**Sumário**

[Introdução 4](#_Toc440922168)

[1.1. Introdução 4](#_Toc440922170)

[1.2. Vantagens X Desvantagens 4](#_Toc440922171)

[1.3. O desenvolvimento Web e o protocolo HTTP 4](#_Toc440922172)

[1.4. O Servidor WEB 4](#_Toc440922173)

[1.5. O protocolo HTTP 5](#_Toc440922174)

[Banco de Dados 6](#_Toc440922175)

[2.1. Instalação das ferramentas 6](#_Toc440922177)

[2.2. Criando o banco da aplicação 7](#_Toc440922178)

[Java EE 8](#_Toc440922179)

[3.1. Algumas especificações do Java EE 8](#_Toc440922183)

[3.2. Servidor de Aplicação 9](#_Toc440922184)

[3.3. Servlet Container 9](#_Toc440922185)

[3.4. Preparando o Tomcat 10](#_Toc440922186)

[Inicializando a Aplicação 10](#_Toc440922187)

[4.1. Criando o Projeto 10](#_Toc440922192)

[4.2. Instalando a aplicação no Tomcat 12](#_Toc440922193)

[4.3. A estrutura do Projeto 12](#_Toc440922194)

[API JDBC 13](#_Toc440922195)

[5.1. Porque usar um banco de dados? 13](#_Toc440922197)

[5.2. O Que é a API JDBC? 13](#_Toc440922198)

[5.3. A conexão em Java 13](#_Toc440922199)

[5.4. Adicionando o Driver ao Projeto. 14](#_Toc440922200)

[5.5. Fábrica de Conexões 14](#_Toc440922201)

[5.6. Testando a conexão 16](#_Toc440922202)

[5.7. Criando o Bean 16](#_Toc440922203)

[5.8. Melhorando as Exceções 17](#_Toc440922204)

[A Documentação do Código 18](#_Toc440922205)

[6.1. As Classes 18](#_Toc440922212)

[6.2. Os Métodos 19](#_Toc440922213)

[6.3. Trechos de código 19](#_Toc440922214)

[6.4. Linhas 19](#_Toc440922215)

[O Padrão DAO 19](#_Toc440922216)

[7.1. A Interface DAO 20](#_Toc440922218)

[7.2. Criando o DAOImpl 20](#_Toc440922219)

[7.3. Garantindo a Conexão 20](#_Toc440922220)

[7.4. Tirando o SQL da aplicação 20](#_Toc440922221)

[7.5. Inserindo um Contato 21](#_Toc440922222)

[7.6. Teste o seu DAO 22](#_Toc440922223)

[7.7. O buscar Todos 23](#_Toc440922224)

[7.8. Removendo um Contato 25](#_Toc440922225)

[7.9. A pesquisa por Id 25](#_Toc440922226)

[7.10. Fechando as conexões de forma segura e elegante. 26](#_Toc440922227)

[A Camada WEB 27](#_Toc440922228)

[8.1. Servlets 27](#_Toc440922230)

[8.2. O BuscaContato 27](#_Toc440922231)

[8.3. O Get 27](#_Toc440922232)

[8.4. Enviando e recuperando parâmetros no doGet 29](#_Toc440922233)

[8.5. Deixando o buscaContato mais flexível 30](#_Toc440922234)

[8.6. Gravando um novo contato 31](#_Toc440922235)

[Colocando o HTML no seu devido lugar 33](#_Toc440922236)

[9.1. O AdicionaContato 33](#_Toc440922246)

[9.2. JavaServer Pages 34](#_Toc440922247)

[9.3. Usando Scriptlets 35](#_Toc440922248)

[9.4. JSTL para escrever menos 36](#_Toc440922249)

[9.5. Trabalhando com Taglibs 36](#_Toc440922250)

[Otimizando nossas Servlets 37](#_Toc440922251)

[10.1. Criando o Controller 38](#_Toc440922262)

[10.2. Melhorando o BuscaContato 40](#_Toc440922263)

[10.3. Melhorando o AdicionaContato 42](#_Toc440922264)

[O Padrão Arquitetural MVC 43](#_Toc440922265)

[Trabalhando com o Bootstrap 43](#_Toc440922266)

[Melhorando nossa Aplicação 44](#_Toc440922267)

[13.1. Criando o Menu 44](#_Toc440922281)

[13.2. Alterando o Index 44](#_Toc440922282)

[13.3. Melhorando nossa Página de Sucesso 45](#_Toc440922283)

[13.4. Lançando Exceções mais elegantes para o usuário 46](#_Toc440922284)

[13.5. Melhorando o BuscaContato 47](#_Toc440922285)

[Removendo um Contato 48](#_Toc440922286)

[14.1.1. O RemoveContato 48](#_Toc440922301)

[14.2. Alterando A JSP 48](#_Toc440922302)

[Desafio... 49](#_Toc440922303)

[15.1. O Editar Contato 49](#_Toc440922319)

[15.2. Filtrando a Busca de Contatos 49](#_Toc440922320)

[Gerando a Documentação do Sistema 50](#_Toc440922321)

Essa apostila está dividida em duas partes, a primeira é uma introdução aos conceitos de WEB que são imprescindíveis para desenvolvimento de qualquer sistema. Enquanto a segunda parte trata do desenvolvimento de uma Agenda WEB, que será construída durante o curso, onde aprenderemos como modelar e desenvolver um sistema utilizando Java WEB. Bons estudos!

Capítulo

# Introdução



## Introdução

Certamente o grande forte do Java é o mercado WEB. Criar um projeto com Java dá muita liberdade. As aplicações WEB são importantes, pois o sistema agora fica instalado em um servidor remoto que pode ser acessado em qualquer momento, em qualquer parte do mundo, necessitando apenas de um browser e de internet. Apesar de tanta popularidade no ambiente Web, o desenvolvimento com Java não é trivial: é necessário conhecer com certa profundidade as APIs de servlets e de JSP, mesmo que sua equipe venha utilizar frameworks como Spring MVC, VRaptor ou JSF. Conceitos de HTTP, session e cookies também são mandatórios para poder enxergar gargalos e problemas que sua aplicação enfrentará.

## Vantagens X Desvantagens

Durante muitos anos, os usuários se habituaram com aplicações Desktop. Este tipo de aplicação é instalada no computador local e acessa diretamente um banco de dados ou gerenciador de arquivos. As aplicações eram criadas com uma série de componentes ricos oferecidos pela plataforma do próprio sistema operacional. Porém existem desvantagens no desenvolvimento Desktop, como por exemplo, cada usuário tem uma cópia integral da aplicação, qualquer alteração precisaria ser propagada para todas as outras máquinas, além disso, o usuário deve se preocupar com as configurações das máquinas (memória, processador, etc.). Torna-se difícil identificar BUGs. Em geral, enfrentamos problemas de manutenção e gerenciabilidade.

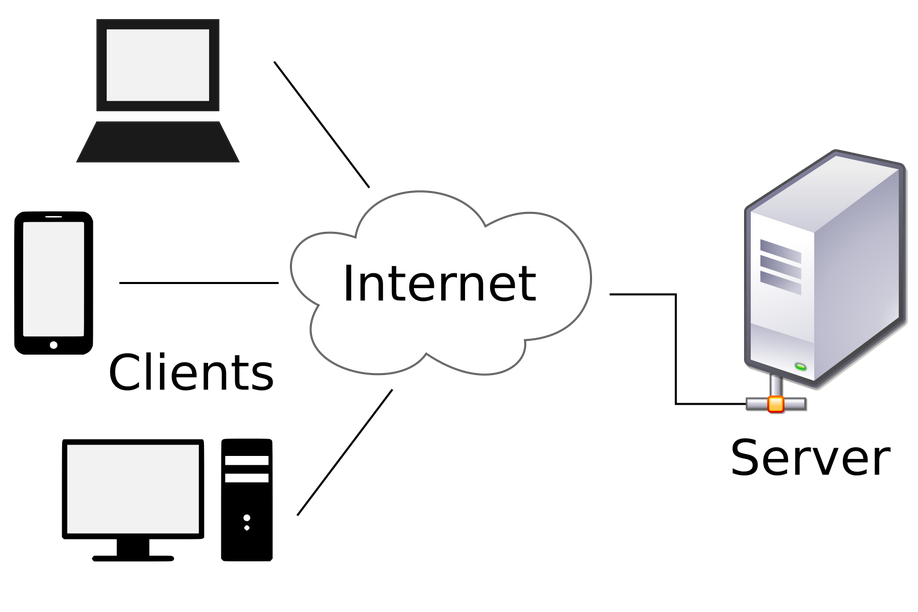
Esse cenário muda no mundo WEB, pois a aplicação irá rodar em um servidor que o usuário vai acessar através do browser, em qualquer lugar, ou seja, em um sistema corporativo, o usuário não precisa estar dentro da empresa para ter acesso ao sistema. A manutenção e o gerenciamento se tornam muito mais fáceis. Entre outros benefícios.

## O desenvolvimento Web e o protocolo HTTP

Há um servidor central onde a aplicação é executada e processada e todos os usuários podem acessá-la através de um cliente simples e do protocolo HTTP. Um navegador web, como Firefox ou Chrome, que fará o papel da aplicação cliente, interpretando HTML, CSS e JavaScript - que são as tecnologias que ele entende. O browser faz solicitações ao servidor (requests), que fornece as respostas (responses). E agora as aplicações trabalham orientadas a requisições e respostas e não orientadas a eventos (Desktop).

## O Servidor WEB

A função do servidor web é receber as requisições e devolver algo para o cliente (respostas).

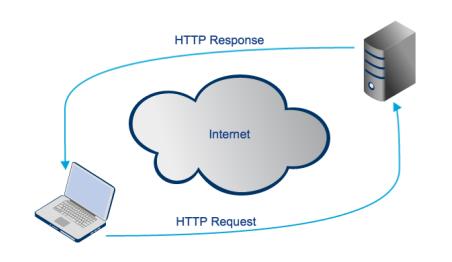


A função do cliente é fazer solicitações ao servidor, receber a resposta e exibir o resultado do pedido. O browser é o software que se comunica com o servidor e “traduz” a resposta do servidor através da camada de aplicação.

## O protocolo HTTP

É o protocolo usado para a comunicação entre os clientes e os servidores. O Conteúdo de uma solicitação (request) geralmente pode ser formado por: Método HTTP, página que será acessada; parâmetros do formulário; enquanto o conteúdo de uma resposta contém, o código de status (informa se a solicitação foi realizada com sucesso ou não), o tipo de Conteúdo (HTML, figuras, textos, etc) e o conteúdo propriamente dito (HTML real, imagem, etc). A solicitação HTTP possui outra solicitação conhecida como URL (Localizador Uniforme de Recursos). A solicitação URL é um recurso que se ativa quando o usuário tenta acessar alguns dos métodos HTTP descritos abaixo:

* **GET** - Solicita ao servidor um recurso chamado de solicitação URl. Este é o método padrão, pois é a forma como o browser chama o servidor quando se digita uma URL para que ele a recupere.
* **POST** - Contém um corpo nos quais seus parâmetros de solicitação já são codificados. O mais frequente uso desse método é na submissão de formulários.
* **HEAD** - Similar ao método GET, o servidor apenas retoma a linha de resposta e os cabeçalhos de resposta.
* **PUT** - Esse método permite o envio de arquivos para o servidor Web.
* **DELETE** - Permite a exclusão de documentos dentro do servidor Web.
* **OPTIONS** - É possível fazer uma consulta de quais comandos estão disponíveis para um determinado usuário.



Capítulo

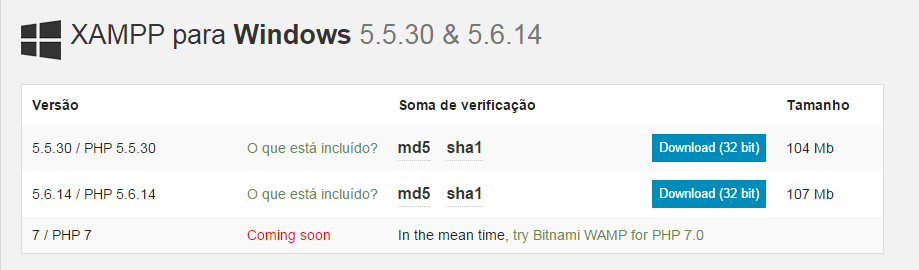
# Banco de Dados

Muitos sistemas precisam manter as informações com as quais trabalham, seja para permitir consultas futuras, geração de relatórios ou possíveis alterações nas informações. Para que esses dados sejam mantidos para sempre, esses sistemas geralmente guardam essas informações em um banco de dados.

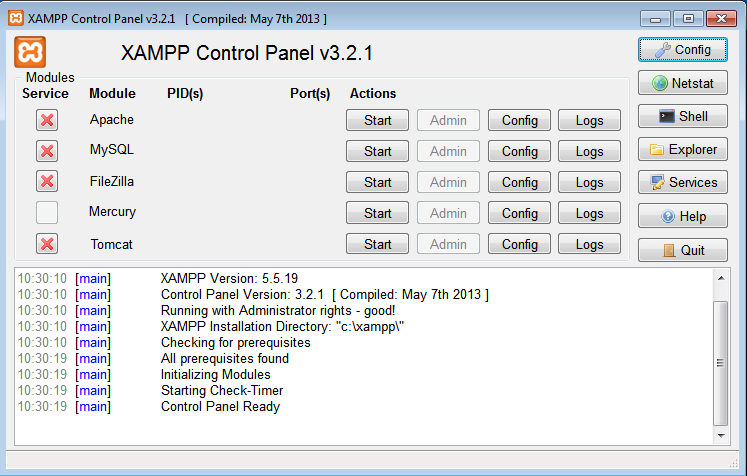


## Instalação das ferramentas

Durante esse curso utilizaremos o banco de dados MySQL, há várias formas de se instalar o mesmo, porém utilizaremos o xampp, que é um pacote que contém diversos ferramentas, inclusive o banco de dados MySQL, que utilizaremos. Faça o download do xampp, de acordo com o seu sistema operacional, diretamente do site oficial: <https://www.apachefriends.org/pt_br/download.html>

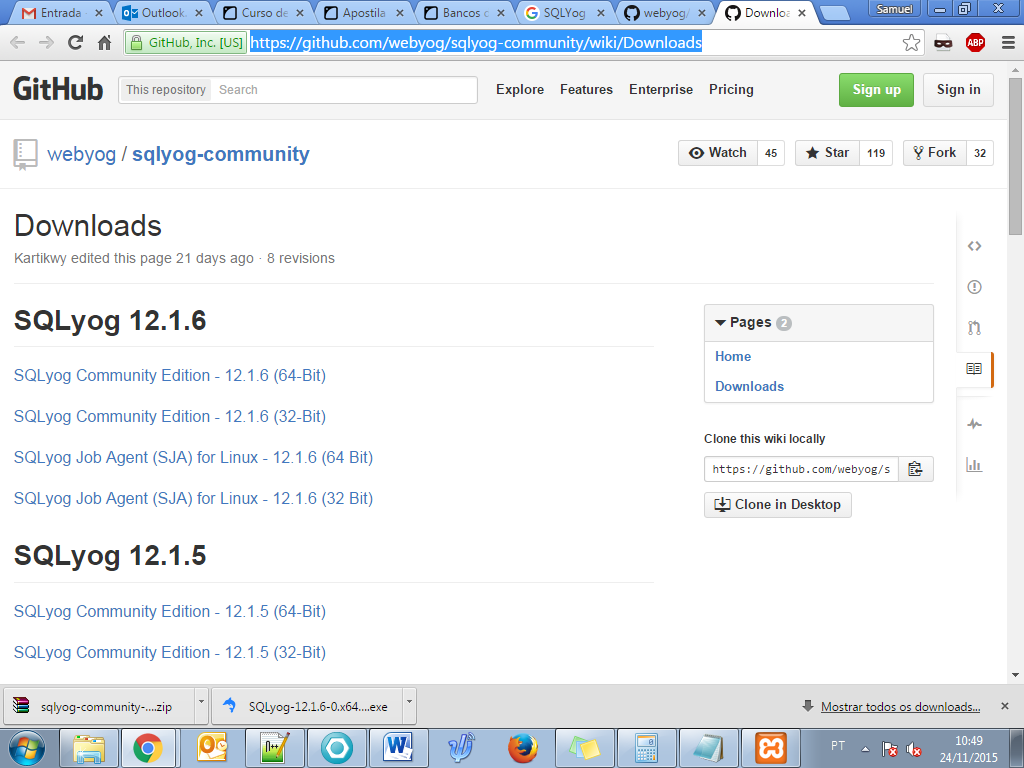


Após o download faça a instalação e execute o Xampp Control Panel (painel de controle do xampp). Para começar a utilizar o serviço do MySQL basta clicar em ‘Start’

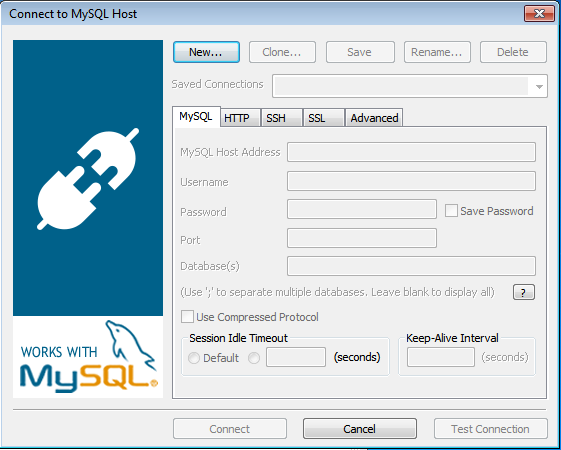


O próximo passo será instalar uma ferramenta para facilitar o trabalho com o MySQL, ou seja, uma IDE para banco de dados. No nosso curso utilizaremos o SQLYog Community Edition, disponível em:

<https://github.com/webyog/sqlyog-community/wiki/Downloads>



Faça a instalação padrão do SQLYog, ao final, clique em executar SQLYog. O próximo passo é criar uma conexão com o MySQL. Na tela de Conexão do SQLYog clique em ‘NEW’, em seguida devemos dar um nome a nova conexão, utilizaremos o nome ‘**localhost’** (padrão).



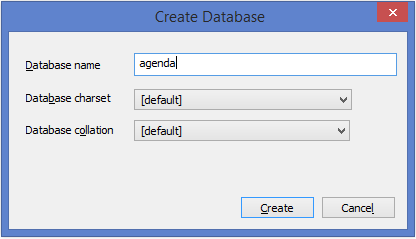
**OBS1**: Você pode trabalhar com o banco via linha de comando (diretamente no prompt), porém para agilizar o trabalho é mais indicado utilizar uma IDE.

**OBS2**: Todas as ferramentas utilizadas nesse curso são livres. Existem várias versões do SQLYog inclusive versões profissionais, mas para esse curso será utilizada apenas a versão Community.

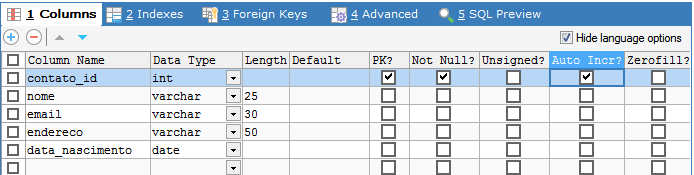
**OBS3**: Para criar uma nova conexão com o MySQL e poder criar um novo banco de dados o serviço do MySQL precisa estar startado no xampp (é comum esquecer desse passo).

## Criando o banco da aplicação

Para criar um novo banco de dados, basta clicar com o botão direito do mouse sobre a conexão e escolher ‘Create Database’, em seguida atribuir um nome, que no nosso exemplo será ‘agenda’.



Após criar o banco, criaremos as tabelas que serão utilizadas por nosso sistema, para isso clique na pasta ‘Tables’. A primeira tabela a ser criada será contato, para isso clique com o botão direito em ‘Tables’. O nome da tabela será contato e terá os seguintes campos:



Após criar a tabela, basta salvar.

**OBS1**: Não se esqueça de deixar o campo *contato\_id* como **PK** (Primary Key), **Not Null** e **Auto Incremento** (para que o banco fique responsável por gerenciar os IDs).

**OBS2**: Os campos do tipo varchar deve sempre ter um length (tamanho).

**OBS3**: Após a criação da tabela Contato, iremos popular a mesma com alguns registros para facilitar nosso trabalho, para isso utilize a interface gráfica do SQLYog ou utilize o comando **INSERT**.

Capítulo

# Java EE

As aplicações Web de hoje em dia já possuem regras de negócio bastante complicadas (requisitos funcionais). Além dessas regras, existem os requisitos não funcionais: persistência em banco de dados, transação, acesso remoto, web services, gerenciamento de threads, gerenciamento de conexões HTTP, cache de objetos, gerenciamento da sessão web, balanceamento de carga, entre outros.

O Java EE (Java Enterprise Edition) foi criado pela Sun e consiste de uma série de especificações bem detalhadas, dando uma receita de como deve ser implementado um software que faz cada um desses serviços de infraestrutura. Porque a Sun faz isso? A ideia é que você possa criar uma aplicação que utilize esses serviços. Como esses serviços são bem complicados, você não perderá tempo implementando essa parte do sistema. Existem implementações tanto open source quanto pagas, ambas de boa qualidade.



## Algumas especificações do Java EE

As APIs a seguir são as principais dentre as disponibilizadas pelo Java Enterprise:

* JavaServer Pages (**JSP**), Java **Servlets**, Java Server Faces (**JSF**) - trabalhar para a Web;
* Enterprise Javabeans Components (**EJB**) e Java Persistence API (**JPA**) - objetos distribuídos, clusters, acesso remoto a objetos etc.;
* Java API for XML Web Services (**JAX-WS**), Java API for XML Binding (**JAX-B**) - trabalhar com arquivos xml e webservices;
* Java Autenthication and Authorization Service (**JAAS**) - API padrão do Java para segurança;
* Java Transaction API (**JTA**) - controle de transação no contêiner;
* Java Message Service (**JMS**) - troca de mensagens assíncronas;
* Java Naming and Directory Interface (**JNDI**) - espaço de nomes e objetos;
* Java Management Extensions (**JMX**) - administração da sua aplicação e estatísticas sobre a mesma;

**Onde encontrar as especificações?**

O grupo responsável por gerir as especificações é Java Community Process: <http://www.jcp.org/>

Lá você pode encontrar tudo sobre as Java Specification Requests (JSR), isto é, os novos pedidos de bibliotecas e especificações para o Java, tanto para JavaSE, quanto EE e outros.

Sobre o Java EE, você pode encontrar em: <http://java.sun.com/javaee/>

## Servidor de Aplicação

O objetivo do servidor de aplicações é disponibilizar uma plataforma que separe do desenvolvedor de software algumas das complexidades de um sistema computacional. No desenvolvimento de aplicações comerciais, por exemplo, o foco dos desenvolvedores deve ser a resolução de problemas relacionados ao negócio da empresa, e não de questões de infraestrutura da aplicação. O servidor de aplicações responde a algumas questões comuns a todas as aplicações, como segurança, garantia de disponibilidade, balanceamento de carga e tratamento de exceções.

## Servlet Container

O Java EE possui várias especificações, entre elas, algumas específicas para lidar com o desenvolvimento de uma aplicação Web:

* Servlet
* JSP
* JSTL
* JSF

Um **Servlet** **Container** é um servidor que suporta essas funcionalidades, mas não necessariamente o Java EE completo. É indicado a quem não precisa de tudo do Java EE e está interessado apenas na parte web (boa parte das aplicações de médio porte encaixam-se nessa categoria). Há alguns servlet containers famosos no mercado. O mais famoso é o **Apache** **Tomcat** (por esse motivo o utilizaremos em nosso curso).

## Preparando o Tomcat

O primeiro passo é fazer o download do Tomcat, disponível no site oficial, utilizaremos a versão 8.0, baixaremos apenas o Core, zipado, que está disponível em: <http://tomcat.apache.org/>

Após o download extraia o conteúdo do arquivo zipado para a pasta C:\Tomcat\.

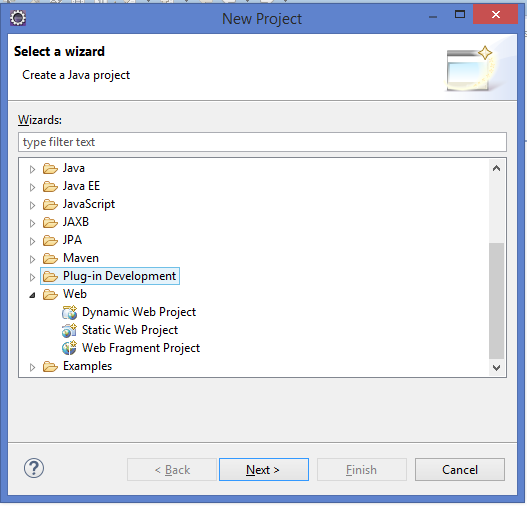
Capítulo

# Inicializando a Aplicação

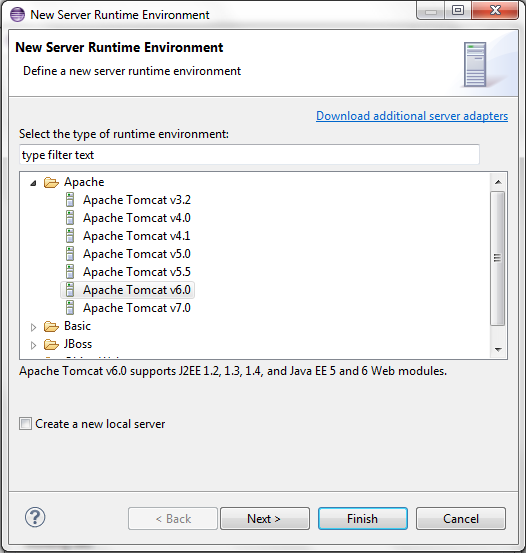


## Criando o Projeto

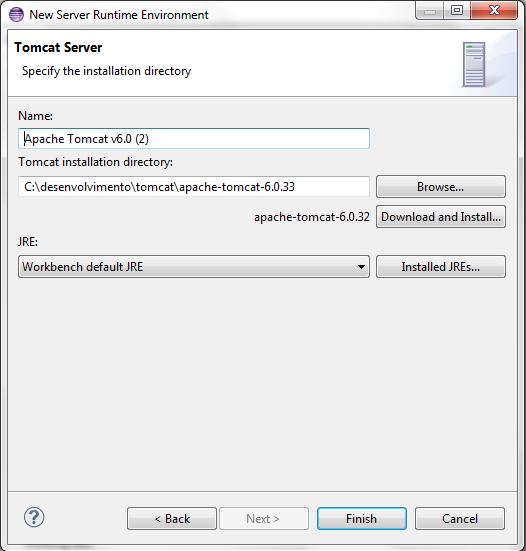
No eclipse, criaremos um novo projeto, que deve ser do tipo **Dynamic Web Project**. Chamaremos o projeto de **Agenda.**



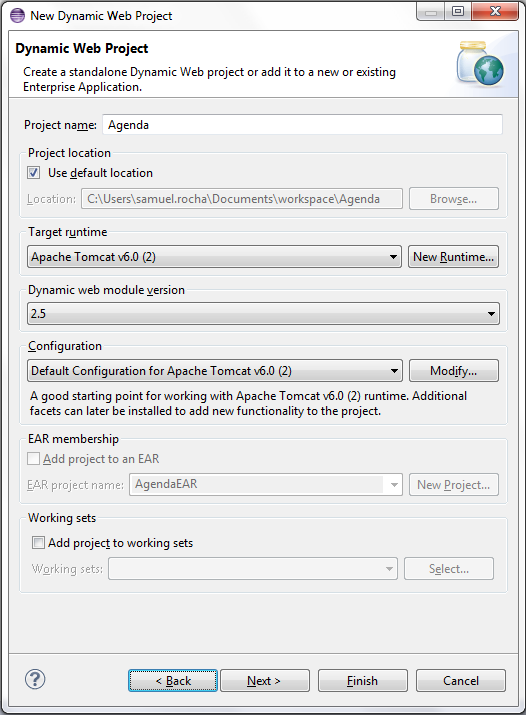
Na opção Target runtime, vamos clicar em New runtime e escolher a opção Apache Tomcat v8 e clicar em Next.



Na tela seguinte, vamos dizer onde extraímos o arquivo zipado, no nosso caso foi em c:\Tomcat, em seguida podemos clicar em Finish:

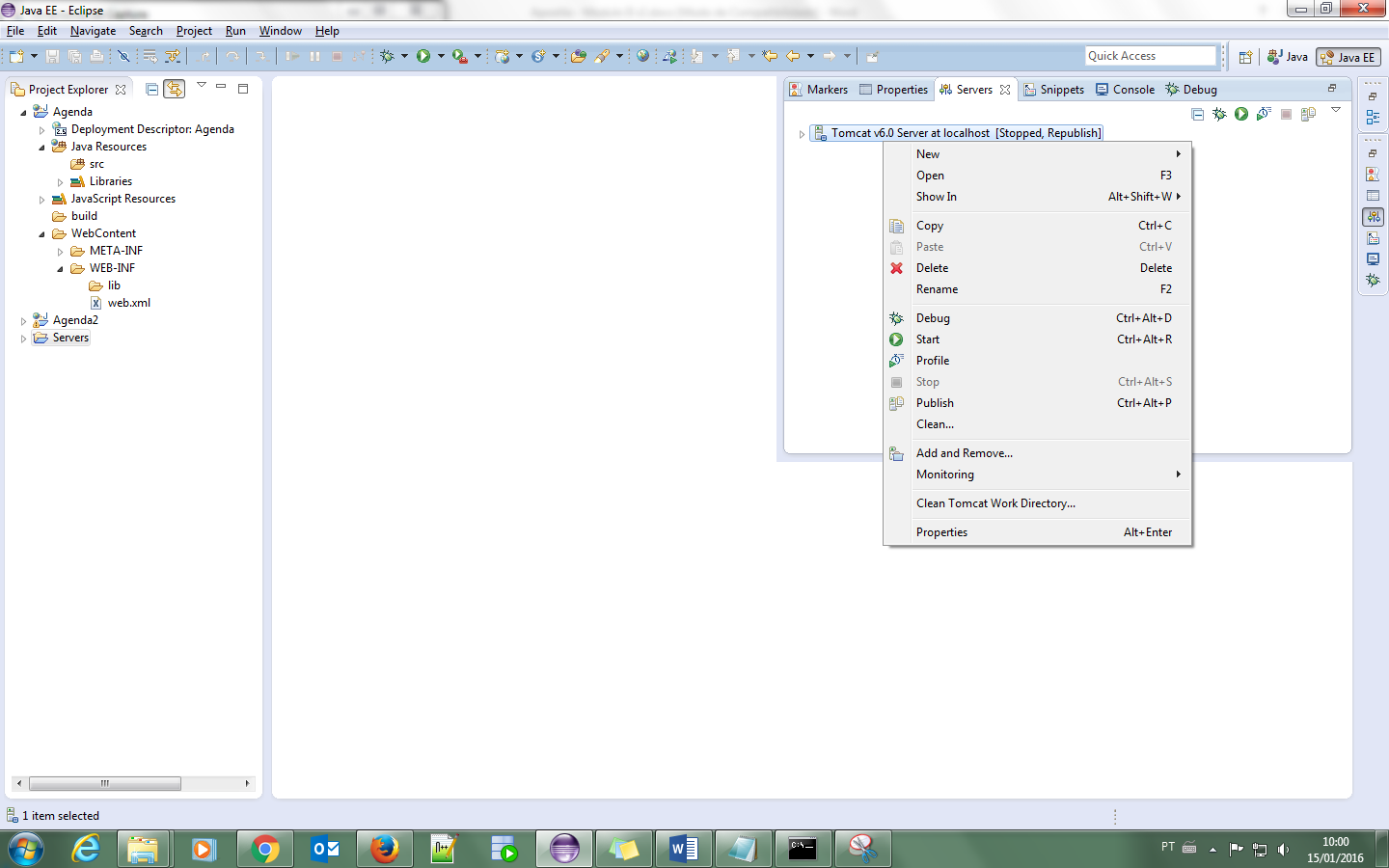


Após a configuração, podemos em Next, Next e em **Finish**. Caso o eclipse solicite alterar a perspectiva, pode aceitar, a partir de agora trabalharemos na perspectiva Java EE.

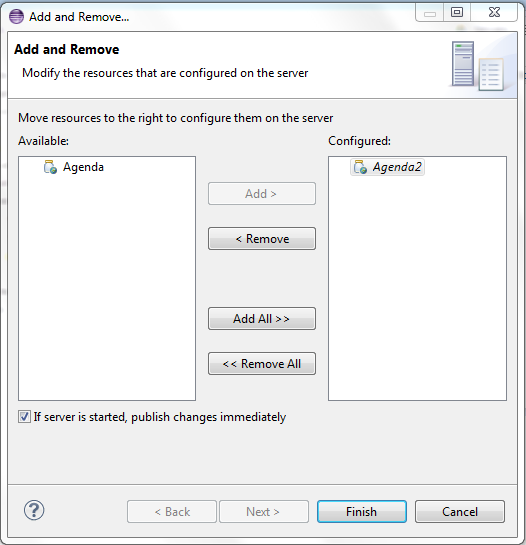


## Instalando a aplicação no Tomcat

Para o Tomcat rodar nossa aplicação, precisamos fazer a instalação da mesma. Para isso vamos clicar no servidor com o botão direito e escolher a opção Add and Remove...



Basta adicionar nosso projeto e clicar em Finish.

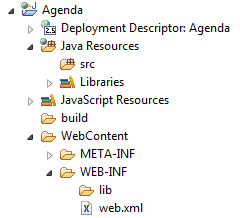


## A estrutura do Projeto

O nosso projeto tem uma estrutura diferente do que estamos acostumados. Na pasta Java Resources, está a conhecida **src**, onde ficará todo o nosso código fonte Java.

Outro diretório muito importante é o **WebContent**, pois é nele que vai ficar todo o nosso conteúdo a ser acessado pelo usuário, ou seja, tudo o que for referente a HTML, CSS e Javascript ficará no WebContent**.**

Repare também que dentro da WebContent há uma pasta chamada **WEB-INF**. Essa pasta é extremamente importante para qualquer projeto web Java EE. Ela contém configurações e recursos necessários para nosso projeto rodar no servidor. O web.xml é o arquivo onde ficará armazenada as configurações relativas à sua aplicação. No **WEB-INF** também ficam as bibliotecas (libs) usadas na sua aplicação. Mas atenção, o **WEB-INF não é visível para o usuário.**



Capítulo

# API JDBC

Após a configuração do nosso ambiente, passaremos a trabalhar em nosso sistema, o primeiro passo é fazer a aplicação se comunicar com o banco de dados, para isso utilizaremos a API JDBC (Java Database Connectivity).



## Porque usar um banco de dados?

Muitos sistemas precisam manter as informações com as quais eles trabalham, seja para permitir consultas futuras, geração de relatórios ou possíveis alterações nas informações. Para que esses dados sejam mantidos para sempre, esses sistemas geralmente guardam essas informações em um banco de dados, que as mantém de forma organizada e prontas para consultas. A maioria dos bancos de dados comerciais são os chamados **relacionais**, que é uma forma de trabalhar e pensar diferente ao paradigma orientado a objetos.

## O Que é a API JDBC?

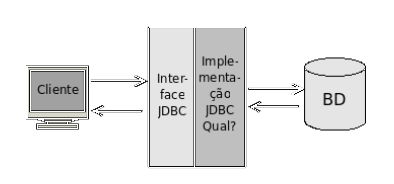
Java Database Connectivity ou JDBC é um conjunto de classes e interfaces (API) escritas em Java que fazem o envio de instruções SQL para qualquer banco de dados relacional. No passado a única forma de fazer a aplicação se comunicar com o banco de dados era conhecendo o protocolo proprietário do banco e programando em cima desse protocolo, o que variava de um fabricante para o outro (Oracle, MySql, SQL Server, etc.). Com o tempo se viu a necessidade de ter uma forma mais simples e genérica de se comunicar com o bando de dados, surgindo assim a API JDBC.

**Mais sobre a API JDBC**

<https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/>

## A conexão em Java

A API JDBC nos oferece uma forma simples e elegante de se comunicar com o banco de dados, porém cada banco trabalha de forma variada entre fornecedores (SQLServer, MySql, etc.), logo precisamos de um agente intermediário que saiba com qual banco estamos trabalhando e que possa “**conversar**” com o mesmo. Esse agente chamamos de **Driver**, todos os principais bancos de dados do mercado possuem drivers JDBC para que você possa utilizá-los com Java. O driver é responsável por fazer a “tradução” entre o código Java e o Banco de Dados utilizado na aplicação.

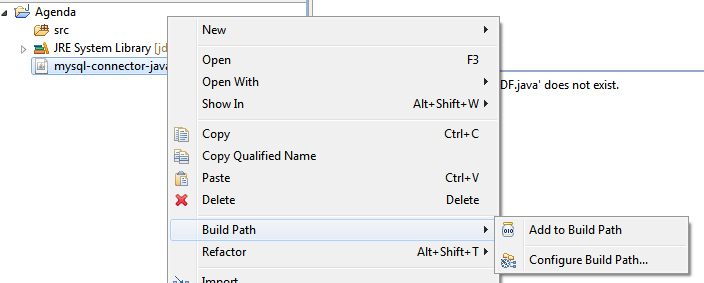


Como utilizaremos o MySql o próximo passo é baixar o seu driver, que está disponível em <http://dev.mysql.com/downloads/connector/j/>

Caso ainda não tenha cadastro na Oracle, faça o seu cadastro e em seguida baixe o driver. Extraia o arquivo .zip para alguma pasta do seu computador.

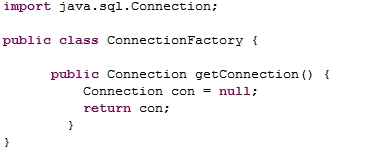
## Adicionando o Driver ao Projeto.

Para adicionar o Driver devemos copiar o arquivo **mysql-connector-java-5.1.37-bin.jar** (que na pasta que você acabou de extrair) e colar na pasta src do nosso projeto, em seguida clicamos com o botão direito -> Build Path -> Add to Build Path. Agora o Driver do MySql está no nosso projeto e podemos utilizar a API JDBC para se comunicar com o nosso banco ‘agenda’.

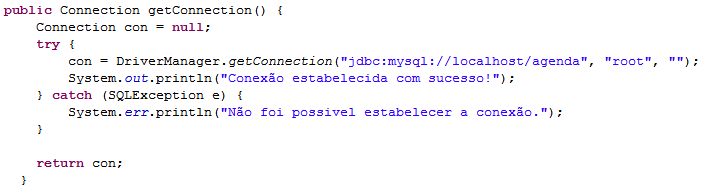


## Fábrica de Conexões

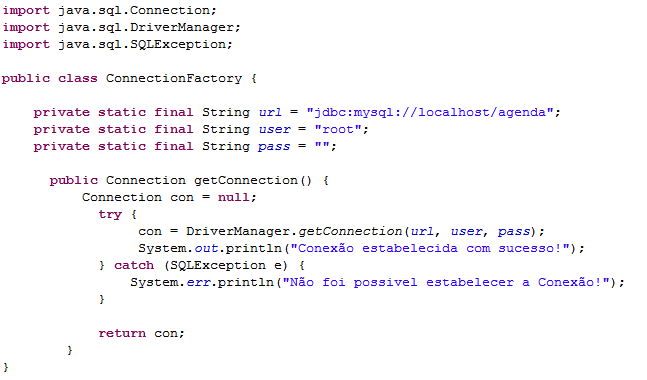
Por **padrão**, sempre que estamos trabalhando com JDBC criamos uma classe que fica responsável apenas pela conexão com o banco de dados, essa classe é chamada **ConnectionFactory**. Deixaremos nossa classe dentro do pacote **br.com.agenda.util**. Essa classe possui apenas um método, **getConnection**, o método responsável por de fato estabelecer a conexão com o banco de dados, ou seja, ele nos retorna uma conexão valida, uma **Connection**.



Para estabelecermos uma conexão com o banco basta solicitamos ao Driver essa conexão, passando a url de conexão com o banco, o usuário e a senha. Observe que ao solicitar uma conexão ao Driver, isso pode gerar uma SQLException, por isso usaremos um Try/Catch para tratar o erro.



Para deixar o nosso código mais elegante e seguir os padrões adotados, trocaremos a url, o usuário e a senha por constantes privadas na nossa classe. Veja abaixo como fica a nossa classe completa.

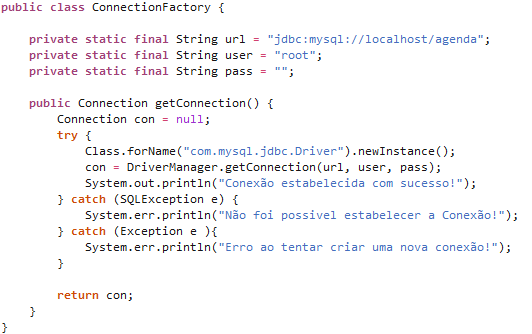


**OBS1:** Tenha cuidado com os imports. Lembre-se que estamos usando os imports do pacote java.sql.

**OBS2:** Por padrão o usuário do MySql é *root* e a não tem senha.

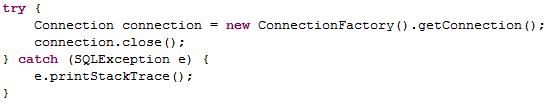
**Design Patterns e o Factory Method**Não foi por acaso que chamamos a classe de ConnectionFactory. Essa é um padrão de projeto muito comum e bastante recomendado. Muitas vezes queremos controlar o processo de criação de um objeto que pode ser repetido diversas vezes em nosso código, como o de abrir uma conexão. O que aconteceria se esse código estivesse espalhado por nossa aplicação e alguma informação mudasse? Com a classe ConnectionFactory, não estamos deixando as informações do banco de dados espalhadas pela aplicação. Essa boa prática de projeto é conhecida como **factory** **method** e compõe um dos diversos **Design** **Patterns** existentes.

Como estamos trabalhando com um sistema WEB precisamos **registrar** **o Drive** do banco que iremos utilizar, garantindo que “a aplicação saiba” com qual banco conversar. Esse passo na verdade só é necessário, porque o servidor que estamos utilizando (Tomcat) nos obriga a registrar o Driver para evitar vazamento de memória, ou seja, é um caso especifico do Tomcat. Desde o Java 6 que não é mais necessário registrar o Drive através do Class.forName. Observe agora como fica nossa Classe:



## Testando a conexão

Para verificar se tudo está conforme esperado criaremos a classe **TestaConexao** dentro do pacote **br.com.agenda.teste**. Todas as nossas classes de teste deverão ficar nesse pacote. A classe TestaConexao terá apenas um método main. Crie a Classe em seguida execute diversos testes para verificar se a conexão está sendo estabelecida ou não, também tente alterar a url de conexão e testar novamente para você ver o que ocorre.



## Criando o Bean

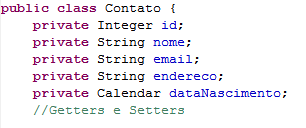
Beans ou Javabeans são classes que possuem o construtor sem argumentos e com métodos de acesso do tipo get e set! Nesse caso o bean será uma representação da nossa tabela contato, uma **entidade**, uma maneira simples de trabalharmos com as informações dos contatos na nossa aplicação.

**JAVABEANS:**

*Javabeans* ou simplesmente *Beans*, também são conhecidos como modelos anêmicos ou classes anêmicas, em algumas literaturas, essas classes também são chamadas de *POJOS.*

Cuidado para não confundir Javabeans (**Beans**) com Enterpise Java Beans (**EJB**), esse último é outro componente do Java EE.

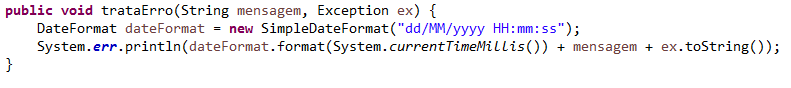
No pacote **br.com.agenda.beans** criaremos a Classe **Contato**, que será uma representação (mapeamento) da nossa tabela Contato.



Observe que a nossa classe Contato é uma representação exata da tabela contato, um mapeamento da tabela e colunas do banco para uma Classe e atributos no Java. Trabalhar dessa forma é uma excelente prática, no passado era comum tratar as informações do banco apenas como **Object**, ao invés de trabalhar com tipos específicos para representar as **entidades** do banco. Essas classes que representam uma tabela do banco de dado comumente são chamadas de **beans**, **entidades** ou **modelos**, pois nelas existem as **mesmas** **informações** **das** **tabelas** **do** **banco**.

## Melhorando as Exceções

A forma que as exceções estão sendo tratadas não estão coerentes, vamos dar um tratamento melhor, para isso criaremos o método **trataErro** na Classe **Util** do pacote **br.com.agenda.util**. Esse método irá printar a data e hora atual mais uma mensagem de erro juntamente com a Exceção que foi gerada.



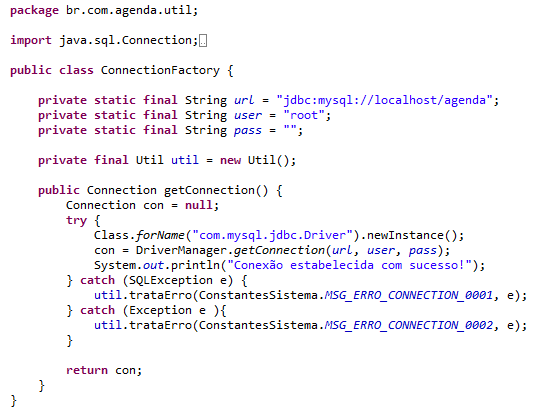
A mensagem de erro que esse método recebe como parâmetro, será uma constante publica criada na Classe ConstantesSistema, você deve cadastrar uma mensagem para cada erro que o sistema pode lançar:



O próximo passo é chamar o método trataErro na nossa Aplicação, não esqueça de passar também a mensagem cadastrada. Observe que agora a nossa Classe ConnectionFactory ficou bem mais elegante:

**O Tratamento de Exceções:**

Em aplicações grandes (comerciais e corporativas) todas as exceções são gravadas em um arquivo de log, arquivo de texto que guarda detalhes sobre as exceções do sistema. Também é comum guardas esses detalhes em uma tabela do banco de dados. O objetivo de guardas esses detalhes é que no futuro a equipe de arquitetura ou de analise possa ver como o sistema está se comportando, quais são os erros mais comum e porque eles ocorrem. Também é comum criar pequenos sistemas apenas para ler os arquivos de log e gerar gráficos e estatísticas, porém como nosso sistema é simples e não temos muito tempo, iremos tratar as exceções de forma mais simples.



Capítulo

# A Documentação do Código

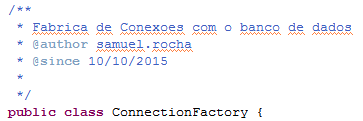
A preocupação com a qualidade do código é extremamente importante, pois sistemas evoluem muito rápido, por isso devemos sempre pensar em código limpo, melhores práticas e código bem documentado. O Java nos fornece um recurso bastante útil para documentarmos nosso sistema, chamado de JAVADOC, que é um gerador de documentação criado pela Sun Microsystems para documentar a API dos programas em Java, a partir do código-fonte. O resultado é expresso em uma página HTML. Porém para essa documentação do seu código ser gerada você precisa anotar as classes e métodos do seu sistema, como vamos ver a seguir.



## As Classes

Antes da assinatura da Classe você deve colocar um breve texto sobre o que aquela Classe faz e em seguida anotar com as tags de documentação. As principais tags para documentar classes são:

* @author - Especifica o autor da classe ou do método em questão.
* @since - Indica desde quando uma classe ou métodos foi adicionado na aplicação.

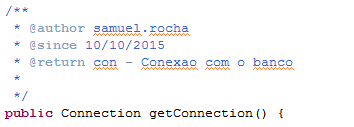


Para documentar basta digitar /\*\* e teclar Enter.

## Os Métodos

Antes da assinatura do método você deve colocar um breve texto sobre o que aquele método faz e em seguida anotar com as tags de documentação. As principais tags para documentar métodos são:

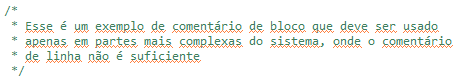
* @author - Especifica o autor da classe ou do método em questão.
* @param - Mostra um parâmetro que será passado a um método.
* @return - Mostra qual o tipo de retorno de um método.
* @since - Indica desde quando uma classe ou métodos foi adicionado na aplicação.
* @throws - Indica os tipos de exceções que podem ser lançadas por um método.



Nem todas as anotações são obrigatórias, use sempre o bom senso e siga o padrão da sua equipe de desenvolvimento. Observe que na imagem anterior, usamos apenas o @author, @since e @return.

## Trechos de código

As vezes precisamos escrever um comentário em alguma parte complexa do nosso código, algum trecho de código com recursos mais avançados ou com uma regra mais difícil, nesses trechos devemos usar comentários, uma opção é usar comentário de bloco:



## Linhas

Também existem os comentários de linha:



Já existem algumas ferramentas que verificam se você fez a documentação corretamente no seu código.

Capítulo

# O Padrão DAO

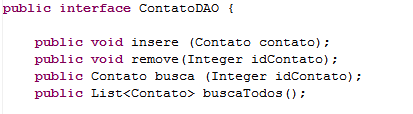
Grande parte das aplicações desenvolvidas em Java que fazem uso de banco de dados utiliza algum tipo de persistência, mas há um sério risco de emaranhar, misturar o código de lógica de negócios mais o código da persistência de dados (**CRUD**), ou seja, misturar código **SQL** com código **JAVA**. Para evitar que exista esse emaranhado no código foi criado o padrão de projeto **DAO** (Data Access Object / Objeto de Acesso a Dados). O DAO se tornou assim um dos padrões de projeto mais utilizados no desenvolvimento de aplicações de softwares e tem como finalidade separar as regras de acesso a banco de dados das regras de negócios e de interface com usuário ou qualquer outro tipo de classe que não tenha relevância alguma com as ações de persistência.

**OBS**: No passado, o **DAO** também era chamado de **repositório**. Não se assuste se ouvir alguém falando dessa forma.



## A Interface DAO

É comum e sempre uma boa prática criar um Interface, um contrato, que será responsável por definir quais métodos serão utilizados dentro do DAO, dessa forma você estará organizando quais responsabilidades cada classe terá. No nosso exemplo criaremos o pacote **br.com.agenda.DAO** e dentro desse pacote a Interface **ContatoDAO**.



A nossa interface terá todos os métodos que queremos utilizar no nosso DAO. Nesse exemplo são os métodos para inserir um novo contato, remover um contato, buscar um contato e buscar todos os contatos.

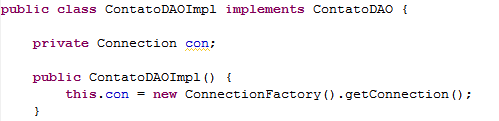
## Criando o DAOImpl

Criaremos agora o pacote **br.com.agenda.DAOImpl**, que ficará responsável por acessar e manipular as informações vindas do banco dados. Nesse pacote criaremos a Classe **ContatoDAOImpl** que irá implementar a interface **ContatoDAO**. Nessa classe ficará toda a responsabilidade de manipular os dados do banco.



## Garantindo a Conexão

Já que vamos manipular os dados do banco, precisamos garantir que haja uma conexão. Para isso, vamos criar uma nova conexão, utilizando a nossa fábrica de conexões (ConnectionFactory), no construtor da Classe **ContatoDAOImpl**, com isso, garantimos que sempre que essa Classe for chamada, haja uma conexão válida.



Agora que o **ContatoDaoImpl** possui uma conexão com o banco, podemos construir nossos métodos de manipulação, utilizando o código SQL necessário.

## Tirando o SQL da aplicação

Para evitar que o código SQL fique misturado com o código Java, é sempre uma boa prática separar uma coisa da outra, deixando um código mais limpo e desacoplado possível. Imagine que por algum motivo você precisa alterar um código SQL, nesse caso você precisa alterar em diversos pontos do seu DAO, quando você tem uma classe para armazenar todo o SQL, a manutenção fica muito mais simples. Para nosso exemplo, criaremos a Classe **SQL**, dentro do pacote **br.com.agenda.util**. Uma Classe apenas com constantes que armazenam todos os nossos códigos SQL.

Vamos criar o código SQL necessário para inserir um novo contato no banco, para isso usaremos a instrução ***INSERT*** ***INTO***.



Agora que temos o código SQL que insere contatos no banco, colocaremos esse código em uma constante Java, dentro da Classe SQL.



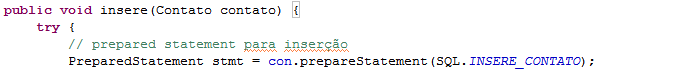
Observe que os parâmetros da instrução INSERT não podem ser fixos, por isso substituímos pelo caractere de ‘?’, afinal esses parâmetros são variados.

**CONSTANTES:**

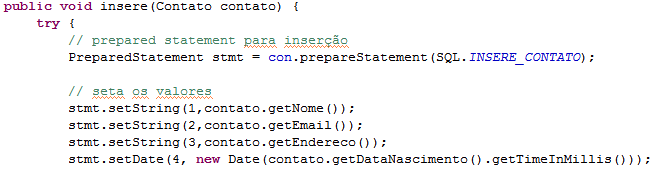
1. As constantes em Java sempre são declaradas em maiúsculas e em caso de nome composto são separados por underline “\_”.
2. Assim como as variáveis, precisamos informar o tipo da constante (String, int, boolean...).
3. Como uma constante não muda seu valor, então, marcamos ela com a palavra-chave ***final***, que fará com que o valor nunca seja modificado.
4. Nossa constante será ***static*** pois só deve haver uma dela na memória.

## Inserindo um Contato

Agora que temos o código SQL, a questão é: como executar esse código no banco de dados? É bem simples, há uma interface chamada **PreparedStatement** para nos ajudar neste trabalho. Basta chamar o método prepareStatement da Connection que criamos, passando o código SQL como argumento, veja como fica o método insere da classe ContatoDAOImpl:



O próximo passo é substitui as ‘?’ do nosso código SQL pelos seus respectivos valores. Para isso a interface PreparedStatement nos oferece uma série de métodos **set** para popularmos os valores variáveis da instrução SQL (setString, setDouble, setDate, etc.) Os métodos set exigem dois parâmetros, o primeiro deve ser o índice e o segundo o valor propriamente dito, por exemplo, se você quer setar uma String na primeira ‘?’ você deve usar o setString(1, “valor”). Observe como fica o nosso método insere:



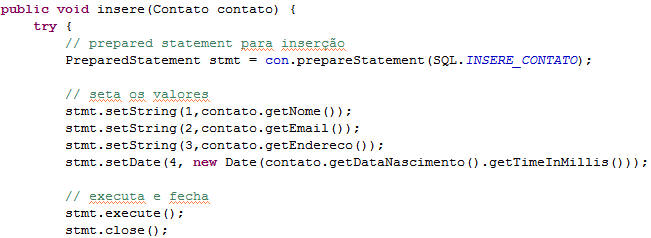
Conheça mais sobre o PreparedStatement em:

<https://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html>

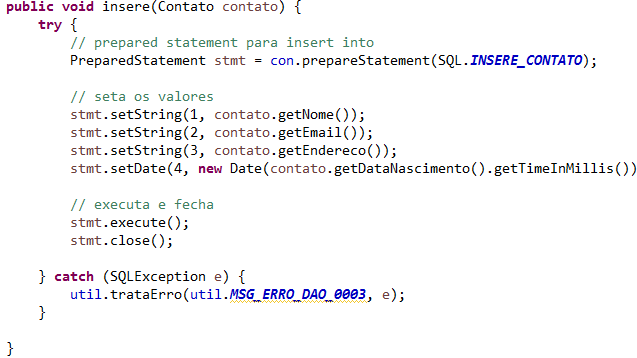
Em casos de dúvidas, utilize o nosso portal:

<http://www.portal.codeshouse.com.br>

Agora basta executar a instrução SQL, chamando o execute():

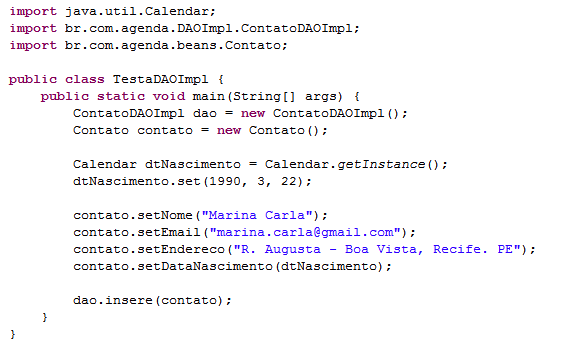


Todo esse código lança uma SQLException, que trataremos com o método **trataErro** da Classe **Util**, que criamos no tópico 5.8.



## Teste o seu DAO

No pacote **br.com.agenda.teste** crie a Classe **TestaDAOImpl** com um método main para testar os métodos do nosso ContatoDAOImpl. Faça diversos testes para cada ação (insere, remove, busca e buscaTodos).

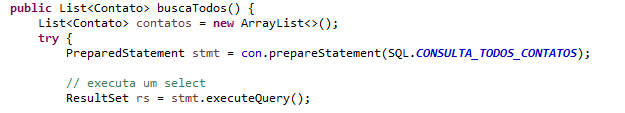


## O buscar Todos

Agora vamos criar o método responsável por buscar todos os contatos no banco, o buscaTodos. Novamente colocaremos nossa instrução SQL dentro de uma constante, utilizando o comando **SELECT** *NOME\_DAS\_COLUNAS* **FROM** *NOME\_TABELA*.

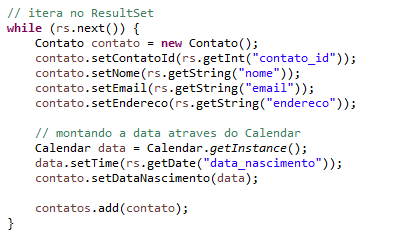


Vamos criar o método buscaTodos() no DAO, criando um PreparedStatement e passando o SQL:



Observe que agora vamos chamar o método executeQuery(), esse método devolve um ResultSet, ou seja um conjunto cheio de dados. Devemos varrer esse conjunto com um while e recuperar cada informação dele. Com as informações em mãos, iremos criar um objeto Contato e adicioná-lo na nossa lista de contatos.

Tome cuidado com o campo dataNascimento, observe que estamos trabalhando com um Calendar e o ResultSet devolve um Date por isso vamos fazer a conversão dos objetos.



Não se esqueça de tratar o SQLException e de fechar as conexões abertas. Veja como ficou o nosso método buscaTodos():



**O Uso do \* nos Selects:**

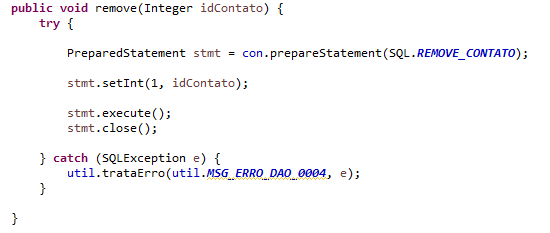
Observe que utilizamos a expressão “*SELECT CONTATO\_ID, NOME, EMAIl, ENDERECO, DATA\_NASCIMENTO FROM CONTATO*” ao invés de simplesmente “*SELECT \* FROM CONTATO*”. O uso do \* é considerada uma má prática de programação, pois você perde o controle do que está sendo retornado na Query, por isso, sempre utilize o nome dos campos no lugar de \*.

## Removendo um Contato

Seguindo o mesmo pensamento vamos construir o nosso método remove que recebe o id do contato a ser removido. Na Classe SQL vamos criar a constante que vai armazenar o nosso código *DELETE*.



Agora vamos construir o nosso método remove. Sem nenhuma novidade. Não se esqueça de setar o parâmetro do id no PreparedStatement. Após construir o seu método remove, volte na Classe **TestaDAOImpl** e faça o teste do mesmo, passando Ids validos. Lembre-se de popular o banco e deixa-lo com dados consistentes para que possamos continuar testando a nossa aplicação.

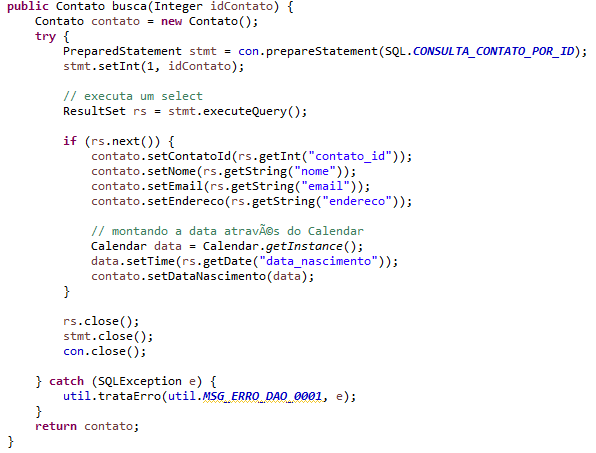


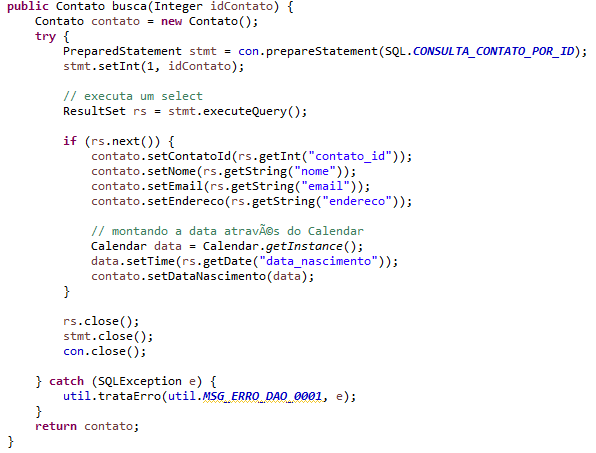
## A pesquisa por Id

A pesquisa por ID vai seguir a mesma ideia do buscaTodos, porém agora temos um parâmetro para ser passado na query. Vamos criar nossa Constante:



Em seguida, vamos implementar nosso método busca. Observe que estamos trabalhando com um Calendar no campo dataNascimento e o ResultSet devolve um Date por isso vamos fazer a conversão dos objetos.





Quer aprender mais sobre as instruções SQL?

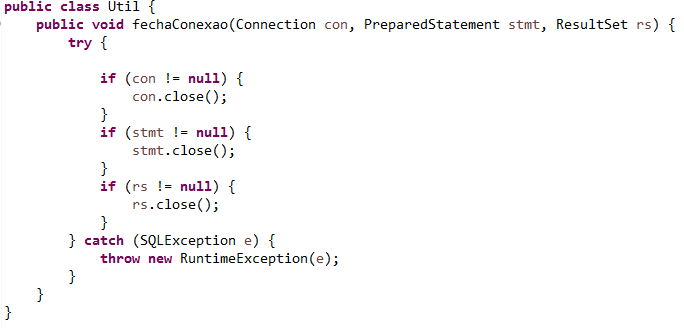
Acesse <http://www.w3schools.com/sql/>

## Fechando as conexões de forma segura e elegante.

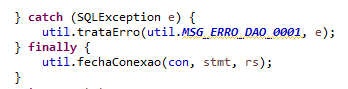
Vamos criar a Classe **Util** dentro do pacote **br.com.agenda.util** para criar uma forma mais elegante de fechar a conexão, o PreparedStatement e o ResultSet. Observe que o trecho de código abaixo está sendo repetido em vários pontos do nosso DAO:



Na Classe **Util** criaremos o método **fechaConexao** que vai receber uma conexão, um PreparedStatement e um ResultSet. Esse método é bastante simples, tudo o que ele faz é fechar os objetos passados, lembrando de verificar se o objeto não está nullo para não gerar um NullPointerException.



Após criar o método **fechaConexao** iremos substituir o con.close(), stmt.close() e rs.close() por fecharConexao(com, stmt, rs). Lembrando que esse método deve estar dentro de um bloco finally. Observe como nosso código fica mais enxuto e mais organizado:



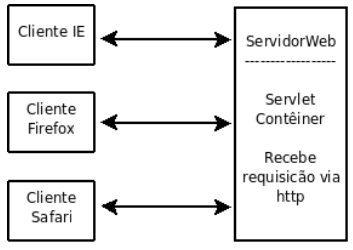
Capítulo

# A Camada WEB



## Servlets

As Servlets são a primeira forma que veremos de criar páginas dinâmicas com Java. Usaremos a própria linguagem Java para isso, criando uma classe que terá capacidade de gerar conteúdo HTML. O nome "servlet" vem da ideia de um pequeno servidor (servidorzinho, em inglês) cujo objetivo é receber chamadas HTTP, processá-las e devolver uma resposta ao cliente. Cada servlet é um objeto Java que recebe as requisições (request) e produz algo (response), como uma página HTML dinamicamente gerada e devolve para o cliente.



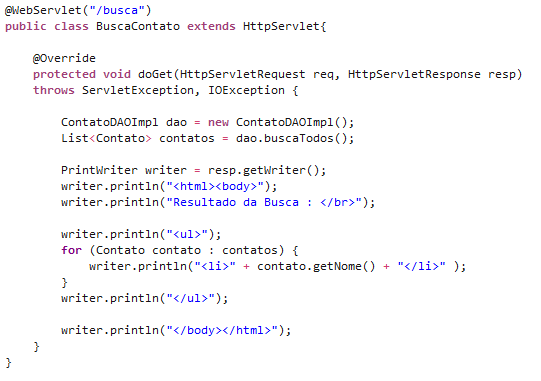
O comportamento das servlets que vamos ver neste capítulo foi definido na classe HttpServlet do pacote javax.servlet.

## O BuscaContato

Para começar a entender o funcionamento de uma Servlet, vamos criar a Classe **BuscaContato** dentro do pacote **br.com.agenda.web**, essa Classe deve herdade de **HttpServlet**.



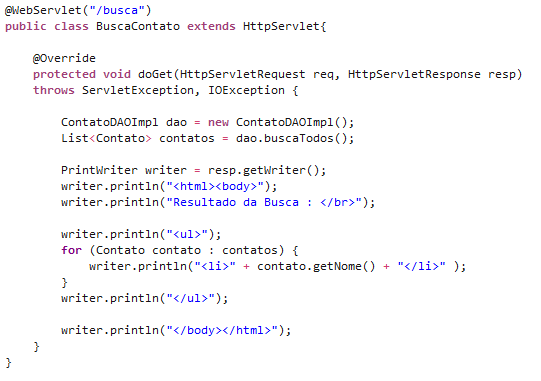
Como ela é uma Servlet e deve ser acessada a partir do nosso navegador, iremos anotar a nossa Classe com @WebServlet e informar o path, ou seja, o recurso, a Servlet que queremos acessar, dessa forma, nossa Classe ficará assim:



O objetivo dessa Classe será buscar os contatos no banco e exibir no navegador e a requisição será dada através de uma URI.

## O Get

Toda requisição que é feita via URI envia uma solicitação do tipo **get** para o servidor, ou seja, o cliente está pedindo alguma coisa ao servidor, nesse caso iremos sobrescrever o método doGet:

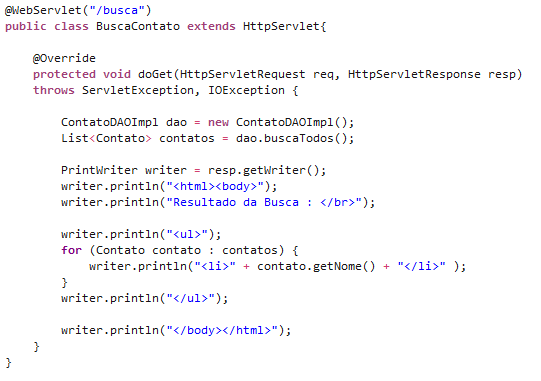


Note que o método doGet é composto por uma requisição (HttpServletRequest req) e por uma resposta (HttpServletResponse resp), ou seja, já sabemos de onde vem nossa requisição e por onde vamos enviar as respostas.

No nosso exemplo, queremos que toda vez que o usuário acessar a nossa Servlet BuscaContato apareça no navegador uma lista com todos os Contatos cadastrados no banco. O usuário irá acessar nosso sistema através do endereço <http://localhost:8080/Agenda> e para ter acesso a nossa Servlet ele deverá acessar <http://localhost:8080/Agenda/busca>. Esse endereço pode mudar, caso o Tomcat esteja rodando em uma porta diferente, por padrão ele roda na 8080.

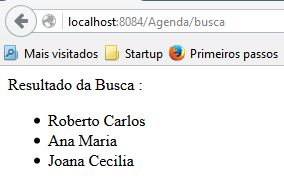
Enquanto isso, o nosso método doGet irá chamar o método buscaTodos() da Classe ContatoDAOImpl e em seguida irá varrer todos os contatos e exibir no navegador. Para exibir os dados iremos utilizar o PrintWriter.println. Porém além de imprimir os dados, também devemos imprimir todos as tags HTML necessárias para que nossos dados apareçam corretamente no navegador.

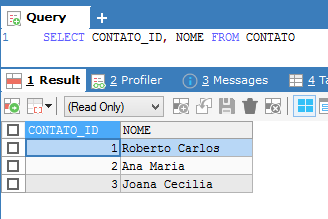
Observe como fica o nosso método doGet:



Agora vamos executar nossa aplicação e vamos até o navegador para testar a nossa Servlet, não se esqueça que o banco de dados deve estar no ar. Para acessar nossa aplicação iremos digitar no navegador a URI <http://localhost:8080/Agenda/busca>. Lembrando que o código <http://localhost:8080/> é onde está rodando o seu servidor, pode ser na porta 8084, 8081, ou outra porta. O **/Agenda** é o nome do nosso projeto, ou seja, do nosso sistema. E **/busca** é o recurso, ou seja, a Servlet que que queremos acessar, acessar através do método doGet.

Observe a seguir o resultado da página no navegador e como está o nosso banco de dados:

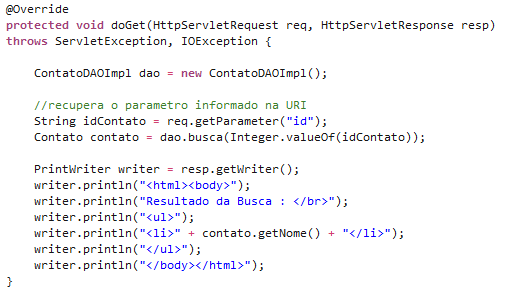




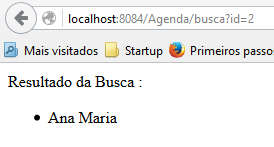
## Enviando e recuperando parâmetros no doGet

Uma URI também pode ser composta por parâmetros, filtros. Vamos melhorar nosso método doGet para buscar um contato de acordo com o id passado na URL, para isso nosso DAO já possui o método busca que traz um contato a partir do seu Id.

Como o Id vai ser passado na URL, iremos recuperar ele na request, e em seguida usar esse parâmetro no método busca de ContatoDAOImpl, veja como fica o nosso método agora:

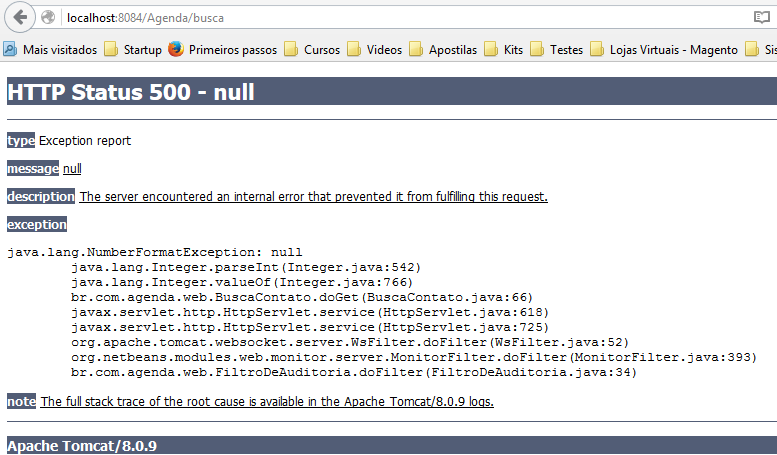


Para testar, devemos restartar o servidor e chamar a URl <http://localhost:8080/Agenda/busca?id=2>. Observe que agora, após o /busca estamos passando o parâmetro **?id=2**.



## Deixando o buscaContato mais flexível

Observe que agora se não passarmos o parâmetro, a nossa página retorna um erro de NumberFormatException, ou seja, somos obrigados a informar o Id.



Para deixar nosso código ainda mais flexível vamos permitir tanto a busca com passagem de parâmetro como sem passagem de parâmetros, isso através de uma simples verificação se o Id foi ou não informado como parâmetro na URI.



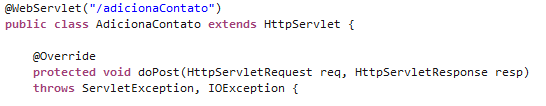
Agora o nosso código está mais flexível, deixando tanto o usuário passar o parâmetro como não e das duas formas ele terá as informações vindas do banco. Faça diversos testes no seu browser.

## Gravando um novo contato

Vamos criar uma nova funcionalidade, a de gravar contato no nosso banco de dados. Desta vez, para enviar as informações necessárias para gravar o contato iremos criar um formulário na nossa página **index.html**, assim o usuário poderá enviar as informações no formulário ao invés de ficar digitando tudo na URL. É um formulário bem simples, mas por enquanto não vamos nos preocupar com características visuais da nossa página. Também usaremos uma tabela apenas para alinhar os campos do formulário. Veja abaixo o código da nossa página index.html:



Observe que o nosso formulário vai enviar para o Servidor um Post, isso porque não é aconselhável enviar várias informações via Get. Por padrão todo formulário envia os dados via Get, por isso devemos especificar quando queremos utilizar o Post. O próximo passo é criar a nossa Servlet AdicionaContato, onde iremos sobrescrever o método doPost:



Nesse método iremos instanciar um objeto DAO, recuperar os dados do request para passa-los para um objeto contato, em seguida chamar o método insere passando o contato que criamos. Lembrando de tratar a Data de forma correta.



Por fim, vamos apenas gerar uma mensagem de sucesso, informando que o contato foi cadastrado com sucesso. Observe como ficou o nosso método doPost:



Não esqueça de fazer seus testes no navegador e verificar se o contato realmente foi gravado no banco de dados.

**GET vs POST**

* O Tamanho da URL pode ser limitado e o método Get pode perder informações devido ao tamanho, por isso o Get não deve ser usada para enviar informações e sim para trazer, solicitar informações.
* Toda requisição do tipo Get pode ser cacheada no navegador.
* O Post envia os parâmetros no corpo da requisição
* O Post também não é cacheado.

O HTTP trabalha com outros métodos além do Get e do Post, porém esses são os que mais usamos.

Leia mais em <http://www.w3schools.com/tags/ref_httpmethods.asp>.

Capítulo

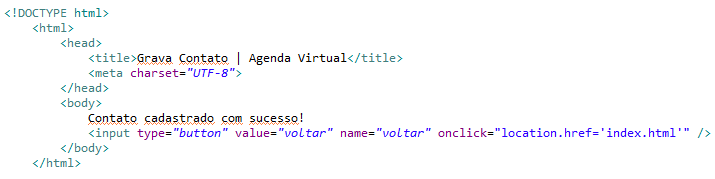
# Colocando o HTML no seu devido lugar

As nossas Servlets estão desempenhando bem suas tarefas, porém temos um grande problema. As Classes Java estão cheias de código HTML, a nossa aplicação está bem simples e bem pequena, porém imagine fazer uma aplicação grande, corporativa e no meio da Classe Java ter um monte de código HTML e pior ainda, além do HTML começar a misturar código CSS e Javascript, tudo no meio do código Java.... Não dá muito certo. Está na hora de começarmos a colocar cada coisa no seu devido lugar.

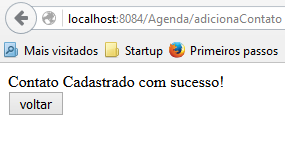


## O AdicionaContato

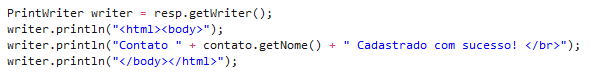
Vamos jogar o código HTML da Servlet AdicionaContato dentro de uma página HTML separando cada coisa (Java e HTML). O primeiro passo será criar o arquivo **sucesso.html**:



Também vamos criar um botão apenas para retornar para a página principal, para isso vamos usar o onclick e fazer o redirecionamento via Javascript. Observe como deve ficar nossa tela:



O próximo passo é alterar a Servlet, a partir de agora ela não vai mais exibir o código HTML, apenas redirecionar o navegador para sucesso.html, onde vai estar todo o código de view (tela). Vamos substituir o trecho abaixo:



Por um novo código que fará o redirecionamento. Para isso usaremos o RequestDispatcher, informando para onde queremos despachar a requisição, que no nosso caso, vai para “/sucesso.html”. Em seguida enviamos a requisição (req) e a resposta (resp) para a página:



Repare como o código ficou bem mais simples e desacoplado. Reinicie o servidor e faça seus testes.

## JavaServer Pages

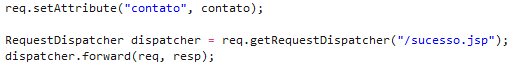
O código ficou bem melhor, porém agora não estamos mais exibindo o nome do Contato que foi cadastrado, como fazíamos antes, agora simplesmente exibimos a mensagem de sucesso sem o dado referente ao nome.

Vamos fazer nossa página de sucesso recuperar o nome do Contato que foi cadastrado e exibir na tela. Porém para fazer isso não vamos utilizar apenas o HTML e sim o **JavaServer Pages** (JSP).

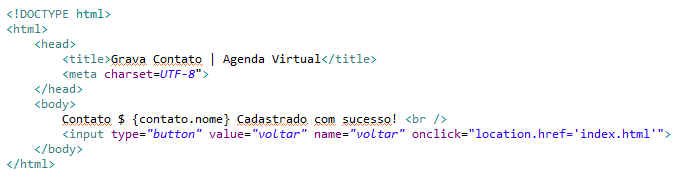
**JavaServer Pages** (JSP) é uma tecnologia que ajuda os desenvolvedores de software a criarem páginas web geradas dinamicamente baseadas em HTML, XML ou outros tipos de documentos. Lançada em 1999 pela Sun Microsystems, JSP é similar ao PHP, mas usa a linguagem de programação Java.

O primeiro passo é trocar (renomear) o arquivo sucesso.html por **sucesso.jsp**, agora temos uma página capaz de recuperar dados Java, uma página onde podemos escrever código Java, para que possamos adicionar comportamento dinâmico, como declaração de variáveis, condicionais (if), loops (for, while) entre outros.

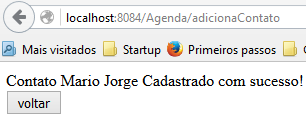
Agora que temos uma página JSP, vamos fazer a Servlet enviar a informação que queremos, para isso a Servlet tem que setar o objeto Contato na requisição que estamos trabalhando e assim a página JSP será capaz de enxergar esse objeto e trabalhar com ele:



O Java já está setando o objeto contato no request, o próximo passo é recuperar esse objeto na JSP, para isso utilizaremos um recurso conhecido como **Expression Language**, que nos permite acessar dados vindos do Java, ou seja, dados que foram setados no request. Observe como fica o nosso arquivo **sucesso.jsp**:

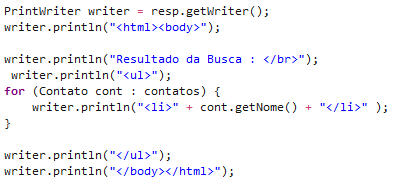


Através do ${ } podemos ter acesso ao objeto Java. Observe que estamos fazendo ${contato.nome}, isso porque o JSP entende que estamos tentando acessar o método getNome() do objeto contato. Reinicie o servidor e faça seus testes no navegador:

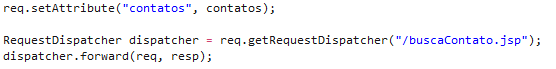


## Usando ****Scriptlet****s

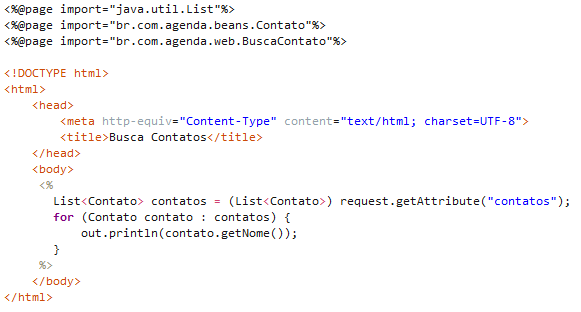
A nossa próxima tarefa será alterar o BuscaContato, que retorna uma lista com todos os Contatos cadastrados. O primeiro passo será setar a lista com todos os contatos na request. Vamos substituir o código abaixo que está na Classe BuscaContato:



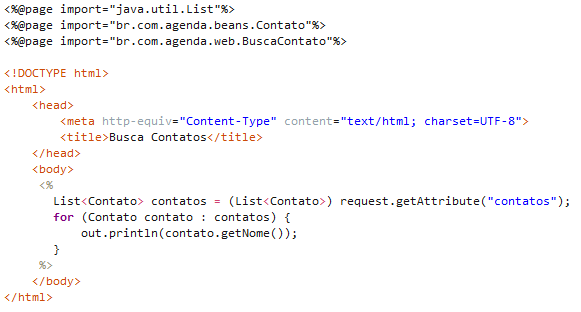
Pelo nosso RequestDispatcher, igual fizemos antes:



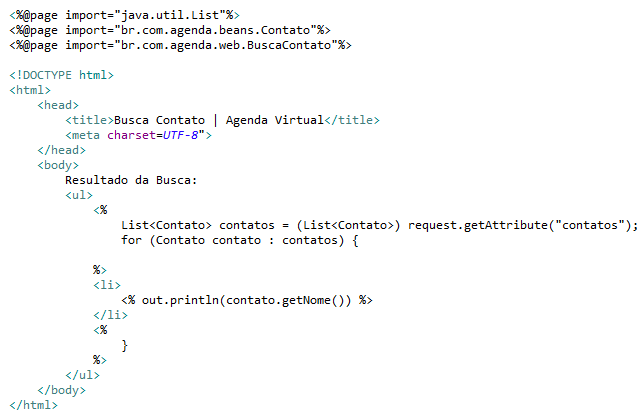
Agora vamos construir nossa JSP, primeiro vamos criar o arquivo **buscaContato.jsp.** Vamos trabalhar de uma forma um pouco diferente, não queremos acessar um método de um objeto mais sim uma lista completa. Já que no Java setamos o atributo “contatos” no request, na JSP vamos recuperar esse atributo através do getAttribute da seguinte forma:



Em seguida vamos varrer essa lista usando o forEach do Java e para imprimir os dados na tela vamos usar o out.println() disponível em toda JSP. Não esqueça que trechos de código Java devem estar dentro de <% e %>, ou seja, Scriptlet. Observe como fica nosso código:



Não se esqueça de fazer os imports necessários. Veja como fica o nosso arquivo **buscaContato.jsp** completo:



Observe que o código Java fica separado do código HTML através do <% e %>, com isso é possível alternar entre os códigos. Para exibir os contatos em forma de lista no navegador utilizamos as tags <ul> e <li>.

**Scriptlet**

Scriptlet é o código escrito entre <% e %> . Esse nome é composto da palavra script (pedaço de código em linguagem de script) com o sufixo let, que indica algo pequeno.

## JSTL para escrever menos

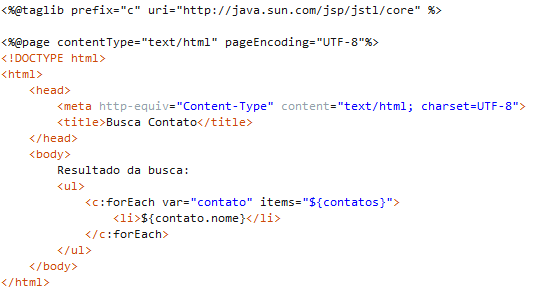
Repare que mesmo separando o código HTML do Java ainda sim escrevemos muito na nossa página JSP para fazer ações simples e o código fica todo bagunçado. Existe uma biblioteca que pode na auxiliar nessa função, a JSTL (JSP Standard Tag Library). O JSTL tem como objetivo enxugar o código da nossa JSP.

A JSTL é a API que encapsulou em tags simples toda a funcionalidade que diversas páginas Web precisam, como controle de laços (fors), controle de fluxo do tipo if else, manipulação de dados XML e a internacionalização de sua aplicação.

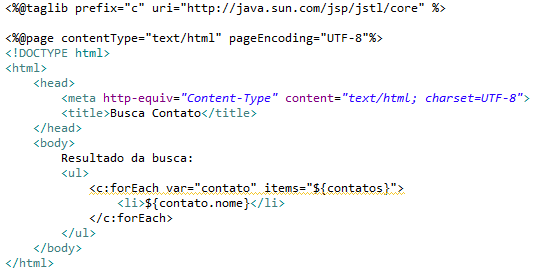
Para utilizar o JSTL precisamos fazer o download da biblioteca e adicionar em nosso projeto. O site oficial é <https://jstl.java.net/>. Após o download basta adicionar o jar ao projeto da mesma forma que fizemos no item 5.4 dessa apostila.

## Trabalhando com Taglibs

Com o JSTL em mãos, o primeiro passo é declarar a sua utilização em nossa página, na primeira linha do arquivo buscaContato.jsp vamos declarar a utilização da taglib c do JSTL/core:



O próximo passo é varrer a lista, agora podemos usar a tag <c:forEach> que vai ter o mesmo papel do forEach do Java. Nesse forEach vamos declarar a lista que vem no request (items=“${contatos}”). Em seguida declaramos uma variável que vai armazenar cada item da lista (var= “contato”). Dentro do forEach recuperamos apenas a informação que queremos exibir (${contato.nome}). Veja como fica nossa página completa:



Observe que utilizamos bem menos código que antes. Reinicie o servidor e faça seus testes.

Capítulo

# Otimizando nossas Servlets

Nosso código está bem melhor que antes e cada vez mais desacoplado, cada coisa está indo para o seu devido lugar. Porém ainda temos um pequeno problema. Observe que estamos sempre repetindo os seguintes passos:

* O cliente pede uma lógica a Servlet
* A Servlet de lógica é executada
* A Servlet redireciona o cliente para alguma JSP
* O fluxo recomeça.

Vamos dar um exemplo, o cliente chama o BuscaContato, o BuscaContato executa toda a lógica e depois o BuscaContato redireciona o usuário para a JSP. Os mesmos passos se repetem para o AdicionaContato.

A medida que nosso sistema crescer vamos estar o tempo todo repetindo esses passos. O que poderíamos fazer para melhorar esse processo?

Poderíamos criar uma **Servlet intermediaria**, onde o cliente iria acessá-la e falar o que ele precisa, e ela vai chamar a lógica correta e em seguida fazer o redirecionamento dele para a JSP correta, ou seja, essa **Servlet intermediaria** seria um **Controller**, um controlador que atende as requisições e chama a lógica correspondente. Agora que ficou claro, vamos ao trabalho....

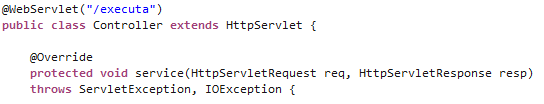


## Criando o Controller

Vamos criar uma Servlet Controller para fazer essa função de **intermediaria**, sua urlPatterns será “/executa”, ou seja, a partir de agora o cliente sempre terá que passar por **/executa** para acessar alguma coisa dentro do nosso sistema.



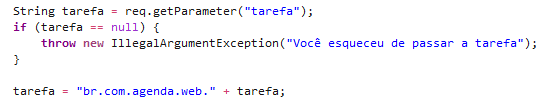
A nossa Servlet tem que ser genérica e aceitar tanto **doGet** como **doPost**, para isso, a API possui o método **service**, que consegue entender qualquer tipo de requisição vinda do cliente.



O próximo passo é saber qual “tarefa” o cliente deseja, é a tarefa de BuscaContato ou de AdicionaContato? Não importa, o que importa é recuperar a tarefa que vai vim na request, para fazer isso basta usar o getParameter que já conhecemos. **Porém** o cliente é obrigado a informar a tarefa, caso contrário vamos lançar uma Exception:

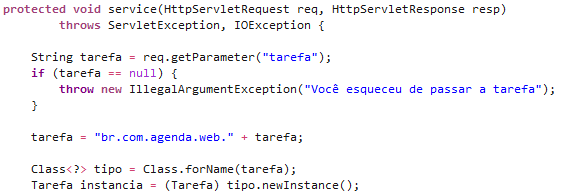


A tarefa nada mais é do que a Servlet que queremos chamar BuscaContato ou AdicionaContato, porém além de saber o nome da Servlet temos que ter o nome do pacote em que ela está por isso vamos concatenar o nome do pacote + a tarefa:



Agora que sabemos qual tarefa queremos, como iremos executá-la? Se o cliente pediu “BuscaContato” Como iremos instanciar uma Classe BuscaContato e poder executar a lógica dentro dela? Para isso, o Java nos fornece o recurso do Class.forName.

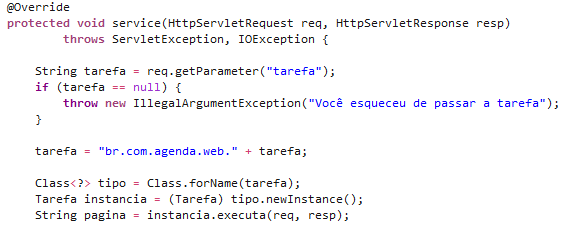
O que vamos fazer agora é usar a String tarefa, que contém o nome de uma Servlet, porém não queremos uma String e sim a Classe mesmo, para isso usamos o Class.forName que vai nos devolver a Classe propriamente dita (Não uma String, nem um Objeto, mas sim a Classe). Em seguida vamos instanciar essa Classe, mas que Classe é essa? BuscaContato ou AdicionaContato? Como não sabemos qual é a Classe, vamos usar o Polimorfismo, criando a Interface Tarefa e a partir de agora tanto BuscaContato como AdicionaContato irão implementar essa Interface:



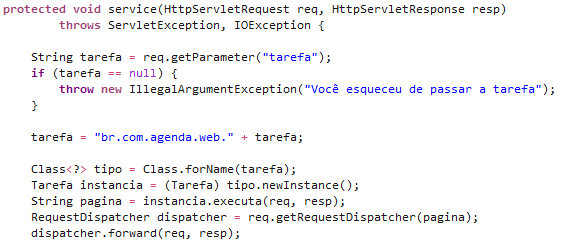
A Interface Tarefa, por sua vez, vai ter o método **executa** que recebe um request e um response. Esse método será implementado em cada Servlet de lógica da nossa aplicação, ou seja, ele será implementado em BuscaContato e em AdicionaContato. E esse método é justamente o processamento que o cliente espera, uma consulta no banco de dados, a devolução de uma lista, um insert.... Veja como fica a Interface Tarefa:

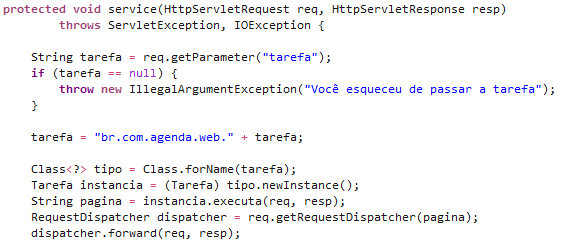


Voltando ao Controller, agora nosso código cria uma instancia de um objeto acordo com a que o cliente pediu e chama o método **executa** desse objeto, que vai fazer toda a lógica de negócio:

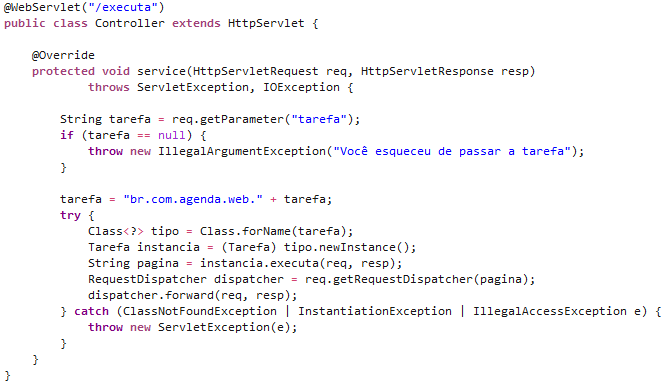


Após executar toda a lógica de negócio, o método **executa** devolve uma String que é justamente o nome da página para onde o cliente deve ser redirecionado, no caso de AdicionaContato, o método **executa** devolte “sucesso.jsp” que é para onde o cliente vai:





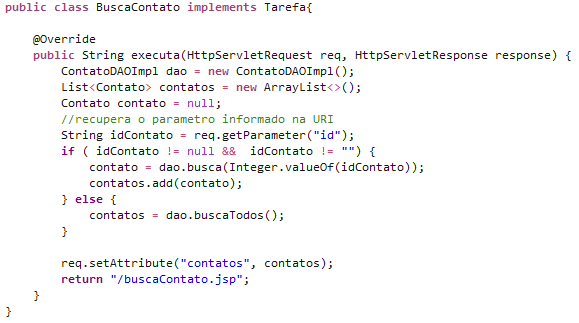
Todo esse código pode gerar algumas exceções, vamos trata-las. Veja como fica nossa Servlet Controller:

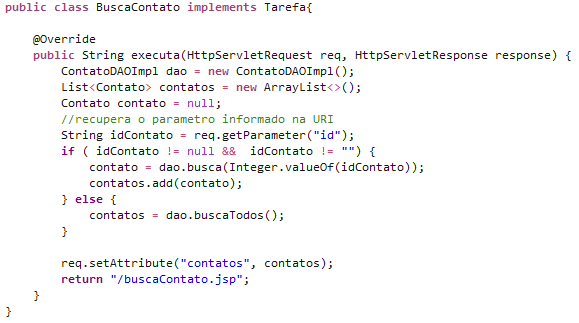


## Melhorando o BuscaContato

Agora que temos a nossa Servlet Controller, que será a Servlet intermediaria, o próximo passo agora é atualizar as demais Servlets, ou seja, vamos deixa-las apenas com lógica de negócio.

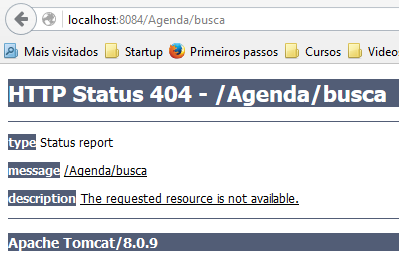
Primeiro vamos a BuscaContato. Agora nossa Classe não vai mais estender de HttpServlet e sim implementar a Interface Tarefa, com isso somos obrigados a escrever o método executa. Executa? Executa o que? A busca de contato que é justamente o que já existe dentro do método doGet que tínhamos feito anteriormente, ou seja, o doGet não vai mais existir e toda lógica que havia dentro dele vai para dentro do método executa. Além disso não vamos mais precisar do RequestDispatcher (pois colocamos ele dentro da Servlet Controller), tudo o que precisamos dizer qual página queremos exibir, quem vai redirecionar é o Controller:



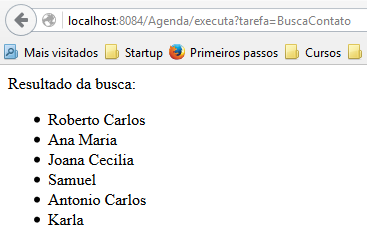


**OBS**: Para manter a boa prática, no futuro, utilize a Classe ConstantesSistema para armazenar os nomes das JSPs dentro de constantes que podem ser usadas em qualquer parte do sistema.

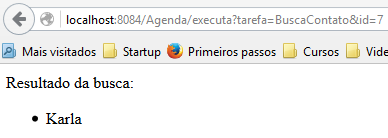
Vamos reiniciar o servidor e testar nossa aplicação? Mais observe o que acontece se eu chamar <http://localhost:8080/Agenda/busca> como fazia antes:



Agora eu não posso mais chamar direto como era feito antes, agora todo o meu sistema tem que passar por executa e dizer qual a tarefa que quer executar:

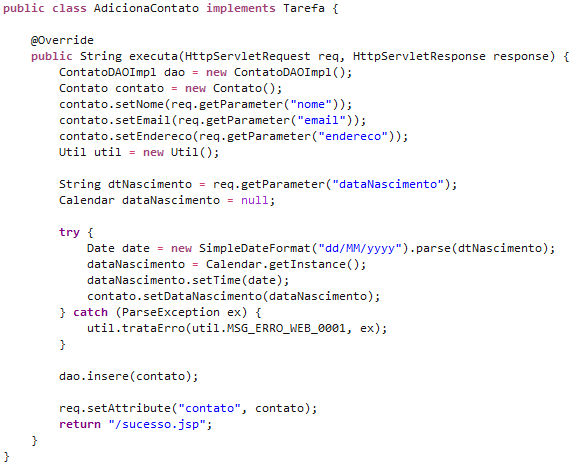


Mas e se eu quiser buscar apenas um único contato passando como parâmetro o seu id? Basta informar isso na URI:

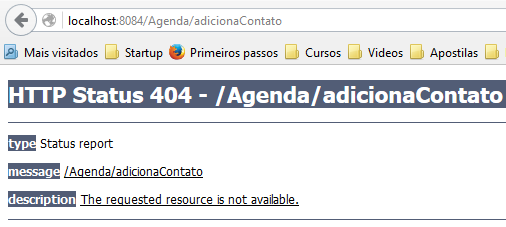


## Melhorando o AdicionaContato

Vamos atualizar agora o AdicionaContato. Seguindo a mesma ideia teremos:



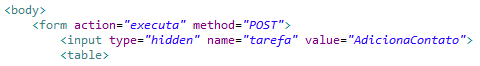
Em seguida vamos reiniciar o servidor e testar a nossa aplicação, preenchendo o formulário do index e clicando no botão salvar:



Novamente uma exceção é lançada, isso porque o nosso formulário está tentando chamar a Servlet AdicionaContato, que não existe mais, quem existe agora é Executa, vamos ajustar nosso formulário e tentar novamente. No index.html faça a seguinte alteração e em seguida teste sua aplicação:



Agora sim, tudo está funcionando como esperado. Porém vamos melhorar a legibilidade do nosso formulário, usando um **input hidden** que passa a tarefa, essa forma é muito mais usada:



Nosso sistema está muito mais organizado e estruturado, obedecendo os padrões da programação WEB e cada coisa está no seu devido lugar.

Capítulo

# O Padrão Arquitetural MVC

Generalizando o modelo acima, podemos dar nomes a cada uma das partes dessa nossa arquitetura. Quem é responsável por apresentar os resultados na página web é chamado de Apresentação (**View**), ou seja, nossos arquivos **html** e **jsp**.

A servlet que faz os dispatches para quem deve executar determinada tarefa é chamada de Controladora (**Controller**).

As classes que possuem a lógica de negócio, representam suas entidades e as que te ajudam a armazenar e buscar os dados são chamadas de Modelo (**Model**).

Essas três camadas separadas por responsabilidades, formam um padrão arquitetural chamado de **MVC**, ou **Model View Controller**. Ele pode sofrer variações de diversas maneiras. O que o MVC garante é a separação de tarefas, facilitando assim a reescrita de alguma parte, e a manutenção do código.

Existem diversos frameworks que utilizam o padrão arquitetural MVC tanto na linguagem Java como em outras linguagens de programação.

Capítulo

# Trabalhando com o Bootstrap

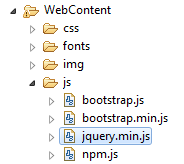
Nossa aplicação já está bem estruturada, agora vamos dar uma melhorada, nosso front-end está muito primitivo e nada atraente. Para facilitar nosso trabalho vamos utilizar o Bootstrap e deixar ele cuidar da nossa camada View. Vamos fazer o download do Bootstrap no próprio site: <http://getbootstrap.com>.

O Bootstrap é um framework front-end que facilita a vida dos desenvolvedores abstraindo o CSS, Jquery e Javascript. Após fazer o download do Bootstrap, vamos extrair o arquivo zipado e copiar as pastas css, fonts e js para dentro do nosso WebContent:



Além do Bootstrap precisamos fazer o download do jquery.min disponível em: <http://code.jquery.com/jquery-2.2.0.min.js>

Ao acessar o link acima, copie todo o conteúdo da página e cole em um arquivo de texto, em seguida o salve com o nome jquery.min.js e salva dentro da pasta js do WebContent:



Agora já estamos prontos para trabalhar com o Bootstrap.

Capítulo

# Melhorando nossa Aplicação

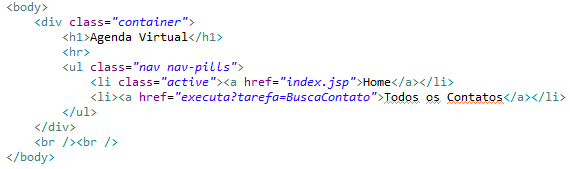


## Criando o Menu

Agora que estamos com Bootstrap em nossa aplicação vamos criar a página **menu.jsp**, onde criaremos um Menu para utilizar em nossas páginas e facilitar a navegação do usuário. O primeiro passo é apontar para o bootstrap dentro da sessão head:



A partir de agora, todas as nossas páginas deverão fazer referência ao Bootstrap. Em seguida iremos criar o menu propriamente dito dentro do body:



## Alterando o Index

Agora que criamos o menu, vamos utilizá-lo nas nossas páginas, começando pelo index, que a partir de agora será **index.jsp** (vamos renomear o arquivo)**.**  O próximo passo é apontar para o Bootstrap e “importar” o Menu através da jstl:





Agora vamos ajustar o formulário:



Essas class que estamos utilizando são estilos do CSS do Bootstrap, incialmente não se preocupe com isso, a documentação do CSS está disponível em: <http://getbootstrap.com/>

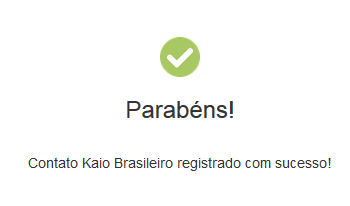
Você também pode conhecer mais em: <http://www.w3schools.com/bootstrap/>

## Melhorando nossa Página de Sucesso

Vamos alterar nossa página de sucesso para que ela fique genérica e possa ser usada em qualquer parte do sistema, quando alguma Classe chamar a tela de sucesso, basta informar o nome do contato e uma ação (Cadastrado, Alterado, Removido, etc.), dessa forma a tela de sucesso deverá exibir “Contato XXX foi XXX com sucesso! ”. Também vamos inserir uma imagem para deixar nossa página mais atraente e importar o menu:



Veja como agora nossa página está bem mais atraente.



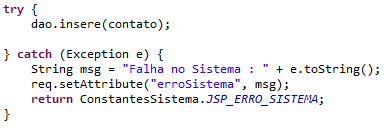
## Lançando Exceções mais elegantes para o usuário

Seguindo a mesma ideia vamos criar a nossa página de **erroSistema.jsp**:

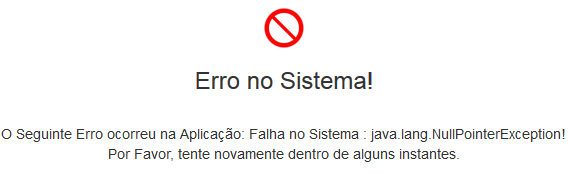




Não há nenhuma novidade nessa tela, apenas vamos declarar o Bootstrap no head, importar o cabeçalho, chamar uma imagem de erro e imprimir uma mensagem na tela. Nas nossas classes de lógica, vamos colocar um try-catch para recuperar os erros e retornar a erroSistema.jsp:



Observe como fica nossa tela caso seja cadastrado um contato sem informações:



Dessa forma a nossa tela de erro fica genérica, pode ser utilizada em qualquer parte do sistema. O toString() da exceção não é interessante para o usuário final, ele não sabe o que é NullPointerException. Porém vamos deixar dessa forma, apenas para fins didáticos.

## Melhorando o BuscaContato

O próximo arquivo que vamos alterar é o buscaContato.jsp, vamos seguir a mesma estrutura das demais classe. A única diferença é que vamos montar uma tabela para os dados ficarem mais bem posicionados na tela. Não se esqueça de referenciar o Bootstrap e de importar o menu:





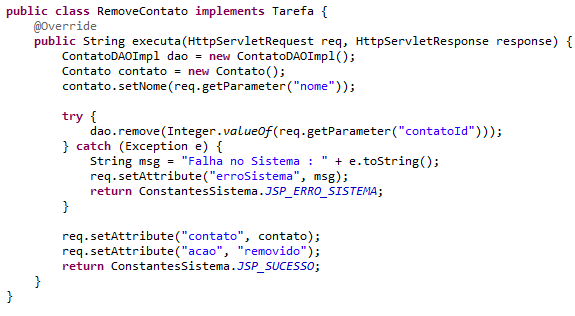
Capítulo

# Removendo um Contato



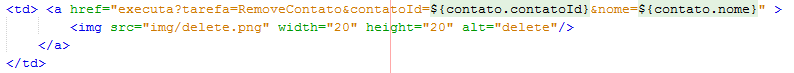
## O RemoveContato

Agora sim, nossa aplicação está atraente, usável e bastante organizada. Vamos agora construir a classe responsável por remover um contato, excluir ele do banco. No DAO já temos o método remove, basta criarmos uma classe de lógica que vai chamar esse método:



## Alterando A JSP

Vamos alterar a página buscaContato.jsp para que que quando o usuário clicar na imagem de excluir, ele será redirecionado para a página de sucesso e o contato será excluído do banco. Para isso usaremos o atributo href passando a URI que queremos solicitar:



O Sistema não está validando antes de fazer a exclusão, esse, obviamente, é o cenário ideal, validar através de Javascript se o usuário realmente quer excluir o registro ou não.

Capítulo

# Desafio...

Para finalizar nosso sistema vamos deixar dois desafios que vão deixar nossa aplicação bem completa.

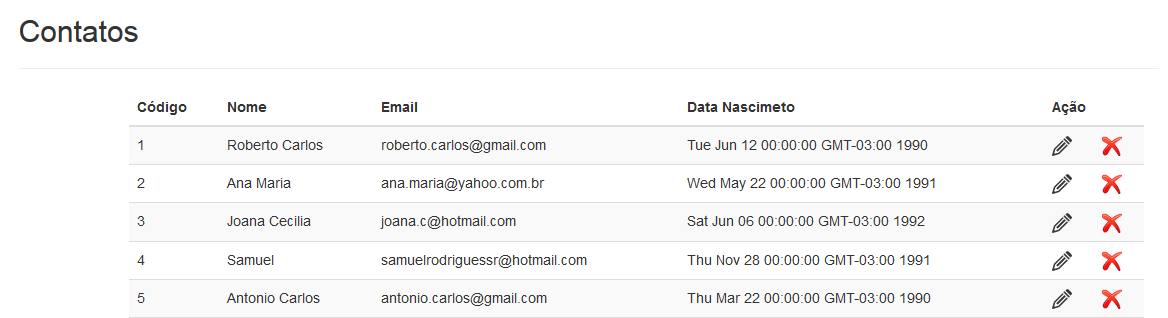


## O Editar Contato

Como desafio dessa apostila você deve criar ao lado do ícone de excluir contato (  ), um ícone de Editar (  ), na tela de BuscaContato.jsp.

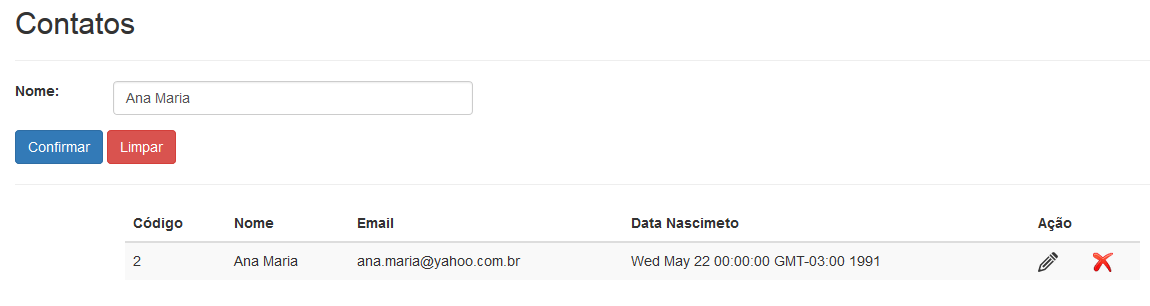
Ao clicar no ícone de Editar o sistema deve abrir outra janela com um formulário contendo todos os dados do contato selecionado, para que o usuário possa editar suas informação, lembrando que o ID nunca pode ser editado, apenas nome, endereço, data de nascimento....

Após editar os campos e clicar em ‘Salvar’ o sistema deve fazer um update no banco de dados, alterando o registro do contato em questão. Divirta-se!



## Filtrando a Busca de Contatos

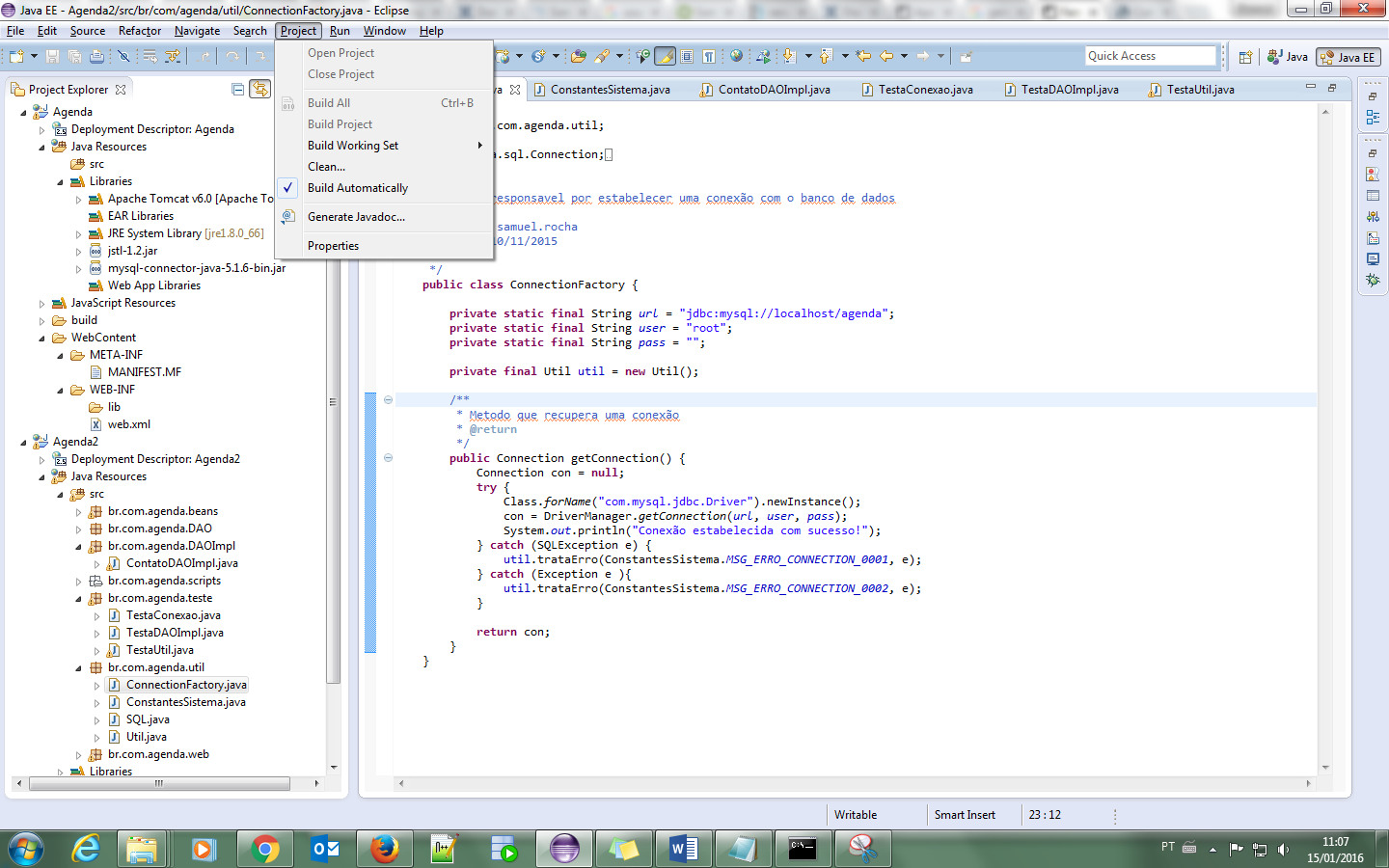
O próximo desafio é criar um formulário na página BuscaContato.jsp, apenas com o campo nome e os botões confirmar e limpar. A ideia é o usuário digitar parte do nome e clicar em confirmar e a partir disso o sistema deve retornar apenas o usuário ou os usuários que foram encontrados com aquele nome. Boa Sorte!



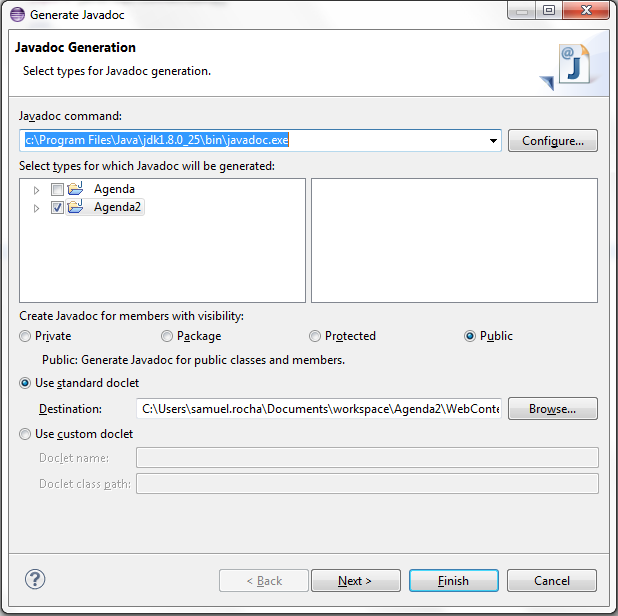
Capítulo

# Gerando a Documentação do Sistema

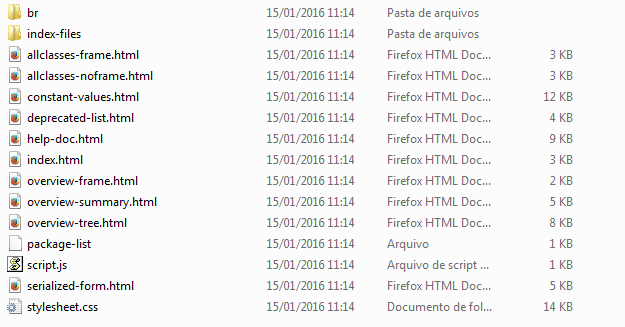
Agora que nossa aplicação está finalizada e funcionando corretamente, vamos gerar a documentação da mesma a partir de todos os JAVADOCs que criamos nas classes e nos métodos. O primeiro passo é clicar em Project e escolher Generate Javadoc...



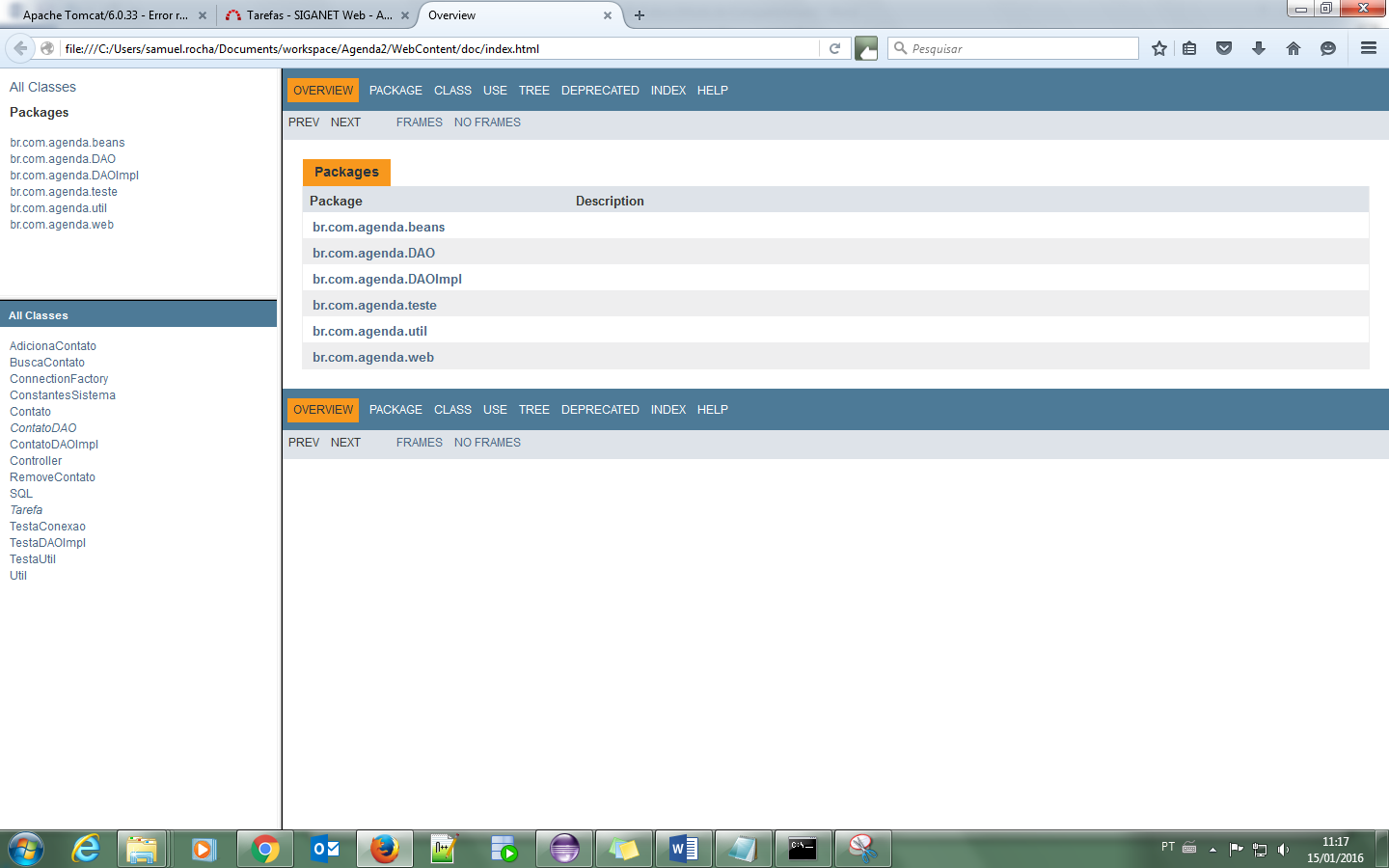
Na tela que abrir selecione o seu projeto. Em javadoc Command coloque o caminho do seu javadoc (ele vai estar dentro da pasta do jdk) e em Destination escolha o local onde será gerada a documentação (não precisa alterar). Em seguida clique em Finish.



Vá até o diretório que você colocou em Destination e abra o arquivo indez.html no seu navegador e veja como ficou a documentação do seu projeto.

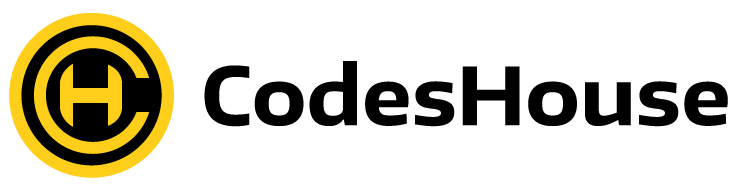


Veja como a documentação do sistema ficou extremamente organizada. Isso é importante em qualquer equipe de desenvolvimento de software.



**ANOTAÇÕES**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



**CodesHouse Treinamentos**

A CodesHouse Treinamentos é uma empresa especializada na capacitação de desenvolvedores de software, que tem como objetivo oferecer treinamentos de máxima qualidade, e relacionados às principais tecnologias utilizadas pelas empresas. Através desses treinamentos, seus alunos tornam-se capacitados para atuar no mercado de trabalho.

**Termo** **de** **Uso**

**Termo de Uso**

Todo o conteúdo desta apostila é propriedade da CodesHouse Treinamentos. A apostila pode ser utilizada livremente para estudo pessoal. É proibida qualquer utilização desse material que não se enquadre na condição acima sem o prévio consentimento formal, por escrito, da CodesHouse Treinamentos.