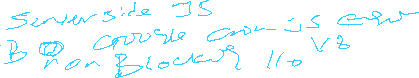
**LECTURE 2 – NODE 1**

* **JavaScript** was traditionally the language of the web browser referred to as "client-side" processing.
* **Node.js** is a server-side platform built on Google Chrome's JavaScript Engine (V8 Engine).
* **ECMAScript** is an organization that keeps the standard of JS.
* **JS Engine:** A program that convert JS code into something that computer processor understands
* **NodeJS**



•Server-side JavaScript

•Built on Google’s V8



•Evented, non-blocking I/O

•Common JS module system



•Allows script programs do I/O in JavaScript



•Focused on Performance



* **Process -**  an instance of a program that lives in the memory.
* **A thread -**  is some sort of instructions to be executed by the CPU



* **OS scheduler**: scheduling is the logic the OS uses to decide which thread to process at any given time.



* **Hyper-threading** - improving scheduling threads by adding more CPU cores.
* **I/O** - A communication between CPU and any other process external to the CPU (memory, disk, network)



* **How to handle I/O:**



•Synchronous code (slow)



•Fork new process (doesn’t scale)

•Threads (complicated and we need to lock shared resources)

•Single thread (event loop)



* how I/O should be done.

E.g Instead of : var result = db.query("select \* from T"); which blocks the entire process



Do: db.query("select..", function (result) {// use result });



* JavaScript/ V8 is Synchronous



* Node does things Asynchronously.



* Node.js was created because JavaScript had no existing I/O libraries, so they could start clean with non-blocking I/O.



* **Node.js features**

1. Asynchronous and Event Driven
2. Very Fast



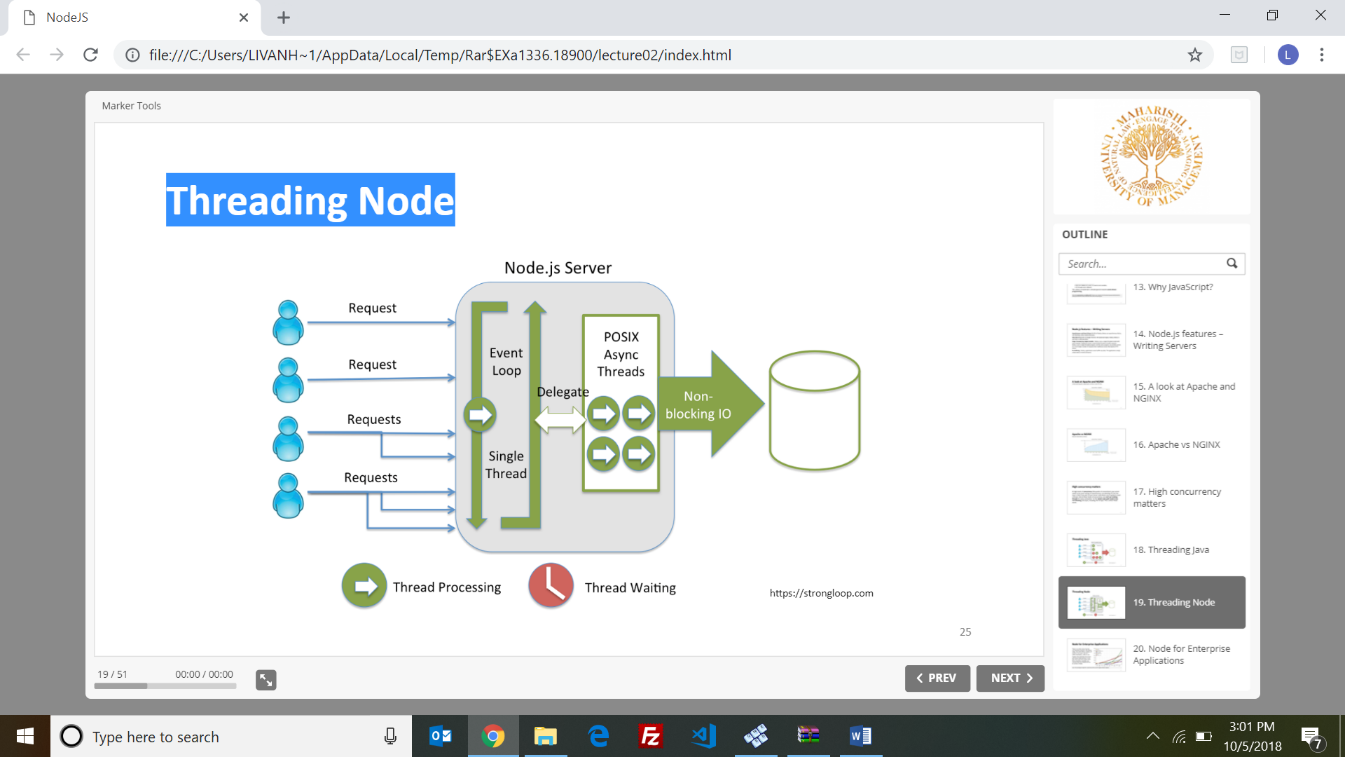
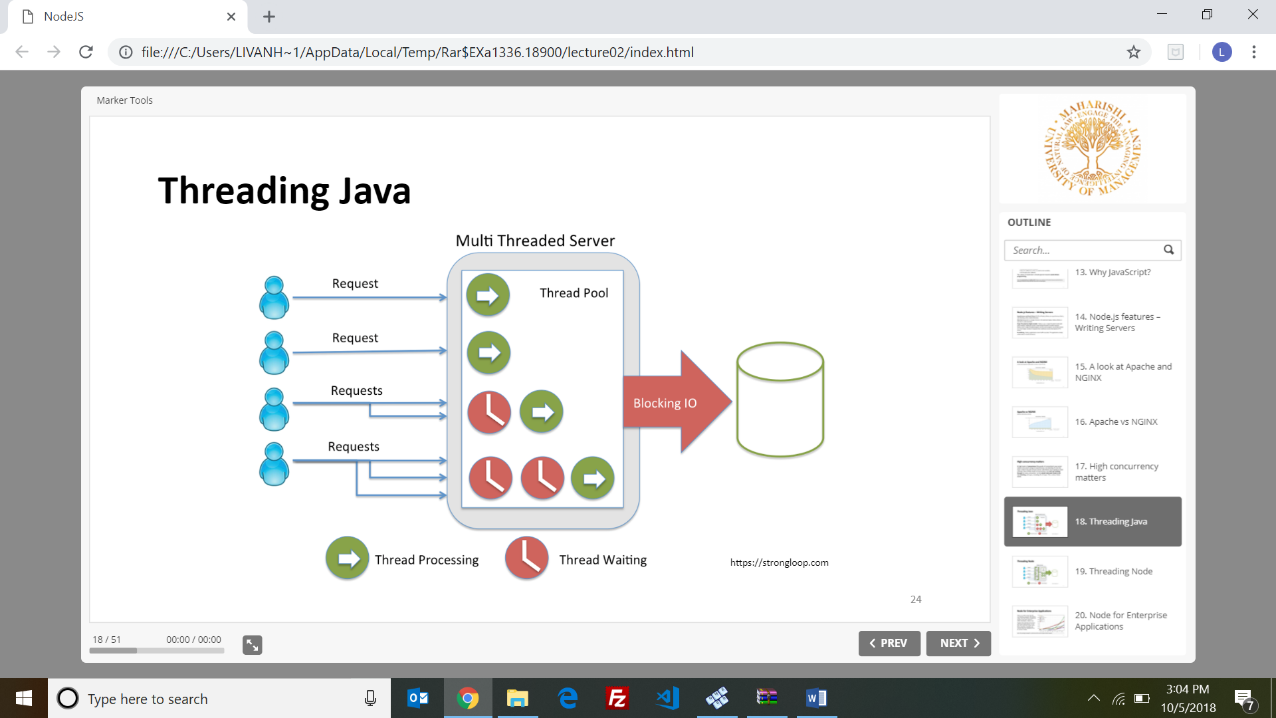
1. Single Threaded but Highly Scalable



1. No Buffering



* **Threading Java Vs**  **Threading Node**



* There is no other server has the non-blocking ecosystem like Node.js today.
* **npm** is the package manager for JavaScript and the world’s largest software registry.
* What does JavaScript need to manage a Server?

•Organize our code into reusable pieces (MVC)

•Ways to deal with Files

•Ways to deal with Databases

•Ability to communicate over the Internet – Accept requests and send

responses in standard format

•Deal with work that takes a long time

* **Example HTTP Server**

var http = require('http');

http.createServer(function(req, res){

res.writeHead(200, {'Content-Type': 'text/plain'});

res.end('Hello World\n');

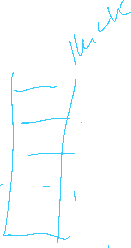
}).listen(1337, '127.0.0.1');

* **Google V8** is Google's open source JavaScript engine, written in C++ and is used in Google Chrome and is very fast.
* Node is written in C++; because V8 that converts JS to machine code is written in C++.



* **What’s inside Node**? C++ Core, The JavaScript Core, Other Node dependencies





* **The Server Global Environment :** Buffer, Console, URL, URL SearchParams, Global,

Process, \_\_dirname, \_\_filename, exports, module, require(), setInterval() and clearInterval(),

setTimeout() and clearTimeout(),setImmediate() and clearImmediate()

* In Node we run JS on the server so we don’t have window object. Instead Node provides us with global modules and methods that are automatically created for us (they aren’t part of ECMA specifications)



* Browser JavaScript by default puts everything into its window global scope. Node.js was designed to behave differently with everything being local by default (variables and functions).
* Declaring a variable without **var** will create the variable on the global object
* The document object that represent DOM of the webpage is nonexistent in Node.js
* Node Single-Thread Model - all events will be processed only one at a time
* **Event loop -** A loop that picks events from the event queue and pushes their callbacks into the call stack
* Node.js runs using a single thread . Under the hood Node uses many threads through libuv.



* **node myFile.js** - Read the file content and run the JS code + track all operations
* **Event Loop order of operations –** Timers=>I/O callbacks=>idle, prepare=>incoming: connections, data..=>check=>Close callbacks
* **setTimeout**schedules a callback to run after a specific time, the functions are registered in the timers phase of the event loop.



* **setImmediate** will schedule a callback to run at check phase of the event loop after IO event’s callbacks.



* **process.nextTick()** is not part of the event loop, it adds the callback into the nextTick queue.

**=======================================================================**

**LECTURE 3 – NODE 2**

* Node implements CommonJS Modules specs. If we simply create a normal JS file it will be a module.



* **How require() works** - 1.Resolve 2.Load 3.Wrap 4.Evaluate 5.Cache



* **require(path**) - Encapsulates my JS code in IIFE (protected) and run it and returns what we



explicitly attach to module.exports object.



* Add **./** to look into local location for the the JS file
* **module.exports -** return statement



**E.g:** // helloModule.js



var sayHi = function(){ console.log('hi');  }

module.exports = sayHi;

// app.js

var hello = require('./helloModule');

hello();

* **exports** and **module.exports** both reference to the same (empty) object at the beginning but only module.exports will be returned.
* **kinds of events:**

•**Custom Events,** Objects using EventEmitter from Node.

•**System Events**  that comes from C++ core (libuv)

* **EventEmitter object:**

•It has **.emit()** on the prototype using .call() .apply() to invoke all the listeners functions.

•It has **.on()** which is a shortcut to addListener() function also on the prototype.

**Buffers and Streams**

* **Buffer:** A chunk of memory allocated outside V8 (intentionally limited in size, we cannot resize an allocated buffer)
* **Stream**: a sequence of data made available over time. Pieces of data that eventually combine into a whole.
* Buffer is a Node.js addition to primitives.

**Character set vs. Encoding**

* **Character set:** A representation of characters as numbers
* **Encoding:** characters are stored in binary
* Buffers data are saved in binary
* **Slice a buffer -** buffer can be sliced into a smaller buffer by using the **slice()** method.

**E.g .** var buffer = new Buffer('this is the string in my buffer');

var slice = buffer.slice(10, 20);

* **Files** -

**Example Read/Write Files**

var fs = require('fs');

var path = require('path');

// Reading from a file:

fs.readFile(path.join(\_\_dirname, 'data/students.csv'),

{encoding: 'utf-8'}, function (err, data) {

if (err) throw err;

console.log(data);

});

// Writing to a file:

fs.writeFile('students.txt', 'Hello World!', function (err) {

if (err) throw err;

console.log('Done');

});

* **Stream types:**

•Readable (fs.createReadStream)

•Writable (fs.createWriteStream)

•Duplex (net.Socket)

•Transform (zlib.createGzip)

* Streams inherit from EvenEmitter. So they have access to on() and emit().
* Use **encoding** to convert data to String of hex and **highWaterMark** to set the size of the chunk of streams.
* **pipe()** - method used to connect two streams.

**E**.g.

var fs= require('fs');

var readable = fs.createReadStream(\_\_dirname + 'sourceFile.txt');

var writable = fs.createWriteStream(\_\_dirname + 'destinationFile.txt');

readable.pipe(writable);

* **Zip file using streams**

**E.g**

var fs = require('fs');

var zlib = require('zlib');

var gzip = zlib.createGzip();

// this is a readable & writable stream and it returns a zipped stream

var readable = fs.createReadStream(\_\_dirname + 'source.txt');

var compressed = fs.createWriteStream(\_\_dirname + 'destination.txt.gz');

readable.pipe(gzip).pipe(compressed);

* A key goal of the stream API, is to limit the buffering of data to acceptable levels such that sources and destinations of differing speeds will not overwhelm the available memory
* Node is very popular today to create and run Web servers.
* Web Server create (shortcut)

require('http').createServer(function(req, res) {

res.writeHead(200, {'Content-Type': 'text/plain'});

res.end('Hello World!');

}).listen(4000, '127.0.0.1');

**Reactive Server with RxJs**

* The benefit of using a reactive server is that many components can listen to the subject and

update themselves accordingly

**Reactive benefits**

* We can add logging, buffer, filter, merge and many more operations on events, because of the large number of operations in RxJS library.
* We can handle errors and memory deallocation easily.
* It is much simpler than using callbacks, and even simpler than using Promises.
* **Send out an HTML file in an easier way**

**E.g.**

var http = require('http');

var fs = require('fs');

http.createServer(function(req, res) {

    var rs = fs.createReadStream('/big/file').pipe(res);

})

* **util.promisify()-** It converts a callback-based function to a Promise-based one.

**=======================================================================**

**LECTURE 4 – NODE 3**

* **Package**is a collection of code (module) you can use it in your code managed by package manager.
* **Package management system**: software that automates installing and updating packages.
* **Dependencies:** code (module) that another set of code (module) depends on to work.
* **npm:** Node Package Manager
* The npm registry: source of all codes we are installing.
* npm is not part of Node, it's a package complementing Node
* **npm install** - will read all dependencies and install them for us.
* **Three kind of packages:** dependency, development and global.
* **npm init**  - will create package.json
* **Semantic Versioning –** used to give a version of code meaning :

Example:  **MAJOR.MINOR.PATCH**

•PATCH: Some bugs were fixed. Your code will work fine

•MINOR: some new features were added. Your code will work fine.

•MAJOR: Big changes. Your code will break (maybe)

* **package.json Manifest :**

**^**  only update minor and patches

**~**  only update patches

* **package-lock** is used to avoid the situation where installing modules from the same package.json results in two different installs.
* package.json overrules the package-lock if package.json has been updated
* **npm install moment** - moment is a package that parse, validate, manipulate and display dates.

**E**.g: var moment = require('moment');

console.log(moment().format("ddd, hA")); // Mon, 10AM

* **How Node resolves a core module path without (./):**

•If it is not a Core Node module then:

•It looks at your local node\_modules first

•Then the global repository on your machine

**Dev and Global Packages**

* **Development Dependencies:** Needed only while developing the app. It’s not needed for running the app. E.g npm install jest --save-dev
* **Global Dependencies:** Available to all applications.

npm install –g nodemon

nodemon app.js

* Handling basic GET & POST requests is relatively simple with Node.js but not file unloads.
* **url.parse(str)** will return URL object with properties (protocol, hostname, port, pathname, hash).
* **Format a URL**

const urlObject = {

protocol: 'http',

host: 'www.mum.edu',

search: '?q=CS572',

pathname: '/search', };

console.log( url.format(urlObject) );  // http://www.mum.edu/search?q=CS572

* **Using querystring core module**

const querystring = require('querystring');

querystring.stringify({

name: 'Asaad Saad',

course: 'CS572 Modern Web Applications'

})

// 'name=Asaad%20Saad&course=CS572%20Modern%20Web%20Applications'

querystring.parse('name=Asaad%20Saad&course=CS572%20Modern%20Web%20Applications');

// { name: 'Asaad Saad', course: 'CS572 Modern Web Applications' }

**Reading Post Data**

* Handling POST data is done in a non-blocking way, by using asynchronous callbacks. Because POST requests can be very large in size and handling it would result in a blocking operation.
* Node.js serves our code the POST data in small chunks (stream) to handle blocking.

**Scalability**

* We can achieve scalability for our Node application in three ways:

•Cloning (using multi processors on single machine)

•Decomposing (micro services)

•Sharding (using multi machines)

* To clone a new child process from the master process you may use:

• spawn() •fork() •exec() •execFile()

* Event loop is the control structure that decides what the thread is doing at any given time
* **pm2 Advanced Process Manager**
* The **cluster mode** allows node.js applications to be scaled across all CPUs available, without any code modifications. This increases the performance and reliability of your applications.
* The best debugger is **console.log()** but sometimes we need to see the callstack and dive in async code a bit more.

**=======================================================================**

**LECTURE 5 – EXPRESS 1**

* **Stateful** - Keeps data
* **Stateless** - Does not keep any data
* **REST = Representational State Transfer -** Architectural style for distributed systems

**Idempotent vs. Safe Methods**

* **Safe methods** are HTTP methods that do not modify resources. E.g OPTIONS, GET, HEAD
* **idempotent HTTP method** is a HTTP method that can be called many times without different outcomes.
* **RESTful API** is an API that uses HTTP requests to GET, PUT, POST and DELETE data.
* **API Versioning** - is useful because it allows you to commit breaking changes to your service without breaking the code for existing consumers.
* It can be implemented in two ways:

1. API version should be set in HTTP headers
2. API version is embedded into the API endpoint prefix:/api/v1/....

* Having the identifier as part of the request endpoint is great because it allows DELETE and GET requests to use the same endpoint

**Response Paging**

* To implement paging, we use the Link header.
* The rel attribute describes the relationship between the requested page and the linked

page. E.g Link: <http://example.com/api/products/?p=1>; rel="first"

* Setting the **Cache-Control** header to **private** bypasses intermediaries.

E.g Cache-Control: private

* The **Expires** header tells the browser that a resource should be cached and not requested again until the expiration date has elapsed. E.g Expires: Fri, 5 Oct 2018 18:31:12 GMT
* It’s hard to define future Expires headers in API responses because if the data in the server changes, it could mean that the client’s cache becomes stale.

**Conditional Requests**

* **Conditional requests** can be time-based, specifying a **Last-Modified** header in your responses.
* It’s best to specify a **max-age** in the Cache-Control header, to let the browser invalidate the
* cache after a certain period of time even if the modification date doesn’t change.

E.g:

Cache-Control: private, max-age=86400

Last-Modified: Fri, 5 Oct 2018 18:31:12 GMT

* The next time the browser requests this resource, it will use the If-Modified-Since request

header: If-Modified-Since: Fri, 5 Oct 2018 18:31:12 GMT

* If the resource hasn’t changed since the specified date, the server will return with an empty body with the 304 Not Modified status code
* **Entity Tag -** A hash that represents the resource in its current state.

Cache-Control: private, max-age=86400

ETag: "d5aae96d71f99e4ed31f15f0ffffdd64"

* **Throttling (rate limiting),** is a technique to limit the number of requests a client can make to your API in a certain window of time.
* The **X-RateLimit-Reset** header should contain a UNIX timestamp describing the moment when the limit will be reset: E.g

X-RateLimit-Limit: 2000

X-RateLimit-Remaining: 1999

X-RateLimit-Reset: 1404429213925

* Once the request quota is drained, the API returns a 429 Too Many Requests response

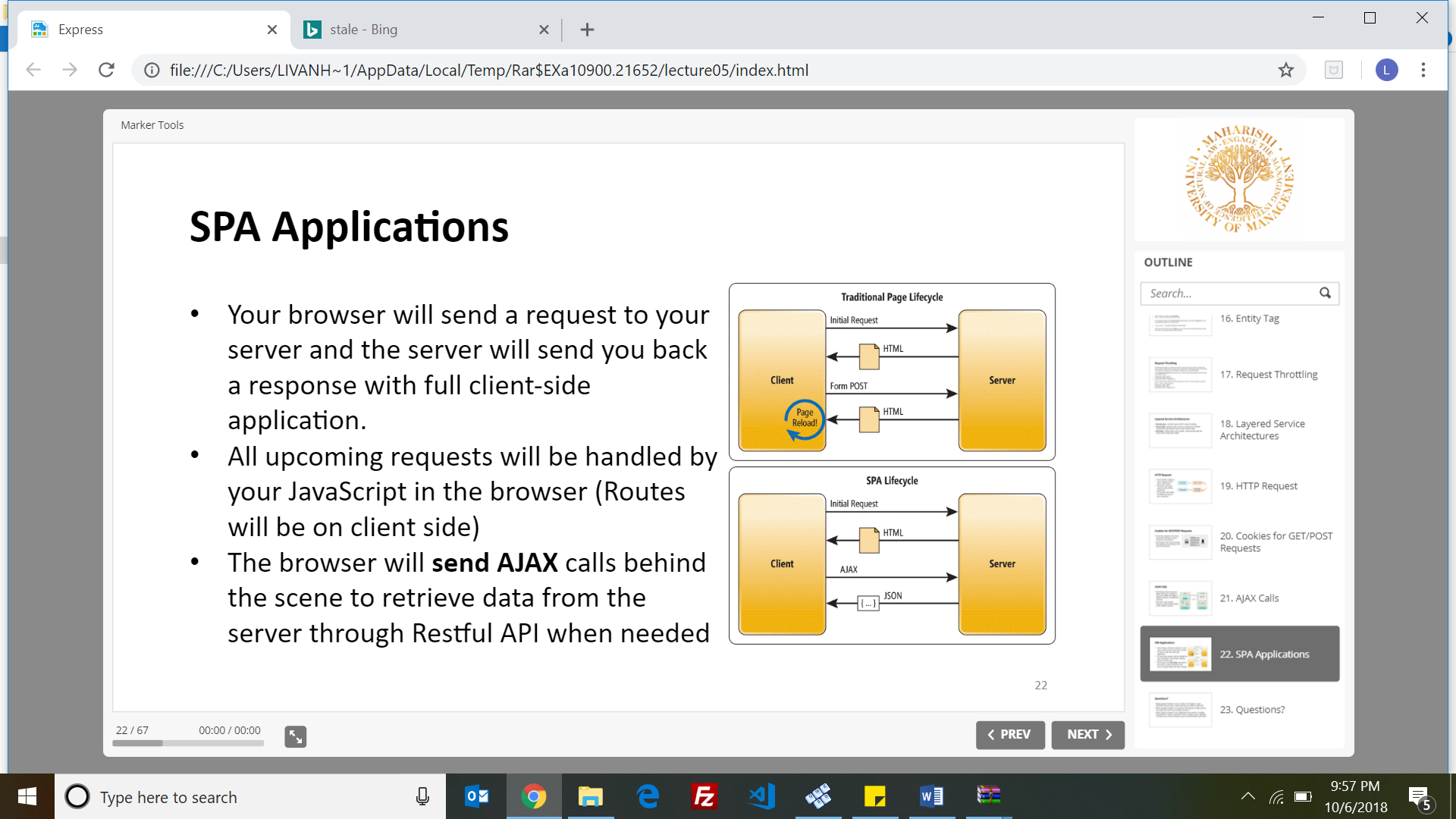
E.g. HTTP/1.1 429 Too Many Requests

* **Layered Service Architectures**

• **Routing layer:** controller logic and API request handling.

• **Service layer:** business logic concerns

• **Data layer:** unified data access models



**AJAX and Cookies**

• Cookies are not set automatically from AJAX response (the response is handled by an AJAX callback function)

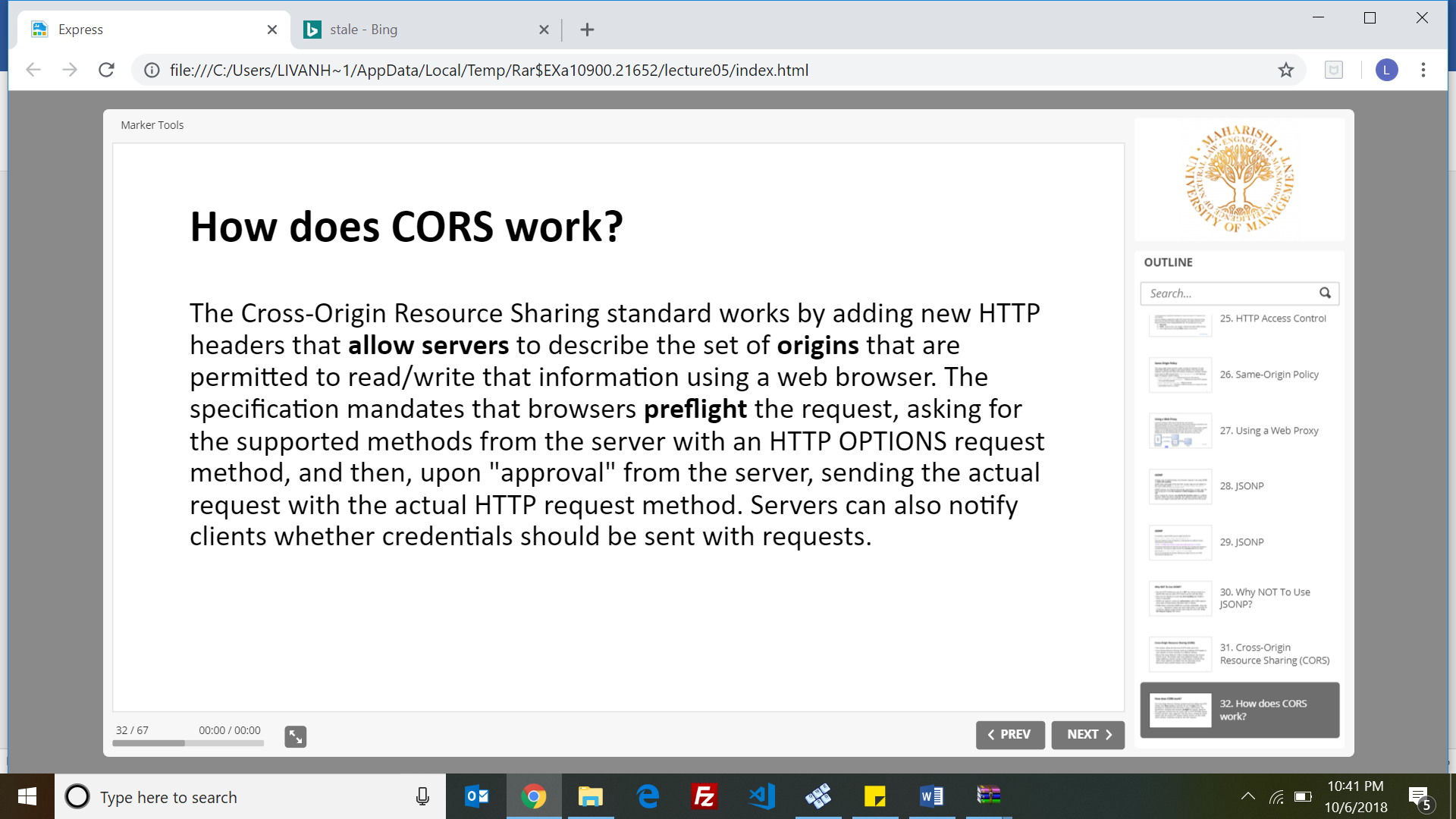
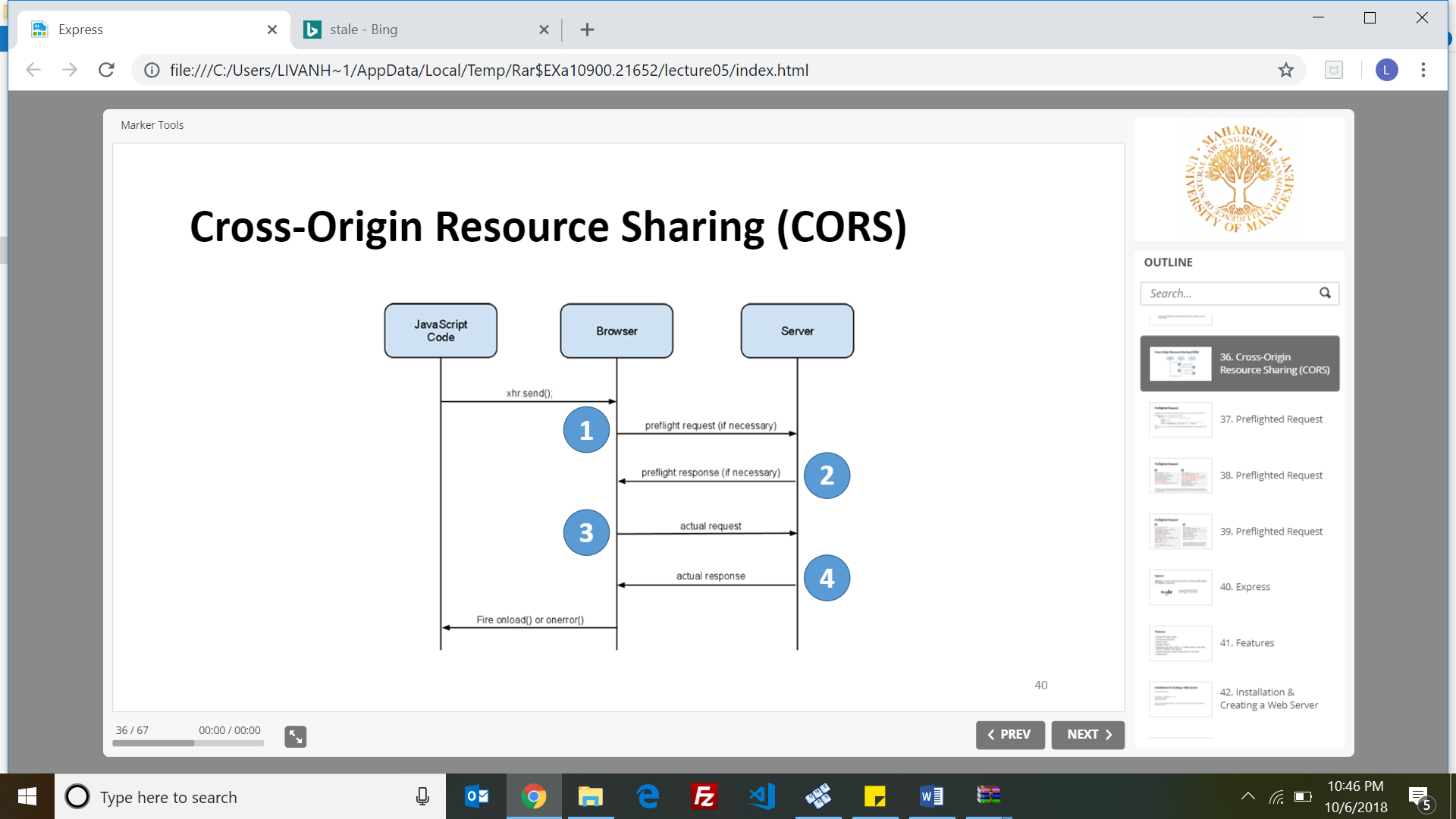
• AJAX calls only send Cookies if the URL you're calling is on the same domain as your calling script.

* **HTTP Access Control using :**



1.Web proxy 2.JSONP  3.Cross-Origin Resource Sharing **(CORS)** headers on the server

* **Same-Origin Policy -** permits scripts running in a browser to only make XHR requests to pages on the same domain



**Simple Requests vs Preflight Requests**

* **Simple requests** are characterized as such they can be made from a browser without using CORS.
* **Preflight Request -** request that does not meet the criteria above is a not-so-simple request, and requires a little extra communication between the browser and the server
* **Express.js** is a web framework based on the core Node.js http module and connect components
* **Features of Express.js**

•Parsing HTTP request bodies

•Extracting URL parameter

•Parsing cookies

•Managing sessions

•Organizing routes with a chain of if conditions based on URL paths and HTTP methods of the requests

•Determining proper response headers based on data types

•Handling errors

* **Installation & Creating a Web Server**

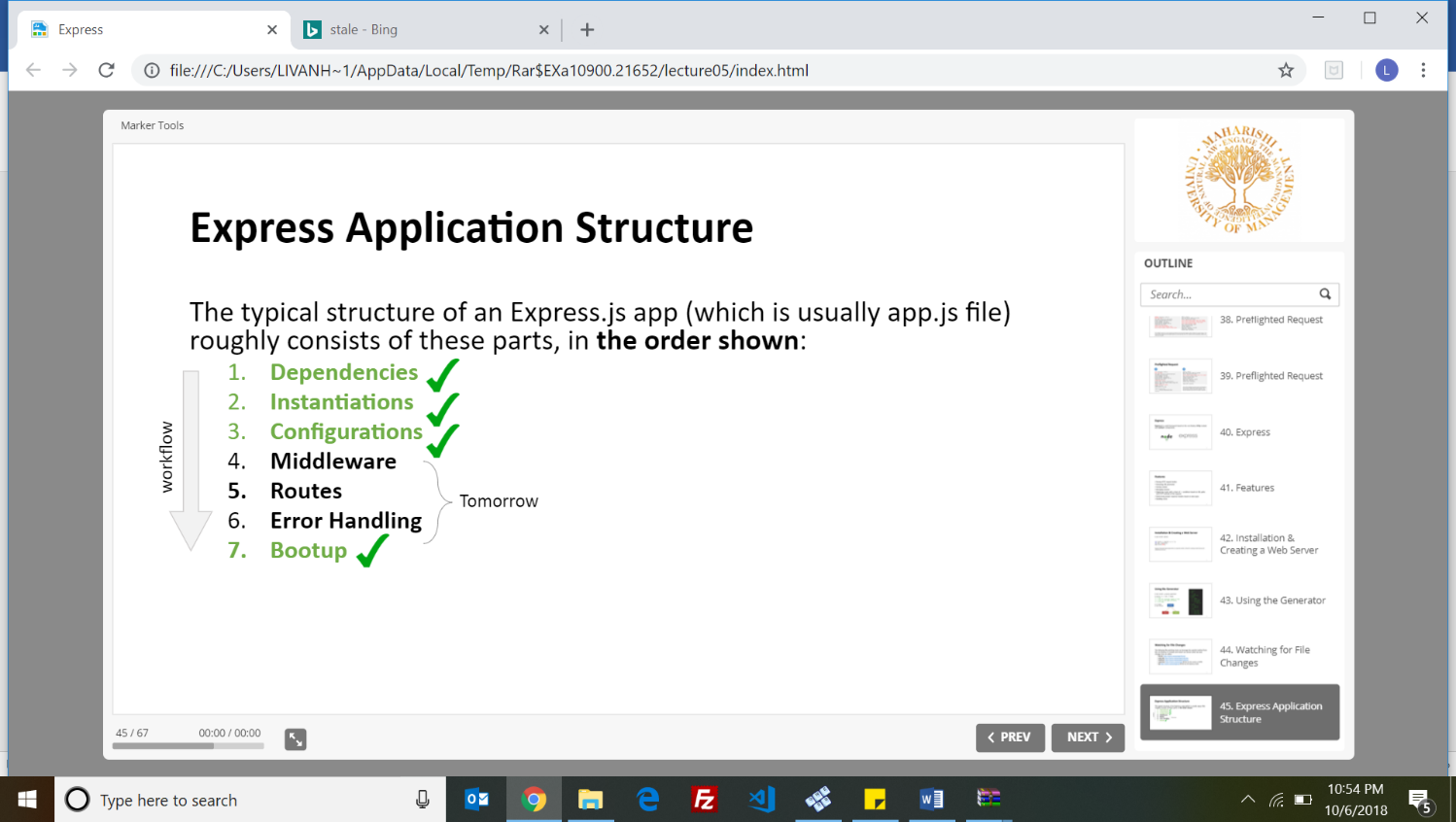
$ npm install express

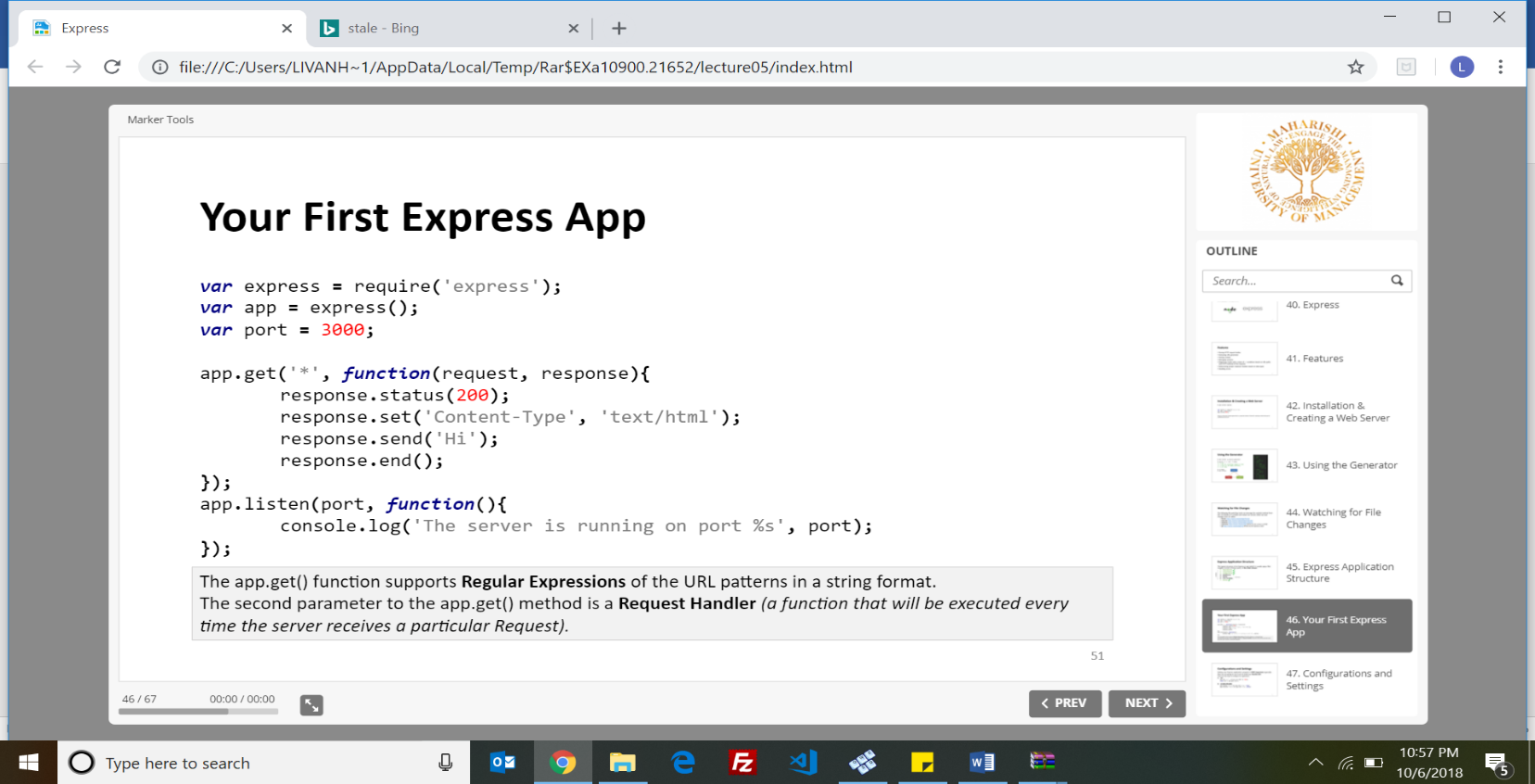
var express = require('express');

var app = express();

app.listen(3000);

* **Express.js Generator** (express-generator) is a separate module. It allows for rapid app creation because its scaffolding mechanism. E.g $ npm install -g express-generator
* **Express Application Structure**



* **First Express App**
* **Two ways to configure an Express application:**

**1.set**

app.set('port', process.env.PORT || 3000);

const port = app.get('port);

**2.enable/disable**

app.enable('etag') === app.set('etag', true)

app.disable('etag') === app.set('etag', false)

**Settings –'env'**

* The better way is to start an app with **$ NODE\_ENV=preview** node app Or to set the **NODE\_ENV** variable on the machine
* The most common values for env setting are: development , test, stage, preview ,production.

**Settings – 'views'**

* has an absolute path to a directory that has all your templates

**Settings – 'view engine'**

* holds the template file extension

**Settings – 'view cache'**

* If 'view cache' is set to false, it allows painless development because templates are read each time the server requests them.
* If 'view cache' is set to true, it facilitates template compilation caching, which is a desired behavior in production.
* If the env setting is production, then 'view cache' is enabled by default. Otherwise it is set to false.

**Settings – 'trust proxy'**

* This setting will permit trusting in the X-Forwarded-\* headers he X-Forwarded-\* request header helps you identify the IP address, Protocol and Port of a client when you use an HTTP or HTTPS load balancer.

**Settings – 'case sensitive routing'**

* Used To disregard the case of the URL paths (its set to false by default)

E.g app.enable('case sensitive routing');

**Settings – 'strict routing'**

* It deals with cases of trailing slashes in URLs.(its set to false by default.

E.g. /users and /users/ will be completely separate routes.

**Settings - x-powered-by**

* sets the HTTP response header X-Powered-By to the Express value (its enables by default).

E.g. app.set('x-powered-by', false)

* Disabling/Removing it from the response is recommended for security reasons, because it’s harder to find vulnerabilities if your platform is unknown.

**Settings – 'etag'**

* ETag (Entity Tag) is a caching tool
* Default value is: true (weak ETag), false (no ETag), and strong (strong ETag)

**Settings – 'query parser'**

* Parse the data sent in the URL after the question mark. Default value is “extended”.
* possible values are

•false: Disable parsing

•extended: Parse based on [qs](https://www.npmjs.org/package/qs" \t "_blank" \o "https://www.npmjs.org/package/qs)(default)

•simple: Parse based on Node’s native query parser, [querystring](http://nodejs.org/api/querystring.html" \t "_blank" \o "http://nodejs.org/api/querystring.html)

**Settings – 'jsonp callback name'**

* default callback name, which is a prefix for our JSONP response
* **Security Headers**

**• CSP** (Content Security Policy) : for Cross Site Scripting (XSS) and data injection attacks.

=> content-security-policy:[CSP\_POLICY]

•**PKP (**Public Key Pins): for attacks with forged certificates

=> public-key-pins:[PKP\_POLICY]

•**STS** (Strict Transport Security): stops the man-in-the-middle attacks by letting sites only be accessed using HTTPS. => strict-transport-security:[STS\_POLICY]

===========================================================================

**LESSON 6 – EXPRESS 2**

* The typical structure of an Express.js app (which is usually app.js file)

roughly consists of these parts, in the order shown: 1. Dependencies 2.Instantiations 3.Configurations 4.Middleware 5.Routes 6.Error Handling 7.Bootup

* They receive the request, response,next objects
* Request Handlers .E.g

const filter = function(req, res, next) {

console.log('filtered');

return next(); };

* **Middleware** is Request Handler, a useful pattern that allows developers to reuse code within their applications and even share it with others in the form of NPM modules.
* Middleware accepts three arguments: 1.request 2.response 3.next
* To use a middleware, we call the app.use()method which accepts:

•One optional string path.

•One mandatory callback function.

E.g. app.use('/register', function(req, res, next) {return next();});

* Express comes with fours built-in middleware :

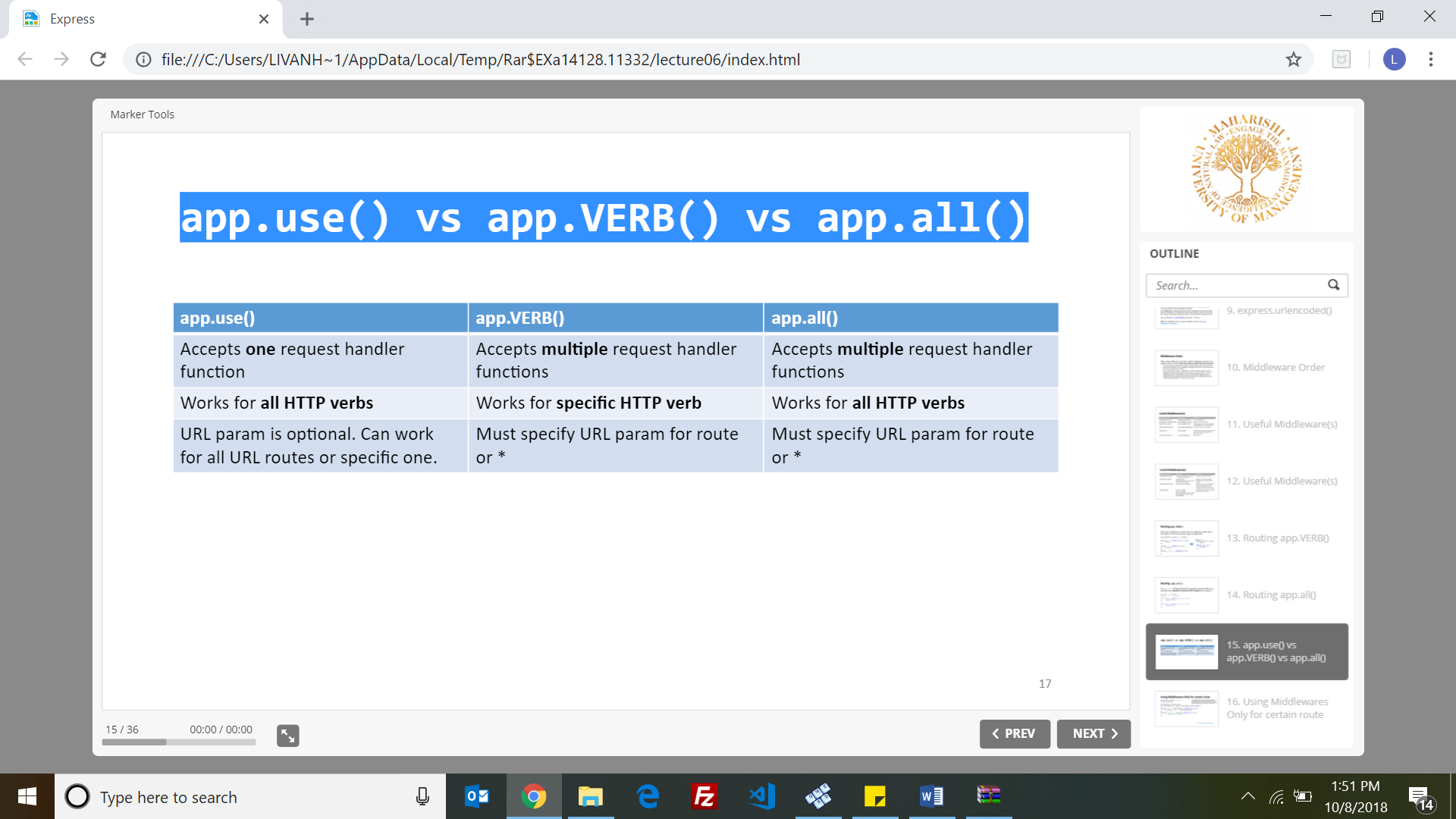
•express.static() : enables pass-through requests for static assets.

•express.json() : parses requests with urlencoded payloads and is based on body-parser.

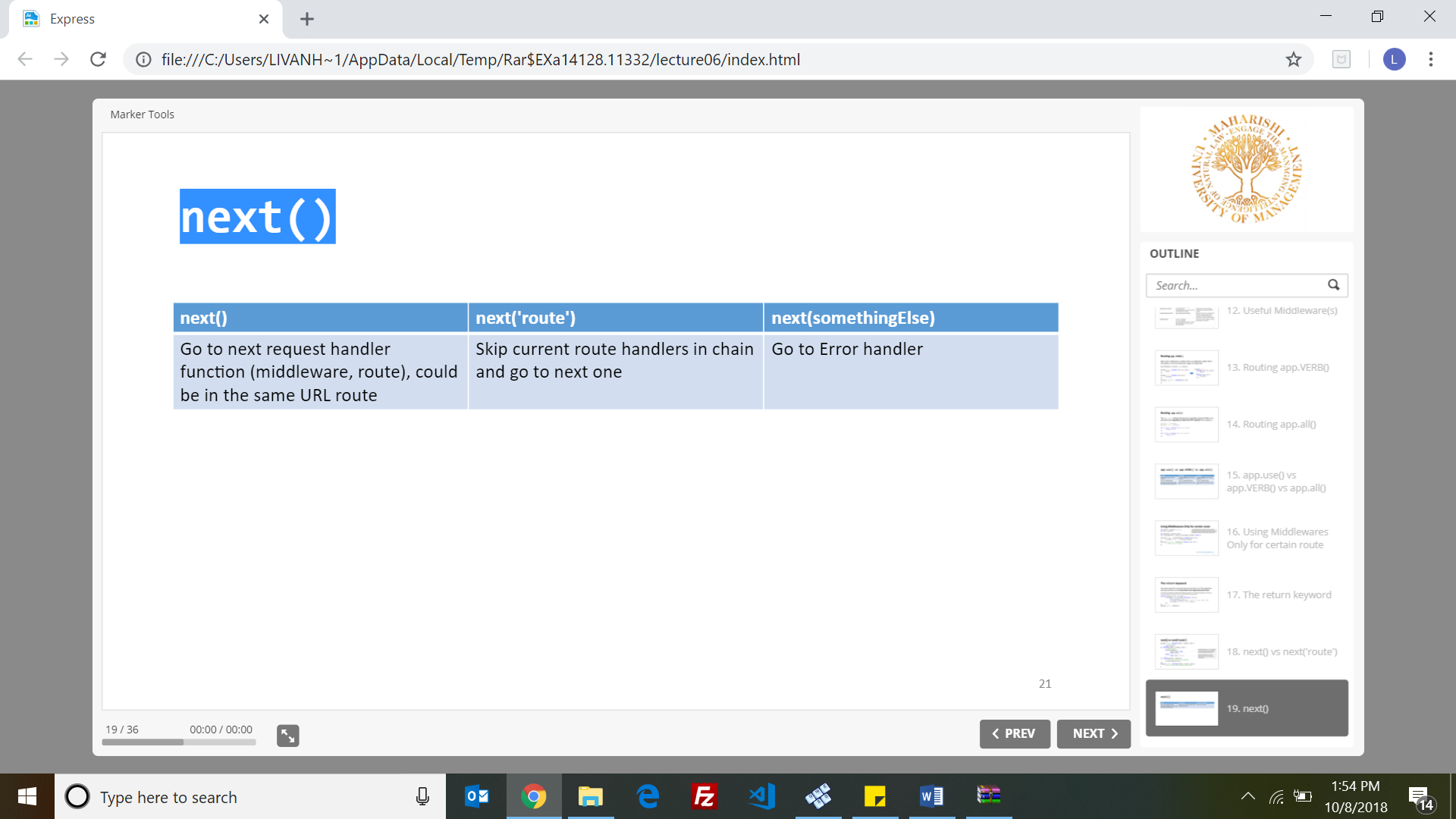
•express.urlencoded() - based on body-parser

•express.Router()

* Middleware statements go before routes.
* The **app.all()** method will execute its specified request handlers on a particular path regardless of what the HTTP method of the request is.
* **app.use() vs app.VERB() vs app.all()**



* The return keyword is essential, because if we don’t use it the application will jump and later we will come back to the original execution stack.
* **express.Router() Middleware** - The **Router** class is a mini Express.js application that has only middleware and routes.
* **next()**



* **try/catch** fails at asynchronous errors. Use try/catch only for synchronous JavaScript/Node.js code only
* **Error Handling in Node:**

const fs = require('fs');

fs.readFile('a\_file\_that\_does\_not\_exist.txt', function(err, data) {

if (err) {

console.error('Error:', err);

return; }

// Otherwise handle the data

});

* **Error Handling in Express**

app.use(function(err, req, res, next) {

console.error(err.stack);

res.status(500).send('Something went wrong!'); });

* error-handling functions have four arguments instead of three: (err, req, res, next).
* You define error-handling middleware last, after other app.use() and routes calls.
* **Helmet** is a collection of security related middleware that provides most of the security headers . E.g

var helmet = require('helmet');

app.use(helmet());

* **Accepting CORS in Express :**

$ npm install cors --save

var express = require('express')

var cors = require('cors')

var app = express()

app.use(cors())

* Express.js doesn’t perform any user/client input sanitation or validation when you use body-parser or query as input data.
* **Security – Server Side Input Validation:** var validator = require('express-validator');

**=======================================================================**

**LECTURE 7 – MONGO DB**

* A record in MongoDB is a document, which is a data structure composed of key and value pairs.
* MongoDB documents are similar to JSON objects.
* Embedded documents & arrays reduce the need for expensive joins.
* Dynamic schema supports fluent polymorphism
* **Non-Relational**

•Scalability and Performance (embedded data models reduces I/O activity)

•Depth of Functionality

•To retains scalability

•MongoDB does not support Joins between two collections

•No relational algebra: tables/columns/rows (SQL)

•Transactions across multiple collections

* Atomic transaction: is either all occur or nothing occur.
* **Key Features of mongoDb:**

•High Availability

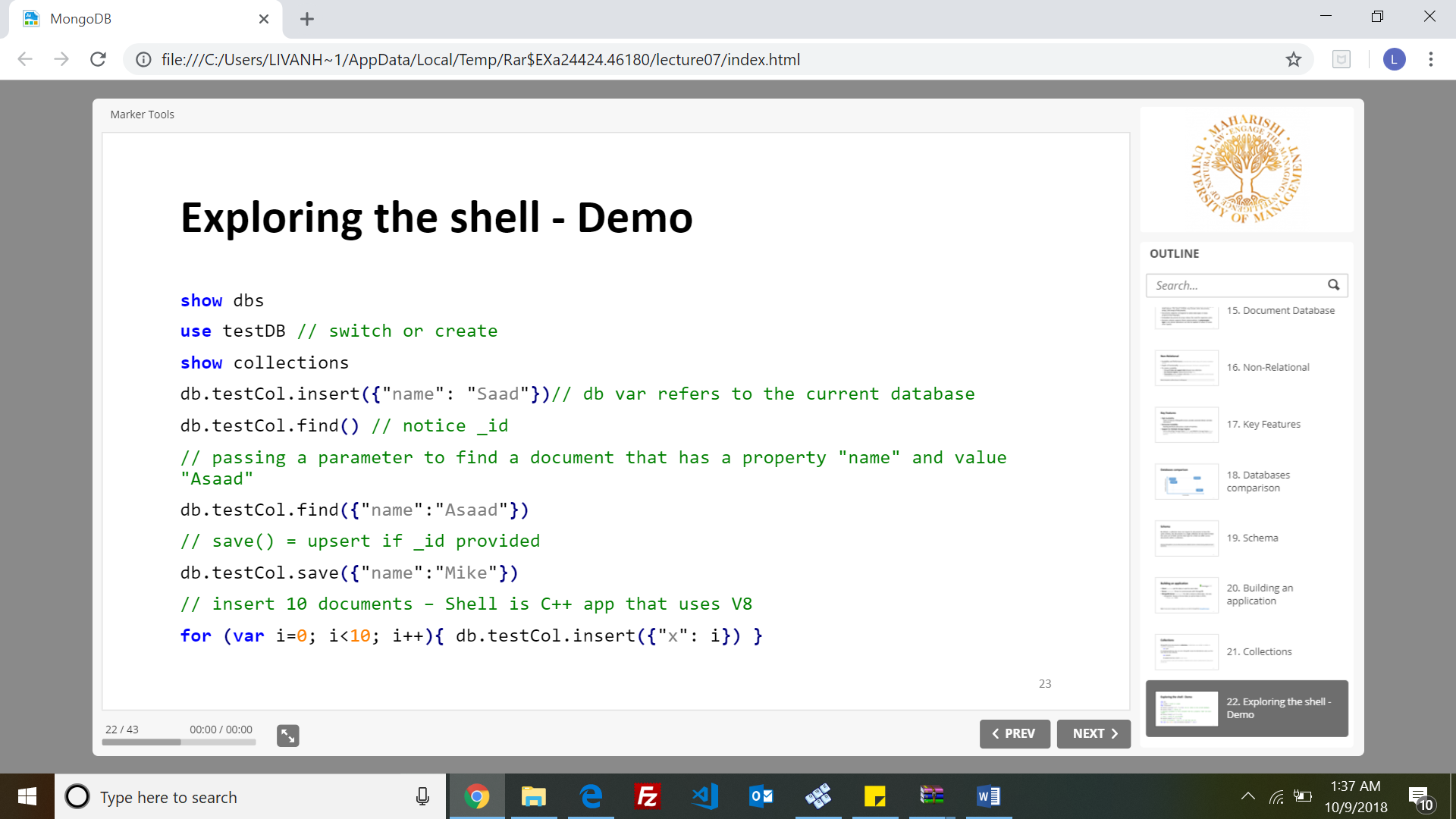
•Horizontal Scalability

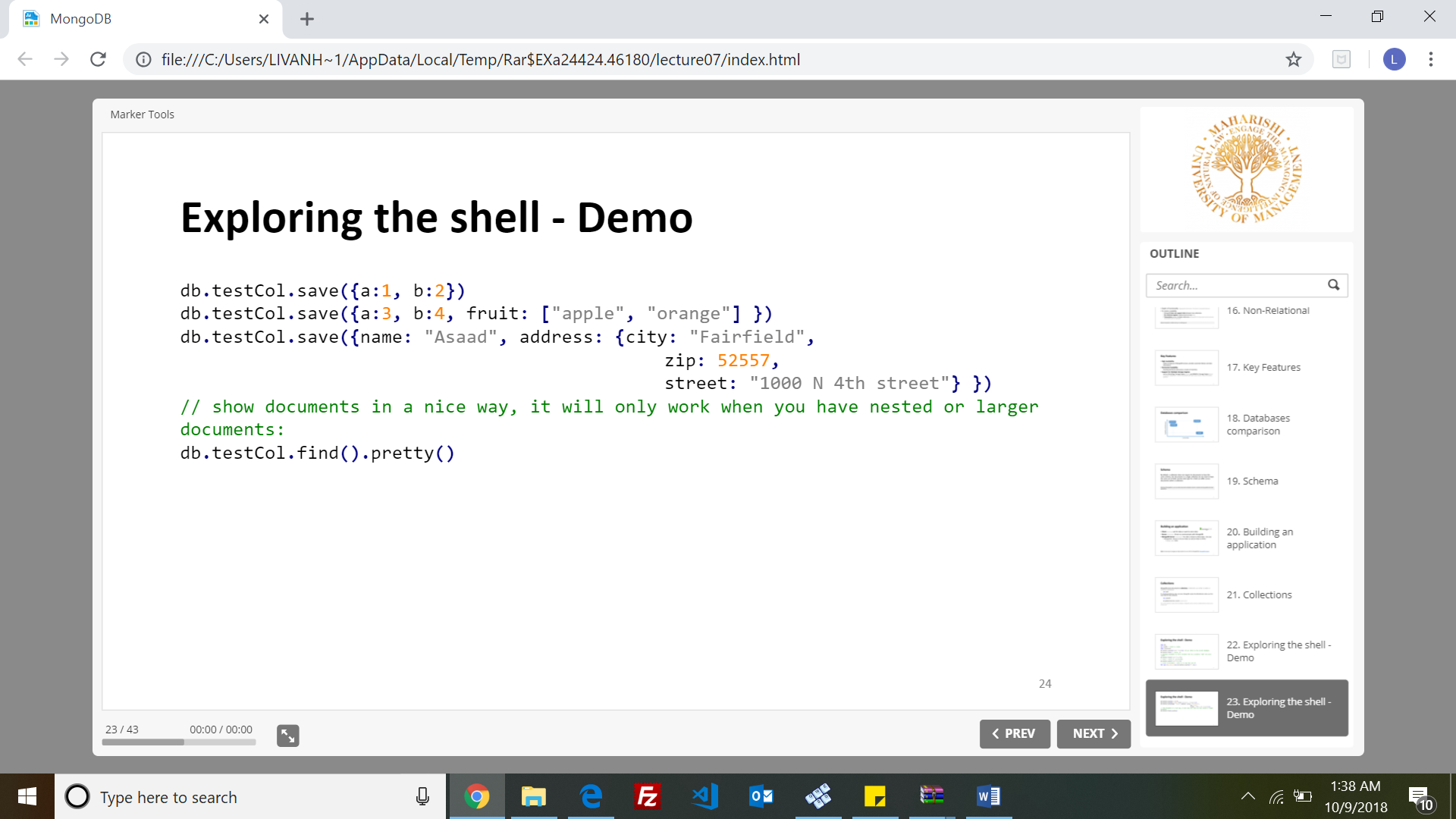
•Support for Multiple Storage Engines

* Databases in mongo compared to other has higher :functionality, performance and scalability
* By default, a collection does not require its documents to have the same schema
* MongoDB stores documents in collections. (**Collections** are similar to tables in relational databases) .E.g creating collection: use myDB
* The insert() operation creates both the database myNewDB and the collection myNewCollection if they do not already exist

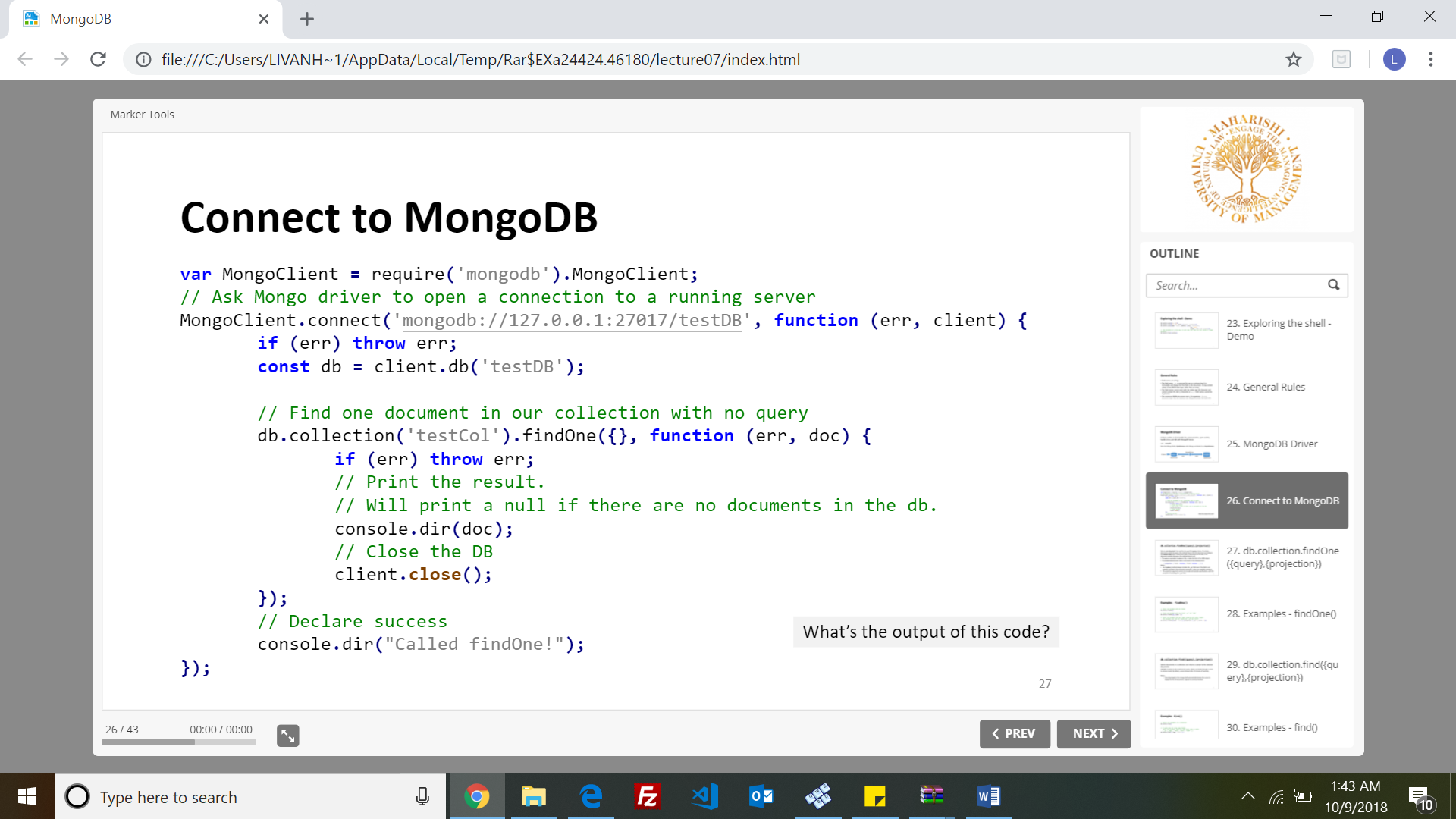
E.g use myNewDB

db.myNewCollection.insert( { x: 1 } )





* The field name \_id is reserved for use as a primary key, and is the first field in the document
* The maximum BSON document size is 16 megabytes. (To store documents larger than the maximum size, MongoDB provides the GridFS API)
* **MongoDB Driver -** A library written in JS to handle the communication, open sockets, handle errors and talk with MongoDB Server. : => npm i mongodb
* **Connect to MongoDB**



* **db.collection.findOne({query},{projection}) -** Returns one document that satisfies the specified query criteria
* The findOne() method includes the \_id field even if the field is not explicitly specified, unless you explicitly exclude it.
* **Examples : findOne()**

// return one document with all fields : db.testCol.findOne()

// return one document with two fields "\_id" and "name"

db.testCol.findOne({}, {name: 1})

// return one document that has "name" property with value "Asaad",

   this document will have all fields but "\_id" and "birth"

db.testCol.findOne({name: 'Asaad'}, {projection: { \_id: 0, birth: 0 }})

* **db.collection.find({query},{projection}) -** Selects documents in a collection and returns a cursor to the selected documents
* **cursor**: A pointer to the result set of a query.
* By default, cursors timeout after 10 minutes of inactivity.
* Executing find() in the mongo shell automatically iterates the cursor to display the first 20 documents. Type “ **it** ”to continue iteration.
* **Examples - find() -:**

// returns all documents in a collection

db.testCol.find()

* **count()** – used to get the count of all the documents that match a certain criteria

**E.g :** db.testCol.count() , E.g: db.testCol.count( { "grade": "A" } )

* MongoDB is **agile**, there will be no need to have same structure in documents. Every document can have its own structure.
* pretty(): makes outputs look better to read. E.g : db.testCol.find().pretty()
* **MongoDB Schema Design - In** MongoDB we use **Application-Driven Schema,** which means we design our schema based on how we access the data.
* The only scenario we cannot embed is when data exceeds 16 MB and we need to put it in separate collection
* **Transactions vs Atomic Operations**

In MongoDB, our data is usually located in one document, and because most operations are Atomic, we accomplish the same thing without the need for transactions.

* **Benefits of Embedding**

•Improved read performance

•One round to the DB

* When embedding, Take into considerations the following:

•Frequently of access and the way you want to access the data

•Size of items (16M)

•Atomicity of data

* The only scenario we cannot embed is when data exceeds 16 MB and we need to put it in separate collection.

===========================================================================