How To: HTML Canvas Buttons

Jake Pine · Follow
6 min read · Oct 25, 2020

In this tutorial, I will walk you through creating interactive buttons on the HTML5 canvas. This tutorial is split up into three sections: Getting Familiarized with the Starter Code, Creating a Button Class, and Adding Event Listeners to the Canvas.

For this tutorial, I will assume a basic understanding of both JavaScript and HTML. While a thorough knowledge of HTML is not necessary, we will be using a base `index.html` file as well as basic HTML events.

The code snippets in this tutorial are written in TypeScript as it more clearly shows the types of variables and functions which will hopefully minimize confusion. If you are new to TypeScript or would prefer to see the code in pure JavaScript, **all source code from this tutorial can be found at this git repo**.

Now, onto the tutorial…

Getting Familiarized with the Starter Code:

The starter code for this project can be found here. I have created a basic `index.html` file with a head and a body. Inside the body, you can find the canvas we will be drawing on, and a link to our JavaScript file:

```
<canvas id="main" width="640" height="480"></canvas>
<script src="main.js"></script>
```

In our `main.js` file, I have laid out the basic structure of a canvas update loop:

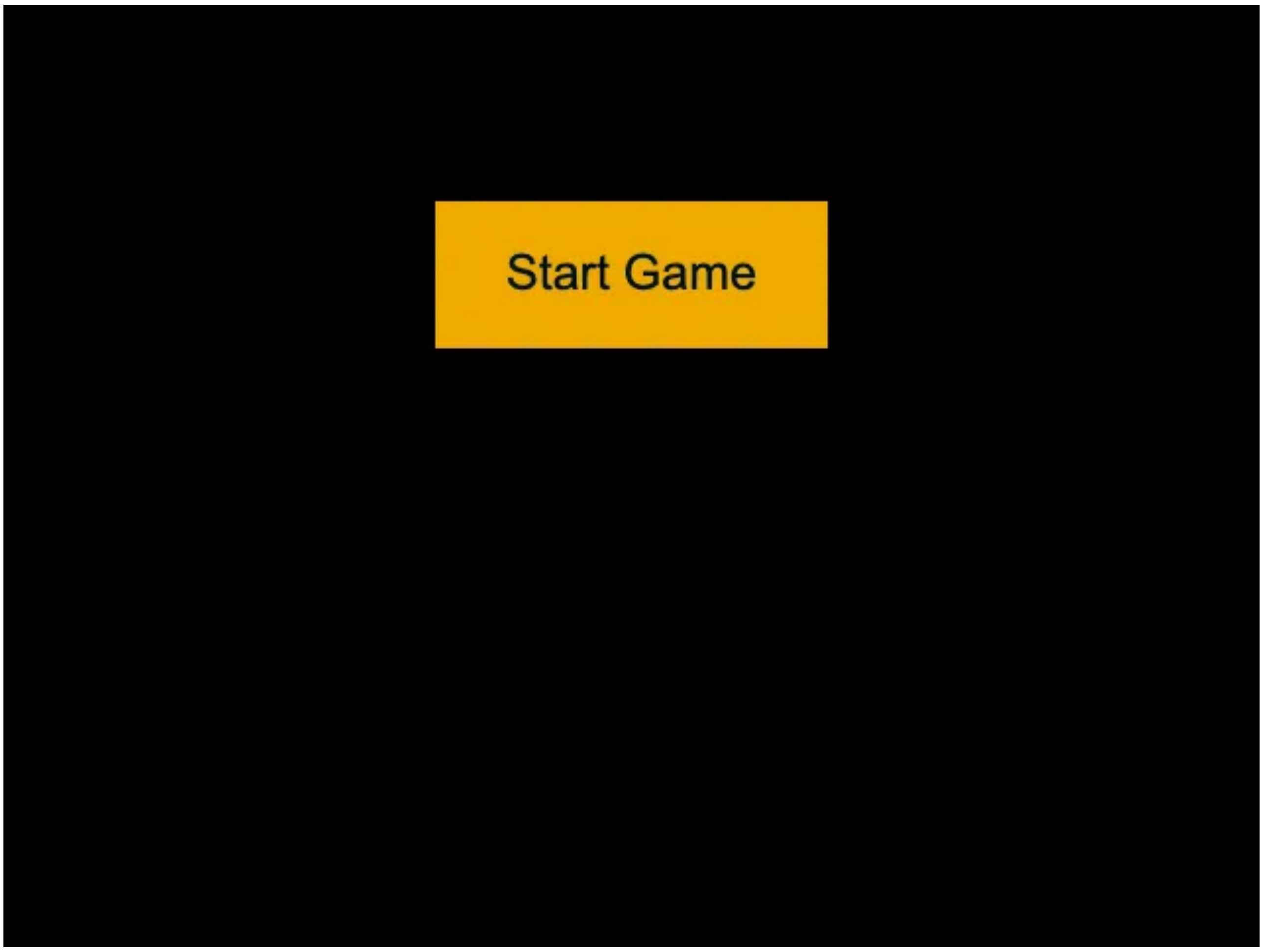
```
let canvas = <HTMLCanvasElement>document.getElementById('main');
let c = canvas.getContext('2d');

function init(): void {
  // initialization
  update();
}

function update(): void {
  ... // recalculate and render graphics
  requestAnimationFrame(update);
}

init();
```

The `init` function serves as a place to setup (initialize) any variables for the program, while the `update` function provides a place to update the scene and redraw every frame.

I went ahead and added to the update loop so that it draws a black background and draws the initial version of our button:

```
// draw background
c.fillStyle = '#000000';
c.fillRect(0, 0, canvas.width, canvas.height);

// draw button
c.fillStyle = '#eeaa00';
c.fillRect(220, 100, 200, 75);
c.fillStyle = '#001122';
c.textAlign = 'center';
c.font = '25px arial';
c.fillText('Start Game', 320, 145, 200);
```

We are now able to see a black canvas with an orange ‘button’ on it:



A live version of this implementation can be found here.

Creating a Button Class:

I could teach you how to create a button at a specific location with a specific size and handle clicks for that area of the screen, but that wouldn't be very scalable. Instead, I will walk you through the process of creating a container class for our button so we can easily create many buttons anywhere on the screen (and even allow for a moving button or one that grows and shrinks).

First, it would be good to decide what variables our button class will contain. Another way to think about this is: *what might differ from button to button?*

The features I have decided that vary between buttons are: text, button color, text color, position, and size. From this, we can create the following fields for our class.

```
class Button {
  text: string;
  fillColor: string;
```

```
    textColor: string;

    // position
    x: number;
    y: number;

    // size
    width: number;
    height: number;

    ...
  }
```

Next, we should make a constructor for our button class. Since the main things that define separate buttons to me is the text and the styling, I made the constructor with only those parameters. I created separate functions for setting the position and size of the button (which also allows for easier animating of the buttons).

```
  // in Button

  constructor(text: string, fillColor: string, textColor: string) {
    this.text = text;
    this.fillColor = fillColor;
    this.textColor = textColor;
  }

  ...

  setPosition(x: number, y: number): void {
    this.x = x;
    this.y = y;
  }
  setSize(width: number, height: number): void {
    this.width = width;
    this.height = height;
  }
```

This class now holds all of our data for a button, but how do we draw it? We could access all of its fields individually, and draw it using the code we had in `main.js` prior, but again, that wouldn't be very scalable. Let's create a `draw` method in Button so each button knows how to draw itself.

Here I have adapted the drawing code from `main.js` to use the fields in our class instead of the hardcoded values:

```
draw(c: CanvasRenderingContext2D): void {
  // draw the button body
  c.fillStyle = this.fillColor;
  c.fillRect(this.x, this.y, this.width, this.height);

  // draw the button text
  c.fillStyle = this.textColor;
  c.textAlign = 'center';
  c.textBaseline = 'middle';
  c.font = '25px arial';
  c.fillText(this.text, this.x + this.width / 2, this.y +
this.height / 2, this.width);
}
```

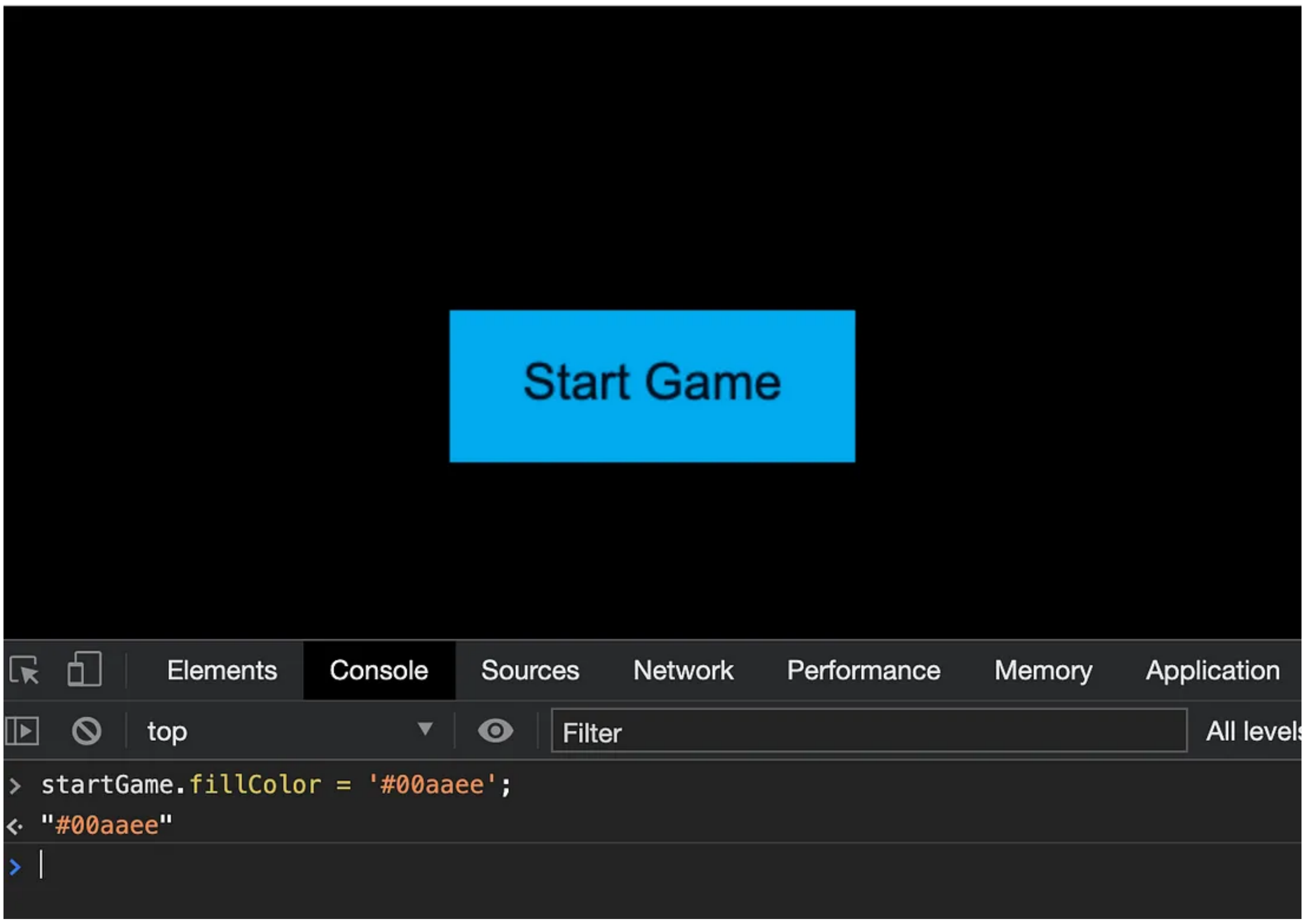
Now in our `main.js` it is time to create an instance of our `Button` class. We will initialize our button in the `init` function:

```
let startGame: Button;

function init(): void {
  // initalization
  startGame = new Button('Start Game', '#eeaa00', '#001122');
  startGame.setPosition(canvas.width / 2 - 100, 150);
  startGame.setSize(200, 75);

  update();
}
```

All we need to do now in order to draw our button is to call our button's `draw` method in the `update` loop:

```
function update(): void {
  c.fillStyle = "#000000";
  c.fillRect(0, 0, canvas.width, canvas.height);

  ...

  startGame.draw(c);

  ...

  requestAnimationFrame(update);
}
```

If we look at the result, we can see that our button is drawing once again and we can tweak the values in our developer console and watch it update live.



```
> startGame.fillColor = '#00aaee';
<· "#00aaee"
>
```

A live version of this implementation can be found here.

If you are following along, it might be a good exercise to try implementing button borders (ie. color, thickness, etc...) yourself.

Adding Event Listeners to the Canvas:

Now we're getting to the main show... almost.

Before we add our button clicks, I suggest we think about the way we call draw on our button. The way we have it currently, we will need to call that function for every button in our scene. Instead of cluttering our code with all of these `button.draw(c);` lines, let's create an array to store our buttons.

```
let buttons: Button[] = [];

function init(): void {
  // initialization
  let startGame = new Button('Start Game', '#eeaa00', '#001122');
```

```
    startGame.setPosition(canvas.width / 2 - 100, 150);
    startGame.setSize(200, 75);

    buttons.push(startGame);
    ...
  }
```

This way we are able to cleanly loop through all of our buttons and call their `draw` methods:

```
  // in update()

  buttons.forEach(button => button.draw(c));
```

In order to add functionality to our buttons, we will need to add two things to our `Button` class. The first is a method that checks if a given position is within the bounds of the button (this will make it very easy to check if the mouse clicked on a button).

```
  inBounds(mouseX: number, mouseY: number): boolean {
    return !(mouseX < this.x || mouseX > this.x + this.width ||
  mouseY < this.y || mouseY > this.y + this.height);
  }
```

The second is a function that should be called when the button is clicked (aka what the button does). Since this function will vary for each button, I will treat it as a field of `Button` with the type of a void lambda function.

```
  onClick: () => void;
```

For our ‘Start Game’ button, I made the following lambda to log “Start Game!” whenever the button is clicked:

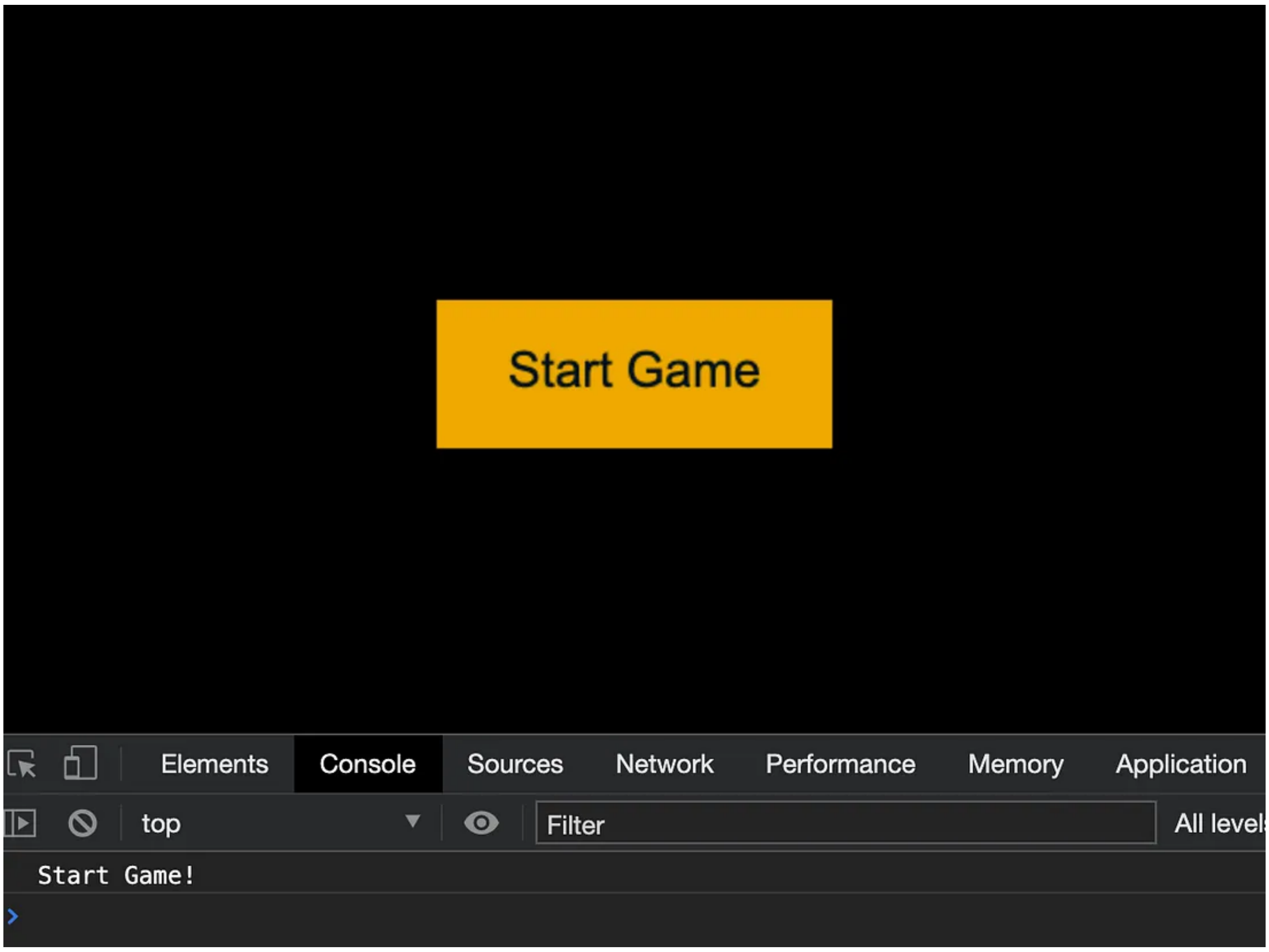
```
  // in init
  let startGame = new Button('Start Game', '#eeaa00', '#001122');
  ...
  startGame.onClick = () => console.log('Start Game!');
```

```
...
buttons.push(startGame);
```

Now we are ready to add our event listener. Our event listener will detect anytime the canvas is clicked, determine the mouse position on the canvas, and call the `onClick` method of all (if any) buttons that the mouse was over when the click occurred.

```
// in main.js

canvas.addEventListener('click', (event: MouseEvent) => {
  let x = event.pageX - (canvas.clientLeft + canvas.offsetLeft);
  let y = event.pageY - (canvas.clientTop + canvas.offsetTop);

  buttons.forEach(b => {
    if (b.inBounds(x, y) && !!b.onClick) b.onClick();
  });
});
```

Now we can see the message “Start Game!” in our console whenever we click our button.



```
Start Game!
```

A live version of this implementation can be found here.

https:jpinedev.github.io/ui-button-tutorial/

Wrapup:

If you would like an exercise to try to build off of what I covered, my suggestions would be to make buttons change color when hovered over or clicked on. If you are not very familiar with HTML Events, this may be a helpful resource.

All source code can be found here, and live versions of each of the three stages of development can be found here.

Html Canvas Html5 Canvas Html5 Game Development Html5 Game Developer UI

Follow

Written by Jake Pine

1 Follower

More from Jake Pine

Jake Pine

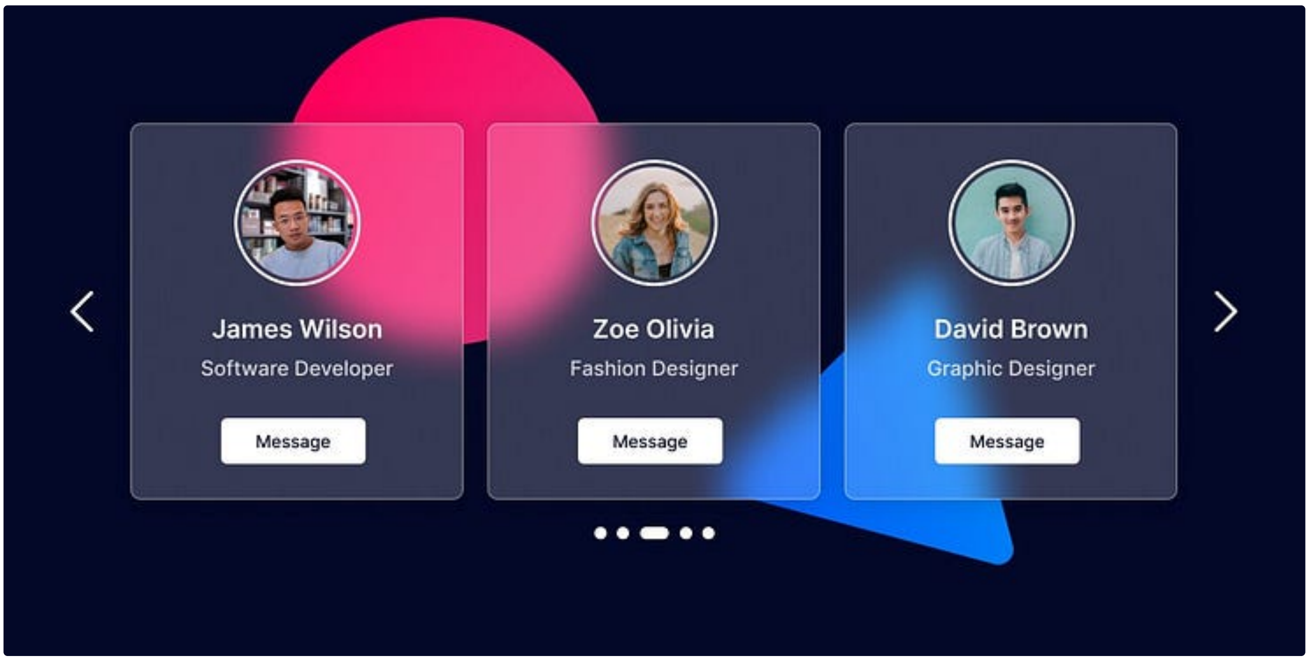
Chicken Dinner: The Meal of Champions

Your adrenaline surges to its peak when you hear the resounding shot of an AWM followed by a message in the kill-feed. Out of the initial...

Sep 28, 2020

See all from Jake Pine

Recommended from Medium

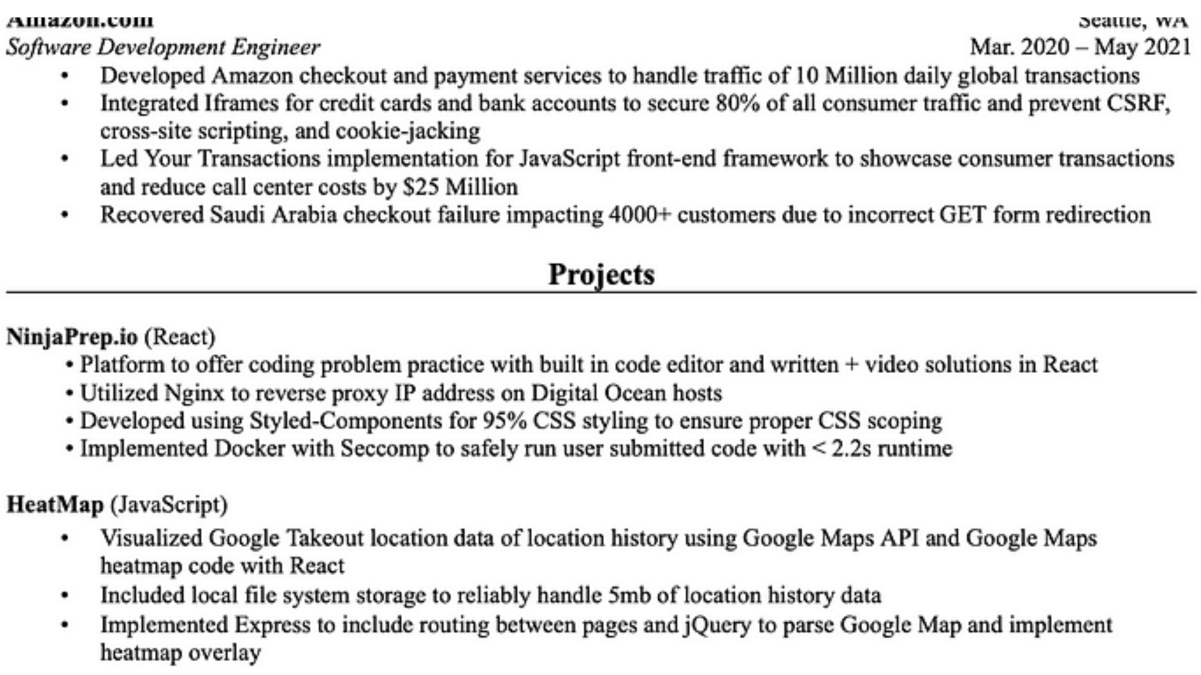


CodingNepal

How to Create Responsive Card Slider in HTML CSS & JavaScript

"This article was originally published at https://www.codingnepalweb.com."

Jul 5 7



Amazon.com Seattle, WA
Software Development Engineer Mar. 2020 – May 2021

- Developed Amazon checkout and payment services to handle traffic of 10 Million daily global transactions
- Integrated Iframes for credit cards and bank accounts to secure 80% of all consumer traffic and prevent CSRF, cross-site scripting, and cookie-jacking
- Led Your Transactions implementation for JavaScript front-end framework to showcase consumer transactions and reduce call center costs by $25 Million
- Recovered Saudi Arabia checkout failure impacting 4000+ customers due to incorrect GET form redirection

Projects

NinjaPrep.io (React)
- Platform to offer coding problem practice with built in code editor and written + video solutions in React
- Utilized Nginx to reverse proxy IP address on Digital Ocean hosts
- Developed using Styled-Components for 95% CSS styling to ensure proper CSS scoping
- Implemented Docker with Seccomp to safely run user submitted code with < 2.2s runtime

HeatMap (JavaScript)
- Visualized Google Takeout location data of location history using Google Maps API and Google Maps heatmap code with React
- Included local file system storage to reliably handle 5mb of location history data
- Implemented Express to include routing between pages and jQuery to parse Google Map and implement heatmap overlay

Alexander Nguyen in Level Up Coding

The resume that got a software engineer a $300,000 job at Google.

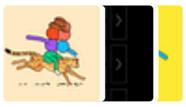
1-page. Well-formatted.

Jun 1 25K 502

Lists

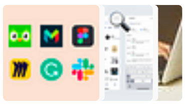
Icon Design

36 stories · 443 saves



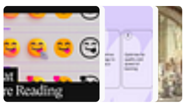
Figma 101

7 stories · 768 saves



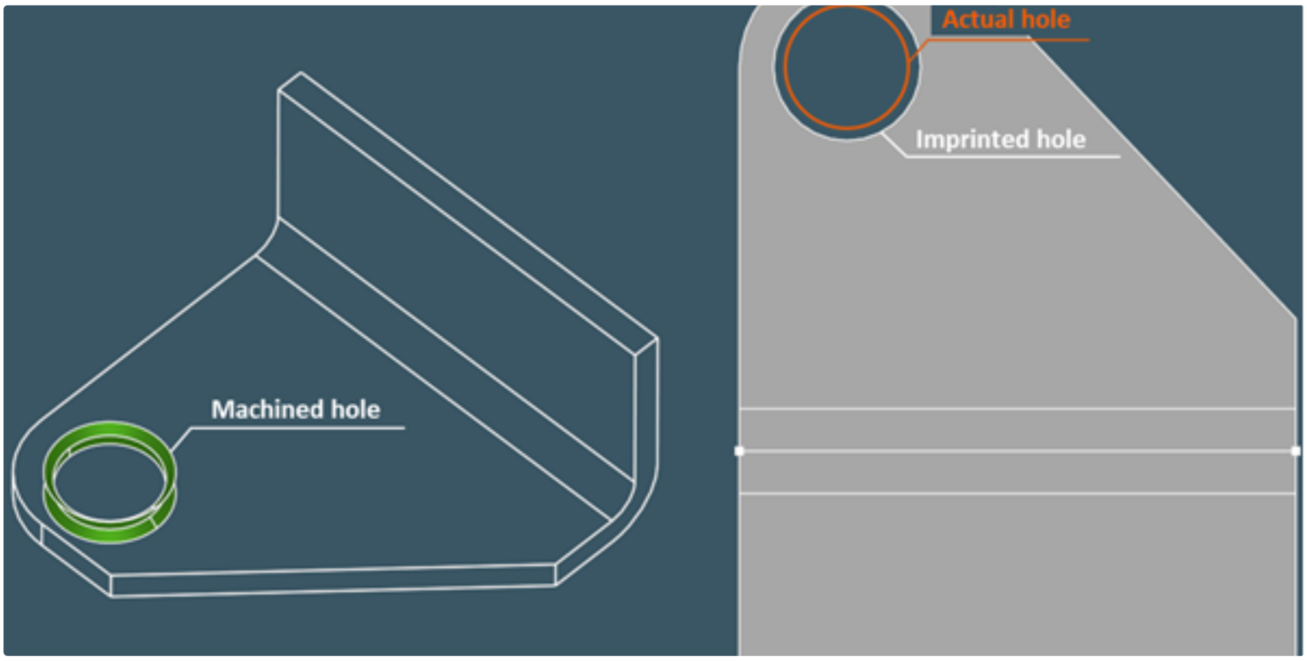
Interesting Design Topics

257 stories · 875 saves



Stories to Help You Grow as a Designer

11 stories · 1017 saves

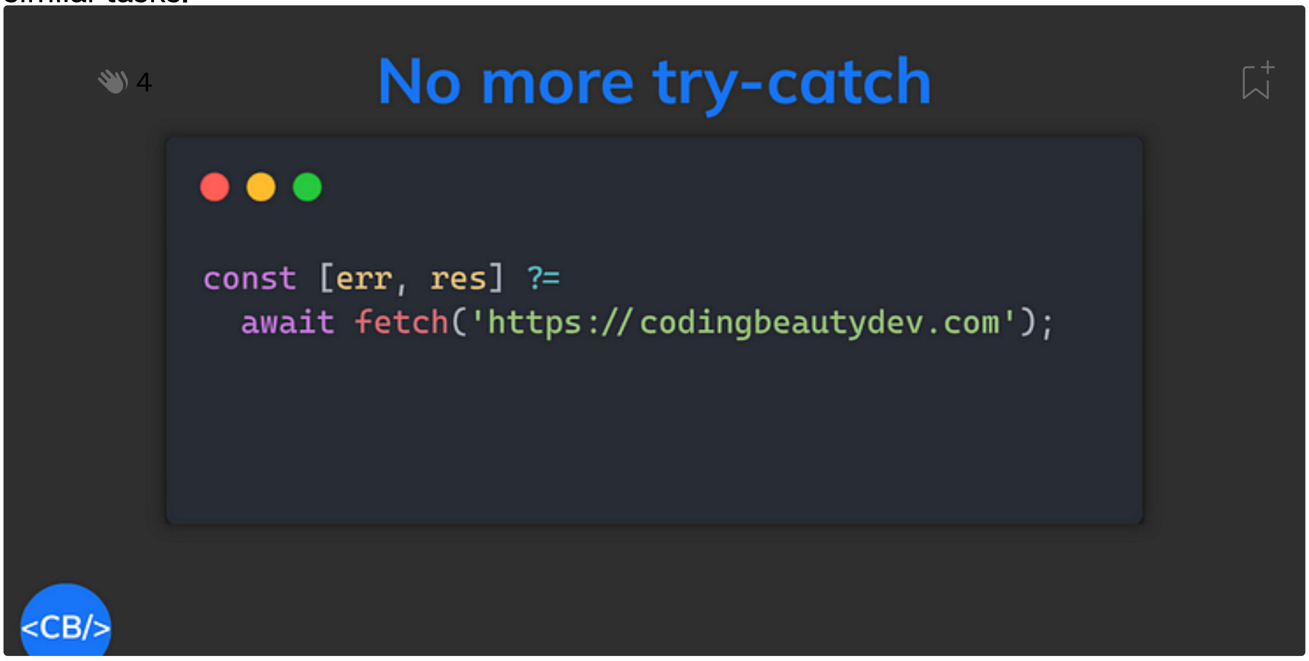


Analysis Situs

On sheet metal unfolding. Part 7: drilled holes

Part 1 gives an introduction to unfolding and reviews the software packages for solving similar tasks.

4



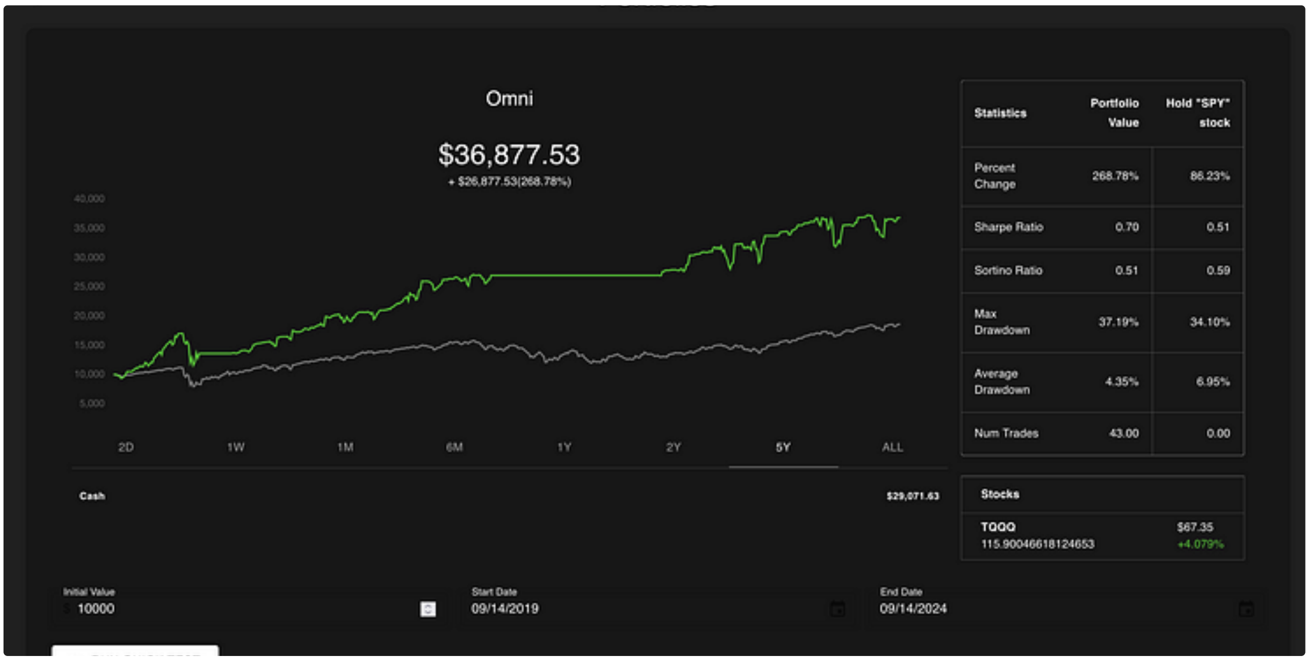
```
const [err, res] ?=
  await fetch('https://codingbeautydev.com');
```

Tari Ibaba in Coding Beauty

This new JavaScript operator is an absolute game changer

Say goodbye to try-catch

Sep 17 3.9K 56



Statistics	Portfolio Value	Hold "SPY" stock
Percent Change	268.78%	86.23%
Sharpe Ratio	0.70	0.51
Sortino Ratio	0.51	0.59
Max Drawdown	37.19%	34.10%
Average Drawdown	4.35%	6.95%
Num Trades	43.00	0.00

Cash $29,071.63

Stocks	
TQQQ 115.90046618124653	$67.35 +4.079%

Initial Value $ 10000

Start Date 09/14/2019

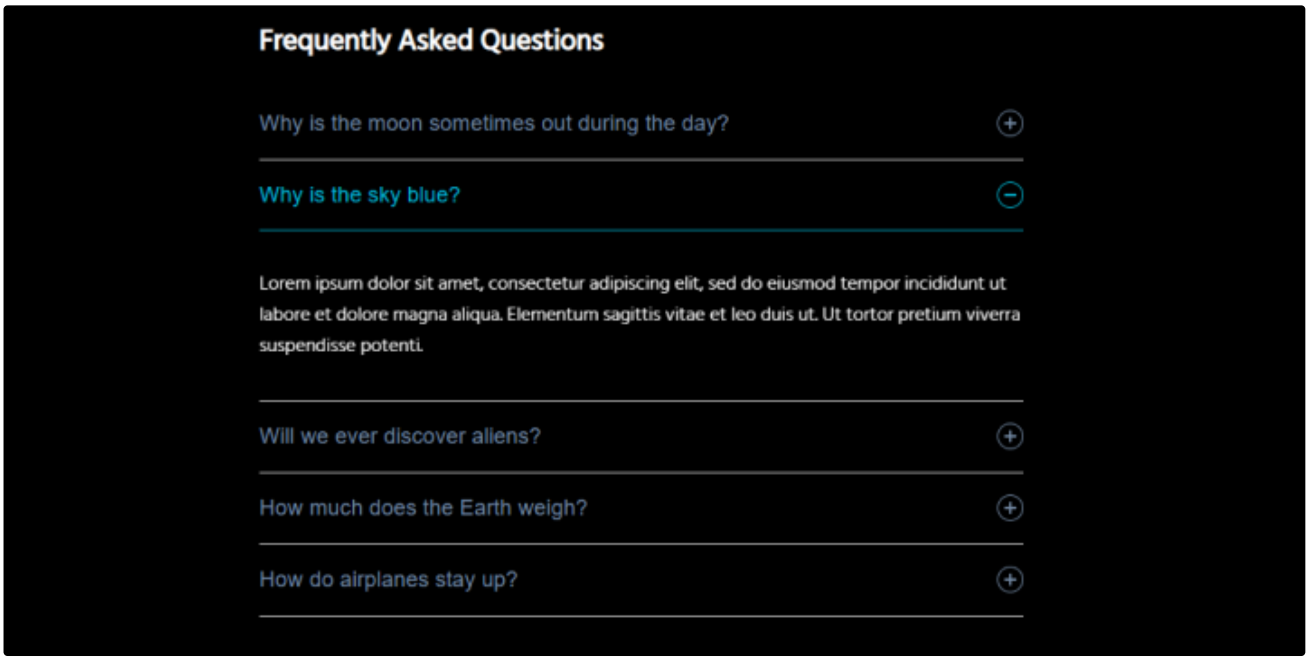
End Date 09/14/2024

Austin Starks in DataDrivenInvestor

I used OpenAI's o1 model to develop a trading strategy. It is DESTROYING the market

It literally took one try. I was shocked.

Sep 15 5.9K 150


Frequently Asked Questions

Why is the moon sometimes out during the day?

Why is the sky blue?

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Elementum sagittis vitae et leo duis ut. Ut tortor pretium viverra suspendisse potenti.

Will we ever discover aliens?

How much does the Earth weigh?

How do airplanes stay up?

Idrak Mirzayev

Accordion/Collapse Module

An accordion, or collapse module, is a user interface design pattern that allows users to expand and collapse content sections. This...

Nov 2 10

See more recommendations