**Topic 1: Introduction to JSON Web APIs**

**Introduction**

In this week's topic you will be introduced to the concept of **web APIs**. You will get an initial idea of why they are useful, and will start to write a simple web API using Node.js and Express. You will also be introduced to ***goormIDE***, a cloud-based environment which you can use to develop Node applications.

**Is HTML always the best output format?**

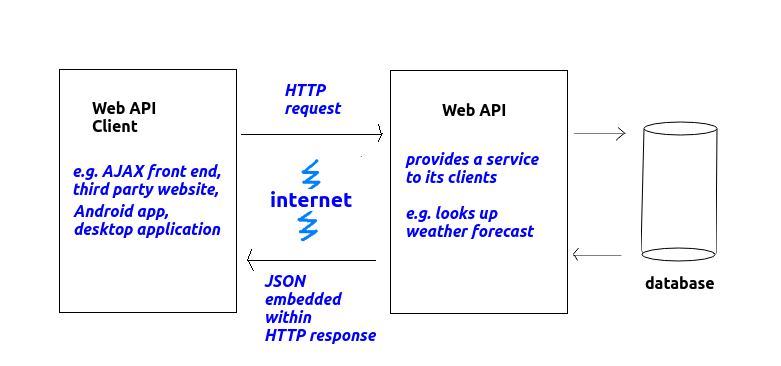
Think about the work you did last year in Web Technologies, in which you wrote a simple Node/Express server to search a database table and display the results using EJS. You generated ***HTML*** as the output format, so that the results look good in a browser.

The problem with HTML, however, is that it is a format ***specific to a browser***. This is fine if we are simply writing a website to be viewed by an end-user in a browser. But the end-user is not the only potential user of the data. ***Other applications*** might want to make use of the content, for example a weather app running on Android or iOS might want to access weather forecasts from a weather website. Apps do not typically use HTML, they use their own user interface, so it would be better if our server delivered the data as ***raw data***, without any presentation or formatting. An example of a raw data format is ***JSON***, which you saw last year in COM419. Raw data could then be easily processed by ***any front end***, for example:

* An AJAX application. As you saw last year, AJAX applications send requests to servers, and in many cases, the server sends back raw data, such as JSON, which can then be ***parsed*** and integrated into the AJAX front-end.
* A smartphone app, as we have seen.
* A third-party website, which might want to use data provided by our server. For example, a tourist website for a particular city might want to display a weather forecast for that city, using a server which provides weather information. This would be easier if the weather server sent back raw data, rather than HTML. With HTML, the relevant data (the weather forecast) would have to be extracted from an HTML page (containing formatting, styling etc) which would be difficult to parse compared to concise, well-structured raw data such as JSON.

**Web APIs - Introduction**

A ***web API*** is an application running on a web server which provides ***raw data*** (such as JSON) to other applications (***client applications***), as we saw above. Web APIs receive HTTP requests from clients, process the request, and deliver the response back as an HTTP response - but unlike the simple servers you saw last year, the response is delivered as raw data, rather than HTML. This is shown on the diagram below:



Here are a few examples of web APIs and their clients:

* ***Flights:*** An airline provides flight times and a booking service available on its own website. However, other organisations might wish to make use of this. Third-party sites such as Skyscanner (which can search all airlines for the cheapest flight on a given date) or travel agents may wish to use the information provided by the airline, and display it to the end-user. In these cases, code on the Skyscanner web server, or a standalone booking application at a travel agent (which might be written in Java, for example) might connect to the airline's web API, and receive flight information back. The clients (Skyscanner and the travel agent application) could also contact the web API to perform bookings.
* ***Weather:*** We mentioned this example above, briefly. A weather web API might make available current conditions in various cities around the world. Then, a tourism website (e.g. for London or Paris) could connect to the weather web API and display the current conditions in that city, retrieved from the web API, integrated seamlessly into the layout of its own site. Furthermore, a smartphone app could connect to the web API and show the weather for the user's current location within the app's interface.
* ***Mapping:*** electronic maps are typically stored in a database, which contains information about the roads (name, number, type) plus the road's route as a series of coordinates (representing points on the Earth's surface). A mapping web API could make available this mapping information as raw data, so that client websites could use this raw data to draw a map in their own style and colour scheme. OpenStreetMap ([openstreetmap.org](http://www.openstreetmap.org/)) is an example of such a mapping web API.

**Raw data is easier to parse**

Web APIs obviously send back information to their clients, but as we have seen, a raw data format such as JSON is preferred to HTML. Why is this? HTML is not considered a good idea because it contains not only data, but also page structure information (headings, paragraphs, tables etc). A client website using the web API, or an app, might wish to arrange the information in a different way.

So what we want is a format which represents the data, and the data alone. There are a number of formats we can use, including JSON (JavaScript Object Notation), and also ***XML*** (eXtensible Markup Language). In this module we will focus on JSON, because it's the leading format, it is easy to generate on the server and to parse (interpret) on the client, and you have met it already.

**JSON - JavaScript Object Notation - Revision**

JSON uses JavaScript syntax (hence the name) to represent data as it gets sent across the web. As you saw last year, JavaScript uses curly brackets {} to represent objects (similar to Python dictionaries in the sense that they consist of key/value pairs, though you can also add ***methods*** to objects) and square brackets [] to represent arrays. So with JSON we reuse this syntax to represent data, using curly brackets {} to represent a single entity (such as a person, a song or a film) and square brackets [] to represent a collection of entities (i.e. an ***array*** of entities).

Here is an example of a JSON object representing a single student.

{

"name": "Tim Smith",

"username": "2smitt82",

"course": "Computer Studies"

}

Note how the JSON object representing the student is defined by curly brackets { and }, and inside the curly bracket, we specify each property of the student (name, username and course) and the corresponding value ("Tim Smith", "2smitt82", and "Computer Studies", respectively). A colon (:) separates the property and the value, and a comma separates each property/value pair.

The next example shows a ***collection (array) of students***. Note how we use the JSON array syntax [ and ] to define the collection, how each individual student object is represented by curly brackets { and }, and how each student object within the array is separated by a comma.

[

{

"name": "Tim Smith",

"username": "2smitt82",

"course": "Computer Studies"

},

{

"name": "Jamie Bailey",

"username": "1bailj39",

"course": "Computer Studies"

},

{

"name": "Deep Patel",

"username": "0pated61",

"course": "Networks and Web Design"

}

]

**Node and Express - Revision**

In other modules you have looked at ***Node.js*** and ***Express***. This section is provided for revision. Node.js can be downloaded from [the website](http://www.nodejs.org/).

**npm**

As you have seen, Node also comes with a ***package manager*** called ***npm***. This allows you to use add-ons to Node (***packages***) which perform additional tasks, not part of the core Node.js distribution, such as communicating with a database. To use npm to install new packages, you enter:

npm install <packagename>

at the command-line (e.g. DOS or Linux shell prompt).

**API documentation**

See [here](https://nodejs.org/api/) for full Node API documentation.

**Creating a Node web server Using Express**

It is possible to create a webserver from first principles in Node, using the HTTP module. However, as you have seen in previous modules, it's easier to use a pre-existing web server framework, and ***Express*** is perhaps the most widely used. The npm command below will install it:

npm install express

This will install it locally to your current project. Or to install it globally so that it's accessible anywhere on your machine, for all projects and all users:

npm install -g express

On Linux and other Unix-based systems, this requires 'sudo' rights (admin privileges).

**Hello World with Express**

The example below is the Hello World with Express.

const express = require('express');

const app = express();

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

res.send('Hello World from Express!');

});

app.listen(3000);

We require the express module, then create an Express app object with express().

With Express, we set up ***routes***. A route is a combination of a particular URL and a handler function for that URL. The URLs for a particular API are called ***API endpoints***. So a route can be described as a handler for a particular endpoint, or set of endpoints (we can set up a single route to handle multiple URLs). In this example we are simply handling the top level, 'root' endpoint, so to request it from a browser, we would enter:

http://localhost:3000/

without any parameters supplied.

The handler for the endpoint is a function which takes two parameters, req representing the HTTP request and res representing the HTTP response. In this example we call the send() method of the response object to send back ***Hello World from Express!*** to the client.

**Running the server**

Remember that you use the node command to run a Node application. So if the above basic Express app is saved as app.js, you would run it with:

node app.js

This will run the server continuously in the background. To stop the server, you need to press Control-C. You must do this if you change the server code as the server will not automatically reload if you change it!

**Multiple routes - example**

Here is an example which sets up two separate routes:

const express = require('express');

const app = express();

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

res.send('Hello World!');

});

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

res.send(`There have been ${Date.now()} milliseconds since 1/1/70.`);

});

app.listen(3000);

This example sets up two routes /hello and /time, to handle the corresponding endpoints. The first will send back 'Hello World!' and the second will send back the number of milliseconds since January 1st 1970.

Note that the /time route uses the ***backtick syntax***, which may be new to you. If we use backticks (`) for strings, we can ***embed expressions directly in the string by surrounding the expression with curly brackets { } and preceding the curly brackets with a dollar sign $***. This avoids us having to use string concatenation with the + operator and leads to more concise syntax. So note how we embed the expression Date.now(), which gets the current time in milliseconds, in the string by surrounding it with curly brackets and preceding it with a dollar sign:

`There have been ${Date.now()} milliseconds since 1/1/70.`

**URL parameters**

Express allows us to easily specify URL parameters. A parameter is a value within a URL which the server will process. So for example we might have an /artist route which takes an artist as a parameter. This route would match both of the endpoints below; Oasis and Madonna are the parameters to the same /artist route.

http://localhost:3000/artist/Oasis

http://localhost:3000/artist/Madonna

You set up the route to contain named parameters and then you can use these named parameters in your JavaScript. This example greets a user, with the name supplied as a parameter:

const express = require('express');

const app = express();

app.get('/greet/:personName', (req,res)=> {

res.send(`Hello ${req.params.personName}!`);

});

app.listen(3000);

Note how this will setup the route /greet followed by a parameter :personName representing a person name. So URLs such as

http://localhost:3000/greet/John

http://localhost:3000/greet/Jane

will be matched, and the greeting with the specified name will be sent back. These two examples will respond with:

Hello John!

Hello Jane!

respectively.

To access the named parameter from the JavaScript code, use the req.params object with the name of the parameter. So in this example req.params.personName will reference the :personName parameter and the server will send back a response containing "Hello" plus the person name supplied as a parameter.

**Outputting JSON from an Express server**

Having covered some basic Node/Express revision, we will now look at how we can ***generate JSON*** from an Express server.

Here is an example to generate JSON from a JavaScript array. To send back JSON to the client, we use res.json() rather than res.send(). Remember that in JavaScript we use the square brackets [ and ] to define arrays.

const express = require('express');

const app = express();

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

const monthLengths = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31];

res.json(monthLengths);

});

app.listen(3000);

This will produce the JSON below when the jsonExample endpoint is requested:

[ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ]

Here is an example showing an ***object***:

const express = require('express');

const app = express();

app.get('/jsonExample2', (req, res) => {

const obama = {"name" : "Barack Obama",

"age" : 56,

"nationality" : "US",

"job" : "US President 2008-16" };

res.json(obama);

});

app.listen(3000);

This would give the JSON below when the jsonExample2 endpoint is requested:

{"name" : "Barack Obama", "age" : 56, "nationality" : "US", "job": "US President 2008-16"}

and finally here is an example showing an array of objects:

const express = require('express');

const app = express();

const politicians = [

{

"name" : "Barack Obama",

"age" : 56,

"nationality" : "US",

"job" : "US President 2008-16"

},

{

"name" : "Donald Trump",

"age" : 75,

"nationality" : "US",

"job" : "US President 2016-20"

},

{

"name" : "Joe Biden",

"age" : 79,

"nationality" : "US",

"job" : "US President 2020-"

}

];

app.get('/jsonExample3', (req, res) => {

res.json(politicians);

});

app.listen(3000);

This will produce this JSON when the jsonExample3 endpoint is requested:

[

{

"name" : "Barack Obama",

"age" : 56,

"nationality" : "US",

"job": "US President 2008-16"

},

{

"name" : "Donald Trump",

"age" : 75,

"nationality" : "US",

"job": "US President 2016-20"

},

{

"name" : "Joe Biden",

"age" : 79,

"nationality" : "US",

"job": "US President 2020-"

}

]

The simple rule is that JavaScript arrays will map to JSON arrays, and JavaScript objects will map to JSON objects.

**Setting the correct content type**

A server should let clients know that JSON is coming back from the server, so that the client knows that they need to process the response as JSON, rather than something else. For example, web browsers will format the JSON nicely to make it easier to test and debug a web API. To do this the Content-Type HTTP header must be set to application/json. ***The great thing if you are using Express, however, is that res.json() will set the content type correctly for you.***

**Exercise 1**

**NOTE the following**:

* To test, you may need to install the JSONViewer plugin, from [the Chrome web store](https://chrome.google.com/webstore/detail/json-viewer/gbmdgpbipfallnflgajpaliibnhdgobh?hl=en). This will format the JSON nicely so it's easy to see the output.
* Some of these may be very easy if you have done the database module last term. So feel free to skip Questions 1, 2 and 4 if you are confident with them: they are mostly intended for revision for those of you who have not done any Node since your first-year module. (You will need question 3 for later exercises, so please complete it even if you consider it very easy).

**Questions**

1. On your local machine, create the simple Hello World Express example above. Run it, and request it from your browser to show that it works.
2. Add the /time route from the second example, and again request it from your browser. Refresh your browser to prove that the number of milliseconds increases as time passes.
3. Add an ***artist*** route to your server. The ***artist*** route should take parameter for the artist name and the application should respond to the request with a message displaying the artist the user requested. So, if the user requested:

http://localhost:3000/artist/Oasis

it should send back the message

You are searching for songs by Oasis

1. Make the application also respond to URLs of this form:

http://localhost:3000/artist/(the artist)/song/(the song)

by specifying parameters for both artist and song. So, for example, if the user requested:

http://localhost:3000/artist/Oasis/song/Wonderwall

it would send back:

You are searching for Wonderwall by Oasis

1. You are now going to create a route which ***generates JSON***. In your Express application, at the start of the script (***before*** any routes), create a JavaScript array of objects containing an array of songs. Each song should have fields for title, artist, year and chart position. If you need some inspiration, look at [this Wikipedia page](https://en.wikipedia.org/wiki/List_of_best-selling_singles_in_the_United_Kingdom) which lists the biggest selling UK singles of all time (look at "Best-selling singles based on paid-for purchases" to see the songs). You need only include about 5 songs, but try and include two by the same artist. Then, create a topSongs route, which returns the songs to the client as JSON. Again, test it in your browser.
2. ***More advanced:*** Modify your artist route to filter the array of songs by the artist parameter passed in, and return the filtered results as JSON, so only songs by that artist are returned to the client. To do this efficiently in JavaScript, you can use the filter() method which filters an array based on a certain condition. The example below would specifically find all songs called 'Crazy' (note how we convert the requested title to lower case, so we can handle requests containing the word 'Crazy' in either lower case, upper case or a mixture of both):

const matchingSongs = allSongs.filter ( song => song.title.toLowerCase() == 'crazy' );

1. Add an additional route to find all songs in a given date range:

/from/:year1/to/:year2

The route should filter the array to show only those songs released between the given years, and again return the results as JSON.

**Deploying a Node application**

**We will do this in the lab session, in the final hour.**

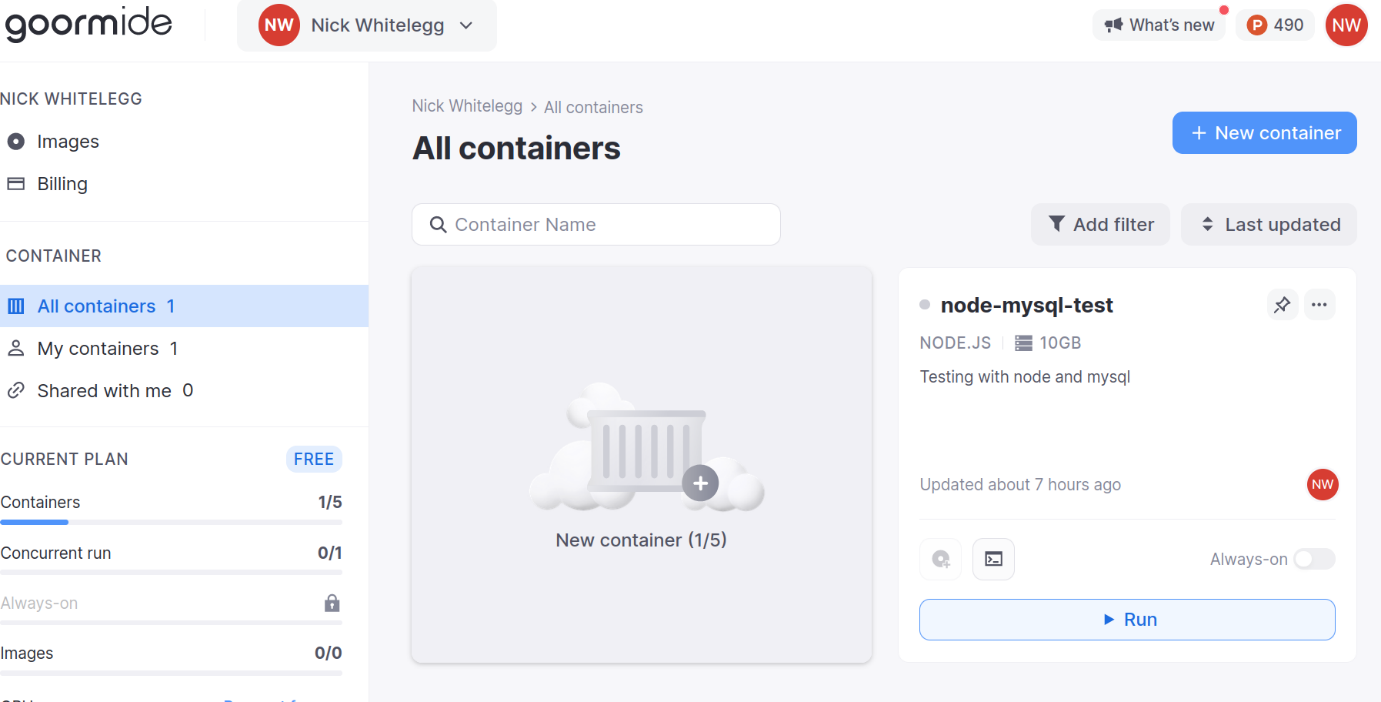
So far we have just run Node servers locally on our own machine. However for a Node application to be accessible to other people, it needs to be deployed on a publicly-accessible ***server environment*** of some kind. There are two main classes of server environment:

* "Classic" web hosting. A classic hosting environment consists of a server machine, which can have software installed on it (such as Node.js or a database) to allow the machine to serve web applications, but has no integrated web-based tools for the end-user to manage the server or the database or write code. You write your code on your local desktop or laptop, and upload it to the server using protocols such as FTP and SSH. Or, alternatively, a developer can log into the server and access a command prompt and work directly on the server machine. In this environment, the developer typically has to manage the server themselves, performing actions such as security updates and server configuration from the server console. The university servers ***Edward2*** and ***Neptune*** are examples of this type of server environment and are available for use should you wish to use them (please ask me for details).
* A ***cloud-based*** environment. A cloud environment is a complete software environment running on the web. A typical cloud server is a server environment which contains not just the storage space and software (such as Node or a database), but also a fully integrated, user-friendly, web-based environment allowing the user to develop code, control databases, install software, manage storage space and RAM, and so on. Many cloud environments can also dynamically increase server capacity (memory, storage space) on demand - though typically at a cost. Unlike "classic" hosting, the user typically does not have to worry about managing and upgrading the server software or database as this is all done automatically as part of the cloud environment.

In this module, both options are available to us. As seen above, there are "classic" servers provided by the university available to you. However, by default, we will use a cloud-based Integrated Development Environment called ***goormIDE***, available at [ide.goorm.io](https://ide.goorm.io/), which contains hosted tools to write code and run and deploy servers. Each server you create is packaged in a ***container*** (a pre-packaged environment with software installed and configured ready to go, capable of being installed on any suitable machine), using [Docker](https://docker.com/), which many of you may have come across.

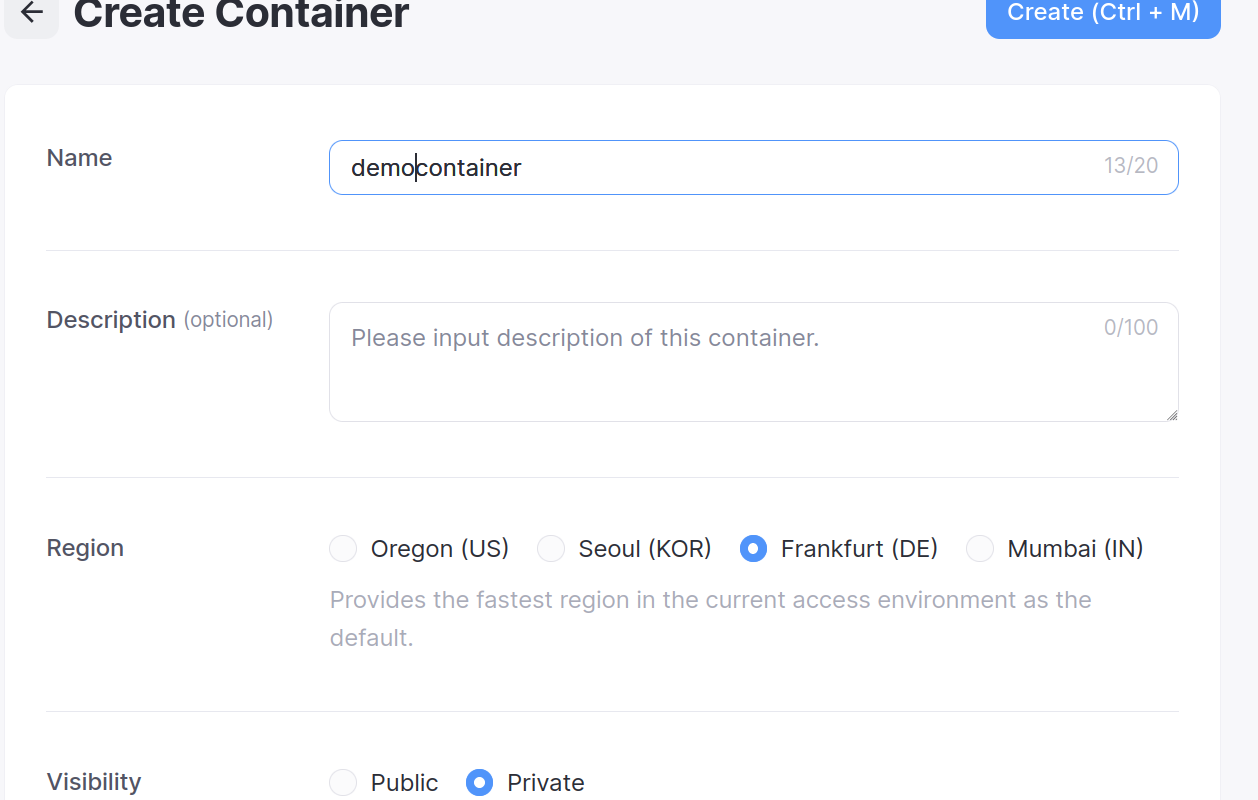
**Using goormIDE**

VIsit goormIDE [here](https://ide.goorm.io/) and log in. You can either signup for an account or use a third-party provider such as Google. Once logged in, you will see a main control panel such as this:

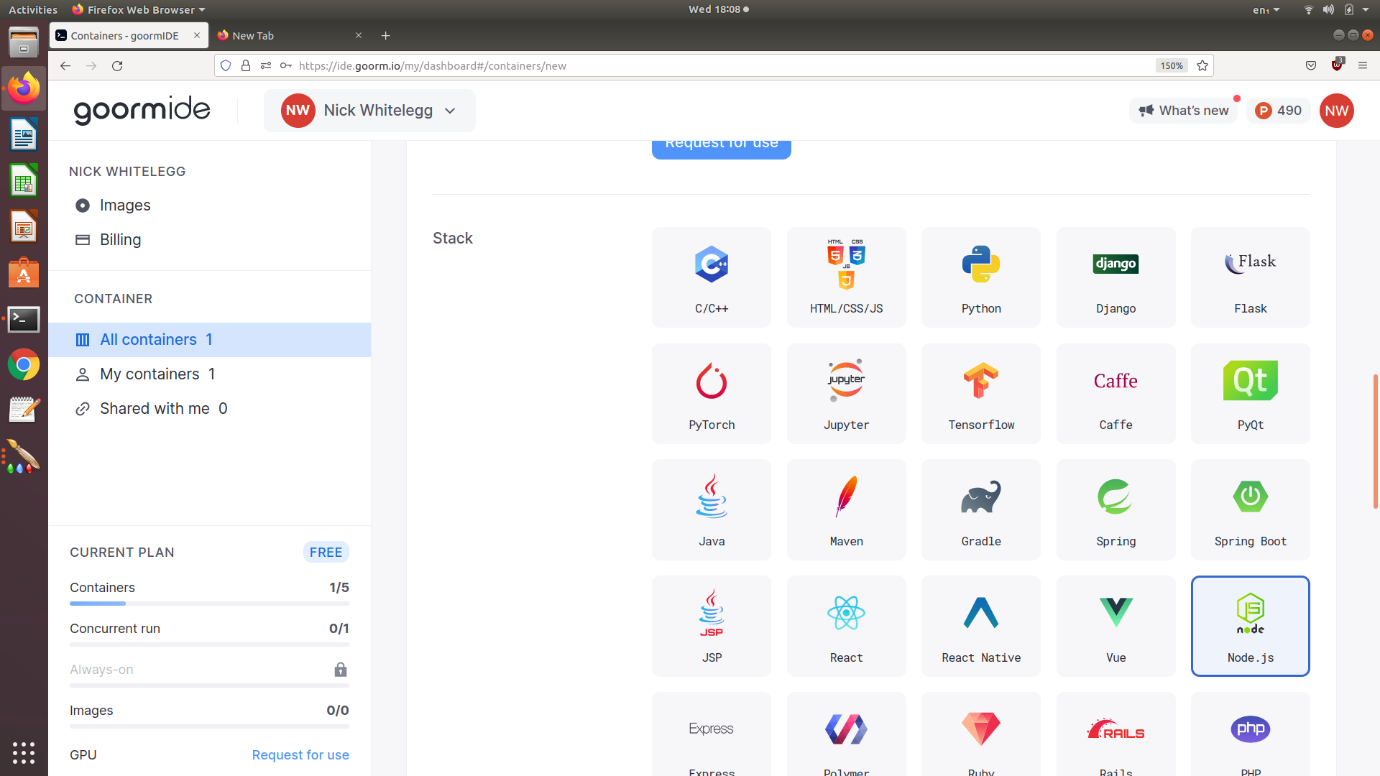


In the control panel you can view all your containers. Each container is an independent virtual server, with its own IP address and domain. In the example above, a container called node-mysql-test has already been created.

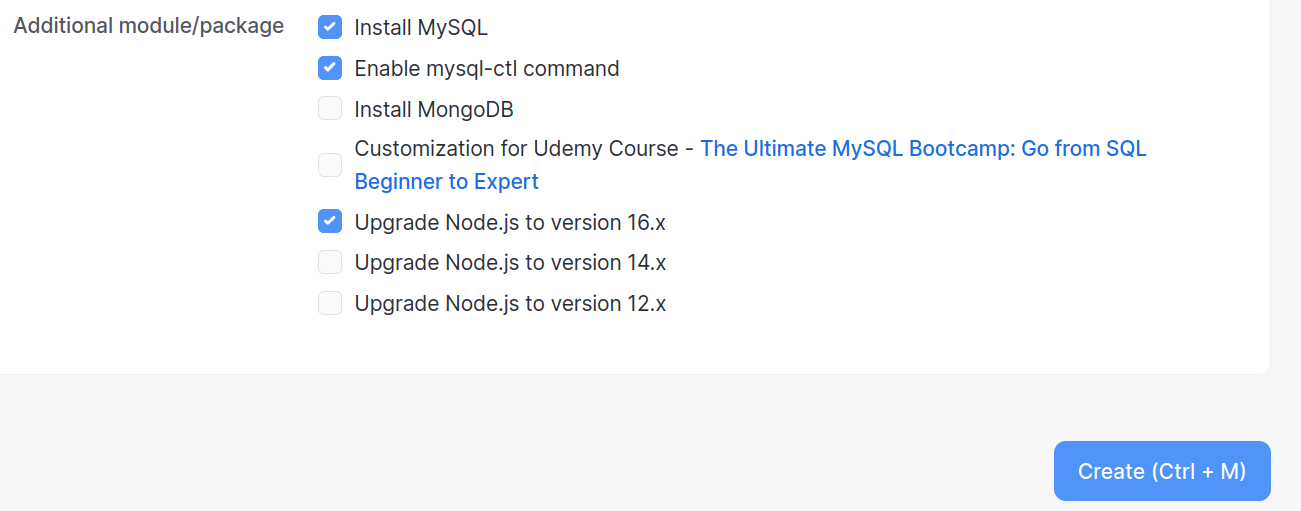
Create a new container with ***New container***. This will bring you to a series of configuration options for the container:



Give your container a name. The other early options can be left as they are, make sure the visibility is ***Private*** so that other people cannot access your container. You will then be asked which development stack to use. Choose Node.js as shown below (do ***not*** choose Express, as that auto-generates quite a bit of code for you and we want to start from scratch):

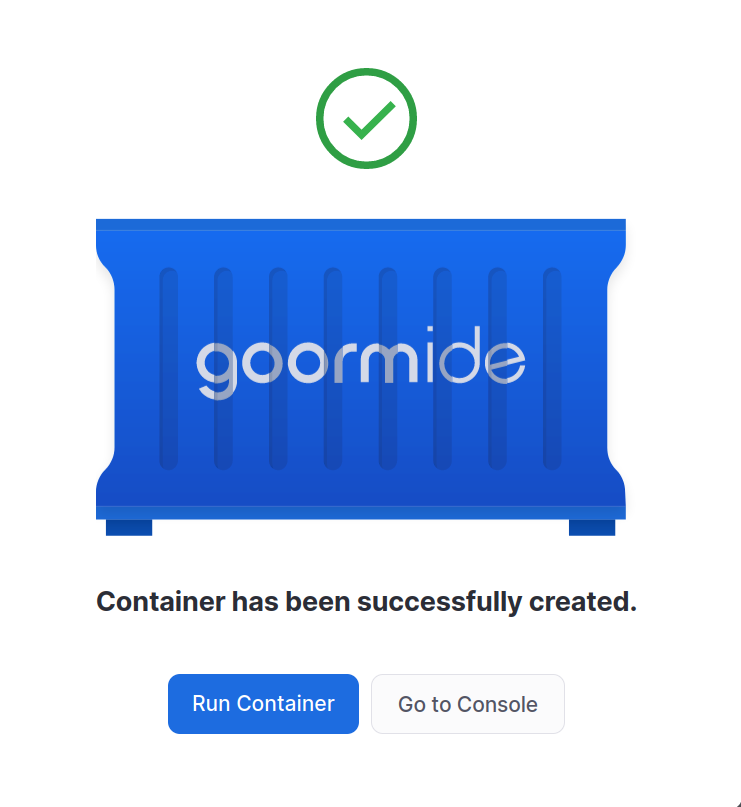


Lower down you can choose software to install to your container, as shown below:

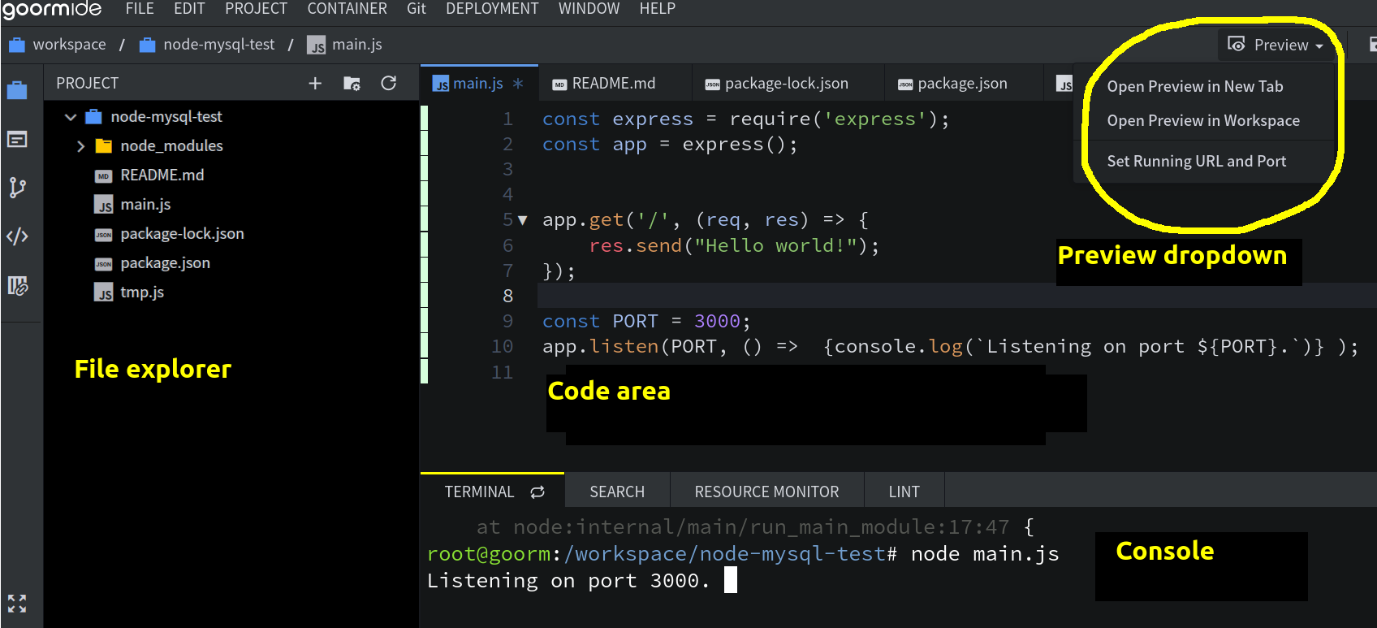


Select the options shown above. Next week you will need MySQL, while mysql-ctl allows you to start up MySQL as a service. Also select "Upgrade Node.js to version 16.x" so that you are working with the latest long-term-support version. Then click ***Create***.

Once created you will see a screen like that below, indicating your container has been created. Select ***Run Container***.



This will run your container and will load up the main IDE workspace screen as seen below.



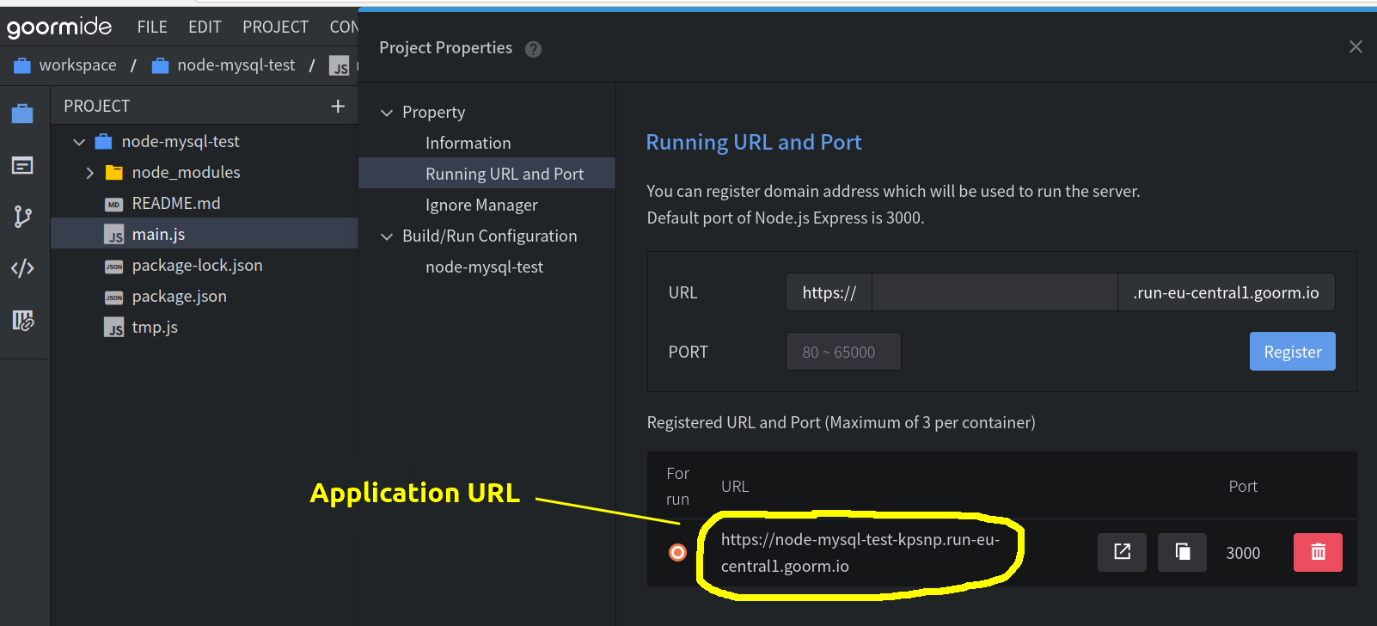
Note the key features which are labelled.

* On the left hand side is a file explorer;
* Occupying most of the screen is the code editor;
* At the bottom of the screen is the console (Linux shell prompt); from here you can run commands such as node and npm (in the example, we have started our server by running the node command) as well as common shell and Linux commands. **You need to install Express and run your Node server from here**, e.g.:
* npm install express

node app.js

* Also note the ***Preview*** dropdown list. As seen, this allows you to view your application running on the web via ***Open Preview in New Tab***. It also allows you to view and edit the application's public URL via ***Set Running URL and Port***.

If you select ***Set Running URL and Port***, you can view your application's public URL and port as shown below.



**Exercise 2**

Run your Node server from Exercise 1 in the goormIDE environment.