PART 1

- Explanation of what is node.js
- Demo Installation.
- You can do many things with node - read files, write files etc. However we are primarily concerned with using node as a web server.
- We can download modules to help us with this. The tool that allows us to browse and choose modules is called npm.
- The web development framework that we will be using is called express.
- Your job as a node developer is to write the callbacks that respond to events and primarily events that transmit data through the http protocol.
- Example of the callback pattern is prevalent in JQuery

Create a folder called todo on your desktop

Create a package file inside todo with express

```
{
    "name": "todo_list",
    "version": "1.0.0",
    "main": "app.js",
    "author": "",
    "license": "ISC",
    "dependencies": {
```

```
        "express": "*"
},
    "description": ""
}
```

In the terminal from the todo directory type npm install

App.js

From here on out all our code will go in a file called: `app.js`

This is placed at the top level of project directory. Create it now.

Basic webserver setup

```
var express = require("express");
var app = express();
app.listen(3000, function(err) {
    if (err) {
        console.log('Server is not working ');
    } else {
        console.log('Server works')
    }
});
```

Go to localhost:3000 to view missing route.

Create a basic route

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

What are the request and response arguments?

Request: Used to request data FROM a client (Typically a form field)

Response: Launches some code when a user goes to a specific URL.

(This will become more clear as we go)

Nodemon

Each time we modify the code we need to refresh

We can automate this by installing a module called Nodemon.

```
npm install -g nodemon
```

Setting up a default route

If you want to set up default route you can create a route at the bottom of the page with an asterisk

selector.

```
app.get("*", function(req, res) {
    res.redirect("/")
});
```

How to Render an HTML page?

Setup a directory called *public* in the project folder.

Then place static_example.html inside of it.

Add the following code to the app.js file.

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

Go to:

localhost:3000

localhost:3000/static_example.**html**

You can create routes to serve the static HTML files

```
app.get('/static', function(req, res) {
      res.sendfile('public/static_example.html');
});
```

View Engine

The previous way of rendering html allows you to do so by specifying html files in the public directory. However a better way is to use a view engine. View engines give us the ability to use “magic” variables that can travel directly from the server to the html code in our webpage.

View Engine Setup

To use a view engine first we need to pick one, there are many choices for this. I prefer to use one called handlebars as I think it is the easiest of the bunch to learn and use.

To get started we must first import the “express-handlebars” module.

We can do this by adding it to our package and then run `npm install` from terminal

We then need to require the module in our app.js file:

```
var expressHbs = require('express-handlebars');
```

Then we set a views directory in our project folder.

Then we need to add the following code to set handlebars as our view template software:

```
app.set('view engine', 'hbs');
app.engine('hbs', expressHbs({
    extname: 'hbs'
}));
```

To create view files you need to use the extension .hbs

Create one called: Index.hbs

You can then render these files AND send variables from the server to the client like this:

```
app.get("/", function(req, res) {
    res.render("index", {
        variableFromServer: "hi there"
```

```
    });
});
```

In index.hbs type the following:

```
<p>{{ variableFromServer }}</p>
```

Looping through data with Handlebars

If the data in `variableFromServer` happens to be an array or object. You can loop through the data of it and render each piece of data to the screen with the following syntax.

```
{{#each item}}
<p>{{this}}</p>
{{/each}}
```

PART 2

Up to this point you have learned how to get data from the server to the client. Now we are going to explore how to get data from the client to the server.

Getting data from the client to the server

You are now going to create a form field and on submission the data from it will be sent to the server. To allow this to work we need to install a module called body-parser.

Add it to the package.json file and run npm install.

When installation is done place the following code in the app.js file.

```
var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({
```

```
    extended: true
}));
```

Creating the form field

Place the following into your index.hbs file. When a user submits the form it will send the data over a route: ”/client_to_server”

```
<form method="post"  action="client_to_server">
      <input type="text" name="userData">
      <input type="submit">
</form>
```

Capturing the data on the server

The following code responds to a posts request that are sent over a route called “/client_to_server”.

The data from the client is transmitted via the `req.body` property as an object.

You then need to redirect the page otherwise it will idle.

```
app.post("/client_to_server", function(req, res) {
          console.log(req.body.userData);
          res.redirect("/");
});
```

Updating the data on the page

You can update the data on the page to reflect the submitted data.

```
var data = undefined
app.get("/", function(req, res) {
          res.render("index", {
              item: data
          });
});

app.post("/client_to_server", function(req, res) {
          console.log(req.body.userData)
          data = req.body.userData;
          res.redirect("/")
})
```

If the data variable in this example was an array you could push form input to it, then loop through it on the client to create a todo list.

```
var data = []
app.get("/", function(req, res) {
    res.render("index", {
        item: data
    });
});

app.post("/client_to_server", function(req, res) {
```

```
    console.log(req.body.userData);
    data.push(req.body.userData);
    res.redirect("/");
});
```

In your index.hbs file:

```
{{#each item}}
      <p>{{this}}</p>
{{/each}}
```

PART 3

How to delete individual todos by clicking on them.

Deleting individual todos

To complete this we will need a way to access individual Todo items, we can accomplish this by adding an id property to each todo object via a counter. Add the following code to app.js

```
var data = [];
var counter = 0;
```

```
app.get("/", function(req, res) {
    res.render("index", {
            item: data
        });
});

app.post("/client_to_server", function(req, res) {

        counter += 1
        req.body.id = counter; // add id to object as identifier
        console.log(req.body); // Request entire object (with id)
        data.push(req.body);
        res.redirect("/");
});
```

Adding a link to each element that allows a get request based on ID

You are now going to set each todo item inside an html link, and when you click on this link it will send a get request to the server that contains the id of the todo. We can then use this data to match the id of the element you clicked to the corresponding todo in our data object and then delete it.

```
<ul>
    {{#each item}}
    <li><a href="delete/{{this.id}}">{{this.userData}}</a></li>
    {{/each}}
</ul>
```

Looping through the data object and deleting it based on ID

```
app.get("/delete/:id", function(req, res) {
          var id = +req.params.id; //  convert id to integer
          data.forEach(function(val, index, arr) {
                    if (val.id === id) { // if id matches
                              data.splice(index, 1);//remove object from
  array
                   };
          });
          res.redirect("/");
});
```

BONUS! PART 4

You will now add functionality that will give the user the ability to edit individual todo items. The end result will be the following:

- User will click a link that will open a new page that allows them to edit the todo item in a form field.
- When they are done editing the todo item and click submit they will be taken back to the homepage and the todo item will reflect the new edit

---

In the index.hbs modify the code to look like the following. This will add a new link titled edit which will direct the page to a url that contains the todo items id.

```
{{#each item}}
      <li>
            <a href="delete/{{this.id}}">
                  {{this.userData}}:
            </a>
            <a href="edit/{{this.id}}">
                  <b>Edit</b>
            </a>
      </li>
{{/each}}
```

Create edit route

Create a variable at the top of the app scope and name it updateId. Then add the following route code:

var updateId = undefined;

```
app.get("/edit/:id", function(req, res) {
           res.render("edit", {
            id: req.params.id    //_________________
          })
})
```

Create a views file named **edit.hbs** with the following form field code:

```
<h1> Editing</h1>
<form method="post" action = "/update/{{id}}">
      <input type="text" name = "userData">
```

```
      <input type = "submit">
</form>
```

Now when you click the EDIT link of a todo item on the homepage it will take you to a route with edit/id.

Example: localhost:3000/edit/1

Create the update view and route

Now we are going to create a post route (**you won't need a view file**) called update. The purpose of this is to get the id and then match it against the object in the data array and update its contents.

```
app.post("/update/:id", function(req, res) {
                var referenceID = +req.params.id;
                data.forEach(function(val, index, arr) {
                          if (val.id === referenceID) {
                              val.userData = req.body.userData
                    }
          })

          res.redirect("/")

})
```