Portfolio 2023

Contents

[Meta 2](#_Toc159014681)

[Pages 2](#_Toc159014682)

[Landing Page 2](#_Toc159014683)

[About Page 4](#_Toc159014684)

[Projects Page 6](#_Toc159014685)

[Contact Page 7](#_Toc159014686)

[Blogs 8](#_Toc159014687)

[Digital Guitar - Hardware 10](#_Toc159014688)

[Digital Guitar - Make Guitar Sounds with Howler 19](#_Toc159014689)

[Validating Dates with JavaScript 24](#_Toc159014690)

[JavaScript's Sorting – What's Behind the Hood 26](#_Toc159014691)

[Green Rooftop - An Arduino IoT Project 30](#_Toc159014692)

[Scheduling Emails with Cyclic 38](#_Toc159014693)

[Brief Anatomy of React 40](#_Toc159014694)

[Git Cheat Sheet 49](#_Toc159014695)

[Funtional Programming with Monet.js 57](#_Toc159014696)

[React Custom Hook Pattern 61](#_Toc159014697)

[Recreating McDonald's Ordering Kiosk UI 61](#_Toc159014698)

# Meta

*[meta describe content]*: Tivadar Debnar | Web Developer Portfolio. Explore advanced web development techniques, coding tips, and projects. Unlock the power of JavaScript for creating dynamic websites. Get inspired now.

# Pages

## Landing Page

**Intro Text**

"Hello there! I'm pleased that you've found your way here. Whether you share my passion for coding and design or need a supporting hand with web development, you've come to the right place."

**About Text**

I'm a web developer based in London. I deeply love crafting beautiful and functional websites and web applications. I strive to create impressive digital experiences that leave an impact and projects I'm proud of. I am especially fascinated by modern web design and user interfaces.

**Options**

How can I help you?

**Seeking a developer to join our team?**

I appreciate your interest. Let me provide you with a brief overview of myself as a developer.

I am passionate about web development and enjoy transforming ideas into reality. For me, coding is not just a job but a creative instrument where I can continually improve and challenge myself.

Although I self-studied web development for many years prior, I have recently completed my formal studies, graduating from Falmouth University in Computing (BSc Hons, first). During my academic journey, I consolidated my programming concepts and gained hands-on experience in various technologies and languages, such as Python or C#. However, my passion lies in web development. I have been creating projects in JavaScript and MERN (Mongo, Express, React, NodeJs) since 2017, and I continued to work on my side projects even during my years in college and university.

**Skills**

HTML5 – 70,

CSS – 80,

JavaScript – 70,

React – 60,

NodeJs with Express – 40,

MongoDB with Mongoose – 40,

Adobe – 30,

**Currently Learning**

Sass – 30,

TypeScript – 35,

Jest – 20,

I continuously learn new things.

I am excited to apply my skills and contribute to a dynamic development team. If my profile aligns with your requirements and you feel I would be a good fit for your team, I would be thrilled to discuss further details. Please don't hesitate to contact me.

**Looking for a programming buddy?**

Are you searching for a programming buddy to embark on coding adventures together? Look no further! I'm excited to connect with fellow developers with the same passion and enthusiasm for programming.

Collaboration and learning are at the heart of my journey, and having a programming buddy can make the experience even more rewarding. We can conquer coding challenges, share knowledge, and push each other to reach our goals. Having a programming buddy means having someone who can provide support and motivation when the going gets tough. We can work on projects, troubleshoot code, and celebrate our achievements together. It's all about fostering a positive environment where we can grow as developers.

We're off to a great start if you're also driven by curiosity and a desire to improve your programming skills. I'm particularly interested in web development and exploring different programming languages and frameworks. I am taking courses in TypeScript and React 18 and planning to move towards TDD with Jest, so if this interests you, it's even better. Alternatively, we also can do coding challenges, and you can find me on codewars:

If you're as enthusiastic as I am about finding a programming buddy, I'd love to hear from you. Feel free to email me at... Let's join forces and make our coding journey even more remarkable!

**Looking for Likeminded Friends?**

**-**

**Just here to get to know me?**

## About Page

**The Beginnings**

I often have nostalgic memories about the first time I immersed myself in a Pascal programming book and began tinkering with my computer, a Z80 Enterprise. My enthusiasm didn't wane regardless that most available books were English in the late 90s, and I was a Hungarian 13-year-old with a minimal English vocabulary. So equipped with a dictionary, I started deciphering my textbooks and creating my first text-based games. The early exposure to programming proved to be a perfect inspiration, fueling my love for languages of all kinds, whether spoken by humans or written in code.

**Self Study**

Curiosity and a mystical fascination with coding have been constant companions throughout my adult life, propelling me to explore browser-based programming technologies like JavaScript, CSS, and HTML. Soon, I found myself coding daily, taking online courses, such as FreeCodeCamp's front-end development course, and attending meetups of different coding-related groups. With each passing moment dedicated to my studies and projects, I came to the profound realisation that programming was my true calling, urging me to pursue a career as a professional web developer.

**College**

To make my commitment official, in 2020, I enrolled in Icon College of Technology and Management to get my Higher National Diploma (HND) in Computing, where I studied some fascinating subjects, such as Algorithms and Data Structures, Databases, Web Development, Project Management and Security. The knowledge gained from my self-guided studies proved invaluable across various subjects, resulting in my graduating with distinction and setting the stage for me to pursue a bachelor's degree.

**Uni**

During my time at Falmouth University, I delved into the field of software engineering, focusing on Mobile Development (React Native), Artificial Intelligence, Big Data, IoT, and several other exciting disciplines. Additionally, I concentrated every spare time on my dissertation project, one of the most stimulating and complex projects I have ever embarked on: a digital guitar console that can be played on a browser. This project involved nearly 800 hours of electric wiring, manufacturing, microcontroller programming, and extensive use of JavaScript / NodeJS for its user interface and server. The final product received an exceptional 90% mark and helped me towards a first-class degree with honours.

You can check out this project in detail here.

**Continued Self Development**

I consider completing my bachelor's degree the beginning of my self-development and an exciting phase of a life-long professional development journey. I am now further expanding my skills by taking courses on TypeScript and refreshing my knowledge in React and Sass to consolidate my expertise in my current stack.

**Certificates**

## Projects Page

Welcome to my projects page! Let me introduce some exciting work showcasing my passion for web development and creativity. Each of these projects handed me opportunities to tackle some interesting problems. From responsive front-end designs with HTML, CSS, JavaScript and React to back-end solutions with Node.js and Express, I've leveraged many tools to bring these projects to life.

Also, feel free to filter my projects by technology you are interested in!

**RiffMaster**

My groundbreaking project aims to revolutionise how we learn and play the guitar. With a custom-designed digital guitar controller and an interactive web application, RiffMaster offers an entertaining and educative musical experience for both beginners and experienced guitarists. The platform provides comprehensive lessons and enables composing and playing real-time music games with RiffMaster's innovative hardware and software fusion.

**Francesco Levo**

I crafted a sleek portfolio website for my personal trainer client to expand his reach. This website features a minimalist dark web design, a powerful logo representing his brand identity, seamless multiple language support, testimonials, personal statements, health and nutritional recommendations, a user-friendly Google Calendar integration for easy bookings, and a contact form to communicate with potential clients.

**Pattern Generator**

Generating intricate CSS code becomes a breeze. With a comprehensive range of linear and radial gradient tools and pattern samples, you can easily adjust your designs using intuitive sliders, bypassing the need for lengthy vendor-specific CSS code. Creating captivating CSS patterns has never been more straightforward, featuring a custom colour picker, with palettes of websafe colours (rgba, hsl and hex), and support for various CSS units and angles.

**RainCheck**

RainCheck is a comprehensive tool designed to provide users with relevant weather information at their fingertips. Offering user-friendly time and location formatting and essential weather features such as temperature, wind speed, precipitation, and humidity. With hourly and daily breakdowns, users can confidently plan their activities. Additionally, the app boasts a settings screen for metric/imperial toggling and location selection, all wrapped in straightforward navigation for seamless usability.

**Green Rooftop IoT Project**

The GreenRoof project is dedicated to building a sustainable urban environment by offering an innovative IoT device that facilitates the monitoring and maintaining green rooftops. This cutting-edge technology allows rooftop owners and service providers to access real-time data on soil moisture, temperature, humidity, light, and rainwater tank status. The website complements the device by providing an intuitive interface for users to view their data effortlessly. GreenRoof is shaping urban landscapes' greener and more sustainable future by promoting eco-awareness and natural living spaces.

**Fruits and Flowers**

Fruits and Flowers began as a personal challenge to test my JavaScript skills and evolved into a captivating game with over 100 levels. Unlike traditional match-three games, Fruits and Flowers offers unique challenges with special gems, adding extra layers of complexity. Players can earn in-game points to acquire unique gems that provide advantages at tougher levels. While this early project may not adhere to all programming best practices, it holds a special place in my heart as a significant milestone in my development journey.

**Pocket Tutor**

Pocket Tutor is a versatile app that facilitates autodidactic learning through interactive flashcards, perfect for on-the-go practice. With the ability to create custom topic badges and flashcards, users can quickly test their knowledge anytime, anywhere. The flashcards support code snippets with syntax highlighting for HTML, CSS, and JavaScript, making them ideal for aspiring developers. To tailor their learning experience, users can set test difficulties and track their progress using insightful performance diagrams. Pocket Tutor empowers learners to take charge of their education and deepen their understanding.

## Contact Page

Let's Get in Contact! I'm excited to hear from you! Whether you have a project in mind, want to collaborate, or simply want to say hello, I'd love to hear from you. Please feel free to fill out the form below, and I'll get back to you as soon as possible. Happy coding!

# Blogs

My blogs are a way of giving back - and paying forward - all the help I have been given through my programming learning journey. Feel free to browse my articles that cover some of my findings, solutions and tutorials on exciting and relevant topics, such as programming languages, web development and project walkthroughs. Happy Coding!

**Digital Guitar**

Let's create a guitar console with Arduino that sends data through a USB cable using a keyboard library to build a guitar web application. Discover matrix wiring and keyboard debounce handling.

**Digital Guitar - Make Guitar Sounds with Howler**

In this article, we create a keyboard listener for our digital guitar and translate our protocols into appropriate notes. We'll also build a digital interface for our guitar jam page that displays the guitar fretboard and user actions.

**Validating Dates with JavaScript**

I bumped into a problem involving JavaScript date validation, formatting and leap years. Let's explore our options.

**Exploring JavaScript's Sorting Mechanism: Behind the Scenes**

Delve into the intricate inner workings of JavaScript's built-in sorting mechanism. Uncover the magic behind Array.sort() as we dissect the comparison functions, sorting stability, and optimisation strategies employed by the language.

**Green Rooftop - An Arduino IoT Project**

An ideal IoT project for beginners. Learn how to measure temperature, humidity, moisture, light, and water tank levels while creating a UI with buttons for easy navigation. We'll also use NodeJS to show you how to build a simple API to access real-time and historical sensory data.

**Scheduling Emails with Cyclic**

Discover a solution for scheduling emails with Node.js in unconventional hosting environments like Cyclic. Learn why traditional Node schedulers may not work and explore my workaround involving a separate post path for scheduling within your application and the Cron scheduler.

**Brief Anatomy of React**

My last interview questions drifted towards the exciting topics of how React works internally, so here is my brief research on this subject in hindsight. We'll cover virtual DOM, component architecture, JSX transformations, rendering processes, and more.

**Git Cheat Sheet**

Git is the ultimate weapon in every professional software engineer's arsenal, as it has become the standard software development tool in the industry for collaborating engineers. This comprehensive cheat sheet is your key to mastering Git, and I'm thrilled to share my personal collection of Git commands, tips and research with you.

**Recreating McDonald's Ordering Kiosk UI**

Discover the secrets of user-friendly UI design as we recreate McDonald's ordering kiosk interface. Learn the techniques behind its user-friendly layout while learning the fundamentals of Flexbox.

**Funtional Programming with Monet.js**

I delved into the devine art of functional programming in JavaScript using Monet.js, and decided to create a pocket guide covering fundamental concepts, monads such as Identity, Maybe, Either, Writer, and Future, to create more robust, composable, and error-resistant code.

**React Custom Hook Pattern**

I've often encountered React hook errors when it came to working with nested functions and hooks together. Luckily, while pair programming with a senior frontend developer, I discovered a pattern that would've been a lifesaver many times to avoid these errors.

## Digital Guitar - Hardware

**Project RiffMaster**

**How to Build Your Digital Guitar Instrument with an Application**

[Intro Image]

We all know the pain when it comes to choosing a dissertation topic, and it is exponentially more excruciating when our project must also be implemented for more practical disciplines, such as computer science and software engineering. While originality is encouraged, graduates must also consider the feasibility and the relatively short deadline. This digital guitar project idea was born in my internal struggle to be innovative and create a project that might contribute to the greater good with some educational and entertainment aspects.

[Documentation Link]

**Project Outline**

We'll create a bespoke music experience by developing a hardware device and accompanying software that offers a naturalistic guitar layout for learning and playing. The device's minimalistic design ensures affordability while connecting via USB to an online platform. This comprehensive solution aims to enhance guitar skill development through playful learning of riffs, chords, and songs.

**The Guitar**

First, we need to build a digital guitar with the following features: realistic guitar dimensions and shape, USB connection, 20 fret buttons in six rows, strum bar switches, three LEDs and a power toggle. Our instrument will feature an Arduino board to detect and communicate user interactions through USB; therefore, an Arduino Due will be used, as it has plenty of digital pins and has bidirectional USB communication (can act as an external keyboard). Alternatively, Arduino Leonardo or Mega can also be used.

[Final Product Image]

I used a Gibson Les Paul template [link here] to ensure the instrument had appropriate dimensions and shape. Although Les Paul is designed for electric guitars, we can build it semi-acoustic to fit the electric components in the body cavity.

**Building the Neck**

We can start manufacturing with the guitar neck, as this component is essential for building a working solution. The user will interact with a range of buttons in a ***mesh arrangement*** to communicate which string has active finger press positions associated. Hence, the neck width and depth must be small enough that all six buttons on a fret may be pressed simultaneously with an index finger for barre chords. However, the neck must be wide enough to leave sufficient space for the electronics (42-57mm for Les Paul). Our console's neck will have a constant width to simplify the hardware development, and 50mm is a compromise that satisfies usability requirements while sufficient for allocating all electrical components. The length of the neck is the standard 460mm, and the fret distances are calculated with the golden ratio (recursively dividing the distance between the neck and saddle by 1.618).

[Guitar Neck Distance Image]

We will use ***polystyrene*** sheets for the prototype in two thickness sizes: 2mm for bending and 3mm for structure. Polystyrene sheets are available in most hardware shops; they are cheap (£30-50), bend on heat well, are relatively light, can be cut without specialised tools, and are excellent electrical insulators. The guitar neck requires no bending, but we can use the 2mm plastic to cut our template to reduce the thickness of the neck.

[Drilling Template]

The drill holes will be evenly allocated alongside the fret's axis, distributing the buttons comfortably with 0.7mm wide switch legs in rectangular arrangements.

[Neck Design Image]

We also need 20 fret pieces from the 2mm material, so our frets may receive string supports with grooves, which offer protection for tactile switches, provide guides to the users' hands, and can keep actual strings in place for an optional stringed solution.

[Fret Design Image]

The buttons will face towards the user's fingers; therefore, no electronic components may be allocated to the otherwise compressed interface. The switch legs must lead through the drill holes by wire extensions to reach the backside circuit. The electronics must be protected from physical impacts; therefore, a back cover and side protection are required. Moreover, the neck length must be extended to hold the neck in the instrument body securely.

[Neck Cover Image]

**Wiring the Neck**

As I mentioned, we will use mesh wiring for our buttons in a 6 x 20 matrix. Keyboards and keypads often use matrices to consolidate greater numbers of switches to microcontroller pins. *When a key is pressed, a column wire contacts a row wire and completes a circuit. The keyboard controller detects this closed circuit and registers it as a key press*. For example, PC keyboards usually range from 63 to 105 keys arranged in a matrix. Meshing the switch wires would result in a drastically reduced digital pin requirement. Matrix keyboards use scanning algorithms to detect button presses, where rows and columns are individually read. After reengineering different matrix examples, I boiled down the algorithms to this simple version:

[Matrix Scanning Image]

We may wire up our button matrix with any N1 range diode to eliminate keypress ghosting and masking issues. (I used N14007 diodes).

[Fret Button Matrix Image]

I used universal PCBs pasted on the backside of the fretboard to simplify soldering. Additionally, I used six stripped wires for the columns to reduce a substantial amount of extra wiring. These stripped wires serve as buses to the Arduino column digital pins.

[Fret Wiring Image]

Finally, we can connect the row and column wires to our Arduino. Although the choice of digital pins may seem arbitrary, the reason I used these pins is that some of my digital pins were faulty. Ideally, I could use pins 22-52, but the main point is that row, and column wires should be close enough to tape them together. (Otherwise, there will be a lot of messy wiring.)

**Testing the Fretboard**

Our first milestone is to read the device's state. When state change happens, they are communicated through USB as sequences of emulated keyboard presses. We may use simplified ***state machines*** to determine currently activated switches. We must differentiate previous and current conditions for frets/strums. We allocate a pair of 6/20 2D matrices and length-six arrays for the frets and strums. The initial values are reversed as we used pull-up resistors: zero means pressed, and one means released.

[Initialisation Code Snippet]

Next, we need to initialise our digital pins. We may apply Arduino's internal pull-up resistor because we did not use resistors in our circuitry.

We activate the column pins with each loop and read individual row values, updating the previous and current button states. We can test our buttons on the serial monitor with the following code snippet:

[Testing Code Snippet]

**Building the Guitar Body**

The main right-hand component is the strum unit, which triggers events that produce sound effects or gameplay actions. But beyond the strum, the instrument's body must allocate a standard-size on/off toggle, LEDs, USB connection, and support elements to attach the neck, cover, and body. Hand manufacturing constraints the range of executable solutions, like bending perfect curves from plastic or creating parts that fit surgically together, and our design must consider an acceptable tolerance of ±1mm.

[Guitar Body Design Image]

We cut two identical sheets for the guitar body's front and back cover, and the front piece must be drilled for the toggle switch and three LEDs.

[Guitar Body Front Cover Image]

**Bending the Side Piece**

A wooden template body was created to bend the side walls to the appropriate curvatures because several attempts of free-handed bending resulted in broken pieces of already cut material. Although the polystyrene side piece reacted to the heat, I highly recommend using a hairdryer and proceeding very slowly with the bending.

[Bending Image]

**Support Elements**

The body must be assembled in a way to allow disassembly for servicing. Support units with embedded nuts and screws will hold the body and cover together. These connectors reinforce the device's structural integrity, as the body may be under more stress than other electrical handheld devices. The body must withstand the pressure of the user's right arm's weight without damaging the open-cavity structure. One typical guitar design failure is the weak neck-body connection, resulting in bent or damaged guitar necks. Substantial pulling or pushing forces generate leverage that may easily deform or break the controller apart, and additional components will reinforce the neck to endure reasonable use.

[Guitar Body Assembly Design]

[Guitar Support Elements Image]

**Finalising the Neck Cover**

Assuming that we tested the fretboard buttons (and button press combinations), we can finalise the guitar neck with side walls and a removable cover. Not doing it right after the fretboard development and waiting to finish the body first gave me enough time to test the fretboard thoroughly. With the sidewalls, correcting wiring or replacing components in this tight circuit would be a nightmare.

[Neck Cover Image]

**Neck Body Attachment**

The neck attachment was the riskiest part of the assembly. Should the neck have failed to hold together precisely, the whole project might have been endangered, as Gorilla glue and CT1 cannot be removed without severe damage.

[Body Neck Joint Image]

We can finish our lovely guitar with a lick of matt black paint.

**Strum Unit**

Guitars are polyphonic instruments, and strings can be strummed simultaneously. Additionally, guitars produce sound when they are strummed, except hammer-on and pull-up sounds, which are beyond our project's scope. We need to find a straightforward solution to trigger a strum event. For authenticity, we can go one step ahead of Guitar Hero consoles that use swinging mechanical toggle switches for strums and equip our instrument with actual strings. The console strings are attached to fixed points, and one end features a momentary tactile switch, and the other may be set to a screw that adjusts the tension. The switch side can also apply a lever with adjustable tension.

[Strum Mechanism Image]

Some of the other feasible solutions are vibration or pressure sensors, ideal for improved versions of the strum mechanism. These could be used as analogue inputs, and some additional features could be implemented, such as muting and strum force. In the future, I intend to experiment with these and create a working analogue solution.

**Strum Unit Design**

The strum component is a separate serviceable unit that can be easily removed; therefore, we may create a frame scaffolding for our strings and a hull that hollows in the guitar body. This design assembles the unit from flat plastic pieces that can be pasted together:

[Strum Components Image]

We can put our components together in the following way:

[Strum Unit Image]

**Strum Assembly**

The strum unit was reasonably straightforward to assemble; however, the screws, levers, and string tension adjustments were time-consuming.

[Strum Assembly Image 1]

[Strum Assembly Image 2]

[Strum Assembly Image 3]

We can wire up our unit and connect it to the Arduino. I used digital pins 16 to 35 for the column (fret) and 2 to 7 for the column (row) wires. As I mentioned, my pin layout is affected by faulty digital pins.

[Strum Wiring Image]

**Toggle and LEDs**

Users should interact with the device without disrupting traditional inputs, and USB keyboard communication may depend on the final communication protocol. The device must have an ON/OFF switch that disables any serial communication towards the computer. Additionally, three LEDs will provide further information about the device's state:

• RED: Device connected to USB,

• GREEN: USB communication is ON,

• BLUE: USB data transfer.

[Toggle and LEDs Circuitry Image]

[Arduino Wiring Image]

[Toggle Wiring Image]

Finally, we can string up the device.

[Assembled Strum Unit Image]

**Registering Strums**

For completeness and common ground, let me include my pin layout and variables that may appear in the upcoming code snippets. Alternatively, you can download the complete controller code [here: controller.ino].

[Constants and Variables Code Snippet]

First, we need to initialise our pins in the setup function.

[Strum Setup Code Snippet]

We can flag cycles where a state change happens by setting a transaction variable false at the beginning of each cycle and flipping it to true whenever a fret change or a state change occurs. The possible state changes are the following: fretboard buttons are pressed/released, or strings are activated or in their default stationary position. These state changes must trigger a controller event by setting the device's transaction state to true.

[Controller Transaction State Code Snippet]

**Communication Protocol**

There are multiple ways to design device communication, such as MIDI or binary protocols; however, I sought an easy-to-implement version in this scenario. Our Arduino will be used as an external keyboard to communicate with our computer, transferring textual information. But there are over 252 possible state combinations and just 62 alphanumeric values that can be safely used if we use both upper and lower case letters and digits. This means that we will send multiple keypress events. The fewer events are used, the more performant the communication will be, and we may easily fit an event into a two-character combination (3844). On the other hand, creating a logical mapping between character pairs and events is cumbersome, and I would rather see a solution with some relation to the physical state of the device. A more straightforward protocol could compromise with four characters using only digits. A flag for the event, a fret number, and a strum string number.

[Protocol Image]

**Debounce Handling**

Momentary tactile switches are not guaranteed to be immune to physical bouncing and may trigger unsolicited actions from the controller. Furthermore, our device has two types of inputs with directly opposing behaviour.

**Fret Buttons**

From the traditional perspective, inputs register events ***when pressed***, just like our fret switches. When activation happens, we assume that state changes are intentional after a particular interval, typically ~50-100ms.

**Strum Switches**

Unlike frets, strum switches are activated after strings are put under pressure and ***released***. We may also expect extra noise, as our switches use strings, levers and springs that may oscillate. After the first release, we measure the interval of state changes and register if the last change happened after the ***debounce time allowance***.

[Debouncing Image]

**Sending an Event Message**

We can invoke a function on every cycle with a transaction state change by listening to it in the main loop.

[Listening Code Snippet]

Now that we have our communication protocol and are aware of the debounce issues, we can send a message from our device through keyboard presses. First, we must initialise a debounce array for strums and a 2D array for frets and measure time with the millis function. Then we traverse both the fret and strum states separately, and any change detected outside the debounce timing will send a four-digit keyboard sequence with the event, fret and strum information using the standard Keyboard library.

[Set Controller State Code Snippet]

**Head Stock**

Although users can interact with the fret by buttons and use chord patterns on the left-hand side, controls may not feel authentic enough. After testing the device for more complex chords and finger patterns, I came to the conclusion that finding the exact locations of fret positions is more problematic, even after equipping my buttons with button caps. Therefore, setting up our instrument with strings would hopefully help guide the user's hand naturally as guitar strings do. To set up the six strings on the fretboard, we have to create a headstock, which provides fixed positions to ***attach strings as guidelines***. The headstock unit is an optional accessory and must be attachable/detachable to the neck.

[Head Stock Design Image]

Now that we have our headstock, we can set up our strings. Depending on the type of button caps used, modifications may be needed; the caps I ordered made the strings slip off from their place when pressed, so I grooved all 120 individually.

[Head Stock Back Image]

[Head Stock Front Image]

Lastly, similar to our fret design at the beginning of this post, we may create a saddle to hold our string in place.

[Saddle Image]

Now that our guitar console is up and running, we are ready to develop our own guitar application. You can read my article about how to build your own guitar hero game prototype [here].

## Digital Guitar - Make Guitar Sounds with Howler

[Equalizer Image]

This article continues my previous blog post, where I provided a walkthrough on creating a digital guitar instrument. This instrument communicates through USB as a keyboard, and for simplicity, we translate individual fretboard and strum events into four-digit messages. Therefore, some of the code snippets I provide here are in the context of the guitar project and may need your refactoring if you want to use them in your application. If you missed my blog or need references on how this guitar works, click [here].

Additionally, this article focuses on creating guitar sounds according to user inputs on the guitar console, and discussing the basic web app that I built for my device would be a distraction. I will dedicate a separate article to that.

[RiffMaster Home Page Image]

**What is Howler JS?**

Howler.js is a popular JavaScript library for working with audio in web applications. It provides an easy-to-use API for playing and controlling audio files in various formats, making it a versatile tool for creating audio experiences on the web. It offers cross-browser and mobile compatibility, supports many audio formats (MP3, WAV, OGG), and features playback controls (play, pause, stop), looping, positional audio and fade effects. [Link: Howler]

To use Howler.js in your vanilla JS application, you include the library in your project, load audio files, and use the provided API to control and manage audio playback. So after creating the html boilerplate for our app, one of the first things we need to do is to import the Howler JS script in our header.

[Import Howler Code Snippet]

However, for React users, installing the package through NPM or Yarn is the best way. If the application is written with TypeScript, the type definitions must also be installed for the TypeScript compiler.

[NPM Howler Code Snippet]

**Audio Files**

As our application simulates standard acoustic guitars, and the fretboard supports 20 frets on six strings, we must have an extensive range of audio sounds available for each possible note, from the lower E2 to the upper D6.

[Fret Board Arrangement Image]

You can create your own audio or download guitar sounds from [Link: splice] or [Link: ]. However, I found collecting the full range of notes with the same instrument challenging and rather time-consuming, so I opted to record my own set of guitar sounds. If you want to spare a few hours, please use my audio library [Link: audio library]! (Please note my naming convention: uppercase note names, optional "S" for sharp notes as # is an illegal filename character, and respective note naming convention (2 - 6))

Try them out!

[Guitar Sound Buttons]

One of the great things about Howler JS is that it handles simultaneous audio playing easily, a crucial feature, as our digital guitar is a polyphonic instrument. Each string should provide its corresponding sound if activated; more often than not, guitarists play on multiple strings at one time. Let's see some basic chords.

[Chord Sound Buttons]

And yes, anyone with a sharp pitch, this is the chord progression from Metallica – Nothing else matters, so you are welcome!

**Reading Controller Messages**

First, we need to be able to read keypresses from our instrument, and for that, we will create our controller listener function. This function is only responsible for listening to keypresses that are digits, and each message from our instrument consists of four digits. The first digit is the event, with possible values 0 and 1, released and pressed, respectively. The second and third digits represent our fret value, which may be between 0 and 20. The fourth digit is the string number (1 - 6). On each keypress, the key is appended to the final controller message until the fourth digit, where we can trigger an action. Additionally, any invalid keys or digits out of the boundary of our protocol must be discarded, and the message should be cleared.

The controller state has two properties: the message and the highest fret position, an array of six numbers. We need to store these values because when the user activates a string, multiple fret positions may be started on that string simultaneously. Guitars always play the highest sound on a given string, and our digital guitar should not be an exception. One way should be to set these restrictions on the hardware level and let only the highest notes trigger a message sequence. However, this would also restrict any future endeavour for, let's say, displaying a visual guide or a graphical UI that shows activated notes. Lastly, separation of concerns: the guitar controller is not responsible for the logic of how the messages are used and interpreted, and they should be implemented on the application level.

[Controller Code Snippet]

Each four-digit message invokes our translate function, which keeps our highest fret positions state up to date and triggers either a strum or a fret activation function. These functions are declared when a page loads, so each page may have different functionalities. For instance, a Jam Session plays the activated notes, and a Play Session may have a Guitar Hero-like game that handles controller events differently.

[Translate Messages Code Snippet]

Now we can include our controller listener in the html file.

[Controller Listener Code Snippet]

**Reading the Fretboard**

In our implementation, the guitar does not make any sound effects when a button on its fretboard is activated, as hammer-down or pull-up motions are out of the scope of this prototype. On the other hand, every fretboard event will update the guitar's state; namely, it sets the highest note position on a given string. One option might be to store the six strings as an array of numbers, and each activation or deactivation would compare and refresh the highest note. However, if the user interacts with multiple buttons on a string, any information about positions under the highest notes is lost.

Alternatively, we can store individual string information as arrays of numbers and pop or push them on interaction. In this case, the default value of the string array would be [[0], [0], [0], [0], [0], [0]], where 0 means a guitar string without any notes registered. Similarly, reading the uppermost value can be done by either iterating through the individual strings and finding the highest number or saving a sorted array and referring to its last item.

Lastly, we can initialise a display action on board function that takes the fret, string and the event as arguments. This function will be implemented when displaying our virtual guitar board on the screen. The guitar note names will be calculated relative to the lower E2 string, and each string has an offset to adjust the fret number.

[Guitar Notes Code Snippet]

[Fret Activated Code Snippet]

**Reading the Strums**

Unlike the fretboard inputs, strum events activate audio, so they must first find the uppermost activated note position on the fretboard. In the case of a strum press, all notes on the string must be stopped; otherwise, they must be played. The highest note search is similar to the fretboard's; however, I provided some extra branches for different displaying cases, which will later be discussed.

[Strum Activated Code Snippet]

**Creating Sounds**

As we need to deal with many Howl objects, it is reasonable to structure them into an audio object, where the key is the audio name and the value is the How object itself. The most important properties that Howler accepts are the source (src) path, volume (default 1), HTML5, and preload. HTML5 property should be used for large audio files so you don't have to wait for the complete file to be downloaded and decoded before playing. Preload property automatically begins downloading the audio file when the Howl is defined. If using HTML5 Audio, you can set this to 'metadata' to only preload the file's metadata (to get its duration without downloading the entire file, for example). For the complete list of properties, see [Link: Howler GitHub Documentation].

[Get Audio Code Shippet]

One of the things that we must be careful of is that the function preloading Howler audio needs to be user-initiated. The reason is the current autoplay policy in most major browsers. Web browsers are moving towards stricter autoplay policies to improve the user experience, minimise incentives to install ad blockers, and reduce data consumption on expensive and constrained networks. These changes are intended to provide further playback control to users and benefit publishers with legitimate use cases. [Link: Developer Chorme]

Therefore, if you want to preload your audio variables, do it after a click event.

[Play/Stop Nets Code Snippet]

You can manipulate audio play by simply calling play or stop functions.

**Guitar Interface**

I wanted to create a simple user interface for the digital guitar so users could visually confirm every interaction, and let's be honest, while the audio experience is fun, some visual aid is necessary for these types of applications. The guitar UI must have all the frets in a six by 20 matrix, with six strings representing our physical guitar. Also, we must ensure that our approach is user-friendly; hence, all notes, fret numbers, and strings will also be textually represented. As the DOM representation of a guitar requires hundreds of elements, and these elements must be referred to for highlighting, we could either use specific IDs for references or store our DOM elements in arrays and refer them by the index.

[DOM Storage Code Snippet]

If we want to create a more authentic guitar fretboard, we need to be able to calculate the distances on the frets, just like for the real instrument, where the spaces between frets decrease from the neck to the saddle. These distances are calculated iteratively. If we divide any scale length by the constant 17.817, we will get the distance from the front edge of the nut to the first fret. [Link: Fret Distances] We can subtract this fret distance from the total and move to the next fret until we reach the last one (20).

[FretDistances Code Snippet]

We will create our guitar component with several different types of DOM elements. The usual steps for appending an element are straightforward. Create a child DOM element with an HTML tag name, set its ID, class name or other attributes, get the parent element, and append the child element to the parent. As this step will heavily feature our guitar-building process, I made it a one-liner utility function so that we can focus on the main task.

[Append Code Snippet]

Now, we can create the guitar component by traversing seven rows (the last row is for the fret numbering) and 21 fret columns (again, the last one is the string component).

[Create Guitar Code Snippet]

You can copy the following CSS code to have the same result.

[Guitar CSS Code Snippet]

The result is this simple GUI for our guitar.

[Guitar Interface Image]

Finally, we need to display the user interactions on the fretboard. Our displayActionOnBoard function manages the visual highlighting of the fret and string elements in the digital guitar interface. Given inputs are finger position, string, and strum activation and highlight. We can adjust the appearance of corresponding elements to create different levels of highlights (actively playing, just pressed). The remove highlight parameter can reset the components to their normal state.

[Display Actions Code Snippet]

Let's test out our Guitar interface. The screenshot below shows when the guitar is played on a barre D minor chord.

[Guitar Highlights Image]

Feel free to share your thoughts on potential improvements.

## Validating Dates with JavaScript

Date validation forms a cornerstone of programming tasks beyond just JavaScript applications, and JavaScript provides seemingly uncomplicated means for conducting elementary Date object validations. However, it's crucial to spotlight certain nuances that require attention. The fundamental approach that may resonate with some:

[Basic Date Validation Code Snippet]

The getTime function returns a date object or Not a Number (NaN) according to the date string. One interesting property of NaN is that it is not equal to itself, even when using the strict equality operator (===), hence the false return for invalid dates. Unfortunately, the date overflows on leap years, which may be problematic when validating date strings.

[Date Overflow Code Snippet]

As it turns out, according to MDN, "If you specify a number outside the expected range, the date information in the [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) object is updated accordingly. For example, if the Date object holds June 1st, a dateValue of 40 changes the date to July 10th, while a dateValue of 0 changes the date to the last day of the previous month, May 31st". [Link - 1]. This feature is bad news and means that we may just as well validate our date strings ourselves.

Let's consider in this scenario that we only accept dates in the English date format, which is [DD.MM.YYYY]. We can match our string with a simple RegEx and extract the day, month and year information as numbers.

[Regex Code Snippet]

The last day of each month can be assigned using an array of numbers:

[Day Array Code Snippet]

However, we must consider leap years for February (the second item of the array day[1]). To check if a year is a leap year, divide the year by 4. If it is fully divisible by 4, it is a leap year. For example, 2016 is divisible by 4, a leap year, whereas 2015 is not. However, century years like 300, 700, 1900, and 2000 need to be divided by 400 to check whether they are leap years. [Link - 2]

So, let's extend our code with some additional validation for leap years; also, we can restrict the minimum and maximum for the accepted years if applicable to our scenario.

[Final Solution Code Snippet]

Our date validation does not accept date strings with day overflow and returns false for all invalid dates.

## JavaScript's Sorting – What's Behind the Hood

*[Sorting Image]*

Sorting is a fundamental operation in computer science and programming. JavaScript, a versatile and widely used programming language, offers several ways to sort arrays and data structures. However, there is more to JavaScript sorting that catches the eye at first sight. JavaScript is unique because its array sorting behaviour may vary depending on the browser vendor and version. While many other programming languages, such as Python, Java, or C#, provide fixed and standardised implementations for sorting arrays, JavaScript's sorting behaviour relies on the specific JavaScript engine used by the browser. Let's start with the basics of JavaScript array sorting.

**Sorting Strings**

Let's see what MDN says about sorting arrays with JavaScript. The sort() method of [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) instances sorts the elements of an array [in place](https://en.wikipedia.org/wiki/In-place_algorithm) and returns the reference to the same array, now sorted. The default sort order is ascending, built upon converting the elements into strings and comparing their UTF-16 code unit value sequences. [Link: MND]

*[Sorting Numbers Code Snippet]*

A critical aspect of the example above is that sorting mutates our original array. When an array is sorted in place, the elements are reordered within the memory locations occupied by the original elements. In-place sorting can be more memory-efficient, especially for large arrays, as it doesn't require allocating memory for a separate sorted array. However, as the sort function returns with the reference of the original array, it can cause unwanted side effects. If we want to keep the original data, we may use the ES6 spread operator to destruct our array. (Deeply nested objects may need some extra tending, but that is for another post.)

*[Sorting Mutation Code Snippet]*

Another thing to remember is that default sorting is case-sensitive. When strings are being compared, they are converted to their equivalent **Unicode value**, unsurprisingly numbers, then sorted sequentially, in ascending order by default. [Link: DigitalOcean] The order is as follows: numbers, upper-case characters and lowercase characters.

*[Sorting Characters Code Snippet]*

**Sorting Numbers**

But what happens if we want to sort numbers instead of strings?

*[Sorting Numbers Code Snippet]*

Knowing that JavaScript uses type conversion, it is unsurprising that our numbers are arranged in a string-like ascending order. Just like string array items ["B", "AA", "A"] are sorted into ["A", "AA", "B"], arrays containing numbers [2, 11, 1] will be sorted into [1, 11, 2]. Fortunately, the sort method also accepts an optional parameter for comparing array items. [Link: HubSpot] The optional comparison function can be used to fine-tune our sorting logic.

There are four ways to call a JavaScript sort method.

1. Functionless,
2. Comparison Function,
3. Inline Comparison Function,
4. Arrow Comparison Function.

*[Sorting Ways Code Snippet]*

We can compare numbers with a comparator in any of the previously mentioned methods (except, of course, the functionless sort call).

*[3 Ways of Number Sorting Snippet]*

**Comparators**

According to the ECMAScript specification (2023), the sort must be stable (elements that compare equal must remain in their original order). If the compare function is not undefined, it should be a function that accepts two arguments, x and y and returns a negative number if x < y, a positive number if x > y, or a zero otherwise. [Link: ECMAScript Spec] This means that comparator functions are highly customisable, and we can use them for a broad range of scenarios.

*[Comparator Code Snippet]*

**Sorting in Descending Order**

There are two simple ways to sort numbers in descending order. The most straightforward way is to reverse the sorted array using the array prototype reverse function.

*[Reverse Version 1 Code Snippet]*

Or we can modify our comparator function in the following way:

*[Reverse Version 2 Code Snippet]*

In most cases, it does not matter which method we use. Reversing the sorted array is an easy-to-read solution, but the second version may fit better for performance-sensitive applications with large arrays. While the complexity of both of our sorting functions is implementation-dependent, likely O (n log (n)), the reverse has an additional O (n) complexity.

**Sorting Objects**

Comparator functions offer much greater versatility than merely using them for descending sorting. One of the most common use cases for more complex sorts in web development is related to sorting objects by one or more of their property value. Typical scenarios include sorting users by their registration date, organising products by price, or arranging posts by their publication timestamps. Let's consider a simple scenario where users are ordered by their name:

*[Users Data Code Snippet]*

We can simply refer to the object property values in our comparison function to sort users by name.

*[Sorting by Name Code Snippet]*

However, please beware of the string comparison because we cannot rely on the subtraction operator, like "A" – "B" returns NaN. To compare strings reliably, we must use the localCompare() function. The localeCompare() method of [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String) values returns a number indicating whether this string comes before, or after, or is the same as the given string in sort order. *[Link: MND]* Additionally, we can fine-tune our comparison by specifying the language, base for accents and cases, or setting numeric options to correct the already mentioned comparison problem. See details here: [Link: MND]

*[Sorting Options Code Snippet]*

Lastly, we can sort our users by dates, for example, their date of birth. In this case, we compare the returned number when creating a new Date instance.

*[Sorting by Date Code Snippet]*

**Edge Cases**

Some edge cases have already been mentioned, like the lexicographic sorting of numbers, language-specific issues like accents, and case sensitivity. However, I want to call your attention to some JavaScript-specific values we must account for to create reliable code. Some falsy values, such as 0, false, null and undefined, may behave unexpectedly. For instance, NaN stayed in its original place at the beginning and the end of a numeric sorting, but it was ordered lexicographically in string conversion.

*[Edge Cases Code Snippet]*

**Implementation Freedom**

As I have written in the introduction, browsers have no restrictions on how they implement their JavaScript sort function as long as they adhere to the ECMAScript specifications. This, of course, leaves us guessing what exactly is going on behind the scenes, and as far as we are concerned, they might use Bubble sort. But jokes apart, most browser sort implementations are highly optimised. Historically, Array prototype sort and Typed Array prototype sort relied on the same Quicksort implementation written in JavaScript. The basis is a Quicksort with an Insertion Sort fall-back for shorter arrays (length < 10). *[Link V8]*

However, while Insertion sort is a stable algorithm, Quicksort is not, and sort stability (two objects with equal keys appear in the same order in sorted output as they appear in the input data set *[Link: GeeksForGeeks]*) was a long-desired feature of the JavaScript community. Therefore, the latest browser implementations, such as Chrome's V8 engine, use TimSort.

**TimSort**

[Tim Sort Cheat Sheet Image]

TimSort is a stable hybrid algorithm combining Merge and Insertion sort, and it takes advantage of the fact that real-world data is often partially sorted. TimSort's time complexity is recorded at O(n log (n)), making its average time complexity equal to that of Quicksort and Mergesort; in best-case scenarios, whether negligible or not, TimSort will typically outperform both Quicksort and Mergesort in time complexity, though it is arguably relegated to cases where data is considered nearly sorted given TimSort's stable characteristics. *[Link Medium]*

First, the algorithm divides our array into groups called runs. The size of the runs is typically the power of two (32 or 64) for optimising Merge Sort's recursive tree. Initially, Insertion Sort will sort the runs, as it performs when the size of the given array is relatively small. When the runs are sorted, we can merge them two by two iteratively until every run is merged.

*[TimSort Code Snippet]*

In summary, JavaScript sorting is a versatile tool in web development but comes with nuances. Be aware of browser-specific variations and consider using custom comparator functions for sorting objects. Watch out for edge cases, such as handling falsy values and language-specific sorting challenges. Modern JavaScript engines use efficient algorithms like TimSort for optimisation.

## Green Rooftop - An Arduino IoT Project

*[Intro Image]*

In this hands-on tutorial, you'll embark on a journey to design and create an intelligent rooftop garden system that can monitor environmental parameters such as temperature, humidity, moisture, and light. We will create a user-friendly interface featuring buttons, LCD, and a buzzer for easy interaction. This project offers an ideal learning experience for those looking to combine Arduino with an API, as we will also create a simple NodeJs server and a website. Let's begin this educational adventure with a greener, more sustainable future in mind.

**Green Rooftops and IoT**

IoT has the potential to play a significant role in helping to address environmental challenges. By leveraging IoT devices and technologies, it is possible to gather data about various environmental factors such as air and water quality, weather patterns, and biodiversity. This data can then inform and improve conservation efforts, monitor and mitigate pollution, and support sustainable agriculture and energy practices. Through the collection and analysis of real-time data, IoT can help optimise resource use, reduce waste and pollution, and support the transition to a more sustainable and circular economy *[Link: WEF]*

One of the main benefits of our application is that we empower urban environment maintenance services to monitor and maintain their contractors' green roofs or terraces centrally. Monitoring may help better organise their workforce, predicting foreseeable labour and material requirements, and combined with meteorological data, can fine-tune the whole operation of environmental businesses.

*[Concept Image]*

**Microcontroller Selection**

Although reading analogue and digital sensors can be done with a basic Arduino Uno, it may not be suitable for this project because most basic microcontroller devices do not have a built-in WiFi module. Even if WiFi modules can be purchased separately, built-in WiFi connectivity is one of the most significant factors in our microcontroller selection. It reduces complexity and frees up physical space, which may be a deal breaker, considering the number of input and output components we use.

Several microcontroller options are available with built-in WiFi capabilities, like the Arduino MKR WiFi-1000 and WiFi Rev-2, Ublox Nina-W10 and W13 modules or the ESP8266 and ESP32. However, the Arduino-based boards will enjoy preference over other options because of extensive Arduino documentation and support. For example, the Arduino Rev-2 is an ideal choice as it has a header pinout that supports jumper cables. And because we might need to experiment extensively to make every component work according to specs, header pinouts make circuit design and tests comfortable.

*[Microcontroller Image]*

**Inputs**

Grove Soil Moisture Sensor (SKU-101020008) is a capacitive sensor that evaluates the soil moisture level by measuring resistance. It "*consists of two probes that allow the current to pass through the soil and then obtain resistance values to measure soil moisture content*" *[Link: Seeed]*. However, it is essential to notice that the values read by these sensors are not proportional to the full range of analogue inputs (0-1023) as the output of the component voltage is limited to 1-3V.

*[Moisture Sensor Image]*

Grove Light Sensor is an analogue module that reads light levels with high accuracy, using a highly sensitive photo-triode.

*[Light Sensor Image]*

Grove Humidity and Temperature Sensor Pro can simultaneously measure the temperature and humidity of the surrounding environment. It is a digital sensor that utilises a single-wire communication protocol and is designed to be highly compatible. The sensor is based on the DHT-22 sensor, a high-performance, reliable sensor for measuring temperature and humidity. The sensor has a temperature measurement range of -40-80°C and 0-99% for humidity. Additionally, DHT-22 has a high accuracy of ±2°C for temperature and ±0.5% for humidity.

*[DHT Image]*

Gravity Photoelectric Water/Liquid Level Sensor: Most liquid-level sensors must be submerged into the liquid to work, unlike Gravity, as it operates on optical principles and lacks mechanical parts that need calibration or maintenance. As a result, it is well suited to applications with an exposal of high pressures or volatile temperatures and can be fitted in several ways.

*[Liquid Level Sensor Image]*

Lastly, three momentary tactile switches will be used for the device user inputs. The left and right buttons will step between menu options, and the select button will either activate a menu option or enter a submenu.

*[User IO Image]*

**Outputs**

LED: Two Light Emitting Diodes will be used, one indicating a successful WiFi connection and one when the device communicates with the server; as a good practice, resistors will be connected to the anode to extend the LEDs' lifetime.

*[LED Image]*

LCD Display (HD-44780): This Liquid Crystal Display is a fundamental part of many Arduino sets, as these are relatively easy to integrate into any project. The display has two rows, each with 16 characters of display space. In addition, the backlight of the component can be set through its V0 pin, which will come in handy for adjusting the display's contrast through an analogue pin.

*[LCD Image]*

Buzzers are great economical ways to extend our user interface, providing additional feedback about actions for our customers. The only complexity of applying audio feedback is to make it optional, as no UI should be delivered with compulsory audio functionality. The Piezo buzzer is a multi-tone audio signalling device that vibrates piezoelectric crystals to produce audio output.

*[Buzzer Image]*

**Protocols**

We will use WiFi to connect our devices to the network, as it may cover sufficient area (using 2.5GHz may reach about 50 meters). Alternatively, GSM cellular connection may be used for longer distances, which is not essential for our prototype implementation. We will use HTTPS for secure communication with our backend server, and our RESTful NodeJS API will listen to the HTTP requests. In exchange, every request will return a response in JSON (JavaScript Object Notation), including errors. The benefits of using JSON over plain text requests are that it is well structured, ready to use after parsing, and platform-independent. Additionally, Restful APIs offer a standardised way of communicating with our server using simple HTML verbs (methods).

Our server has to be able to send notification emails to our clients, so we will use SMTP to send our emails using NodeMailer (naturally, our client will use POP3 to retrieve our emails). Lastly, internal Arduino components may use protocols abstracted by libraries, like DHT22 uses I2C communication.

*[Protocols Image]*

**Design**

In our scenario, the design must be approached from multiple angles, as the project incorporates the IoT device's physical arrangement, communication to the server, and a software platform. Our main goal is for users to read specific device values in person or online and for the service team to monitor device activity through the application to make decisions about maintenance.

*[Use Case Diagram]*

The prototype will not include login procedures as we have one device; however, in the commercial product, we must ensure that users can access only their data while the maintenance team can read all devices' information.

*[Sequence Diagram]*

Our IoT device has four main steps to achieve a successful reporting functionality. First, we boot our device by loading saved settings, initialising our pin layout and connecting to WiFi. Then, the main loop only checks the elapsed time and calls a reporting procedure. At the same time, the LCD and the connection/report LEDs continuously inform the user about the ongoing processes.

*[Activity Diagram]*

As most of our sensors work with analogue pins, we have just enough digital pins to accommodate our LCD, LEDs and buttons without additional support. The electrical layout of the device can be divided into four different units: the microcontroller, control panel components with LEDs and input buttons, outputs, such as LCD and speaker, and sensors.

*[Electric Design]*

**Modelling**

Our project is an outdoor device, so we must create an enclosure to protect the electronic components from the elements. We will use Polystyrene sheets as our building materials, as they are suitable for outdoor use; however, our commercial devices should be manufactured with more eco-friendly materials such as carbon fibre sheets. Our design must be compact yet provide enough space for our components, and removing the bottom screws should make it serviceable.

*[Modelling Image]*

**User Interface**

The users can communicate through the menu by manipulating three buttons (Left, Right, and Select). The menu is non-iterative; pressing right on the last item will not go to the first one. The select button can refresh a sensor reading or enter a submenu. The following is the design of our device's interface:

*[Menu Image]*

**Backend Design**

Even though only one prototype will be built, we may design our API to be scalable because the commercial solution will have multiple users and devices. Therefore, apart from the reporting API routes, we'll have an index route for checking connectivity, a device route for registering instruments, and users for the application. We'll use HTML methods aligning RESTful conventions:

*[RESTFul Image]*

Following our route plan, we can create a simple database schema with three tables: Users, Devices and Reports. Then, we can work with an initial schema using MongoDB. Adding properties is not restricted in NoSQL databases, and a commercial version may extend it by adding login and profile details.

[ER Image]

**Frontend Design**

Lastly, we may create our application design with mobile devices in focus. Because responsive design is out of our current scope, only mobile sizes will be developed (<720 pixels). Our specifications require two pages, one for reading the latest device information and one for listing historical measurements.

[Wireframe Image]

**Manufacturing**

First, the polystyrene sheet was cut to size, drilled and pasted together according to our design plan, and then the cover received a matte black painting.

*[Plastic Components Image]*

*[Electric Housing Image]*

Next, the Arduino was fixed to a central place and connected to the USB. The sensors required extensive wiring; therefore, we used universal printed circuit boards (PCBs) to join the sensors to VCC and ground wires to reduce cabling.

*[Assembly 1 Image]*

*[Assembly 2 Image]*

Lastly, the LCD, LEDs and buttons were assembled, with a second universal PCB for VCC and ground. Unfortunately, the second PCB made our wiring messy, and for a commercial device, we should use a custom-designed PCB to minimise wiring complexity.

*[Assembly 3 Image]*

*[Assembly 4 Image]*

The assembled product is now ready for further development.

*[Final Product Image]*

**Sensor Readings and Menu**

You can download the complete project code from here, as I my code snippets do not cover pin layouts, and some fundamental setups. *[Link: download controller.ino]*

After finalising our wiring, we can continue developing by reading sensors. But first, we must set the pin modes according to the pin layout and create a function responsible for assigning sensor values to globally available variables. Analogue sensor values range from 0–1023, so some readings must be turned into percentages.

*[Reading Sensors Code Snippets]*

We will use LCD cursor settings and printing extensively in our nested menu. A print LCD subroutine that encapsulates these functionalities is easier to reuse. Additionally, we can add a function responsible for text content centring.

*[LCD Printing Code Snippet]*

Finally, we can listen to button presses. Left, right and select buttons will receive their handling functions, and they may stop incrementing/decrementing the current index to stay within the menu (or submenu) boundaries or trigger an appropriate action. The menu state will be stored in an array, and each item will represent a submenu.

*[Handle Button Press Code Snippet]*

**Report Intervals**

In a successful business, thousands of devices might communicate with our servers. Even if only a subset of them are set to a fixed time reporting, for instance, 4 PM, servers may be overwhelmed if all requests come in simultaneously. However, we may calculate the elapsed time from the devices' start to disperse possible floods of POST requests.

*[Intervals Code Snippet]*

**WiFi Connection**

We must establish a WiFi connection to send our reports to the server. This connection may be set from start-up and must be double-checked before server communication. One of the simplest ways to connect our Arduino is to use the WIFININA library. Our algorithm will be based on our earlier the diagram.

*[WiFi Code Snippet]*

**Post Requests**

Finally, we may be able to connect to our server and send our sensor reports to our database. We can use the ArduinoHTTPClient library for HTTP communication and create a WiFiSSLClient. First, we check our WiFi status and build a payload string for sensor values. As this library had difficulty correctly sending our data in the request body, we used the query string as a workaround. Additionally, our WiFi password and SSID are hardcoded for now, but the commercial application should have alternatives to set these values dynamically.

*[POST Request Code Snippet]*

**Request Routes**

After completing our POST request from the device, we may consume those requests from the server side. We connect to our database with a connection string and listen to incoming requests. We modularise our schemas and route handlers into index, users, devices and reports. After registering a mock user and a device for testing, we can validate and save sensor data on our database.

*[Post Route Code Snippet]*

**Email Alert**

Our report route is also responsible for sending alert emails to the users' email addresses if specific values are out of boundaries, such as water tank and soil moisture. We register the application to our Google mail server and construct an email template with Nodemailer using HTML content to produce mail services.

*[Email Code Snippet]*

**Frontend**

Lastly, we need to create the application's client side, where the users may access their device's report data. In our presentation prototype, we hard-coded the user id, so we could concentrate on the main object of our code, which is successfully accessing services and visualising reports. However, commercial applications would use login procedures to provide user permissions.

**API Calls**

Several acceptable server request methods exist in JavaScript, like traditional async callbacks, promises (then/catch) or the modern fetch (async/await). Using async-await is beneficial because it reads like synchronous programming, is concise, and is not nested, meaning multiple calls would never end up in a so-called callback hell.

*[Fetch Data Code Shippet]*

**Conclusion**

In conclusion, this project showcases some exciting intersections of environmental monitoring, IoT technology, and sustainable practices. Through this tutorial, you've learned how to create and design an intelligent rooftop garden system project that can monitor environmental parameters.

## Scheduling Emails with Cyclic

*[Intro Image]*

Email scheduling is a valuable tool for optimising communication. It enables you to send emails conveniently, improving productivity and ensuring messages reach their audience when they're most likely to engage. Whether for follow-ups, marketing campaigns, or avoiding clutter in inboxes, it enhances organisation and supports thoughtful, efficient communication.

In NodeJs, scheduling emails can be done effortlessly using npm NodeCron and NodeMailer. NodeMailer is a Node JS module that allows you to send emails from your server easily. It is a zero-dependency module for all [Node JS](https://www.turing.com/kb/node-js-vs-asp-net)-compatible applications. The emails sent can be plain text, attachments, or HTML. *[Link: Turing.com]* Cron is a job scheduling function that allows users to trigger specific tasks at designated times. It is a background process executing non-interactive jobs. *[Link: Airplane.dev]* NodeCron is a cron task scheduler for NodeJs applications.

**Why NodeCron Doesn't Work on Cyclic?**

A simple NodeCron solution looks like the following:

*[NodeCron Code Snippet]*

However, NodeCron relies on a continuous server environment to run scheduled tasks at specific times or intervals. In serverless environments like Cyclic.sh, the server is not continuously running; it's event-driven and only executes code in response to specific events or triggers. This serverless architecture doesn't support traditional cron jobs because there's no persistent server to run them. So, serverless means applications are only on for the time it takes to process individual requests. They are suspended immediately after each response is sent. *[Link: Cycle.sh]* So, your scheduler will silently fail without any logging or error messages.

**The Workaround**

I will present a workaround solution using a simple example scenario. Suppose I want to create an email scheduler that triggers every midnight and sends an email to a specific email address with some essential information, for instance, a daily breakdown of website activity statistics. I will use NodeMailer for my email-sending function, so let's install it:

*[NPM Install Code Snipppet]*

We define a function called sendEmail that sends an email message. We then set up the sender's and recipient's email addresses, email subject, and the email's content (HTML format in this case). We also have to configure the email server settings, including authentication and secure communication, using TLS, specifically for Gmail in this example. Our sendEmail is asynchronous and returns a promise that resolves to information about the sent email if successful or an error if there's an issue.

*[Send Email Code Snippet]*

Now that we have an email sender function, we may set up an API endpoint to POST emails. However, as I mentioned, this endpoint is not automated with NodeJS.

*[API Endpoint Code Snippet]*

**Set Up a Cron Job on Cyclic**

Now, I assume that you are registered and already have a Git Project hosted by Cyclic, but in case you need some help with the basic repository deployment, please check out this link. *[Link: Cyclic]* Go to your project and click the Cron menu option.

*[Cron Menu Option Image]*

Create a new cron task:

*[New Cron Task Image]*

Set the method to POST and the API endpoint path you defined for your email scheduling. You can choose between running once or schedule options, so if you want to schedule your emailer for daily reports, select the schedule option.

*[Edit Cron Task Image]*

Lastly, set the schedule time using cron expressions. Cron Expressions are, in most cases, a string of five fields, which make up the building blocks of every Cron expression. Each field represents a specific time unit, so the different fields represent minutes, hours, days of the month, months, and days of the week, respectively *[Link: CronToGo]*

[Cron Syntax Image]

To set our schedule to midnight that repeats every day, we can use the "0 0 \* \* \*" expression, where the first 0 means the minute, the second the hour, and the last three values are left as wildcards. After saving the cron job, you should see your cron task and the next scheduled run time.

*[Final Cron Task Image]*

## Brief Anatomy of React

This article is an extract of my research on React fundamentals. After tackling some React interview questions, I realised that my theoretical understanding had gaps that needed an immediate fix. As a result of this research, here are some of my findings that may prove helpful in a React interview and definitely will deepen your confidence as a developer if you are at the beginning of your journey to mastery.

**Library or Framework**

React.js is a free, open-source frontend JavaScript library extensively used for designing web UI interfaces. However, React is often mentioned as a framework in statistics and job listings. Framework and library as terminologies may be interchangeable in certain circumstances, as both refer to reusable code that simplifies complex functionalities.

**Library**

A library is a collection of pre-defined methods and classes developers can use to ease their work and accelerate development. *[Link: Sencha]* Additionally, libraries often target specific functionalities of an application.

**Framework**

A framework gives the fundamental structure while indicating the required customisation from the coder. The framework defines the workflow of a software application, informs the developer of what he needs, and invokes the developer's code when necessary.

React lacks the completeness or restrictions of large ecosystems, so it is more commonly considered a library rather than a framework. Even though it is often listed as one of the most popular web development frameworks, it relies on third-party libraries for many features, like routing or server-side rendering (SSR), otherwise provided by other frameworks, like Next.js. One great advantage of React.js is that it offers freedom for structuring and customising our application architecture.

**DOM vs Virtual DOM**

To better understand how React renders a UI in the browser, we need to distinguish between DOM and Virtual DOM.

**DOM**

The Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents. The DOM is a structural representation of the web document as nodes and objects, which defines the logical structure of documents and how a UI document is accessed and manipulated.

Normally, whenever a user requests a webpage, the browser receives an HTML document for that page from the server. The browser then constructs a logical, tree-like structure from the HTML to show the user the requested page in the client. *[Link: LogRocket]*

With the DOM, programmers can build documents, navigate their structure, and add, modify, or delete elements and content. Anything found in an HTML or XML document can be accessed, changed, deleted, or added using the DOM. *[Link: W3]* The DOM represents the HTML page as a tree of elements called nodes and serves as an interface for the browser to enable interaction with JavaScript. The document is the root node, and the child nodes form the subtree.

*[DOM Image]*

**Virtual DOM**

The virtual DOM (VDOM) is a programming concept where an ideal, or "virtual", representation of a UI is kept in memory and synced with the "real" DOM by a library such as ReactDOM. As the name implies, the virtual DOM is a much lighter in-memory replica of the actual DOM in the form of objects. This process is called [reconciliation](https://legacy.reactjs.org/docs/reconciliation.html). *[Link: ReactJS]*

When a change occurs in a real DOM element, the DOM has to re-render not only that element but also all of its child elements. This traditional approach can be slow and inefficient, especially in complex web applications with high interactivity and frequent state changes. In contrast, React adopts the virtual DOM concept during the rendering process, aligning with its declarative approach: you tell React what state you want the UI to be in, and it makes sure the DOM matches that state. With this method, developers can specify the desired UI state, and React takes care of the rest.

After updating the state, React compares it to a previous snapshot, identifying the exact element that changed. It then optimises performance by updating only that specific element in the real DOM. We will delve deeper into this optimisation technique shortly. By abstracting manual DOM manipulations, the virtual DOM simplifies the development process, allowing developers to focus on creating components and writing code that is more predictable and less error-prone.

Thanks to the virtual DOM, developers are relieved from the burden of managing state transitions. Once the state is updated, React ensures that the DOM reflects that updated state seamlessly. The only disadvantage of VDOM is the higher memory usage, as the diffing algorithms need to keep comparing the elements to know which components need to be updated or changed.

**Rendering Components**

In React, the rendering occurs when a React component's render method is called. This method returns a representation of the user interface as a Virtual DOM (a lightweight copy of the actual DOM). React then compares this Virtual DOM to the previous one, identifying the differences or updates needed. These updates are calculated efficiently, and only the necessary changes are applied to the HTML DOM. This process, called reconciliation, is a key part of what makes React efficient and performant. It ensures the application's UI stays in sync with its underlying data, providing a smooth and responsive user experience.

**How Reconciliation Works**

The reconciliation algorithm compares the current virtual DOM tree to the updated virtual DOM tree. It makes the minimum number of changes necessary to bring the virtual DOM in line with the updated state. *[Link: Medium]*

**Element in different types:** Whenever the type of the element changes in the root, react will scrap the old tree and build a new one i.e a full rebuild of the tree.

**Elements of the same type:** When the type of changed element is the same, React checks for attributes of both versions and then only updates the node that has changes without any changes in the tree. The component will be updated in the next lifecycle call.

**Batching:** React batches multiple changes into a single update, reducing the number of updates to the virtual DOM and, in turn, the real DOM.

**Reconciliation Algorithms**

React Fiber is just a plain JS object with properties, and since React 16, Fiber is the default reconciler. Fiber focuses on animations and responsiveness by chunking and prioritising tasks. It can reuse, abort, pause and resume work processes; unlike previous versions, it is asynchronous. The older versions of reconcilers worked like a stack, hence the name Stack Reconciler. As this reconciler worked synchronously, it had some significant flaws; to re-render the stack, it needed to be empty.

**Stack Reconciler**

A good example of when Stack Reconciling might become problematic is when we want to render something, and there are unfinished processes. Let's see the steps:

**Initialisation**: When the ReactDOM.render() function is called for the first time, we will create a virtual DOM by iterating the first element, usually the <App /> component.

**Rendering**: While creating VDOM, we will make the corresponding DOM nodes and append the element to its parent.

**State Change**: when calling the setState() function, it marks the VDOM as dirty, passing the execution to the reconciler.

**Reconciliation:** The reconcile() function accepts the current node as its parameter and recursively reconciles every child of the current node tree to detect changes and update the real DOM.

**Align DOM and VDOM:** If the current node has changed, we align (modify, add, delete) its attributes to both DOMs (virtual and real). Additionally, shouldComponentUpdate(), React.memo(), and PureComponent checks will be handled in this phase.

**Traverse Nodes**: Iterate through every child of the current node and reconcile them.

*[Image: Stack Reconciliation]*

Consider a web application with a long list of items, such as a product catalogue, where each item corresponds to a list element. When a user interacts with the application, the following issues can arise:

**UI Responsiveness**: If each list item takes significant time to reconcile and render, the main thread will be occupied with these rendering tasks, leading to noticeable delays in responding to user interactions, such as typing or clicking.

**Blocking User Input**: While the main thread is busy reconciling and rendering items, it may not respond promptly to user input, resulting in a frustrating user experience, as users expect real-time or near-real-time responsiveness to their actions.

**High-Priority Updates**: In scenarios with critical, high-priority updates that users need to see immediately (e.g., error messages, alerts, or dynamic search results), the stack reconciliation process can cause delays in rendering these updates.

**Fiber Reconciler**

While Fiber has significant performance improvements, it also makes React development easier. Every Fiber is a plain old JavaScript object, and a Fiber also represents a unit of work. React processes Fibers ending with finished works and commits them by rendering them to the visible DOM. This process has two phases: rendering and committing. In the render phase, React does all kinds of asynchronous tasks, which can be prioritised, paused or discarded. The commit phase is a synchronous phase that cannot be interrupted.

Every Fiber has a one-to-one relationship to things, like components or DOM nodes. The type of Fiber is stored in the tag property, which accepts a number between 0 and 24 and represents a type, like function component, class component, fragment, etc. The state node property represents the element's reference to access the state associated with the Fiber.

The difference between Fiber and React elements is that React elements are continuously recreated, while Fibers are primarily created during the initial mount and constantly reused. Fibers form a tree structure, and the child, sibling and return properties handle the relationship between Fibres. In a Fiber, the child refers to the first child, all other elements will be connected through the sibling property, and the return is the reference to the parent.

*[Fiber Image]*

Fibers also represent a unit of work, and these works can be the following: state changes, lifecycle functions, DOM changes, and react can handle these works directly or scheduled. Using time slicing, React can split work into chunks, and high-priority works, such as user interaction or animations, can be scheduled directly using the requestAnimationFrame() function. Low-priority works, like a network request, can be delayed and scheduled with the requestIdleCallback for an idle period.

In fact, two trees are used in the reconciliation process: the current tree that is visible on the screen and the work-in-progress tree. The Fiber alternate property refers to the Fiber pair on the work-in-progress tree to help reuse Fibers. React steps into a Fiber with the beginWork() function and keeps stepping in children until there are none, calling the completeWork() function. After React completes the final or root Fiber, the commitWork() function is called, and the changes are flushed to the DOM. There is only one child per node, so the traversal is not recursive but uses a while loop.

**Batching**

React can face challenges in keeping pace with user inputs or actions, mainly when several state updates are in play. For instance, when a component needs to update two or more state variables in response to user interactions, React may initiate multiple re-renders, potentially leading to a less-than-smooth or sluggish user experience. *[Link: Shevchuk]*

Before React 18, not all state updates were batched, though. For example, state updates using asynchronous code (e.g. Promise) or third-party APIs (e.g. setTimeout) weren't batched and therefore triggered two re-renderings (for two respective state updates) of the component. However, with React's additions in React 18, [automatic batching](https://github.com/reactwg/react-18/discussions/21) became the default. If there are situations where a React developer would want to opt out of batching, one could use the flushSync top-level API of React. *[Link: Wieruch]*

**JSX**

JavaScript XML (JSX) is a syntax extension for JavaScript often used with the React library. It allows you to write HTML-like code within JavaScript files. JSX provides a more concise and readable way to describe the structure of user interfaces. React components use JSX to define the user interface's layout, structure, and appearance by creating a hierarchy of virtual DOM elements that are later rendered to the actual DOM. JSX code is transpiled into regular JavaScript using tools like Babel, making it compatible with web browsers. This approach simplifies the creation of dynamic and interactive web applications, making it easier for developers to work with the React library.

Fundamentally, JSX provides syntactic sugar for the React.createElement(component, props, ...children) function. React components use capitalised JSX tags, and native, built-in components, such as a div, must be lowercase. Also, JSX accepts dot notation; therefore, MyComponents.ColourPicker is a valid JSX syntax and can be used to modularise complex components. A closing tag can also be applied if a JSX component has no children. Note that JSX must return a single root element, and multiple root elements must be wrapped with a container component or a fragment.

JSX also accepts any JavaScript expression to assign dynamic properties using bracket notation. The default property value is true if only the property name is applied. In JSX, JavaScript expressions enclosed in curly braces can evaluate to strings, React elements, or lists of these values, providing a versatile way to define the structure and content of React components. Additionally, the props.children prop works like any other prop, allowing the passing of various data types, not limited to what React typically renders. For instance, custom components can receive callback functions or custom JSX structures as children, adding flexibility and composability to React applications.

**Babel and JSX Transpilation**

Babel is a JavaScript compiler that is often used in React development. It is crucial in transforming modern JavaScript code, including JSX, into versions compatible with older browsers and environments. Babel is essential in the React ecosystem for the following reasons:

**1. JSX Transformation**: Babel is used to transpile JSX code, allowing developers to write JSX syntax in their React components. It converts JSX into standard JavaScript code that browsers can understand.

2. **ES6/ESNext Support**: Babel enables modern JavaScript features (ECMAScript 6 and beyond) in React applications. It transforms code into ES5 or other compatible versions, ensuring broad browser compatibility.

3. **Optimising Code**: Babel can optimise and minimise the code, reducing its size and improving application performance.

4. **Plugin Ecosystem**: Babel has a rich ecosystem of plugins that can be added to tailor the transformation process to specific project requirements. Developers can customise Babel's behaviour with plugins and presets.

**Class vs Functional Components**

We can declare React components in two different ways, using classes or functions. Functional components are some of the more common components encountered while working in React. These are simple JavaScript functions that return JSX. A class component is a JavaScript class that extends React.Component which has a render method.

*[Functional vs Class Syntax Code Snippet]*

A class component requires you to import and extend the Component class from React.Component and create a render function which returns a React element. Using classes results in additional boilerplate injected into the output compared to the function example. Stateless components are the simplest components in the React ecosystem; however, if we want to use functional components statefully, we must use hooks. Transpiling the hook-enabled function component adds significantly less boilerplate to the transpiled code.

**Hooks**

With React Hooks, we can use state and other React features in a functional component. It empowers developers to do functional programming in React. *[Link: FCC]* React Hooks let you "hook into" state and lifecycle features in functional components. They were introduced in React 16.8 to provide a way to use state, context, refs, and other React features without the need for class components. React Hooks offer several built-in functions, each designed for a specific purpose:

useState: Allows functional components to manage local state. It takes an initial state value and returns an array with the current state and a function to update it.

useEffect: Enables functional components to perform side effects, like data fetching or DOM manipulation, after rendering. It replaces lifecycle methods like componentDidMount and componentDidUpdate.

useContext: Provides access to the context API, allowing components to consume context values.

useRef: Creates a mutable ref object that can be used to directly access and interact with DOM elements or store mutable values.

useReducer: Provides an alternative to useState for managing more complex state logic using a reducer function.

useMemo and useCallback: Memorisation functions that optimise performance by memoising values and functions to prevent unnecessary re-renders.

Hooks allow functional components to maintain their state, side effects, and other features previously only available in class components. They make it easier to reuse stateful logic and encourage the development of smaller, more focused components, leading to cleaner, more maintainable code.

**Component Lifecycle Methods**

In React class components, there are several lifecycle methods that you can override to add functionality to your components at different stages of their existence. Here is a list of the main React lifecycle methods:

**componentDidMount**: This method is called after the component is inserted into the DOM. It is often used for initiating AJAX requests, adding event listeners, or setting up timers.

**componentDidUpdate**: This method is called after a component's updates are flushed to the DOM. It helps handle updates or perform actions after a re-render.

**componentWillUnmount**: This method is called just before the component is removed from the DOM. It's typically used for cleanup, such as removing event listeners or cancelling timers.

**shouldComponentUpdate**: This method allows you to control whether the component should re-render. It returns a Boolean value that determines whether the re-render should proceed.

In a React class component, the order in which the lifecycle methods are called follows a specific sequence. Here is the typical order of component lifecycle methods:

1. **Constructor**: This method is called when an instance of the component is created. It's used for initialising state and setting up initial values.

2. **static getDerivedStateFromProps**: This is a static method that's called before `render`. It allows you to update the component's state based on changes in props.

3. **render**: This is the core method for rendering the component's UI. It should be a pure function without side effects.

4. **componentDidMount**: This method is called after the component is inserted into the DOM. It's often used for initiating AJAX requests, adding event listeners, or setting up timers.

5. **shouldComponentUpdate**: This method is called before re-rendering to determine whether the component should update. The subsequent lifecycle methods will not be invoked if it returns' false'.

6. **render**: If `shouldComponentUpdate` allows the update, `render` is called again to re-render the component's UI.

7. **componentDidUpdate**: This method is called after the component's updates are flushed to the DOM. It helps handle updates or perform actions after a re-render.

8. **componentWillUnmount**: This method is called just before the component is removed from the DOM. It's typically used for cleanup, such as removing event listeners or cancelling timers.

*[Component Lifecycle Image]*

It's important to note that with the introduction of function components and React Hooks, these lifecycle methods have evolved. Function components rely on the `useEffect` hook for handling side effects and state management. If you're using function components, the order and usage of lifecycle methods are quite different from class components.

**Synthetic Events**

JavaScript events serve as the means for users to engage with a web application, enabling actions like capturing clicks, focus, mouseover, and keypress interactions when they occur within the web browser. Each JavaScript event is paired with an event handler, which collaborates with an event listener. The event listener's role is to observe a specific event's occurrence, while the event handler is a function containing executable code that triggers once the event is detected or activated.

React Synthetic Events are very similar to Native Events; however, with Synthetic Events, the same API interface is implemented across multiple browsers. *[Link: Britto, V]* Both Synthetic Events and Native Events can implement the preventDefault and stopPropagation methods. Synthetic events are **cross-browser**, as they wrap the browser’s native event through the nativeEvent attribute and provide a uniform API and consistent behaviour. Additionally, they perform better than native events as synthetic events are [delegated](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events#event_delegation) to document through [bubbling](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events#event_bubbling).

**useCallback and useMemo**

Both useCallback and useMemo are hooks in React that help optimise performance by memoising values and functions. They are often used when you want to avoid unnecessary calculations or re-renders. UseCallback is used to memorise functions. It returns a memorised version of the function that only changes if one of the dependencies passed to it has changed. UseCallback is particularly useful when you pass functions as props to child components to prevent unnecessary re-renders of those components. We must remember that every time a component re-renders, its functions get recreated. Because of this, our function has changed because of the referential equality.

*[UseCallback Code Snippet]*

UseMemo is used to memoize values. It takes a function and an array of dependencies. It returns a memoised value recalculated when a dependency has changed; it is useful when you want to optimise expensive calculations in terms of time or computation.

*[UseMemo Code Snippet]*

The useCallback and useMemo Hooks are similar. The main difference is that useMemo returns a memoised value and useCallback returns a memoised function. [Link: w3school]

**Context API**

Effectively handling state is a fundamental aspect of building applications with React. One prevalent method of state management involves passing props between components. However, when you find yourself repeatedly passing the same data to numerous components or dealing with components scattered across your application, it can become cumbersome and potentially lead to performance issues, making your application less efficient and more challenging to maintain. Passing props through multiple levels of child components in React is called prop drilling or prop threading. Context API allows data to be passed through a component tree without having to pass props manually at every level. This makes it easier to share data between components. *[Link: Boateng, D]*

You start by creating a Context using the React.createContext() function. This function returns an object with two properties: Provider and Consumer. The Provider component wraps the part of your component tree where you want to make certain data available. It accepts a value prop: the data you want to share. The Consumer component is used to access the data provided by the Provider. It can be placed in any component that needs access to the data, and it will automatically subscribe to updates when the data changes.

**Conclusion**

In this journey through React, we've explored the core concepts of rendering, delving into the virtual DOM, reconciliation, component lifecycles and JSX. We've also uncovered the power of React Hooks and introduced some state management techniques like the Context API. This article is far from finished, and I will extend its content as I go along on my personal learning journey. I hope you found it useful.

## Git Cheat Sheet

Before the advent of Version Control Systems (VCS), the traditional method involved the laborious practice of manually saving complete project copies in assorted folders (a common occurrence prior to Git's introduction in 2005). This approach quickly became impractical, particularly when multiple individuals collaborated on a single project, necessitating the manual exchange and integration of changes.

**What is Git**

Git stands as the most widely embraced version control system globally. Version control systems are designed to track the changes in our code across time and maintain a dedicated database about the changes in a so-called repository. Within this repository, we gain insight into the historical evolution of our projects, identifying the contributors behind each modification, along with the timing and rationale for those adjustments. Furthermore, in the event of an error, we can restore our project to a prior state.

**Categories of Versioned Systems**

**Centralised:** In a centralised version control system, every team member relies on a central server to access the most recent project copy and to exchange modifications. Systems like Subversion and Microsoft Team Foundation Server are examples of this approach. However, this centralised model poses a notable vulnerability: if the server experiences downtime, collaborative work comes to a halt until the server is operational once more.

**Distributed**: In a distributed system, each team member maintains a local copy of the project on their own machine, allowing them to save snapshots of the project independently. Git and Mercurial are examples of distributed version control systems. Git, in particular, is both free and open source, boasting exceptional speed and scalability.

**Installing Git**

Firstly, we can check if we have Git installed on our machine by viewing the current version:

[Git Version Code Snippet]

You can download the latest version of Git on [Link: Git Downloads]

**Git Bash**

The Bourne shell is the original [Unix](https://www.techtarget.com/searchdatacenter/definition/Unix) shell -- command execution program, often called a command interpreter -- that was developed in 1979 at what at the time was Bell Labs. Named for its developer, English computer scientist Stephen Bourne, the Bourne shell is also known by its program name, sh.

Git Bash (Bourne Again Shell) is a command-line interface (CLI) for Git that allows you to interact with Git and execute Git commands in a Unix-like terminal environment on Windows. While you can use Git through graphical user interfaces (GUIs) or directly in the Windows Command Prompt, having Git Bash provides several advantages:

**Consistency**: Git Bash provides a consistent and Unix-like command-line environment, which is more familiar to developers who work in Unix-based systems (Linux or macOS). This consistency is especially valuable if you switch between Windows and other platforms.

**Access to Unix Commands**: In addition to Git commands, Git Bash gives you access to a variety of Unix commands, such as ls, grep, find, and others, which can be beneficial for tasks beyond version control.

**Flexibility**: Git Bash allows you to work with Git in a way that may be more efficient for your specific needs. You can use Git commands directly in the terminal without relying on a GUI.

While Git Bash is a valuable tool, it's important to note that you can use Git through other interfaces, including GUI clients like GitKraken, SourceTree, and GitHub Desktop.

**Git Configuration**

Git configuration has three levels: system (system-wide settings), global (user-specific settings), and local (repository-specific settings). Local configurations take precedence over global and system configurations. In Git, you can configure various settings, including your identity (name and email), core behaviour, custom commands, remote repository details, and more. These configurations allow you to customize your Git workflow and behaviour to suit your specific needs and preferences.

*[Config User Name Code Snippet]*

Beware that the double quote is necessary, as the name contains white space. Also, a single dash means that the following flags are single-character only and generally means that more than one flag can be passed. In contrast, the double dash connotes a single positional flag argument to a command line tool. Alternatively, we can open the settings file with our default code editor:

*[Open Config Code Snippet]*

The "Auto Carriage Return Line Feed" property, often called autocrlf in Git, handles line endings in text files, as Windows handles line feeds differently than Linux and macOS. Windows prefixes carriage return before line feeds (/R/N versus /N).

**Initialising Repositories**

First, we must create a folder for our new project and enter the new folder. Then, we must initialise our repository:

*[Git Init Repo Code Snippet]*

The command line will return with the full path of our initialised repository and its folder. This directory will contain one subfolder called .git/, which is a hidden folder by default, as users should not directly modify this folder. If we want to see this folder, the traditional LS command will not show it, and we need to use LS -A (for Linux) or DIR /A (for Windows) to list all items, including hidden ones. We can use OPEN (for Linux) or START (for Windows) commands to open the folder from the bash terminal.

*[List Hidden Git Folder Code Snippet]*

This folder contains all the information and metadata that Git needs to manage the repository and track changes in the project. This folder contains all the information and metadata that Git needs to control the repository, such as configuration settings, object database, logs, staging and descriptions to track changes in the project. You'll lose the project history if you corrupt or remove this folder.

**Git Workflow**

In our daily work, we often make changes to project files, and when we reach a point where we want to capture the project's state, we create a "commit" in Git. Think of a commit as a project snapshot at a specific time. What sets Git apart from many other version control systems is the "staging area" or "index", an intermediate step where we prepare our changes for the next snapshot, deciding what goes into the next commit.

So, once we've finished making changes, we add the modified files to the staging area, review our changes, and if everything looks good, we make a commit. The contents in the staging area represent the state of the project just before our latest changes. If there are changes we don't want in the next snapshot, we can unstage them and commit only what's intended. When committing, we also provide a meaningful message describing what the snapshot signifies, which is essential for a clear project history.

A common misconception about Git is that the staging area becomes empty after committing changes. In reality, the staging area contains the previous version of the project. Similarly, when deleting files, we stage this change to reflect it in the staging area. Each commit in Git contains a unique identifier, a message, date and time information, the author, and a complete snapshot of the project's state at that particular moment.

Unlike some other version control systems, Git doesn't store just the changes that occurred; it retains the full content of the project. This allows Git to quickly restore the project to a previous snapshot without calculating the changes. For efficiency, Git avoids storing identical files multiple times and instead references them. Therefore, a snapshot represents a commit pointing to a directory structure's content. Git manages its data as a series of snapshots, much like a miniature file system.

*[Git Storage 1 Image]*

Whenever you commit in Git, you're essentially taking a picture of your project's current state, capturing all your files and storing a reference to that snapshot. To optimise storage, Git doesn't re-store unchanged files; it simply links to the previously stored, identical files.

*[Git Storage 2 Image]*

**Delta vs Snapshots**

**Snapshot storage** saves the entire state of a file or system as a single instance, known as a snapshot. Each snapshot represents the system at a given time and serves as a reference point for future changes. This method is simple, efficient, and straightforward, but it can consume a lot of storage space over time if there are frequent changes to the system.

**Delta storage**, on the other hand, saves only the changes made to a file or system rather than the entire state. Each change is stored as a delta or a difference from the previous version, and the system's current state is reconstructed by combining all the deltas. This method is more efficient in terms of storage space, as it only stores the differences between versions but can be more complex to implement and manage. *[Link: Maata]*

Consider your bank statement as an analogy: a bank statement reflects your current balance rather than tracking every transaction. Similarly, a snapshot stores the state of a file or system at a specific moment, providing a clear, straightforward representation in contrast to cumulative deltas, which record every change. Retrieving balances or changes between days is more efficient with snapshots because calculating the current balance would require us to traverse every account history change.

*[Snapshot vs Delats Image]*

**Staging**

Let’s make some changes to our project using the ECHO command. To see the status of the working directory of the staging area, we use the GIT STATUS. For shorter, less verbose status results, we also can use GIT STATUS -S.

*[Changes Code Snipet]*

The primary function of the git ADD command is to promote pending changes in the working directory to the git staging area. The staging area is one of Git's more unique features. It helps to consider it a buffer between the working directory and the project history. *[Link: Atlassian]*

*[Add Code Snippet]*

Since we have added our files to the staging area, modifying any file and querying a git status will result in a message where the file appears green as a new file already added and red as unstaged changes are present in the file.

**Committing Changes**

Adding commits keeps track of our progress and changes as we work. Git considers each commit change point or "save point". You can return to a point in the project if you find a bug or want to make a change. Note, that you must explicitly tell Git which changes you wish to include in a commit before running the GIT COMMIT command. *[Link: Git Tower]* Files won't be automatically included in the next commit unless you use the GIT ADD command to mark the desired changes for inclusion. Commits are not automatically transferred to the remote server as committing saves a new commit object only in the local repository. Exchanging commits has to be performed manually and explicitly with the GIT FETCH, GIT PULL, or GIT PUSH commands.

The quickest way to write a git commit is to use the command with the -M message flag, enabling us to write a short descriptive message under 72 characters. However, for longer and more meaningful messages, we can use the GIT COMMIT command without the flag and edit our commit in our default code editor. The text up to the first blank line in a commit message is treated as the commit title, and the rest of the commit is the body. The 50/72 rule is a set of standards that are pretty well agreed upon in the industry to standardise the format of commit messages. 50 is the maximum number of characters of the commit title, and 72 is the maximum character length of the commit body.

*[Commit Code Snippet]*

**Commit Best Practices**

Each commit should represent a logical unit of work and focus on a specific feature, bug fix, or improvement. Avoid committing unrelated changes together!

Write meaningful commit messages. The message should concisely describe what the commit does and why it is necessary. A good commit message helps others understand the context without delving into the code.

Keep code formatting and style changes separate from functional code changes. It's a good practice to have dedicated commits for formatting improvements.

Long messages can be hard to read. Keep messages concise and to the point while providing enough information to understand the purpose of the change.

Write commit messages in the imperative mood, as if you're giving a command. For example, "Fix a bug" or "Add a new feature." This convention is widely accepted and helps maintain consistency.

Always run tests and ensure that the code compiles without errors before committing to reduce the likelihood of breaking the build.

**Skipping the Staging**

The primary purpose of the Git staging area (also known as the index) is to allow you to review and prepare the changes committed to the repository. The staging area serves as an intermediate step between your working directory and the Git repository. It allows you to select changes (modifications, additions, or deletions) from your working directory for inclusion in the next commit.

However, if you are confident that the code to be committed does not need to be reviewed, you can skip the staging area. Skipping the staging area and making commits directly from your working directory is a practice often referred to as a "direct commit" or a "commit without staging". Skipping the staging area is acceptable for minimal, self-contained changes ready to be committed without further review. For example, fixing a simple typo in a comment.

So after changes in the working directory, we use direct commit all with a message instead of the ADD command (COMMIT -A -M or COMMIT -AM).

*[Skip Staging Code Snippet]*

**Removing Files**

We can use the RM or DEL commands if we want to remove unwanted files in our project. However, it is important to remember that even though we removed the file from our working directory, it will still be in the staging area. To confirm that, we can list our files in the staging area with GIT LS-FILES. When we remove files, we need to stage them with ADD command. As a shorthand, we can use GIT RM instead of the plain Linux one.

**Renaming and Moving Files**

Renaming can be done with the MV “move” command. If we run git STATUS, we will see two changes: the file with the previous name is deleted, and a new file with the current name is added. As Git doesn’t automatically track new files, we need to stage these changes. Because renaming is a two-step operation, once we modify the working directory, which stages both the deletion of the old file and the addition of a new file, Git provides its own MV function.

**Git Ignore**

For most projects, there will be files and folders that should not be included or tracked in the repository. Such files either do not add any value to our project or pose a security threat:

* Npm and Yarn: dependencies can be regenerated from the package JSON,
* Build and Dist folders: they are compiled from the source code,
* Development log files,
* Environment configurations, such as .ENV files, may contain sensitive information, such as API keys and credentials, and exposing them may cause security breaches.

*[Gitignore Code Snippet]*

However, if Git has already tracked a file, it won’t be able to ignore it anymore, and we need to remove it from the staging area.

*[Remove from Index Code Snippet]*

**Viewing History and Commits**

Using the GIT LOG, we can list the history of all the commits made to the repository. The listed commits are sorted from latest to earliest and include the identifier, author and date. The git identifier or a Git Commit ID is a unique 40-character SHA-hash value generated automatically and assigned to commits whenever a new commit is made to the repository. Commit ID is used while merging commits or checking out files from different commits. *[Link: Hameed]* To get a less verbose version of the history, we use the GIT LOG with the --ONELINE flag, and to reverse the sorting of the list, use the –REVERSE flag.

*[Git Log Code Snippet]*

To inspect the changes in a commit, we type the GIT SHOW in the terminal with the unique ID of the commit. There is no need to type the complete ID. As long as there are no multiple matches, the first few characters of the ID can identify a commit. Alternatively, to see the last commit, use the HEAD and an optional tilde ~ to signify how many steps need to go back. Additionally, we can view the whole directory structure using the LS-TREE command.

*[Git Show Code Snippet]*

In Git the folders are represented as trees and the files as blobs. BLOB is a “Binary Large Object,” a data type that stores binary data. Binary Large Objects (BLOBs) can be complex files like images or videos, unlike other data strings that only store letters and numbers. *[Link: Burchfiel]* A Git tree object creates the hierarchy between files in a Git repository. You can use the Git tree object to create the relationship between directories and the files they contain. These endpoints allow you to read and write [tree objects](https://git-scm.com/book/en/v2/Git-Internals-Git-Objects#_tree_objects) to your Git database on GitHub.

**Unt­­­staging and Restoring**

Because every commit should be one logically comprehensible unit, sometimes we need to unstage some of the changes we want to omit from the next commit. We can undo the ADD operation with the RESTORE function.

*[Restore Code Snippet]*

We can also discard local changes in our working directory with the GIT CLEAN command. This action would permanently delete the files from our working directory, so we must use the –FORCE and optional -D (directories) flags.

**Restoring Previous Versions**

To restore the last commit, list the commits with the GIT LOG –ONELINE to see the commit ID, or to restore the last commit, use the HEAD~1.

## Funtional Programming with Monet.js

The latest challenge I encountered in my work involved delving into functional programming using Monet.js and Ramda. While Ramda had an abundance of resources available, I struggled to find a straightforward begginers guide about Monet.js, apart from the official GitHub documentation. *[Link: Monet Git]*

Although numerous tutorials exist on using monads and functional programming in general, most are based on Haskell, Lisp, or other languages that natively embrace functional programming paradigms. Therefore, I took it upon myself to create my own tutorial, experiment, and conduct research to share my findings. I hope that, through this article, everything will eventually click together, making your journey into functional programming less troublesome than mine. However, as with most of the things on this site, I am writing this article for my own benefit as well, as part of my learning process, so if I made any errors please let me know.

**Functional Programming**

Functional programming languages, unlike many others, have a strong foundation in mathematics, benefitting from centuries of development in mathematics field. This solid mathematical basis provides them with significant advantages compared to more recent programming paradigms, like object-oriented programming, which are considered less mature due to their shorter history and development.

In functional programming, there are no statements, only *expressions*. Thanks to functional purity, you can reason about code and reduce code complexity in the same way you reduced the complexity of equations back in algebra class. In non-functional languages (imperative languages), there is no equivalent mechanism for reasoning about how the code works. *[Link: Spectrum]*

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. In functional programming, functions are treated as first-class citizens, meaning they can be passed as arguments to other functions, returned as values from other functions, and assigned to variables.

**Gloassary**

To begin learning functional programming, let's cover some of the most important key concepts. I'll introduce each term with examples in case you're new to them. Feel free to skip ahead if you're already familiar with the terminology.

**Immutability:** Data, once created, cannot be changed or mutated. Instead of modifying existing data, functional programming emphasizes creating new data structures.

**First-Class Functions:** Functions are treated as first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned as values from other functions.

**Pure Functions:** Functions in functional programming are pure, meaning their output is solely determined by their input and they have no side effects, so same input will always produce the same output. These side effects may be: modifying a variable or a data structure outside of the function scope, modify a field of an object, throw exceptions, IO operations etc.

**Referential Transparency:** It emphasizes referential transparency, where a function call with a given set of parameters can be replaced by its corresponding result without changing the program's behavior.

**Higher-Order Functions:** Functions that can accept other functions as arguments or return functions are called higher-order functions. They enable more concise and modular code.

**Recursion:** Functional programming often uses recursion as an alternative to traditional iterative loops.

**Declarative Style:** Functional programming emphasizes a declarative style, where the programmer describes what the program should accomplish rather than specifying how to achieve it.

**What are functors?**

When programmers started writing programs with for loops, they have noticed that you can iterate over many different data structures (arrays, linked lists, various types of dictionaries, graphs) that have one thing in common - we can write a for loop which somehow iterates over their elements. In essence mapping works similarly, where we apply a function to a each item of a container, and return a new container. A a map function never changes the container, instead it just act upon its contents.

In mathematics, a functor is a map between [categories](https://en.wikipedia.org/wiki/Category_(mathematics)), and in functional programming they are a powerful concept that can be applied to JavaScript as well. They allow for the manipulation of data in a way that is both predictable and safe. *[Link: Levelup]* A functor is a container that holds a value and can be used to apply a function to that value (a simple example would be to map a JavaScript array). So, in the simples term a funtcor is an object that can implement a map. However, surprisingly Promises are also functors, but instead of map we use our function in the then method which returns a new Promise.

**What are monads**

Monads are a concept from functional programming that provide a way to structure computations and handle side effects in a pure and composable manner. In essence, monads are a design pattern used to manage the flow of data between functions while encapsulating behaviors like state, asynchronous operations, or error handling. They help address the challenges of working in a purely functional paradigm where side effects and mutable state are typically avoided.

1. ***What is Functional Programming?***
   * *Basics of functional programming paradigm.*
   * *Key concepts: immutability, pure functions, and first-class functions.*
2. ***Why Functional Programming in JavaScript?***
   * *Advantages of using functional programming in JavaScript.*
   * *Real-world use cases.*

***Module 2: Introduction to Monads***

1. ***Understanding Monads***
   * *Definition and purpose of monads.*
   * *Monad laws.*
2. ***Introduction to Monet.js***
   * *Overview of Monet.js library.*
   * *Installing Monet.js.*

***Module 3: Identity Monad***

1. ***Introduction to Identity Monad***
   * *Explanation of the Identity Monad.*
   * *Use cases for the Identity Monad.*
2. ***Working with Identity Monad in Monet.js***
   * *Creating an Identity Monad.*
   * *Extracting values from the Identity Monad.*
   * *Mapping over the Identity Monad.*

***Module 4: Maybe Monad***

1. ***Introduction to Maybe Monad***
   * *Handling null and undefined with Maybe.*
   * *Avoiding null/undefined-related errors.*
2. ***Working with Maybe Monad in Monet.js***
   * *Creating a Maybe Monad.*
   * *Handling optional values using Maybe.*
   * *Chaining Maybe Monads.*

***Module 5: Either Monad***

1. ***Introduction to Either Monad***
   * *Dealing with errors using Either.*
   * *Representing success and failure.*
2. ***Working with Either Monad in Monet.js***
   * *Creating an Either Monad.*
   * *Transforming and chaining Either Monads.*

***Module 6: Writer Monad***

1. ***Introduction to Writer Monad***
   * *Capturing logs and side effects.*
   * *Structuring computations with logging.*
2. ***Working with Writer Monad in Monet.js***
   * *Creating a Writer Monad.*
   * *Combining and transforming logs.*

***Module 7: Future Monad***

1. ***Introduction to Future Monad***
   * *Asynchronous programming with Future.*
   * *Representing values to be available in the future.*
2. ***Working with Future Monad in Monet.js***
   * *Creating a Future Monad.*
   * *Composing asynchronous operations.*

## React Custom Hook Pattern

In this article, I will discuss React hooks and fundamental React hook patterns, as well as how to address the common error message: “Uncaught error: React components can only be called inside the body of a function component,” in a specific way that caught my attention.

If you are new to the functional React style and have primarily used classes in your React application, hooks can manage component state and lifecycle in classless function components, as functions lack these features. In class components, **setState** is used for state management, while function components may use built-in or custom hooks to handle dynamic components, lifecycle, and DOM access. Hooks can generally provide a more declarative approach for sharing stateful logic without altering the component hierarchy. React core includes three built-in hooks—**useState**, **useEffect**, and **useContext**—along with seven additional hooks. The most important ones are **useState**, **useEffect**, **useContext**, and **useRef**. By convention, hook names are prefixed with the word "use", and for consistency its recomended to maintain this convention.

**Fundamental hooks**

One of the leading problem with classes is that sharing stateful logic with multiple nested components required a treelike structure of data management, which could get particularly complex fairly easily. Hooks simplify state creation and access and modification drastically, without creating your component a state object. The useState function accepts an optional parameter for setting default values and returns an accessor and a mutator, and using an array we can destructure them.

*[Code Snippet UseState]*

The **useEffect** hook enables us to implement logic for all component lifecycles - such as mount, update, and unmount - within a single function. **useEffect** takes a function and an optional dependency array. Without dependencies, **useEffect** runs on mount and every time stateful data is modified in the component. However, if we want the effect to trigger only when certain dependencies change, we provide the optional dependency array. An empty dependency array causes the hook to run only once, during mounting. Additionally, we can listen to state changes and update the component by including relevant state values in the dependency array. Finally, we can use a tear-down function by returning a function from the hook, which runs before the component unmounts. Open the console (F12 – Win / Cmd + F12 - Mac) and press the button to see the function logged on the dev tool.

*[Code Snippet UseEffect]*

*[Button]*

The useContext hook allows you to work with the React context API. Context helps you share data between components without passing them through properties the component chain, also called props drilling. Note, that only data that does not need to be updated often should be placed on React context, like theme, UI state or shared global state for a group of components, and is not designed to be an application state management tool. First create a context and a context provider that accepts React Nodes as its children, then wrap the context provider around the component tree you want to share context with. Beware that the context must be only accessed within the context provider!

*[Code Snippet UseContext]*

*[Button]*

The useRef hook is used to create a mutable reference that persists across renders. Unlike **useState**, the value returned by **useRef** does not trigger re-renders when it changes. It's commonly used to access or store references to DOM elements or other values that need to persist between renders.

*[Code Snippet UseRef]*

*[Button]*

Check your dev tools to see the result.

Other notable built-in hooks, which are beyond the scope of our discussion, include **useReducer**. With this hook we can update the state following the Redux pattern, by dispatching actions and reducers with optional payloads to compute the next state. Additionally, there's **useMemo**, which is employed to cache function call results that may be expensive to compute on each render, and **useCallback**, used to cache entire functions. In React, functions defined in function components get recreated on every render.

*[Code Snippet Custom Hook]*

*[Button]*

**Custom Hooks**

Building your own hooks lets you extract component logic into reusable functions, that can be applied in multiple components. In your custom hooks you are free to use any number of built-in hooks, such as useState, useEffect or useRef and other custom hooks built by you or third-party APIs. In essence, React hooks are plain JS functions which purpose is to interact with (hook on) the React component state and lifecycle, and they are implemented so that they work correctly only if they are called while a functional component is rendering. So keep in mind that if your function simply calls a hook but doesn't return JSX or doesn't represent a UI element, it's considered a custom hook.

**3 Rules of Using Hooks**

Hooks should only be called at the top level of functional components or custom hooks. They should not be called inside loops, conditions, or nested functions. This ensures that hooks are called in the same order on every render, allowing React to properly maintain the state of each hook between renders.

Hooks should only be called from within functional components or custom hooks. They should not be called from regular JavaScript functions, event handlers, or asynchronous functions. This ensures that React can properly associate each hook call with the component that owns it.

Hooks should always be called unconditionally, meaning that their calls should not be placed inside conditional statements. Instead, you can use conditional logic inside the hook itself if necessary. This ensures that hooks are called consistently on every render, preventing bugs related to missing or incorrect hook calls.

**Invalid Hook Call Error**

While using custom hooks are great ways to organise the state and lifecycle of your React components, they may get increasingly complicated to handle as your application grows in size and complexity. Often times what tripped me off is applying to the hook rules that helps maintain the order of hook calls. So to demonstrate it I created a simplified scenario that reproduces the issue I faced. Suppose I have a component that will fetch album data using custom fetch hooks, and display them in a table. Under the album table I placed a button that would trigger another custom fetch hook, which would load comments; however, the comments can only be fetched after the button click as they were highly dependent of the current state of the application.

*[Code Snippet Example Component]*

In my custom fetch hooks, I've employed **useQuery** to facilitate basic data fetching, resulting in an object comprising the query status (loading, error, or success) along with the retrieved data. While going into the specifics of **useQuery** may seem irrelevant, rest assured that the focus lies in the hooks' ability to retrieve the album and comment data.

*[Code Snippet Example Hooks]*

Unfortunately, executing this program and pressign the button will inevitably lead to the all-too-familiar error message: Invalid hook call. And rightfully so, as I violated the rules of hooks. Check the Component.tsx above and see if you can spot where things went south.

*[Image Error Message]*

*[Code Snippet Error Line]*

As I wanted to fetch the data no sooner or later than when the button is pressed, I placed the hook in the buttons event handler function.

**Wrapping and Extracting the Fetch**

Fortunately, there are multiple ways around these kinds of problems, and one of the most intuitive that I found so far came to me when I was pair programming with one of my senior collegue. The solution goes like this: in this scenario useFetchComments hook is only responsible for fetching the comment data. So we should provide a hook that exposes functionality related to the comments, including data fetching.

*[Code Snippet Wrapped Hook]*

This pattern doesn't have a specific name in React terminology, but it can be considered a form of abstraction or encapsulation. Essentially, you're creating a custom hook (**useGetComments**) that internally utilizes another hook (**useComments**) to fetch data and provides a simplified interface (**getComments**) for consuming that data.

This approach is a common pattern in React development, where you encapsulate complex logic or data-fetching operations within custom hooks to provide a cleaner and more reusable interface for components. It's a form of modularization that promotes separation of concerns and improves code maintainability.

## Recreating McDonald's Ordering Kiosk UI