Capítulo 3

Desenvolvimento Full-Stack com Javascript: uma Visão Geral e Prática

Bruna C. R. Cunha
Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP)
São Carlos – SP – Brasil
brunaru@icmc.usp.br

Abstract

This chapter presents basic principles for developing a service-based full-stack Web application using REST. Considering the current Web development scenario, the specific objectives are to use the JavaScript language for both front-end and back-end development, and to apply frameworks and libraries for developing modern Web applications, such as Express, a framework for web server constructions, and Sequelize, a library used for Object-Relational Mapping. We discuss basic principles, including technologies and the main architectural patterns currently used, that support the use of technologies for the Web. At the end, we present a Web page that uses JS to consume the developed services.

Resumo

Este capítulo apresenta aspectos básicos do desenvolvimento de uma aplicação Web full-stack baseada em serviços utilizando REST. Considerando o atual cenário do desenvolvimento Web, os objetivos específicos são utilizar a linguagem JavaScript em ambas as camadas de desenvolvimento, front-end e back-end, e aplicar frameworks e bibliotecas para o desenvolvimento de aplicações Web modernas, como o Express, framework para construções de servidores web, e o Sequelize, biblioteca utilizada para o Mapeamento Objeto-Relacional. São discutidos princípios básicos, incluindo as tecnologias e os principais padrões arquitetônicos utilizados atualmente, de forma a fundamentar o uso de tecnologias para a Web. Ao final, é apresentada uma página Web que utiliza JS para consumir os serviços desenvolvidos.

3.1. Introdução

Uma aplicação Web é um *software* distribuído e acessível por meio da Internet por clientes, mais especificamente navegadores Web (*i.e., Web browser*, em inglês). Esses sistemas são organizados de forma que o usuário interage com um programa em execução no computador local (*i.e.*, um navegador) que, por sua vez, realiza requisições pela Internet (Sommerville, 2019). As requisições são atendidas por computadores remotos, ou seja, servidores, que hospedam o sistema e seus demais recursos. Um servidor Web fornece serviços, como acesso a páginas da Web, recursos multimídia, dados e aplicações, de acordo com as funções determinadas pela arquitetura adotada. Em outros termos, uma aplicação web trata-se de um *software*, hospedado em um servidor, que apresenta funcionalidades diversas.

Nesse domínio, é importante clarificar também a notação de servidor Web. Um servidor Web é referente a um *hardware* ou um *software* (ou o conjunto de ambos) que atende requisições de clientes (Mozilla Foundation, 2023). Enquanto o *software* é composto de diferentes componentes que permitem que clientes acessem diferentes recursos, o *hardware* trata-se de um computador que armazena o *software* e, possivelmente, outros elementos relacionados, como arquivos e bancos de dados. Esse último trata-se de um *software* dedicado a armazenar, manter e manipular coleções organizadas de informações estruturadas.

Bancos de dados podem ser relacionais, os quais organizam dados em tabelas relacionadas entre si por meio de chaves e utilizam a *Structured Query Language* (SQL) como linguagem padrão. Exemplos são MySQL, Oracle Database, PostgreSQL. Dado que o formato relacional foi dominante por muitos anos, bancos não relacionais são chamados NoSQL. O MongoDB e o Neo4j são duas alternativas não relacionais populares atualmente, sendo um baseado em documentos e o outro orientado a grafos, respectivamente.

Considerando esses elementos, uma aplicação Web pode apresentar um ou mais servidores, responsáveis por entregar uma interface, executável em navegadores Web, que se comunicam com servidor(es) lógico(s), responsáveis, minimamente, pelo processamento de requisições e acesso a banco(s) de dados.

Aplicações Web são apoiadas pelo modelo cliente-servidor, considerando que o cliente, um navegador, exibe e executa parte do código recebido do servidor, enquanto o servidor Web responde requisições e as processa. Considerando a organização de aplicações Web, é importante contextualizar os termos programação *front-end* e *back-end*. *Front-end* é o termo utilizado para definir a aplicação executada pelo cliente (navegador) e trata-se do código responsável, principalmente, pela interface com o usuário e pela lógica de processamento e comunicação com o servidor. Já a expressão *back-end* se refere ao(s) componente(s) executados em servidores (físicos), como regras de negócio e acesso ao(s) banco(s) de dados (Marinho e da Cruz, 2020). Por fim, o termo *full-stack* é utilizado para denominar a programação de uma aplicação Web por completo.

O desenvolvimento *front-end* é apoiado pelo uso de linguagens interpretadas pelo navegador e que permitem a exibição de conteúdo interativo. Atualmente, são três linguagens que lideram esse cenário: HTML, Cascading Style Sheets (CSS) e JavaScript

(JS). Enquanto o HTML é utilizado para estruturar o conteúdo e a semântica de uma página Web, o CSS é o recurso utilizado para formatar e estruturar as páginas e o JavaScript é a principal linguagem de programação utilizada para atribuir interatividade e permitir a comunicação com servidores Web. Essas tecnologias possuem organizações responsáveis por manter sua evolução e seus padrões de implementação, o HTML e o CSS são mantidas pelo World Wide Web Consortium (W3C), enquanto o JS segue a especificação de linguagem mantida pela ECMA Script.

Na outra ponta, o desenvolvimento *back-end* utiliza uma ampla variedade de linguagens. Para que uma determinada linguagem possa ser utilizada, é preciso, apenas, que ela seja executável pelo *hardware* do servidor e que, de alguma forma, disponibilize recursos de comunicação HTTP que permitam a implementação de aplicações Web. Entre as linguagens utilizadas pela programação *back-end* temos JavaScript, Java, Python, C#, PHP, entre outras.

Tanto o desenvolvimento *front-end* como o *back-end* são apoiados por bibliotecas e *frameworks* que auxiliam o trabalho de desenvolvedores. Bibliotecas são coleções de classes e métodos empacotados para reusabilidade. Ou seja, uma biblioteca implementa um conjunto conexo e específico de funcionalidades visando resolver um problema particular. Por outro lado, um *framework* pode ser definido como uma coleção de bibliotecas que incorporam uma abordagem específica de desenvolvimento que estruturam e agilizam o processo de criação de um *software*.

3.2. Tecnologias

Esta seção apresenta as tecnologias envolvidas no desenvolvimento de aplicações Web. Os conceitos aqui discutidos fundamentam explicações sobre o funcionamento de recursos aplicados na programação JS que serão trabalhados em seções futuras.

3.2.1. O Protocolo HTTP

A Internet apresenta uma arquitetura de protocolos organizada em cinco camadas: aplicação, transporte, rede, enlace e física. O *Hypertext Transfer Protocol* (HTTP), assim como sua variante segura, o *Hypertext Transfer Protocol Secure* (HTTPS), está localizado na camada de aplicação, sendo responsabilidade dos navegadores e servidores Web implementarem o seu padrão. Por estar na camada de aplicação, o HTTP dialoga com protocolos da camada de transporte, especialmente o *Transmission Control Protocol* (TCP).

O HTTP é um protocolo de comunicação baseado em mensagens de requisição e resposta, partindo de um cliente para um servidor. Uma característica importante dessa comunicação é que ela é sem estado (*i.e.*, *stateless*), ou seja, após receber uma requisição de um cliente, o servidor processa e responde, sem guardar informações de estado sobre a requisição realizada. Essa característica torna o modelo da Web bastante escalável. O entendimento do HTTP é essencial para o desenvolvimento consciente de aplicações e arquiteturas baseadas em seu funcionamento. Isso ficará mais evidente no decorrer deste capítulo.

Em termos de estrutura, uma mensagem de requisição é composta por um método HTTP (*e.g.*, verbos GET, POST, PUT, DELETE), pelo endereço do recurso (*i.e.*, seu *Uniform Resource Identifier*, ou URI) e pela versão do protocolo utilizada. A

mensagem também pode, opcionalmente, conter um cabeçalho, com informações adicionais sobre a requisição, como *tokens* de autenticação, tipo de dados, tipo de *enconding*, *cookies* etc. Dependendo do método, a mensagem pode apresentar um corpo, que contém os dados da requisição. O corpo pode conter dados em XML e JSON ou até mesmo uma mídia, a depender do verbo utilizado. A Figura 3.1 ilustra a estrutura de uma mensagem de requisição (*request*).

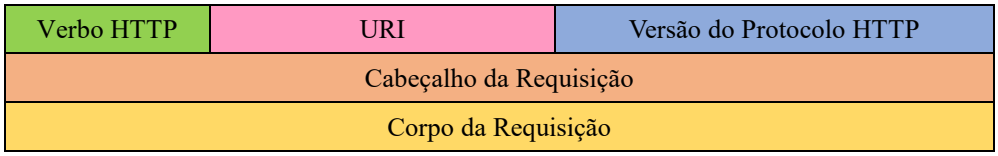

Figura 3.1. Estrutura de uma mensagem de requisição HTTP

Uma requisição do tipo POST, por exemplo, é utilizada para enviar dados a um servidor e seu corpo irá conter informações, como dados de um formulário para cadastro. Já uma requisição do tipo GET não apresenta corpo e é utilizada, geralmente, para recuperar um recurso. A extensibilidade do protocolo permite que ele seja utilizado para a recuperação de diferentes tipos de recurso. A própria ação de acessar um website gera uma requisição do tipo GET para recuperar a página correspondente. Além da página, outras requisições são disparadas para recuperar os demais recursos associados, como arquivos CSS e JavaScript, imagens e demais arquivos multimídia.

Mensagens HTTP utilizadas na resposta seguem uma estrutura similar: uma primeira linha, contendo a versão do protocolo e um código de resposta, acompanhada por cabeçalho e corpo opcionais. A Figura 3.2 ilustra a estrutura de uma mensagem de resposta (*response*). O código da resposta (*status*) é um valor numérico, definido no servidor, e, quando corretamente implementado, segue uma lógica de acordo com intervalos definidos. Por exemplo, mensagens de *status* na casa dos 200 representam o sucesso da execução da requisição, valores na casa dos 400 são requisições malformadas (entre as quais consta o famoso erro 404, de recurso não encontrado) e entre os 500 estão os erros de servidor. O cabeçalho pode conter informações adicionais sobre a resposta, como o tipo de conteúdo, controle de cache e *enconding*. O corpo pode conter uma página, texto ou qualquer outro dado ou arquivo requisitado.

Figura 3.2. Estrutura de uma mensagem de resposta HTTP

3.2.2. A linguagem HTML

Proposta em 1990 por Berners-Lee, a linguagem HTML trata-se de uma “linguagem de marcação de hipertexto”, ou seja, como o nome indica, é uma linguagem que permite a definição de textos enriquecidos pela inserção de conexões para outros documentos e recursos. O HTML passou por diversas versões e atualmente está em sua versão 5, publicada como padrão em 2016.

Utiliza-se a "marcação" do HTML para definir os elementos e descrever a estrutura de uma página Web. A marcação é realizada por marcadores chamados "*tags*". Esses elementos indicam ao navegador como exibir o conteúdo. O HTML apresenta um conjunto extenso de marcadores, os quais permitem definir diversos tipos de conteúdo, como parágrafos (*p*), cabeçalhos (*h1*, *h2*, *h3*, etc), seções (*div*, *section*, *nav*, *footer*, etc), links (*a*), listas (*ul*, *ol*), tabelas (*table*), imagens (*img*), vídeos (*video*) entre outros. As *tags* também podem ser acompanhadas por atributos, que variam com o tipo da *tag*, que são utilizados para definir comportamentos do elemento. A marcação ocorre com o nome da *tag* entre os símbolos "<" e ">". Na maioria dos casos, a marcação necessita de ser repetida para indicar o fim daquele conteúdo, isso é realizado repetindo a marcação ao final do conteúdo, porém com um símbolo de "/" antes do nome da *tag*. Alguns exemplos de uso:

- <html> </html>: elemento raiz que indica o início e o fim de uma página HTML;
- : define seções (divisões) na página;
- <form> </form>: define um formulário com auxílio de outros elementos, como marcadores do tipo "input" para indicar diferentes tipos de entrada do usuário (*e.g.*, <input type="text">);
- <img src="">: permite a inclusão de imagens, indicadas no atributo "scr".

Basicamente, toda página HTML é um documento iniciado pelo seu elemento raiz (*i.e.*, <html>) que contém as demais marcações. Esse documento pode ser estruturado em uma árvore hierárquica e acessado programaticamente por meio da interface *Document Object Model* (DOM). A manipulação da árvore DOM pela linguagem JavaScript permite a modificação dinâmica de páginas HTML, o que será demostrado na Seção 3.2.4 e nos exemplos práticos.

Este capítulo pressupõe que o leitor possua conhecimentos básicos em HTML. Caso seja necessário, o tutorial sobre HTML[1] da MDN Web Docs, um projeto colaborativo da Mozilla (2023), é indicado para referência e estudo.

3.2.3. Folhas de Estilo em Cascata

Cascading Style Sheets (CSS), ou Folhas de Estilo em Cascata em português, é uma linguagem de estilo para definição da apresentação de elementos HTML. O nome "cascata" refere-se às regras de aplicação dos estilos, que consideram a ordem e a especificidade de seus seletores. A primeira versão do CSS foi proposta em 1996, enquanto sua atual versão, o CSS3, teve o desenvolvimento iniciado em 1999 e têm recebido atualizações e melhorias constantes ao longo dos anos.

O CSS permite estilizar textos, imagens, seções, tabelas, listas, entre outros componentes, considerando aspectos de sua aparência, como tamanho, cor, fonte, fundo, margens, borda e posicionamento na tela. Além disso, a linguagem permite descrever visualizações distintas de acordo com o meio de apresentação (*e.g.*, tela ou impressão) ou a resolução dos dispositivos, flexibilizando a criação de designs adaptativos e responsivos.

[1] developer.mozilla.org/en-US/docs/Web/HTML

Até sua versão 4, o HTML permitia formatar seus elementos por meio de atributos, o que deixava o código poluído e de difícil manutenção, pois alterações visuais ficavam incorporadas às marcações. O HTML5 eliminou esse comportamento, de forma que folhas de estilos se tornaram obrigatórias para a configuração de sua apresentação.

A inclusão do CSS pode ser feita pela inclusão de um arquivo externo ao HTML (forma mais indicada, exemplo: <link rel="stylesheet" href="./style.css">) ou pela adição de definições diretamente no código, por meio do marcador "<style>" ou do atributo "*style*".

A nomenclatura utilizada na construção do CSS segue o padrão *dashed-case*, ou seja, separado por hifens. Declarações são feitas em blocos para um ou mais seletores. Cada bloco declarativo é identificado por um seletor e delimitado por chaves, as quais abrigam propriedades com valores que definem seu estilo, conforme ilustra a Figura 3.3.

Figura 3.3. Exemplo de um bloco declarativo em CSS

O seletor indica qual será o alvo de uma declaração. No exemplo da Figura 3 o seletor é a *tag* HTML "*p*", o que indica que a formatação declarada será aplicada a todos os parágrafos do documento. Seletores podem ser *tags* HTML (*e.g.*, *p, h1, h2, a, body, div*), identificadores (*#id-do-elemento*) e classes (*.nome-da-classe*) criadas para fins de aplicação de estilo. O seletor também pode ser universal (*), ou seja, aplicado em todos os elementos. Pode-se associar um único bloco a vários seletores ou realizar combinações entre eles.

A chave define a propriedade CSS que está sendo considerada, enquanto o seu valor é o comportamento aplicado. Cada propriedade aceita valores específicos. Por exemplo, a chave "color" aceita valores relativos à cor, codificados em inglês, hexadecimal ou modelo RGB.

O capítulo espera que o leitor possua conhecimentos básicos em CSS. Caso seja necessário, o tutorial sobre CSS[2] da MDN Web Docs é indicado para referência e estudo.

[2] developer.mozilla.org/en-US/docs/Web/CSS

3.2.4. A linguagem JavaScript

O terceiro pilar do desenvolvimento *front-end*, ou seja, da parte referente ao código executado pelo navegador, é o JavaScript (JS). Diferentemente de HTML e CSS, que são, respectivamente, linguagens de marcação e de estilo, o JavaScript é uma linguagem de programação. É o JS que permite programar o comportamento da interface, habilitando a criação de páginas Web interativas e dinâmicas, ou seja, o conteúdo apresentado na página irá mudar de acordo com a interação do usuário e dos dados armazenados. JavaScript é uma linguagem de *script*, ou seja, seu código roda em um ambiente de execução, sendo interpretado, e não é submetido a um processo de compilação. Os navegadores atuais implementam mecanismos para interpretar *scripts* em JS e, mais recentemente, a plataforma Node.js tornou possível que a linguagem seja utilizada em outros cenários, particularmente na programação de servidores Web.

Criado em 1995, o JavaScript foi bem aceito devido a sua simplicidade e por ser atualizado com frequência. Desde 1997 a linguagem é padronizada pela ECMAScript, a qual garante a interoperabilidade entre as páginas Web e navegadores. Porém, pode-se dizer que o ponto de virada do JS foi em 2005, quando o mecanismo *Asynchronous JavaScript and XML* (AJAX) foi proposto (Garrett, 2005). O objetivo do AJAX é permitir o tratamento de requisições assíncronas por parte do cliente, ou seja, ele possibilita requisitar um conteúdo e tratar seu resultado sem interrupções ou atualizações completas da página. Essa abordagem proporciona um diálogo interativo com o servidor, sem a necessidade de recarregar a página no navegador, o que amplia a experiência do usuário na Web, deixando-a similar a experiência "*desktop*" de um *software* nativo.

A sintaxe da linguagem JavaScript não difere muito de suas predecessoras, como C ou Java, de forma que suas estruturas de blocos, condicionais, de repetição e declaração de funções se assemelham. O Quadro 3.1 apresenta um exemplo de sua sintaxe.

Quadro 3.1. Exemplo de código JS com estruturas de condição (*if*) e repetição (*for*)

```
const valor = 'O valor é igual a '
for (let i = 1; i < 6; i++) {
   console.log(valor + i)
   if (i == 5) {
      console.log('fim...')
   }
}
```

A tipagem dinâmica, também referenciada como tipagem "fraca", é um dos diferenciais da linguagem JS, a qual infere o tipo atribuído em tempo de execução. Em JS, a declaração de variáveis ocorre por meio das palavras-chave "*var*" e "*let*", em escopo global e local respectivamente, e "*const*", para a declaração de constantes. O Quadro 3.2 exemplifica a declaração de uma variável local que, inicialmente, recebe um valor numérico, imprime esse valor no *log* e, em seguida, seu valor é alterada para um texto, que também é impresso.

Quadro 3.2. Demonstração da tipagem dinâmica em JS

```
let a = 3
console.log(a)
a = 'Um texto!'
console.log(a)
```

A tipagem dinâmica exige uma maior cautela quando operações entre variáveis, principalmente advindas de formulários ou serviços, são realizadas. Considerando, por exemplo, que o operador de adição é aplicado tanto para concatenação de textos (*i.e., strings*) como para a soma de números, a soma de um número representado em um formato textual não irá gerar o resultado esperado. O Quadro 3.3 ilustra essa situação. Uma variável com o valor textual "5" é declarada e, ao aplicar o operador de adição com o valor número 5, o resultado é uma concatenação de texto. Para obter um resultado numérico é necessário converter o valor do texto em um inteiro, por meio do método "*parseInt()*".

Quadro 3.3. Concatenação ou soma de valores de acordo com sua tipagem

```
let num = '5'
console.log(num + 5) // o resultado é 55
console.log(parseInt(num) + 5) // o resultado é 10
```

Assim como em outras linguagens, em JS é possível organizar blocos de código em unidades invocáveis, ou seja, encapsular partes do algoritmo em funções. Funções em JS são definidas pela palavra-chave "*function*", são delimitadas por chaves e podem (ou não) possui parâmetros e retornar um valor (o retorno não é obrigatório). As funções também podem ser anônimas, o que implica que seu nome não será declarado e a mesma será atribuída a uma variável. A definição de funções é uma das características que demonstra a flexibilidade da linguagem. Há ainda o formato de expressão "*arrow*", em que a definição de uma função é reduzida, não havendo a necessidade da palavra-chave "*function*". O Quadro 3.4 exemplifica esses três tipos de função. Priorizando a didática, este documento irá utilizar apenas o formato clássico.

Quadro 3.4. Três formas diferentes de definir funções em JS

```
/* Formato tradicional. */
function quadrado_a(num) {
   return num * num
}

/* Função anônima. */
const quadrado_b = function (num) {
   return num * num
}

/* Função arrow. */
const quadrado_c =  num => num * num

console.log(quadrado_a(2))
```

```
console.log(quadrado_b(3))
console.log(quadrado_c(4))
```

Outro conceito importante do JS são as funções do tipo "*callback*". De forma simplificada, uma função "*callback*" é uma função que é passada como argumento para outra função, sendo invocada dentro de outro contexto. Ou seja, funções em JS funções aceitam funções como argumentos, além de valores e referências. Essas funções podem ser invocadas dentro do bloco de execução daquela que a recebeu. O Quadro 3.5 apresenta um exemplo simples em que uma função chamada "soma" recebe dois valores numéricos e uma função como argumentos, realizando a somatória dos valores e invocando a função recebida com o resultado da operação. Já o Quadro 3.6 ilustra a situação da função "*forEach*", pertencente ao objeto do tipo "*array*" (lista), a qual recebe uma função como argumento e a invoca para cada item presente na lista.

Quadro 3.5. Definição de uma função que aceita uma *callback* como argumento

```
/* Função que escreve um texto na tela. */
function minhaCallback(texto) {
   console.log('Valor: ' + texto)
}

/* Função soma */
function soma(x, y, func) {
   const res = x + y
   func(res)
}

soma(3, 4, minhaCallback)
```

Quadro 3.6. Exemplo do *forEach*, o qual recebe uma função *callback* como argumento

```
/* Array (lista) de números. */
const numeros = [23, 44, 1, 8, 71, 92, 33, 47, 55, 60]

/* Função que escreve um texto na tela. */
function minhaCallback(texto) {
   console.log('Valor: ' + texto)
}
/* Percorre a lista chamando a callback. */
numeros.forEach(minhaCallback)

/* A função pode ser escrita diretamente como argumento. */
numeros.forEach(function(num) {
   console.log('Valor: ' + num)
})
```

Ainda no contexto de funções, é importante considerar que essas podem ser síncronas ou assíncronas. No contexto deste tutorial, algumas operações assíncronas são realizadas, como consultas a bancos de dados e a serviços. É imprescindível que tais tarefas sejam assíncronas pois envolvem requisições fora do escopo do programa que

podem ter um retorno demorado. Um dos recursos utilizados para tratar operações assíncronas em JS são as promessas (*Promisses*). Uma *Promisse* é um objeto que representa uma conclusão ou uma falha de uma tarefa assíncrona. A conclusão é indicada pela função "*then*", a qual recebe uma função *callback* como argumento, enquanto o erro é tratado pelo "*catch*", que também recebe uma *callback*. O Quadro 3.7 demonstra uma chamada para uma função assíncrona "*findAll*()", da classe "Client", que recebe uma função para tratar os dados retornados e, ao final, trata um possível erro, caso sua execução não seja bem sucedida.

Quadro 3.7. Chamada de uma função assíncrona (findAll) tratada por meio de Promisse

```
import Client from '../models/client.model.js'
function findAll(request, response) {
   Client.findAll().then(function(results) {
      /* aqui é realizado o tratamento dos resultados da promisse */
   }).catch(function(err) {
      /* aqui é realizado o tratamento de erros */
   })}
```

Além de suportar os estilos de programação funcional, imperativo e orientada a objetos, o JS permite manipular o DOM de uma página HTML. O DOM organiza os elementos da página HTML em uma árvore. Esses elementos podem ser recuperados pela manipulação da árvore ou, mais comumente, pelas suas características, como identificador (*getElementById*), classe (*getElementsByClass*), nome (*getElementsByName*) ou *tag* (*getElementsByTagName*). Além disso, eventos que mapeiam funções JS podem ser associados aos elementos (*e.g.*, *onClick*, *onMouseOver*, *onLoad*). O Quadro 1.8 exemplifica um código que modifica um texto.

Quadro 3.8. Código HTML e JS: um botão que permite alterar o texto do cabeçalho

```
<!DOCTYPE html>
<html>
<body>
   <h1 id="titulo">Bem vindo!</h1>
   <button onclick="mudar()">Clique</button>
</body>
<script>
   const textoNovo = "Pague seus boletos!"
   function mudar() {
      let t = document.getElementById("titulo")
      t.innerText = textoNovo
   }
</script>
</html>
```

Nesse exemplo, a função "*mudar()*", invocada ao pressionar o botão, recupera o elemento *h1* por meio de seu identificador (*i.e., id*) e altera o seu texto. Esse exemplo poderia conter uma consulta assíncrona a um serviço, o qual enviaria dados textuais (ou até mídias) que poderiam ser utilizados para alterar ou criar elementos na página. Ou

seja, ao identificar e tratar um evento, que pode ser uma ação do usuário ou da própria página, é possível processar esse evento de forma dinâmica e interativa, sem a necessidade de carregamentos de página.

Este capítulo não propõe ser um guia extensivo da linguagem JavaScript, mas utilizará a mesma para construção de um serviço Web com acesso ao banco e uma página que permite listar e cadastrar dados referentes a esse serviço. Para mais detalhes sobre a linguagem, recomenda-se o tutorial da MDN Web Docs[3] (2023).

3.3. Arquiteturas Web

A forma como uma aplicação Web é organizada afeta diretamente a escolha das tecnologias utilizadas. Em outras palavras, as soluções arquiteturais, propostas para contemplar demandas e melhorar a qualidade de um produto de *software*, influenciam diretamente na escolha das tecnologias e na proposição de bibliotecas e *frameworks* associados, os quais estruturam a aplicação e agilizam sua implementação.

Entre as arquiteturas aplicadas no desenvolvimento de sistemas Web, a Arquitetura Cliente-Servidor de Duas Camadas (*two-tier*, em inglês) trata-se de uma abordagem simples e clássica, de forma que o sistema é centralizado por motivos de segurança ou simplicidade (Sommerville, 2019). São possíveis diferentes configurações quanto à divisão de processamento (*i.e.*, regras de negócio) entre o cliente e o servidor, porém, o cliente é sempre responsável pela apresentação enquanto o servidor lógico é responsável pela manipulação de dados e acesso ao banco.

Além de apresentar uma baixa manutenibilidade (*i.e.*, qualidade relacionada à facilidade de manutenção), a organização em duas camadas é pouco escalável, dado que a existência de um único componente lógico impede que o trabalho seja distribuído em servidores físicos. Em casos de sistemas de complexidade superior ou que atendam muitas requisições, uma arquitetura multicamadas (*n-tier*) é mais adequada, de forma que a aplicação é dividida em camadas lógicas de acordo com suas funcionalidades.

Em termos de divisão do processamento, a aplicação pode priorizar uma abordagem baseada em um maior processamento pelo cliente ou pelo servidor. No modelo de cliente magro (*thin client*) a maior parte do processamento é gerenciado pelo servidor. Já na abordagem de cliente gordo (*thick client*), a situação é inversa, de forma que a maior parte ou todo o processamento é realizado no cliente com a intenção de reduzir a carga do servidor. Considerando as tecnologias atuais, JavaScript é uma linguagem que permite que o processamento de dados e ações seja realizado no navegador, tornando possível que o modelo cliente gordo seja adotado sem a necessidade de instalação de um *software* adicional.

Por muitos anos a programação de sistemas Web foi estruturada por uma arquitetura *Model–View–Controller* (MVC) com alto acoplamento entre cliente e servidor. As tecnologias utilizadas no *front* e *back-end* eram dependentes e as "páginas dinâmicas" eram geradas no servidor, não no navegador. Um bom exemplo disso é a tecnologia *JavaServer Pages* (JSP), utilizada com a linguagem de programação Java.

[3]developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/JavaScript_basics

Por mais que o código fosse inserido nas páginas HTML, o processamento em si era realizado no servidor, o qual era responsável por gerar a visão (*i.e.*, a *View*).

No entanto, com o crescimento da Web e o aumento do número de usuários, novos requisitos surgiram devido à necessidade de maior escalabilidade, componentização e comunicação entre sistemas de diferentes organizações. A Arquitetura Orientadas a Serviços (*Service-Oriented Architecture*, SOA) foi proposta como alternativa para esses problemas. A proposta da SOA é o desenvolvimento de sistemas distribuídos em que os componentes são serviços independentes. As principais características dessa arquitetura são (Sommerville, 2019):

- **Acoplamento fraco**: uma mudança efetuada em um componente deve afetar o mínimo possível os outros componentes;
- **Interoperabilidade**: toda troca de mensagem é independente de plataforma;
- **Reuso**: serviços podem ser utilizados por diferentes aplicações clientes, de forma independente;
- **Capacidade de descoberta**: o ambiente de execução e o contrato de um serviço devem ser identificados e vinculados durante a execução.

Nesse contexto, os Serviços Web (*Web Services*, ou WS, em inglês, notação mais utilizada) são a estratégia mais utilizada para a implementação da Arquitetura Orientada a Serviços. Um serviço Web é um componente de *software* reutilizável e fracamente acoplado, que encapsula funcionalidades discretas e que pode ser distribuído e acessado programaticamente. (Mahmoud, 2005). Serviços Web são acessíveis de forma independente da plataforma e da linguagem de programação utilizadas, pois sua comunicação segue padrões para transmissão de dados, considerando protocolos da Internet e arquivos textuais (geralmente em XML ou JSON). Esses serviços podem ser implementados utilizando o *Simple Object Access Protocol* (SOAP) ou o *Representational State Transfer* (REST), conforme discutido a seguir.

O SOAP trata-se de uma especificação para troca de informações estruturadas baseada em mensagens XML. Apesar de robusto e seguro, o SOAP adiciona um *overhead* considerável em forma de metainformação às suas mensagens, de modo que a serialização, a desserialização e a troca de mensagens é dispendiosa. Por esse motivo, a tecnologia é chamada de “Big Web Services”.

Outra solução para implementação de serviços é o REST, um estilo arquitetural baseado na transferência de representações de recursos de um servidor para um cliente (Fielding, 2000). A proposta do REST é utilizar apenas o protocolo HTTP e os métodos já implementados pelo mesmo, mais especificamente os verbos GET, PUT, POST e DELETE, para definir as ações a serem aplicadas em um recurso, enquanto os recursos são definidos em mensagens textuais no formato JSON ou XML. Devido a essa proposta enxuta, serviços REST envolvem uma sobrecarga menor do que os definidos pela especificação SOAP. Os princípios do estilo arquitetural REST são (Fielding, 2000):

1. **Cliente-Servidor**: cliente e servidor são desacoplados e comunicam-se por uma interface uniforme, sendo o desenvolvimento entre as partes é independente;

2. **Sem Estado:** o servidor não guarda o estado da aplicação para nenhum cliente, cada requisição deve enviar toda informação necessária para seu entendimento;
3. **Cache**: resultados gerados podem ser armazenados temporariamente para atender requisições iguais;
4. **Sistema em camadas**: a arquitetura do sistema é composta por camadas hierárquicas;
5. **Código sob Demanda**: servidores podem retornar pedaços de código (*scripts*) para serem executados no cliente;
6. **Interface Uniforme**: característica central do padrão, definida por quatro restrições de interface:
    a. **Identificação de recursos**: recursos devem ser acessados por meio de endereços únicos (URI);
    b. **Representação de recursos**: cliente e servidor devem compreender o formato padrão (dados e metadados);
    c. **Mensagens auto descritivas**: cada requisição do cliente e cada resposta do servidor devem ser uma mensagem padronizada que contém dados que a descrevem;
    d. **Hipermídia:** links devem ser usados dentro de recursos para encontrar outros recursos.

O estilo REST também pode ser uma abordagem para criação de microsserviços. Microsserviço é uma abordagem arquitetônica para desenvolver uma única aplicação dividida em pequenos serviços, construídos através de pequenas responsabilidades e publicados de forma independente. Cada serviço é executado em um processo próprio, os quais se comunicam por meio de mecanismos leves (LEWIS; FOWLER, 2014).

Na arquitetura de microsserviços o *back-end* é fragmentado em várias partes com escopos bem definidos. Trata-se de uma arquitetura altamente distribuída, que pode incluir pacotes independentes e diversos bancos de dados, sendo que o sistema pode ser facilmente descentralizado em múltiplos servidores físicos. Essas diferentes partes podem ser implementadas em linguagens distintas, de acordo com as demandas advindas de seus requisitos. É importante destacar que nem todo serviço Web é um microsserviço: é possível que a aplicação seja constituída por apenas um (grande) serviço monolítico, centralizado em um único pacote e um único banco.

O tutorial proposto neste capítulo utiliza o REST para implementar serviços Web com a linguagem JavaScript na plataforma Node.js. Destaca-se que essa abordagem pode ser utilizada tanto para construção de um serviço único como para uma implementação descentralizada, em que várias instâncias são produzidas para gerar microsserviços completamente independentes.

3.4. Programação Back-end com JavaScript

A plataforma Node.js permite a execução de código JavaScript e o desenvolvimento de aplicações do lado servidor em diferentes sistemas operacionais. Nesta seção será explorada a criação de um serviço Web no estilo REST utilizando JS no ambiente Node.js.

3.4.1. Instalação do software necessário

Para seguir este tutorial é necessário baixar o Node.js pelo seu site oficial[4] em versão compatível com o seu Sistema Operacional (SO). Ao instalar o Node.js você também deverá instalar o *Node Package Manager* (NPM), incluído no pacote de instalação. Os códigos apresentados foram testados no Node.js versão 21.1.0 e NPM versão 10.2.0. Para verificar as versões instaladas, execute os comandos "*node -v*" e "*npm -v*", respectivamente, no terminal do SO. O ambiente de desenvolvimento recomendado é o Visual Studio Code[5] (VSCode), mas é possível utilizar qualquer editor ou ambiente de sua preferência. O banco utilizado será o PostgreSQL em sua versão 16, caso se utilize outro banco relacional, é necessário instalar a biblioteca adequada.

A instalação e manejo de bibliotecas é uma atividade essencial em aplicações do lado servidor. O NPM é um gerenciador de pacotes que facilita a instalação e o gerenciamento de bibliotecas que podem ser usadas em aplicações executadas em Node.js. Sendo assim, ele será uma ferramenta essencial no projeto apresentado, sendo utilizado para a instalação das bibliotecas empregadas pela aplicação. O NPM não é a única opção de gerenciador de pacotes para Node.js, sua principal alternativa é o Yarn[6].

3.4.2. Inicialização do projeto

Após a instalação das ferramentas, pode-se iniciar a criação do projeto. Para isso, crie um diretório para o projeto e execute o comando "*npm init*" no terminal, já dentro do diretório de destino. O comando irá gerar um processo interativo que solicita informações sobre o projeto, como nome, versão, autor, descrição e licenças. Ao finalizar, o NPM criará um arquivo "*package.json*" que contém as informações sobre o projeto. Esse arquivo é indispensável pois irá permitir o gerenciamento das bibliotecas instaladas pelo seu criador e colaboradores.

O próximo comando do NPM a ser utilizado será o "*npm install*" ou "*npm* i". Esse é o comando utilizado para instalar pacotes no projeto, de forma que o NPM procura o(s) pacote(s) especificado(s) em seu repositório e o(s) baixa para o diretório "*node_modules*" dentro projeto. O comando pode ser executado indicando o pacote desejado, por exemplo, o comando "*npm i express*" instala a versão mais recente e estável do *framework Express*, ou sem nenhum argumento. Nesse último caso, o NPM irá baixar as bibliotecas listadas no arquivo "*package.json"*. A cada instalação, o pacote baixado é adicionado à seção "*dependencies*" do arquivo "*package.json*".

Para o projeto descrito será necessário instalar as seguintes dependências: *express, sequelize, pg* e *cors*. Isso pode ser realizado pelo comando "*npm i express sequelize pg cors*". Cada um desses pacotes é explicado a seguir.

O Express[7] trata-se de um *framework* de aplicações Web que fornece uma maneira conveniente e flexível de criar e gerenciar aplicações. O Express fornece várias funcionalidades, como gerenciamento de rotas, tratamento de erros, integração com banco de dados e recursos de segurança, automatizando tarefas repetitivas de

[4] nodejs.org
[5] code.visualstudio.com
[6] yarnpkg.com
[7] https://expressjs.com/

desenvolvimento. Em particular, neste tutorial o *framework* irá auxiliar na criação de um serviço Web baseado em *Application Programming Interfaces* (APIs) do tipo REST.

O Sequelize[8] é uma ferramenta de Mapeamento Objeto-Relacional (ORM) (*Object Relational Mapper*, em inglês), técnica de programação que permite relacionar objetos com dados em um banco de dados. Ao utilizar uma ferramenta ORM, o desenvolvedor não precisa se preocupar com a escrita de código SQL, dado que a técnica realiza o mapeamento dos objetos em tabelas e dados no banco, automaticamente.

Por fim, as bibliotecas "*pg*"[9] e "*cors*"[10] serão utilizadas para conexão com o banco PostegreSQL e para configuração do *Cross-Origin Resource Sharing*, respectivamente.

Para finalizar a configuração do projeto, é necessário incluir a linha *"type": "module" no* arquivo "*package.json*". Essa configuração é essencial para se utilizar o novo sistema de importação definido pela ECMAScript 6 (ES6). O Quadro 3.9 mostra o arquivo de configuração resultante.

Quadro 3.9. Arquivo *package.json* após instalação dos pacotes via *npm*

```
{
 "name": "pet-app-back-pg",
 "version": "1.0.0",
 "description": "back-end do pet-app",
 "main": "index.js",
 "type": "module",
 "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
 },
 "author": "bruna",
 "license": "ISC",
 "dependencies": {
  "cors": "^2.8.5",
  "express": "^4.18.2",
  "pg": "^8.11.3",
  "sequelize": "^6.35.1"
 }
}
```

3.4.3. Início do Código

Após a configuração do projeto pelo NPM, é possível iniciar a implementação da aplicação. Primeiramente, é proposta a criação de um serviço de teste que retorna apenas um texto, indicando que a aplicação está sendo executada e está acessível. Para isso, crie um diretório "*src*" dentro do diretório do projeto: esse será o diretório que irá abrigar todo o código fonte em JS. Em seguida, crie um arquivo "app.js" com o código do

[8] https://sequelize.org/
[9] https://www.npmjs.com/package/pg
[10] https://www.npmjs.com/package/cors

Quadro 3.10. Para iniciar a aplicação basta executar o comando "*node ./src/app.js*" no terminal (ou utilizar uma extensão como o *Code Runner*[11] no VSCode). O resultado pode ser acessado pela URL: *http://localhost:3000/hello*.

Observe que o código do Quadro 3.10 importa o *express* (conforme o padrão ES6) e inicializa uma instância do mesmo em uma constante, nomeada "*app*". Essa constante é chamada para definir um serviço, mapeado pelo método GET e pela URI "*/hello*", e interceptado por uma função *callback* que retorna o texto "*Hello World!*". Esse processo é chamado de roteamento, de forma que o *framework* irá mapear um método HTTP e uma URI a uma função desejada. Em resumo, o roteamento realizado pela instância do Express é feito por meio de um método HTTP (verbos GET, PUT, POST e DELETE, conforme discutido na Seção 3.3) que requere como argumentos, minimamente, um caminho (URI do recurso) e uma função que será executada. A função roteada receberá os atributos "*request*" e "*response*", referentes à sua chamada. Ao final do código, é indicado que a aplicação deve ouvir as requisições na porta 3000 do servidor (essa é a porta padrão recomendada).

Quadro 3.10. Código fonte do arquivo *app.js*: exemplo de um primeiro serviço do tipo REST.

```
import express from 'express'

const app = express()
const port = 3000
app.get('/hello', function(request, response) {
   response.send('Hello World!')
})
app.listen(port, function() { console.log('Servidor na porta ' + port) })
```

O próximo passo do tutorial envolve a criação do Banco de Dados e dos modelos que irão gerar as tabelas no banco.

3.4.4. Modelagem de dados

O serviço que será implementado permitirá cadastrar clientes e seus animais de estimação, de forma que é preciso representar essas informações em duas tabelas, relacionadas entre sim. Como a proposta do tutorial é utilizar uma ferramenta de ORM, no caso, a biblioteca Sequelize, é necessário apenas a criação de um usuário e um banco, os quais serão informados no momento da conexão. Como mencionado na etapa de instalação, o tutorial se baseia no PostgreSQL (versão 16.1.1). Considera-se que o banco foi instalado na porta 5432. Recomenda-se criar um usuário e um banco dedicados à aplicação, conforme mostra o Quadro 3.11. Caso prefira, essa criação pode ser realizada também pelo *software* gráfico *pgAdmin*, o qual acompanha a instalação.

Quadro 3.11. Comandos para criação de usuário e banco via terminal (Unix). Usuário "userdev", senha "123456" e banco "devpet"

```
psql -d postgres
```

[11] https://marketplace.visualstudio.com/items?itemName=formulahendry.code-runner

```
CREATE ROLE userdev WITH LOGIN PASSWORD '123456';
ALTER ROLE userdev CREATEDB;
\q
psql -d postgres -U userdev
CREATE DATABASE devpet;
```

Retornando ao código JS, dentro do diretório "*src*", crie um arquivo "*model.js*". No ambiente Node.js cada arquivo é visto como um módulo JS e esse será o módulo que irá conter a conexão com o banco e criação dos modelos. O código é apresentado no Quadro 3.12. Inicialmente, são importadas as classes necessárias da biblioteca Sequelize. Em seguida, o primeiro passo cria uma conexão com o banco ao instanciar um objeto, nomeado "*database*", do tipo Sequelize com os dados (nome do banco, usuário e senha) do banco recém-criado, além da indicação do dialeto (*postgres*) e domínio (*localhost*). Tanto na inicialização da conexão como na instanciação dos modelos é requerida cautela, pois, o uso de separadores, como colchetes e vírgulas, deve ser respeitado. Por exemplo, ao instanciar um novo objeto Sequelize, os primeiros argumentos são configurações textuais de banco, usuário e senha, enquanto o terceiro argumento trata-se de um conjunto de opções delimitado por chaves.

Ainda no Quadro 3.12, após a instanciação do objeto "*database*", são inicializados os modelos de cliente e animal de estimação, as classes "*Client*" e "*Pet*", respectivamente. Ambas as classes estendem a classe "*Model*" da biblioteca do Sequelize e em seguida utiliza o método herdado "*init*" para definição de suas tabelas. A definição de uma tabela se dá pela indicação de seus atributos (colunas) e, ao final, das configurações da tabela. Por exemplo, o modelo de "Pet" inclui os atributos "*id, name, type, breed* e *birth*", sendo *id* a chave-primária (um inteiro auto incrementável), os demais são atributos textuais, sendo *name* e *type* duas colunas que não aceitam valores nulos. Novamente, é importante destacar que é necessário cautela na colocação dos separadores.

Ao final da chamada de "*init*", nas opções do modelo, é indicado um objeto Sequelize, o qual é a instância de conexão, nesse caso, o objeto "*database*", e demais opções. No exemplo, indica-se que *timestamps* não são desejadas. Outra opção comum é nomear a tabela, o que é feito pela propriedade "*tableName*". Quando a tabela correspondente ao modelo não é nomeada, a biblioteca pluraliza o nome do modelo (*e.g.*, Pet se torna a tabela Pets).

Quadro 3.12. Arquivo *models.js*

```
import { Sequelize, Model, DataTypes } from 'sequelize'

const database = new Sequelize('devpet', 'userdev', '123456',
   { dialect: 'postgres', host: 'localhost' })

class Client extends Model {}
Client.init( {
   id: { type: DataTypes.INTEGER, autoIncrement: true, primaryKey: true },
   name: { type: DataTypes.STRING, allowNull: false },
   document: { type: DataTypes.STRING, allowNull: false }
```

```
}, { sequelize: database, timestamps: false })

class Pet extends Model {}
Pet.init({
   id: { type: DataTypes.INTEGER, autoIncrement: true, primaryKey: true },
   name: { type: DataTypes.STRING, allowNull: false },
   type: { type: DataTypes.STRING, allowNull: false },
   breed: DataTypes.STRING,
   birth: DataTypes.STRING
}, { sequelize: database, timestamps: false })

Pet.belongsTo(Client)
Client.hasMany(Pet)
export default { Client, Pet }
```

Após definidos os modelos, as relações entre eles podem ser adicionadas. O Sequelize suporta as associações clássicas de Um-Para-Um, Um-Para-Muitos e Muitos-Para-Muitos. Isso pode ser realizado por meio dos comandos: *HasOne, BelongsTo, HasMany* e *BelongsToMany*. No código do exemplo (Quadro 3.12), o cliente pode possuir vários animais de estimação (*pets*), enquanto um *pet* deve ser de responsabilidade de um cliente. Esse relacionamento foi indicado pelas operações *Pet.belongsTo(Client)* e *Client.hasMany(Pet)*. Como resultado, a tabela *Pets* no banco irá conter uma chave estrangeira para a tabela *Clients*, chamada *ClientId*. A chave estrangeira também pode ser renomeada, utilizando a opção *foreignKey* na definição da relação *belongsTo*. Note que a relação foi estabelecida em duas vias, o que permitirá a recuperação de informações do cliente pelo modelo *Pet*, e a recuperação de *pets* pelo modelo *Client*.

Por fim, para que os modelos criados sejam acessíveis em outros módulos do projeto, ambos são exportados pelo comando *export default*.

Para sincronizar os modelos implementados, foi criado um arquivo (*dbsync.js*) que os importa e invoca o método "*sync*" (Quadro 3.13). Esse método sincroniza as tabelas com o banco, criando-as caso não existam. Caso as tabelas existam, pode-se utilizar a opção "{ *alter: true* }" para forçar uma atualização do banco (não indicado para sistemas em produção). O código pode ser executado pelo comando "*node ./src/dbsync.js*".

Quadro 3.13. Sincronização dos modelos (arquivo *dbsync.js*)

```
import model from './model.js'
await model.Client.sync()
await model.Pet.sync()
```

A biblioteca do Sequelize permite outros comandos para criação dos modelos. Sua documentação pode ser consultada em seu site oficial[12]. Além disso, caso se utilize outro banco, como MySQL, é necessário instalar a biblioteca adequada (via NPM) e apontar o dialeto correspondente na instância do objeto Sequelize. A mudança do

[12] https://sequelize.org/docs/v6/core-concepts/model-basics/

número da porta também implica que essa deve ser indicada entre os parâmetros de configuração.

3.4.5. Controlador

Com os modelos prontos, pode-se realizar as ações de criação, leitura, alteração e remoção dos dados no banco. Como visto anteriormente, esse processo pode ser realizado ao mapear um método REST, disponibilizado pela API do Express, a uma função *callback*, técnica chamada de roteamento. No entanto, priorizando a organização do código, será criado um módulo controlador, o qual conterá as funções roteadas que farão uso do modelo. Nesse exemplo, foi criado apenas um modelo e apenas um controlador, porém, esses podem ser divididos em vários módulos, cada um dedicado a uma entidade.

O Quadro 3.14 apresenta parte do código do controlador (*controller.js*), referente às funções relacionadas ao modelo do cliente. Todas as funções do controlador já são exportadas por padrão no início do módulo. Observe que todas as funções recebem os argumentos "*request*" e "*response*", pois, conforme explicado na Seção 3.4.2, elas serão roteadas pela aplicação. Esses atributos irão conter os dados da requisição e da resposta às mensagens HTTP (conforme discutido na Seção 3.2.1), respectivamente, de forma que é possível recuperar informações da requisição e determinar os dados de retorno.

Para realizar as operações no banco, são utilizados os métodos gerados para os modelos pela biblioteca do Sequelize. Por exemplo, o método "*findAll()*" recupera todos os registros no banco, enquanto o "*findByPk(id)*" retorna um registro específico. Portanto, a utilização desses métodos é feita de acordo com a documentação de sua biblioteca (a qual é acessível no próprio VSCode). Por envolverem acesso ao banco, todos esses métodos são executados de forma assíncrona, o que implicou em seu tratamento via promessas. Nesse caso, o resultado de cada promessa é referente a uma operação realizada no banco e que pode ser convertido e enviado em formato JSON no objeto *response*. Já os valores de entrada são recuperados pelo objeto *request*.

Como mencionado, uma mensagem de requisição HTTP pode ou não apresentar um corpo. Mensagens do tipo GET não apresentam um corpo, de forma que são utilizados parâmetros na URI para o envio dados na requisição. Isso pode ser observado na função "*findClientByPk*" no Quadro 3.14. Ao utilizar o método de busca de um registro pelo *id*, o valor de *id* é recebido via parâmetro, sendo recuperado ao acessar o "*request.params.id*". Nas operações de alteração e remoção o parâmetro *id* é utilizado para formar uma cláusula de consulta, determinada pela condição "*where: { id: request.params.id }*".

Por outro lado, quando o verbo POST é utilizado, o corpo da mensagem contém dados. No Quadro 3.14, a função "*createClient*" utiliza o método "*create*" do modelo e recupera informações do corpo da requisição para determinar os valores dos atributos "*name"* e "*document"*. Nesse caso, espera-se que os valores recebidos estejam em "*request.body.name*" e "*request.body.document*".

Considerando a resposta, é comum que sejam retornados dados em formato JSON, de forma que na maioria dos casos foi utilizado o comando "*response.json(result)*". Em algumas situações pode ser adequado um resultado vazio ou até mesmo uma mensagem de texto, o que pode ser obtido pelo método "*send()*". Essas

chamadas podem ser encadeadas com o *status* da resposta. O *status* retornado deve estar de acordo com a convenção para mensagens HTTP. *Status* na casa de 200 indicam sucesso, enquanto na casa de 400 são requisições malformadas e na casa de 500 são relacionados a erros no servidor.

Quadro 3.14. Código fonte do módulo *controller.js* (parte 1)

```
import model from './model.js'
export default { findAllClients, findAllPets, findClientByPk, findPetByPk, findPetsOfClient,
createClient, createPet, updateClient, updatePet, deleteClientByPk, deletePetByPk }

function findAllClients(request, response) {
  model.Client.findAll().then(function (results) {
    response.json(results).status(200)
  }).catch(function (e) { console.log(e) })
}
function findClientByPk(request, response) {
  model.Client.findByPk(request.params.id).then(function (result) {
    response.json(result).status(200)
  }).catch(function (e) { console.log(e) })
}

function createClient(request, response) {
  model.Client.create(
    { name: request.body.name, document: request.body.document }
  ).then(function (result) {
    response.status(201).json(result)
  }).catch(function (e) { console.log(e) })
}

function updateClient(request, response) {
  model.Client.update(
    { name: request.body.name, document: request.body.document },
    { where: { id: request.params.id } }
  ).then(function (result) {
    response.status(200).send()
  }).catch(function (e) { console.log(e) })
}
```

```
function deleteClientByPk(request, response) {
  model.Client.destroy({ where: { id: request.params.id } }
  ).then(function (result) {
    if (result == 1) {
      response.status(200).send()
    } else {
      response.status(404).send()
    }
  }).catch(function (e) { console.log(e) })
}
```

O Quadro 3.15 contém a continuação do código do controlador, referente às operações com o modelo *Pet*. Observe que as tarefas são similares, com exceção da função "*findPetsOfClient*", a qual retorna os animais de estimação de um cliente específico. Nesse caso é utilizado o método "*findAll*" do modelo com uma cláusula de consulta que indica que deverão ser retornados apenas os registros cuja chave-estrangeira seja igual a chave do cliente indicado no parâmetro da requisição (*i.e.*, *where: { ClientId: request.params.id }*).

Quadro 3.15. Código fonte do módulo *controller.js* (parte 2)

```
function findAllPets(request, response) {
  model.Pet.findAll().then(function (results) {
    response.json(results).status(200)
  }).catch(function (e) { console.log(e) })
}

function findPetByPk(request, response) {
  model.Pet.findByPk(request.params.id, { include: model.Client }
  ).then(function (result) {
      response.json(result).status(200)
    }).catch(function (e) { console.log(e) })
}
function findPetsOfClient(request, response) {
  model.Pet.findAll({ where: { ClientId: request.params.id } }
  ).then(function (results) {
    response.json(results).status(200)
  }).catch(function (e) { console.log(e) })
}

function createPet(request, response) {
  model.Pet.create({
    name: request.body.name, type: request.body.type,
    breed: request.body.breed, birth: request.body.birth, ClientId: request.body.ClientId }
```

```
  ).then(function (result) {
    response.json(result).status(201)
  }).catch(function (e) { console.log(e) })
}

function updatePet(request, response) {
  model.Pet.update({
    name: request.body.name, type: request.body.type,
    breed: request.body.breed, birth: request.body.birth, ClientId: request.body.client },
  { where: { id: request.params.id } }
  ).then(function (result) {
    response.json(result).send()
  }).catch(function (e) { console.log(e) })
}
function deletePetByPk(request, response) {
  model.Pet.destroy({ where: { id: request.params.id } }
  ).then(function (result) {
    if (result == 1) { response.status(200).send()
    } else { response.status(404).send()
    }
  }).catch(function (e) { console.log(e) })
}
```

3.4.6. Rotas

Com o controlador pronto, podem ser indicadas as rotas da API. Isso poderia ser implementado diretamente no arquivo "*src/app.js*", no entanto, visando uma melhor organização, foi criado o arquivo individualizado "*src/routes.js*". Nesse tutorial, há apenas um módulo de rotas, no entanto, podem ser criados vários, o que é indicado para diferenciar funcionalidades (*e.g.*, um módulo dedicado às rotas de autenticação e segurança).

O Quadro 3.16 demonstra a definição das rotas associadas às funções implementadas pelo controlador. O objeto "*routes*" é uma instância da classe "*Routes*" do Express e, por meio desse, são mapeados os métodos e as URIs às funções criadas. Ao final, o objeto "*routes*" é exportado. Observe que nenhuma função recebe argumentos, de forma que apenas seus nomes são indicados. Isso ocorre pois o *framework* irá invocar as funções com os argumentos *request* e *response*. Por exemplo, o cadastro de um cliente é roteado para a função "*createClient*" pelo método "*post*" no recurso "/*clients*".

Outro ponto essencial do roteamento é a indicação do nome dos parâmetros na URI. No exemplo, as operações de busca, alteração e deleção de recursos fazem uso de um parâmetro *id*, indicado nas URIs por "*:id*" (o caractere de dois pontos é utilizado na definição de parâmetros nas rotas do Express). O nome do parâmetro definido pela rota

deverá ser respeitado tanto pela URI da requisição quanto pela função implementada no controlador.

Quadro 3.16. Arquivo *routes.js*

```
import express from 'express'
import controller from './controller.js'

const routes = express.Router()

routes.get('/clients', controller.findAllClients)
routes.get('/clients/:id', controller.findClientByPk)
routes.post('/clients', controller.createClient)
routes.put('/clients/:id', controller.updateClient)
routes.delete('/clients/:id', controller.deleteClientByPk)

routes.get('/pets', controller.findAllPets)
routes.get('/pets/:id', controller.findPetByPk)
routes.post('/pets', controller.createPet)
routes.put('/pets/:id', controller.updatePet)
routes.delete('/pets/:id', controller.deletePetByPk)
routes.get('/clients/:id/pets', controller.findPetsOfClient)

export default routes
```

É importante destacar que a nomenclatura aplicada nas rotas do Quadro 3.16 respeita as convenções do estilo REST, ou seja, a pluralização dos nomes dos recursos, a identificação de itens específicos pelo caminho dos recursos e seu identificador (*i.e.*, */recursos/id*) e de recursos relacionados pela composição da URI (*i.e.*, */recursos/id/recursoRelacionado*). A rota definida para a recuperação dos animais de um cliente específico é um exemplo dessa última situação, pois sua URI é determinada pelo caminho desse cliente ("*/clients/:id/pets*").

3.4.7. Finalização da configuração do serviço

Para finalizar a API é necessário incluir as rotas programadas na instância do Express. Além disso, devem ser adicionadas algumas configurações para que a API aceite o formato JSON e as requisições da página que será implementada.

O Quadro 3.17 apresenta a versão final do arquivo "*src/app.js*". Nele foram importados a biblioteca do CORS, instalada em passos anteriores, e o recém-criado módulo de roteamento ("*src/routes.js*"). A linha "*app.use(cors())*" foi adicionada a fim de permitir requisições "*cross-origin*", ou seja, basicamente, o CORS permite informar ao navegador que a aplicação terá seus recursos acessados por outra aplicação em uma origem distinta (em termos de domínio). Caso essa linha não seja incluída, será gerado um erro de CORS no navegador quando a página Web, que será executada em um servidor de porta diferente da aplicação Node.js, realizar uma requisição HTTP para a API. As duas linhas que seguem a habilitação do CORS são necessárias para que as requisições do tipo POST e PUT possam enviar dados em JSON no corpo da mensagem

HTTP. Após as configurações, o módulo de rotas é associado à instância do Express, pelo comando "*app.use(routes)*".

Quadro 3.17. Arquivo *app.js*

```
import express from 'express'
import cors from 'cors'
import routes from './routes.js'
const app = express()
const port = 3000
/* Habilita requisições de diferentes origens */
app.use(cors())
/* Permite receber dados em json */
app.use(express.json())
/* Indica como o parser do json será realizado */
app.use(express.urlencoded({extended: true}))

app.use(routes)
app.listen(port, function() { console.log('Servidor rodando na porta ' + port) } )
```

3.4.8. Testando a API por requisições

Antes de iniciar a criação da página Web que será o cliente da API implementada, é possível realizar testes para garantir que o código implementado não possui erros e está de acordo com os requisitos planejados. Existem ferramentas que simulam clientes REST ao executar requisições iguais àquelas que são envidas pelos navegadores. Além de permitir apontar o verbo e a URI da requisição, essas ferramentas possibilitam personalizar as mensagens HTTP, incluindo, por exemplo, o corpo da mensagem, e exibir o resultado retornado pelo servidor. A ferramenta que foi utilizada para testar o código do exemplo foi a Thunder Client[13], a qual pode ser instalada como extensão no VSCode.

A Figura 3.4 mostra a configuração da ferramenta para inclusão de um registro via API pelo método POST. A interface da ferramenta permite a seleção do verbo, nesse caso o POST, e a inclusão da URI do recurso (*localhost:3000/clients*). Na aba *body* é possível enviar dados em diferentes formatos no corpo da mensagem sendo que, no exemplo da figura, foi incluído um JSON com nome e documento do cliente a ser cadastrado. A parte direita da ferramenta mostra o resultado da requisição, conforme definido pela API implementada, que lista o cliente cadastrado com o seu *id* e o *status* 201 (criado). Os clientes cadastrados podem ser recuperados via método GET, utilizado a mesma URI de recurso. A Figura 3.5 mostra o resultado após o cadastro de três clientes.

[13] https://www.thunderclient.com/

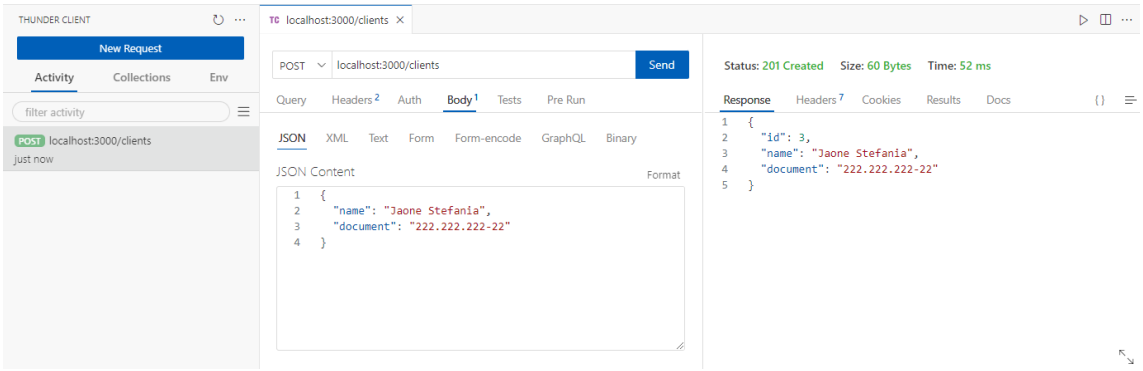

Figura 3.4. Cadastro de um cliente via método POST

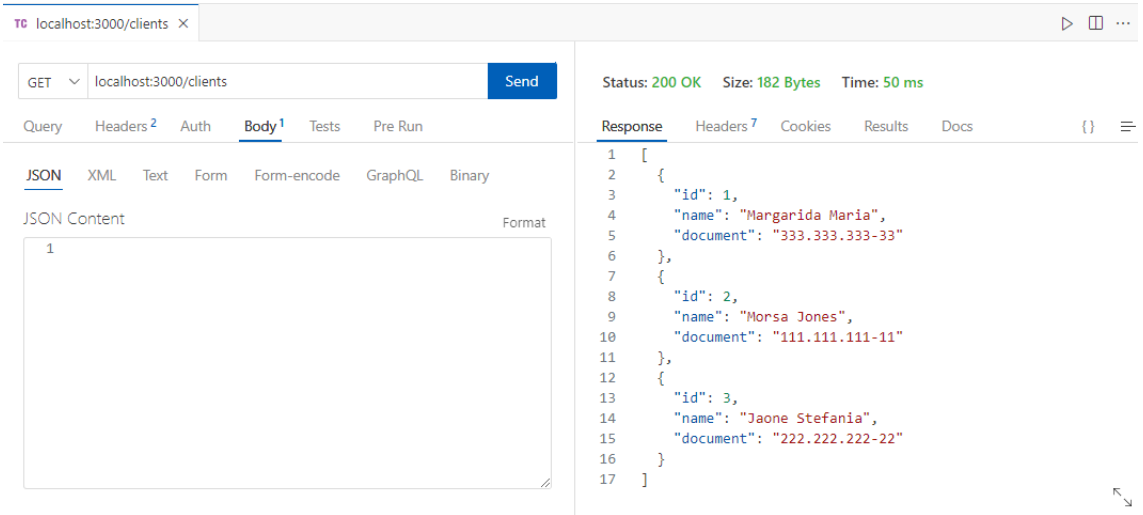

Figura 3.5. Consulta por todos os clientes cadastrados

Na Figura 3.6 é apresentado mais um exemplo em que é feito o cadastro de um animal de estimação para um cliente específico, cujo *id* é 3. O JSON correspondente contém os dados do modelo *Pet*, incluindo a chave estrangeira do cliente. Após essa inserção, é possível buscar pelos animais do cliente de *id* igual a 3, como demonstrado na Figura 3.7 (*localhost:3000/clients/3/pets*).

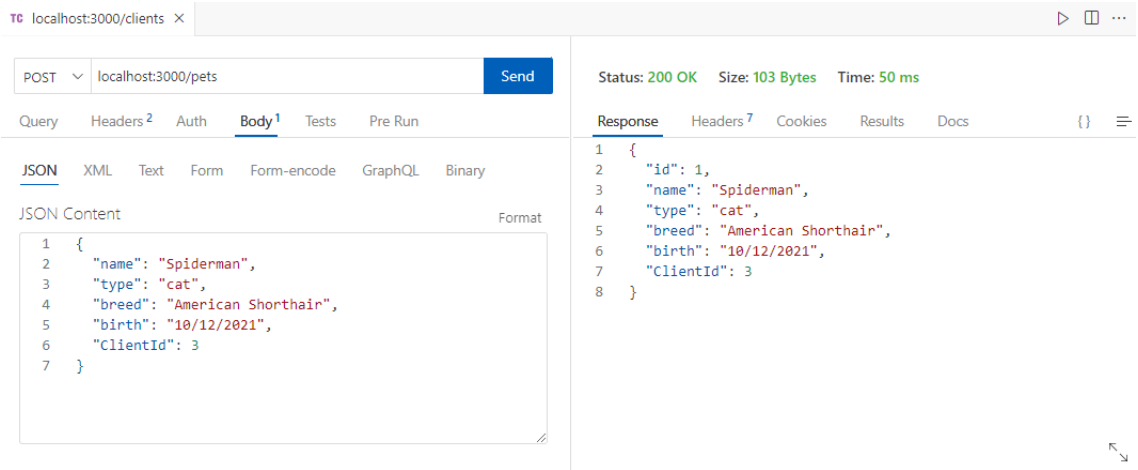

Figura 3.6. Cadastro de um pet via método POST

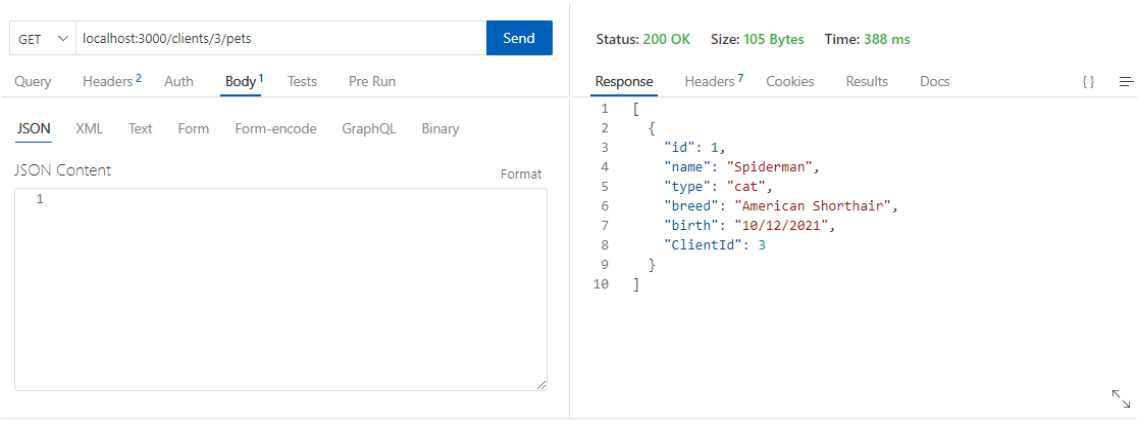

Figura 3.7. Listagem dos pets do cliente de id igual a 3

Operações de alteração e remoção demandam que a URI aponte para um recurso de *id* específico. A Figura 3.8 exemplifica a operação de PUT na URI do cliente de *id* 3 (*localhost:3000/clients*) para alteração de seu nome.

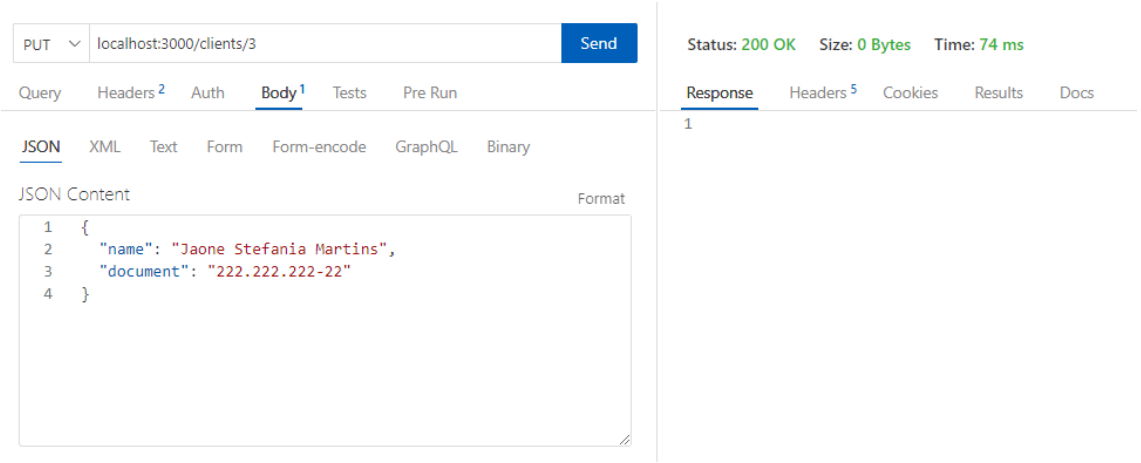

Figura 3.8. Alteração do nome do cliente de id igual a 3

Para garantir a corretude do código implementado é recomendando testar todas as funções de cadastro, consulta, alteração e remoção de registros.

3.5. Programação Front-end com JavaScript

Esta seção irá demonstrar como criar uma página Web que seja capaz de utilizar o pequeno serviço Web com API REST criado. A linguagem JavaScript apresenta diversos *frameworks* e bibliotecas para a programação *front-end*, no entanto, objetivando o foco na linguagem, esse tutorial utilizará o JS "puro", ou seja, sem esse ferramental de apoio. O uso da linguagem JS sem o suporte de um *framework* ou biblioteca de grande porte é chamado de *Vanilla JavaScript* pela comunidade. Por ter como alvo de estudo o uso de JavaScript, este tutorial não entrará em detalhes sobre o código HTML e CSS apresentado. As Seções 3.2 e 3.3 apresentam conceitos básicos e indicam um material complementar para o estudo dessas tecnologias.

O Quadro 3.18 contém o código HTML da página (*index.html*). Observe que dois arquivos JS são incluídos em seu cabeçalho. O primeiro (*https://unpkg.com/axios/dist/axios.min.js*) trata-se da biblioteca Axios[14], a qual será utilizada para realizar requisições HTTP para o serviço Web criado. O Axios simplifica essas chamadas e será utilizado a fim de deixar o código mais claro, porém o JS apresenta uma API própria, chamada *Fetch*, a qual também pode ser utilizada para requisições baseadas em promessas. O próximo script incluído trata-se do código JavaScript que será desenvolvido (*script.js*). Esse código é referenciado logo no início da página pelo evento "*onload*", o qual invoca a função "*getClients()*". No cabeçalho também é incluído um arquivo CSS (*style.css*) que é listado no Quadro 3.19.

O código HTML apresenta a estrutura básica de formulários para o cadastro de registros no banco. Inicialmente, o formulário para inclusão de animal do cliente não deverá estar visível na tela, devendo aparecer apenas quando um cliente for aberto. O controle de visibilidade será feito por meio de CSS, mais especificamente pelas classes "*.hide*" e "*.show*". O Quadro 3.19 contém essas classes e as demais definições de estilo.

[14] https://axios-http.com/

Quadro 3.18. Código da página *index.html*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="https://unpkg.com/axios/dist/axios.min.js"></script>
  <script src="./script.js"></script>
  <link rel="stylesheet" href="./style.css">
  <title>Pet App</title>
</head>
<body onload="getClients()">
  <h1>Pet App</h1>
  <div id="div-clients">
    <h3>Add Client</h3>
    <form>
      <label for="name">Name:</label>
      <input type="text" name="name" id="name" required>
      <label for="document">Document:</label>
      <input type="text" name="document" id="document" required>
      <input type="button" value="Submit" onclick="postClient()">
    </form>
  </div>
  <div id="div-client-pets"></div>
  <div id="div-form-pet">
    <h3>Add New Pet</h3>
    <form>
      <label for="petname">Name:</label>
      <input type="text" name="petname" id="petname" required>
      <label for="type">Type:</label>
      <select name="type" id="type" required>
        <option value="Dog">Dog</option>
        <option value="Cat">Cat</option>
        <option value="Bird">Bird</option>
        <option value="Turtle">Turtle</option>
        <option value="Rabitt">Rabitt</option>
      </select>
      <label for="breed">Breed:</label>
      <input type="text" name="breed" id="breed" required>
      <label for="birth">Birth:</label>
      <input type="date" name="birth" id="birth" required>
      <input type="button" value="Submit" id="submit-pet">
    </form>
  </div>
</body>
</html>
```

Quadro 3.19. Código da folha de estilos *style.css*

```
body {
   font-family: 'Gill Sans', 'Gill Sans MT', Calibri, 'Trebuchet MS', sans-serif;
}
h1 {
   color: white;
   background-color: #4B99AD;
   margin: 0px;
   padding: 10px;
}
.client-list {
   background-color: gainsboro;
   padding-left: 10px;
   border-bottom: 1px solid gray;
}
.pets-list {
   background-color: aliceblue;
   padding-left: 10px;
   border-bottom: 1px solid cyan;
}
.client-list spam {
   cursor: pointer;
}
.client-list p {
   display: inline-block;
   margin-right: 10px;
}
.pets-list p {
   display: inline-block;
   margin-right: 10px;
}
form {
   margin: 10px;
}
button {
   margin-left: 5px;
}
.hide {
   display: none;
}
.show {
   display: block;
}
```

Com a estrutura da página pronta, o código JS correspondente pode ser integrado. O código será contínuo e fará parte do arquivo "*script.js*", porém, para fins didáticos, ele será dividido em múltiplos quadros. O Quadro 3.20 apresenta a primeira parte, a qual é responsável por listar os clientes cadastrados na tela. A primeira linha do

código define uma constante com o endereço do serviço. Em seguida, vem a função "*getClients*()", a qual é chamada pelo evento "*onload*" do "*body*" da página. A função inicialmente garante que o formulário para cadastro de animais está oculto, ao recuperar seu container (*div*) e excluir e adicionar as classes adequadas. Em seguida, é realizada uma requisição do tipo GET pela biblioteca Axios na URI dos clientes, a qual retorna, por meio de promessa, uma resposta ou um erro. No caso de uma resposta, um JSON que contém a lista de clientes é recuperado do objeto de resposta pelo comando "*response.data*". A lista de clientes é percorrida e, para cada um, é invocada a função "addClient", a qual é apresentada no Quadro 3.21.

Quadro 3.20. Arquivo *scripts.js* (parte 1): listagem dos clientes

```
const api = 'http://localhost:3000/'
function getClients() {
    document.getElementById('div-form-pet').classList.remove('show')
    document.getElementById('div-form-pet').classList.add('hide')
    axios.get(api + 'clients').then(function(response) {
        const clients = response.data
        for(let client of clients) {
            addClient(client)
        }
        console.log(clients)
    })
    .catch(
        function (error) {
            console.error(error)
        })
}
```

O Quadro 3.21 apresenta a continuação do código com a função "*addClient*", responsável por adicionar cada cliente na tela. Conforme explanado na Seção 3.4, esse comportamento pode ser obtido ao manipular o DOM da página. São criados quatro elementos: (1) um parágrafo, que contém o nome e o documento do cliente; (2) um botão para abrir o cliente; (3) um botão para deletar o cliente; e (4) um container, que agrupa os três elementos anteriores e é anexado a uma *div* já presente no documento. O container também guarda o *id* do cliente, dado que esse será indispensável para sua posterior identificação. Na criação dos botões, note que são atribuídas funções aos eventos de clique por meio da definição de atributo (*i.e.*, pelo método "*setAttribute*"). Essas funções são demonstradas no Quadro 3.22.

Conforme mostra o Quadro 3.21, as funções referenciadas pelos botões criados de forma dinâmica são a de visualização de animais de um cliente ("*getPets*") e a de remoção do cliente ("*deleteClient*"). Em ambos os casos, o evento associado é o de clique ("*onclick*") e são incluídos argumentos à chamada das funções. O Quadro 3.22 apresenta a implementação do código. A primeira função apresentada, "*deleteClient(id)*", recebe o identificador do cliente a ser removido e realiza uma requisição do tipo DELETE, com o auxílio da biblioteca do Axios, na URI dos clientes concatenada com uma barra e um *id*. Na Seção 3.4.5, foi demonstrado que na construção das rotas o *id* é mapeado como um parâmetro na própria URI. Em caso de sucesso, o cliente removido do sistema deve, também, ser removido da interface, o que é executado

pelo comando "*document.getElementById(id).remove()*" (esse comando depende da definição do *id* do container, conforme feito no Quadro 3.21).

Quadro 3.21. Arquivo *scripts.js* (parte 2): adição de cliente no documento HTML

```
function addClient(client) {
    const clientItem = document.createElement('p')
    clientItem.innerText = client.name + ', ' + client.document

    const viewButton = document.createElement('button')
    viewButton.innerText = 'Open'
    viewButton.setAttribute('onclick','getPets(' + client.id + ', "' + client.name + '")')

    const trashButton = document.createElement('button')
    trashButton.innerText = 'Delete'
    trashButton.setAttribute('onclick','deleteClient(' + client.id + ')')

    const container = document.createElement('div')
    container.classList.add('client-list')
    container.id = client.id
    container.appendChild(clientItem)
    container.appendChild(viewButton)
    container.appendChild(trashButton)

    const divClients = document.getElementById('div-clients')
    divClients.appendChild(container)
}
```

As próximas funções do Quadro 3.22 tratam da listagem dos animais de estimação do cliente selecionado. Os primeiros comandos tratam da configuração do formulário para adição de um novo animal. Nesse caso, além de configurar a visibilidade do formulário, a função do botão de submissão é definida dado que é necessário que o *id* do cliente, responsável por aquele *pet*, seja enviado como argumento para a função. Em seguida, a biblioteca do Axios é utilizada para realizar uma requisição do tipo GET, nesse caso, na rota definida para o retorno dos animais de um cliente específico (*api + 'clients/' + clientId + '/pets'*). Em caso de sucesso, são listados os animais na tela, utilizando operações similares às utilizadas para a listagem de clientes.

Quadro 3.22. HTML Arquivo *scripts.js* (parte 3): adição de cliente no documento HTML

```
function deleteClient(id) {
    axios.delete(api + 'clients/' + id
    ).then(function (response) {
        document.getElementById(id).remove()
    }).catch(function (error) {
        console.error(error)
    })
}
```

```
function getPets(clientId, clientName) {
   document.getElementById('div-form-pet').classList.remove('hide')
   document.getElementById('div-form-pet').classList.add('show')
   document.getElementById('submit-pet').setAttribute('onclick','postPet(' + clientId + ')')


   const divPets = document.getElementById('div-client-pets')
   divPets.innerHTML = ''

   axios.get(api + 'clients/' + clientId + '/pets').then(function (response) {
      const h2 = document.createElement('h2')
      h2.innerText = 'Pets of ' + clientName
      divPets.appendChild(h2)
      const pets = response.data
      for (let pet of pets) {
         addPet(pet)
      }
   }).catch(function (error) {
      console.error(error)
   })
}

function addPet(pet) {
   const p = document.createElement('p')
   p.innerText = pet.name + ', ' + pet.type + ', ' + pet.breed + ', ' + pet.birth
   const container = document.createElement('div')
   container.classList.add('pets-list')
   container.id = pet.id
   container.appendChild(p)
   document.getElementById('div-client-pets').appendChild(container)
}
```

No código HTML apresentado no Quadro 3.18, o formulário dedicado à inclusão de clientes possui um botão que faz referência à função "*postClient()*", enquanto o Quadro 3.22 possui uma referência à função "*postPet(clientId)*". O Quadro 3.23 contém o código dessas duas funções, as quais possuem um funcionamento muito similar: ambas recuperam os dados dos formulários, realizam uma requisição do tipo POST passando esses valores no formato JSON e, em caso de sucesso, adicionam o registro no documento e limpam o formulário.

Quadro 3.23 - Arquivo *scripts.js* (parte 4): funções responsáveis pelo cadastro de clientes e animais de estimação

```
function postClient() {
   const name = document.getElementById('name').value
   const doc = document.getElementById('document').value
   axios.post(api + 'clients', {
      name: name, document: document
    })
    .then(function (response) {
      addClient(response.data)
      document.getElementsByTagName('form')[0].reset()
    })
    .catch(function (error) {
      console.log(error)
    })
}

function postPet(clientId) {
   const name = document.getElementById('petname').value
   const type = document.getElementById('type').value
   const breed = document.getElementById('breed').value
   const birth = document.getElementById('birth').value
   console.log(clientId)
   axios.post(api + 'pets', {
      name: name, type: type, breed: breed, birth: birth, ClientId: clientId })
    .then(function (response) {
      addPet(response.data)
      document.getElementsByTagName('form')[1].reset()
    })
    .catch(function (error) {
      console.log(error)
    })
}
```

A página resultante trata-se de uma *Single Page Application* (SPA) que permite listar, cadastrar e apagar clientes, assim como listar e cadastrar seus animais de estimação, utilizando o serviço Web desenvolvido para Node.js. A Figura 3.9 ilustra uma tela dessa aplicação. Sugere-se que a página seja estendida para incluir as demais funcionalidades disponíveis no serviço.

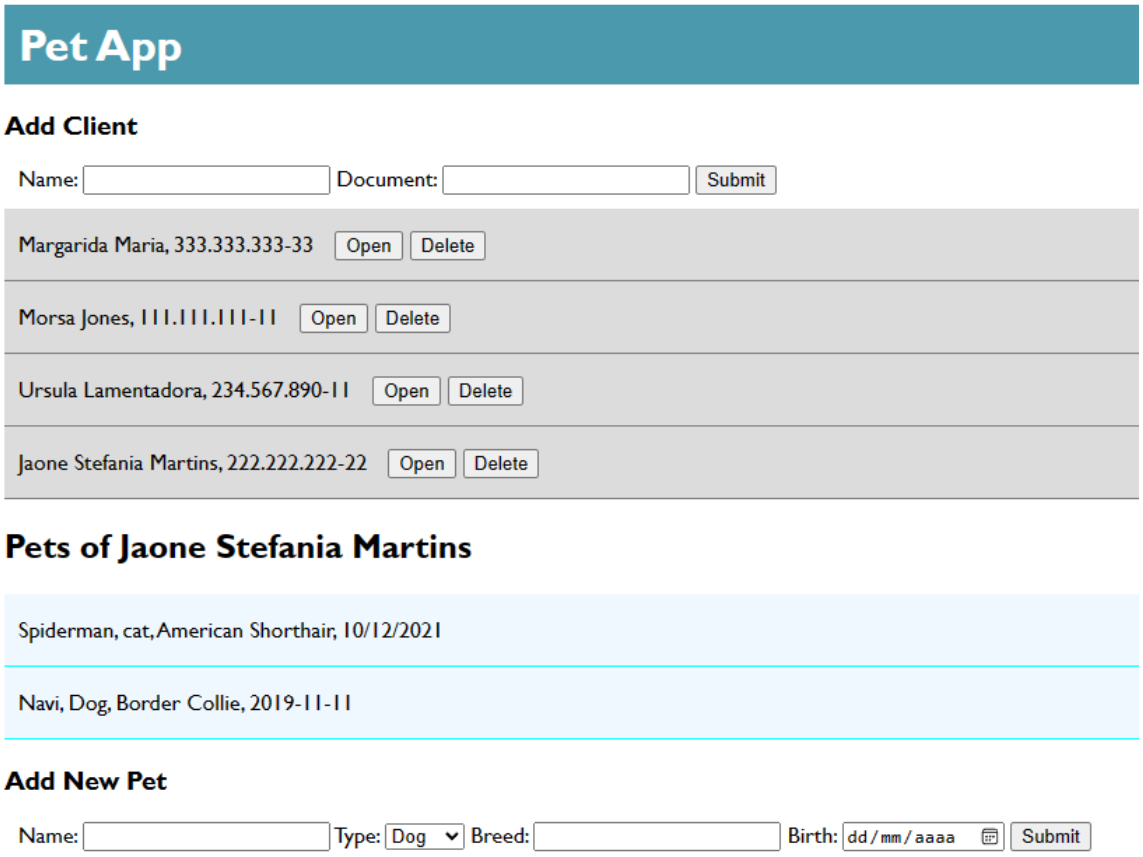

Figura 3.9. Tela da página Web resultante

3.6. Conclusão

A introdução e a evolução da linguagem JavaScript foi o que permitiu que páginas Web se tornassem verdadeiras aplicações, com funcionalidades interativas e um processo de comunicação pervasivo com servidores. Essa rápida progressão revolucionou a experiencia dos usuários e permitiu que aplicações Web substituíssem diversas aplicações *desktop*. Em consonância, arquiteturas de *software* são propostas para resolver problemas relacionados a atributos de qualidade, os quais são resultados de tendências de desenvolvimento tecnológico, de forma geral. Ao passo que tecnologias associadas evoluíram, o desenvolvimento Web passou a ser orientado por arquiteturas altamente componentizadas e fracamente acopladas. É nesse contexto que se encaixam as arquiteturas orientadas a serviço. Considerando o estilo arquitetural REST e o uso da linguagem JavaScript, este capítulo se propôs a apresentar conceitos fundamentas para o desenvolvimento Web *back-end* e *front-end*. O uso da mesma linguagem nas duas frentes é uma vantagem em termos operacionais, dado que facilita a atuação de desenvolvedores em diferentes âmbitos. Nesse sentido, é importante apontar que o uso do JS para a implementação de um servidor Web depende do uso da plataforma Node.js, enquanto o lado cliente permite o uso da linguagem sem nenhum componente adicional. Todavia, o desenvolvimento *front-end* pode ser beneficiado em vários aspectos quando um *framework* ou biblioteca (*e.g.*, Vue.js ou React) é incorporado ao projeto. A manutenibilidade é um desses principais aspectos, considerando que essas tecnologias aprimoram a componentização e o reuso de código.

3.7. Referências

Berners-Lee, T. and Fischetti, M. (1999) “Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor” Harper San Francisco.

Fielding, R. T. (2000) “Architectural styles and the design of network-based software architectures”. University of California, Irvine.

Garrett, J. J. et al. (2005) “Ajax: A new approach to web applications”.

Marinho, A. L. and Da Cruz, J. L. (org.). (2020) “Desenvolvimento de aplicações para Internet”. 2º ed. São Paulo: Pearson.

Mahmoud, Q. (2005) “Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI)”. Disponível em: <https://www.oracle.com/technical-resources/articles/javase/soa.html>. Acesso realizado em: 30 de ago. de 2023.

Mozilla Foundation. “MDN Web Docs.” Disponível em: <https://developer.mozilla.org/en-US>

Sommerville, I. (2019) “Engenharia de Software”. 10º ed. São Paulo: Pearson.