Τεχνολογία Διαδικτύου
10. Express

Γρηγόρης Τζιάλλας

Καθηγητής

Τμήμα Πληροφορικής και Τηλεπικοινωνιών

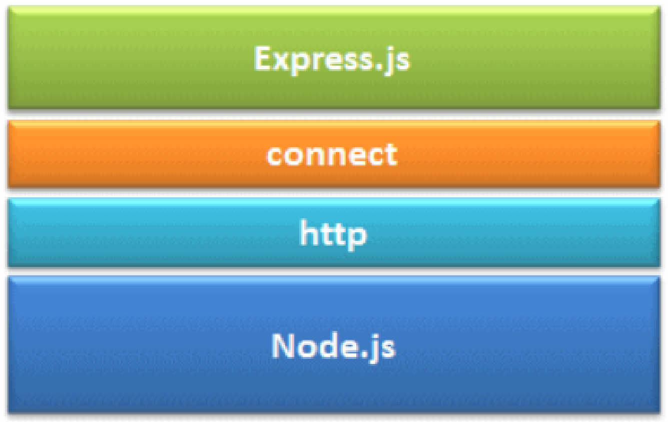
Σχολή Θετικών Επιστημών

Πανεπιστήμιο Θεσσαλίας

Express.js

Express.js

- "Express is a fast, unopinionated minimalist web framework for Node.js" - official web site: http://expressjs.com
- Express.js provides various features that make web application development fast and easy which otherwise takes more time using only Node.js.
- Express.js is based on the Node.js middleware module called connect which in turn uses http module. So, any middleware which is based on connect will also work with Express.js.



Advantages of Express.js

- Makes Node.js web application development fast and easy.
- Easy to configure and customize.
- Allows you to define routes of your application based on HTTP methods and URLs.
- Includes various middleware modules which you can use to perform additional tasks on request and response.
- Easy to integrate with different template engines like Jade, Vash, EJS etc.
- Allows you to define an error handling middleware.
- Easy to serve static files and resources of your application.
- Allows you to create REST API server.
- Easy to connect with databases such as MongoDB, Redis, MySQL

Express.js installation

- You can install express.js using npm.
- The following command will install latest version of express.
    - npm install express

Express Hello World example

```
const express = require('express');
const app = express();
const port = 4000;
app.get('/', function (req, res) {
   res.send('Welcome to Express');
})
app.listen(port, function () {
   console.log("Server listening at " + port)
})
```

Express methods for HTTP verbs

- There is a method for every HTTP verb: get(), post(), put(), delete(), patch():
  - app.get('/', (req, res) => { /* */ })
  - app.post('/', (req, res) => { /* */ })
  - app.put('/', (req, res) => { /* */ })
  - app.delete('/', (req, res) => { /* */ })
  - app.patch('/', (req, res) => { /* */ })
- Those methods accept a callback function - which is called when a request is started - and we need to handle it.
- Express sends us two objects in this callback, which we called req and res, they represent the Request and the Response objects.
- Both are standards. You can read more info at:
  - https://developer.mozilla.org/en-US/docs/Web/API/Request
  - https://developer.mozilla.org/en-US/docs/Web/API/Response

Request parameters

.app	holds a reference to the Express app object
.baseUrl	the base path on which the app responds
.body	contains the data submitted in the request body (must be parsed and populated manually before you can access it)
.cookies	contains the cookies sent by the request (needs the cookie-parser middleware)
.hostname	the hostname as defined in the Host HTTP header value
.ip	the client IP
.method	the HTTP method used
.params	the route named parameters
.path	the URL path
.protocol	the request protocol
.query	an object containing all the query strings used in the request
.secure	true if the request is secure (uses HTTPS)
.signedCookies	contains the signed cookies sent by the request (needs the cookie-parser middleware)
.xhr	true if the request is an XMLHttpRequest

Sending a response to the client

- The send() method of the Response object is used to send a simple string as a response, and to close the connection:

```
(req, res) => res.send('Hello World!')
```

- If you pass in a string, it sets the Content-Type header to text/html.
- If you pass in an object or an array, it sets the application/json Content-Type header, and parses that parameter into JSON.
- After this, send() closes the connection.
- send() automatically sets the Content-Length HTTP response header, unlike end() which requires you to do that.
- Use end() to send an empty response

Sending response status to the client

- Send Status method:

```
res.sendStatus(200)
// === res.status(200).send('OK')

res.sendStatus(403)
// === res.status(403).send('Forbidden')

res.sendStatus(404)
// === res.status(404).send('Not Found')

res.sendStatus(500)
// === res.status(500).send('Internal Server Error')
```

Reading and Setting response headers

- You can access all the HTTP headers using the Request.headers property.
- You can change any HTTP header value using Response.set():

```
res.set('Content-Type', 'text/html')
```

- There is a shortcut for the Content-Type header, however:
  - res.type('.html’) // => 'text/html'
  - res.type('html’)   // => 'text/html'
  - res.type('json’)   // => 'application/json'
  - res.type('application/json’) // => 'application/json'
  - res.type('png’)   // => image/png:

Handling redirects

- Redirects are common in Web Development. You can create a redirect using the Response.redirect() method:

  ```
  res.redirect('/go-there')
  ```

  This creates a 302 redirect.

- A 301 redirect is made in this way:

  ```
  res.redirect(301, '/go-there')
  ```

- You can specify an absolute path (/go-there), an absolute url (https://anothersite.com), a relative path (go-there) or use the .. to go back one level:

  ```
  res.redirect('../go-there')
  res.redirect('..')
  ```

- You can also redirect back to the Referer HTTP header value (defaulting to / if not set) using

  ```
  res.redirect('back')
  ```

301 redirect indicates that a page has permanently moved to a new location, meanwhile, a 302 redirect says that the page has moved to a new location, but that it is only temporary.

Basic routing with Express.js

- Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).
- Using the app object methods get(), post(), put() and delete() we can define routes for HTTP GET, POST, PUT and DELETE requests respectively.

In the example, app.get() and app.post() define routes for HTTP GET and POST.

The first parameter is a path of a route which will start after base URL. The callback function includes request and response object which will be executed on each request.

```
const express = require('express');
const app = express();
const port = 4000;
app.get('/', (req, res) => {
   res.send('Get Received')
 })
app.get("/about", function (req, res) {
  res.send("About this wiki");
});
 app.post('/', (req, res) => {
   res.send('Post Received')
 })
var server = app.listen(port, function () {
 console.log("Server listening at "+  port)
})
```

You can find the example at https://codesandbox.io/p/github/uthcst/express_basic

Named parameters for routing

- If we don't want a parameter to be sent as a query string, but part of the URL, we use named parameters.

```
app.get('/uppercase/:theValue', (req, res) =>
    res.send(req.params.theValue.toUpperCase()))
```

- You can use multiple named parameters in the same URL, and they will all be stored in req.params.

Use a regular expression to match a path

- You can use regular expressions to match multiple paths with one statement:

```
app.get(/post/, (req, res) => { /* */ })
```

will match /post, /post/first, /thepost, /posting/something, and so on.

Handling post requests

- To handle HTTP POST request in Express.js version 4 and above, you need to install middleware module called body-parser.
- This body-parser module parses the JSON, buffer, string and url encoded data submitted using HTTP POST request. Install body-parser using NPM

Example posting and receiving new book data

Index.html css style

```
        form{
            display: grid;
            grid-template-columns: 1fr 4fr;
        }
        form *{
            padding: 3px;
            margin: 3px;;
        }
```

indexl.html for posting data

```
    <h1>Add new book</h1>
    <form action="/addBook" method="post">
        <label for="author">Author</label>
        <input name="author" type="text" />
        <label for="title">Title</label>
        <input name="title" type="text" />
        <label for="year">Year</label>
        <input name="year" type="text" />
        <input type="reset" /><input
type="submit" />
    </form>
```

Express app showing form and receiving post data

```
const express = require('express');
const bodyParser = require("body-parser");
const app = express();
const port = process.env.port || 4000;
app.use(bodyParser.urlencoded({ extended: false }));

app.get('/', function (req, res) {
    res.sendFile(__dirname + '/www/index.html');
});
app.post('/addBook', function (req, res) {
    let book = {
        author: req.body.author,
        title: req.body.title,
        year: req.body.year
    }
    res.send(book.title + ' Submitted Successfully!');
});
const server = app.listen(port, function () {
    console.log('Server is running at ', port);
});
```

You can find the example at https://codesandbox.io/p/github/uthcst/express_form

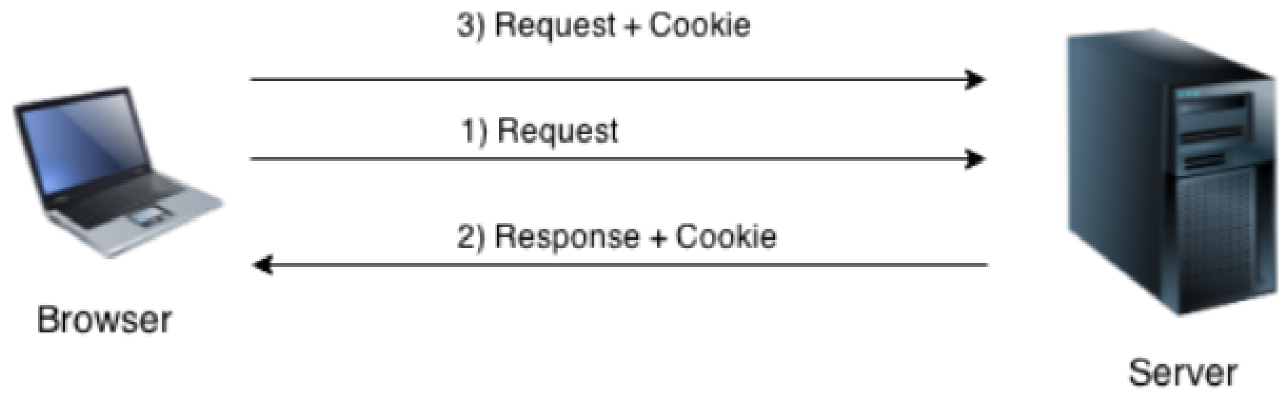
Serving static files

- Using the express.static() method, you can server static resources directly by specifying the folder name of the static resources.

```
const express = require('express');
const app = express();
const port = 4000;
app.use(express.static("www"));
//Serves requests starting with /pictures from another folder
app.use('/pictures', express.static('www/images'));
const server = app.listen(4000, function () {
  console.log("Server listening at port " + port)
})
```

Express.js Cookies Management

- Cookies are small piece of information i.e. sent from a website and stored in user's web browser when user browses that website. Every time the user loads that website back, the browser sends that stored data back to website or server, to recognize user



- A cookie was initially used to store information about the websites that you visit. But with the advances in technology, a cookie can track your web activities and retrieve your content preferences.

Browser cookies

- Cookies are sent on each client request to the server. Every domain name serves as a bucket that stores all the created cookies.
- Web browser cookies can be found under the application storage tab in the browser's DevTools.
- Cookies are set in the browser whenever the set-cookie header is attached to the http headers.
- This set-cookie header has the properties of key-value pairs where the key represents the name, and the value is the cookie data to be set.
- You can set the cookie on the client-side of this domain by running the following code in the console.

```
document.cookie="example_key=example_value"
```

Browser cookies properties

value	In the same instance, it stores the value of the cookie data. This is the main information that we want the browser to remember.
domain	It stores whatever URL the client request was sent to. That is the domain name that fetches the cookie data to the browser
name	This is the name given to the particular cookie saved in the browser. This can be filed using the key in the cookie data
expires	Developers use this option to set the maximum time that the cookie will last in the client browser. This can be set on the server then stored on the client-side when a request is made If missing, or 0, the cookie is a session cookie
httpOnly	Set the cookie to be accessible only by the web server.
maxAge	Set the expiry time relative to the current time, expressed in milliseconds
path	It denotes the specific path or URL that will push the cookie to the browser if requested. Defaults to '/'
secure	Marks the cookie HTTPS only
signed	Set the cookie to be signed
sameSite	Value of SameSite

Setting, getting and clearing cookies example

Requires the cookie-parser module

Setting a cookie to the client browser

Reads the cookie and send it back

Clear the created cookie

Service static files

```
const express = require('express');
const bodyParser = require('body-parser');
const cookieParser = require('cookie-parser');
const port = process.env.port || 4000;
const app = express();
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
//set a cookie
app.get('/setCookie', (req, res) => {
    res.cookie("myCookie", "myCookieValue");
    res.send(`<div>myCookie have been saved successfully</div>
          <div>Check your browser at Developer Tools - Application -
          Storage - Cookies </div>`);
});
//show the saved cookies
app.get('/getCookie', (req, res) => {
    console.log(req.cookies.myCookie);
    res.send(req.cookies.myCookie);
});
// routing for logout
app.get('/clearCookie', (req, res) => {
    res.clearCookie('myCookie');
    res.send('Cookie cleared');
})
app.use(express.static(__dirname + "/www"));
//Serves requests starting with /pictures
app.use('/pictures', express.static(__dirname + '/www/images'));
const server = app.listen(4000, function () {
    console.log("Server listening at port " + port);
});
```

You can find the example at https://codesandbox.io/p/github/uthcst/express_static

Securing cookies

- One precaution that you should always take when setting cookies is security.
- You can access any cookie on a browser console using JavaScript (document.cookie) or using the browser dev tools
- We can add several attributes to make this cookie more secure.
    - HTTPonly ensures that a cookie is not accessible using the JavaScript code. This is the most crucial form of protection against cross-scripting attacks.
    - A secure attribute ensures that the browser will reject cookies unless the connection happens over HTTPS.
    - sameSite attribute improves cookie security and avoids privacy leaks.
- You can also add the maximum time you want a cookie to be available on the user browser. When the set time elapses, the cookie will be automatically deleted from the browser.

Read this guide to learn more attributes and how you can use them in JavaScript and Node.js

Sessions

- By default Express requests are sequential and no request can be linked to each other. There is no way to know if this request comes from a client that already performed a request previously.
- Users cannot be identified unless using some kind of mechanism that makes it possible.
- That’s what sessions are.
- When implemented, every user of your API or website will be assigned a unique session, and this allows you to store the user state.
- The express-session module, which is maintained by the Express team, can be used for implementing sessions.

Express-session options and how to use them

- To set up the session, you need to set a couple of Express-session options, as shown below.

```
const oneDay = 1000 * 60 * 60 * 24;
app.use(sessions({
    secret: "thisismysecrctekeyfhrgfgrfrty84fwir767",
    saveUninitialized:true,
    cookie: { maxAge: oneDay },
    resave: false
}));
```

- secret - a random unique string key used to authenticate a session. It is stored in an environment variable and can't be exposed to the public.
- resave - takes a Boolean value. It enables the session to be stored back to the session store, even if the session was never modified during the request.
- saveUninitialized - this allows any uninitialized session to be sent to the store. When a session is created but not modified, it is referred to as uninitialized.
- cookie: { maxAge: oneDay } - this sets the cookie expiry time. The browser will delete the cookie after the set duration elapses.

Example using sessions

Requires the express-session module

Initializes session parameters

Access session property page_views

Increase page_views

Set initial value of page_views to 1

```
const express = require('express');
const port = process.env.port || 4000;
const session = require('express-session');

const app = express();
app.use(session({
  'secret': '343ji43j4n3jn4jk3n',
  resave: true,
  saveUninitialized: true
}))

app.use('/', express.static('www'));
app.get('/getSession', (req, res) => {
  if (req.session.page_views) {
    req.session.page_views++;
    res.send("You visited this page " +
req.session.page_views + " times");
  } else {
    req.session.page_views = 1;
    res.send("Welcome to this page for the first time!");
  }
});
app.listen(port, function () {
  console.log('Server is running at ', port);
});
```

You can find the example at https://codesandbox.io/p/github/uthcst/express_session

Middleware

- A middleware is a function that hooks into the routing process, performing an arbitrary operation at some point in the chain (depending on what we want it to do).
- It’s commonly used to edit the request or response objects, or terminate the request before it reaches the route handler code.
- Middleware is added to the execution stack like so:

```
app.use((req, res, next) => { /* */ })
```

- This is similar to defining a route, but in addition to the Request and Response objects instances, we also have a reference to the next middleware function, which we assign to the variable next.
- We always call next() at the end of our middleware function, in order to pass the execution to the next handler. That is unless we want to prematurely end the response and send it back to the client.

Example of middleware

- One example is cookie-parser, which is used to parse cookies into the req.cookies object.

  app.use(cookieParser());

- We can also set a middleware function to run for specific routes only (not for all), by using it as the second parameter of the route definition

```
const myMiddleware = (req, res, next) => {
    /* ... */
    next()
}

app.get('/', myMiddleware, (req, res) => res.send('Hello World!'))
```

Templates

- Express is capable of handling server-side template engines.
- Template engines allow us to add data to a view, and generate HTML dynamically.
- There are a number of template engines available today, and the more popular ones include Pug, Handlebars, EJS, Mustache, Swig, and others.
- Integrating a template engine into your Express application only takes a few lines of code.
- Just after assigning the Express function (before creating your routes), add the following app settings:
  - views, the directory where the template files are located e.g.

    app.set('views', './views').
  - view engine, your template engine. For example, to use the Pug template engine:
    app.set('view engine', 'pug')

The pug template engine https://pugjs.org/

- Pug has a very distinct syntax, favoring indentation and spaces over the traditional angle brackets in HTML tags. A typical page with head and body segments looks like this:

There are no opening or closing tags. Instead, the enclosing tag is declared, and its children are indented just below, like in Python.

The content of each tag is declared beside the tag, while attributes are declared inside parentheses. Classes are indicated with . and ids with #

Variables can be defined in two ways:

- Using the #{variable} syntax
- Using the equals (=)

Pug supports two primary methods of iteration:

- each
- while

```
doctype html
html
    head
        meta(name='viewport', content='width=device-width')
        link(rel="stylesheet", href="/css/style.css")
        title = Pug example
    body
        include  _navbar.pug
        div#main
            header
                h1 Pug example
                p  Here is the homepage for #{name}
            section
                h2 #{subject}
                p Lorem ipsum dolor sit, amet consectetur
                       adipisicing elit. Totam, repellendus!
                p
                    a(href=link) DIT
                ul
                each val in [1, 2, 3, 4, 5]
                    li= val
            footer
                h2 Here is the footer
```

You can find the example at https://codesandbox.io/p/github/uthcst/express_pug

Using partials

- Template engines allow us to reuse other templates through partials. Partials are normal templates that may be called directly by other templates.

```
doctype html
html
    head
        meta(name='viewport', content='width=device-width')
        link(rel="stylesheet", href="/css/style.css")
        title = Pug example
    body
        include  _navbar.pug
        div
            header
        ...
```

File _navbar.pug

```
nav.menu
    ul
        li
            a(href='#')
                span Home
        li
            a(href='#')
                span About
        li
            a(href='#')
                span Menu
        li
            a(href='#')
                span Contact
        li
            a(href='#')
                span  Login
```

You can find the example at https://codesandbox.io/p/github/uthcst/express_pug

Sanitizing input

- There’s one thing you quickly learn when you run a public-facing server: never trust the input.
- Even if you sanitize and make sure that people can’t enter weird things using client-side code, you’ll still be subject to people using tools (even just the browser devtools) to POST directly to your endpoints.
- Or bots trying every possible combination of exploit known.
- The express-validator package can be used to perform sanitization https://express-validator.github.io/

Sanitization methods

- trim()
  - trims characters (whitespace by default) at the beginning and at the end of a string
- escape()
  - replaces <, >, &, ', " and / with their corresponding HTML entities
- normalizeEmail()
  - canonicalizes an email address. Accepts several options to lowercase email addresses or subaddresses
- blacklist()
  - remove characters that appear in the blacklist
- whitelist()
  - remove characters that do not appear in the whitelist
- unescape()
  - replaces HTML encoded entities with <, >, &, ', " and /
- ltrim()
  - like trim(), but only trims characters at the start of the string
- rtrim()
  - like trim(), but only trims characters at the end of the string
- stripLow()
  - remove ASCII control characters, which are normally invisible

Validation methods

contains()	isIP()
equals()	isIn()
isAlpha()	isInt()
isAlphanumeric()	isJSON()
isAscii()	isLatLong()
isBase64()	isLength()
isBoolean()	isLowercase()
isCurrency()	isMobilePhone()
isDecimal()	isNumeric()
isEmpty()	isPostalCode()
isFQDN()	isURL()
isFloat()	isUppercase()
isHash()	isWhitelisted()
isHexColor()	

Conversion methods

- toBoolean()
  - convert the input string to a boolean
- toDate()
  - convert the input string to a date, or null if the input is not a date
- toFloat()
  - convert the input string to a float, or NaN if the input is not a float
- toInt()
  - convert the input string to an integer, or NaN if the input is not an integer

Validation and sanitization example

Requires the express-validator module

Check name for length and sanitize

Check email and normalize

Check if numeric and sanitize

Check for errors found and return status

```javascript
const express = require('express');
const bodyParser = require("body-parser");
const { check, validationResult } = require('express-validator');

const app = express();
const port = process.env.port || 4000;
app.use('/', express.static('www'));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(express.json());
app.post('/form', [
  check('name').isLength({ min: 3 }).trim().escape(),
  check('email').isEmail().normalizeEmail(),
  check('age').isNumeric().trim().escape()
], (req, res) => {
  const errors = validationResult(req)
  if (!errors.isEmpty()) {
    return res.status(422).json({ errors: errors.array() })
  }
  const name = req.body.name;
  const email = req.body.email;
  const age = req.body.age;
  res.send(`${name} ${email} ${age} submitted successfully!`)
});
const server = app.listen(port, function () {
  console.log('Server is running at ', port);
});
```

```html
<h1>Sanitization Example</h1>
<form action="/form" method="post">
    <label for="name">Name</label>
    <input name="name" type="text" />
    <label for="email">Email</label>
    <input name="email" type="text" />
    <label for="age">Age</label>
    <input name="age" type="text" />
    <input type="reset" /><input type="submit" />
</form>
```

You can find the example at https://codesandbox.io/p/github/uthcst/express_sanitize

Authentication

Authentication and Authorization

- Authentication is the process of confirming a user's identity by obtaining credentials and using those credentials to validate their identity. If the certificates are valid, the authorization procedure begins.
- Authorization is the process of granting authenticated users access to resources by verifying whether they have system access permissions or not. It also allows you to restrict access privileges by granting or denying specific licenses to authenticated users.
- After the system authenticates your identity, authorization occurs, providing you full access to resources such as information, files, databases, finances, locations, and anything else.

Basic Authentication

- HTTP Basic authentication (BA) implementation is the simplest technique securing web resources because it does not require cookies, session identifiers, or login pages.
- HTTP Basic authentication uses standard fields in the HTTP header.
- The Basic authentication mechanism does not provide confidentiality protection for the transmitted credentials. Therefore, basic authentication is typically used in conjunction with HTTPS to provide confidentiality.

Basic Authentication Protocol

- Server side
  - When the server wants the user agent to authenticate itself towards the server after receiving an unauthenticated request, it must send a response with a HTTP 401 Unauthorized status line and a WWW-Authenticate header field.
- Client side
  - When the user agent wants to send authentication credentials to the server, it may use the Authorization header field.
  - The Authorization header field is constructed as follows:
    - The username and password are combined with a single colon (:). This means that the username itself cannot contain a colon.
    - The resulting string is encoded into an octet sequence. The character set to use for this encoding is by default unspecified, as long as it is compatible with US-ASCII, but the server may suggest use of UTF-8 by sending the charset parameter.
    - The resulting string is encoded using a variant of Base64 (+/ and with padding).
    - The authorization method and a space character (e.g. "Basic ") is then prepended to the encoded string.
  - For example, if the browser uses **Aladdin** as the username and **open sesame** as the password, then the field's value is the Base64 encoding of
    - Aladdin:open sesame, or QWxhZGRpbjpvcGVuIHNlc2FtZQ==.
    - Then the Authorization header field will appear as: Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==

Basic Authentication Example

Define static user names and passwords

Get authorization data

Get part of data after the first space

Get user name and password from buffer

Check user name and password

User is authenticated - continue with request

Send request to browser to authenticate user

Check if user is authenticated and return json data

Read more at https://dev.to/anasaijaz/basic-authentication-vanilla-js-3fc

```
const express = require('express');
const app = express();
const port = process.env.port || 4000;
const users = { 'user1': 'password1', 'user2': 'password2' };
const isAuthenticated = (req, res, next) => {
  const encodedAuth = (req.headers.authorization || '')
    .split(' ')[1] || '';
  const [name, password] = Buffer.from(encodedAuth, 'base64')
    .toString().split(':')
  // Check users credentials and return next if ok
  if (name && password === users[name])
        return next();
  // User is not authenticated give a reponse 401
  res.set('WWW-Authenticate', 'Basic realm="Access to Index"')
  res.status(401).send("Unauthorised access")
}
app.use(express.static(__dirname + "/www"));
app.get('/api/data', isAuthenticated, function (req, res) {
  const data = [{ name: 'Nick', age: 21}, { name: 'Maris', age: 22 }];
  res.status(200).send(JSON.stringify(data));
})
app.listen(4000, function () {
  console.log("Server listening at port " + port)
})
```

You can find the example at https://codesandbox.io/p/github/uthcst/express_basicAuth

Authentication based on cookies

Define static user names and passwords

Check password and authenticate user

Check if user is authenticated

Folder admin is accessible only to authenticated users

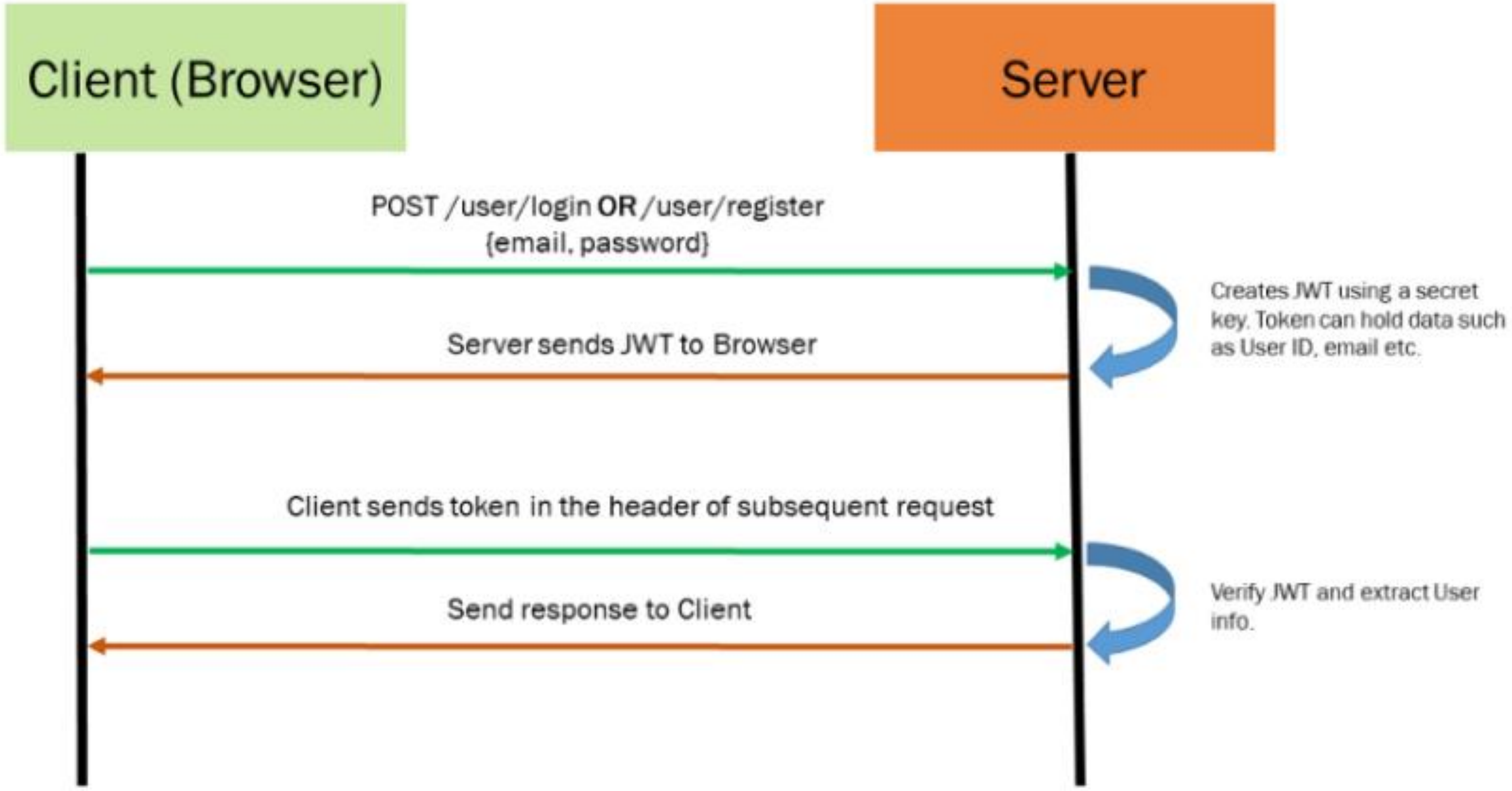
```
const express = require('express')
const bodyParser = require('body-parser');
const cookieParser = require('cookie-parser');
const port = process.env.port || 4000;
const app = express();
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(__dirname + '/www'));
const users = {'user1': 'password1','user2': 'password2'};
app.post('/login', (req, res) => {
    // get username from the client form data
    const username = req.body.username;
    const password = users[username];
    // only if the passwords are equal
    if (password && password === req.body.password) {
        res.cookie('username', username);
        res.redirect("/admin");
    }
    res.send('Failed to login!')
})
app.get('/logout', (req, res) => {
    res.clearCookie('username');
    res.redirect('/’);
});
const isAuthenticated = (req, res, next) => {
    if (!req.cookies.username){
        res.status(401);
        res.send('Access Forbidden');
    }
    next();
}
app.use("/admin",isAuthenticated,express.static(__dirname + '/admin'));
console.log("Server listening at " + port);
app.listen(port);
```

You can find the example at https://codesandbox.io/p/github/uthcst/express_cookiesAuth

JWT based authentication

- JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.
- This information can be verified and trusted because it is digitally signed.
- JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA. -Jwt.io
- A JWT is an encoded string (token) that can be shared between a server and client. The encoded string can hold data inside what is called a payload.
- This information, though protected against tampering, is readable by anyone. Do not put secret information in the payload or header elements of a JWT unless it is encrypted.

How JWT works in securing an application


Client (Browser)
Server
POST /user/login **OR** /user/register
{email, password}
Creates JWT using a secret key. Token can hold data such as User ID, email etc.
Server sends JWT to Browser
Client sends token in the header of subsequent request
Verify JWT and extract User info.
Send response to Client

Server side JWT authentication example 1/2

Require json web token module

Create jwt token and return it to the client

```
const express = require('express');
const bodyParser = require('body-parser');
const jwt = require('jsonwebtoken');
const app = express();
app.use(express.json());
app.use(bodyParser.urlencoded({ extended: false }));
const port = process.env.port || 4000;
const users = { 'user1': 'password1', 'user2': 'password2','1':'1' };

app.use(express.static(__dirname + "/www"));

app.post('/login', (req, res) => {
    // get username from the client form data
    const username = req.body.username;
    const password = users[username];
    if (password === req.body.password) {
        jwt.sign({ username: username }, 'secretkey', (err, token) => {
          res.status(200).json(token);;
        })
    }
    else {
        res.sendStatus(401);
    }
})
```

The full example is at https://codesandbox.io/p/github/uthcst/express_JWTAuth

Server side JWT authentication example 2/2

Read authorization header

Check if header is defined

Split header to remove "Bearer " from the beginning and take the jwt token

Verify jwt token and extract the token data to variable authData

Check user credentials

Definition of middleware function to check if user is authenticated

```
//JWT Authentication
function isAuthenticated(req, res, next) {
    const bearerHeader = req.headers['authorization'];
    if (typeof bearerHeader !== 'undefined') {
        const bearer = bearerHeader.split(' ');
        const bearerToken = bearer[1];
        jwt.verify(bearerToken, 'secretkey', (err, authData) => {
            if (err)
                res.sendStatus(403);
            else if (users[authData?.username])
                next();
            else
                res.sendStatus(403);
        });
    } else {
        res.sendStatus(403);
    }
}
app.get('/api/data', isAuthenticated, function (req, res) {
    var data = [{ name: 'Nick', age: 21 }, { name: 'Maria', age: 22 }];
    res.status(200).send(JSON.stringify(data));
})
app.listen(port, function () {
    console.log("Server listening at port " + port)
});1
```

Client Side JWT authetication example 1/2

Definition of a Variable to store the jwt token

Get values of username and password from the corresponding html input elements

Send Credentials to the server

Store jwt token returned by the server

```
let jwtToken;
function login() {
    jwtToken="";
    let credentials = {
        username: document.querySelector("[name='username']").value,
        password: document.querySelector("[name='password']").value
    }
    fetch("/login", {
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        method: "POST",
        body: JSON.stringify(credentials)
    })
        .then((res) => res.json())
        .then((json) => {
            if (json) {
                jwtToken = json;
                document.querySelector("[name='username']").value = "";
                document.querySelector("[name='password']").value = "";
                document.getElementById("data").innerText = "Logged In";
            }
        })
        .catch(e => showError(e));
}
```

Client Side JWT authetication example 2/2

Send Authorization header with value consisting of “Bearer “ concatenated with the jwt token that has been returned by the server

```
function getData() {
    fetch("/api/data", {
        headers: {
            Authorization: `Bearer ${jwtToken}`
        },
        method: "GET"
    })
    .catch(e => showError(e))
        .then((res) => res.json())
        .then((json) => showData((json)))
        .catch(e => showError(e));
}

function showError(error) {
    document.getElementById("data").innerHTML="Error ";
    console.log(error);
}
function showData(jsonData) {
    document.getElementById("data").innerHTML = arrayToTable(jsonData);
}

function arrayToTable(anArray) {
    //converts anArray of objects to an HTML table
}

document.getElementById("ButtonLogin").addEventListener("click", login);
document.getElementById("ButtonGetData")
                         .addEventListener("click", getData);
```

Web Services

Web services

- A web service is a collection of open protocols and standards used for exchanging data between applications or systems.
- Applications use web services to exchange data over computer networks or the Internet.
- Main Web Services Standards
  - **SOAP (Simple Object Access Protocol)**
    - is a standards-based web services access protocol that has been around for a long time. Originally developed by Microsoft, SOAP isn’t as simple as the acronym would suggest.
  - **REST (Representational State Transfer)**
    - is another standard, made in response to SOAP’s shortcomings. It seeks to fix the problems with SOAP and provide a simpler method of accessing web services.

Principles of REST API

- Statelessness
  - Statelessness enforces servers to remain unaware of the clients' state. The server would not store anything about HTTP requests made by the client or vice versa.
- Cacheable
  - A well-established cache mechanism would drastically reduce the server's average response time by avoiding submitting the same request twice
- Decoupled
  - REST is a distributed approach where client and server applications completely decouple (independent) each other.
- Uniform Interface
  - REST API follows the principles that define a uniform interface and prohibits using self or multiple interfaces within an API. It should also ensure that similar data belongs to only one uniform resource identifier (URI)

A Quick Overview of REST

- The following four HTTP methods are commonly used in REST based architecture.
  - GET − This is used to provide a read only access to a resource.
  - PUT − This is used to create a new resource.
  - DELETE − This is used to remove a resource.
  - POST − This is used to update a existing resource or create a new resource.

CRUD (Create, Read, Update and Delete)

CRUD is an acronym for Create, Read, Update, and Delete commands to interact with databases. HTTP methods of REST Services are used for the CRUD operations:

- Create: (Put HTTP method)
  - This is a procedure that generates new records through an 'INSERT' statement.
- Read: (Put GET method)
  - This is a procedure used to read/retrieve data based on desired input parameters.
- Update: (Post HTTP method)
  - This is a procedure used to modify records without overwriting them.
- Delete: (Delete HTTP method)
  - This is a procedure used to remove one or more entries entirely.

CRUD Rest API using files for storing data

Define a filename for persons json data

Define route for getting all persons

Define route for getting a person by id

Define route for adding a person

Define route for updating a person by id

Define route for deleting a person by id

```
onst express = require('express');
const fs = require("fs");
const bodyParser = require('body-parser');
const app = express();
app.use(express.json());
app.use(bodyParser.urlencoded({ extended: false }));
const port = process.env.port || 4000;

let aFileName = __dirname + '/www/data/persons.json';

app.get('/api/persons', (req, res) => getPersons(req, res));
app.get('/api/persons/:id', (req, res) => getPerson(req, res));
app.post('/api/persons', (req, res) => addPerson(req, res));
app.put('/api/persons/:id', (req, res) => updatePerson(req, res));
app.delete('/api/persons/:id', (req, res) => deletePerson(req,
res));

app.use(express.static(__dirname + "/www"));

app.listen(port, function () {
    console.log("Server listening at port " + port)
})
```

CRUD Rest API using files: get all persons or a person

Set initial return value to an empty array

Set initial return value to an empty array

```
function getPersons(req, res) {
    fs.readFile(aFileName, function (err, data) {
        let persons = [];
        if (!err) persons = JSON.parse(data);
        res.status(200).json(persons);
    });
}

function getPerson(req, res) {
    const id = parseInt(req.params.id)
    fs.readFile(aFileName, function (err, data) {
        let persons = [];
        if (!err) persons = JSON.parse(data);
        res.status(200).json(persons.filter(p => p.id
=== id));
    });
}
```

CRUD Rest API using files: add a Person

```
function addPerson(req, res) {
    const { id, name, phone, age } = req.body;
    const newPerson = {id:parseInt(id), name, phone, age};
    fs.readFile(aFileName, function (err, data) {
        let persons = [];
        if (!err) persons = JSON.parse(data);
        persons.push(newPerson);
        fs.writeFile(aFileName,JSON.stringify(persons),function(err){
                if (err){
                    res.status(200).json(`Error adding id: ${id}`);
                }
                else{
                    res.status(200).json(`Person added with id: ${id}`);
                }
            });
    });
}
```

CRUD Rest API : update a Person

```
function updatePerson(req, res) {
    const { id, name, phone, age } = req.body
    const aPerson = {id:parseInt(id), name, phone, age};
    fs.readFile(aFileName, function (err, data) {
        let persons = [];
        if (!err) persons = JSON.parse(data);
        const anIndex = persons.findIndex(p=>p.id===aPerson.id);
        if (anIndex < 0 ) {
            res.status(200).json(`Cannot find ID: ${id}`);
            return;
        }
        persons[anIndex] = aPerson;
        fs.writeFile(aFileName,JSON.stringify(persons),function(err){
                if (err){
                    res.status(200).json(`Error updating id: ${id}`);
                }
                else{
                    res.status(200).json(`Updated id: ${id}`);
                }
            });
    });
}
```

CRUD Rest API using files: delete a Person

```
function deletePerson(req, res) {
    const id = parseInt(req.body.id)
    fs.readFile(aFileName, function (err, data) {
        let persons = [];
        if (!err) persons = JSON.parse(data);
        const anIndex = persons.findIndex(p=>p.id===id);
        if (anIndex < 0 ) {
            res.status(200).json(`Cannot find ID: ${id}`);
            return;
        }
        persons.splice(anIndex, 1)
        fs.writeFile(aFileName,JSON.stringify(persons),function(err){
                if (err){
                    res.status(200).json(`Error deleting id: ${id}`);
                }
                else{
                    res.status(200).json(`Deleted id: ${id}`);
                }
            });
    });
}
```

CRUD Rest API using files: persons.json

```json
[{"id":1,"name":"nick","phone":"11111","age":20},
{"id":2,"name":"Peter","phone":"22222","age":21},
{"id":3,"name":"Mary","phone":"5555","age":"21"}]
```

CRUD Rest API using files: index.html

```html
    <h1>Persons Form</h1>
    <div id="personsTable"></div>
    <div id="personsForm">
        <label for="id">Id</label>
        <input name="id" type="number" />
        <label for="name">Name</label>
        <input name="name" type="text" />
        <label for="phone">phone</label>
        <input name="phone" type="text" />
        <label for="age">age</label>
        <input name="age" type="text" />
    </div>
    <div id="toolbar">
        <button id="ButtonGetPersons">Get Persons</button>
        <button id="ButtonAddPerson">Add Person</button>
        <button id="ButtonUpdatePerson">Update Person</button>
        <button id="ButtonDeletePerson">Delete Person</button>
    </div>
```

CRUD Rest API using files: client-side code

```javascript
function getPersons() {
    fetch("/api/persons", { method: "GET" })
        .then((res) => res.json())
        .then((json) => showPersons((json)))
        .catch((err) => console.error("error:", err));
}
function getFormData() {
    let aPerson = {
        id: document.querySelector("#personsForm [name='id']").value,
        name: document.querySelector("#personsForm [name='name']").value,
        phone: document.querySelector("#personsForm [name='phone']").value,
        age: document.querySelector("#personsForm [name='age']").value,
    };
    return JSON.stringify(aPerson);
}
function addPerson() {
    let bodyData = getFormData();
    fetch("/api/persons", {
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        method: "POST",
        body: bodyData
    })
        .then((res) => res.json())
        .then((json) => {
            alert(json);
            getPersons();
        })
        .catch((err) => alert("error:", err));
}
```

CRUD Rest API using files: client-side code

```
function updatePerson() {
    let bodyData = getFormData();
    fetch("/api/persons/"+bodyData.id, {
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        method: "PUT",
        body: bodyData
    })
        .then((res) => res.json())
        .then((json) => {
            alert(json);
            getPersons();
        })
        .catch((err) => alert("error:", err));
}

function deletePerson() {
    let bodyData = getFormData();
    fetch("/api/persons/"+bodyData.id, {
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        method: "DELETE",
        body: bodyData
    })
        .then((res) => res.json())
        .then((json) => {
            alert(json);
            getPersons();
        })
        .catch((err) => alert("error:", err));
}
```

CRUD Rest API using files: client-side code

```
function selectPerson(event) {
    if (!event.target.outerHTML.startsWith("<td>")) return;
    let aRow = event.target.parentNode;
    let inputs=document.querySelectorAll("#personsForm input");
    for (let i=0; i < inputs.length; i++){
        inputs[i].value = aRow.children[i].innerText;
    }
    let rows = [...aRow.parentNode.children];
    rows.forEach((r => r.classList.remove("selected")));
    aRow.classList.add("selected");
}
function showPersons(persons) {
    let anHTML = `<table><tr><th>Id</th>
        <th>Name</th><th>Phone</th><th>Age</th></tr>`
    for (let aPerson of persons) {
        anHTML += `<tr>
            <td>${aPerson.id}</td>
            <td>${aPerson.name}</td>
            <td>${aPerson.phone}</td>
            <td>${aPerson.age}</td>
            </tr>`;
    }
    anHTML += "</table>";
    document.getElementById("personsTable").innerHTML = anHTML;
}
getPersons();
document.getElementById("personsTable").addEventListener("click", selectPerson);
document.getElementById("ButtonGetPersons").addEventListener("click", getPersons);
document.getElementById("ButtonAddPerson").addEventListener("click", addPerson);
document.getElementById("ButtonUpdatePerson").addEventListener("click", updatePerson);
document.getElementById("ButtonDeletePerson").addEventListener("click", deletePerson);
```

CRUD Rest API with POSTGRES

- PostgreSQL, commonly referred to as Postgres, is a free, open source relational database management system.
- PostgreSQL is a robust relational database that has been around since 1997 and is available on all major operating systems, Linux, Windows, and macOS. Since PostgreSQL is known for stability, extensibility, and standards compliance, it’s a popular choice for developers and companies.
- node-postgres
    - node-postgres, or pg, is a nonblocking PostgreSQL client for Node.js. Essentially, node-postgres is a collection of Node.js modules for interfacing with a PostgreSQL database.
    - It can be installed with the command: **npm install pg**

Tutorial on installing postgres and setting up a REST api
https://blog.logrocket.com/crud-rest-api-node-js-express-postgresql/

CRUD Rest API with POSTGRES: Connection

Require postgres package

Set credentials to be used for the connection

Values are read from the environment for security reasons.
You can specify absolute values instead

Function to return SQL query results to the browser

```
const express = require('express');
const fs = require("fs");
const bodyParser = require('body-parser');
require('dotenv').config();
const app = express();
app.use(express.json());
app.use(bodyParser.urlencoded({ extended: false }));
const port = process.env.port || 4000;
const { Client, Pool } = require('pg');
const credentials = {
    user: process.env.db_user_,
    host: process.env.db_host_,
    database: process.env.db_database_,
    password: process.env.db_password_,
    port: process.env.db_port_
};
const showResults = (response, error, results) => {
    if (error) {
        console.log(error);
        results.status(200).json([]);
    }
    response.status(200).json(results.rows);
}
```

CRUD Rest API with POSTGRES: Reading data

Connect to database using credentials →

SQL query to read the first 50 records of table persons →

Execute query and show results →

SQL query to select person by id →

Connect to database using credentials →

SQL query to select person by id →

Array with query parameters. It has to be used in order to avoid SQL injection →

Return results →

```
async function getPersons(req, res) {
    const pool = new Pool(credentials);
    const text = `SELECT * FROM persons order by id
limit 50`;
    pool.query(text, (error, results) => {
        showResults(res, error, results);
    });
}
async function getPerson(req, res) {
    const id = parseInt(req.params.id)
    const pool = new Pool(credentials);
    const text = `SELECT * FROM persons WHERE id = $1`;
    const values = [id];
    return pool.query(text, values, (error, results) =>
{
        showResults(res, error, results);
    });
}
```

CRUD Rest API with POSTGRES: add person

Connect to database using credentials

SQL query to insert a person

Array with query parameters

Execute query

Return result

```
async function addPerson(req, res) {
    const { id, name, phone, age } = req.body;
    const pool = new Pool(credentials);
    const text = `
      INSERT INTO persons (id, name, phone, age)
      VALUES ($1, $2, $3, $4)
      RETURNING id`;
    const values = [id, name, phone, age];
    return pool.query(text, values, (error, results) =>
{
        if (error) {
            console.log(error);
        }
        res.status(200).json(`Person added with ID:
              ${results.rows[0].id}`);
    });
}
```

CRUD Rest API with POSTGRES: update person

Connect to database using credentials

SQL query to update a person

Array with query parameters

Execute query

Return result

```
async function updatePerson(req, res) {
    const { id, name, phone, age } = req.body
    const pool = new Pool(credentials);
    const text = `UPDATE persons SET (name, phone, age) =
                 ($2, $3, $4) WHERE id = $1`;
    const values = [id, name, phone, age];
    return pool.query(text, values, (error, results) => {
        if (error) {
            res.status(200).json(error);
            return;
        }
        res.status(200).json(`Person with ID:
             ${id} updated`);
    });
}
```

CRUD Rest API with POSTGRES: delete a person

Connect to database using credentials

SQL query to delete a person

Array with query parameters

Execute query

Return result

```
async function deletePerson(req, res) {
    const id = parseInt(req.body.id)
    const pool = new Pool(credentials);
    const text = `DELETE FROM persons WHERE id = $1`;
    const values = [id];
    return pool.query(text, values, (error, results) => {
        if (error) {
            res.status(200).json(error);
            return;
        }
        res.status(200).json(`Person with ID:
            ${id} deleted`);
    });
}
```

CRUD Rest API with POSTGRES: routing

Route for getting all persons

Route for getting a person by id

Define route for adding a person

Route for updating a person by id

Route for deleting a person by id

```
app.get('/api/persons', (req, res) => getPersons(req, res));
app.get('/api/persons/:id', (req, res) => getPerson(req, res));
app.post('/api/persons', (req, res) => addPerson(req, res));
app.put('/api/persons/:id', (req, res) => updatePerson(req, res));
app.delete('/api/persons/:id', (req, res) => deletePerson(req, res));

app.use(express.static(__dirname + "/www"));

app.listen(port, function () {
    console.log("Server listening at port " + port)
})
```

Αναφορές

Javascript

- https://developer.mozilla.org/en-US/docs/Learn/JavaScript
- https://www.w3schools.com/js/default.asp
- https://www.w3schools.com/js/js_examples.asp
- https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/#basic-javascript

Express

- https://thevalleyofcode.com/express/
- https://www.javatpoint.com/expressjs-tutorial
- https://www.tutorialspoint.com/expressjs/