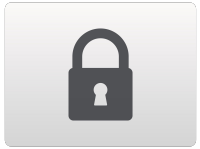
& Server Communication

Mike Rabayda

Overview

1. AJAX JS
    - XMLHttpRequest & Callback functions
    - jQuery AJAX
    - FetchAPI - better way of making async requests
2. RESTful APIs
    - Overview
    - Autocomplete example
3. Password Authentication
    - Basics
    - Bcrypt
    - Example Implementation

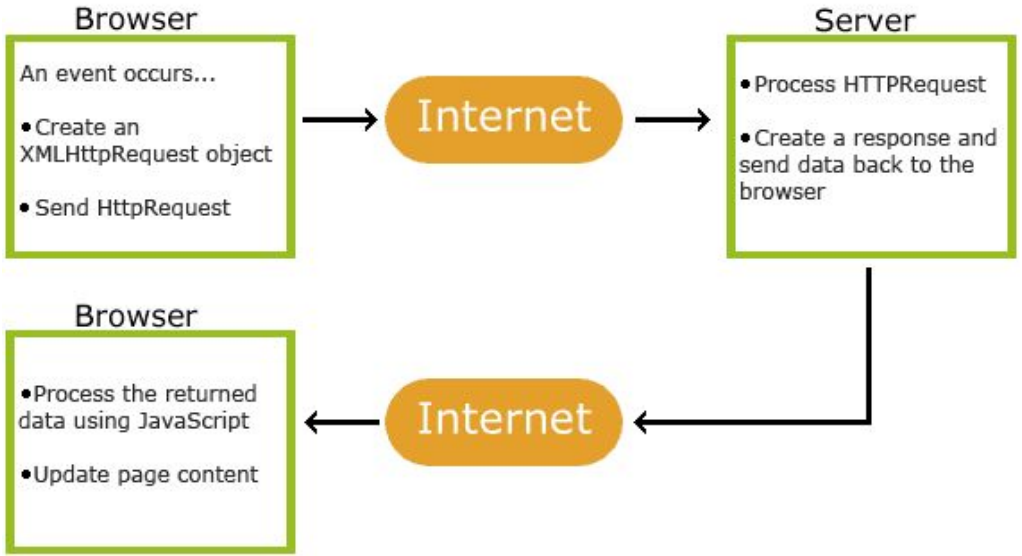


Part 1

What is Ajax?

Asynchronous **J**avascript **A**nd **X**ML

- Allows asynchronous exchange of data between web page and web server
  - Misleading acronym, works with JSON too



Why do we want asynchronous updates?

- So that JS can change website elements in the background without interrupting interaction between user and web browser
- This way JS does not have to wait for server response, it can execute other scripts while waiting

How does AJAX send/receive data to/from server?

1. XmlHttpRequest(XHR) object
2. Define .onload callback function
    - Store and handle the JSON/XML response
3. Send a Request
    - XHR.open(“GET/POST/PUT/DELETE”, “urlOnServer”, async)
    - XHR.send()
        - Empty for GET, xHR.send(“dataToBeSent)” in POST

Responses:

Property	Description
responseText	get the response data as a string
responseXML	get the response data as XML data

Method	Description
getResponseHeader()	Returns specific header information from the server resource
getAllResponseHeaders()	Returns all the header information from the server resource

- XHR.onload = function(){//HERE}

https://www.w3schools.com/js/tryit.asp?filename=tryjs_ajax_xml2

Basic XmlHttpRequest


```
// Create an XMLHttpRequest object
const xhttp = new XMLHttpRequest();

// Define a callback function
xhttp.onload = function() {
  // Here you can use the Data
}

// Send a request
xhttp.open("GET", "ajax_info.txt");
xhttp.send();
```

https://www.w3schools.com/js/tryit.asp?filename=tryjs_ajax_xmlhttp

Requests: GET or POST?

GET

- Simpler, faster than POST, can be used in most cases
- Get uncached result
  - https://www.w3schools.com/js/tryit.asp?filename=tryjs_ajax_get_unique
- Send info with GET
  - Try it Yourself »

POST

- Always use POST when:
  - A cached file is not an option (update a file or database on the server).
  - Sending a large amount of data to the server (POST has no size limitations).
    - Sending user input (which can contain unknown characters), POST is more robust and secure than GET.
- Basic POST
  - Try it Yourself »
- Send an HTML form through POST
  - Try it Yourself »

jQuery & AJAX


```
$(selector).load(URL,data,callback);
```

- loads data from a server and puts the returned data into the selected element.
- data = specifies a set of querystring key/value pairs to send along with the request
- callback = name of a function to be executed after the load() method is completed

https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery_ajax_load

```
$.get(URL,callback);
```

https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery_ajax_get

```
$.post(URL,data,callback);
```

https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery_ajax_post

```
jQuery.ajax(url, [settings])
```

Performs asynchronous HTTP (Ajax) request

Ajax/jQuery Example

HTML Reference

```
</head> <body>
Search for:
<select id="persontype">

<option value="student" selected>Student </option>

<option value="instructor"> Instructor </option> </select> <br>

Name: <input type=text size=20 id="name">
<button onclick="loadTableAsync()"> Show details </button>

<table id="personTable" border="1">

<thead>
<tr> <th>ID</th> <th>Name</th> <th>Dept. Name</th> </tr>

</thead> </table> </body> </html>
```

//HTML for Student or Teacher dropdown options

//defines #name input element, Show Details button to call loadTableAsync() onclick

//HTML for results table once selection is chosen

STOP

Task: Turn to the person next to you and discuss this next slide as to what you think the JS accomplishes with jQuery's form of AJAX

Ajax & jQuery: Autocomplete Example

*/*IMPORT jQuery, jQuery UI, DataTables CSS* - not shown/*

```
/*****************************************************************************************************/

/* This part*/

<script>

var myTable;

$(document).ready(function() {

$("#name").autocomplete({ source: "/autocomplete name" });

myTable = $("#personTable").DataTable({
columns: [{data:"id"}, {data:"name"}, {data:"dept name"}] }); });

function loadTableAsync() {

        var params = {persontype:$("#persontype").val(), name:$("#name").val()};

        var url = "/person query ajax?" + jQuery.param(params);

        myTable.ajax.url(url).load();}

</script>
```

1. jQuery event handler runs after HTML DOM is loaded
2. specifies #name ID as input element for /autocomplete route to fill
3. initializes DataTable object for selection information to be sent back to

1. defines params object to get partial input from #name element, persontype from separate dropdown
2. defines url string to fetch autocomplete suggestions from /person route

Ajax: Autocomplete Example

‘Show Details’ Clicked

Partial Input Request from #name to ‘/person’

Partial Input Request from #name to ‘/autocomplete name’

Search for: Student ▾

Name: Br

Brandt

Brown

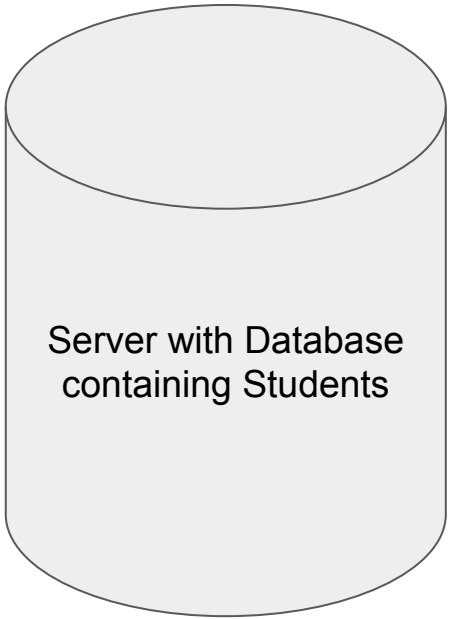
Matched Persons in JSON/XML

Matched Student in JSON/XML

Server with Database containing Students, ‘/autocomplete name’, ‘/person’ route

76543	Brown	Comp. Sci.
77777	Brown	Biology
88888	Brown	Finance

Ajax: Autocomplete Example



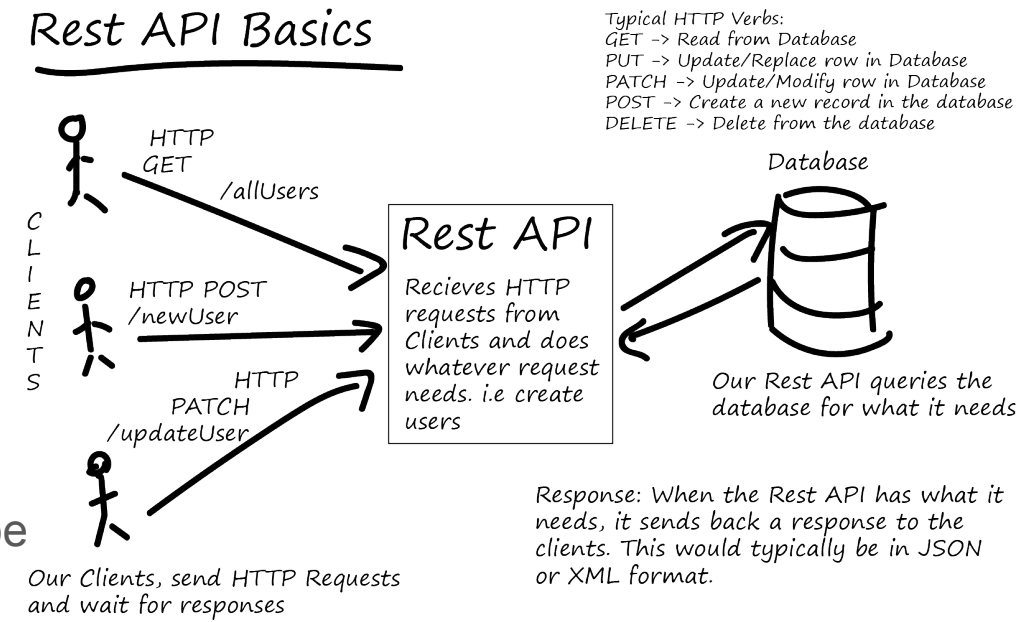
Assumption:

Within this server that contains the names of the Students, some sort of /autocomplete name or /person route returns JSON/XML of names of students/instructors that match letters in the term parameter

Part 2

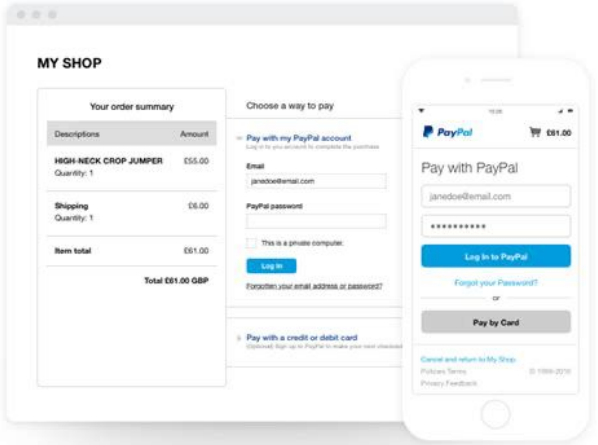
RESTful APIs:

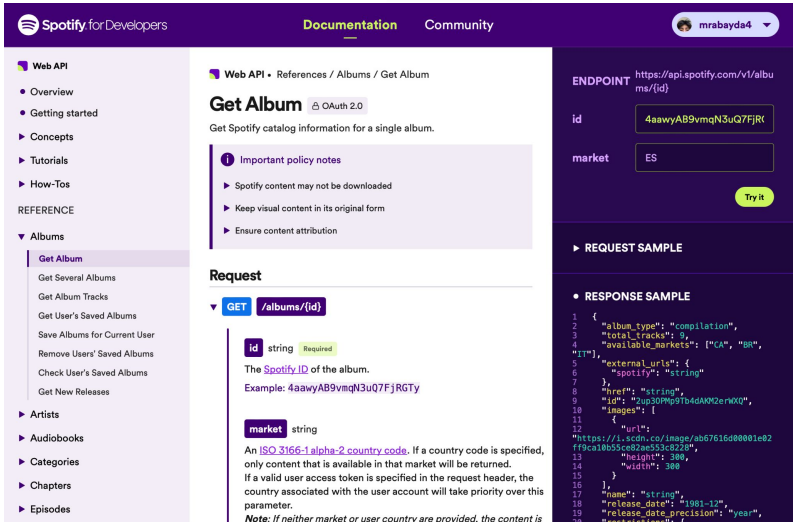
- An interface that 2 computer systems use to exchange information securely over the internet
- Architecture
  - Uniform interface
  - Statelessness
    - Each request is independent and can be executed
  - Layered System
  - Cacheability
  - Code on demand

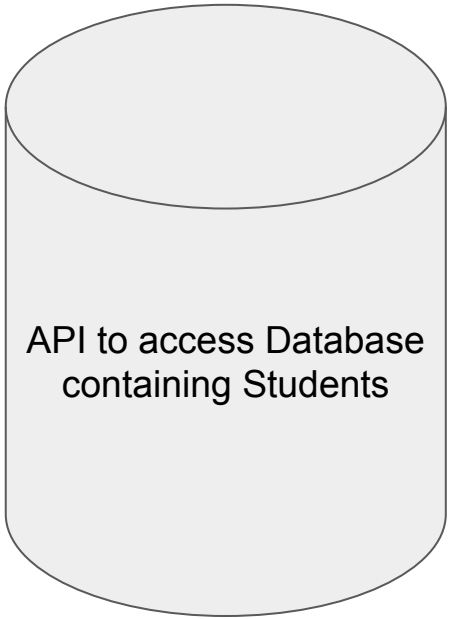


Web Services: SOAP vs. RESTful APIs

- Simple Object Access Protocol (SOAP)
  - Created first by Microsoft
  - Exposes components of application logic as services
  - Only XML
  - Old Paypal API
- Representational State Transfer (REST)
  - Consistent interface to access data in a server
  - JSON/XML
  - Examples
    - SpotifyAPI
    - Duo
    - TwitterAPI

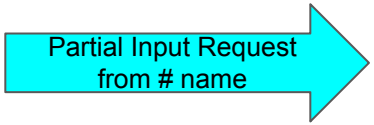


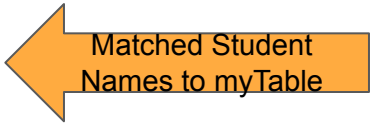





```
const express = require('express');
const app = express();
const PORT = 3000;

const names = ["Alice", "Bob", "Charlie", "David", "Eva", "Frank", "Georgia", "Henry"];

app.get('/autocomplete name', (req, res) => {
    const term = req.query.term; // Retrieve the 'term' query parameter from the request
    if (!term) {
        res.json([]); // If no term is provided, return an empty array
        return;
    }

    // case-insensitive search
    const filteredNames = names.filter(name => name.toLowerCase().includes(term.toLowerCase()));

    res.json(filteredNames); // Return the filtered list of names as a JSON response
});

app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

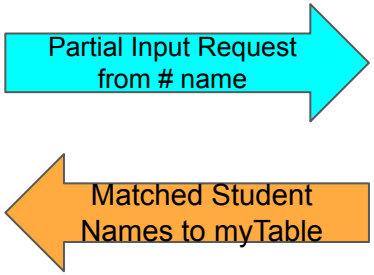
Ajax: Rewriting our Requests with Fetch






```
document.addEventListener("DOMContentLoaded", function() {
    const inputElement = document.getElementById("name"); // Assuming there
exists  an input box with ID 'name'

    inputElement.addEventListener("input", function() {
        const searchTerm = inputElement.value;

        if (searchTerm.length > 1) {
            characters are typed
            fetchNames(searchTerm).then(matchingNames => {
                console.log(matchingNames); // Handle the matching names (e.g.,
display suggestions)

            //Initialize Table - not shown
                // Function to reload the DataTable with new data
            window.loadTableAsync = function() {
                var params = {
                    persontype: $("#persontype").val(),
                    name: $("#name").val()
                };
                var url = "/person query ajax?" + jQuery.param(params);
                myTable.ajax.url(url).load();
```

Ajax: Rewriting our Requests ith Fetch




```
function fetchNames(term) {
    // The URL of the RESTful API endpoint
    const apiUrl = `/autocomplete name?term=${encodeURIComponent(term)}`;

    return fetch(apiUrl)
        .then(response => {
            if (!response.ok) {
                throw new Error('Network response was not ok');
            }
            return response.json(); // Parses the JSON response
        })
        .then(data => {
            // `data` is the JSON array of matching names returned by the server
            return data; // This is the array of names that match the `term`
        })
        .catch(error => console.error('There has been a problem with your fetch operation:', error));
}
```

XMLHttpRequest

- More boilerplate
- Callback Hell

FetchAPI

- Newer standard
- Uses Promises
  - .then() for success scenarios
  - .catch() for errors
- Likely to be better supported
- Supported on all modern browsers
- Requires explicit parsing of JSON/XML

Modern Browsers (Fetch API)

Modern Browsers can use Fetch API instead of the XMLHttpRequest Object.

The Fetch API interface allows web browser to make HTTP requests to web servers.

If you use the XMLHttpRequest Object, Fetch can do the same in a simpler way.

JS & RESTful APIs: FetchAPI

```
async function logMovies() {
  const response = await fetch("http://example.com/movies.json");
  const movies = await response.json();
  console.log(movies);
}
```

- fetch(resource, options)
  - Options in the form of JSON
  - https://developer.mozilla.org/en-US/docs/Web/API/Request

FetchAPI: Example POST request

```
// Example POST method implementation:
async function postData(url = "", data = {}) {
  // Default options are marked with *
  const response = await fetch(url, {
    method: "POST", // *GET, POST, PUT, DELETE, etc.
    mode: "cors", // no-cors, *cors, same-origin
    cache: "no-cache", // *default, no-cache, reload, force-cache, only-if-
cached
    credentials: "same-origin", // include, *same-origin, omit
    headers: {
      "Content-Type": "application/json",
      // 'Content-Type': 'application/x-www-form-urlencoded',
    },
    redirect: "follow", // manual, *follow, error
    referrerPolicy: "no-referrer", // no-referrer, *no-referrer-when-
downgrade, origin, origin-when-cross-origin, same-origin, strict-origin,
strict-origin-when-cross-origin, unsafe-url
    body: JSON.stringify(data), // body data type must match "Content-Type"
header
  });
  return response.json(); // parses JSON response into native JavaScript
objects
}

postData("https://example.com/answer", { answer: 42 }).then((data) => {
  console.log(data); // JSON data parsed by `data.json()` call
});
```

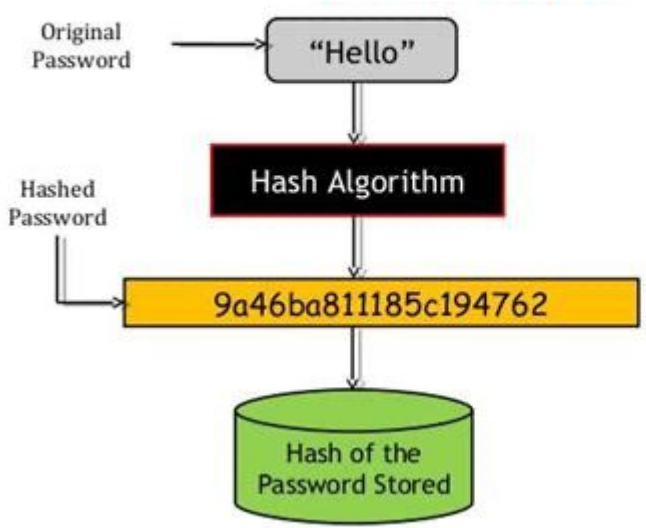
Part 3

express

Authentication

Data Structures Review

- What user info should be stored in a DB
  - Username ✅
  - Password ❌
- NEVER store actual passwords
  - Store hashed and salted passwords instead
  - Salting
    - adds a random value to each password before hashing
    - lets 2 users with same password to have different hashed passwords
    - When users submit password attempts to log in, the server retrieves the salt, appends it to submitted password, hashes and checks the stored password



How can we implement basic user authentication securely in apps?

A: Many different options - JWT, OAuth, But here’s an example of a couple of common ones I found people use online

- Node.js
  - Open source JS runtime environment, allows us to JS without the website
- Express.js
  - makes it easier to organize app functionality with middleware and routing
- Passport.js
  - authentication middleware for Node.js that can be seamlessly integrated with Express.js
- Bcrypt
  - Does the actual password hashing and salting
- Express-sessions
  - Middleware that allows us to persist multiple requests for a user
- Cookie-parser
  - allows us to save an authentication token for logged in users

Bcrypt (in Node.js project), after npm install

async (recommended)

```
const bcrypt = require('bcrypt');
const saltRounds = 10;
const myPlaintextPassword = 's0/\/\P4$$w0rD';
const someOtherPlaintextPassword = 'not_bacon';
```

To hash a password:

Technique 1 (generate a salt and hash on separate function calls):

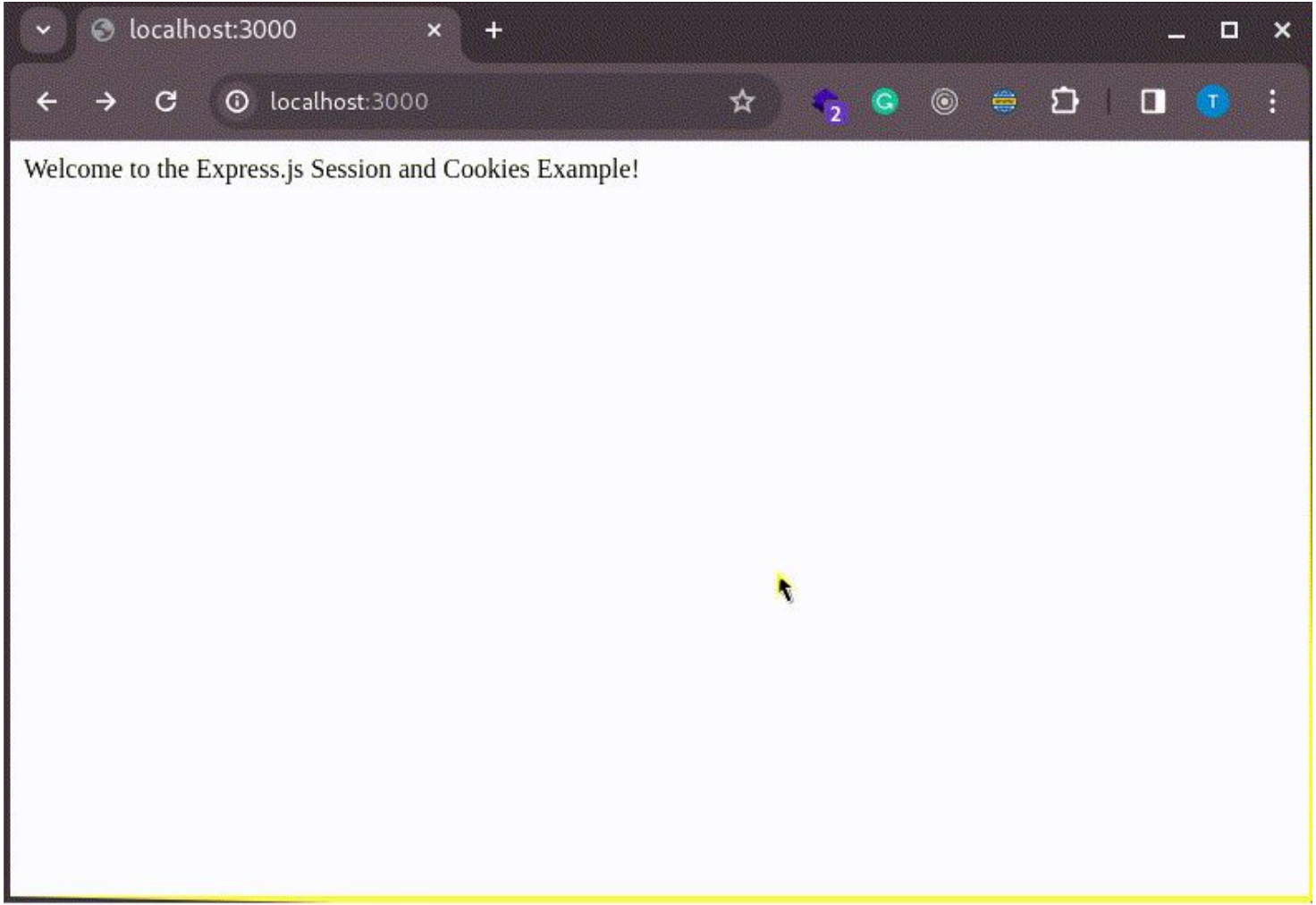
```
bcrypt.genSalt(saltRounds, function(err, salt) {
    bcrypt.hash(myPlaintextPassword, salt, function(err, hash) {
        // Store hash in your password DB.
    });
});
```

Technique 2 (auto-gen a salt and hash):

```
bcrypt.hash(myPlaintextPassword, saltRounds, function(err, hash) {
    // Store hash in your password DB.
});
```

To check a password:

```
// Load hash from your password DB.
bcrypt.compare(myPlaintextPassword, hash, function(err, result) {
    // result == true
});
bcrypt.compare(someOtherPlaintextPassword, hash, function(err, result) {
    // result == false
});
```


localhost:3000
localhost:3000
Welcome to the Express.js Session and Cookies Example!

```
const express = require("express");
const session = require("express-session");
const cookieParser = require("cookie-parser");

const app = express();

// Middleware setup
app.use(
    session({
        secret: "your-secret-key",
        resave: false,
        saveUninitialized: false,
    })
);

app.use(cookieParser());

// Sample user data for demonstration purposes

// Middleware to check if the user is authenticated
const isAuthenticated = (req, res, next) => {
    if (req.session.user) {
        next();
    } else {
        res.redirect("/login");
    }
};
```

```
// Routes
app.get("/", (req, res) => {
    res.send("Welcome to the Express.js Session and Cookies Exampl
});

app.get("/login", (req, res) => {
    res.sendFile(__dirname + "/login.html");
});

app.post("/login", express.urlencoded({ extended: true }), (req, r
    const { username, password } = req.body;

    // Check if the provided credentials are valid
    if (username === "admin" && password === "admin") {
        // Store user data in the session
        req.session.user = username;
        res.cookie("sessionId", req.sessionID);

        res.redirect("/profile");
    } else {
        res.send("Invalid credentials. Please try again.");
    }
});

app.get("/profile", isAuthenticated, (req, res) => {
    // Retrieve user data from the session
    const userData = req.session.user;
    res.send(`Welcome, ${userData.username}!
  <a href="/logout">Logout</a>`);
});

app.get("/logout", (req, res) => {
    // Destroy the session and redirect to the login page
    req.session.destroy(() => {
        res.clearCookie("sessionId");
        res.redirect("/login");
    });
});
```

```
// Start the server
const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

Works Consulted - Sites

- Ajax
  - https://www.w3schools.com/js/js_ajax_intro.asp
    - Other various subpages within w3
  - jQuery/AJAX autocomplete example
    - “Database System Concepts” (423 - 426)
  - Autocomplete picture - ajax-autocomplete-jquery.jpg
  - jQuery/AJAX: https://www.w3schools.com/jquery/jquery_ajax_load.asp
- RESTful APIs
  - https://www.w3schools.in/restful-web-services/intro
  - https://aws.amazon.com/what-is/restful-api/
- REST vs. SOAP
  - https://stackify.com/soap-vs-rest
- Bcrypt
  - https://www.npmjs.com/package/bcrypt
- All other websites linked