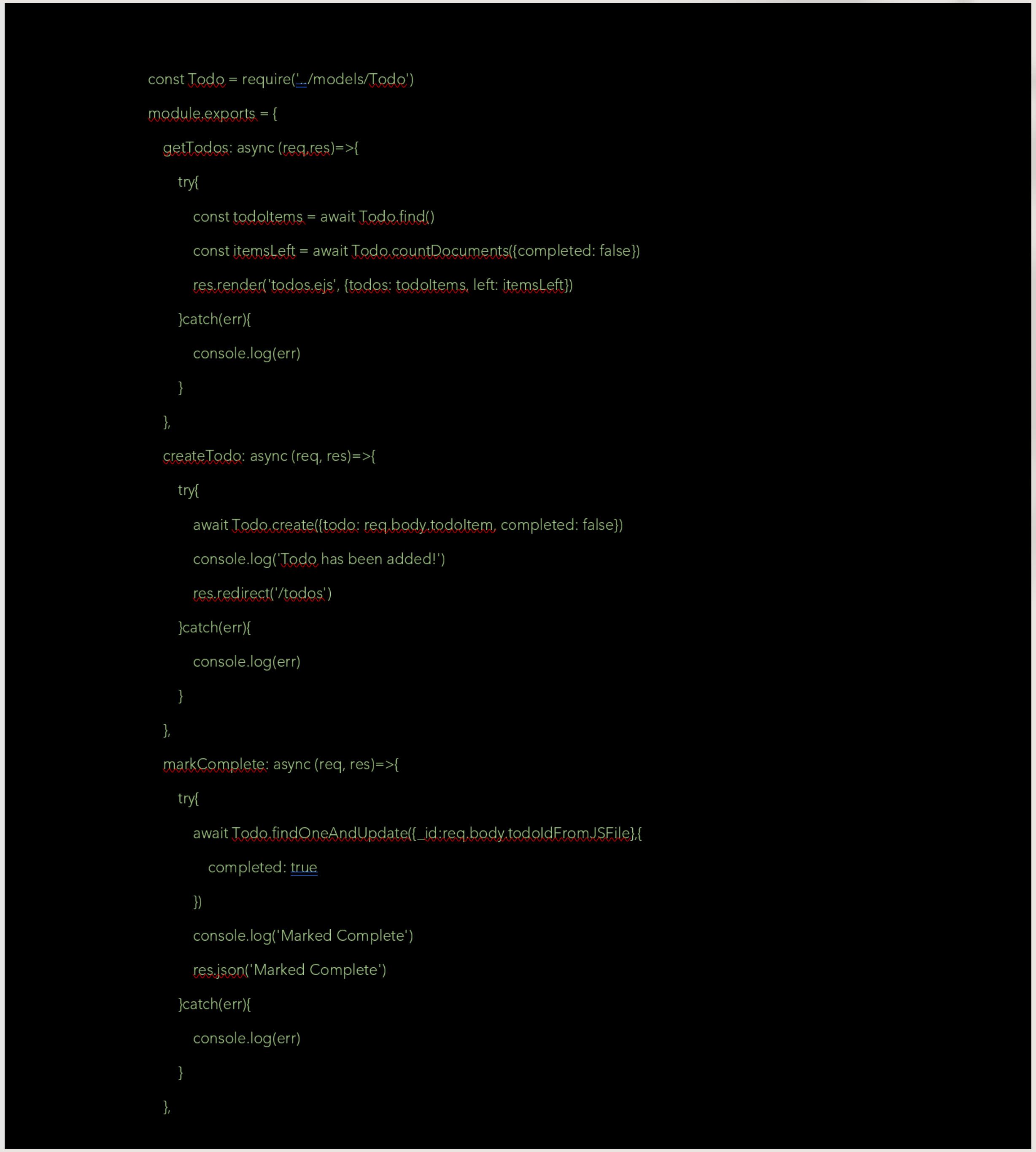

1. Someone or something sends a request to the API.
REST API
Application
Request
Response
2. The API processes the request, talking to a database if necessary.
3. The API always sends a response back to the requestor.

Four common request options for defining a call to an API

Option	Description	Required
url	Full URL of the request to be made, including protocol, domain, path, and URL parameters	Yes
method	Method of the request, such as GET, POST, PUT, or DELETE	No—defaults to GET if not specified
json	Body of the request as a JavaScript object; an empty object should be sent if no body data is needed	Yes—ensures that the response body is also parsed as JSON
qs	JavaScript object representing any query string parameters	No

Routes and controllers in Express.js

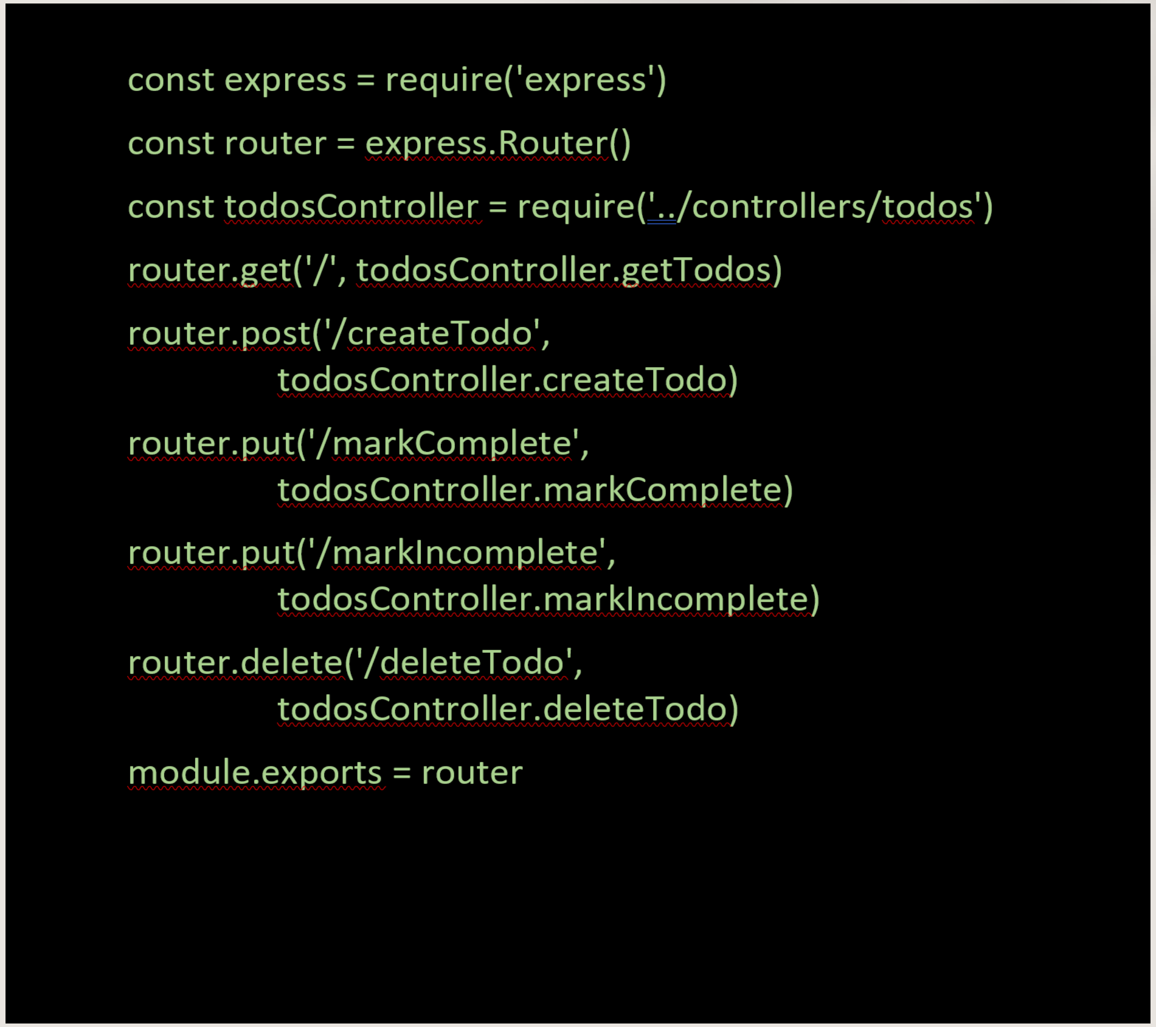
- Route is a section of express code that associates an HTTP verb (GET, POST, PUT, DELETE, etc.), a URL path/pattern, and a function that is called to handle that pattern.
- Creating routes can be done in many ways. We use the express. Router middleware as it allows us to group the route handlers for a particular part of a site together and access them using a common route-prefix .
- Router functions are Express middleware which means that they must either complete (respond to) the request or call the next function in the chain. In the case above we complete the request using send(), so the next argument is not used (and we choose not to specify it).
- The code example given here uses get(), post(),update().



```
const Todo = require('../models/Todo')
module.exports = {
  getTodos: async (req,res)=>{
    try{
      const todoItems = await Todo.find()
      const itemsLeft = await Todo.countDocuments({completed: false})
      res.render('todos.ejs', {todos: todoItems, left: itemsLeft})
    }catch(err){
      console.log(err)
    }
  },
  createTodo: async (req, res)=>{
    try{
      await Todo.create({todo: req.body.todoItem, completed: false})
      console.log('Todo has been added!')
      res.redirect('/todos')
    }catch(err){
      console.log(err)
    }
  },
  markComplete: async (req, res)=>{
    try{
      await Todo.findOneAndUpdate({_id:req.body.todoIdFromJSFile},{
        completed: true
      })
      console.log('Marked Complete')
      res.json('Marked Complete')
    }catch(err){
      console.log(err)
    }
  },
```

Routes and HTTP Request

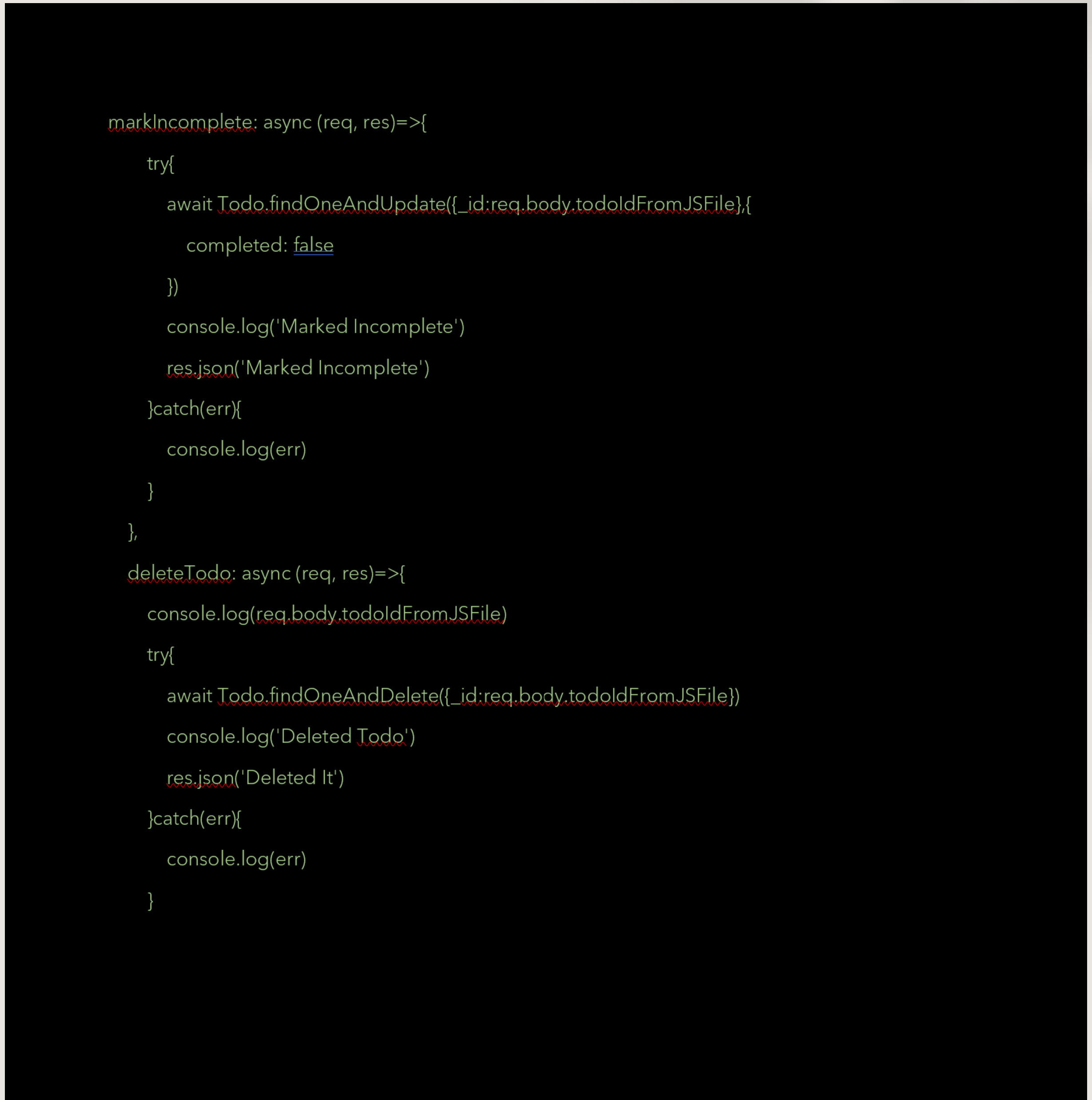
- GET route –request data from the database through server
- POST route-create a new record or collection
- PUT route- update the record in the database
- DELETE route-delete a collection in the database



```
const express = require('express')
const router = express.Router()
const todosController = require('../controllers/todos')
router.get('/', todosController.getTodos)
router.post('/createTodo',
          todosController.createTodo)
router.put('/markComplete',
          todosController.markComplete)
router.put('/markIncomplete',
          todosController.markIncomplete)
router.delete('/deleteTodo',
          todosController.deleteTodo)
module.exports = router
```

Routing defines the way in which the client requests are handled by the application endpoints.

- The Router.get() method to respond to HTTP GET requests with a certain path. The Router also provides route methods for all the other HTTP verbs, as put(), post(), delete(),patch () etc.
- The route paths define the endpoints at which requests can be made.
- Route paths can also be string patterns. String patterns use a form of regular expression syntax to define *patterns* of endpoints that will be matched. The route paths can also be JavaScript regular expressions.
- Route parameters are *named URL segments* used to capture values at specific positions in the URL. The named segments are prefixed with a colon and then the name (e.g. /**:**your_parameter_name/. The captured values are stored in the req.params object using the parameter names as keys (e.g. req.params.your_parameter_name).



```
markIncomplete: async (req, res)=>{
    try{
        await Todo.findOneAndUpdate({_id:req.body.todoIdFromJSFile},{
            completed: false
        })
        console.log('Marked Incomplete')
        res.json('Marked Incomplete')
    }catch(err){
        console.log(err)
    }
  },
  deleteTodo: async (req, res)=>{
    console.log(req.body.todoIdFromJSFile)
    try{
        await Todo.findOneAndDelete({_id:req.body.todoIdFromJSFile})
        console.log('Deleted Todo')
        res.json('Deleted It')
    }catch(err){
        console.log(err)
    }
```

Routing in express, node.js

Routing in ExpressJS is used to subdivide and organize the web application into multiple mini-applications each having its own functionality. It provides more functionality by subdividing the web application rather than including all of the functionality on a single page. These mini-applications combine together to form a web application. Each route in Express responds to a client request to a particular route/endpoint and an HTTP request method (GET, POST, PUT, DELETE, UPDATE and so on). Each route basically refers to the different URLs in the website.

Routing refers to how a server side application responds to a client request to a particular endpoint. This endpoint consists of a URI (a path such as / or /books) and an HTTP method such as GET, POST, PUT, DELETE, etc.

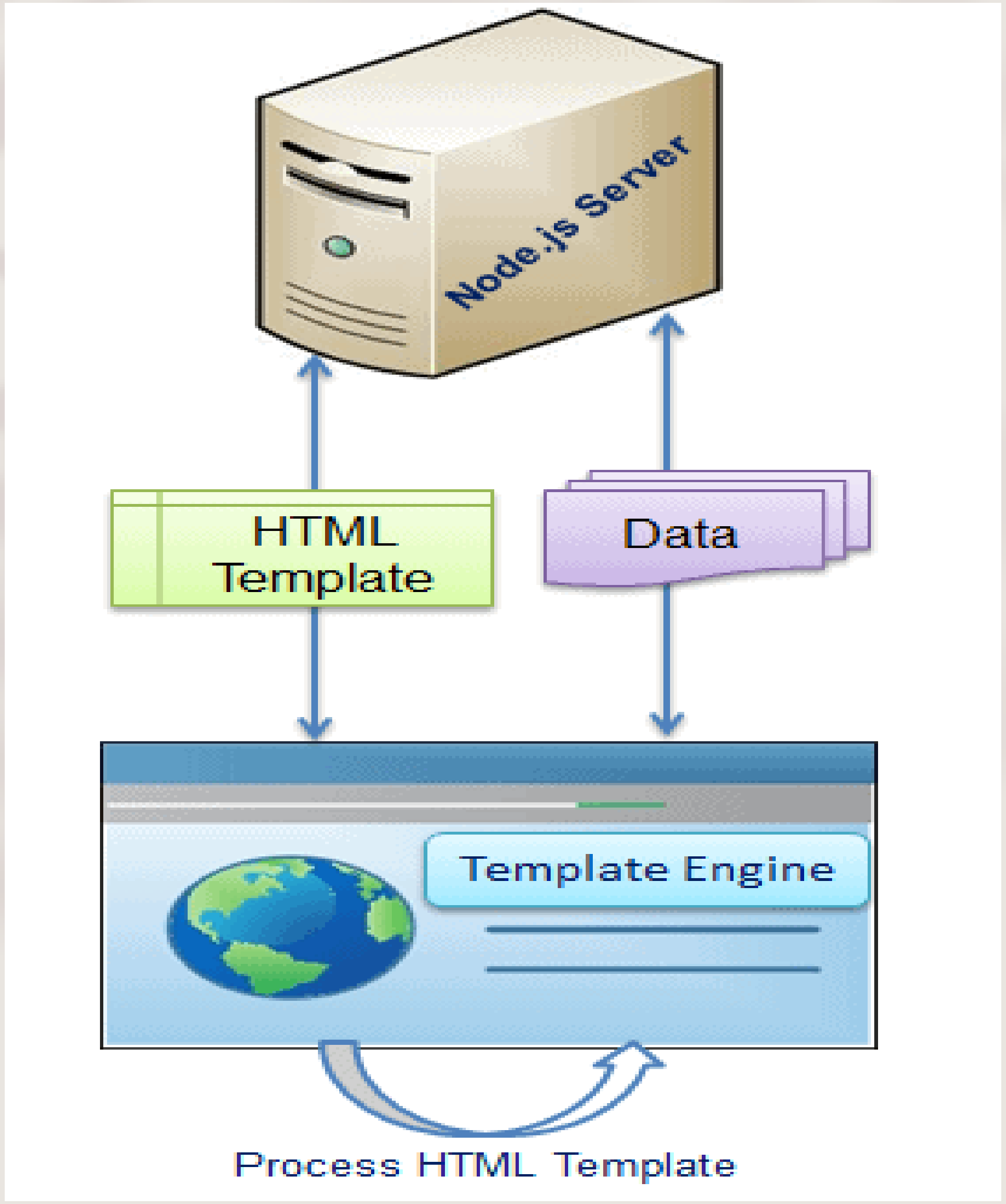
Middleware

Middleware is a term for any software or service that enables the parts of a system to communicate and manage data. It is the software that handles communication between components and input/output, so developers can focus on the specific purpose of their application.

```
const express = require('express')
const app = express()
const connectDB = require('./config/database')
const homeRoutes = require('./routes/home')
const todoRoutes = require('./routes/todos')

require('dotenv').config({path: './config/.env'})

connectDB()

app.set('view engine', 'ejs')
app.use(express.static('public'))
app.use(express.urlencoded({ extended: true }))
app.use(express.json())

app.use('/', homeRoutes)
app.use('/todos', todoRoutes)
```

View Engine Responsibilities

- Any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants
- The Razor syntax provides a simple, clean and lightweight way to render HTML content based on your view. Razor lets you render a page using a programming language, producing fully HTML5 compliant web pages.
- view renders presentation of the model in a particular format Views are responsible for presenting content through the user interface. They use the to embed .NET code in HTML markup. There should be minimal logic within views, and any logic in them should relate to presenting content.
- The View component is used for all the UI logic of the application. It presents the model’s data to the user. Basically it just receives the data from the model and displays it to the user.
- Template engines are libraries that allow us to use different template languages. A template language is a special set of instructions (syntax and control structures) that instructs the engine how to process data. Using a template engine is easy with Express. The popular template engines such as Pug, EJS, Swig, and Handlebars are compatible with Express. However, Express comes with a default template engine, Jade, which is the first released version of Pug.



Template Engine in Node.js

Client-side browser loads HTML template, JSON/XML data and template engine library from the server. Template engine produces the final HTML using template and data in client's browser. However, some HTML templates process data and generate final HTML page at server side also.

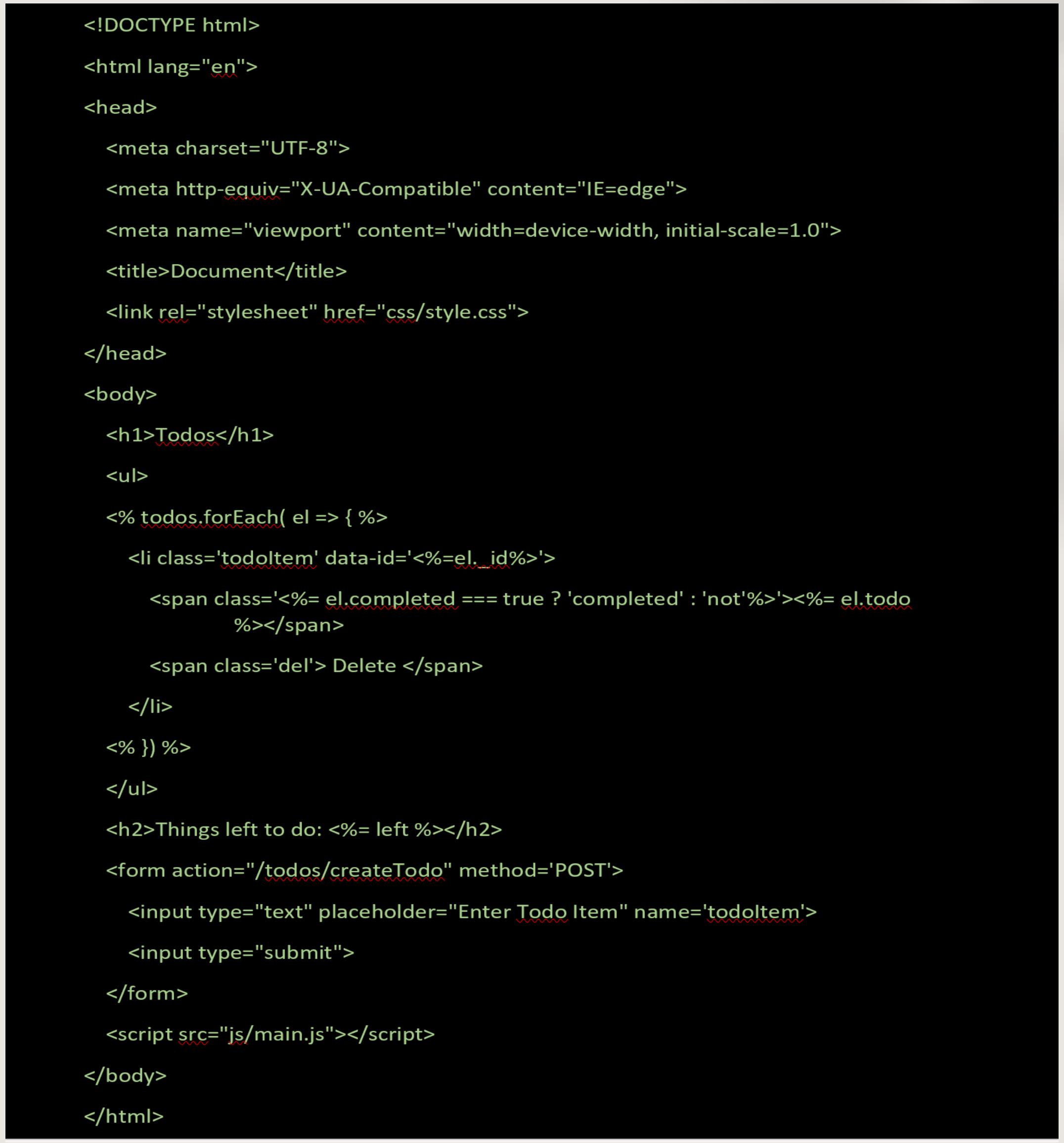
Use of Template Engine

- View engines are useful for rendering web pages. Template engine helps us to create an HTML template with minimal code. Also, it can inject data into HTML template at client side and produce the final HTML.
- There are many view engines available in the market like Mustache, Handlebars, EJS, etc but the most popular among them is EJS which simply stands for Embedded JavaScript. It is a simple templating language/engine that lets its user generate HTML with plain javascript.

1. Advantages of using Template Engine are improves developer's productivity, readability, faster performance, single template for multiple pages and maximizes client side processing.

Template Engines

- View engines are useful for rendering web pages. Template engine helps us to create an HTML template with minimal code. Also, it can inject data into HTML template at client side and produce the final HTML.
- There are many view engines available in the market like Mustache, Handlebars, EJS, etc but the most popular among them is EJS which simply stands for Embedded JavaScript. It is a simple templating language/engine that lets its user generate HTML with plain javascript.
- Advantages of using Template Engine are improves developer's productivity, readability, faster performance, single template for multiple pages and maximizes client side processing.



```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <link rel="stylesheet" href="css/style.css">
</head>
<body>
  <h1>Todos</h1>
  <ul>
  <% todos.forEach( el => { %>
    <li class='todoItem' data-id='<%=el._id%>'>
      <span class='<%= el.completed === true ? 'completed' : 'not'%>'><%= el.todo
              %></span>
      <span class='del'> Delete </span>
    </li>
  <% }) %>
  </ul>
  <h2>Things left to do: <%= left %></h2>
  <form action="/todos/createTodo" method='POST'>
    <input type="text" placeholder="Enter Todo Item" name='todoItem'>
    <input type="submit">
  </form>
  <script src="js/main.js"></script>
</body>
</html>
```