Comparação de desenvolvimento de aplicações web com ASP.NET MVC 5 e Spring MVC 4

José Rafael Vasconcelos Cavalcante

January 5, 2011

1 Introdução

1.1 Objetivo

Este trabalho tem como objetivo comparar a criação de uma aplicação utilizando duas tecnologias, *Java* com o *Spring Framework* e *Hibernate*, e o *ASP.NET MVC 5* e *Entity Framework* utilizando *C#*. é usado como exemplo, a criação de um projeto de uma aplicação web de três camadas (apresentação, serviços e persistência) que utiliza o padrão *Model View Controller* (*MVC*).

A abordagem utilizada consiste em dividir as tarefas necessárias para a criação da aplicação em cinco capítulos, e nesses capítulos serão demonstrados exemplos de código com as duas tecnologias. Depois de expostos os exemplos de como executar as tarefas, serão ressaltadas as vantagens, desvantagens e características distintas de cada tecnologia. As tarefas a serem executadas serão as seguintes:

- Preparação do ambiente de desenvolvimento.
- Criação e configuração de um novo projeto.
- Criação de *controllers*.
- Criação de *views*.
- Criação de classes de serviço.
- Configuração de injeção de dependências.
- Criação de entidades.
- Criação de repositórios.

1.2 Estado da arte

No momento de criação desse trabalho, a literatura conta com vários trabalhos sobre medição de desempenho de aplicações *web* e poucos comparando como se cria uma aplicação. E mesmos trabalhos comparando o esforço do desenvolvimento de aplicações utilizando tecnologias diferentes, utilizam versões antigas de tais tecnologias.

Atul Mishra em seu trabalho *"Critical Comparison Of PHP And ASP.NET For Web Development"*, por exemplo, compara a criação de um site utilizando uma versão antiga do *ASP.NET* e utilizando *PHP*. Nesse trabalho não são expostos exemplos de código, apenas as conclusões tiradas pelo autor. Mostafa Pordel e Faranhaz Yekeh, autores do artigo " *"JSF vs ASP.NET, what are their limits?"* também comparam o desenvolvimento de aplicações *web* utilizando as tecnologias *Java* e *.NET*. Mas também utilizam versões antigas de ambas as tecnologias e mostram poucos exemplos de código.

Tiago Bencardino, em seu trabalho de conclusão do curso de engenharia de teleinformática de titulo " *iQuizzer*: Integrando aplicações web e dispositivos móveis em um ambiente para criação e execução de *quizzes*", comparou o desenvolvimento de uma aplicação para as plataformas moveis *Android* e *iOS*. Bencardino utilizou uma quantidade maior de exemplos de código, comparando o desenvolvimento das mesmas tarefas em ambas as plataformas alvos de seu trabalho.

1.3 Resumo dos resultados alcançados

Ambas as tecnologias possuem seus pontos fortes e fracos. De modo geral, o uso de *Java* com o *Spring Framework* proporciona melhor controle do projeto ao desenvolvedor e a liberdade de utilizar uma gama maior de bibliotecas para sua configuração. Mas essa liberdade tem um custo, o desenvolvedor de modo geral, precisa escrever mais código de configuração e pesquisar como os componentes escolhidos interagem para que o projeto seja bem sucedido.

A tecnologia *ASP.NET MVC* é mais fechada (apesar de ter seu código ser aberto), existem muitas convenções que o desenvolvedor deve obedecer e não existe tanta liberdade para configuração, tudo é mais padronizado. A vantagem disso é que o desenvolvedor se preocupa menos em como vai configurar sua aplicação e ele pode usar esse tempo para escrever código que realmente agrega valor ao produto.

De modo geral, se o desenvolvedor dispuser de tempo e precisar de um controle maior sobre o projeto, o uso da tecnologia *Java* é aconselhável. Por outro lado, se o projeto tiver um prazo reduzido ou o desenvolvedor preferir seguir convenções e não quiser se preocupar em como vai ter que configurar o projeto, a tecnologia *.NET* pode ser uma melhor opção.

1.4 Publico alvo

Esse trabalho pode beneficiar gerentes de projetos e lideres técnicos à escolher qual das duas tecnologias utilizar para iniciar um novo projeto. Estudantes interessados em começar a desenvolver para a *web* também podem se beneficiar desse trabalho e aprender o básico sobre as duas tecnologias.

1.5 Metodologia de pesquisa

A maior fonte da informação para esse trabalho foram livros, em sua maioria as edições mais recentes. Além dos livros, foram utilizados como fonte de pesquisa as documentações online de ambas as tecnologias. Também foram utilizados como fonte de pesquisa, exemplos encontrados em sites de desenvolvedores na internet, sendo o mais utilizado o *stack overflow* americano.

2 Configuração de ambiente de desenvolvimento

Neste capitulo é demonstrado o preparo do ambiente de desenvolvimento em um computador rodando o sistema operacional *Windows 8.1* de 64 *bits*. Primeiramente, instala-se um sistema gerenciador de banco de dados para trabalhar tanto com a plataforma *Java/Spring MVC* quanto com a plataforma *ASP.NET MVC* 5 . O banco de dados usado é o *MySQL Community Server* versão 5.6.22 (a versão mais atual até o momento de criação desta monografia). A *Integrated Development Environment* (*IDE*) utilizada para escrever código em *Java*, é o *Eclipse Luna*. O *Gradle* é usado como *build tool* e o *Apache Tomcat* como *container Java Enterprise Edition* (*JEE*). Para desenvolver em *C#*, é usado o *Visual Studio 2013 Community*. Considerando que o leitor já possui entendimentos sobre informática necessários para instalar programas no *Windows*, as instruções de instalação serão sucintas.

2.1 Instalação do MySQL

O download do instalador do *MySQL Community* foi feito no seguinte endereço `https://dev.mysql.com/downloads/windows/installer/5.6.html`. O instalador está disponível duas versões, *web installer* e *off-line installer*. O *web installer* é um arquivo pequeno que quando executado irá baixar os arquivos do *MySQL* para a máquina, o *off-line installer* é maior e vem com todos os arquivos necessários para a instalação do *MySQL*. Qualquer que seja o método de instalação escolhido, eles terão as mesmas opções.

Executando o instalador, é escolhida a opção *Custom* e na árvore de opções que aparecerá na tela a seguir, são escolhidos o *MySQL Server*, o *MySQL Workbench*, o *Connector/J* (para *Java*) e o *Connector/NET* (para *.NET*), como mostrado na Figura 1. Pode acontecer do instalador pedir para instalar o *Microsoft Visual C++ 2013* como dependência do *MySQL Workbench*, se isso acontecer, o próprio instalador proverá um botão para instalar essa dependência.

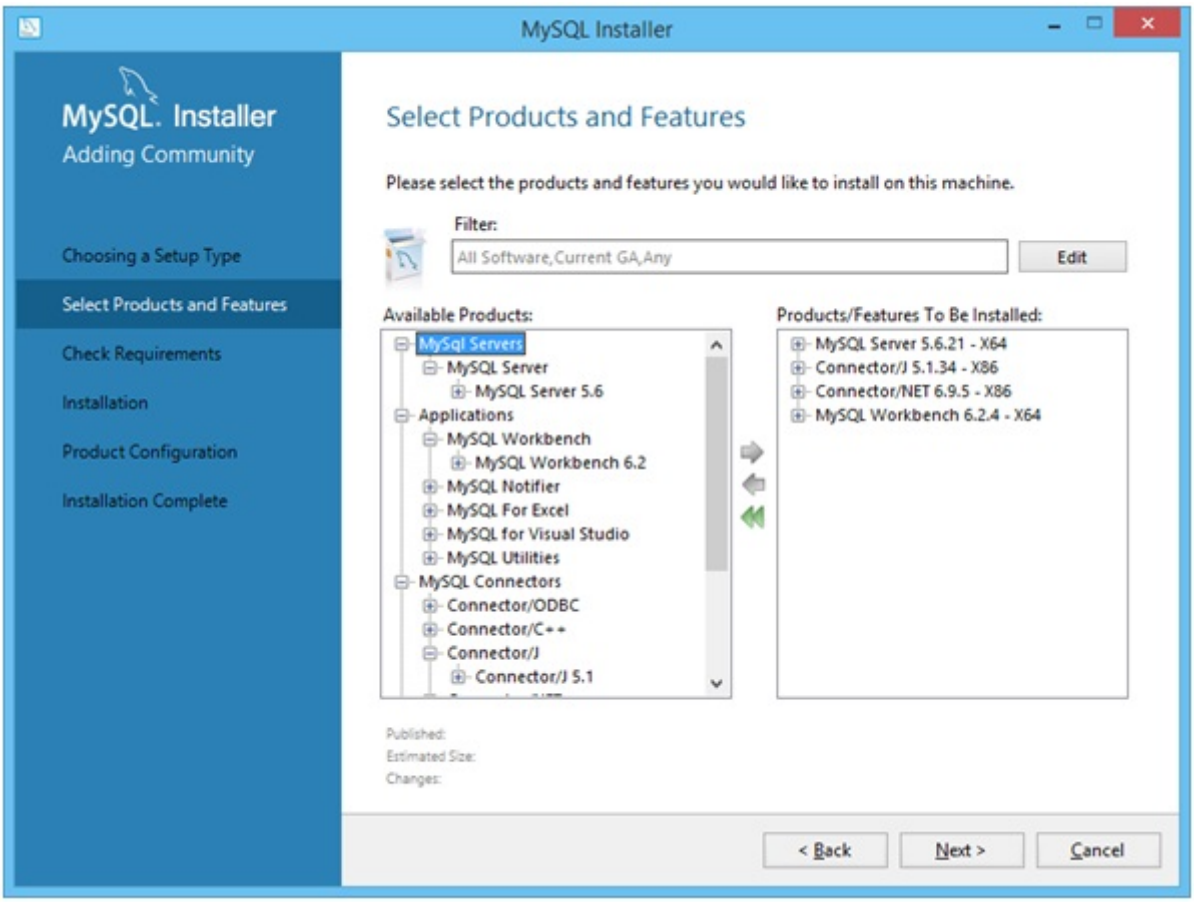

Figura 1: Opções de instalação do *MySQL*

Concluída a instalação, é hora de configurar o serviço do *MySQL*. Deixa-se selecionado o tipo de configuração como *Development Machine* e as configurações de rede padrão (protocolo *TCP/IP*, porta 3306). Quando for necessária a senha do usuário *Root*, é usada “1234”, é uma senha fraca que não se recomenda usar em ambiente de produção, mas serve para propósito de exemplo. Finaliza-se a configuração deixando marcados os restantes das opções de configuração como padrão do instalador.

Com o objetivo de testar o sucesso da instalação, o desenvolvedor pode executar o *MySQL Workbench*, como ilustrado na Figura 2, e tentar se conectar à instancia do MySQL.

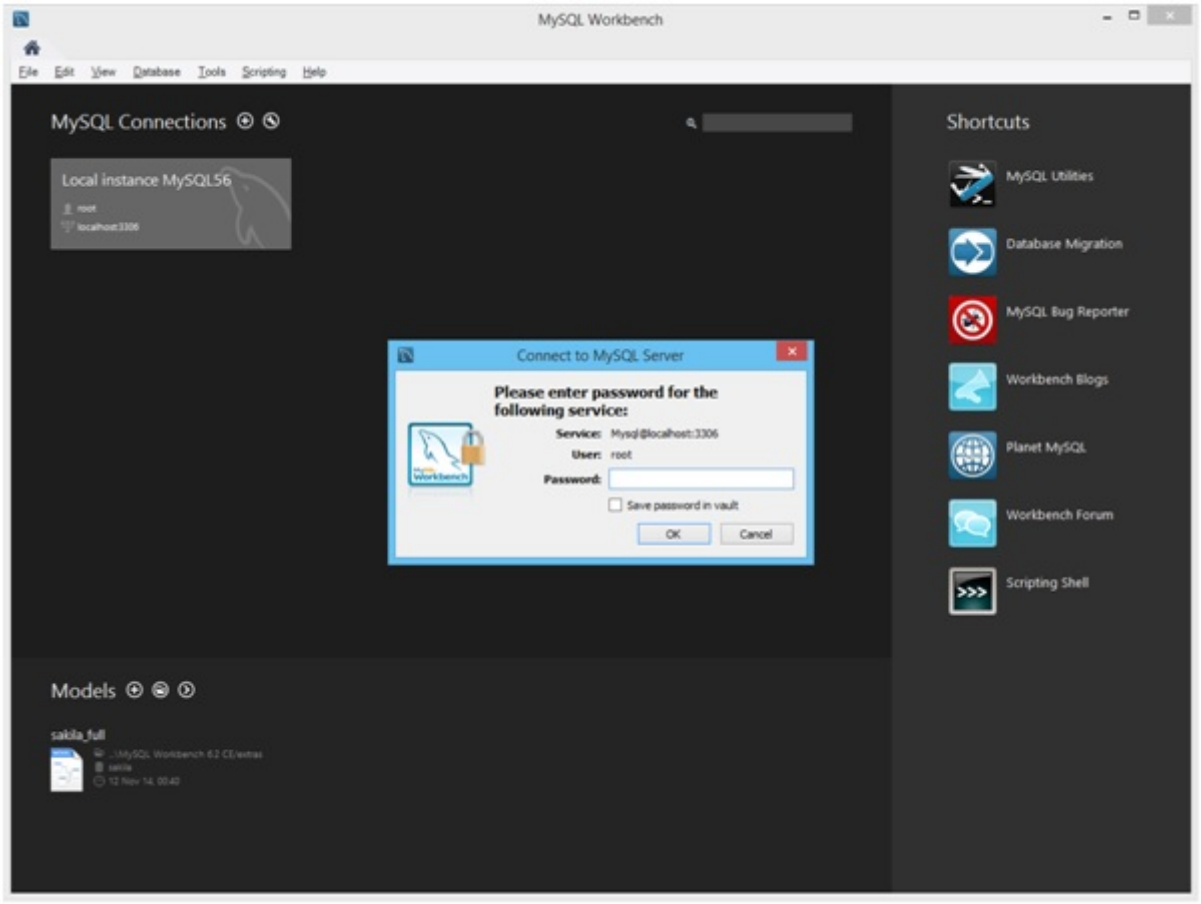

Figura 2: O *MySQL Workbench*

Para mais informações sobre o *MySQL*, visite a página oficial do projeto, `https://www.mysql.com/`.

2.2 Preparando o ambiente *Java*

Para desenvolver em *Java*, é utilizado o *Eclipse Luna* e o *Java Development Kit 8* (*JDK 8*). é usado o *Gradle* como *build tool* através de um *plugin* do *Eclipse* e o servidor *web* utilizado é o *Apache Tomcat*.

2.2.1 Instalando JDK 8

O instalador do *JDK 8* pode ser adquirido no endereço `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`. Existem diversas versões para diversos sistemas operacionais, é usado nesse trabalho a versão para *Windows* de 64 *bits*.

Para fazer a instalação do *JDK*, é executado o arquivo de instalação seguindo as suas instruções. A única configuração possível durante a instalação é a mudança da sua pasta de destino, mas é mantido o diretório padrão como mostrado na Figura 3.

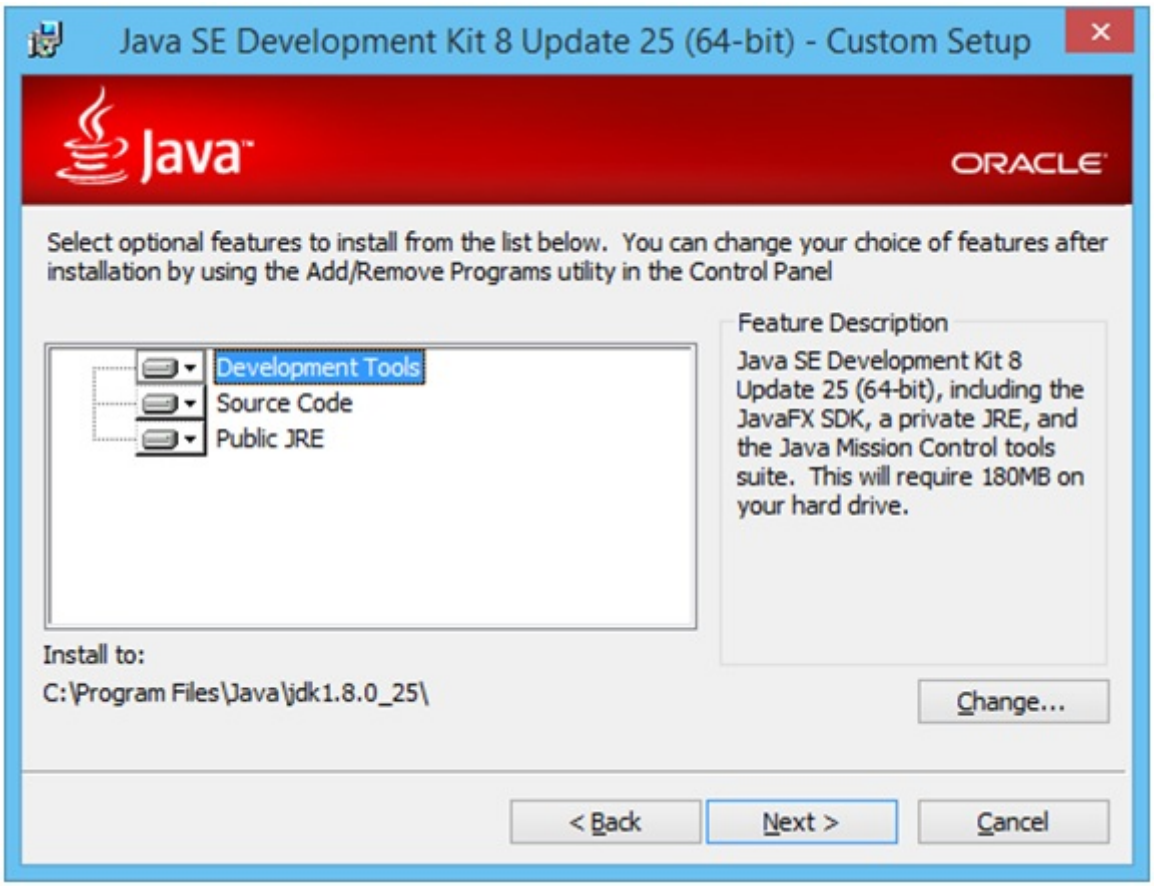

Figura 3: O instalador do *JDK 8*

Terminada a instalação, é necessário configurar a variável *PATH* para que o sistema encontre os arquivos do *Java*. Essas opções de configuração estão no painel de controle do *Windows*, no caminho Sistema/Configurações avançadas do sistema/Variáveis de ambiente. Na janela de variáveis do sistema, é editada a variável *PATH*. Se adiciona o caminho onde o *JDK* foi instalado acrescido da pasta bin (C:\Program Files\Java\jdk1.8.0_25\bin) como mostrado na Figura 4.

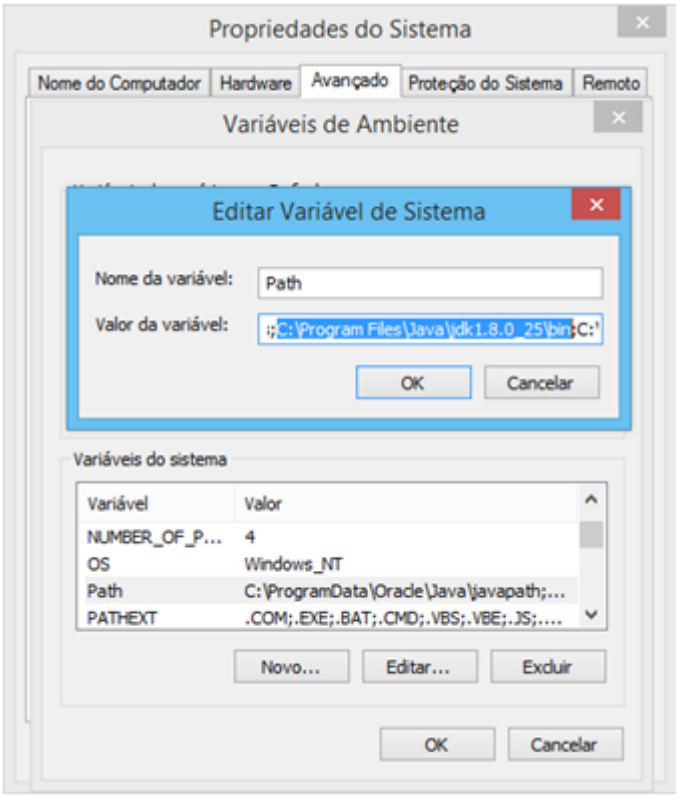

Figura 4: Configurando a variável *PATH*

Para testar se tudo foi instalado corretamente, abre-se uma janela do *prompt* de comando e digita-se o comando *"java –version"* (sem aspas). Se não existirem problemas, é exibida na tela o número da versão do *JDK* instalado. Caso isso não aconteça, é aconselhável desinstalar o *JDK* e repetir o processo de instalação.

2.2.2 Instalando o *Eclipse Luna*

O *Eclipse Luna* para Desenvolvedores *Java EE* é encontrado no endereço `https://www.eclipse.org/downloads/`. Terminando o *download*, a pasta *"eclipse"* pode ser descompactada para qualquer diretório do computador. Coloca-se um atalho na sua área de trabalho para o executável do *Eclipse* (eclipse.exe) para facilitar o acesso.

Na primeira vez que o *Eclipse* é executado é exibida uma janela para configurar o *Workspace* padrão (uma pasta onde serão guardados projetos e configurações), como está ilustrado na Figura 5.

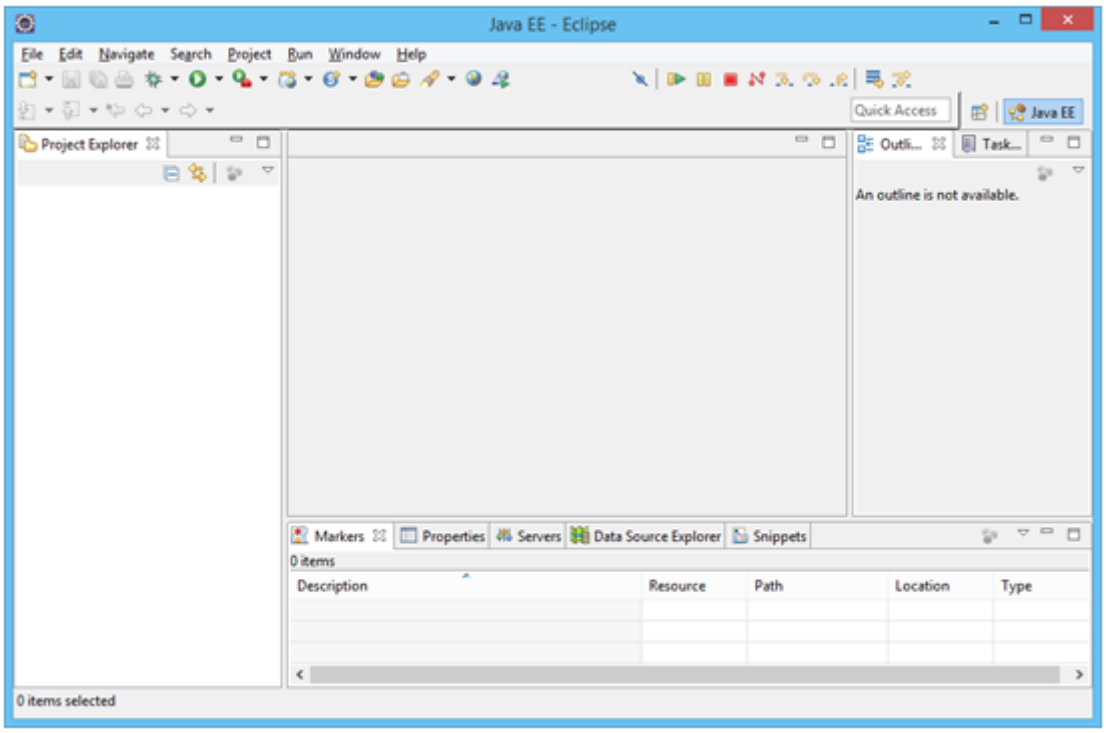

Figura 5: *Eclipse* recém instalado

2.2.3 Instalando o *plugin* do *Gradle*

O *Gradle* é a *build tool* que é utilizada nos exemplos do projeto *Java/Spring MVC* . Ele faz o mesmo trabalho que o *ANT* associado ao *Ivy* ou o *Maven* fazem, mas ele é considerado por alguns autores como o mais moderno em se tratando de *build tools*, pois seus *scripts* são escritos em *Groovy* em vez de *XML* e ele permite configurações que o *Maven* não permite. O *Gradle* está presente em todo ciclo de vida do software (ele gera artefatos, executa teste unitários, resolve dependências e executa integração continua), mas nesse trabalho é usada apenas uma pequena parte do que ele pode oferecer. Para mais informações sobre o *Gradle* acesse `https://www.gradle.org`.

No *Eclipse Marketplace* (repositório de *plugins* do *Eclipse*), faz-se uma pesquisa por *"Gradle"* na barra de buscas, entre os resultados está o *Gradle IDE Pack*. Esse *plugin* é usado nos exemplos desse trabalho. O *Eclipse* pede confirmação para instalação de todos os pacotes necessários, todos são selecionados e instalados. Aceita-se os termos de uso do *Gradle* e ao aparecer uma janela de alerta confirmando a instalação, clica-se em OK. Quando a instalação terminar, o *Eclipse* é reiniciado.

Para verificar se o *plugin* foi instalado com sucesso, a pasta *Gradle* deve aparecer na arvore de tipos de projetos, no menu de novos projetos, como pode ser observado na Figura 6.

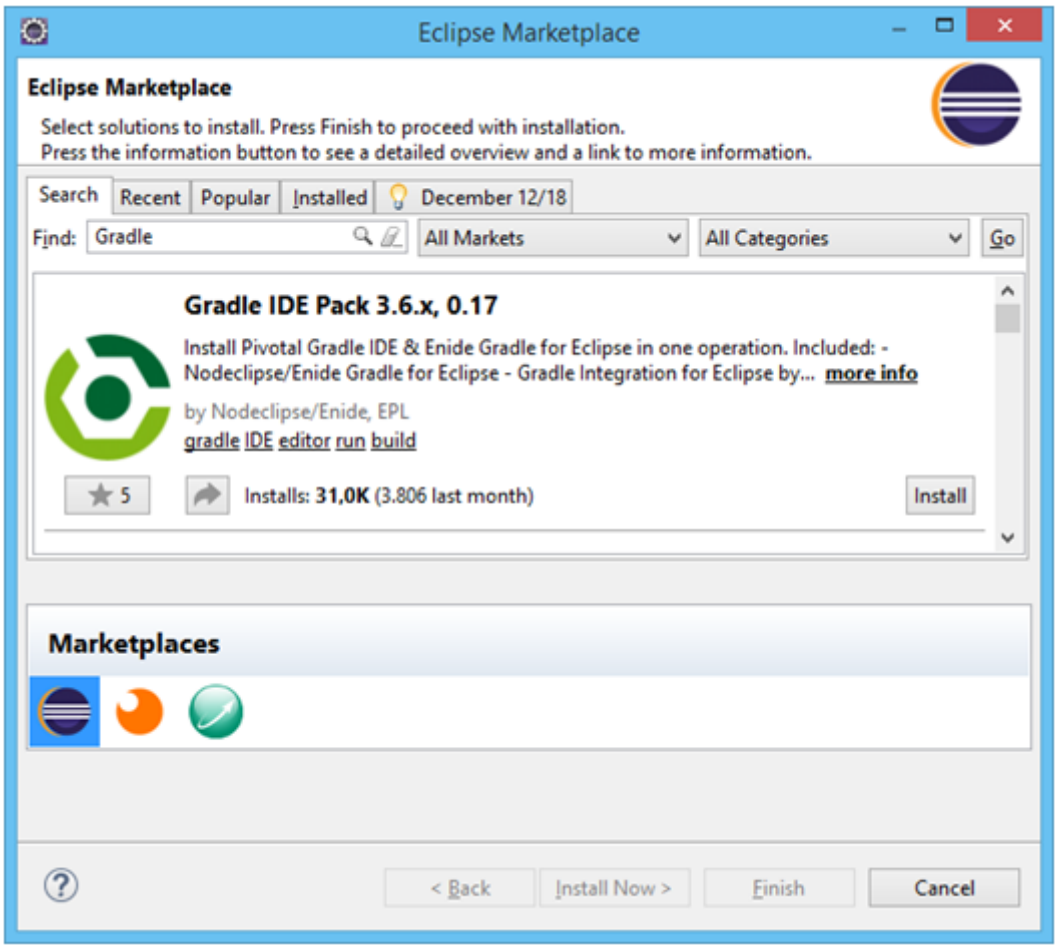

Figura 6: Instalando o *plugin* do *Gradle*

2.2.4 Instalando o *Apache Tomcat 8* como servidor de desenvolvimento do *Eclipse*

O *Java Enterprise Edition* é um conjunto de especificações que precisam ser implementadas por um *container* (um servidor de aplicação ou servidor *web*) que irá executar efetivamente a aplicação. Existem diversos *containers* disponíveis no mercado, sendo o *GlassFish* o próprio *container* da *Oracle*. Nesse trabalho é usado o *Apache Tomcat*, pois o *Eclipse* tem integração nativa com ele. O *Tomcat* pode ser gerenciado pela aba de servidores do *Eclipse*.

O instalador do *Tomcat 8* pode ser encontrado no endereço `https://tomcat.apache.org/download-80.cgi`. Para os exemplos, usa-se a distribuição para *Windows* de 64 bits no formato *zip*. A pasta *apache-tomcat-8.0.15* pode ser descompactada para qualquer diretório no disco rígido (como exemplo é usada a raiz do disco C:).

Na aba *Servers* do Eclipse existe um *link* auto descritivo para adicionar um novo servidor. Clicando no *link* e expandindo pasta *Apache*, é escolhido o *Tomcat 8* na árvore de opções, o que pode ser observado na Figura 7. Na janela seguinte, em *Tomcat installation directory*, o botão *browse...* é usado para escolher o caminho de instalação do *Tomcat* (C:\apache-tomcat-8.0.15). Clicando no botão *Finish*, o *Tomcat* está pronto para uso com o *Eclipse*.

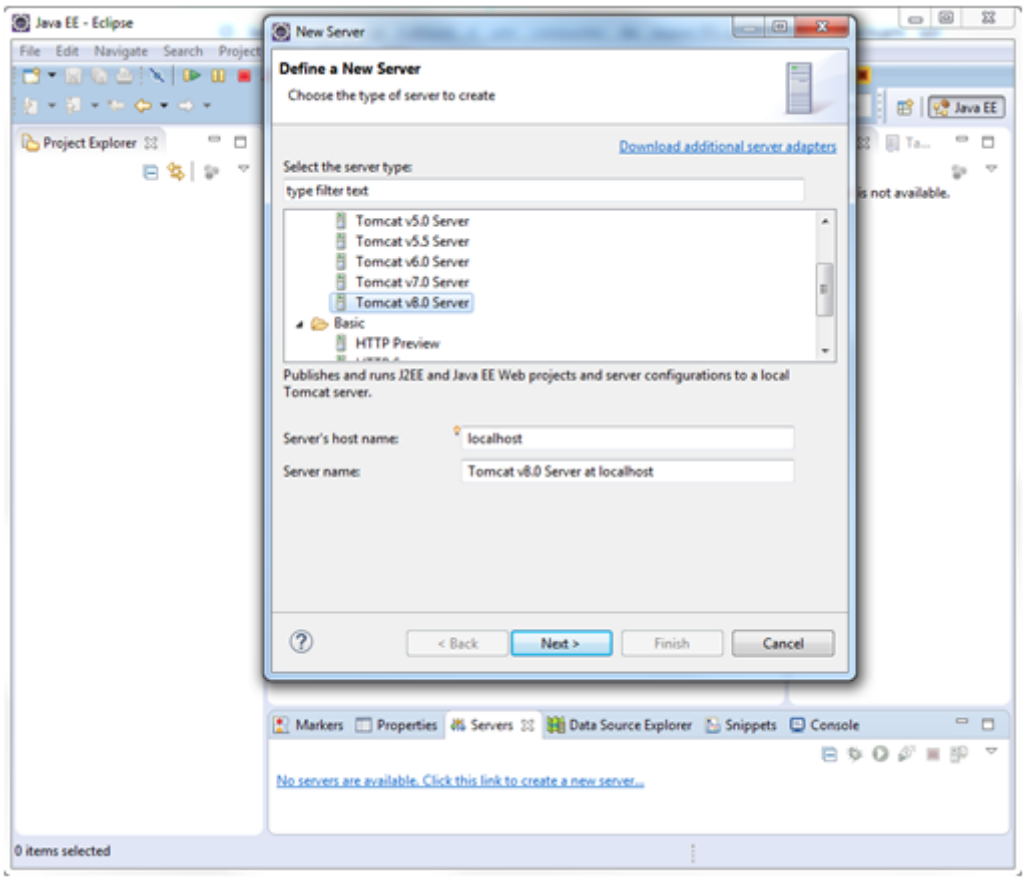

Figura 7: Janela pra adicionar servidores no *Eclipse*

2.3 Instalando o *Visual Studio Community 2013*

O instalador do *Visual Studio Community 2013* pode ser encontrado no endereço `http://www.visualstudio.com/en-us/news/vs2013-community-vs.aspx`. Na Figura 8 temos uma ilustração do instalador em questão. Esse é um instalador *online*, ele baixa os arquivos do *Visual Studio* e opcionais selecionados à medida que a instalação for progredindo. Uma imagem do DVD de instalação *offline* também está disponível na sessão de *downloads* do site `http://www.visualstudio.com`.

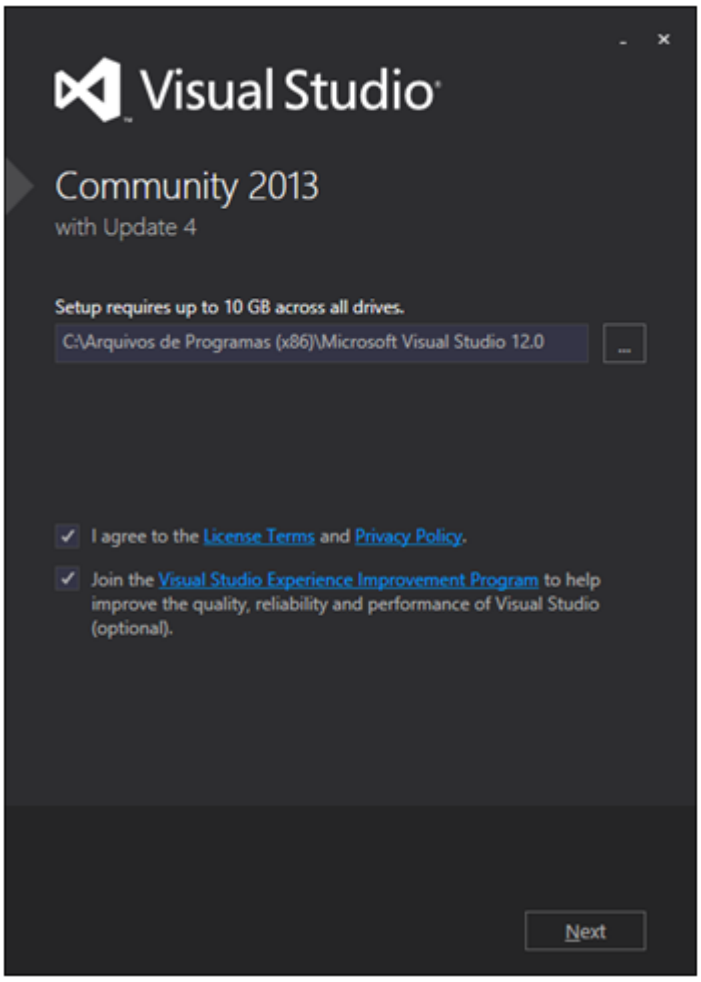

Figura 8: Instalador do Visual Studio Community 2013

Clicando no botão *Next*, a próxima tela que o instalador exibe uma lista de componentes opcionais como o *kit* de desenvolvimento do *Windows Phone 8* e do *Silverlight*. Desses componentes opcionais, é aconselhável instalar pelo menos o *Microsoft Web Developer Tools* para facilitar o desenvolvimento de aplicações *web*.

Após a instalação, o desenvolvedor pode utilizar sua conta da *Microsoft* como perfil no *Visual Studio* e publicar suas aplicações no *Microsoft Azure*, porém isso é opcional. Quando se executa o *Visual Studio* pela primeira vez, ilustrado na Figura 9, o desenvolvedor pode escolher as opções de desenvolvimento e um esquema de cores que irá usar. Como exemplo, é escolhida a opção de desenvolvimento *Web Development*.

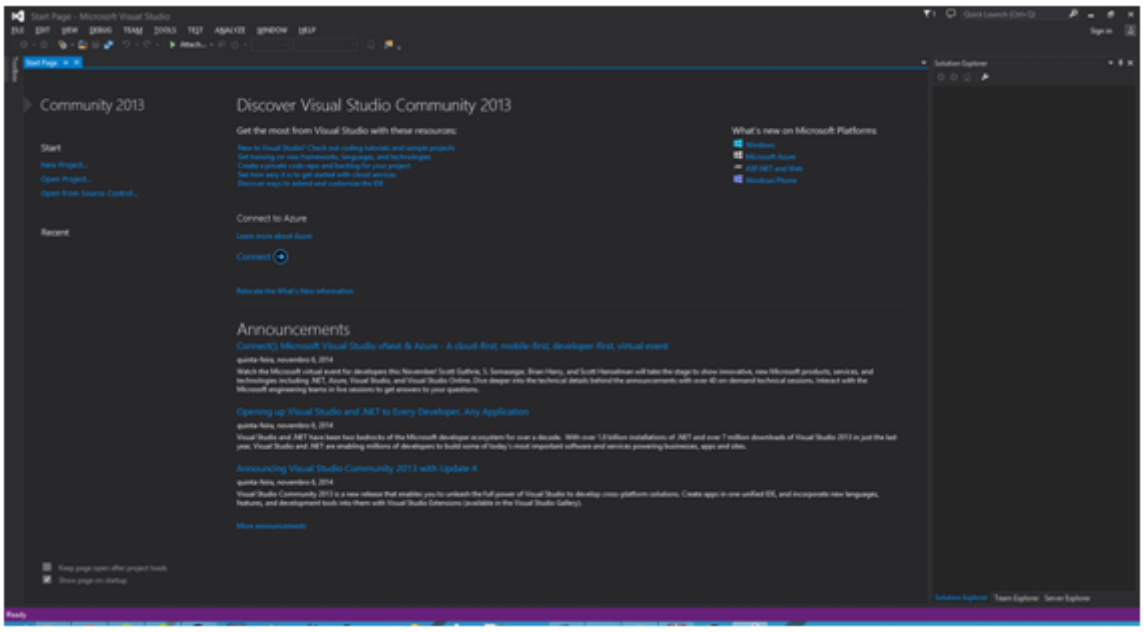

Figura 9: Tela inicial do *Visual Studio Community 2013*

O *Visual Studio* possui sua própria ferramenta de geração de *builds* (*MsBuild*), gerenciador de pacotes para obter bibliotecas de terceiros (*Nuget*) e um servidor de desenvolvimento minimalista baseado no *Internet Information Services* (servidor *web* do *Windows Server*) para executar e depurar aplicações *web*.

2.4 Conclusão

A preparação de um ambiente de desenvolvimento *Java/Spring MVC* requer mais passos, dentre eles instalar o kit de desenvolvimento do *Java*, uma *IDE* (*Eclipse*), um *container Java Enterprise Edition* (*Tomcat*) e uma *build tool* (*Gradle*), enquanto todo o *software* necessário para se desenvolver com *.NET* é adquirido em um único instalador.

A vantagem do ambiente *Java* é que ele fornece ao desenvolvedor mais opções de como configurar seu ambiente, além disso o tamanho em *megabytes* do *software* necessário é consideravelmente menor do que o ambiente *.NET*. O desenvolvedor tem a liberdade de escolher outras *build tools* disponíveis no mercado (Ex: *Ant*, *Maven*), outras *IDEs* (Ex: *Netbeans*) e outros servidores *Java EE* (Ex: *Jetty*, *Glassfish*). O lado negativo dessa liberdade é que, com essa variedade de opções, o desenvolvedor tem que pesquisar mais sobre cada solução até decidir como vai montar seu ambiente de desenvolvimento. Então depois de escolher quais produtos irá utilizar, pode ser que precise de mais algum tempo estudando como eles interagem.

A vantagem do ambiente *.NET* está na facilidade de obter todo o *software* necessário para o desenvolvimento em um único pacote, o *.NET Framework*, *Visual Studio Community 2013* e demais ferramentas. A desvantagem é que esse pacote pode conter componentes que o desenvolvedor não precisa ou deseja, baixando arquivos desnecessários e tomando espaço em disco.

No próximo capitulo é demonstrado como criar um projeto de uma aplicação *web* nas duas plataformas.

3 Criando projetos web no padrão *MVC*

Nesse capitulo é mostrado como criar um novo projeto para uma aplicação *web* que utiliza o padrão *MVC* nas plataformas *Java/Spring MVC* e *ASP.NET MVC* 5 . Para *Java* serão usadas as bibliotecas *Spring Framework*, que cuidará da arquitetura *MVC* e injeção de dependências, e o *Hibernate*, que cuidará da persistência e acesso a dados. Na plataforma *.NET* é utilizado o *ASP.NET MVC 5*, que cuida de toda estrutura de uma aplicação web *MVC*, o *Entity Framework 6* para acesso a dados e o *Ninject* para injeção de dependências.

3.1 Criando um projeto do *Gradle* no *Eclipse*

O modelo de projeto usado nos exemplos desse trabalho é o *Gradle Project*. Esse modelo de projeto está localizado na pasta *Gradle* na arvore de novos projetos do *Eclipse*. Na Figura 10 pode-se observar a localização do modelo e a criação do novo projeto.

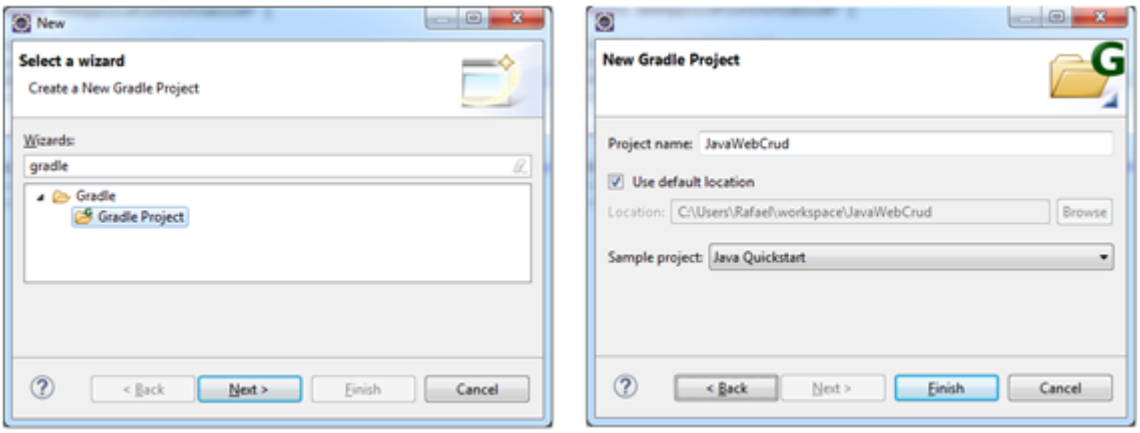

Figura 10: Criando um projeto do *Gradle*

É criado um projeto com a estrutura mostrada na Figura 11.

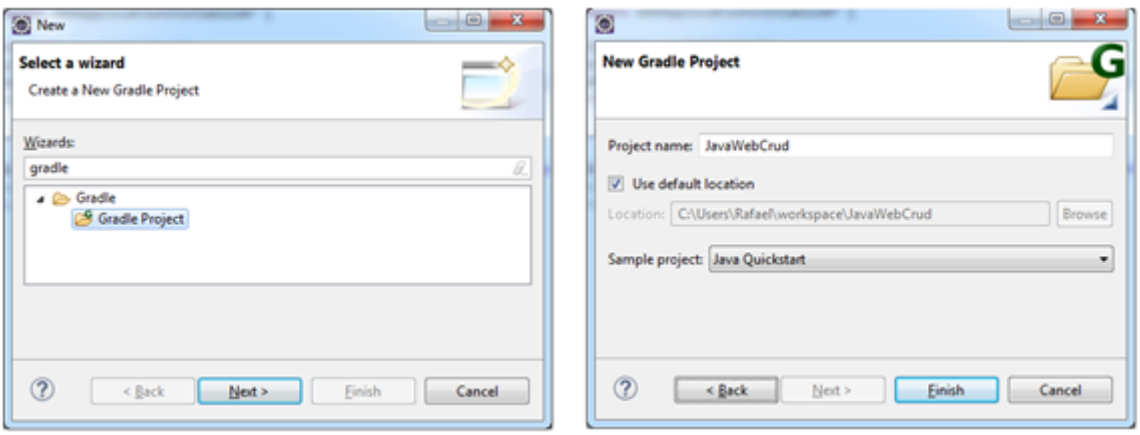

Figura 11: Estrutura inicial de um projeto do *Gradle*

As pastas *"src/main/java"* e *"src/main/resources"* devem armazenar, respectivamente, código fonte *Java* e recursos utilizados na aplicação. As pastas *"src/test/java"* e *"src/test/resources"* são utilizadas para testes unitários. As pastas de testes não serão utilizadas nos exemplos e são removidas. A pasta *build* é para onde irão todos os artefatos gerados pelo projeto.

O arquivo *build.gradle*, exibido no quadro 1, é o arquivo de definição de projeto do *Gradle*. Nele podem ser criados *scripts* para controlar a construção de artefatos, adquirir bibliotecas de terceiros, configurar testes unitários e realizar várias outras tarefas. Os *scripts* do *Gradle* são escritos em *Groovy*, outra linguagem compatível com a máquina virtual *Java*.

```
apply plugin: 'java'
apply plugin: 'eclipse'

sourceCompatibility = 1.5
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart', '
            Implementation-Version': version
    }
}

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'commons-collections', name: 'commons-collections',
        version: '3.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}

test {
    systemProperties 'property': 'value'
}

uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

Código 1: O arquivo *build.gradle*

A estrutura do projeto e o arquivo *build.gradle* gerados devem ser modificados para servir à uma aplicação *web*. é adicionada uma nova pasta, chamada de *WebContent*, para armazenar conteúdo especifico para *web* (páginas *jsp/html*, arquivos *javascript* e *css*). Dentro dela deve ser criada uma pasta chamada *WEB-INF* para armazenar páginas *jsp*. Por padrão, usuários de sistemas *web* *Java Enterprise Edition* não

possuem acesso direto à pasta *WEB-INF*, então é um lugar seguro para armazenar páginas. Na Figura 12 pode ser observada a estrutura do projeto com a pasta *WEB-INF*.

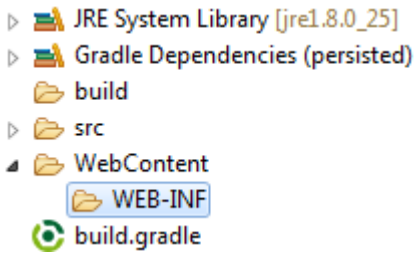

Figura 12: Estrutura do projeto com a pasta *WebContent*

Em seguida, é necessário modificar o arquivo *build.gradle*. O *Gradle* dispõe de vários *plugins* que facilitam seu uso em diversas situações e servem a propósitos específicos. Para gerar arquivos *.war* (artefatos de aplicações *web*) e informar ao *Gradle* que a pasta *WebContent* armazenará o conteúdo especifico para *web*, é utilizado o *plugin* chamado *war*. Para fazer com que o *Eclipse* veja o projeto como um projeto para *web* que possa ser enviado para o *Tomcat*, é utilizado o *plugin eclipse-wtp*. O início do arquivo *build.gradle* deverá ficar como no exemplo do quadro 2.

```
apply plugin: 'java'
apply plugin: 'war'
apply plugin: 'eclipse-wtp'

project.webAppDirName = 'WebContent'
...
```

Código 2: Arquivo *build.gradle* com novos *plugins*

Com tudo devidamente configurado, é necessário atualizar o tipo de projeto para que o *Eclipse* o veja como um projeto de uma aplicação *web*. Pressionando Ctrl + Alt + Shift + R o *Eclipse* abrirá uma caixa de texto onde poderão ser executadas tarefas do *Gradle*. Uma dessas tarefas é o comando *"eclipse"* que gera arquivos adicionais para que o projeto possa ser detectado como uma aplicação *web*. O comando é executado como na Figura 13 e o *Eclipse* agora poderá publicar a aplicação no *Tomcat*.

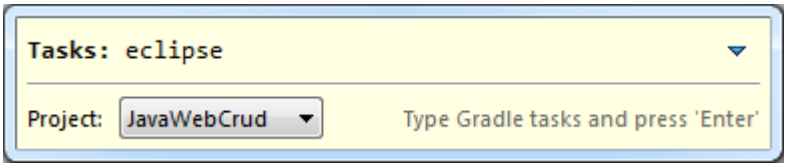

Figura 13: *Gradle Task Quick Launcher*

Após executar esse comando, a estrutura do projeto mudará um pouco. As pastas destinadas a conter código *Java* e bibliotecas usadas no projeto serão movidas para uma pasta chamada *Java Resources*.

3.1.1 Adicionando o projeto ao *Tomcat*

Para adicionar uma aplicação *web* ao servidor *Tomcat* configurado no *Eclipse*, na aba *"servers"* dá-se um duplo clique no servidor para abrir suas configurações, e na aba *"modules"* clica-se no botão *"Add Web Module"*. Escolhido o projeto *web* que se deseja adicionar, o caminho da aplicação pode ser configurado. A Figura 14 ilustra esse procedimento. Com a configuração padrão do *Tomcat*, a aplicação estará disponível no endereço com o seguinte formato: `http://<endereço-ip>:8080/<caminho-da-aplicaç˜ao>`. Esse endereço é referenciado no restante do trabalho como raiz da aplicação.

Iniciando o *Tomcat*, clicando nos botões *"Start the server"* ou *"Start the server in debug mode"*, e acessando o endereço da aplicação, é mostrado uma página de erro 404. Isso acontece porque ainda não existem as configurações do *servlet* padrão, do *Spring MVC* e ainda não foi criado nenhum *controller* e nenhuma *view*. Nesse capítulo ainda é demonstrado como configurar o *servlet* padrão e o *Spring MVC*, no capitulo 3 é demonstrado como se criam *controllers* e *views*.

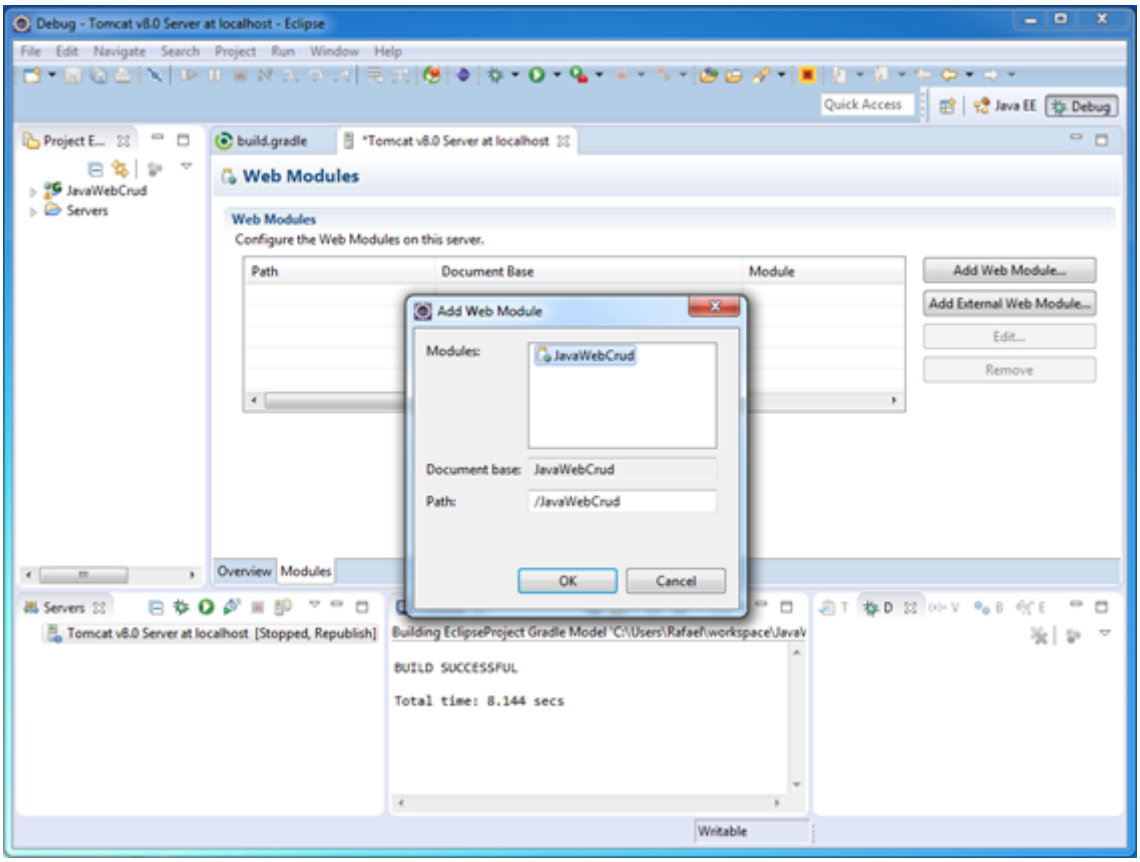

Figura 14: Adicionando um projeto web ao *Tomcat*

3.1.2 Adicionando dependências ao projeto

Abaixo estão listadas as bibliotecas que são usadas no projeto *Java/Spring MVC* . O *Gradle* pode adquirir essas bibliotecas e suas dependências do repositório central

do *Maven*.

- *Java Servlet API 3.1.0* - Biblioteca base para programar em *Java* para *web* usando *servlets*.
- *Spring Web MVC 4.1.4 RELEASE* - Biblioteca para criar um projeto *MVC* com *Spring*.
- *Spring Transaction 4.1.4 RELEASE* - Gerenciador de transações com o banco de dados.
- *Spring Object/Relational Mapping 4.1.4 RELEASE* - Gerenciador de entidades e mapeamento objeto/relacional.
- *Jackson Databind 2.5.1* - Serializa e de deserializa objetos *Java* no formato *JSON*.
- *MySQL Java Connector 5.1.34* - Comunicação com o banco de dados *MySQL*.
- *Hibernate JPA Support 4.3.8 Final* - Adaptador do *Hibernate* para padrões *Java Persistence API*.
- *Hibernate Validator Engine 5.1.3 Final* - Biblioteca para validar dados de entidades.
- *Hibernate c3p0 Integration 4.3.8* - Gerenciador de conexões com o banco de dados.

Os detalhes sobre as funcionalidades das bibliotecas de comunicação com o banco de dados e mapeamento objeto/relacional são abordados no capítulo 6.

Para fazer com que o *Gradle* adquira as bibliotecas necessárias e suas dependências, a seção *"dependencies"* do arquivo *build.gradle* é alterada como mostra o quadro 3.

```
dependencies {
        compile 'javax.servlet:javax.servlet-api:3.1.0'
        compile 'org.springframework:spring-webmvc:4.1.4.RELEASE'
        compile 'org.springframework:spring-tx:4.1.4.RELEASE'
        compile 'org.springframework:spring-orm:4.1.4.RELEASE'
        compile 'com.fasterxml.jackson.core:jackson-databind:2.5.1'
        compile 'mysql:mysql-connector-java:5.1.34'
        compile 'org.hibernate:hibernate-entitymanager:4.3.8.Final'
        compile 'org.hibernate:hibernate-validator:5.1.3.Final'
        compile 'org.hibernate:hibernate-c3p0:4.3.8.Final'
}
```

Código 3: Adicionando dependências ao projeto

Com o arquivo *build.gradle* alterado e salvo, o comando para baixar as dependências para o projeto está localizado no menu de contexto do projeto (clicando com o botão direito do *mouse* sobre ele) no caminho *Gradle/Refresh Dependencies*, que

pode ser observado na Figura 15. O *Gradle* irá adquirir todas as bibliotecas especificadas no arquivo *build.gradle* e suas dependências automaticamente. O *Spring MVC*, por exemplo, depende do *Spring Core* para funcionar e o *Hibernate JPA Support* precisa do *Hibernate Core*. Todas as dependências do projeto ficam visíveis na seção *Libraries/Gradle Dependencies*.

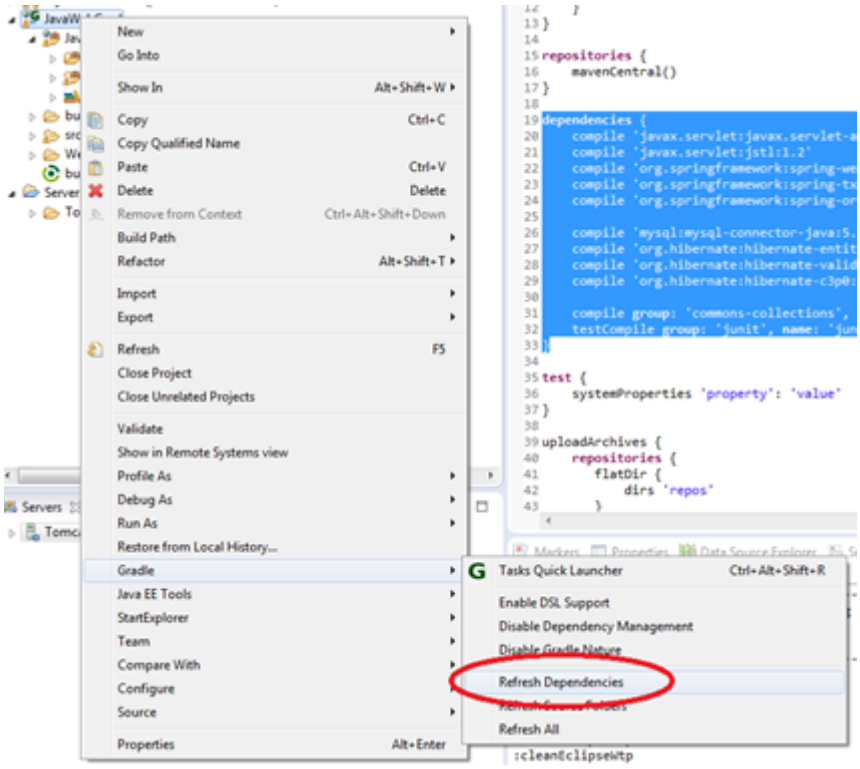

Figura 15: Atualizando dependências de um projeto *Gradle*

3.1.3 Configurando a aplicação *web*

Existem duas maneiras de se configurar uma aplicação *Java*, usando arquivos *XML* ou escrevendo a configuração em *Java*. Nos exemplos dessa seção, é usada a segunda opção.

Os arquivos de configuração são armazenados no pacote *br.uece.webCrud.config*. Dentro desse pacote são criadas duas classes, *AppInitializer* e *SpringMvcConfig*, como mostra a Figura 16.

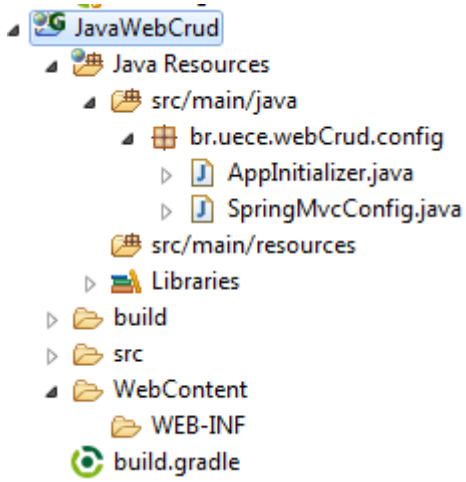

Figura 16: Estrutura do projeto *Java/Spring MVC* com pacote para arquivos de configuração

A primeira configuração que deve ser feita é usar a classe *DispatcherServlet* como o *servlet* padrão e configurar o contexto da aplicação (configurações especificas do *Spring Framework*). *Servlets* são classes *Java* que processam requisições para aplicação, elas podem servir páginas para o usuário, retornar objetos *JSON*, redirecionar requisições para outros *servlets*, ETC. Para mais informações sobre *servlets* recomenda-se o livro *Murach's Java Servlets and JSP*.

A classe *AppInitializer* herda da interface *WebApplicationInitializer*. Para inicializar a aplicação, o *Spring Framework* irá procura classes que herdam dessa *interface* dentro dos pacotes. O conteúdo da classe deve estar como no quadro 4.

A interface *WebApplicationInitializer* possui a assinatura de apenas um método, *onStartup*. Na implementação desse método é escrito código para configurar tanto o contexto da aplicação quando o *servlet* primário.

Primeiro é criado um objeto do tipo *AnnotationConfigWebApplicationContext* e é usado o seu método *register* para adicionar a classe *SpringMvcConfig* como uma classe de configuração do *Spring*. A classe *AnnotationConfigWebApplicationContext* habilita o uso de anotações *Java* para configurar demais classes, o conteúdo da classe *SpringMvcConfig* é exposto mais adiante.

É criado então um *servlet* do tipo *DispatcherServlet* que recebe o contexto como parâmetro e é usada a classe *ServletRegistration* para registrar o *servlet* à aplicação. O método *addMapping* recebe como parâmetro o caractere "/", isso quer dizer que esse *servlet* é usado para todas a páginas *web* e demais caminhos dentro da aplicação. O método *setLoadOnStartup* recebe como parâmetro o valor 1, para configurar o *servlet* como o primeiro que o servidor deve usar.

```
package br.uece.webCrud.web.config;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;
import org.springframework.web.WebApplicationInitializer;
```

```
import org.springframework.web.context.support.
    AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

public class AppInitializer implements WebApplicationInitializer {
        public void onStartup(ServletContext servletContext) throws
            ServletException {
                AnnotationConfigWebApplicationContext context = new
                    AnnotationConfigWebApplicationContext();
                context.register(SpringMvcConfig.class);

                DispatcherServlet springServlet = new DispatcherServlet(
                    context);

                ServletRegistration.Dynamic springServletRegistration =
                    servletContext.addServlet("dispatcher", springServlet);
                springServletRegistration.addMapping("/");
                springServletRegistration.setLoadOnStartup(1);
        }
}
```

Código 4: Classe *AppInitializer*

Vários objetos que fazem parte da estrutura da aplicação (também conhecidos como *Spring Beans*) são configurados na classe *SpringMvcConfig*, como mostra o quadro 5. A anotação *@Bean* do *Spring Framework* decora métodos na classe *SpringMvcConfig* que retornam objetos que são usados no núcleo da aplicação e definem seu contexto.

A classe *SpringMvcConfig* começa decorada com as seguintes anotações:

- *@Configuration* - Define a classe como uma classe de configuração do *Spring Framework*.
- *@EnableWebMvc* - Habilita o *Spring MVC*.
- *@EnableTransactionManagement* - Habilita o controle automático de transações com o banco de dados pelo *Spring Framework*.
- *@ComponentScan* - Configura os nomes de pacotes onde o *Spring IoC Container* deve procurar classes decoradas a anotação *@Component* e suas especializações, como *@Repository*, *@Service* e *@Controller* (Mais detalhes sobre essas anotações em capítulos posteriores).

```
package br.uece.webCrud.web.config;
import java.beans.PropertyVetoException;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.http.MediaType;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.annotation.
    EnableTransactionManagement;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.
    ContentNegotiationConfigurer;
import org.springframework.web.servlet.config.annotation.
    DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.
    WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.ContentNegotiatingViewResolver;
import org.springframework.web.servlet.view.InternalResourceViewResolver;
import com.mchange.v2.c3p0.ComboPooledDataSource;

@Configuration
@EnableWebMvc
@EnableTransactionManagement
@ComponentScan({"br.uece.webCrud.controller", "br.uece.webCrud.service", "
    br.uece.webCrud.repository", "br.uece.webCrud.model"})
public class SpringMvcConfig extends WebMvcConfigurerAdapter {
        private final String DATABASE_DRIVER = "com.mysql.jdbc.Driver";
        private final String DATABASE_URL = "jdbc:mysql://localhost/
            web_crud_java";
        private final String DATABASE_USER = "root";
        private final String DATABASE_PASSWORD = "1234";
        private final String JPA_DATABASE_CONSTRUCTION = "create";
        private final String JPA_DATABASE_DIALECT = "org.hibernate.dialect.
            MySQL5Dialect";
        private final String JPA_DEFAULT_SCHEMA = "web_crud_java";

        @Override
        public void configureDefaultServletHandling(
            DefaultServletHandlerConfigurer configurer) {
                configurer.enable();
        }

        @Override
        public void configureContentNegotiation(
            ContentNegotiationConfigurer configurer)
        {
                configurer.mediaType("html", MediaType.TEXT_HTML).mediaType
                    ("json", MediaType.APPLICATION_JSON);
        };

        @Bean
        public InternalResourceViewResolver internalResourceViewResolver()
        {
                InternalResourceViewResolver irvr = new
                    InternalResourceViewResolver();
                irvr.setPrefix("/WEB-INF/");
                irvr.setSuffix(".jsp");

                return irvr;
        }
```

```
@Bean
public ContentNegotiatingViewResolver contentNegotiatingiewResolver
    ()
{
        List<ViewResolver> viewResolverList = new ArrayList<
            ViewResolver>();
        viewResolverList.add(internalResourceViewResolver());

        ContentNegotiatingViewResolver cnvr =  new
            ContentNegotiatingViewResolver();
        cnvr.setViewResolvers(viewResolverList);

        return cnvr;
}

//Database beans
@Bean
public ComboPooledDataSource mySqlDataSource() throws
    PropertyVetoException
{
        ComboPooledDataSource cpds = new ComboPooledDataSource();
        cpds.setDriverClass(DATABASE_DRIVER);
        cpds.setJdbcUrl(DATABASE_URL);
        cpds.setUser(DATABASE_USER);
        cpds.setPassword(DATABASE_PASSWORD);
        return cpds;
}

@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory
    () throws PropertyVetoException
{
        LocalContainerEntityManagerFactoryBean emf = new
            LocalContainerEntityManagerFactoryBean();
        HibernateJpaVendorAdapter hjva = new
            HibernateJpaVendorAdapter();
        emf.setJpaVendorAdapter(hjva);
        emf.setPackagesToScan("br.uece.webCrud.model");
        emf.setDataSource(mySqlDataSource());
        emf.setJpaProperties(this.jpaProperties());
        return emf;
}

@Bean
public JpaTransactionManager transactionManager() throws
    PropertyVetoException
{
        JpaTransactionManager jtm = new JpaTransactionManager();
        jtm.setEntityManagerFactory(entityManagerFactory().
            getObject());
        return jtm;
}

private Properties jpaProperties()
{
        Properties p = new Properties();
        p.setProperty("hibernate.show_sql", "true");
```

```
                p.setProperty("hibernate.format_sql", "true");
                p.setProperty("hibernate.hbm2ddl.auto",
                    JPA_DATABASE_CONSTRUCTION);
                p.setProperty("hibernate.default_schema",
                    JPA_DEFAULT_SCHEMA);
                p.setProperty("hibernate.dialect", JPA_DATABASE_DIALECT);
                return p;
        }
}
```

Código 5: Classe *SpringMvcConfig*

A classe começa definindo várias propriedades que são usadas para a criação da conexão com o banco de dados, como o *driver* a ser utilizado, localização, nome do banco, usuário e senha. Outras propriedades armazenam configurações do *JPA* (implementadas pelo *Hibernate*), como qual dialeto do *SQL* usar para gerar consultas no banco (dialeto do *MySql 5*), qual a ação executar na configuração *hibernate.hbm2ddl.auto* (*"create"* para criar o banco de dados e gerar tabelas a partir de classes decoradas com a anotação *@Entity*) e qual o esquema padrão a ser usado nas consultas (normalmente o mesmo nome do banco de dados no *MySql*)

A classe *SpringMvcConfig* herda da classe *WebMvcConfigurerAdapter* para ter acesso a alguns métodos que facilitam a configuração da aplicação. Esses métodos decorados com *@Override* configuram o uso do *ServletHandler* padrão do *Spring* e como a aplicação deve servir diferentes tipos de conteúdo ao usuário.

Os dois primeiros métodos decorados com *@Bean* retornam objetos com informações de onde e como encontrar as *views* da aplicação. O *InternalResourceViewResolver* armazena configurações sobre em que diretório estão as *views* (*WEB-INF*) e suas extensões (*.jsp*). Assim é dito ao *Spring Framework* para procurar páginas *.jsp* dentro do caminho */WebContent/WEB-INF*.

O Segundo método configura o *ContentNegotiatorViewResolver*, uma classe de uso interno do *Spring* que resolve *views* baseadas no cabeçalho das requisições *HTTP*. Note que ele recebe uma coleção de *ViewResolvers* como parâmetro e é adicionado o *InternalResourceViewResolver* do método anterior nessa lista.

Os três últimos métodos decorados com a anotação *@Bean* retornam objetos relacionados ao uso do banco de dados e manipulação de entidades. O primeiro deles retorna um objeto do tipo *ComboPooledDatasource* que recebe vários parâmetros para se conectar ao banco de dados e servir como fonte de dados para a aplicação.

O segundo método cria um objeto do tipo *LocalContainerEntityManagerFactoryBean* que gerenciará a criação de uma implementação da interface *EntityManager*. Implementações de *EntityManager* são usadas nos repositórios para persistir objetos e consultar o banco de dados. Entre os parâmetros que o *LocalContainerEntityManagerFactoryBean* recebe estão um adaptador para o uso do *Hibernate*, o nome do pacote onde são criadas entidades e o *ComboPooledDatasource* configurado anteriormente para ser usado como fonte de dados. Ele também recebe um objeto do tipo *Properties* contendo várias configurações do *Java Persistence API* (*JPA*). Para informações mais

detalhadas sobre o *JPA*, recomenda-se o livro *Pro JPA 2.0*, 2ª edição, da editora *Appress*.

O último método retorna um objeto *JpaTansactionManager*. Esse objeto é o responsável por gerenciar as transações com o banco de dados. Ele recebe a instancia do objeto *LocalContainerEntityManagerFactoryBean* configurado anteriormente como parâmetro.

3.2 Criando um projeto *ASP.NET MVC* 5 no *Visual Studio Community 2013*

O link *"new project..."*, na tela inicial do Visual *Studio Community 2013*, mostra um menu com diversos modelos para criação de projetos. Para o projeto desse trabalho, é usando o *"ASP.NET Web Application"* o menu *"Visual C#"*.

Um projeto no *Visual Studio* faz parte de uma solução. Uma solução além de ser uma coleção de projetos, contém informações de dependências e configurações. A Figura 17 ilustra a criação de um novo projeto. Os arquivos de solução junto com os de projeto são os equivalentes aos arquivos de projeto do *Eclipse* e o arquivo *build.gradle*. vsnewproject.png

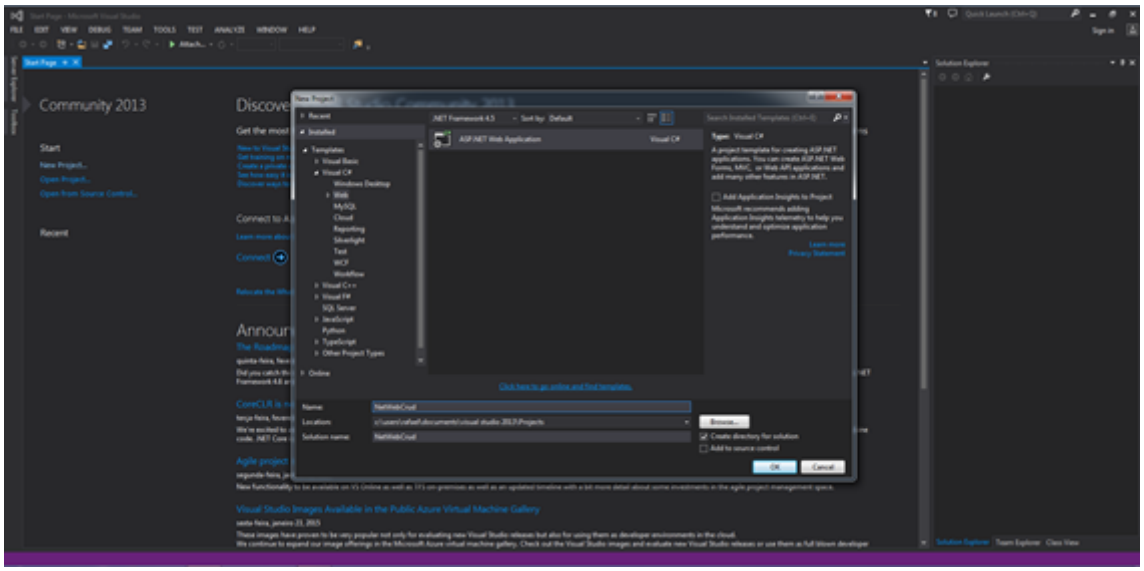

Figura 17: Criando um novo projeto no *Visual Studio Community 2013*

O nome da solução de exemplo é *"NetWebCrud"*. O *Visual Studio* automaticamente dá o mesmo nome da solução para o primeiro projeto criado. O local de armazenamento dos arquivos também pode ser configurado nessa tela. Clicando no botão "OK", aparecerá o menu mostrado na Figura 18.

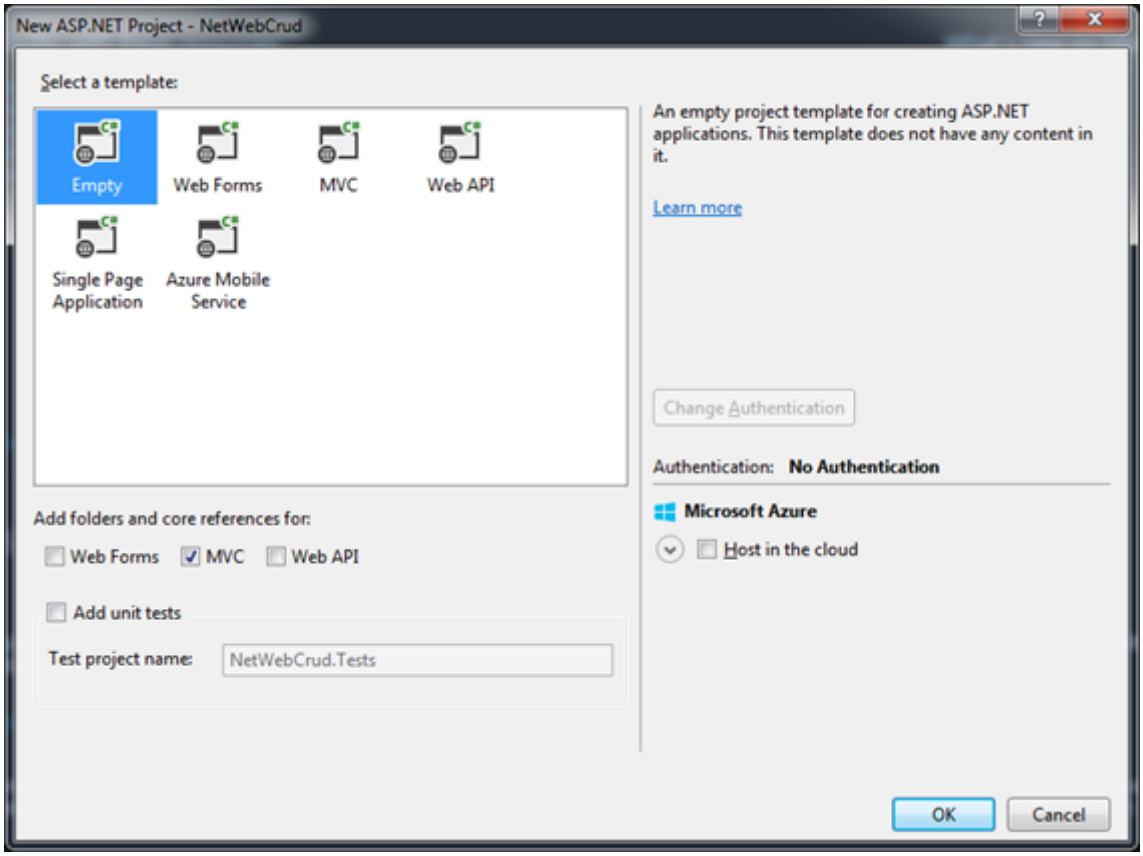

Figura 18: Opções de um novo projeto *web* no *Visual Studio*

Existem várias opções de projetos para *web*. Para um projeto *MVC* contendo apenas o mínimo necessário para seu funcionamento, é escolhido o modelo *"Empty"* e é marcada a opção *MVC*. O serviço *Microsoft Azure* não é utilizado, então a opção *"Host in the Cloud"* é desmarcada. Clicando em "OK", o *Visual Studio* irá gerar um projeto com a estrutura mostrada pela Figura 19.

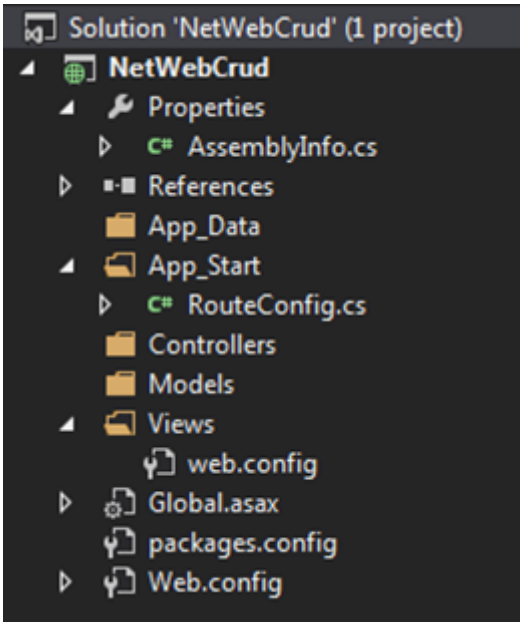

Figura 19: Estrutura de um projeto *ASP.NET MVC* 5

O *ASP.NET MVC* 5 funciona seguindo o princípio de convenção à configuração. O desenvolvedor deve seguir várias convenções para o desenvolvimento de sua aplicação, em compensação, a quantidade de configuração necessária é mínima.

Todas as páginas, e subpastas que contém páginas, devem ficar inseridas na pasta *Views*, todos os *controllers* devem fica na pasta *Controllers*. No *ASP.NET MVC* existe o conceito de *Areas*, onde o desenvolvedor pode organizar melhor seu código, mas esse conceito não é abordado nesse trabalho. A pasta *Models* pode conter entidades de negócio, repositórios e quaisquer outras classes da camada *Model* do padrão *MVC*, mas o desenvolvedor também pode criar tais classes em outros projetos dentro da solução.

A classe *AssembyInfo* contém informações como nome da aplicação, empresa desenvolvedora e número de versão. Esse arquivo pode ser editado pelo desenvolvedor para atualizar os metadados da aplicação. A pasta *References*, contém as bibliotecas que são usadas no projeto. A pasta *App_Data* pode conter um arquivo de banco de dados minimalista para armazenar dados, mas como é usado o banco de dados *MySql*, essa pasta é apagada manualmente.

A pasta *App_Start* existe para que o desenvolvedor possa armazenar arquivos de inicialização da aplicação. Um modelo de um novo projeto *ASP.NET MVC* 5 vêm com a classe *RouteConfig* onde a tabela de rotas que mapeia *URLs* para *controllers* é configurada. Esse conteúdo desse arquivo é explicado no próximo capítulo durante a criação de *controllers*.

O arquivo *Global.asax* contém o método *Application_Start*. Esse método é chamado uma vez quando a aplicação é iniciada e nele podem ser incluídas linhas de código que serão executadas quando a aplicação iniciar. No modelo de projeto escolhido, ele já vem com uma chamada ao método *RegisterRoutes* da classe *RouteConfig* e *RegisterAllAreas* da classe *AreaRegstration*.

O Arquivo *Web.config* na pasta raiz do projeto é um arquivo *XML* que contém diversas configurações para o projeto. O arquivo *web.config* na pasta *Views* contém configuração apenas para a criação de *views*.

3.2.1 Adicionando dependências usando o *NuGet*

Do mesmo modo que o *Gradle* pode adicionar bibliotecas externas ao projeto *Java*, o *NuGet* é uma ferramenta que adiciona e gerencia pacotes nos projetos do Visual Studio. Nesse projeto de exemplo são usadas as seguintes bibliotecas:

- *MySql.Data.Entity 6.9.5* - Biblioteca necessária para o *MySql* funcionar com o *Entity Framework 6*. Adicionando essa biblioteca, o *NuGet* também adiciona o *driver do MySql* e o *Entity Framework 6* ao projeto.
- *Ninject MVC5 3.2.2* - Injeção de dependências para projetos *ASP.NET MVC* 5 .

O console do *NuGet* pode ser acessado no menu *TOOLS/NuGet Package Manager/Package Manager Console*, como mostra a Figura 20.

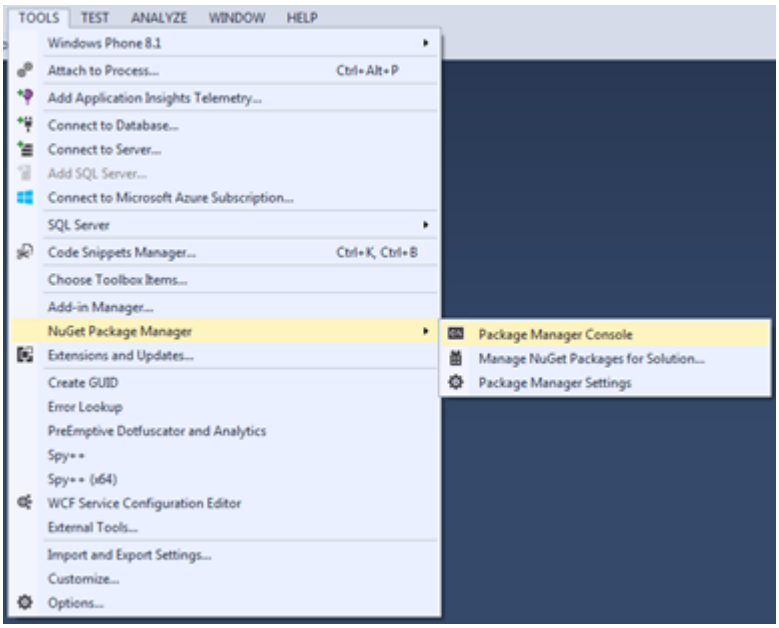

Figura 20: Acessando o console do *NuGet*

O console do *NuGet* é exibido no *Visual Studio* como mostra a Figura 21. Nele o desenvolvedor pode digitar comandos para procurar e adquirir bibliotecas para seus projetos.

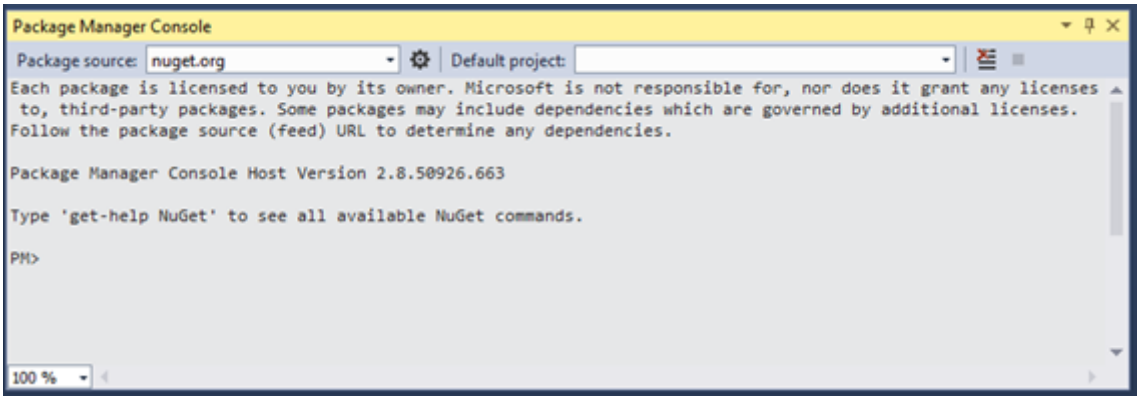

Figura 21: Console do NuGet

Para instalar as bibliotecas citadas anteriormente, executa-se os seguintes comandos no console do *NuGet*:

- *Install-package mysql.data.entity*
- *Install-package ninject.mvc5*

Quando o *NuGet* terminar de baixar todos os arquivos, eles são adicionado ao projeto. A Figura 22 ilustra a instalação das bibliotecas.

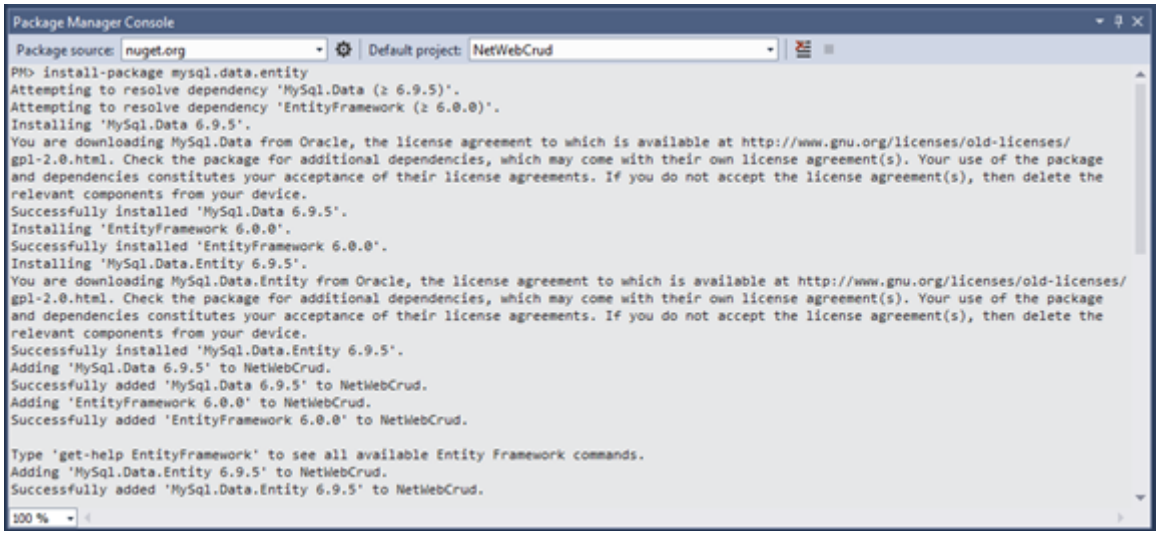

Figura 22: Console do *NuGet*

Ao terminar a instalação, os arquivos *packages.config* e *Web.config* são atualizados com as referências e configurações das novas bibliotecas.

A instalação do *Ninject Mvc 5* adiciona o arquivo *NinJectWebCommon.cs* à pasta *App_Start*. Na Figura 23 pode-se observar o arquivo adicional citado. O conteúdo e funcionalidade desse arquivo é abordado no capitulo 5.

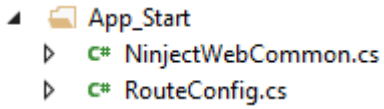

Figura 23: Arquivo adicional do *Ninject*

3.2.2 Adicionando informações de conexão com o banco de dados

O acesso ao banco de dados é configurado no arquivo *Web.config* na raiz do projeto. Para adicionar informações de conexão com o banco *MySql*, é adicionada a *tag connectionStrings* como no exemplo mostrado no quadro 6.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="entityFramework"
                type="System.Data.Entity.Internal.ConfigFile.
                    EntityFrameworkSection, EntityFramework, Version
                    =6.0.0.0, Culture=neutral, PublicKeyToken=
                    b77a5c561934e089" requirePermission="false" />
  </configSections>
  <connectionStrings>
    <add name="NetWebCrudContext" connectionString="Server=localhost;
        Database=net_web_crud;Uid=root;Pwd=1234;" providerName="MySql.Data.
        MySqlClient"/>
  </connectionStrings>
...
```

Código 6: Adicionando configurações de conexão ao *Web.xml*

A *tag configuration* é a raiz do arquivo de configuração. A *tag configSections* deve ser a primeira filha da *tag* raiz, caso não seja, a aplicação irá reportar um erro de configuração. Note que já existem referências ao *Entity Framework*. A *tag connectionStrings* deve ser filha da *tag configuration*.

A *tag add* é usada para adicionar informações de conexão com o banco de dados. É necessário dar um nome para a conexão, configurar a *connection string* propriamente dita e o nome da classe responsável pelo acesso ao banco. A *connection string* deve conter no mínimo informações sobre para qual servidor apontar, qual o nome do banco de dados e usuário/senha utilizados.

3.3 Conclusão

Criar um novo projeto com o *Spring MVC* requer escrita de código, mais passos e conhecimento de como o *framework* funciona internamente. A vantagem de um projeto com *Spring* é a questão da modularidade de componentes (o desenvolvedor adiciona ao projeto somente aquilo que precisa) e uma liberdade maior de configuração.

Uma maneira de tornar a configuração inicial de projetos *Java* com *Spring* mais

simples é usar o módulo *Spring Boot*. O *Spring Boot* configura automaticamente a aplicação a partir dos arquivos *.jar* contidos no projeto. Ele detecta as bibliotecas populares e "adivinha"como o projeto deve ser configurado. Internamente ele cria de forma automática a configuração que foi demonstrada na seção 2.1.3.

As vantagens de criação de um novo projeto *ASP.NET MVC* no *Visual Studio* são os modelos de projetos que já vem prontos logo após a instalação. Não existe a necessidade de realizar tantas configurações iniciais, contudo menos aspectos do projeto são configuráveis. O desenvolvedor deve obedecer as convenções do *ASP.NET MVC*.

No próximo capitulo será mostrado como criar *controllers* e *views* nos dois tipos de projetos.

4 Criando controllers e views

Um *controller* é uma classe que recebe requisições, processa informações (ou chama outras classes processa-las) e devolve um resultado ao usuário. Resultados comuns são páginas da *web* e objetos *JSON* ou *XML* para processamento no navegador (usando Javascript). *Views*, no contexto de uma aplicação *web*, são as próprias páginas que serão exibidas ao usuário.

Nesse capitulo é demonstrado como criar *controllers* e *views* em ambos os ambientes de desenvolvimento. São expostos exemplos de como criar *controllers* com ações que retornam páginas, objetos *JSON* e exemplos de ações que aceitam parâmetros. São dados também exemplos de como usar as *view engines* que geram páginas *HTML* dinamicamente.

4.1 Criando um *controller* com uma ação que retorna uma página

Uma das funções mais comuns dos *controllers* é retornar páginas para o usuário. Nas próximas seções é criado um *controller* simples que irá retornar uma página estática para exibição no navegador.

Numa aplicação *MVC*, um controller não deve instanciar nem usar métodos de outro *controller*. Se o desenvolvedor achar que isso é necessário, é prudente considerar incluir a funcionalidade necessária em uma classe de serviços (ou classe de negócios). *Controllers* devem apenas receber requisições, se for o caso chamar outras classes para processar requisições, e no fim devolver o resultado ao usuário. O capitulo 5 abordará a criação de classes de serviço.

Os exemplos desse capítulo, irão manipular uma classe chamada *Person*. Os quadros 7 e 8 mostram o código das classes em *Java* e *C#*.

```
package br.uece.webCrud.model;
import java.util.Date;

public class Person {
        private String name;

        @DateTimeFormat(pattern = "yyyy-MM-dd")
        private Date birthDate;

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public Date getBirthDate() {
                return birthDate;
```

```
        }

        public void setBirthDate(Date birthDate) {
                this.birthDate = birthDate;
        }
}
```

Código 7: Classe *Person* em Java

A anotação *@DateTimeFormat* especifica em qual formado o capo de data será escrito. A escolha desse formato será explicada na seção 4.1.2.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace NetWebCrud.Models
{
    public class Person
    {
        public string Name { set; get; }
        public DateTime BirthDate { set; get; }
    }
}
```

Código 8: Classe *Person* em C#

4.1.1 Java

Controllers no *Spring MVC* são classes decoradas com a anotação *@Controller*. Essas classes podem estar contidas em qualquer pacote do projeto. No projeto exemplo, elas estarão contidas no *pacote br.uece.webCrud.controller*. O código contido no quadro 9 mostra um exemplo de um *controller*, com o nome de *PersonController* e com uma ação que retorna uma página.

```
package br.uece.webCrud.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/person")
public class PersonController {

        @RequestMapping(value = "/add", method = RequestMethod.GET)
        public String addPage() {
                return "person/add";
        }
```

```
}
```

Código 9: *PersonController* no projeto *Spring*

A anotação *@RequestMapping* configura com qual o caminho, a partir da raiz da aplicação, as ações do *controller* são acessadas. A mesma anotação em uma ação configura qual o caminho para acessa-la a partir do caminho do controller. Por exemplo, para acessar a ação *addPage* o usuário entraria com o endereço <raiz da aplicação>/person/add. É possível não usar o *@RequestMapping* no *controller* e usá-lo apenas na ação, mas fazendo isso o caminho para a ação será em relação à raiz da aplicação. Também é possível configurar para qual método *HTTP* a ação irá responder. No exemplo acima, a ação é executada apenas para requisições usando o método *GET*.

No *Spring MVC* uma das maneiras de retornar uma página para o usuário é retornar um objeto *String* com o caminho do arquivo da página. Lembrando que na seção 3.1.3 foram configurados um prefixo e um sufixo no objeto *InternalResourceViewResolver*, e na seção 2.1 o *Gradle* foi configurado com o nome da pasta onde ficará armazenado o conteúdo especifico da *web*. Então o caminho completo para o arquivo que será retornado é <raiz do projeto>/WebContent/WEB-INF/person/add.jsp.

Agora é criada a página que essa primeira ação irá retornar. É criada uma pasta chamada *person* dentro de *WEB-INF*, e dentro dessa pasta é criado o arquivo *add.jsp*, como pode ser visto na Figura 24.

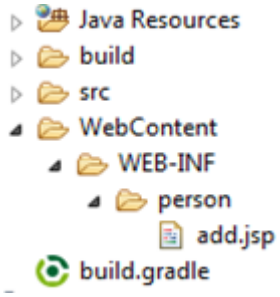

Figura 24: Primeira *view* adicionada ao projeto *Java*

O conteúdo do arquivo *add.jsp* é editado como no exemplo do quadro 10.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.
    w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Add Person</title>
</head>
<body>
        <h1>Add Person</h1>
        <form method="POST">
                <p>Name: <input type="text" name="name" /></p>
```

```
                <p>Birth: <input type="date" name="birthDate"/></p>
                <p><input type="submit" value="save"/></p>
        </form>
</body>
</html>
```

Código 10: Arquivo *add.jsp*

A diretiva *page* no cabeçalho da página diz ao servidor como tratar o arquivo, no caso como uma página *HTML*. O resto arquivo é um formulário *HTML* comum, que recebe dois valores (o nome e a data de nascimento de uma pessoa) e tem um botão que os envia para o servidor. Note que o formulário faz o envio usando o método *POST*.

4.1.2 ASP.NET MVC

Um *controller* é adicionado ao projeto *ASP.NET MVC* clicando com o botão direito do mouse na pasta *Controllers*, em seguida selecionado as opções *"Add"* e *"Controller"*. A Figura 25 ilustra a adição do *controller*.

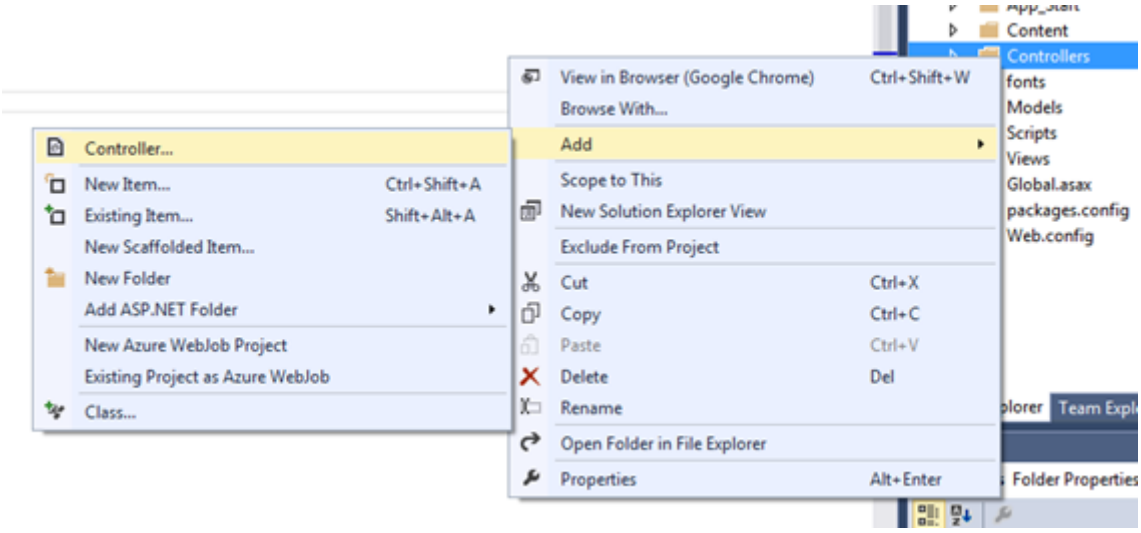

Figura 25: Adicionando um controller no Visual Studio

São exibidas diversas opções de modelos de *controllers*, a opção usada é a *"MVC 5 Controller – Empty"* e o *controller* foi também nomeado como *PersonController*. Uma das convenções do *ASP.NET MVC* é que *controllers* sempre tem que terminar com a palavra *"Controller"*. Normalmente é recomendado que se nomeie *controllers* com o nome da entidade que eles manipulam como prefixo, para facilitar a organização do código.

Também é adicionada uma ação chamada *Add* que retorna uma página. O código do *controller* deve ficar como no quadro 11. Note que *PersonController* é subclasse da classe *Controller*.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace NetWebCrud.Controllers
{
    public class PersonController : Controller
    {
        [HttpGet]
        public ActionResult Add()
        {
            return View();
        }
    }
}
```

Código 11: *PersonController* em *C#*

A ação *Add* retorna um objeto do tipo *ActionResult*, que é uma classe pai de vários tipos de resultados no *ASP.NET MVC* como *JSONResult* e *ViewResult*. O método *View* retorna um *ViewResult*, que é uma página *web*. A ação *Add* também está decorada com anotação *HttpGet*, logo essa ação irá responder apenas ao método *GET*.

No *ASP.NET MVC*, o mapeamento de endereço para *controllers* e ações é centralizado pela tabela de rotas, não por anotações isoladas em cada *controller* e método. Quem configura a tabela de rotas é a classe *RouteConfig*, mostrada no quadro 12.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace NetWebCrud
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new {
                                    controller = "Home",
                                    action = "Index",
                                    id = UrlParameter.Optional
                                }
            );
        }
```

```
    }
}
```

Código 12: A classe *RouteConfig*

A configuração da tabela de rotas começa ignorando qualquer requisição a arquivos com extensão *.axd*, que são arquivos de recursos do *ASP.NET*. Logo em seguida é configurada a rota padrão. O padrão controller/action/id determina que o acesso às ações devem seguir o caminho <raiz da aplicação>/<nome do controller>/<nome da ação>/<parâmetro opcional chamado "id». Então para acessar a ação *Add* de *PersonController* o caminho é <raiz da aplicação>/person/add. Note que não é necessário digitar *personController* no caminho, apenas o prefixo é necessário. A rota padrão vem configurada com o apontando para uma ação chamada *Index* de um *controller* chamado *HomeController* se apenas o endereço da raiz da aplicação for requisitado. O desenvolvedor pode mudar esses valores.

Para adicionar uma *view*, se clica com o botão direito do *mouse* em uma ação e então na opção *"Add View..."* como mostrado na Figura 26.

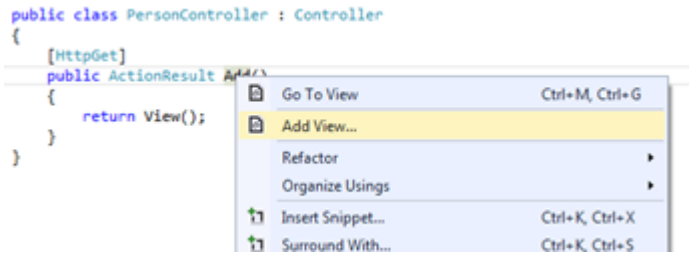

Figura 26: Adicionando uma *view* no *ASP.NET MVC*

Outra convenção do *ASP.NET MVC* é que views devem ter o nome das ações que as retornam e ficar dentro de uma pasta com o nome do *controller*, como na Figura 27. É usado o modelo *"Empty"* na criação dessa primeira *view*. A extensão de arquivos de views em projetos *ASP.NET MVC 5* é *.cshtml*.

Figura 27: *View* adicionada ao projeto *ASP.NET MVC*

A pasta *Shared* pode conter *views* acessíveis à todos os *controllers*. Em alguns modelos de projetos do *Visual Studio* ela contém um arquivo chamado *_Layout.cshtml*. Esse arquivo é uma página modelo (ou *layout*) usada quando se quer aproveitar uma mesma estrutura para várias *views* diferentes. Essa pasta também pode armazenar *Partial Views*, que são pedaços de *views* e componentes que o desenvolvedor pode usar em diversas páginas. *Layouts* e *Partial Views* não são abordados nesse trabalho.

O conteúdo do arquivo *Add.cshtml* é aditado para que fique igual ao exemplo do quadro 13.

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Add</title>
</head>
<body>
    <h1>Add Person</h1>
    <form method="POST">
        <p>Name: <input type="text" name="name" /></p>
        <p>Birth: <input type="date" name="birthDate" /></p>
        <p><input type="submit" value="save" /></p>
    </form>
</body>
</html>
```

Código 13: O arquivo *Add.cshtml*

Tanto no projeto *Java/Spring* como no projeto *ASP.NET MVC*, quando acessada pelo Google Chrome, a view é exibida como na Figura 28.

Figura 28: Resultado de exibição da *view*

4.2 Criando ações parametrizadas

Na seção 4.1, foram criadas ações que respondem ao método *HTTP GET* e retornam uma página ao usuário. Nessa seção essa página será utilizada para enviar informações ao servidor utilizando o método *HTTP POST*. Na *views* de exemplo, o atributo *action* não está definido nos formulários, então ele vai enviar informações ao servidor para o mesmo caminho usado para acessá-lo (<raiz da aplicação>/person/add).

Devem ser criadas ações nos *controllers* que respondam ao mesmo endereço da página, mas usem método *POST*. Essas ações também devem receber as informações que o formulário irá enviar. Isso é feito configurando anotações e adicionando parâmetros às ações em ambos os projetos. Nos exemplos das seções 4.2.1 e 4.2.2 são feitas ações que recebem um objeto do tipo *Person* e o adicionam à uma coleção armazenada em memória.

4.2.1 Java

O código presente no quadro 14 é adicionado à classe *PersonController*:

```
private List<Person> persons;

@RequestMapping(value = "/add", method = RequestMethod.POST)
public String addPost(@ModelAttribute Person person) {

        if(persons == null) {
                persons = new ArrayList<Person>();
        }
        persons.add(person);

        return "redirect:/person/list";
}
```

Código 14: Ação de *PersonController* no projeto Java que responde ao método *POST*

A anotação *@RequestMapping* no método *addPost* aponta para o mesmo caminho que o método *addPage* mas ela responde às chamadas usando o método *POST*. O método *addPost* recebe como parâmetro um objeto do tipo *Person*. Os parâmetros de métodos também podem ser decorados com anotações.

A anotação *@ModelAttribute* permite que o *Spring MVC* procure e mapeie valores do cabeçalho *HTTP* para o objeto *person*. No quadro 10, o atributo *name* dos elementos *input* onde são digitadas as informações de um novo objeto são *"name"* e *"birthDate"*, os mesmos nomes dos atributos da classe *Person*.

A Figura 29 mostra as ferramentas de desenvolvedor do *Google Chrome* com um exemplo do que acontece quando se envia os dados da página ao servidor.

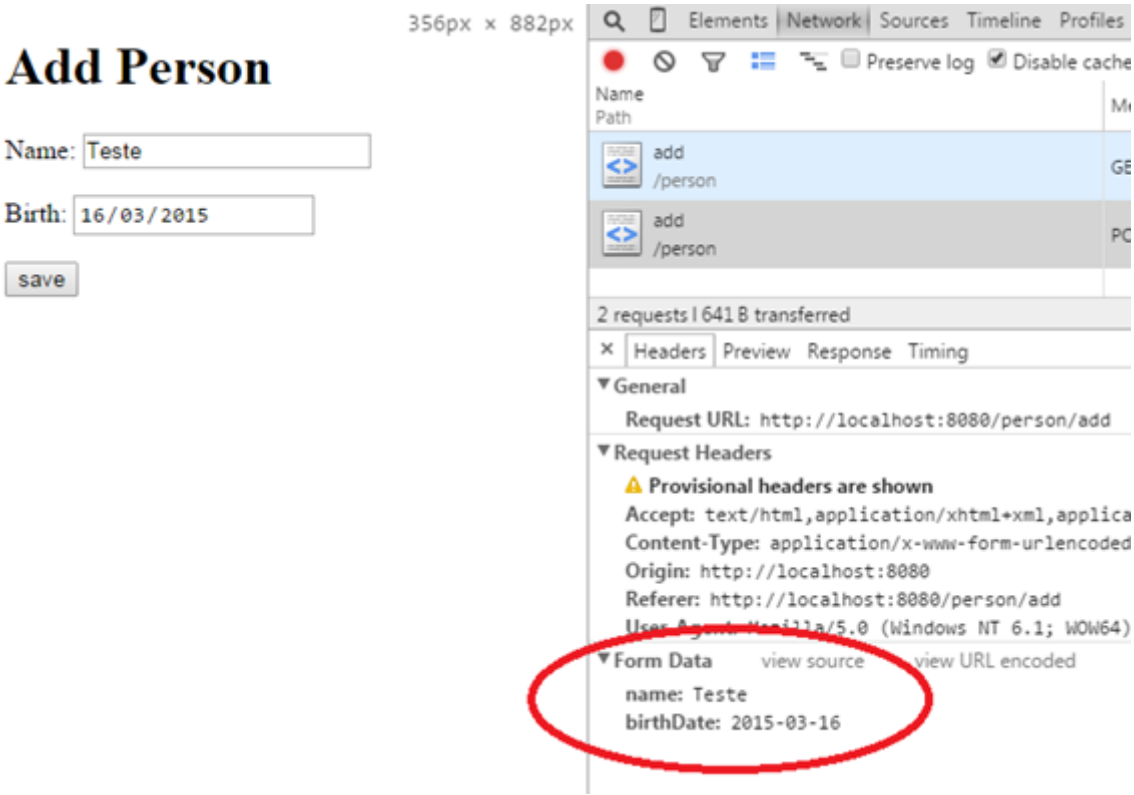

Figura 29: Enviando dados para o servidor

No quadro 7, o atributo *birthDate* da classe *Person* foi decorado com a anotação *@DateTimeFormat* para que aceite o formato de data “yyyy-MM-dd”, a Figura 29 mostra que esse é o formato de data que o *Google Chrome* envia para o servidor. Se o formato de data não estivesse configurado ou fosse enviado em outro padrão, o servidor enviaria uma reposta de erro 400 (*Bad Request*).

Quando chegam ao servidor, os dados são mapeados automaticamente para o parâmetro *person* do método *addPost* como mostrado na Figura 30, e a partir daí o desenvolvedor pode trabalhar com os dados.

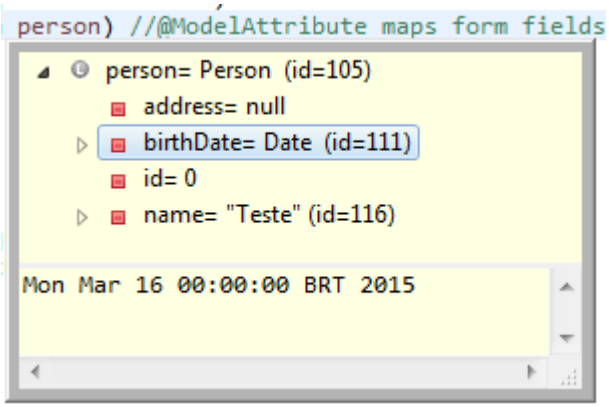

Figura 30: Objeto *person* com parâmetros corretamente populados no projeto *Java*

O retorno do método *addPost* redireciona a requisição para outra ação que mostra uma página com uma tabela com todas pessoas adicionadas à lista. As implementações

dessa ação e página serão demonstradas em seções posteriores.

4.2.2 ASP.NET MVC

O código apresentado no quadro 15 foi adicionado ao *PersonController*.

```
private IList<Person> Persons
{
        get
        {
                var persons = HttpContext.Application["persons"];
                if (persons == null)
                {
                        HttpContext.Application["persons"] = persons = new
                            List<Person>();
                }
                return (IList<Person>)persons;
        }
}

[HttpPost]
public ActionResult Add(Person person)
{
        persons.Add(person);
        return RedirectToAction("List");
}
```

Código 15: *PersonController* do projeto *ASP.NET* com nova ação

O *ASP.NET MVC* não guarda o estado de objetos entre requisições, então a lista deve ser guardada como uma variável da aplicação. Armazenada dessa forma, o estado da lista se preserva e está disponível para todos os usuários. Lembrando que em capítulos posteriores, a lista será substituída por serviços que persistem as entidades no bando de dados.

Por convenção do *ASP.NET MVC*, a ação do *controller* que reponde ao método *POST* deve ter o mesmo nome da *view* a qual ele responde. A única configuração necessária é decorar a ação com a anotação *HttpPost*. O *ASP.NET MVC* faz o mapeamento dos dados para o objeto *person* de forma semelhante ao *Spring MVC*, só que não é necessário especificar o formato do campo *birthDate*. A Figura 31 mostra o resultado do mapeamento.

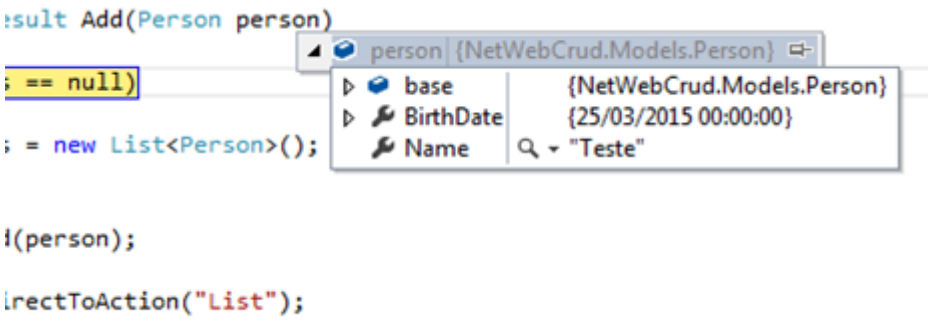

Figura 31: Objeto *person* no *ASP.NET MVC* corretamente populado

Assim como no projeto *Java/Spring*, o objeto *person* é adicionado em uma lista e a ação redireciona o usuário para uma página que mostra todos os objetos da lista.

4.3 Retornando um objeto JSON

Aplicações web modernas utilizam a técnica *AJAX* para atualizar partes de páginas em vez de enviar uma requisição para que o servidor envie uma página completa novamente. Essa técnica é essencial para dar ao usuário uma melhor experiência de uso da aplicação. Os dados são enviados e recebidos assincronamente utilizando código *Javascript*, esses dados podem estar no formato *XML* ou *JSON*. Nessa seção é demonstrado como retornar dados no formato *JSON*. Como exemplo é retornado um único objeto do tipo *Person* para o usuário.

Para mais informações sobre como manipular dados com *Javascript* e *AJAX*, é recomendada a leitura dos livros *Pro Javascript For Web Apps* e *Pro jQuery* da editora *Apress*.

4.3.1 Java

A biblioteca *Jackson Databind* é responsável por converter objetos *Java* no formato *JSON*. Essa biblioteca foi adicionada ao projeto durante a configuração do arquivo *build.gradle* na seção 3.1.2.

Para uma ação retornar um objeto *JSON*, ela deve retornar um objeto *Java* decorado com a anotação *@ResponseBody*, como no código do quadro 16.

```
@RequestMapping(value = "/getOne", method = RequestMethod.GET)
public @ResponseBody Person getOne()
{
        Person p = new Person();
        p.setName("Teste");
        p.setBirthDate(new Date());
        return p;
```

```
}
```

Código 16: Ação no *Spring MVC* que retorna um objeto *JSON*

Acessando a ação *getOne* pelo navegador, é obtida a resposta exposta na Figura 32.

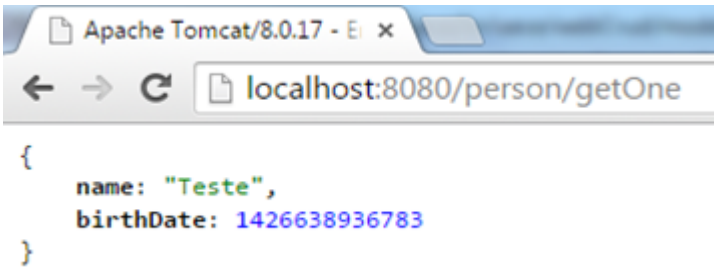

Figura 32: Resultado *JSON* no projeto *Java*

O atributo *birthDate* é enviado ao navegador como um número que pode ser usado para criar um objeto *Date* do *Javascript*.

4.3.2 ASP.NET MVC

O *ASP.NET MVC* pode trabalhar com objetos no formato *JSON* sem precisar de uma biblioteca de terceiros. O exemplo do quadro 17 mostra uma ação que retorna um objeto *JSON*.

```
[HttpGet]
public ActionResult GetOne()
{
        var p = new Person
        {
                Name = "Test",
                BirthDate = DateTime.Now
        };

        return Json(p, JsonRequestBehavior.AllowGet);
}
```

Código 17: Ação no *ASP.NET MVC* que retorna um objeto *JSON*

O método *Json* no final da ação faz a conversão de objetos para o formato *JSON*. Por padrão o *ASP.NET MVC* só envia objetos *JSON* usando o método *HTTP POST*, para a habilitar o uso de *HTTP GET* é necessário um parâmetro adicional. A Figura 33 mostra a resposta à ação *GetOne*.

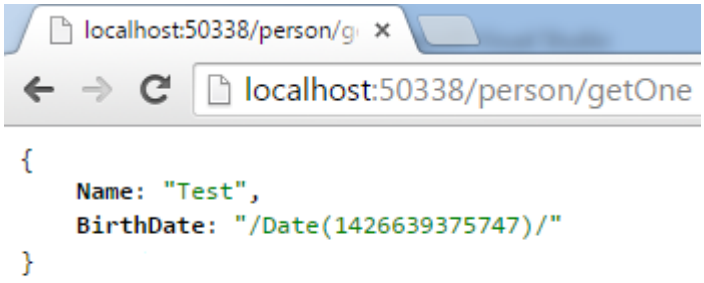

Figura 33: Resultado *JSON* no projeto *ASP.NET MVC*

Aqui existe uma inconveniência. O modo como o *ASP.NET* envia datas faz com que seja necessário mais código *Javascript* para extrair o número que realmente representa a data e criar um objeto *Date*.

4.4 Gerando *Views* Dinâmicas

Nessa seção é demonstrado como fazer uma página que exibe todos os itens das listas de objetos *Person* em ambos os projetos. São feitas ações que enviam as listas para as *views* que montam o código *HTML* usando as *view engines* padrão de cada tecnologia, *JSP Standart Tag Library* (*JSTL*) no projeto *Java/Spring* e *Razor* no projeto *ASP.NET MVC*.

4.4.1 Java (JSTL)

Observe o método *listPage* na classe *PersonController* no exemplo do quadro 18.

```
@RequestMapping("/list")
public ModelAndView listPage()
{
        if(persons == null) {
                persons = new ArrayList<Person>();
        }

        ModelAndView result = new ModelAndView("person/list");
        result.addObject("personsList", persons);
        return result;
}
```

Código 18: Ação no projeto *Java* que retorna uma página com a lista de *Persons*

O tipo de retorno do método *listPage* e um objeto do tipo *ModelAndView*. Esse objeto representa uma página dinâmica que pode conter e manipular objetos *Java*. O método é mapeado para o endereço <raiz da aplicação>/person/list.

A ação começa com a verificação de existência, e se for o caso criação da lista. Logo em seguida ele cria um objeto *ModelAndView* que aponta para uma nova página que será criada dentro da pasta *person*, *list.jsp*. O método *addObject* adiciona a lista *persons* para que a página possa utilizá-la com o nome de referência *personsList*. O Resultado então é retornado para o usuário. O quadro 19 mostra o código da página *list.jsp*.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.
   w3.org/TR/html4/loose.dtd">
<html>
<head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO
            -8859-1">
        <title>List Persons</title>
</head>
        <body>
                <h1>Person List</h1>
                <a href="add">Add Person</a>
                <table>
                        <thead>
                                <tr>
                                        <th>Name</th>
                                        <th>Birth Date</th>
                                </tr>
                        </thead>
                        <tbody>
                                <c:forEach items="${personsList}" var="
                                    person">
                                    <tr>
                                        <td>${person.name}</td>
                                        <td>${person.birthDate}</td
                                            >
                                    </tr>
                                </c:forEach>
                        </tbody>
                </table>
        </body>
</html>
```

Código 19: Código da página *list.jsp*

A página começa com o cabeçalho padrão de páginas *jsp*. Logo abaixo, o cabeçalho *taglib* adiciona a referência à biblioteca *JSTL* básica. Essa biblioteca contém um conjunto de *tags JSTL*, pedaços de código *Java* que ajudam a montar páginas dinâmicas. As *tags JSTL* são escritas na página como pedaços de código que lembram *tags HTML*, mas devem começar utilizando prefixo configurado no cabeçalho *taglib* (no caso do exemplo acima, a letra "c"). Até o fim da *tag HTML* <thead>, o que está escrito na página é apenas *HTML* puro. A parte onde o *JSTL* entra em ação é dentro do corpo da tabela.

A *tag forEach* cria um pedaço de bloco *HTML* para cada item em uma coleção. Essa coleção é configurada no atributo *items* e usando a sintaxe do *JSTL* é dito que a coleção se chama *personList*, o mesmo nome que foi usado como referencias para passar a lista de objetos *Person* para a página. O atributo *var* configura um nome de referência para o objeto que é usado em cada iteração da lista.

Dentro de cada iteração é criada uma nova linha para a tabela, e em cada linha é criada duas colunas, uma mostrando o atributo *name* e outra o *birthDate* do objeto *person*. Note que não é necessário chamar os métodos *getName* e *getBirthDate*, é necessário apenas referenciar o nome do atributo que o desenvolvedor deseja mostrar na página. Depois de adicionar alguns objetos à lista, é obtido o resultado exposto na Figura 34.

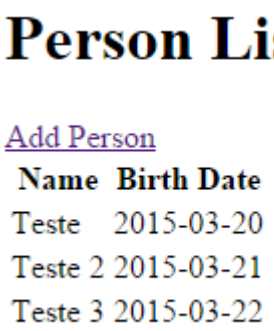

Figura 34: Resultado da página *list.jsp*

O *JSTL* possui muitas outras *taglibs*. Para mais informações, consulte a documentação do *JSTL*.

4.4.2 ASP.NET (Razor)

O exemplo do quadro 20 mostra a ação que retorna a página que exibe o conteúdo da lista.

```
[HttpGet]
public ActionResult List()
{
        return View(Persons);
}
```

Código 20: Ação que retorna a página *list.cshtml*

O método *View* pode receber como parâmetro qualquer objeto que o desenvolvedor queira passar para a página, aqui ele recebe a lista de objetos do tipo *Person*. O quadro 21 mostra o código fonte da página *list.cshtml*.

```
@using NetWebCrud.Models
@model List<Person>
```

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>List Persons</title>
</head>
<body>
    <h1>Person List</h1>
    <a href="add">Add Person</a>
    <table>
        <thead>
            <tr>
                <th>Name</th>
                <th>Birth Date</th>
            </tr>
        </thead>
        <tbody>
            @foreach(var person in Model) {
                <tr>
                    <td>@person.Name</td>
                    <td>@person.BirthDate</td>
                </tr>
            }
        </tbody>
    </table>
</body>
</html>
```

Código 21: Código fonte da página *list.cshtml*

Além da sintaxe da *view engine Razor* (que mistura código *C#* com *HTML*) uma diferença em relação à página do projeto *Java* é que uma página no *ASP.NET MVC* pode ser fortemente tipada.

A primeira linha da página importa o *namespace NetWebCrud.Models* para se utilizar de suas classes. A segunda linha informa qual é o tipo de objeto que será o modelo da página, uma lista de objetos *Person* como a que foi passada na ação do *controller*.

O objeto *Model* dentro do laço *foreach* é a referência ao objeto que foi enviado para a página. *Model* faz referência a qualquer tipo de objeto e a diretiva *@model* no começo da página especifica seu tipo para que os recursos de auto completar do *Visual Studio* estejam disponíveis e o *ASP.NET MVC* possa gerar a página. O resultado pode ser visto na Figura 35.

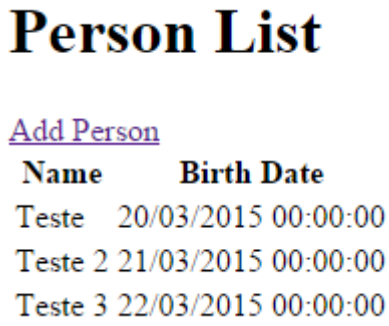

Figura 35: Resultado da página *list.cshtml*

Note que o *ASP.NET MVC* exibe também a hora do atributo *BirthDate*.

4.5 Conclusão

Enquanto no *Spring MVC* o desenvolvedor deve decorar seus *controllers* e ações com várias anotações *@RequestMapping*, no *ASP.NET MVC* o mapeamento de endereços é centralizado inteiramente na tabela de rotas. Isso junto com as convenções do *ASP.NET MVC* ajudam o desenvolvedor a escrever menos código.

Os *controllers* no *Spring MVC* podem ser qualquer classe decorada com a anotação *@Controller*, no *ASP.NET MVC* devem ser classes que herdem da superclasse *Controller* e tenham o sufixo *"Controller"* no seu nome. O modo de criação de *controllers* no *Spring MVC* é mais flexível, mas é uma boa prática também adicionar o sufixo *"Controller"* em tais classes e deixa-las em um pacote especifico para *controllers* objetivando uma melhor organização de código.

Ao enviar dados para o servidor, a única diferença significativa é o modo de enviar datas. O mapeamento dos dados da requisição *HTTP* para objetos é simples e funcional em ambas as tecnologias.

O *ASP.NET MVC* dá suporte nativo para manipulação de objetos no formato *JSON*, bastando apenas que o desenvolvedor utilize o método *Json* para gerar um objeto *JSONResult*. O *Spring MVC* não vem com suporte nativo para trabalhar com objetos *JSON*, sendo necessária a biblioteca *Jackson Databind* e a configuração de anotações.

A *view engine* *JSTL* utiliza uma sintaxe parecida com o *HTML*, se integrando melhor ao código da página, enquanto o *Razor* tem a vantagem da possibilidade de ser fortemente tipado e utilizar as vantagens do *C#* como a função de auto completar código do *Visual Studio*. Em ambas as tecnologias o desenvolvedor pode estender as *view engines*, em projetos *Java* novas *taglibs* podem ser criadas e no *ASP.NET MVC* podem ser feitas *View Helpers* e *Partial Views*. Um vantagem do *ASP.NET MVC* em relação ao *Java Enterprise Endition* em se tratando de *views* é o suporte nativo à *layouts* (também conhecidas como *master pages*). Caso o desenvolvedor *Java* queria usar *layouts*, deve recorrer às bibliotecas de terceiros como o *Sitemesh* ou utilizar outras *view engines* como o *FreeMarker*. E aqui encontramos a maior

vantagem do *Java Enterprise Edition* em relação à *views*, o desenvolvedor pode escolher simplesmente utilizar outras *view engines*.

No próximo capitulo são abordados a criação de classes de negócio/serviços e injeção de dependências.

5 Classes de serviços e injeção de dependências

Uma aplicação de três camadas típica se divide em camadas de apresentação, negócios/serviços e persistência de dados. A camada de serviços é onde se escreve o código que representa regras de negócio das aplicações, enquanto a camada de persistência é responsável por guardar os dados da aplicação para uso posterior. É importante que essas camadas estejam fracamente acopladas, pois o desenvolvedor pode, por exemplo, aproveitar as classes de negócio que ele escreveu para uma aplicação *web* em uma aplicação para celular.

Nos exemplos desse trabalho, a camada de apresentação é formada pelas páginas *web* escritas no capítulo anterior, os *controllers* funcionam como uma ponte entre a camada de apresentação e de negócios. *Controllers* devem apenas receber e responder requisições, o ideal é que não exista código de regras de negócios em suas ações.

Nesse capítulo é demonstrado como escrever classes de serviço (também chamadas de classes de negócio) fracamente acopladas à camada de apresentação e aos *controllers*.

5.1 Classes de serviços

É criada como exemplo uma classe de serviço em cada projeto que contêm um método com uma regra de negócio simples. As listas de objetos *Person*, até agora pertencente aos *controllers*, são movidas para essas classes como uma forma de representar a camada de persistência. Em ambos os projetos, o serviço implementa uma *interface*. Diferente do que é sugerido para *controllers*, que um *controller* não deve chamar métodos de outro, classes de serviço podem utilizar outras classes de serviço.

5.1.1 Java

No projeto *Java*, a *interface* e classe concreta são criadas no pacote *br.uece.webCrud.service*. Primeiro é feita a interface *PersonService* que o serviço irá implementar, como podemos observar no quadro 22.

```
package br.uece.webCrud.service;
import br.uece.webCrud.model.Person;

public interface PersonService {
        public Person add(Person p);
        public List<Person> getAll();
}
```

Código 22: *Interface PersonService*

No quadro 23 pode-se observar o serviço *PersonServiceImpl* implementando a *interface PersonService*.

```
package br.uece.webCrud.service;
import java.util.ArrayList;
import java.util.List;
import org.springframework.stereotype.Service;
import br.uece.webCrud.model.Person;

@Service
public class PersonServiceImpl implements PersonService {

        private List<Person> persons;

        public PersonServiceImpl() {
                persons = new ArrayList<Person>();
        }

        public void add(Person p) {
                if(persons.stream().anyMatch(pp -> pp.getName().equals(p.
                    getName()))) {
                        throw new RuntimeException("Person already exists."
                            );
                }

                persons.add(p);
        }

        public List<Person> getAll() {
                return persons;
        }
}
```

Código 23: Classe *PersonServiceImpl*

A classe *PersonServiceImpl* começa decorada com a anotação *@Service*. Essa anotação é usada pelo *Spring Framework* para marcar classes da camada de serviços. A anotação *@Service* herda da anotação *@Component*, então essa é uma classe que pode ser usada pelo *Spring IoC Container* durante injeção de dependências. Mais detalhes sobre injeção de dependências e o modo como o *Spring* injeta uma instância dessa classe serão explicados nas seções 5.2 e 5.2.1. Segundo a documentação do *Spring Framework 4.1.4*, a anotação *@Service* tem o mesmo efeito que *@Component*, mas é recomendado o uso de *@Service* pois em versões futuras do *framework*, essa anotação poderá conter funcionalidades adicionais.

O construtor de *PersonServiceImpl* inicializa a lista de objetos *Person*. O método *add* recebe um objeto do tipo *Person* como parâmetro. Ele verifica se já existe um objeto com o mesmo nome na lista, se sim, dispara uma exceção, se não, adiciona o objeto à lista. Essa verificação é um exemplo simples de uma regra de negócios.

5.1.2 .NET

As classes de serviços no projeto *ASP.NET MVC* são criadas dentro da pasta *Models/Services*. Para o *ASP.NET MVC* a camada *model* de uma aplicação *MVC* contém tanto entidades quanto regras de negócios. Assim como no projeto *Java*, primeiro é feita a *interface IPersonService* mostrada no quadro 24.

```
namespace NetWebCrud.Models.Services
{
    public interface IPersonService
    {
            void add(Person p);
            IList<Person> GetAll();
    }
}
```

Código 24: *Interface IPersonService*

O quadro 25 mostra a implementação da *interface* pela classe *PersonService*.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace NetWebCrud.Models.Services
{
    public class PersonService : IPersonService
    {
        private IList<Person> persons
        {
            get
            {
                var persons = HttpContext.Current.Application["persons"];
                if (persons == null)
                {
                    HttpContext.Current.Application["persons"] = new List<
                        Person>();
                }
                return (IList<Person>)persons;
            }
        }

        public void add(Person p)
        {
            if (persons.Any(per => per.Name.Equals(p.Name)))
            {
                throw new Exception("Person already exists.");
            }

            persons.Add(p);
        }

        public IList<Person> GetAll()
```

```
        {
            return persons;
        }
    }
}
```

Código 25: Classe *PersonService*

A classe *PersonService* no projeto *ASP.NET MVC* é uma classe comum para onde foi movida a lista de objetos *Person* e contém um método idêntico ao do projeto *Java*. Nenhuma anotação ou configuração especial é feita na classe para identificá-la como serviço.

5.2 Injeção de dependências

Injeção de dependências é uma técnica utilizada para diminuir o acoplamento entre classes. Nos exemplos dessa seção, as classes de serviço são injetadas nos *controllers* criados no capitulo anterior.

Considere o exemplo do quadro 26.

```
package br.uece.webCrud.examples;

public interface Bar {
        public void sayHello();
}

public class BarImpl implements Bar {
        public void sayHello() {
                System.out.println("Hello World");
        }
}
public class Foo {
        private Bar bar;
        public Foo() {
                bar = new BarImpl();
        }

        public Bar getBar() {
                return bar;
        }
}
public class App {
        public static void main(String[] args) {
                Foo foo = new Foo();
                foo.getBar().sayHello();
        }
}
```

Código 26: Exemplo de classes fortemente acopladas

No exemplo acima, a classe *Foo* usa uma classe chamada *BarImpl* que implementa uma *interface* chamada *Bar*. Nesse exemplo a classe *Foo* fica fortemente acoplada à implementação *BarImpl*. Imagine se a classe *BarImpl* fosse uma classe de persistência que salva informações em arquivos de texto e um dia o desenvolvedor precisasse portar a aplicação para mais um ambiente onde seria usado um banco de dados. Sem usar injeção de dependências, o desenvolvedor também teria que escrever uma nova versão da classe *Foo* que usasse outra implementação de *Bar*.

O exemplo do quadro 27 mostra como funciona a injeção de dependências.

```
public class Foo {
        private Bar bar;
        public Foo(Bar bar) {
                bar = bar;
        }
        public Bar getBar() {
                return bar;
        }
}
public class App {
        public static void main(String[] args) {
                Bar bar = new BarImpl();
                Foo foo = new Foo(bar);
                foo.getBar().sayHello();
        }
}
```

Código 27: Exemplo de classes fracamente acopladas

No exemplo acima, a classe *Foo* agora recebe em seu construtor um abjeto que implemente *Bar*. Uma instância de *BarImpl* é criada dentro do método *main* e injetada dentro da instância de *Foo*. Nesse cenário, as classes não estão tão acopladas e o desenvolvedor pode usar outras implementações de *Bar* mais facilmente. A injeção de dependências também é conhecida como inversão de controle (*Inversion of Control* ou *IoC*), pois dá a responsabilidade de inicializar dependências de uma classe para classes externas.

Essa técnica ainda pode ser melhorada se a injeção de dependências for automatizada, requerendo menos código. É isso que o *IoC Container* do *Spring Framework* e o *Ninject* fazem. O uso dessas ferramentas será demonstrado nas seções seguintes.

5.2.1 Java

O *Spring IoC container* resolve dependências procurando por classes para serem injetadas dentro dos pacotes da aplicação de modo automático. Na seção 3.1.3., a classe de configuração *SpringMvcConfig* foi decorada com a anotação *@ComponentScan*. Os nomes dos pacotes passados como parâmetro para essa anotação são os pacotes onde o *Spring Framework* procura por classes para serem instanciadas e injetadas.

Para serem utilizadas pelo *Spring IoC container*, as classes devem estar decoradas com a anotação *@Component* ou alguma de suas especializações, como *@Service*, *@Repository* e *@Controller* ou métodos decorados com a anotação *@Bean*. Por padrão o *Spring IoC container* injeta as classes como *singletons*, criando apenas uma instância da classe para ser usada várias vezes. Esse comportamento pode ser alterado pela anotação *@Scope* nas classes que se deseja injetar.

A diferença entre as anotações *@Component* e *@Bean* (também utilizada na seção 3.1.3) é que *@Component* é usado em classes que o próprio *Spring IoC container* procura e instancia enquanto *@Bean* é usado em métodos que retornam objetos que o próprio desenvolvedor criou. Classes anotadas com *@Component* ou uma de suas especializações devem ter um construtor que não receba parâmetros.

O código do quadro 28 mostra uma nova versão do *controller PersonController* do projeto *Java/Spring*, agora com uma instância de *PersonServiceImpl* sendo injetada e utilizada.

```
package br.uece.webCrud.controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import br.uece.webCrud.model.Person;
import br.uece.webCrud.service.PersonService;

@Controller
@RequestMapping("/person")
public class PersonController {

        @Autowired
        private PersonService personService;

        @RequestMapping("/list")
        public ModelAndView listPage()
        {
                ModelAndView result = new ModelAndView("person/list");
                result.addObject("personsList", personService.getAll());
                return result;
        }

        @RequestMapping(value = "/add", method = RequestMethod.GET)
        public String addPage()
        {
                return "person/add";
        }

        @RequestMapping(value = "/add", method = RequestMethod.POST)
        public String addPost(@ModelAttribute Person person)
        {
                personService.add(person);
                return "redirect:/person/list";
        }
```

```
}
```

Código 28: *PersonController* utilizando *PersonService* no projeto *Java/Spring*

No lugar da lista em memória, agora o *controller* possui e utiliza um *PersonService* que está decorado com a anotação *@Autowired*. Essa anotação marca propriedades onde o *Spring IoC container* deve injetar dependências. No caso acima, o *container* procura por classes decoradas com *@Component* ou suas especializações, que implementam *PersonService* e estão contidas nos pacotes configurados por *@ComponentScan*. Encontrada a classe *PersonServiceImpl*, uma instância dessa classe é criada e atribuída à propriedade *personService* para uso no *PersonController*.

Essa não é a única maneira de injetar dependências usando o *Spring*. Para conhecer mais opções de injeção de dependências, aconselha-se consultar sua documentação.

5.2.2 .NET

O *ASP.NET MVC 5* não possui injeção de dependências automática nativa, então é usado a biblioteca *Ninject* para essa funcionalidade. Diferente do *Spring*, o *Ninject* não procura por classes de modo automático para injetar dependências, as classes que o desenvolvedor deseja injetar devem ser configuradas por código. Na seção 3.2.1, o *Ninject MVC 5* foi adicionado ao projeto e uma classe chamada *NinjectWebCommon* (exibida no quadro 29) foi criada na pasta *App_Start*.

```
namespace NetWebCrud.App_Start
{
    using System;
    using System.Web;
    using Microsoft.Web.Infrastructure.DynamicModuleHelper;
    using Ninject;
    using Ninject.Web.Common;
    using NetWebCrud.Models.Services;

    public static class NinjectWebCommon
    {
        private static readonly Bootstrapper bootstrapper = new
            Bootstrapper();

        public static void Start()
        {
            DynamicModuleUtility.RegisterModule(typeof(
                OnePerRequestHttpModule));
            DynamicModuleUtility.RegisterModule(typeof(NinjectHttpModule));
            bootstrapper.Initialize(CreateKernel);
        }

        public static void Stop()
        {
            bootstrapper.ShutDown();
        }
```

```
        private static IKernel CreateKernel()
        {
            var kernel = new StandardKernel();
            try
            {
                kernel.Bind<Func<IKernel>>().ToMethod(ctx => () => new
                                                                    Bootstrapper
                                                                        ()
                                                                        .
                                                                        Kernel
                                                                        )
                                                                        ;

                kernel.Bind<IHttpModule>().To<
                    HttpApplicationInitializationHttpModule>();

                RegisterServices(kernel);
                return kernel;
            }
            catch
            {
                kernel.Dispose();
                throw;
            }
        }

        private static void RegisterServices(IKernel kernel)
        {
        }
    }
}
```

Código 29: Classe *NinjectWebCommon*

O método *Start* é executado quando a aplicação é inicializada, ele registra módulos do *Ninjet* para uso interno da aplicação, cria e inicializa um *kernel* do *Ninject*. O *kernel* é o objeto onde é feita a configuração de injeções de dependência, essa configuração é feita no método *RegisterServices* como mostra o quadro 30.

```
private static void RegisterServices(IKernel kernel)
{
        kernel.Bind<IPersonService>().To<PersonService>().InRequestScope();
}
```

Código 30: Registro de dependências no *Ninject*

O *kernel* do *Ninject* pode usar uma sintaxe fluente para configurar dependências. Aqui usa-se o método genérico *Bind* recebendo a interface *IPersonService* seguido do método *To* configurando a classe *PersonService* para ser a classe concreta injetada quando o *Ninject* encontra aquela interface. Por último, o método *InRequestScope* cria uma instancia de *PersonService* para cada requisição do usuário ao servidor. Existe o método *InSingletonScope* que pode criar apenas uma instancia da classe para todos

os usuários e todas as requisições, mas a necessidade de se utilizar *InRequestScope* é abordada na seção 6.3.2.

O *Ninject* é uma biblioteca extensa que contém módulos que funcionam em vários tipos de projetos *.NET*. Para mais informações aconselha-se acessar a página do *Ninject* (`http://www.ninject.org/`).

O *controller* *PersonController* é modificado para utilizar o serviço *PersonService*, como pode ser observado no quadro 31.

```
namespace NetWebCrud.Controllers
{
    public class PersonController : Controller
    {
        [Inject]
        public IPersonService PersonService { set; private get; }

        [HttpGet]
        public ActionResult Add()
        {
            return View();
        }

        [HttpPost]
        public ActionResult Add(Person person)
        {
            PersonService.Add(person);
            return RedirectToAction("List");
        }

        [HttpGet]
        public ActionResult List()
        {
            return View(PersonService.GetAll());
        }
    }
}
```

Código 31: *PersonController* do projeto *ASP.NET MVC* usando *PersonService*

A anotação *Inject* decora a propriedade *PersonService* onde é injetada a instância de classe *PersonService*, do mesmo modo que *@Autowired* no projeto *Java/Spring*. A propriedade deve ter um método *set* público para que o *Ninject* possa ter acesso e injetar a classe concreta, o método *get* pode ser mais restrito.

5.3 Conclusão

O *Spring* possui a anotação *@Service* para marcar classes de serviço e classes de negócio, enquanto no *ASP.NET MVC* qualquer classe pode ser uma classe de serviço. Atualmente nenhuma das tecnologias mostra vantagem na criação de serviços mas no futuro, talvez sejam adicionadas funcionalidades para a anotação *@Service*

que possam beneficiar o *Spring Framework*.

O *Spring* é superior ao *ASP.NET MVC 5* em se tratando de injeção de dependências. Além de ter um *IoC container* nativo, o *Spring* resolve dependências automaticamente enquanto o *ASP.NET MVC 5* precisa de bibliotecas de terceiros, e o *Ninject* precisa de configurações via código para todas as classes que são injetadas.

Na próxima versão do *ASP.NET*, também chamada de *vNext*, será adicionado um *IoC Container* nativo. Até o momento em que este trabalho está sendo escrito, o *vNext* está em fase de testes público e pode passar por mudanças, então o uso do *IoC Container* da próxima versão não foi considerado.

No próximo capítulo é abordada a criação do banco de dados a partir de entidades de negócio e acesso a dados usando repositórios do *Spring* e a classe *DbContext* do *Entity Framework*.

6 Entidades e repositórios

Entidades são classes que representam os objetos de negócio, a classe *Person* utilizada até agora é um exemplo de entidade. Em ambos os projetos, essa classe é usada para gerar uma tabela no banco de dados e substituir as listas em memória que foram usadas até então. Essa técnica é chamada de mapeamento objeto/relacional, pois mapeia propriedades de objetos para tabelas e colunas no banco de dados relacional. Nos exemplos desse capitulo são usados o *Hibernate* no projeto *Java/Spring* e o *Entity Framework* no projeto *ASP.NET MVC* como bibliotecas para mapeamento objeto/relacional.

Além de a classe *Person*, nesse capitulo é criada a classe *Contact* que conterá o *e-mail* e o telefone de uma pessoa. Essa classe terá um relacionamento de um para um com a classe *Person* e irá gerar uma chave estrangeira no banco de dados.

Com as entidades prontas, são feitos repositórios para persisti-las no banco de dados. Será feito um repositório genérico e um repositório especifico para cada classe.

6.1 Entidades

As seções a seguir mostram como utilizar uma classe para gerar tabelas no banco de dados. Entidades precisam de um identificador único para ser sua chave primária, essa propriedade será chamada *Id* e será do tipo inteiro.

Como são usadas duas entidades nesse capitulo e ambas precisam do identificador único, é criada uma classe base com essa propriedade em comum chamada *BaseEntity*. O intuito da criação dessa classe é mostrar como é possível o uso de herança nas entidades. As classes *Person* e *Contact* são especializações de *BaseEntity*.

6.1.1 Java

Entidades no projeto *Java/Spring* são criadas no pacote *br.uece.webCrud.model*. A classe *BaseEntity* é criada como mostra o quadro 32.

```
package br.uece.webCrud.model;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.MappedSuperclass;

@MappedSuperclass
public abstract class BaseEntity {

        @Id
        @GeneratedValue
        private int id;

        public int getId() {
```

```
                return id;
        }

        public void setId(int id) {
                this.id = id;
        }
}
```

Código 32: Classe *BaseEntity* no projeto *Java*

A classe *BaseEntity* é decorada com a anotação *@MappedSuperclass* que diz ao *Hibernate* que entidades que sejam suas subclasses devem levar em consideração as propriedades herdadas no mapeamento objeto/relacional. A propriedade *id* é decorada com as anotações *@Id*, que configura o *Hibernate* para usá-la como chave primária e única, e *@GeneratedValue*, que delega ao banco de dados a lógica de geração do valor da chave.

A classe *Person* então é modificada como no quadro 33.

```
package br.uece.webCrud.model;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import org.springframework.format.annotation.DateTimeFormat;

@Entity
public class Person extends BaseEntity {
        @Column(nullable = false)
        private String name;

        @Temporal(TemporalType.DATE)
        @DateTimeFormat(pattern = "yyyy-MM-dd")
        private Date birthDate;

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public Date getBirthDate() {
                return birthDate;
        }

        public void setBirthDate(Date birthDate) {
                this.birthDate = birthDate;
        }
}
```

Código 33: Classe *Person* herda de *BaseEntity*

A classe *Person* agora é subclasse de *BaseEntity* e está decorada com a anotação *@Entity*. A anotação *@Entity* diz ao *Hibernate* que essa classe deve ser mapeada para uma tabela e as propriedades da classe para as colunas da tabela.

Por padrão o *Hibernate* procura por uma tabela com o mesmo nome da classe e colunas com o mesmo nome das propriedades para realizar o mapeamento, mas esse comportamento pode ser modificado com as anotações *@Table*, *@Column*, *@Transient* e muitas outras. A propriedade *name* por exemplo, está decorada com a anotação *@Column* especificando que aquela propriedade não pode ser nula no banco de dados e a propriedade *birthDate* está decorada com a anotação *@Temporal* especificando que a coluna no banco deve ser do tipo *Date*.

Na seção 3.1.3, durante a configuração do projeto, a propriedade *hibernate.hbm2ddl.auto* foi configurada com o valor *"create"*, assim o *Hibernate* cria as tabelas no banco de dados de acordo com as configurações de classes decoradas com a anotação *@Entity*. Outros valores para essa configuração podem ser *"update"*, para atualizar a estrutura das tabelas sem perder dados, e *"validate"*, para apenas verificar se o mapeamento objeto/relacional está consistente. O *Hibernate* cria tabelas, mas não cria o banco de dados, para que a criação de tabelas funcione é necessário que o banco de dados já exista. As tabelas são criadas no momento em que a aplicação é iniciada.

6.1.2 .NET

Entidades no projeto ASP.NET MVC são criadas na pasta *Model/Entities*. Considere que a classe *Person* é movida para essa pasta. O quadro 34 mostra a classe *BaseEntity* do projeto *ASP.NET MVC*.

```
namespace NetWebCrud.Models.Entities
{
    public class BaseEntity
    {
        public int Id { set; get; }
    }
}
```

Código 34: Classe *BaseEntity* no projeto *ASP.NET*

Uma convenção do *Entity Framework* é que se uma classe tiver uma propriedade chamada *Id* de tipo numérico ou *guid* (tipo de identificar único), essa propriedade será usada como chave primária. Assim nenhuma configuração adicional é necessária.

Assim como no projeto *Java*, a classe *Person* do projeto *ASP.NET MVC* é modificada para ser subclasse de *BaseEntity*, como mostrado no quadro 35.

```
using System.ComponentModel.DataAnnotations;

namespace NetWebCrud.Models.Entities
{
```

```
    public class Person : BaseEntity
    {
        [Required]
        public string Name { set; get; }
        public DateTime BirthDate { set; get; }
    }
}
```

Código 35: Classe *Person* no projeto *ASP.NET* agora herda de *BaseEntity*

Assim como no projeto *Java*, entidades do *Entity Framework* podem ser decoradas com anotações que controlam como as tabelas serão geradas. No exemplo acima a anotação *Required* também diz que a coluna correspondente à propriedade *Name* não pode aceitar valores nulos.

6.1.2.1 Criando o contexto do banco de dados Não existe configuração nas classe do projeto *ASP.NET MVC* que indique que elas são entidades e que devem ser usadas no mapeamento objeto/relacional. Esse papel é da classe *DbContext* do *Entity Framework*. A classe *DbContext* representa o banco de dados como um todo. Para criar o banco de dados a partir das entidades, o desenvolvedor deve criar uma classe que herde de *DbContext* e tenha propriedades do tipo *DbSet* como a classe *NetWebCrudContext* no exemplo do quadro 36.

```
using NetWebCrud.Models.Entities;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;
using System.Linq;
using System.Web;

namespace NetWebCrud.Models
{
    public class NetWebCrudContext : DbContext
    {
        public NetWebCrudContext() : base("NetWebCrudContext")
        {
            Database.SetInitializer<NetWebCrudContext>(new
                                DropCreateDatabaseIfModelChanges<
                                    NetWebCrudContext>());
        }

        public DbSet<Person> Persons { set; get; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder
            )
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention
                >();
        }
    }
```

```
}
```

Código 36: Classe *NetWebCrudContext* representa o contexto do banco de dados

A classe *NetWebCrudContext* possui um *DbSet* de objetos do tipo *Person* que representa a tabela que é gerada no banco de dados. Para utilizar o banco, o desenvolvedor deve criar uma instância do contexto e fazer consultas aos *DbSets* usando o Language Integrated Query (*LINQ*). O *LINQ* é uma linguagem para realizar consultas em coleções do *.NET Framework*. Usado em *DbSets*, ele gera consultas *SQL* automaticamente para manipular o banco de dados. Consultas com *LINQ* são abordadas na seção 6.4.

O construtor de *NetWebCrudContext* executa o construtor de *DbContext* recebendo como parâmetro o nome da *connection string* que contém informações de conexão com o banco. Ele recebe o nome da *connection string* *NetWebCrudContext*, a mesma que foi configurada no arquivo *Web.xml* na seção 3.2.2.

O construtor do contexto do banco de dados também é usado para configurar o tipo de inicialização do banco. O construtor de *NetWebCrudContext* utiliza como inicializador a classe *DropCreateDatabaseIfModelChanges*, então se qualquer entidade for modificada, o banco de dados é totalmente reconstruído e todos os dados perdidos. Existem outras classes que podem ser usadas como inicializadores como *CreateDatabaseIfNotExists*, para criar o banco apenas uma vez, e *DropCreateDatabaseAways* para sempre recriar o banco de dados quando a aplicação iniciar. O desenvolvedor pode estender esses inicializadores ou construir um completamente novo que implemente a interface *IDatabaseInitializer*.

O *Entity Framework* não atualiza automaticamente a estrutura do banco de dados sem perder informações, para isso é necessário utilizar o *Entity Framework Migrations*. Essa funcionalidade permite utilizar as linguagens *C#* ou *VB.NET* para executar *scripts* no banco de dados e atualizar sua estrutura. O *Entity Framework Migrations* não é abordado nesse trabalho, para mais informações recomenda-se consultar a documentação do *Entity Framework*.

O método *OnModelCreating* é usado para configurar diversos aspectos da criação do banco de dados. No exemplo do quadro 36, a convenção de pluralizar o nome das tabelas é removida.

Diferente do *Hibernate* que cria ou atualiza o banco de dados quando a aplicação é iniciada, o *Entity Framework* cria o banco de dados na primeira vez em que uma instancia de *DbContext* é criada.

Uma instancia de *NetWebCrudContext* precisa ser instanciada para que se possa usar o banco de dados. No projeto exemplo, cada requisição de um usuário deve criar uma nova instancia de *NetWebCrudContext*. Se mais de um usuário utilizar a mesma instancia, poderão haver erros de concorrência e um usuário poderá ter acesso os dados de outro que ainda não foram salvos para o banco de dados. O *Ninject* é responsável por criar as instancias de *NetWebCrudContext* e injeta-las nos repositórios que serão criados na seção 6.3.2. O quadro 37 mostra a configuração de injeção de

NetWebCrudContext.

```
private static void RegisterServices(IKernel kernel)
{
        kernel.Bind<IPersonService>().To<PersonService>().InRequestScope();
        kernel.Bind<DbContext>().ToSelf().InRequestScope();
}
```

Código 37: Injeção do contexto do banco de dados

6.2 Relacionamentos entre entidades

Entidades podem se relacionar com outras com o uso de chaves estrangeiras. O relacionamento entre a classe *Person* e a classe *Contact* é usado nos exemplos dessa seção.

6.2.1 Java

A classe *Contact* é criada no pacote *br.uece.webCrud.model* como mostra o quadro 38.

```
package br.uece.webCrud.model;
import javax.persistence.Entity;
import javax.persistence.OneToOne;

@Entity
public class Contact extends BaseEntity {
        private String phone;

        private String email;

        @OneToOne(mappedBy = "contact")
        private Person person;

        public String getPhone() {
                return phone;
        }

        public void setPhone(String phone) {
                this.phone = phone;
        }

        public String getEmail() {
                return email;
        }

        public void setEmail(String email) {
                this.email = email;
        }
```

```
        public Person getPerson() {
                return person;
        }

        public void setPerson(Person person) {
                this.person = person;
        }
}
```

Código 38: Classe Contact no projeto Java

Assim como a classe *Person*, a classe *Contact* herda de *BaseEntity* e é decorada com a anotação *@Entity*. O Relacionamento com a classe *Person* é configurado pela anotação *@OneToOne* decorando o atributo *person*. O atributo *mappedBy* recebendo o valor *"contact"* quer dizer que na classe *Person* deve existir um atributo com esse nome, configurado como o dono do relacionamento entre as tabelas. Resumindo, a tabela correspondente à classe *Person* deve ter a coluna com a chave estrangeira para *Contact*.

Além da anotação *@OneToOne*, o *JPA* possui as anotações *@OneToMany*, *@ManyToOne* e *@ManyToMany* para criar relacionamentos. As anotações *@OneToMany* e *@ManyToMany* devem ser usadas em uma coleção de objetos, quando uma entidade tem um relacionamento de "um para muitos"ou de "muitos para muitos"com outra entidade.

O relacionamento na classe *Person* também deve ser configurado como mostra o quadro 39.

```
package br.uece.webCrud.model;
import java.util.Date;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import org.springframework.format.annotation.DateTimeFormat;

@Entity
public class Person extends BaseEntity {
        @Column(nullable = false)
        private String name;

        @Temporal(TemporalType.DATE)
        @DateTimeFormat(pattern = "yyyy-MM-dd")
        private Date birthDate;

        @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
        @JoinColumn(name = "contactId")
        private Contact contact;
```

```
        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public Date getBirthDate() {
                return birthDate;
        }

        public void setBirthDate(Date birthDate) {
                this.birthDate = birthDate;
        }

        public Contact getContact() {
                return contact;
        }

        public void setContact(Contact contact) {
                this.contact = contact;
        }
}
```

Código 39: Classe *Person* com relacionamento para *Contact*

A classe *Person* agora possui uma propriedade do tipo *Contact*, também decorada com a anotação *@OneToOne* e com a anotação *@JoinColumn*. A primeira anotação é configurada para cascatear todas as operações de *Person* para *Contact*. Se por exemplo, um objeto do tipo *Person* for atualizado no banco de dados, a sua propriedade *contact* também será atualizada. A configuração *fetch* recebe como valor *FetchType.EAGER*, configurando a propriedade *contact* para ser recuperada do banco de dados por *eager loading* (executando uma única consulta). A propriedade *fetch* também pode receber *FetchType.LAZY*, fazendo com que *contact* só seja carregado quando necessário, mas gerando uma nova consulta ao banco de dados.

A anotação *@JoinColumn* configura a classe *Person* para criar a coluna da chave estrangeira na sua tabela correspondente, essa coluna tem o nome de *contactId*. O *Hibernate* detecta automaticamente as chaves decoradas com *@Id* e as utiliza como chaves estrangeiras em tabelas relacionadas. A Figura 36 mostra o resultado de criação das tabelas no banco de dados *MySQL*.

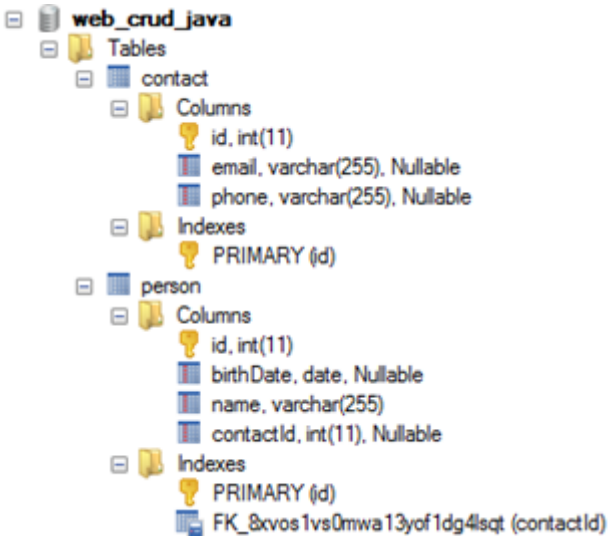

Figura 36: Tabelas geradas pelo *Hibernate*

6.2.2 .NET

A classe *Contact*, ilustrada no quadro 40, também é criada no projeto *ASP.NET MVC*.

```
namespace NetWebCrud.Models.Entities
{
    public class Contact : BaseEntity
    {
        public String Phone { set; get; }
        public String Email { set; get; }
    }
}
```

Código 40: Classe *Contact* no projeto *ASP.NET MVC*

A classe *Contact* não requer nenhuma configuração, sendo apenas preciso herdar de *BaseEntity* para adquirir a propriedade *Id*. Por outro lado, a classe *Person* recebe mais propriedades como mostra o quadro 41.

```
namespace NetWebCrud.Models.Entities
{
    public class Person : BaseEntity
    {
        [Required]
        public string Name { set; get; }
        public DateTime BirthDate { set; get; }
        public int? ContactId { set; get; }

        public virtual Contact Contact { set; get; }
    }
}
```

Código 41: Classe *Person* no projeto *ASP.NET* com novas propriedades

Na classe *Person* é adicionada uma propriedade *Contact* modificada com a palavra chave *"virtual"* para configurá-la como uma propriedade de navegação que pode ser carregada por *lazy loading*. Isso resulta em mais uma consulta no banco, mas com o uso do *LINQ* é possível recuperar as informações de *Person* e *Contact* com apenas uma consulta.

A propriedade *ContactId* gera a coluna da chave estrangeira. Por convenção do *Entity Framework*, propriedades de chaves estrangeiras devem ter o nome da propriedade de navegação (*Contact*) seguido do nome da sua chave primária (*Id*).

Por último, é necessário adicionar um *DbSet* para *Contact* no contexto do banco de dados, como mostra o quadro 42.

```
using NetWebCrud.Models.Entities;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;
using System.Linq;
using System.Web;

namespace NetWebCrud.Models.Context
{
    public class NetWebCrudContext : DbContext
    {
        public NetWebCrudContext() : base("NetWebCrudContext")
        {
            Database.SetInitializer<NetWebCrudContext>(new
                                DropCreateDatabaseIfModelChanges<
                                    NetWebCrudContext>());
        }

        public DbSet<Person> Persons { set; get; }
        public DbSet<Contact> Contacts { set; get; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder
            )
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention
                >();
        }
    }
}
```

Código 42: Classe *NetWebCrudContext* com o *DbSet* para *Contact*

O resultado no banco é semelhante ao do projeto *Java/Spring*, como mostra a Figura 37. Uma diferença que deve ser ressaltada é a criação da tabela *__migrationhistory*. Essa tabela é usada pelo *Entity Framework* para armazenar dados sobre o histórico da estrutura do banco de dados. Assim o *Entity Framework Migrations* sabe quais modificações devem ser feitas no banco quando as entidades são modificadas.

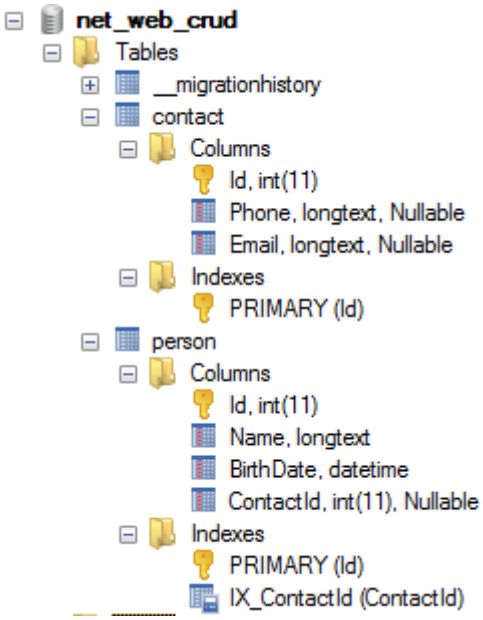

Figura 37: Tabelas geradas pelo *Entity Framework*

6.3 Repositórios

A persistência e acesso aos dados são feitos utilizando o padrão de repositórios. Em cada projeto é criado um repositório genérico, que pode realizar operações básicas (salvar, atualizar, recuperar e apagar) em qualquer classe que seja subclasse de *BaseEntity*. Um repositório especifico para a entidade *Person* é criado nos dois projetos e outro para *Contact* apenas no projeto *ASP.NET MVC*. Eles são especializações do repositório genérico e possuem operações especificas para essas entidades.

É demonstrado como utilizar a *Java Persistence Query Language* (*JPQL*) e a *Language Integrated Query* (*LINQ*) para realizar consultas ao banco de dados e como gerenciar transações. Por último, os repositórios são injetados nas classes de serviço, substituindo as listas em memória.

6.3.1 Java

Nos exemplos dessa seção, os repositórios e interfaces que eles implementam no projeto *Java/Spring* são criados no pacote *br.uece.webCrud.repositories*.

Primeiramente, são criadas a interface *GenericRepository* e a classe *GenericRepositoryImpl*, demonstradas nos quadros 43 e 44 respectivamente. Essa classe é o repositório genérico mencionado anteriormente.

```
package br.uece.webCrud.repository;
import java.util.List;
import br.uece.webCrud.model.BaseEntity;

public interface GenericRepository<T extends BaseEntity> {
        public List<T> getAll();
```

```
        public T getById(int id);
        public T add(T entity);
        public T update(T entity);
        public void delete(int id);
}
```

Código 43: Interface GenericRepository

```
package br.uece.webCrud.repository;
import java.lang.reflect.ParameterizedType;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import org.springframework.stereotype.Repository;
import br.uece.webCrud.model.BaseEntity;

@Repository
public abstract class GenericRepositoryImpl<T extends BaseEntity>
    implements GenericRepository<T> {

        @PersistenceContext
        protected EntityManager em;

        protected Class<T> t;

        public GenericRepositoryImpl() {
                this.t = ((Class) ((ParameterizedType) getClass()
                                .getGenericSuperclass()).
                                    getActualTypeArguments()[0]);
        }

        @Override
        public List<T> getAll() {
                String typeString = t.getSimpleName();
                Query q = em.createQuery("SELECT e FROM " + typeString + "
                    e");
                return q.getResultList();
        }

        @Override
        public T getById(int id) {
                return em.find(t, id);
        }

        @Override
        public T add(T entity) {
                em.persist(entity);
                return entity;
        }

        @Override
        public T update(T entity) {
                em.merge(entity);
                return entity;
        }
```

```
        @Override
        public void delete(int id) {
                em.remove(getById(id));
        }
}
```

Código 44: Classe GenericRepositoryImpl

A classe *GenericRepositoryImpl* é decorada com a anotação *@Repository*, então instâncias dela podem ser injetadas pelo *Spring IoC Container*. Erros de seus métodos em tempo de execução, dispararam a exceção *DataAccessException* do *Spring Framework*.

Essa classe faz uso de *Java Generics*, podendo realizar suas operações com qualquer classe que herde de *BaseEntity*. Seu construtor de extrai o tipo da classe com a qual ele está trabalhando a utiliza em suas operações. *GenericRepositoryImpl* é uma classe abstrata (que não pode ser instanciada), quem efetivamente poderá executar suas operações serão os repositórios que herdarão dessa classe.

O objeto da classe que executa as operações no banco de dados é do tipo *EntityManager*, decorado com a anotação *@PersistenceContext*. A instancia de *EntityManager* é criada pelo *bean LocalContainerEntitityManagerFactoryBean*, configurado na seção 3.1.3, e injetado pelo *Spring IoC Container*. Para objetos de acesso a dados, a anotação *@PersistenceContext* funciona do mesmo modo que *@Autowired*.

O método *getAll* retorna todas as entradas de uma determinada entidade armazenada no banco de dados. Ele mostra um exemplo de como criar uma consulta usando o método *createQuery* de *EntityManager*, recebendo o texto da consulta escrito em *JPQL*. Essa linguagem possui similaridades com *SQL*. O método *getResultList* executa a consulta criada, convertendo o texto *JPQL* em *SQL*, realizando a consulta no banco de dados, mapeando tuplas para objetos, e devolvendo o resultado como uma lista. Para mais informações sobre o *JPQL*, é recomendado consultar a documentação da linguagem no site `http://docs.oracle.com/cd/E15523_01/apirefs.1111/e13946/ejb3_langref.html`.

O método *findById* usa o método *find* do *EntityManager* para encontrar uma entidade com chave primária e do tipo que foram passados como parâmetros. O método *add* chama o método *persists* para inserir um novo objeto no banco de dados. Esse objeto não pode ter uma chave primária de valor já existente no banco, ou o *Hibernate* irá disparar uma exceção *PersistenceException*.

O método *merge*, chamado por *update*, realiza atualização dos valores de uma entidade no banco de dados. Para que ele funcione é necessário que a chave primária do objeto a ser atualizado exista, ou o *Hibernate* também disparará uma exceção. O método *delete*, usa *getById* para recuperar uma entidade pela sua chave primária e chama o método *remove* de *EntityManager* para remove-la do banco de dados.

Com o repositório genérico criado, é possível criar um repositório especifico para a classe *Person*, como mostram os quadros 45 e 46. A classe *PersonRepositoryImpl*

adiciona o método *existsWithName*, que verifica a existência de uma pessoa com um certo nome. Nesse método é criada uma consulta mais complexa, que retorna o valor booleano *true*, se existir uma entidade no banco de dados com a propriedade *name* igual a passada como parâmetro e propriedade *id* diferente da passada como parâmetro. Esse exemplo também mostra como adicionar parâmetros às consultas, utilizando método *setParameter*.

```
package br.uece.webCrud.repository;
import br.uece.webCrud.model.Person;

public interface PersonRepository extends GenericRepository<Person> {
        public boolean existsWithName(Person p);
}
```

Código 45: Interface PersonRepository

```
package br.uece.webCrud.repository;
import javax.persistence.TypedQuery;
import org.springframework.stereotype.Repository;
import br.uece.webCrud.model.Person;

@Repository
public class PersonRepositoryImpl extends GenericRepositoryImpl<Person>
    implements PersonRepository {
        @Override
        public boolean existsWithName(Person p) {
                TypedQuery<Boolean> query = em
                        .createQuery(
                                "SELECT CASE WHEN COUNT(*) > 0 THEN true
                                    ELSE false END " +
                                "FROM Person p WHERE p.name = :name AND p.
                                    id <> :id",
                                Boolean.class);
                query.setParameter("id", p.getId());
                query.setParameter("name", p.getName());
                return (boolean) query.getSingleResult();
        }
}
```

Código 46: Classe PersonRepositoryImpl

Por último, a lista em memória da classe *PersonServiceImpl* é substituída por *PersonRepository* e a anotação *@Transactional* é adicionada à classe, como mostra o quadro 47.

```
package br.uece.webCrud.service;
import java.util.List;
import javax.transaction.Transactional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import br.uece.webCrud.model.Person;
import br.uece.webCrud.repository.PersonRepository;
```

```
@Service
@Transactional
public class PersonServiceImpl implements PersonService {

        @Autowired
        private PersonRepository pr;

        @Override
        public List<Person> getAll() {
                return pr.getAll();
        }

        @Override
        public Person add(Person p) {
                if (pr.existsWithName(p)) {
                        throw new RuntimeException("Person already exists."
                            );
                }

                return pr.add(p);
        }

        @Override
        public Person getById(int id) {
                return pr.getById(id);
        }

        @Override
        public Person update(Person p) {
                return pr.update(p);
        }

        @Override
        public void delete(int id) {
                pr.delete(id);
        }
}
```

Código 47: *PersonServiceImpl* utilizando *PersonRepository*

A anotação *@Transactional* do *Spring Framework* pode ser utilizada em classes ou métodos. Essa anotação configura o gerenciamento de transações com o banco de dados. Decorando um método, essa anotação faz com que uma única transação seja usada em todas as operações de acesso a dados dentro daquele método. Decorando uma classe, como no exemplo do quadro 47, a anotação aplica o seu efeito a todos os métodos da classe. O *Spring* controla internamente a criação e reutilização de transações já existentes usando o *bean JpaTransactionManager* também configurado na seção 3.1.3. Se um método decorado com essa anotação for chamado internamente por outro também decorado com ela, o método interno irá aproveitar a transação que já existe e não irá criar uma nova. Esse comportamento padrão pode ser modificado configurando atributos da anotação.

Caso não ocorram erros nas operações com o banco de dados, as operações da

transação são persistidas de forma atômica. Caso haja algum erro, as operações são revertidas e uma exceção é disparada. De toda forma, a transação é destruída automaticamente quando não é mais utilizada.

Como o relacionamento de *Person* com *Contact* configurado com *CascadeType.ALL* na seção 6.2.1, quando um objeto do tipo *Person* é salvo ou atualizado no banco de dados, o seu contato também é. Então nos exemplos desse capitulo para *Java*, não houve a necessidade de se criar um repositório para a classe *Contact*.

6.4 .NET

Os repositórios no projeto *ASP.NET MVC* são criados na pasta *Models/Repositories*. Assim como no projeto *Java/Spring*, começa-se criando o repositório genérico e sua *interface*. O quadro 48 mostra a *interface* do repositório genérico, *IGenericRepository*, e o quadro 49 mostra sua implementação, *GenericRepository*.

```
using NetWebCrud.Models.Entities;
using System.Collections.Generic;
using System.Data.Entity;

namespace NetWebCrud.Models.Repositories
{
    public interface IGenericRepository<T> where T : BaseEntity
    {
        T GetById(int Id);
        IList<T> GetAll();
        T Save(T entity);
        T Update(T entity);
        void Delete(int Id);
        DbContextTransaction GetTransaction();
    }
}
```

Código 48: *Interface IGenericRepository*

```
using NetWebCrud.Models.Context;
using NetWebCrud.Models.Entities;
using Ninject;
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Web;

namespace NetWebCrud.Models.Repositories
{
    public abstract class GenericRepository<T> :
                                IGenericRepository<T> where T : BaseEntity
    {
        [Inject]
```

```
        public NetWebCrudContext context { set; protected get; }

        protected DbSet<T> EntitySet
        {
            get { return context.Set<T>(); }
        }

        public T GetById(int Id)
        {
            return EntitySet.Where<T>(e => e.Id == Id).SingleOrDefault();
        }

        public IList<T> GetAll()
        {
            return EntitySet.ToList<T>();
        }

        public T Save(T entity)
        {
            EntitySet.Add(entity);
            context.SaveChanges();
            return entity;
        }

        public T Update(T entity)
        {
            EntitySet.Attach(entity);
            context.Entry(entity).State = EntityState.Modified;
            context.SaveChanges();
            return entity;
        }

        public void Delete(int Id)
        {
            var entity = GetById(Id);
            if (entity != null)
            {
                EntitySet.Remove(entity);
                context.SaveChanges();
            }
        }

        public DbContextTransaction GetTransaction()
        {
            return context.Database.BeginTransaction();
        }
    }
}
```

Código 49: Classe *GenericRepository*

Assim como no projeto *Java/Spring*, o repositório genérico é uma classe abstrata que realiza operações no banco de dados para classes que herdam de *BaseEntity*. Também semelhante ao outro projeto, existe uma propriedade no repositório que efetivamente irá persistir dados e consultar o banco, a propriedade *context* do tipo *NetWebCrudContext*. A propriedade *EntitySet* é usada para facilitar o acesso ao

DbSet correto da entidade a qual o repositório genérico irá manipular.

O método *GetById* mostra um exemplo de como se usar expressões *lambda* e a sintaxe fluente do *LINQ* para realizar uma consulta. Métodos de filtragem e ordenação da sintaxe fluente do *LINQ* retornam coleções do tipo *IQueryable*, essa *interface* permite que mais métodos sejam chamados em cascata para a criação de consultas complexas. Uma consulta do *LINQ* só é realmente executada no bando de dados quando se chamam certo métodos, como *Single*, *First*, *ToList* e *ToArray*, e essas consultas também são transformadas em *scripts SQL* quando são executadas no banco de dados. No método *GetAll*, o método *ToList* do *DbSet* retorna todas as tuplas do banco de dados transformadas em uma lista de objetos.

Dentro do método *Save*, um objeto é adicionado ao *DbSet* usando o método *Add*, equivalente a uma operação de inserção na tabela. Operações de inserção, atualização e deleção de dados só são persistidas após a chamada do método *SaveChanges* do contexto. Esse método cria uma transação que em caso de erro reverte a transação e dispara uma exceção. É possível ter um controle melhor sobre transações, como ainda será visto nessa seção.

O método *Update* é um pouco mais complexo. Para persistir dados atualizados de uma entidade no banco de dados, caso a entidade não esteja anexada ao contexto, é necessário anexa-la e utilizar o contexto para modificar seu estado para *Modfied*. Assim, quando *SaveChanges* for chamado, o contexto saberá que aquela entidade é de um objeto que já existe no banco de dados (através da chave primária já existente) e irá atualizar suas informações.

O método *Delete* é bastante similar ao do repositório do projeto *Java/Spring*. Uma entidade é recuperada pela sua chave primária e o método *Remove* do *DbSet* a remove do banco de dados.

O método *GetTransaction* proporciona um melhor controle sobre transações. O controle automático de transações que o *Entity Framework* possui no método *SaveChanges* pode não ser suficiente para conversações extensas com o banco de dados em regras de negócio complexas. Então é possível criar uma transação manualmente, com o método *BeginTransation*, e utilizar essa transação para persistir ou reverter alterações. Mesmo com o uso dessa transação, e necessário chamar o método *SaveChanges* antes executar a persistência de dados.

Os repositórios para as entidades *Person* e *Contact* são criados como mostram os quadros 50 e 51. O *Entity Framework* cascateia automaticamente operações de inserção e deleção entre entidades relacionadas, mas não de atualização. Sendo assim, é necessária a existência de repositório de *Contact* para que atualizações dessa entidade sejam persistidas.

```
using NetWebCrud.Models.Entities;
using System;

namespace NetWebCrud.Models.Repositories
{
    public interface IPersonRepository : IGenericRepository<Person>
```

```
    {
        bool ExistsWithName(Person p);
    }
}
using NetWebCrud.Models.Entities;

namespace NetWebCrud.Models.Repositories
{
    public interface IContactRepository : IGenericRepository<Contact>
    {
    }
}
```

Código 50: *Interfaces IPersonRepository* e *IContactRepository*

```
using NetWebCrud.Models.Entities;
using System;
using System.Collections.Generic;
using System.Linq;

namespace NetWebCrud.Models.Repositories
{
    public class PersonRepository : GenericRepository<Person>,
        IPersonRepository
    {
        public bool ExistsWithName(Person p)
        {
            return EntitySet.Any(e => e.Name.Equals(name) && e.Id != p.Id);
        }
    }
}

using NetWebCrud.Models.Entities;

namespace NetWebCrud.Models.Repositories
{
    public class ContactRepository : GenericRepository<Contact>,
        IContactRepository
    {
    }
}
```

Código 51: Classes *PersonRepository* e *ContactRepository*

Assim como no projeto *Java/Spring*, o repositório para a classe *Person* possui um método que verifica a existência de uma outra entidade que possua o mesmo nome. A consulta é feita com uma expressão *lambda* do *LINQ*. O repositório para *Contact* não possui nenhuma operação adicional, usa apenas as operações herdades de *GenericRepository*. Como esses repositórios também serão injetados na classe de serviço *PersonService*, sua configuração de injeção é adicionada à classe *NinjectWebCommon*, como mostra o quadro 52.

```
private static void RegisterServices(IKernel kernel)
```

```
{
        kernel.Bind<IPersonService>().To<PersonService>().InRequestScope();
        kernel.Bind<IPersonRepository>().To<PersonRepository>().
            InRequestScope();
        kernel.Bind<IContactRepository>().To<ContactRepository>().
            InRequestScope();
        kernel.Bind<DbContext>().ToSelf().InRequestScope();
}
```

Código 52: Configuração de injeção de repositórios no projeto *ASP.NET MVC*

Por último, a classe de serviço *PersonService* é atualizada como no quadro 53. O método *Update* mostra como fazer controle manual de transações. Chamando o método *GetTransaction* de *PersonRepository*, ou de qualquer outro repositório, uma transação é criada para o contexto *NetWebCrudContext* que é injetado no escopo da requisição do usuário. Dessa forma, só existe uma instância do contexto do banco de dados sendo utilizada por todos os repositórios, então a transação também é única.

Usando uma transação dessa forma, o desenvolvedor pode fazer as operações que desejar com quaisquer repositórios, e chamar o método *Commit* da transação para finalizar a persistência. Em caso de erros, é chamado o método *Rollback* para reverter às alterações no banco de dados. Como a transação está dentro do escopo da palavra chave *using*, o coletor de lixo do *.NET Framework* se encarrega de chamar o método *Dispose* automaticamente para finalizar a transação.

```
using NetWebCrud.Models.Entities;
using NetWebCrud.Models.Repositories;
using Ninject;
using System;
using System.Collections.Generic;
using System.Linq;

namespace NetWebCrud.Models.Services
{
    public class PersonService : IPersonService
    {
        [Inject]
        public IPersonRepository PersonRepository { set; private get; }
        [Inject]
        public IContactRepository ContactRepository { set; private get; }

        public void Add(Person p) {
            if (PersonRepository.ExistsWithName(p)) {
                throw new Exception("Person already exists.");
            }
            PersonRepository.Save(p);
        }

        public Person GetById(int id) {
            var result = PersonRepository.GetById(id);

            if (result == null) {
                throw new Exception("Person doesn't exists.");
            }
```

```
            return result;
        }

        public Person Update(Person p) {
            Person result = null;

            using (var transaction = PersonRepository.GetTransaction()) {
                try {
                    if (p.ContactId != null && p.ContactId > 0) {
                        ContactRepository.Update(p.Contact);
                    }

                    result = PersonRepository.Update(p);

                    transaction.Commit();
                }
                catch (Exception ex) {
                    transaction.Rollback();
                }
            }
            return result;
        }

        public void Delete(int id) {
            PersonRepository.Delete(id);
        }

        public IList<Person> GetAll() {
            return PersonRepository.GetAll();
        }
    }
}
```

Código 53: Classe *PersonService* utilizando ambos os repositórios no projeto *ASP.NET MVC*

6.5 Conclusão

A criação de entidades para o modelo objeto/relacional com *Hibernate* e *Entity Framework* possuem abordagens distintas. No primeiro, classes precisam de anotações para serem tratadas como entidades, e no segundo não há essa necessidade. No *Entity Framework*, classes mais simples podem ser usadas e convenções cuidam de lógica no mapeamento objeto/relacional. A desvantagem do uso do *Entity Framework* é a necessidade de criação de uma classe de contexto que representa o banco de dados. Dependendo da complexidade da aplicação, a codificação e manutenção da classe de contexto pode se tornar uma inconveniência.

O relacionamento entre entidades no *Hibernate* é feito com anotações na própria entidade, e a configuração de relacionamentos de muito para muitos, por exemplo, pode se tornar complicada de entender para um desenvolvedor iniciante. O *Entity Framework* abstrai essa configuração, sendo apenas necessária a criação de propriedades de navegação entre entidades e propriedades que representem chaves estrangeiras.

Repositórios em projetos *Spring* com *Hibernate* utilizam uma implementação de *EntityManager* para manipular entidades, podendo usar *Criteria Queries* ou *JPQL* para gerar consultas. *Criteria Queries* utilizam vários métodos para gerar uma consulta, que no fim resulta em um código complexo, de baixa legibilidade e de manutenção questionável. A linguagem *JPQL* se assemelha a *SQL* e é mais legível, mas tem a desvantagem de ser escrita com *strings*, o que a torna mais propensa a erros. Em projetos com o *Entity Framework*, a linguagem *LINQ* é usada para gerar consultas. Essa linguagem é de fácil entendimento, e pode utilizar as vantagens do *IntelliSense* (ferramenta de auto completar código do *Visual Studio*), tornando a detecção de erros mais fácil e a escrita de código mais eficiente.

Quanto ao controle de transações, o modo automático como o *Spring Framework* lida com transações o tornam uma melhor opção nesse aspecto do que o *Entity Framework*.

7 Conclusão

Nesse trabalho foi comparada a criação de uma aplicação *web* utilizando as tecnologias *Java* com o *Spring MVC* e o *Hibernate*, e o *ASP.NET MVC* utilizando *C#*. Foram abordadas a configuração do ambiente de desenvolvimento, configuração inicial de projeto e a criação das três camadas da aplicação, apresentação (subdividida entre *controllers* e *views*), serviços (incluindo injeção de dependências) e persistência.

As conclusões a que se chegaram por esse trabalho em relação ao tópicos abordados foram as seguintes:

- Preparação do ambiente de desenvolvimento - A tecnologia *.NET* é mais prática e poupa tempo do desenvolvedor. Enquanto com o *.NET* o necessário para a criação de um projeto se encontra pronto para uso em um único instalador, utilizando a tecnologia *Java* o desenvolvedor deve adquirir vários *softwares* de fontes diferentes e configura-los.
- Criação e configuração de um novo projeto - Aqui a tecnologia *.NET* também obteve vantagem. Os modelos de projeto *ASP.NET MVC* do *Visual Studio* e convenções adotadas, ajudam o desenvolvedor à escrever sua aplicação imediatamente. Com o *Spring Framework* é necessária configuração de diversos objetos da estrutura da aplicação.
- Criação de *controllers* - Na criação de *controllers* em si, nenhuma tecnologia mostrou grande superioridade. Mas o *ASP.NET MVC* demonstrou maior praticidade no mapeamento de endereços para ações, centralizando essa configuração em um único lugar e, vez de anotações espalhadas pelo código como no *Spring Framework*.
- Criação de *views* - A possibilidade de *views* fortemente tipadas e utilização de funcionalidades de auto completar código, deram vantagem à *view engine Razor* do *ASP.NET MVC* em relação ao *JSTL* do *Java Enterprise Edition*. Mas seria injusto não mencionar que a liberdade de escolher outras *view engines* é uma grande vantagem da plataforma *Java*.
- Criação de classes de serviço - Nenhuma tecnologia se mostrou superior à outra em relação a criação de classes de serviço.
- Configuração de injeção de dependências - A existência de um *container* nativo de injeção de dependências no *Spring Framework*, junto com configuração automática de instanciação e injeção de classes mostram superioridade dessa tecnologia em relação ao *ASP.NET MVC* nesse aspecto. Além de precisar adquirir uma biblioteca externa (o *Ninject*, utilizado como exemplo) também foi necessária configuração adicional para cada classe a ser injetada no projeto *.NET*.
- Criação de entidades - Pela não necessidade da criação de um contexto do banco de dados e possuir atualização da estrutura do banco de forma automática, o *Hibernate* implementando o *JPA* se mostrou superior ao *Entity Framework*, apesar de requerer que as entidades sejam decoradas com várias anotações.

- Criação de repositórios - A sintaxe fluente do *LINQ* (*.NET*) se mostrou superior ao *JPQL* (*Java*/ *JPA*) por utilizar expressões *lambda*, ser fortemente tipada e fazer uso das funções de auto completar código do *Visual Studio*. Apesar disso, desenvolvedores mais familiarizados em trabalhar com *scripts SQL* podem preferir o *JPQL*. Em relação ao controle de transações, o *Spring Framework* se mostrou superior, pois automatiza a criação e reuso de transações ao mesmo tempo que dá ao desenvolvedor controle sobre elas.

Existem outras carateristas inerentes às tecnologias utilizadas que devem ser lavadas em consideração na hora de escolher alguma delas para um novo projeto. A versão atual do *ASP.NET* por exemplo, não funciona em sistemas operacionais que não sejam o *Windows*. Então se nos requisitos do sistema especificasse que ele deve rodar em *Linux*, por exemplo, *Java* deverá ser a escolha entre as duas tecnologias. A familiaridade da equipe de desenvolvimento e suas preferencias subjetivas também impactam na escolha da tecnologia.

De modo geral, a conclusão a que esse trabalho chega é que se um projeto precisar ficar pronto rápido e a equipe de desenvolvimento não tiver uma preferencia clara por uma tecnologia, o uso do *ASP.NET* pode ser uma boa opção. Em contra partida, se o time de desenvolvimento quiser ter controle granular sobre o projeto, dispor de tempo para testar várias soluções diferentes e dominar o *Spring Framework*, utilizar o ambiente *Java* é uma melhor opção.

Como trabalho futuro, o desempenho das duas tecnologias pode ser analisado. Quando o *ASP.NET MVC 6* for lançado, pode ser feita uma nova comparação para desenvolvimento em ambiente *Linux*. Pode-se também utilizar o *Spring Boot* em um trabalho futuro e verificar como ele auxilia a configuração de um projeto *Java*.

Esse trabalho foi importante para auxiliar a ter uma compreensão melhor das vantagens e desvantagens entre as duas tecnologias de modo mais lógico. Ouve-se muitas discussões entre desenvolvedores em momentos de descontração que dizem preferir uma tecnologia à outra por motivos subjetivos, muitos deles não conhecendo realmente a tecnologia rival. Assim, esse trabalho também auxilia desenvolvedores experientes em fundamentar seus elogios e críticas à uma ou outra tecnologia.