Interface Pessoa-Máquina

Licenciatura em Engenharia Informática

Ficha Prática #08

Rafael Braga
d13414@di.uminho.pt

Daniel Murta
d6203@di.uminho.pt

José Creissac Campos
jose.campos@di.uminho.pt

(v. 2025.1)

Conteúdo

1 Objetivos **2**

2 Routing e pedidos HTTP em Vue.js **2**

2.1 Routing 2

2.2 Pedidos HTTP 3

2.2.1 API Fetch 3

2.2.2 Métodos HTTP 5

3 Exercícios **5**

3.1 Jogo 4 em linha 6

1 Objetivos

1. Praticar a utilização da *framework* Vue.js, em particular foco no Vue Router e pedidos HTTP.

2 Routing e pedidos HTTP em Vue.js

Nesta ficha irá praticar o condigiuração de *routing* (roteamento) e a realização de pedidos HTTP em Vue.js.

2.1 Routing

O *routing* é um aspeto importante da construção de *Single Page Applications* (SPAs). Permite associar endereços URL a diferentes componentes da aplicação, possibilitando que estes sejam acedidos diretamente, através desses URL.

O *routing* no Vue.js é gerido pelo Vue Router, o *router* oficial para Vue.js. Para usar o Vue Router, primeiro é necessário instalá-lo. Se estiver a começar um novo projeto, pode adicioná-lo ao criar o projeto com Vite, selecionando a opção correspondente. Num projeto existente, é possível adicioná-lo através do Node.js:

```
% npm install vue-router@latest
```

No caso do projeto fornecido com esta ficha o Vue Router já está configurado.

A configuração do *router* é feita através da função `createRouter`:

```
import { createRouter, createWebHistory } from 'vue-router'

const router = createRouter({
    history: createWebHistory(),
    routes: [
        { path: '/', component: Home }
        { path: '/about', component: About }
    ]
});
```

Neste caso, a rota `"/"` é mapeada para o componente `Home` e a rota `"/about"` para o componente `About`. A função `createWebHistory` indica que deverá ser usada a gestão de história de navegação do *browser*.

Para que o *router* seja utilizado, deverá ser integrado na aplicação:

```
const app = createApp(App)

app.use(router)   // usar o router em app

app.mount('#app')
```

Para indicar, onde, no `template` de `App`, os componentes `Home` e `About` devem ser apresentados, o Vue Router fornece a tag `<router-view>`. Para navegação entre rotas fornece a tag `<router-link>`. Assim, o `template`:

```
<template>
    <router-link to="/">Home</router-link>
    <router-link to="/about">About</router-link>

    <router-view></router-view>
</template>
```

corresponde a uma interface que permite navegar entre `Home` e `About`, sendo o conteúdo do componente selecionado apresentado abaixo dos links de navegação.

Mais exemplos de configuração de rotas, incluindo rotas aninhadas, podem ser consultados nos exemplos fornecidos nas aulas teóricas.

2.2 Pedidos HTTP

O Vue.js não tem suporte específico para a realização de pedidos HTTP. Para os realizar recorremos a *frameworks* como Axios[1] ou ao suporte nativo dos *browsers* através da API Fetch. Neste tutorial iremos utilizar esta segunda opção.

2.2.1 API Fetch

A API Fetch permite realizar pedidos HTTP de forma assíncrona utilizando a função `fetch()`. Na sua utilização mais simples, a função aceita apenas um argumento, o URL do recurso que se deseja obter, e retorna uma promessa (*promise*) de resposta.

Uma *promise* é um objeto que representa a conclusão eventual ou falha de uma operação assíncrona. Os estados possíveis de uma *promise* são:

- `Pending`: Estado inicial, nem cumprida nem rejeitada.

[1] https://axios-http.com/, visitado em 09/04/2024.

- `Fulfilled`: Operação completada com sucesso. O método `then()` permite registar uma função para tratar a resposta obtida.
- `Rejected`: Operação falhou. O método `catch()` permite registar uma função para para tratar o erro.

O método `finally()` permite registar uma função que é executada após a *promise* ser resolvida (independentemente do seu resultado).

Um exemplo deste tipo de utilização simples é:

```
fetch('https://exemplo.com/dados')
    .then(resposta => {
        if (!response.ok) throw new Error("Erro na resposta!");
        return resposta.json();
    })
    .then(dados => console.log(dados))
    .catch(error => console.error("Erro a obter dados", error))
    .finally(() => console.log("Pedido terminado."));
```

O método `json()` devolve uma *promise* que permite obter o objeto Javascript correspondente aos dados obtidos na resposta recebida.

Outro modo de fazer pedidos HTTP é através de métodos assíncronos com o uso das funcionalidades `async/await`. Um exemplo deste tipo de utilização simples é:

```
async function fetchData() {
    try {
        const response =
            await fetch('https://exemplo.com/dados');
        if (!response.ok) {
            throw new Error("Erro na resposta!");
        }
        const dados = await response.json();
        console.log(dados);
    } catch (error) {
        console.error("Erro a obter dados", error);
    } finally {
        console.log("Pedido terminado.");
    }
}
```

2.2.2 Métodos HTTP

Um pedido HTTP pode ser de diferentes tipos. O HTTP define um conjunto de métodos (também chamados verbos HTTP) para indicar a ação desejada.

Os métodos HTTP mais relevantes para este tutorial são[2]:

- GET – usado para obter dados;
- POST – usado para enviar dados para processamento (potencialmente com efeitos laterais);
- PUT – usado para atualizar um recurso existente ou criar um novo recurso (sem efeitos laterais);
- PATCH – usado para aplicar modificações a um recurso existente.

A utilização de `fetch()` com apenas o URL corresponde a um pedido GET. Para os restantes pedidos é necessário passar um segundo parâmetro à função, com um objeto que indica o método do pedido e os dados a enviar. Um pedido PUT, por exemplo, poderia ser feito do seguinte modo:

```
fetch('https://exemplo.com/dados', {
        method: 'PUT',
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify(dados)
});
```

O método `stringify()` do objeto `JSON` transforma um objeto Javascript na sua representação textual em JSON.

Para mais informações e exemplo de utilização da API fetch aconselha-se a consulta de: https://developer.mozilla.org/en-US/docs/Web/API/fetch

3 Exercícios

Resolva os seguintes exercícios.

[2] Sobre métodos HTTP, ver também https://developer.mozilla.org/docs/Web/HTTP/Methods, visitado em 09/04/2024.

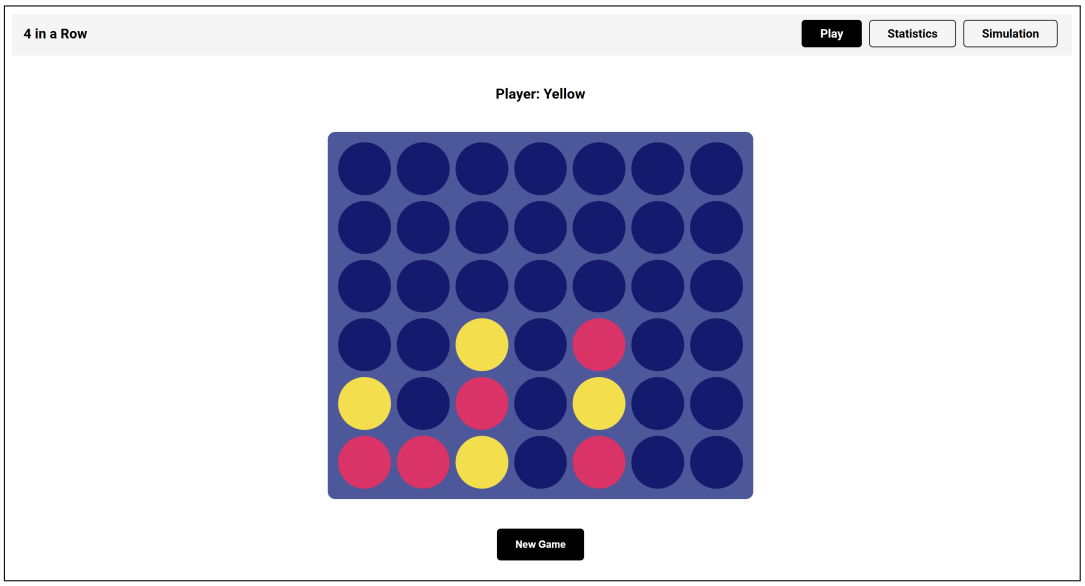

Figura 1: O jogo 4 em linha

3.1 Jogo 4 em linha

Com o conjunto de exercícios abaixo, pretende-se implementar uma aplicação que apresenta um conjunto de funcionalidades referentes ao jogo "*4 em linha*". O resultado final pretendido é o apresentado na Figura 1.

A aplicação a desenvolver deverá apresentar as seguintes funcionalidades/páginas:

- Uma página de jogo, onde será possível jogar 4 em linha e obter o resultado do jogo.
- Uma página de estatísticas que deverá apresentar o número de vitórias, para as equipas amarela e vermelha, e o número de empates. Nesta página poder-se-á também consultar o histórico dos jogos.
- A partir do histórico de jogos a aplicação deverá permitir que um utilizador selecione um jogo e consulte o estado final do tabuleiro desse jogo.
- Uma funcionalidade de simulação que deverá permitir ver uma animação das jogadas efetuadas durante o último jogo realizado.

Com esta ficha, é fornecida uma base de trabalho para a implementação do jogo. Esta, contém uma implementação parcial da aplicação (diretoria `4inarow`) e um *backend*, implementado através da ferramenta `json-server` (diretoria `backend`). O *backend* permite:

- Consultar/atualizar as estatísticas do jogo.
- Consultar/guardar o histórico dos jogos.
- Consultar/guardar as jogadas efetuadas no último jogo. Estas jogadas podem ser consultadas para efetuar a funcionalidade de simulação.

Tomando como base a implementação parcial da aplicação e a implementação do seu *backend*, fornecidas com esta ficha, resolva então os seguintes exercícios:

1. Estude a implementação do *backend* fornecido. Para tal execute, no terminal, os seguintes comandos, a partir da diretoria `backend`:

   - `npm install` - para instalar todas as dependências necessárias.
   - `npx json-server db.json` - para correr o `json-server` com a base de dados JSON `db.json`.

   Após a execução destes comandos, deverá ser apresentada, no terminal, a lista de *endpoints* disponíveis no *backend*. Verifique o bom funcionamento do *backend* efetuando pedidos a partir do seu browser para os *endpoints* listados. Analise as estruturas de dados devolvidas em cada pedido.

2. Mude agora para a diretoria `4inarow` e execute o comando `npm install`, para instalar as dependências deste outro projeto. Configure agora as rotas da aplicação, através do *Vue Router*, no ficheiro `router.js`. Para tal, complete as seguintes alíneas:

   (a) Adicione as rotas `"/game"`, `"/statistics"` e `"/simulation"` e associe a estas rotas, respetivamente, os componentes que definem as páginas[3] `GamePage`, `StatisticsPage` e `SimulationPage`. No ficheiro `App.vue` renderize o conteúdo associado às rotas. Utilize o seu browser para verificar o bom funcionamento destas rotas (aceda a cada uma delas através do seu endereço).

   (b) Acrescente à `NavBar` os *router links* associados às rotas acima e verifique que a navegação está a funcionar corretamente. Como são aplicados os estilos a cada *router link* da navegação?

[3] Por convenção, chamamos páginas aos componentes que representam as páginas da aplicação. Note que estes componentes foram colocados numa diretoria `pages`. Na diretoria `components` foram colocados componentes auxiliares, utilizados nas páginas. Esta organização, permite uma melhor estruturação do projeto.

(c) Acrescente ao *router* a rota base `"/"` que deverá ser redirecionada para a rota `"/game"`. Acrescente também uma rota que trata de caminhos não suportados que deverá mostrar o componente referente à página `NotFoundPage`.

3. Codifique o método `updateStatistics()` da página `GamePage`. Este método deverá atualizar as estatísticas no `json-server`. Para tal, utilize o método `PATCH` para enviar a atualização das estatísticas, através do seguinte *endpoint*:

   `http://localhost:3000/statistics/1`

4. Complete a funcionalidade do histórico de jogos efetuados. Para tal complete as seguintes alíneas:

   (a) Codifique o método `saveGame()` da página `GamePage`. Este método deverá salvaguardar o conteúdo da variável `game` e a data atual em formato ISO. Para tal, utilize o método `POST`, através do seguinte *endpoint*:

   `http://localhost:3000/games`

   (b) Codifique o método `getGames()` da página `StatisticsPage`. Este método deverá obter todo o histórico de jogos salvaguardado no `json-server` e deverá atualizar a variável `games` com o valor devolvido deste histórico. Para tal, utilize o método `GET`, através do seguinte *endpoint*:

   `http://localhost:3000/games`

   (c) Ainda na página `StatisticsPage`, adicione um evento de click a cada `CardComponent` que ilustra um jogo do histórico. Este evento deverá redirecionar para a rota `"/game/:id"`, sendo o valor de `":id"` o identificador do jogo associado ao `CardComponent` onde foi efetuado o click.

   (d) No ficheiro `router.js`, adicione a rota `"/game/:id"` que permite visualizar o estado final do jogo com o identificador igual ao fornecido pelo valor `":id"`. Esta rota deverá mostrar o componente referente à página de resultados do jogo, `GameResultPage`. Faça com que o identificador da rota seja passado como `props`.

   (e) Codifique o método `getGame()` da página `GameResult`. Este método deverá obter o estado final de um jogo salvaguardado no `json-server` e deverá atualizar a variável `game` com o valor devolvido. Para tal, utilize o método `GET`, através do seguinte *endpoint*:

   `http://localhost:3000/games/:id`

Em que `":id"` corresponde ao identificador recebido como `props`.

5. Complete a funcionalidade de simulação do último jogo efetuado. Para tal complete as seguintes alíneas:

   (a) Faça alterações na página `GamePage` de modo a que seja possível guardar informação sobre o jogador que iniciou o jogo (`startPlayer` – um valor boolean que indica o primeiro jogador a jogar, vermelho ou amarelo) e ainda um array de inteiros (array `plays`) que deverá conter os índices de cada coluna correspondente a uma jogada. Estes dados deverão ser guardados no `json-server`. Para tal, codifique o método `updateSimulation()`, que deverá utilizar o método `PATCH`, através do seguinte *endpoint*:

   `http://localhost:3000/simulation/1`

   (b) Codifique o método `getSimulation()` da página `SimulationPage` que deverá obter do `json-server` os dados referentes à simulação do último jogo. Estes dados deverão ser guardados na variável `simulation`.

   (c) Adicione ao botão de simular um evento de click que deverá executar o método `simulate()`. O botão deverá estar *disabled* sempre que uma simulação estiver a correr.

   (d) Codifique o método `simulate()` que deverá realizar a simulação recebida. Caso não exista nenhuma simulação disponível deverá mostrar uma mensagem apropriada. Para mostrar a simulação com maior clareza efetue cada jogada pertencente à simulação a cada 1 segundo. Para tal use a função `setInterval`. Para terminar a simulação e limpar o intervalo definido com a função mencionada, utilize a função `clearInterval`.

6. O jogo não fornece *feedback* visual sobre as operações em curso (por exemplo, carregar estatísticas ou guardar o jogo). Adicione uma mensagem de carregamento/gravação para melhorar a experiência do utilizador.