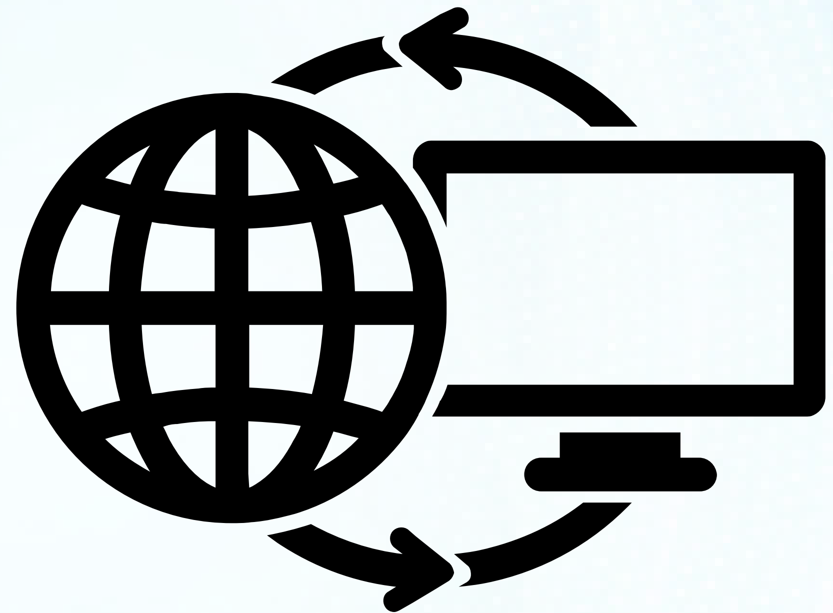
Web Services aplicados aos Jogos

Curso Técnico em
Programação de Jogos Digitais

Ernesto Henrique Radis Steinmetz



Presidência da República Federativa do Brasil
Ministro da Educação
Secretaria de Educação Profissional e Tecnológica

Este material foi produzido pela Direção de Educação a Distância do Instituto Federal de Brasília – IFB

Reitora
Luciana Massukado

Pró-Reitora de Ensino - PREN
Yvonete Bazbuz da Silva Santos

Diretora de Educação a Distância - DEaD
Jennifer de Carvalho Medeiros

Apoio Administrativo e Tecnológico
Cláudia Sabino Fernandes
Joscélia Moreira de Azevedo
Noeme César Gonçalves

Coordenador Adjunto Ensino
Elias Vieira de Oliveira

Coordenadora Pedagógica
Eliziane Rodrigues de Queiroz

Orientador de Ensino e Aprendizagem
Antonio Gomes da Costa Neto
Norma Lúcia Néris de Queiroz

Coordenadora de Curso
Alessandro Borges Lima

Equipe de Elaboração

Coordenadora de Produção de Material Didático
Joscélia Moreira de Azevedo

Conteudista
Ernesto Henrique Radis Steinmetz

Desenhista Instrucional
Maria do Socorro de Lima

Revisora de conteúdo
Karina Mendes Nunes Viana

Ilustrador e Produtor de vídeos animados
Márlon Cavalcanti Lima

Diagramador
Fábio Lucas Vieira

Gestora de Ambiente Virtual de Aprendizagem
Maria Jesus Rezende

Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-NãoComercial-CompartilhaIgual 4.0 Internacional

SUMÁRIO

APRESENTAÇÃO 6

Unidade 1 – Introdução a Web Services..7

1.1 Objetivos específicos da Unidade 1... 7

1.2 O que é Web Services? 8

1.3 E qual a finalidade dos Web Services? 8

1.4 Você sabe como funcionam os Web Services?..10

1.5 Web Services e desenvolvimento em camadas 13

1.6 Os protocolos SOAP e REST...14

1.7 Benefícios dos Web Services...18

Atividades de Revisão 21

Referências 22

Unidade 2 – Instalação e configuração de um WebServices...23

2.1 Objetivos específicos da Unidade 2.. 23

2.2 O que é o Node.js 24

2.3 Como funciona o Node.js 26

2.4 O que podemos ver de diferente no Node.js em relação a programação?........................27

2.5 Instalando e executando o Node.js 28

2.6 Instalando e criando módulos no node.js...33

2.7 Criando uma aplicação Web com Node.js..38

Atividades de Revisão 45

Referências 46

Unidade 3 – Desenvolvimento uma aplicação básica de Back-end com Banco de Dados 47
3.1 Preparando servidor HTTP..48
3.2 Conectando o Banco de Dados com Sequelize 57
Atividades de Revisão 66

Unidade 4 – Desenvolvendo um jogo com WebServices e Banco de Dados 67
4.1 Criando as rotas do Controller.. 77
4.2 Iniciando o servidor com as rotas 81

Atividades de Revisão 86
Referências...87

APRESENTAÇÃO

Olá, estudante!

Bem-vindo(a) a mais uma disciplina do nosso curso! Estou certo de que este estudo contribuirá bastante para a sua formação profissional.

Esta disciplina está estruturada em 4 unidades temáticas com seções específicas.

Na Unidade 1, iremos compreender os conceitos de Web Services e seu funcionamento na integração de aplicações diferentes. Também iremos revisar os conceitos de aplicações em camadas, definindo Front-End e Back-End.

Na Unidade 2, iremos focar em instalação e configuração do ambiente de WebServices para o desenvolvimento de aplicações de Back-End.

Na Unidade 3, iremos desenvolver uma aplicação básica de Back-End baseada em Serviços Web em interação com Banco de Dados.

Na Unidade 4, iremos desenvolver um jogo simples, utilizando os conceitos de uma aplicação básica de Back-End apoiada em WebServices.

Unidade 1 Introdução a *Web Services*

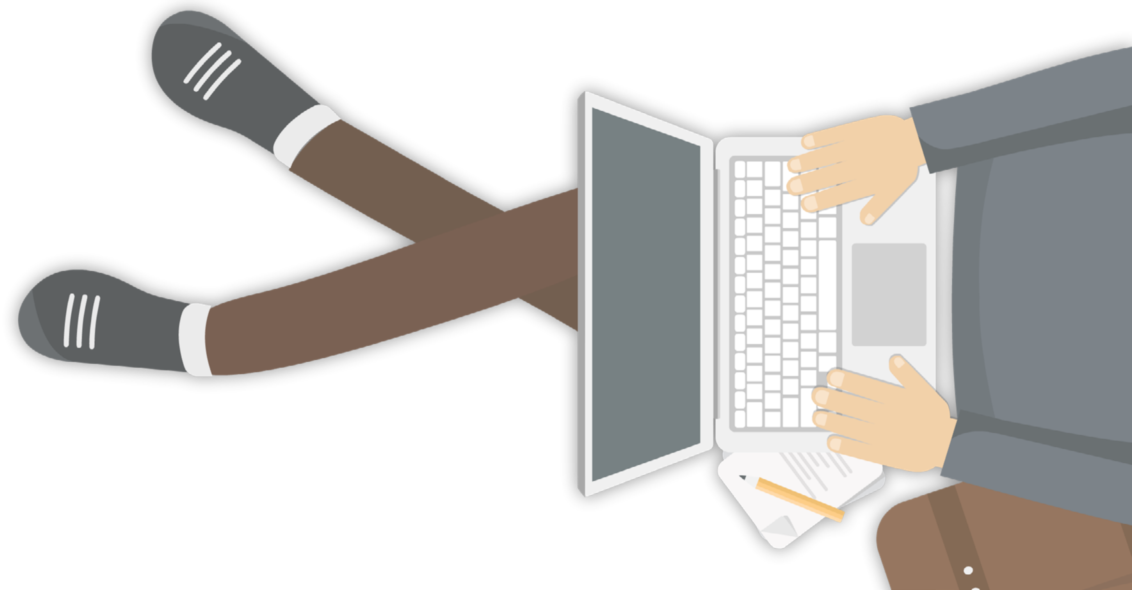
Nesta unidade iremos conhecer os conceitos básicos de Web Services, suas vantagens e aplicabilidade. Além disso, veremos um pouco da história da sua criação e quais as organizações internacionais responsáveis pela padronização dos Web Services e as tecnologias envolvidas na sua implementação. Boa leitura a todos(as).

1.1 Objetivos específicos da Unidade 1

Ao final desta unidade, espero que você seja capaz de:

conceituar Web Services;

compreender as finalidades e o funcionamento dos Web Services;



1.2 O que é Web Services?

Quem estabelece os conceitos de definição e utilização de Web Services, em âmbito mundial, é o consórcio W3C (World Wide Web Consortium). É o mesmo consórcio responsável por definir e regulamentar as normas da Web. Assim, o W3C (2011) define Web Service como: sistema de software projetado para suportar a interoperabilidade entre máquinas sobre rede. A W3C e a OASIS (Organization for the Advancement of Structured Information Standards) são as instituições responsáveis pela padronização dos WebServices. Empresas como IBM e Microsoft, duas das maiores do setor de tecnologia, apoiam o desenvolvimento deste padrão.

Quando falamos de Web Services aplicados ao desenvolvimento de jogos, as vantagens da sua utilização, nas diversas aplicações, é perfeitamente percebida em ambientes com multiusuários, ou seja, multijogadores.

1.3 E qual a finalidade dos Web Services?

Os Web Services buscam prover uma solução para integração de sistemas, visando à comunicação entre aplicações diferentes. Essa tecnologia permite que novas aplicações possam interagir com aquelas que já existem, proporcionando compatibilidade entre sistemas desenvolvidos em plataformas diferentes.

Teoricamente, Web Services são um conjunto de métodos chamados e invocados por outros programas, utilizando tecnologias Web. Assim, os Web Services são componentes que permitem às aplicações enviar e receber dados utilizando estes métodos. Cada aplicação pode ter a sua própria linguagem de programação, que é traduzida para uma linguagem comum, ou seja, um formato intermediário que permite a comunicação entre elas. Alguns exemplos destas linguagens são: XML, Json, CSV, etc.

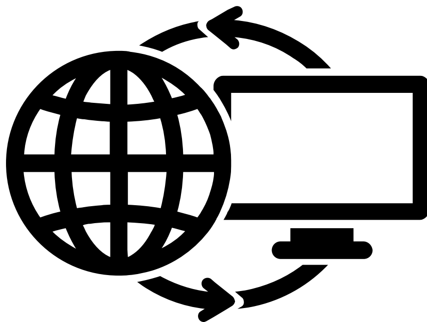
Atualmente existem jogos para plataformas Android utilizando Web services.

Bem, agora que você já sabe o conceito e a finalidade dos Web Services, iremos entender um pouco sobre as suas funções. Vamos lá?

A principal função dos Web Services é transferir dados por meio de protocolos de comunicação para diferentes plataformas, independentemente das linguagens de programação utilizadas nessas plataformas. Bem como, funcionar com qualquer sistema operacional, plataforma de hardware ou linguagem de programação de suporte Web.

O objetivo dos Web Services é a comunicação de aplicações através da Internet. Esta comunicação é realizada com intuito de facilitar a EAI (Enterprise Application Integration) que significa a integração entre diversas aplicações, ou seja, interoperabilidade entre a informação que circula numa organização, nas diferentes aplicações. Quando aplicamos este conceito ao desenvolvimento de jogos, temos a vantagem de disponibilizar vários serviços de aplicações de Back-End a vários jogadores/usuários independentemente da tecnologia utilizada nas aplicações de Front-End.

Compreender o conceito e os propósitos dos Web Services é muito importante. Porém, como profissional da área de jogos, é indispensável que você saiba como funcionam os Web Services. Então, fique ligado no que vamos descrever no próximo tópico.



1.4 Você sabe como funcionam os Web Services?

A atual dinâmica da Web exige a interação entre as diversas aplicações, organizando e trocando dados de diferentes formas. No entanto, percebemos que nem sempre a comunicação entre as aplicações acontece de forma natural e tranquila. Assim, é cada vez mais comum a necessidade de trocar dados entre diferentes sistemas.

Neste contexto, os Web Services surgem como uma solução prática e de baixo custo para sanar a incompatibilidade de sistemas e garantir a comunicação entre as diversas aplicações. Os Web Services permitem ligar diferentes aplicações que integram um sistema, independentemente do tipo de plataforma ou linguagens de programação. Para isso, são utilizados alguns padrões tecnológicos e de comunicação definidos e aceitos mundialmente.

No ano de 2000, a W3C aceitou a submissão do Simple Object Access Protocol (SOAP). O SOAP é um formato de mensagem baseado em XML que estabelece uma estrutura básica de transmissão e comunicação entre aplicações e serviços diferentes, via protocolo de rede HTTP. Sendo uma tecnologia não amarrada a fornecedor, o SOAP disponibilizou uma alternativa atrativa em relação aos protocolos proprietários tradicionais, tais como CORBA e DCOM.

No decorrer do ano seguinte, o W3C publicou a especificação WSDL. Uma nova implementação do XML, este padrão forneceu uma linguagem para descrever a interface dos Web Services. Posteriormente suplementada pela especificação UDDI (Universal Description, Discovery and Integration), que proporcionou um mecanismo padrão para a descoberta dinâmica de descrições de serviço, a primeira geração da plataforma de Web Services foi estabelecida. A Figura 1, abaixo, ilustra em alto nível o relacionamento entre estes padrões (DEVMEDIA, 2019).

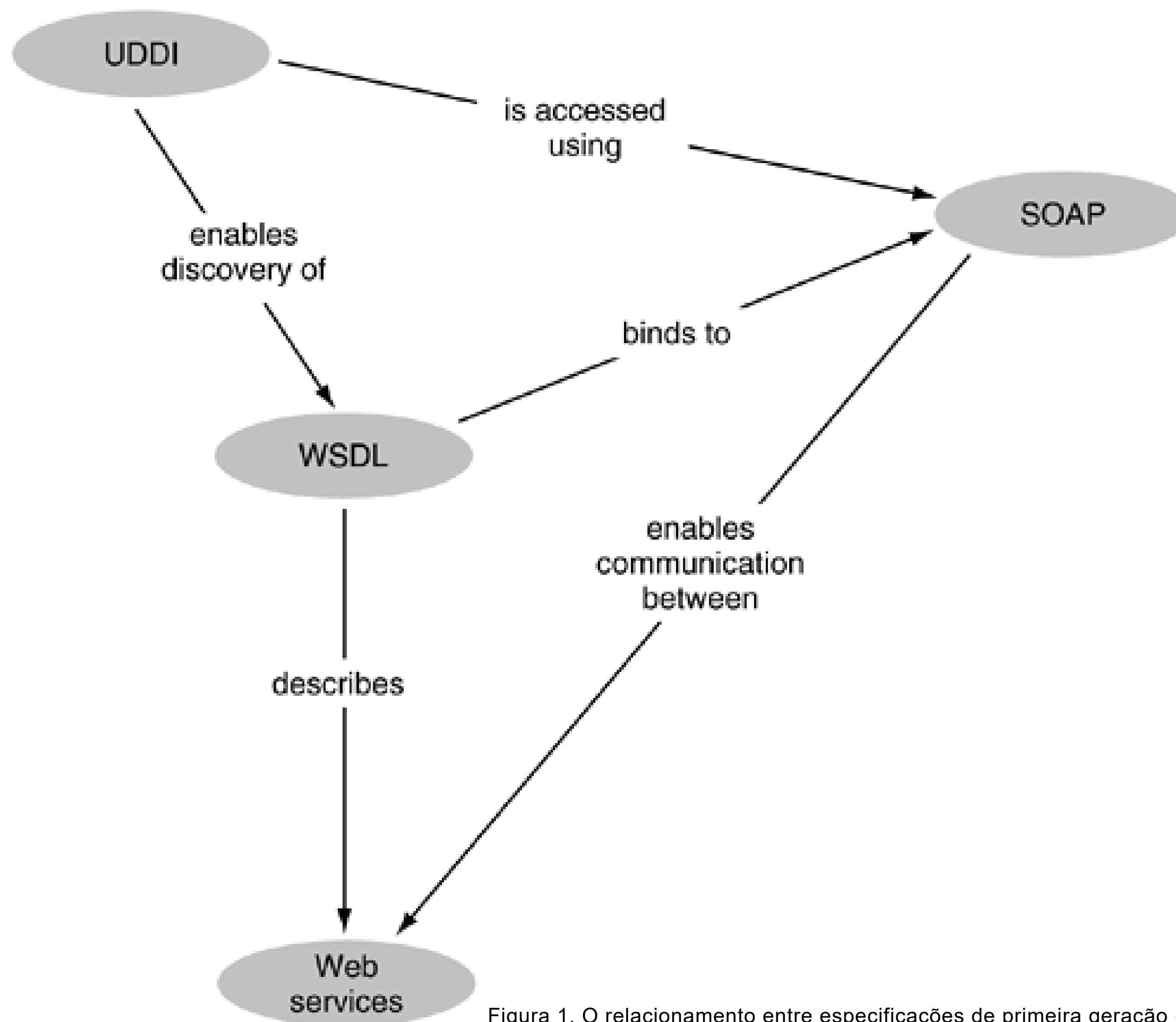


Figura 1. O relacionamento entre especificações de primeira geração (DEVMEDIA, 2019).

Para entender o processo ilustrado, seguem as traduções das expressões empregadas:

Is accessed using: é acessado utilizando.

Enables discovery of: permite a descoberta de.

Describes: descreve.

Enables communication between: permite a comunicação entre.

Binds to: ligação para.

Com essas especificações, os Web Services passam a ser crescentemente adotados por vendedores e fabricantes de software. O suporte amplo da indústria de software impulsionou à popularidade e importância desta plataforma e de princípios de projeto orientados a serviço. Assim, surgiu a segunda geração de especificação de Web Services. Os Web Services com arquitetura orientada a serviços (SOA - Service-oriented architecture).

O conceito de serviços em uma aplicação existe já há algum tempo. Serviços, assim como componentes, são considerados blocos de construção independentes, os quais coletivamente representam um ambiente de aplicação. No entanto, diferentemente de componentes tradicionais, serviços têm algumas características únicas que lhes permitem compor partes de uma arquitetura orientada a serviços. Uma destas características é a completa autonomia em relação a outros serviços. Isto significa que cada serviço é responsável por seu próprio domínio, o que tipicamente corresponde a limitar seu alcance para uma função de negócio específica.

Em Web Services com arquitetura orientada a serviços, podemos ter várias aplicações, trocando dados com vários serviços e, além disso, em determinadas situações, os próprios serviços podem se comunicar entre si, potencializando a comunicação entre aplicações diferentes.

1.5 Web Services e desenvolvimento em camadas

Quando olhamos para o desenvolvimento de software em camadas, temos Front-End e Back-End. Front-End é a interface amigável que permite ao usuário interagir com o sistema, observando as características do ambiente do cliente segundo o hardware e o software. O código do lado servidor é definido como Back-End. Nele, temos as regras de negócio da aplicação, que definem os acessos ao banco de dados, gerando os resultados dos fluxos de processamento.

Neste contexto, podemos dizer que Web Services correspondem ao Back-End da aplicação? Sim, mas Web Services vão além disso, fazem o papel de Back-End, permitindo a comunicação e a integração entre várias aplicações de Front-End diferentes. Web Services permitem que os recursos da aplicação do software estejam disponíveis sobre a rede de forma normalizada. Outras tecnologias fazem a mesma coisa; por exemplo, os browsers da Internet acessam as páginas Web disponíveis, usando por norma as tecnologias da Internet, HTTP e HTML. No entanto, estas tecnologias não são bem-sucedidas na comunicação e integração de aplicações. O grande diferencial da tecnologia Web Services sobre as demais aplicações, é que ela possibilita que diferentes aplicações se comuniquem entre si e utilizem recursos diferentes.

Assim, Web Services possuem uma determinada quantidade de operações pré-definidas. Considerando essas operações disponíveis nos Web Services, a aplicação solicita uma dessas operações. Os Web Services recebem a solicitação e efetuam o processamento, enviando os dados de resposta para a aplicação que requereu a operação. A aplicação recebe os dados e faz a sua interpretação, convertendo-os para a sua linguagem própria e apresentando o resultado do processamento ao usuário no Front-End. Em outras palavras, os Web Services fazem com que os seus recursos estejam disponíveis para qualquer aplicação cliente, dentro dos parâmetros definidos para cada operação em conforme.

Quando observamos a aplicabilidade destes conceitos no desenvolvimento de jogos, percebemos que algumas funcionalidades desenvolvidas em Back-End podem ser compartilhadas entre vários usuários e/ou entre vários jogos diferentes. A exemplo disso,

podemos ter uma funcionalidade de movimentação de um personagem que pode ser desenvolvida para um determinado jogo e, ao ser incluída em Web Services, pode ser utilizada em outros jogos com a mesma dinâmica. Outro exemplo desta interoperabilidade é o registro de scores e records dos jogadores, que pode ser utilizado em vários jogos diferentes.

1.6 Os protocolos SOAP e REST

Para que a comunicação entre aplicações ocorra, é necessária uma linguagem intermediária que garanta a comunicação entre a linguagem dos Web Services e o sistema que faz o pedido aos Web Services. Para isso, são utilizados protocolos de comunicação que padronizam a troca de informações, como o SOAP (Simple Object Access Protocol) e o REST (Representational State Transfer).

Os Web Services são identificados por um URI (Uniform Resource Identifier), descritos e definidos usando XML (Extensible Markup Language). A utilização de tecnologias standards como o XML e o HTTP (Hypertext Transfer Protocol) tornam os Web Services atrativos e de fácil utilização. Os WebServices são utilizados para disponibilizar serviços interativos na Web, podendo ser acessados por outras aplicações, usando um protocolo SOAP ou REST.

O protocolo SOAP utiliza XML para enviar mensagens e, geralmente, serve-se do protocolo HTTP para transportar os dados. Associado ao protocolo SOAP, está o documento WSDL (Web Services Definition Language) que descreve a localização dos Web Services e as operações de que dispõe. Além disso, fornece a informação necessária para que a comunicação entre sistemas seja possível.

Saiba Mais!

Para entender melhor os conceitos de XML e WSDL acesse os seguintes links:

XML - https://pt.wikipedia.org/wiki/Web_service#XML

WSDL - https://pt.wikipedia.org/wiki/Web_service#WSDL

O REST é um protocolo de comunicação mais recente, que surgiu com o objetivo de simplificar o acesso aos Web Services. Este baseia-se no protocolo HTTP e permite utilizar vários formatos para representação de dados, como JSON , XML, RSS, entre outros.

Saiba Mais!

REST (Representational State Transfer) significa, em português, Transferência de Estado Representacional. É um estilo de arquitetura que define um conjunto de restrições e propriedades baseados no protocolo HTTP. Web Services que seguem o padrão arquitetural REST, ou Web Services RESTful, fornecem interoperabilidade entre sistemas de computadores na Internet. Web Services compatíveis com REST permitem que os sistemas solicitantes acessem e manipulem representações textuais de recursos da Web, usando um conjunto uniforme e pré-definido de operações sem estado. Outros tipos de Web Services, como Web Services SOAP, expõem seus próprios conjuntos arbitrários de operações

SOAP (Simple Object Access Protocol) significa, em português, Protocolo Simples de Acesso a Objetos. Ele é um protocolo que permite a troca de informações estruturadas em uma plataforma descentralizada e distribuída. O SOAP utiliza a Linguagem de Marcação Extensível (XML) para definição do formato das mensagens e, normalmente, baseia-se em outros protocolos da camada de aplicação. Dentre eles, podemos destacar o Protocolo de transferência de hipertexto (HTTP).

Fonte do Glossário: Wikipedia

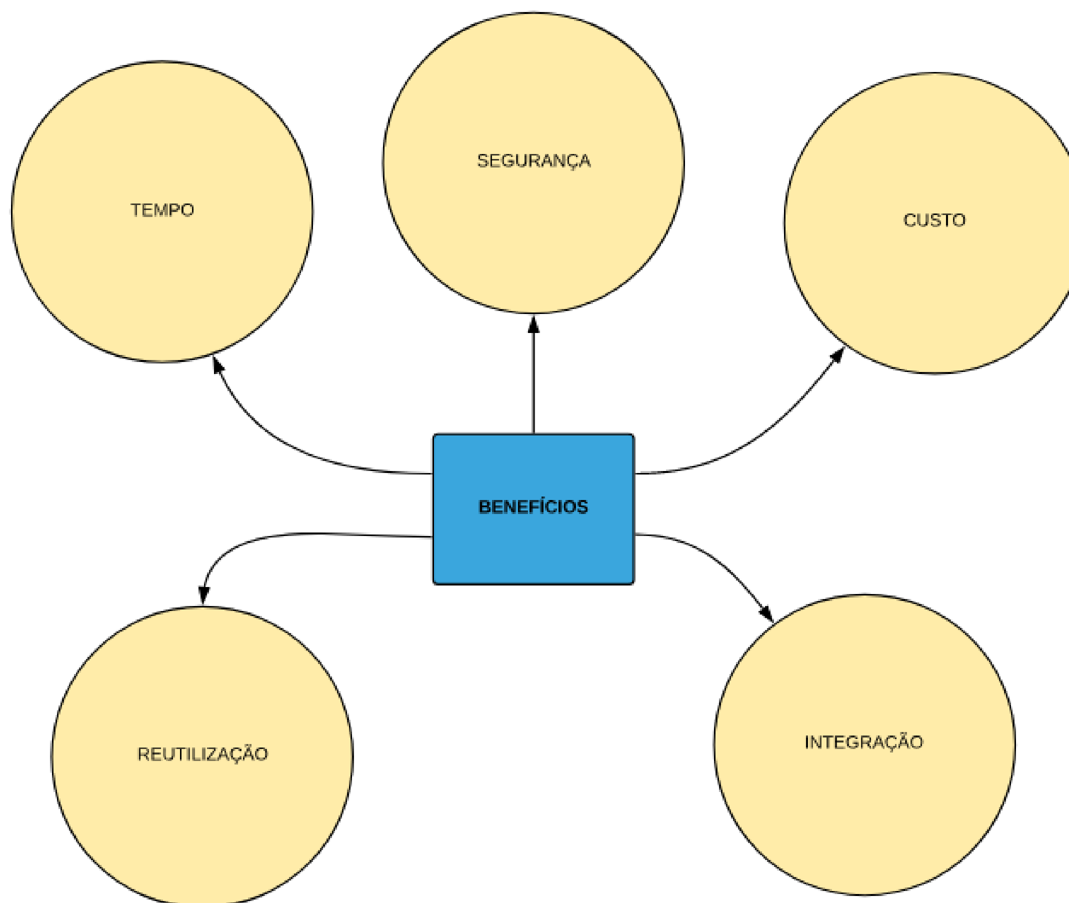
Assim, uma das grandes vantagens do REST é a flexibilidade, já que não limita os formatos de representação de dados. O protocolo REST é também utilizado quando a performance é importante, uma vez que é um protocolo ágil e com a capacidade de transmitir dados diretamente via protocolo HTTP.

Saiba Mais!

Entenda melhor a diferença entre REST e SOAP, veja o artigo no link a seguir: https://www.devmedia.com.br/web-services-rest-versus-soap/32451

1.7 Benefícios dos Web Services

Os benefícios dos Web Services podem ser percebidos tanto a nível tecnológico, como a nível de negócios. Dentre eles, podemos destacar:



- Integração de informação e sistemas: com o uso de CML/JSON e protocolos HTTP tudo torna-se a ser mais simplista no funcionamento entre aplicações e sistemas. . Assim, Web Services permite a troca de informação entre sistemas, sem a necessidade do entendimento de informação detalhado sobre o funcionamento deles. Os Web Services permite a interligação seja entre Sistemas Operacionais como também trabalhar as diversas linguagens de programação nas plataformas de forma independente.

- Reutilização de código: Web Services pode possuir a reutilização dos seus códigos com diferentes tipos de plataformas com implementações e objetivos diferentes. . O código escrito único é usado por diversas aplicações.

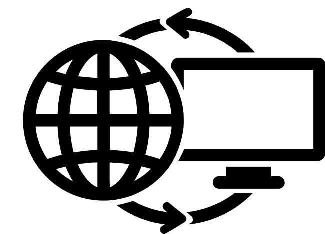
- Redução do tempo de desenvolvimento: o desenvolvimento com Web Services tendem possuir uma melhor agilidade pois a implementação não são totalmente a partir do zero nos sistemas. A redução de tempo permite uma forma melhor de trabalho dentro do desenvolvimento produzindo uma implementação também melhor com funcionalidades existentes ou novas e por diversos tipos de aplicações ganhando uma característica de opção para desenvolvimento de programa sob demanda.

- Melhor segurança: Web Services trabalha com aplicações evitando a comunicação direta por meio de um serviço implementado entre a aplicação e o banco de dados. . Assim, a segurança do sistema que fornece os dados está garantida pela camada de Back-End.

- Reduzir os custos: Ao usar o Web Services não é necessário criar aplicações sob demanda para a integração de dados. Em alguns casos, a criação de aplicações de integração de sistemas pode ser tão dispendiosa quanto o desenvolvimento do próprio sistema. Os Web Services tiram partido de protocolos e da infraestrutura Web já existente na organização, exigindo um investimento mais modesto e atrativo.

Resumo

Nesta unidade abordamos os conceitos básicos sobre Web Services, sua organização, vantagens, aplicabilidade, como ocorre seu funcionamento e os protocolos REST e SOAP. Além disso, revisamos o conceito do desenvolvimento em camadas, relembrando a função do Front-End e do Back-End e como Web Services são aplicados nestes conceitos.



Atividades de Revisão

1. Explique o que são Web Services e como funcionam.

2. Dentre benefícios e vantagens do uso de Web Services, escolha um de cada e justifique a sua resposta, considerando o contexto de desenvolvimento de jogos.

3. Qual a função dos protocolos REST e SOAP e qual a diferença entre eles?

4. Considerando o conceito de desenvolvimento de aplicações em camadas (Front-End e Back-End), explique a utilização dos Web Services aplicado ao desenvolvimento de jogos.

Referências

DEVMEDIA. WebServices: Tutorial sobre WebServices. Disponível em: <https://www.devmedia.com.br/web-services/2873>. Acesso em: fev 2019.

W3C - World Wide Web Consortium. The Web Services Resource Access Working Group, 2011. Disponível em: <https://www.w3.org/2002/ws/ra/>. Acesso em: fev 2019.

Unidade 2 Instalação e configuração de um WebServices

Nesta unidade conheceremos o processo de instalação e configuração de Web Services. Para isso, utilizaremos o Node.js que permite o desenvolvimento de aplicações, no lado servidor, utilizando JavaScript. Ao final desta unidade, teremos um Serviço Web instalado e rodando, bem como, todas a ferramentas para o desenvolvimento de aplicações e de jogos utilizando esta tecnologia. Boa leitura a todos.

2.1 Objetivos específicos da Unidade 2

Ao final desta unidade, espero que você seja capaz de:

Instalar e configurar o ambiente para desenvolvimento de um WebServices;

Desenvolver WebServices utilizando NodeJS e Javascritpt/TypeScript;

2.2 O que é o Node.js

O criador de Node.js, Ryan Dahl, ao utilizar uma página do Flickr , queria realizar upload de uma foto com uma barra de progresso para verificar o progresso do upload, percebeu que no navegador padrão daquela época era complicado implementar essa informação devido a linguagem dinâmica.

Sendo um interpretador de código, o Node.Js foi uma ideia de usar o Java Script para desempenhar funções de WebServices. A manipulação de conexões simultâneas e o próprio Java Script, possibilitou facilitar os programadores.

Você já pensou em trabalhar com códigos que controlam muitas conexões simultâneas em um único servidor?

Ahh!!! Viu?! Como não pensar no html 5 com jogos multiplayers.

Saiba Mais!

No Brasil, a NodeBR ficou responsável pela divulgação da tecnologia do uso do Node.js segundo as definições mundiais da Nodejs.org.

O Node.js é considerado recente. Foi criado foi em 2009. Para alguns estudiosos, isso é uma desvantagem em relação às linguagens mais maduras como a linguagem Python - 1991, a JavaVM - 1995, e até mesmo o .NET - 2000. Mas o Node.js já possui muita coisa criada pra ele e já pode ser considerado maduro, o que nos garante mais confiança para executar projetos maiores e mais pesados. Para conhecer mais, acesse o link: http://nodebr.com/o-que-e-node-js/

Para entendermos o Node.js, vamos compreender que o Java Script é uma linguagem que veio com desenvolvimento de muitos anos. Foi sempre desenvolvida para clientes e todas as outras formas para servidores falharam. Com as melhorias das tecnologias, a maior característica de facilidade encontrada no Node.js é o fato de ter um único ponto de execução do programa chamado de single-thread, enquanto todos os outros eram multi-thread. Vamos pensar num servidor web tradicional, em que cada requisição esperava terminar a execução para alocar recursos, no Node.js, apenas um ponto de execução trata todas as requisições através do uso de eventos. Nos servidores tradicionais, a execução para dar continuidade esperava as operações como o acesso ao banco de dados e a leitura dos arquivos. No Node.js, por sua vez, isso não ocorre, porque essas atividades não são realizadas ao mesmo tempo, não bloqueando a execução do programa. Além disso, enquanto nos servidores tradicionais a execução espera as conclusões das operações de entrada e saída, o Node.js usa o evento chamado de event Loop para tratar as requisições concorrentes, em pilha de eventos, permitindo superar o modelo tradicional.

O Node.js usa o mesmo interpretador do Google Chrome para intepretação do Java Script chamado de Máquina Virtual V8. Isso permite que o Node.js seja eficiente e também que consuma menos recursos.

Fique Ligado!

O conceito de máquina virtual, além da execução de programas, poderá ser visto melhor na disciplina concomitante de Sistemas Operacionais e Arquitetura Mobile!!!

Saiba Mais!

Existe um site muito bom que explica as características do Node.Js, o problema das arquiteturas bloqueantes entre outras questões. Sugiro a consulta, pois ajudará a entender melhor os textos da unidade 2. Acesse o artigo "Programação assíncrona com Node.JS" no endereço:

https://www.devmedia.com.br/programacao-assincrona-com-node-js/31509#arquiteturas-bloqueantes

Procure pela Guia do artigo desse site. Recomendo!

2.3 Como funciona o Node.js

Como o tempo de execução de uma aplicação JavaScript é orientada por eventos assíncronos, o Node.js foi projetado para criar aplicativos de rede escalonáveis. No exemplo da Figura 2, temos o código para criação de uma aplicação "hello world", utilizando Node.js.

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Figura 2. Código base para criação de uma aplicação em Node.js (Nodejs.org, 2019).

No exemplo “hello world”, podemos ter muitas conexões que podem ser acessadas e tratadas simultaneamente. Em cada conexão, o retorno de chamada é acionado, retornando à frase “hello world” que será apresentada no Front-End, mas se não houver mais nenhuma requisição, o Node.js entrará no modo de espera, até que uma nova requisição seja enviada ao servidor.

2.4 O que podemos ver de diferente no Node.js em relação a programação?

Podemos, facilmente, entender essa diferença, quando nos referimos a java e a java script. Enquanto, numa linguagem java, o código precisa ser compilado, permitindo a criação de aplicações executáveis, o java script é totalmente composto por texto e é executado em um navegador. Dessa forma, o Node.js não é uma linguagem de programação e utiliza o mesmo interpretador do navegador Google Chrome.

Na linguagem de programação, a cada requisição recebida, cria-se uma nova chamada de execução para atendê-la. Chamamos, isso de multi-threading. Cada thread (encadeamento de execução) para uma requisição facilita a programação, pois divide o programa em vários processos, permitindo criar um programa desenvolvido em módulos, já que enquanto um processo aguarda, o outro processo pode trabalhar. Porém, a concomitância de vários processos funcionando torna tudo muito complexo. Esse comportamento de aguardar é chamado de bloqueante. O conceito de multi-threading funciona muito bem na área de programação de jogos multiplayers. No entanto, os recursos de máquina serão enormes.

Já em Node.js, usa-se uma única thread de execução e não é um framework java script. Podemos ver semelhanças na programação ou no framework quando nos referimos ao Node.js, mas não podemos confundi-los. O Node.js não bloqueia o processo, os processos ocorrem de forma paralela, possuindo apenas um fluxo de execução chamado de evento loop.

Saiba Mais!

Para saber mais sobre o Node.js, acesse o site oficial do ambiente: https://nodejs.org/en/about/

2.5 Instalando e executando o Node.js

Para o desenvolvimento em Node.js, é necessária a instalação e a configuração do ambiente. O Node.js é multiplataforma, ou seja, está disponível para vários sistemas operacionais. No nosso caso, iremos utilizar o Windows. Para isso, devemos acessar o site https://nodejs.org/en/, conforme a figura 3 e efetuar o download do instalador do ambiente.

Node.js
https://nodejs.org/en/
HOME | ABOUT | DOWNLOADS | DOCS | GET INVOLVED | SECURITY | NEWS | FOUNDATION
Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.
Download for Windows (x64)
Versão mais utilizada
10.15.3 LTS
Recommended For Most Users
Other Downloads | Changelog | API Docs
Versão mais atualizada
11.12.0 Current
Latest Features
Other Downloads | Changelog | API Docs
Or have a look at the Long Term Support (LTS) schedule.
Sign up for Node.js Everywhere, the official Node.js Monthly Newsletter.
LINUX FOUNDATION COLLABORATIVE PROJECTS
Report Node.js issue | Report website issue | Get Help
© Node.js Foundation. All Rights Reserved. Portions of this site originally © Joyent.
Node.js is a trademark of Joyent, Inc. and is used with its permission. Please review the Trademark Guidelines of the Node.js Foundation.

Figura 3. Site oficial do Node.js.

Na Figura 3, temos duas opções de download. Uma versão recomendada para a maioria dos usuários, mais estável e mais utilizada no momento. A outra versão é mais atual, com as últimas atualizações disponíveis. Para a nossa instalação, iremos baixar a versão mais utilizada e mais estável.

Após efetuar o download do instalador, execute o programa. Após a instalação, podemos abrir o prompt de comandos do Windows e verificar a versão do Node.js que foi instalada, utilizando o comando “node -v”. Isso indica que a instalação foi bem-sucedida. Veja a Figura 4.

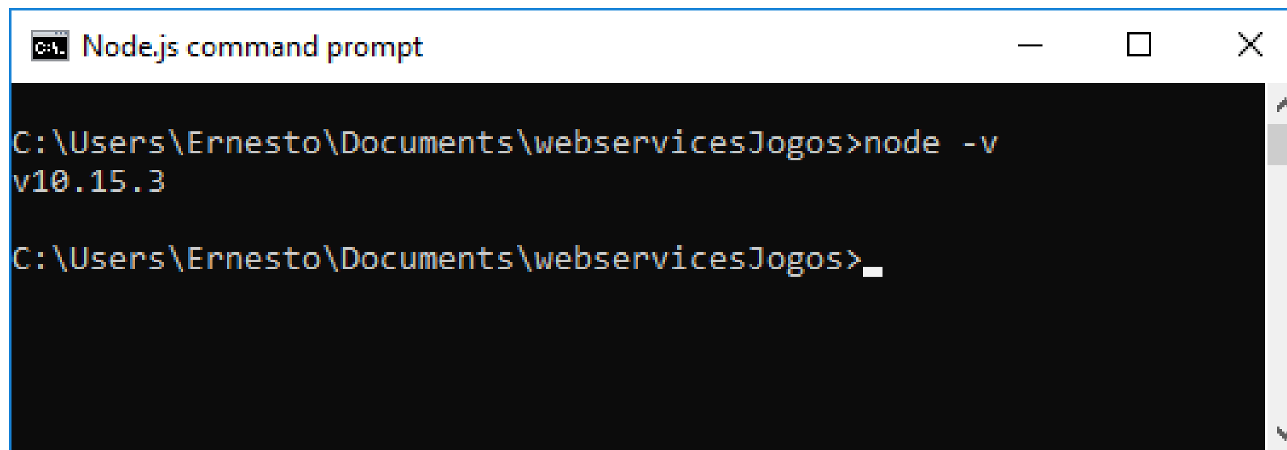


Figura 4. Prompt de comandos do windows com o comando “node -v”.

Curte aí!

O Node.js possui uma vasta documentação e uma API. Para conhecê-la, acesse o link oficial do projeto, a seguir: https://nodejs.org/en/docs/

Com o Node.js instalado, criaremos uma aplicação de teste para verificar o funcionamento do nosso ambiente. Para isso, vamos criar uma pasta “webservicesJogos” no prompt de comandos do Windows e entrar nesta pasta. Dentro dela, criaremos um arquivo com extensão .js “app.js”. Este arquivo pode ser editado em qualquer editor de textos simples, como, por exemplo: o bloco de notas e/ou o notepad++. Em nossos exemplos, utilizaremos o notepad ++.

Saiba Mais!

Para saber mais sobre o notepad++, como informações, download e funcionalidades, acesse o site oficial do notepad++, a seguir: https://notepad-plus-plus.org/

Com o arquivo “app.js” e o notepad++ instalado, abra o arquivo no editor e insira um comando javascript simples:

```
console.log(“Teste node.js”);
```

Feito isso, vamos salvar o arquivo e executar o nosso primeiro exemplo em node.js, no prompt de comandos do windows, utilizando o comando “node app.js”. A nossa aplicação imprimirá a mensagem Teste node.js, demonstrando que nosso ambiente está instalado e funcionando corretamente. Veja o resultado deste procedimento na Figura 5.

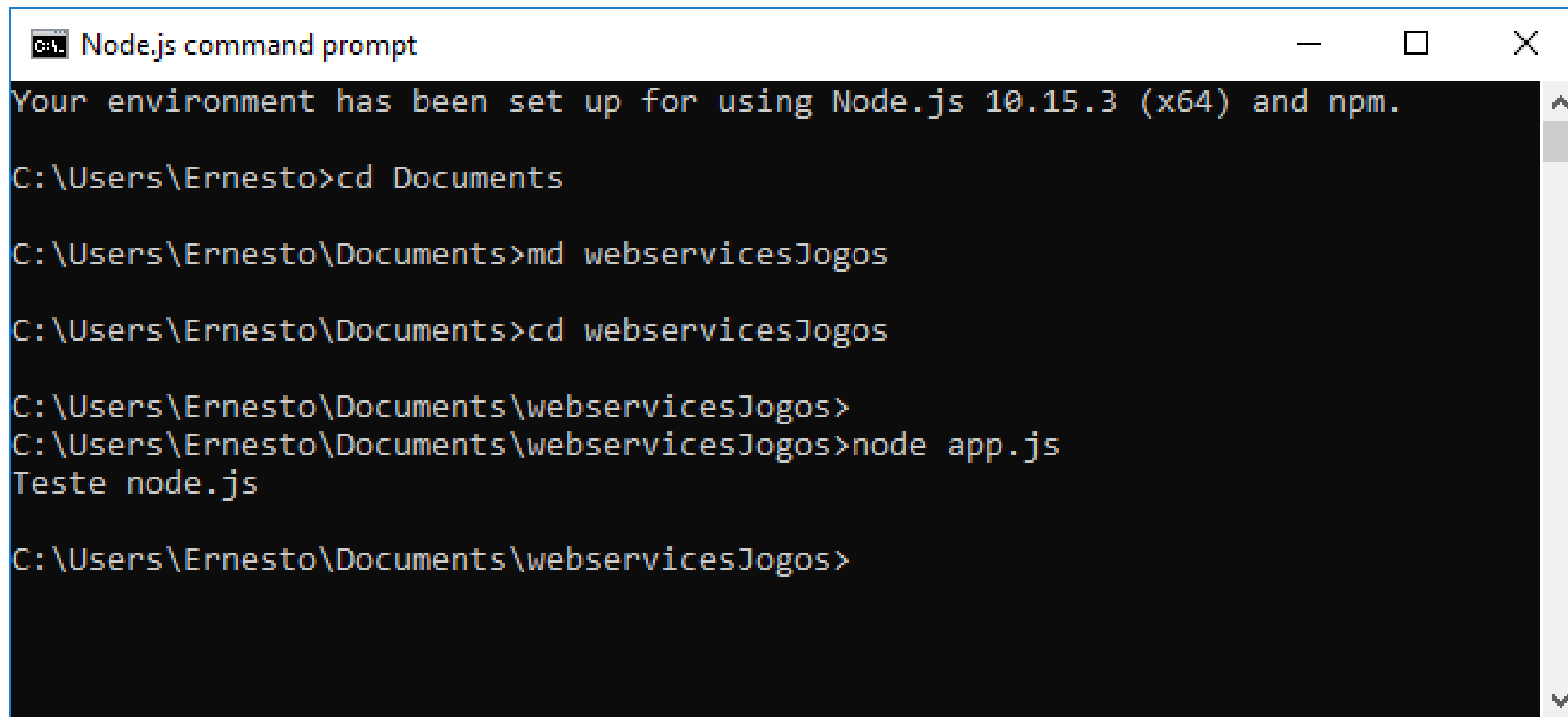


Figura 5. Prompt de comandos do windows com o comando “node app.js”.

Uma das grandes vantagens do node.js é que os desenvolvedores não precisam aprender uma nova linguagem de programação para criar suas aplicações. No exemplo, abaixo, temos uma aplicação em javascript, no qual declaramos a variável texto = “Node.js”; seguida pela definição da função mostraMsg, que recebe um parâmetro msg e mostra uma mensagem no prompt de comandos do Windows, finalizando com a chamada da função mostraMsg, que executa esta função.

EXEMPLO:

```
var texto = “Node.js”;
function mostraMsg(msg){
        console.log(“Ola, “, msg);
}
mostraMsg(texto);
```

Observe, na Figura 6, o resultado da execução do exemplo anterior.

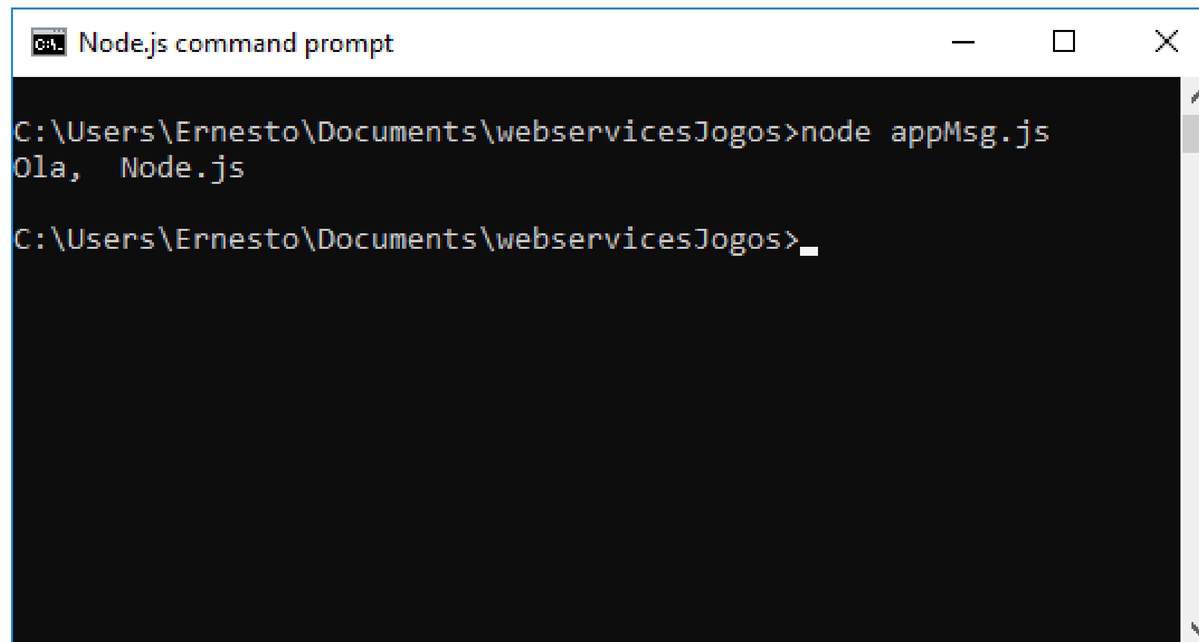


Figura 6. Prompt de comandos do windows com o comando “node appMsg.js”.

Saiba Mais!

Para a codificação dos nossos jogos em node.js, utilizaremos a linguagem javascript. Aproveite para revisar seus conceitos sobre esta linguagem de programação, acessando o site:

https://www.w3schools.com/js/

2.6 Instalando e criando módulos no node.js

Após a instalação do Node.js, aprenderemos a instalar os módulos que permitem a execução de funcionalidades mais complexas, como impressões diferenciadas no console, servidores HTTP, etc. Estes pacotes são conjuntos de classes e funções javascript pré-implementadas, que serão instalados e/ou importados para nossa aplicação, somente quando forem necessários.

Como o Node.js é modular, a instalação inicial disponibiliza o Node Package Manager - NPM, que é o instalador padrão de pacotes do Node.js. Com o NPM, podemos instalar, atualizar e desinstalar os pacotes em nossas aplicações. Podemos verificar a versão do NPM, instalada junto com o Node.js, ao executarmos o comando “npm -v”, no prompt de comandos do Windows.

Da mesma forma, para instalar um novo pacote, devemos executar o comando “npm install cli-color”, ou seja, o comando npm install + <o nome do pacote>. Neste caso, instalaremos o pacote cli-color, que permite imprimir textos coloridos na console de saída. Durante este processo, o NPM busca os arquivos na Web e instala o pacote no diretório “node_modules”, criado dentro do diretório da aplicação. Veja, na Figura 7, o exemplo da instalação do pacote.

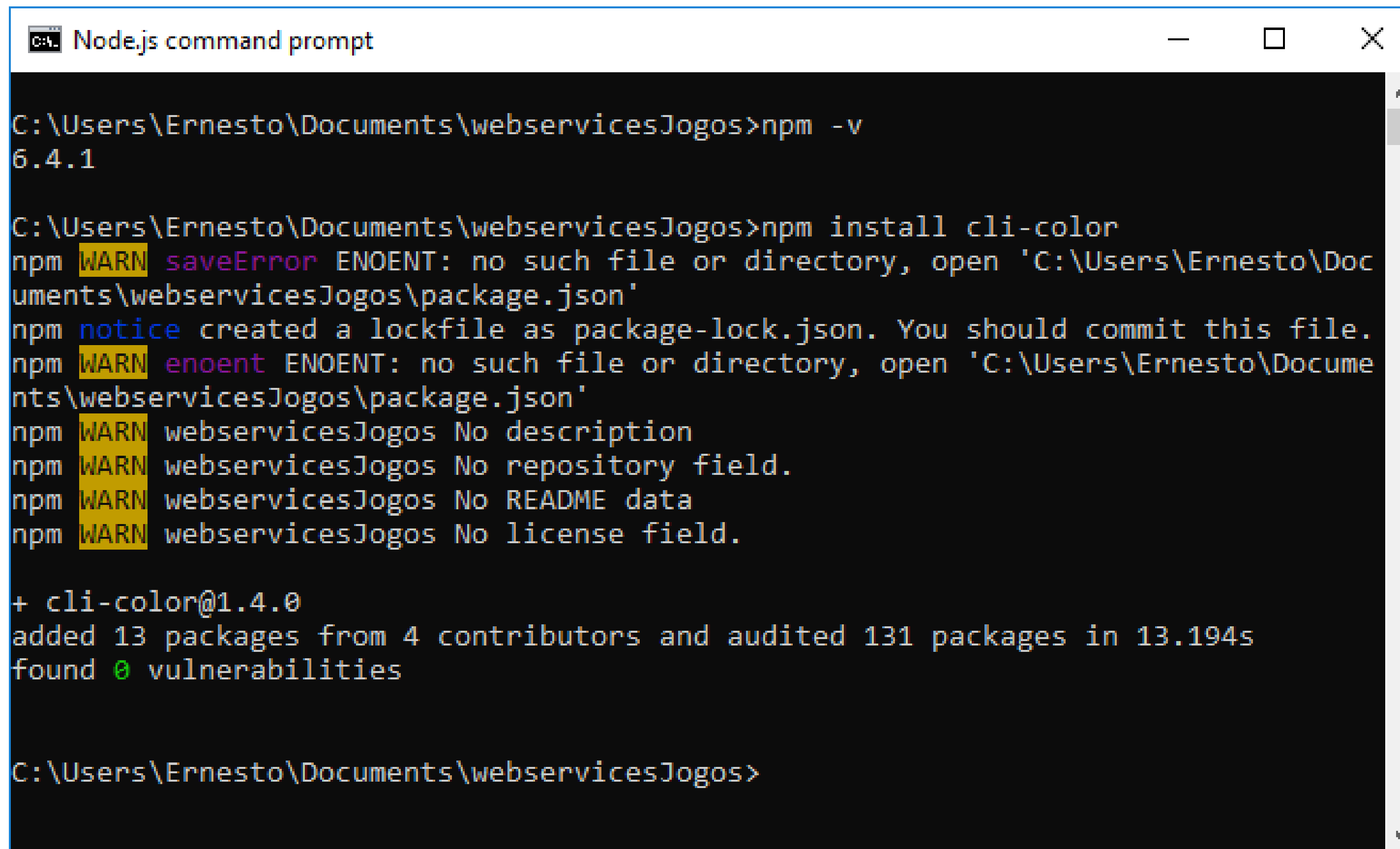


Figura 7. Prompt de comandos do windows com o comando “npm install cli-color”.

Para utilizarmos este pacote em nossas aplicações, devemos importá-lo para nossa aplicação, declarando uma variável que faz referência ao pacote que queremos importar, utilizando a função require("cli-color"), passando o nome do pacote como parâmetro. Após isso, podemos utilizar as funções deste pacote conforme o exemplo de código que segue:

```
var pqg= require("cli-color");

console.log(pqg.grenn("texto verde");
```

Ao atualizarmos este exemplo no arquivo app.js e executarmos no prompt de comandos do Windows, temos o resultado apresentado na Figura 8.

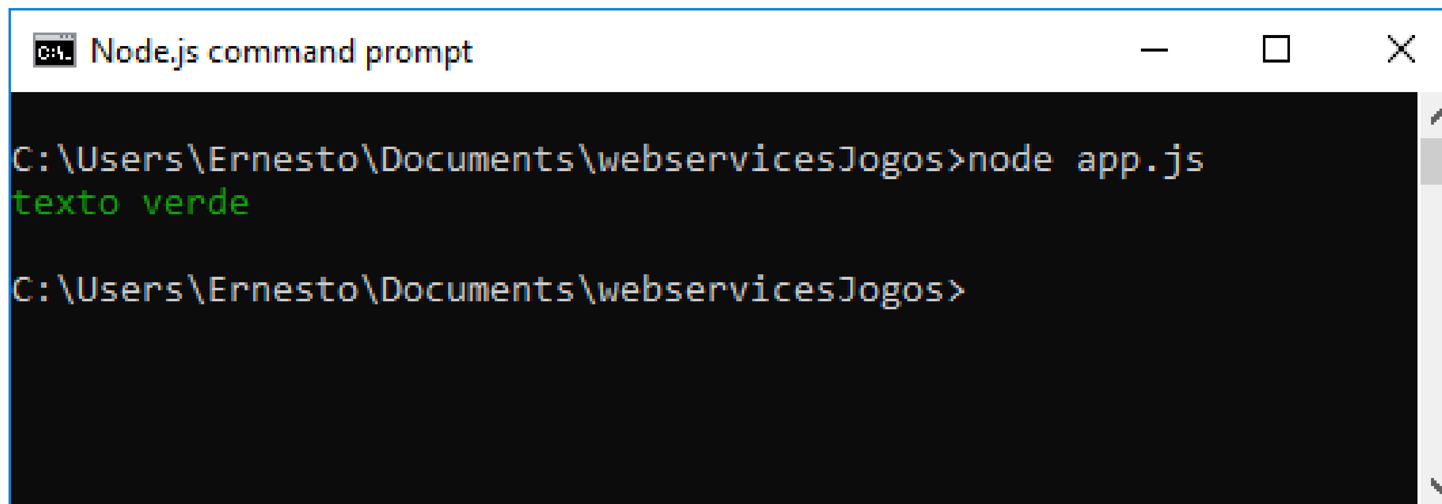


Figura 8. Prompt de comandos do windows com o comando "node app.js" utilizando o pacote cli-color.

Além de importarmos e utilizarmos pacotes prontos, o Node.js permite que possamos criar nossos próprios pacotes. Isso permite ao desenvolvedor criar e organizar os módulos e funcionalidades das aplicações, oportunizando sua utilização em diversos softwares diferentes.

Para isso, criaremos o pacote funcoes.js, no qual configuraremos uma função “logar”, que receberá dois parâmetros: login e senha. A validação destes valores retornará à expressão true (quando eles estiverem corretos) ou false (quando estiverem errados). Ao final, temos o comando de exports.validarLogin, o qual permitirá que outras aplicações façam uso desta função. Veja o exemplo a seguir:

EXEMPLO:

```
//declaração da função logar
function logar(login, senha){
    //comando de verificação
    if(login==”Jose” && senha==”123”)
        return true;
    else
        return false
}
// comando de exports da função validarLogin
exports.validarLogin = logar;
```

No exemplo acima, podemos perceber que a linguagem de programação utilizada para definição da função é javascript, a mesma linguagem que já era utilizada no front-end. Com isso, podemos utilizar todos os comandos desta linguagem (if/else; while; for; do/while; etc).

Com o código do pacote funcoes pronto, vamos adicionar este pacote ao nosso arquivo app.js, incluindo uma nova variável logar com o require para o pacote. Feito isso, vamos utilizar um comando if para verificar o retorno “correto”, quando os dados de login estiverem certos, e “incorreto”, quando os dados de login estiverem errados.

EXEMPLO:

```
var cores= require(“cli-color”);

var logar= require(“./funcoes.js”);

if (logar.validarLogin(“Jose”, “123”))

        console.log(cores.green(“correto”));

else

        console.log(cores.red(“incorreto”));
```

O resultado da execução do arquivo app.js com a chamada para o pacote funcoes.js está apresentado na Figura 9. Na primeira execução, com os dados corretos e, na segunda, com os dados incorretos. Para visualizar este resultado, substitua a linha: if (logar. validarLogin(“Jose”, “123”)) por: if (logar.validarLogin(“Jose”, “000”)).

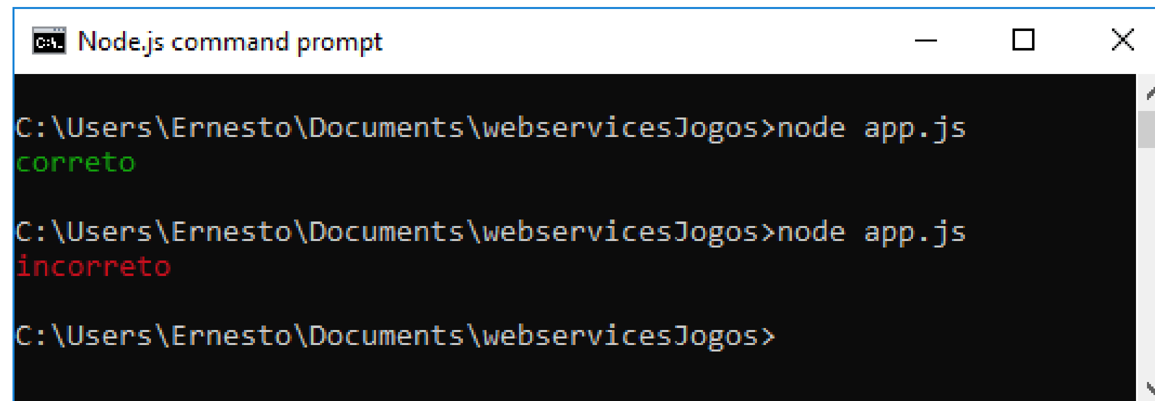


Figura 9. Prompt de comandos do windows com o comando “node app.js” utilizando o pacote funcoes.js.

2.7 Criando uma aplicação Web com Node.js

Até agora, as nossas aplicações apenas mostravam as saídas no prompt de comandos do Windows. Mas, como vimos, a principal vantagem do Node.js é o desenvolvimento para Web. Assim, o nosso próximo passo será criar uma aplicação Web com Node.js e, para isso, utilizaremos o pacote HTTP para criar um servidor e acessá-lo por meio do browser.

Vamos praticar? Crie um arquivo appWeb.js e digite o código a seguir:

EXEMPLO:

```
var http= require(“http”);
http.createServer(function(request, response){
        response.write(“App web com Node.js”);
        response.end();
}).listen(8081);
console.log(“Servidor rodando em http://localhost:8081/”);
```

Neste exemplo, temos a declaração e a criação do objeto http. A partir deste objeto, chamaremos a função createServer, que vai receber como parâmetro outra função. Essa função nos permitirá tratar o request e o response do browser. No final, o método listen(8081) indica a porta em que o servidor estará rodando.

Ao executarmos o arquivo appWeb.js no prompt de comandos do Windows, teremos um servidor Web rodando e pronto para responder a requisições na porta 8081 do br. Em seguida, digite a url localhost:8081 no browser. Veja o resultado na Figura 10.

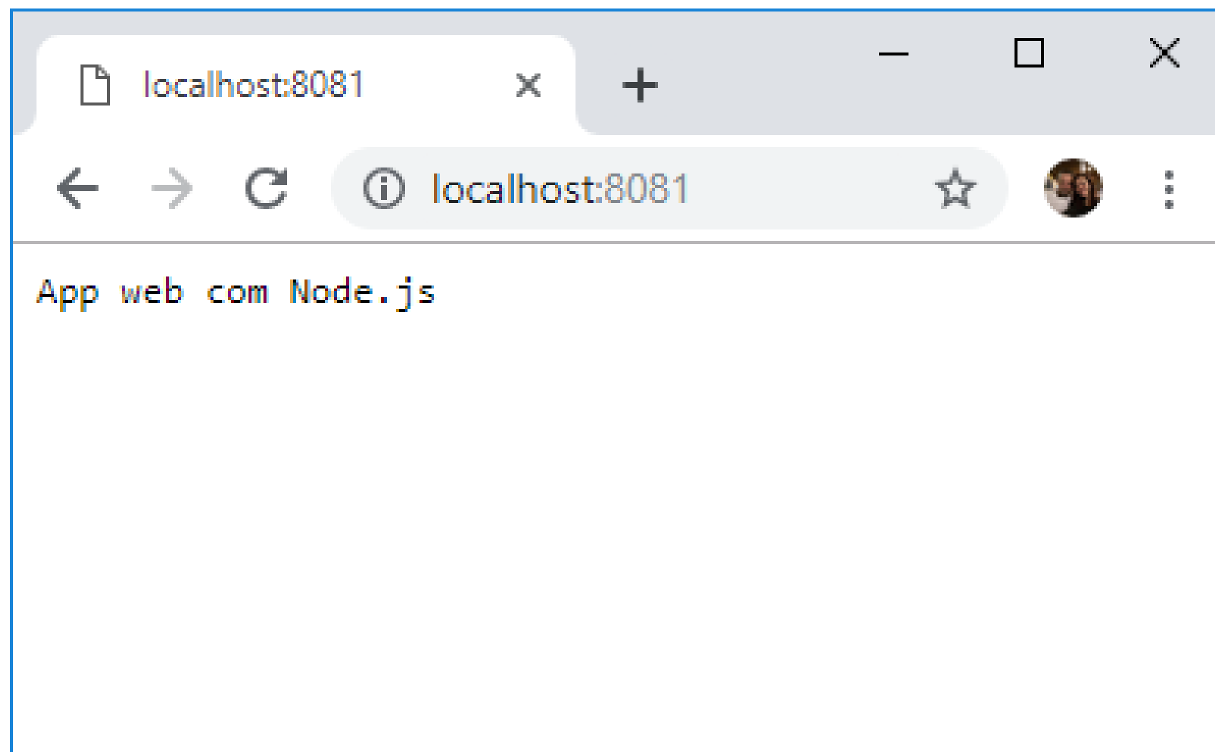


Figura 10. Servidor http do Node.js rodando na porta 8081.

Fique Ligado!

Quando executarmos o comando node appWeb.js, no prompt de comandos do Windows, o servidor ficará rodando. Mas sempre que houver a necessidade de atualizar o arquivo appWeb.js, o servidor deverá ser reiniciado para que estas alterações tenham efeito.

Para isso, no prompt de comandos do Windows, utilize o comando ctrl+c, para a execução do comando corrente e reinicie o servidor usando o comando node appWeb.js.

Com um servidor Web rodando, o nosso próximo passo é a criação de aplicações Web que permitam carregar arquivos HTML estáticos e entregar seu conteúdo como resposta às requisições do cliente, visualizando seu conteúdo no browser.

Para isso, crie inicialmente um arquivo index.html composto por um código html simples que mostra um título h1 no servidor, conforme o código a seguir:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <title>Html com Node.js</title>
    <meta charset="utf-8">
  </head>
  <body>
```

```
      <h1>Primeiro arquivo html com Node.js</h1>
   </body>
</html>
```

Após criarmos o arquivo html, temos que reescrever o arquivo appWeb.js, para que ele envie este arquivo com resposta da requisição do browser. Veja o exemplo abaixo:

```
var http= require("http");
var fs= require("fs");
http.createServer(function(request, response){
        fs.readFile("index.html", function(erro, conteudo){
                if(erro){
                        console.log(erro);
                        response.write("Erro no servidor");
                }
                else response.write(conteudo);
                response.end(); })
}).listen(8081);
console.log("Servidor rodando na porta 8081");
```

Ao editarmos o arquivo appWeb.js, incluiremos a declaração e a criação do objeto fs, fazendo referência com o pacote fs - file system, que permite a manipulação de arquivos. Dentro da função http.createServer, utilizaremos a função fs.readFile, que recebe o index.html como parâmetro, junto com outra função que permite escrever este arquivo na saída para o browser. Em caso de erro, será apresentada uma mensagem ao usuário e o erro ocorrido será mostrado no prompt de comandos do servidor, conforme mostra a Figura 11.

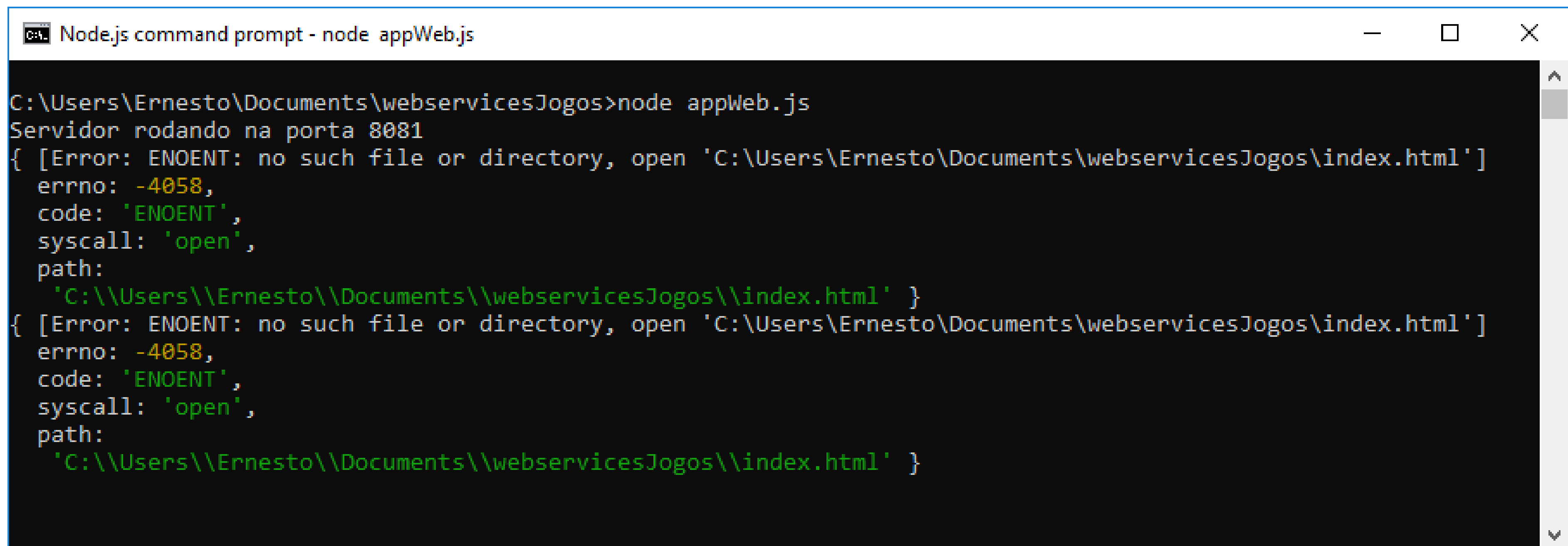


Figura 11. Mensagem de erro no prompt de comandos do servidor.

Neste caso, removemos o arquivo index.html do diretório da aplicação e provocamos um erro ao digitarmos a url localhost:8081 no browser. Para corrigir este problema, basta recolocar o arquivo na pasta e atualizar o browser. Assim, apresentaremos a mensagem: “Primeiro arquivo html com Node.js”.

Veja o resultado correto da execução do arquivo index.html na Figura 12.

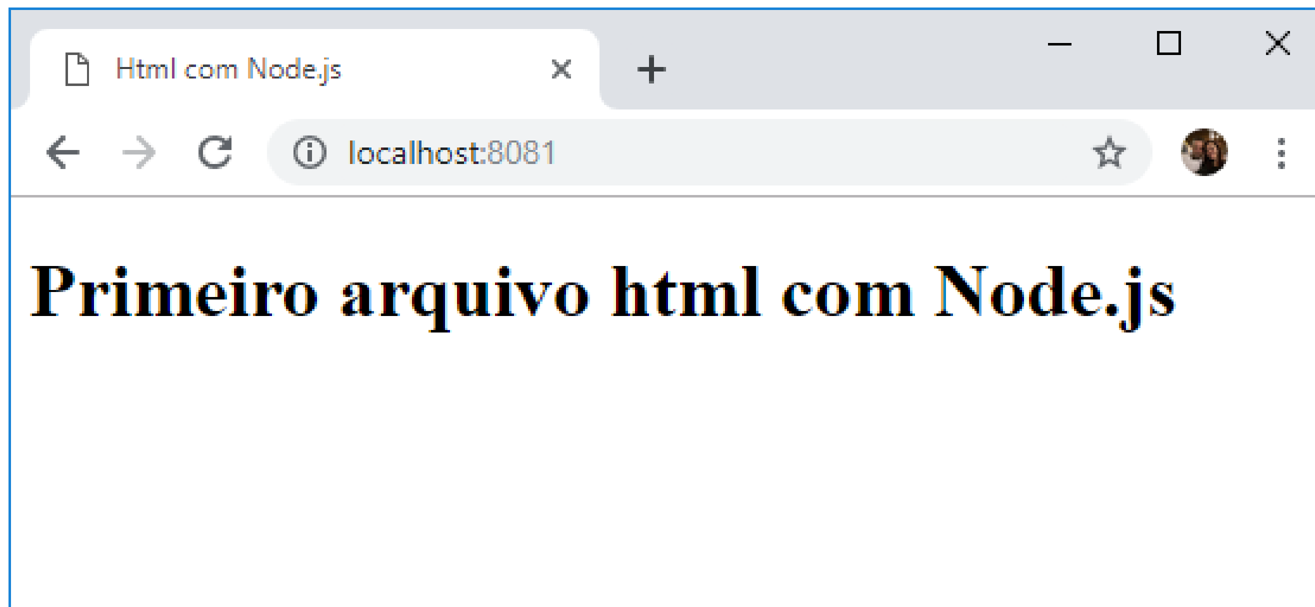
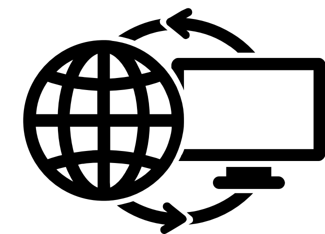


Figura 12. Arquivo index.html executado no servidor http do node.js.

Resumo

Nesta unidade vimos o processo de instalação e configuração de um WebServices. Para isso, utilizamos o Node.js que permite o desenvolvimento de aplicações, no lado servidor, utilizando JavaScript. Ao final temos um WebServices instalado e rodando, bem como, todas a ferramentas para o desenvolvimento de aplicações e jogos utilizando esta tecnologia.



Atividades de Revisão

1- Explique o que é o Node.js e como ele funciona.

2- Explique como criar e instalar módulos em uma aplicação do Node.js.

3- Explique como mostrar arquivos html, utilizando o Node.js.

4- Crie um arquivo html que mostra a mensagem “hello world” no browser, utilizando um servidor http do Node.js.

Referências

DEVMEDIA. WebServices: Tutorial sobre WebServices. Disponível em: <https://www.devmedia.com.br/web-services/2873>. Acesso em: fev 2019.

Nodejs.org. About Node.js. Disponível em: <https://nodejs.org/en/about/>. Acesso em: fev 2019.

W3C - World Wide Web Consortium. The Web Services Resource Access Working Group, 2011. Disponível em: <https://www.w3.org/2002/ws/ra/>. Acesso em: fev 2019.

Unidade 3 Desenvolvimento uma aplicação básica de Back-end com Banco de Dados

Nesta unidade, aplicaremos os conceitos WebServices ao desenvolvermos uma aplicação de Back-end com acesso ao banco de dados. Para isso, utilizaremos o ambiente instalado e configurado na unidade anterior.

Interaja com o assuntos selecionados para essa unidade e lembre-se de executar os exemplos sugeridos no decorrer do material.

3.1 Preparando servidor HTTP

Conforme vimos na unidade anterior, para o desenvolvimento de aplicações com o Node.js precisamos do próprio Node.js instalado na nossa máquina, bem como do NPM para gerenciamento dos pacotes.

Com isto feito, vamos criar um novo diretório para nossa aplicação com o nome web-api e dentro desta pasta, executaremos o comando npm init para iniciarmos o NPM no projeto e, criarmos o arquivo package.json. Esse arquivo conterá as configurações do projeto como as dependências e as scripts de execução da aplicação.

Para que o arquivo seja criado, o NPM efetuará algumas perguntas, como:

1. Nome do pacote, sugerindo web-api. Para confirmar, basta teclar enter.

    package name: (web-api)

2. Versão do pacote, sugerindo 1.0.0. Para confirmar, basta teclar enter.

    version: (1.0.0)

3. Descrição, que pode ser deixado em branco. Para confirmar, basta teclar enter.

    description:

4. Ponto de partida da aplicação, que é o arquivo inicial da nossa aplicação, sugerindo index.js. Neste caso, alteraremos digitando src/app.js para redefinir este parâmetro.

```
entry point: (index.js) src/app.js
```

5. Os demais itens podem ser mantidos no valor default, teclando enter para cada um deles.

```
test command:
git repository:
keywords:
author:
license: (ISC)
```

6. No final, o NPM mostra uma prévia do arquivo packaje.json que está sendo criado. Para confirmar, devemos digitar yes e confirmar.

```
{
  “name”: “web-api”,
  “version”: “1.0.0”,
  “description”: “”,
  “main”: “src/app.js”,
  “scripts”: {
    “test”: “echo \”Error: no test specified\” && exit 1”
```

```
    },
    "author": "",
    "license": "ISC"
  }
  Is this OK? (yes) yes
```

Veja o procedimento completo na Figura 12:Antes da decisão final da execução de um evento, faz-se um estudo de via- bilidade. Planejamento bem organizado e planejamento acompanhado são sempre as palavras-chave para evitar transtornos financeiros para organiza- dores e promotores do evento.

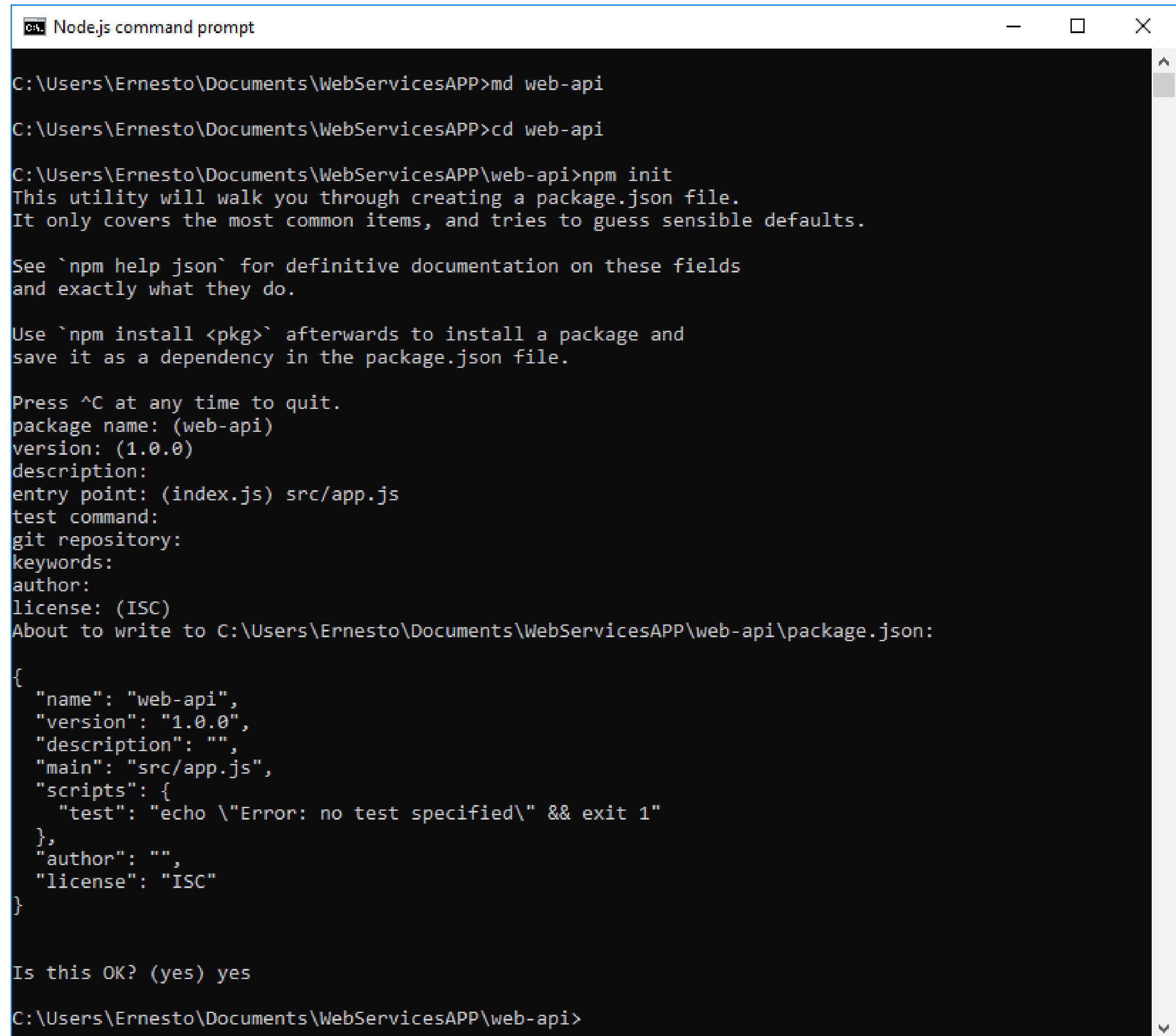


Figura 12. Procedimento de criação do arquivo packaje.json, utilizando o comando npm init.

Após a criação do packaje.json criaremos o pacote src dentro do pacote web-api. Com isso, separaremos os arquivos de código dos arquivos de configuração e demais arquivos do nosso projeto, organizando estes arquivos e facilitando a sua localização dentro da árvore de diretórios.

Dentro do pacote src criaremos o arquivo app.js que será o arquivo de start da nossa aplicação e a primeira coisa a ser feita é a inicialização do nosso servidor HTTP. Para isso, importaremos o módulo HTTP utilizando o comando require, conforme vimos na unidade anterior. Veja a Figura 13 com o código completo.

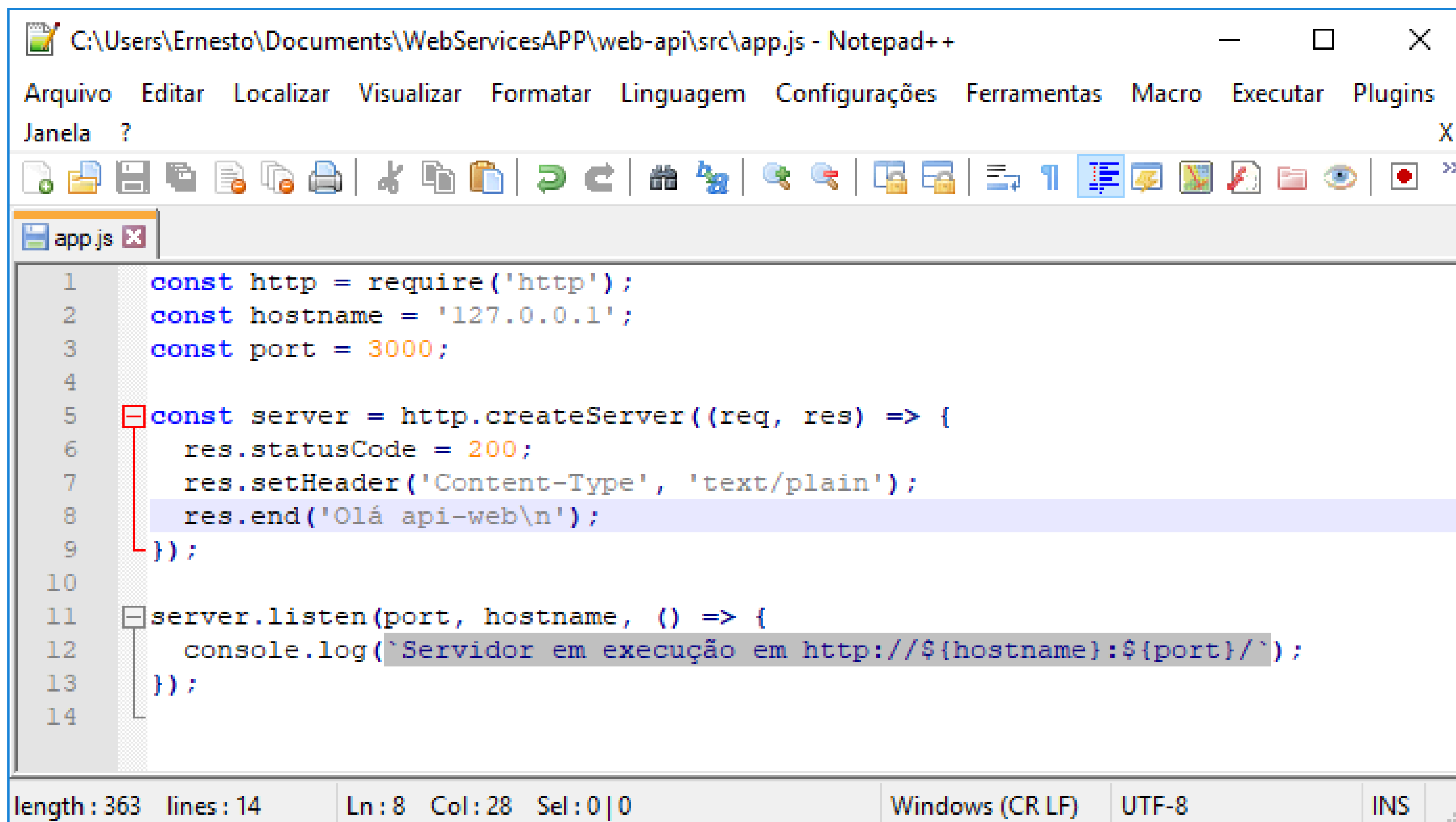


Figura 13. Código do arquivo app.js que cria o servidor HTTP do projeto web-api.

- Linha 1: Temos a importação da biblioteca http o.

- Linha 2: Salvamos o valor do hostname.

- Linha 3: Definimos o valor da porta da rede, no caso porta 3000.

- Linha 5: Criamos a constante server para armazenar o objeto referente ao servidor da aplicação em si, fazemos isso por executar o método createServer() da biblioteca HTTP. Este método recebe como parâmetro uma função, então lhe é passado uma função anônima seguindo a notação de flecha do JavaScript. O método createServer passará como parâmetros da função anônima dois objetos:

    - O primeiro é um objeto referente à requisição HTTP feita ao servidor.

    - O segundo é um objeto referente à resposta que será enviada a quem fez a requisição.

- Linha 6: Aqui já estamos dentro da função anônima passada como parâmetro de createServer. A primeira coisa que será feita: definir que o atributo statusCode do objeto res receberá o valor 200. Isto significa que a resposta que será enviada ao cliente terá o status HTTP de 200, que significa OK.

- Linha 7: Executamos o método statusCode do objeto res para adicionar um cabeçalho à resposta que será enviada. Neste caso, estamos adicionando o cabeçalho “Content-Type: text/plain”.

- Linha 8: Executamos o método end do objeto res para finalizar o ciclo requisição-resposta e enviar a resposta ao cliente que tiver feito a requisição.

Com isso, temos nosso servidor HTTP rodando. Teste o seu funcionamento digitando http://127.0.0.1:3000/ na url do browser. Porém, com esta configuração básica, nosso servidor não consegue responder nada além de “Olá api-web”. Ele nem mesmo consegue responder erro404, no caso de uma requisição para uma outra url não encontrada.

Para tratar essas requisições genéricas de um servidor HTTP, instalaremos o express que é um framework para o Node.js, que permite a definição das rotas do servidor. Estas rotas definirão as respostas do servidor para cada tipo de requisição. Desse modo, podemos criar um servidor de alto nível mais completo.

Curte aí!

Para saber mais detalhes sobre o Express acesse o site do NodeBR e leia o artigo “Primeiros passos com Express em Node.js” no link: http://nodebr.com/primeiros-passos-com-express-em-node-js/

Para instalar o express, pararemos a execução do servidor HTTP do Node.js e digitaremos o comando “npm i express --save” que fará o download e a instalação do pacote do express no nosso servidor e o comando npm install http-status --save A opção --save irá fazer o registro no arquivo package.jason. Veja o exemplo da instalação do pacote, na Figura 14:

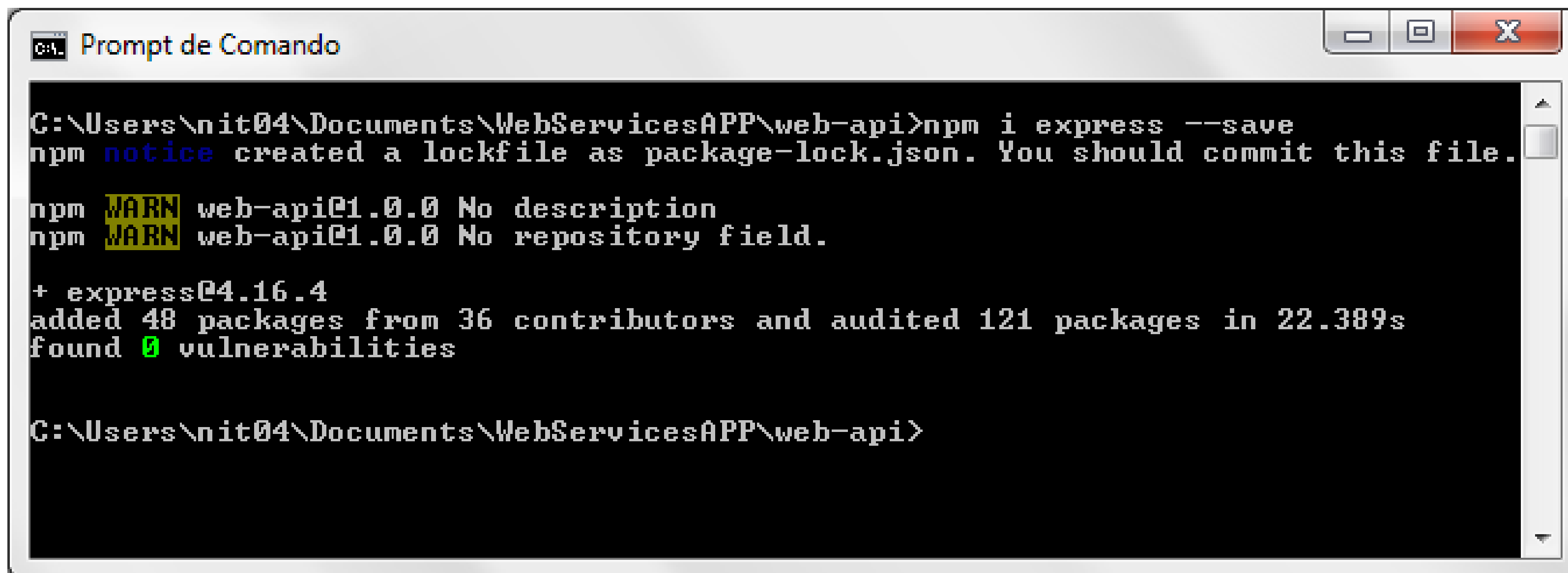


Figura 14: Instalação do pacote do framework express no projeto web-api.

Após a instalação do express, acrescentaremos também o pacote http-satus no projeto web-api.

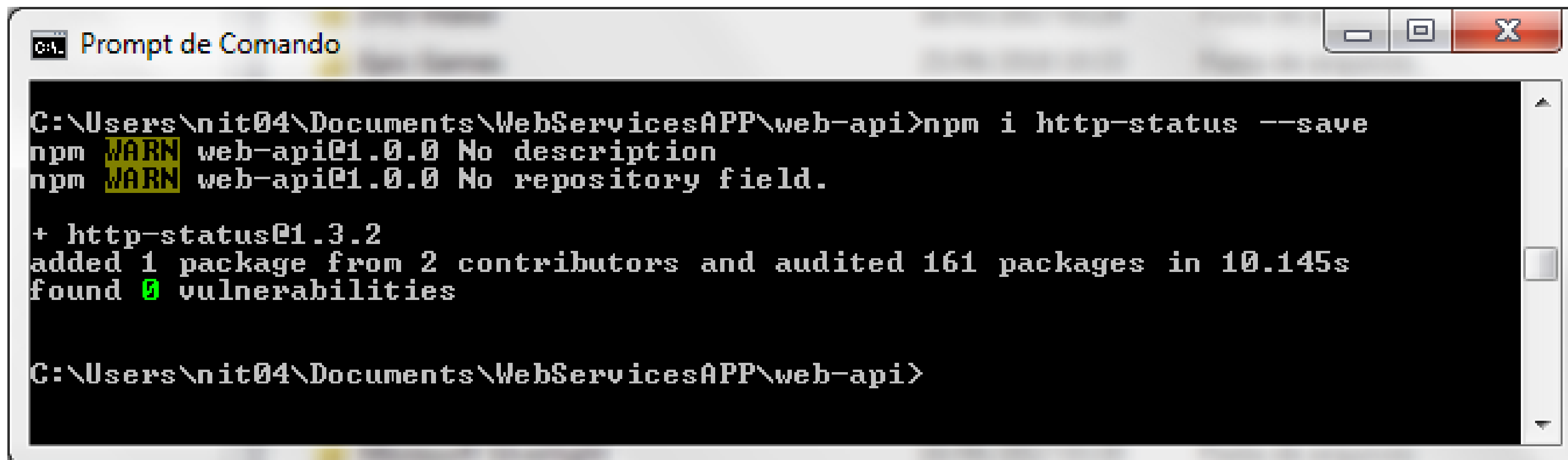


Figura 15: Instalação do http-satus no projeto web-api.

Com isso, nosso servidor está pronto para tratar as requisições e nós podemos criar, agora, as novas rotas do nosso servidor. Inicialmente, definiremos o retorno do servidor para o erro404. Para isso, editaremos o arquivo app.js, incluindo algumas novas linhas de código, conforme a figura 16.

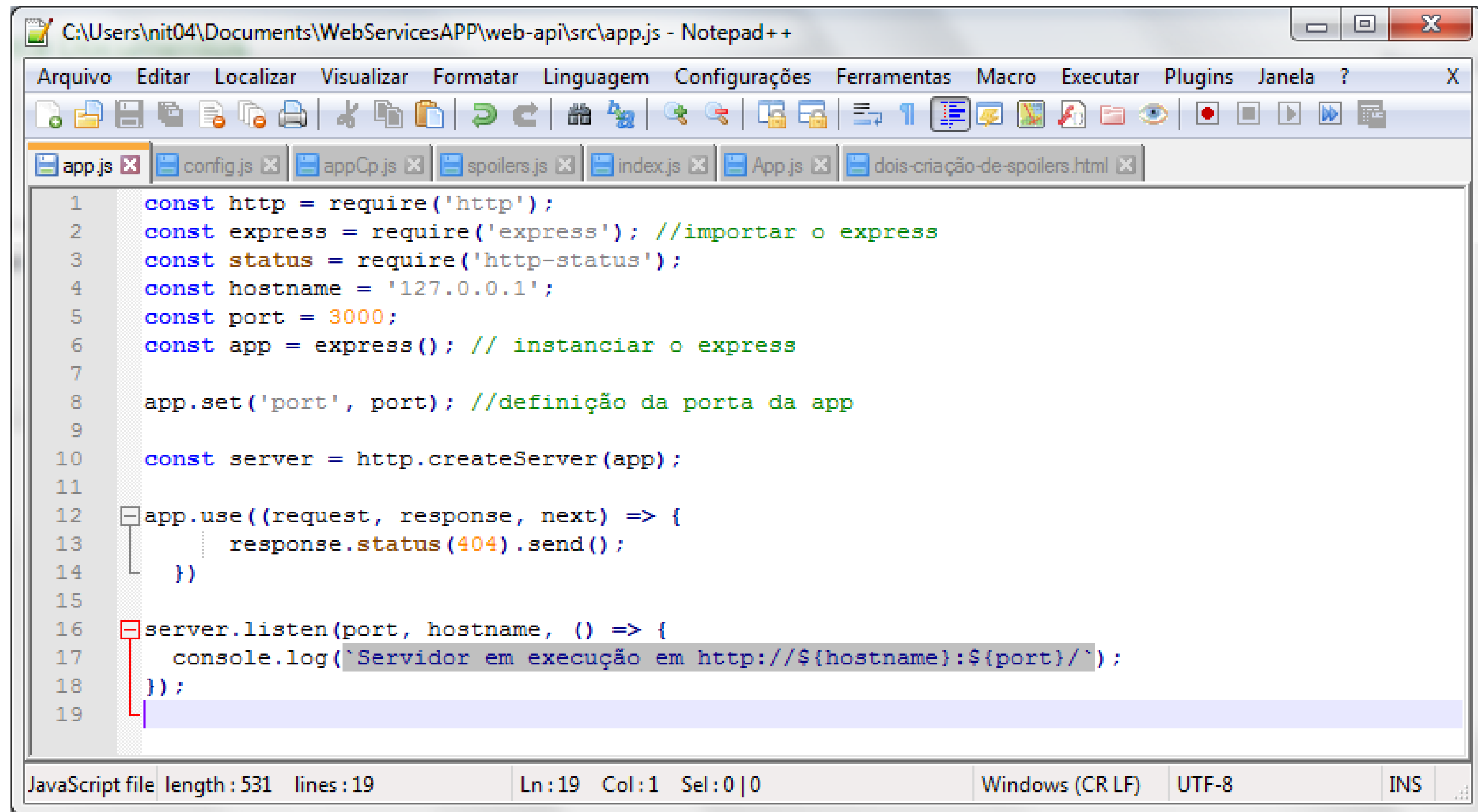


Figura 16: Código do arquivo app.js com express e http-status.

- Linha 2: declaramos a constante express e importamos o pacote do express.

- Linha 3: declaramos a constante status e importamos o pacote do http-status.

- Linha 6: declaramos a constante app e instanciamos o express vinculado a esta constante.

- Linha 8: Definimos a porta da rede do servidor, no caso porta 3000.

- Linha 10: Criamos a constante server para armazenar o objeto referente ao servidor da aplicação. Fazemos isso, por executar o método createServer(). Este método recebe como parâmetro a constante app que instanciou todas as funcionalidades do express.

- Linha 13 a 15: define o retorno do servidor para o erro404.

3.2 Conectando o Banco de Dados com Sequelize

Após a configuração do servidor HTTP com o express, o próximo passo é configurar e conectar o banco de dados na aplicação. Para isso, criaremos o diretório "/config" dentro do diretório "/src" da aplicação. Neste diretório, produziremos um arquivo chamado config.js com o seguinte código:

```
module.exports = {
    development: {
      database: {
        host: “localhost”,
        port: 3306,
        name: “jogo”,
```

```
    dialect: “mysql”,
    user: “root”,
    password: “”
  }
},
production: {
  database: {
    host: process.env.DB_HOST,
    port: process.env.DB_PORT
  }
}
};
```

Aqui, temos as configurações utilizadas em ambiente de desenvolvimento (development:), que, geralmente, é o computador utilizado pelo programador envolvido no projeto. As configurações do servidor estão na seção de produção (production:) que devem ser utilizadas no servidor de produção, ou seja, no local onde a aplicação será hospedada.

Com as configurações iniciais para conexão ao banco de dados, estabeleceremos essa conexão utilizando a biblioteca Sequelize, que permite fazer o mapeamento objeto-relacional entre a aplicação e o banco de dados. Para instalar o sequelize, interromperemos a execução do servidor e digitaremos o comando “npm i mysql2 sequelize --save” que fará o download e a instalação da biblioteca. Veja o exemplo da instalação, na Figura 17:

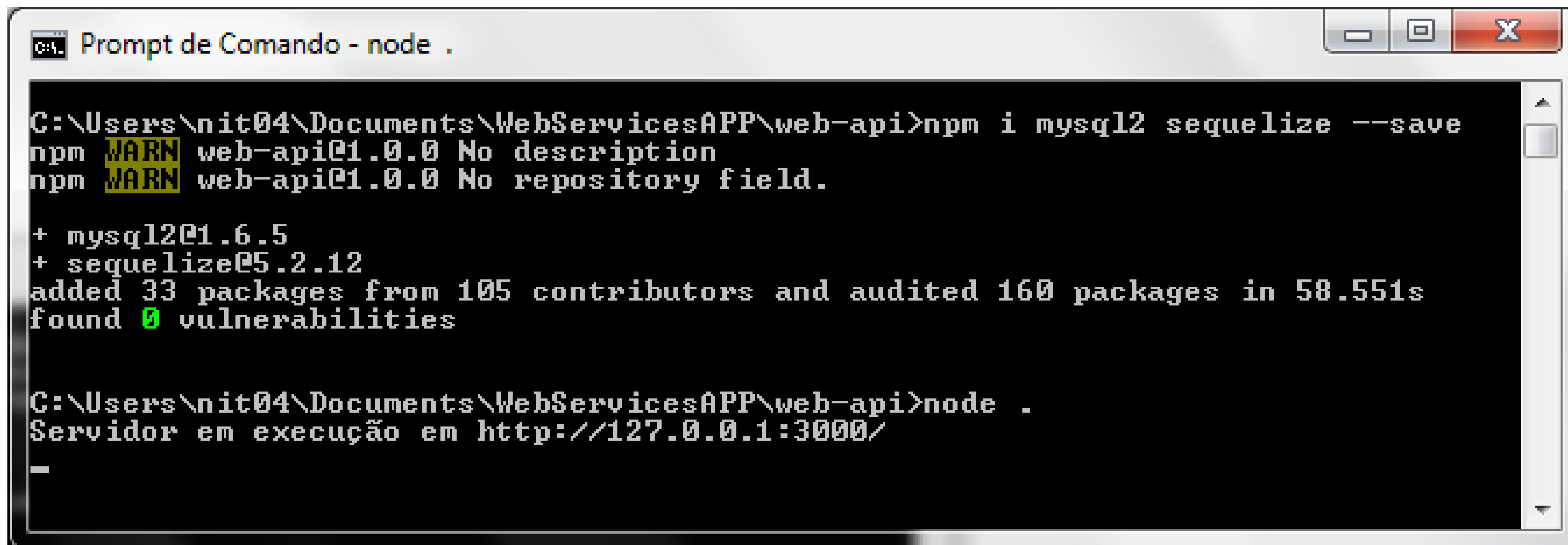


Figura 16: Código do arquivo app.js com express e http-status.

Após a instalação do sequelize, acrescentaremos um novo diretório com o nome database e, dentro deste diretório, incluiremos um arquivo chamado database.js. Esse arquivo fará a conexão com o banco de dados, utilizando as informações do arquivo config.js. Veja o código a seguir:

```
const Sequelize = require("sequelize");
const environment = process.env.NODE_ENV || "development";
const config = require("../config/config.js")[environment];
const sequelize = new Sequelize(
   config.database.name,
```

```
    config.database.user,
    config.database.password,
    {
      host: config.database.host,
      dialect: config.database.dialect
    }
);
module.exports = sequelize;
```

Com a criação do database, devemos agora criar o objeto da nossa aplicação. Este objeto será o principal dado da nossa aplicação. É com base nele que faremos as quatro funcionalidades básicas de um banco de dados. Estas funcionalidades são definidas no CRUD - acrônimo de Create, Read, Update e Delete, que correspondem às quatro operações básicas utilizadas em bases de dados relacionais.

No nosso exemplo, produziremos um objeto com os dados para criar um ranking dos melhores desempenhos de jogadores para algum jogo. Esse objeto será composto por três campos:

- id:INTEGER - campo com o id do jogador
- nome:STRING - campo nome do jogador
- score:INTEGER - campo com o score do jogador

Para isso, devemos inicialmente criar um novo diretório model, dentro da pasta src da aplicação. Neste diretório, criaremos um arquivo jogo.js que define os atributos da tabela no banco de dados, conforme o código a seguir:

```
const Sequelize = require("sequelize");
const sequelize = require("../database/database");
const Jogo = sequelize.define("Jogo", {
  id: {
    allowNull: false,
    primaryKey: true,
    type: Sequelize.INTEGER
  },
  nome: {
    allowNull: false,
    type: Sequelize.STRING(255),
    validate: {
      len: [2, 255]
    }
  },
  score: {
    allowNull: false,
    type: Sequelize.INTEGER
  }
 });
module.exports = Jogo;
```

Após a definição do model, devemos configurar o banco de dados. Para isso, utilize o banco de dados mySql e crie um banco de dados com o nome jogo. A título de ilustração, veja o exemplo apresentado na Figura 18.

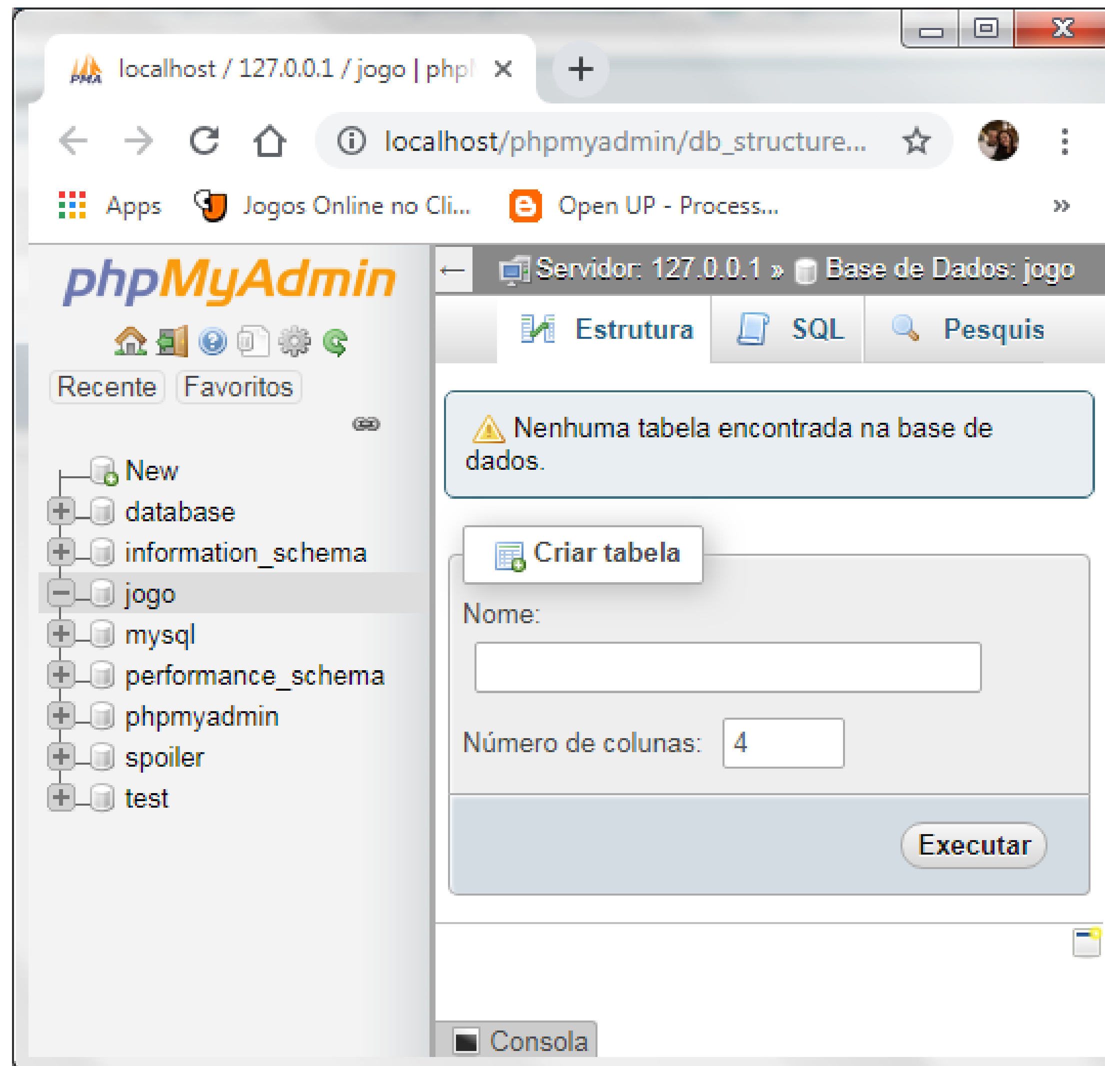


Figura 18: Banco de dados jogo criado no mySql.

Após a criação do banco de dados, vamos organizar nosso arquivo app.js incluindo o sequelize e a conexão ao banco de dados?! Observe a figura 19:

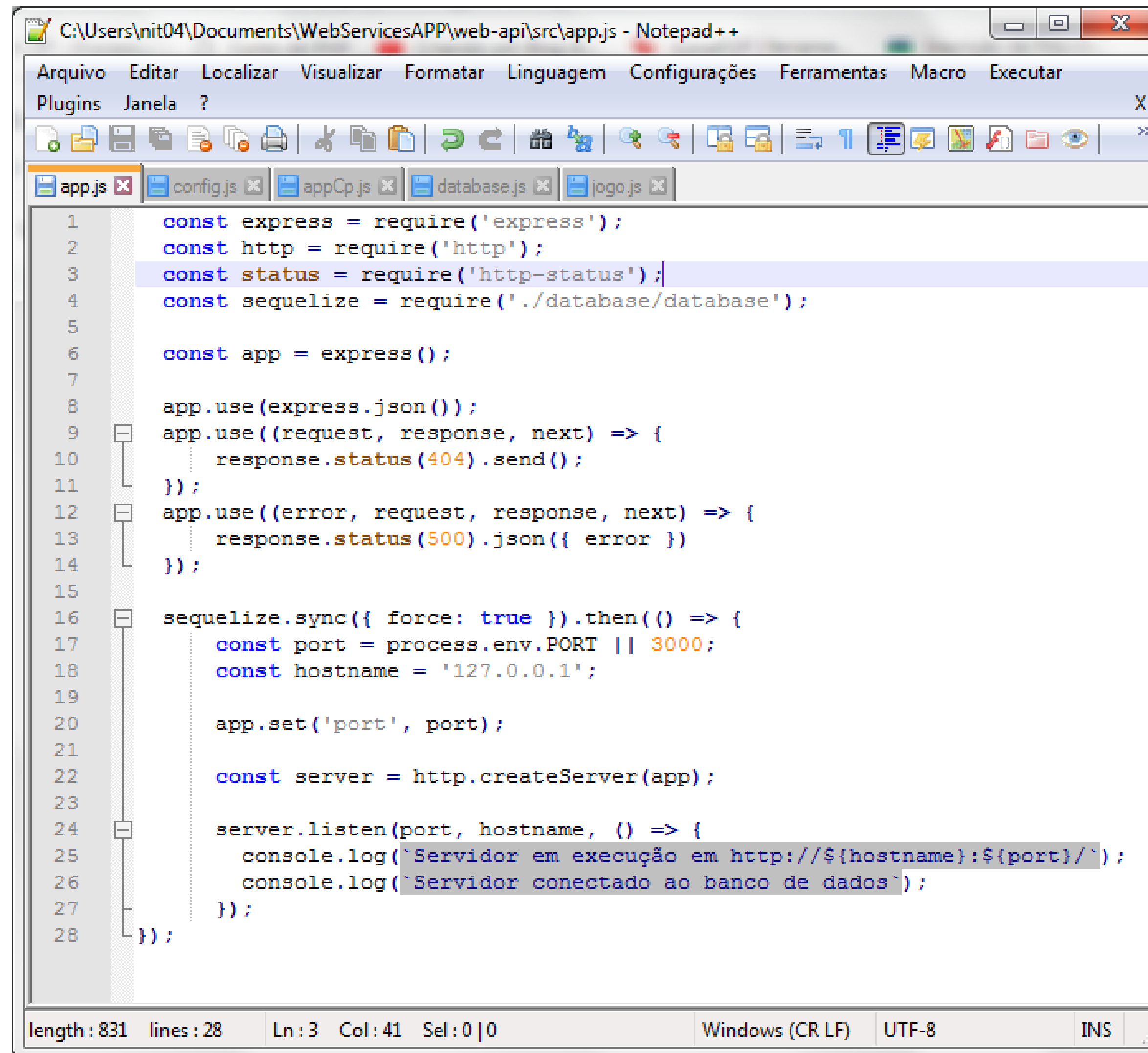


Figura 19: arquivo app.js com a conexão ao banco de dados.

- Linhas 2 a 4: declaramos as constantes dos pacotes que são importados.

- Linha 6: declaramos a variável app e instanciamos o express.

- Linhas 8 a 14: definimos as rotas padrões do express.

- Linhas 16 a 27: o sequelize cria a conexão com o banco de dados e inicia o servidor na porta indicada.

Com isso pronto, podemos executar nosso servidor com o comando “node .” e iniciaremos o servidor com uma conexão no banco de dados. Veja o resultado no prompt de comando, conforme a Figura 20.

A instrução comando “Executing <default>: SELECT 1+1 AS result” indica que a conexão ao banco de dados foi criada e que está funcionando.

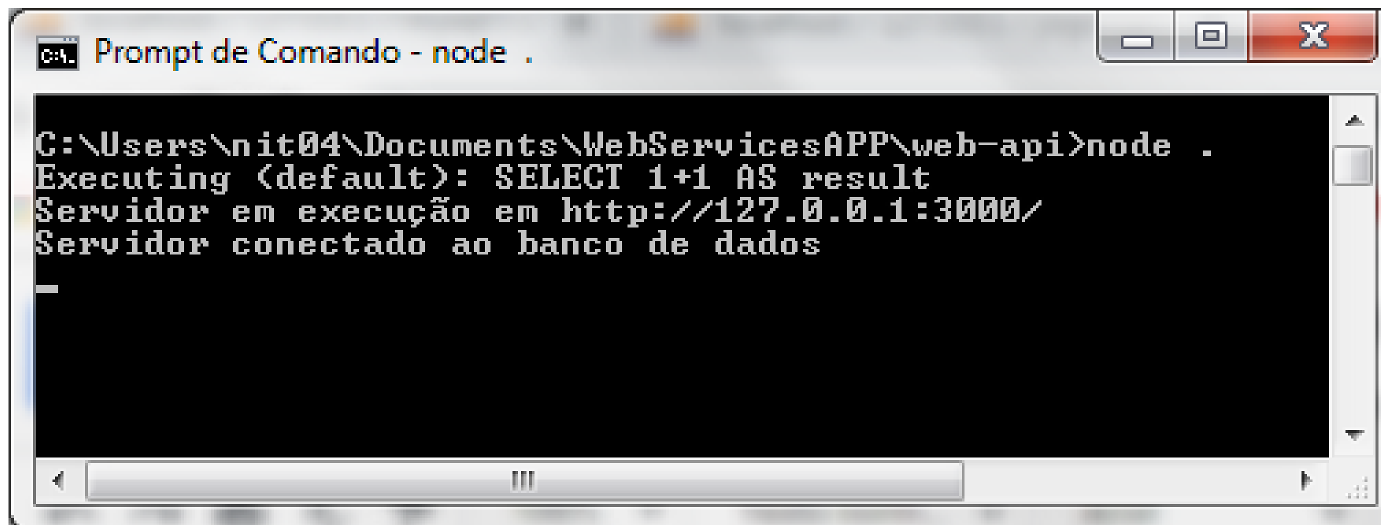
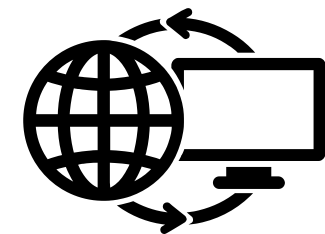


Figura 20: servidor do node.js executando e conectado ao banco de dados.

Resumo

Nesta unidade estudamos a criação de uma conexão do nosso servidor com o banco de dados mySql. Para isso, instalamos o pacote sequelize que permite fazer o mapeamento objeto relacional entre os objetos da aplicação e banco de dados. Ao final, temos um WebServices rodando e acessando o banco de dados. Na próxima unidade, prepararemos nosso servidor para efetuar um crud completo, permitindo que os dados de nossos jogos sejam persistidos no servidor.



Atividades de Revisão

1. Faça uma breve pesquisa e explique o que significa mapeamento objeto-relacional.

2. Explique o que é o sequelize e pra que ele é utilizado no node.js.

3. Crie um banco de dados jogo e configure a conexão com este banco de dados, conforme descrito nesta unidade.

Unidade 4

Desenvolvendo um jogo com WebServices e Banco de Dados

Nesta unidade iremos aplicar todos os conhecimento adquiridos durante nossa disciplina de WebServices aplicados à jogos. Para isso vamos desenvolver uma funcionalidade de jogo simples na aplicação de Back-end com acesso ao banco de dados.

Mas o que significa desenvolver uma funcionalidade de jogo simples na aplicação de Back-end? Como já estudamos ao longo do nosso curso e nos conceitos iniciais desta disciplina, para termos uma aplicação completa em um WebServices precisamos da camada de apresentação(Front-end) que recebe os dados dos usuários (requisições) e apresenta o resultado dos processamentos(respostas).

Entretanto, a camada de não faz parte do escopo da nossa disciplina, estes conceitos serão vistos em outras componentes do curso. Assim, iremos aplicar os nossos conhecimentos preparando um serviço que permite gerar um ranking dos jogadores de um determinado jogo. Este serviço será disponibilizado de forma genérica para que qualquer jogo(aplicação) possa utilizá-lo conforme prevê o conceito de WebServices. Ou seja, qualquer jogo novo ou já desenvolvido, poderá utilizar este serviço para gerar e mostrar o ranking dos melhores jogadores e resultados.

Lembre-se de interagir com o assuntos selecionados para essa unidade e de executar os exemplos sugeridos no decorrer do material. Boa leitura a todos.

Criando as Funcionalidades de CRUD no WebServices

Na unidade anterior, criamos o pacote model com um arquivo jogo.js que define os atributos da tabela no banco de dados. Após isso configuramos a conexão com o servidor com o banco de dados mySql utilizando a biblioteca sequelize que permite fazer o mapeamento objeto relacional entre os objetos da aplicação e banco de dados.

Assim, o nosso próximo passo é a implementação das funcionalidades do CRUD (Create, Read, Update e Delete), que são as operações básicas utilizadas em bases de dados relacionais. Para isso, iremos criar os controllers da aplicação.

Inicialmente devemos criar um novo diretório com o nome controller dentro do diretório src, nesta nova pasta vamos criar um novo arquivo com o nome jogo.js que irá conter as funcionalidades do CRUD. Veja o código da primeira funcionalidade que permite consultar um registro do nosso ranking conforme a Figura 21.

```
const Jogo = require('../model/jogo')
const Status = require('http-status')

exports.buscarUm = (request, response, next) => {
    const id = request.params.id

    Jogo.findById(id).then((jogo) => {
        if (jogo) {
            response.send(jogo)
        } else {
            response.status(Status.NOT_FOUND).send()
        }
    }).catch((error) => next(error))
}
```

Figura 21: Arquivo controller/jogo.js com a funcionalidade buscarUm() registro do ranking.

- Linha 1: declaração da constante jogo que instancia o objeto jogo do pacote model.

- Linha 2: declaração da constante Status do pacote http-status para tratamento de erros.

- Linha 4: comando exports que torna a funcionalidade buscarUm() pública.

- Linha 5: declaração da constante id que recebe o valor de consulta passado como parâmetro no request.

- Linha 7: Chamamos o método findById(id) que consulta o registro correspondente ao id informado e retorna um objeto jogo.

- Linha 8: verificamos se o objeto jogo retornado existe, ou seja, se o objeto correspondente ao id informado foi encontrado. Se verdadeiro: retorna objeto jogo na linha 9. Senão retorna Status.NOT_FOUND na linha 11.

- Linha 9: retorna objeto jogo no response.

- Linha 11: retorna Status.NOT_FOUND no response.

- Linha 11: retorna um erro ocorrido durante a execução do método findById(id).

Após a funcionalidade que permite consultar um registro do nosso ranking vamos implementar outra funcionalidade de consulta, mas agora iremos retornar todos os registros em um lista de ocorrências do objeto jogo. Veja o código do arquivo controller/jogo.js com a essa nova funcionalidade conforme a Figura 22.

```
}

exports.buscarTodos = (request, response, next) => {
    let limite = parseInt(request.query.limite || 0)
    let pagina = parseInt(request.query.pagina || 0)

    if (!Number.isInteger(limite) || !Number.isInteger(pagina)) {
        response.status(Status.BAD_REQUEST).send()
    }

    const ITENS_POR_PAGINA = 10

    limite = limite > ITENS_POR_PAGINA || limite <= 0 ? ITENS_POR_PAGINA : limite
    pagina = pagina <= 0 ? 0 : pagina * limite

    Jogo.findAll({ limit: limite, offset: pagina }).then((jogo) => {
        if (jogo && jogo.length) {
            response.send(jogo)
        } else {
            response.status(Status.NOT_FOUND).send()
        }
    }).catch((error) => next(error))
}
```

Figura 22: Arquivo controller/jogo.js com a funcionalidade buscarTodos() registro do ranking.

- Linha 16: comando exports que torna a funcionalidade budcarTodos() pública.

- Linha 17: declaração da variável limite que recebe o valor passado como parâmetro no request.

- Linha 18: declaração da variável pagina que recebe o valor passado como parâmetro no request.

- Linhas 20 a 22: verifica se os valores recebidos como parâmetros são válidos, em caso falso retorna o erro Status.BAD_REQUEST.

- Linha 24: declaração da constante ITENS_POR_PAGINA que recebe o valor 10.

- Linhas 26 e 27: valida os valores do limite e da página com base no valor de ITENS_POR_PAGINA.

- Linha 29: Chamamos o método findAll( ) que consulta todos os registros e retorna uma lista de objetos jogo.

- Linha 30: verificamos se a lista de objetos jogo retornado existe. Se verdadeiro: retorna a lista de objetos jogo na linha 31. Senão retorna Status.NOT_FOUND na linha 33.

- Linha 31: retorna a lista de objetos jogo no response.

- Linha 33: retorna Status.NOT_FOUND no response.

- Linha 35: retorna um erro ocorrido durante a execução do método findAll( ).

Após criarmos as funcionalidades buscarUm() e buscarTodos() finalizamos as funcionalidades de READ do nosso controller. O próximo passo será criarmos a funcionalidade de CREATE. Para isso, iremos implementar uma nova funcionalidade criar() que permite a inclusão de novos registros no nossos banco de dados. Os valores passados como parâmetros são o nome do jogador e o seu score, lembrando que o número de id é gerado automaticamente pelo banco de dados. Veja o código do arquivo controller/jogo.js com a funcionalidade criar() conforme a Figura 23.

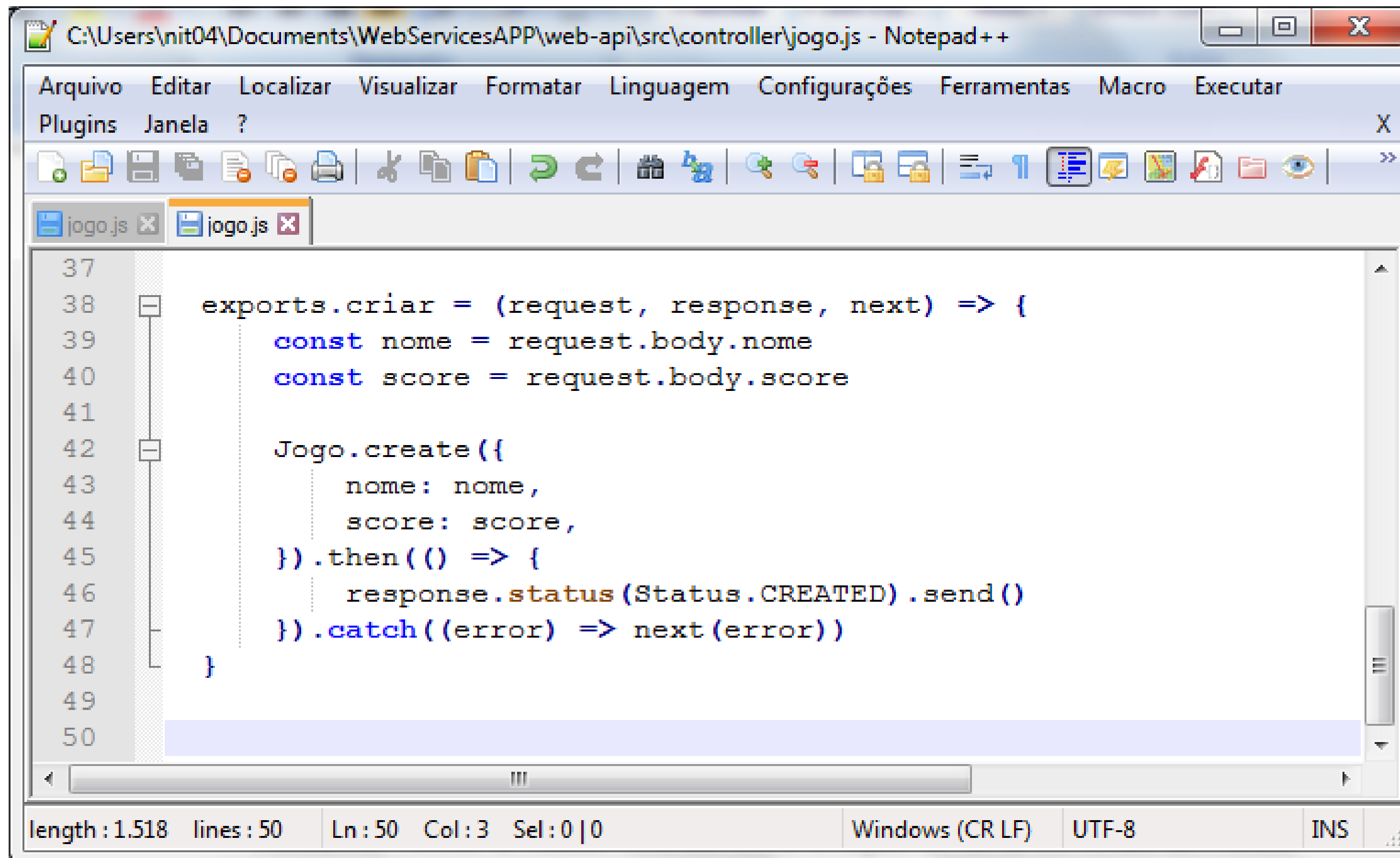


Figura 23: Arquivo controller/jogo.js com a funcionalidade criar() registro no ranking.

- Linha 38: comando exports que torna a funcionalidade criar() pública.

- Linha 39: declaração da constante nome que recebe o valor passado como parâmetro no request.

- Linha 40: declaração da constante score que recebe o valor passado como parâmetro no request.

- Linha 42: Chamamos o método create( ) que cria um objeto jogo e armazena no banco de dados.

- Linha 46: retorna Status. CREATED no response.

- Linha 47: retorna um erro ocorrido durante a execução do método create( ).

A próxima funcionalidade a ser criada é o atualizar, que permite alterar o secore de um jogador que já esteja na nossa lista de resultados, para isso recebemos o id, o nome e o novo score o do jogador. Estes dados serão atualizados no banco de dados. Podemos perceber que nesta funcionalidade estamos fazendo um mistura dos métodos criar() e buscar(), veja na Figura 24.

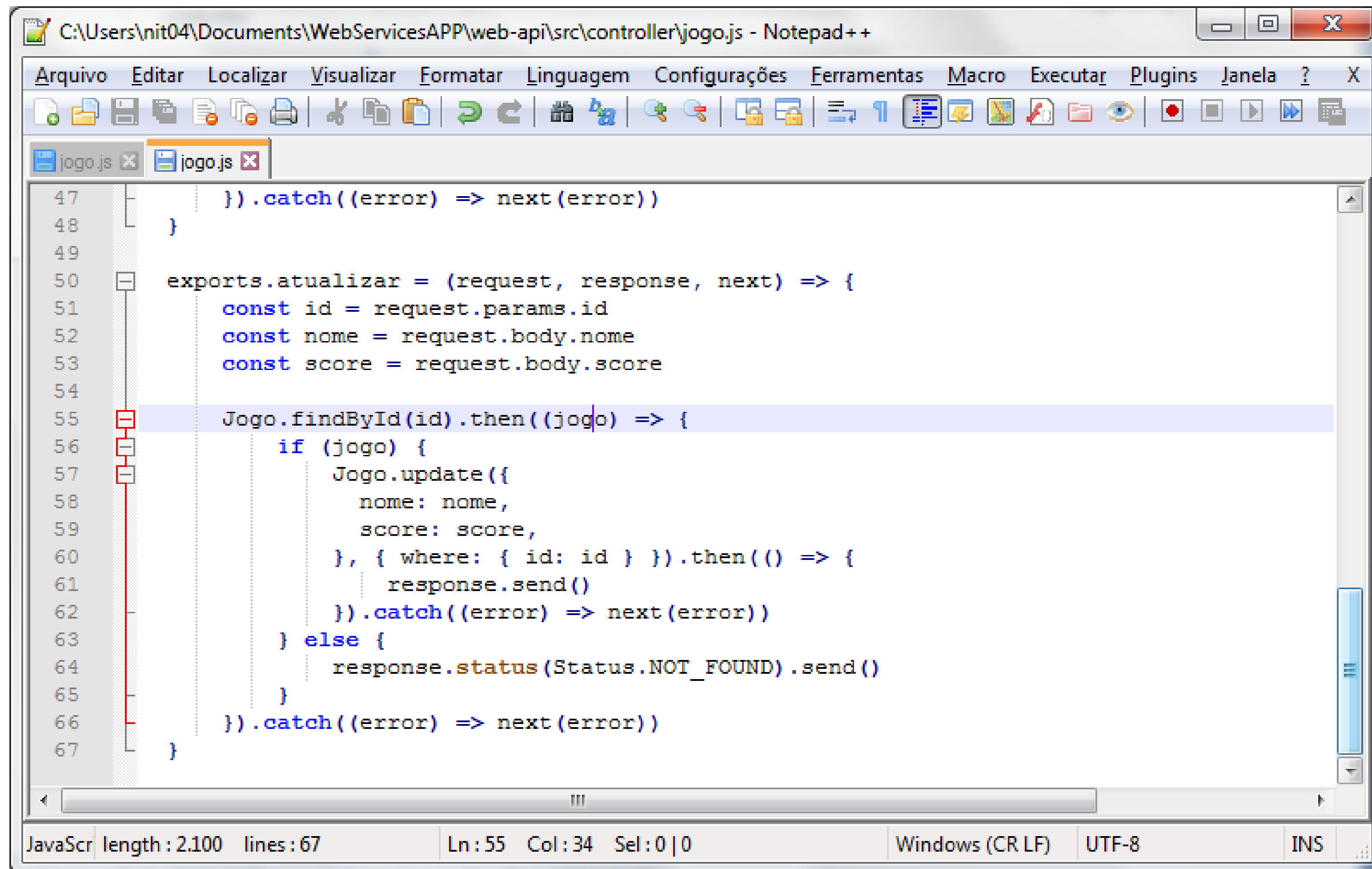


Figura 24: Arquivo controller/jogo.js com a funcionalidade atualizar() registro no ranking.

- Linha 50: comando exports que torna a funcionalidade atualizar() pública.

- Linha 51: declaração da constante id que recebe o valor de consulta passado como parâmetro no request.

- Linha 52: declaração da constante nome que recebe o valor de consulta passado como parâmetro no request.

- Linha 53: declaração da constante score que recebe o valor de consulta passado como parâmetro no request.

- Linha 55: Chamamos o método findById(id) que consulta o registro correspondente ao id informado e retorna um objeto jogo.

- Linha 56: verificamos se o objeto jogo retornado existe, ou seja, se o objeto correspondente ao id informado foi encontrado. Se verdadeiro: chama o método update() na linha 57. Senão retorna Status.NOT_FOUND na linha 64.

- Linha 57: chamamos o método update() para o objeto jogo retornado da consulta do método findById(id).

- Linha 58: atualizamos o valor o atributo nome no objeto jogo.

- Linha 59: atualizamos o score o atributo nome no objeto jogo.

- Linha 60: quando id=id atualiza o objeto jogo no banco de dados.

- Linha 61: retorna o resultado da atualização para o response.

- Linha 62: retorna um erro ocorrido durante a execução do método update (id).

- Linha 64: retorna Status.NOT_FOUND no response.

- Linha 66: retorna um erro ocorrido durante a execução do método findById(id).

Após a criarmos o método atulizar(), ficou faltando somente o método excluir() para completarmos o nosso CRUD dos scores. Semelhante ao método atualizar(), iremos efetuar um busca pelo id do score a ser excluído, para então efetuar a exclusão, veja o código na Figura 25.

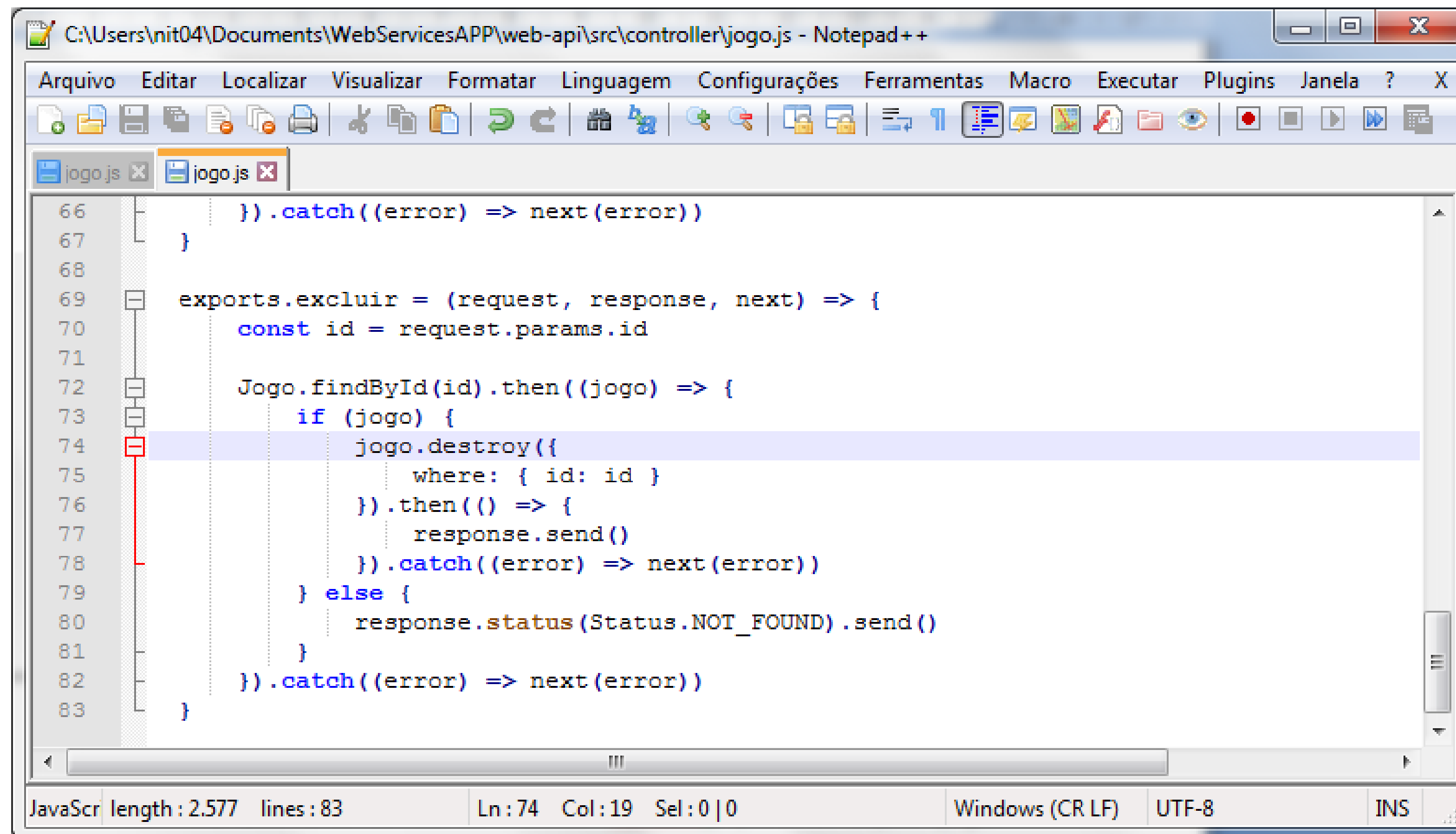


Figura 25: Arquivo controller/jogo.js com a funcionalidade excluir() registro no ranking.

- Linha 69: comando exports que torna a funcionalidade excluir() pública.

- Linha 70: declaração da constante id que recebe o valor de consulta passado como parâmetro no request.

- Linha 72: chamamos o método findById(id) que consulta o registro correspondente ao id informado e retorna um objeto jogo.

- Linha 73: verificamos se o objeto jogo retornado existe, ou seja, se o objeto correspondente ao id informado foi encontrado. Se verdadeiro: chama o método destroy() na linha 74. Senão retorna Status.NOT_FOUND na linha 80.

- Linhas 74 a 77: chamamos o método destroy () para o objeto jogo retornado da consulta do método findById(id). Quando id=id exclui o objeto jogo no banco de dados, retornando o resultado da exclusão para o response na linha 77.

- Linha 78: retorna um erro ocorrido durante a execução do método destroy (id).

- Linhas 80: retorna Status.NOT_FOUND no response.

- Linhas 82: retorna um erro ocorrido durante a execução do método findById(id).

4.1 Criando as rotas do Controller

Após criamos todos os métodos do controller, o nosso próximo passo é criar as rotas para as aplicações que irão utilizar este métodos, estas rotas podem ser consideradas “conectores” que permitem as aplicações, ou seja, os jogos, acessem os método definidos no nosso WebService.

Inicialmente devemos criar um novo diretório dentro do src com o nome routes, dentro dessa pasta criaremos o arquivo jogos.js. Nesse arquivo iremos criar as rotas para que vinculam os métodos criados no controller/jogo.js com os métodos do protocolo HTTP (get, post, put e delete), conforme a Figura 26 a seguir:

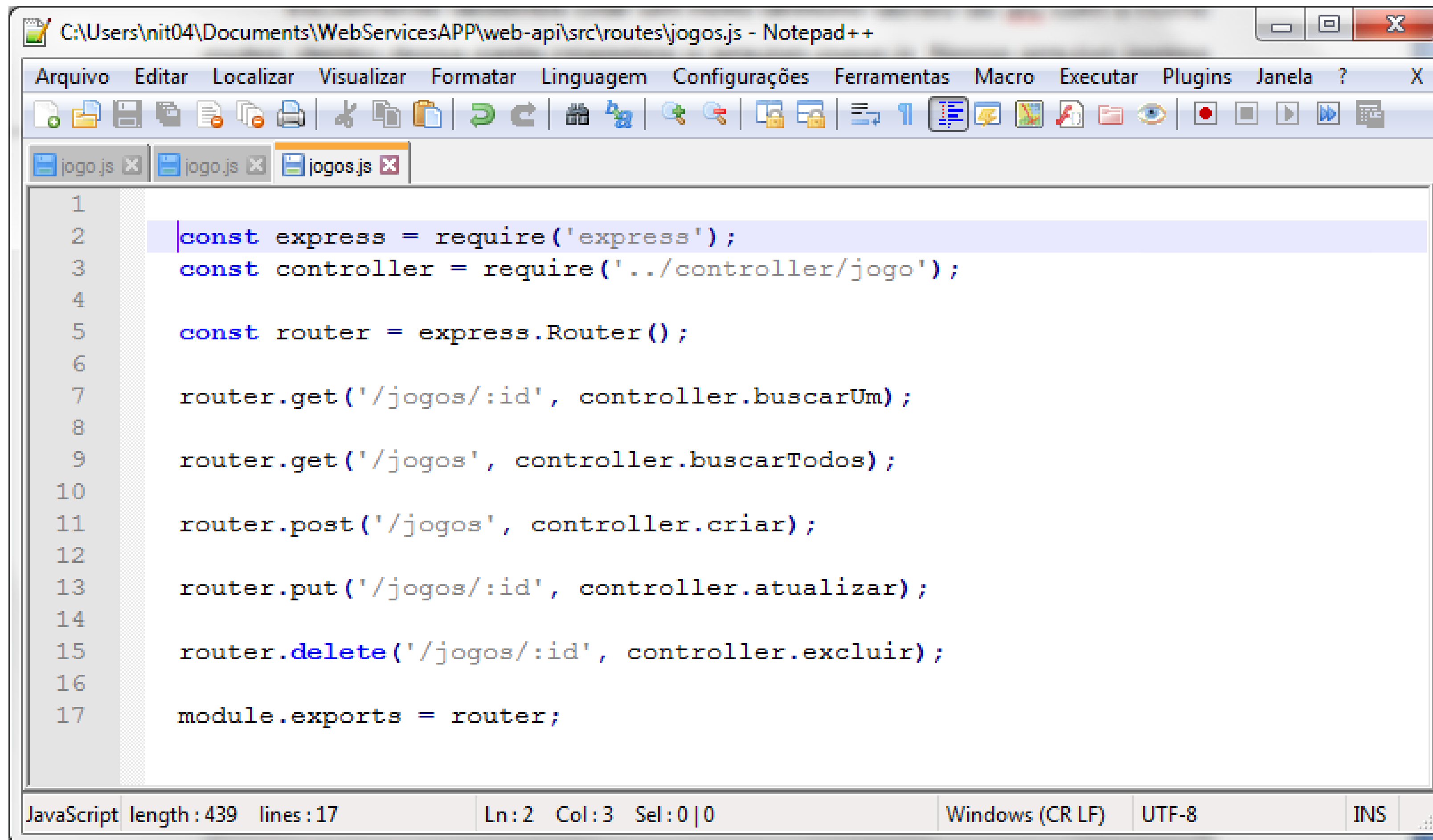


Figura 26: Arquivo routes/jogos.js com as rotas para os métodos do controller/jogo.js.

- Linha 1: cria a constante express importando o pacote express.

- Linha 3: cria a constante controller importando o pacote controller/jogo.js.

- Linha 5: cria a constante router instanciando um objeto Router().

- Linha 7: cria a rota do método get com o método buscarUm(id) do controller.

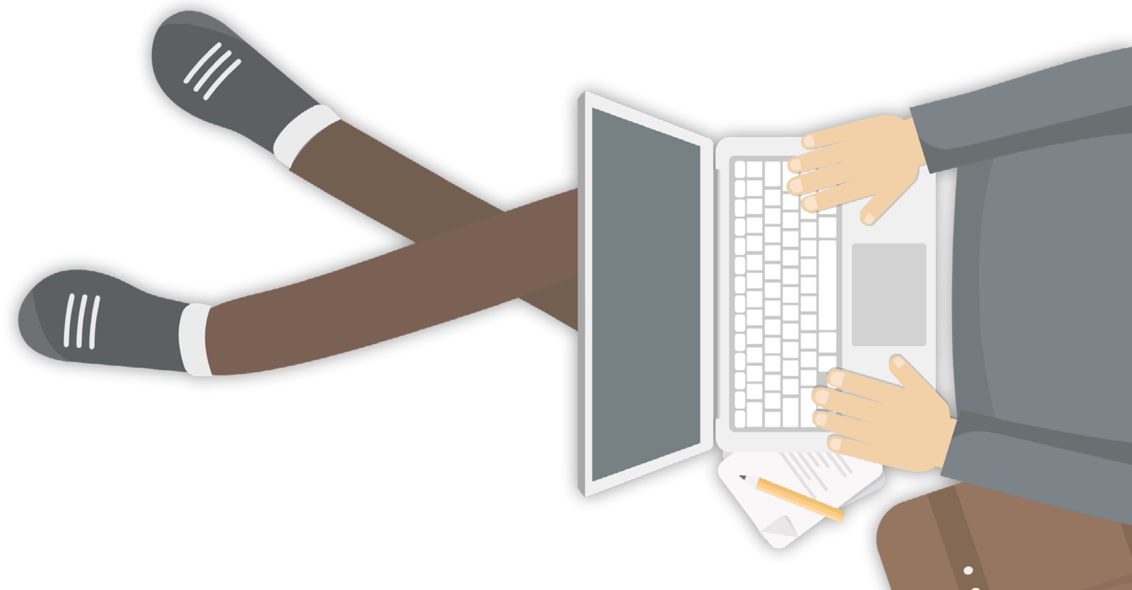
- Linha 9: cria a rota do método get com o método buscarTodos() do controller.

- Linha 11: cria a rota do método post com o método criar() do controller.

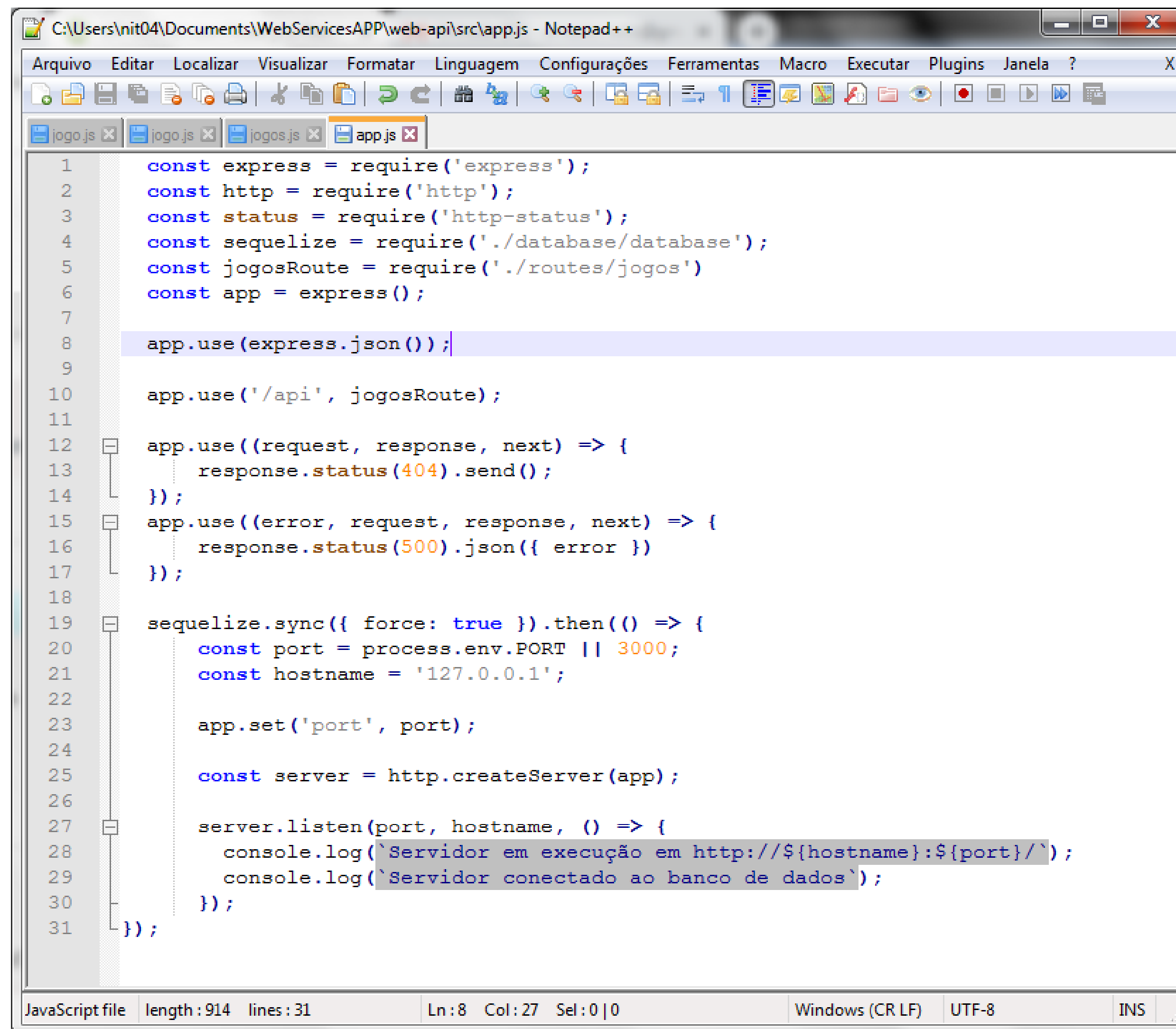
- Linha 13: cria a rota do método put com o método atualizar() do controller.

- Linha 15: cria a rota do método delete com o método excluir() do controller.

- Linha 17: comando exports que torna o objeto Router() público.



Após a definição das rotas precisamos informar no arquivo app.js onde o express irá encontrar essas definições de rotas. Para isso devemos incluir o pacote definido no diretório routes o comando app.use com o caminho para este diretório, conforme podemos ver na Figura 27.



```
const express = require('express');
const http = require('http');
const status = require('http-status');
const sequelize = require('./database/database');
const jogosRoute = require('./routes/jogos')
const app = express();

app.use(express.json());

app.use('/api', jogosRoute);

app.use((request, response, next) => {
    response.status(404).send();
});
app.use((error, request, response, next) => {
    response.status(500).json({ error })
});

sequelize.sync({ force: true }).then(() => {
    const port = process.env.PORT || 3000;
    const hostname = '127.0.0.1';

    app.set('port', port);

    const server = http.createServer(app);

    server.listen(port, hostname, () => {
      console.log(`Servidor em execução em http://${hostname}:${port}/`);
      console.log(`Servidor conectado ao banco de dados`);
    });
});
```

Figura 27: Arquivo app.js com a inclusão das rotas para os métodos do controller/jogo.js.

- Linha 5: cria a constante jogosRoute importando o pacote routes/jogos.

- Linha 10: definimos as rotas do express para os métodos do controller/jogo.js conforme a constante jogosRoute.

- Linha 19: cria a tabela Jogos quando o parâmetro “force: true”, apagando todos os registros previamente existentes. O parâmetro “force: false” não apaga a tabela Jogos se ela já existir.

4.2 Iniciando o servidor com as rotas

Após concluída as configurações de rotas podemos então iniciar o nosso WebService com os serviços e a conexão ao bando de dados. Para isso, devemos abrir o Prompt de comandos do windows e executamos o comando “node .”, que irá criar a tabela definida no arquivo model/jogo.js na base de dados jogo no MySql, além de iniciar o serviço do servidor e estabelecer a conexão com o bando de dados. A Figura 28 mostra o resultado deste comando.

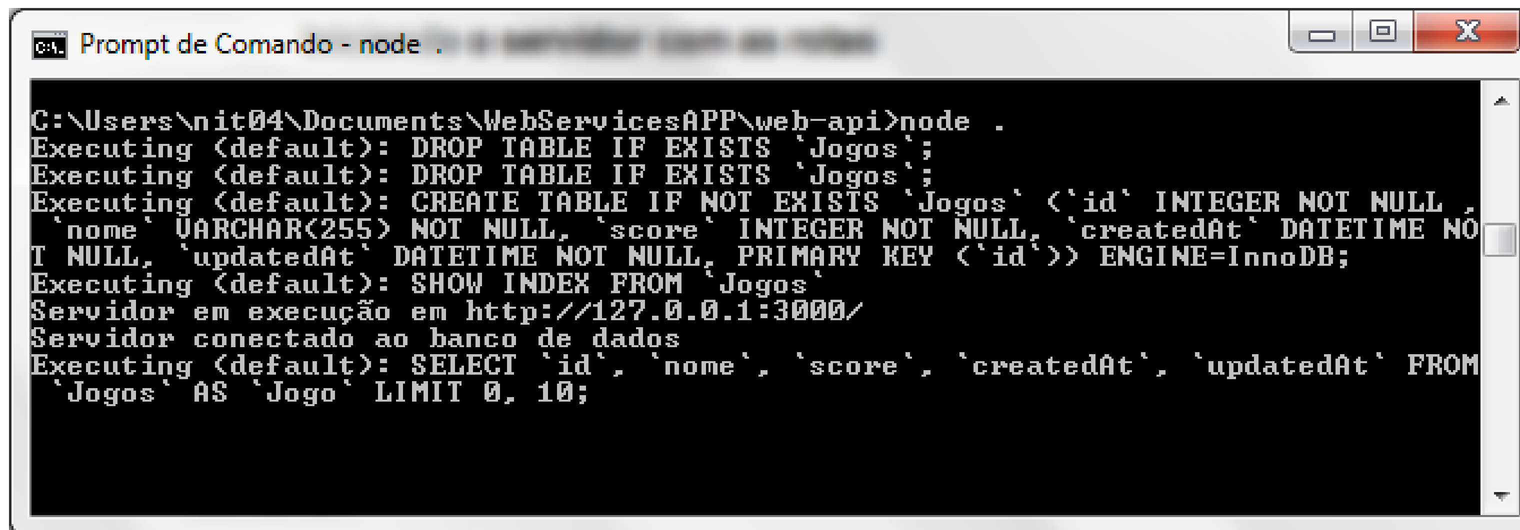


Figura 28: Inicialização do servidor com a conexão ao bando de dados e a criação da tabela Jogos.

Com o servidor rodando podemos inserir manualmente dois registros na tabela Jogos para verificarmos a consulta e a rota padrão definida pelo método controller/buscarTodos(), faça isso utilizando a interface gráfica do MySql conforme mostra a Figura 29.

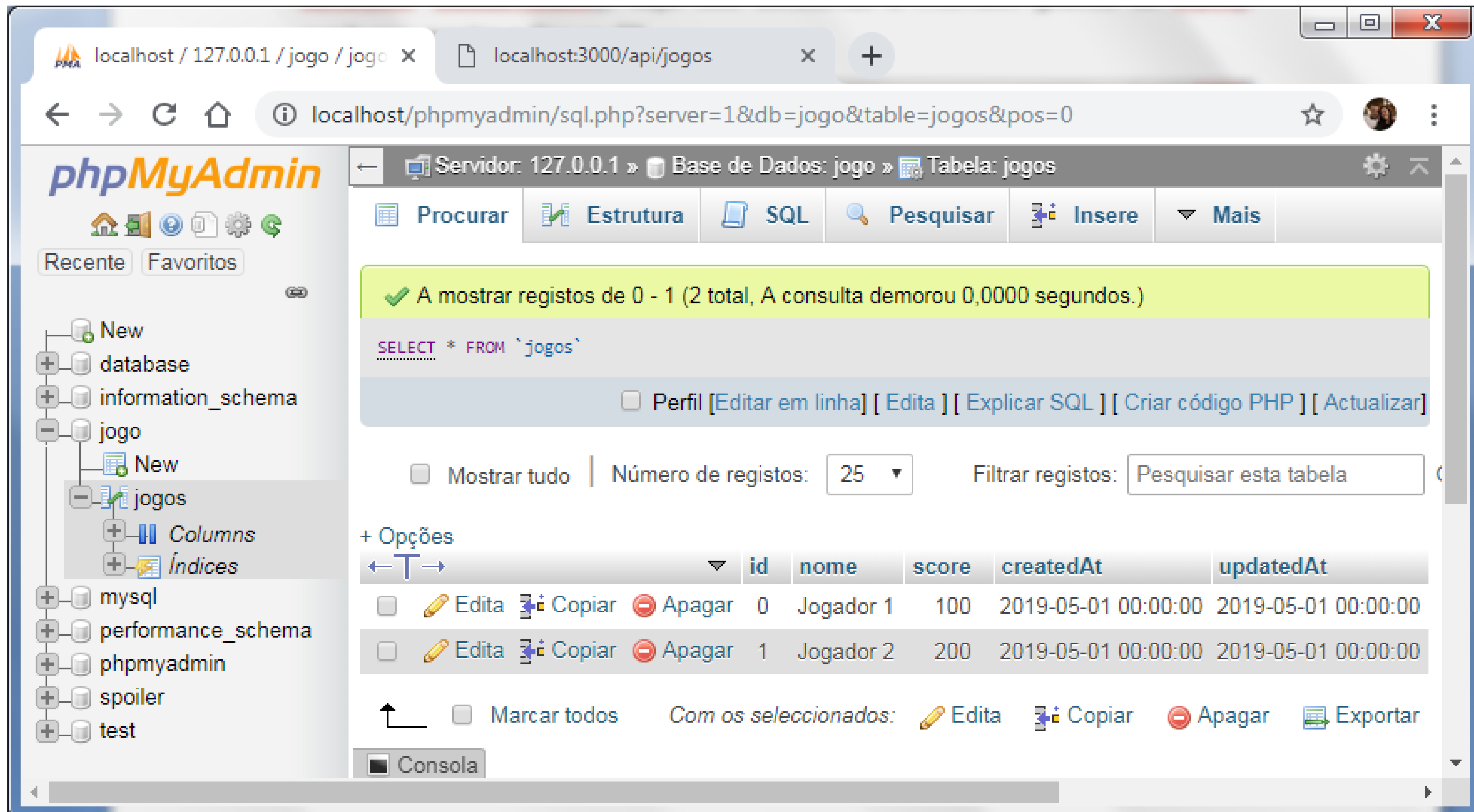


Figura 29: Tabela jogos com os registros de Jogador 1 e Jogador 2 inseridos.

Com os dados inseridos podemos testar o método controller/buscarTodos(), abrindo o navegador e digitando a URL - http://localhost:3000/api/jogos. Que retorna uma lista com todos os registros encontrados na tabela jogos, conforme mostra a Figura 30.

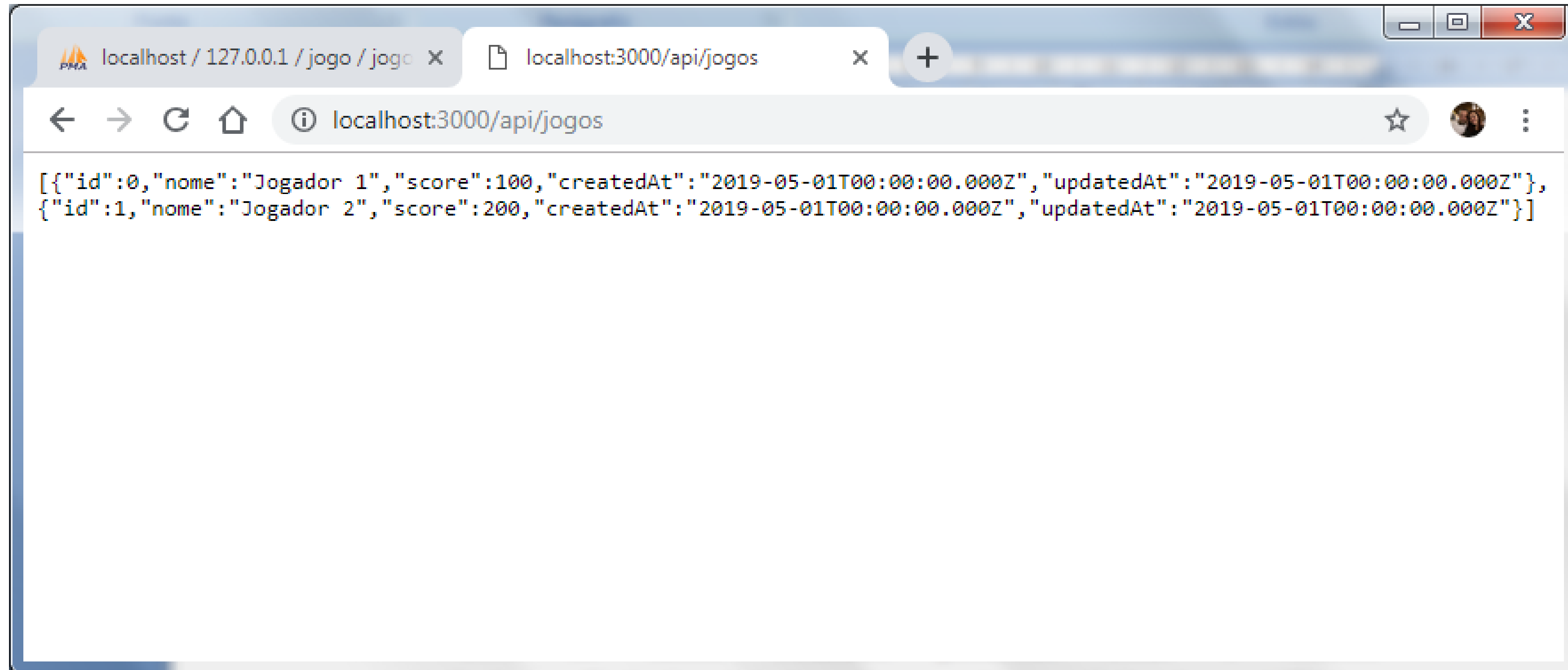


Figura 30: Resultado da consulta utilizando o método controller/buscarTodos().

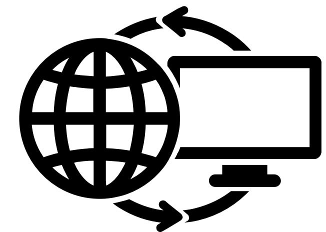
Com isso finalizamos a instalação e configuração de um WebServices para Jogos implementando a camada de back-end para nossas aplicações. Lembrando que para executarmos os demais métodos definidos nas rotas do nosso servidor precisamos da camada de front-end que deve receber os parâmetros do jogo e chamar os métodos correspondentes. Bem como, devemos desenvolver paginas que tratam as respostas retornadas do servidor após as consultas no banco de dados. Mas isso é assunto para outras disciplinas do nosso curso.

Ao chegarmos no final da nossa disciplina desenvolvemos um serviço Web para criação do módulo de controle de um ranking para Jogos On-line. Para isso iniciamos compreendendo os conceitos básicos de WebServices e descrevemos as vantagens e aplicabilidade dos conceitos de WebServices.

Após a parte teórica e introdutória partimos para a prática com a instalação de um WebServices utilizando REST, NodeJS e Javascritpt/TypeScript. Com o ambiente instalado, integramos o nosso Web-Services com Bancos de Dados e desenvolvemos um serviço que permite gerenciar um ranking para Jogos On-line.

Resumo

Nesta unidade prepararemos nosso servidor para efetuar um CRUD completo, permitindo que os dados de nossos jogos sejam persistidos no servidor. Para isso definimos o módulo controller e seus métodos que foram vinculados aos métodos do protocolo HTTP (GET, POST, PUT e DELETE) para prover os serviços.



Atividades de Revisão

1. Faça uma breve pesquisa e explique o que significa um CRUD

2. Faça uma breve pesquisa e explique o que fazem os métodos do protocolo HTTP (GET, POST, PUT e DELETE).

3. Explique o que são as rotas de um WebServices.

4. Explique como os métodos do módulo controller são vinculados aos métodos do protocolo HTTP.

5. Explique porque um WebServices implementa a camada de back-end e não a camada de front-end.

Referências

DEVMEDIA. WebServices: Tutorial sobre WebServices. Disponível em: <https://www.devmedia.com.br/web-services/2873>. Acesso em: fev 2019.

Nodejs.org. About Node.js. Disponível em: <https://nodejs.org/en/about/>. Acesso em: fev 2019.

W3C - World Wide Web Consortium. The Web Services Resource Access Working Group, 2011. Disponível em: <https://www.w3.org/2002/ws/ra/>. Acesso em: fev 2019.