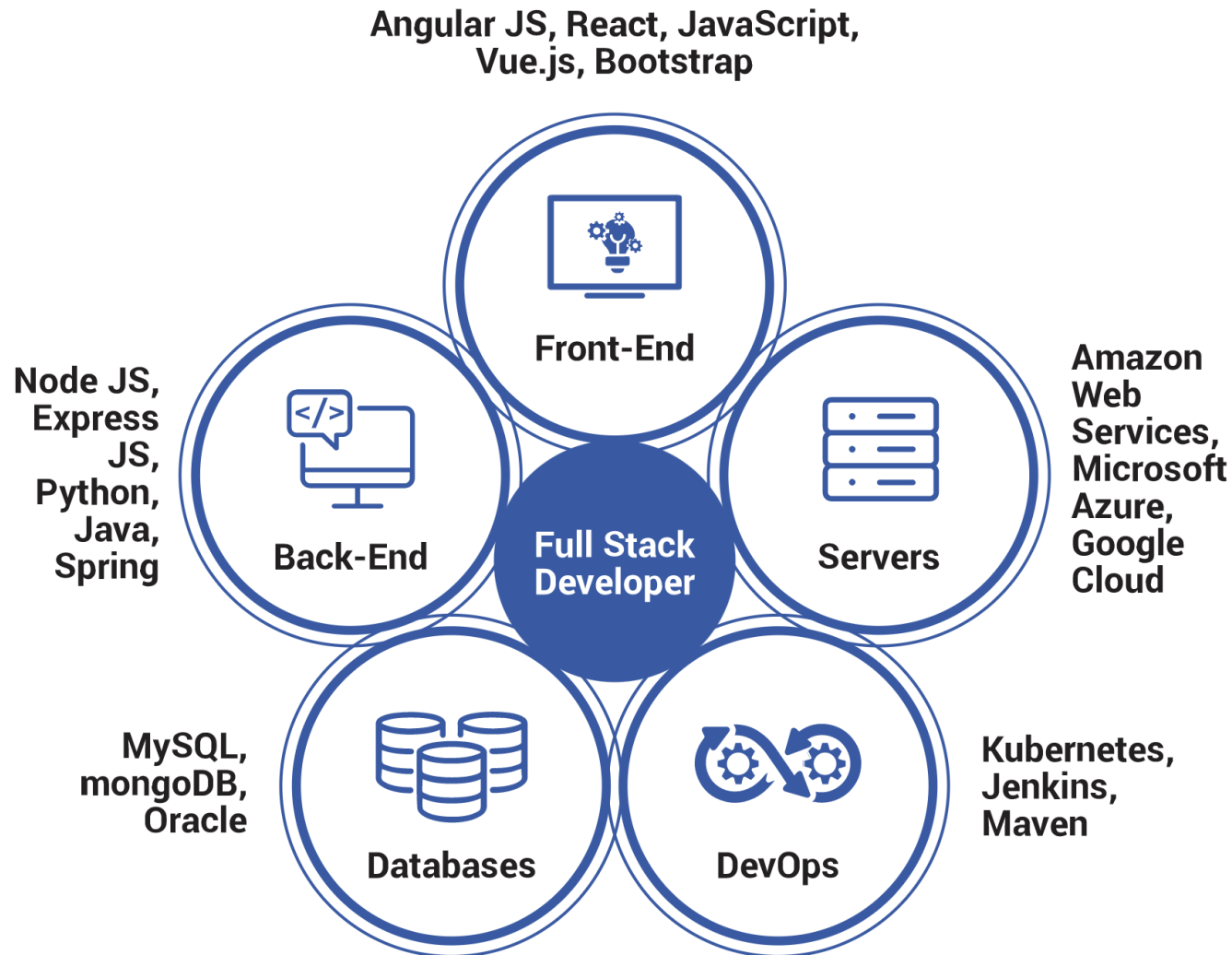


Chapter 1

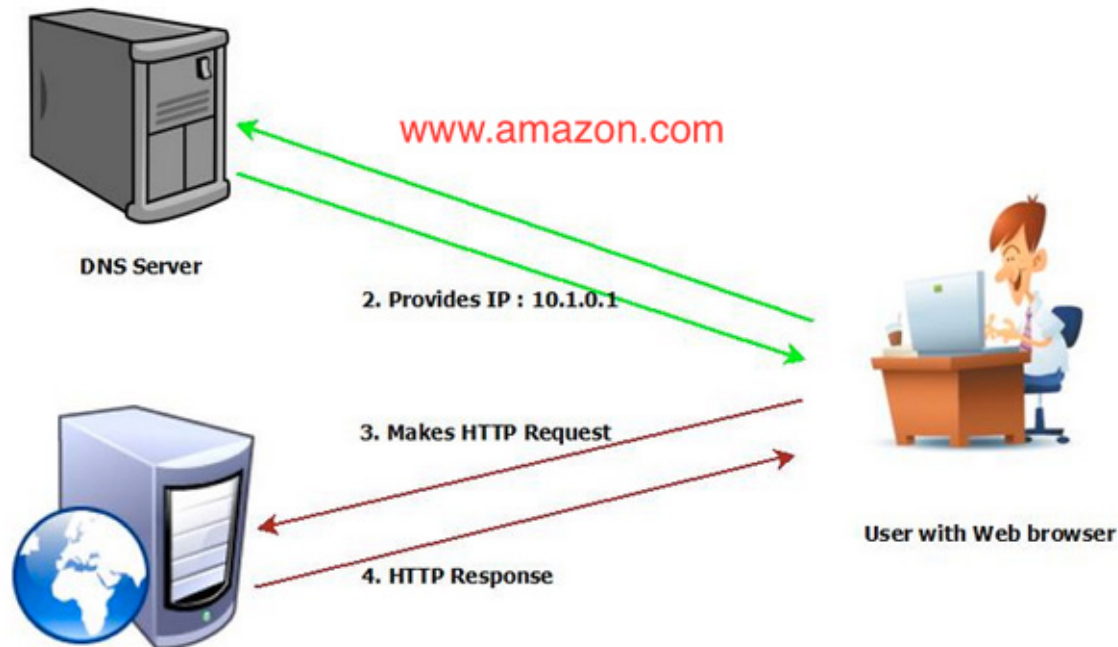
Introduction

Full Stack Development

- Basically just developing all parts of a website
- Includes the database and web server, the logic and the user interface
- A full-stack web developer is expected to take care of all of these... and more



How the Internet works?



What is Node.js

- A JavaScript library that took V8, Google Chrome's JavaScript engine out of the browser and lets you create and run your own web server
- Built in HTTP library so no requirement to use Apache or others

Benefits of Node.js

- **Speed** - It uses Google Chrome's powerful Javascript engine V8, so it can execute thousands of instructions a second
- **Asynchronism** - encourages an asynchronous coding style making for faster code to manage concurrency while avoiding multithreaded problems.
- **Popularity** - JavaScript offers Node.js access to many useful libraries
- **Consistency** - JavaScript everywhere
- **Simplicity** - Single-threaded web server as opposed to multi-threaded

Full JavaScript Tech Stacks

- **MEAN** stack - MongoDB, Express, AngularJS, NodeJS
- **MERN** stack - MongoDB, Express, ReactJS, NodeJS
- **MEN** for this course - MongoDB, Express, NodeJS

Install Node.js and verify version

- Book page: 12 Install Node.js
- Check version: `node -v`
- Check Node Package Manager (NPM) version: `npm -v`

Create first Web Server

Code on book page: 13

```
const http = require('http')
const server = http.createServer((req, res) => {
  console.log(req.url)
  res.end('Hello Node.js')
})
server.listen(3000)
```

CONST vs VAR vs LET?

Request and Response

Code on book page: 13

```
const http = require('http')
const server = http.createServer((req, res) =>{
  if(req.url === '/about')
    res.end('The about page')
  else if(req.url === '/contact')
    res.end('The contact page')
  else if(req.url === '/')
    res.end('The home page')
  else {
    res.writeHead(404)
    res.end('page not found')}
})
server.listen(3000)
```

Response with Html

Code on book page: 18

```
const http = require('http')
const fs = require('fs')
const aboutPage = fs.readFileSync('about.html')
const server = http.createServer((req, res) =>{
  if(req.url === '/about')
    res.end(aboutPage)//response with about.html page
```

Chapter 2

Introduction to NPM & Express

What is Express

- Express is a framework that acts as a light layer atop the Node.js web server making it easier to develop Node.js web applications
- It simplifies the APIs of Node.js
- Adds helpful features, helps organizes our application's functionality with middleware and routing
- Adds helpful utilities to Node.js 's HTTP objects and facilitates rendering of dynamic HTML views
- `Npm init` // Generates package.json
- `Npm install express` // Install the express

Use Express

```
const express = require('express') // require
express module
const app = express() // calls express function to start
new Express app
app.listen(3000, ()=>{
  console.log("App listening on port 3000")
})
```

- Express takes care of the http, request and response objects behind the scenes. The callback function provided in the 2nd argument in app.listen() is executed when the servers starts listening.

Run the App

Book code page: 28

```
node index.js // Run the app
```

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

```
const app = express()
```

```
app.listen(3000, ()=>{
```

```
  console.log("App listening on port 3000") })
```

```
app.get('/', (req, res)=>{
```

```
  res.json({
```

```
    name: 'Welcome to Express' })
```

```
})
```

JSON (JavaScript Object Notation) is a lightweight data-interchange format. JSON (JavaScript Object Notation) is a lightweight data-interchange format.

Handling requests with Express

- Express has populated more operations for us to respond better to browser requests.
- We can build APIs with Node.
- We can define specific routes and its response our server gives when a route is hit
- Routing where we map requests to specific handlers depending on their URL

Refactor routing code using Express

- Book code page: 29

```
const server = http.createServer((req, res) =>{
  if(req.url === '/about')
    res.end(aboutPage)
  else if(req.url === '/contact')
    res.end(contactPage)
  else if(req.url === '/')
    res.end(homePage)
  else {
    res.writeHead(404)
    res.end(notFoundPage)
  }
})
```

Refactor routing code using Express (2)

- Book code page: 29

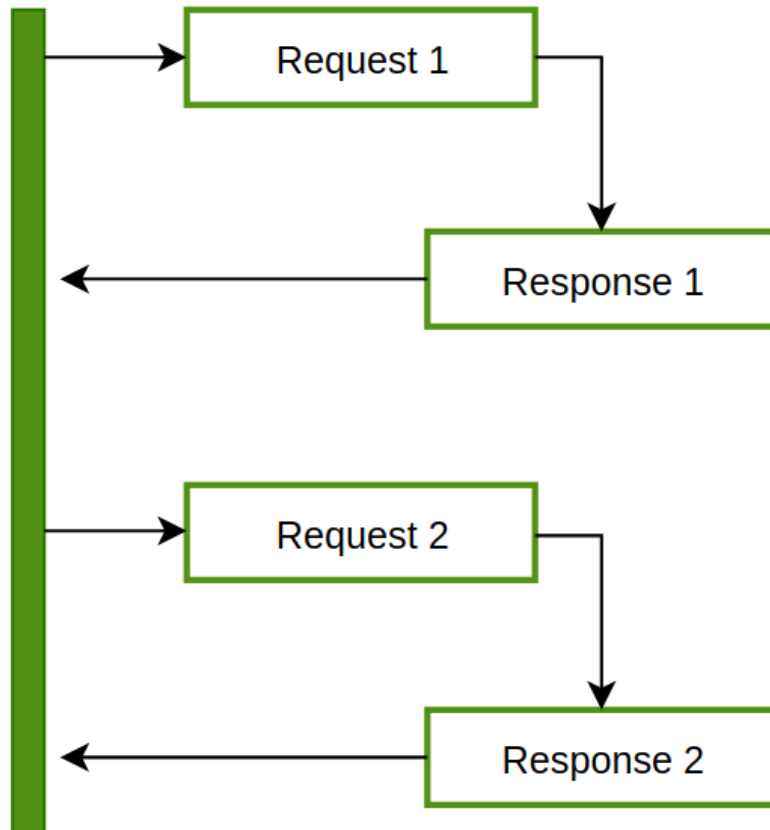
```
const path = require('path')
```

```
...
```

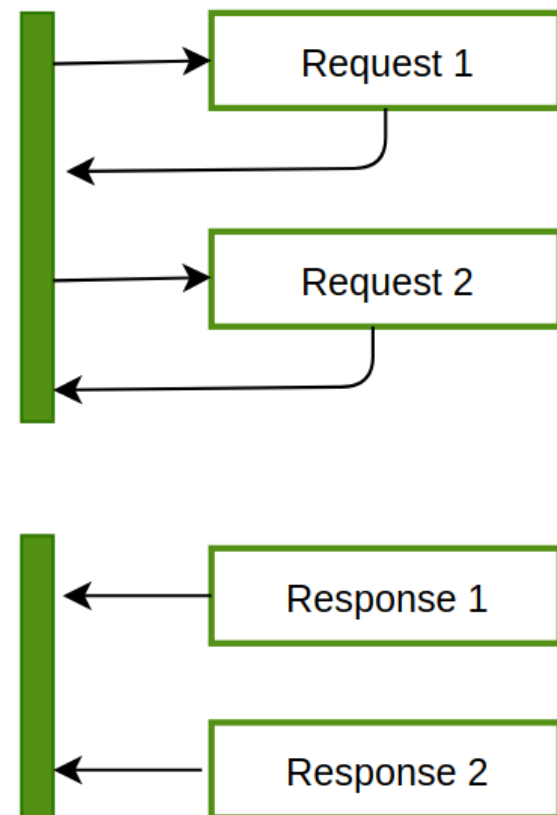
```
app.get('/', (req, res)=>{  
  res.sendFile(path.resolve(__dirname, 'index.  
html'))  
})
```

`path.resolve(__dirname, 'index.html')` helps us get the full absolute path which otherwise changes based on different Operating Systems

Synchronous



Asynchronous



Serving other HTML files

Book code page: 30

```
app.get('/contact', (req, res)=>{    //called when  
request to /contact comes  
res.sendFile(path.resolve(__dirname, 'contact  
.html'))  
})
```

- we use app.get to handle HTTP GET requests which are the most common HTTP method

CRUD operations



Serving static files with Express

- Book code page: 32

```
const path = require('path')
app.use(express.static('public'))
app.listen(3000, ()=>{
  console.log("App listening on port 3000")
})
```

- `express.static` is a packaged shipped with Express that helps us serve static files.
- With `express.static(' public ')`, we specify that any request that ask for assets should get it from the ' public ' directory

Summary

- Benefits of using Node.js
- How a request and response work between client and server
- We learn to install third party custom packages
- Express helps to make it easier to handle requests and responses
- Generate package.json and maintain project metadata
- Serve static files

Chapter 3

Node.js project and Page routes

Template code

- To quick start we will download the template code
- Visit the link and download the code
<https://startbootstrap.com/theme/clean-blog>

Setup project

- Within new folder : run *npm init*
- Install express
npm install express
- Create Index.js file and below code to it:

```
const express = require('express')  
const app = new express()  
app.listen(4000, ()=>{  
  console.log('App listening on port 4000')  
})
```
- Application is ready to listen on port 4000

Server restart (1)

- During the development we need to continuously monitor the changes we do in our application
- It would be pain, if we restart the node.js server and verify the changes
- Nodemon package monitor the code changes and auto-restart the server (see how we added `–save-dev`)

npm install nodemon –save-dev

Server restart (2)

- Modify the start script in package.json to use nodemon

```
scripts": {  
  "start": "nodemon index.js"  
},
```
- Run the application using *npm start* instead of *node index.js*
- *npm start* will look inside the package.json and run the associated command
- Nodemon watches for the file changes and auto restart server

Serving static files

- Register public folder to serve static files

```
const app = new express()  
app.use(express.static('public'))
```

- Create public folder in the application folder
- Move all html and other downloaded template to public folder

Test the App

- Nodemon should restart server
- Go to browser and test
 - <http://localhost:4000>
- Navigate to different link and verify the content

Create page routes (1)

- So far all html pages are serve from public folder
- These files should be serve from specific routes using `app.get`
- Add pages folder and move all html files into it
- Modify the `index.js` file to use path

Create page routes (2)

- Index.js file should invoke "path" package

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

```
const path = require('path')
```

```
const app = new express()
```


Create page routes (3)

- Modified index.js file to use home page route: "/",
- Request is made to home page route: " / ", index.html will be served

```
app.get('/',(req,res)=>{  
  res.sendFile(path.resolve(__dirname, 'pages/index.html'))  
})
```

Express supports methods that correspond to all HTTP request methods: get, post, and so on. For a full list, see [app.METHOD](#).

Create page routes (4)

```
app.get('/about',(req,res)=>{  
  res.sendFile(path.resolve(__dirname, 'pages/about.html'))  
})  
app.get('/contact',(req,res)=>{  
  res.sendFile(path.resolve(__dirname, 'pages/contact.html'))  
})  
app.get('/post',(req,res)=>{  
  res.sendFile(path.resolve(__dirname, 'pages/post.html'))  
})
```

Link pages (1)

- Modified index.html file to use correct href link,

```
<li class="nav-item">
```

```
  <a class="nav-link" href="/">Home</a>
```

```
</li>
```

```
<li class="nav-item">
```

```
  <a class="nav-link" href="/about">About</a>
```

```
</li>
```

```
<li class="nav-item">
```

```
  <a class="nav-link" href="/post">Sample Post</a>
```

```
</li>
```

Link pages (2)

- With the above approach we can certainly navigate from index.html to other pages
- But the problem is we need to add these links to all the other pages
- This is repetitive work and not scalable

Summary

- Began using template code
- Integrated node.js project
- Use of Nodemon to auto restart server
- Serve static pages from public folder
- Setup page routes
- Use it to navigate various html pages

Routing Method

There is a special routing method, `app.all()`, used to load middleware functions at a path for ***all*** HTTP request methods. For example, the following handler is executed for requests to the route “/secret” whether using GET, POST, PUT, DELETE, or any other HTTP request method supported in the [http module](#).

```
app.all('/secret', (req, res, next) => {  
  console.log('Accessing the secret section ...')  
  next() // pass control to the next handler  
})
```

Routing Path

Route paths can be strings, string patterns, or regular expressions.

This route path will match `abcd`, `abxcd`, `abRANDOMcd`, `ab123cd`, and so on.

```
app.get('/ab*cd', (req, res) => {  
  res.send('ab*cd')  
})
```

This route path will match `/abe` and `/abcde`.

```
app.get('/ab(cd)?e', (req, res) => {  
  res.send('ab(cd)?e')  
})
```

Route paths based on regular expressions:

This route path will match anything with an “a” in it.

```
app.get(/a/, (req, res) => {  
  res.send('/a/')  
})
```

This route path will match butterfly and dragonfly, but not butterflyman, dragonflyman, and so on.

```
app.get(/.*fly$/, (req, res) => {  
  res.send('/.*fly$/')  
})
```


Route parameters

Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the `req.params` object, with the name of the route parameter specified in the path as their respective keys.

Route path: `/users/:userId/books/:bookId`

Request URL: `http://localhost:3000/users/34/books/8989`

`req.params`: `{ "userId": "34", "bookId": "8989" }`

To define routes with route parameters, simply specify the route parameters in the path of the route as shown below.

```
app.get('/users/:userId/books/:bookId', (req, res) => {  
  res.send(req.params)  
})
```

The name of route parameters must be made up of “word characters” (`[A-Za-z0-9_]`).

Route handlers (1)

More than one callback function can handle a route (make sure you specify the next object). For example:

```
app.get('/example/b', (req, res, next) => {  
  console.log('the response will be sent by the next function ...')  
  next()  
}, (req, res) => {  
  res.send('Hello from B!')  
})
```

Route handlers (2)

An array of callback functions can handle a route. For example:

```
const cb0 = function (req, res, next) {  
  console.log('CB0')  
  next()  
}
```

```
const cb1 = function (req, res, next) {  
  console.log('CB1')  
  next()  
}
```

```
const cb2 = function (req, res) {  
  res.send('Hello from C!')  
}
```

```
app.get('/example/c', [cb0, cb1, cb2])
```

Route handlers (3)

A combination of independent functions and arrays of functions can handle a route. e.g.

```
const cb0 = function (req, res, next) {  
  console.log('CB0')  
  next()  
}  
  
const cb1 = function (req, res, next) {  
  console.log('CB1')  
  next()  
}  
  
app.get('/example/d', [cb0, cb1], (req, res, next) => {  
  console.log('the response will be sent by the next function ...')  
  next()  
}, (req, res) => {  
  res.send('Hello from D!')  
})
```

Response Methods

Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile()	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.

Chained router handlers

```
app.route('/book')  
  .get((req, res) => {  
    res.send('Get a random book')  
  })  
  .post((req, res) => {  
    res.send('Add a book')  
  })  
  .put((req, res) => {  
    res.send('Update the book')  
  })
```

Router Module (1)

Use the `express.Router` class to create modular, mountable route handlers. A Router instance is a complete middleware and routing system;

Create a router file named `birds.js` in the app directory, with the following content:

```
const express = require('express')
const router = express.Router()

// middleware that is specific to this router
router.use((req, res, next) => {
  console.log('Time: ', Date.now())
  next()
})

// define the home page route
router.get('/', (req, res) => {
  res.send('Birds home page')
})

// define the about route
router.get('/about', (req, res) => {
  res.send('About birds')
})

module.exports = router
```

Router Module (2)

Then, load the router module in the app:

```
const birds = require('./birds')
```

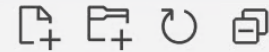
```
// ...
```

```
app.use('/birds', birds)
```


Best Practices

- Always begin a node project using **npm init**.
- Always install dependencies with a **--save** or **--save-dev**. This will ensure that if you move to a different platform, you can just run *npm install* to install all dependencies.
- Stick with lowercase file names and camelCase variables. If you look at any npm module, its named in lowercase and separated with dashes. Whenever you require these modules, use camelCase.
- Don't push node_modules to your repositories. Instead npm installs everything on development machines.
- Use a **config** file to store variables
- Group and isolate routes to their own file.

✓ EXPRESSJS-STRUCTURE



> node_modules

✓ src

> configs

> controllers

> middlewares

> models

> routes

> services

> utils

✓ test/unit

> services

> utils

◆ .gitignore

JS index.js

{ } package-lock.json

{ } package.json

i README.md

Chapter 4

Templating Engines

Static page

- In the previous chapter we ended up linking pages to the routes
- Issue with the linking was, we need to add nav bar into all other pages
- Redundant code, scalability issue, managing code would be difficult

Dynamic pages

- Dynamically render html pages
- Avoid code duplication and adds scalability
- Minimize code and helps to move duplicate code into single file

Templating Engine

- Helps to dynamically render HTML pages
- Refactor code and allows us to abstract our app into different layout files
- Don't have to repeat common code
- Helps us to create an HTML template with minimal code
- It can inject data into HTML template at client side and produce the final HTML

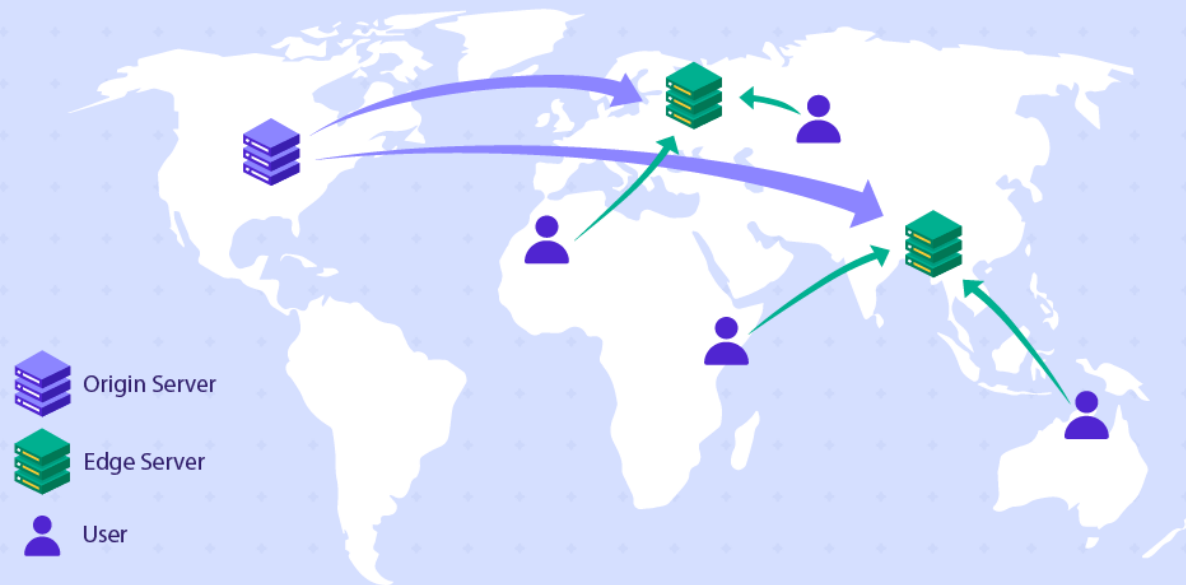
Templating Engines

- Many templating engines ie Twig, Pug, EJS, Hamlet.js etc
- We will use EJS (Embedded JavaScript)
- Install using npm: (see how we added --save)
npm install ejs --save

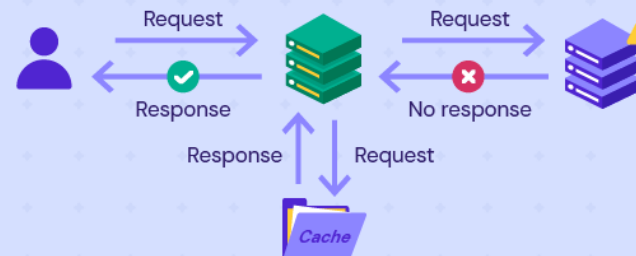
Advantages

1. Improves developer's productivity.
2. Improves readability and maintainability.
3. Faster performance.
4. Maximizes client side processing.
5. Single template for multiple pages.
6. Templates can be accessed from CDN (Content Delivery Network).

How Does a CDN Work?

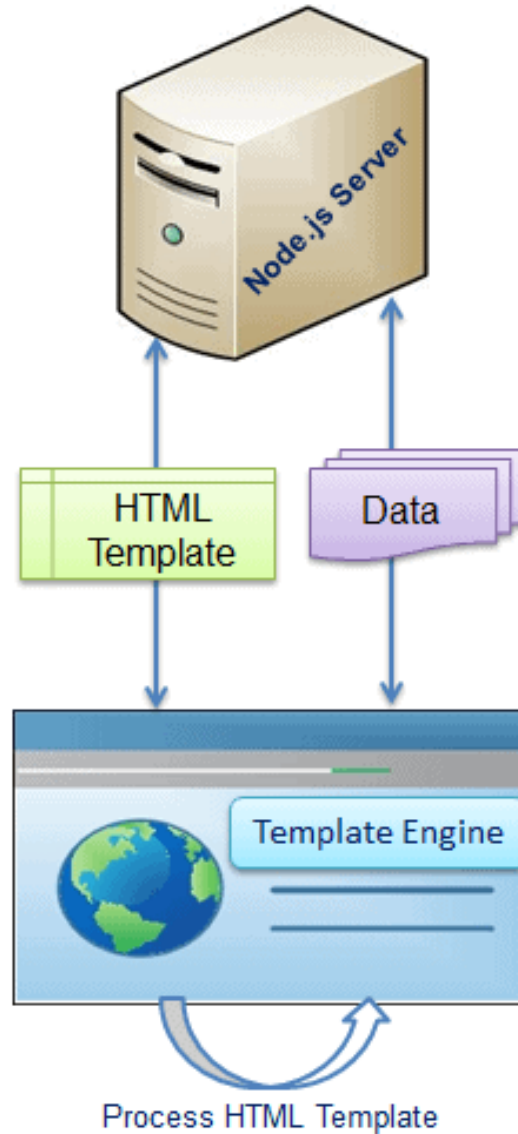


When a user visits a static website for the first time, a CDN server will transfer the files from the origin server. Note that you can access the content even when the origin server is not working since a CDN server typically stores and serves the static cached data.



How template engines work?

When you build a server-side application with a template engine, the template engine replaces the variables in a template file with actual values, and displays this value to the client. This makes it easier to quickly build our application.



Use EJS in app

- Add Ejs into index.js
- Index.js will look like below:

```
const express = require('express')  
const path = require('path')  
const app = new express()  
const ejs = require('ejs')  
app.set('view engine', 'ejs')
```

Explain code

- With `app.set`, we tell express to use EJS as our templating engine
- Any file ending with `.ejs` should be rendered with the EJS package

Change in code with EJS (1)

- Previously: Get request was like this:

```
app.get('/',(req,res)=>{  
    res.sendFile(path.resolve(__dirname,'pages/index.html'))  
})
```

- With the EJS, above code can written like this:

```
app.get('/',(req,res)=>{  
    res.render('index');  
})
```

- Send a view using res.render()

Change in code with EJS (2)

- Express adds the render method to the response object
- `Res.render('index')` will look in a views folder for the file `index.ejs`
- Get request handlers with `res.render` for about routes:

```
app.get('/about',(req,res)=>{  
  //res.sendFile(path.resolve(__dirname,'pages/about.html'))  
  res.render('about');  
})
```

Layouts (1)

- Solve the problem of repetitive code
- Create a Layout file
- Put the common code inside the layout file
- Layout file should have navbar, header, footer, scripts,

Layouts (2)

- Each page includes layout file along with its own content
- Separate `<head>`, `<nav>`, `<footer>`, code into their respective files
- Create `header.ejs`, `navbar.ejs`, `footer.ejs`
- Include these files instead of repeating entire code

Sample ejs file

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<!-- Bootstrap core CSS -->

<link href="vendor/bootstrap/css/bootstrap.min.css" rel="stylesheet">

<!-- Custom fonts for this template -->

<!-- Custom styles for this template -->

<link href="css/clean-blog.min.css" rel="stylesheet">

</head>

Script.ejs file

```
<!-- Bootstrap core JavaScript -->  
  <script src="vendor/jquery/jquery.min.js"></script>  
  <script src="vendor/bootstrap/js/bootstrap.bundle.min.js"></script>  
<!-- Contact Form JavaScript -->  
  <script src="js/jqBootstrapValidation.js"></script>  
  <script src="js/contact_me.js"></script>  
<!-- Custom scripts for this template -->  
  <script src="js/clean-blog.min.js"></script>
```

Put all together

```
<!DOCTYPE html>
  <html lang="en">
    <%- include('layouts/header'); -%>
  <body>
    <%- include('layouts/navbar'); -%>
    <!-- Page Header -->
    <!-- Main Content -->
    <%- include('layouts/footer'); -%>
    <%- include('layouts/scripts'); -%>
  </body>
</html>
```

Code explain

```
<%- include('layouts/header'); -%>
```

- include call receives a path relative to the template
- Path for header.ejs would be /views/layouts/header.ejs
- Reference: EJS tags <https://ejs.co/#docs>

Features

- Control flow with `<% %>`
- Escaped output with `<%= %>` (escape function configurable)
- Unescaped raw output with `<%- %>`
- Newline-trim mode ('newline slurping') with `-%>` ending tag
- Whitespace-trim mode (slurp all whitespace) for control flow with `<%_ _%>`
- Custom delimiters (e.g. `[? ?]` instead of `<% %>`)
- Includes
- Client-side support
- Static caching of intermediate JavaScript
- Static caching of templates
- Complies with the [Express](#) view system

Templating view files

```
<html>
<body>
  <h1>Welcome to User Details</h1>
  <p><b>Name:</b> <%= user.name %></p>
  <p><b>Email:</b> <%= user.email %></p>
  <p><b>Stack:</b> <%= user.stack %></p>
  <u><b>Hobbies</b></u>
  <% user.hubby.forEach(hubby =>{ %>
    <li><%= hobby %></li>
  <% })%>
</body>
</html>
```

Note the `<%= variable %>` pattern of displaying values. That is the way it is used in `ejs`. Also notice the `user.forEach()`; this is to show how powerful template engines can be.

Dynamic Rendering

```
app.get('/user/:id/:name', (req, res) => {  
  const user = {  
    id: req.params.id,  
    name: req.params.name,  
    stack: "MERN",  
    email: "theodoreonyejiaku@gmail.com",  
    hubby: ["singing", "playing guitar", "reading",  
"philosoph"]  
  }  
  res.render("user", {user})  
})
```


Tags

- `<%` 'Scriptlet' tag, for control-flow, no output
- `<%=` 'Whitespace Slurping' Scriptlet tag, strips all whitespace before it
- `<%=` Outputs the value into the template (HTML escaped)
- `<%-` Outputs the unescaped value into the template
- `<%#` Comment tag, no execution, no output
- `<%%` Outputs a literal '<%'
- `%>` Plain ending tag
- `-%>` Trim-mode ('newline slurp') tag, trims following newline
- `_>` 'Whitespace Slurping' ending tag, removes all whitespace after it

Summary

- Refactored code to use EJS
- Install EJS package and use it into node.js project
- Use EJS engine to dynamically render HTML pages
- Call `res.render` to dynamically render view

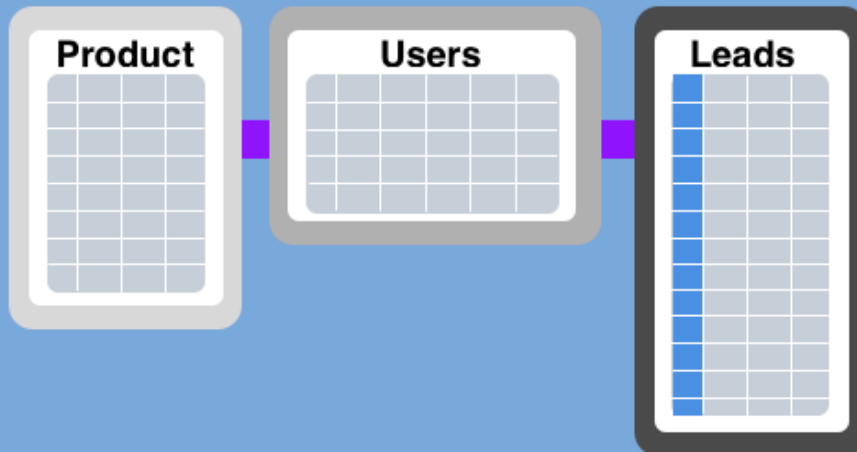
Chapter 5

Introduction to MongoDB

SQL/No-SQL Database

- Relational database is like spreadsheet
- Data is structured and each entry is row in table
- RD are controlled with SQL/ structured query language
- No-SQL database are called non-relational db
- Data is not structured in table and row
- No-SQL have collections and documents
- ***NOTE:** we will not be installing the mongodb locally rather create cloud account: <https://www.mongodb.com/>*

SQL vs. NoSQL



Architecture of MongoDB

- No-SQL stores data in collections and documents
- One of more collections
- Collections represents single entity in app ie user, organization etc
- Collection contains documents
- It is an instance in the entity ie record of the one user

Mongodb Collection-document

Database

→ Products collection

→ Product document

```
{  
  price: 26,  
  title: "Learning Node",  
  description: "Top Notch Development book",  
  expiry date: 27-3-2020  
}
```

→ Product document

- Above is an example of how collection-document looks like
- Documents look a like JSON object with various properties
- It is actually BSON – Binary JSON

MongoDB Cloud

- Visit: <https://www.mongodb.com/>
- Create free cloud account (This is easier) and use free tier
- Skip local installation

MongoDB Compass

Install: <https://www.mongodb.com/downloadcenter/compass>

- This is mongodb client
- Create test database and connect mongodb using compass
- Create few collections and documents using compass

Mongoose

- Install mongoose : *npm install mongoose*
- Connecting to MongoDB from Node
- index.js, add the following code

```
const mongoose = require('mongoose');  
mongoose.connect('mongodb+srv://<username>:<password>@<url_from_mon  
godb>/myFirstDatabase?retryWrites=true&w=majority', {useNewUrlParser:  
true})
```

The MongoDB Node.js driver rewrote the tool it uses to parse [MongoDB connection strings](#). Because this is such a big change, they put the new connection string parser behind a flag. To turn on this option, pass the `useNewUrlParser` option to `mongoose.connect()` or `mongoose.createConnection()`.

DB Connection String

- Screen shot from mongodb cloud

×

Connect to Demo

✓ Setup connection security

✓ Choose a connection method

Connect

1

Select your driver and version

DRIVER

Node.js

VERSION


4.0 or later

2

Add your connection string into your application code

☐ Include full driver code example

mongodb+srv://<username>:<password>@<cluster>.mongodb.net/myFirstDatabase?
retryWrites=true&w=majority



Replace **<password>** with the password for the **<username>** user. Replace **myFirstDatabase** with the name of the database that connections will use by default. Ensure any option params are [URL encoded](#).

Having trouble connecting? [View our troubleshooting documentation](#)

Go Back

Close

Defining a Model (1)

- Create directory models inside the project
- Create file: BlogPost.js
- Add below code:

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema;
const BlogPostSchema = new Schema({
  title: String,
  body: String
})
const BlogPost = mongoose.model('BlogPost', BlogPostSchema);
module.exports = BlogPost
```

Defining a Model (2)

- Models are defined through the *Schema* interface
- A *schema* represents how a collection looks like
- access the database via *mongoose.model*
- The first argument is the singular name of the collection your model is for
- Mongoose automatically looks for the plural version of your model name

CRUD Operation

- Create operation:

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

```
const BlogPost = require('./models/BlogPost')
```

```
mongoose.connect('mongodb+srv://<username>:<password>@<url_from_mon  
godb>/myFirstDatabase?retryWrites=true&w=majority', {useNewUrlParser: true} BlogF  
(  
  title: 'Your custom title ',  
  body: 'your custom body message'  
, (error, blogpost) =>{  
  console.log(error, blogpost)  
})
```

Code Explain

```
const BlogPost = require('./models/BlogPost')
```

- import the *BlogPost* model

```
mongoose.connect('mongodb+srv://<username>:<password>@<url_from_mon  
godb>/myFirstDatabase?retryWrites=true&w=majority', {useNewUrlParser: true})
```

- connect to the database

```
BlogPost.create({}, {}, (error, blogpost) => {console.log(error, blogpost)})
```

- create a new *BlogPost* document
- first argument, we pass in the data
- 2nd argument, we pass in a call back function

Visualizing Data (1)

- Execute code using node: node test.js
- Verify the data using mongo Compass

test.blogposts

1 1
DOCUMENTS INDEXES

Documents Aggregations Schema Explain Plan Indexes Validation

Filter   Type a query: { field: 'value' }

Reset

Find



More Options ▶

 ADD DATA ▼

 EXPORT COLLECTION

1 - 1 of 1 

< >







```
_id: ObjectId('63d966ec1e0595be144e77c1')
title: "blog title"
body: "blog body"
__v: 0
```


ObjectId

In MongoDB, each document stored in a collection requires a unique [_id](#) field that acts as a [primary key](#). If an inserted document omits the `_id` field, the MongoDB driver automatically generates an [ObjectId](#) for the `_id` field.

ObjectIds are small, likely unique, fast to generate, and ordered. ObjectId values are 12 bytes in length, consisting of:

- A 4-byte timestamp, representing the ObjectId's creation, measured in seconds since the Unix epoch.
- A 5-byte random value generated once per process. This random value is unique to the machine and process.
- A 3-byte incrementing counter, initialized to a random value.

Read, Update

- ***Read: Display all records***

```
BlogPost.find({}, (error, blogspot) =>{  
  console.log(error, blogspot)  
})
```

- ***Update:***

```
var id = "5cb436980b33147489eadfbb";  
BlogPost.findByIdAndUpdate(id, {  
  title: 'Updated title'  
}, (error, blogspot) => {  
  console.log(error, blogspot)  
})
```

Delete

- *Delete:*

```
var id = "5cb436980b33147489eadfbb";  
BlogPost.findByIdAndDelete(id, (error, blogspot) =>{  
  console.log(error, blogspot)  
})
```

Chapter 6

Apply MongoDB to project

Handle POST Request

```
app.post('/posts/store',(req,res)=>{  
  console.log(req.body)  
  res.redirect('/')  
})
```

- we get the form data from the browser via the request *body* attribute

Save Data to DB

```
const BlogPost =require('./models/BlogPost.js')
```

```
...
```

```
app.post('/posts/store',(req,res)=>{
```

```
// model creates a new doc with browser data
```

```
  BlogPost.create(req.body,(error,blogpost) =>{
```

```
    res.redirect('/')
```

```
  })
```

```
})
```

- Run and app and fill the form data
- Verify the data stored in MongoDB using compass

Display List of Blog Posts

- Continue working from previous week
- Ref: page 68

```
app.get('/', async (req, res) => {  
  const blogposts = await BlogPost.find({})  
  res.render('index', {  
    blogposts: blogposts    // assigning data to blogposts variable  
  });  
})
```

Dynamic Data with Templating Engine

- Use EJS templating engine to dynamically display data
- Ref: page 69-70
- Add the code in index.ejs
- Ref: page 71
- Display the data on the page

Single Blog Post

- Page 63
- Add code in index.js

```
app.get('/post/:id', async (req, res) => {  
  const blogpost = await BlogPost.findById(req.params.id)  
  res.render('post', {  
    blogpost  
  })  
})
```

- Code explanation page: 74
- Modify post.ejs

Modify Schema

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema;
const BlogPostSchema = new Schema({
  title: String,
  body: String,
  username: String,
  datePosted: { /* can declare property type with an object like this because we
need 'default' */
    type: Date,
    default: new Date()
  }
});
```

Update Code

- Modify index.ejs and post.ejs (page: 75 - 76)

Summary

- Introduced to MongoDB
- NoSQL database that stores data in the form of collections and documents
- Using Mongoose, connect to Node application
- CRUD operation using Mongoose
- Use MongoDB compass to display data
- Use MongoDB model inside the project
- Add form and save data from form inside mongodb
- Used MongoDB to build blog app
- Implemented form to create blog post
- Used model to store data
- Display list of blog and individual blog using EJS templating engine

Chapter 7

Upload an Image with Express

Express-fileupload

- Download package:
(<https://www.npmjs.com/package/express-fileupload>)

npm install --save express-fileupload

Create.ejs

```
<form action="/posts/store" method="POST" enctype="multipart/form-data">
  <div class="control-group">
    <div class="form-group floating-label-form-group controls">
      <label>Image</label>
      <input type="file" class="form-control" id="image" name="image">
    </div>
  </div>
  <br>
  <div class="form-group">
    <button type="submit" class="btn btn-primary"
    id="sendMessageButton">Send</button>
  </div>
</form>
```

Code Explanation (1)

```
<form action="/posts/store" method="POST" enctype="multipart/form-data">
```

- *enctype="multipart/form-data"* is for the browser to know that the form contains multi media
- browser will then encrypt the multi media before sending it to the server

```
const fileUpload = require('express-fileupload')
```

- *express-fileupload* adds the *files* property to the *req* object so that we can access the uploaded files using *req.files*.
- We then register the package in Express with *app.use(fileUpload())*

Code Explanation (2)

```
let image = req.files.image;  
image.mv(path.resolve(__dirname, 'public/img', image.name), async  
(error) => {  
    await BlogPost.create(req.body)  
    res.redirect('/')  
})
```

- *req.files.image* object contains certain properties like *mv* - a function to move the file elsewhere on your server and *name*
- *image.mv* moves the uploaded file to *public/img* directory with the name from *image.name*

Saving uploaded Image to DB (1)

```
const BlogPostSchema =  
new Schema({  
  title: String,  
  body: String,  
  username: String,  
  datePosted:{  
    type: Date,  
    default: new Date()  
  },  
  image: String  
});  
• Update the model
```

Saving Uploaded Image to DB (2)

- Update Index.js

```
app.post('/posts/store', (req,res)=>{  
  let image = req.files.image;  
  image.mv(path.resolve(__dirname,'public/img',image.name),async  
    (error)=>{  
      await BlogPost.create({  
        ...req.body,  
        image: '/img/' + image.name  
      })  
      res.redirect('/')  
    })  
})
```

Chapter 8

Introduction to Express Middleware

Middleware

- Middleware are functions that Express executes in the middle after the incoming request which then produces an output which could either be the final output or be used by the next middleware
- middlewares might make changes to the *request* and *response* objects

Middleware Example

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

```
app.use(bodyParser.json())
```

```
app.use(bodyParser.urlencoded({extended:  
true}))
```

```
app.use(fileUpload())
```

- Use function registers middleware with Express app
- Express will execute all the ' use ' statements sequentially before handling the request

Custom Middleware

```
const customMiddleWare = (  
  req,res,next)=>{  
    console.log('Custom middle ware called')  
    next()  
  }  
app.use(customMiddleWare)
```

- *next()* tells Express that the middleware is done and Express should call the next middleware function
- All middlewares called by *app.use* calls *next()*.

Registering Validation Middleware

- Use middleware for form validation
- Validation middleware

```
const validateMiddleWare = (  
  req,res,next)=>{  
  if(req.files == null || req.body.title == null || req.body.title == null){  
    return res.redirect('/posts/new')  
  }  
  next()  
}
```


Use Custom Validation Middleware

- The validateMiddleWare middleware simply checks if any of the form fields are null (which means that they are not entered by the user) and if so, redirect them back to the create post page
- Apply middleware using: `app.use(validateMiddleware)`
- Apply middleware for specific url:
 - `app.use('/posts/store',validateMiddleWare)`

Summary

Chapter 9

Introduction to MVC

Model-View-Controller

- To keep entire code into single file is not best approach
- It adds complexity and code becomes unmanageable
- Refactor the code using MVC
- Helps to develop code faster

Model

- Represents the structure of the data, the format and the constraints with which it is stored
- It is the database part of the application

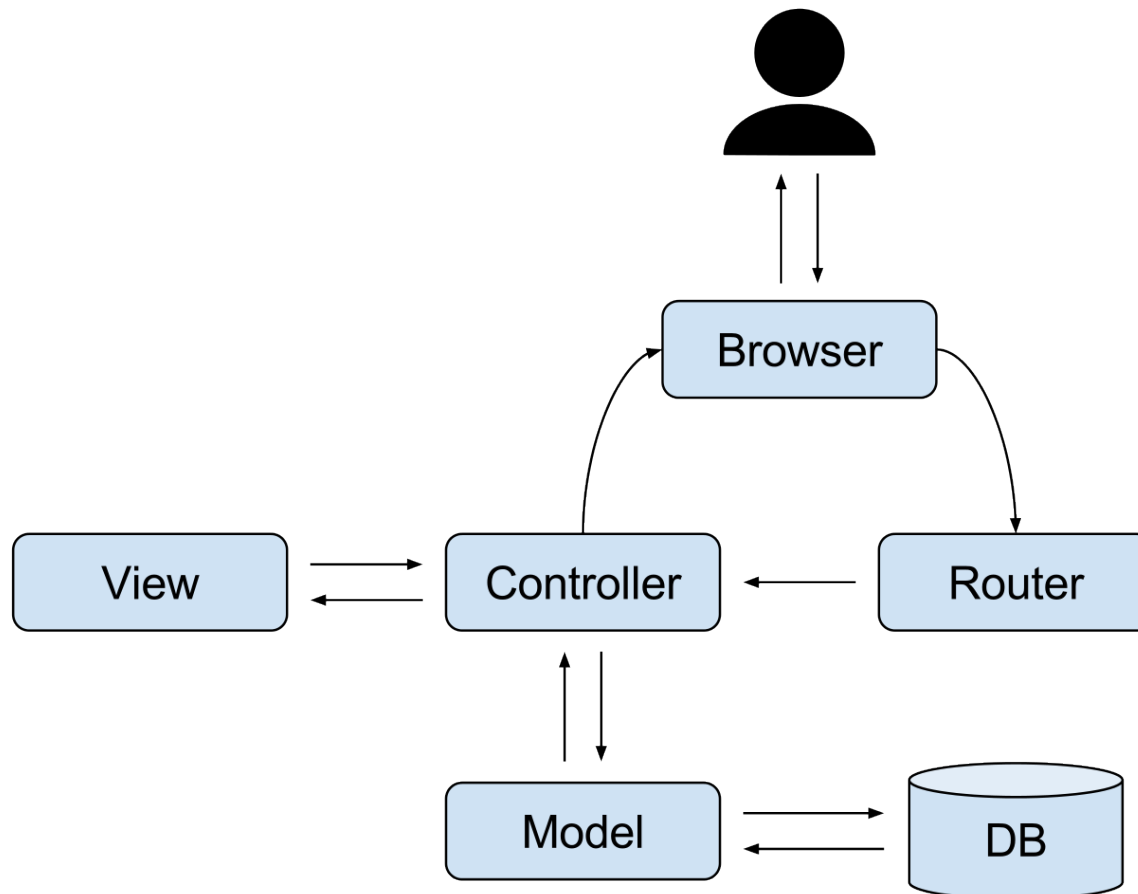
View

- View is what is presented to the user
- View makes use of model and present data to the user
- User can make request/ change the data from view
- View consist of static /dynamic webpages
- Mostly pages are stored in views folder

Controller

- Controls the requests from the user
- Generates appropriate response and render back to user
- In this chapter we will refactor code into controller

MVC Architecture Pattern



Refactor Code (1)

- Add new folder controller
- Add new file newPost.js
- This file contain the controller handling request

```
module.exports = (req, res) =>{  
    res.render('create')  
}
```

- Code inside the index.js

```
const newPostController = require('./controllers/newPost');  
app.get('/posts/new',newPostController);
```

Refactor Code (2)

- Create home.js, getPost.js and storePost.js
- Add code to find all the blog, single blog and create blogs inside above js files
- Index.js would be like below:

```
const homeController = require('./controllers/home')  
const storePostController = require('./controllers/storePost')  
const getPostController = require('./controllers/getPost')
```

```
app.get('/',homeController)  
app.get('/post/:id',getPostController)  
app.post('/posts/store', storePostController)
```

Refactor Validation Layer

- validationMiddleware.js

```
module.exports = (req,res,next)=>{  
    if(req.files == null || req.body.title == null || req.body.body == null){  
        return res.redirect('/posts/new')  
    }  
    next()  
}
```

- Modify the code in index.js

```
const validateMiddleware = require("../middleware/validateMiddleware");
```

Chapter 10

User Registration

Add View

- Ref code: page: 95-96
- This code will create view and controller for the user registration

User Model

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema;
const UserSchema = new Schema({
  username: String,
  password: String
});
• // export model
const User = mongoose.model('User',UserSchema);
module.exports = User
```

Password Encryption

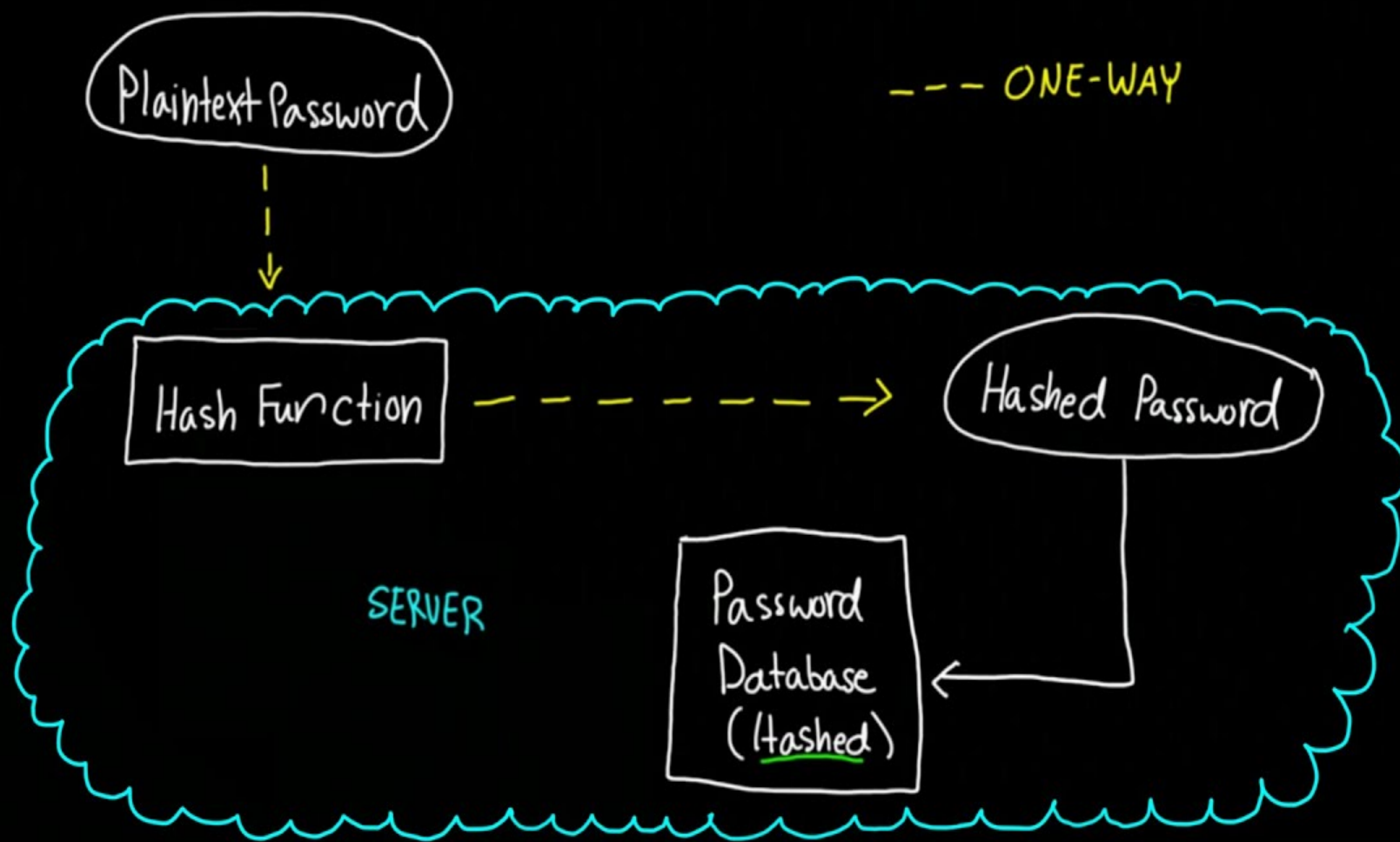
- Use mongoose model hook to encrypt password
- Hook is act as middleware
- Package used for this is: bcrypt
- Bcrypt helps to hash password



CONESTOGA
Connect Life and Learning

WHAT YOU DO HERE...
COUNTS OUT THERE®

Hashing



Modified User Model

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema;
const bcrypt = require('bcrypt')
const UserSchema = new Schema({
  username: String,
  password: String
});
UserSchema.pre('save', function(next){
  const user = this
  bcrypt.hash(user.password, 10, (error, hash) => {
    user.password = hash
    next()
  })
})
const User = mongoose.model('User', UserSchema);
module.exports = User
```

Mongoose Validation (1)

- Mongoose will be used for the data validation before saving to db
- Modify user Schema:

```
const UserSchema = new Schema({  
  username: {  
    type: String,  
    required: true,  
    unique: true  
  },  
  password: {  
    type: String,  
    required: true  
  }  
});
```

Mongoose Validation (2)

- Username is required and it should be unique
- Password is required too
- *required* field specifies that this field is required
- Set the *unique* to true

Log Errors

- storeUser.js

```
const path = require('path')
module.exports = (req,res)=>{
  User.create(req.body, (error, user) => {
    console.log(error)
    res.redirect('/')
  })
}
```

User Login Process

- Code Ref – Pages: 103-106

Summary

- Refactor the code into MVC pattern
- Create controller and separate js code into different files
- Use bcrypt package to hash user password
- Use mongoose validation to validate user entered data