Week 2 Node.js

Use the event-driven nature of JavaScript to support non-blocking operations in the platform:

- You register code to specific events
- Code will be executed once the event is emitted

Use asynchronous programming (operations can execute without blocking other operations):

Eliminate waiting and continue with the next request

Memory efficient: single-threaded, non-blocking

ECMAScript primitive data types:

- Boolean
- Null
- Undefined
- Number
- String
- Symbol
- Object

Built-in modules

- http:

  to make Node.js act as an HTTP server

  import command: const http=require ('http');
- fs: to handle file system

  import command: const http=require ('fs');
- url: to parse URL strings

  import command: const http=require ('url');

Prerequisite:

npm init

- every Node.js application or module will contain a package.json file
- specify the version of current release, the name of application, the main application file
- essential for npm to save any package to the node community online

```
let http = require("http");
let fs = require("fs");
http.createServer(function (request, response) {
    console.log("request ", request.url);
    let filePath = "." + request.url;
    if (filePath === "./") {
      filePath = "./index";
    }
    fs.readFile(filePath+".html", function (error, content) {
      if (error) {
        fs.readFile("./404.html", function (error, content) {
          response.writeHead(404, { // file not found
            "Content-Type": "text/html",
          });
          response.end(content, "utf-8");
        });
      } else {
        response.writeHead(200, { // OK
          "Content-Type": "text/html",
        });
        response.end(content, "utf-8");
      }
    });
  })
  .listen(8080);
console.log("Server running at http://127.0.0.1:8080/");
```

```
http.createServer(function (req, res){
        ....
}).listen (8080);
```

http.createServer( ): turn your computer into an HTTP server & creates an HTTP Server object

function (req, res): a callback method (requestListener)

req: get details about the coming request (eg. method, URL, headers & body of request)

res: contain many methods for sending data back to the client

server.listen(): specify the port number the server listens on

```
fs.readFile(filePath+ “.html”, function (error, content){
        ….
});
```

fs.readFile (path, callback):

asynchronously reads the entire contents of a file and invokes the callback upon completion

Param1: file path

Param2: callback function (err,data)

- err: error object (get a value that contains the error code and description)
- data: the contents of the file

```
res.writeHead(404, {
        “Content-Type”: “text/html”
});
```

res.writeHead (statusCode, headers):

Param1: the status code

- 200: all is OK
- 404: page not found

Param2: an object containing the response headers

```
res.end(content, “utf-8”);
```

res.end ([data], [encoding], [callback]):

the method must be called on each response

used to end the response process

param1: the data with which the user wants to end the response

param2: default encoding is ‘utf8’

```
response.writeHead(200);
response.write('Hello from FIT2095!! the time is : ' + currentTime);
response.end();
```

res.write(chunk, [encoding], [callback]): respond to a client's request by a string/buffer

```
let http = require("http");
let url = require("url");
http.createServer(function (req, res) {
   console.log("URL=" + req.url);
   res.writeHead(200, {
     "Content-Type": "text/html",
   });
   var baseURL = "http://" + req.headers.host + "/";
   console.log(req.headers.host);
   var url = new URL(req.url, baseURL);
   console.log(url);
   let params = url.searchParams;
   console.log(params);
   let msg = params.get("year") + " " + params.get("month");
   res.end(msg);
 }).listen(8080);
```

address:

http://localhost:8080/getDate?year=2021&month=8

req.url:

/getDate?year=2021&month=8

req.headers.host:

localhost:8080

var url=new URL(req.url, baseURL);

const url= new URL(url, [base]);

URL() constructor returns a newly created URL object representing the URL defined by the parameters

url:

```
URL {
  href: 'http://localhost:8080/getDate?year=2021&month=8',
  origin: 'http://localhost:8080',
  protocol: 'http:',
  username: '',
  password: '',
  host: 'localhost:8080',
  hostname: 'localhost',
  port: '8080',
  pathname: '/getDate',
  search: '?year=2021&month=8',
  searchParams: URLSearchParams { 'year' => '2021', 'month' => '8' },
  hash: ''
}
```

url.searchParams:

return a URLSearchParams object

```
URLSearchParams { 'year' => '2021', 'month' => '8' }
```

params.get("year"): 2021

params.get("month"): 8

http://localhost:8080/getDate?year=2021&month=8

url.parse(req.url,true).query: {‘year’: 2021, ‘month’: 8}

url.parse(req.url,true).pathname: /getDate

url.parse(req.url,true).host: localhost:8080

url.parse(req.url,true).search: ?year=2021&month=8

Workshop:

http://localhost:4000/weeks/2/wsq/?q=1&mark=20

query string: q=1&mark=20

protocol: HTTP (HyperText Transfer Protocol)

port number: 4000

pathname: /weeks/2/wsq/

server address: localhost (the reserved address for the local computer, equal to IP address 127.0.0.1

Week 3 Express

Web application framework for Node.js

- provide all tools and basic building blocks you need to get a web server up and running by writing very little code
- allow us to focus on logic & content

Prerequisite:

npm init

npm install express

```
let express = require("express");
let app = express();
let url = require("url");

let db = [];
let rec = {
    name: "Tim",
    age: 23,
    address: "Mel",
};
db.push(rec);

app.get("/", function (req, res) {
    res.send("Hello from FIT2095");
});
app.get("/list", function (req, res) {
    res.send(generateList());
});
app.get("/newuser", function (req, res) {
    // let baseURL = "http://" + req.headers.host + "/";
    // let url = new URL(req.url, baseURL);
    // let params = url.searchParams;
    // console.log(params);
    let newRec = {
        name: req.query.name,
        age: req.query.age,
        address: req.query.address,
    };
    db.push(newRec);
    res.send(generateList());
});
app.get("/delete", function (req, res) {
    //   let baseURL = "http://" + req.headers.host + "/";
    //   let url = new URL(req.url, baseURL);
    //   let params = url.searchParams;
    //   console.log(params);
    deleteUser(req.query.id);
    res.send(generateList());
});
app.listen(8080);

function deleteUser(id) {
    db.splice(id, 1);
}

function generateList() {
    let st = "Name  Age   Address  </br>";
    for (let i = 0; i < db.length; i++) {
        st += db[i].name + " | " + db[i].age + " | " + db[i].address + "</br>";
    }
    return st;
}
```

let app=express();

create an instance of Express app

db.push (rec);

array.push(item1, item2, …,itemX):

add new items to the end of an array

return the new length of the array

```
app.get("…", function (req,res){
        res.send(…);
});
```

app.get(path, callback): define a route handler for GET requests to a given URL

res.send(…): send some data and end the response

http://localhost:8080/newuser?name=Max&age=22&address=ACT

req.query: {name: 'Max', age: '22', address: 'ACT'}

db.splice(id, 1);

array. splice(start, [deleteCount]):

start: the index at which to start changing the array

deleteCount: the number of elements to remove from start

```
Route path: /users/:userId/books/:bookId
Request URL: http://localhost:3000/users/34/books/8989
req.params: { "userId": "34", "bookId": "8989" }
```

Workshop:

```
let express=require ('express');
let app = express();
app.get('/week3/marks/:prerq/:wsq/:lab', function (req, res){
   let prerq=req.params.prerq;
   let wsq=req.params.wsq;
   let lab=req.params.lab;
   let weekMark=prerq*0.1+wsq*0.1+lab*0.2;
   res.send('Week 3 Mark is '+weekMark);
});
app.listen (8080);
```

router paths based on string patterns

'ab?cd': include b or not (acd, abcd)

'ab+cd': multiple b (abcd, abbcd, abbbcd)

'ab*cd': anything between ab & cd (abcd, abxcd, abRANDOMcd)

'/ab(cd)?e': include cd or not (abe, abcde)

/a/: anything with "a" in it

/.*fly$/: anything ends with "fly"

Workshop:

```
app.get('/week3*', function (req, res){
res.send('Welcome to week 3');
});
```

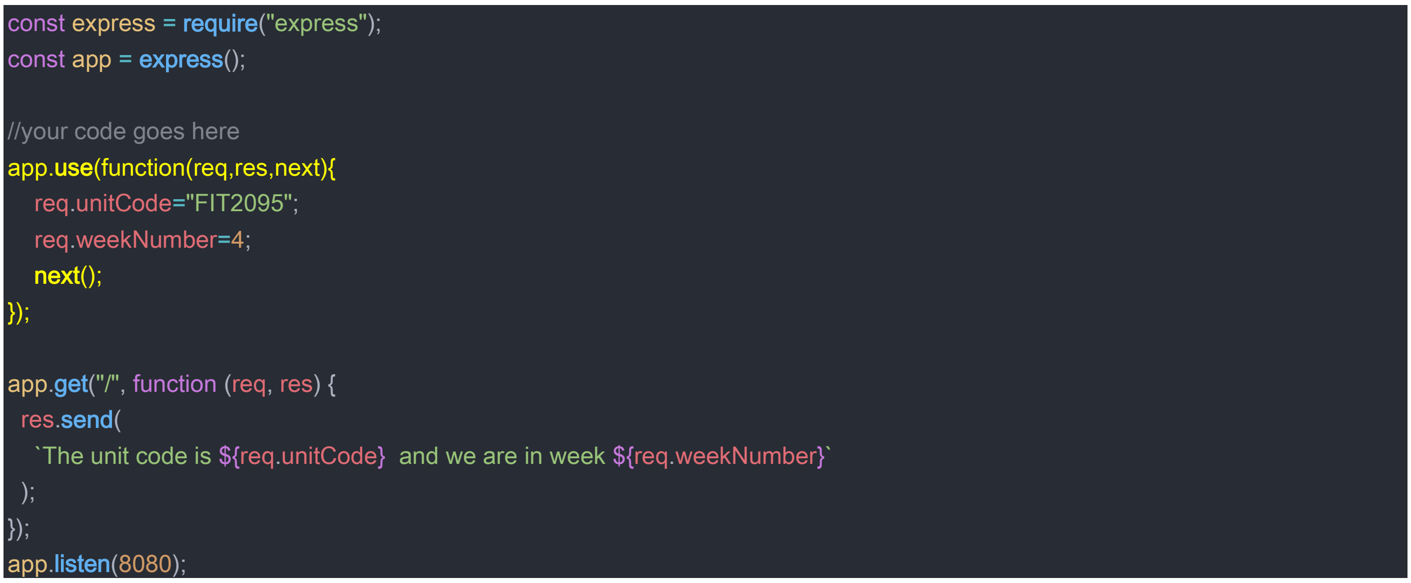
Accept requests with the prefix 'week3'

Week 4 Advanced Express (EJS)

Middleware:

A function that has access to Request and Response objects and gets invoked by Express to do some tasks

Workshop:



```
const express = require("express");
const app = express();

//your code goes here
app.use(function(req,res,next){
    req.unitCode="FIT2095";
    req.weekNumber=4;
    next();
});

app.get("/", function (req, res) {
  res.send(
    `The unit code is ${req.unitCode}  and we are in week ${req.weekNumber}`
  );
});
app.listen(8080);
```

```
app.use (function (req,res,next){
        …
        next();
});
```

app.use(callback):

callback: middleware function(s)

- application-level middleware
- the middleware function will be executed for every request to the app
- intercept all incoming requests and apply some pre-processing by using middleware functions

next():

represent the next middleware function to be executed

if not called, the request will be left hanging

POST request

Prerequisite:

npm init

npm install express

```
<html>
<body>
  <form action="/data" method="POST">
    User Name:
    <input type="text" name="username" /> </br>
    User Age:
    <input type="number" name="userage" /> </br>
    <button>Submit</button>
  </form>
</body>
</html>
```

<form action="/data" method= "POST">

action attribute: specifies where to send the data when a form is submitted (states the URL that will process the contents of a form

method attribute: specifies how the data in the form is sent ("POST")

- form data is appended inside the body of the HTTP request
- form data will not be visible in the URL

```
let express = require('express');
let app = express();
app.use(express.urlencoded({extended: true}));
app.use(express.json());

app.get('/', function (req, res) {
  res.sendFile(__dirname + '/index.html');
});
app.post('/data', function (req, res) {
  console.log(req.body.username);
  console.log(req.body.userage);
  res.send('Thank You')
})
app.listen(8080);
```

express.urlencoded():

- built-in middleware function
- parse incoming requests with urlencoded payloads

- based on body-paser

express.json():

- built-in middleware function
- parse incoming requests with JSON playloads
- based on body-parser

res.sendFile(path): transfer the file at the given path

req.body:

{username: 'Yidie Hu', userage: '25'}

The body parser middleware parses the data and make it available under the properties of req.body object

POST VS GET

Post: send data to a server to create/update a resource

Get: request data from a specified source

Webpage rendering

Rendering engine enables you to use static files in the application.

At runtime, the engine replaces variables in a template file with actual values and transforms the HTML file sent to the client

EJS (Embedded JavaScript):

- most popular view engine
- a simple templating language that lets you generate HTML markup with plain JavaScript

Prerequisite:

npm init

npm install express

npm install ejs

```
let express = require('express');
let app = express();

//configure Express app to handle the EJS engine
app.engine('html', require('ejs').renderFile);
app.set('view engine', 'html');

let db = [];
db.push({
    carId: 0,
    carMake: 'BMW',
    carModel: '735',
    carYear: 2014
});
db.push({
    carId: 1,
    carMake: 'Mercedes',
    carModel: 'C250',
    carYear: 2017
});
db.push({
    carId: 3,
    carMake: 'Audi',
    carModel: 'A6',
    carYear: 2019
});

app.get('/', function (req, res) {
    res.render('index.html', {username: "Guest", carDb: db});
});
app.listen(8080);
```

res.render (‘index.html’, {username: “Guest”, carDb: db});

can also be written as res.render (‘index’,…)

res.render (view, [locals],[callback]):

to render a view and send the rendered HTML string to the client

- view: a string that is the file path of the view file to render
- locals: an object whose properties define local variables for the view

  eg. username & carDb in index.html

- callback: function(err, html)

```
<body>
   <h1> Welcome
      <%= username%>
   </h1>
   </br>
   <h3>The available cars are:
      <table>
         <tr>
            <th>ID</th>
            <th>Maker</th>
            <th>Model</th>
            <th>Year</th>
         </tr>
         <% for ( let i =0; i< carDb.length;i++){ %>
            <tr>
               <td>
                  <%= carDb[i].carId %>
               </td>
               <td>
                  <%= carDb[i].carMake %>
               </td>
               <td>
                  <%= carDb[i].carModel %>
               </td>
               <td>
                  <%= carDb[i].carYear %>
               </td>
            </tr>
            <% }%>
    </table>
   </h3>
</body>
```

!!! By default, res.render needs the HTML files to be in a directory called 'views' (all HTML files should be in a folder 'views')

<%= %> tag: print out the value of the attribute

Static asset

app.use (express.static('public'));

serve images, CSS files, JavaScript files in a directory named public

express.static(root):

- built-in middleware function in Express

```
<head>
        <link rel= “stylesheet” href= “style.css”>
</head>

<body>
        <img src= “logo.png”>
</body>
```

style.css & logo.png: in the folder ‘public’

Week 5 MongoDB

Free document database management system

The equivalent of a record is a document or an object

Primary key: reserved field name _id (auto-generate a unique key for every document if not supplied)

BSON (Binary JSON) documents: binary representation of JSON documents

- JSON (JavaScript Object Notation): literals surrounded by {} and contain key-value pairs
- BSON has been extended to include more data types (eg. dates, timestamps, regular expressions, binary data)

Schema not required

MongoDB VS SQL

Database	Database
Collection	Table
Document	Row
Field	Column
Index	Index

Prerequisite:

npm init

npm install express

npm install ejs

npm install mongodb

(npm install morgan)

MongoDB CURD operations (Create Update Retrieve Delete)

Create:

.insertOne()

Update:

.updateOne()

.updateMany()

Retrieve:

.findOne()

.find()

Delete:

.deleteOne()

.deleteMany()

```
const express = require("express");
const mongodb = require("mongodb");
const morgan = require("morgan");
const ejs = require("ejs");
const app = express();
app.engine("html", ejs.renderFile);
app.set("view engine", "html");
app.use(express.static("public"));
app.use(express.urlencoded({ extended: false }));
app.use(morgan("common"));
app.listen(8080);

const MongoClient = mongodb.MongoClient;
const url = "mongodb://localhost:27017/";

let db;

MongoClient.connect(url, { useNewUrlParser: true }, function (err, client) {
  if (err) {
    console.log("Err  ", err);
  } else {
    console.log("Connected successfully to server");
    db = client.db("fit2095db");
  }
});

app.get("/", function (req, res) {
  res.sendFile(__dirname + "/views/index.html");
});

app.post("/addnewuser", function (req, res) {
  let userDetails = req.body;
  db.collection("users").insertOne({
    name: userDetails.uname,
    age: userDetails.uage,
    address: userDetails.uaddress,
  });
  res.redirect("/getusers");
});
```

```
app.get("/getusers", function (req, res) {
  db.collection("users").find({}).toArray(function (err, data) {
      res.render("listusers", { usersDb: data });
    });
});

app.get("/updateuser", function (req, res) {
  res.sendFile(__dirname + "/views/updateuser.html");
});

app.post("/updateuserdata", function (req, res) {
  let userDetails = req.body;
  let filter = { name: userDetails.unameold };
  let theUpdate = {
    $set: {
      name: userDetails.unamenew,
      age: userDetails.uagenew,
      address: userDetails.uaddressnew,
    },
  };
  db.collection("users").updateOne(filter, theUpdate);
  res.redirect("/getusers");
});

app.get("/deleteuser", function (req, res) {
  res.sendFile(__dirname + "/views/deleteuser.html");
});

app.post("/deleteuserdata", function (req, res) {
  let userDetails = req.body;
  let filter = { name: userDetails.uname };
  db.collection("users").deleteOne(filter);
  res.redirect("/getusers");
});
```

const url= “mongodb://localhost:27017/”;

mongodb://serveAdress: port

MongoDB default port: 27017

```
mongoClient.connect(url, {useNewUrlParser: true}, function (err, client){
        if (err){
                …
        }else{
                db=client.db(…);
        }
```

});

.connect (url, [options], callback):

url: (string) connection url for MongoDB

options: (object) optional options for insert command

callback (function): function (err, client)

- param1: Error object (gets value if an error occurs)
- param2: initialised db object (connects to the database)

mongodbClient.db(name):

name: (string) the name of the database

return a database object that allow you to access collections in the specified database

db.collection(“users”).insertOne(object);   **Create**

- references a collection named ‘users’
- creates the collection if it doesn’t exist

res.redirect(path): redirect to the URL derived from the specified path

db.collection(“users”).find({}).toArray(function(err,data){   **Retrieve**

                    …

});

Return all occurrences in the selection

Can be written as :

db.collection(“users”).find(query).toArray…

a query object should be used to filter the result of find() method

let query={name: ‘Alex’};

Regular Expressions:

let query={name: /^T/}; (start with T)

let query={name: /x$/}; (end with x)

Comparison Expression Operators

let query={age: {$gte:25}};

db.collection(“users”).findOne(query).toArray…

return the first document that satisfies the specified query criteria on the collection

db.collection(“users”).find(query).sort(…).limit(…).toArray…

```
let query = { age: { $gte: 25 } };
let sortBy={age:-1,name:1}
db.collection("week5table").find(query).sort(sortBy).toArray(function
(err, result) {
if (err) throw err;
console.log(result);
});
```

1: ascending

-1: descending

```
let query = { age: { $gte: 25 } };
let sortBy = { age: -1 }
db.collection("week5table").find(query).sort(sortBy).limit(5).toArray(fu
nction (err, result) {
if (err) throw err;
console.log(result);
});
```

return the top 5 document

db.collection(“users”).updateOne(filter, theUpdate); **Update**

db.collection.updateOne(filter, update, [options]):

- filter: the selection criteria for the update
- update: the modifications to apply

  eg. {$set: {age:31}}

  operators:

  $set: set the value of a field in a document

$inc: increment the value of the field by specified amount

$mul: multiply the value of the field by specified amount

- options: eg. upsert

```
db.collection("week5table").updateOne({ name: 'Tim' }, { $set: {age: 31 } },
{ upsert: true }, function (err, result) {
});
```

upsert: update & insert

default: false

true: create a new document if no document matches the filter

db.collection.updateMany (filter, update, [options]):

```
db.collection("week5table").updateMany({ name: /x$/ }, { $inc: { age:
2 } }, { upsert: true }, function (err, result) {
});
```

db.collection("users").deleteOne(filter);   **Delete**

db.collection("users").deleteOne(filter):

delete the first document that matches the filter

db.collection("users").deleteMany(filter):

```
db.collection("week5table").deleteMany({age: { $gte: 25 }}, function
(err, obj) {
console.log(obj.result);
});
```

Comparison Expression Operators for filter

{field: {operator: value}}

operators:

$eq: equivalent

$gt: greater than

$gte: greater than or equal to

$lt: less than

$lte: less than or equal to

$ne: not equivalent

Workshop:

```
db.collection("flights").find({$or:[{from: 'SA', to: 'SYD'},{from: 'SYD', to: 'SA'}]}).sort({cost:-1}).toArray(function (err, result){
   console.log(result);
});
```

Retrieve all fights between SA and SYD

Week 6 Mongoose

A JavaScript framework that is commonly used in a Node.js application with a MongoDB database

An object data modelling (ODM) library that provides a modelling environment for your collections

Enforces structure as needed while still keeping the flexibility & scalability of MongoDB

Includes built-in typecasting, validation, query building, business logic hooks and more

Uses MongoDB driver to interact with MongoDB storage

Each **schema** maps to a MongoDB collection and defines the shape of the documents (structure of the document, default values, validators etc) within the collection

Mongoose SchemaTypes:

- String
- Number
- Date
- Buffer
- Boolean
- Mixed
- ObjectId
- Array
- Decimal128
- Map

models folder: book.js & author.js

author.js

```
const mongoose = require('mongoose');
let authorSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    name: {
        firstName: {
            type: String,
            required: true
        },
        lastName: String
    },
    age: {
        type: Number,
        validate: {
            validator: function (ageValue) {
                return ageValue >= 10 && ageValue <= 110;
            },
            message: 'Age should be a number between 10 and 110'
        }
    },
//   age: { type: Number, min: 5, max: 20 },
    created: {
        type: Date,
        default: Date.now
    }
});
module.exports = mongoose.model('Author', authorSchema);
```

required: true

the field is mandatory

```
validate:{
        validator: function (…){
                return …;
        },
        message: '…';
}
```

- validator: Boolean function

  return true/false;
- message: a string represents the output that will be printed out when an invalid value is detected

age: {type: Number, min: 5, max: 20} (using min & max properties)

default: Date.now

default value of the field if no value is provided

module.exports=mongoose.model (‘Author’, authorSchema);

mongoose.model(param1, param2):

model()constructor

create a new Model for the schema

- a Mongoose model is a wrapper on the Mongoose schema
- a Mongoose model provides an interface to the database for creating, querying, updating, deleting records etc
- a document is an instance of a Mongoose model

param1:

- (string) the singular name of the collection the model is for (eg. ‘Author’ for authors collection)
- also the name of the model

param2:

(mongoose.Schema) a reference to the Schema

module.exports: export the model

book.js

```
const mongoose = require('mongoose');
let bookSchema = mongoose.Schema({
   _id: mongoose.Schema.Types.ObjectId,
   title: {
      type: String,
      required: true
   },
   isbn: String,
   author: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Author'
```

```
    },
    created: {
        type: Date,
        default: Date.now
    }
});
module.exports = mongoose.model('Book', bookSchema);
```

ref: ‘Author’

indicates the name of the schema

app.js

```
const mongoose = require('mongoose');
const Author = require('./models/author');
const Book = require('./models/book');
mongoose.connect('mongodb://localhost:27017/libDB', function (err) {
    if (err) {
        console.log('Error in Mongoose connection');
        throw err;
    }
     console.log('Successfully connected');

    let author1 = new Author({
        _id: new mongoose.Types.ObjectId(),
        name: {
            firstName: 'Tim',
            lastName: 'John'
        },
        age: 80
    });
    author1.save(function (err) {
        if (err) throw err;
        console.log('Author successfully Added to DB');

        var book1 = new Book({
            _id: new mongoose.Types.ObjectId(),
            title: 'FIT2095 Book ',
            author: author1._id,
            isbn: '123456',
        });
        book1.save(function (err) {
            if (err) throw err;
            console.log('Book1 successfully Added to DB');
        });

        var book2 = new Book({
            _id: new mongoose.Types.ObjectId(),
            title: 'MEAN Stack with FIT2095',
            author: author1._id
        });
        book2.save(function (err) {
            if (err) throw err;
            console.log('Book2 successfully add to DB');
        });
    });
});
```

```
mongoose.connect('mongodb://localhost:27017/libDB', function (err){
        …
}
```

connect to MongoDB with mongoose.connect() method

URL syntax: mongodb://ServerAddress: Port//DbName

```
author1.save(function (err){
        var book1=new Book({
        …
        });
        book1.save(function (err){
                …
        });
        var book2=new Book({
                …
        });
        book2.save(function (err){
                …
        });
});
```

save():

a method on a Mongoose document

asynchronous method, so return a promise that you can await on

Creating & saving book1, book2 have been implemented inside the callback function of author1.save function:

Node.js is asynchronous, book1, book2 require the author's ID. So have to be after the save operation of the author is done

Mongoose models CURD operations

Update:

Model.updateOne()

Model.updateMany()

Model.findOneAndUpdate()

Model.replaceOne()

Model.findOneAndReplace()

Retrieve:

Model.findOne()

Model.find()

Model.findById()

Delete:

Model.deleteOne()

Model.deleteMany()

Model.findOneAndDelete()

Model.findOneAndRemove()

Model.findByIdAndDelete()

Model.findByIdAndRemove()

```
Model.findOne({'name.firstName':'Tim'}, 'age', function (err, doc) {
});
```

Output: { _id: new ObjectId("618fa7421eb5d6e79251d0c2"), age: 80 }

doc: object

Only show _id & age

```
Author.find({ 'name.firstName': 'Tim' }, 'age', function (err, docs) {
console.log(docs);

});
```

docs: array

where clause

```
Author.where({ 'name.firstName':
/^T/ }).where('age').gte(25).lte(35).limit(10).sort('age').exec(function
(err, docs) {
console.log(docs);

});
```

.exec(function (err, docs)): execute a query

Populate()

```
Book.find({}).populate('author').exec(function (err, data) {
console.log(data);
});
```

Book.find ({}).populate(‘author’).exec(function(err,data){

…

}

.populate(fieldName):

Allow you to reference documents in other collections

fieldname: the field in this collection that references to the other collection

Week 7 RESTFul API

Representational State Transfer

An application programming interface that uses HTTP requests to perform the CURD operations on data

<u>Stateless architecture</u>:

- no information is retained by either sender or receiver
- dependent only on the input parameters that are supplied

REST architecture composition:

- clients
- servers
- resources
- request methods (HTTP operations eg. GET, PUT, POST…)

models folder:

actor.js & movie.js

routers folder:

actor.js & movie.js

actor.js

```
const mongoose = require('mongoose');
const Actor = require('../models/actor');
const Movie = require('../models/movie');
module.exports = {
   getAll: function (req, res) {
       Actor.find(function (err, actors) {
           if (err) {
               return res.status(404).json(err);
           } else {
               res.json(actors);
           }
       });
   },
   createOne: function (req, res) {
       let newActorDetails = req.body;
       newActorDetails._id = new mongoose.Types.ObjectId();
```

```
    let actor = new Actor(newActorDetails);
    actor.save(function (err) {
      res.json(actor);
    });
  },
  getOne: function (req, res) {
    Actor.findOne({ _id: req.params.id }).populate('movies').exec(function (err, actor) {
        if (err) return res.status(400).json(err);
        if (!actor) return res.status(404).json();
        res.json(actor);
      });
  },
  updateOne: function (req, res) {
    Actor.findOneAndUpdate({ _id: req.params.id }, req.body, function (err, actor) {
      if (err) return res.status(400).json(err);
      if (!actor) return res.status(404).json();
      res.json(actor);
    });
  },
  deleteOne: function (req, res) {
    Actor.findOneAndRemove({ _id: req.params.id }, function (err) {
      if (err) return res.status(400).json(err);
      res.json();
    });
  },
  addMovie: function (req, res) {
    Actor.findOne({ _id: req.params.id }, function (err, actor) {
      if (err) return res.status(400).json(err);
      if (!actor) return res.status(404).json();
      Movie.findOne({ _id: req.body.id }, function (err, movie) {
        if (err) return res.status(400).json(err);
        if (!movie) return res.status(404).json();
        actor.movies.push(movie._id);
        actor.save(function (err) {
          if (err) return res.status(500).json(err);
          res.json(actor);
        });
      })
    });
  }
};
```

res.status(code): set the HTTP status for the response

- 400: bad request (client error)
- 404: not found (client error)
- 500:internal server error (server error)
- 200: OK

res.json ([body]): send a JSON response

Actor.findOneAndUpdate({_id: req.params.id}, **req.body**, function (err,actor){

…

});

If no operations specified, treated as set operation

`

movie.js

```
var Actor = require('../models/actor');
var Movie = require('../models/movie');
const mongoose = require('mongoose');
module.exports = {
   getAll: function (req, res) {
      Movie.find(function (err, movies) {
         if (err) return res.status(400).json(err);
         res.json(movies);
      });
   },
   createOne: function (req, res) {
      let newMovieDetails = req.body;
      newMovieDetails._id = new mongoose.Types.ObjectId();
      Movie.create(newMovieDetails, function (err, movie) {
         if (err) return res.status(400).json(err);
         res.json(movie);
      });
   },
   getOne: function (req, res) {
      Movie.findOne({ _id: req.params.id }).populate('actors').exec(function (err, movie) {
            if (err) return res.status(400).json(err);
            if (!movie) return res.status(404).json();
            res.json(movie);
         });
   },
   updateOne: function (req, res) {
      Movie.findOneAndUpdate({ _id: req.params.id }, req.body, function (err, movie) {
         if (err) return res.status(400).json(err);
         if (!movie) return res.status(404).json();
         res.json(movie);
      });
   }
};
```

Model.create(docs):

Creates a new document

does new Model(doc).save() for every doc in docs

app.js

```
const express = require('express');
const mongoose = require('mongoose');
const actors = require('./routers/actor');
const movies = require('./routers/movie');
const app = express();
app.listen(8080);
app.use(express.json());
app.use(express.urlencoded({ extended: false }));

mongoose.connect('mongodb://localhost:27017/movies', function (err) {
    if (err) {
        return console.log('Mongoose - connection error:', err);
    }
    console.log('Connect Successfully');
});

app.get('/actors', actors.getAll);
app.post('/actors', actors.createOne);
app.get('/actors/:id', actors.getOne);
app.put('/actors/:id', actors.updateOne);
app.post('/actors/:id/movies', actors.addMovie);
app.delete('/actors/:id', actors.deleteOne);

app.get('/movies', movies.getAll);
app.post('/movies', movies.createOne);
app.get('/movies/:id', movies.getOne);
app.put('/movies/:id', movies.updateOne);
```

`

HTTP methods

1. GET:

- request data from the source
- no side effect

2. POST:

- request server to create a resource in the database (mostly when a web form is submitted)
- <u>non-idempotent</u>: multiple requests have side effects of creating the same resource multiple times

3. **PUT:**

- request server to update a resource <u>or create the resource if it doesn't exist</u>
- <u>idempotent:</u> multiple will always produce the same result

4. **DELETE:** request the resources or instances should be removed from the database

Week 8 Angular

A front-end platform and framework for building client applications in HTML and TypeScript (a strict superset of JavaScript)

Prerequisite:

sudo npm install -g @angular/cli

install CLI (Command Line Interface)

used for:

- ng new
- ng generate
- ng serve

ng new w8ang

- create a new Angular application (a folder named ‘w8ang’ will be created with several folders & files)
- install all required packages into ‘node_modules’ subfolder

cd w8ang

get into the folder

to run:

ng serve

build the application and start a web server (CLI comes with a simple HTTP server)

address: http://localhost:4200 (4200: default port number for ‘ng serve’)

Data binding:

- property binding
- event binding
- two-way binding

Property & event binding code

app. component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'FIT2095';
  counter: number = 0;

  incCounter() {
    this.counter++;
  }
  resetCounter(){
    this.counter=0;
  }
}
```

counter: number=0;

TypeScript has strong static typing feature:

variables are declared to be a specific type and the compiler will check the validity of their values

app.component.html

```
<p>
  Welcome to {{title}}!!
</p>
<p>
  The count is: {{counter}}
</p>
<button (click)="incCounter()">Click Me</button>
<button (click)="resetCounter()">Reset</button>
```

{{title}}:

Inside {{ }}: the name of a component property

Property binding:

- Passing data from the component class and set the value of a given element in the view
- One-way: from component to HTML

- Allow you to control element property values from the component and change them whenever needed

<button (click)= “intCounter()”> Click Me </button>

Bind click event to intCounter() function

Event binding:

- Listen for certain events (eg. click, double-click) and call the component’s method whenever the event occurs
- One-way: from HTML to component

Two-way binding code

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

!!! Must import FormModule and add it to NgModule’s imports to use the ngModel directive in two-way binding

app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  message = "Default Value";
  resetMsg() {
    this.message = "";
  }
}
```

app.component.html

```
<input [(ngModel)]="message" />
<br>
<br>
<div>Your Message: {{message}}</div>
<br>
<button (click)="resetMsg()" >Reset Message </button>
```

[(ngModel)]= “message”

Each time the user updates the input element (bound to ‘message’), Angular set the component variable ‘message’ with the new value and re-evaluates all the expression that use ‘message’

[( )]: two-way binding syntax (“banana-in-a-box” syntax)

two-way binding: Both display a data property and update that property when user makes changes

built-in structural directives (*ngIf, *ngFor)

app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'for';
  data:any = [];
  item = "";
  newItem() {
    this.data.push(this.item);
  }
  clearItems() {
    this.data = [];
  }
}
```

app.component.html

```
Enter Your Items:
<input [(ngModel)]="item" />
<br>
<button (click)="newItem()">Add Item</button>
<button (click)="clearItems()">Clear Items</button>
<br>
<div *ngIf="data.length>0">You have {{data.length}} items</div>
<div *ngIf="data.length==5">Caution: 5 items in your array</div>
<br>
<table>
  <tr>
    <th> # </th>
    <th> Item </th>
  </tr>
  <tr *ngFor="let item of data; let i = index">
    <td>{{i+1}}</td>
    <td>{{item}}</td>
  </tr>
</table>
```

*ngIf= “data.length>0”

A directive used when you want to display or remove an element based on a condition

Only display when condition is true

*ngFor = “let item of data; let i=index”>

A structural directive that renders a template for each item in the list

Week 9 AngularII

Dependency Injection (DI)

A software design pattern that deals with how components get hold of their dependencies (services/objects that a class needs to perform its function)

Angular injector subsystem: create components, resolve dependencies, provide them to other components as requested

The list of dependencies of a component can be found in the constructor parameter types

Prerequisite:

sudo npm install -g @angular/cli

ng new movieAng

ng build

- the project will be served using node HTTP server, so we need to build the Angular project each time a file at the client side gets changes
- compiles Angular project and generate new HTML and JS files in ‘dist’ folder at the root of the project

  ‘dist’ folder: a source for the client UI (any request other than the CURD operation should be redirected to this folder)

ng generate component Actor / ng g c Actor

create a new component

- components: classes that interact with the HTML file of the component
- component: template(view)+a class(TypeScript code)+metadata
- component controls view (HTML)

ng generate service database / ng g s database

create a new service

- service: an object that gets instantiated only once during the lifetime of an application
- service contains methods that contain data throughout the life of an application
- benefit: organise & share methods, models or data with different components of Angular application

to run:

ng build --watch

automate the process of building the Angular project with each change of its files

node server.js

replace the **app.component.html** code with the selector (i.e. tag) of the actor component.

```
<app-actor></app-actor>
@Component({
  selector: 'app-actor',
  templateUrl: './actor.component.html',
  styleUrls: ['./actor.component.css']
})
```

if HTML contains <app-actor></app-actor>, then Angular inserts an instance of ActorComponent view between those tags.

selector:

tell Angular to create and insert an instance of this component wherever it finds the corresponding tag in the template HTML

add '**DatabaseService**' to the list of providers in **app.module.ts**.

```
providers: [DatabaseService],
```

tell the app module that this new service should be available application-wide

import the Angular HttpClientModule in the root AppModule app.module.ts.

```
imports: [BrowserModule, FormsModule, HttpClientModule]
```

the database service depends on the HTTP service in its communications with the RESTFul servers

add this line to list of imports at the list of imports in the root AppModule app.module.ts.

```
import { FormsModule } from "@angular/forms";
```

open database.service.ts and import both **HttpClient** and **HttpHeaders** from angular http package:

```
import { HttpClient, HttpHeaders } from "@angular/common/http";
```

In the service's constructor, add a reference to the **httpClient**:

```
constructor(private http: HttpClient) {}
```

inject the httpClient to the service and make it available with reference http

any method inside the service can make http requests by calling this.http

```
const httpOptions = {
headers: new HttpHeaders({ "Content-Type": "application/json" }),
};
```

httpOptions is an object that specifies a set of options for the request such as the format of the body (JSON)

import the service to app.component.ts

```
import { DatabaseService } from "../database.service";
```

Similar to injecting the HTTP service to the database service, we will use the constructor to provide a reference to the database service in app.component.ts

```
constructor(private dbService: DatabaseService) {}
```

inject the DatabaseService to the actor component and make it available with reference dbService

dbService: a reference to a dependency which is class represents a service

DatabaseService code:

```
getActors() {
return this.http.get("/actors");
}
```

```
getActor(id: string) {
let url = "/actors/" + id;
return this.http.get(url);
}
```

```
createActor(data) {
return this.http.post("/actors", data, httpOptions);
}
```

```
updateActor(id, data) {
let url = "/actors/" + id;
return this.http.put(url, data, httpOptions);
}
```

```
deleteActor(id) {
let url = "/actors/" + id;
return this.http.delete(url, httpOptions);
}
```

```
onGetActors() {
   this.dbService.getActors().subscribe((data: any) =>{
     this.actorsDB = data;
   });
 }
```

the output of getActor() is observable

Observables: things you want to observe and take action on

You can define a function for publishing values (observable objects) but the function is not executed until a customer subscribes to it. The subscribed customer will receive notifications until the function completes or they unsubscribe

```
onSaveActor() {
   let obj = { name: this.fullName, bYear: this.bYear };
   this.dbService.createActor(obj).subscribe(result => {
     this.onGetActors();
```

```
  });
 }
```

```
ngOnInit() {
   this.onGetActors();
   //added
   this.onGetMovies();
 }
```

ngOnInit():

initialise the component

called shortly after creating the component

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { ActorComponent } from './actor/actor.component';

import { FormsModule } from "@angular/forms";
import { DatabaseService } from "./database.service";
import { HttpClientModule } from "@angular/common/http";

@NgModule({
  declarations: [
    AppComponent,
    ActorComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [DatabaseService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

declarations:

declare components, directives, pipes that belong to the current module

imports:

make exported declarations of other modules available in current module

providers:

create services that can be used in the global collection of services (application-wide)

bootstrap:

set the root component (the main application view) which host all other application views

Week 11 Socket.io

A library that enables real-time, bidirectional and event-based communication between the browser and the server

Consist of:

- Node.js server
- JavaScript client library for the browser

Features:

- Reliability
- Auto-reconnection support

  Disconnected client will try to reconnect forever until the server is available
- Disconnection detection

  Heartbeat mechanism
- Binary support

  Any serialised data structure can be emitted
- Multiplexing support
- Room support

Prerequisite:

Server side:

```
npm install socket.io
npm install express
```

client side:

```
ng new chatapp
npm install socket.io-client
npm install @types/socket.io-client
```

app.component.html

```
<div id="chatContainer">
```

```
  <div class="chatform">
    <div class="form-group">
      <label for="userName">Your Name</label>
      <input type="text" class="form-control" id="userName" placeholder="Your name" [(ngModel)]="userName">
    </div>
    <div class="form-group">
      <label for="message">Message</label>
      <input type="text" class="form-control" id="message" placeholder="Your message" [(ngModel)]="messageText">
    </div>
    <button (click)="sendMessage()" type="submit" class="btn btn-primary">Send </button>
  </div>
  <div class="container" *ngFor="let item of messages.slice().reverse() ">
    <p>From: {{item.userName}}</p>
    <div>
      <h3>{{item.msg}}</h3>
    </div>
    <span class="time-right">{{item.timeStamp}}</span>
  </div>
</div>
```

app.component.ts

```
import { Component } from '@angular/core';
import { io } from 'socket.io-client';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'chatapp';

  userName:string="";
  messageText: string="";
  messages: Array<any> = [];
  socket:any;
  constructor() {
    this.socket = io();
  }
  ngOnInit() {
    this.messages = new Array();
    this.listen2Events();
  }
  listen2Events() {
    this.socket.on("msg", (data:any) => {
      this.messages.push(data);
    });
  }
  sendMessage() {
    let info={messageText: this.messageText, userName: this.userName};
    this.socket.emit("newMsg", info);
    this.messageText = "";
  }
}
```

socket:any

create an instance of the socket.io client

```
constructor(){
        this.socket=io();
}
```

Send a connection request to the server

```
this.socket.on (event, function (param){
        …
});
```

The client listens to messages from server

this.socket.emit("event", message);

client send a message to the server

server.js

```
let express = require("express");
let path = require("path");
let app = express();
let server = require("http").Server(app);
let io = require("socket.io")(server);
let port = 8080;
app.use("/", express.static(path.join(__dirname, "dist/chatApp")));

io.on("connection", socket => {
  console.log("new connection made from client with ID="+socket.id);
  socket.on("newMsg", data => {
    io.sockets.emit("msg", { msg: data.messageText, userName: data.userName,timeStamp: getCurrentDate() });
  });
});
server.listen(port, () => {
  console.log("Listening on port " + port);
});

function getCurrentDate() {
  let d = new Date();
  return d.toLocaleString();
}
```

```
let path = require ('path');
```

path module provides utilities for working with file and directory paths

```
let io=require (“socket.io”)(server);
```

create an instance of Socket.io server

```
io.on (“connection”, function (socket){
});
```

Wait for connection and function will be called after the “connection” event happens successfully

io.on(param1,param2):

param1: (string) the name of the event

param2: (object which is callback function): the function to call when the event happens

socket (param): a socket to a new client

```
socket.on (“newMsg”, function (data){
        io.sockets.emit(“msg”, {…});
});
```

socket.on(event, callback):

wait for event and after listening to the event, execute callback function

io.sockets.emit (param1,param2);

param1: (string) the name of the event

param2: (any serializable data structure) the message to all connected sockets (clients)

```
// sending to the client
socket.emit('hello', 'can you hear me?', 1, 2, 'abc');
```

eg. send to the new client

```
// sending to all clients except sender
socket.broadcast.emit('broadcast', 'hello friends!');
```