# **Javascript: 1**

# **Introduction to the topic:**

## Arrow functions are introduced in ES6. These provide concise and shorter sytanx for writing functions. The major difference from traditional functions is that , arrow function doesn’t bind “this” to the calling object. It instead binds “this” to the enclosing context.

## Split method splits a string into an array of substrings by applying the provided input as separator.

## Slice mthod returns substring of a given string based on input indexes.

## Includes returns whether array includes given object.

## TypeOf operator returns the datatype of the variable.

## Parse is used to form javascript object from JSON string.

## Stringify is used to get JSON representation of the Javascript object.

# **Programming Question:**

## The restaurant manager of a famous restaurant wants to use an application to manage realtime table allocations for the customers. The restaurant is currently providing discount code to all the customers as a limited offer. When a customer visits, the application should allocate table to the user, save the user details and provide discount code to the user. Before allocating, it should be validated that the table is free and not occupied. When the user checks out, the application should set the table free for other customer and remove the user from list. This application only maintains real time users data hence the user should be removed.

use strict';

class Reservation {

constructor(users, tables) {

this.users = users;

this.tables = tables;

/\* Arrow functions - demonsrates the usage of this

Reservation.prototype.reserveTable = (tableNumber, userName) => {

console.log(`Trying to reserve table ${tableNumber} for user: ${userName}`);

if (validateTableNumber(tableNumber)) {

let user = addNewUser(userName, tableNumber);

// Includes is used to see if the new user is added to active users list.

if (this.users.includes(user)) {

// occupy table

let table = this.tables.find(item => item.number === tableNumber);

table.occupied = 1;

                    // Stringify is used to get JSON string from JS object.

let userString = JSON.stringify(user);

console.log(`New user added succesfully: ${userString}`);

console.log(`Provide this code to the user: ${user.code}`);

} else {

console.log(`New user wasn't added\n`);

}

printUserDetails();

}

};

Reservation.prototype.vacateTable = (tableNumber) => {

console.log(`Trying to vacate table ${tableNumber}`);

let table = this.tables.find(item => item.number === tableNumber);

if (table == null || table == undefined) {

console.log('This table is not reserved');

printUserDetails();

return;

} else {

let user = this.users.find(item => (item.tableNumber === tableNumber));

if (user) {

this.users = this.users.filter(e => e !== user)

}

table.occupied = 0;

console.log(`Table ${tableNumber} is now free`);

printUserDetails();

}

};

let printUserDetails = () => {

console.log('Current active users: ');

// Stringify is used to get string from JSON object to send as HTML response.

console.log(JSON.stringify(this.users, null, 1) + '\n\n');

console.log('-----------------------------------------------------------');

};

let addNewUser = (userName, tableNumber) => {

// Split is used to get firstname & lastname

let firstname = userName.split(" ")[0];

let lastname = userName.split(" ")[1];

let id = Math.floor(1000 + Math.random() \* 9000);

// Slice is used to form a discount code to be given to the user

let code = firstname.slice(0, 3) + lastname.slice(0, 3) + id;

// Parse is used to form javascript object from json string.

let user = JSON.parse(`{ "name": "${userName}","id": "${id}","tableNumber":

${tableNumber}, "code" : "${code}"}\n`);

this.users.push(user);

return user;

};

let validateTableNumber = (tableNumber) => {

// Check if tableNumer is a valid number

if (typeof tableNumber != 'number' || tableNumber <= 0 || tableNumber > 5) {

console.error(`Invalid table number: ${tableNumber}\n`);

console.log('-------------------------------------------------------');

return false;

}

// Check if table is available.

let table = this.tables.find(item => item.number === tableNumber);

if (table.occupied != 0) {

console.error(`Table ${tableNumber} is already reserved. Try other table\n`);

printUserDetails();

return false;

}

return true;

};

}

}

let users = [];

let tables = [{

number: 1,

occupied: 0

},

{

number: 2,

occupied: 0

}

]

let reservation = new Reservation(users, tables);;

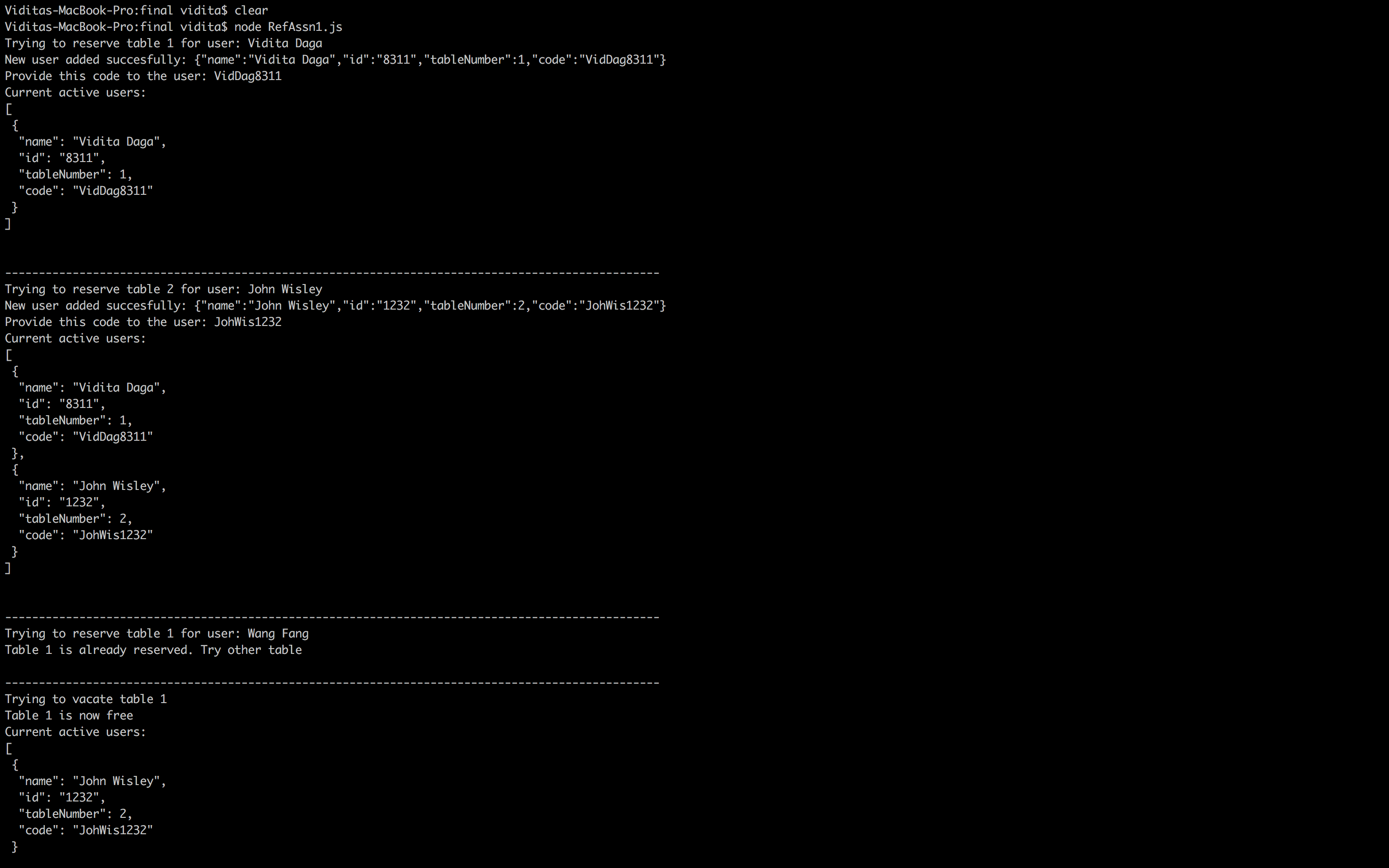
reservation.reserveTable(1, 'Vidita Daga');

reservation.reserveTable(2, 'James Brown');

reservation.reserveTable(1, 'John Wisley');

reservation.vacateTable(4, 'David Miller')

reservation.vacateTable(1);



# **Javascript: 2**

# **Introduction to topic:**

## Destrucuring object enables to retrieve specific attribute from the object. Destructuring array enables to retrieve specific values from array.

## Spread operator can be used to make copy of existing object or to make object with additional properties.

## The rest parameter syntax allows us to represent an indefinite number of arguments as an array.

## Closure provides access to an outer function’s scope from an inner function.

# **Programming Question:**

## A software company provides “Accounting Software” online on lease. As a promotional offer, the company is planning to provide free ugrade to the top 2 users who have used the software for most number of hours. The application should find the top 2 users and send them notification regarding the offer. The top 2 users might or might not change over time. Also, the application should keep a track of how many times this offer is provided.

"use strict";

class CustomerManagement {

constructor(customers) {

this.customers = customers;

CustomerManagement.prototype.findTopUsers = () => {

let noOfTimesOfferProvided = 0;

let notifyCustomers = () => {

let sortedCustomers = this.customers.sort((a, b) => (a.usageHours < b.usageHours) ? 1 : -1);

// Destructure array to get top 2 users from sorted array.

let [first, second] = sortedCustomers;

// Spread operator to add rank

first = { ...first, rank: 1 };

second = { ...second, rank: 2 };

console.log(`Top 2 users:`);

console.log(first);

console.log(second);

// Destructure objects to get only emails.

var { email } = first;

const email1 = email;

var { email } = second;

const email2 = email;

sendMessage(email1, email2);

noOfTimesOfferProvided++;

console.log(`Offer provided ${noOfTimesOfferProvided} times`);

console.log('-------------------------------------------------------\n')

}

// Rest operator

let sendMessage = (...emails) => {

// This method will actually send email using some EMAIL server.

console.log(`Sent email to: `);

console.log(emails);

}

return notifyCustomers;

}

}

}

let customers = [{

id: 654,

name: 'Davis Miller',

usageHours: 189,

email: 'dmiller@xyz.com'

},

{

id: 989,

name: 'James Wisley',

usageHours: 90,

email: 'james@xyz.com'

},

{

id: 100,

name: 'Robert Chris',

usageHours: 100,

email: 'rchris@xyz.com'

},

{

id: 101,

name: 'Wang Fang',

usageHours: 50,

email: 'wang@xyz.com'

},

{

id: 12,

name: 'Monica Sharma',

usageHours: 250,

email: 'msharma@xyz.com'

}];

let custManagement = new CustomerManagement(customers);

//closure is used to keep track of how many times offer is provided

let notifyCustomers = custManagement.findTopUsers();

notifyCustomers();

customers[1].usageHours = 400;

customers[2].usageHours = 500;

notifyCustomers();

customers[4].usageHours = 800;

notifyCustomers();



# **Javascript: 3**

# **Introduction to topic:**

## Export is used to make functions available to external javascript files. Core reusable functionalities are generally exported so that other modules can import and reuse the functionalities.

## Static methods provide class-level functions in javascript.

## Default params have default values for the method parameters.

## Regular expressions are used to match string against patterns.

# **Programming Question:**

## An online Gaming company provides 2 types of games :Type A & Type B. The price of playing any game online depends on the type of game, the duration for which game is played and the traffic demand at that time. In addition 10 % tax is added to the final price. Write functions to calculate product price based on the given inputs. The default duration of playing the game is 24 hrs if not specified. If not specified, there is no high demand multiplying factory. Since the price calculation functionality can be used from different departments like the Sales, Marketing, Billing etc, the functions should be available in form of exported modules.

Example of JS file using the exported module. Exported module defined in next page.

'use strict';

// Method imported from other module.

let calculateProductPrice = require("./RefAssn3\_Exported.js");

let productPrice = calculateProductPrice('mafia game\_a', 10, 48);

console.log(`Price of product : ${productPrice}`);

productPrice = calculateProductPrice('counterstrike game\_b', 10, 48 ,1.5);

console.log(`Price of product : ${productPrice}`);

productPrice = calculateProductPrice('age of empires game\_a', 10);

console.log(`Price of product : ${productPrice}`);

Exported module:

'use strict';

const BASE\_PRICE\_PRODUCT\_GAME\_A = 20;

const BASE\_PRICE\_PRODUCT\_GAME\_B = 50;

const HOURLY\_PRICE = 2;

class ProductManagement {

/\* static method defined to calculate product price

Default parameters are used for highDemandMultiplier and productTime

\*/

static calculateProductPrice (productType, tax, productimeInHours = 24, highDemandMultiplier = 1) {

let basePrice = ProductManagement.getBaseProductPrice(productType);

let price = basePrice;

productimeInHours = productimeInHours - 24;

if (productimeInHours >= 0) {

price = basePrice + (productimeInHours \* HOURLY\_PRICE);

}

price = price \* highDemandMultiplier;

price = price + ( price \* ( tax / 100 ) );

return price;

}

static getBaseProductPrice(productType) {

// Regular expression is used to find the product base price

let found = productType.search(/game\_a/i);

if (found > -1) {

return BASE\_PRICE\_PRODUCT\_GAME\_A;

}

found = productType.search(/game\_b/i);

if (found > -1) {

return BASE\_PRICE\_PRODUCT\_GAME\_B;

}

return -1;

}

}

// The module function is exported to be used by other modules.

module.exports = ProductManagement.calculateProductPrice;

# 

# **Javascript: 4**

# **Introduction to topic:**

## Inheritance can be achieved in javascript by creating classes and their sub-classes.

## The dervied classes can override methods from base class to define runtime behavior that is specific to the derived class.

## Object.assign is used to copy or clone an object.

# **Programming Question:**

## A company owns : ELearning module and Gaming softwares. User can look up product details and request to access any of these 2 products. Design an application for this system and provide functionality to give access to the user to use these products and maintain the active users. Giving access involves multiple steps like unlocking learning material for ELearning module & unlocking hardward needed for games etc. based on the type of product. The methods should show this change in behavior based on the product type.

'use strict';

class Product {

constructor(id, name, basePrice) {

this.id = id;

this.name = name;

this.basePrice = basePrice;

this.users = [];

// Method overriden by subclasses to add specific product details.

Product.prototype.getProductDetails = () => {

return `Product id: ${this.id}, Product Name: ${this.name}, Product Base Price: ${this.basePrice} `;

}

Product.prototype.provideAccessToUser = (userId, userName) => {

let date = new Date();

date.setDate(date.getDate() + 1);

let dateString = date.toString();

// This method is implemented by subclasses.

this.allocateResources(userId);

let newUser = JSON.parse(`{ "name": "${userName}", "id": "${userId}" , "productId": "${this.id}", "expiry": "${dateString}"}`);

this.users.push(newUser);

let usersString = JSON.stringify(this.users, null, 2);

console.log(`Current active users of this product: ${usersString}\n`);

console.log(`\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n`)

}

}

}

// Subclass for Game products

class Game extends Product {

constructor(id, name, basePrice, type) {

super(id, name, basePrice);

this.type = type;

// Overriden

Game.prototype.getProductDetails = () => {

console.log(super.getProductDetails() + `Game type: ${this.type}`);

}

// Method called in base class but implemented in subclass.

Game.prototype.allocateResources = (userId) => {

unlockGame(userId);

unlockGameResources(userId);

console.log('\n');

}

let unlockGame = (userId) => {

console.log(`Game ${this.name} is unlocked for user: ${userId}`);

}

let unlockGameResources = (userId) => {

console.log(`Game ${this.name} resources are now available to the user: ${userId}`);

}

}

}

// Subclass for ELearning products

class ELearning extends Product {

constructor(id, name, basePrice, category) {

super(id, name, basePrice);

this.category = category;

//Overriden

ELearning.prototype.getProductDetails = () => {

console.log(super.getProductDetails() + `Category: ${this.category}`);

}

// Method called in base class but implemented in this class.

ELearning.prototype.allocateResources = (userId) => {

unlockMaterial(userId);

unlockDiscussionForum(userId);

console.log('\n');

}

let unlockMaterial = (userId) => {

console.log(`ELearning product material ${this.name} is unlocked for user: ${userId}`);

}

let unlockDiscussionForum = (userId) => {

console.log(`User ${userId} can participate in discussions for ELearning product: ${this.name}`);

}

}

}

let game1 = new Game('1234', 'Counter Strike', 10, 'MultiPlayer');

game1.getProductDetails();

game1.provideAccessToUser('5678', 'Vidita Daga');

game1.getProductDetails();

game1.provideAccessToUser('9999', 'Huang Wang');

//Assign is used to copy the game1 object and change specific attributes only.

let game2 = Object.assign(game1);

game2.id = '9876';

game2.type = 'With computer';

game2.users = [];

game2.getProductDetails();

game2.provideAccessToUser('5678', 'Vidita Daga');

let eLearning = new ELearning('7878', 'English Grammar', 30, 'Language');

eLearning.getProductDetails();

eLearning.provideAccessToUser('9786', 'James Roy');



# **Javascript: 5**

# **Introduction to topic:**

## fetch API provides interface to retrieve data. It accepts argument which is the API or path to fetch the resource . It returns a Promise back which has the response if the fetch operation is succesfull and error otehrwise.

# **Programming Question:**

## Provide a function to display the last closing stock price of given companies. Use the IEX Trading Open Source API for fetching data.

let fetch = require("node-fetch");

// Open source API to get last closing stock price of given companies

fetch('https://api.iextrading.com/1.0/tops/last?symbols=GLUU,HUYA,ZNGA')

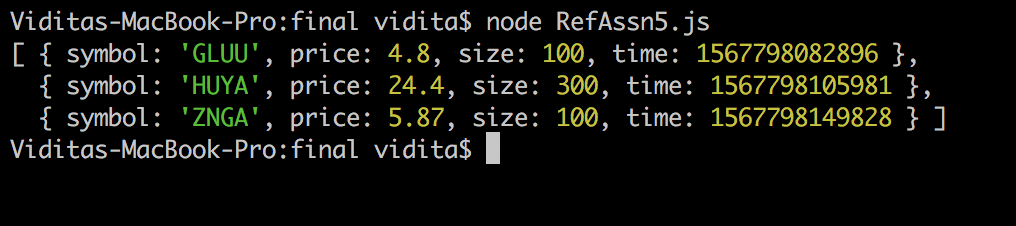
.then(response => response.json())

.then(data => {

console.log(data)

})

.catch(error => console.error(error))



# **Javascript: 6 – Part 1 ( Difference between call, bind and apply )**

# **Introduction to topic:**

## Call is used to call a function and pass the object to which “this” refers to.

## Apply is same as Call except that we can pass an array of arguments instead of individual arguments.

## Bind returns back a function in which “this” refers to the context that is passed in.

# **Programming Question:**

## A team is following agile methodology to deliver the release of a product. User stories are defined and are available in product backlog. Create an application to assign stories to the employees. One employee can work on one or more user stories. Also in some cases it is possible that multiple employees are working on one user story.

'use strict';

function assignUserStory(story1, story2, story3) {

if (undefined != story1) {

this.stories.push(story1);

console.log(`Assigned story: ${story1} to employee: ${this.name}\n`);

}

if (undefined != story2) {

this.stories.push(story2);

console.log(`Assigned story: ${story2} to employee: ${this.name}\n`);

}

if (undefined != story3) {

this.stories.push(story3);

console.log(`Assigned story: ${story3} to employee: ${this.name}\n`);

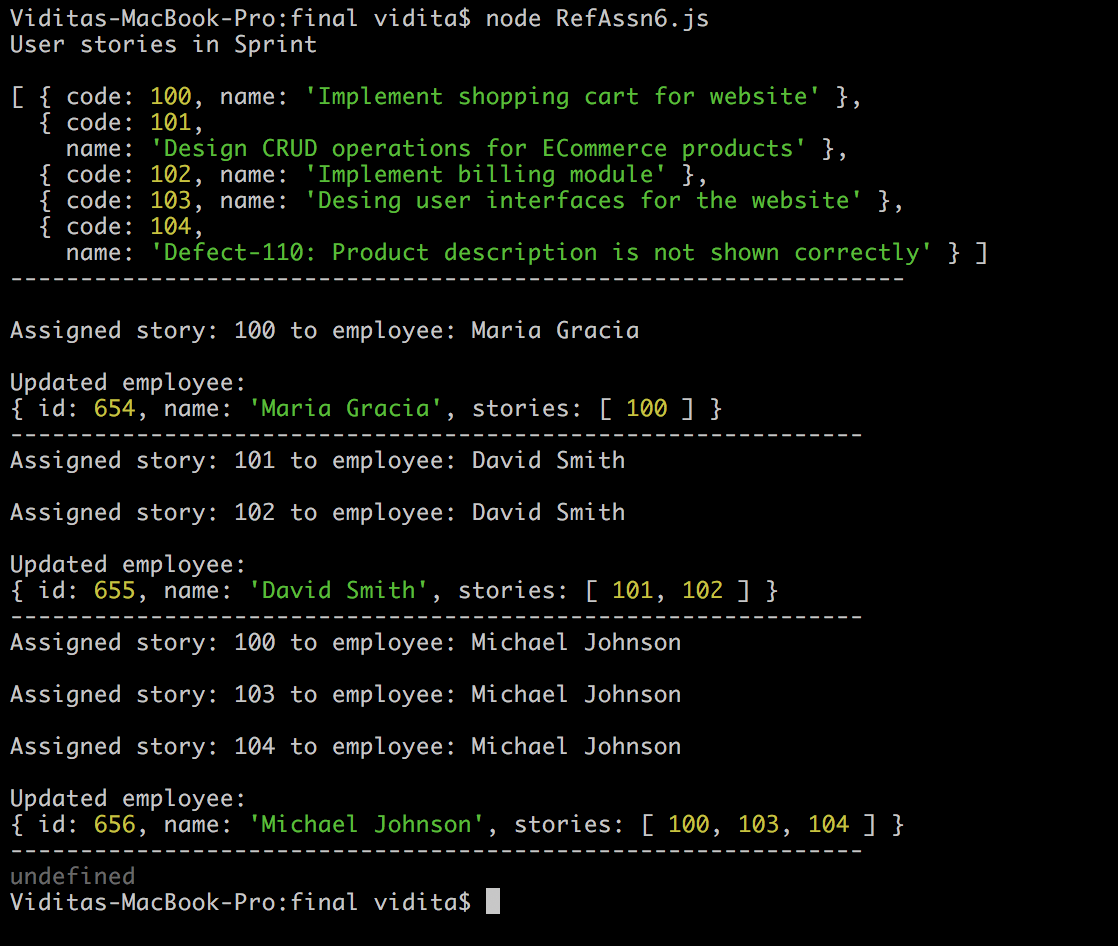
}

console.log(`Updated employee: `);

console.log(this);

console.log('-------------------------------------------------------------');

}



let employees = [

{

id: 654,

name: 'Maria Gracia',

stories: []

},

{

id: 655,

name: 'David Smith',

stories: []

},

{

id: 656,

name: 'Michael Johnson',

stories: []

}

];

let userstories = [

{

code: 100,

name: 'Implement shopping cart for website'

},

{

code: 101,

name: 'Design CRUD operations for ECommerce products'

},

{

code: 102,

name: 'Implement billing module',

},

{

code: 103,

name: 'Desing user interfaces for the website'

},

{

code: 104,

name: 'Defect-110: Product description is not shown correctly'

}];

console.log('User stories in Sprint\n');

console.log(userstories);

console.log('----------------------------------------------------------------\n');

assignUserStory.call(employees[0], 100);

assignUserStory.apply(employees[1], [101, 102]);

let bound = assignUserStory.bind(employees[2]);

console.log(bound(100, 103, 104));

# **Javascript: 6 – Part 2 ( Difference between var, let & const )**

# **Introduction to topic:**

## let is introduced in ES6 and is scoped to the block in which it is defined. var is scoped to the function in which it is defined. Also let is not attached to the global window object while var is attached to global window object. let overcomes problems that can arise due to using var.

## const is used to define constant values and cannot be changed once defined.

# **Programming Question:**

## Calculate Simple Interest and show difference between var, let & const.

//const

const roi = 10;

// Var can be used out of block in which it is defined

let calculateSimpleInterest = (principal, period) => {

var interest = [];

for (var i = 0; i < principal.length; i++) {

var intr = (principal[i] \* period \* roi) / 100;

interest.push(intr);

}

console.log(intr);

};

let principal = [200, 300, 400];

calculateSimpleInterest(principal, 2);

// let is scoped to block hence this code throws error

let calculateSimpleInterest2 = (principal, period) => {

let interest = [];

for (let i = 0; i < principal.length; i++) {

let intr = (principal[i] \* period \* roi) / 100;

interest.push(intr);

}

console.log(intr);

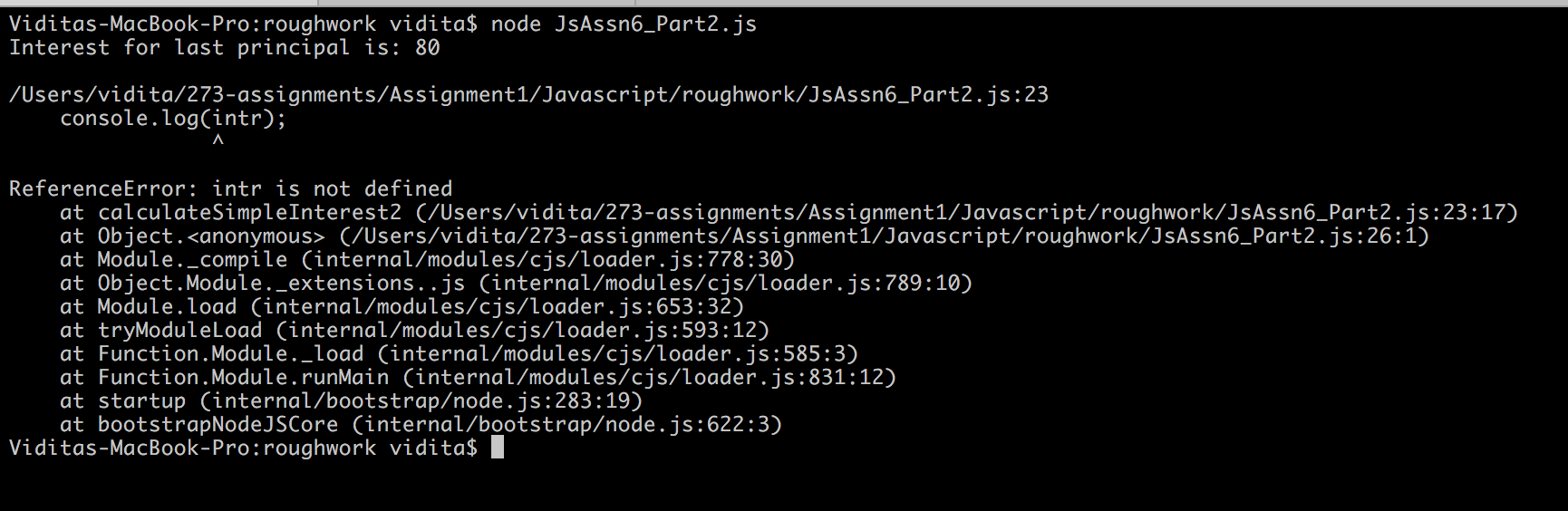
console.log(interest);

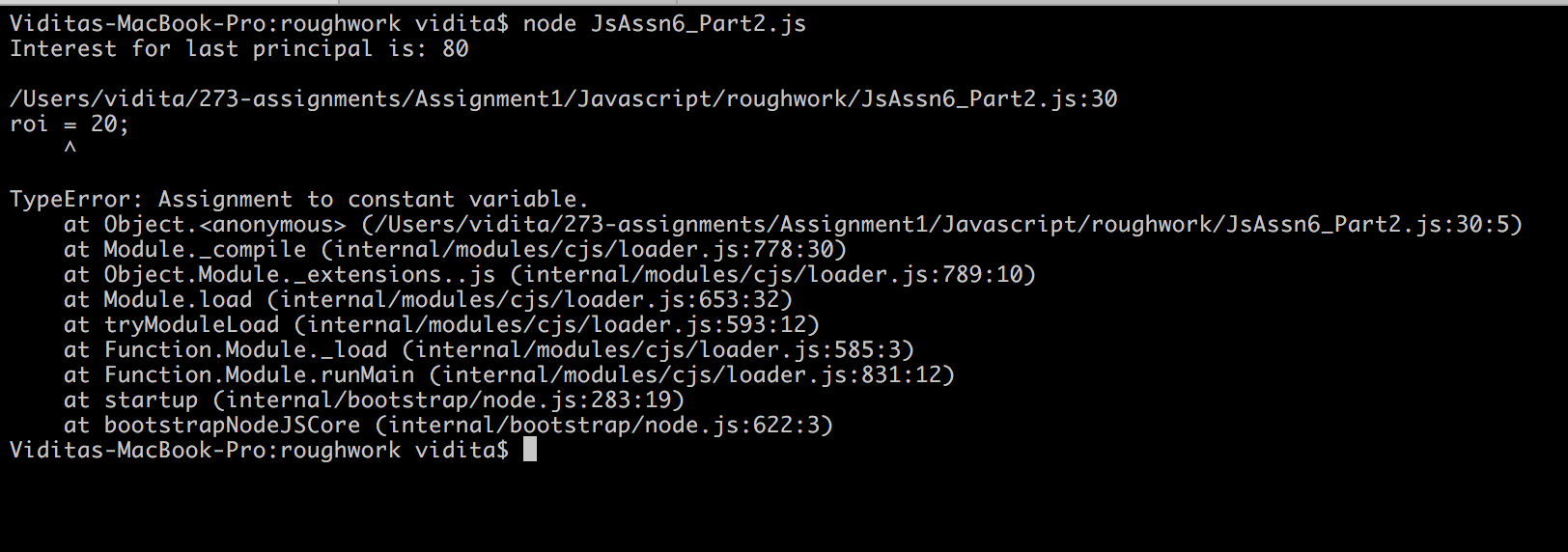
}

calculateSimpleInterest2(principal, 2);

//thorws error

roi = 20;





# **Javascript: 6 – Part 3 ( Difference between callback, promises, async await )**

# **Introduction to topic:**

## A Callback Function is a function that is passed as a parameter to another function and the callback function is run inside the another function.

## Promises are used to handle asynchronous calls. They are used to handle multiple and/or nested callbacks which can lead to unreadable code.

## Aync-Await helps to work with Promises in a more readable way. Making a function async means it always returns a promise. Await is used to wait until the promise is resolved before executing next instruction.

# **Programming Question:**

## Create an application to cook vegetables and soup. Vegetables can be cooked only after all vegetables are available. Soup can be prepared independent of cooking vegetables.

## Implement this using callback function.

## Extend above application to create Vegetable Sandwich after cooking vegetables. Soup can again be prepared independently of creating sandwich. Use promises to avoid nested callbacks.

## Implement same functionality of creating sandwich and soup using async-await.

**Callbacks**:

let getVegetables = (callback) => {

console.log(`Getting onions`);

console.log(`Getting potatoes`);

console.log(`Getting tomatoes`);

console.log('Got vegetables');

callback();

};

let cookVegetables = () => {

console.log(`Cooking vegetables`);

setTimeout(() => {

console.log('Cooked vegetables');

}, 5000);

};

let makeSoup = () => {

console.log('Soup is ready');

}

// Vegetables should be cooked only when they are all available.`

getVegetables(cookVegetables);

// Making soup can be started while the vegetables are being cooked.

makeSoup();

let getVegetables = (callback) => {

console.log(`Getting onions`);

console.log(`Getting potatoes`);

console.log(`Getting tomatoes`);

console.log('Got vegetables');

callback();

};

let cookVegetables = () => {

console.log(`Cooking vegetables`);

setTimeout(() => {

console.log('Cooked vegetables');

}, 5000);

};

let makeSoup = () => {

console.log('Soup is ready');

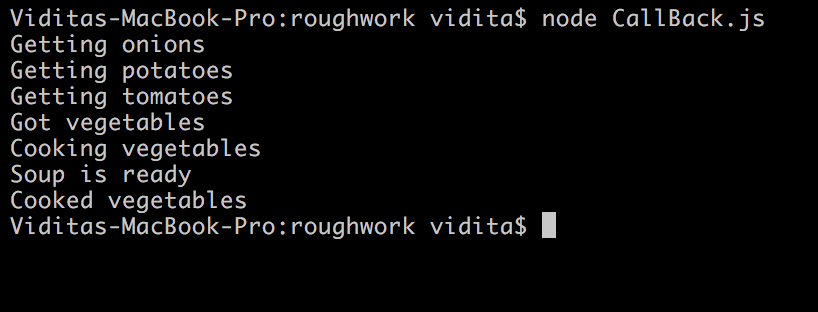
}

// Vegetables should be cooked only when they are all available.`

getVegetables(cookVegetables);

// Making soup can be started while the vegetables are being cooked.

makeSoup();



**Promises**:

let getVegetables = () => {

let promise = new Promise((resolve, reject) => {

console.log(`Getting onions`);

console.log(`Getting potatoes`);

console.log(`Getting tomatoes`);

setTimeout(() => {

console.log('Got all the vegetables');

resolve();

}, 5000);

});

return promise;

};

let cookVegetables = () => {

let promise = new Promise((resolve, reject) => {

console.log('Cooking vegetables');

setTimeout(() => {

console.log('Cooked vegetables');

resolve();

}, 5000);

});

return promise;

};

let putVeggiesInBread = () => {

return new Promise((resolve, reject) => {

console.log('Putting cooked veggies in bread');

resolve();

});

};

let makeSandwich = () => {

getVegetables()

.then(cookVegetables)

.then(putVeggiesInBread)

.then(() => {

console.log('Sandwich is ready');

});

};

let makeSoup = () => {

console.log('Soup is ready');

};

let makeLunch = () => {

// Vegetables should be cooked only when they are all available.

makeSandwich();

// Making soup can be started while the vegetables are being cooked.

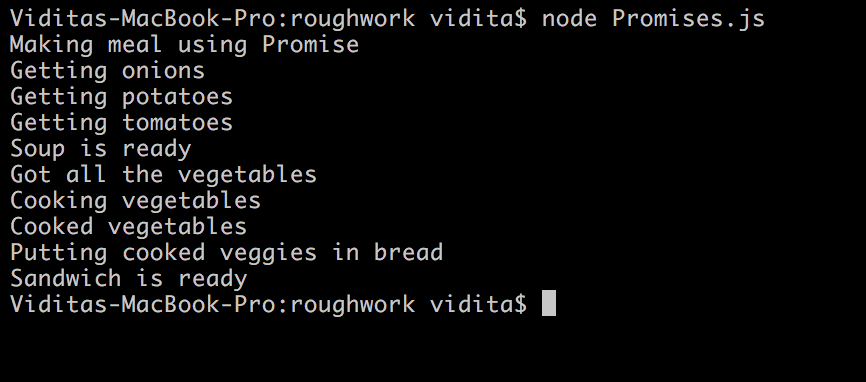
makeSoup();

};

console.log('Making meal using Promise');

makeLunch();

**Output Promises:**



**Async - Await**

let getVegetables = () => {

let promise = new Promise((resolve, reject) => {

console.log(`Getting onions`);

console.log(`Getting potatoes`);

console.log(`Getting tomatoes`);

setTimeout(() => {

console.log('Got all the vegetables');

resolve();

}, 5000);

});

return promise;

};

let cookVegetables = () => {

let promise = new Promise((resolve, reject) => {

console.log('Cooking vegetables');

setTimeout(() => {

console.log('Cooked vegetables');

resolve();

}, 5000);

});

return promise;

};

let putVeggiesInBread = () => {

return new Promise((resolve, reject) => {

console.log('Putting cooked veggies in bread');

resolve();

});

};

async function createSandwich () {

await getVegetables();

await cookVegetables();

await putVeggiesInBread();

console.log('Sandwich is ready');

}

let makeSoup = () => {

console.log('Soup is ready');

};

let makeMeal = () => {

createSandwich();

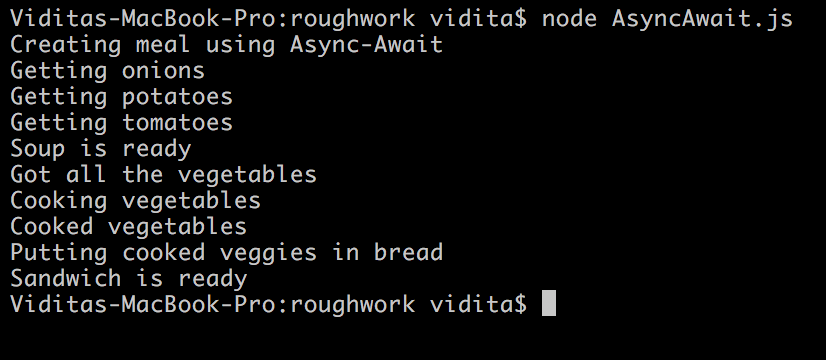
makeSoup();

}

console.log('Creating meal using Async-Await');

makeMeal();

**Output Aync-Await:**



# **HTML: 1**

# **Introduction to topic:**

## LocalStorage is used to save data in client permanently.

## SessionStorage is used to save data in client till the time session is present. The data is lost when the tab is closed.

## Input properties / options used: required, autofocus, pattern, email, password, maxLength, number, textarea with rows and columns.

# **Programming Question:**

## Create an application for blog subscription. User should be able to select categories and subscribe to various blog categories. The user selection should persist even if browser is closed just like shopping cart saves your items even when you close the browser. Also, the application should have registration page. First page should accept basic user details and second page should accept contact details from user. The data entered by user should be persisted as long as user is actively using the website. So if user navigates to page2, the form filled on first page should persist the data temporarily.

**RegistrationPage1.html**

<html>

<head>

<title>Registration Form</title>

<script language="JavaScript">

function saveData() {

sessionStorage.setItem('firstname', document.getElementById("firstname").value);

sessionStorage.setItem('lastname', document.getElementById("lastname").value);

sessionStorage.setItem('dob', document.getElementById("dob").value);

sessionStorage.setItem('email', document.getElementById("email").value);

sessionStorage.setItem('password', document.getElementById("password").value);

alert('Saved to draft');

};

function setData() {

document.getElementById("firstname").value = sessionStorage.getItem('firstname');

document.getElementById("lastname").value = sessionStorage.getItem('lastname');

document.getElementById("dob").value = sessionStorage.getItem('dob');

document.getElementById("email").value = sessionStorage.getItem('email');

document.getElementById("password").value = sessionStorage.getItem('password');

};

</script>

</head>

<body onload="setData()">

<h3 align="center">REGISTRATION FORM</h3>

<form>

<table align="center" cellpadding="10">

<tr>

<td>FIRST NAME</td>

<td><input id="firstname" type="text" name="firstname" required autofocus pattern="[A-Z][a-z]{1,50}"}/>

</td>

</tr>

<tr>

<td>LAST NAME</td>

<td><input id="lastname" type="text" name="lastname" required pattern="[A-Z][a-z]{1,50}" />

</td>

</tr>

<tr>

<td>DATE OF BIRTH</td>

<td><input id="dob" type="date" name="dob" />

</td>

</tr>

<tr>

<td>EMAIL ID</td>

<td><input id="email" type="email" name="emailId" required /></td>

</tr>

<tr>

<td>PASSWORD</td>

<td><input id="password" type="password" name="password"

required /></td>

</tr>

<tr>

<td colspan="2" align="center">

<input type="submit" value="Save" name="save" formaction="javascript:saveData()" />

</td>

</tr>

<tr>

<td colspan="2" align="center">

<input type="submit" value="Next" name="next" formaction="RegistrationPage2.html" />

</td>

</tr>

</table>

</form>

</body>

</html>

RegistrationPage2.html

<html>

<head>

<title>Registration Form</title>

<script language="JavaScript">

function submit() {

sessionStorage.setItem('mobile', document.getElementById("mobile").value);

sessionStorage.setItem('address', document.getElementById("address").value);

alert('Registration succesfull');

// Save data permanently to a DB

};

function setData() {

document.getElementById("mobile").value = sessionStorage.getItem('mobile');

document.getElementById("address").value = sessionStorage.getItem('address');

};

</script>

</head>

<body onload="setData()">

<h3 align="center">REGISTRATION FORM</h3>

<form>

<table align="center" cellpadding="10">

<tr>

<td>MOBILE NUMBER</td>

<td>

<input id="mobile" type="text" name="mobile" maxlength="10" />

</td>

</tr>

<tr>

<td>ADDRESS <br /><br /><br /></td>

<td><textarea id="address" name="address" rows="4" cols="30"></textarea></td>

</tr>

<tr>

<td colspan="2" align="center">

<input type="submit" value="Back" name="back" formaction="RegistrationPage1.html" />

</td>

<td colspan="2" align="center">

<input type="submit" value="Submit" name="save" formaction="javascript:submit()" />

</td>

</tr>

</table>

</form>

</body>

</html>

SubscriptionPage.html

<!DOCTYPE html>

<html>

<head>

<title>Blog Subscription</title>

<script language="JavaScript">

function saveData(id) {

if (document.getElementById(id).checked) {

localStorage.setItem(id, 'true');

} else {

localStorage.setItem(id, 'false');

}

};

function setData() {

document.getElementById("food").checked = (localStorage.getItem('food') === 'true');

document.getElementById("technology").checked = (localStorage.getItem('technology') === 'true');

document.getElementById("health").checked = (localStorage.getItem('health') === 'true');

document.getElementById("music").checked = (localStorage.getItem('music') === 'true');

document.getElementById("travel").checked = (localStorage.getItem('travel') === ' true');

};

function subscribe() {

alert('You are now subscribed to these blogs');

}

</script>

</head>

<body onload="setData()">

<h3 align="center">Subscribe to blogs</h3>

<form>

<table align="center" cellpadding="10">

<tr>

<td>

<input id="technology" type="checkbox" name="technology" value="technology" onclick="saveData('technology')" />

TECHNOLOGY<br>

</td>

</tr>

<tr>

<td>

<input id="food" type="checkbox" name="food" value="food" onclick="saveData('food')" />

FOOD<br>

</td>

</tr>

<tr>

<td>

<input id="health" type="checkbox" name="health" value="health" onclick="saveData('health')" />

HEALTH<br>

</td>

</tr>

<tr>

<td>

<input id="music" type="checkbox" name="music" value="music" onclick="saveData('music')" />

MUSIC<br>

</td>

</tr>

<tr>

<td>

<input id="travel" type="checkbox" name="travel" value="travel" onclick="saveData('travel')" />

TRAVEL<br>

</td>

</tr>

<tr>

<td colspan="2" align="center">

<input type="submit" value="SUBSCRIBE" name="subscribe" formaction="javascript:subscribe()" />

</td>

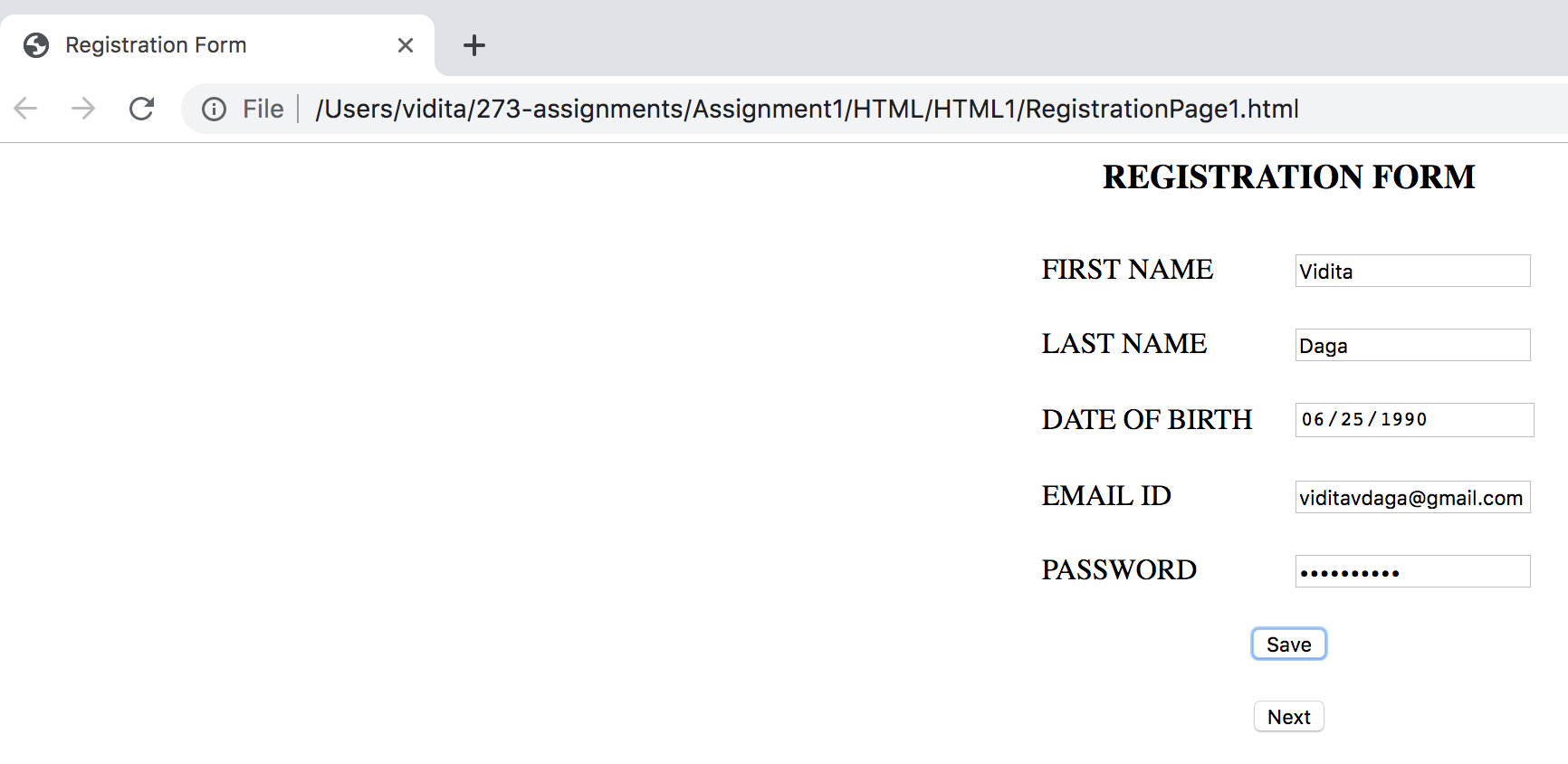
</tr>

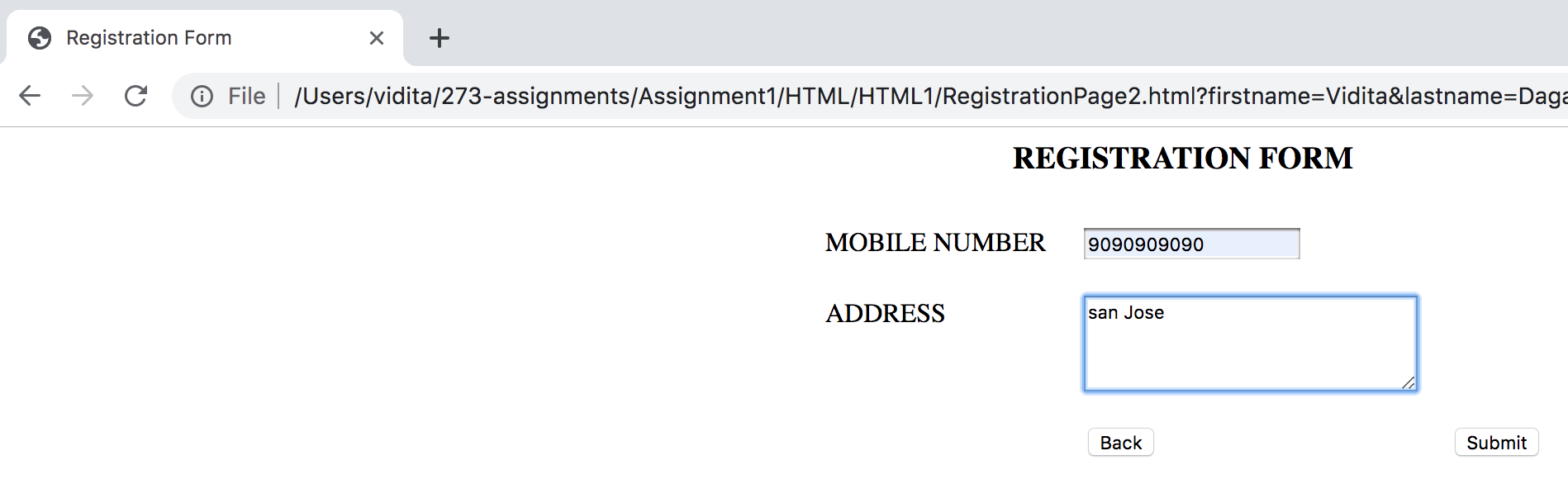
</table>

</form>

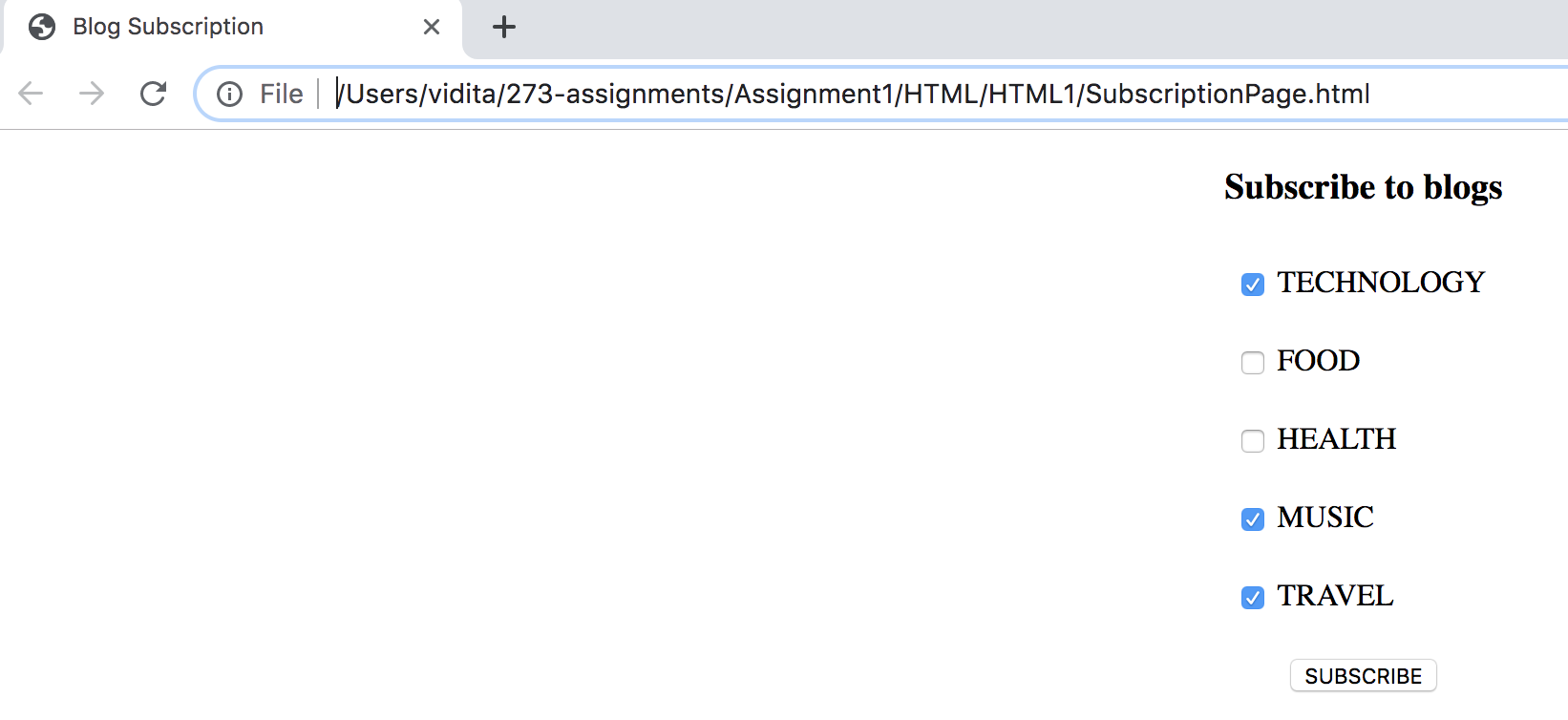
</body>

</html>





Data stored in Local Storage so that selection is retained



# **HTML: 2**

# **Introduction to topic:**

## HTML5 Geolocation APIs provide APIs to get current position & watch the position as the location of the device changes.

## Events in HTML5 help respond to user requests and enable user interaction.

# **Programming Question:**

## A food delivery app allows it’s user to track the location of the delivery person. User can either chose to see the exact current location or track the location of the delivery person to know the route. Implement this functionality using HTML Geolocation and events.

## Note: Only Latitude and Longitude is shown since showing the location on map requires Google API key.

<!DOCTYPE html>

<html lang="en">

<head>

<title>Location Tracker</title>

<script>

var id;

let getLocation = () => {

if(navigator.geolocation){

navigator.geolocation.getCurrentPosition(displayPos, handleErr);

} else{

alert("Browser does not support HTML5 Geolocation");

}

}

let trackLocation = () => {

if (navigator.geolocation) {

id = navigator.geolocation.watchPosition(displayPos, handleErr);

} else {

alert("Browser does not support HTML5 Geolocation");

}

}

let clearWatch = () => {

navigator.geolocation.clearWatch(id);

}

let displayPos = (position) => {

document.getElementById('latitude').value = position.coords.latitude;

document.getElementById('longitude').value = position.coords.longitude;

}

function handleErr(error) {

alert(`Error occured: ${error.message}`);

}

</script>

</head>

<body>

<h3 align="center">Location finder & Tracker</h3>

<form>

<table align="center" cellpadding="10">

<tr>

<td>

<input type="submit" value="GET CURRENT POSITION" formaction="javascript:getLocation()" />

</td>

</tr>

<tr>

<td>

<input type="submit" value="TRACK POSITION" formaction="javascript:trackLocation()" />

</td>

</tr>

<tr>

<td>

<input type="submit" value="STOP TRACKING" formaction="javascript:clearWatch()" />

</td>

</tr>

<tr>

<td>LATITUDE</td>

<td><input id="latitude" type="text" name="latitude" />

</td>

<td>LONGITUDE</td>

<td><input id="longitude" type="text" name="longitude" />

</td>

</tr>

</table>

</form>

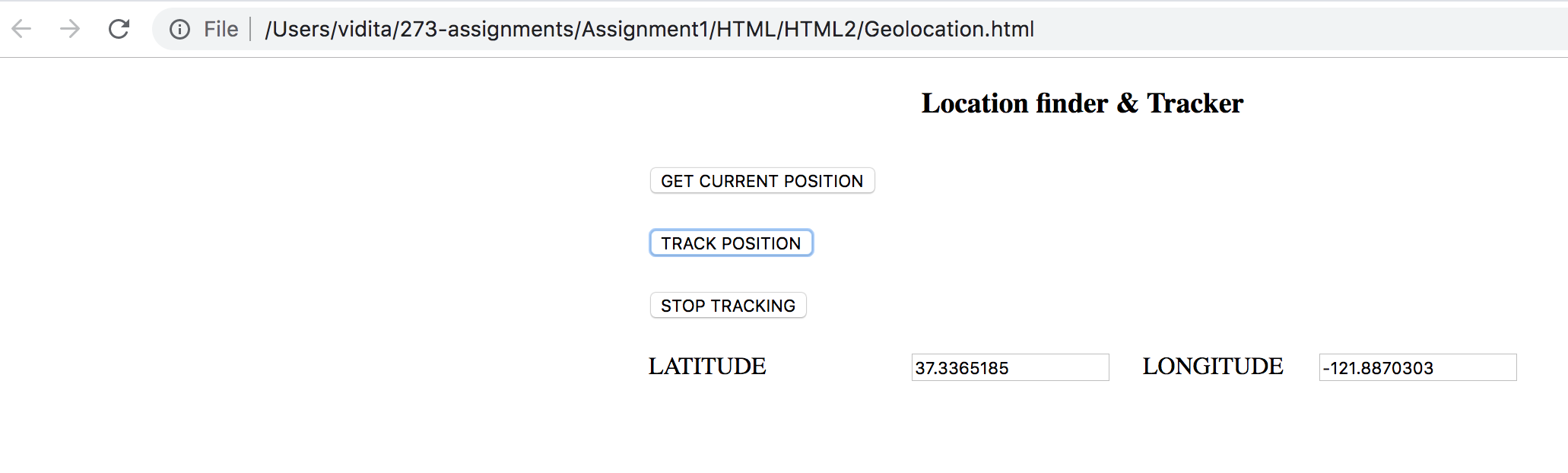
</body>

</html>

# **Application Deployment:**

## Create an application and include above programs. Create a docker image of this and deploy to AWS ECS

# **Application URL:**



# **Application & Deployment:**

## Create an application for parking allotment given that 5 parking slots numbered from 1 to 5 are available. Provide functionalities to display current allocation, allocate slot to a new user and vacate slot. Create docker image and deploy it to AWS ECS.

**Application URL:** <http://ec2-18-206-91-33.compute-1.amazonaws.com:8080/parkingallotment/>