# Unit-4

## File Based Systems

> Program defines and manages its own data.

- Limitations :
  - Separation
    - each program maintains its own data and there is increase in data redundancy.
    - Connecting these separate files requires even more error prone methodologies.
  - Duplication
    - As information is separate for each program there is huge scope for data redundancy.
    - Moreover if data is to be updated at one point it must be updated everywhere which increases chances of errors.
  - Program and data dependance
    - The data is tightly tied to the application/program. Hence any changes in data will require calibration in the application code. This makes the entire system inflexible.
  - Fixed queries
    - queries are hardcoded into the program which restricts its usage
  - Proliferation of application programs
    - As we scale the application we need to have dedicated programs to maintain the data. In such a file based approach, we will need a lot of them.

## MongoDB

> MongoDb is an open-source document-oriented database designed with scalability and development agility in mind.

- How is this different from a traditional relational database? We store objects in JSON like documents with dynamic schemas.

> Schema : Structure of defining data in a database. Mongo is a schema-less database as we can define our own for each document.

- `collections` : analogous to `tables`
- `documents` : analogous to `row/entry`
- `embedded document` : analogous to `table join`
- `column` : analogous to `field`
- MongoDB doesn't enforce any particular schema onto the developers but it still required a schema while adding to the database.
- This allows projects with evolving datasets to be implemented with ease.
- Data stored in `BSON` format.
- Mongo requires overall lesser indices than an RDBMS
  - This is because join operators

## Create Read Update Delete

- Create:

```
db.collection.insert<document>)
// creates the collections if it doesn't exist
// if _id isn't specified, mongo automatically adds it before insertion
// can be used to insert multiple documents
// set the ordered:true param to insert in order. If error occurs it'll halt further insertions. True by default.
db.collection.save(<document>)
// if _id is not specified , this is equivalent to an insert command
// if _id is specified, this is equivalent to an update command
db.collection.update(<query>,<update>,{upsert:true})
// updated the document that fulfills the query
// upsert: if true it inserts a new document if none match the query
// defaults to false
```

- Read:

```
db.collection.find(<query>,<projection>)
// to return all omit query and pass {} as param
// projection specifies the fields to return
db.collection.findOne(<query>,<projection>)
// returns the first document matching the query
```

- Update:

```
db.collection.update(<query>,<update>,<options>)
```

- Delete

```
db.collection.remove(<query>,<justOne>)
// removes the document from the collection
```

## Node & Mongo

- NodeJS driver used to connect to MongoDB instance
- Node uses promises to interface with mongo
- Third party modules like mongoose can be used to extend ODM capabilites.

## Connecting to Mongo

- Connect using IP Address of the mongo server
- Database will be created if it doesn't exist

```
const {MongoClient} = require('mongodb')
const path = "mongodb://127.0.0.1:27017"

let connector
MongoClient.connect(path).then((connection)⇒{
   connector = connection
   let db = connector.db("deebee") // creates db if not existing
   return db.createCollection("NewCollect")// creates collection if not existing
}).then(()⇒{
   const data = {field1: "whatever",field2:"whatever"}// inserted everytime this script is run
   let collect = connector.db("deebee").collection("NewCollect")
   return collect.insertOne(data)
}).then(()⇒{
   // projection will enable or disable display of fields
   // toArray takes the documents returned by the find function and turns them into an array of documents.
       connector.db("deebee").collection("NewCollect").find({}, { projection: {_id:0,name: 1,
           phone :1} }).toArray(function(error, result)
               {
                   if (error) throw error;
                   console.log(result);

                   connector.close();
               })
}).then(()⇒{
   let query = {uname:"@vorrtt3x"}
   let newvals = {qualities:"gigachad"}
   connector.db("deebee").collection("NewCollect").updateOne(query,newvals,(error,result)⇒{
       if (error) throw error
       console.log(result)
       connector.close()
   })
}).then(()⇒{
   let query = {name:"vorrtt3x"}
   connector.db("deebee").collection("NewCollect").deleteOne(query)
}).then(()⇒{
   connector.db("deebee").collection("NewCollect").drop()
}).then(()⇒{
   connector.close()
})
   .catch((error)⇒{
       console.log("Whups",error)
   })
```

### Function Return Values(not in portions):

- `connect` : returns a promise
- `insert` : `WriteResult` object
- `find` : returns a cursor to the documents
- `deleteOne` : `true` if operation ran with `writeconcern` and `false` if `writeconcern` is disabled. `deletedCount` containing number of elements
- `drop` : `true` on success `false` otherwise
- `update` : `WriteResult` object

## Events

- Help in maintaining concurrency even though Node is a single threaded application.

  > Blocking Process: any process on the call stack that is slow and hinders the proceedings of the application.
  > Async Callbacks: We give a functiona call back and run it later

- Event Loop : enables async non-blocking IO operations
- Whenever a task is completed it fires a corresponding event that executes the event listeners.

- Event loop continuously checks the event queue for events pending. When found its associated callbacks(observer functions) are executed.
- Event Emitter : Node module that lets us create event driven APIs
- Whenever an API event is emitted it is pushed onto the event queue
- The event loop waits till the call stack is empty and then pushes this event onto the stack.
- Event emmitters register listener functions that are triggered when the particular event is emitted

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();

myEmitter.on('customEvent', () => {
  console.log('Custom event emitted!');
});

myEmitter.emit('customEvent');
```

- EventEmitter methods:
  - addListener(event,listener)
  - on(event,listener)
  - once(event,listener)
  - removeListener(event,listener)
  - removeAllListeners([event])
  - setMaxListeners([event])
  - listeners(event)
  - emit(event,[arg1],[arg2],[...])
- Class Methods:
  - listenerCount(eventName): Returns number of events for given event
- Events:
  - newListener
  - removeListener

## Node-React Integration

- 2 Approaches:
  - Client Side: by writing HTML and executing on browser
  - Server Side: by writing React Apps and executing on server

### React Routing

- Traditional server side routing performs a `GET` request to the server everytime a link is clicked. This discards the old page and entirely refreshes the entire screen.
- Modern Single Page Applications(SPAs) have a need for fewer page refreshes and hence implement what is known as client side routing which is handled by the JS loaded onto the client and there is not request made to the server.

```
<a href="fun.com">Click Here</a>   // traditional serverside linking
<Router>
<Link to=“/”>Home</Link>
<Link to=“/contact”>Contact</Link>
<Link to=“/about”>About Us</Link>
  <Routes>
    <Route path="/" element={<Home/>}/>
    <Route path="/about" element={<About/>} />
    <Route path="/contact" element={<Contact/>} />
  </Routes>
</Router>  // React Routing
```

```
// To style the links
import styled from ‘styled-components’;
import {NavLink} from ‘react-router-dom’;

export const StyledLink = styled(NavLink)`

margin-right: 5px;
`;
export default StyledLink;
```

## Express & REST

- A service is software that exposes functionality via an API.
- Services can be shared among multiple clients

- Webservices are meant to be accessed by software

> Web Service : Software system designed to support interoperable machine-to-machine interactions over a network

- Characteristics:
  - Made for machine-to-machine interaction
  - Platform independent
  - Leverage the WWW
  - Loose Coupling
  - Must communicate over a network

## REpresentational State Transfer(REST)

- Design Principles for web services
- REST Constrains:
  - Client-Server Independence
  - Stateless : requests carry all context required in a client-server communication. Nothing stored on server
  - Cacheable: data specified as cacheable. HTTP responses are cacheable by clients
  - Uniform Interface: All requests made using uniform HTTP interface
  - Layered System : composed of heirarchical layers. Each components can't see beyond its layer
  - Code on Demand: allows client to download code as scripts or applets
- HTTP methods:
  - GET
  - POST
  - PUT
  - DELETE
  - HEAD
  - OPTIONS

## RESTful Process Control flow

### Design

1. Identify Resources that expose themselves on the network
2. Map these to Uniform Resource Identifiers(URI)
3. Use HTTP Interface to facilitate communication
4. Implement authentication and security
5. Resource Representations - JSON/XML and alternate representations
6. Metadata provisions

### Implementation

1. Parse the request:
   1. Identify the resource from the URI
   2. Map URI variables to resource variables
   3. Identify the HTTP method
   4. Read Resource representation
2. Authenticate the user
3. Perform processing on the resource retreived
4. Generate response:
   1. Proper status code
   2. Description
   3. Resource representation if applicable

```
Request:
GET /news/ HTTP/1.1
Host: example.org
Accept-Encoding: compress,
gzip
User-Agent: Python- httplib2

Response:
HTTP/1.1 200 Ok
Date: Thu, 07 Aug 2008 15:06:24 GMT
Server: Apache
ETag: "85a1b765e8c01dbf872651d7a5"
Content-Type: text/html
Cache-Control: max-age=3600
```

## Express

- Fast, unopinionated, minimal backend framework for Node
- Express is built on top of node's `http` module which provides rich abstraction and makes it easier to create web servers

```
const xpres = require('express')
const app = xpres()
app.get('/',(req,res)⇒{
        res.send("<strong>EDM was a mistake</strong>")
})
app.listen(8008,()⇒{
        console.log("Server running on 8008s")
})
```

## Routing

- Determining how an application responds to client requests.
- `app.METHOD(PATH,HANDLER)`
- `Handler` is executed when the route is matched.
- Serving the requested info to the client is *routing*
- The handler is passed `(req,res)`
- Incase of multiple requests to the same route, express/Node always take the first one.
- If we need to route using patterns add the generic one only *after* ass the specific ones
- Components of a route:
    - HTTP method
    - Route/path
    - Handler callback

```
app.all() // to match all kinds of methods
app.get("/*.test") // regex path matching
```

## Route Params

- parameters can be supplied to the request object via the path

```
app.get("/data:name",(req,res)⇒{
        console.log(req.params.name)
})
```

- more generic routes must be declared *after* the specific ones

```
app.get('/users/:id', (req, res) ⇒ {
    const userId = req.params.id // returns the requested id
    res.send(`User ID: ${userId}`)
})
app.get('/search', (req, res) ⇒ {
  res.send(`Search query: ${query}`)
const query = req.query.q // returns search query
})
app.post('/users', (req, res) ⇒ {
  const userName = req.body.name; // body contains json request
  res.send(`User created with name: ${userName}`)
})
app.get('/headers', (req, res) ⇒ {
  const userAgent = req.headers['user-agent'] // returns HTTP request headers
  res.send(`User-Agent header: ${userAgent}`)
})
app.all('/example', (req, res) ⇒ {
  const method = req.method  // returns http method
  res.send(`Request method: ${method}`)
})
app.get("/url",(req, res) ⇒ {
  const path = req.url // returns request url
  res.send(`Requested path: ${path}`)
})
app.get("/path",(req, res) ⇒ {
  const path = req.path // returns url without the query string
  res.send(`path: ${path}`)
})
app.get("/ogrl",(req, res) ⇒ {
  const originalUrl = req.originalUrl // returns url as received by the server without alterations by any middlewares
  res.send(`Original URL: ${originalUrl}`)
})
app.listen(8008,()⇒{
```

```
    console.log("Server running fine")
})
```

```
req.send() // if content is object or array its converted to json and sent. Content-type is preserved.
req.sendFile() // sends a specified file. type is guessed from file extension
req.json() // converts the given args into json string using json.stringify().type is json
req.status()
```

- URL binding : assigning certain functions to certain routes
- Static routing: hard set paths that can't be changed.
- Dynamic routing: parametrized paths.

```
app.get("/data/:name/:class", ... )
```

## Error Handling

- How express catches errors and exceptions.
- Error response :
  - Statuscode from `err.status`
  - Status message as per the code
  - Body : html of status code message
  - Any headers in `err.headers`
- Express has a built in error handler
- if we pass the error object into the next function there is no need for any external error handlers .

## Middleware

- Functions that take in request, response and next function.
- next function returns the next middleware in the stack
- if there is no next function, control won't pass to next middleware in the chain and request will hang.
- functions:
  - make changes to request and response
  - execute any code
  - end request-response cycle
  - call next middleware
- Types:
  - App level
  - Router level
  - Third Party
    - example `body-parser`
  - builtin
    - example `express-static`
  - Error
- `use()` is used to mount the middleware onto the application
- `body-parser` : third-party middleware to parse json and url-encoded data
- Templating Engines: replace variable names with content at runtime and sent to client.
  - `pug`
  - `EJS`
  - `mustache`

```
// bleh.pug
html
  head
    title Test
  body
    h1 Welcome to #{appName}
    h1 Firstname : #{firstName}
    h1 Lastname : #{lastName}
```

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

app.set('view engine', 'pug');

app.set('views', './puggy');

app.get('/', (req, res) => {
  res.render('bleh', {appName: 'Pug Test',firstName: 'Ben',lastName: 'Chode' });
});
```

```
const PORT = 8008;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

## File Upload

- `multipart/form-data`
- middlewares : `multer`, `connect`, `bodyparser`
- Form Processing : Busboy,Formidable
- `req.files` used to handle file uploads
- `express-fileupload` simple middleware for uploading files
- Assume input field has name `test`
- Then object req.files.test contains:
  - `name` : name of file
  - `mv` : function to move file elsewhere
  - `mimetype`
  - `data` : buffer representation of file. Empty if `useTempFiles` is true
  - `tempFilePath` : temp path if `useTempFiles` is true
  - `size` : size in bytes
  - `md5` : checksums