

Lab 09: *ExpressJS*

Backend

Web Server II

Get & Post Requests

Express JS

Web Server - (Express)

IMPORTANT NOTE: Install Node JS

Table of Content: *Implementing a Simple (HTTP) Web Server*

# of Parts	Duration	Topic	Page
Introduction	5 minutes	Lab Introduction Define JSON, Express, REST, Heroku	1
Iteration 1	10 minutes	Setup Express App with NPM Initialize an empty Express application from NPM	4
Iteration 2	10 minutes	Server sends a Response Send a 'Hello World' response message from server	5
Iteration 3	10 minutes	Serve a HTML file Serve an index.html file to client	7
Iteration 4	10 minutes	Serve another HTML file Serve more than one html file on a different url route	9
Iteration 5	10 minutes	POST routes Bind an action to a POST request from the submit form	11
Iteration 6	10 minutes	POST Query String Parse the client data from the POST Request	13
Iteration 7	10 minutes	GET for Submission data GET sends submission data from POST	15
Iteration 8	10 minutes	Deploy into Production Deploy app to the internet using Heroku	17
End	5 minutes	Concluding Notes Summary and Submission notes	20

Lab Introduction

Prerequisites

Lab 14. This is the second lab in the Web Server sequence. (This is also backend of Lab 10, 11)

Motivation

Use Express to build a web application & Heroku and git to deploy

Goal

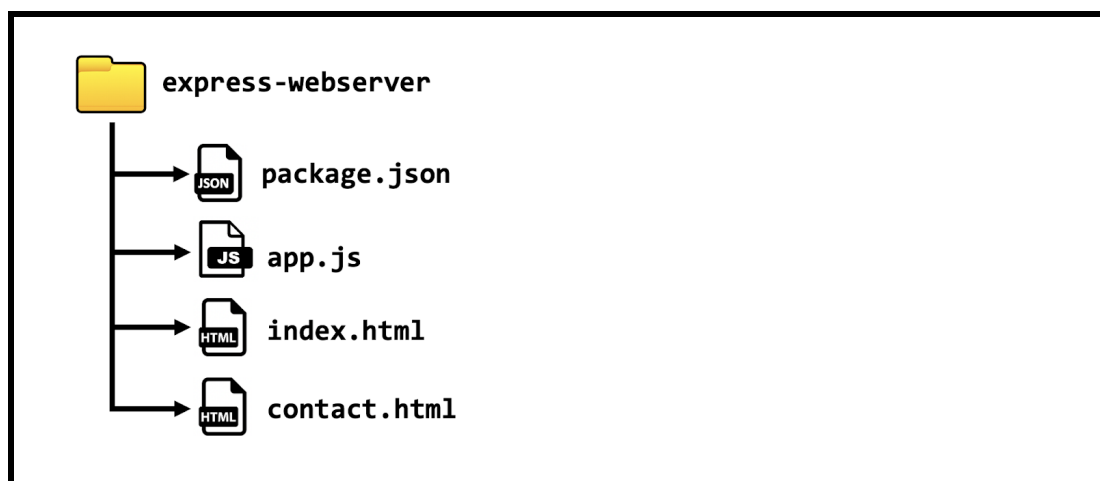
Build an Express app & deploy to the internet

Learning Objectives

- Express
- GET, POST requests
- body-parser & Response body data
- Serve files and JSON data to clients
- Deploy on Heroku

Server-side Architecture:

Start this project by making a project folder where all your assets & scripts will be organized. Create all necessary files and folders as illustrated below.



Concepts

JSON files

JSON (JavaScript Object Notation) is a syntax for serializing objects, arrays, numbers, strings, booleans, and null. It is based upon JavaScript syntax but is distinct from it: some JavaScript is not JSON.

- **package.json:** an app's configuration file. NPM uses it to initialize & launch the app.
<https://docs.npmjs.com/files/package.json>
-

Express JS

Express is a full web framework and server. <http://expressjs.com/>

- **Middleware:** Express can import other modules to make handling requests and responses much easier
-

REST Endpoints

When an API interacts with another system, the touchpoints of this communication are considered endpoints. For APIs, an endpoint can include a URL of a server or service.

- **GET:** Listen for GET request for HTML files or for JSON data
 - **POST:** Listen for POST request submitted from contact form
-

Heroku

A free cloud service that hosts web applications. Deploy any web application to Heroku that requires node to run. Node applications offer more powerful & flexibility than just client-side apps.

<https://www.heroku.com/>

- **Heroku CLI:** The Heroku Command Line Interface (CLI) makes it easy to create and manage your Heroku apps directly from the terminal.
<https://devcenter.heroku.com/categories/command-line>
 - **git:** Heroku uses git to deploy a local project to their remote production servers
<https://devcenter.heroku.com/articles/git>
-

Iteration 1: Setup Express App with NPM

'Approach' → Plan phase

Goal #1: Initialize an empty Express application from NPM

Approach: Use NPM (Node Package Manager) to set up Backend App

NPM uses a configuration file named `package.json` that installs all of an app's dependencies and defines a launch command for starting up your app.

'Apply' → Do phase

Setup Steps

Step 1 (json): Initialize the `package.json` file

The `package.json` file contains all the metadata for managing, building, and launching your node application. JSON (JavaScript Object Notation) is a declarative format to easily specify an app's configurations.

`package.json`

```
{
  "name": "webserver",
  "version": "1.0.0",
  "description": "Web Server",
  "main": "app.js",
  "author": "Your Name",
  "dependencies": {
    "express": "*"
  }
}
```

Step 2 (bash): Use NPM to install dependencies

NPM uses the `package.json` file to fetch and download the app's dependencies with the `install` command.

```
npm install
```

'Assess' → Test phase

All the app's dependencies should be downloaded into a `node_modules` folder.

Iteration 2: Server sends a Response

'Approach' → Plan phase

Goal #2: Send a 'Hello World' response message from server

Approach: Use Express to Listen and Respond to Requests

Express has many 'middleware modules' that make building backend services much easier. You import the modules and use them in your app. This will make it easy to set up a web server.

'Apply' → Do phase

JavaScript Steps

Step 1 (JS): app.js → import modules and setup data

Import the express module and instantiate it into a variable. Setup a port number.

app.js

```
const express = require('express');
const app = express();
const port = 3000;
```

Step 2 (JS): app.js → define GET route for '/'

The express app easily binds HTTP methods (get,post,delete, put) to url routes and callback functions. The callback for the root directory should be to serve the index

app.js

```
app.get('/', serveIndex);
```

Step 3 (JS): app.js → define callback function: serveIndex

The response to a request will be to send a response message of "Hello World"

app.js

```
function serveIndex(request, response){
  response.send('Hello World');
}
```

Step 4 (JS): app.js → listen for request on port

The express app listens to port for requests. Print message to console to show server launched properly.

app.js

```
app.listen(port) ;  
console.log(`Server is running on port ${port}...`);
```

'Assess' → Test phase

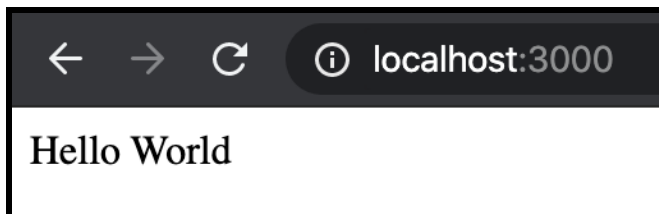
Launch the web server application from the bash terminal with node:

```
node app
```

You should get a display message from the terminal:

```
Server running on port 3000...
```

Open browser at localhost:3000:



You should get a 'Hello World' message

Iteration 3: Serve a HTML file

'Approach' → Plan phase

Goal #3: Serve an index.html file to client

Approach: Use Express to serve files to clients

Express has built-in modules for managing and parsing the content-types (MIME) that make handling requests easy. Web servers are easy to build in express.

'Apply' → Do phase

HTML Steps

Step 1 (HTML): index.html → Make an index.html file

Create an index.html file in the project directory. Same directory as app.js and package.json

index.html

```
<h1>Hello World</h1>
```

JavaScript Steps

Step 1 (JS): app.js → Refactor the serveIndex function

Refactor the callback function for serveIndex such that it send the index.html file

app.js

```
function serveIndex(request, response) {  
  response.sendFile('index.html', {root: dirname});  
}
```


'Assess' → Test phase

Launch the web server application from the bash terminal with command:

```
node app
```

You should get a display message from the terminal:

```
Server running on port 3000...
```

Open browser at localhost:3000:



Iteration 4: Serve another HTML file

'Approach' → Plan phase

Goal #4: Serve more than one html file on a different url route

Approach: Use Express to serve a HTML file on a different route

This second HTML file will contain a contact form that the client will be able to fill out and submit back to the express app.

'Apply' → Do phase

HTML Steps

Step 1 (HTML): contact.html → make a new HTML file

Create a contact.html file in the project directory. Same directory as index.html

contact.html

```
<form method='post' action='contact/send'>
  <input type='text' name='name' placeholder='Enter Name'></input>
  <input type='email' name='email' placeholder='Enter Email'></input>
  <input type='text' name='message' placeholder='Enter Message'></input>
  <button type='submit'>Submit</button>
</form>
```

JavaScript Steps

Step 1 (JS): app.js → define GET for route: '/contact.html'

Use express to bind a GET request to url routes with a callback function

app.js

```
app.get('/contact.html', serveContact);
```

Step 2 (JS): app.js → serveContact (callback function)

The action is to send the contact.html file to client as a response

app.js

```
function serveContact(request, response){  
  response.sendFile('contact.html', {root:   dirname});  
}
```

'Assess' → Test phase

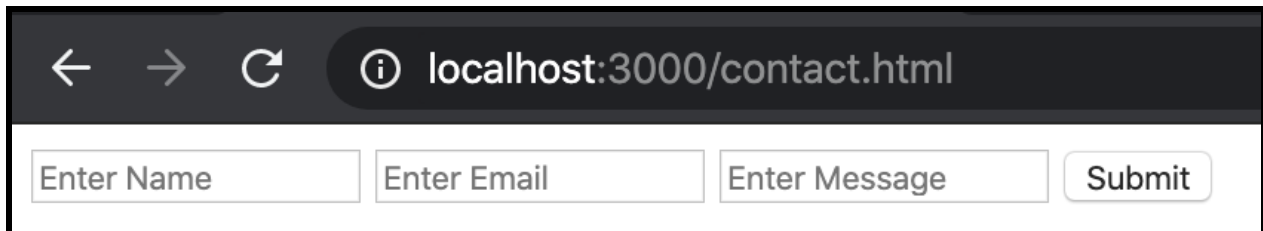
Launch the web server application from the bash terminal with command:

```
node app
```

You should get a display message from the terminal:

```
Server running on port 3000...
```

Open browser at localhost:3000/contact.html:



The screenshot shows a web browser window with the address bar displaying 'localhost:3000/contact.html'. Below the address bar, there is a contact form with three input fields labeled 'Enter Name', 'Enter Email', and 'Enter Message', followed by a 'Submit' button.

Iteration 5: POST routes

'Approach' → Plan phase

Goal #5: Bind an action to a POST request from the submit form

Approach: Use Express to setup a POST route

In this iteration, we will just console log on server that we received a post request from a post route

'Apply' → Do phase

JavaScript Steps

Step 1 (JS): app.js → POST route: 'contact/send'

Bind a POST route to the callback function contactHandler

app.js

```
app.post('/contact/send', contactHandler);
```

Step 2 (JS): app.js → contactHandler (callback function)

Define callback function that prints a message into the server terminal. Then redirect browser to index

app.js

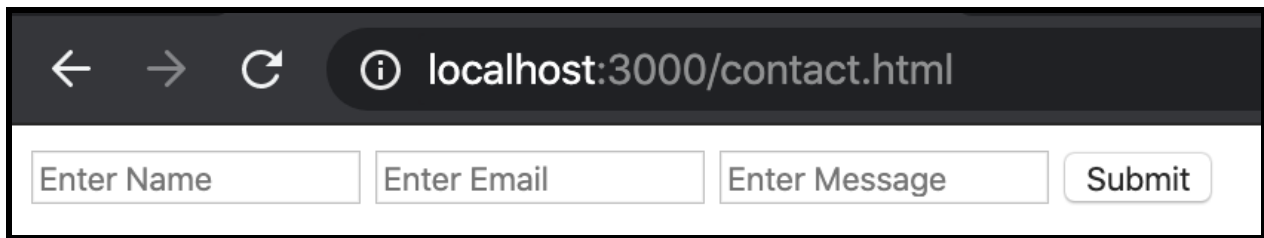
```
function contactHandler(request, response) {  
  console.log('Received POST Request');  
  response.redirect('/');  
}
```

'Assess' → Test phase

Launch the web server application from the bash terminal with command:

```
node app
```

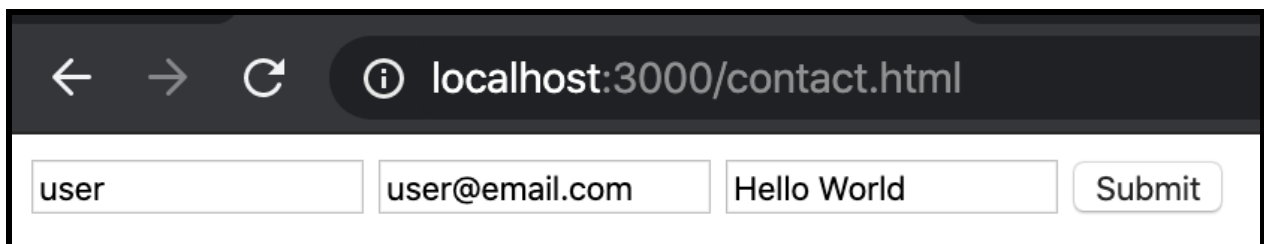
Open browser at localhost:3000/contact.html:



← → ↻ ⓘ localhost:3000/contact.html

Enter Name Enter Email Enter Message Submit

Input data into the fields



← → ↻ ⓘ localhost:3000/contact.html

user user@email.com Hello World Submit

Then click the 'Submit' button

After submitting, you should get a display message from the server terminal:

```
Received POST Request
```

Iteration 6: POST Query String

'Approach' → Plan phase

Goal #6: Parse the client data from the POST Request

Approach: Body-Parser module

The Body-Parser module (middleware) parses HTTP Bodies for their data values.

'Apply' → Do phase

JavaScript Steps

Step 1 (JS): app.js → 'body-parser' module

import the body-parser module to be able to use it in this app. Imports should happen at top of codebase.

app.js

```
const bodyParser = require('body-parser');
```

Step 2 (JS): app.js → express uses body-parser

Express apps use middleware modules such as body-parser to quickly solve problems. body-parser contains methods to convert URL encoded HTTP query string into JSON (JavaScript Objects)

app.js

```
app.use( bodyParser.json() );  
app.use( bodyParser.urlencoded({extended:false}) )
```

Step 3 (JS): app.js → Refactor: contactHandler

Refactor contactHandler to destructure the request object's data and print it. Also

app.js

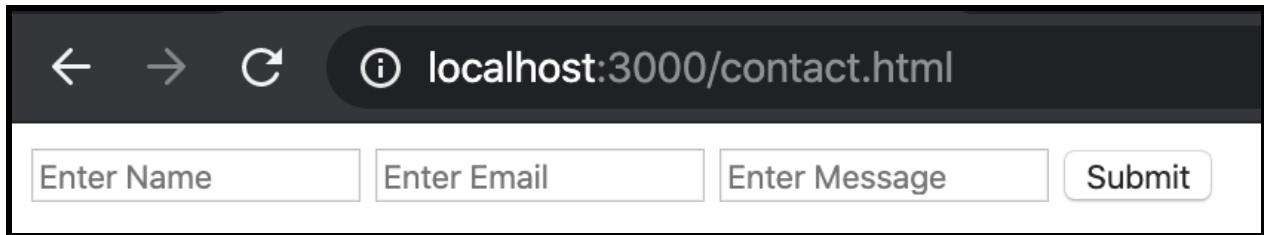
```
function contactHandler(request, response){  
  console.log(request.body);  
  response.redirect('/');  
}
```

'Assess' → Test phase

Launch the web server application from the bash terminal with command:

```
node app
```

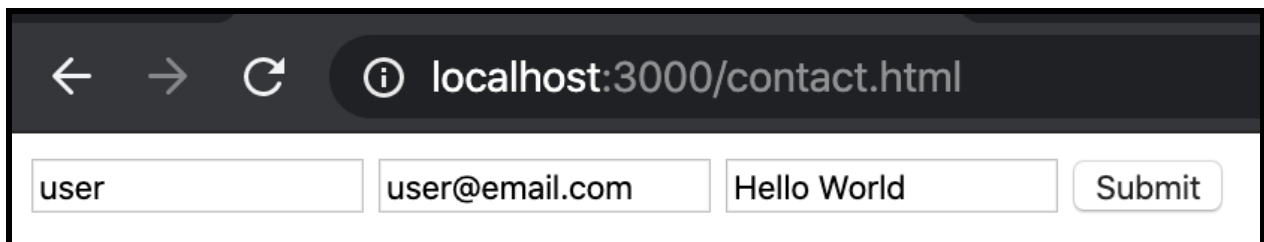
Open browser at localhost:3000/contact.html:



localhost:3000/contact.html

Enter Name Enter Email Enter Message Submit

Input data into the fields



localhost:3000/contact.html

user user@email.com Hello World Submit

Then click the 'Submit' button

After submitting, you should get a display message from the server terminal:

```
user user@email.com Hello World
```

Iteration 7: GET for Submission data

'Approach' → Plan phase

Goal #7: GET route that sends the submission data from the POST

Approach: Express to get data from server

Express provides ability to respond to client by sending json data to a client

'Apply' → Do phase

JavaScript Steps

Step 1 (JS): app.js → submissions list

Initialize an array to hold the submissions. The variable should be initialized near the top of the codebase.

app.js

```
const submissions = [];
```

Step 2 (JS): app.js → Refactor: contactHandler

Refactor contactHandler to push a Request's body into the submissions array.

app.js

```
function contactHandler(request, response){  
  submissions.push(request.body);  
  response.redirect('/')  
}
```

Step 3 (JS): app.js → GET: '/submissions'

Bind a GET to '/submissions' route and a callback function to serveSubmissions..

app.js

```
app.get('/submissions', serveSubmissions);
```


Step 4 (JS): app.js → serveSubmissions callback

serveSubmissions is callback function that sends submissions list to client as a json response

app.js

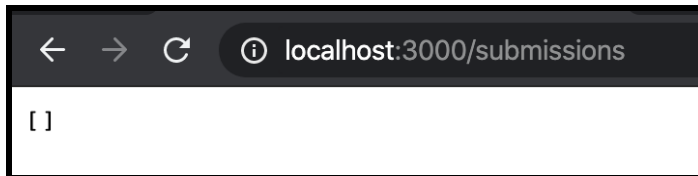
```
function serveSubmissions(request, response){  
  response.json(submissions);  
}
```

'Assess' → Test phase

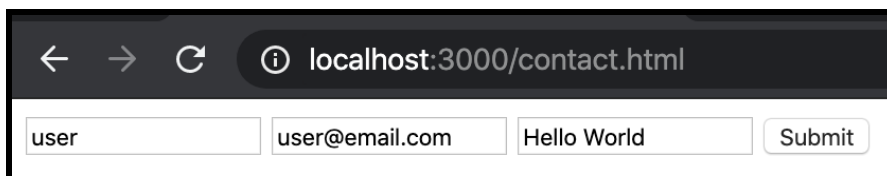
Launch the web server application from the bash terminal with command:

```
node app
```

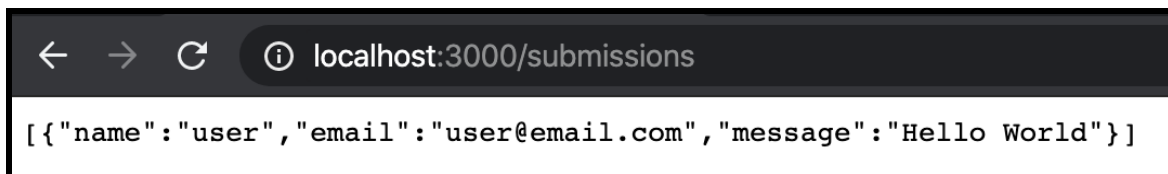
Open browser at localhost:3000/submissions (Note: It should display an empty list.)



Open browser at localhost:3000/contact.html: (Input data into fields & submit)



Open browser at localhost:3000/submissions (Note: It should display the submitted data)



Iteration 8: Deploy into Production

'Approach' → Plan phase

Goal #8: Deploy app to the internet using Heroku

Approach: Heroku

Heroku provides free web app hosting services. Web apps are deployed to Heroku with git.

'Apply' → Do phase

JavaScript Steps

Step 1 (JS): app.js → Refactor: port

For production, assign the port number to Heroku's environmental port, if it exists, otherwise assign it as 3000 for local development.

app.js

```
const port = process.env.PORT || 3000;
```

Step 2 (JS): app.js → import path

Heroku production server will need the 'path' module imported to access html files in local directory.

app.js

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

Step 3 (JSON): package.json → App Start

The package.json is the app's configuration file. Must define to Heroku how to "start" the web application from the command line

package.json

```
{
  "name": "webserver",
  "version": "1.0.0",
  "description": "Web Server",
  "main": "app.js",
  "scripts": {
    "start": "npm install && node app"
  },
  "author": "Your Name",
  "dependencies": {
    "express": "*"
  }
}
```

Terminal Steps

Step 1 (bash): Download Heroku CLI

Instructions: (<https://devcenter.heroku.com/articles/heroku-cli#download-and-install>)

Step 2 (bash): Heroku Login

From terminal, go into project root directory and run bash command to login to heroku

```
heroku login
```

Then enter your Heroku credentials

Step 3 (bash): Create Heroku App & Register it with git

Make a new heroku app from this project directory, by running the bash command

```
heroku create
```

Then initialize the project with git

```
git init
```

Then set heroku as the remote git repo for the app

NOTE: Use name Heroku provided.

```
heroku git:remote -a heroku_app_name
```

NOTE: Don't type in heroku_app_name, it should be a unique name Heroku gives you.

Step 4 (bash): Push to Heroku

Use git to push the app to Heroku.

```
echo node_modules > .gitignore  
git add .  
git commit -m "Version 1.0.0"  
git push heroku master
```

Note: Mustn't push the node_modules folder to the production servers. They must install their own dependencies based on their version of node. Thus .gitignore file is created to prevent it.

'Assess' → Test phase

Access the web application from the browser:

- https://heroku_app_name.herokuapp.com/
- https://heroku_app_name.herokuapp.com/contact.html
- https://heroku_app_name.herokuapp.com/submissions

Conclusions

Final Comments

In this lab you implemented a basic web application in Express that responds to GET & POST requests and serves HTML files & JSON data to the web browser. This lab covered: Express, body-parser, REST endpoints, Heroku deployment

Future Improvements

- Prevent public from accessing the submissions route
- Email the submission to the web master
- Store the submission in a database

Lab Submission

Compress your project folder into a zip file and submit on Moodle.