Relatório do Projeto Final de Programação Web II

Carlos Barreiro nª20360
José Castro nº20351
Lucas SIlva nº20611
Tiago Sampaio nº20366

Curso Técnico Superior Profissional em Desenvolvimento Web e Multimédia

Famalicão, Junho, 2021

Resumo

Neste relatório iremos abordar todos os passos necessários na criação de uma página web contendo dois jogos, um de Black Jack e um jogo de compra e venda de Criptomoedas. Este trabalho serve principalmente para demonstrar todo o conhecimento adquirido ao longo da Unidade Curricular de Programação Web.

Lista de Abreviaturas e Siglas (se aplicável)

***SVG* -** É um vetor suportado nos navegadores e pode ser alterado em css ou javascript.

CSS - Uma mecanismo para estilizar os elementos HTML

HTML - Apesar de não ser uma linguagem de programação é bastante utilizada na construção de páginas web.

TS - Typescript é uma linguagem de programação proveniente do JavaScript porém sendo tipada, ou seja, é necessário definir os tipo das variáveis.

Palavras-Chave:

Event onclick - É um evento que aciona uma ou várias funções ao ser pressionado.

Moustache notations - Uma referência que está ligada ao serviço com a importação do mesmo no ficheiro .ts.

Viewchild - As viewchild servem para ir buscar os elementos ao HTML sem utilizar Javascript (getElementById).

Component - Um componente é um template que pode ser instanciado por um ngModule.

Div - Uma tag define uma divisão ou secção num documento HTML e pode ser usado como *container* para elementos HTML que podem ser então estilizados com CSS ou manipulado com JavaScript.

Angular - Angular é uma framework baseada em javascript que conta com diversas funcionalidades e tendo como linguagem principal o typescript, tem várias vantagens tais como o carregamento dinâmico das páginas e compilação assíncrona.

Índice

1 Introdução

O presente trabalho é sobre dois jogos que se interligam, ambos desenvolvidos por nós, onde um é sobre criptomoedas e o outro sobre blackjack, basicamente o usuário investe onde pode aumentar ou diminuir as suas economias, e com o mesmo dinheiro jogar no blackjack onde também pode apostar as suas economias.

Quanto às inspirações, no blackjack experimentamos alguns jogos mas não nos inspiramos em nenhum jogo em particular, simplesmente criamos o nosso próprio design, procurando várias imagens e completando com algumas imagens produzidas em photoshop, com as regras originais. No caso do crypto foi igual, vimos alguns sites mas não nos inspiramos com nenhum em especial.

Os objetivos deste trabalho são fazer a implementação da matéria lecionada ao longo do semestre e desenvolver uma dinâmica de pesquisa na programação.

O código foi inteiramente feito em html, css e typescript logo, framework Angular.

2 Caracterização

Este relatório esclarece os temas pedidos para o projeto através de uma breve introdução ao tema, seguindo-se das atividades realizadas. Cada atividade é instituída por um título, uma breve introdução e comandos ou passos necessários para a realização da atividade, quer seja na forma de texto ou imagens ilustrativas.

3 Atividades realizadas

3.1 Projetos em Angular

Para dar início ao projeto tivemos de criar uma aplicação em Angular e utilizando o Visual Code executamos o seguinte comando para criar a aplicação.

ng new projeto *--routing*

Nota que no comando supracitado, o *--routing* serve para que o site carregue em apenas uma página, funcionando assim quase como uma aplicação móvel.

Em seguida criamos componentes para cada uma das páginas presentes no nosso site e para isso executamos este comando.

ng generate create components/jogo

Agora que a base do projeto em Angular foi criado podemos começar por demonstrar como cada página foi criada/programada.

3.2 Blackjack

Começaremos por falar do jogo do Blackjack.

Para preparar todas as funcionalidades do Blackjack tivemos de preparar primeiro a base da página com o HTML e alguns aspectos da aparência da página.

Começamos então por definir as cartas tanto do Dealer quanto do Jogador.

```
<div class="dealer" #dealer>
  <img src="../../../assets/Images/card.png" #card1>
  <img src="../../../assets/Images/card.png" #card3>
  <img src="../../../assets/Images/card.png" #card5>
  <img src="../../../assets/Images/card.png" #card7>
  <img src="../../../assets/Images/card.png" #card9>

</div>
<div class="player" #player>
  <img src="../../../assets/Images/card.png" #card0>
  <img src="../../../assets/Images/card.png" #card2>
  <img src="../../../assets/Images/card.png" #card4>
  <img src="../../../assets/Images/card.png" #card6>
  <img src="../../../assets/Images/card.png" #card8>
</div>
```

Como diz a regra do jogo Blackjack, as cartas são retiradas do baralho e dadas tanto ao Dealer quanto ao Jogador de forma alternada, por isso que cada carta é marcada com

referências de números alternados, por exemplo o Jogador fica com as cartas com a carta *0, 2, 4, 6* e *8*, enquanto o Dealer fica com as restantes cartas.

Em seguida definimos os botões para que o jogador pudesse decidir dar hit (pedir outra carta) ou então stay (ficar com as cartas que já tem na mão).

Para isso criamos uma *div* para que todo o código ficasse mais organizado e definimos esse *div* com a classe *"decision".* Dentro desse *div* definimos dois <*svg*> que servem, como acima explicado, para que o jogador possa escolher como jogar.

```
<div class="decision" #decisao>
  <svg (click)="hit()" style="fill: green" enable-background="new 0 0 477.867 477.867" version="1.1" viewBox="0 0 477.87 477.87" xml:space
  <path d="m392.53 0h-307.2c-47.105 0.056-85.277 38.228-85.333 85.333v307.2c0.056 47.105 38.228 85.277 85.333 85.333h307.2c47.105-0.056 85.2
  <path d="M324.267,221.867H256V153.6c0-9.426-7.641-17.067-17.067-17.067s-17.067,7.641-17.067,17.067v68.267H153.6    c-9.426,0-17.067,7.641-
  </svg>
  <svg (click)="stay()"style="fill: red" height="512pt" viewBox="0 0 512 512" width="512pt" xmlns="http://www.w3.org/2000/svg"><path d="m256
</div>
```

Após a implementação dos botões de escolha configuramos as moedas de aposta, onde cada uma aumenta a aposta total do jogador dependendo do seu valor. Para isso usamos *event onclick* definido no código como (click) seguido então da função para incrementar a aposta com o valor definido pelas moedas, para isso usamos as funções aumentarAposta().

É importante ressaltar que o jogador só pode ter no máximo 5 cartas pois, e só pode atingir no máximo 21 pontos , de acordo com as regras do Blackjack. Caso passe dos 21 pontos, o jogador perde.

Em relação ao dealer, o limite de cartas e o máximo de pontos é o mesmo.

Se o dealer tiver 16 ou menos pontos com as suas cartas, este é obrigado a virar uma nova carta.

Caso o dealer e o jogador tenha o mesmo número de pontos é considerado empate e o valor apostado volta para o jogador. No caso de ambos os jogadores não excedarem os 21 pontos ganha quem tiver a melhor mão, ou seja, mais pontos.

Na eventualidade do jogador ter "BlackJack" nas duas primeiras cartas(21 pontos) o jogador e o dealer não tenha 21 pontos a aposta do jogador é multiplicada por 2.5.

```
<div class="chips" #chips *ngIf="showChips == false">
    <img (click)="aumentarAposta(5)" src="../../../assets/Images/chip5.png">
    <img (click)="aumentarAposta(10)" src="../../../assets/Images/chip10.png">
    <img (click)="aumentarAposta(25)" src="../../../assets/Images/chip25.png">
    <img (click)="aumentarAposta(100)" src="../../../assets/Images/chip100.png">
    <img (click)="aumentarAposta(500)" src="../../../assets/Images/chip500.png">
</div>
```

Para finalizar a parte do HTML definimos um *div* para mostrar os pontos do Dealer e do Jogador de acordo com as cartas que eles obtiveram. Além disso, este *div* mostra também o saldo restante do jogador e é atualizado de acordo com o resultado da jogada.

Para mostrar tanto o saldo quanto os pontos atuais são usados *moustache notations*, passando a informação que queremos apresentar, por exemplo {{this.saldo.Playersaldo.saldo}} que mostra o saldo do jogador guardado no serviço.

Por fim contamos com dois botões para que o jogador possa escolher se quer sair da aposta ou continuar a apostar utilizando as funções *reset()* e *leave()*.

```
<div class="pontos">
  <div class="pontosDealer">
    <p>Dealer</p>
    <p>{{this.pontosDealer}}</p>
  </div>
  <div class="pontosPlayer">
    <p>Player</p>
    <p>{{this.pontosPlayer}}</p>
  </div>
  <div class="saldos">
    <p>Saldo</p>
    <p>{{this.saldo.Playersaldo.saldo}}</p>
    <p>Aposta</p>
    <p>{{this.aposta}}</p>
  </div>
</div>
<div class="result" #result>
  <h1 #h1 >Win</h1><h1> {{this.aposta}}$</h1>
  <div class="buttons">
    <button (click)="reset()">Again</button>
    <button (click)="leave()">Leave</button>
  </div>
</div>
```

Em relação ao css, decidimos manter a aparência comum de um jogo de Blackjack num tema preto e dourado, parecendo assim ser um jogo mais premium. Apesar de não ter ficado 100% responsivo, o blackjack ficou bastante utilizável mesmo para utilizadores com ecrãs mais reduzidos.

```css
.chips{
    width: 20%;
    height: 70%;
    position: absolute;
    top: 50%;
    left: 90%;
    transform: translate(-50%, -50%);
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: space-evenly;
}
```

Já no componente do typeScript começamos por chamar os serviços a serem utilizados neste *components* e igualamos as variáveis aos serviços para poder usar os objetos ou variáveis como serviço público.

```typescript
import { BlackjackService } from 'src/app/services/blackjack.service';
import { SaldoService } from 'src/app/services/saldo.service';

@Component({
  selector: 'app-jogo',
  templateUrl: './jogo.component.html',
  styleUrls: ['./jogo.component.css']
})
export class JogoComponent implements OnInit {

  saldo : SaldoService
  constructor(private getDeck: BlackjackService, private getSaldo : SaldoService) {
    this.saldo = getSaldo;
  }
```

Em seguida utilizamos *@viewchild* que serve para ir buscar os elementos ao HTML (neste caso as cartas) sem utilizar JavaScript, que é a melhor maneira de o fazer em Angular. Com os @viewchild podemos alterar as características dos elementos HTML.

```typescript
ngAfterViewInit() : void{
  this.cartas2.push(this.card0);
  this.cartas2.push(this.card1);
  this.cartas2.push(this.card2);
  this.cartas2.push(this.card3);
  this.cartas2.push(this.card4);
  this.cartas2.push(this.card5);
  this.cartas2.push(this.card6);
  this.cartas2.push(this.card7);
  this.cartas2.push(this.card8);
  this.cartas2.push(this.card9);

}
```

```typescript
@ViewChild('card0') card0 : any;
@ViewChild('card1') card1 : any;
@ViewChild('card2') card2 : any;
@ViewChild('card3') card3 : any;
@ViewChild('card4') card4 : any;
@ViewChild('card5') card5 : any;
@ViewChild('card6') card6 : any;
@ViewChild('card7') card7 : any;
@ViewChild('card8') card8 : any;
@ViewChild('card9') card9 : any;
```

A função start() vai buscar ao serviço *blackjack.service.ts* um baralho e guarda-o numa classe o deck_id.

```
start(){ // vais buscar um getdeck e
this.getDeck.getdeck().subscribe(
  x =>{
   this.deckid = new Deckinit(x);
   this.getDeckCards();
  })
}
```

A função *getDeckCards() vai buscar as cartas conforme o código do deck_id faz a conversão dos dados para a classe Card.*

```
cartasUso : Array<Card> = [];
getDeckCards(){
  if(this.deckid)
    this.getDeck.getCards(this.deckid.deck_id).subscribe(
      data =>
      {this.cartasUso = data['cards'].map(y=> new Card(y));
      });
}
```

A função *verificar()* serve para verificar os pontos obtidos pelo jogador. Caso os pontos do jogador sejam maiores que 21 a função retorna ***loser*** para que seja mostrado na tela a mensagem de derrota e para que seja deduzido do saldo.

Caso os pontos do jogador sejam menores ou iguais a 21 e os do dealer também, porém os pontos do jogador forem maiores que os pontos do dealer então a função retorna ***winner*** para que seja apresentado na tela a mensagem de vitória e para que seja adicionado o valor ganho pelo jogador.

Se tantos os pontos do jogador quanto do dealer forem iguais e menores que 21, a função devolve ***tie***, mostrando assim uma mensagem na tela a dizer que foi empate e é nos devolvido o valor da aposta.

Caso os pontos do jogador sejam menores ou iguais a 21 e os do dealer também, porém os pontos do jogador forem menores que os pontos do dealer então a função retorna ***loser*** para que seja apresentado na tela a mensagem de derrota e para que seja deduzido o valor apostado pelo jogador.

Caso os pontos do dealer sejam maiores do que 21 e os pontos do jogador forem menores que 21, a função devolve **winner** para que seja apresentado na tela a mensagem de derrota, adicionando assim o pontos ao jogador.

Por fim, tanto se o jogador quanto o dealer tiveram BlackJack (as duas primeiras cartas serem um ás e um 10 [rei, dama ou valete]) e o outro não conseguir igualar o número de pontos a função devolve **winner** ou **loser** consoante o resultado.

```
verificar(){
  if (this.pontosPlayer > 21) {
    return 'loser'
  } else if (this.pontosPlayer < 21 && this.pontosDealer < 21 && this.pontosPlayer > this.pontosDealer) {
    return 'winner'
  } else if (this.pontosPlayer == this.pontosDealer) {
    return 'tie'
  } else if (this.pontosPlayer < 21 && this.pontosDealer < 21 && this.pontosPlayer < this.pontosDealer) {
    return 'loser'
  } else if (this.pontosDealer > 21) {
    return 'winner'
  } else if (this.pontosDealer == 21) {
    return 'l this: this
  } else if (this.pontosPlayer == 21) {
    return 'blackjack'
  }
  return "0";
}
```

Em seguida temos a função *play(),* em que o seu objetivo é simplesmente para começar o jogo e assim que o jogo começa ele fica com o atributo *display none* para que ele desapareça do ecrã.

```
// Ao clickar no botao "Start" o jogo  vai começar e o mesmo é ocultado com display none
play(){
  this.showChips = true;
  this.startdiv.nativeElement.style.display = "none";
  this.start();
  this.givecards();

}
```

A função *givecards()* é a função que dá as duas primeiras cartas tanto ao dealer quanto ao jogador alternadamente mantendo a segunda carta do dealer virada para baixo.

Esta função permite:

- Virar a segunda carta do dealer assim que o jogador não fizer mais jogadas;
- Verificar se a carta que o jogador recebe é um ás e se caso for verificar se os pontos do jogador já passam dos 21 e se caso passar, o valor do ás deixa de ser 11 para ser 1;
- Incrementa os pontos do jogador à medida que ele vai pedindo cartas;

```
givecards(){
  if(this.contador != 3){
    //mostra a face da carta
    this.cartas2[this.contador].nativeElement.src = this.cartasUso[this.contador].image;
    this.cartas2[this.contador].nativeElement.style.display = "block";
    //Condiçao para verificar se a carta que o "jogadorAtual" receber um "As" e os pontos somados ultrapassarem 21 entao o "As" vale 1 ponto - Segundo Regras Oficiais
    if(this.contador % 2 == 0){
      if(parseInt(this.cartasUso[this.contador].value) == 11 && (parseInt(this.cartasUso[this.contador].value) + this.pontosPlayer) > 22){
        this.cartasUso[this.contador].value = "1";
      }
      //incrementar os pontos do Player
      this.pontosPlayer += parseInt(this.cartasUso[this.contador].value);
    }else{
      if(parseInt(this.cartasUso[this.contador].value) == 11 && (parseInt(this.cartasUso[this.contador].value) + this.pontosPlayer) > 22){
        this.cartasUso[this.contador].value = "1";
      }
      this.pontosDealer += parseInt(this.cartasUso[this.contador].value);
    }
    this.contador++;
    // uma pequena pausa de 600 milisegundos exclusivamente para aspectos visuais e logicos
    setTimeout(() => {
      this.givecards();
    }, 600);
  }else {
    //caso a proxima carta a ser dada seja a "quarta" ou seja a segunda carta do dealer apenas mostra o verso
    this.cartas2[this.contador].nativeElement.src = "../../../assets/Images/card.png";
    this.cartas2[this.contador].nativeElement.style.display = "block";
    this.decisao.nativeElement.style.display = "flex";
    this.contador++;
  }
};
```

Em seguida temos as funções *hit()* e *stay()* que determinam as jogadas do jogador.

Caso o jogador queira pedir mais uma carta ele utiliza a função *hit()* que lhe dá uma carta e se ao receber esta carta o player ultrapasse os 21 pontos a jogada dele termina e avança para o dealer.

Caso o jogador não queira mais pedir cartas é utilizada então a função *stay()* dando assim a vez ao dealer para começar a jogar.

```
//Caso o jogador escolha "hit" vai lhe ser dada uma nova carta, se o jogador tiver blackjack, ou seja, 21 pontos nao podera pedir uma carta
hit(){
  if(this.pontosPlayer < 21){
    this.cartas2[this.contador].nativeElement.src = this.cartasUso[this.contador].image;
    this.cartas2[this.contador].nativeElement.style.display = "block";
    if(parseInt(this.cartasUso[this.contador].value) == 11 && (parseInt(this.cartasUso[this.contador].value) + this.pontosPlayer) > 22){
      this.cartasUso[this.contador].value = "1";
    }
    this.pontosPlayer += parseInt(this.cartasUso[this.contador].value);
    this.contador+=2;
    /*Verificar se é maior ou igual ou nao ----------------------------------------------------------------------------------------------------*/
    if(this.pontosPlayer > 21){
      this.contador = 3;
      setTimeout(() => {
        this.dealerCards();
      }, 600);
    }
  }
}
```

```
// Caso o player escolha ficar com as cartas que tem a jogada passa para o dealer e o contador é ajustado ás cartas do dealer
stay(){
  this.stayoption = true;
  this.contador = 3;
  this.dealerCards();
}
```

Assim que o jogador escolher stay ou os seus pontos não ultrapassarem 21, a jogada passa para o dealer, é então que a função *dealercards()* é executada.

Esta função dá cartas ao dealer e caso os seus pontos forem iguais ou inferiores a 16 e ainda não tiver ganho ao jogador, o dealer vai receber cartas -Regras Oficiais- .

Além disso, esta função verifica se caso o que foi descrito em cima não se confirme, ela chama uma outra função que verifica o resultado do jogo e finaliza a jogada e caso o jogador ultrapasse os 21 pontos antes da jogada do dealer, a segunda carta do dealer é mostrada.

```
  //Caso o jogador escolha "stay" ou se o player nao ultrapassar os 22 pontos a jogada passa para o dealer
dealerCards(){
  if(this.pontosPlayer < 22 || this.stayoption == true){
    //se o dealer tiver 16 ou menos pontos e ainda nao tiver ganho ao player , é obrigatorio ele receber uma nova carta - Regras oficiais
    if(this.pontosDealer <= 16 && this.pontosDealer < this.pontosPlayer || this.pontosDealer <= 16 && this.pontosDealer == this.pontosPlayer){
      this.cartas2[this.contador].nativeElement.src = this.cartasUso[this.contador].image;
      this.cartas2[this.contador].nativeElement.style.display = "block";
      if(parseInt(this.cartasUso[this.contador].value) == 11 && (parseInt(this.cartasUso[this.contador].value) + this.pontosDealer) > 22){
        this.cartasUso[this.contador].value = "1";
      }
      this.pontosDealer += parseInt(this.cartasUso[this.contador].value);
      this.contador+=2;
      setTimeout(() => {
        this.dealerCards();
      }, 600);
      //caso o if em cima ja nao se verificar, iremos chamar a funçao para saber quem foi o vencedor e passamos para o fim da jogada
    }else{
      this.verify  = this.verificar();
      this.fim(this.verify);
    }
    //Caso o jogador ultrapasse os 22 pontos a ultima carta do dealer é mostrada
  }else{
    this.cartas2[this.contador].nativeElement.src = this.cartasUso[this.contador].image;
    this.cartas2[this.contador].nativeElement.style.display = "block";
    this.verify  = this.verificar();
    this.fim(this.verify);
  }

}
```

A função *fim()* é a função que é chamada dentro da função *dealerCards()* que verifica o resultado do jogo e mostra ao jogador o quanto ele ganhou ou perdeu dependendo do resultado da jogada.

```
// Na funçao fim é mostrado o resultado da jogada e o valor da aposta ganhou ou perdido respectivamente
fim(verify){
  this.result.nativeElement.style.display = "flex";
  if(verify == "winner"){
    this.h1.nativeElement.textContent ="Winner";
    this.saldo.Playersaldo.saldo += this.aposta
  }else if(verify == "loser"){
    this.h1.nativeElement.textContent ="Lose";
    this.aposta -= 2*this.aposta;
    this.saldo.Playersaldo.saldo += this.aposta
  }else if(verify == "tie"){
    this.h1.nativeElement.textContent ="Tie";
  }else if(verify == "blackjack"){
    this.h1.nativeElement.textContent ="BlackJack";
    this.saldo.Playersaldo.saldo += 1.5*this.aposta
  }
}
```

A seguir temos a função *reset()* que é a função que iguala todas as variáveis e esconde as cartas para que seja possível fazer uma nova jogada.

```
//Reset é a funçao para igualar todas as variaveis necesarias e esconder as cartas para voltar a fazer uma nova jogada
reset(){
  setTimeout(() => {
    this.contador = 0;
    this.pontosPlayer = 0;
    this.pontosDealer = 0;
    this.stayoption = false;
    this.aposta = 0;
    this.result.nativeElement.style.display = "none";
    for (let i = 0; i < 10; i++) {
    this.cartas2[i].nativeElement.style.display = "none";
    }
    this.startdiv.nativeElement.style.display = "flex";
    this.decisao.nativeElement.style.display="none";
    this.showChips = false;
    localStorage.setItem('balance', this.saldo.Playersaldo.saldo)
  }, 600);
}
```

Por fim temos as funções *aumentarAposta()* e *leave().*

A função *aumentarAposta()* recebe o valor de cada uma das fichas em HTML e junta esse valor a apostar.

A função *leave()* atualiza o valor do saldo dentro do *localstorage* e volta para a página home.

4 Crypto

Sobre o Crypto, muito resumidamente simula os investimentos em crypto, onde o usuário pode comprar e mais tarde vender, após ter subido ou descido de valor.

Começamos por criar a base em html.

Na seguinte imagem podemos ver a parte do loading. Onde criamos uma pequena animação de passagem entre componentes.

```html
<!-- before screen -->
<div *ngIf="beforeScreen == false" class="before-screen column-flex-center position-fix" #beforeScreen>
    <h1>Stocks</h1>
    <div class="spinner">
        <div class="bar1"></div>
        <div class="bar2"></div>
        <div class="bar3"></div>
        <div class="bar4"></div>
        <div class="bar5"></div>
        <div class="bar6"></div>
        <div class="bar7"></div>
        <div class="bar8"></div>
        <div class="bar9"></div>
        <div class="bar10"></div>
        <div class="bar11"></div>
        <div class="bar12"></div>
    </div>
</div>
```

ngIf - é uma diretiva estrutural que permite, fazer aparecer ou não uma secção de código.

No nosso caso, utilizamos para definir quando o div de transição é apagado.

```css
.before-screen {
    width: 100%;
    height: 100%;
    position: absolute;
    top: 50%;
    left: 50%;
    background-position: center;
    background-repeat: no-repeat;
    background-size: 100% auto;
    background-color: black;
    animation: slide-out-top 1s cubic-bezier(0.550, 0.085, 0.680, 0.530) both;
    z-index: 2;
}

.before-screen h1 {
    font-size: 5vw;
    color: whitesmoke;
    transition-duration: 0.5s;
}

@keyframes slide-out-top {
    0% {
        transform: translateY(0) translate(-50%, -50%);
    }
    100% {
        transform: translateY(-150%) translate(-50%, -50%);
        display: none;
    }
}
```

```css
.spinner {
    position: relative;
    width: 54px;
    height: 54px;
    padding: 10px;
    border-radius: 10px;
}

.spinner div {
    width: 5%;
    height: 16%;
    background: #FFF;
    position: absolute;
    left: 49%;
    top: 43%;
    opacity: 0;
    border-radius: 50px;
    box-shadow: 0 0 3px rgba(0, 0, 0, 0.2);
    animation: fade 1s linear infinite;
}

@keyframes fade {
    from {
        opacity: 1;
    }
    to {
        opacity: 0.25;
    }
}
 .spinner .bar1 {
    transform: rotate(0deg) translate(0, -130%);
    animation-delay: 0s;
 }

 .spinner .bar2 {
    transform: rotate(30deg) translate(0, -130%);
    animation-delay: -0.9167s;
 }

 .spinner .bar3 {
    transform: rotate(60deg) translate(0, -130%);
    animation-delay: -0.833s;
 }

 .spinner .bar4 {
    transform: rotate(90deg) translate(0, -130%);
    animation-delay: -0.7497s;
 }

 .spinner .bar5 {
    transform: rotate(120deg) translate(0, -130%);
    animation-delay: -0.667s;
 }

 .spinner .bar6 {
    transform: rotate(150deg) translate(0, -130%);
    animation-delay: -0.5837s;
 }

 .spinner .bar7 {
    transform: rotate(180deg) translate(0, -130%);
    animation-delay: -0.5s;
 }
```

Para que a animação funcionasse, tivemos de usar *transforms* para modificar as suas posições e *delays* para modificar o tempo que cada barra começasse a rodar criando assim um efeito de *loading*.

Na seguinte imagem, temos a parte principal do componente, onde temos, uma coluna lateral com as criptomoedas mais conhecidas e um input para fazer uma pesquisa de qualquer outra criptomoeda disponível na api, um gráfico na parte direita superior, e o comprar e vender na parte direita inferior.

Usamos o grid, pois foi uma maneira mais organizada, em comparação ao display flex.

```
    <!-- stocks -->
    <div class="left column-space-evenly">
        <div class="input display-grid-center" #error>
            <input type="text" #search (keyup.enter)="getStock(search.value)" placeholder="Search for a crypto">
        </div>
        <div class="stocks column-space-evenly">
            <div class="stock row-space-evenly" *ngFor="let item of cryptos">
                <h1>{{item.ticker.base || "COIN"}}</h1>
                <h2>Price <br> {{item.ticker.price || 0}}€</h2>
                <h2>Change <br> {{item.ticker.change || 0}}€</h2>
            </div>
        </div>
    </div>
    <div class="right column-space-evenly">
        <div class="top column-space-evenly">
            <div class="graph">
                <img src="assets/images/chart.png" alt="graph">
            </div>
            <div class="bottom-showdown row-space-evenly">
                <h1>{{this.stock.ticker.base || "COIN"}}</h1>
                <h2>Price <br> {{this.stock.ticker.price || 0}}€</h2>
                <h2>Change <br> {{this.stock.ticker.change || 0}}€</h2>
            </div>
        </div>
        <div class="bottom display-grid-center">
            <div class="balance display-grid-center">
                <h2>Balance <br>{{this.saldoService.balance || 0}}€</h2>
            </div>
            <div class="button column-flex-center">
                <button class="buy" (click)="buy(buyInput)">Buy</button>
                <div class="input display-grid-center">
                    <input type="text" placeholder="Crypto to buy" #buyInput>
                </div>
            </div>
            <div class="button column-flex-center">
                <button class="sell" (click)="sell(sellInput)">Sell</button>
                <div class="input display-grid-center">
                    <input type="text" placeholder="Crypto to sell" #sellInput>
                </div>
            </div>
        </div>
    </div>
</body>
```

Em termos de responsividade, também está muito completo, apesar de não estar 100%, está completamente utilizável ao usuário.

Já no typescript, começamos por declarar as variáveis e importar os serviços necessários.

Tivemos que igualar os serviços a variáveis para que pudessem ser utilizados fora daqueles componentes, pondo assim o serviço em modo público pois por padrão são

```
//declaraçao de variaveis e import dos servicos
stocksService : StocksApiService
saldoService : SaldoService
constructor(private service : StocksApiService, private saldo : SaldoService) {
  this.stocksService = service
  this.saldoService = saldo
}

@ViewChild("error") input : any;
@ViewChild("search") search : any;

beforeScreen : boolean = false
```

privados

Ao iniciar o componente, desliga o 'before Screen' e chama a função ‘getDefaultStocks’ e guarda o balance do local Storage no nosso serviço.

```
// ao iniciar desliga o beforescreen, chama a funçao getDefaultStocks e vai buscar o balance ao localstorage
ngOnInit(): void {
  setTimeout(() => {
    this.beforeScreen = true
  }, 1000);
  this.getDefaultStocks()
  this.saldo.balance = localStorage.getItem("balance")
}
```

Depois, declaramos o objeto do stock.

```
//declaraçao do objeto do stock
stock : any = {
  error: "",
  success: false,
  ticker: {
    base: "",
    price: 0,
    change: 0,
    target: "",
    volume: 0
  },
  timestamp: 0
}
```

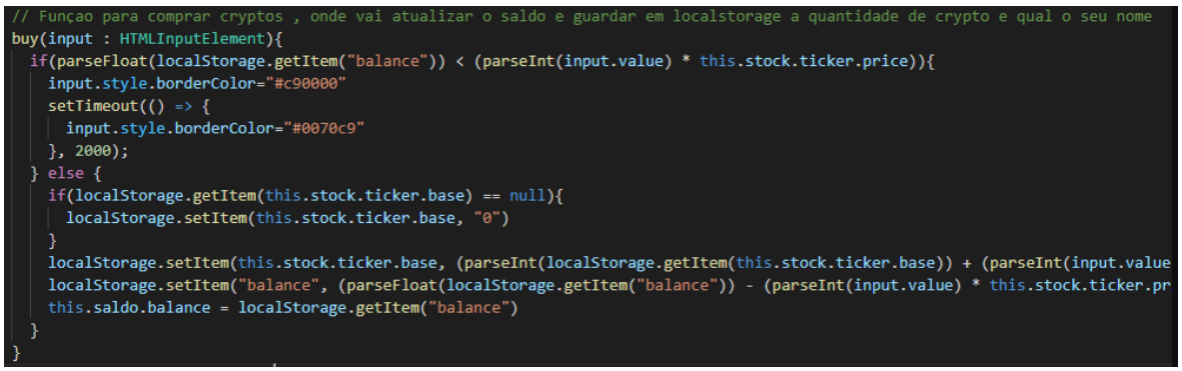
Declaramos um array com as 6 criptomoedas mais famosas do momento, pois podem ser as que os utilizadores mais tenham interesse, e também declaramos um array vazio para adicionar as moedas que o utilizador pesquisa.

```
//Declaraçao do array das moedas que vao aparecerp por default
searchForCoins : string[] = ['btc', 'eth', 'xrp', 'ada', 'doge', 'bch']
// Array vazio onde vai ficar a informaçao da api de todas as coins
cryptos : any = []
```

ngFor - O array criado acima é utilizado junto com o ngFor presente no HTML que é uma diretiva estrutural presente em angular utilizada para construir secções de código presentes numa lista.

```
        <div class="stock row-space-evenly" *ngFor="let item of cryptos">
            <h1>{{item.ticker.base || "COIN"}}</h1>
            <h2>Price <br> {{item.ticker.price || 0}}€</h2>
            <h2>Change <br> {{item.ticker.change || 0}}€</h2>
        </div>
    </div>
```

A função “buy” foi construída para comprar cryptos, que atualiza o saldo e guarda no “localStorage” o que compramos e a sua quantidade.



```
// Funçao para comprar cryptos , onde vai atualizar o saldo e guardar em localstorage a quantidade de crypto e qual o seu nome
buy(input : HTMLInputElement){
  if(parseFloat(localStorage.getItem("balance")) < (parseInt(input.value) * this.stock.ticker.price)){
    input.style.borderColor="#c90000"
    setTimeout(() => {
      input.style.borderColor="#0070c9"
    }, 2000);
  } else {
    if(localStorage.getItem(this.stock.ticker.base) == null){
      localStorage.setItem(this.stock.ticker.base, "0")
    }
    localStorage.setItem(this.stock.ticker.base, (parseInt(localStorage.getItem(this.stock.ticker.base)) + (parseInt(input.value
    localStorage.setItem("balance", (parseFloat(localStorage.getItem("balance")) - (parseInt(input.value) * this.stock.ticker.pr
    this.saldo.balance = localStorage.getItem("balance")
  }
}
```

Já a função “sell” faz o contrário, ou seja, vende os cryptos, usando as mesmas técnicas do “buy”.

```
sell(input : HTMLInputElement){
  if(localStorage.getItem(this.stock.ticker.base) == null || parseInt(localStorage.getItem(this.stock.ticker.base)) < parseInt(input.value)){
    input.style.borderColor="#c90000"
    setTimeout(() => {
      input.style.borderColor="#0070c9"
    }, 2000);
  } else {
    localStorage.setItem(this.stock.ticker.base, (parseInt(localStorage.getItem(this.stock.ticker.base)) - (parseInt(input.value))).toString())
    localStorage.setItem("balance", (parseFloat(localStorage.getItem("balance")) + (parseInt(input.value) * this.stock.ticker.price)).toString())
    this.saldo.balance = localStorage.getItem("balance")
  }
}
}
```

Caso o utilizador não tenha saldo suficiente para a compra, essa não poderá ser efetuada, e se o caso for que não tenha criptomoedas em stock, também não conseguirá vendê-las para aumentar o seu saldo.

Para segurança da aplicação, o input da pesquisa das criptomoedas é desabilitado temporariamente após ser executado.

5 404

Para finalizar temos a página 404, em que o seu propósito é informar ao utilizador que o conteúdo que ele procura não existe ou não se encontra disponível.

Para isso foi preciso, além de criar o componente e trabalhar todo o seu HTML e CSS, definir no ***Routing*** que este componente tenha este efeito. Para isso, no ***app-routing.module.ts*** foi preciso definir o path **"**"**(path predefinido para indicar as páginas não encontradas) para redirecionar para o componente 404.

```
const routes: Routes = [
  {path: "", component: HomeComponent},
  {path: "home", component: HomeComponent},
  {path: "stocks", component: StocksComponent},
  {path: "blackjack", component: JogoComponent},
  {path: "404", component: FourOfourComponent},
  {path: "**", redirectTo: "404" }

];
```

6 Serviços

6.1 Blackjack Service

O serviço do blackjack foi criado para fazer a ligação a api das cartas, onde tem duas funções, "getdeck" e "getCards".

O "getdeck" vai buscar um id à api e o "getCards" pega em 10 cartas da api conforme o id do deck.

```
linkdeck ="https://deckofcardsapi.com/api/deck/new/shuffle/?deck_count=1";

getdeck(){
  return this.httpAsk.get(this.linkdeck)
}

getCards(deckid : string){
  let link = "https://deckofcardsapi.com/api/deck/"+ deckid +"/draw/?count=10";
  return this.httpAsk.get(link);
}
```

6.2 Crypto Service

Para o jogo do Stock precisávamos uma maneira de ir buscar informações à API e além disso guardas as informações, por isso foi necessário criar um serviço.

Neste serviço basicamente definimos o link onde o jogo iria buscar as criptomoedas e os seus preços e definimos que a moeda básica de compra e venda dessas moedas seria o euro.

Dentro deste serviço temos então um função *getStock()* que, recebendo uma informação inserida no input pelo HTML, faz uma pesquisa na API e procura a moeda introduzida.

```
stocks : any = {
  link: "https://api.cryptonator.com/api/ticker/",
  currency: "eur"
}

getStock(search : string){
  return this.http.get(this.stocks.link + search + "-" + this.stocks.currency)
}
}
```

6.3 Saldo Service

O nosso serviço de saldo serve para interligar o dinheiro ganho a apostar no Blackjack com o dinheiro que podemos utilizar para comprar as criptomoedas. Para isso temos a variável *balance* que guarda o saldo compartilhado entre os dois jogos que guarda a informação no localStorage (guarda as informações desejadas no browser).

```
balance : string = localStorage.getItem("balance")
```

7 Classes

As classes funcionam como objetos no nosso caso nós criamos uma classe para guardar o deck_id e outra para guardar arrays das cartas e das suas características.

```
export class Card {
  image: string;
  value: string;
  suit: string;
  code: string;

  constructor(cartaRecebida : any){
    if(cartaRecebida.value == 0 || cartaRecebida.value == "JACK" || cartaRecebida.value == "QUEEN" || cartaRecebida.value == "KING"){
      cartaRecebida.value = 10;
    }
    if(cartaRecebida.value == "ACE"){
      cartaRecebida.value = 11;
    }
    this.image = cartaRecebida.image;
    this.value = cartaRecebida.value;
    this.suit = cartaRecebida.suit;
    this.code = cartaRecebida.code;
  }


}
```

8 Conclusão

Neste trabalho abordamos vários, senão todos os nossos conhecimentos adquiridos nas aulas para o desenvolvimento deste projeto, onde o projeto ficou funcional e concluímos que os conhecimentos adquiridos nas aulas foram muito bem absorvidos e solidificados para a concretização deste trabalho.

Cumprimos quase todos os nossos objetivos pessoais e também os propostos pelo professor para a realização do mesmo, faltando apenas ter criado uma carteira no jogo do crypto para sabermos a quantidade de moedas que compramos.

Este trabalho foi bastante importante para algumas informações frescas ficarem solidificadas, para na nossa perspetiva, o desenvolvimento em grupo e o aperfeiçoamento de matérias anteriormente dadas.

Em termos de dificuldades, houveram algumas, como por exemplo a condição que faz virar as cartas uma de cada vez, sendo que a primeira tinha de ser a do jogador, a segunda do dealer, novamente o jogador e novamente o dealer, tendo que estar viradas para baixo, onde as próximas cartas teriam de ser para o jogador, e só consoante os pontos ou decisão do jogador para manter as cartas e não pedir mais é que o dealer jogava.

9 Referências bibliográficas

W3Schools:

https://www.w3schools.com/

CSS Reference:

https://cssreference.io/