Projeto Classificatório

Processo seletivo - *Web Analytics*

PEDRO HENRIQUE **MAGALHÃES BOTELHO**

GRADUANDO EM ENGENHARIA DE COMPUTAÇÃO

QUIXADÁ - CE
2022

Conteúdo

1 Introdução

Olá! Me chamo Pedro M. Botelho e este é o relatório simplificado do projeto "Broken Database". O código está no arquivo "resolucao.js", e os arquivos de banco de dados estão na pasta "database", onde "saida.json"é a resposta para o problema do banco de dados corrompido.

O código está documentado, em língua inglesa, e explica, em si, a integridade e funcionamento do código. Nesse relatório falarei sobre questões de implementação.

2 A organização do código

O código está dividido em 6 seções, de forma a organizar da melhor forma possível o código. Cada seção traz um elemento fundamental para o código, e a separação busca seccionar o código quanto à sua utilidade. A primeira seção é apenas uma explicação da usabilidade do código. As outras seções serão discutidas em breve.

O código se utiliza bastante de *arrow functions*, já que trazem uma simplicidade visual e não contém um contexto "this". A simplicidade teve de ser, algumas vezes, desprezada para introduzir robustez ao código, como laços for...in e tratamento de erros.

A entrada e saída do sistema é feita através dos arquivos na pasta **database**/, que são manipulados utilizando constantes no código, que podem ser modificadas para referenciar outros arquivos. O código, em sim, foi feito para ser responsivo o suficiente para aplicar as modificações e validações requisitadas a qualquer arquivo JSON solicitado, desde que incluído seu caminho nas respectivas constantes.

Dito isso, veremos agora as principais do código, de forma que vejamos os principais pontos de implementação do código. A maior parte do código utilizou-se do **paradigma funcional**, para facilitar a manutenção do programa e a manipulação de dados.

3 Variáveis Constantes

Alguns valores são utilizados no decorrer de todo o código. Para assegurar o reúso de software, a robustez e a legibilidade esses valores foram colocados no começo, como forma de cabeçalho, como constantes, para a segurança do programa, já que não devem ser modificados.

Cada constante tem um propósito específico:

- **namesCorrection**: Guarda um array de pares de string, onde cada par representa um caractere a ser substituído e o novo caractere. É usado para corrigir os valores da propriedade "name"dos objetos do arquivo JSON. Poderia ter sido definida dentro da única função que a usa, fixNames(), mas foi definida em uma seção dedicada para melhorar a organização do código.
- **brokenDatabase**: Guarda o caminho do arquivo JSON a ser recuperado. Se trata do indicador do arquivo de **entrada** do sistema.
- **okDatabase**: Guarda o caminho de onde o arquivo JSON ficará após ser recuperado. Se trata do indicador do arquivo de **saída** do sistema.
- **fs**: Uma constante especial que importa o módulo de *sistema de arquivos*, do **Node.js**. Isso nos permite ter acesso ao sistema de arquivos da máquina.
- **FileError**: Um objeto utilizado para indicar um erro de arquivo. Quando o caminho indicado na constante **brokenDatabase** não existir, ou não puder ser acessado ou ter uma estrutura inválida é lançado um erro, representado por esse objeto. O objeto tem duas propriedades:
  - **name**: O nome do erro representado pelo objeto.
  - **message**: Uma mensagem que deve ser emitida ao usuário quando o erro for lançado.

Os arquivos de banco de dados, como dito antes, estão na pasta database/, para melhorar a organização do projeto. A localização do arquivo desejado pode ser modificado na respectiva constante.

4 Funções

O sistema conta com várias funções com diferentes propósitos, mas com um mesmo fim: agregar à reparação do banco de dados. Todas as funções foram definidas como **arrow functions** constantes.

Algumas funções poderiam ter sido feitas em um única linha, mas foi prezado pela legibilidade do código e pela robustez(como o tratamento de erros da função **importDatabase()**).

4.1 Funções do Banco de Dados NoSQL

Essas funções realizam o contato direto com os arquivos que representam o banco de dados, ou seja, os arquivos JSON. A constante **fs** é utilizada aqui para invocar os métodos próprios para manipular arquivos de maneira síncrona.

A função **importDatabase()**, como mostrada na listagem 1, retorna uma lista de objetos obtidos do caminho do banco de dados, especificado pelo argumento, por meio da função readFileSync(). Um **bloco try-catch** irá lançar um erro **FileError** caso haja algum problema com o caminho especificado ou com o arquivo em si, permitindo ao usuário localizar e consertar o erro.

```
1 const importDatabase = (databaseFile) => {
2     try {
3         return JSON.parse(fs.readFileSync(databaseFile, 'utf-8'));
4     }
5     catch (fileError) {
6         throw FileError;
7     }
8 }
```

Listing 1: Função importDatabase()

Já a função **exportDatabase()**, como mostrada na listagem 2, realiza a operação contrária: exporta a lista de objetos para um banco de dados JSON, utilizando a função writeFileSync(). A função stringify() irá converter a lista de objetos em uma string JSON facilmente legível, para ser exportada.

```
1 const exportDatabase = (databaseFile, databaseRaw) => {
2     fs.writeFileSync(databaseFile, JSON.stringify(databaseRaw, null, 2));
3 }
```

Listing 2: Função exportDatabase()

4.2 Funções de Reparo

Essas funções tem como foco principal reparar o banco de dados, baseando-se nas especificações definidas. São funções interoperáveis, não dependendo umas das outras para funcionar. As funções recebem a lista de objetos a ser manipulada e a retorna, ao final, já consertada. A estrutura **for...in** foi utilizada para iterar na lista de objetos.

A função **fixNames()**, como mostrada na listagem 3, substitui caracteres inválidos pelos respectivos caracteres válidos. Os pares que definem a substituição estão na constante namesCorrection. A função itera nos pares da constante e substitui todas as ocorrências dos caracteres inválidos na propriedade "name"do objeto, utilizando a função replaceAll() e passando os dois elementos do par.

```
1 const fixNames = (database) => {
2     for (let i in database) {
3         namesCorrection.forEach(correct => database[i]["name"] =
              database[i]["name"].replaceAll(correct[0], correct[1]));
4     }
5     return database;
6 }
```

Listing 3: Função fixNames()

A função **fixPrices()**, mostrada na listagem 4, repara o tipo dos valores da propriedade "price", de string para número. Para isso é verificado, na linha 3, se o tipo do valor da propriedade "price", do objeto da iteração, não é um valor numérico. Se não for então é realizado um *cast* do valor, utilizando a função Number(), e o atribuindo de volta ao objeto.

```
1 const fixPrices = (database) => {
2     for (let i in database) {
3         if (typeof database[i]["price"] != Number) {
4             database[i]["price"] = Number(database[i]["price"]);
5         }
6     }
7     return database;
8 }
```

Listing 4: Função fixPrices()

Já a função **fixQuantities()**, como mostra a listagem 5, adiciona uma propriedade "quantity"ao objeto, caso ele não a tenha, a atribuindo valor zero. Para realizar essa verificação é utilizado a função hasOwnProperty("quantity"), na linha 3, pra verificar se o objeto em questão tem a propriedade "quantity". Caso

não a tenha é atribuído a propriedade ao objeto, e a ela, valor zero.

```
const fixQuantities = (database) => {
     for (let i in database) {
          if (!database[i].hasOwnProperty("quantity")) {
               database[i]["quantity"] = 0;
          }
     }
     return database;
}
```

Listing 5: Função fixQuantities()

4.3 Funções de Validação

Essas funções tem como objetivo verificar se o banco foi devidamente recuperado. Sua principal ação é mostrar ao usuário, no console, informações do banco de dados, como o nome dos produtos após a reparação e o valor total do estoque.

Essas funções tem algumas características diferenciais que valem a pena serem mencionadas:

- **Sequências de Escape ANSI**: Utilizadas para estilizar os títulos de cada etapa da validação, para que fique fácil sua identificação. A cor verde utilizada é definida pela sequência 0;32m.
- **Parâmetro *categories***: Lista de categorias dos produtos, sem repetições, e previamente obtida do arquivo JSON. É falado mais sobre esse argumento na seção 5.
- **Intenso uso do Paradigma Funcional**: O uso do paradigma funcional foi incrementado nessas funções, justamente para maximizar a legibilidade e a densidade de código. As funções forEach(), filter(), map(), reduce() e sort() dão ao programa muito mais robustez!

A função **listProducts()**, como é mostrado na listagem 6, tem como objetivo listar o nome dos produtos, agrupados por categoria e ordenados por ID. Para isso a função itera sobre a lista de categorias, adicionando a uma lista de produtos ordenados apenas os produtos que correspondem à categoria da iteração em questão. Como a estrutura itera sobre todas as categorias ao final teremos uma lista de produtos com todos os produtos agrupados por categoria.

A cada iteração é filtrado apenas os objetos de uma categoria, criando um grupo, utilizando o método filter(). Os elementos desse grupo são ordenados por ID, em ordem crescente, utilizando o método sort(), e, ao final, mapeados à lista de produtos como uma lista dos nomes dos produtos, por meio de map().

```
const listProducts = (database, categories) => {
     let products = [];
     for(let cat of categories){
          products = products.concat(database
               .filter(item => item["category"] == cat)
               .sort((x, y) => x["id"] - y["id"])
               .map(item => item["name"]));
     }
     console.log("\033[0;32m%s\033[0m",
     "List of categorized and sorted products:");
     products.forEach(item => console.log(item));
}
```

Listing 6: Função listProducts()

Já a função **stockValue()**, como mostra a listagem 7, mostra ao usuário o valor total do estoque de cada categoria. Essa função itera sobre cada elemento da lista de categorias, imprimindo uma string personalizada com o valor total, referente a cada categoria. O valor total é calculado pela função categoryTotal(), que recebe uma categoria e retorna o valor total do estoque da categoria. Esse valor total é formatado para suas casas decimais com a função toFixed(2).

Essa função é chamada em cada iteração, assim operando em todas as categorias no processo. Ela primeiro filtra os elementos do banco de dados pela categoria passada como argumento. Após isso é calculado o valor total de cada elemento, ou seja, o preço multiplicado pela quantidade. Então é realizado um processo de redução, onde o valor total de cada produto é acumulado com os outros. Ao final o valor total referente à categoria é retornado.

```
const stockValue = (database, categories) => {
     const categoryTotal = (category) => database
          .filter(item => item["category"] == category)
          .map(item => item["price"] * item["quantity"])
          .reduce((total, item) => total + item);

     console.log("\033[0;32m%s\033[0m",
     "\nTotal inventory value by category:");

     categories.forEach(cat =>
     console.log(`Category ${cat}: R$${categoryTotal(cat).toFixed(2)}`));
}
```

Listing 7: Função stockValue()

5 O código principal

É só na última seção do código que são invocadas as funções. Primeiramente é utilizada a função importDatabase(), passando a constante que representa o caminho do arquivo de entrada como argumento. A função retorna uma lista de objetos, que é utilizada como argumento nos métodos fixNames(), fixPrices() e fixQuantities(), que retornam a mesma lista de objetos, porém com a respectiva propriedade consertada. Após isso é utilizada a função exportDatabase() para exportar a lista para um arquivo, passando a constante com o caminho do arquivo de saída e a lista de objetos.

É utilizada uma função autoexecutável para atribuir à constante productCategories uma lista das categorias dos produtos, em ordem alfabética. Essa função é mostrada na listagem 8. É utilizado um objeto da classe Set para guardar as categorias sem que hajam repetições. Para isso obtemos as categorias de cada produto e iteramos sobre elas, adicionando uma a uma no conjunto. Ao final obtemos um array a partir do conjunto, em ordem alfabética. Esse array então é atribuído à productCategories, que será passado como argumento para as funções de validação, junto ao banco de dados corrigido.

```
1 const productCategories = (() => {
2     let categories = new Set();
3     databaseRaw.map(item => item["category"])
4     .forEach(cat => categories.add(cat));
5     return Array.from(categories).sort();
6 })();
```

Listing 8: Lista productCategories

É possível perceber que essa função só executará uma vez, e de forma automática, e atribuirá à productCategories seu retorno.

Posicionamento da constante productCategories

Essa função não pode ser executada antes de importar o banco de dados, logo não é possível posicionar a constante na seção das outras constantes. Portanto, teve de ser colocada na última seção. Poderíamos defini-la como “var", o que iria realizar o *hoisting*, podendo ser acessada livremente nas funções acima, mas isso impactaria uma futura modularidade do código. Dito isso a melhor abordagem era definir como constante na última seção e passá-la como parâmetro para as funções.

Toda a função principal foi envelopada com um bloco **try-catch** para tratar erros no arquivo de entrada, por mais que apenas a função importDatabase() lance um objeto FileError. Isso foi feito para que o código permaneça mais organizado. Caso haja um erro no arquivo de entrada este é tratado no bloco **catch** da seção principal, mostrando ao usuário o nome do erro, em negrito de cor branca, e a mensagem em si, em vermelho, como mostra a listagem 9.

```
1 catch (fileError) {
2       console.log("\033[1;37m%s\033[0m: \033[0;31m%s\033[0m",
3       fileError.name, fileError.message);
4 }
```

Listing 9: Tratamento do erro FileError

Importante lembrar que, para que não haja um erro de arquivo, o programa deve ser executado a partir de sua própria pasta. Ou seja, o diretório de trabalho, ao executar o programa, deve ser o diretório brokendatabase/, ou outro em que o programa esteja localizado.

6 Conclusões

Eu me diverti bastante fazendo esse projeto. Digo que aprendi muita coisa, e que foi um desafio muito interessante, principalmente a função listProducts(). Essa função foi um grande desafio, pois queria fazer o código da maneira mais responsiva o possível, de forma que as categorias não estivesse pré-selecionadas e, ao mesmo tempo, queria uma solução com um bom desempenho.

Utilizar o paradigma funcional em um projeto como esse realmente me serviu de grande aprendizado, e espero ter mais oportunidades como essa! Dito isso, gostaria de agradecer a você que acompanhou o meu projeto e aos prezados da Rocky, que me deram a excelente oportunidade de concorrer a uma vaga de estágio em sua empresa. Foi uma jornada cheia de aprendizados, e espero que continue por muito tempo!

Até a próxima!

Pedro M. Botelho
Engenharia de Computação