

Desenvolvimento de Web APIs com Java

Ivan Salvadori

Desenvolvimento de Web APIs com Java

Ivan Salvadori



Esse é um livro Leanpub. A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. Publicação Lean é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

© 2016 - 2018 Ivan Salvadori

Conteúdo

Sobre Este Livro	i
Público Alvo	i
Pré-requisitos	i
Recursos	i
Sobre o Autor	i
Agradecimentos	i
Introdução	1
Web APIs	1
Princípios Arquiteturais REST	3
Jersey	4
Spring Boot	5
Projeto	9
Visão Geral	9
Modelagem do Domínio da Aplicação	9
Integração de Dados	10
Ferramentas Utilizadas	11
Configuração Inicial	11
Contratos	15
Configuração do Banco de Dados	16
Implementação das Funcionalidades	19
Cadastramento	19
Consulta a Todos os Contatos	29
Consulta a um Contato Específico	32
Alteração e Remoção	36
Tratamento de Exceções	43
Criação de Exceções de Negócio	43
Implementação de Providers	46
Contato Não Encontrado	47
Aplicação Cliente da Web API	51
Visão Geral	51

CONTEÚDO

Listagem dos Contatos	52
O Problema de Cross Origin Request	55
Cadastro	57
Consulta	60
Alteração	62
Construção e Implantação do Projeto	67
Próximos Passos	71

Sobre Este Livro

Público Alvo

Este livro é destinado a estudantes de programação interessados em desenvolver sistemas para a Web. Este livro também pode ser interessante para programadores experientes que buscam atualizar seus conhecimentos sobre o desenvolvimento de Web APIs utilizando a linguagem de programação Java.

Pré-requisitos

Para acompanhar adequadamente os assuntos abordados neste livro, recomenda-se que o leitor possua conhecimentos básicos de programação na linguagem Java, de gerenciamento de banco de dados, de modelo cliente/servidor, além de conhecer o protocolo HTTP.

Recursos

O código fonte dos projetos está disponível em <https://bitbucket.org/salvadori/livro-java-web-apis>.

Sobre o Autor

Ivan Salvadori é bacharel (2009), mestre (2015) e doutorando em ciência da computação pela Universidade Federal de Santa Catarina. É membro do Laboratório de Pesquisas em Sistemas Distribuídos (LAPESD-UFSC¹). Atua na área de sistemas distribuídos, com foco em Web services semânticos. Atualmente está pesquisando mecanismos de composição para arquitetura de Microservices.

Agradecimentos

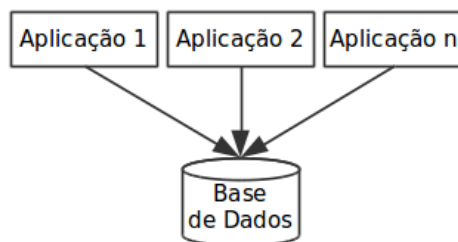
Agradeço carinhosamente a contribuição de José Gilson da Rosa na revisão de texto deste livro.

¹<http://lapesd.inf.ufsc.br/>

Introdução

Web APIs

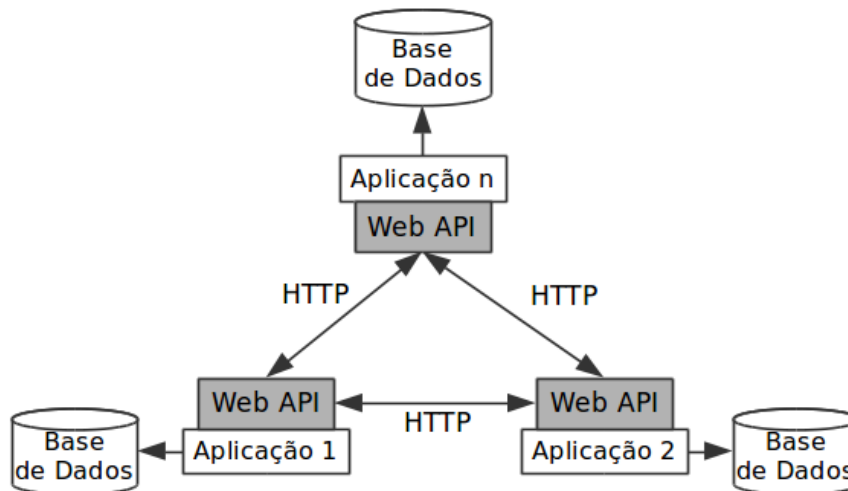
Organizações necessitam interligar sistemas e trocar informações internamente e também com outras organizações. Uma solução simples e muito utilizada para este tipo de integração é através do compartilhamento de banco de dados, onde tabelas são criadas para armazenar e compartilhar os dados dos sistemas. Esta forma de integração é relativamente simples e rápida de ser implementada, porém apresenta algumas desvantagens. Com a evolução dos sistemas, é inevitável que ocorram alterações (estruturais ou de conteúdo) nas bases de dados. Como diversas aplicações utilizam tabelas em comum, uma alteração pontual no banco de dados pode afetar diversas aplicações, dificultando a evolução e manutenção dos sistemas integrados.



Integração de aplicações através de banco de dados

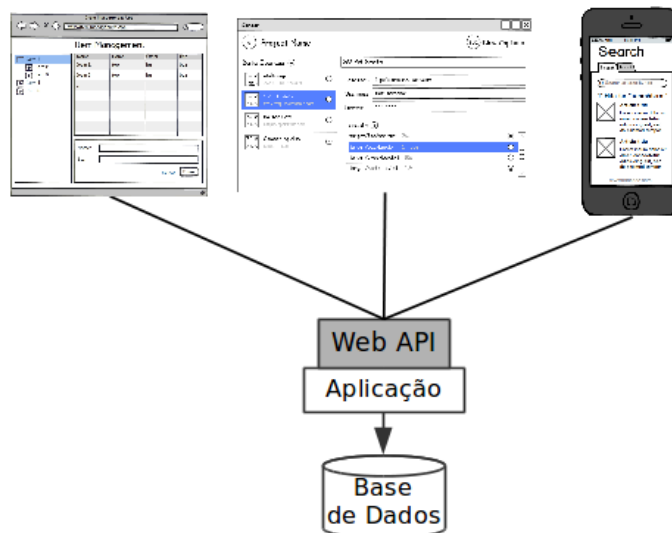
Outra alternativa é realizar a integração através de Web APIs, que disponibilizam as funcionalidades das aplicações em rede local ou na Web. A principal diferença entre Web APIs e aplicações Web tradicionais é o usuário. Aplicações Web tradicionais são manipuladas diretamente por seres humanos, enquanto Web APIs são projetadas para operar com outros sistemas. No cenário de integração através de Web APIs, cada aplicação possui sua própria base de dados, sem compartilhá-la com os demais sistemas. As informações são expostas através de Web APIs, que formam uma camada de integração. Os dados são geralmente representados nos formatos JSON ou XML, e transportados via HTTP. Com esta abordagem de integração, a Web se torna uma infraestrutura para construção de sistemas distribuídos.

A integração por Web APIs possui a vantagem de reduzir o acoplamento entre as aplicações, possibilitando que evoluam em ritmos diferentes, pois alterações nos modelos de dados não influenciam diretamente a integração. Outro ponto positivo é a possibilidade da integração se estender para fora dos domínios da organização. Como a base de dados não é compartilhada, apenas dados específicos são disponibilizados, atuando como *backend* para outras aplicações. Por outro lado, acrescenta a complexidade de implementação de uma camada extra, responsável por realizar e atender chamadas a outras Web APIs, além converter os dados nos formatos estipulados. Além disso, Web APIs são fundamentais para aplicações *mobile*, que geralmente utilizam o suporte *server-side* para atender aos seus objetivos.



Integração de aplicações através de Web APIs

Na integração através de banco de dados, as aplicações executam operações de consulta, criação, alteração e remoção de dados, conhecidas como operações CRUD (*Create, Read, Update, Delete*). É esperado que a integração através de Web APIs seja capaz de realizar as mesmas operações. Na realidade, grande parte das Web APIs desenvolvidas possuem esta característica. Aplicações dessa natureza parecem se encaixar adequadamente ao estilo arquitetural REST. Sendo assim, aplicações CRUD que disponibilizam suas funcionalidades através de uma Web API podem ser facilmente integradas com outras aplicações.



Web APIs como backend para outras aplicações

Princípios Arquiteturais REST

REST (*REpresentational State Transfer*) é uma coleção de princípios e restrições arquiteturais para o desenvolvimento de aplicações distribuídas na Web. REST é uma abordagem leve para o desenvolvimento de Web Services, que busca simplicidade e baixo acoplamento. Recursos formam a base dos princípios REST. Um recurso agrupa um conjunto de dados que juntos representam uma unidade de informação coesa. Recursos são acessíveis a clientes remotos através de representações, que são endereçadas através de um identificador único, denominado URI (*Uniform Resource Identifier*). A representação de um recurso é uma amostra dos valores de suas propriedades em um determinado momento do tempo.

JSON é um dos formatos mais utilizados em Web APIs para representar a estrutura e os dados dos recursos. JSON utiliza uma coleção de pares de chave/valor, onde a chave sempre é descrita como texto, e o valor pode ser expresso como literal, numérico, booleano, nulo, objeto ou uma sequência ordenada de valores. É muito utilizado no intercâmbio de informações, pois é independente de linguagem de programação e fácil criação, manipulação e análise.

Dentre os princípios arquiteturais REST está o estabelecimento de uma interface uniforme entre cliente e servidor. Uma das formas para estabelecer uma interface uniforme é respeitar a semântica do protocolo utilizado pela Web API. O HTTP é o protocolo mais utilizados em Web APIs REST, e respeitar a semântica do protocolo significa utilizar adequadamente os seus verbos. Os verbos HTTP mais utilizados são:

- GET - Obter a representação de um recurso;
- POST - Criar um novo recurso;
- PUT - Alterar um recurso;
- DELETE - Remover um recurso.

Espera-se que o significado dos verbos HTTP sejam respeitados, empregando o verbo adequado para cada ação, embora muitas implementações REST negligenciem esta restrição e utilizam GET para obter, criar, alterar e remover recursos, dentre outras combinações. Outra restrição imposta pelo REST é a correta utilização de códigos de *status* ou mensagens. Todas as requisições tratadas pelo servidor recebem um código de *status* que informa ao cliente o resultado da requisição. Os códigos possuem tamanho fixo de três dígitos e estão organizados da seguinte forma:

- 1XX - Informações;
- 2XX - Sucessos;
- 3XX - Redirecionamentos;
- 4XX - Erros causados pelo cliente;
- 5XX - Erros causados no servidor.

Outra restrição arquitetural REST exige que as requisições contenham todas as informações necessárias para sua execução, sem recorrer a dados armazenados em sessões do usuário, ou seja,

requisições auto-descritivas. Não é esperado que o servidor mantenha dados na sessão do usuário, tornando a aplicação *stateless*, ou seja, o servidor não deve manter nenhuma informação sobre as requisições realizadas. Esta restrição é importante para promover a escalabilidade do sistema, pois diversas instâncias da Web API podem ser iniciadas para realizar o balanceamento de carga. Considerando que as requisições dos clientes sejam auto-descritivas, qualquer Web API pode atender a qualquer requisição sem necessidade de compartilhamento de estados entre os servidores.

Jersey

Jersey é a implementação de referência da especificação JAX-RS, que estabelece os mecanismos para o desenvolvimento de Web Services REST para a linguagem Java. O framework Jersey permite a manipulação de requisições HTTP, a serialização de representações de recursos em diversos formatos, além de mecanismos para tratamento de exceções.

O código de exemplo de utilização do Jersey apresenta uma classe que recebe as anotações do framework para manipular requisições HTTP. A anotação `@Path("caminho1")` é aplicada diretamente sobre a classe e determina uma URL de acesso. As anotações `@GET`, `@POST`, `@PUT` e `@DELETE` são utilizadas para associar os métodos da classe aos respectivos verbos HTTP. As anotações `@Consumes` e `@Produces` especificam o formato das representações que são esperadas e retornadas pelos métodos, respectivamente. Neste exemplo, as representações serão serializadas em JSON. Através da anotação `@Path`, aplicada sobre um método, é possível adicionar trechos adicionais à URL, além de definir variáveis através de `@PathParam` ou de `@QueryParam`, como exemplificado no método `carregar`. Os valores das variáveis do tipo `@PathParam` são atribuídos como parte integrante da URL, enquanto os valores de `@QueryParam` são associados aos nomes das variáveis.

Exemplo de anotações Jersey

```

@Path("caminho1")
public class ExemploJersey {


    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("caminho2/{var1}")
    public Response carregar(@PathParam("var1") String x, @QueryParam("var2") String y){
        // codigo para carregar um recurso
        String retorno = String.format("var1: %s var2: %s", x, y);
        return Response.ok(retorno).build();
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response criar() {
        // codigo para criar um recurso
        return Response.ok("mensagem de retorno").build();
    }

    @PUT
    @Consumes(MediaType.APPLICATION_JSON)
    public Response modificar() {
        // codigo para modificar um recurso
        return Response.ok("mensagem de retorno").build();
    }

    @DELETE
    public Response remover() {
        // codigo para remover um recurso
        return Response.ok("mensagem de retorno").build();
    }
}

```



var1: Ordem var2: Progresso

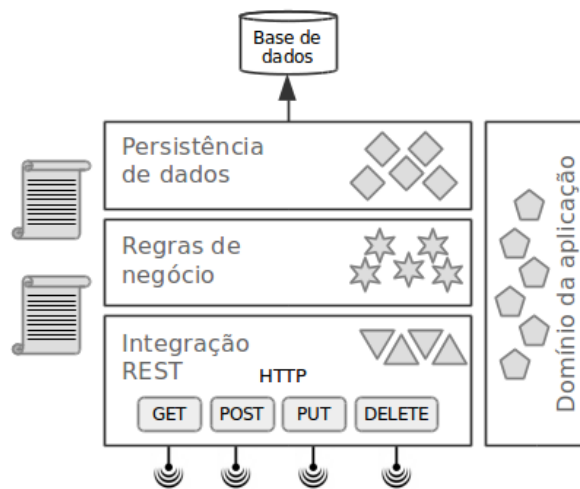
Exemplo de manipulação de variáveis

Spring Boot

Spring Boot é um framework para o desenvolvimento de aplicações baseadas em Spring. Sua principal contribuição é a facilidade de configuração do projeto e aumento de produtividade. Além

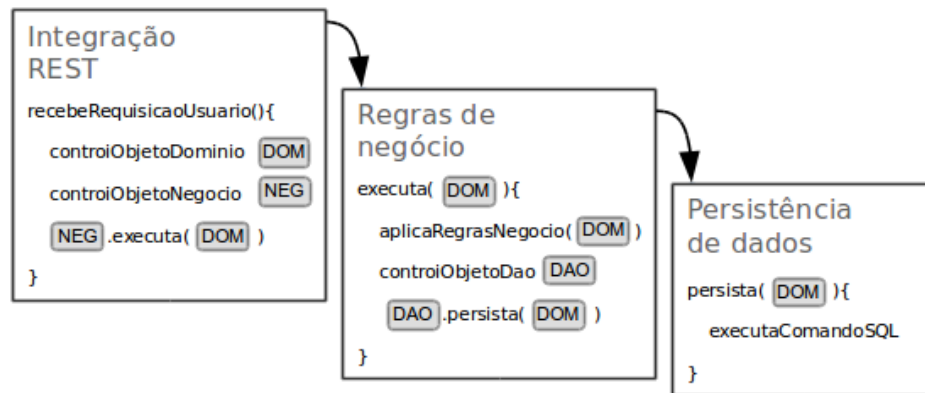
disso, o Spring Boot é uma das opções mais adotadas para o desenvolvimento de Web APIs em Java, principalmente para a arquitetura de microservices. Caso você não tenha experiência com o Spring framework não se preocupe, pois uma breve introdução será apresentada a seguir. Se você domina os conceitos básicos do Spring, fique a vontade para seguir em frente.

Antes de falar sobre o Spring framework, primeiro vamos discutir um pouco sobre design de software. Uma das formas mais tradicionais de modelagem é o design em camadas, que agrupa o sistema em classes que possuem a mesma responsabilidade, tais como: persistir informações em um banco de dados, aplicar regras de negócio ou interagir com os usuários. Além disso, existe a camada de domínio de aplicação, que descreve as informações manipuladas pelo sistema, sendo utilizada pelas demais camadas. Neste exemplo, a camada de persistência de dados presta serviços para a camada de regras de negócio, que por sua vez, presta serviços para a camada de integração. Os serviços são descritos por meio de contratos, que estabelecem as diretrizes para a execução das funcionalidades.



Design em camadas

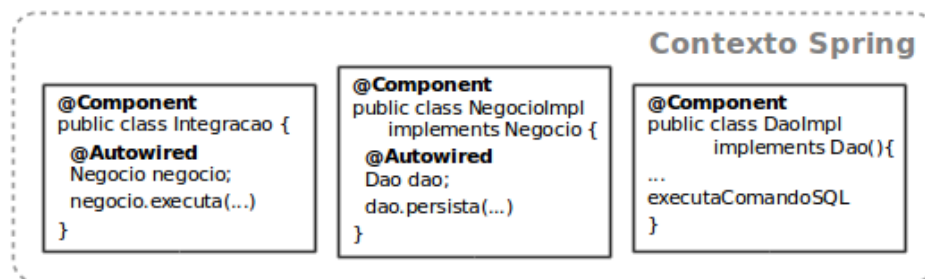
As camadas do sistema trocam mensagens através de um fluxo bem definido, como mostra a figura a seguir. Ao receber uma requisição do usuário, a camada de integração converte os dados recebidos em um objeto de domínio (DOM). Em seguida, a informação (objeto de domínio) é passada para a camada de negócio através da construção de um objeto (NEG) e a invocação de uma de suas funcionalidades descritas no seu contrato. Por sua vez, a camada de negócio aplica as regras necessárias e solicita serviços de persistência (DAO). Por fim, a camada de persistência recebe o objeto de domínio e executa alguma operação de banco de dados.



Interação entre camadas

Através deste modelo de interação, é possível dizer que a camada de integração depende da camada de negócios, que por sua vez, depende da camada de persistência. Entretanto, para manter o baixo acoplamento, as camadas se comunicam com base somente nos contratos de serviço. As classes que implementam os serviços não devem ser compartilhadas entre as camadas. É neste ponto que o Spring framework entra em ação. Ele é capaz de realizar a *injeção de dependências*, que a partir do contrato de serviço (interface), cria um objeto que implementa esta interface.

A anotação `@Component` define uma classe como um *bean* do Spring que pode ser injetado em outro *bean*, fazendo parte do contexto das classes gerenciadas pelo framework. A classe *Integracao* apenas informa que depende de um objeto que implementa o contrato definido pela interface *Negocio*. O mesmo ocorre na classe *NegocioImpl*, que depende de um objeto que implementa a interface *Dao*. Estes pontos de injeção são demarcados através da anotação `@Autowired`, e durante o carregamento da aplicação, o Spring framework providencia a criação dos objetos necessários. Nas próximas seções serão apresentados exemplos concretos que utilizam a injeção de dependências. A injeção de dependências é apenas uma das funcionalidades disponibilizadas pelo Spring. Vários outros módulos fazem parte da pilha de tecnologias do framework. Neste projeto, será utilizado também o suporte JDBC do Spring, que facilita a manipulação de banco de dados, além de oferecer controle de transações, fundamental para garantir a integridade dos dados.



Contexto Spring e injeção de dependências

É fundamental compreender corretamente o comportamento dos *beans* dos Spring. Por padrão, quando o Spring cria uma instância de um *bean*, este objeto segue o comportamento *singleton*, onde apenas um objeto é construído e utilizado nos pontos de injeção. Ao anotar um endpoint com *@Component*, adota-se o comportamento *prototype*, onde os valores dos atributos da classe serão mantidos entre as requisições. Este entendimento sobre os beans do Spring é fundamental para garantir o comportamento correto da aplicação.



```
SingletonEndpoint.java
9 @Component
10 @Path("singleton")
11 public class SingletonEndpoint {
12     int acessos = 0;
13
14     @GET
15     public Response teste() {
16         acessos++;
17         System.out.println(acessos);
18         return Response.ok(acessos).build();
19     }
20 }
21
```

Markers Properties Servers Data Source Explorer Snippets Console

WebApiApplication [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (Dec 16, 2015, 10:45:19 AM)

```
1
2
3
4
5
```

Endpoint singleton



```
PrototypeEndpoint.java
6
7 @Path("prototype")
8 public class PrototypeEndpoint {
9     int acessos = 0;
10
11     @GET
12     public Response teste() {
13         acessos++;
14         System.out.println(acessos);
15         return Response.ok(acessos).build();
16     }
17 }
18
```

Markers Properties Servers Data Source Explorer Snippets Console

WebApiApplication [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (Dec 16, 2015, 10:45:19 AM)

```
1
1
1
1
1
```

Endpoint prototype

Projeto

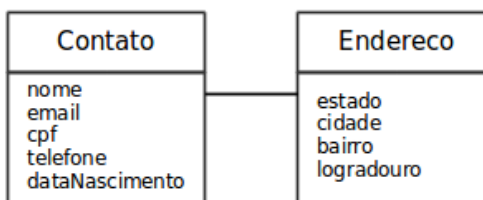
Este capítulo apresenta os detalhes do projeto de uma Web API que será implementada com Spring Boot e Jersey. Primeiramente é apresentada a visão geral, seguida da modelagem do domínio da aplicação e da integração de dados, ferramentas utilizadas, contratos de serviço e configuração de banco de dados.

Visão Geral

Uma aplicação de gerenciamento de uma agenda de contatos é utilizada como exemplo para aplicar as tecnologias abordadas neste livro. Diversas simplificações foram realizadas no projeto para manter o foco nas tecnologias, simplificar o entendimento e a implementação. Embora o exemplo seja baseado em um estudo de caso simples, o projeto apresenta os requisitos mais comuns em aplicações reais. O projeto contempla o desenvolvimento de uma Web API para a manipulação de dados de contatos. Através de requisições HTTP, deve ser possível cadastrar novos contatos, consultar contatos anteriormente cadastrados, além de alterar e remover os dados.

Modelagem do Domínio da Aplicação

O domínio da aplicação especifica quais são as informações manipuladas pelo sistema. O domínio é constituído apenas pelas classes *Contato* e *Endereco*. A aplicação deve gerenciar o *nome*, *email*, *cpf*, *telefone*, *telefone* e o *endereço* dos contatos. O endereço é formado pelo *estado*, *cidade*, *bairro* e *logradouro*. As classes de domínio da aplicação são implementadas como *Plain Old Java Object* (POJOs), constituídas apenas por atributos privados, construtor padrão e métodos acessores. Embora esta modelagem resulte em objetos de domínio anêmicos, ainda assim é uma abordagem tradicional e muito utilizada.



Modelo conceitual do domínio da aplicação

Contato.java

```
public class Cliente {  
    private String id;  
    private String nome;  
    private String email;  
    private String cpf;  
    private String telefone;  
    private Date dataNascimento;  
    private Endereco endereco;  
    //gets e sets omitidos  
}
```

Endereco.java

```
public class Endereco {  
    private String estado;  
    private String cidade;  
    private String bairro;  
    private String logradouro;  
    //gets e sets omitidos  
}
```

Integração de Dados

A integração de dados representa a interface com o usuário do sistema, que no contexto de Web APIs são outras aplicações. Com base nas funcionalidades descritas na visão geral do projeto, pode-se modelar as classes de integração por meio de dois recursos. O recurso *ListaDeContatos* agrupa todos os contatos cadastrados no sistema, e disponibiliza dois métodos para interação. O primeiro método utiliza HTTP GET, que retorna a representação da lista ao usuário. O segundo método utiliza HTTP POST para adicionar um novo contato à lista. O recurso *Contato* manipula as informações de um contato específico e disponibiliza três métodos de interação. O primeiro método utiliza HTTP GET que retorna os dados de um contato, enquanto o segundo e o terceiro método utilizam HTTP PUT e DELETE para alterar e remover um contato, respectivamente.



Recursos e métodos para integração de dados

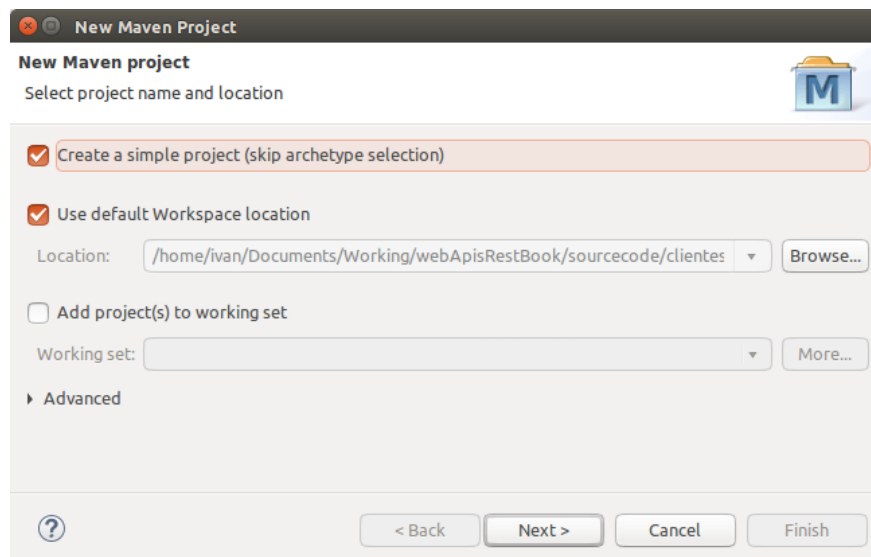
Ferramentas Utilizadas

A IDE utilizada para implementar o projeto foi o Eclipse versão Mars Release (4.5.0). Entretanto, outras IDEs podem ser utilizadas sem prejuízo para o desenvolvimento. O MySQL versão 5.5.46 foi utilizado com SGBD da aplicação. Novamente, outros bancos de dados podem ser utilizados para implementar o projeto. O Apache Maven foi utilizado para gerenciar o projeto. Foi utilizada a versão que acompanha o Eclipse, sem necessidade de nenhuma instalação externa.

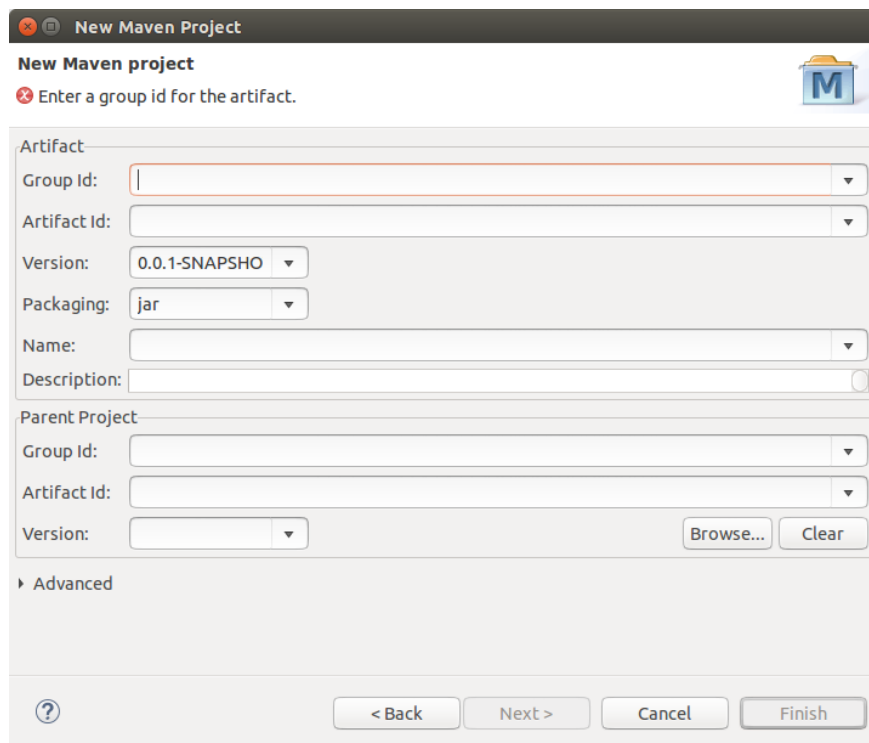
Configuração Inicial

Como dito anteriormente, o projeto é gerenciado pelo Apache Maven. Para criar um projeto Maven basta selecionar *New > Maven Project* no menu *File*. Através da seleção da opção *Create a simple project*, será criado um projeto Maven simples, sem nenhuma pré-configuração. Na próxima janela serão preenchidas as informações de *Group Id* e *Artifact Id*, que representam a organização que desenvolve o projeto e o nome do projeto, respectivamente. Neste exemplo, o valor de *Group Id* é *br.com.exemplo* e de *Artifact Id* é *agenda-api*. Embora o projeto seja uma aplicação Web, o *Packaging* selecionado é *jar*.

Com o projeto criado, é o momento de organizar as classes em pacotes. O pacote *config* agrupa todas as classes relacionadas com a configuração da aplicação. Os pacotes *dao* e *negocio* agrupam as classes e interfaces de persistência e regras de negócio, respectivamente. As classes do domínio da aplicação são agrupadas no pacote *dominio*. As classes responsáveis por manipular as requisições dos usuários são agrupadas no pacote *endpoint*. Por fim, as representações de recursos que exigem algum tratamento de apresentação serão agrupadas no pacote *representacao*.



Novo projeto Maven parte 1



Novo projeto Maven parte 2

```
▼ agenda-api [webApisRestBook master]
  ▼ src/main/java
    ► br.com.exemplo.agenda.api.config
    ► br.com.exemplo.agenda.api.dao
    ► br.com.exemplo.agenda.api.dominio
    ► br.com.exemplo.agenda.api.endpoint
    ► br.com.exemplo.agenda.api.negocio
    ► br.com.exemplo.agenda.api.representacao
  ► JRE System Library [JavaSE-1.8]
  ► Maven Dependencies
  ► src
  ► target
  ► application.yml
  ► pom.xml
```

Estrutura de pacotes

O projeto também possui dois arquivos de configuração: *application.yml* e *pom.xml*. O arquivo *application.yml* é responsável por externalizar a configuração da aplicação, como por exemplo, dados de conexão ao banco de dados, porta HTTP para receber as requisições, além de outras configurações necessárias. No arquivo *pom.xml* são descritas as dependências (bibliotecas) externas, além de diretrizes para compilação do projeto.

pom.xml

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.8</source> <target>1.8</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <configuration>
        <archive>
          <manifest>
            <addDefaultImplementationEntries>
              true
            </addDefaultImplementationEntries>
          </manifest>
        </archive>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jersey</artifactId>
  </dependency>
</dependencies>

```

Certifique-se de adicionar o conteúdo do arquivo *pom.xml* em seu projeto conforme o exemplo anterior. Ao declarar as dependências e salvar o arquivo, o Apache Maven se encarrega de realizar

o download das bibliotecas e importar ao projeto. Este procedimento pode levar alguns minutos. Quando o arquivo *pom.xml* é modificado, é necessário atualizar o projeto da seguinte forma: clique com o botão direito do mouse sobre o nome do projeto, em seguida selecione o menu *Maven > Update Project...* na próxima janela certifique-se de que o projeto está selecionado e confirme.

Com o projeto criado e as bibliotecas configuradas, é hora de configurar o framework Jersey. Crie uma classe no pacote *config* conforme descrito em *JerseyConfig.java*. A anotação *@ApplicationPath* define a URL padrão da aplicação. Todos os endpoints da aplicação devem ser registrados. Um endpoint pode ser registrado individualmente ou pode-se registrar todos os endpoints de um pacote. A opção escolhida foi registrar o pacote *endpoint*, dessa forma, todos os endpoints deste pacote estão automaticamente registrados.

JerseyConfig.java

```
@Component
@ApplicationPath("/agenda-api")
public class JerseyConfig extends ResourceConfig {
    public JerseyConfig() {
        this.register(RequestContextFilter.class);
        this.packages("br.com.exemplo.agenda.api.endpoint");
    }
}
```

O Spring Boot permite que uma aplicação Web seja executada a partir de um arquivo *jar* executável, semelhante a uma aplicação *stand-alone*. Sendo assim, é preciso implementar uma classe que implementa o método *main*. Crie uma classe no pacote *config* com o conteúdo de *WebApiApplication.java*. A anotação *@SpringBootApplication* define a classe como a responsável por iniciar a aplicação. A anotação *@ComponentScan* recebe o nome do pacote para iniciar a varredura dos beans do Spring anotados com *@Component*. De acordo com o exemplo, a varredura contempla todos os pacotes a partir de “*br.com.exemplo.agenda.api*”, localizando beans nos pacotes, *config*, *dao*, *dominio*, *endpoint*, *negocio* e *representacao*.

WebApiApplication.java

```
@SpringBootApplication
@ComponentScan("br.com.exemplo.agenda.api")
public class WebApiApplication {
    public static void main(String[] args) {
        SpringApplication.run(WebApiApplication.class, args);
    }
}
```

Neste momento a aplicação está pronta para manipular requisições HTTP. Vamos fazer um teste para verificar se tudo está configurado corretamente. Crie uma classe no pacote *endpoint* com o conteúdo de *TesteEndPoint.java*. Este endpoint manipula requisições HTTP GET mapeadas para a URL “*teste*”. O resultado da requisição é uma mensagem de texto informando que o teste foi bem

sucedido. Os endpoints são acessados através de URLs resultantes da concatenação da URL base definida na configuração do Jersey com os caminhos definidos em cada endpoint e seus respectivos métodos. Por padrão, o Spring Boot utiliza a porta HTTP 8080. Para realizar o teste, execute a classe `main*` e digite a seguinte URL em seu navegador: `localhost:8080/agenda-api/teste`.

TesteEndPoint.java

```
Path("teste")
public class TesteEndPoint {
    @GET
    public Response teste() {
        return Response.ok("Teste bem sucedido").build();
    }
}
```

Contratos

Os contratos são as descrições dos serviços prestados pelas camadas do sistema. Na linguagem de programação Java os contratos são desenvolvidos através de *interfaces*. A seguir, são definidos os dois contratos da camada de persistência. Cada contrato descreve os serviços de persistência associados a uma classe de domínio da aplicação. Sendo assim, o contrato especificado em *ContatoDao.java* define os serviços de persistência para a classe *Contato*, enquanto o contrato especificado em *EnderecoDao.java* define os serviços para a classe *Endereco*.

ContatoDao.java

```
public interface ContatoDao {
    void cadastrar(Contato contato);
    void alterar(Contato contato);
    void remover(String idContato);
    Contato consultar(String idContato);
    List<Contato> listarTodos();
}
```

EnderecoDao.java

```
public interface EnderecoDao {
    void cadastrar(Endereco endereco, String idContato);
    Endereco consultar(String idContato);
    void remover(String idContato);
}
```

Apenas um contrato é estabelecido na camada de negócio, como especificado em *RegrasContatos.java*. Este contrato considera que o objeto *contato* é composto por um objeto *endereco*. Dessa forma, o endereço é manipulado juntamente com os dados do contato, mesmo que persistido de forma independente.

RegrasContatos.java

```
public interface RegrasContatos {  
    void cadastrar(Contato contato);  
    List<Contato> listarTodos();  
    Contato consultar(String idContato);  
    public void alterar(Contato contato);  
    public void remover(String idContato);  
}
```

Configuração do Banco de Dados

A primeira parte da configuração é a criação da base de dados utilizada pela aplicação. As tabelas do banco de dados foram criadas com base nas classes de domínio da aplicação. Seguindo o domínio da aplicação foram criadas duas tabelas, uma para armazenar os dados do contato e outra para o endereço. A tabela *endereco* não possui chave primária, apenas uma chave estrangeira relacionada ao *id* do contato.

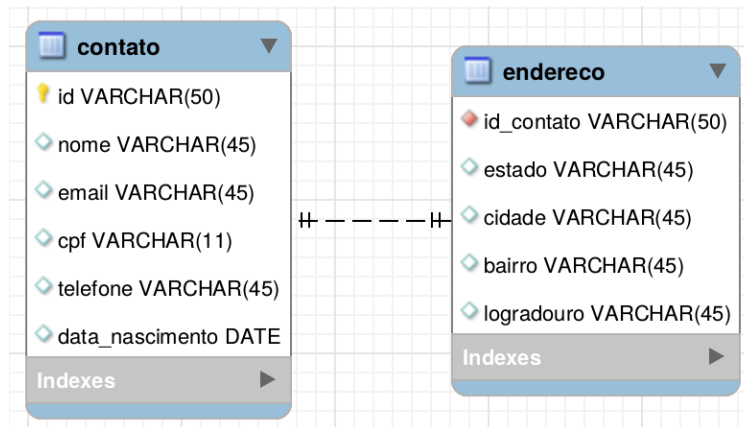


Diagrama do banco de dados

A próxima parte da configuração é adicionar as dependências do driver JDBC - MySQL e do módulo Spring JDBC ao arquivo *pom.xml*. O driver JDBC é uma biblioteca necessária para que uma aplicação Java se comunique com o sistema de banco de dados. O módulo Spring JDBC oferece uma série de mecanismos para facilitar e aumentar a produtividade no desenvolvimento de classes que interagem com o banco de dados.

Dependências para manipular banco de dados (pom.xml)

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.37</version>
</dependency>
```

Por fim, as configurações de acesso ao banco de dados, como por exemplo: URL, porta, nome de usuário e senha, devem ser realizadas no arquivo *application.yml*. Além das configurações básicas é apresentada a configuração necessária para realizar a verificação das conexões, evitando que a aplicação utilize uma conexão inválida.

Configuração do banco de dados (application.yml)

```
spring.datasource.url: "jdbc:mysql://<endereçoServidor>:3306"
spring.datasource.username: <usuario>
spring.datasource.password: <senha>
spring.datasource.driver-class-name: com.mysql.jdbc.Driver
spring.datasource.max-active: 10
spring.datasource.initial-size: 5
spring.datasource.max-idle: 5
spring.datasource.min-idle: 1
spring.datasource.test-while-idle: true
spring.datasource.test-on-borrow: true
spring.datasource.validation-query: "SELECT 1"
spring.datasource.time-between-eviction-runs-millis: 5000
spring.datasource.min-evictable-idle-time-millis: 60000
```

Implementação das Funcionalidades

Chegou a hora de implementar as funcionalidades da aplicação. Este capítulo apresenta os detalhes para implementar a agenda de contatos, contemplando todo o ciclo de vida das informações. De acordo com as especificações estabelecidas anteriormente, são apresentados os detalhes de implementação das funcionalidades de cadastramento, consulta, alteração e remoção de contatos.

Cadastramento

O cadastramento é a funcionalidade responsável por criar um novo contato na agenda. O processo de cadastramento é iniciado através de uma requisição HTTP, que solicita que a representação informada seja mantida no banco de dados. Sendo assim, vamos iniciar a implementação pelo endpoint responsável por manipular as requisições do usuário.

Crie uma classe no pacote *endpoint* com o conteúdo de *ListaContatosEndpoint.java*. A anotação *@Path("listaDeContatos")* define que a classe tem o comportamento de endpoint e estabelece uma URL de acesso. Nas linhas 4 e 5 é demarcado um ponto de injeção de dependência para uma instância de objeto que representa as regras de negócio. Este é o ponto onde a informação passa da camada de integração para a camada de negócio. Entre as linhas 7 e 13 é implementado o método que recebe a requisição para o cadastro do contato. O método é anotado com *@POST*, que define o verbo HTTP a ser utilizado. As anotações *@Produces* e *@Consumes* definem o JSON como formato de representação de entrada e de saída do método. O método recebe um objeto que é automaticamente convertido de JSON para um objeto do tipo *Contato*. Na sequência, é invocado o método *cadastrar* do contrato das regras de negócio e retornado ao usuário o objeto armazenado no banco de dados.

O próximo passo é implementar a classe responsável pelas regras de negócio para o cadastramento de contatos. O código apresentado em *GerenciadorContatos.java* apresenta a classe que implementa a interface *RegrasContatos* com todos os métodos definidos no contrato. Deve-se anotar a classe com *@Component* para defini-la como um bean do Spring. O cadastramento exige a manipulação das informações do contato e de seu endereço. Sendo assim, nas linhas 4 a 8 são definidos os pontos de injeção de dependência para os Daos responsáveis pela persistência dos dados. O método *cadastrar* apenas gera um id aleatório para o contato, além de solicitar para a camada de persistência o armazenamento das informações.

ListaContatosEndpoint.java - cadastramento

```
1 @Path("listaDeContatos")
2 public class ListaContatosEndpoint {
3
4     @Autowired
5     private RegrasContatos regrasContatos;
6
7     @POST
8     @Produces(MediaType.APPLICATION_JSON)
9     @Consumes(MediaType.APPLICATION_JSON)
10    public Response cadastrarContato(ContatoRep contato) {
11        regrasContatos.cadastrar(contato);
12        return Response.ok(contato).build();
13    }
14
15    @GET
16    @Produces(MediaType.APPLICATION_JSON)
17    public Response carregarListaContatos() {...}
18 }
```

Note que os métodos *cadastrar*, *alterar* e *remover* são anotados com *@Transactional*. Esta anotação define uma transação de negócio, que garante a execução atômica de todas as instruções do método. Caso alguma exceção seja lançada durante a execução do método, o Spring framework garante o retorno do banco de dados para o estado inicial da transação.

Imagine o seguinte cenário onde não seja definida uma transação. Na linha 15 de *GerenciadorContatos.java* é solicitada a gravação das informações do contato no banco de dados. Considere que as informações foram persistidas na tabela de contatos. Durante a gravação dos dados de endereço (linha 16) ocorre alguma exceção que não permita a persistência na tabela de endereço. Entretanto, existe uma restrição que todo contato deve obrigatoriamente possuir informações do endereço. Neste cenário, o banco de dados está em um estado de inconsistência, pois os dados do contato foram armazenados sem os dados de endereço. Por outro lado, quando o método é anotado com *@Transactional*, a transação garante que os dados armazenados na tabela do contato sejam desfeitos, resultando no *rollback* automático dos dados.

GerenciadorContatos.java - cadastramento

```
1 @Component
2 public class GerenciadorContatos implements RegrasContatos {
3
4     @Autowired
5     private ContatoDao contatoDao;
6
7     @Autowired
8     private EnderecoDao enderecoDao;
9 }
```

```

10  @Override
11  @Transactional
12  public void cadastrar(Contato contato) {
13      String idContato = UUID.randomUUID().toString();
14      contato.setId(idContato);
15      contatoDao.cadastrar(contato);
16      enderecoDao.cadastrar(contato.getEndereco(), idContato);
17  }
18
19  @Override
20  public List<Contato> listarTodos() {...}
21
22  @Override
23  public Contato consultar(String idContato) {...}
24
25  @Override
26  @Transactional
27  public void alterar(Contato contato) {...}
28
29  @Override
30  @Transactional
31  public void remover(String idContato) {...}
32  }

```

Os arquivos *JdbcContatoDao.java* e *JdbcEnderecoDao.java* apresentam a implementação das classes de persistência do contato e do endereço. A manipulação do banco de dados é realizada através do módulo Spring JDBC. Além disso, ambas as classes são anotadas com *@Component* para que possam ser injetadas nas instâncias que necessitam dos serviços de persistência.

JdbcContatoDao.java - cadastramento

```

@Component
public class JdbcContatoDao implements ContatoDao {

    @Autowired
    private NamedParameterJdbcTemplate jdbcTemplate;

    @Override
    public void cadastrar(Contato contato) {
        StringBuilder sql = new StringBuilder();
        sql.append("insert into agenda.contato ");
        sql.append("(id, nome, email, cpf, telefone, data_nascimento) ");
        sql.append("values (:id, :nome, :email, :cpf, :tel, :dataN)");

        Map<String, Object> parametros = new HashMap<>();
        parametros.put("id", contato.getId());
        parametros.put("nome", contato.getNome());
    }

```

```

        parametros.put("email", contato.getEmail());
        parametros.put("cpf", contato.getCpf());
        parametros.put("tel", contato.getTelefone());
        parametros.put("dataN", contato.getDataNascimento());

        jdbcTemplate.update(sql.toString(), parametros);
    }

    @Override
    public List<Contato> listarTodos() {...}

    @Override
    public Contato consultar(String idContato) {...}

    @Override
    public void alterar(Contato contato) {...}

    @Override
    public void remover(String idContato) {...}
}

```

JdbcEnderecoDao.java - cadastramento

```

@Component
public class JdbcEnderecoDao implements EnderecoDao {

    @Autowired
    private NamedParameterJdbcTemplate jdbcTemplate;

    @Override
    public void cadastrar(Endereco endereco, String idContato) {
        StringBuilder sql = new StringBuilder();
        sql.append("insert into agenda.endereco ");
        sql.append("(estado, cidade, bairro, logradouro, id_contato) ");
        sql.append("values (:estado, :cidade, :bairro, :logradouro, :idContato)");

        Map<String, Object> parametros = new HashMap<>();
        parametros.put("idContato", idContato);
        parametros.put("estado", endereco.getEstado());
        parametros.put("cidade", endereco.getCidade());
        parametros.put("bairro", endereco.getBairro());
        parametros.put("logradouro", endereco.getLogradouro());

        jdbcTemplate.update(sql.toString(), parametros);
    }

    @Override

```

```
public Endereco consultar(String idContato) {...}

@Override
public void remover(String idContato) {...}
}
```

O suporte do Spring JDBC é disponibilizado através de um *NamedParameterJdbcTemplate* injetado diretamente nas classes Dao. Este objeto oferece a abstração necessária para executar instruções SQL. Utilizando o *Stringbuilder*, é escrita uma instrução SQL para inserir os dados no banco. As variáveis são associadas à chaves precedidas por dois pontos `':'`. Em seguida, é construído um mapa para relacionar as chaves aos valores extraídos do objeto de domínio *contato*. Por fim, é invocado o método *update* que recebe a String que representa o SQL e o mapa de parâmetros. O armazenamento do endereço segue o mesmo procedimento, modificando apenas o comando SQL e a extração das informações do objeto de domínio relacionado ao endereço.

Para testar a implementação do cadastro de contatos é necessário realizar uma requisição HTTP POST. Para isso, vamos utilizar uma extensão de navegador chamada *Postman*. A parte superior da ferramenta mostra os dados enviados para Web API, enquanto a parte inferior mostra as informações retornadas. A mensagem de retorno contém os dados enviados com a adição do *id* gerado pela aplicação. Entretanto, é possível notar que a data de nascimento retornou com um valor diferente. Analisando com mais detalhes, vamos até o banco de dados para verificar como o registro foi armazenado. Como verificado, a data de nascimento foi armazenada com um valor incorreto (“2000-10-09”), sendo que o valor informado foi “2000-10-10”. Isto ocorre devido à conversão direta de uma String para um objeto do tipo *Date*. Uma forma de corrigir este erro é converter os dados manualmente para uma representação personalizada do recurso, ao invés de utilizar diretamente o objeto de domínio da aplicação.

localhost:8080/agenda-api/listaDeContatos POST URL params Headers (1)

Content-Type application/json

Header Value

Manage presets

form-data x-www-form-urlencoded raw JSON

```

1 {
2   "nome": "João da Silva",
3   "email": "joao@silva.com",
4   "cpf": "111",
5   "telefone": "55 48 12345",
6   "dataNascimento": "2000-10-10",
7   "endereco": {
8     "estado": "Santa Catarina",
9     "cidade": "Florianópolis",
10    "bairro": "Trindade",
11    "logradouro": "Rua x, número y, ao lado de z."
12  }
13 }

```

Send Preview Add to collection Reset

body Headers (8) STATUS 200 OK TIME 1070 ms

Pretty Raw Preview JSON XML

```

1 {
2   "id": "6a6503a8-451a-4dc6-aedf-822e65aa1264",
3   "nome": "João da Silva",
4   "email": "joao@silva.com",
5   "cpf": "111",
6   "telefone": "55 48 12345",
7   "dataNascimento": "971136000000",
8   "endereco": {
9     "estado": "Santa Catarina",
10    "cidade": "Florianópolis",
11    "bairro": "Trindade",
12    "logradouro": "Rua x, número y, ao lado de z."
13  }
14 }

```

Requisição de teste para cadastramento de contato

1 SELECT * FROM agenda.contato;

#	id	nome	email	cpf	telefone	data_nascimento
1	6a6503a8-451a-4dc6-aedf-822e65aa1264	João da Silva	joao@silva.com	111	55 48 12345	2000-10-09
*	NULL	NULL	NULL	NULL	NULL	NULL

Registro de teste armazenado no banco de dados

Criação de Representações

Representações são classes de apoio para a troca de informações entre Web APIs e seus clientes. Elas são utilizadas em situações em que compartilhar diretamente os objetos de domínio da aplicação não

é adequado, principalmente quando é necessário atender questões de formatação de dados, valores numéricos ou a própria estrutura das informações.

A classe *ContatoRep.java* mostra o código da representação do contato e de seu endereço. Todos os atributos são valores textuais, inclusive a data de nascimento. Dessa forma, as informações enviadas pelos clientes serão tratadas como String e convertidas adequadamente. Outra diferença entre a representação e as classes de domínio da aplicação é a forma como os atributos estão estruturados, pois todos os atributos estão organizados de forma plana, sem composição de classes. Sendo assim, é possível criar diversas representações para atender diferentes expectativas e propósitos das aplicações clientes.

ContatoRep.java

```

1 public class ContatoRep {
2     private String id;
3     private String nome;
4     private String email;
5     private String cpf;
6     private String telefone;
7     private String dataNascimento;
8     private String estado;
9     private String cidade;
10    private String bairro;
11    private String logradouro;
12
13    public ContatoRep() {}
14
15    public ContatoRep(Contato contato) {
16        this.id = contato.getId();
17        this.nome = contato.getNome();
18        this.email = contato.getEmail();
19        this.cpf = contato.getCpf();
20        this.telefone = contato.getTelefone();
21        this.dataNascimento = serializarData(contato.getDataNascimento());
22
23        if (contato.getEndereco() != null) {
24            this.estado = contato.getEndereco().getEstado();
25            this.cidade = contato.getEndereco().getCidade();
26            this.bairro = contato.getEndereco().getBairro();
27            this.logradouro = contato.getEndereco().getLogradouro();
28        }
29    }
30
31    public Contato converterParaDominio() {
32        Contato contato = new Contato();
33        contato.setId(this.id);
34        contato.setNome(this.nome);
35        contato.setEmail(this.email);

```

```
36     contato.setCpf(this.cpf);
37     contato.setTelefone(this.telefone);
38
39     Date dataN = converterData(this.dataNascimento);
40     contato.setDataNascimento(dataN);
41
42     Endereco endereco = new Endereco();
43     endereco.setEstado(this.estado);
44     endereco.setCidade(this.cidade);
45     endereco.setBairro(this.bairro);
46     endereco.setLogradouro(this.logradouro);
47     contato.setEndereco(endereco);
48     return contato;
49 }
50
51 private Date converterData(String dataTextual) {
52     DateTimeFormatter dtf = DateTimeFormat.forPattern("dd/MM/yyyy");
53     DateTime dataConvertida = dtf.parseDateTime(dataTextual);
54     return dataConvertida.toDate();
55 }
56
57 private String serializarData(Date data) {
58     DateTimeFormatter dtf = DateTimeFormat.forPattern("dd/MM/yyyy");
59     LocalDateTime dt = new LocalDateTime(data, DateTimeZone.UTC);
60     return dtf.print(dt);
61 }
62 //gets e sets omitidos
63 }
```

Além do construtor padrão, está disponível um construtor que preenche os atributos a partir de um objeto de domínio, além de um método conversor de representação para domínio. Existe também o conversor para manipular datas. O método *converterData* converte uma data representada textualmente no formato *dd/MM/yyyy* em um objeto do tipo *Date*. A conversão inversa é realizada pelo método *serializarData* capaz de transformar em String um objeto do tipo *Date*. Para manipular a data foi utilizada a biblioteca *joda-time*. Entretanto, é necessário incluir ao *pom.xml* esta dependência.

Dependência joda-time (pom.xml)

```
<dependency>
  <groupId>joda-time</groupId>
  <artifactId>joda-time</artifactId>
</dependency>
```

O próximo passo é substituir a classe de domínio pela representação em *ListaContatosEndpoint.java*. O método *cadastrarContato* recebe agora uma representação e não mais um objeto de domínio. Na linha 5 é realizada a conversão da representação para o domínio, que é repassado para a camada de negócio, seguindo o fluxo previamente estabelecido.

ContatoRep.java - cadastramento

```
1 @POST
2 @Produces(MediaType.APPLICATION_JSON)
3 @Consumes(MediaType.APPLICATION_JSON)
4 public Response cadastrarContato(ContatoRep contato) {
5     Contato contatoDominio = contato.converterParaDominio();
6     regrasContatos.cadastrar(contatoDominio);
7     ContatoRep contatoCadastrado = new ContatoRep(contatoDominio);
8     return Response.ok(contatoCadastrado).build();
9 }
```

Em seguida, realiza-se a requisição para cadastro do contato utilizando a nova estrutura da representação. Note que a data foi enviada respeitando o formato esperado e a representação retornada pela Web API está correta. Por fim, verifica-se que as informações do contato foram corretamente armazenadas no banco de dados.

localhost:8080/agenda-api/listaDeContatos POST [URL params](#)

Content-Type: application/json

Header Value

form-data x-www-form-urlencoded raw **JSON**

```

1 {
2   "nome": "Maria da Silva",
3   "email": "maria@silva.com",
4   "cpf": "10987654321",
5   "telefone": "55 48 12345",
6   "dataNascimento": "10/10/2000",
7   "estado": "Santa Catarina",
8   "cidade": "Florianópolis",
9   "bairro": "Trindade",
10  "logradouro": "Rua x, número y, ao lado de z."
11 }

```

Send Preview Add to collection

Body Headers (8) STATUS 200 OK TIME 1043 ms

Pretty Raw Preview JSON XML

```

1 {
2   "id": "437a1cfe-f65f-461b-ba4c-c17240a97bd0",
3   "nome": "Maria da Silva",
4   "email": "maria@silva.com",
5   "cpf": "10987654321",
6   "telefone": "55 48 12345",
7   "dataNascimento": "10/10/2000",
8   "estado": "Santa Catarina",
9   "cidade": "Florianópolis",
10  "bairro": "Trindade",
11  "logradouro": "Rua x, número y, ao lado de z."
12 }

```

Cadastro de contato através da representação

contato x

1 • **SELECT * FROM agenda.contato;**

Result Set Filter: Edit: Export/Import: Wrap Cell Content:

#	id	nome	email	cpf	telefone	data_nascimento
1	437a1cfe-f65f-461b-ba4c-c17240a97bd0	Maria da Silva	maria@silva.com	10987654321	55 48 12345	2000-10-10
2	6a6503a8-451a-4dc6-aedf-822e65aa1264	João da Silva	joao@silva.com	111	55 48 12345	2000-10-09
*	NULL	NULL	NULL	NULL	NULL	NULL

Informações armazenadas no banco de dados

Consulta a Todos os Contatos

A próxima funcionalidade a ser implementada é a consulta de todos os contatos cadastrados. A classe *ListaContatosEndpoint* implementa o método *carregarListaContatos*, associado ao verbo HTTP GET, que retorna a lista de todos os contatos serializados em JSON. O método obtém os contatos através da invocação de um serviço da camada de negócio (linha 12). Na linha 13, é criada uma lista responsável por agrupar as representações dos objetos de domínio. Entre as linhas 14 e 16 todos os objetos de domínio são convertidos em representações. Por fim, a lista de representações é retornada ao cliente.

ListaContatosEndpoint.java - consultar todos os contatos cadastrados

```
1  @Path("listaDeContatos")
2  public class ListaContatosEndpoint {
3
4      @Autowired
5      private RegrasContatos regrasContatos;
6
7      //implementacao do cadastramento omitida
8
9      @GET
10     @Produces(MediaType.APPLICATION_JSON)
11     public Response carregarListaContatos() {
12         List<Contato> lista = regrasContatos.listarTodos();
13         List<ContatoRep> representacoes = new ArrayList<>();
14         for (Contato contato : lista) {
15             representacoes.add(new ContatoRep(contato));
16         }
17         return Response.ok(representacoes).build();
18     }
19 }
```

O próximo passo é implementar em *GerenciadorContatos.java* a regra de negócio para listagem de todos os contatos. Nenhuma restrição é definida, sendo assim, a camada de negócio apenas solicita o serviço da camada de persistência para carregar os objetos desejados.

GerenciadorContatos.java - consultar todos os contatos cadastrados

```

@Component
public class GerenciadorContatos implements RegrasContatos {
    @Autowired
    private ContatoDao contatoDao;

    @Override
    public List<Contato> listarTodos() {
        return contatoDao.listarTodos();
    }
    //demais metodos omitidos
}

```

A ultima parte da funcionalidade é a implementação da consulta de todos os registros no banco de dados. A classe *JdbcContatoDao* mostra a utilização do Spring JDBC para a recuperação de informações do banco de dados. Primeiramente, a instrução SQL de consulta é construída nas linhas 9 e 10. Na sequência, o objeto *jdbcTemplate* injetado pelo Spring executa o SQL e constrói um objeto de domínio com base em um *RowMapper*. O *RowMapper* implementa o método *mapRow* que cria e popula um objeto de domínio através da manipulação do *resultSet*. O método *rowMap* é executado para cada registro retornado pelo banco de dados, que é armazenado em uma lista. Note que o endereço não está sendo carregado juntamente com o contato. Estas informações somente estão disponíveis quando consultada as informações de um contato específico, que será a próxima funcionalidade a ser implementada.

Para finalizar, vamos realizar uma requisição através do Postman para consultar todos os contatos cadastrados. Por meio de uma requisição HTTP GET na URL “localhost:8080/agenda-api/listaDe-Contatos” a Web API retorna um documento JSON com as informações cadastradas.

JdbcContatoDao.java - consultar todos os contatos cadastrados

```

1  @Component
2  public class JdbcContatoDao implements ContatoDao {
3
4      @Autowired
5      private NamedParameterJdbcTemplate jdbcTemplate;
6
7      @Override
8      public List<Contato> listarTodos() {
9          StringBuilder sql = new StringBuilder();
10         sql.append("select * from agenda.contato");
11
12         return jdbcTemplate.query(sql.toString(), new RowMapper<Contato>() {
13             @Override
14             public Contato mapRow(ResultSet rs, int rowNum) throws SQLException {
15                 Contato contato = new Contato();
16                 contato.setId(rs.getString("id"));

```

```

17     contato.setNome(rs.getString("nome"));
18     contato.setEmail(rs.getString("email"));
19     contato.setCpf(rs.getString("cpf"));
20     contato.setTelefone(rs.getString("telefone"));
21     contato.setDataNascimento(rs.getDate("data_nascimento"));
22     return contato;
23 }
24 });
25 }
26 //demais metodos omitidos
27 }

```

localhost:8080/agenda-api/listaDeContatos GET [URL params](#)

Content-Type application/json

Header	Value
--------	-------

[Send](#) [Preview](#) [Add to collection](#)

Body Headers (7) **STATUS** 200 OK **TIME** 27 ms

[Pretty](#) [Raw](#) [Preview](#) [JSON](#) [XML](#)

```

1  [
2    {
3      "id": "437a1cfe-f65f-461b-ba4c-c17240a97bd0",
4      "nome": "Maria da Silva",
5      "email": "maria@silva.com",
6      "cpf": "10987654321",
7      "telefone": "55 48 12345",
8      "dataNascimento": "10/10/2000",
9      "estado": null,
10     "cidade": null,
11     "bairro": null,
12     "logradouro": null
13   },
14   {
15     "id": "6a6503a8-451a-4dc6-aedf-822e65aa1264",
16     "nome": "João da Silva",
17     "email": "joao@silva.com",
18     "cpf": "111",
19     "telefone": "55 48 12345",
20     "dataNascimento": "09/10/2000",
21     "estado": null,
22     "cidade": null,
23     "bairro": null,
24     "logradouro": null
25   }
26 ]

```

Requisição para listar todos os contatos

Consulta a um Contato Específico

Esta funcionalidade consulta as informações de um contato específico. A classe *ContatoEndpoint* implementa o endpoint que manipula as requisições destinadas a manipular um único contato da agenda. Como descrito anteriormente, este endpoint disponibiliza as funcionalidades para consulta, alteração e remoção de recursos.

ContatoEndpoint.java - consulta a um contato específico

```
1  @Path("contato")
2  public class ContatoEndpoint {
3
4      @Autowired
5      private RegrasContatos regrasContatos;
6
7      @QueryParam("idContato")
8      private String idContato;
9
10     @GET
11     @Produces(MediaType.APPLICATION_JSON)
12     public Response obterContato() {
13         Contato contato = regrasContatos.consultar(idContato);
14         return Response.ok(new ContatoRep(contato)).build();
15     }
16
17     @PUT
18     @Produces(MediaType.APPLICATION_JSON)
19     @Consumes(MediaType.APPLICATION_JSON)
20     public Response alterarContato(ContatoRep contato) {...}
21
22     @DELETE
23     @Produces(MediaType.TEXT_PLAIN)
24     public Response removerContato() {...}
25 }
```

Na linha 1, a URL *contato* é associada ao endpoint. As linhas 4 e 5 definem o ponto de injeção do objeto responsável pelas regras de negócio. O contato é especificado através de seu identificador, definido pela propriedade *idContato*. O identificador é informado através de *QueryParam*, e a variável fica disponível a todos os métodos, como mostrado nas linhas 7 e 8. O parâmetro *idContato* poderia utilizar *PathParam*, *entretanto, foi escolhido o formato *QueryParam* apenas como uma opção. Além disso, a declaração da variável *idContato* e a anotação de mapeamento *QueryParam* podem ser realizadas na assinatura do método.

Entre as linhas 10 e 15 é implementado o método de consulta aos dados do contato. O método recebe a anotação *@GET* além definir o JSON como formato da representação retornada. Em seguida, é invocado o método *consultar* disponibilizado pelo objeto de negócio, que retorna um objeto de

domínio correspondente ao contato desejado. Por fim, é retornada a representação do recurso a partir do objeto de domínio. Os demais métodos (linhas 17 a 25) serão implementados nas próximas seções.

A classe *GerenciadorContatos* implementa a regra de negócio para esta consulta. O método *consultar* não aplica nenhuma restrição de negócio, apenas solicita à camada de persistência que consulte as informações do contato e de seu respectivo endereço. Por fim, o endereço é associado ao contato e retornado ao endpoint.

GerenciadorContatos.java - consulta a um contato específico

```
1  @Component
2  public class GerenciadorContatos implements RegrasContatos {
3      @Override
4      public Contato consultar(String idContato) {
5          Contato contato = contatoDao.consultar(idContato);
6          Endereco endereco = enderecoDao.consultar(idContato);
7          contato.setEndereco(endereco);
8          return contato;
9      }
10     //demais metodos omitidos
11 }
```

A implementação dos métodos que consultam as informações do contato e do endereço no banco de dados é realizada nas classes *JdbcContatoDao* e *JdbcEnderecoDao*, respectivamente. Primeiramente, é construído o comando SQL de consulta e associado o parâmetro para identificador do contato. Em seguida, é executado o método *queryForObject* disponibilizado pelo *jdbcTemplate*, que retorna um único objeto. O registro retornado pelo banco de dados é manipulado por um *RowMapper*, responsável por construir um objeto de domínio e popular os atributos com os dados do *resultset*.

JdbcContatoDao.java - consulta a um contato específico

```
@Component
public class JdbcContatoDao implements ContatoDao {

    @Autowired
    private NamedParameterJdbcTemplate jdbcTemplate;

    @Override
    public Contato consultar(String idContato) {
        StringBuilder sql = new StringBuilder();
        sql.append("select * ");
        sql.append("from agenda.contato ");
        sql.append("where id = :id");

        MapSqlParameterSource params = new MapSqlParameterSource("id", idContato);

        return jdbcTemplate.queryForObject(sql.toString(), params, new RowMapper<Contato>() {
            @Override
            public Contato mapRow(ResultSet rs, int rowNum) throws SQLException {
                Contato contato = new Contato();
                contato.setId(rs.getString("id"));
                contato.setNome(rs.getString("nome"));
                contato.setEmail(rs.getString("email"));
                contato.setCpf(rs.getString("cpf"));
                contato.setTelefone(rs.getString("telefone"));
                contato.setDataNascimento(rs.getDate("data_nascimento"));
                return contato;
            }
        });
    }

    //demais metodos omitidos
}
```

JdbcEnderecoDao.java - consulta a um contato específico

```

@Component
public class JdbcEnderecoDao implements EnderecoDao {

    @Autowired
    private NamedParameterJdbcTemplate jdbcTemplate;

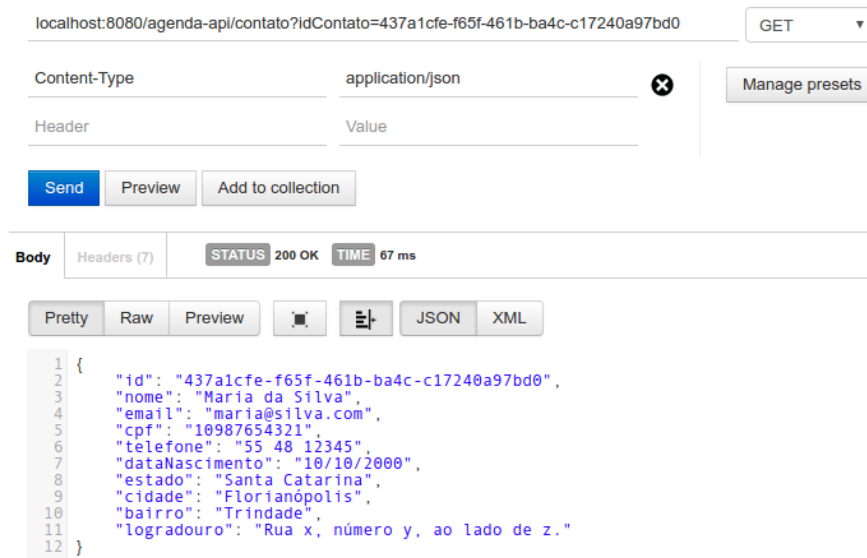
    @Override
    public Endereco consultar(String idContato) {
        StringBuilder sql = new StringBuilder();
        sql.append("select * ");
        sql.append("from agenda.endereco ");
        sql.append("where id_contato= :id");

        MapSqlParameterSource params = new MapSqlParameterSource("id", idContato);

        return jdbcTemplate.queryForObject(sql.toString(), params, new RowMapper<Endereco>() {
            @Override
            public Endereco mapRow(ResultSet rs, int rowNum) throws SQLException {
                Endereco endereco = new Endereco();
                endereco.setBairro(rs.getString("bairro"));
                endereco.setCidade(rs.getString("cidade"));
                endereco.setEstado(rs.getString("estado"));
                endereco.setLogradouro(rs.getString("logradouro"));
                return endereco;
            }
        });
    }
}

```

Para testar a funcionalidade vamos executar uma requisição no Postman com HTTP GET aplicada sobre a URL *localhost:8080/agenda-api/contato*, adicionando o *queryParam idContato* com o identificador desejado. É importante adicionar o cabeçalho HTTP *Content-Type* configurado para *application/json*.



Requisição para listar um contato específico

Alteração e Remoção

As últimas funcionalidades que faltam ser implementadas são a alteração e a remoção dos contatos cadastrados. Vamos começar com a implementação da classe *ContatoEndpoint*. O identificador do contato é atribuído à variável *idContato* por meio da anotação *QueryParam*, utilizada por ambos os métodos. Além disso, os dois métodos respeitam a semântica do protocolo HTTP e utilizam *PUT* para alteração e *DELETE* para remoção de recursos.

O método *alterarContato* recebe do cliente uma representação com os dados atualizados do contato. Note que o identificador obtido a partir do atributo associado ao *QueryParam* deve ser atribuído ao objeto e convertido para o modelo de domínio antes de ser repassado à camada de negócio. O método *removerCliente* apenas utiliza o identificador para solicitar a remoção do contato. Ao remover um contato, a Web API retorna apenas uma mensagem de sucesso, sendo assim, a anotação *@Produces* define o formato da representação como texto plano.

ContatoEndpoint.java - alteração e remoção

```
@Path("contato")
public class ContatoEndpoint {

    @Autowired
    private RegrasContatos regrasContatos;

    @QueryParam("idContato")
    private String idContato;

    //metodo de obtencao de um cliente omitido

    @PUT
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_JSON)
    public Response alterarContato(ContatoRep contato) {
        contato.setId(idContato);
        Contato contatoDominio = contato.converterParaDominio();
        regrasContatos.alterar(contatoDominio);
        return Response.ok(new ContatoRep(contatoDominio)).build();
    }

    @DELETE
    @Produces(MediaType.TEXT_PLAIN)
    public Response removerContato() {
        regrasContatos.remover(idContato);
        return Response.ok("Contato removido com sucesso").build();
    }
}
```

A classe *GerenciadorContatos* implementa os métodos da camada de negócio para alteração e remoção. Ambos os métodos recebem a anotação *@Transactional*, pois deve-se garantir que os dados do cliente e do endereço sejam alterados ou removidos de forma atômica. A camada de negócio apenas solicita serviços da camada de persistência. Entretanto poderiam ser implementadas restrições para garantir determinadas regras de negócio.

A camada de persistência não oferece o serviço de alteração de endereço. Sendo assim, a alteração deve remover o endereço e depois cadastra-lo novamente. Esta é uma característica importante, pois as funcionalidades disponibilizadas pelas diferentes camadas não precisam implementar necessariamente as mesmas funcionalidades. Cada camada disponibiliza serviços de forma relativamente independente.

GerenciadorContatos.java - alteração e remoção

```
@Component
public class GerenciadorContatos implements RegrasContatos {

    @Autowired
    private ContatoDao contatoDao;

    @Autowired
    private EnderecoDao enderecoDao;

    @Override
    @Transactional
    public void alterar(Contato contato) {
        contatoDao.alterar(contato);
        enderecoDao.remover(contato.getId());
        enderecoDao.cadastrar(contato.getEndereco(), contato.getId());
    }

    @Override
    @Transactional
    public void remover(String idContato) {
        enderecoDao.remover(idContato);
        contatoDao.remover(idContato);
    }
    //demais metodos omitidos
}
```

A classe *JdbcContatoDao* implementa os métodos que manipulam o banco de dados para alterar e para remover um contato. Os dois métodos utilizam o mesmo princípio: primeiramente é construído o comando SQL com base nas variáveis do objeto de domínio; em seguida é criado um mapa com os parâmetros; por fim, o comando SQL é executado com base no mapa dos parâmetros. A remoção de um endereço, funcionalidade implementada pela classe *JdbcEnderecoDao*, segue o mesmo procedimento.

JdbcContatoDao.java - alteração e remoção

Component

```
public class JdbcContatoDao implements ContatoDao {

    @Autowired
    private NamedParameterJdbcTemplate jdbcTemplate;

    @Override
    public void alterar(Contato contato) {
        StringBuilder sql = new StringBuilder();
        sql.append("update agenda.contato set ");
        sql.append("nome= :nome, ");
        sql.append("email= :email, ");
        sql.append("cpf= :cpf, ");
        sql.append("telefone= :telefone, ");
        sql.append("data_nascimento= :dataNascimento ");
        sql.append("where id=:id");

        Map<String, Object> parametros = new HashMap<>();
        parametros.put("id", contato.getId());
        parametros.put("nome", contato.getNome());
        parametros.put("email", contato.getEmail());
        parametros.put("cpf", contato.getCpf());
        parametros.put("telefone", contato.getTelefone());
        parametros.put("dataNascimento", contato.getDataNascimento());

        jdbcTemplate.update(sql.toString(), parametros);
    }

    @Override
    public void remover(String idContato) {
        StringBuilder sql = new StringBuilder();
        sql.append("delete from agenda.contato ");
        sql.append("where id = :id");
        MapSqlParameterSource params = new MapSqlParameterSource("id", idContato);
        jdbcTemplate.update(sql.toString(), params);
    }
    //demais metodos omitidos
}
```

JdbcEnderecoDao.java - remoção


@Component

```
public class JdbcEnderecoDao implements EnderecoDao {
```

@Override

```
public void remover(String idContato) {  
    StringBuilder sql = new StringBuilder();  
    sql.append("delete from agenda.endereco ");  
    sql.append("where id_contato = :idContato");  
    MapSqlParameterSource params = new MapSqlParameterSource("id", idContato);  
    jdbcTemplate.update(sql.toString(), params);  
}  
  
//demais metodos omitidos  
}
```

localhost:8080/agenda-api/contato?idContato=437a1cfe-f65f-461b-ba4c-c17240a97bd0 PUT [URL params](#)

Content-Type application/json  [Manage presets](#)



Header	Value
--------	-------

form-data x-www-form-urlencoded raw **JSON**

```
1 {  
2   "nome": "Maria da Glória e Silva",  
3   "email": "maria.gloriasilva@silva.com",  
4   "cpf": "10987654321",  
5   "telefone": "55 48 12345",  
6   "dataNascimento": "10/10/1980",  
7   "estado": "Paraná",  
8   "cidade": "Curitiba",  
9   "bairro": "Batel",  
10  "logradouro": "Rua a, número 1, ao lado de b."  
11 }
```

[Send](#) [Preview](#) [Add to collection](#)

Body Headers (8) **STATUS** 200 OK **TIME** 50 ms

Pretty Raw Preview   **JSON** XML

```
1 {  
2   "id": "437a1cfe-f65f-461b-ba4c-c17240a97bd0",  
3   "nome": "Maria da Glória e Silva",  
4   "email": "maria.gloriasilva@silva.com",  
5   "cpf": "10987654321",  
6   "telefone": "55 48 12345",  
7   "dataNascimento": "10/10/1980",  
8   "estado": "Paraná",  
9   "cidade": "Curitiba",  
10  "bairro": "Batel",  
11  "logradouro": "Rua a, número 1, ao lado de b."  
12 }
```

Requisição para alterar um contato



localhost:8080/agenda-api/contato?idContato=437a1cfe-f65f-461b-ba4c-c17240a97bd0 DELETE ▾

form-data x-www-form-urlencoded raw

Key Value Text ▾

Send Preview Add to collection

Body Headers (8) STATUS 200 OK TIME 81 ms

Pretty Raw Preview   JSON XML

1 Contato removido com sucesso

Requisição para remover um contato

Tratamento de Exceções

Quando desenvolvemos sistemas, temos em mente que tudo irá funcionar perfeitamente. Entretanto, não podemos ignorar o fato que erros e situações não planejadas podem e irão acontecer. Quando o sistema atinge um estado de não conformidade, por exemplo: um erro de execução de um comando SQL ou alguma informação inválida proveniente do usuário, são lançadas exceções, que se não tratadas adequadamente se transformam em erros do sistema. Este capítulo apresenta como definir as exceções de negócio e como tratar os erros do sistema.

Criação de Exceções de Negócio

Exceções de negócios são lançadas quando alguma restrição do próprio domínio da aplicação não é respeitada. Vamos criar a seguinte restrição de negócio: “*Apenas contatos com idade igual ou superior a 18 anos podem ser cadastrados*”. Quando solicitado o cadastro de um contato que não atenda a esta regra, deverá ser lançada uma exceção.

A classe *IdadeContatoException* implementa uma exceção relacionada à idade mínima para o cadastro de contatos. Sem entrar no mérito de exceções *cheçadas* ou *não cheçadas*, vamos implementar as exceções através da herança de *RuntimeException*. Neste projeto, as classes de exceção são agrupadas no pacote *negocio*.

IdadeContatoException.java

```
public class IdadeContatoException extends RuntimeException {  
    public IdadeContatoException(String msg) {  
        super(msg);  
    }  
}
```

Vamos implementar agora a verificação da data de nascimento no momento em que o contato é cadastrado e alterado. A classe *GerenciadorContatos* mostra a implementação do método privado *validarDataNascimento*, que calcula a quantidade de anos entre a data de nascimento do contato e a data atual, lançando a exceção caso a diferença seja menor que a idade mínima estabelecida. Os métodos *cadastrear* e *alterar* incluem agora a verificação da data de nascimento.

GerenciadorContatos.java

```

@Component
public class GerenciadorContatos implements RegrasContatos {

    private final int IDADE_MINIMA = 18;

    @Autowired
    private ContatoDao contatoDao;

    @Autowired
    private EnderecoDao enderecoDao;

    @Override
    @Transactional
    public void cadastrar(Contato contato) {
        validarDataNascimento(contato.getDataNascimento());
        String idContato = UUID.randomUUID().toString();
        contato.setId(idContato);
        contatoDao.cadastrar(contato);
        enderecoDao.cadastrar(contato.getEndereco(), idContato);
    }

    @Override
    @Transactional
    public void alterar(Contato contato) {
        validarDataNascimento(contato.getDataNascimento());
        contatoDao.alterar(contato);
        enderecoDao.remover(contato.getId());
        enderecoDao.cadastrar(contato.getEndereco(), contato.getId());
    }

    private void validarDataNascimento(Date dataNascimento) {
        DateTime dateTimeDn = new DateTime(dataNascimento);
        DateTime hoje = new DateTime();
        int idade = Years.yearsBetween(dateTimeDn, hoje).getYears();
        if (idade < IDADE_MINIMA) {
            String msgErro = "Contato com menos de %s anos";
            msgErro = String.format(msgErro, IDADE_MINIMA);
            throw new IdadeContatoException(msgErro);
        }
    }

    //demais metodos omitidos
}

```

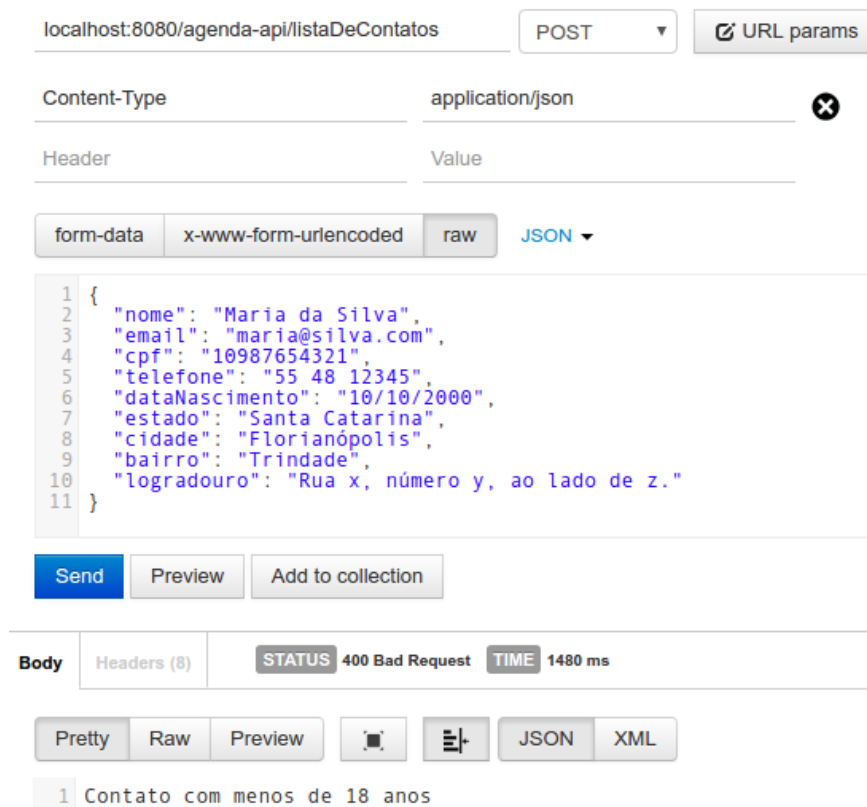
Uma vez lançada, a exceção precisa ser tratada para disponibilizar uma resposta adequada ao cliente da aplicação. Uma forma é envolver a chamada do método de negócio com *try/catch*. O exemplo a

seguir mostra a modificação na classe *ListaContatosEndpoint*, necessária para tratar a exceção no momento de cadastramento do contato. O “caminho feliz” é implementado no escopo *try*, enquanto *catch* contém o código que será executado caso a exceção seja lançada.

ListaContatosEndpoint.java - try/catch

```
@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Response cadastrarContato(ContatoRep contato) {
    try {
        Contato contatoDominio = contato.converterParaDominio();
        regrasContatos.cadastrar(contatoDominio);
        ContatoRep contatoCadastrado = new ContatoRep(contatoDominio);
        return Response.ok(contatoCadastrado).build();
    }
    catch (IdadeClienteException e) {
        return Response.status(Status.BAD_REQUEST).entity(e.getMessage()).build();
    }
}
```

O tratamento de exceções no contexto de Web APIs implica em retornar uma resposta adequada ao cliente, considerando a semântica do protocolo de comunicação utilizado. No caso do protocolo HTTP, a exceção de idade mínima para o cadastro do contato é resultado de uma informação inválida proveniente do próprio cliente. Uma boa forma de informar ao cliente que ele está enviando informações inválidas é através de uma resposta com o código 400 (HTTP BAD REQUEST). Sendo assim, aplicações clientes de Web APIs devem ser capazes de interpretar corretamente os códigos retornados, resultando em sistemas mais robustos e confiáveis aos usuários finais. Por fim, a figura a seguir mostra a execução de uma requisição contendo dados inválidos e o resultado retornado pela Web API.



Requisição com dados inválidos

Implementação de Providers

Tratar exceções diretamente no endpoint pode tornar o código pouco legível e agradável. Entretanto, é possível utilizar *Providers* para mapear e tratar as exceções lançadas durante a execução da aplicação. A classe *IdadeContatoExceptionHandler* mostra a implementação do Provider responsável por tratar as exceções relacionadas à idade do contato. Primeiramente, a classe precisa ser anotada com `@Provider`. Além disso, a classe deve implementar a interface *ExceptionHandler* para uma determinada exceção ou hierarquia de exceções. Por fim, a resposta adequada para a exceção é construída no método *toResponse*. Por se tratar de uma configuração da Web API, os providers foram agrupados no pacote *config*. Os providers ou seus pacotes devem ser registrados no arquivo de configuração do Jersey, conforme mostrado em *JerseyConfig.java*. Dessa forma, o bloco *try/catch* pode ser retirado sem prejuízo ao funcionamento da aplicação.

IdadeContatoException.java

```

@Provider
public class IdadeContatoExceptionHandler
    implements ExceptionMapper<IdadeContatoException> {

    @Override
    public Response toResponse(IdadeContatoException exception) {
        return Response.status(Status.BAD_REQUEST).entity(exception.getMessage()).build();
    }
}

```

JerseyConfig.java - registro de provider

```

@Component
@ApplicationPath("/agenda-api")
public class JerseyConfig extends ResourceConfig {

    public JerseyConfig() {
        this.register(RequestContextFilter.class);
        this.packages("br.com.exemplo.agenda.api.endpoint");
        this.register(IdadeContatoExceptionHandler.class);
    }
}

```

Contato Não Encontrado

Uma situação muito comum em Web APIs é a inexistência de um recurso solicitado pelo cliente. Ao consultar ou solicitar alterações de um contato inexistente, deve-se retornar uma resposta adequada ao cliente. Para tratar este problema, vamos implementar uma classe de exceção para esta situação.

ContatoNaoEncontradoException.java

```

package br.com.exemplo.agenda.api.negocio;

public class ContatoNaoEncontradoException extends RuntimeException {
    public ContatoNaoEncontradoException(String msg) {
        super(msg);
    }
}

```

A identificação de recursos inexistentes é realizada diretamente na classe de manipulação do banco de dados, uma vez que o Spring JDBC proporciona mecanismos para facilitar o tratamento de exceções. O arquivo *JdbcContatoDao.java* mostra o ponto de tratamento das exceções quando um

contato não é encontrado na base de dados. Quando uma instrução SQL é executada através do método *queryForObject*, espera-se que um registro seja retornado, caso contrário, será lançada a exceção do tipo *IncorrectResultSizeDataAccessException*. Dessa forma, o método *consultar* trata esta exceção e lança uma exceção de negócio em seu lugar. No caso da alteração e remoção, o número de registros afetados por uma instrução SQL é retornando pelo método *update*. Dessa forma, é possível verificar se a alteração ou remoção de um recurso foi realizada. Caso o número de registros afetados seja zero, será lançada uma exceção de contato não encontrado.

JdbcContatoDao.java - tratamento de recursos inexistentes

```

@Component
public class JdbcContatoDao implements ContatoDao {
    @Override
    public Contato consultar(String idContato) {
        //codigo omitido
        try {
            return jdbcTemplate.queryForObject(... { // codigo omitido});
        } catch (IncorrectResultSizeDataAccessException e) {
            String msgErro = "Contato nao encontrado";
            throw new ContatoNaoEncontradoException(msgErro);
        }
    }

    @Override
    public void alterar(Contato contato) {
        //codigo omitido
        int update = jdbcTemplate.update(sql.toString(), parametros);
        if (update == 0) {
            throw new ContatoNaoEncontradoException("Contato nao encontrado");
        }
    }

    @Override
    public void remover(String idContato) {
        //codigo omitido
        int removido = jdbcTemplate.update(sql.toString(), parametros);
        if (removido == 0) {
            String msgErro = "Contato nao encontrado";
            throw new ContatoNaoEncontradoException(msgErro);
        }
    }
}

```

A classe *ContatoNaoEncontradoExceptionHandler* mostra a implementação do provider responsável por tratar as exceções lançadas quando um contato não for encontrado na base de dados. O provider retorna uma resposta com o código HTTP 404 NOT FOUND, informando adequadamente

à aplicação cliente que o recurso solicitado não existe. Não se esqueça de registrar este provider em *JerseyConfig.java*.

ContatoNaoEncontradoExceptionHandler.java

```
@Provider
public class ContatoNaoEncontradoExceptionHandler
    implements ExceptionMapper<ContatoNaoEncontradoException> {
    @Override
    public Response toResponse(ContatoNaoEncontradoException exception) {
        return Response.status(Status.NOT_FOUND).entity(exception.getMessage()).build();
    }
}
```

The screenshot shows a REST client interface. The URL bar contains `localhost:8080/agenda-api/contato?idContato=idInvalido` and the method is set to `GET`. The `Content-Type` header is `application/json`. Below the header section are buttons for `Send`, `Preview`, and `Add to collection`. The response section shows a `STATUS 404 Not Found` and `TIME 544 ms`. The response body is displayed in a table with one row: `1 Contato nao encontrado`. The response is formatted as `JSON`.

Requisição de um recurso inexistente

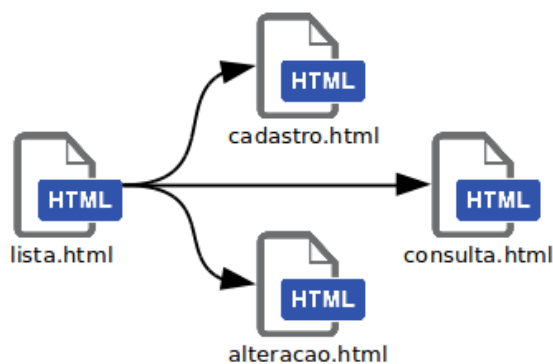
Aplicação Cliente da Web API

Com o objetivo de oferecer uma visão mais completa e prática do uso de Web APIs, este capítulo apresenta os detalhes de desenvolvimento de uma aplicação cliente que interage com a Web API desenvolvida no decorrer deste livro.

Visão Geral

De forma geral, Web APIs não são manipuladas diretamente pelos usuários finais, mas por aplicações intermediárias, conhecidas como aplicações clientes. Estas aplicações se comunicam com uma ou mais Web APIs e oferecem uma interface gráfica adequada ao usuário final do sistema. Existem diversas tecnologias para o desenvolvimento de clientes de Web APIs. Dentre as mais comuns destacam-se as aplicações desktop desenvolvidas com diferentes linguagens de programação, aplicações nativas ou híbridas para dispositivos móveis e aplicações Web. Para este exemplo, vamos desenvolver uma aplicação Web com HTML e JQuery com Ajax, pois é uma opção muito utilizada e relativamente simples de ser implementada.

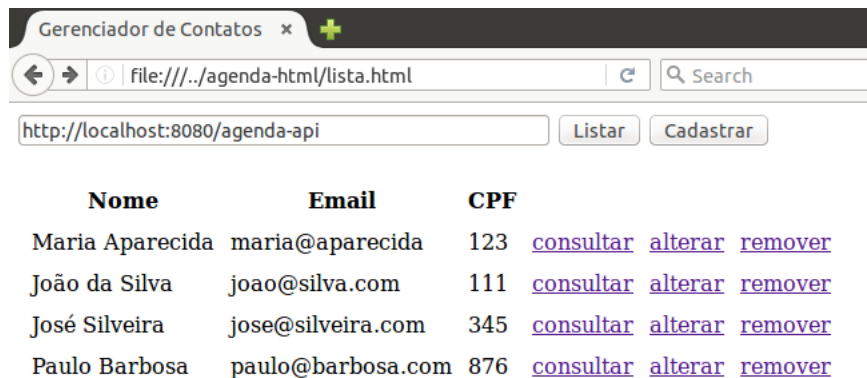
Neste capítulo vamos implementar uma aplicação cliente capaz de se comunicar com a Web API de contatos que desenvolvemos anteriormente. A aplicação cliente é constituída por quatro páginas HTML. A partir da *lista.html*, que apresenta todos os contatos cadastrados, é possível cadastrar novos contatos em *cadastro.html*, consultar todas as informações de um contato específico em *consulta.html* ou modificar os dados em *alteracao.html*. A remoção de um contato não exige uma página dedicada, sendo realizada diretamente na página da listagem.



Arquivos da aplicação cliente

Listagem dos Contatos

Para a aplicação cliente, a listagem significa consultar a Web API e apresentar as informações retornadas em uma página HTML. As informações dos contatos podem ser facilmente apresentadas em uma tabela. Além de algumas propriedades, cada linha da tabela apresenta também controles (links) para consultar mais informações, alterar e remover.



Aplicação cliente - listagem dos contatos

No topo da página HTML existe um campo de texto que define a URL da Web API, além dos botões de listar e cadastrar. Embora especificar o endereço da Web API pareça irrelevante inicialmente, este mecanismo permite que uma única aplicação cliente se comunique com diversas Web APIs de agenda. Isto é desejável em cenários onde existe um grande nível de distribuição de dados, por exemplo: departamentos, filiais ou parceiros que possuem sua própria instância da Web API, e os dados precisam ser reunidos em uma aplicação cliente.

O arquivo *lista.html* apresenta o código da página HTML necessário para representar as informações dos contatos. As páginas HTML são capazes apenas de apresentar as informações desejadas, a interação com a Web API deve ser feita com javascript. Neste exemplo, vamos utilizar o apoio do JQuery. Sendo assim, deve-se primeiro obter o arquivo JQuery² e importá-lo na página (linha 6). Além disso, o código javascript responsável pela interação com a Web API é desenvolvido em um arquivo separado (lista.js) e também importado pela página (linha 7).

²<https://jquery.com/download/>

lista.html

```

1 <html>
2 <head>
3   <title>Gerenciador de Contatos</title>
4   <meta charset="UTF-8">
5   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6   <script src="jquery/jquery-2.1.4.min.js"></script>
7   <script src="lista.js"></script>
8 </head>
9 <body>
10  <input id='apiPath' style='width: 400px;' value='http://localhost:8080/agenda-api'>
11  <button id='botaoListarTodos' type="button">Listar</button>
12  <button id='botaoCadastrar' type="button">Cadastrar</button>
13  <br><br>
14  <div id="listaDeContatos">
15    <table id="tabelaContatos" border="0" cellspacing="10">
16      <thead> <tr>
17        <th>Nome</th>
18        <th>Email</th>
19        <th>CPF</th>
20        <th></th>
21        <th></th>
22        <th></th>
23      </tr> </thead>
24      <tbody></tbody>
25    </table>
26  </div>
27 </body>
28 </html>

```

O arquivo *lista.js - parte 1* mostra a implementação responsável por realizar a consulta dos contatos cadastrados na Web API. O arquivo começa com a função *listarTodosContatos*, que realiza uma requisição HTTP com ajax. Com base na URL da Web API, são criadas duas variáveis, uma para montar a URL da lista, e outra para criar a URL de acesso aos contatos (linhas 2 e 3). Entre as linhas 6 e 9 são definidos os detalhes da requisição ajax.

Basicamente duas coisas podem acontecer em uma requisição ajax com JQuery, sucesso ou erro. Em caso de sucesso, será atribuída à variável *contatos* a lista de registros retornados pela Web API, que serão iterados e apresentados em uma linha da tabela HTML. Primeiramente, é executada a função *limparTabela* (*lista.js - parte2*) para garantir que a tabela está vazia. Através do comando *\$.each*, todos os registros são iterados e transformados em uma nova linha, que é adicionada à tabela HTML. As funcionalidades *consultar*, *alterar* e *remover* são disponibilizadas através de links associados com cada linha da tabela. Funções são associadas aos links, que recebem a URL do contato e executam as funcionalidades desejadas.

lista.js - parte 1

```
1 var listarTodosContatos = function() {
2   pathLista = $('#apiPath').val()+"/listaDeContatos";
3   pathContato = $('#apiPath').val()+"/contato?idContato=";
4
5   $.ajax({
6     url: pathLista,
7     type: 'GET',
8     async: true,
9     contentType: 'application/json',
10    success: function(contatos) {
11      limparTabela();
12      $.each(contatos, function(index, contato) {
13        var novaLinha =
14          '<tr>' +
15          '<td>' + contato.nome + '</td>' +
16          '<td>' + contato.email + '</td>' +
17          '<td>' + contato.cpf + '</td>' +
18          '<td><a href="#" onclick=consultar("' + pathContato + contato.id + "')>consultar</a></td>' +
19          '<td><a href="#" onclick=alterar("' + pathContato + contato.id + "')>alterar</a></td>' +
20          '<td><a href="#" onclick=remover("' + pathContato + contato.id + "')>remover</a></td>' +
21          '</tr>';
22        $("#tabelaContatos tr:last").after(novaLinha);
23      });
24    },
25    error: function() { }
26  });
27  };
```

As funções *consultar* e *alterar* recebem a URL do contato como parâmetro de entrada, que é armazenada na área de memória local do navegador, denominada *sessionStorage*. Os itens armazenados nesta área de memória podem ser acessados mesmo quando ocorre troca de página. Dessa forma, as funções de consulta e alteração direcionam o usuário para outras páginas e recuperam o item armazenado na memória local do navegador. Mais adiante será apresentada a recuperação do item. A função *remover* também recebe a URL do contato como parâmetro para executar uma requisição HTTP DELETE, que remove o contato na Web API. Note que a funcionalidade de remoção é implementada no mesmo arquivo javascript da listagem, uma vez que não exige uma página dedicada. Entretanto, mecanismos mais elaborados para remoção poderiam ser implementados, como por exemplo a confirmação de remoção.

lista.js - parte 2

```
1  var limparTabela = function() {
2    $("#tabelaContatos").find("tr:gt(0)").remove();
3  }
4
5  var consultar = function(urlContato) {
6    sessionStorage.setItem("urlContato", urlContato);
7    window.location.href = "consulta.html";
8  }
9
10 var alterar = function(urlContato) {
11   sessionStorage.setItem("urlContato", urlContato);
12   window.location.href = "alteracao.html";
13 }
14
15 var remover = function(urlContato) {
16   $.ajax({
17     url: urlContato,
18     type: 'DELETE',
19     async: true,
20     success: function() {
21       listarTodosContatos();
22     }
23   });
24 }
25
26 $(document).ready(function() {
27   $("#botaoListarTodos").click(function() {
28     listarTodosContatos();
29   });
30
31   $("#botaoCadastrar").click(function() {
32     window.location.href = "cadastro.html";
33   });
34 });
```

O Problema de Cross Origin Request

Ao realizar o teste da listagem de contatos, nos deparamos com uma mensagem de erro. Isto ocorre devido ao fato da página da listagem (*file://*) estar em um domínio diferente da Web API (*localhost:8080*). Por motivos de segurança, navegadores restringem a execução de requisições entre diferentes domínios realizadas através de scripts. Para solucionar este problema é possível implementar o mecanismo de *Cross-Origin Resource Sharing* (CORS), que permite à Web API controlar o acesso de recursos de diferentes domínios. Dessa forma, deve-se implementar o CORS

diretamente na Web API. No pacote *config* do projeto da Web API, crie um provider de acordo com o arquivo *CorsInterceptor.java*, que deve ser registrado em *JerseyConfig.java*. A partir deste momento, a aplicação cliente deve ser capaz de interagir corretamente com a Web API.



Erro - Cross origin request

CorsInterceptor.java

```
package br.com.exemplo.agenda.api.config;

@Provider
public class CorsInterceptor implements ContainerResponseFilter {
    private final Integer corsPreflightMaxAgeInSeconds = 30 * 60;

    @Override
    public void filter(ContainerRequestContext req, ContainerResponseContext resp)
        throws IOException {
        resp.getHeaders().add("Access-Control-Allow-Origin", req.getHeaderString("origin"));
        resp.getHeaders().add("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");
        resp.getHeaders().add("Access-Control-Allow-Credentials", "true");
        List<String> allowedHeaders = req.getHeaders().get("Access-Control-Request-Headers");
        if (allowedHeaders != null) {
            for (String allowedHeader : allowedHeaders) {
                resp.getHeaders().add("Access-Control-Allow-Headers", allowedHeader);
            }
        }
        resp.getHeaders().add("Access-Control-Max-Age", this.corsPreflightMaxAgeInSeconds);
    }
}
```

Cadastro

O cadastro dos contatos é realizado pela página *cadastro.html* e pelo arquivo *cadastro.js*. Primeiramente, são definidos os elementos HTML que compõem a página, juntamente com algumas instruções de estilo css que atuam na aparência da página (é desejável que o css seja implementado em um arquivo distinto e importado na página html). Note que os campos de texto e seus rótulos não estão inseridos em um formulário HTML, como ocorre em aplicações tradicionais. Uma vez que o JQuery é o responsável pela realização da requisição via ajax, nenhum formulário precisa ser submetido. As informações escritas nos campos de texto são obtidas com base no identificador dos elementos. Estas informações são utilizadas para construir um documento JSON que é enviado para a Web API para o cadastro de um novo contato.



Gerenciador de Contatos x +

file:///.../agenda-html/cadastro.html Search » ≡

Cadastro de Contato

Dados do contato

Nome:

Email:

CPF:

Telefone:

Data Nascimento:

Endereço

Estado:

Cidade:

Bairro:

Logradouro:

Aplicação cliente - cadastro de contato

cadastro.html

```
<html>
<head>
  <title>Gerenciador de Contatos</title>
  <meta charset="UTF-8">
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <script src="jquery/jquery-2.1.4.min.js"></script>
  <script src="cadastro.js"></script>
  <style>
    label {
      float: left;
      width: 120px;
      clear: both;
      margin: 2px;
    }
    input {
      width: 250px;
      clear: both;
      margin: 2px;
    }
  </style>
</head>
<body>
  <h1>Cadastro de Contato</h1>
  <div id="dadosContato">
    <h3>Dados do contato</h3>
    <label>Nome:</label><input id="nome" type="text"></input><br>
    <label>Email:</label><input id="email" type="text"></input><br>
    <label>CPF:</label><input id="cpf" type="text"></input><br>
    <label>Telefone:</label><input id="telefone" type="text"></input><br>
    <label>Data Nascimento:</label><input id="data" type="text"></input><br>
    <h3>Endereço</h3>
    <label>Estado:</label><input id="estado" type="text"></input><br>
    <label>Cidade:</label><input id="cidade" type="text"></input><br>
    <label>Bairro:</label><input id="bairro" type="text"></input><br>
    <label>Logradouro:</label><input id="logradouro" type="text"></input><br><br>
    <button id="botaoCadastrar">Cadastrar</button><br><br>
    <div id="resultado"></div>
  </div>
</body>
</html>
```

O arquivo *cadastro.js* implementa a funcionalidade para o cadastramento do contato. A função *cadastrear* obtém as informações inseridas pelo usuário e constrói o objeto *dadosContato* com os dados obtidos. Em seguida, é implementado o código que realiza uma requisição HTTP POST para a URL da Web API, contendo a representação JSON do documento *dadosContato*. A função *cadastrear* é invocada quando o usuário pressionar botão **botaoCadastrar*. Ao final da requisição, é apresentada ao usuário uma mensagem de sucesso ou erro de acordo com a resposta da Web API.

cadastro.js

```

var cadastrar = function() {
    var dadosContato = {
        nome: $("#nome").val(),
        email: $("#email").val(),
        cpf: $("#cpf").val(),
        telefone: $("#telefone").val(),
        dataNascimento: $("#datan").val(),
        estado: $("#estado").val(),
        cidade: $("#cidade").val(),
        bairro: $("#bairro").val(),
        logradouro: $("#logradouro").val()
    };
    $.ajax({
        url: "http://localhost:8080/agenda-api/listaDeContatos",
        type: 'POST',
        async: true,
        contentType: 'application/json',
        data: JSON.stringify(dadosContato),
        success: function() {
            $("#resultado").empty();
            $("#resultado").append("Contato cadastrado com sucesso")
        },
        error: function(xhr, status, error) {
            $("#resultado").empty();
            $("#resultado").append("Erro ao cadastrar: " + xhr.responseText)
        }
    });
};
$(document).ready(function() {
    $("#botaoCadastrar").click(function() {
        cadastrar();
    });
});

```

Consulta

A consulta busca as informações de um contato específico e apresenta na página *consulta.html*. O arquivo *consulta.js* implementa a requisição ajax que busca as informações na Web API. Note que no momento que página está pronta (contexto *\$(document).ready()*), a função *consultarContato* é invocada com a URL do contato como parâmetro de entrada. Entretanto, a URL é obtida a partir do *sessionStorage*, armazenada anteriormente pela página de listagem. Essa é uma forma de tornar acessíveis as informações que precisam ser acessadas entre troca de páginas. Uma vez que a função *consultarContato* obtém os dados da Web API, as informações são associadas aos elementos HTML de acordo com seus identificadores.



Aplicação cliente - consulta de um contato específico

consulta.html

```
1 <html>
2 <head>
3   <title>Gerenciador de Clientes</title>
4   <meta charset="UTF-8">
5   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6   <script src="jquery/jquery-2.1.4.min.js"></script>
7   <script src="consulta.js"></script>
8 </head>
9 <body>
10  <h1>Detalhes do Contato</h1>
11  <h3>Dados do contato</h3>
12  <b>Nome:</b><span id="nome"></span><br>
13  <b>Email:</b><span id="email"></span><br>
```

```

14 <b>CPF:</b><span id="cpf"></span><br>
15 <b>Telefone:</b><span id="telefone"></span><br>
16 <b>Data Nascimento:</b><span id="datan"></span><br>
17 <h3>Endereco</h3>
18 <b>Estado:</b><span id="estado"></span><br>
19 <b>Cidade:</b><span id="cidade"></span><br>
20 <b>Bairro:</b><span id="bairro"></span><br>
21 <b>Logradouro:</b><span id="logradouro"></span><br>
22 </body>
23 </html>

```

consulta.js

```

var consultarContato = function(urlContato) {
  $.ajax({
    url: urlContato,
    type: 'GET',
    async: true,
    contentType: 'application/json',
    success: function(contato) {
      $("#nome").text(contato.nome);
      $("#email").text(contato.email);
      $("#cpf").text(contato.cpf);
      $("#telefone").text(contato.telefone);
      $("#datan").text(contato.dataNascimento);
      $("#estado").text(contato.estado);
      $("#cidade").text(contato.cidade);
      $("#bairro").text(contato.bairro);
      $("#logradouro").text(contato.logradouro)
    },
    error: function() { }
  });
};

$(document).ready(function() {
  consultarContato(sessionStorage.getItem('urlContato'));
});

```

Alteração

A alteração do contato é realizada pela página *alteração.html*, que apresenta rótulos e campos de texto para que as informações de um contato sejam alteradas. No arquivo *alteracao.js*, duas funções são implementadas: *consultarContato* e *consultarContato*. A função de consulta é a mesma do exemplo anterior, que busca as informações do contato desejado para que suas informações sejam apresentadas nos campos de texto. Esta funcionalidade poderia ser reutilizada com algumas modificações no arquivo *consulta.js*, entretanto, para facilitar o entendimento, permitimos algumas duplicações de código. Quando o botão com o identificador *botaoAlterar* é pressionado, a função *alterar* é invocada. Esta função obtém os dados dos campos de texto, constrói um objeto JSON e realiza uma requisição HTTP PUT para a URL do contato armazenada no *sessionStorage*.

Gerenciador de Contatos x +

file:///.../agenda-html/alteracao.html Search » ≡

Alteração de Contato

Dados do contato

Nome:

Email:

CPF:

Telefone:

Data Nascimento:

Endereço

Estado:

Cidade:

Bairro:

Logradouro:

Aplicação cliente - alteração dos dados do contato

alteracao.html

```

<html>
<head>
  <title>Gerenciador de Contatos</title>
  <meta charset="UTF-8">
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <script src="jquery/jquery-2.1.4.min.js"></script>
  <script src="alteracao.js"></script>
  <style>
    label {
      float: left;
      width: 120px;
      clear: both;
      margin: 2px;
    }
    input {
      width: 250px;
      clear: both;
      margin: 2px;
    }
  </style>
</head>
<body>
  <h1>Alteração de Contato</h1>
  <div id="dadosContato">
    <h3>Dados do contato</h3>
    <label>Nome:</label> <input id="nome" type="text"></input> <br>
    <label>Email:</label> <input id="email" type="text"></input> <br>
    <label>CPF:</label> <input id="cpf" type="text"></input> <br>
    <label>Telefone:</label> <input id="telefone" type="text"></input> <br>
    <label>Data Nascimento:</label> <input id="datan" type="text"></input>
    <h3>Endereço</h3>
    <label>Estado:</label> <input id="estado" type="text"></input> <br>
    <label>Cidade:</label> <input id="cidade" type="text"></input> <br>
    <label>Bairro:</label> <input id="bairro" type="text"></input> <br>
    <label>Logradouro:</label> <input id="logradouro" type="text"></input><br><br>
    <button id="botaoAlterar">Alterar</button> <br><br>
    <div id="resultado"></div>
  </div>
</body>
</html>

```

alteracao.js

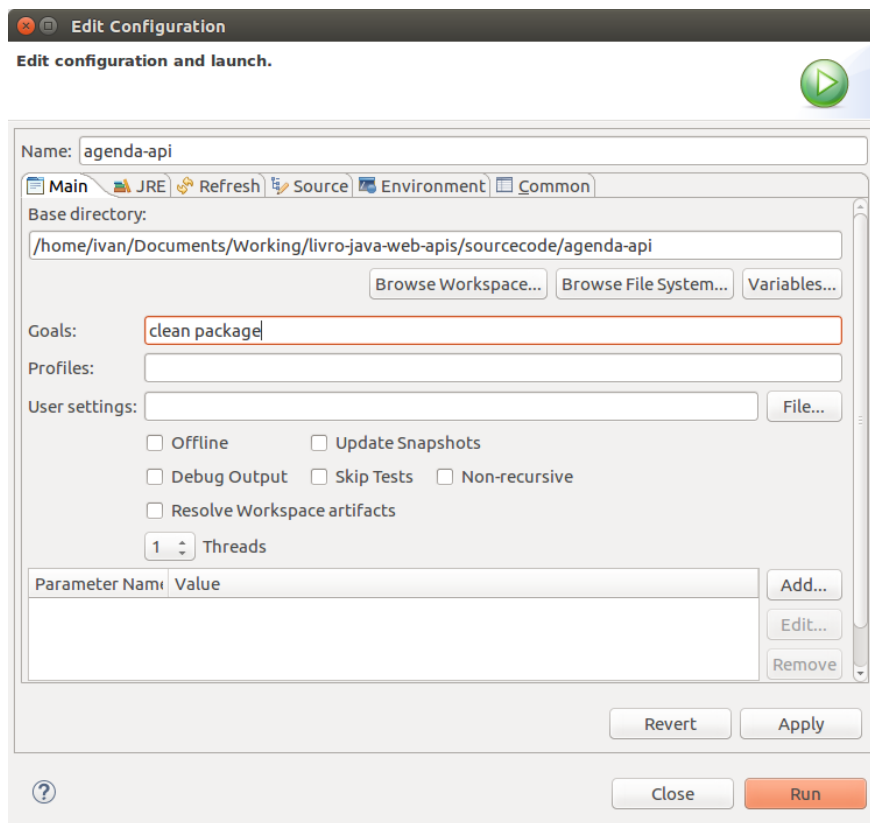
```
1 var consultarContato = function(urlContato) {
2     $.ajax({
3         url: urlContato,
4         type: 'GET',
5         async: true,
6         contentType: 'application/json',
7         success: function(contato) {
8             $("#nome").val(contato.nome);
9             $("#email").val(contato.email);
10            $("#cpf").val(contato.cpf);
11            $("#telefone").val(contato.telefone);
12            $("#datan").val(contato.dataNascimento);
13            $("#estado").val(contato.estado);
14            $("#cidade").val(contato.cidade);
15            $("#bairro").val(contato.bairro);
16            $("#logradouro").val(contato.logradouro)
17        },
18        error: function() {}
19    });
20 };
21
22 var alterar = function(urlContato) {
23     var dadosContato = {
24         nome: $("#nome").val(),
25         email: $("#email").val(),
26         cpf: $("#cpf").val(),
27         telefone: $("#telefone").val(),
28         dataNascimento: $("#datan").val(),
29         estado: $("#estado").val(),
30         cidade: $("#cidade").val(),
31         bairro: $("#bairro").val(),
32         logradouro: $("#logradouro").val()
33     };
34     $.ajax({
35         url: urlContato,
36         type: 'PUT',
37         async: true,
38         contentType: 'application/json',
39         data: JSON.stringify(dadosContato),
40         success: function() {
41             $("#resultado").empty();
42             $("#resultado").append("Contato alterado com sucesso")
43         },
44         error: function(xhr, status, error) {
45             $("#resultado").empty();
46             $("#resultado").append("Erro ao alterar: " + xhr.responseText)
```

```
47     }
48   });
49 };
50 $(document).ready(function() {
51   consultarContato(sessionStorage.getItem('urlContato'));
52   $("#botaoAlterar").click(function() {
53     alterar(sessionStorage.getItem('urlContato'));
54   });
55 });
```

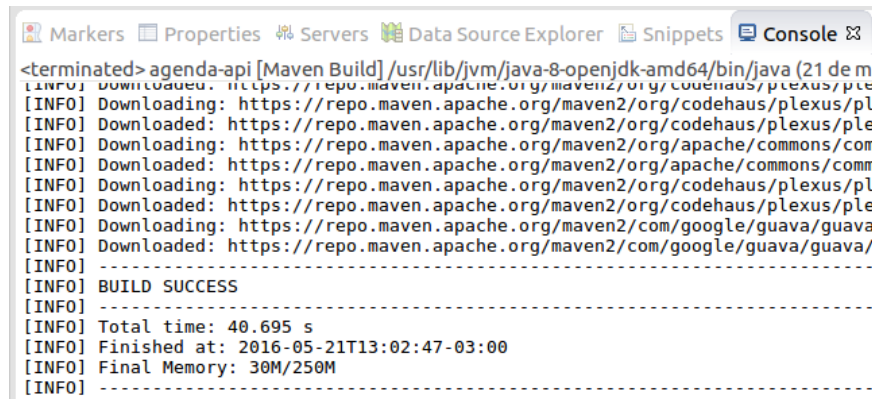
Construção e Implantação do Projeto

Até este momento a Web API foi executada dentro do ambiente de desenvolvimento. Este capítulo apresenta os passos para construir o projeto de forma a possibilitar que a Web API seja implantada em um ambiente de produção ou teste.

O primeiro passo é criar o artefato executável através do apache maven. Clique com o botão direito do mouse sobre o projeto, em seguida selecione o menu *Run As > Maven build....* No campo *Goals* digite *clean package*, em seguida pressione o botão *Run*. Este procedimento empacota o projeto em um arquivo JAR executável.



Configuração para construção do artefato executável



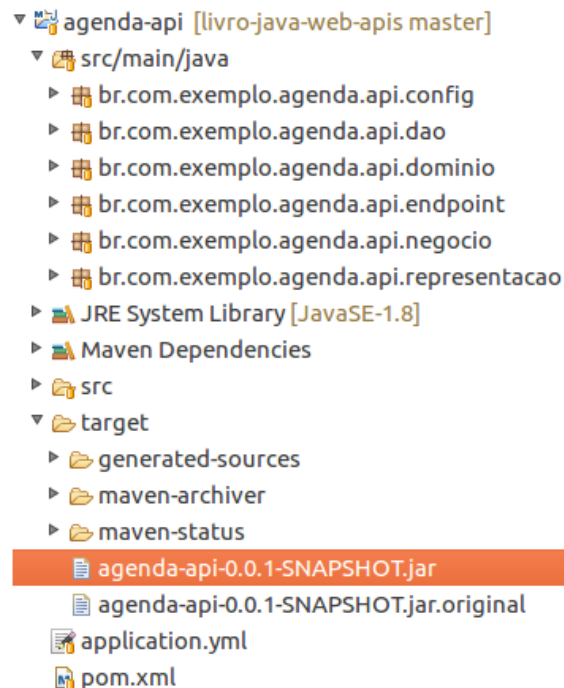
```

<terminated> agenda-api [Maven Build] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (21 de ma
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plex
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/ple
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/ple
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/commons/comr
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/commons/commo
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/ple
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/ple
[INFO] Downloading: https://repo.maven.apache.org/maven2/com/google/guava/guava/1
[INFO] Downloaded: https://repo.maven.apache.org/maven2/com/google/guava/guava/1
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 40.695 s
[INFO] Finished at: 2016-05-21T13:02:47-03:00
[INFO] Final Memory: 30M/250M
[INFO] -----

```

Andamento da construção do projeto

Quando a construção é finalizada, o arquivo JAR executável é criado na pasta *target* do projeto. Sendo assim, para implantar a Web API fora do ambiente de desenvolvimento basta copiar este arquivo para o servidor de destino. É importante manter o arquivo de configurações *application.yml* junto com o JAR executável, pois as configurações de acesso ao banco de dados e o número da porta HTTP para acessar a Web API são configurados neste arquivo. Para alterar o número da porta HTTP da Web API, basta adicionar a linha *server.port: <PORTA>* no arquivo *application.yml*.



Arquivo executável gerado

Para executar a Web API basta digitar o comando “`java -jar agenda-api-0.0.1-SNAPSHOT.jar`” na linha de comando. A execução pode ser configurada em arquivos batch em ambientes Windows ou scripts shell em ambientes Linux. Por exemplo, para executar a Web API em *background* em ambiente Linux, basta digitar o comando “`java -jar agenda-api-0.0.1-SNAPSHOT.jar &`”. Dessa forma, A Web API está pronta para ser executada em produção ou teste.



```

Terminal File Edit View Search Terminal Help

Spring Boot (v1.3.0.RELEASE)

2016-05-21 13:13:56.021 INFO 7567 --- [main] b.c.e.a.api.config.WebApiApplication
: Starting WebApiApplication v0.0.1-SNAPSHOT on notebook with PID 7567 (/home/ivan/Documents/Working/livro-java-web-apis/projectBuild/agenda-api-0.0.1-SNAPSHOT.jar started by ivan in /home/ivan/Documents/Working/livro-java-web-apis/projectBuild)
2016-05-21 13:13:56.030 INFO 7567 --- [main] b.c.e.a.api.config.WebApiApplication
: No profiles are active
2016-05-21 13:13:56.168 INFO 7567 --- [main] ationConfigEmbeddedWebApplicationContext
: Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationCon
text@1cc7b584: startup date [Sat May 21 13:13:56 BRT 2016]; root of context hierarchy
2016-05-21 13:13:58.485 INFO 7567 --- [main] f.a.AutowiredAnnotationBeanPostProcessor
: JSR-330 'javax.inject.Inject' annotation found and supported for autowiring
2016-05-21 13:13:59.110 INFO 7567 --- [main] trationDelegate$BeanPostProcessorChecker
: Bean 'org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration' of t
ype [class org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration$En
hancerBySpringGLIBS$8800db85] is not eligible for getting processed by all BeanPostProcessors (f
or example: not eligible for auto-proxying)
2016-05-21 13:13:59.971 INFO 7567 --- [main] e.j.JettyEmbeddedServletContainerFactory
: Server initialized with port: 8080
2016-05-21 13:13:59.978 INFO 7567 --- [main] org.eclipse.jetty.server.Server
: jetty-9.2.14.v20151106
2016-05-21 13:14:00.224 INFO 7567 --- [main] application
: Initializing Spring embedded WebApplicationContext
2016-05-21 13:14:00.224 INFO 7567 --- [main] o.s.web.context.ContextLoader
: Root WebApplicationContext: initialization completed in 4060 ms
2016-05-21 13:14:00.883 INFO 7567 --- [main] o.s.b.c.embedded.FilterRegistrationBean
: Mapping filter: 'characterEncodingFilter' to: [/]
2016-05-21 13:14:00.883 INFO 7567 --- [main] o.s.b.c.embedded.FilterRegistrationBean
: Mapping filter: 'requestContextFilter' to: [/]
2016-05-21 13:14:00.884 INFO 7567 --- [main] o.s.b.c.e.ServletRegistrationBean
: Mapping servlet: 'jerseyServlet' to: [/agenda-api/*]
2016-05-21 13:14:01.283 INFO 7567 --- [main] o.e.jetty.server.handler.ContextHandler
: Started o.s.b.c.e.j.JettyEmbeddedWebAppContext@1ebab64[/,file:/tmp/jetty-docbase.7656321182544
888838.8080/,AVAILABLE]
2016-05-21 13:14:01.284 INFO 7567 --- [main] org.eclipse.jetty.server.Server
: Started @7980ms
2016-05-21 13:14:03.889 INFO 7567 --- [main] o.s.j.e.a.AnnotationMBeanExporter
: Registering beans for JMX exposure on startup
2016-05-21 13:14:06.167 INFO 7567 --- [main] o.eclipse.jetty.server.ServerConnector
: Started ServerConnector@67579d34{HTTP/1.1}{0.0.0.0:8080}
2016-05-21 13:14:06.173 INFO 7567 --- [main] .s.b.c.e.j.JettyEmbeddedServletContainer
: Jetty started on port(s) 8080 (http/1.1)
2016-05-21 13:14:06.180 INFO 7567 --- [main] b.c.e.a.api.config.WebApiApplication
: Started WebApiApplication in 12.071 seconds (JVM running for 12.876)

```

Execução da Web API

Próximos Passos

Uma tendência que pode ser claramente observada é a migração de aplicações monolíticas para aplicações que seguem uma arquitetura de microservices. O uso de Web APIs RESTful é umas das opções mais utilizadas para implementação desta nova arquitetura. Entretanto, é necessário que as Web APIs sejam modeladas de acordo com as especificações de microservices. Sendo assim, é interessante conhecer mais profundamente como o desenvolvimento de Web APIs RESTful pode contribuir para o desenvolvimento de microservices.

Além disso, a fragmentação de aplicações monolíticas em diversos microservices resulta em uma série de desafios, como por exemplo, a utilização de mecanismos mais sofisticados para estabelecer a comunicação entre microservices. Outro desafio importante é reunir as funcionalidades, distribuídas em diversos microservices, para disponibilizar os recursos necessários aos usuários finais da aplicação. Diversas soluções estão sendo lançadas no mercado para solucionar alguns desses problemas. Entretanto, é fundamental que as soluções contemplem uma maior autonomia dos componentes na tomada de decisão e auto adaptação dos componentes quanto a futuras mudanças na implementação dos serviços. Para atingir essas características é fundamental a adoção de tecnologias semânticas. Quando a implementação de Web APIs semânticas for uma realidade, diversos problemas serão solucionados e uma nova ordem muito mais interessante de problemas entrará em cena.