INTERFACE TECNOLÓGICA

IMPACTO QUE O *SEQUELIZE* TRAZ PARA O DESENVOLVIMENTO DE UMA *API* CONSTRUÍDA EM *NODE.JS* COM *EXPRESS.JS*

IMPACT THAT SEQUELIZE *BRINGS TO THE DEVELOPMENT OF AN* API *BUILT IN* NODE.JS *WITH* EXPRESS.JS

Nathan Barsoti – nathanbarsoti@hotmail.com
Faculdade de Tecnologia (Fatec) – Taquaritinga – SP – Brasil

Daniela Gibertoni – daniela.gibertoni@fatectq.edu.br
Faculdade de Tecnologia (Fatec) – Taquaritinga – SP – Brasil

DOI: 10.31510/infa.v17i2.964
Data de publicação: 18/12/2020

RESUMO

Como o mercado do desenvolvimento de softwares vem aumentando consideravelmente a sua demanda e com o objetivo de solucionar e otimizar problemas, garantindo a qualidade do produto e a satisfação do cliente, são exigidos prazos e tempo de entrega cada vez mais curtos. Por conta de tanta concorrência por melhorias e entregas ágeis, surgem mecanismos e facilitadores para aumentar a produtividade na codificação de software, facilitar a manutenibilidade e confidenciabilidade através dos *frameworks*. Este artigo aborda, por meio de pesquisas bibliográficas exploratória e experimental, o uso do *Sequelize*, um ORM (Object-Relational Mapper), no impacto do desenvolvimento de uma API (lado do servidor) em *Node.js* com *Express.js*, a partir do qual foi desenvolvido um sistema *web* para uma rede de dentistas situada na região de Itápolis/SP, com a finalidade de identificar as principais vantagens de usar o *Sequelize* neste estudo de caso. Como principais resultados obtidos, destacam-se a busca da manutenção e confiabilidade do código, e da otimização no processo de criação da API, por meio das funcionalidades do *Node.js*, *Express.js* e *Sequelize*.

Palavras-chave: Sequelize. Node.js. Express.js. Desenvolvimento de API. ORM.

ABSTRACT

As the software development Market has been considerably increasing its demand, aiming to solve and optimize problems, ensuring product quality and customer satisfaction, increasingly shorter deadlines and delivery times are required. Because of so much competition for improvements and fast deliveries, mechanisms and facilitators emerge to increase the productivity in codifying and to facilitate the maintainability and confidentiality through frameworks. This article addresses, through experimental and exploratory bibliographical research, how the use of Sequelize, an ORM (Object-Relational Mapper), impacts on the development of an API (server side) built in Node.js with Express.js, from which it was developed a web system for a range of dentists located in the Itápolis/SP region, with the

purpose of identifying the main vantages in using Sequelize in this case study. As main results, there are the search for maintaining and reliability of the code, and for optimization in the process of API creation, through Node.js, Express.js and Sequelize functionalities.

Keywords: Sequelize. Node.js. Express.js. API Development. ORM.

1 INTRODUÇÃO

O mercado de desenvolvimento de softwares vem evoluindo, e com essa evolução, o mercado vem se tornando um dos mais importantes da era moderna. Hoje, o software está presente em inúmeras atividades do dia a dia, desde as mais simples como controlar uma caminhada no parque ou processar informações básicas de compra e venda, até atividades mais complexas como controlar carros ou aviões (PRIKLADNICK, 2014).

Ainda para o mesmo autor Prikladnick (2014), o ambiente no qual as empresas de desenvolvimento de software fazem parte é altamente competitivo. Sendo assim, elas se esforçam para lançar suas novidades, produtos e serviços antes das empresas concorrentes. Com base nesse pensamento, durante o desenvolvimento de determinado produto ou serviço, as empresas buscam entregas contínuas e dentro de um prazo de tempo menor.

Para que essas entregas aconteçam de maneiras mais rápidas, se faz o uso de alguns facilitadores. Neste caso, um ORM (Object-Relational Mapper), que retrata uma das técnicas utilizadas para realizar a abstração da base de dados para a aplicação criada, ou seja, encarrega-se de criar a estrutura da base de dados e fazer a gestão dos objetos da aplicação, transformando os dados da representação interna em memória para o formato relacional, retirando a responsabilidade dos programadores de ter que gerir estas transformações (SATO, 2013). Dessa forma, o software passa a interagir com o ORM e não com a base de dados diretamente, aumentando a produtividade devido ao fato de não precisar escrever códigos SQL e poder fazer diretamente pelo código da aplicação.

O objetivo desse artigo é mostrar como o Sequelize, um ORM (Object-Relational Mapper) para Node.js, impacta no desenvolvimento de uma API Node com Express.js (framework para Node.js, utilizado para otimizar a construção de aplicações web e API's), mostrando as vantagens de utilizá-lo durante o desenvolvimento.

A metodologia utilizada neste artigo é a pesquisa bibliográfica exploratória e experimental, no qual foi realizada uma abordagem dos temas de uma maneira prática e dinâmica das metodologias e técnicas para o desenvolvimento de uma API Node com Express.js

utilizando o ORM Sequelize, capaz de elaborar um estudo de caso ao desenvolver um sistema para uma rede de dentistas.

Este artigo está estruturado em quatro sessões: a primeira sessão é a introdução que traz a contextualização proposta deste trabalho. Na segunda sessão é apresentada a fundamentação teórica que contém informações sobre o tema principal do artigo e temas complementares para o entendimento completo do assunto. A sessão três é destinada para a metodologia utilizada nesta pesquisa, sendo que na sessão quatro serão apresentadas as tecnologias e meios utilizados para o desenvolvimento do estudo de caso e, por fim, a sessão de número cinco traz uma conclusão acerca deste tema.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção dedica-se a apresentar os conceitos referentes ao tema central deste artigo, que é o impacto que utilizar um ORM traz no desenvolvimento de um sistema bem como apresentar as tecnologias que foram utilizadas no estudo de caso no qual foi desenvolvido uma API Node com Express.js.

2.1 Node.js

O Node.js foi desenvolvido em 2009, por Ryan Dahl e mais 14 colaboradores. Eles buscavam uma tecnologia inovadora com uma arquitetura *non-blocking thread* (arquitetura não bloqueante). O principal motivo para o desenvolvimento do Node.js foi dado por conta de que os sistemas webs desenvolvidos em Java, Python, PHP, .NET possuem uma característica em comum: suspender todo o processamento que está sendo realizado enquanto esperam por uma entrada/saída de dados no servidor, que vem a ser o modelo *blocking-thread* (arquitetura bloqueante) (PEREIRA, 2014).

Neto (2017) complementa que trabalhar de forma não bloqueante facilita a execução paralela e o aproveitamento de recursos. Ou seja, as arquiteturas não bloqueantes permitem que possam ser realizadas várias ações de maneira simultânea, já em relação as arquiteturas bloqueantes, cada ação será executada somente depois que a ação anterior for encerrada.

Segundo Nandaa (2018) o Node.js é um ambiente JavaScript, conhecido como *server-side* (que fica do lado do servidor), sendo orientado a eventos. O JavaScript é executado usando

o mecanismo V8 desenvolvido pelo Google para utilizar em seu navegador Google Chrome. A utilização desse mecanismo proporciona um ambiente de tempo de execução do lado do servidor. Com isso, o JavaScript é compilado e executado de maneira mais rápida.

Vale destacar que o Node.js possui uma estrutura orientada a eventos e uma forma de I/O (input/output) que faz com que ele seja leve e eficiente. Isso torna o Node.js "poderoso", porque essas características são essenciais para um melhor desempenho de intenso tráfego de rede e para aplicações em *real-time* (tempo real), que hoje, são considerados grandes empecilhos da web (MORAES, 2018).

Para Rauch (2012), um dos benefícios é que os desenvolvedores podem utilizar a mesma linguagem de programação tanto no lado do cliente, quanto no do servidor. Com base nisso e considerando o JavaScript muito dinâmico, o Node.js atingiu um sucesso instantâneo, muito por razão de sua simplicidade e produtividade aprimorada de programação e alto desempenho.

2.2 Express.js

Express.js é um web framework que atua como uma camada no topo do Node.js, facilitando e deixando o desenvolvimento de APIs em node mais prático. Além disso, com ele é mais fácil organizar as funcionalidades da aplicação usando *middleware* e roteamento, facilita a renderização de páginas HTML dinâmicas. Ele define um padrão de extensibilidade facilmente implementado e adiciona utilitários para os objetos HTTP do Node.js (HAHN, 2018).

Segundo Mardan (2014) o Express.js é um web framework no qual é baseado no *core* (núcleo) do módulo HTTP do Node.js e conecta os *middlwares* da aplicação. Dessa forma, os desenvolvedores são livres para escolher as bibliotecas que quiserem, gerando flexibilidade e alta capacidade de personalização.

Mardan (2014) ainda cita que o Express.js ajuda a resolver alguns problemas que os desenvolvedores têm com o Node.js, como gerenciar corpos de solicitações HTTP, gerenciar cookies, gerenciar sessões, organização de rotas, determinar cabeçalhos de respostas adequados.

O Express.js é considerado minimalista, porém, permite aos desenvolvedores uma liberdade para criar pacotes *middleware* específicos com o objetivo de resolver problemas que surgem durante o desenvolvimento de uma aplicação. Além disso, é um framework não

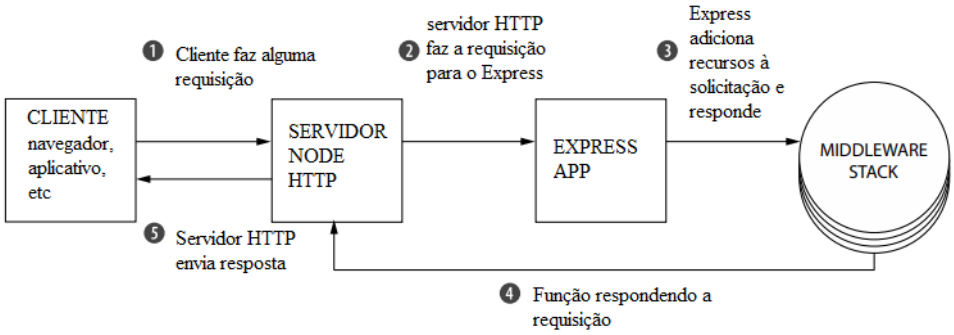
opinativo, ou seja, o desenvolvedor pode inserir qualquer *middleware* que prefira para o manuseio das solicitações e pode estruturar a aplicação em um arquivo ou em vários, usar qualquer estrutura de pastas dentro do diretório (MDN WEB DOCS, 2020).

Para Hahn (2018) o Express.js adiciona dois grandes recursos ao servidor HTTP Node.js:

- Adiciona uma série de conveniências úteis ao servidor HTTP do Node.js, abstraindo muito de sua complexidade. Por exemplo, enviar um único arquivo JPEG é complexo no Node.js bruto (especialmente pensando em desempenho), já com Express.js é possível fazer em apenas uma linha de código.
- Permite refatorar uma função monolítica para várias pequenas funções. Isso é mais sustentável para o código e mais modular.

A figura 1 mostra como seria o fluxo de uma solicitação por meio do Express.js.

Figura 1. Fluxo de uma solicitação por meio do Express.js



Fonte: Adaptado de Hahn (2018).

2.3 ORM (Object-Relational Mapping)

O ORM é uma técnica de mapeamento objeto relacional que permite fazer uma relação dos objetos com os dados que eles representam. Essa técnica tem sido muito utilizada e vem crescendo nos últimos anos. Este crescimento se dá principalmente pelo fato de muitos desenvolvedores não se sentirem à vontade em escrever código SQL e pela produtividade que esta técnica proporciona (SATO, 2013).

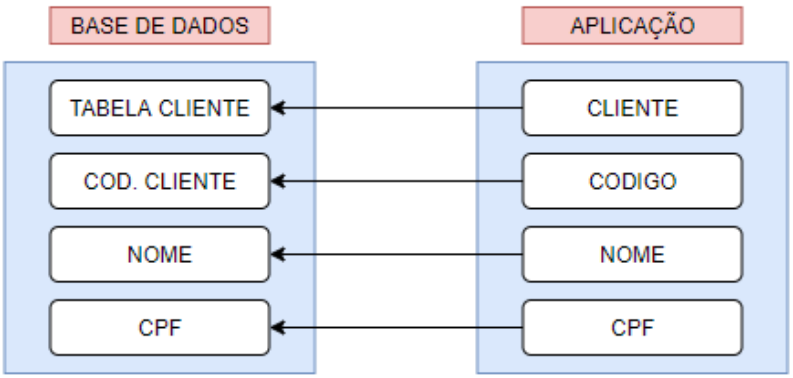
O mapeamento objeto relacional, para Bauer e King (2007), é a base teórica sobre a qual é possível mapear objetos em tabelas de banco de dados relacionais. Quando se mapeia os dados do modelo relacional, cria-se o efeito de que o banco de dados seja orientado a objetos, de modo que o desenvolvedor não precise se preocupar com a disposição dos dados nas tabelas. Assim, pode manter o foco na manipulação dos objetos e nos problemas de negócio da aplicação.

Segundo Keith e Schincariol (2009), a maior parte de uma API que persiste objetos em um banco de dados relacional acaba fazendo o uso de um componente de mapeamento objeto-relacional (ORM). Os autores também citam que o ORM inclui desde como o objeto é mapeado para as colunas do banco de dados até como fazer consultas entre os objetos.

Os desenvolvedores preferem lidar com dados persistentes ao programar em vez de lidar com instruções SQL para recuperar dados na base de dados. Além disso, estruturas ORM reduzem a lacuna de incompatibilidade entre a OOP (programação orientada a objetos) e RDBs (bancos de dados relacionais) ao converter dados de RDBs em objetos de programação apropriados e vice-versa. O ORM desempenha o papel de mediador para mapeamento entre objetos e banco de dados (ALGHAMDI; OWDA; CROCKETT, 2017).

O ORM funciona através do mapeamento das características da base de dados para os objetos da aplicação. O primeiro conceito chave é traçar um paralelo entre Classe x Tabela e Propriedade x Coluna. Ele também permite informar em qual tabela cada classe será persistida e em que coluna do SGBD (Sistema de gerenciamento de banco de dados) cada propriedade ficará armazenada (SATO, 2013).

Figura 2. Ilustração de uma classe para uma tabela da base de dados.



Fonte: Adaptado de Sato (2013).

Na figura 2, é ilustrado uma tabela na base de dados uma classe dentro da aplicação, com as setas indicando o mapeamento da classe para a tabela. Este mapeamento considera coisas como o nome da coluna, o tipo da coluna, tamanho e precisão.

2.4 Sequelize

Sequelize é um ORM baseado em *promises* (objeto usado para processamento assíncrono no qual um *promise* representa um valor que pode estar disponível de modo instantâneo, no futuro ou nunca). Ele suporta os dialetos PostgreSQL, MySQL, MariaDB, SQLite e MSSQL, oferecendo suporte a transações sólidas, relações, replicações de leituras (SEQUELIZE ORG).

Além disso, o próprio SEQUELIZE ORG (2018) cita que o framework permite criar, buscar, alterar e remover dados da base da dados utilizando métodos JavaScript. Permite, também, a modificação da estrutura das tabelas e como consequência os desenvolvedores têm maior facilidade na criação, população e migração de banco de dados.

3 PROCEDIMENTOS METODOLÓGICOS

Para o desenvolvimento deste artigo, foi realizada a pesquisa bibliográfica exploratória experimental, composta por dados obtidos através de pesquisas em livros, dissertações, artigos e teses que abordam os temas relacionados ao assunto em questão (GIL, 2008). Além disso, com todo conhecimento teórico adquirido durante a pesquisa, foi possível realizar um estudo de caso ao desenvolver um sistema, no qual a API foi construída utilizando as técnicas de mapeamento objeto relacional, para que a aplicação pudesse ser desenvolvida de maneira mais rápida e, com isso, gerando maior produtividade durante o desenvolvimento.

4 RESULTADOS E DISCUSSÃO

O projeto desenvolvido foi um sistema web, nomeado *Dentist Control*, para uma rede de dentistas situada na região de Itápolis, no interior do estado de São Paulo, visando facilitar o agendamento de consultas para os pacientes da clínica. Trate-se de uma rede na qual

disponibiliza várias opções de atendimento, desde procedimentos cirúrgicos até procedimentos estéticos.

O problema enfrentado pela organização era o fato de todas as consultas serem agendadas em caderno, ou seja, todo o processo era realizado de forma manual. A organização até chegou a ter um sistema para agendamento de consultas, porém esse sistema era muito complexo e tinha um grande custo de manutenção.

Como aluno do curso de Análise e Desenvolvimento de Sistemas da Fatec Taquaritinga e paciente da rede de dentistas, foi realizada uma proposta sistêmica para o problema enfrentado pela organização. O planejamento foi o desenvolvimento de outro sistema com menor complexibilidade e, como consequência, o custo de manutenção não seria tão alto para que não ocorresse o mesmo problema enfrentado anteriormente. Foi de responsabilidade do autor fazer o levantamento de requisitos, documentar os mesmos e garantir que todo o sistema esteja funcionando de acordo com o planejado.

Para desenvolver esse sistema, que deve entregar uma boa experiência para o usuário e reduzindo o tempo de desenvolvimento, as tecnologias escolhidas para o desenvolvimento foram *Angular* para o *front-end* e *Node.js*, utilizando o *Express.js* e o framework *Sequelize* para o *back-end*.

A escolha dessas tecnologias se deu por conta da baixa curva de aprendizado oferecida pelo conjunto (pois ambos compartilham da mesma linguagem). Embora sejam diferentes bases de desenvolvimento, *Angular* é *client-side* (lado do cliente) e *Node.js* é *server-side* (lado do servidor), além de um conhecimento mínimo do autor por essas tecnologias. Vale ressaltar também que essas tecnologias possuem grande desempenho e produtividade durante o desenvolvimento.

A escolha dessa tecnologia para o desenvolvimento do *back-end* se deu por conta de o Node.js ter uma boa performance, otimizando a taxa de transferência e a escalabilidade em aplicações web. O Express.js foi escolhido por conta de fazer um gerenciamento de diferentes verbos HTTP em diferentes URLs, definir as configurações da aplicação, como a porta a ser usada para conexão e a localização dos modelos que são usados para renderizar respostas. Já a escolha do Sequelize se dá por conta de ser um facilitador entre a comunicação da API (*back-end* da aplicação) com a base de dados, fazendo o mapeamento de dados relacionais (tabelas, colunas e linhas) para objetos JavaScript.

4.1 Resultados obtidos

Para ser possível o desenvolvimento de uma API em Node.js é necessário ter um conhecimento mínimo sobre NPM (*Node Package Manager*), um gerenciador de pacotes para a linguagem de programação JavaScript, que possui um repositório online para a publicação de projetos de código aberto. É necessário, também, ter conhecimento em JavaScript, linguagem na qual a API foi desenvolvida.

Com o uso do Sequelize foi possível a criação do banco de dados diretamente pelo código, sem precisar usar um SGBD para isso. Tudo graças as *migrations* (uma espécie de versionamento do banco de dados, como se fosse o GIT da aplicação) que é possível gerar pelo framework. A Figura 3 mostra um exemplo de como seria o código de uma *migration*.

Figura 3. Exemplo de é o código de uma *migration* com Sequelize.

Fonte: Elaborado pelo autor (2020).

Na figura 3, nota-se que há dois métodos, ***up*** e ***down***. O método ***up*** é responsável por criar os atributos juntamente com suas características (*nullable*, *primaryKey* etc.). O método ***down*** é responsável por fazer um *rollback*, ou seja, se o desenvolvedor desejar desfazer a criação dessa tabela, deve ser executado esse método. Com o avanço do sistema e a necessidade

de fazer algumas mudanças na base de dados, foi possível fazer tudo gerando *migrations*, o que não atrasou o desenvolvimento, pelo contrário, aumentou a produtividade por não haver a necessidade de ter que usar códigos SQL e, também, não precisar de um SGBD para isso. Além disso, automaticamente, o Sequelize cria uma tabela chamada **SequelizeMeta**, na qual guarda as informações sobre as *migrations* que já foram executadas, para que na próxima execução seja executado apenas o que houver de novo e não todas as *migrations* novamente. A Figura 4 apresenta como seria a *Model* através do Sequelize.

Figura 4. Exemplo de como é a *Model* (representação da tabela) no Sequelize.

```
const { Model, DataTypes } = require('sequelize');

class User extends Model {
    static init(sequelize) {
        super.init({
            id: {
                type: DataTypes.UUID,
                primaryKey: true
            },
            email: DataTypes.STRING,
            password: DataTypes.STRING,
        }, {
            sequelize
        });
    }
}

module.exports = User;
```

Fonte: Elaborado pelo autor (2020).

A figura 4 mostra como é possível construir um *Model* com o Sequelize. Sendo assim, os tipos dos atributos devem ser correspondentes ao da base de dados. Nota-se que quando foi criada a tabela, foram colocados os atributos *CreatedAt* e *UpdatedAt*. No *Model* não é necessário colocá-los pois o Sequelize preenche esses campos automaticamente quando se cria ou altera um registro. Então, é usado o método ***init*** para configurar o *model*. Se a tabela possuir alguma chave estrangeira, é necessário o uso do método ***associate***. A Figura 5 mostra um exemplo de como seria o código do método ***associate***.

Figura 5. Exemplo de como é o método associate.

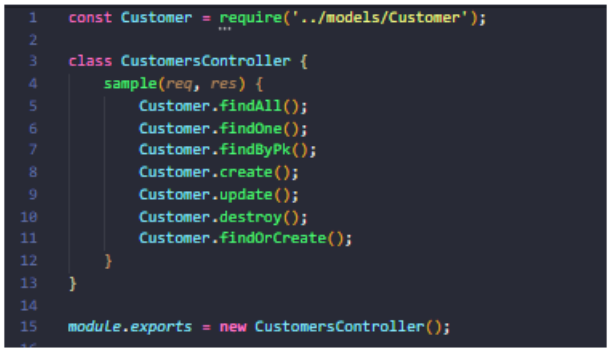
```
22      static associate(models) {
23          this.hasOne(models.Scheduling);
24          this.hasMany(models.Phone, {
25              foreignKey: 'customerId',
26              as: 'phones'
27          });
28          this.hasOne(models.Address, {
29              foreignKey: 'customerId',
30              as: 'address'
31          });
32      }
33  }
```

Fonte: Elaborado pelo autor (2020).

A figura 5 mostra como é o método **associate**, no qual são feitos os relacionamentos com outras tabelas. Nesse caso, o *model* em questão é da tabela *Customers* e mostra que ela tem uma chave estrangeira em *Scheduling*, *Phone* e *Address*.

Quando se trata de fazer requisições da aplicação para o banco de dados, o Sequelize fornece diversos métodos para isso, desde o CRUD (*create*, *read*, *update*, *delete*) até outros métodos como *findOrCreate*. A Figura 6 apresenta os métodos que o Sequelize fornece.

Figura 6. Exemplo dos métodos que o Sequelize fornece.



```
1   const Customer = require('../models/Customer');
2
3   class CustomersController {
4       sample(req, res) {
5           Customer.findAll();
6           Customer.findOne();
7           Customer.findByPk();
8           Customer.create();
9           Customer.update();
10          Customer.destroy();
11          Customer.findOrCreate();
12      }
13  }
14
15  module.exports = new CustomersController();
```

Fonte: Elaborado pelo autor (2020).

A figura 6 mostra como fazer as requisições do Sequelize para a base de dados, basta importar a *model* que foi criada e, a partir disso, o próprio Sequelize já vai mostrar as opções de métodos para se utilizar. Além dos que estão no exemplo, existem muitos outros métodos. O Sequelize também tem um próprio método para paginação, assim, o desenvolvedor não precisa fazer via código.

Seguindo esses princípios do Sequelize, houve um grande aumento de desempenho e produtividade. Como consequência, a aplicação foi desenvolvida em um menor tempo. Além

do mais, para efetuar manutenções na API, tanto como melhorias, não será um processo árduo para o desenvolvedor, por conta de o Sequelize ser um facilitador e possuir, ainda, uma documentação explicando o que cada método faz.

5 CONSIDERAÇÕES FINAIS

Com o desenvolvimento do sistema foi possível atingir os objetivos propostos: sistema simples, sem muita complexidade e com baixo custo de manutenção, facilitando, assim, o cotidiano dos funcionários, uma vez que o sistema antigo parou de ser usado por ser muito complexo e ter um alto custo de manutenção. Vale ressaltar que o sistema atendeu todas as necessidades da organização, possibilitando um gerenciamento adequado das consultas.

Um dos maiores desafios no desenvolvimento de sistemas é buscar otimizações durante o processo. Levando isso em consideração, ter um facilitador ajuda de diversas formas. Por conta disso, quando se tratar de uma API construída em Node.js, pode ser indicado o uso do Sequelize, para aqueles que já possuem um breve conhecimento de *frameworks* similares ou tecnologias *server-side* (lado do servidor). Lembra-se que quando entendido o seu funcionamento e o desenvolvedor passar a ser detentor de conhecimento de forma parcial, a sua utilização no dia a dia se torna algo muito mais prático e eficiente, capaz de reduzir o tempo de desenvolvimento de uma aplicação.

Desta forma, o uso do *framework* Sequelize se tornou um grande aliado durante o desenvolvimento do sistema, pois a prática e a experiência trouxeram uma grande facilidade enquanto era desenvolvida a aplicação. Com base nisso, resultaram-se entregas mais eficazes, com fácil manutenção de código.

O Sequelize, então, é um grande facilitador para o desenvolvimento de aplicações.

Com o conhecimento adquirido de funcionamento do Sequelize citado neste estudo, pode-se aplicar em projetos, para tornar o trabalho mais prático e eficiente. Além disso, ele permite reduzir consideravelmente o tempo de desenvolvimento de uma aplicação. É, portanto, uma excelente escolha para todos os desenvolvedores, independentemente do nível de conhecimento.

REFERÊNCIAS

ALGHAMDI, A; OWDA, M; CROCKETT, K. **Natural Language Interface to Relational Database (NLI-RDB).** Springer Internation Publishing. 2017.

BAUER, C; KING, G. **Java Persistence com Hibernate.** 1. ed. Ciência Moderna Ltda. 2007.

GIL, A. C. **Métodos e Técnicas de Pesquisa Social.** 6. ed. São Paulo: Editora Atlas S.A., 2008.

HAHN, E. M. **Express in action:** Writing, building, and testing Node.js applications. 1. ed. Manning, 2016.

KEITH, M; SCHINCARIOL, M. **Pro JPA 2.** 2. ed. Apress, 2009.

MARDAN, A. **Express.js Guide:** The compreensive book on Express.js. 1. 2014.

MDN WEB DOCS. **Introdução Express/Node.** 2020. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/Express_Nodejs/Introdu%C3%A7%C3%A3o>. Acesso em: 15 out. 2020.

MORAES, W. B. **Construindo aplicações com NodeJS.** 2. ed. São Paulo: Novatec Editora, 2018.

NANDAA, A. **Beginning API Development with Node.js.** 1. ed. United Kingdom: Packt Pubishing, 2018.

NETO, W. **Construindo APIs testáveis com Node.js.** 2017. Disponível em: <https://www.academia.edu/32568937/Construindo_apis_testaveis_com_nodejs>. Acesso em: 13 out. 2020.

PEREIRA, C. R. **Node.js:** Aplicações web real-time com Node.js. 1. ed. Editora Casa do Código, 2014.

PRIKLADNICKI, R; WILLI, R; MILANO, F. **Métodos Ágeis para Desenvolvimento de Software.** Porto Alegre; Bookman, 2014.

RAUCH, G. **Smashing Node.js: JavaScript Everywhere.** 2. ed. Wiley, 2017.

SATO, P. M. **ORM - Object Relational Mapping. DEVMEDIA**, 2013. Disponível em: <https://www.devmedia.com.br/orm-object-relational-mapping-revista-easy-net-magazine-28/27158>. Acesso em: 12 out. 2020.

SEQUELIZE ORG. **SEQUELIZE.** Disponível em: <https://sequelize.org/v3/>. Acesso em: 16 out. 2020.