**Setting up Project Dependencies**

We want to setup our Development Environment. We're going to have a Node Web server and going to write React components for the Front End. The Node Server will then render initial static HTML from the our Front End React components. We also want a Node API Server to talk to the data-base. We will create one Node project to perform all 3 tasks.

Check out article as jsComplete.com/reactfull

Create package.json file to store general info about the project and track its dependencies - npm init

There's two types of packages, the main dependencies package is used in production. Dev dependencies are only used in a local development environment.

Main dependencies, to create a Node Sever we will use Express js – npm install Express

Mongo DB – to connect from Node we need a driver. The official driver – npm install mongodb

Front End dependency will be React – npm install react react-dom. The React library will be used to describe the UI and React Dom to render both those UI's on the front and back end.

Now our Dev dependencies…

Npm install -D (the capital D marks the dependency as a Dev dependency)

Webpack and Webpack CLI

Since we're going to be writing modular code everywhere we need a tool to translate modular code into something Browsers understand. Webpack is the most popular tool for that job.

**Modular programming is** a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.

Express is serving the public files directly with its static middleware

Api folder is for the back-end Server

**Course details**

Learn by doing! Learn full-stack JavaScript development by building a web application with MongoDB, Node.js, and React.js.  
  
Learn how to use Node as a web server and an API server, how to consume data and build user interfaces with React, and how to read and write data with a MongoDB database. React allows you to build full-featured, data-driven applications faster and more efficiently than what is possible with JavaScript alone.  
  
Samer Buna introduces modern JavaScript programming concepts, with an emphasis on functional methods and open-source tools.

**Learning objectives**

* Using modern JavaScript features
* Configuring webpack and Babel
* Working with native Node modules
* Creating an Express server
* Working with React components and JSX
* Loading and working with test mock data
* Fetching data from a remote API
* Working with data in MongoDB
* Isomorphic rendering on the server
* Front-end routing and back-end routing

## **The HTTP/HTTPS modules**

## **Using the HTTP module as both a Browser (Client) and Server Side**

## The HTTP modules are Core modules so you do not have to import them.

## Request or get method to get data. You can pipe | the output to paginate it.

import https from 'https';

https.get('https://www.lynda.com', res => {

console.log('Response status code:', res.statusCode);

});

**Using the HTTP Module as a Server**

Let's do the equivalent of what we did with the HTTP module, but with Express instead. We first need to import Express, and to create a server with Express, we just invoke the imported Express variable as a function. If you read the source code of the Express package, you'll find out that the default export there is just a function, similar to the HTTP module. We also do a .listen method on the Express server. I'll read the port from the configuration file this time. We just import config and use config.port in the listen call. The second argument to the Express listen is the success handler, which is just a function. So let's output a line that the Express server is listening on the configured port. Instead of listening to a single request event, an Express server also handles server side routing for us. So it exposes an API to listen to certain routes. We do a server.get, specify the route we're interested in as the first argument, which is the route / in this case, and the second argument to this .get call is the event handler, which similar to the HTTP module, receives both a request and a response object. In Express, we can use other methods to send things to the requestor. For example, to send a simple string, we can use response.send here. Let's test, curl http 8080 and we get Hello Express. Let's add another route. Similar syntax, this time /about.html, and here we're going to send The about page. Let's go ahead and test that, curl /about.html and we get The about page. Let's also make sure that we can access this on the web. So Hello Express, /about.html, and about.html is The about page. A web server however does not return simple strings like these, it returns the content of a file, like HTML or JS. So let's create a very simple about.html file in here, and I'm going to copy here a simple HTML template for it, and we want this template to be our reply to our requestor for the /about.html route. We can simply use the fs core module. I'm going to go ahead and import fs from fs, and inside about.html, the API is readFile, the path to the file, then the callback is going to give us access to the file's data as a buffer, so we can send as a response data.toString. And we can test this by refreshing the about page, we get the HTML file. It's really that simple, but it can actually be simpler. Express has a static middleware that we can use to automatically serve static assets like this about.html file. All we need to do is this magic line, .use is how we put a middleware in the Express middleware stack, and public is where we want our static assets to be hosted on the file system. With this line, we don't need to handle the route for /about.html, or even use the fs module. We just need to move about.html to under public, and Express will take care of serving its content to a requestor. Restart the server, and make sure the about.html is still working. In a production environment, we should manage static assets separately from the node server, as there are much faster options like NGINX. To keep things simple in development though, I'm just going to use the simple Express static middleware. With Express, we can also manage a group of routes in their own module, instead of having everything here in server.js. For example, we're going to manage all API requests in this API module index.js. So we import Express here, then we create a route object by calling the router from channel in Express. If you're now wondering how come we can invoke Express itself, and we can also invoke properties on Express, remember that a function in JavaScript is just an object, so we can attach other properties on that object. This router object is similar to the server object we used before, we can define .get calls on it and handle them in the second argument. This is an API call, so I'll send a JSON response by sending an object here. Finally, to be able to use this router, we need to export it. In the main server.js, we first import this API router, and we can use it just like any other Express middleware. First argument is the route prefix, and the API router is the second argument. To test this, go to /api, and we should see the empty data object we send in the API module.

import https from 'https';

//After we import it, we use createServer call to create the Server

const server = http.createServer();

// Use the listen method to run the Server on a certain port

server.listen(8080);

// Create Server gives us an event emitter object which we can subscribe to. The most important emitted event is the request event. The callback for this event receives two special objects (req, res). A request object and a response object. The user initiating the request will see anything we write to the response object. While we can use the request object to read things from the request. For example, what url was requested? Were there any query string parameters with the request and many other things. Let's write a simple hello http string back to the requester. The response object is a writable stream, which means we can use it to stream data to the user and that's extremely powerful. Let's write another line to the user after three seconds using a timer like this and after that let's go ahead and terminate the stream response with the .end call.

server.on('request', (req,res) => {

})